Advanced Computer Architectures
Theory

Christian Rossi

Academic Year 2023-2024

Abstract

The course topics are:

- Review of basic computer architecture: the RISC approach and pipelining, the memory hierarchy.
- Basic performance evaluation metrics of computer architectures.
- Techniques for performance optimization: processor and memory.
- Instruction level parallelism: static and dynamic scheduling; superscalar architectures: principles and problems; VLIW (Very Long Instruction Word) architectures, examples of architecture families.
- Thread-level parallelism.
- Multiprocessors and multicore systems: taxonomy, topologies, communication management, memory management, cache coherency protocols, example of architectures.
- Stream processors and vector processors; Graphic Processors, GP-GPUs, heterogeneous architectures.

Contents

CHAPTER 1

Introduction

1.1 Architectures' classification

In 1966, Michael Flynn introduced a taxonomy outlining the architecture of calculators. This classification divides architectures into four categories:

- *Single Instruction Single Data*: utilized by uniprocessor systems.
- *Multiple Instruction Single Data*: although theoretically possible, this architecture lacks practical configurations.
- *Single Instruction Multiple Data*: features a straightforward programming model with low overhead and high flexibility, commonly employed in custom integrated circuits.
- *Multiple Instruction Multiple Data*: known for its scalability and fault tolerance, this architecture is utilized by off-the-shelf microservices.

1.1.1 Single instruction single data

The traditional concept of computation involves writing software for serial execution, typically on a single computer with a lone Central Processing Unit (CPU). Tasks are divided into a sequence of discrete instructions executed sequentially, allowing only one instruction to be processed at any given moment. This arrangement is illustrated by the single instruction single data architecture.

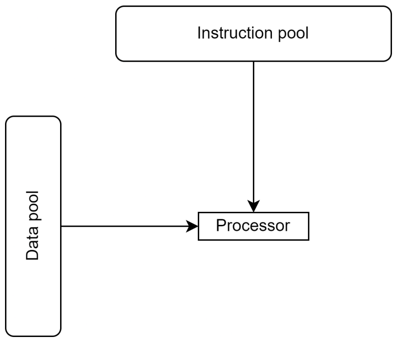


Figure 1.1: Single Instruction Single Data (SISD)

In a single instruction single data architecture:

- *Single instruction*: only one instruction is processed by the CPU in each clock cycle.
- *Single data*: only one data stream is utilized as input during each clock cycle.

Execution in this setup is deterministic, meaning the outcome is predictable and follows a defined sequence of steps. Single instruction single data architecture architectures represent the most prevalent type of computers.

1.1.2 Single instruction multiple data

In the single instruction multiple data architecture, the following characteristics apply:

- *Single instruction*: all processing units execute the same instruction simultaneously during each clock cycle.
- *Multiple data*: each processing unit can handle a different data element independently.

This architecture is particularly well-suited for specialized problems with a high level of regularity, such as graphics and image processing.

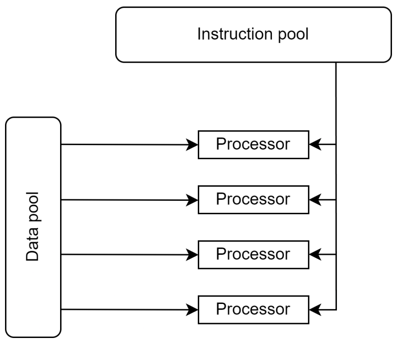


Figure 1.2: Single Instruction Multiple Data

1.1.3 Multiple instructions architectures

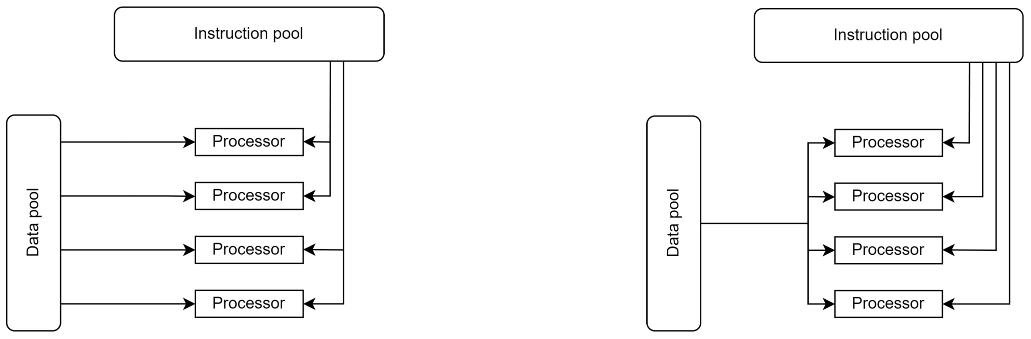
Hardware parallelism can be achieved through various methods:

- *Instruction-level parallelism*: this method harnesses data-level parallelism at different levels. Compiler techniques such as pipelining exploit modest-level parallelism, while speculation techniques operate at medium levels of parallelism.
- *Vector architectures and graphic processor units*: these architectures leverage data-level parallelism by executing a single instruction across a set of data elements simultaneously.
- *Thread-level parallelism*: this approach exploits either data-level or task-level parallelism within a closely interconnected hardware model that enables interaction among threads.
- *Request-level parallelism*: this method capitalizes on parallelism among largely independent tasks specified by either the programmer or the operating system.

Currently, the most common type of parallel computer features:

- *Multiple instruction*: each processor may execute a different instruction stream.
- *Multiple data*: each processor may operate with a distinct data stream.

Execution in parallel computing can occur synchronously or asynchronously, and it may be deterministic or non-deterministic.



(a) Multiple Instruction Multiple Data (b) Multiple Instruction Single Data

Figure 1.3: Possible architectures for hardware parallelism

CHAPTER 2

MIPS

2.1 Characteristics

MIPS embodies the principles of Reduced Instruction Set Computer (RISC) architecture, focusing on streamlined execution by employing simple instructions within a condensed basic cycle. This design aims to enhance the efficiency of Complex Instruction Set Computer (CISC) CPUs.

As a load-store architecture, MIPS operates such that Arithmetic Logic Unit operands are sourced exclusively from the CPU's general-purpose registers, precluding direct retrieval from memory. Dedicated instructions are thus essential for:

- Loading data from memory into registers.
- Storing data from registers into memory.

A pipeline architecture is a pivotal technique aimed at performance optimization. It capitalizes on the concurrent execution of multiple instructions derived from a sequential execution flow.

Furthermore, the Instruction Set Architecture (ISA) of MIPS encompasses a defined set of operations, instruction formats, supported hardware data types, named storage, addressing modes, and sequencing protocols.

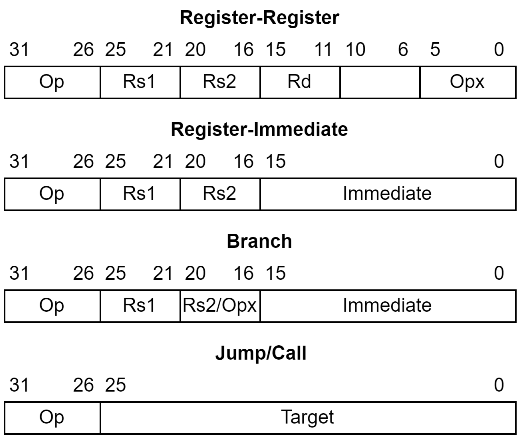


Figure 2.1: MIPS instruction set architecture

2.1.1 MIPS CPU

Within a MIPS CPU, the datapath encompasses the necessary components such as storage, functional units (FUs), and interconnects to execute desired operations effectively. In this setup, control points serve as inputs while signals serve as outputs.

The controller, functioning as a state machine, coordinates the activities within the datapath by directing operations based on the desired function and the signals received.

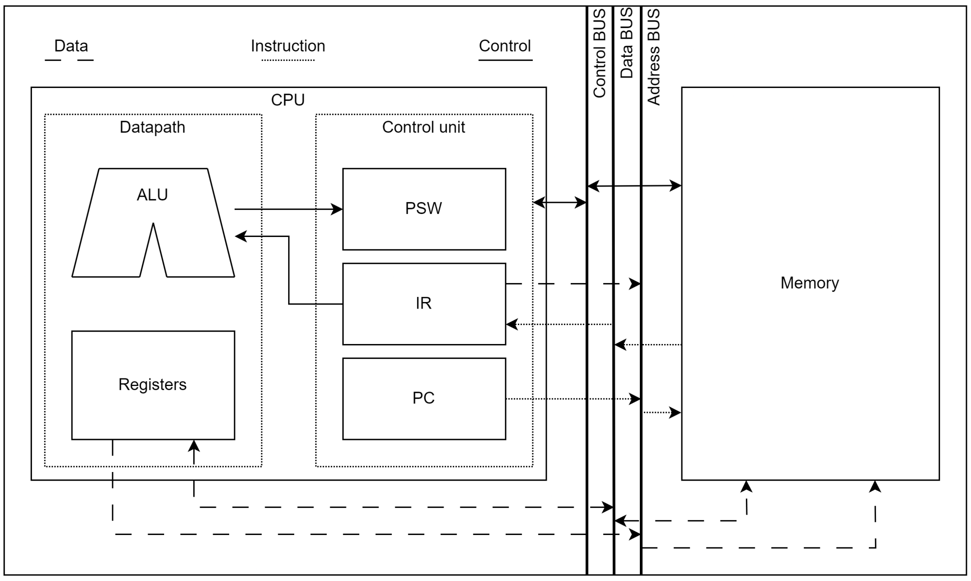


Figure 2.2: MIPS CPU

2.1.2 Program execution

At the core, a program is segmented into instructions, with the hardware focusing on individual instructions rather than the entire program. At a lower level, the hardware divides instructions into clock cycles, with lower-level state machines transitioning states with each cycle.

2.2 MIPS instruction execution

Each instruction within the MIPS subset can be executed within a maximum of five clock cycles, as outlined below:

1. *Instruction fetch cycle*:
   - Transfer the content of the program counter register to the instruction memory and retrieve the current instruction.
   - Update the program counter to the next sequential address by incrementing it by 4 (since each instruction occupies 4 bytes).
2. *Instruction decode and register read cycle*:
   - Decode the current instruction using fixed-field decoding.
   - Access the register file to read one or two registers as specified by the instruction fields.

- Perform sign-extension of the offset field of the instruction if necessary.

3. *Execution cycle*:
    - For register-register ALU instructions, the ALU performs the specified operation on the operands retrieved from the register file (RF).
    - For register-immediate ALU instructions, the ALU performs the specified operation on the first operand retrieved from the RF and the sign-extended immediate operand.
    - For memory reference instructions, the ALU computes the effective address by adding the base register and the offset.
    - For conditional branches, it compares the two registers read from the RF and calculates the potential branch target address by adding the sign-extended offset to the incremented program counter (PC).
4. *Memory access*:
    - Load instructions entail a read access to the data memory using the effective address.
    - Store instructions require a write access to the data memory using the effective address to store the data from the source register read from the RF.
    - Conditional branches may update the content of the PC with the branch target address if the conditional test evaluates to true.
5. *Write-back cycle*:
    - Load instructions write the data retrieved from memory into the destination register of the RF.
    - ALU instructions store the ALU results into the destination register of the RF.

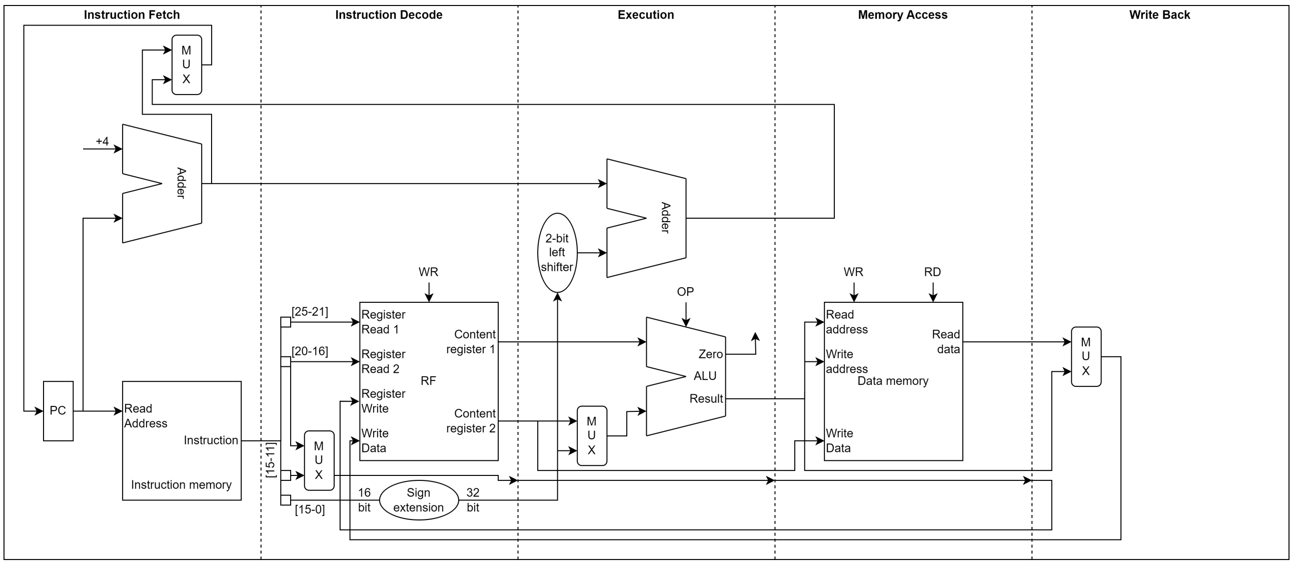


Figure 2.3: MIPS CPU architecture

Below are the durations of each instruction:

Instruction type	**Instruction memory**	**Register read**	**ALU operations**	**Data memory**	**Write back**	**Total latency**
ALU instruction	2	1	2	0	1	$6ns$
Load	2	1	2	2	1	$8ns$
Store	2	1	2	2	0	$7ns$
Conditional branch	2	1	2	0	0	$5ns$
Jump	2	0	0	0	0	$2ns$

The duration of each clock cycle is determined by the critical path established by the load instruction, denoted as $T = 8ns$ (equivalent to a frequency of $f = 125MHz$). We assume a single-clock cycle execution for each instruction, wherein each module is utilized once within a cycle. Modules utilized more than once within a cycle necessitate duplication for efficiency. Furthermore, to ensure separate functionality, an instruction memory distinct from the data memory is required.

Certain modules must be duplicated, while others are shared across different instruction flows. To facilitate sharing a module between two distinct instructions, a multiplexer is utilized. This device enables multiple inputs to access a module and allows the selection of one input among several based on the configuration of control lines.

In the multi-cycle implementation of CPU, the execution of instructions spans across multiple cycles, with MIPS typically utilizing five cycles. The fundamental cycle is shorter at $2ns$, leading to an instruction latency of $10ns$. Key aspects of the multi-cycle CPU implementation include:

- Each phase of instruction execution necessitates a clock cycle.
- Modules can be utilized multiple times per instruction across different clock cycles, allowing for potential module sharing.
- Internal registers are required to retain values for subsequent clock cycles. These registers store data to be utilized in future stages of the instruction execution process.

2.3 Pipelining

Pipelining is an optimization method aimed at enhancing performance by overlapping the execution of multiple instructions originating from a sequential execution flow. It capitalizes on the inherent parallelism among instructions within a sequential instruction stream.

The fundamental concept involves breaking down the execution of an instruction into distinct phases, also known as pipeline stages. Each stage requires only a portion of the time needed to complete the instruction. These stages are interconnected to form a pipeline: instructions enter at one end, traverse through the stages, and exit from the other end, akin to an assembly line. This facilitates a continuous flow of instruction execution, leading to improved efficiency and throughput.

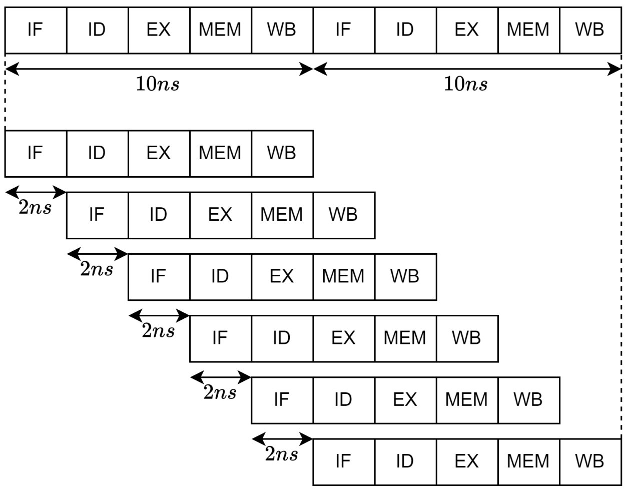


Figure 2.4: Sequential execution and pipelining execution

In pipelining, each stage of the pipeline corresponds to the time required to advance an instruction by one clock cycle. It's crucial to synchronize the pipeline stages, with the duration of a clock cycle determined by the slower stage of the pipeline, typically $2ns$.

