BASIC CONCEPTS IN PROCESSOR MICROARCHITECTURE

3. Run - On Delays (in Order Execution Only)

When instructions must complete the WB (writeback) in program order, any delay in execution (as in the case of multiply or divide) necessarily delays the instruction execution in the pipeline.

4. Branches

The pipeline is delayed because of branch resolution and/or delay in the fetching of the branch target instruction before the pipeline can resume execution.

Branch prediction, branch tables, and buffers all can be used to minimize the effect of branches.

BASIC ELEMENTS IN INSTRUCTION HANDLING

- An instruction unit consists of the state registers as defined by the instruction set — the instruction register — plus the instruction buffer, decoder, and an interlock unit.

- The **instruction buffer's** function is to fetch instructions into registers so that instructions can be rapidly brought into a position to be decoded.
- The **decoder** has the responsibility for controlling the cache, ALU, registers, and so on.

BASIC ELEMENTS IN INSTRUCTION HANDLING

- The **instruction unit** sequencing is managed strictly by **hardware,** but the execution unit may be microprogrammed so that each instruction that enters the execution phase will have its own microinstruction associated with it.

- The ***interlock unit*** is responsible for maintaining the integrity of instruction execution by ensuring that even though multiple instructions may be processed simultaneously, the outcome remains the same as if the instructions were processed one by one.

The Instruction Decoder and Interlocks

When an instruction is decoded, the decoder must provide more than control and sequencing information for that instruction. Proper execution of the current instruction depends on the other instructions in the pipeline.

Schedules the current instruction:

If an instruction relies on data not yet available (e.g., a prior instruction hasn't produced the necessary data), or if an exception occurs (e.g., cache miss or TLB miss), the current instruction is delayed.

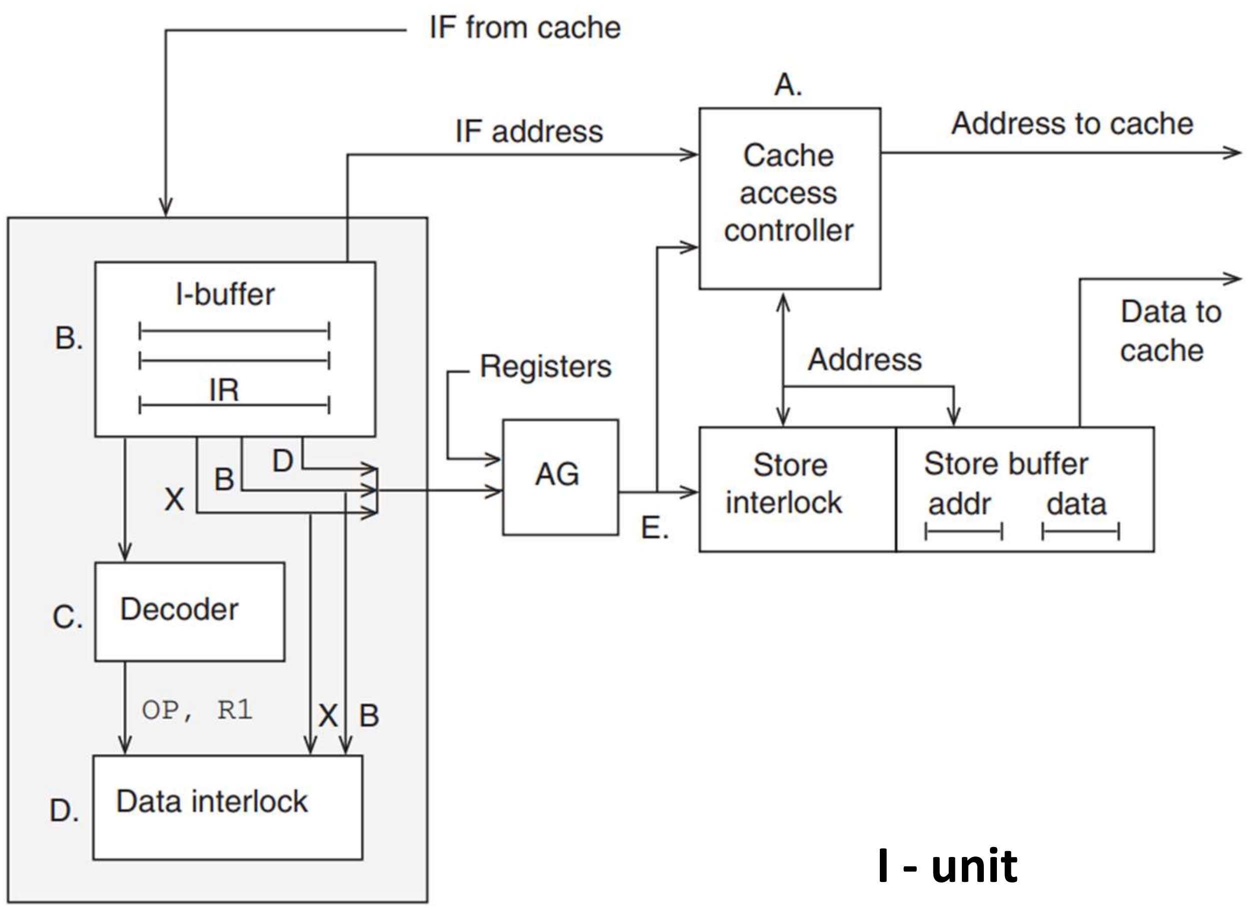
Schedules subsequent instructions:

If an instruction takes multiple cycles to execute, subsequent instructions must wait to ensure instructions complete in the correct order.

Handles branches:

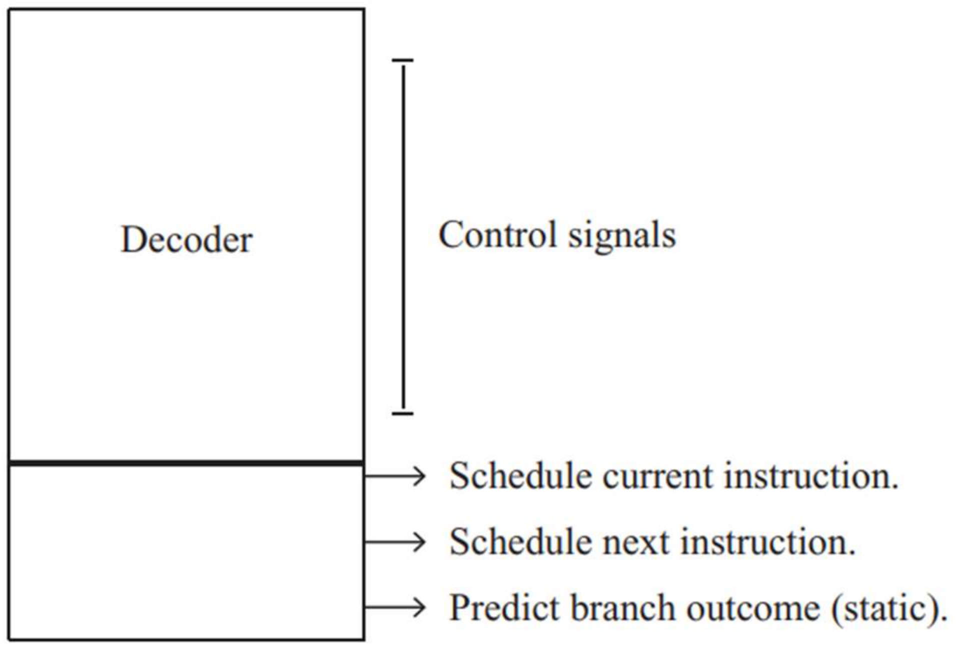
In case of branch instructions (e.g., conditionals), the decoder must predict the correct path of execution and schedule accordingly.

The Instruction Decoder and Interlocks



I - unit

The Instruction Decoder and Interlocks


Decoder
Control signals
Schedule current instruction.
Schedule next instruction.
Predict branch outcome (static).

The Instruction Decoder and Interlocks

Data Interlocks

- These are mechanisms that manage *data dependencies* between instructions.

Example of a Dependency:

If instruction A produces a result that instruction B needs, instruction B cannot proceed until A has completed.

Function of Interlocks:

- Ensure an instruction doesn't attempt to use a result from a previous instruction until that result is ready.
- The interlocks also control when the *Address Generate* (AG) and *Execution* (EX) units can be used.

The Instruction Decoder and Interlocks

Execution Controller & In-order Execution

- The execution controller ensures subsequent instructions don’t enter the pipeline until the current instruction has been executed properly. This is essential for maintaining *in-order completion*.

In-order Completion: The instructions should complete in the order they were issued, even though they may be executed out-of-order inside the pipeline. The interlocks help enforce this.

The Instruction Decoder and Interlocks

Dependency Checking

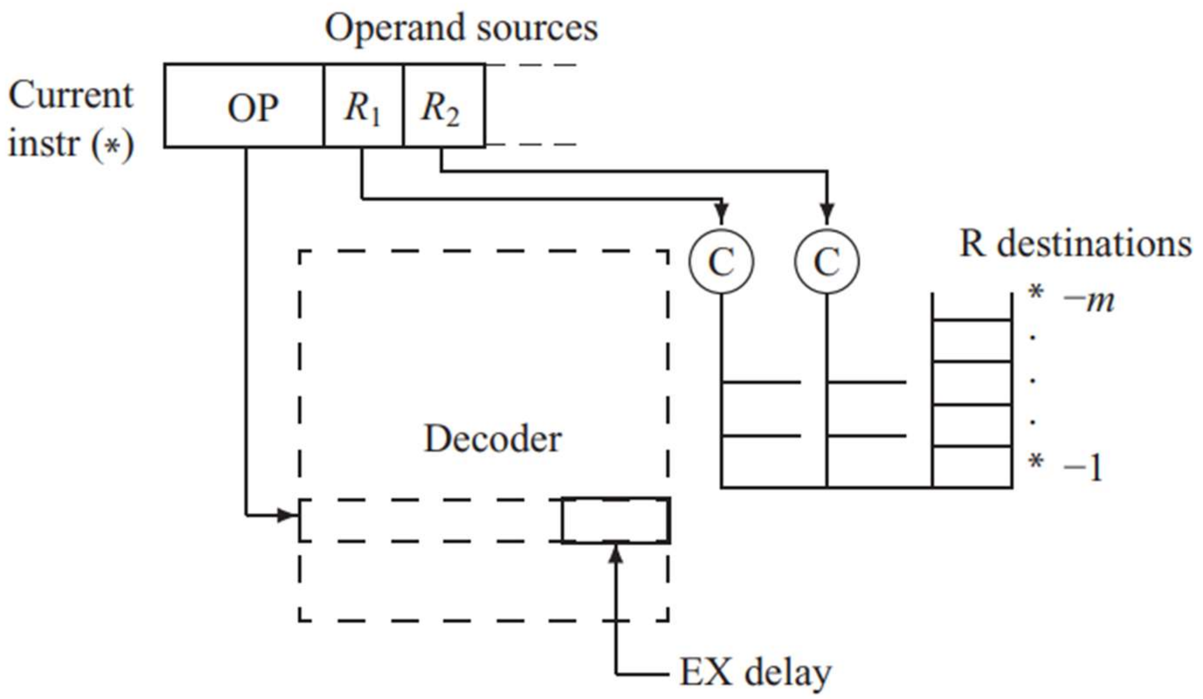
As each instruction is decoded, the decoder checks the *source registers* (input data) of the current instruction and compares them with the *destination registers* (output data) of previous uncompleted instructions. This is done to detect dependencies.

Why is this important? If a current instruction needs a value from a previous instruction that hasn't finished, it has to wait. The interlocks delay execution to prevent errors.

EX Cycles & Timing

- The *execution (EX) cycles* indicate how long an instruction takes to execute. If an instruction requires more EX cycles than expected, subsequent instructions must be delayed accordingly.

The Instruction Decoder and Interlocks



The Instruction Decoder and Interlocks

Store Interlocks (E)

- These perform a similar function for memory access (STORE instructions). Before a STORE instruction can execute, the store interlocks check the address against other pending instructions that may be reading from or writing to the same location. This ensures proper memory access ordering.

- For example, if an instruction wants to store data at a memory location, subsequent instructions trying to read from or write to that location will be delayed until the store operation is complete.

The Instruction Decoder and Interlocks

- **Bypassing (Forwarding)**
- **What is Bypassing?**

Bypassing (or forwarding) is a technique used in pipelined processors to improve efficiency by reducing delays caused by data dependencies.

- **How Does It Work?**

In a typical pipeline, results from one stage are written to a destination register (a storage location for data), and subsequent instructions can use those results after they are stored.

- However, this introduces a delay because the subsequent instruction has to wait until the register is updated.
- Bypassing allows a result produced by the ALU (Arithmetic Logic Unit) to be *directly* sent to the next instruction that needs it, without waiting for it to be written to the register. This reduces delays and improves performance.

Execution Unit

What is the Execution Unit?
The execution unit is a critical component of the processor responsible for carrying out arithmetic and logic operations, including both *integer* and *floating-point* calculations.

Impact on Performance and Area:

- In a processor, especially in the case of a *floating-point unit (FPU)*, the execution unit can significantly affect both performance and the physical size (area) of the processor chip.
- Floating-point calculations are complex and often require more sophisticated hardware than integer calculations.
- Therefore, even a basic FPU can occupy as much or even more *physical area* on the chip than a simple integer core processor (excluding the cache).

Execution Unit

Execution Delay (Run-on):

In simple *in-order pipelines*, where instructions are executed strictly in the order they are issued, the time it takes for an instruction to complete (execution delay or run-on) can heavily influence the overall performance.

If an instruction, especially one involving floating-point calculations, takes too long to complete, it may slow down the entire pipeline.

Advanced Pipelines and Algorithms:

More advanced processors use better algorithms for arithmetic operations to reduce these delays. For example, the use of more efficient algorithms for floating-point and integer operations can help improve the performance of the execution unit and overall processor.

What Are Buffers in a Processor?

- **Buffers** temporarily hold data to reduce pipeline delays.
- They decouple the timing of events from when input data is used.
- Buffers help tolerate **latency** without impacting performance.
- Essential for ensuring smooth data flow between pipeline stages.

How Buffers Minimize Delays

- Buffers allow events to occur at different times than when the data is used.
- They prevent performance degradation by holding data until it can be processed.
- Example: A store buffer holds data waiting to be written to memory.

Buffer Design for Request Rates

Mean Request Rate Buffers:

- Store buffers are typically designed for the average number of memory write requests.
- Buffer overflow can cause pipeline stalls if not managed efficiently.
- Balances size with performance, ensuring optimal request handling most of the time.
- Knowing the expected number of requests, we can trade off buffer size against the probability of an overflow.

Maximum Request Rate Buffers:

- Used for performance-sensitive operations (e.g., video data entry).
- Buffers should be large enough to handle maximum request rates.
- Ensures no delays in critical operations like fetching instructions or video data.

Handling Buffer Overflows

- Overflow in CPU buffers doesn’t mean data is lost, but the pipeline may stall.
- **Overflow condition:** When a buffer is full and new requests are made.
- Processor slows down to clear entries and reduce buffer load.
- Example: Store buffer stalls until memory accesses are completed.

Mean Request Rate Buffers and Overflow Probability

Buffers manage pending requests to optimize performance.

We can model request size as a random variable to design the buffer.

Key variables:

- q: Request size (number of pending requests)
- Q: Mean of the request size distribution
- σ: Standard deviation of the request size distribution

Little's Theorem

The mean request size is the product of:

- Mean request rate (requests per cycle)
- Mean time to service a request

Formula:

- Mean Request Size = Mean Request Rate × Mean Time to Service

Buffer Overflow Probability (p)

- Buffer Size (BF): Capacity of the buffer.
- p: Probability of buffer overflow.

- Two key inequalities provide upper bounds on overflow probability:

- Markov’s Inequality
- Chebyshev’s Inequality

Markov's and Chebyshev's Inequality for Overflow Probability

$$\text{Prob}\{q \geq BF\} \leq \frac{Q}{BF}$$

- **Explanation:**
  - This inequality gives a simple upper bound on the probability of buffer overflow based on the mean request size.

$$\text{Prob}\{q \geq BF\} \leq \frac{\sigma^2}{(BF - Q)^2}$$

- **Explanation:**
  - Chebyshev's inequality provides a more precise bound by incorporating the standard deviation (σ) of the request size distribution.

Selecting Buffer Size (BF)

Using these two inequalities, for a given probability of overflow (p), we can conservatively select BF , since either term provides an upper bound, as

$$BF = \min\left(\frac{Q}{p}, Q + \frac{\sigma}{\sqrt{p}}\right).$$

Selecting Buffer Size (BF)

Suppose we wish to determine the effectiveness of a two-entry write buffer. Assume the write request rate is 0.15 per cycle, and the expected number of cycles to complete a store is two. The mean request size is $0.15 \times 2 = 0.3$, using Little's theorem. Assuming $\sigma^2 = 0.3$ for the request size, we can calculate an upper bound on the probability of overflow as

$$p = \max\left(\frac{Q}{BF}, \frac{\sigma^2}{(BF - Q)^2}\right) = 0.10.$$

Buffers Designed for a Fixed or Maximum Request Rate

- Buffers designed for a fixed or maximum request rate are essential for maintaining performance.
- These buffers mask access latency and ensure smooth data or instruction flow.
- Examples: Instruction buffers, video buffers, and multimedia buffers.

Concept of a Fixed Rate Buffer

- Designed to supply a fixed rate of data or instructions.
- Primary goal: **Mask access latency** to prevent performance bottlenecks.
- Example: If one item is processed per cycle and it takes 3 cycles to access an item, a buffer of at least 3 (or 4) is required.

Reducing the Cost of Branches in Processor Performance

• Branches are a critical issue in processor optimization.

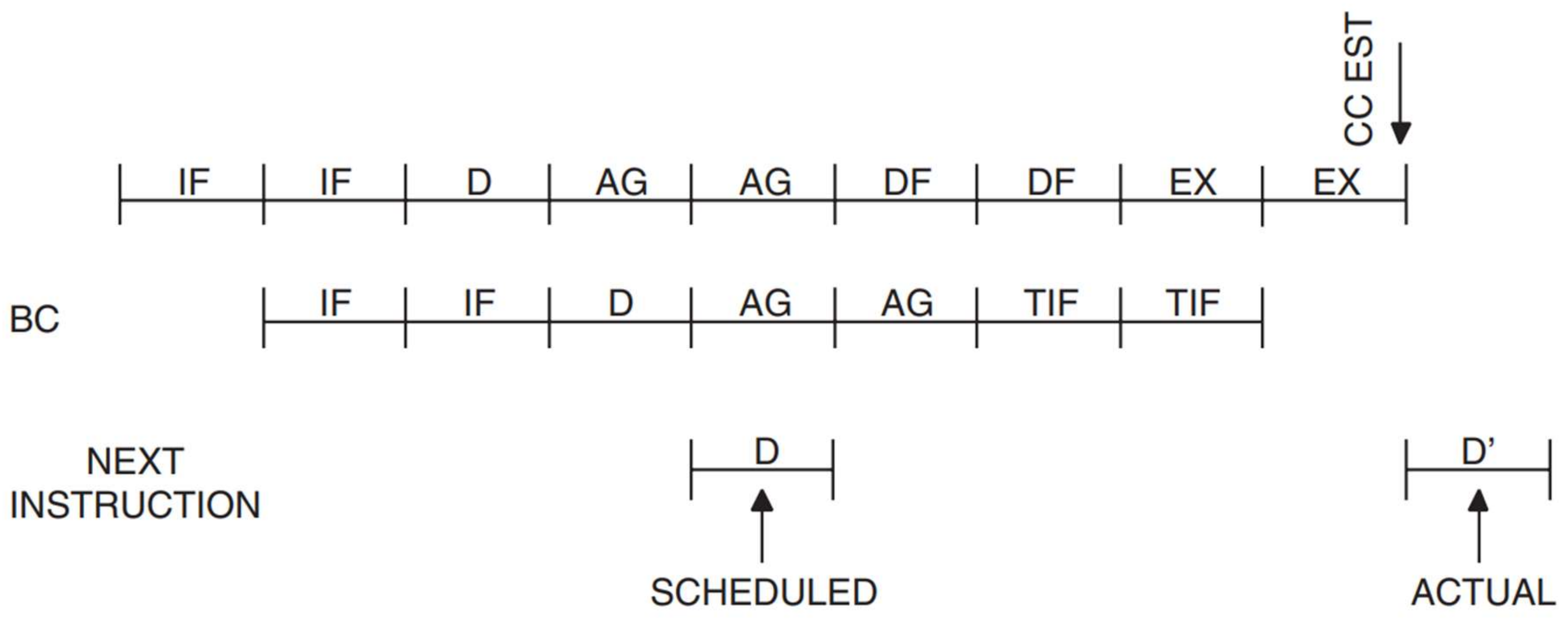
•Conditional branch instructions (BC) significantly affect performance.

• There may be a number of cycles between the decoding of the branch and the setting of the CC (Conditional Code).

•The simplest strategy is for the processor to do nothing but simply to await the outcome of the CC set and to defer the decoding of the instruction following the BC until the CC is known.

•Complex strategies attempt to predict branch outcomes.

The delay caused by a branch (BC)



Simple Sequential Processor

- Sequential processors directly implement the sequential execution model. These processors process instructions sequentially from the instruction stream.
- The next instruction is not processed until all execution for the current instruction is complete and its results have been committed.

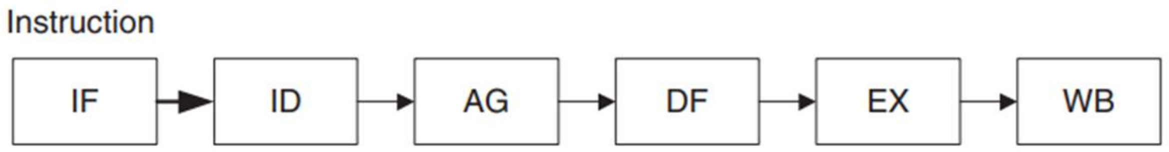


Figure 1.7 Instruction execution sequence.

1. fetching the instruction into the instruction register (IF),
2. decoding the opcode of the instruction (ID),
3. generating the address in memory of any data item residing there (AG),
4. fetching data operands into executable registers (DF),
5. executing the specified operation (EX), and
6. writing back the result to the register file (WB).

Performance Impact of Branches

- Example: Conditional Branch instruction (BC).
- Timing impact: 5 cycles branch penalty.
- Delayed fetch when branch is taken.
- Branches limit processor performance.
- There are two simple and two substantial approaches to the branch problem.

Simple Approaches to Reduce Branch Cost

Branch Elimination:

- Replace branch with another operation in some code sequences.

Simple Branch Speedup:

- Reduce time for target instruction fetch and CC determination.

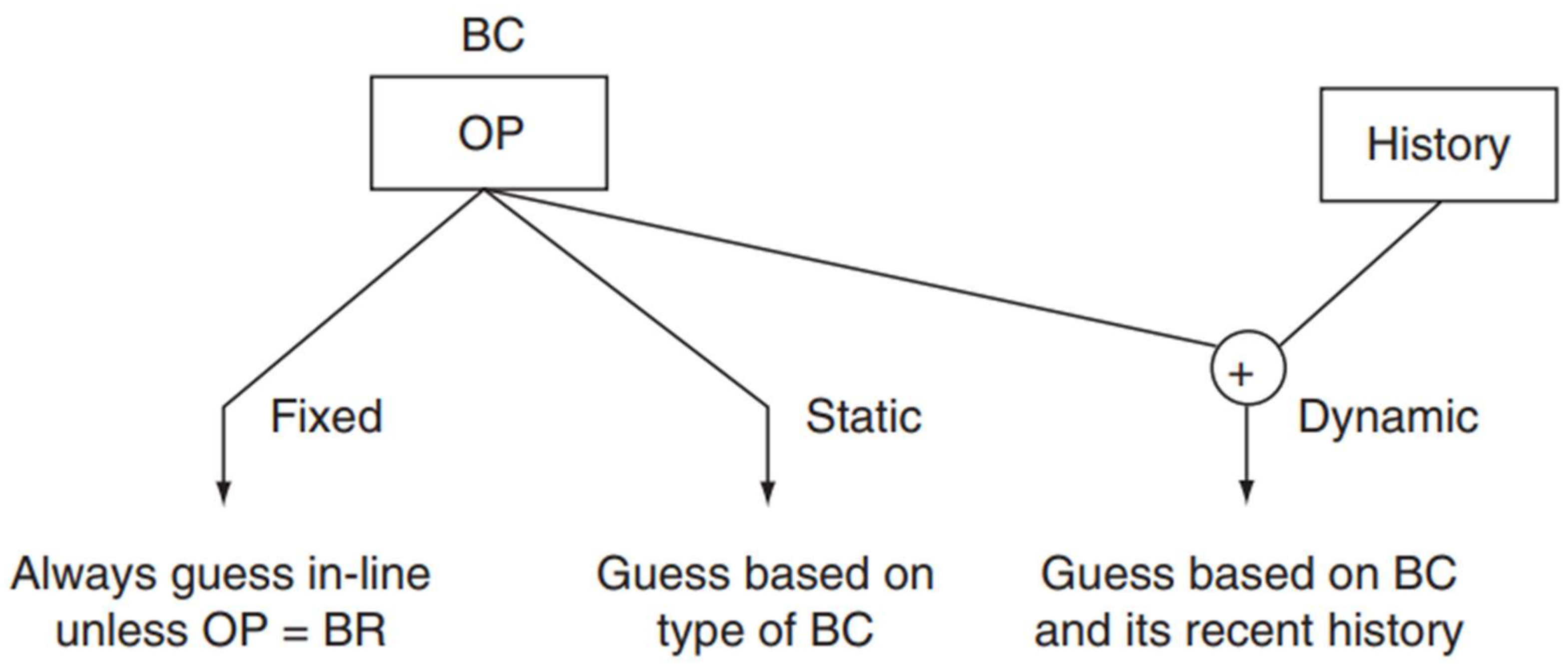
Complex Approaches to Branch Optimization

- **Branch Target Capture**:
  - After a branch has been executed, we can keep its target instruction (and its address) in a table for later use to avoid the branch delay.
  - No delay if branch path is predicted correctly.
- **Branch Prediction**:
  - Using available information about the branch, one can predict the branch outcome and can begin processing on the predicted program path.
  - Types of strategies: Fixed, Static, Dynamic.