The objective is to achieve a balance in the length of each pipeline stage. When stages are perfectly balanced, the ideal speedup resulting from pipelining is equal to the number of pipeline stages. This ensures optimal utilization of the pipeline, enhancing overall performance and efficiency.

In the ideal scenario, we compare a single-cycle unpipelined CPU1 with a clock cycle of $8ns$ to a pipelined CPU2 with five stages of $2ns$ each. In this case the latency (total execution time) of each instruction is increased from $8ns$ to $10ns$ due to the pipeline overhead. However, the throughput (number of instructions completed in a given time unit) is enhanced by four times: CPU1 completes one instruction every $8ns$, while CPU2 completes one instruction every $2ns$.

In the ideal scenario, when comparing a multi-cycle unpipelined CPU3 consisting of five cycles of $2ns$ each to a pipelined CPU2 with five stages of $2ns$ each. The latency (total execution time) of each instruction remains constant at $10ns$. However, the throughput (number of instructions completed in a given time unit) is enhanced by five times: CPU3 completes one instruction every $10ns$, while CPU2 completes one instruction every $2ns$.

2.3.1 Possible issues

A potential concern arises due to the two-stage nature of the register file: read access during the instruction decode stage and write access during the write-back stage. When a read and a write operation target the same register within the same clock cycle, it necessitates the insertion of a stall to prevent issues.

Definition (*Optimized pipeline*)**.** An optimized pipeline is achieved when the register file read operation takes place in the second half of the clock cycle, while the register file write operation occurs in the first half of the clock cycle.

Another potential issue is the occurrence of hazards within the pipeline. Hazards arise when there is a dependency between instructions, and the pipelining process causes a change in the order of accessing operands involved in the dependency, thereby preventing the next instruction from executing during its designated clock cycle. Hazards diminish the performance from the ideal speedup achieved by pipelining. Hazards can be categorized into three main types:

- *Structural hazards*: these occur when different instructions attempt to use the same resource simultaneously. For example, there may be a conflict when both instructions require access to a single memory unit for instructions and data.
- *Data hazards*: these occur when an instruction tries to use a result before it is ready. For instance, an instruction might depend on the result of a previous instruction that is still in the pipeline.
- *Control hazards*: these occur when a decision regarding the next instruction to execute is made before the condition for the decision is evaluated. For instance, issues arise during conditional branch execution.

If dependent instructions are executed closely within the pipeline, data hazards become more prevalent.

2.3.2 Data hazards

Read after write A data hazard of the "read after write" type occurs when an instruction j attempts to read an operand before instruction i has written to it. This hazard, known as a "dependence" in compiler terminology, arises from a genuine necessity for communication between instructions. The potential solutions for mitigating this hazard include:

- *Compilation techniques*:
  - Insertion of "nop" (no operation) instructions.
  - Instruction scheduling: the compiler arranges instructions to ensure that dependent instructions are not placed too close together. It attempts to intersperse independent instructions among dependent ones. If independent instructions cannot be found, the compiler inserts "nop" instructions.
- *Hardware techniques*:
  - Insertion of "bubbles" or stalls in the pipeline.
  - Data Forwarding or Bypassing: this technique involves using temporary results stored in the pipeline registers instead of waiting for the results to be written back to the register file. Multiplexers are added at the inputs of the ALU to fetch inputs from pipeline registers, thus avoiding the need to insert stalls in the pipeline.

Utilizing forwarding allows for resolving this conflict without introducing stalls in most cases. However, for load/use hazards, it is imperative to insert one stall to properly address the issue.

Write after write A data hazard of the "write after write" type arises when instruction j writes operand before instruction i writes it. This situation results in write operations being executed in an incorrect order. Notably, this type of hazard does not occur in the MIPS pipeline since all register write operations take place in the write-back stage. Compiler writers classify this hazard as an output dependence.

Write after read A data hazard of the ”write after write” type arises when an instruction j writes operand before instruction i reads it. This scenario leads to the possibility of reading an incorrect value. However, such hazards do not occur in the MIPS pipeline because operand read operations take place in the instruction decode stage, while write operations occur in the write-back stage. Similarly, assuming that register writes in ALU instructions occur in the fourth stage and that two stages are needed to access the data memory, some instructions might read operands too late in the pipeline. Compiler writers classify this hazard as an anti-dependence.

CHAPTER 3

Performance evaluation

3.1 Introduction

Creating software has become progressively more difficult, to the extent that manually handling all the constraints has become almost unmanageable. Despite the abundance of processor cores due to the unprecedented computational power, energy consumption has become a crucial limitation. Consequently, there is an urgent requirement for software to prioritize energy efficiency and be mindful of space constraints.

3.2 Speed measures

To evaluate the speed of two computers, we can utilize two metrics:

- *User perspective* Users aim to minimize the elapsed time for program execution, which is measured by the response time:
$$T_{response} : T_{execution} = T_{end} - T_{start}$$
- *System manager perspective*: system managers aim to maximize the completion rate, or throughput, which refers to the total amount of work done within a specified timeframe.

These metrics can be compared using the formula:

$$T_{throughput} = \frac{1}{T_{response}}$$

This equality holds true if there are no overlaps; otherwise, a greater throughput is achieved.

Typically, we focus on the common case as it's usually simpler and quicker to assess than the uncommon case.

Theorem 3.2.1 (*Amdahl law*)**.** *The performance improvement of a system achieved by optimizing a particular part of the system is limited by the fraction of time that part is utilized.*

Suppose an enhancement E accelerates a fraction F of the task by a factor S, while the remainder of the task remains unaffected. The overall speedup can be calculated as:

$$S_{overall} = \frac{1}{(1 - F_{enhanced}) + \frac{F_{enhanced}}{S_{enhanced}}}$$

As a result the best we could ever hope to do is:

$$\text{Speedup}_{overall} = \frac{1}{1 - F_{enhanced}}$$

Example:
Considering a new CPU that is 10 times faster, and an I/O-bound server where 60% of the time is spent waiting for I/O, the overall speedup would be:

$$S_{overall} = \frac{1}{(1 - F_{enhanced}) + \frac{F_{enhanced}}{S_{enhanced}}} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.56$$

It's important to note that despite the allure of a 10x speed increase, the actual improvement is only 1.6x. This illustrates a fundamental aspect of human perception regarding speed improvements.

Corollary 3.2.1.1 (*Amdahl's law*)**.** *If an enhancement is only applicable for a fraction of a task, the task's speed cannot be increased by more than the reciprocal of one minus that fraction.*

3.3 Performance measures

The system's performance can be characterized by the following components:

- *Response time*: this encompasses the latency incurred while completing a task, which includes factors such as disk accesses, I/O activity, and operating system overhead. The elapsed time is determined by the sum of CPU time and I/O wait time:

$$T_{elapsed} = T_{CPU} + T_{I/O\ wait}$$

- *CPU time*: this denotes the time spent waiting for I/O operations and corresponds to the CPU's processing time. It can be computed as:

$$T_{CPU}(P) = \frac{\text{clock cycles needed to execute } P}{\text{clock frequency}}$$

$$T_{CPU}(P) = \text{clock cycles needed to execute } P \cdot \text{clock cycles time}$$

3.3.1 CPU time

The CPU time can be determined by the following formula:

$$T_{CPU} = \underbrace{\frac{\text{Instructions}}{\text{Program}}}_{\text{IC}} \cdot \underbrace{\frac{\text{Cycles}}{\text{Instruction}}}_{\text{CPI}} \cdot \underbrace{\frac{\text{Seconds}}{\text{Cycle}}}_{\text{CT}}$$

Here are the components involved:

- *Instruction count* (IC): this denotes the number of instructions executed, not the static code size. It is influenced by the algorithm, compiler, and Instruction Set Architecture (ISA).

- *Cycles per instructions* (CPI): determined by the ISA and CPU organization, this metric accounts for overlap among instructions (pipelining) which reduces this term.
- *Cycle time* (CT): determined by technology, organization, and circuit design.

The objective of CPU performance is to minimize time, which is the product of these three terms because they are interconnected.

Cycles per instruction As mentioned, cycles per instruction are defined as:

$$\text{CPI}(P) = \frac{\text{clock cycles needed to execute } P}{\text{number of instructions}}$$

We can also define the CPU time for every single instruction as:

$$T_{CPU} = \text{CT} \cdot \sum_{i=1}^{n} \text{CPI}_i \cdot \text{I}_i$$

Thus, better results are obtained by allocating resources to the instructions where time is spent.

3.3.2 Other metrics

Other metrics utilized to evaluate hardware performance include:

- *Million of instructions per second* (MIPS):

  $$\text{MIPS} = \frac{\text{IC}}{T_{execution} \cdot 10^6} = \frac{\text{Clock frequency}}{\text{CPI} \cdot 10^6}$$

  MIPS quantifies the rate of operations per unit time, with faster machines having higher MIPS ratings. However, MIPS has significant limitations.

- *Million of floating point operations* (MFLOPS):

  $$\text{MFLOPS} = \frac{\text{Floating point operations in a program}}{T_{CPU} \cdot 10^6}$$

  Assuming floating-point operations are independent of compiler and ISA, MFLOPS can be a reliable metric for numeric codes based on the matrix size, determining the number of floating-point operations in a program. However, it's not always reliable due to factors like missing instructions or optimizing compilers.

3.4 Benchmarks

The conventional method for conducting performance tests on programs involves using benchmarks. In this methodology, certain groups select programs available to the community to measure performance. These programs are executed on machines, and their performance is reported, allowing for comparison with reports from other machines.

The most commonly used benchmarks include:

- *Real programs*: these are representative of real workloads and provide the most accurate way to characterize performance. Occasionally, modified CPU-oriented benchmarks may eliminate I/O operations.

- *Kernels* or microbenchmarks: these are representative program fragments that are useful for focusing on individual features.
- *Synthetic benchmarks*: similar to kernels, these benchmarks attempt to match the average frequency of operations and operands from a large set of programs.
- *Instruction mixes* for CPI.

3.4.1 System performance evaluation cooperative

The System Performance Evaluation Cooperative was established in 1989 to address bench marketing issues. SPEC2000, for example, is based on 12 integer and 14 floating-point programs, and it is utilized to evaluate the CPU, memory architecture, and compilers' compute-intensive performance.

3.4.2 Benchmark problems

Benchmarks may not be representative if the workload is I/O bound, rendering certain benchmarks like SPECint ineffective. Benchmarks also become obsolete over time, and aging benchmarks can be problematic as bench marketing pressure incentivizes vendors to optimize compiler/hardware/software to specific benchmarks. Therefore, benchmarks need to be periodically refreshed.

3.4.3 Alternatives

A straightforward method for comparing relative performance is to use the total execution time of the two programs. Another option is to calculate the arithmetic mean of the execution times, which is valid only if the programs run equally often. If the programs have different frequencies of execution, the weighted arithmetic mean is used:

$$\left\{\sum_{i=1}^{n} \text{weight}(i) \cdot \text{time}(i)\right\} \div \frac{1}{n}$$

In general:

- Use the arithmetic mean for times.
- Use the harmonic mean if rates must be used.
- Use the geometric mean if ratios must be used.

3.5 Energy and power

Energy and power consumption pose constraints for a variety of systems, including embedded systems, IoT devices, and mobile devices due to battery capacities, as well as for desktops, servers, and HPC clusters in terms of energy costs. Consequently, it becomes necessary to establish an energy and power budget for these systems. Achieving optimization in this regard requires a thorough understanding of the underlying phenomenon.

3.5.1 Energy consumption

Thermal Design Power (TDP) serves as a metric to characterize sustained power consumption. It is used as a target for power supply and cooling systems and typically falls between peak power and average power consumption. The clock rate can be dynamically reduced to limit power consumption, and measuring energy per task often provides a more accurate assessment.

Various techniques are employed to reduce dynamic power consumption:

- Optimizing existing processes.
- Implementing dynamic voltage-frequency scaling.
- Employing low-power states for DRAM and disks.
- Utilizing methods like overclocking and turning off cores.

Additionally, static power consumption, which scales with the number of transistors, needs to be considered. To mitigate static power, a technique called power gating is commonly employed.

CHAPTER 4

Caches and memory

4.1 Memory hierarchy

Since 1980, there has been a notable divergence in performance between CPUs and DRAM. To bridge this gap, architects introduced small, high-speed cache memories between the CPU and DRAM, establishing a memory hierarchy.

During the period from 1980 to 1986, DRAM latency decreased by 9% annually, while CPU performance saw a steady increase of 1.35 times per year. Post-1986, CPU performance accelerated further to 1.55 times per year, while DRAM performance remained relatively constant.

With the advent of recent multicore processors, the design of memory hierarchy has become increasingly critical.

Example:
Consider the Intel Core i7 processor, which boasts the capability to generate two references per core clock cycle. With a total of four cores operating at a clock frequency of $3.2\,GHz$, it can achieve a good throughput. Specifically, it can handle 25.6 billion 64-bit data references per second and 12.8 billion 128-bit instruction references per second, resulting in a combined throughput of $409.6\,GB/s$. However, this remarkable processing power highlights a contrast with the DRAM bandwidth, which is merely 6% of the total throughput, amounting to a modest $25\,GB/s$.

To address this challenge, several solutions are necessary:

- Implementation of multi-port, pipelined caches to enhance data access efficiency.
- Adoption of a two-level cache structure per core to optimize data retrieval.
- Integration of a shared third-level cache directly on the chip to further streamline memory access.

In modern high-end microprocessors, the on-chip cache capacity has surpassed 10 MB, albeit at the cost of significant area and power consumption.

The ultimate goal of memory hierarchy is to create the illusion of a vast, speedy, and cost-effective memory system that allows programs to access a memory space scalable to the size of the disk, with speeds comparable to register access. Achieving this necessitates the establishment of a memory hierarchy comprising various technologies, costs, and sizes, each with distinct access mechanisms.

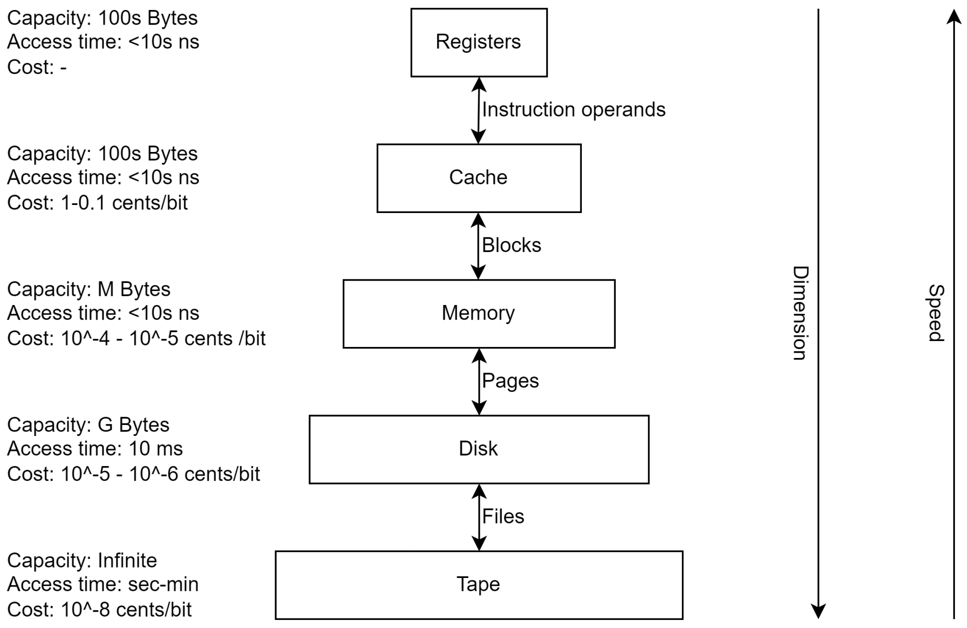


Figure 4.1: Memory hierarchy

4.2 Principle of locality

The principle of locality asserts that programs tend to access only a fraction of the total address space at any given moment. This principle is further elaborated through two key properties:

1. *Temporal locality*: this property suggests that if a memory location is accessed, it is probable that it will be accessed again in the near future.

2. *Spatial locality*: this property indicates that if a memory location is accessed, it is likely that nearby locations will also be accessed in the near future.

Caches leverage both forms of predictability. They exploit temporal locality by retaining the contents of recently accessed memory locations, anticipating their future use. Additionally, they exploit spatial locality by pre-fetching blocks of data surrounding recently accessed locations, capitalizing on the likelihood of adjacent memory access.