Branch Prediction Strategies

- **Fixed Strategy**: Always fetch inline on true conditional branches.
- **Static Strategy**: Varies by opcode or target direction.
- **Dynamic Strategy**: Adapts to current program behavior.

Branch prediction


BC
OP
History
+
Fixed
Static
Dynamic
Always guess in-line unless OP = BR
Guess based on type of BC
Guess based on BC and its recent history

Techniques Summary

- Simple and complex approaches reviewed.
- Branch elimination and prediction summarized.
- Importance of optimizing processor performance through branch handling.

Branch Target Capture and Branch Prediction in Processors

- BTBs (Branch target buffer) store target instructions from previous branch executions.
- Each entry has the instruction address, target address, and most recent target instruction.
- BTBs help reduce branch delays by predicting whether a branch will be taken.

Branch Target Buffer (BTB) Functionality

- BTBs are indexed with each instruction fetch.
- A prediction is made if the instruction matches a stored branch.
- If the branch is predicted to occur, the target instruction is fetched with no delay.
- BTBs are updated if actual target differs after execution.

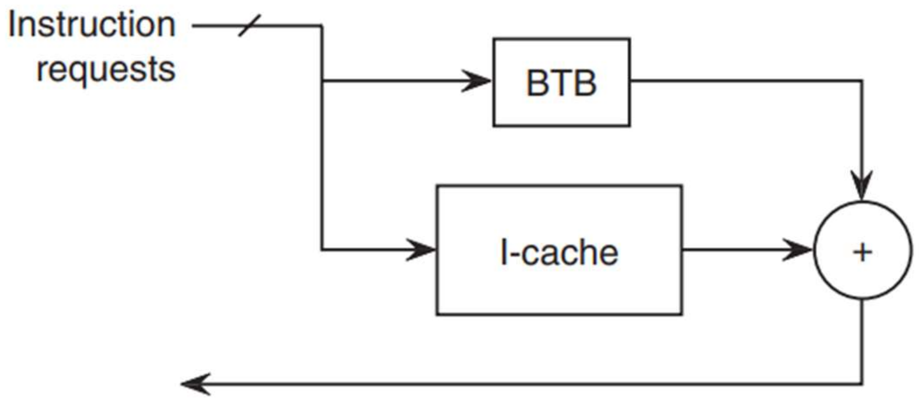
BTB Effectiveness

- BTB hit ratio: Probability of finding the branch at fetch time.

•For a 512-entry BTB, hit rate varies from 70% to over 98%.

•BTBs reduce branch delays when used with the instruction cache (I-cache)

Typical BTB structure



- Typical BTB structure. If “ hit ” in the BTB, then the BTB returns the target instruction to the processor; CPU guesses the target.
- If “ miss ” in the BTB, then the cache returns the branch and in - line path; CPU guesses in - line

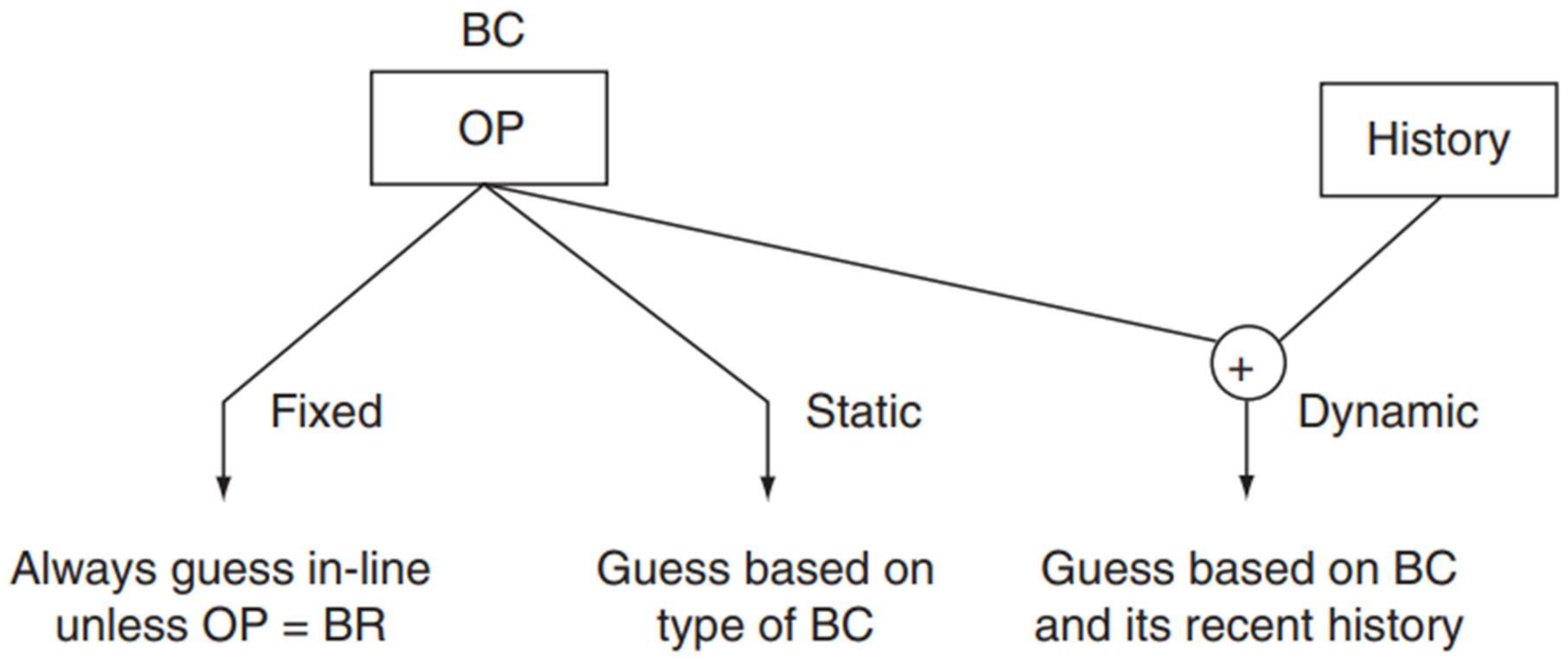
Static vs. Dynamic Branch Prediction

•**Static Prediction**: Based on branch opcode or target direction.

•**Dynamic Prediction**: Based on the recent history of branch activity.

•Even perfect prediction doesn’t fully eliminate branch delays.

Branch prediction


BC
OP
History
+
Fixed
Static
Dynamic
Always guess in-line unless OP = BR
Guess based on type of BC
Guess based on BC and its recent history

Static Prediction Strategies

- Static prediction relies on decoding the branch and guessing the outcome.
- Typically achieves 70-80% effectiveness.
- Target instruction stream is fetched and decoded if the branch is predicted to succeed.

Dynamic Prediction: Bimodal Strategy

•Dynamic predictions are based on past branch outcomes.

•Uses a small up/down saturating counter.

•Prediction accuracy varies between 83.4% to 96.5% depending on the application.

•Table organization can create aliasing problems when branches share history.

Two-Level Adaptive Prediction

- Uses a shift register to record branch history.
- Counters track patterns like "1100" (branch taken/not taken).
- Adaptive methods improve prediction rates up to 95% in large programs (e.g., 6-bit or 24-bit entries).

Combined Prediction Methods

- Combines bimodal and adaptive approaches using a "vote" table of counters.
- When predictions differ, the vote table selects the best predictor.
- Combined methods improve prediction accuracy further but require more hardware.

Branch Prediction in Processors

- Various processors use branch prediction strategies, from complex workstations to simpler System-on-Chip (SoC) processors.

- SOC processors use simpler techniques compared to high-end workstation processors.

Advanced Processor Architectures

- To go beyond one cycle per instruction (CPI), the processor must be able to execute multiple instructions at the same time.

- Concurrent processors must be able to make simultaneous accesses to instruction and to simultaneously execute multiple operations.

- Simultaneous accesses to the data memory.

- Processors that achieve a higher degree of concurrency are called concurrent processors, short for processors with instruction - level concurrency.

Introduction to Concurrent Processors

- Concurrent processors are more complex than simple pipelined processors.

- In these processors, performance depends in greater measure on compiler ability, execution resources, and memory system design.

- Concurrent processors depend on sophisticated compilers to detect the instruction - level parallelism that exists within a program.

- Concurrent processors require additional execution resources, such as adders and multipliers, as well as an advanced memory system to supply the operand and instruction bandwidth required to execute programs at the desired rate.

Types of Concurrent Processors

- **Vector Processors**
- **Very Long Instruction Word (VLIW) Processors**
- **Superscalar Processors**

Vector Processors

Boosting Performance: Vector instructions boost performance in 3 ways:

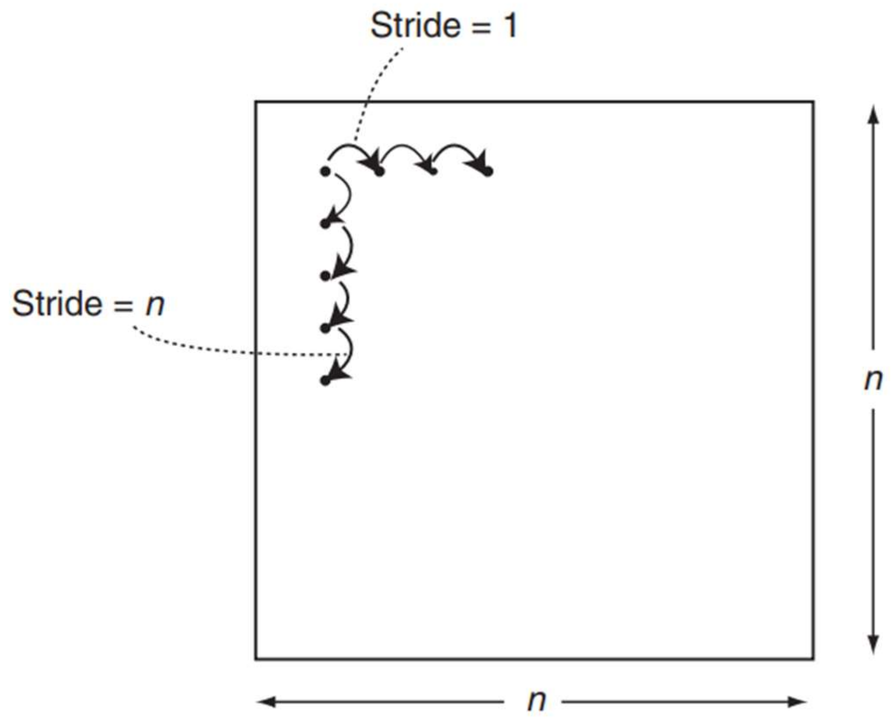
1. Reducing the number of instructions (I-bandwidth)

2. Organizing data efficiently

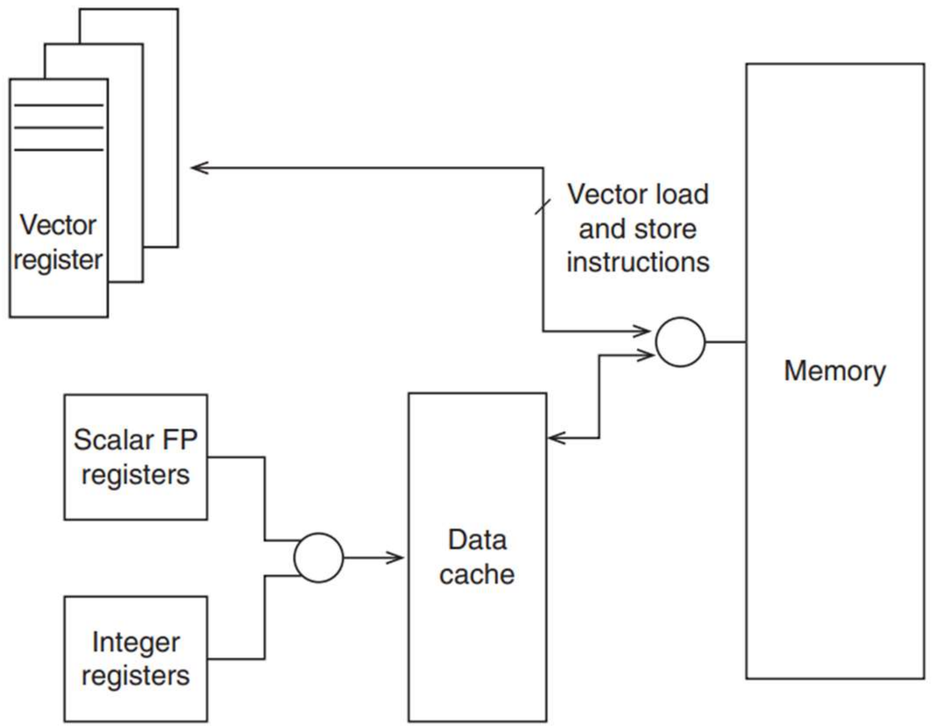
3. Removing control overhead for loop execution

Vector Processors

- Vector processors usually include vector register (VR) hardware to decouple arithmetic processing from memory.
- The VR set is the source and destination for all vector operands.
- In many implementations, accesses bypass the cache.
- The cache then contains only scalar data objects — objects not used in the VRs.