When examining a processor address, the cache tags are searched to locate a match. Subsequently, one of the following actions occurs:

- If a match is found in the cache (HIT), the data copy is retrieved from the cache and returned.

- If the address is not found in the cache (MISS), a block of data is read from the main memory. There is a wait period, after which the data is returned to the processor, and the cache is updated accordingly.

Based on these operations, several metrics can be defined.

Definition (*Hit rate*)**.** The hit rate is the fraction of accesses that are found in the cache.

Definition (*Miss rate*)**.** The miss rate is the complement of the hit rate, indicating the fraction of accesses that result in cache misses.

Definition (*Hit time*)**.** The hit time encompasses the time required for RAM access along with the time needed to determine whether the access resulted in a HIT or MISS.

Definition (*Miss time*)**.** The miss time comprises the time necessary to replace a block in the cache and the time taken to deliver the block to the processor.

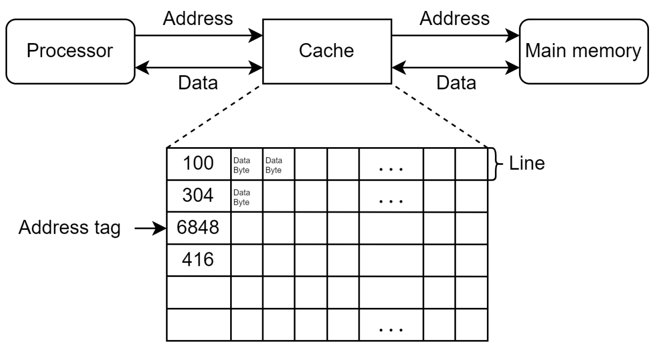
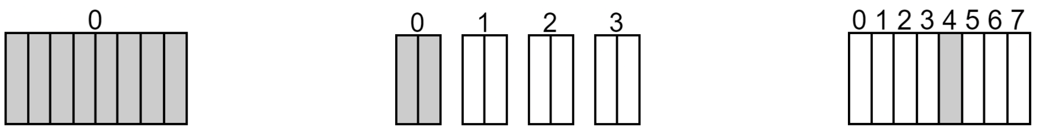


Figure 4.2: Cache and processor interaction

4.2.1 Block placement

Depending on the chosen memory type, the block numbered 12 can be positioned as follows:

- *Fully associative*: it can be placed anywhere within the memory.
- *Two-way set associative*: it can be placed anywhere within set zero, which corresponds to $12 \mod 4$.
- *Direct mapped*: it can only be placed into block four, determined by $12 \mod 8$.



(a) Fully associative (b) Two-way set associative (c) Direct mapped

Figure 4.3: Possible placement of blocks

In this diagram, the placement of block 12 is illustrated based on the different memory configurations mentioned above.

4.2.2 Block identification

A cache miss can occur for several reasons:

1. *Compulsory miss* (cold start or process migration): this happens during the first access to a block, such as during a cold start or when a process migrates. It's essentially an unavoidable aspect of system operation, and there's little that can be done to mitigate it.

2. *Capacity miss*: This occurs when the cache is unable to accommodate all the blocks accessed by the program. Increasing the cache size is a potential solution to reduce the frequency of these misses.

3. *Conflict miss* (collision): multiple memory locations are mapped to the same cache location, resulting in conflicts. This can be addressed by either increasing the cache size or increasing associativity, which allows more flexibility in mapping memory locations to cache locations.

4. *Coherence Miss* (invalidation): This type of miss occurs when another process, such as I/O operations, updates memory, leading to inconsistencies in cached data. Ensuring cache coherence mechanisms are in place can help mitigate this issue.

To locate a block, the cache index is used to select the set to search within, while the tag identifies the actual copy. If no matching candidates are found, a cache miss is declared.

The structure of a memory address typically includes a field for selecting data within a block. However, some caching applications may not utilize this field.

Block address		Block offset
Tag	Index	

Figure 4.4: Memory address general structure

Increasing associativity reduces the index size and expands the tag. Fully associative caches, for example, do not have an index field and can directly access any block in the cache.

The fully associative cache, characterized by requiring only a tag and block offset, is depicted as follows.

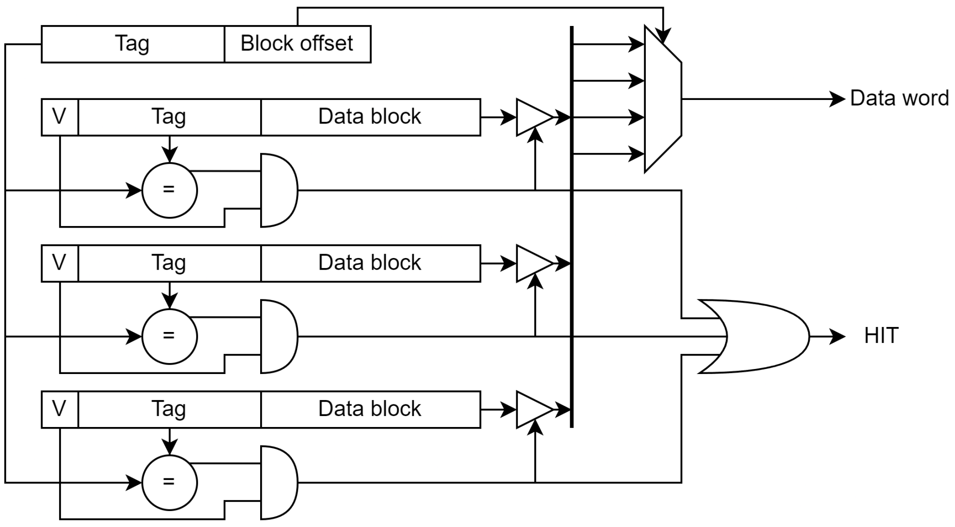


Figure 4.5: Fully associative cache

The two-way set associative cache, which necessitates a tag, index, and block offset, is represented as follows.

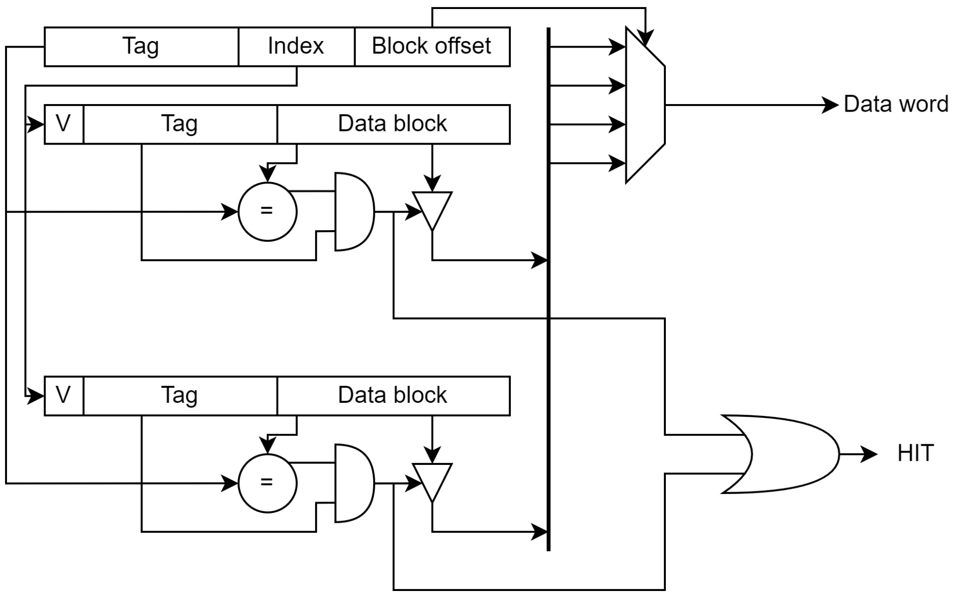


Figure 4.6: Two-way set associative cache

The direct mapped cache, which also requires a tag, index, and block offset, is illustrated as follows.

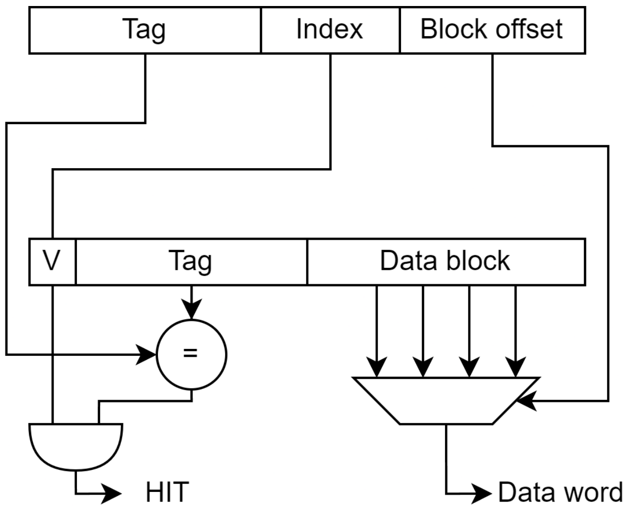


Figure 4.7: Direct mapped cache

Each type of cache has its own specific structure, with varying degrees of associativity and corresponding requirements for indexing and tagging.

4.2.3 Block replacement

In the context of cache misses, block replacement is straightforward for direct-mapped caches. However, for set-associative or fully associative caches, the choice of replacement policy has a significant impact because replacements only occur upon misses. Here are some commonly used replacement policies:

- *Random*: blocks are replaced randomly, without any specific order or pattern.

- *Least recently used* (LRU): this policy replaces the block that has been accessed least recently. Although effective, implementing LRU requires tracking the access history of each block, making it feasible only for caches with a few sets due to the computational overhead.
- *First in first out* (FIFO): blocks are replaced based on the order they were brought into the cache. FIFO is commonly used in highly associative caches where keeping track of access history for LRU may not be practical.

Each of these replacement policies has its advantages and trade-offs, and the choice depends on factors such as cache size, associativity, and the desired balance between complexity and performance.

4.2.4 Write strategy

In the event of a cache hit, we have two options for handling writes:

- *Write through*: this strategy involves writing the data both to the cache and to main memory simultaneously. While this approach typically results in higher traffic, it simplifies cache coherence management.
- *Write back*: with this approach, the data is written only to the cache. The corresponding entry in main memory is updated only when the cache block is evicted. A dirty bit per block helps reduce traffic by indicating whether the block in the cache has been modified.

In the case of a cache miss, we also have two alternatives:

- *No write allocate*: with this method, data is written directly to main memory without being fetched into the cache.
- *Write allocate* (also known as fetch on write): In this scenario, the data is fetched into the cache upon a write miss.

The most common combinations of these strategies are:

- *Write through with no write allocate*: data is written to both the cache and main memory simultaneously, and in the event of a write miss, no data is brought into the cache.
- *Write back with write allocate*: data is written only to the cache, and in the case of a write miss, the data is fetched into the cache before being modified.

We can also use a write buffer for write-through caches to avoid CPU stalls.

4.3 Cache performance

Definition (*Memory stall cycles*)**.** Memory stall cycles is the number of cycles in which the CPU is not working (stalled) waiting for a memory access.

We assume that the cycle time includes the time necessary to manage a cache hit and that during a cache miss the CPU is stalled. We can now compute:

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \cdot \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \cdot \text{Miss penalty}$$

We can simplify by averaging reads and writes:

$$\text{Memory stall cycles} = \text{IC} \cdot \left(\frac{\text{Memory accesses}}{\text{Instruction}}\right) \cdot \text{Miss rate} \cdot \text{Miss penalty}$$

This value is independent of the hardware implementation, but dependent on the architecture.

On the other hand the average access time is computed as:

$$\begin{aligned}
T_A &= H_{rate} \cdot H_{time} + M_{rate} \cdot M_{time} \\
&= H_{rate} \cdot H_{time} + M_{rate} \cdot (H_{time} + M_{penalty}) \\
&= H_{time} \cdot (H_{rate} + M_{rate}) + M_{rate} \cdot M_{penalty} \\
&= H_{time} + M_{rate} \cdot M_{penalty}
\end{aligned}$$

4.3.1 Basic cache optimizations

The access time for a cache can be optimized by:

- Reducing M_{rate}:
  - Larger Block size (compulsory misses):
  - Larger Cache size (capacity misses):
  - Higher Associativity (conflict misses):
- Reducing $M_{penalty}$:
  - Multilevel Caches:
- Reducing H_{time}:
  - Giving Reads Priority over Writes:

4.3.2 Cache design

Within the realm of cache design, various factors interplay: cache size, block size, associativity, replacement policy, and the choice between write-through and write-back mechanisms. Determining the best option involves finding a balance influenced by access patterns (workload and usage) and technological expenses. Often, simplicity emerges as the preferred solution.

The performance metrics used to evaluate the performance are:

- Latency is concern of cache.
- Bandwidth is concern of multiprocessors and I/O.
- Access time: time between read request and when desired word arrives.
- Cycle time: minimum time between unrelated requests to memory.

DRAM used for main memory, SRAM used for cache.

SRAM SRAM memory requires low power to retain bit and requires six transistors for each bit.

DRAM The DRAM must be re-written after being read and also periodically refreshed (each row simultaneously every eight milliseconds). However, this type of memory requires only one transistor per bit. The address lines are multiplexed:

- Upper half of address: row access strobe (RAS).
- Lower half of address: column access strobe (CAS).

Flash memory Flash memory belongs to the category of EEPROM, necessitating block erasure prior to overwrite. It retains data without power, constituting non-volatile storage. With a finite number of write cycles, it falls in price between SDRAM and disk storage. Although slower than SRAM, it outpaces traditional disk speeds.

Optimizations The Amdahl law states that the memory capacity should grow linearly with processor speed. Unfortunately, memory capacity and speed has not kept pace with processors. To overcome this issue some optimizations are possible:

- Multiple accesses to same row.
- Synchronous DRAM: added clock to DRAM interface, and burst mode with critical word first.
- Wider interfaces.
- Double data rate (DDR).
- Multiple banks on each DRAM device.

4.4 Virtual memory

Virtual memory is utilized to confine processes within their allocated memory space boundaries. This architecture serves multiple functions:

- It facilitates user mode and supervisor mode.
- It safeguards specific CPU state components.
- It incorporates mechanisms for transitioning between user mode and supervisor mode.
- It provides tools for restricting memory accesses.
- It includes a Translation Lookaside Buffer (TLB) for address translation.

4.4.1 Virtual machines

Virtual memory's concept enables the creation of virtual machines. These machines support isolation and security, allowing multiple unrelated users to share a computer. This capability is made feasible by the processors' raw speed, which mitigates the associated overhead.

Virtual machines enable the presentation of different Instruction Set Architectures (ISAs) and operating systems to user programs.

The software responsible for system virtual machines is called a hypervisor, with individual virtual machines operating under it referred to as guest virtual machines. Each guest operating system maintains its set of page tables:

- The hypervisor introduces a layer of memory between physical and virtual memory, termed real memory.
- The hypervisor maintains a shadow page table mapping guest virtual addresses to physical addresses. This necessitates the hypervisor's ability to detect changes made by the guest to its page table, which naturally occurs if accessing the page table pointer is a privileged operation.

CHAPTER 5

Exception handling

5.1 Introduction

Definition (*Interrupt*)**.** An interrupt refers to an external or internal event necessitating processing by another system program.

Typically unexpected or infrequent from the program's perspective, interrupts can stem from various causes:

- *Asynchronous*: stemming from external events, such as input/output device service requests, timer expirations, power disruptions, or hardware failures.
- *Synchronous*: arising from internal events (exceptions), including undefined opcodes, privileged instructions, arithmetic overflows, FPU exceptions, misaligned memory access, virtual memory exceptions (such as page faults, TLB misses, and protection violations), and traps (system calls).

5.1.1 History

The first system to incorporate exceptions was the Univac-I in 1951, where an arithmetic overflow would either trigger the execution of a two-instruction fix-up routine at address 0 or, optionally, cause the computer to halt.

The Univac 1103, modified in 1955, introduced external interrupts for gathering real-time wind tunnel data.

The DYSEAC in 1954 was the first system with I/O interrupts, featuring two program counters, and an I/O signal facilitated switching between them. Additionally, it was the first system to incorporate DMA (Direct Memory Access) by I/O devices.

5.2 Taxonomy

Exceptions can be categorized as follows:

- *Synchronous* or *asynchronous*: asynchronous events stem from devices external to the CPU and memory, manageable after the current instruction completes (easier to handle).

- *User requested* or *coerced*: user requested exceptions are predictable and treated similarly to exceptions, utilizing the same mechanisms for saving and restoring state; they are handled after instruction completion. Coerced exceptions arise from hardware events beyond the program's control.
- *User maskable* or *user nonmaskable*: the mask dictates whether the hardware responds to the exception.
- *Within instructions* or *between instructions*: exceptions occurring within instructions are typically synchronous as the instruction initiates the exception. The instruction must halt and restart. Asynchronous exceptions between instructions arise from critical situations, leading to program termination.
- *Resume* or *terminate*: terminating events result in program execution always halting after the interrupt. Resuming events allow program execution to continue after the interrupt.

5.3 Interrupts

Asynchronous interrupt In case of asynchronous interrupt an I/O device signals the need for attention by activating a prioritized interrupt request line. Upon the processor's decision to handle the interrupt:

1. Execution halts at instruction I_i of the current program, ensuring completion of all instructions up to I_{i-1} (precise interrupt).
2. The processor stores the program counter (PC) value of instruction I_i in a dedicated register (EPC).
3. Interrupts are disabled, and control shifts to a specified interrupt handler operating in kernel mode.

The interrupt handler:

- Before enabling interrupts to accommodate nested interrupts:
  - It saves the program counter (PC) to facilitate nested interrupts.
  - Requires an instruction to transfer the PC into general-purpose registers (GPRs).
  - Requires a mechanism to temporarily block further interrupts until the PC is saved.
- It retrieves information about the interrupt cause from a designated status register.
- Utilizes a specialized indirect jump instruction called Return-From-Exception (RFE), which:
  - Enables interrupts.
  - Restores the processor to user mode.
  - Reinstates hardware status and control state.

Synchronous interrupt A synchronous interrupt, also known as an exception, is triggered by a specific instruction. Typically, the instruction cannot finish execution and must be restarted after handling the exception. This necessitates undoing the impact of one or more partially executed instructions. However, in the scenario of a system call trap, the instruction is deemed as fully executed. This involves a special jump instruction that transitions to privileged kernel mode.

5.3.1 Precise interrupt

Definition (*Precise interrupt*)**.** An interrupt or exception is deemed precise when a singular instruction (or interrupt point) exists at which all preceding instructions have finalized their state, and no subsequent instructions, including the interrupting instruction, have altered any state.

This implies that you can effectively resume execution from the interrupt point and obtain the correct outcome.

Precise interrupts are desirable for several reasons. They facilitate the restart of various interrupt and exception types. Additionally, they simplify the process of determining the exact cause of the interruption.

While restartability doesn't mandate preciseness, it significantly enhances the ease of restarting. Preciseness notably streamlines the task for the operating system: less state must be preserved when unloading processes, and restarts are quicker.

5.4 Exception handling in five stage pipelines

Exceptions may arise at various stages within the pipeline. However, the recommended approach for interrupt handling is to minimize pipeline interruption as much as possible.

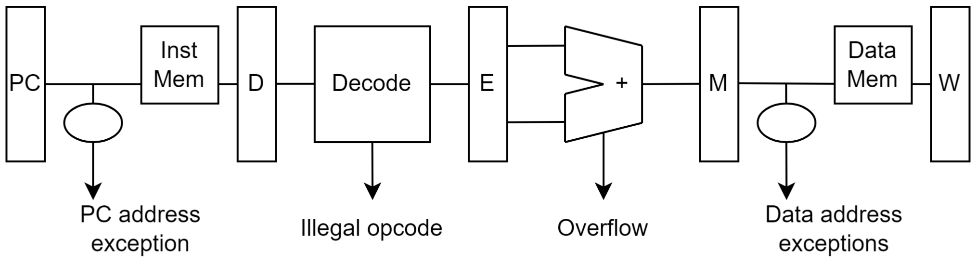


Figure 5.1: Exception origins

To address this, instructions in the pipeline can be tagged to indicate whether they cause exceptions or not. This tagging process waits until the memory stage concludes before flagging an exception. Interrupts are then represented as No-Operation (NOP) placeholders inserted into the pipeline instead of regular instructions.

In cases where a NOP is flushed, it is assumed that the interrupt condition persists. However, managing interrupt conditions becomes complex due to requirements such as supervisor mode switching and saving one or more program counters (PCs).

Additionally, optimizing instruction fetch to start fetching instructions from the interrupt vector may be challenging due to these complexities.

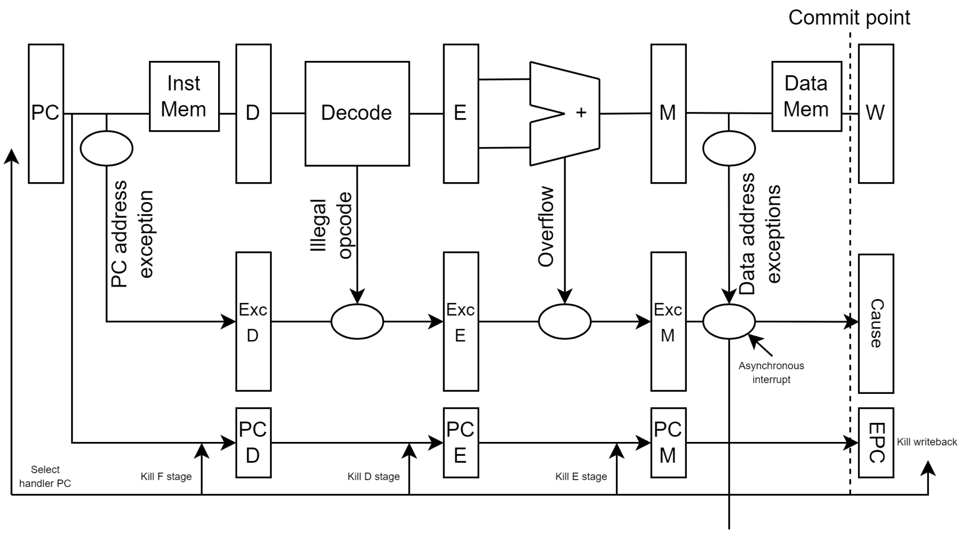


Figure 5.2: Exception handling

Maintain exception flags within the pipeline until reaching the commit point (M stage). Exceptions occurring in earlier pipeline stages take precedence over later ones for a particular instruction. External interrupts are injected at the commit point, overriding any other exceptions present.

If an exception occurs at the commit point, cause and EPC registers are updated, all pipeline stages are terminated, and the handler PC is injected into the fetch stage.

Jim Smith's paper explores various techniques for achieving precise interrupts, including in-order instruction completion, reorder buffer, and history buffer methodologies.

5.4.1 Remarks

The possible methods to anticipate the exceptions are:

- *Prediction mechanism*: due to the infrequent occurrence of exceptions, a straightforward prediction of no exceptions yields high accuracy.
- *Verification of prediction mechanism*: exceptions are identified at the conclusion of the instruction execution pipeline, utilizing specialized hardware tailored for different exception types.
- *Recovery mechanism*:architectural state is exclusively written at the commit point, enabling the discarding of partially executed instructions after an exception. Following the exception, the pipeline is flushed, and the exception handler is initiated.
- *Bypassing*: bypassing facilitates the utilization of uncommitted instruction outcomes by subsequent instructions.

CHAPTER 6

Branch prediction

6.1 Introduction

Every branch incurs a single stall to fetch the correct instruction flow: either the (PC+4) or the branch target address. Our goal is to predict the outcome of a branch instruction as early as possible to enhance performance. The performance of a branch prediction technique depends on three key factors:

1. *Accuracy*: this is determined by the percentage of incorrect predictions made by the predictor.
2. *Cost of misprediction*: this refers to the time lost executing unnecessary instructions due to an incorrect prediction (misprediction penalty). This cost increases notably in deeply pipelined processors.
3. *Branch frequency*: the frequency of branches within the application plays a significant role. Accurate branch prediction is particularly crucial in programs with higher branch frequencies.

Various methods address the performance degradation resulting from branch hazards:

- *Static branch prediction techniques*: predefined actions for each branch remain constant throughout program execution, determined at compile time.
- *Dynamic branch prediction techniques*: decisions leading to branch prediction may alter during program execution.

In both approaches, caution is essential to avoid altering the processor state until the branch outcome is definitively determined.

6.2 Static branch prediction techniques

Static Branch Prediction is employed in processors where it's anticipated that the branch behavior remains highly predictable at compile time. Additionally, Static Branch Prediction can complement dynamic predictors. Several techniques fall under Static Branch Prediction:

- *Branch always not taken* (predicted-not-taken).
- *Branch always taken* (predicted-taken).
- *Backward taken forward not taken* (BTFNT).
- *Profile-driven prediction.*
- *Delayed branch.*

6.2.1 Branch always not taken

In Static Branch Prediction, when assuming that the branch will not be taken, the sequential instruction flow that has been fetched can proceed as if the branch condition was not met.

- If the condition in the instruction decode (ID) stage indicates that the branch will not be taken (correct prediction), performance is preserved.
- However, if the condition in the ID stage indicates that the branch will be taken (incorrect prediction), the next instruction that has already been fetched must be flushed (turned into a nop), and execution restarts by fetching the instruction at the branch target address, incurring a one-cycle penalty.

6.2.2 Branch always taken

An alternative approach is to treat every branch as taken. Once the branch is decoded and the branch target address is computed, we assume the branch is taken and immediately commence fetching and executing at the target.

This predicted-taken scheme is advantageous for pipelines where the branch target is ascertainable before the branch outcome. However, in the MIPS pipeline, the branch target address isn't determined earlier than the branch outcome. Hence, there is no benefit in adopting this approach for this particular pipeline.

6.2.3 Backward taken forward not taken

The prediction relies on the direction of the branch:

- Backward-going branches are anticipated as taken.
- Forward-going branches are expected as not taken.

6.2.4 Profile-driven prediction

Branch prediction relies on profiling information gathered from previous executions. This approach can incorporate compiler hints to enhance prediction accuracy.

6.2.5 Delayed branch

The compiler statically arranges an independent instruction in the branch delay slot. The instruction in the branch delay slot executes regardless of whether the branch is taken or not. Assuming a branch delay of one cycle (as in MIPS), there is typically only one delay slot. While some deeply pipelined processors may have longer branch delays, most processors with delayed branches feature a single delay slot, as it is often challenging for the compiler to fill more than one delay slot.

For MIPS, the compiler consistently schedules an independent instruction after the branch. For instance, a preceding add instruction without any impact on the branch may be placed in the branch delay slot.

The behavior of the delayed branch remains consistent irrespective of whether the branch is taken or not:

- If the branch is untaken, execution proceeds with the instruction following the branch.
- If the branch is taken, execution continues at the branch target.

The compiler's task is to ensure the instruction placed in the branch delay slot is valid and beneficial. There are three typical strategies for scheduling the branch delay slot:

1. *From before*: an independent instruction from before the branch is placed in the branch delay slot. The instruction in the branch delay slot is always executed, regardless of whether the branch is taken or not.
2. *From target*: the branch delay slot is filled with an instruction from the target of the branch. This is often accompanied by using a register (e.g., \$1) in the branch condition to prevent certain instructions (e.g., add) from being moved after the branch. This strategy is preferred for branches that are likely to be taken, such as loop branches (backward branches).
3. From fall-through: The branch delay slot is occupied by an instruction from the fall-through path, where the branch is not taken. Similar to the from target strategy, this may involve using a register (e.g., \$1) in the branch condition to control instruction placement. This strategy is favored for branches that are unlikely to be taken, such as forward branches.

For optimizations to be valid in both target and fall-through cases, it must be acceptable to execute the moved instruction when the branch takes an unexpected direction. This means that while the instruction in the branch delay slot is executed, the work done by it is wasted, yet the program still executes correctly. For instance, if the destination register is an unused temporary register when the branch takes an unexpected direction.

Generally, compilers are able to fill approximately 50% of delayed branch slots with valid and useful instructions, with the remaining slots filled with nops. However, in deeply pipelined processors where delayed branches span multiple cycles, it becomes more challenging to populate all slots with useful instructions. The primary constraints on delayed branch scheduling stem from:

- Limitations on the instructions that can be scheduled in the delay slot.
- The compiler's capability to accurately predict the branch outcome statically.

To enhance the compiler's ability to populate the branch delay slot, many processors have introduced a canceling or nullifying branch. This instruction indicates the predicted branch direction:

- When the branch behaves as predicted, the instruction in the delay slot executes normally.
- However, if the branch prediction is incorrect, the instruction in the delay slot is replaced with a nop (flushed).

This approach enables the compiler to be less conservative when filling the delay slot.

6.3 Dynamic branch prediction techniques

The main idea of dynamic branch prediction techniques is to leverage past branch behavior to forecast future outcomes.

We utilize hardware to dynamically anticipate branch outcomes, where predictions adapt based on real-time branch behavior during execution.

Initially, we implement a basic prediction scheme and then explore methods to enhance prediction accuracy. Dynamic branch prediction integrates two key mechanisms:

- *Branch outcome predictor*: determines the likelihood of branch direction (taken or not taken) based on historical behavior.
- *Branch target predictor*: forecasts the target address for a taken branch.

These prediction modules collaborate within the instruction fetch unit, aiding in the prediction of the subsequent instruction to fetch from the instruction cache:

- If the branch isn't taken, the Program Counter (PC) increments.
- In the case of a taken branch, the BTP provides the target address.

6.3.1 Branch outcome predictor

Branch history table The branch history table is a table featuring one bit for each entry, indicating whether a branch was recently taken or not. It is indexed by the lower portion of the branch instruction's address.

Prediction involves assuming the correctness of a hint, initiating fetching in the predicted direction. Should the hint prove incorrect, the prediction bit is flipped and stored anew. Subsequently, the pipeline is cleared, and the accurate sequence is executed.

This table lacks tags, resulting in every access being a hit. Moreover, it's plausible for the prediction bit to have been set by another branch sharing the same lower-order address bits; however, this doesn't impact prediction accuracy as it merely serves as a hint.

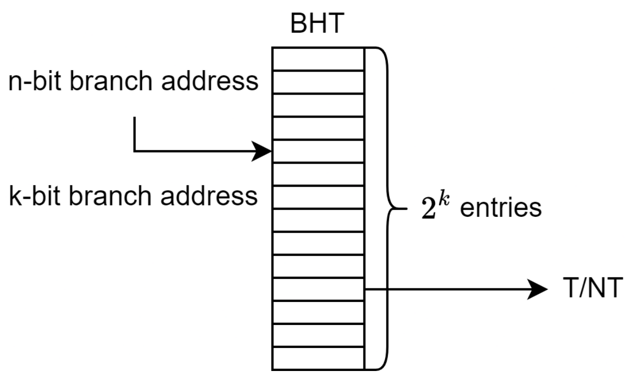


Figure 6.1: Branch history table structure

Accuracy A misprediction arises when either the prediction is erroneous for a particular branch or when the same index has been referenced by two distinct branches, and the previous history pertains to the other branch. To mitigate this issue, increasing the number of rows in the BHT or employing a hashing function (e.g., as in GShare) can be effective solutions.

One-bit branch history table The 1-bit BHT exhibits a notable limitation in scenarios such as loop branches. Even if a branch is predominantly taken throughout a loop but is not taken once, the 1-bit BHT may mispredict twice instead of once. This scheme results in two erroneous predictions:

- At the conclusion of the loop iteration, where the prediction bit indicates a taken branch, contradicting the need to exit the loop.
- Upon re-entering the loop, following the first loop iteration's end, the branch should be taken to maintain loop execution. However, the prediction bit suggests exiting the loop, stemming from the previous execution of the loop's final iteration where the prediction bit was flipped.

Two-bit branch history table In the two-bit branch history table the prediction must fail twice before it undergoes alteration. Within a loop branch, on the final iteration, there's no requirement to modify the prediction. Each index in the table employs 2 bits to encode the four states of a finite state machine.

N-bit branch history table For the n-bit Branch History Table, we require an n-bit saturating counter for each entry in the prediction buffer. This counter has a range of values between 0 and $2^n - 1$. When the counter equals or exceeds half of its maximum value $(2^n - 1)$, the branch is predicted as taken; otherwise, it's predicted as untaken. Similar to the 2-bit scheme, the counter increments with a taken branch and decrements with an untaken branch. Research on n-bit predictors indicates that 2-bit predictors perform nearly as effectively.

6.3.2 Branch target predictor

Branch target buffer Branch target buffer (BTB), also known as branch target predictor, functions as a cache that stores the anticipated branch target address for the instruction following a branch. During the Instruction Fetch (IF) stage, the BTB is accessed using the instruction address of the fetched instruction, which could potentially be a branch, to index the cache. Typical entries in the BTB include:

- The exact address of a branch.
- The predicted target address.

The predicted target address is typically represented as PC-relative.