For an array in memory, different accessing patterns use different strides in accessing memory.



The primary storage facilities in a vector processor. Vector LD/ST usually bypasses the data cache.

Vector Functional Units

- **Vector Registers (VRs)**: 8 or more register sets, each with 16–64 vector elements (floating-point words)

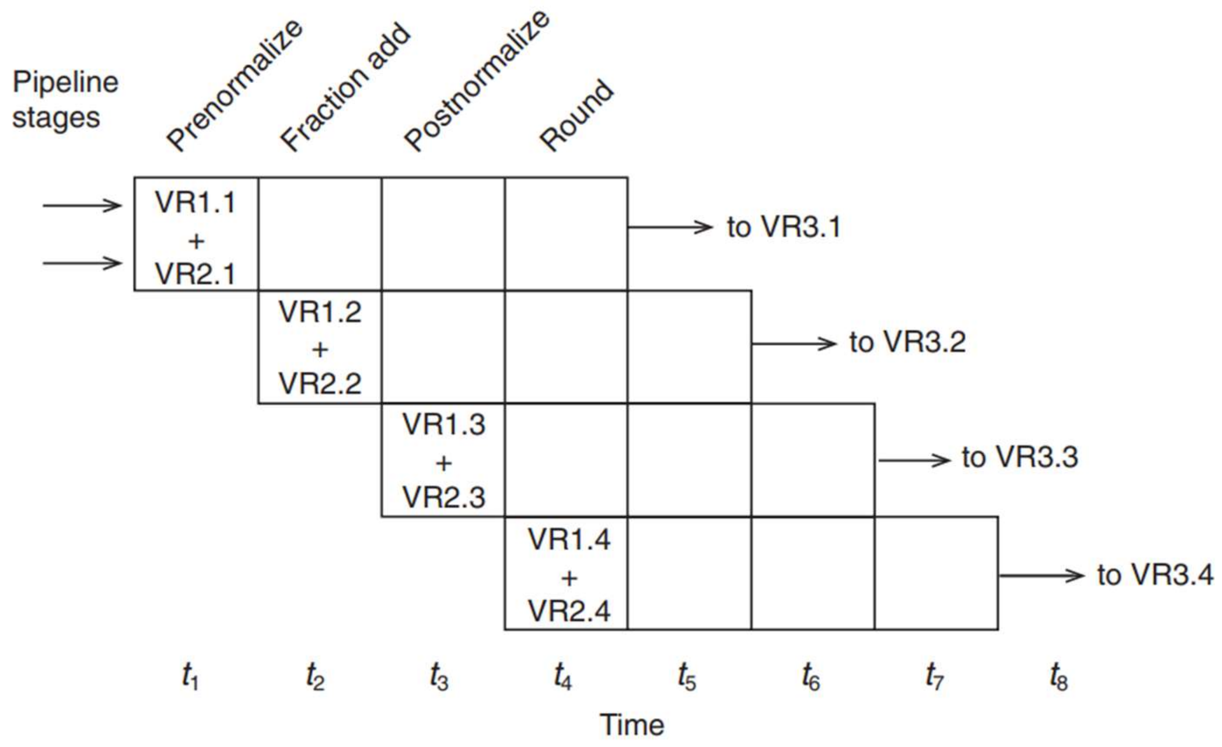
- **Vector Execution Units**: Independent functional units for each instruction class (e.g., add/subtract, multiplication, division, logical operations)
- A vector add **(VADD)** sequence passes through various stages in the adder.
- The sum of the first elements of VR1 and VR2 (labeled VR1.1 and VR2.1) are stored in VR3 (actually, VR3.1) after the fourth adder stage.

- **Pipelining** of the functional units is more important for vector functional units than for scalar functional units, where **latency** is of primary importance.

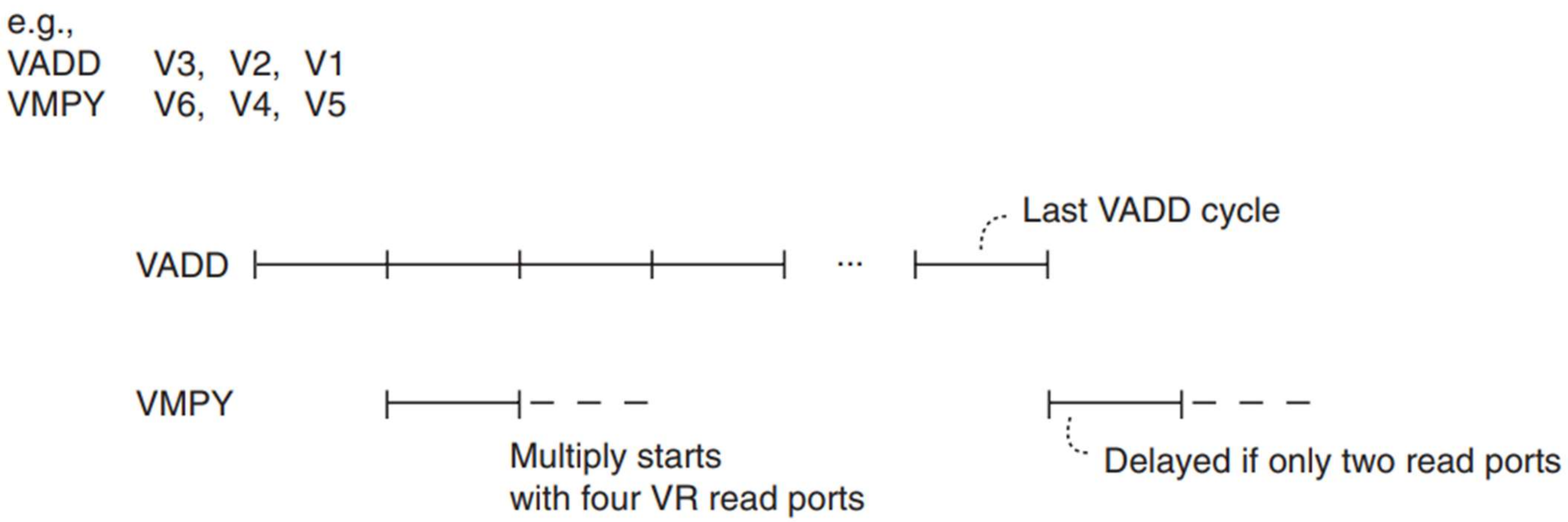
- The advantage of vector processing is that **fewer** instructions are required to execute the vector operations.

- A single (overlapped) vector load places the information into the VRs.

- The vector operation executes at the clock rate of



Approximate timing for a sample four - stage functional pipeline.



```
e.g.,
VADD    V3,  V2,  V1
VMPY    V6,  V4,  V5
```

Figure 3.20 For logically independent vector instructions, the number of access paths to the vector register (VR) set and vector units may limit performance. If there are four read ports, the vector multiply (VMPY) can start on the second cycle. Otherwise, with two ports, the VMPY must wait until the VADD completes use of the read ports.

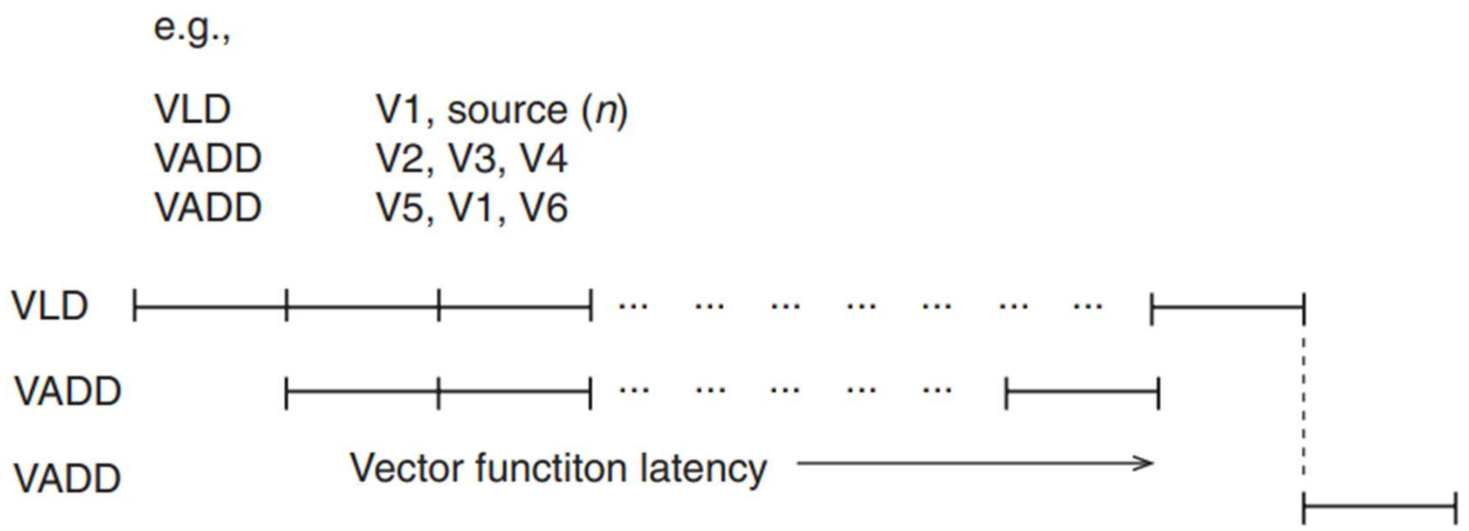


Figure 3.21 While independent `VLD` and `VADD` may proceed concurrently (with sufficient VR ports), operations that use the results of `VLD` do not begin until the `VLD` is fully complete.

Vector Processors

- The ability of the processor to concurrently execute multiple (independent) vector instructions is also limited by the number of VR ports and vector execution units.
- Each concurrent vector load or store requires a VR port.
- vector ALU operations require multiple ports.
- possible to execute more than one vector arithmetic operation per cycle.
- As with bypassing, the results of one vector arithmetic operation can be directly used as an operand in subsequent vector instructions without first passing into a VR.

Vector Processors

- It is illustrated in Figure by a chained ADD - MPY with each functional unit having four stages.

- If the ADD - MPY were unchained, it would take 4 (startup) + 64 (elements/VR) = 68 cycles for each instruction — a total **of 136 cycles**.

- With chaining, this is reduced to 4 (add startup) + 4 (multiply startup) + 64 (elements/VR) = **72 cycles.**

Example of vector chaining

- Vector Addition: **A = B + C**
- Vector Multiplication**: D = A * E**

- Without chaining, the multiplication **(A * E)** would wait until the entire vector addition **(B + C)** finishes.
- However, with vector chaining, the multiplication can begin as soon as the **first element of A** is ready (i.e., after the addition of the first elements of B and C), thus overlapping the execution of the two operations.

For these two instructions,

```
VADD    VR3, VR1, VR2
VMPY    VR5, VR3, VR4
```

the timing would be

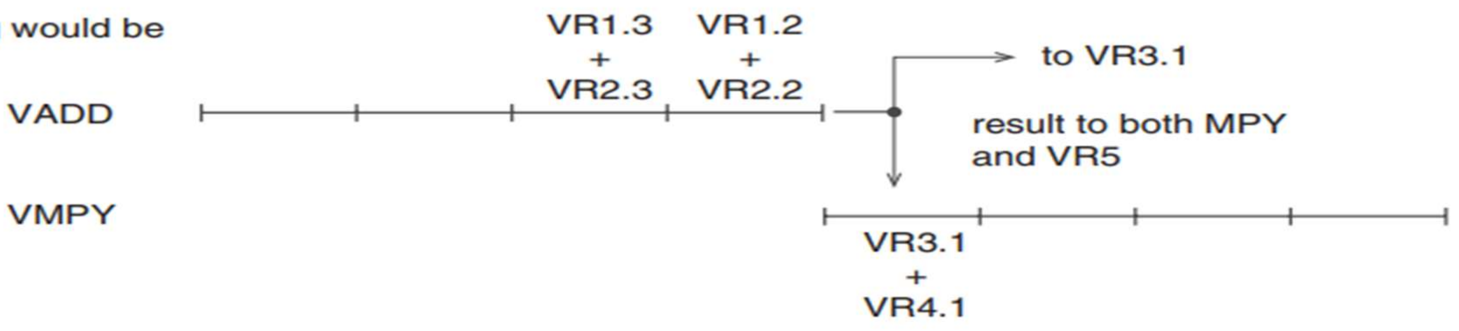


Figure Effect of vector chaining.