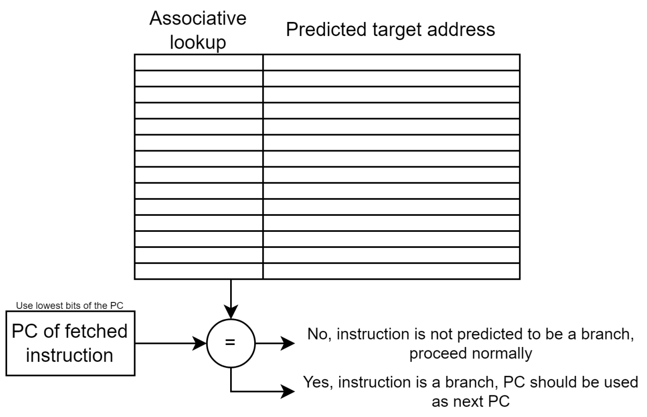

Associative lookup
Predicted target address
Use lowest bits of the PC
PC of fetched instruction
=
No, instruction is not predicted to be a branch, proceed normally
Yes, instruction is a branch, PC should be used as next PC

Figure 6.2: Branch target buffer structure

6.3.3 Correlating branch predictors

BHT predictors rely solely on the recent behavior of a single branch to forecast its future behavior.

Correlating branch predictors, on the other hand, operate on the principle that recent branches exhibit correlations. This means that the recent behavior of not just the current branch under consideration but also other branches can influence the prediction of the current branch.

Predictors that leverage the behavior of other branches to make predictions are known as correlating predictors or 2-level predictors.

In a general (m, n) correlating predictor, the system stores the last m branches to select from 2^m branch history tables (BHTs), with each BHT being an n-bit predictor.

Accuracy A 2-bit predictor lacking global history can be regarded as a $(0, 2)$ predictor. Comparing the performance of a 2-bit simple predictor featuring four thousand entries with that of a $(2, 2)$ correlating predictor with one thousand entries. The (2,2) predictor not only surpasses the simple 2-bit predictor having the same total number of bits (four thousand total bits), but it frequently outperforms a 2-bit predictor regardless of the number of entries.

Two-level adaptive branch predictors The first level history is stored in one or more k-bit shift registers known as the branch history register (BHR), which records the outcomes of the k most recent branches. The second level history is stored in one or more tables called pattern history table (PHT), consisting of two-bit saturating counters. The BHR is utilized to index the PHT to determine which 2-bit counter to use.

Once the two-bit counter is selected, the prediction is made using the same method as in the two-bit counter scheme. We may have:

- *BHT*: local predictor, indexed by the low-order bits of the program counter (branch address).

- *GAs*: local and global predictor, a 2-level predictor: PHT indexed by the content of BHR (global history).
- *GShare*: local XOR global information, indexed by the exclusive OR of the low-order bits of the program counter (branch address) and the content of BHR (global history).

CHAPTER 7

Hardware-level parallelism

7.1 Introduction

Instruction-level parallelism refers to the potential overlap of execution among unrelated instructions. Overlap is achievable when:

- There are no structural hazards.
- There are no stalls due to Read-After-Write (RAW), Write-After-Read (WAR), or Write-After-Write (WAW) dependencies.
- There are no stalls due to control dependencies.

7.1.1 Pipeline evaluation

The CPI of a pipeline is computed as:

$$\text{CPI}_{pipeline} = \text{CPI}_{ideal\ pipeline} + \text{Stalls}_{structural} + \text{Stalls}_{data\ hazard} + \text{Stalls}_{control}$$

Here:

- Ideal pipeline CPI represents the maximum performance achievable by the implementation.
- Structural hazards occur when the hardware cannot support a specific combination of instructions.
- Data hazards arise when an instruction depends on the result of a prior instruction still in the pipeline.
- Control hazards are caused by delays between the fetching of instructions and decisions regarding changes in control flow, such as branches, jumps, or exceptions.

Hazards constrain performance in several ways:

- Structural hazards necessitate additional hardware resources.
- Data hazards require mechanisms like forwarding and compiler scheduling.

- Control hazards can be mitigated through early evaluation, program counter adjustments, delayed branch techniques, and predictors.

As the length of the pipeline increases, the impact of hazards becomes more significant.
Pipelining primarily enhances instruction bandwidth rather than reducing latency.

7.1.2 Data hazard

Data hazards can occur in the following scenarios:

- *Read-after-write* (RAW): when an instruction attempts to read data from a register or memory location that has been modified by a preceding instruction.
- *Write-after-read* (WAR): when an instruction attempts to write data to a register or memory location before it has been read by a preceding instruction.
- *Write-after-write* (WAW): when two instructions attempt to write to the same register or memory location in quick succession without an intervening read operation.

7.2 Complex pipelining

Pipelining complexity increases significantly when aiming for high performance, especially in the presence of:

- Long latency or partially pipelined floating-point units.
- Multiple function and memory units.
- Memory systems with variable access time.
- Precise exception handling.

One potential solution is the complex in order pipeline:

- Introduce a delay in the write-back stage to ensure all operations have the same latency to the write-back stage.
- Avoid over subscription of write ports, allowing for one instruction to enter and one instruction to exit every cycle.
- Enforce in-order instruction commitment, which simplifies the implementation of precise exception handling.

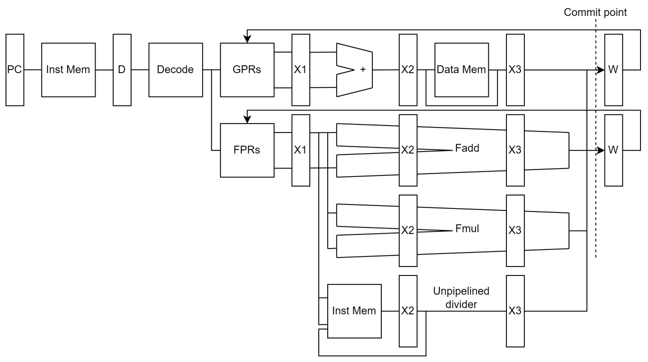


Figure 7.1: Complex in order pipeline

Issues Structural conflicts arise during the execution stage if certain Floating Point Units (FPU) or memory units are not pipelined, causing them to take longer than one cycle. There are also structural conflicts at the write-back stage because different functional units may have variable latencies. Additionally, out-of-order write hazards occur due to the variable latencies of different functional units.

7.3 CPI and dependencies

In a pipelined machine, actual CPI is derived as:

$$\text{CPI}_{pipelined} = \text{CPI}_{ideal} + \text{Stalls}_{structural} + \text{Stalls}_{data\ hazard} + \text{Stalls}_{control}$$

Decreasing any term on the right-hand side reduces the CPI (Cycles Per Instruction) of the pipeline, thereby increasing the Instructions Per Clock (IPC). However, techniques aimed at increasing the CPI of the pipeline might exacerbate issues related to hazards.

To achieve higher performance within a given technology, it's imperative to extract more parallelism from the program. In essence, this entails detecting and resolving dependencies, as well as ordering (scheduling) instructions to attain the highest possible execution parallelism that aligns with the available resources.

Dependencies Establishing dependencies among instructions is crucial for identifying the level of parallelism present in a program. If two instructions exhibit dependence, they cannot execute concurrently; rather, they must be executed sequentially or only partially overlapped. There are three distinct types of dependencies:

- Name dependencies.
- Data dependencies (also known as true data dependencies).
- Control dependencies.

7.3.1 Name dependencies

A name dependence arises when two instructions utilize the same register or memory location (referred to as a name), but there's no flow of data between these instructions associated with that name.

There are two types of name dependencies between an instruction i preceding instruction j in program order:

- Anti-dependence: occurs when j writes to a register or memory location that instruction i reads. Preserving the original instruction ordering ensures that i reads the correct value.
- Output dependence: arises when both i and j write to the same register or memory location. Preserving the original instruction ordering guarantees that the value eventually written corresponds to j.

Name dependencies are distinct from true data dependencies as there is no transmission of values (no data flow) between instructions.

If the name (register number or memory location) used in the instructions could be altered, the instructions do not conflict.

Detecting dependencies through memory locations poses more challenges (referred to as the memory disambiguation problem) since two addresses may refer to the same location but appear different. Register renaming is comparatively easier to implement.

Register renaming can be accomplished either statically by the compiler or dynamically by the hardware.

7.3.2 Data dependencies

A data/name dependence can potentially generate a data hazard (RAW, WAW, or WAR), but the actual hazard and the number of stalls to eliminate the hazards are a property of the pipeline. – RAW hazards correspond to true data dependences. – WAW hazards correspond to output dependences – WAR hazards correspond to antidependences. • Dependences are a property of the program, while hazards are a property of the pipeline.

7.3.3 Control dependencies

Control dependence determines the ordering of instructions and it is preserved by two properties: – Instructions execution in program order to ensure that an instruction that occurs before a branch is executed before the branch. – Detection of control hazards to ensure that an instruction (that is control dependent on a branch) is not executed until the branch direction is known. • Although preserving control dependence is a simple way to preserve program order, control dependence is not the critical property that must be preserved.

7.4 Parallelism

Two essential properties ensure program correctness and are typically preserved by maintaining both data and control dependencies:

- *Exception behavior*: preserving exception behavior ensures that any alterations in the execution order of instructions do not affect how exceptions are triggered within the program.
- *Data flow*: this pertains to the actual movement of data values among instructions, ensuring that correct results are produced and consumed.

There are two strategies to support instruction-level parallelism:

- *Dynamic scheduling*: this approach relies on hardware to identify and exploit parallelism within the program.
- *Static scheduling*: this method depends on software to identify potential parallelism beforehand.

In the desktop and server markets, hardware-intensive approaches tend to dominate.

7.4.1 Dynamic scheduling

Hardware reorders instruction execution to mitigate pipeline stalls while still upholding data flow and exception behavior.

Advantages The main advantages of this approach include:

- Handling cases where dependencies are unknown at compile time.
- Simplifying compiler complexity.
- Facilitating efficient execution of compiled code on different pipeline architectures.

Disadvantages However, these advantages come with certain costs:

- Significant increase in hardware complexity.
- Higher power consumption.
- Potential for imprecise exceptions.

In essence, instructions are fetched and issued in program order (in-order issue). Execution initiates as soon as operands become available, potentially allowing out-of-order execution. Note that out-of-order execution is feasible even in pipelined scalar architectures.

Out-of-order execution introduces the possibility of write-after-read and write-after-write data hazards. Additionally, out-of-order execution implies out-of-order completion unless a re-order buffer is present to ensure in-order completion.

7.4.2 Static scheduling

Compilers employ sophisticated algorithms for code scheduling to harness instruction-level parallelism. However, the instruction-level parallelism available within a basic block—a straight-line code sequence with minimal branching—is often limited.

To achieve significant performance improvements, instruction-level parallelism must be exploited across multiple basic blocks, transcending branches.

Static scheduling involves the compiler's detection and resolution of dependencies by reordering code to avoid dependencies. The compiler's output typically comprises dependency-free code, which is particularly suited for architectures like Very Long Instruction Word (VLIW) processors.

However, there are limits to instruction-level parallelism exploitation:

- Unpredictable branches.
- Variable memory latency, such as unpredictable cache misses.
- Code size explosion.
- Increased compiler complexity.

7.5 Scoreboard

Imagine a scenario where a data structure maintains records of all instructions across functional units. Prior to the issue stage dispatching an instruction, several checks must be conducted:

1. Availability of the required functional unit.
2. Availability of input data, considering Read-After-Write (RAW) hazards.

3. Safety in writing to the destination, taking into account Write-After-Read (WAR) and Write-After-Write (WAW) hazards.
4. Detection of structural conflicts at the Write-Back (WB) stage.

We introduce a data structure called the Correct Issues Table to keep track of the status of functional units. The fields are as follows:

Name	**Busy**	**Op**	**Dest**	**Src1**	**Src2**
Int					
Mem					
Add1					
Add2					
Add3					
Mult1					
Mult2					
Div					

At the issue stage, the instruction i consults this table using the following rules:

- To check if the required functional unit is available, consult the busy column.
- For Read-After-Write (RAW) hazard detection, search the destination column for i's sources.
- For Write-After-Read (WAR) hazard detection, search the source columns for i's destination.
- For Write-After-Write (WAW) hazard detection, search the destination column for i's destination.

An entry is added to the table if no hazard is detected. An entry is removed from the table after the Write-Back stage.

Dynamic scheduling idea In the context of computer architecture and pipelined execution, dynamic scheduling addresses the issue of hardware stalls caused by data dependencies that cannot be resolved through bypassing or forwarding mechanisms.

Instead of letting instructions stall the pipeline, dynamic scheduling allows the hardware to rearrange the execution order of instructions dynamically. This enables instructions that are not dependent on stalled instructions to proceed, thus reducing pipeline stalls and improving overall performance.

Dynamic scheduling facilitates out-of-order execution, where instructions are executed as soon as their operands are available, regardless of their original order in the program. However, it also introduces the need to handle hazards such as Write-After-Read (WAR) and Write-After-Write (WAW) data hazards, which may arise due to the out-of-order execution.

The concept of dynamic scheduling was first implemented in the CDC6600 supercomputer in 1963, marking a significant advancement in computer architecture by allowing for more efficient utilization of hardware resources and better performance in pipelined execution.

7.5.1 CDC6600 scoreboard

The CDC6600 Scoreboard is designed to manage instruction execution efficiently while ensuring data dependencies are handled properly. Here's a revised version of its key features:

- Instructions are dispatched in sequential order to functional units, as long as there are no structural hazards or Write-After-Write (WAW) hazards.
- If a structural hazard occurs or no functional units are available, the execution is stalled.
- Each register has only one pending write operation at a time.
- Instructions may execute out-of-order to handle Read-After-Write (RAW) hazards, waiting for input operands before execution.
- To prevent Write-After-Read (WAR) hazards, instructions wait until preceding instructions have read the output register. The result remains in the functional unit until the register is free for writing.

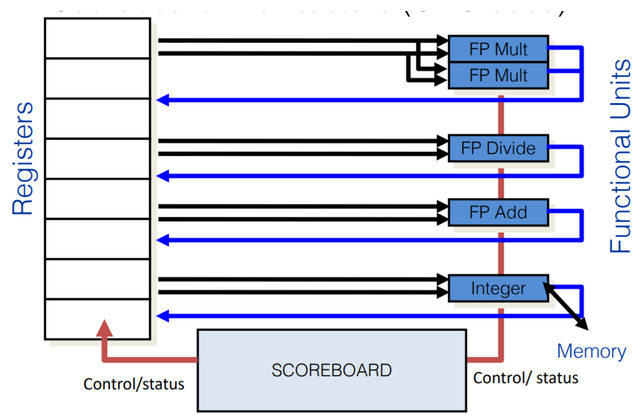


Figure 7.2: CDC6600 scoreboard

Scoreboard operation The Scoreboard serves as the hub for hazard management in the new pipeline architecture:

- All instructions are routed through the Scoreboard for processing.
- It orchestrates the timing for instructions to access their operands and commence execution.
- Continuously monitors hardware alterations, facilitating the resolution of stalled instructions for execution.
- Governs the timing for instructions to commit their results.

This centralized system optimizes performance and ensures efficient instruction flow within the updated pipeline framework.

Scoreboard scheme The ID stage is now bifurcated into two distinct phases:

1. Issue: responsible for instruction decoding and structural hazard verification.
2. Read operands: holds instructions until data hazards are resolved, with an out-of-order execution approach.

While instructions are issued in sequence, the reading of operands occurs out of order. This innovative approach, facilitated by the Scoreboard, enables the execution of instructions without dependencies, enhancing overall efficiency and throughput.

Scoreboard control The four stages of scoreboard control are:

1. Issue: decoding instructions and detecting structural hazards. Instructions are decoded and checked for hazards in program order. If the functional unit corresponding to the instruction is available and there is no other active instruction with the same destination register (WAW hazard), the scoreboard dispatches the instruction to the functional unit and updates its internal state. If either a structural hazard or a WAW hazard is detected, the instruction issuing process is stalled. No further instructions will be dispatched until these hazards are resolved.
2. Reading operands: wait until there are no data hazards, then proceed to read operands. A source operand is deemed available if no earlier issued active instruction will modify it, or a functional unit is currently writing its value into a register. Once the source operands are available, the scoreboard instructs the functional unit to proceed with reading the operands from the registers and commence execution. RAW (Read After Write) hazards are dynamically resolved during this stage, enabling instructions to be executed out of order. Note that there is no data forwarding in this model.
3. Execution: the functional unit initiates operations on the provided operands. Upon completion of the operation, it informs the scoreboard about the execution's conclusion. Functional Units (FUs) are defined by:
   - Latency: the time required to complete one operation effectively.
   - Initiation interval: the number of cycles necessary to elapse between issuing two operations to the same functional unit.
4. Write result: upon completion of execution, the scoreboard checks for Write After Read (WAR) hazards. If there are no WAR hazards detected, it proceeds to write the results. However, if a WAR hazard is identified, the instruction is stalled. Assuming overlap between issuing instructions and writing results is permitted.

Implications The scoreboard plays a crucial role in managing hazards and dependencies in the instruction pipeline. Without register renaming, it must detect Write After Write (WAW) hazards and stall the issuance of new instructions until the conflicting instruction completes execution. To accommodate multiple instructions in the execution phase, the system requires either multiple execution units or pipelined execution units. The scoreboard maintains the state of operations and tracks dependencies to ensure correct execution sequencing and hazard detection.

Scoreboard structure The structure if the scoreboard is composed by:

1. Instruction status.
2. Functional unit status: indicates the state of the functional unit (FU):
3.
    - Busy: indicates whether the unit is busy or not.
    - Operation (Op): the operation to be performed in the unit (+, -, etc.).
    - Destination register (Fi).
    - Source register numbers (Fj, Fk).
    - Functional units producing source registers (Qj, Qk).
    - Flags indicating when Fj, Fk are ready (Rj, Rk).
4. Register result status: indicates which functional unit will write each register. It's left blank if no pending instructions will write that register.

7.5.2 Summary

The key idea behind the Scoreboard is to enable instructions behind a stall to proceed, allowing for the decoding, issuing instructions, and reading operands. This design achieves a speedup of 2.5 compared to no dynamic scheduling. Additionally, a speedup of 1.7 is achieved by reorganizing instructions at the compiler level. However, the benefits are limited by the slow memory (lacking cache) of the CDC 6600. Limitations of the CDC 6600 scoreboard include:

- Lack of forwarding hardware.
- Limited to instructions within a basic block, leading to a small window.
- A few functional units, particularly in integer and load/store units, leading to structural hazards.
- Instructions are not issued on structural hazards.
- The scoreboard waits for Write After Read (WAR) hazards.
- Measures are taken to prevent Write After Write (WAW) hazards.

7.6 Tomasulo algorithm

The Tomasulo Algorithm is a dynamic approach designed to enable execution to continue even in the presence of dependencies. It was developed at IBM, emerging three years after the CDC 6600, primarily for the IBM 360/91. Like its predecessors, its aim is to achieve high performance without the need for specialized compilers.

In Tomasulo's design, both the control logic and buffers are decentralized, contrasting with the centralized approach of a scoreboard. Buffers for operands are termed reservation stations, with each instruction represented as an entry within these stations.

Register renaming In the Tomasulo Algorithm, operands are substituted with values or pointers, a technique known as Register Renaming. This approach helps in circumventing Write-After-Read and Write-After-Write hazards. Unlike traditional registers, reservation stations in Tomasulo are more versatile, enabling optimizations beyond the capabilities of a compiler.

Results are disseminated to other Functional Units via a Common Data Bus, which carries both data and its corresponding source. Additionally, Load/Store operations are treated as Functional Units within the algorithm.

7.6.1 Tomasulo algorithm

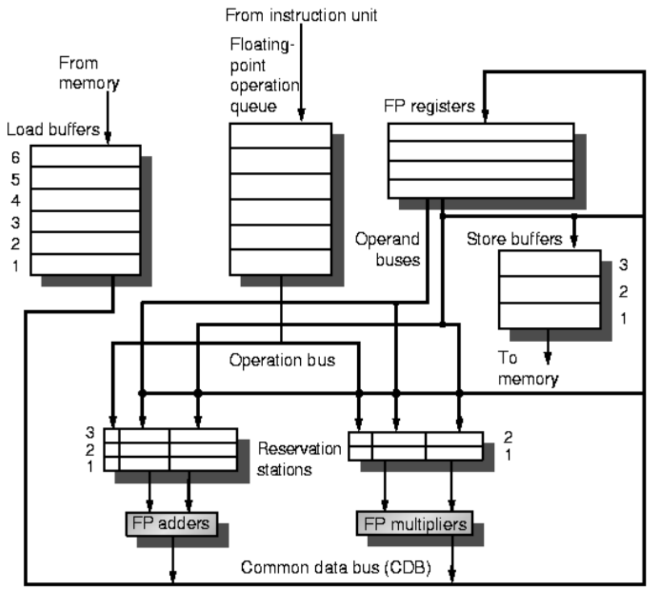


Figure 7.3: Illustration of the Tomasulo Algorithm for a Floating Point Unit

Reservation station A reservation station is equipped with the following components:

- *Tag*: identifies the reservation station.
- *OP*: specifies the operation to be executed on the component.
- V_j, V_k: values of the source operands.
- Q_j, Q_k: pointers to reservation stations that produce V_j, V_k.
- *Zero value*: indicates that the source operand is already available in either V_j or V_k.
- *Busy*: indicates whether the reservation station is currently occupied.

It's important to note that for each operand, only one of the V-field or Q-field is valid.

Additional Components Aside from reservation stations, there are other essential elements within the Tomasulo Algorithm:

- *Register file and the store buffer*: both feature a Value (V) and a Pointer (Q) field. The Pointer (Q) field corresponds to the number of the reservation station producing the result to be stored in the register file or store buffer. If it's zero, it indicates no active instructions producing the result, implying that the register file or store buffer content is correct.
- *Load buffers*: these buffers include an address field (A) and a busy field. They are utilized for holding information related to memory address calculation for load/store operations. Initially, they contain the instruction offset (immediate field). After address calculation, they store the effective address.
- *Store buffers*: similar to load buffers, store buffers also feature an address field (A). They are instrumental in managing store operations within the algorithm.

Algorithm The Tomasulo algorithm unfolds across three main stages:

- *Issue*: retrieve an instruction I from the instruction queue. If it's a floating-point operation, check if a reservation station is available (i.e., check for structural hazards). Perform register renaming and resolve Write-After-Read hazards. Handle Write-After-Write hazards by linking the register file to the instruction I due to in-order issuance.
- *Execution*: once both operands are available, execute the instruction. If operands aren't ready, monitor the common data bus for results. Delay execution until operands are ready to avoid Read-After-Write hazards. Multiple instructions may become ready simultaneously for the same Functional Unit. For load and store instructions, perform a two-step process: compute the effective address and store it in the load or store buffer. Loads in the load buffer execute as soon as the memory unit is available, while stores in the store buffer wait until the value is ready to be stored before being sent to the memory unit.
- *Write*: once the result is available, write it onto the common data bus. From there, distribute it into the register file and all reservation stations (including store buffers) waiting for this result. Stores write data to memory during this stage and mark reservation stations as available for the next instructions.

Load and store Loads and stores undergo effective address computation in a functional unit before proceeding to their respective effective load and store buffers. Loads require a second execution step to access memory before transitioning to the Write Result stage to send the value from memory to the register file and/or reservation stations.

Stores complete their execution in their Write Result stage, which involves writing data to memory. All write operations occur in the Write Result stage, simplifying the Tomasulo algorithm.

Load and Store operations can be executed in a different order, provided they access different memory locations. Otherwise, hazards such as Write-After-Read or Read-After-Write may occur (with Write-After-Write if two stores are interchanged). Loads can be freely reordered.

To detect such hazards, the data memory addresses associated with any earlier memory operation must have been computed by the CPU.

When a Load executes out of order with a previous Store, it's assumed that the address was computed in program order. Once the Load address is computed, it's compared with the address fields in active store buffers. If there's a match, the Load isn't sent to the Load buffer until the conflicting store completes.

Stores must check for matching addresses in both load and store buffers (dynamic disambiguation), an alternative to the static disambiguation performed by the compiler. However, this approach requires a significant amount of hardware. Each reservation station must contain a fast associative buffer, and a single common data bus may limit performance.

7.6.2 Tomasulo and scoreboard comparison

The key features of Tomasulo algorithm are:

- Issue window size of 14.
- No issue on structural hazards.
- Write-After-Read and Write-After-Write hazards are avoided with renaming.
- Results are broadcasted from functional units.
- Control is distributed across reservation stations.
- Allows loop unrolling in hardware.

The key features of scoreboard algorithm are:

- Issue window size of 5.
- No issue on structural hazards.
- Stall the completion for WAW and WAR hazards.
- Results are written back on registers.
- Control is centralized through the Scoreboard.

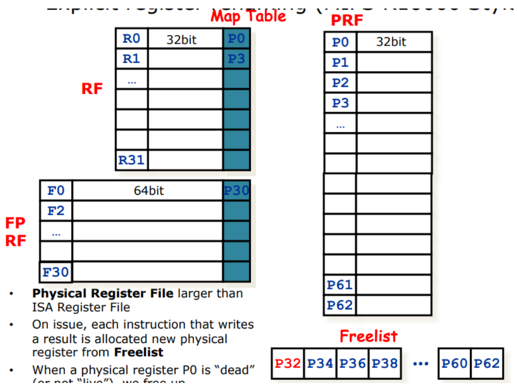
Both approaches distribute control and buffers with functional units versus centralizing them in the scoreboard. Reservation stations in Tomasulo serve as buffers for pending operands. Registers in instructions are replaced by values or pointers to reservation stations, a technique known as register renaming, which helps avoid WAR and WAW hazards. Tomasulo can utilize more reservation stations than registers, enabling optimizations beyond compilers' capabilities.

Results are sent to functional units from reservation stations, not through registers, via a Common Data Bus that broadcasts results to all functional units. Additionally, both load and stores are treated as functional units with associated Reservation Stations.

Furthermore, in Tomasulo, integer instructions can proceed past branches, allowing floating-point operations beyond the basic block in the floating-point queue.

7.7 Explicit register renaming

Tomasulo's approach involves Implicit Register Renaming, where user registers are renamed to reservation station tags. Now, let's introduce Explicit Register Renaming, which utilizes a physical register file larger than the number specified by the Instruction Set Architecture. The main idea is to allocate a new physical destination register for each instruction that writes. This resembles a compiler technique known as Static Single Assignment (SSA) form, but implemented in hardware. This strategy eliminates all possibilities of Write-After-Read (WAR) or Write-After-Write (WAW) hazards, similar to Tomasulo's method, making it beneficial for enabling full out-of-order completion.



- **Physical Register File** larger than ISA Register File
- On issue, each instruction that writes a result is allocated new physical register from **Freelist**
- When a physical register P0 is "dead"

Figure 7.4: Explicit register renaming

Expanding the Physical Register File beyond the size of the Instruction Set Architecture (ISA) Register File involves several steps:

1. Upon issuance, every instruction that generates a result is assigned a new physical register from the available free list.
2. When a physical register, denoted as P0, is no longer in use (i.e., it becomes "dead" or not "live"), it is released back to the free list for future allocations.

Explicit Register Renaming involves maintaining a translation table that maps ISA registers to physical registers. When an instruction writes to a register, its corresponding entry in the translation table is updated with a new register from the available pool. Physical registers are released when they are no longer utilized by any active instructions.

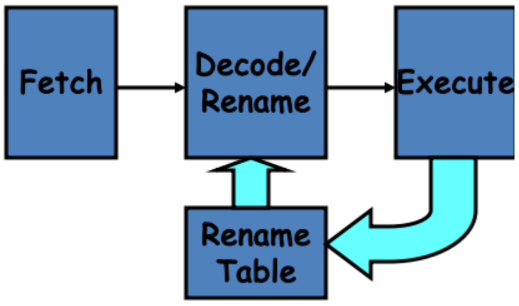


Figure 7.5: Explicit register renaming mechanism

Unified physical register file In the Unified Physical Register File approach, architectural registers are unified into a single physical register file during the decoding stage, without reading register values. Functional units interact with this unified register file, accessing and updating both committed and temporary registers during execution. The process of committing involves updating the mapping of architectural registers to physical registers without moving any data.

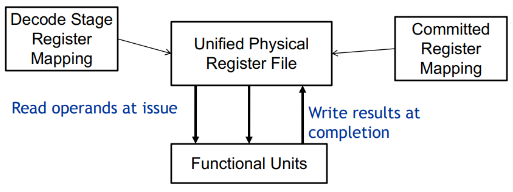

Decode Stage Register Mapping
Unified Physical Register File
Committed Register Mapping
Read operands at issue
Write results at completion
Functional Units

Figure 7.6: Unified Physical Register File

Instruction commit Instruction Commit involves several steps:

- It marks the mapping between an architectural register number and a physical register number as non-speculative, indicating that it's finalized.
- It releases any physical registers used to store the previous value of the architectural register.
- Deallocating registers is a bit complex:
  - Before freeing a physical register, it's necessary to ensure it no longer corresponds to an architectural register and there are no further outstanding uses of it.
  - A physical register remains associated with an architectural register until that register is overwritten.
  - However, there might still be outstanding uses of the physical register. The processor checks if any source operand in the functional units queue corresponds to that register. If not, it can be deallocated.

  Alternatively, the processor can wait until another instruction that writes to the same architectural register commits. This method may slightly prolong the usage of a physical register but is easier to implement.

7.7.1 Hardware register renaming

To perform hardware register renaming, the following components are necessary:

- *Renaming map*: this is a simple data structure that provides the physical register number corresponding to the requested architectural register.
- *Instruction commit*: this process involves permanently updating the renaming table to indicate that the physical register holding the destination value corresponds to the actual architectural register.

Using a ReOrder Buffer (ROB) enforces in-order commit. Use ROB to enforce in-order commit

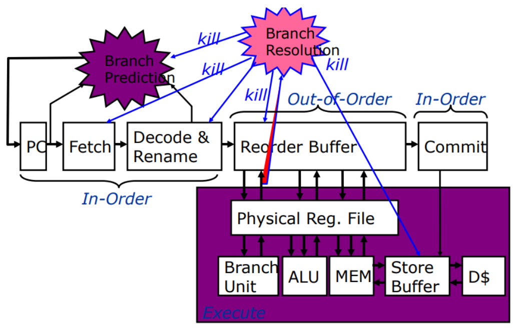


Figure 7.7: Hardware explicit register renaming

Advantages The main advantages include:

- *Decoupling renaming from scheduling*: this approach allows for flexibility in pipeline design. The pipeline can resemble a standard DLX pipeline with the possibility of issuing multiple operations per cycle. Alternatively, it could adopt methodologies like Tomasulo's algorithm or scoreboard-based approaches. Standard forwarding or bypassing techniques can be employed.
- *Single register file access*: data retrieval from a single register file is facilitated. This eliminates the need to bypass values from the reorder buffer, which can be crucial for maintaining pipeline balance.
- *Widespread adoption*: many processors have adopted variants of this technique, including the R10000, Alpha 21264, and HP PA8000 processors.

Support Ensuring swift access to a translation table involves deploying a physical register file with a capacity exceeding that specified by the Instruction Set Architecture (ISA). This setup enables quick determination of available physical registers, crucial for seamless operation. In instances where no registers are available, the system pauses during instruction issuance. Consequently, reservation stations are unnecessary for register renaming. However, many modern architectures combine explicit register renaming with Tomasulo-like reservation stations to effectively control execution flow.

7.7.2 Explicit register renaming and scoreboard

Explicit Register Renaming involves using a physical register file that surpasses the number of registers specified by the ISA. Here's how it works:

- *Translation table*: a translation table maintains the mapping between ISA registers and physical registers. When a register is written to, its entry in the table is updated with a new register from the free list. Physical registers are marked as free when not in use by any active instructions.
- *Pipeline structure*: the pipeline structure can mirror a standard DLX pipeline with stages like Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), etc.

The advantages are:

- Eliminates Write-After-Read (WAR) and Write-After-Write (WAW) hazards.

- Facilitates full out-of-order completion similar to Tomasulo's approach.
- Simplifies architectural design by enabling data retrieval from a single register file.
- Simplifies speculative execution and precise interrupt handling: only requires undoing the table mappings for precise breakpoint restoration.

Scoreboard Control Stages with Explicit Renaming:

1. *Issue*: decode instructions, check for structural hazards, and allocate new physical registers for results. Instructions are issued in program order for hazard checking. Issuance is withheld if there are no free physical registers or if there's a structural hazard.
2. *Read operands*: wait until no hazards, then read operands. All real dependencies (RAW hazards) are resolved in this stage since it waits for instructions to write back data.
3. *Execution*: functional units begin execution upon receiving operands. When the result is ready, it notifies the scoreboard.
4. *Write result*: execution finishes, and results are written.

Note that there are no checks for WAR or WAW hazards in this approach.