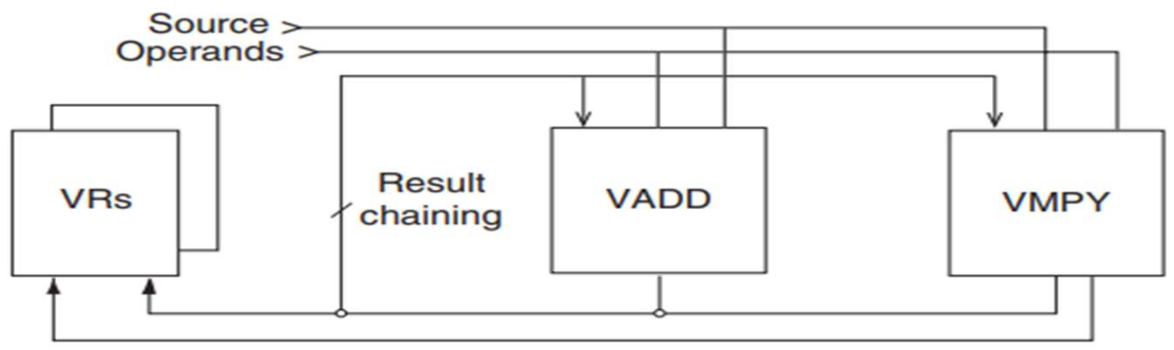


Figure Vector chaining path.

Benefits of Vector Chaining

- **Reduced Latency:** Operations that depend on the output of others can start earlier, reducing overall latency.

- **Higher Throughput:** It increases the parallelism and utilization of functional units, leading to better throughput.

- **Improved Resource Utilization:** Instead of keeping functional units idle, chaining ensures that they are used as soon as they are ready to process the next set of data.

Memory Management in Vector Processors

- Vector processors frequently access memory, since vector data needs to be loaded into Vector Registers (VRs) for processing, and results need to be stored back into memory

- Arithmetic operations complete one per cycle.

- Sufficient memory bandwidth is critical for fast execution.
    - For efficient performance, the memory system should support at least **two memory accesses per cycle**—one for reading new data (input vectors) and one for writing results (output vectors).

    - Preferably , **three memory accesses per cycle** are needed (two reads for input vectors and one write for the result), which allows vector operations to overlap and proceed without delays.

    - Enables concurrent vector reads/writes with arithmetic operations.

Concurrency in Vector Operations

- **Overlapping Memory Accesses and Arithmetic Operations**:
- The passage highlights that vector operations can run concurrently with memory accesses.
- For instance, while the processor is performing arithmetic operations on a vector, memory should simultaneously be loading new data into the VRs and writing old results out to memory.
- This concurrency minimizes idle time for the processor, making it more efficient.
- **Idle Processor Risk**: If the memory system does not provide sufficient bandwidth to feed the VRs with data quickly, the processor has to wait for memory operations to complete.

Elements of the Vector Processor

- **Functional Units**: These are the arithmetic units (e.g., add, multiply) that perform vector operations.

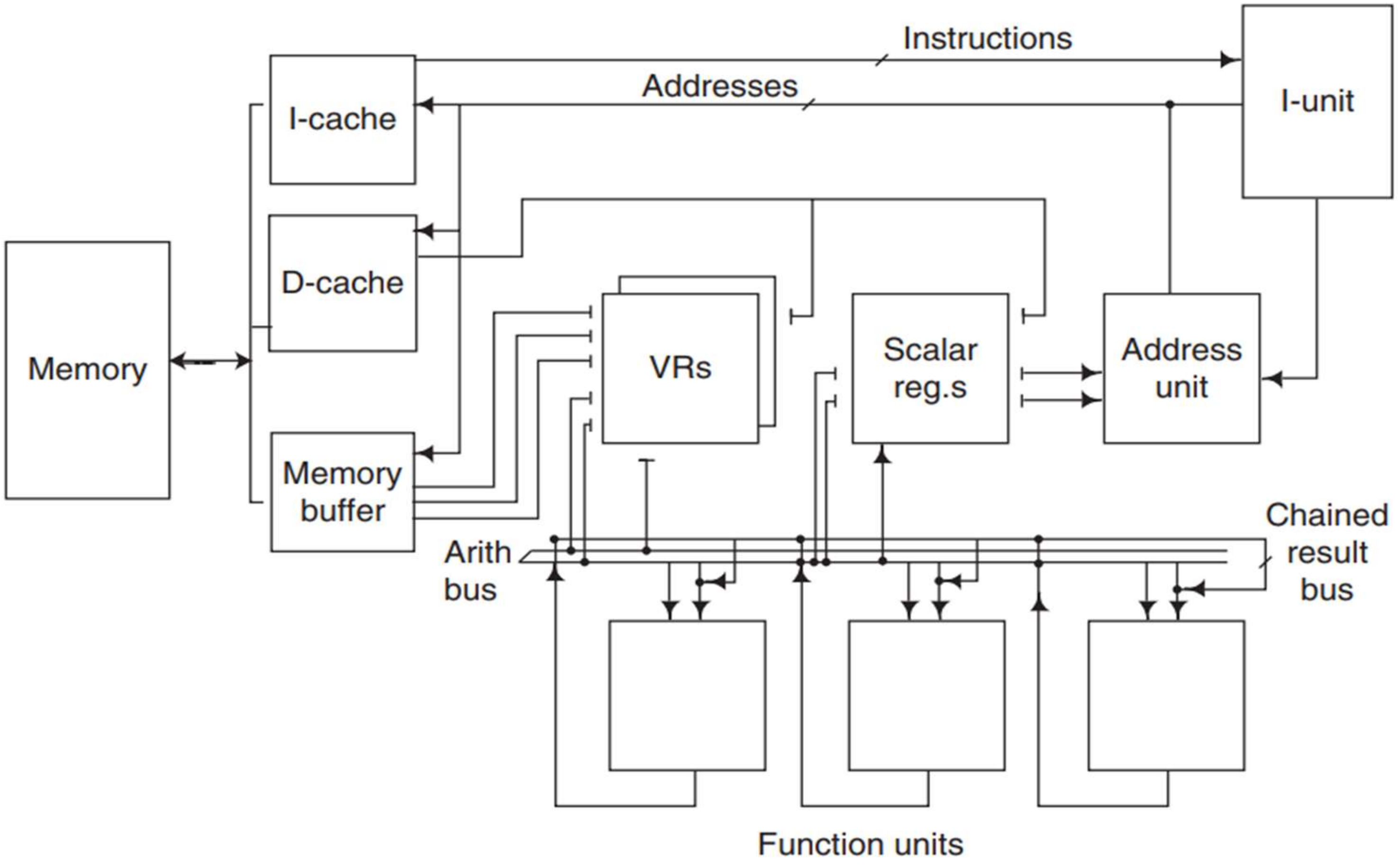
- **Vector Registers (VRs) and Scalar Registers**: The VRs store the vector data being processed, while scalar registers handle single data values.

- **Chaining**: If **chaining** is allowed, it enables multiple operations to be **executed in parallel** by accessing multiple source operands from the VRs and storing the result back to VRs, further increasing efficiency by reducing idle time between dependent operations.

Vector processor Architecture

- The system uses **multiple buses** to connect different components (VRs, memory, functional units) efficiently.
- This architecture ensures that data can move quickly between memory, VRs, and functional units, supporting fast vector operations.
- The processor also has typical **caches** (I-cache for instructions, D-cache for data) and general-purpose registers for handling non-vector (scalar) operations,

Major data paths in a generic vector processor



VLIW PROCESSORS

Two Classes of Multiple-Issue Machines

Statically Scheduled Processors:

These processors rely on the **compiler** to detect instruction dependencies and form groups of independent instructions into **instruction packets** before execution.

Dynamically Scheduled Processors:

In dynamically scheduled processors, instruction dependencies are also analyzed at compile time, but the **ultimate decision** about which instructions to execute together is made by the **hardware at runtime**.

VLIW (Very Long Instruction Word) Machines

- VLIW processors execute multiple operations in parallel by using very wide instruction words, each containing multiple **instruction fragments**. Each fragment controls a specific **execution unit** (e.g., arithmetic unit, load/store unit).

- The **register set** in VLIW processors needs to be highly multi-ported, meaning that multiple data elements must be accessed simultaneously by the various execution units.

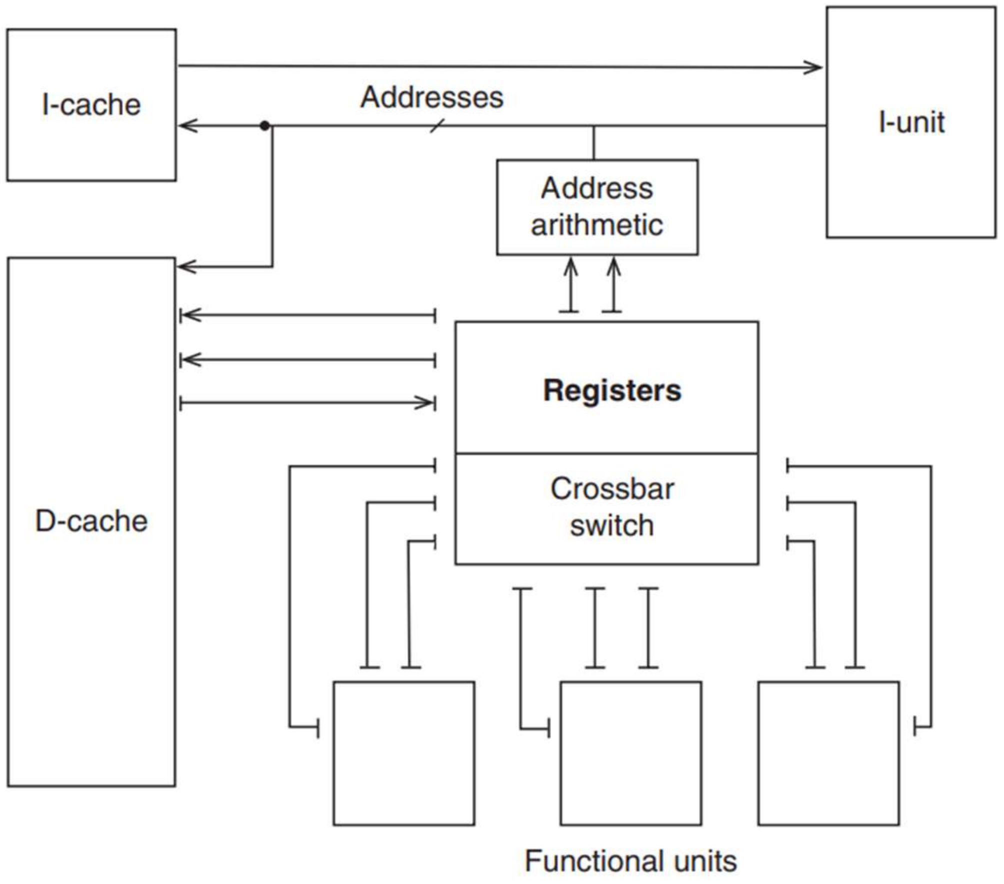
- This can potentially become a **bottleneck**, as the register set must handle many simultaneous accesses to maintain performance.

Simultaneous Multithreading (SMT)

- SMT allows multiple threads (or programs) to run simultaneously on the same processor, sharing execution hardware like adders, decoders, and execution units, but each thread has its own **register set** and **instruction counter**.

- **SMT** allows multiple threads to use the same processor hardware concurrently, increasing efficiency and throughput.

Major data paths in a generic VLIW processor



Superscalar Processors

- **Architecture**: Superscalar processors use **multiple functional units** and **buses** to execute multiple instructions simultaneously.

- These buses connect the **register set** to the functional units, allowing the processor to execute several instructions in parallel.

- **Concurrency Limitation**: While superscalar processors aim to maximize concurrency (parallel execution of instructions), the use of **shared buses** may limit the degree of concurrency.

- **Instruction Independence**: Superscalar processors detect independent instructions that can be executed in parallel.

Data Dependencies

- **Read-after-Write (RAW) Dependency (Essential Dependency)**

This type of dependency occurs when an instruction I_i writes to a register (or memory location), and a subsequent instruction I_j reads from that same location. This creates a dependency because I_j must wait until I_i finishes writing before it can read the correct value.

I1: DIV R3, R1, R2 (writes to R3)

I2: ADD R5, R3, R4 (reads from R3)

instruction I_2 depends on the result of I_1 because I_2 reads from register R_3, which is written to by I_1.