Register renaming and ROB When contrasting Register Renaming with ReOrder Buffer:

- Committing instructions is simpler in Register Renaming compared to ROB.
- However, deallocating registers poses more complexity in Register Renaming.
- The dynamic mapping of architectural to physical registers adds intricacy to the design and debugging processes.
- Register Renaming is employed in several architectures such as PowerPC603/604, Pentium II-III-4, MIPS 10000/12000, Alpha 21264, and Sandy-Bridge. These architectures typically incorporate an additional 20 to 80 registers.

7.7.3 Summary

In Explicit Renaming, there are more physical registers available than needed by the ISA. Here's a breakdown:

- *Rename table*: this keeps track of the current association between architectural registers and physical registers.
- *Translation table*: used to perform compiler-like transformations on-the-fly.

With Explicit Renaming:

- All registers are concentrated in a single register file.
- It allows for the utilization of a bypass network that resembles a 5-stage pipeline.
- Introduces a register-allocation problem that needs to be managed.
- Handling branch misprediction and precise exceptions differs from other methods, but ultimately simplifies processes.
- For precise exceptions and branch prediction, a mechanism like a reorder buffer is essential.

Multiple issue To handle multiple instructions simultaneously, including potential dependencies between them, is crucial for dynamically scheduled superscalar processors. This capability represents one of the fundamental bottlenecks in their design:

- *Issue logic complexity*: designing logic to manage all possible combinations of dependent instructions within a single clock cycle is challenging. As the number of instructions issued per cycle increases, the complexity grows exponentially.
- *Basic strategy*:
  - Assign a reservation station and a reorder buffer entry for each instruction in the next issue bundle. If unavailable, only a subset of instructions is considered sequentially.
  - Analyze all dependencies among the instructions.
  - Update the reservation table for dependent instructions using the assigned reorder buffer number if an instruction in the bundle relies on an earlier one within the same bundle.

This process is executed in parallel within a single clock cycle. Additionally, the ability to commit multiple instructions in one clock cycle is essential. Intel I7 processors employ a similar technique.

Multiple Instructions Issue with Register Renaming:

- The issue logic pre-reserves adequate physical registers for the entire issue bundle.
- It identifies dependencies within the bundle:
  - If no dependence exists within the bundle, the register renaming structure determines the physical register holding or will hold the required result from an earlier issue bundle.
  - If an instruction depends on an earlier one within the bundle, the pre-reserved physical register for the result is used to update the issuing instruction's information.

Again, all these operations are conducted simultaneously in a single clock cycle.

Superscalar Register Renaming: two-issue In the context of Superscalar Register Renaming with a two-issue capability:

- Allocation during decode: instructions are assigned new physical destination registers.
- Operand renaming: source operands are renamed to the physical register holding the most recent value.
- Execution unit view: execution units exclusively utilize physical register numbers.
- Hazard checking: it's imperative to inspect for Read-After-Write (RAW) hazards between instructions issuing in the same cycle. This verification can be conducted concurrently with the rename lookup process.

Speculation and energy efficiency Speculation tends to elevate power consumption while reducing execution time to a greater extent, ultimately resulting in overall energy savings. The total energy consumed may decrease depending on the number of incorrectly executed instructions. Empirical findings indicate that misspeculation in scientific code is generally minimal, whereas it can be more substantial, averaging around 30%, in integer code.

Summary Modern computer architects employ extensive prediction techniques for branches, data dependencies, and data access. While predicting branches can be achieved with relatively simple hardware structures such as the Branch Target Buffer (BTB) and Branch History Table (BHT), more sophisticated prediction methods like correlation are also utilized to handle interdependent branches.

Explicit Register Renaming involves employing more physical registers than specified by the ISA. This approach decouples renaming from scheduling, offering flexibility in resolving Read-After-Write (RAW) hazards. It relies on a rename table to track the current association between architectural and physical registers, although managing this table can be complex.

However, achieving parallelism efficiently from real hardware remains challenging despite these advancements.

7.8 Instruction level parallelism limits

We previously defined Instruction-Level Parallelism (ILP) as the potential overlap of execution among unrelated instructions. This overlapping becomes feasible under certain conditions:

- Absence of Structural Hazards.
- Absence of Read-After-Write (RAW), Write-After-Read (WAR), or Write-After-Write (WAW) stalls.
- No Control stalls.

7.8.1 Superscalar execution

Enabling more than one instruction to begin execution in each clock cycle requires:

- *Fetching more instructions per clock cycle*: this typically involves fetching multiple instructions simultaneously. As long as the instruction cache can handle the increased bandwidth and manage multiple requests concurrently, there are usually no major issues with this approach.
- *Managing data and control dependencies*: to handle data and control dependencies efficiently, dynamic scheduling and dynamic branch prediction mechanisms are crucial. These techniques allow the processor to dynamically schedule instructions and predict branch outcomes, facilitating the execution of multiple instructions concurrently.

Superscalar processors issue multiple instructions per clock cycle, with the number of instructions varying from one to eight. Instruction scheduling can be performed either by the compiler or by hardware mechanisms like Tomasulo's algorithm.

The success of superscalar architectures led to the adoption of Instructions Per Clock cycle (IPC) as a performance metric, contrasting with the traditional notion of Cycles Per Instruction (CPI). In an ideal scenario, where all available instruction slots are filled every cycle, CPIideal equals 1 divided by the issue width.

7.8.2 Limits

Assumptions for an ideal or perfect machine to begin with:

1. *Register renaming*: assume infinite virtual registers are available, and all Write-After-Write (WAW) and Write-After-Read (WAR) hazards are effectively avoided through renaming.
2. *Branch prediction*: assume perfect branch prediction, where there are no mispredictions.
3. *Jump Prediction*: assume all jumps are perfectly predicted, indicating a machine with flawless speculation and an unlimited buffer of instructions readily available.
4. *Memory-address alias analysis*: assume perfect knowledge of memory addresses, allowing movements of stores before loads as long as the addresses are not identical.
5. *Uniform latency*: assume a consistent one-cycle latency for all instructions, enabling unlimited instructions to be issued per clock cycle.

Initial assumptions Initial assumptions:

- The CPU can issue an unlimited number of instructions simultaneously, with the ability to look arbitrarily far ahead in computation.
- There are no restrictions on the types of instructions that can be executed in one cycle, including loads and stores.
- All functional unit latencies are assumed to be one cycle, and any sequence of dependent instructions can issue on successive cycles.
- Perfect caches are assumed, meaning all loads and stores execute in one cycle. This scenario only considers fundamental limits to Instruction-Level Parallelism (ILP).
- It's important to note that the results obtained from these assumptions are highly optimistic, as such a CPU cannot be realized in practice.
- Benchmark programs used include six from SPEC92, comprising three floating-point-intensive programs and three integer programs.

Limits on window size Dynamic analysis becomes imperative to approach perfect branch prediction, as achieving perfection at compile time is unattainable. For a perfectly dynamic-scheduled CPU, the following conditions should ideally be met:

1. The CPU should have the capability to look arbitrarily far ahead to identify a set of instructions to issue and predict all branches perfectly.
2. All register uses should be renamed to avoid Write-After-Write (WAW) and Write-After-Read (WAR) hazards.
3. The CPU should ascertain whether there are any data dependencies among instructions in the issue packet and rename them if necessary.
4. It should determine the presence of memory dependencies among issuing instructions and handle them accordingly.

5. Sufficient replicated functional units should be provided to enable all ready instructions to issue simultaneously.

The size of the window impacts the number of comparisons required to identify Read-After-Write (RAW) dependencies. In modern CPUs, constraints arise from the limited number of registers, coupled with the need to search for dependent instructions and adhere to in-order issue policies. Consequently, all instructions within the window must be retained in the processor.

Present-day CPUs typically feature window sizes ranging from 32 to 200, necessitating up to over 2400 comparisons to determine dependencies.

7.8.3 Current CPU Limitations

Several factors limit CPU performance:

- Number of functional units.
- Number of buses.
- Number of ports for the register file.

These constraints dictate the maximum number of instructions that can be issued, executed, or committed in a single clock cycle, which is typically much smaller than the window size. Currently, the maximum achievable number of instructions per cycle is rare and limited to 6 in some cases. Processor widths vary from single-issue (e.g., ARM11, UltraSPARC-T1) through 2-issue (e.g., UltraSPARC-T2/T3, Cortex-A8 & A9, Atom, Bobcat) to 3-issue (e.g., Pentium-Pro/II/III/M, Athlon, Pentium-4, Athlon 64/Phenom, Cortex-A15), 4-issue (e.g., UltraSPARC-III/IV, PowerPC G4e, Core 2, Core i, Core i*2, Bulldozer), 5-issue (e.g., PowerPC G5), or even 6-issue (e.g., Itanium, although it's a VLIW architecture). However, deciding which 8 or 16 instructions can execute every cycle is incredibly challenging due to the sheer complexity involved:

- It requires significant computational effort.
- This would necessitate a decrease in processor frequency due to longer computation times.

7.8.4 Current Superscalar and VLIW processors

Dynamically-scheduled superscalar processors represent the commercial state-of-the-art for general-purpose computing. Present implementations of architectures like Intel Core i, PowerPC, Alpha, MIPS, SPARC, and others are all examples of superscalar processors. VLIW (Very Long Instruction Word) processors, on the other hand, find primary success in embedded media processors for consumer electronic devices. Itanium 2 stands out as the only general-purpose VLIW processor, adopting a hybrid VLIW approach known as EPIC (Explicitly Parallel Instruction Computing).

Limits Doubling issue rates from today's typical 3-6 instructions per clock, say to 6 to 12 instructions, would likely necessitate a processor capable of:

- Issuing 3 or 4 data memory accesses per cycle.
- Resolving 2 or 3 branches per cycle.

- Renaming and accessing more than 20 registers per cycle.
- Fetching 12 to 24 instructions per cycle.

Implementing these capabilities is complex and would likely require sacrifices in the maximum clock rate. Most techniques for increasing performance tend to increase power consumption. The crucial question lies in whether a technique is energy-efficient, i.e., whether it increases power consumption faster than it enhances performance. Multiple issue processor techniques are generally energy-inefficient:

- Issuing multiple instructions incurs overhead in logic that grows faster than the issue rate.
- This leads to a growing gap between peak issue rates and sustained performance.
- The number of transistors switching is a function of the peak issue rate, while performance is a function of the sustained rate, resulting in an increasing energy per unit of performance ratio.

7.8.5 Conclusions

To advance performance in the future, we need to consider the limitations of Instruction-Level Parallelism (ILP) and explore explicit parallelism that programmers can directly leverage, as opposed to relying solely on implicit parallelism exploited by compilers and hardware. However, several challenges remain:

- *Processor-memory performance gap*: this gap poses a significant hurdle to achieving further performance gains.
- *VLSI scaling problems*: challenges related to wiring and other VLSI scaling issues need to be addressed.
- *Energy and leakage problems*: increasing energy consumption and leakage present additional concerns.

Despite these challenges, alternative forms of parallelism offer potential solutions:

- *Multi-core architectures*: embracing multi-core architectures allows for parallel processing across multiple cores.
- *SIMD revival*: sub-word parallelism, as seen in SIMD (Single Instruction, Multiple Data) architectures, is experiencing a resurgence and offers another avenue for improving performance.

CHAPTER 8

Software-level parallelism

8.1 Introduction

Instruction Level Parallelism (ILP) aims to overlap individual machine operations, such as additions, multiplications, and loads, allowing them to execute in parallel. This process is designed to be transparent to the user, with the primary goal of speeding up execution.

In contrast, parallel processing involves using separate processors to handle different chunks of a program, with each processor programmed to perform its tasks independently. This method is nontransparent to the user, and it aims to improve both the speed and quality of execution. To achieve performance improvements beyond a single cycle per instruction (CPI) using parallel processing, we can employ techniques such as Superscalar architecture or Very Long Instruction Word.

8.1.1 Very Long Instruction Words

VLIW architecture uses a fixed number of instructions, typically ranging from 4 to 16, which are scheduled by the compiler and placed into wide templates. This approach has found notable success in digital signal processing (DSP) and multimedia applications. A significant milestone in VLIW development was the joint agreement between HP and Intel in 1999/2000, leading to the creation of the Intel Architecture-64 (Merced/A-64) with a 64-bit address space. This style of architecture is known as an Explicitly Parallel Instruction Computer (EPIC).

A processor can initiate multiple operations per cycle, with all operations specified entirely by the compiler, unlike in Superscalar architectures. This approach results in low hardware complexity as there is no need for scheduling hardware and reduced support for variable latency instructions. Additionally, the hardware does not perform any instruction reordering, ensuring explicit parallelism and a single control flow.

In this context, an operation refers to a unit of computation, such as an addition, load, or branch, similar to an instruction in a sequential architecture. An instruction, however, is a set of operations intended to be issued simultaneously. The compiler decides which operations go into each instruction through scheduling. All operations that are supposed to start at the same time are packaged into a single VLIW instruction.

Multiple operations are packed into one instruction, with each operation slot designated for a specific function. Constant operation latencies are specified, and the architecture requires a guarantee of parallelism within an instruction, meaning there is no need for cross-operation read-after-write (RAW) checks. Furthermore, no data is used before it is ready, eliminating the need for data interlocks.

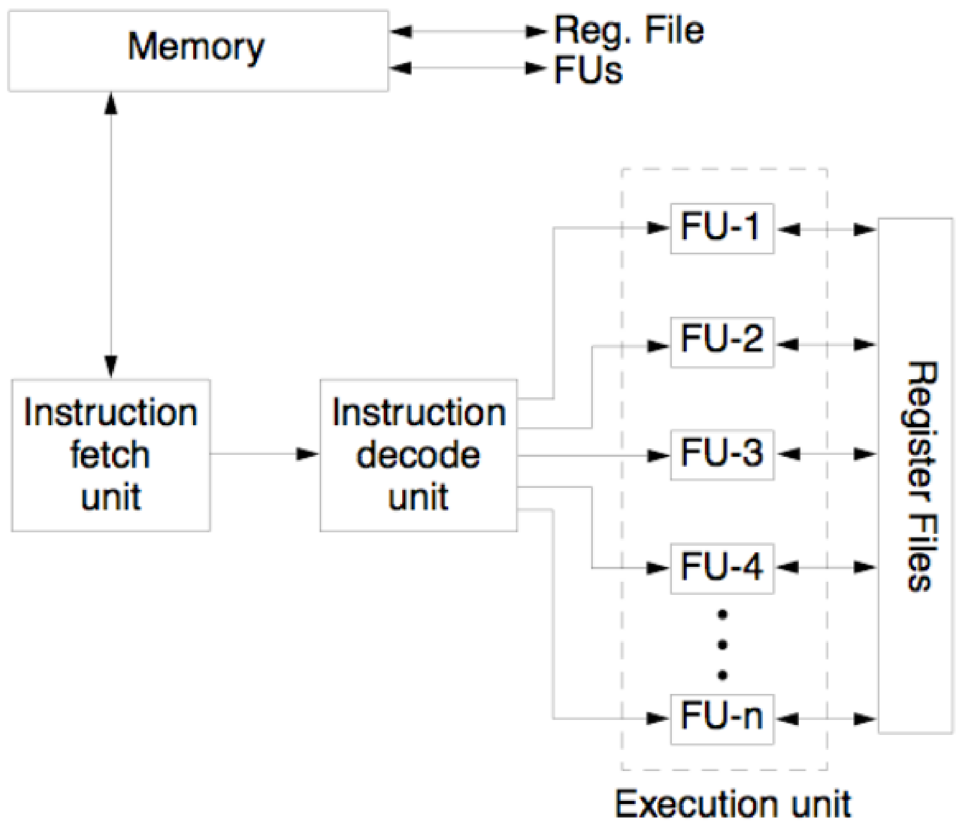


Figure 8.1: VLIW machine configuration

8.2 VLIW compiler

The VLIW compiler is responsible for scheduling instructions to maximize parallel execution, exploiting both Instruction Level Parallelism (ILP) and Loop Level Parallelism (LLP). This involves mapping instructions to the machine's functional units while accounting for time constraints and task dependencies. The compiler ensures intra-instruction parallelism and schedules operations to avoid data hazards, typically separating them with explicit NOPs (No Operation instructions). The primary goal is to minimize the program's total execution time.

To support ILP, there are two main strategies:

1. *Dynamic scheduling*: relies on hardware to locate parallelism.
2. *Static scheduling*: depends on software to identify potential parallelism.

8.2.1 Static scheduling

Static scheduling aims to keep the pipeline full in single-issue pipelines or utilize all functional units in each cycle in VLIW architectures. This maximizes ILP and achieves higher parallel speedups.

General formulation Compilers use sophisticated algorithms for code scheduling to exploit ILP. Within a basic block, which is a sequence of straight non-branch instructions, the amount of parallelism is limited. Data dependencies further restrict the available ILP within a basic block. Therefore, substantial performance enhancements require exploiting ILP across multiple basic blocks, including across branches.