Data Dependencies

Write-after-Read (WAR) Dependency (Ordering Dependency)

This type of dependency occurs when a later instruction I_j writes to a register or memory location that an earlier instruction I_i reads from. This happens only in **out-of-order execution**, when the order of execution changes from the original program sequence.

I1: DIV R3, R1, R2 (writes to R3)
I2: ADD R5, R3, R4 (reads from R3)
I3: ADD R3, R6, R7 (writes to R3)

If instruction I_3 executes before I_2, it could **overwrite the value** in R_3 before I_2 reads it. This would result in I_2 using an incorrect value.

Data Dependencies

Write-after-Write (WAW) Dependency (Output Dependency)

This occurs when two instructions, I_i and I_j, write to the **same destination** (register or memory location). If executed out of order, the final value in the register could be incorrect.

I1: MUL R3, R1, R2 (writes to R3)

I2: ADD R3, R4, R5 (writes to R3)

if I_1 completes after I_2, it would **overwrite** the result of I_2, leaving an incorrect value in R_3. This is called a **WAW dependency**, and it affects the order in which results are stored in registers.

Handling Dependencies

- **RAW dependencies** are essential because they deal with the data flow of a program and must be respected to ensure correct execution.
- **WAW and WAR dependencies** are mainly about managing execution order and only arise in **out-of-order processors**.
- The **fewer dependencies** there are in a sequence of instructions, the more instructions can be executed in parallel.
- The fewer the dependencies in the code, the more **concurrent execution** is possible, which speeds up the overall program execution.

Detecting Instruction Concurrency

- **Compile-time and Run-time Detection**: The detection of independent instructions that can be executed concurrently can happen both during **compilation** (by the compiler) and at **run-time** (by the hardware).
- The **compiler** can optimize the code by unrolling loops and creating larger blocks of code with fewer branches, which increases the opportunities for parallel execution.
- At **run-time**, the hardware has the full knowledge of the system's current state (e.g., cache hits or misses) and can dynamically adjust the execution of instructions based on available resources.

Detecting Instruction Concurrency

```
for (int i = 0; i < 4; i++) {
    array[i] = array[i] * 2;
}

array[0] = array[0] * 2;
array[1] = array[1] * 2;
array[2] = array[2] * 2;
array[3] = array[3] * 2;
```

Why it's beneficial:

Fewer iterations,

Better use of pipelines, Unrolling loops allows the processor to make better use of its pipelines because it has more consecutive instructions to work on without interruption.

Checking for Dependencies During Decode

- **Instruction Decode**: When the processor decodes instructions, it checks for dependencies between them to determine which instructions can be executed concurrently.
- If there are no dependencies and there are enough resources (functional units, registers), the instruction is issued to the appropriate **functional unit** for execution.
- **Instruction Window**: The number of instructions checked at a time is determined by the size of the **instruction window**.

Examples of Dependencies

I1: DIV R3, R1, R2 (writes to R3)

I2: ADD R3, R4, R5 (reads from R3)

Instruction I2 depends on the result of I1 because both instructions use R3. This is a read-after-write (RAW) dependency, meaning I2 cannot execute until I1 completes.

I1: DIV R3, R1, R2 (writes to R3)

I2: ADD R5, R3, R4 (reads from R3)

I3: ADD R3, R6, R7 (writes to R3)

I3 might overwrite **R3** before **I2** has finished using it, which creates a **write-after-read (WAR) dependency**.

Dependencies like these can limit how many instructions are issued simultaneously.

Instruction Issuing and Scheduling

- **Renaming Registers**: To handle **ordering and output dependencies**, modern processors often use **register renaming**. This involves assigning temporary or extra registers to avoid conflicts when multiple instructions try to use the same register. A typical processor may extend its register set (e.g., from 32 to 45–60 registers) to support this renaming.
- **Instruction Scheduling**: This is the process of assigning instructions to the appropriate resources (e.g., functional units) at the correct time. There are two main approaches to scheduling:

- Control Flow Scheduling: Dependencies are resolved centrally during the decode stage, and instructions are issued only when their dependencies have been resolved.

- Data Flow Scheduling: Instructions are issued as soon as they are decoded and held in buffers at the functional units until their operands are ready and the functional unit is available.

Memory Design: System - on - Chip and Board - Based Systems

Memory Design's Central Role in System Performance:

- Memory design is often the most significant aspect of SoC systems in terms of both cost (area or die size) and its impact on performance.
- Memory systems hold essential data (operands) and instructions required by the processor.
- Therefore, regardless of the power or speed of the processor and interconnects, the system speed is primarily bound by the memory's ability to quickly supply data and instructions.

Memory Design: System - on - Chip and Board - Based Systems

Considerations in Memory Design:

- Memory requirements are driven by the specific application requirements: the operating system in use, the application's data size, and the degree of variability in application processes.
- These factors decide how much memory is needed and whether addressing will be real or virtual.

Memory Design: System - on - Chip and Board - Based Systems

On-Die and Off-Die Memory Access:

- on-die memory (integrated on the same chip as the processor) versus off-die memory (external to the chip).

- **Access Time Difference**: On-die memory access is significantly faster (3-10 cycles) than off-die access, which takes 30-100 cycles. This performance difference impacts how memory should be managed and structured to meet system speed requirements.

Off-Die Memory and Cache Requirements:

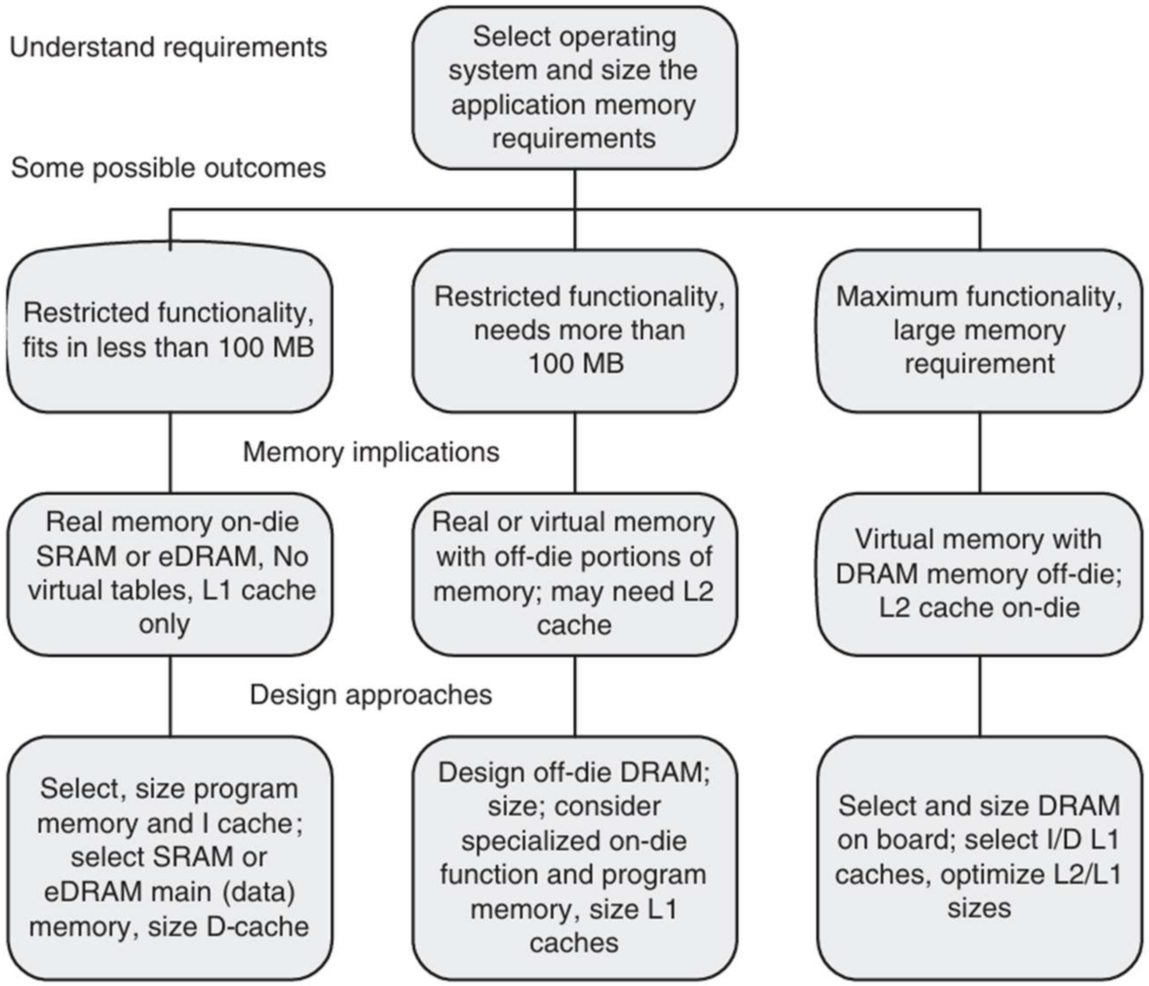
- If memory needs are large and cannot all fit on-die, systems rely on off-die memory.
- To offset the slower off-die access speeds, more cache memory is needed, often organized in multiple levels (L1, L2, etc.) to reduce latency.
- However, this additional cache is **costly**, as cache memory cells (bits) can be 50 times larger than those in embedded DRAM (eDRAM) used for on-die memory.

Memory Design: System - on - Chip and Board - Based Systems

- **Access Time Difference**: On-die memory is faster, typically accessible in 3–10 cycles, while off-die memory takes much longer (30–100 cycles).
- Thus, to achieve similar performance, off-die systems require larger caches, often split into multiple levels.

- **Cache Size and Cost**: Off-die memory requires substantially more cache, which can be much larger than on-die embedded DRAM (eDRAM) bits.
- The example compares cache bits to eDRAM bits, suggesting that a single off-die cache bit can occupy 50 times the area of an eDRAM bit.

An outline for memory design


Understand requirements
Select operating system and size the application memory requirements
Some possible outcomes
Restricted functionality, fits in less than 100 MB
Restricted functionality, needs more than 100 MB
Maximum functionality, large memory requirement
Memory implications
Real memory on-die SRAM or eDRAM, No virtual tables, L1 cache only
Real or virtual memory with off-die portions of memory; may need L2 cache
Virtual memory with DRAM memory off-die; L2 cache on-die
Design approaches
Select, size program memory and I cache; select SRAM or eDRAM main (data) memory, size D-cache
Design off-die DRAM; size; consider specialized on-die function and program memory, size L1 caches
Select and size DRAM on board; select I/D L1 caches, optimize L2/L1 sizes

Flash memory technology and its integration into System-on-Chip (SoC) designs

- Flash memory is an evolving technology that is not necessarily a direct replacement for traditional memory types (like DRAM) but is more effectively viewed as a replacement for disk storage.

- It provides a form of nonvolatile storage, meaning that it retains data even when power is turned off.

- Flash can serve both as memory and as a backup storage solution, allowing for flexibility in system design.

Structure of Flash Memory:

- Flash memory uses an array of floating-gate transistors, which resemble MOS transistors but with a two-gate structure (a control gate and an insulated floating gate).

- The floating gate traps charge, making flash memory nonvolatile (data persists without power).

- Flash data is rewritable but has a limited number of write cycles, typically fewer than one million.

Structure of Flash Memory

- **Flash Memory in Various Package Formats:**

Flash cards come in various sizes, with larger cards often being older technology.

Flash can be “stacked” in SoCs, meaning small flash chips are combined with SoC chips to create compact single-package systems or larger flash stacks for high-capacity storage (up to 64–256 GB).

SoC Internal Memory: Placement and Performance Factors

- SoC Internal Memory: Placement and Performance Factors
- **Importance of Memory Placement**: The main memory can be either on-die (on the same chip as the processor) or off-die (on a separate chip).
- On-die placement provides faster access and is often preferred for high performance, distinguishing SoC designs from traditional workstations.
- **Two Key Performance Parameters**:

Access Time: The time it takes for data to travel between the processor and memory. Access time depends on the physical distance (bus delay, chip delay, etc.) between memory and processor.

Memory Bandwidth: The volume of data the memory can supply to the processor within a certain time.

Bandwidth is influenced by memory organization, the number of independent memory arrays, and modes for sequential data access.

SoC Internal Memory: Placement and Performance Factors

- **Role of Cache**: Cache helps bridge the performance gap caused by limits in memory access time and bandwidth, ensuring that data the processor frequently accesses is readily available.

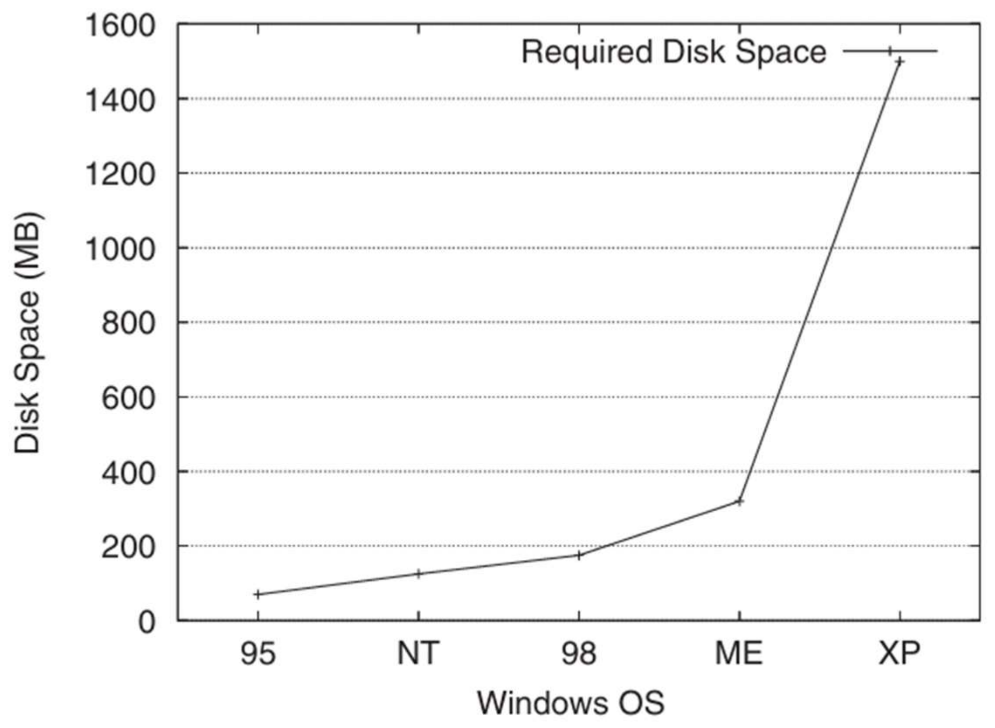
- **Design Challenges for Workstation Processors and Board-Based Systems**: For workstations with off-die memory, memory design becomes complex and costly. Effective cache design is crucial to mitigate the challenges of off-die placement.

The Size of Memory for SoC

- **Off-Die Memory as a Key Challenge**:
- In system board designs, managing large off-die memory (memory not on the main chip with the processor) is a significant challenge because accessing it takes longer and can slow down performance.
- This issue becomes central when designing systems requiring substantial memory.

- **Why Not Limit Memory Size to Fit On-Die?**:
- Although it would simplify the design, restricting memory to sizes that fit on-die (integrated into the chip with the processor) isn’t feasible. This is due to the large memory demands of modern applications, especially in virtual memory systems.

The Size of Memory



Required disk space for several generations of Microsoft ’ s Windows operating system. The newer Vista operating system requires 6 GB.

The Size of Memory for SoC

- **Growing Memory Needs in Workstations**: In workstations, the operating system and applications continue to grow in size and complexity.
- This growth means the "working set" (or active pages of data currently in use) is larger, requiring more physical memory to avoid performance-degrading page swapping.

- **Memory Demands in Board-Based Systems**: Board-based systems face a different challenge: handling large media-based data sets (like videos or graphics), which require high memory **bandwidth** and significant processing power from the media processor.
- Unlike workstations, board-based systems don’t have as much trouble with memory access time (the time it takes to access data in memory), as long as they meet the bandwidth requirements.

SCRATCHPADS AND CACHE MEMORY

Scratchpad Memory is directly controlled by the programmer.

Programmers manually manage what data is stored in the scratchpad memory, which can lead to optimized performance if they anticipate what data will be needed.

Cache Memory, on the other hand, is managed by the hardware.

The processor automatically stores frequently accessed data or instructions here, making it easier to use for general-purpose applications because the management process is automated.

SOC (System-on-Chip) and Scratchpad Memory:

- While general-purpose computers mostly use cache memory due to the complexity of managing memory manually, SOCs allow for the possibility of using scratchpad memory.
- In an SOC, if the application is well-defined, the programmer can optimize performance by carefully controlling data storage and transfers.
- **Advantages of Scratchpad in SOC**: Eliminating cache control hardware saves space on the chip, allowing for a larger scratchpad memory.

Caches and Locality Principles:

- Caches work on the basis of the locality of program behavior , Three principle involved.
  - **Spatial Locality**: If a specific memory location is accessed, neighboring locations will likely be accessed soon.
  - **Temporal Locality**: Data or instructions that have been used once are likely to be reused.
  - **Sequentially**: If location "s" is accessed, it's likely that "s+1" will be accessed next, which is a type of spatial locality.
- **Cache Design**: The cache must strike a balance between the **processor's needs** for quick access and the **memory system's capacity**, while keeping costs low.

Key Parameters of the Processor-Cache Interface

Cache Hits and Misses:

- A **cache hit** occurs when the processor finds the needed data in the cache, allowing for fast access.

- A **cache miss** happens when the data isn't in the cache, forcing the system to retrieve it from a slower memory source (such as main memory).

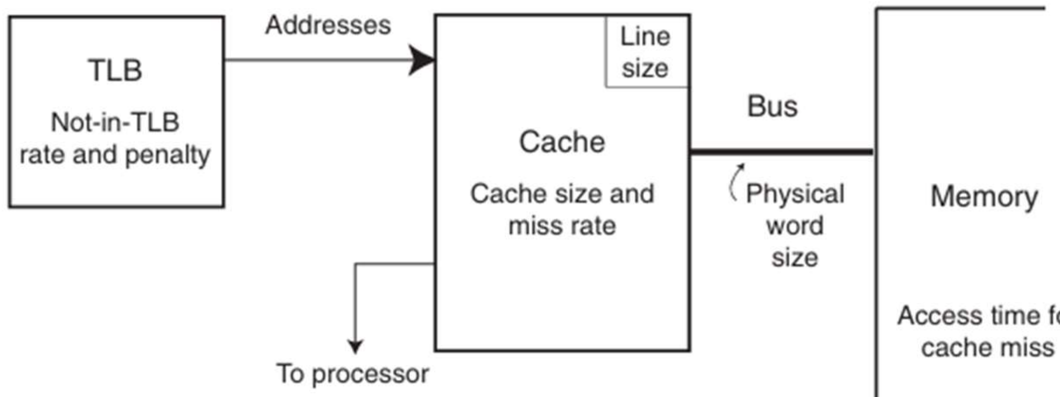
Key Parameters of the Processor-Cache Interface

- **Physical Word**: This is the basic unit of data transferred between the processor and cache. sizes ranging from 2–4 bytes in smaller processors to 8 bytes or more in more advanced processors.
- **Block Size (or Line)**: The block size determines how much data is transferred between the cache and main memory in one go.
- **Access Time for a Cache Hit**: This measures how quickly the cache can deliver data to the processor on a hit.
- **Cache Miss Rate**: The **miss rate** is the frequency with which cache misses occur.
- **Access Time for a Cache Miss**: The time it takes to retrieve data from main memory and place it in the cache after a miss.
- **Real Address Calculation**: When a virtual address (used by programs) is converted to a physical memory address, additional time may be needed if it's not in the Translation Lookaside Buffer (TLB).

Cache as Part of the Processor:

- In many designs, **first-level cache** is embedded within the processor, enhancing speed by providing data close to the processing unit.
- This integration is especially important in **System-on-Chip (SOC)**, where multiple processors may share memory.
- In SOCs, managing the memory hierarchy across multiple processors (not just isolated caches) is essential to ensure data consistency and system efficiency.

Parameters affecting processor performance



CACHE ORGANIZATION

Fetch Strategies:

Fetch-on-Demand:

- This strategy loads data into the cache only when it's specifically requested by the processor.
- It's commonly used in simple processors and is straightforward, fetching data only when a **cache miss** occurs.

Prefetch:

- Here, the cache anticipates what data the processor might need next and loads it in advance.
- This strategy is often used in instruction caches (I-caches) to improve speed by proactively fetching instructions.

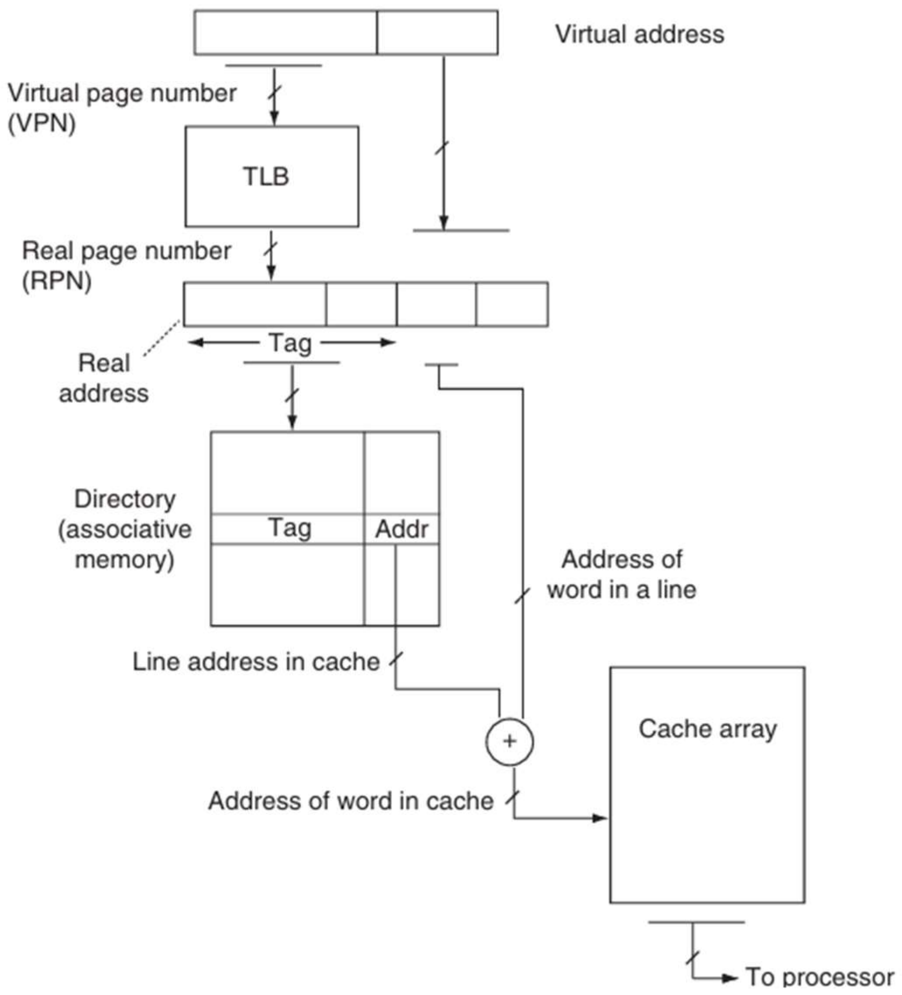
Types of Cache Mapping

- **Fully Associative (FA) Mapping**:
- **Direct Mapping**:
- **Set-Associative Mapping**:

Fully associative mapping.

- In this type, a requested data address is compared with all addresses stored in the cache to find a match.
- This exhaustive search makes FA caches flexible but can slow down access due to the high comparison workload, especially in large caches.

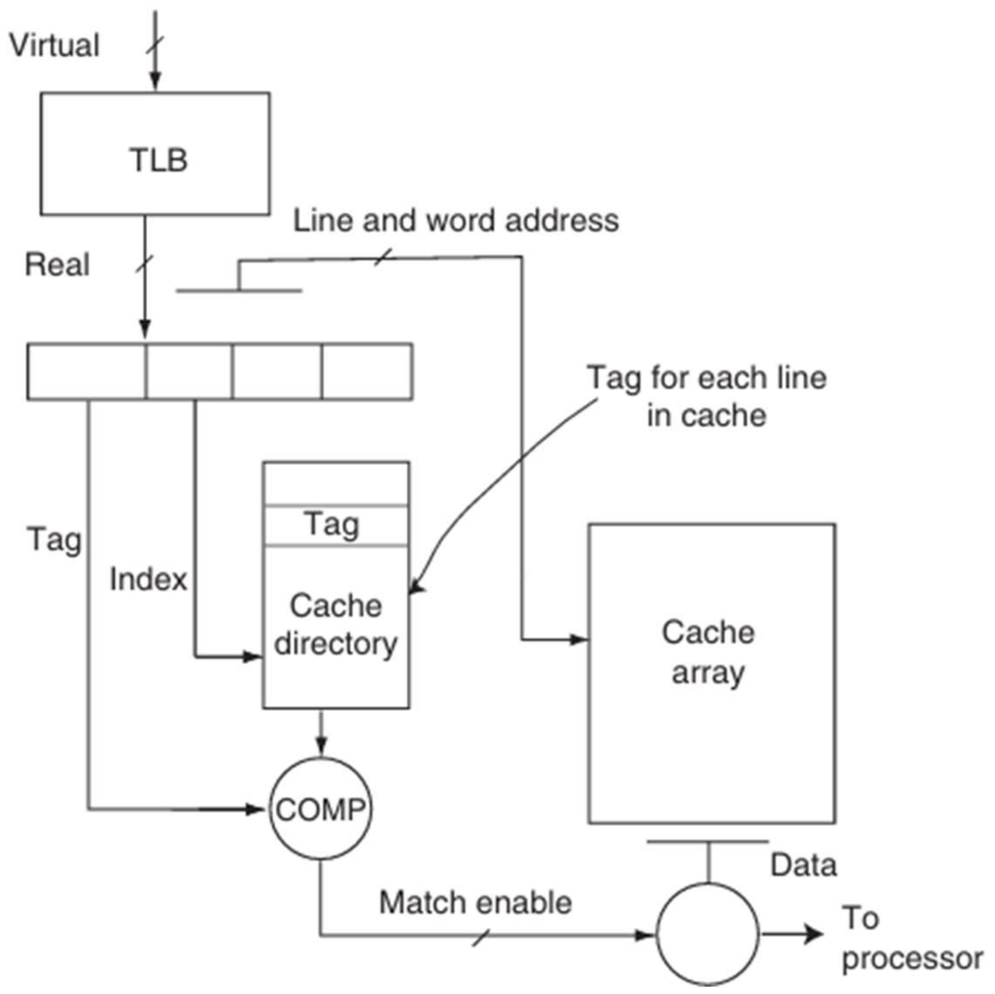
Fully associative mapping



Direct Mapping:

- Each memory location has only one possible location in the cache (based on the address's lower bits, known as **index bits**).
- However, several memory locations may share this one slot, which means that conflicts can occur.
- The **tag bits** are then used to verify that the correct data is in the cache.
- The advantage of direct mapping is that both the cache and directory can be accessed simultaneously, enabling quick access when the required data is in cache.