Static scheduling involves the compiler detecting and resolving dependencies through code reordering. The compiler's output is a dependency-free code, which is typical for VLIW processors. This method has several advantages and disadvantages:

8.2.2 Summary

The VLIW approach benefits from simple hardware requirements, making it easier to design and maintain. It also allows for straightforward extension of the number of functional units, facilitating scalability. Additionally, good compilers can effectively detect and exploit parallelism, maximizing performance gains.

However, VLIW requires a large number of registers to keep all functional units active, and it necessitates substantial storage for operands and results. The architecture demands a high data transport capacity between functional units, register files, and memory, and a high bandwidth between the instruction cache and fetch unit. This results in a larger code size. Binary compatibility poses challenges, as does the need for profiling to understand branch probabilities, adding a significant extra step in the build process. Scheduling for statically unpredictable branches is also difficult, since the optimal schedule can vary with different branch paths.

Static Scheduling Methods Several methods are used for static scheduling, including:

- *Simple code motion*: reordering instructions to avoid stalls.
- *Loop unrolling and loop peeling*: increasing loop body size to reduce overhead and improve parallelism.
- *Software pipelining*: overlapping the execution of operations from different iterations of a loop.
- *Global code scheduling*: optimizing code across multiple basic blocks.
  - *Trace scheduling*: identifying and optimizing frequently executed paths.
  - *Super-block scheduling*: grouping basic blocks to optimize their execution as a single unit.
  - *Hyper-block scheduling*: extending super-blocks to include predicated instructions for increased parallelism.
  - *Speculative trace scheduling*: scheduling instructions based on probable execution paths, even if they involve branches.

8.3 Trace Scheduling

Trace scheduling focuses on optimizing sequences of instructions known as traces within a control flow graph. A trace, defined as a loop-free sequence of basic blocks, represents an execution path for a given set of inputs (Fisher). The likelihood of a trace being executed depends on the input set, with some traces being executed more frequently than others.

The process of trace scheduling begins with selecting a sequence of basic blocks that represents the most frequent branch path. Profiling feedback or compiler heuristics are used to identify these common paths. Once identified, the entire trace is scheduled at once, with fix-up code added to handle branches that jump out of the trace.

One limitation of trace scheduling is that it cannot extend beyond a loop barrier. This is typically addressed through loop unrolling, which allows scheduling to proceed beyond loops but can result in increased code size and performance degradation due to the costs of starting and ending iterations.

Despite these drawbacks, trace scheduling offers significant advantages by prioritizing the scheduling of traces based on their execution probability. This ensures that the most frequently executed traces receive the best optimization. Traces are treated as basic blocks during scheduling, without special handling for branches.

A trace is a sequence of instructions that may include branches but does not include loops. The objective of trace scheduling is to identify common paths and schedule the traces within those paths independently. Scheduling within a trace relies on basic code motion, extending globally across multiple basic blocks through appropriate renaming.

Compensation codes are necessary for handling side entry points (points within the trace except the beginning) and side exit points (points within the trace except the end). However, blocks on non-common paths may incur additional overhead, necessitating a high probability of following the common path based on profiling data. Generating compensation codes, especially for entry points, can be particularly challenging.

8.3.1 Code Motion

In addition to the need for compensation codes, there are restrictions on code movement within a trace. The dataflow of the program must remain unchanged, and exception behavior must be preserved. Dataflow integrity is maintained by ensuring two types of dependencies: data dependency and control dependency.

To eliminate control dependency, two solutions are typically employed: using predicate instructions (Hyper-block scheduling) to remove the branch, or using speculative instructions (Speculative Scheduling) to move an instruction before the branch speculatively. These techniques allow for greater flexibility and efficiency in scheduling traces.

CHAPTER 9

Multithreading and multiprocessors

9.1 Multithreading

Modern processors often fail to utilize their execution resources efficiently due to various issues such as memory conflicts, control hazards, branch misprediction, and cache misses. Addressing these problems individually tends to have limited effectiveness. Consequently, a general latency-tolerance solution that can hide all sources of latency is needed to significantly enhance performance.

9.1.1 Parallel programming

Explicit parallelism involves structuring applications into concurrent and communicating tasks. Operating systems support different types of tasks, with processes and threads being the most important and frequent. The implementation of multitasking varies based on the processor's characteristics, whether it is single-core, single-core with multithreading support, or multicore.

9.1.2 Multithreaded execution

Multithreading enables multiple threads to share the functional units of a single processor by overlapping their execution. The processor must duplicate the independent state of each thread, such as having separate copies of the register file and program counter (PC), and for running independent programs, separate page tables. Memory is shared through virtual memory mechanisms, which already support multiple processes. Hardware is designed for fast thread switching, which is much quicker than a full process switch.

Thread switching can occur in two ways:

- *Fine-grained multithreading*: the processor switches from one thread to another at each instruction, interleaving the execution of multiple threads. The CPU must be capable of changing threads every clock cycle, necessitating duplicated hardware resources.

- *Coarse-grained multithreading*: the processor switches from one thread to another only during long stalls, such as a miss in the second-level cache. Two threads share many system resources, and switching between threads requires several clock cycles to save the context. This approach does not slow down a single thread under normal conditions and does not require very fast thread-switching. However, it does not reduce throughput loss for short stalls, as the CPU must empty the pipeline before starting a new thread.

9.1.3 Thread-level parallelism

ILP and TLP ILP and TLP exploit different types of parallelism in a program. A processor designed for ILP often has idle functional units due to stalls or dependencies in the code. TLP can provide independent instructions to keep the processor busy during these stalls and utilize the otherwise idle functional units. By leveraging the resources of a superscalar processor, both ILP and TLP can be exploited simultaneously.

The key motivation is that modern CPUs have more functional resources than a single thread can utilize. With register renaming and dynamic scheduling, independent instructions from different threads can be issued without concerns about dependencies, which are managed by the hardware.

9.1.4 Simultaneous multithreading

Simultaneous multithreading (SMT) builds on the idea that dynamically scheduled processors already have many hardware mechanisms to support multithreading. These include a large set of virtual registers to hold the register sets of independent threads and register renaming to provide unique register identifiers. This allows instructions from multiple threads to be mixed in the datapath without confusing sources and destinations across threads. Out-of-order completion further enhances hardware utilization.

SMT can be implemented by adding a per-thread renaming table and maintaining separate PCs for each thread. Independent commitment can be supported by logically keeping a separate reorder buffer for each thread. The system can dynamically adapt to the environment, allowing the execution of instructions from each thread and utilizing all functional units if one thread encounters a long latency event. More threads increase the CPU's issue possibilities per cycle, ideally limited only by the imbalance between resource requests and availability.

9.2 Parallel architectures

Increasing the performance and clock frequency of a single core has become increasingly difficult due to several factors. Deep pipeline designs, which have been a common strategy, face significant issues:

- *Heat dissipation*: managing the heat generated by high-frequency operations is problematic.
- *Speed of light limitations*: The physical limits of light speed affect transmission speed in wires.
- *Design complexity*: designing and verifying deep pipelines is challenging and requires large design teams.
- *Multithreaded applications*: many new applications are inherently multithreaded, demanding better support for parallel execution.

Beyond Instruction-Level Parallelism ILP architectures, such as superscalar and VLIW, are designed to exploit fine-grained, instruction-level parallelism. However, they are not well-suited for supporting large-scale parallel systems. Multiple-issue CPUs, which attempt to issue multiple instructions per cycle, have become extremely complex, and the benefits of extracting additional parallelism are diminishing. Consequently, extracting parallelism at higher levels has become more attractive.

9.2.1 Process and Thread-Level Parallel Architectures

To achieve higher performance, the next step involves process- and thread-level parallel architectures. This approach focuses on connecting multiple microprocessors in a complex system to handle large-scale parallel tasks efficiently.

Definition (*Parallel architecture*)**.** A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems quickly.

The goal is to replicate processors to enhance performance rather than merely designing a faster single processor. Parallel architecture extends traditional computer architecture by incorporating a communication architecture that includes:

- *Abstractions*: defining the hardware/software interface.
- *Efficient structures*: developing various structures to realize these abstractions efficiently.

By leveraging parallel architectures, systems can better support multithreaded applications and large-scale computational tasks, addressing the limitations of ILP and deep pipeline designs.

Many of the early multiprocessors were based on the SIMD (Single Instruction, Multiple Data) model, which garnered significant attention in the 1980s. Today, SIMD is applied only in specific instances, such as vector processors and multimedia instructions. Conversely, MIMD (Multiple Instruction, Multiple Data) has become the architecture of choice for general-purpose multiprocessors.

9.2.2 Vector processing

Vector processors are designed to perform high-level operations on linear arrays of numbers, known as vectors. A typical vector processor consists of a pipelined scalar unit (which may be out-of-order or VLIW) and a vector unit. There are two main types of vector processors: memory-memory vector processors, where all vector operations are memory to memory, and vector-register processors, where all vector operations occur between vector registers, except for load and store operations. This latter type is akin to load-store architectures.

Vector processors are particularly useful in multimedia processing, including tasks such as compression, graphics, audio synthesis, and image processing. They are also employed in standard benchmark kernels like Matrix Multiply, FFT, Convolution, and Sort, as well as in both lossy (e.g., JPEG, MPEG video and audio) and lossless (e.g., Zero removal, RLE, Differencing, LZW) compression techniques. Other applications include cryptography (e.g., RSA, DES/IDEA, SHA/MD5), speech and handwriting recognition, operating systems and networking (e.g., memcpy, memset, parity, checksum), databases (e.g., hash/join, data mining, image/video serving), and language runtime support (e.g., stdlib, garbage collection), even extending to benchmarks like SPECint95.

MIMD architectures have gained popularity due to their flexibility. They can function as single-user machines for high performance on specific applications, as multiprogrammed multiprocessors running many tasks simultaneously, or as a combination of these functions. These systems can be constructed using standard CPUs, which is the case for nearly all modern multiprocessors. Each processor in an MIMD system fetches its own instructions and operates on its own data, often using off-the-shelf microprocessors. This scalability allows for a variable number of processor nodes and provides cost/performance advantages. MIMD systems can focus on single-user high-performance tasks, support multi-programmed environments, or combine these functionalities, although fault tolerance remains a concern.

To effectively utilize an MIMD system with n processors, there must be at least n threads or processes to execute. These independent threads are typically identified by the programmer or created by the compiler, encapsulating parallelism within the threads, a concept known as thread-level parallelism. A thread can range from a large, independent process to parallel iterations of a loop, with parallelism identified by the software, unlike in superscalar CPUs where it is hardware-determined.

MIMD machines Existing MIMD machines fall into two classes based on the number of processors involved, which dictates their memory organization and interconnection strategy.

Centralized shared-memory architectures can support at most a few dozen processor chips (less than 100 cores). These systems feature large caches and multiple memory banks, and are often referred to as symmetric multiprocessors (SMP) with a Uniform Memory Access (UMA) architecture.

Distributed memory architectures are designed to support a larger number of processors. These systems require a high-bandwidth interconnect but face the disadvantage of complex data communication among processors.

9.3 Multiple Instruction Multiple Data architectures

In shared memory architectures, processors operate within a single logically shared address space, allowing any processor to reference any memory location. This means that the address space is shared among processors, so the same physical address refers to the same memory location for all processors. In contrast, message passing architectures use multiple and private address spaces where processors communicate through send and receive primitives. Here, each processor has its own separate address space, meaning the same physical address in different processors refers to different memory locations.

Shared address spaces In shared address space architectures, processors communicate through shared variables in memory. Communication management is implicit, handled through load/store operations. This model is the oldest and most popular in parallel computing. It is important to note that shared memory does not necessarily imply a single centralized memory structure.

Private address spaces In private address space architectures, processors communicate by sending and receiving messages. This communication is managed explicitly through send/receive operations to access private memory. One advantage of this setup is the absence of cache coherency issues among processors, which can be a significant challenge in shared memory systems.

9.3.1 Physical memory organization

Centralized shared-memory architectures typically support up to a few dozen processor chips (fewer than 100 cores) and feature large caches and multiple memory banks. These systems are often referred to as symmetric multiprocessors (SMP) with a Uniform Memory Access (UMA) architecture. On the other hand, distributed memory architectures are designed to support a larger number of processors and require a high-bandwidth interconnect. These systems, known as Non-Uniform Memory Access (NUMA) architectures, present challenges in managing data communication among processors.

It's important to understand that the concepts of addressing space (single vs. multiple) and physical memory organization (centralized vs. distributed) are orthogonal. A multiprocessor system can have a single addressing space with distributed physical memory.

Shared programming model In the shared memory programming model, a program is a collection of threads of control, which can be created dynamically during execution in some languages. Each thread has a set of private variables (e.g., local stack variables) and a set of shared variables (e.g., static variables, global heap). Threads communicate implicitly by reading and writing shared variables and coordinate by synchronizing on these shared variables.

Private programming model In the message passing programming model, a program consists of a collection of named processes, usually fixed at startup. Each process has a local address space with no shared data, and logically shared data is partitioned over local processes. Processes communicate by explicit send/receive pairs, and coordination is implicit in every communication event. The Message Passing Interface (MPI) is a commonly used software implementation of this model.

Comparison The choice between shared memory and message passing depends on the specific program. Both models are communication Turing complete, meaning each can simulate the other. Shared memory has the advantage of implicit communication through loads/stores and low overhead when cached. However, it can be complex to scale effectively, requires synchronization operations, and makes data placement control within the caching system difficult.

Message passing, particularly in massively parallel processors, requires software to handle all data layout and remote data access via message requests and replies. It incurs high software overhead for message passing, as early machines required operating system invocation for each message. Even with user-level access, there is significant overhead. While sending messages can be relatively inexpensive, receiving messages is costly due to the need for polling or interrupt handling.

Bus-based symmetric shared memory Bus-based symmetric shared memory architectures dominate the server market and are making their way into desktop systems. They are attractive for throughput servers and parallel programs due to their fine-grain resource sharing, uniform access through loads/stores, automatic data movement, and coherent cache replication. These architectures represent a cost-effective and powerful extension of uni-processor mechanisms for data access, with the key feature being the extension of the memory hierarchy to support multiple processors.

9.3.2 Shared memory machines

Shared memory machines can be categorized into two main types: non-cache coherent and hardware cache coherent. Hardware cache coherent machines work with any data placement, although performance may vary. Optimization efforts can be focused on critical portions of the code. These machines use load and store instructions for data communication, which does not involve the operating system and results in low software overhead. Special synchronization primitives are usually employed. In large-scale systems, logically distributed shared memory is implemented as physically distributed memory modules.

Cache coherence problem Shared-memory architectures cache both private data (used by a single processor) and shared data (used by multiple processors). When shared data are cached, the value may be replicated in multiple caches. This replication reduces access latency and required memory bandwidth, while also decreasing contention when multiple processors read shared data simultaneously. However, private processor caches can create issues because copies of a variable might be present in multiple caches. Consequently, a write by one processor may not immediately be visible to others, leading to the problem of cache coherence.

Informally, coherency means any read must return the most recent write, but this definition is too strict and difficult to implement. A more practical definition is any write must eventually be seen by a read, with all writes being seen in the proper order, a concept known as serialization. To ensure coherence, two rules must be followed:

1. If Processor P writes x and Processor P_1 reads it, P's write will be seen by P_1 if the read and write are sufficiently far apart and no other writes to x occur between the two accesses.

2. Writes to a single location are serialized, meaning two writes to the same location by any two processors are seen in the same order by all processors.

These rules ensure that the latest write will be seen and prevent the possibility of observing writes in an illogical order.

It is impractical to require that a read of x by Processor P_1 can instantaneously see the write of x by another processor if the write precedes the read by a small amount of time. This introduces the problem of memory consistency. Coherence and consistency are complementary concepts: coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations. For now, we assume the following:

- A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write.

- The processor does not change the order of any write with respect to any other memory access.

This implies that if a processor writes A followed by B, any processor that sees the new value of B must also see the new value of A.

Coherent caches A program running on multiple processors will typically have copies of the same data in several caches. In a coherent multiprocessor, caches support both migration and replication of shared data items.

- *Migration*: a data item can be moved to a local cache and used there transparently. This reduces both the latency to access a remotely allocated shared data item and the bandwidth demand on the shared memory.

- *Replication*: for shared data being simultaneously read, caches make a copy of the data item in the local cache. This reduces both latency of access and contention for reading shared data.

To maintain coherency, hardware-based solutions such as cache-coherence protocols are employed. The key issue in implementing a cache-coherent protocol in multiprocessors is tracking the status of any sharing of a data block. There are two primary classes of protocols: snooping protocols and directory-based protocols.