Direct Mapping



Set-Associative Mapping

- This is a hybrid between FA and direct mapping. Here, each memory location can map to several possible slots in the cache (often two or four).
- This organization divides the cache into multiple "sets" (subcaches), and each set can hold more than one line address, allowing for flexibility and faster access than FA.
- Multiple locations can be checked simultaneously for a match, so it generally provides faster access than fully associative caches.

Set-Associative Mapping

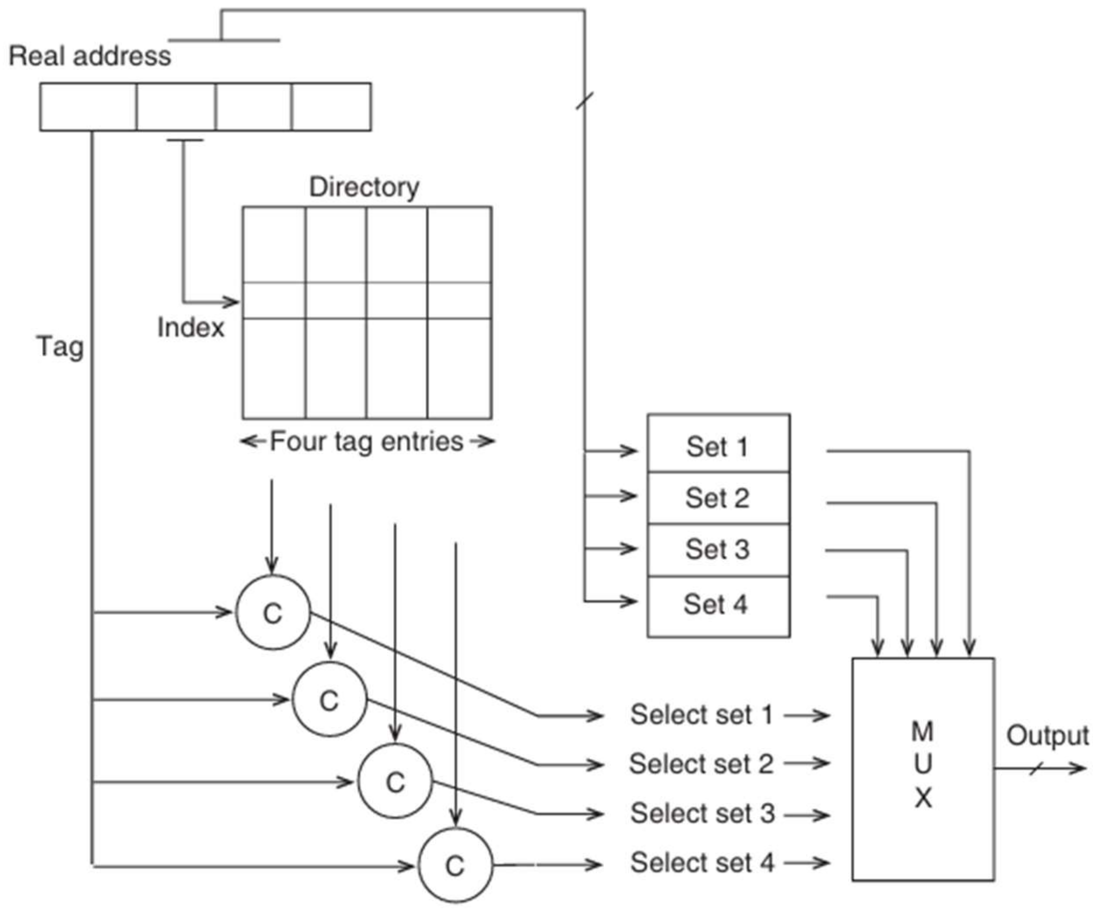
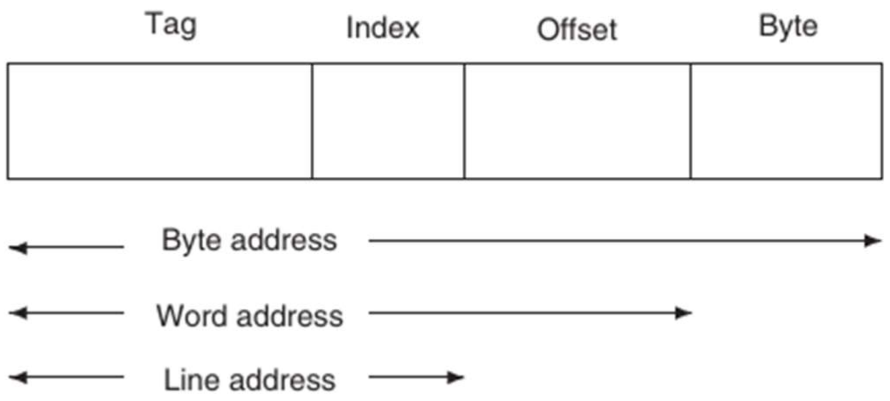


Figure 4.7 Set associative (multiple direct-mapped caches) mapping.

Definitions of Address Components

- **Tag**: The most significant address bits used for matching in the directory to check if the requested data is present.
- **Index**: The middle portion of the address, used to locate a line within the cache directory.
- **Offset**: This specifies the exact location within a cache line (line is the unit of data transferred between cache and memory).
- **Byte**: The least significant bits that specify a particular byte in a word, though these are typically only relevant for specific write operations.

Address partitioned by cache usage



Trade-offs Between Cache Types

- **Fully Associative**: Offers flexibility but may slow down access due to the time needed to search all entries.
- **Direct Mapping**: Fastest for accessing data but limited because each memory address can map to only one cache line, increasing the chance of conflict.
- **Set-Associative**: Balances between FA and direct mapping, allowing some flexibility with generally faster access times than FA, though slightly slower than direct-mapped.

Cache Size and Miss Rate

- **Cache size** is a critical factor in determining **cache performance**, specifically the miss rate (the frequency with which data is not found in the cache and must be fetched from main memory).
- **Larger caches** generally have **lower miss rates** because they can store more data, reducing the chance of a "miss" when requested data isn't in the cache.
- **Empirical limitations**: Most cache miss rate data is based on empirical (measured) data. These results depend heavily on the specific program being run, as cache behaviour is influenced by the program's structure and memory access patterns.

Cache Size and Miss Rate

- **Historical data bias**: Early cache studies were done on smaller programs and smaller memory sizes, so they show lower miss rates for small caches. As programs and memory sizes have grown, miss rates have tended to increase over time.
- **Smith's DTMRs**: A designer named Smith developed a set of **design target miss rates** (DTMRs) that provide estimated miss rates for caches based on size. Typically, doubling the cache size halves the miss rate, though this doesn’t hold as well for transaction-based programs.

Write Policies for Cache and main memory

- **Write-through (WT)**: Data is written simultaneously to both the cache and main memory.

  Advantage: Keeps main memory and cache data consistent.

  Disadvantage: Increases memory bandwidth usage, as every write operation goes to both places.

- **Copy-back (CB)** (or **write-back**): Data is only written to the cache, and main memory is updated only when the data is replaced in the cache.

  Advantage: Reduces the number of writes to main memory, lowering memory traffic.

  Disadvantage: Requires tracking modified lines (using a "dirty bit") that haven’t been written back to main memory.

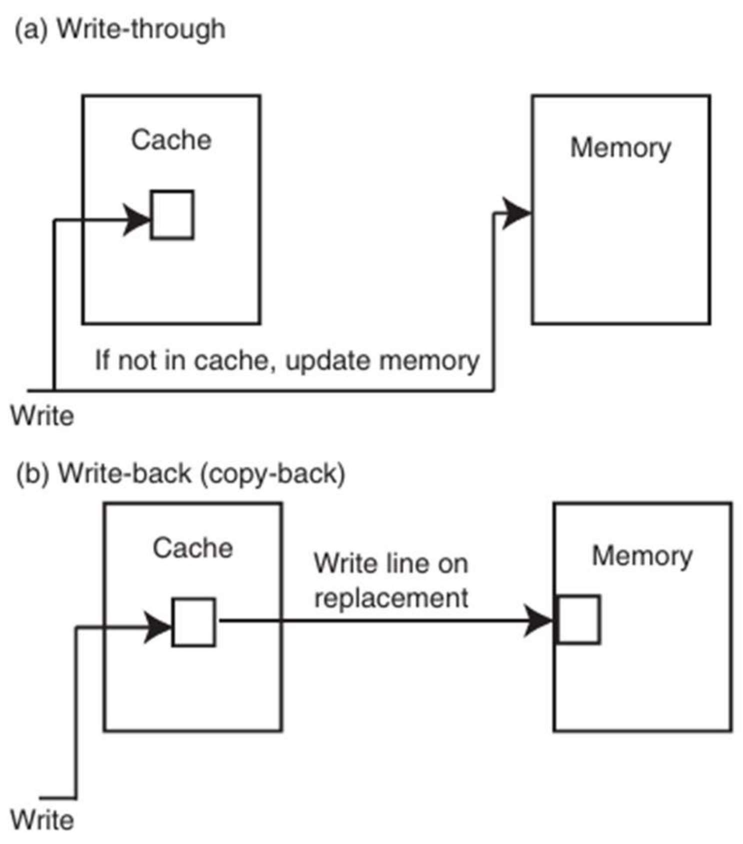
Cache Types and Write Misses

- Copy-back is generally preferred for larger caches due to its efficiency in reducing memory traffic.
- Write-through is often used for smaller or special-purpose caches to ensure memory consistency.

Write miss handling: When a write operation does not find the data in the cache (a write miss), there are two policies:

- Write allocate (WA): The missing data is fetched into the cache before the write, commonly used with copy-back caches.
- No write allocate (NWA): Data is written directly to memory without bringing it into the cache, usually found in write-through caches.

Cache Types and Write Misses


(a) Write-through
Cache
Memory
If not in cache, update memory
Write
(b) Write-back (copy-back)
Cache
Write line on replacement
Memory
Write

Write policies: (a) write - through cache (no allocate on write) and (b) copy - back cache (allocate on write).

Cache miss and the strategies

When a **cache miss** occurs (i.e., the requested data is not found in the cache):

1. **Fetching the missed line**: The required data line must be fetched from main memory and brought into the cache.

2. **Replacing an existing line**: Since cache space is limited, a line currently in the cache must be designated for replacement by the new line (missed line).

Cache miss and the strategies

- **Fetching a Line**

Write-through cache: The missed line is fetched, and the line it replaces is discarded. No additional action is needed because all writes have already been propagated to main memory.

Copy-back (write-back) cache: If the line to be replaced is "dirty" (i.e., it has been modified in the cache but not yet written to memory), it must first be written back to main memory before it can be discarded.

If the line is "clean," it can be discarded immediately.

Cache miss and the strategies

- Line Replacement Policies: When a line must be replaced, the cache uses a **replacement policy** to decide which line to discard.

Least Recently Used (LRU):

- The line that hasn’t been accessed for the longest time is replaced. is therefore the most effective choice.
- However, it is complex to implement, as each line needs a counter to track how recently it was accessed.
- In practice, approximations of LRU with smaller counters are often used.

First In, First Out (FIFO):

- The line that has been in the cache the longest is replaced.
- This method is simpler but doesn’t always match access patterns as closely as LRU.

Random Replacement (RAND):

- A line is chosen randomly for replacement. While less efficient, it’s very simple to implement.

Cache Environment: Effects of System, Transactions, and Multiprogramming

- **System Effects on Cache:**

Most cache performance data are obtained from application traces, which capture cache usage patterns of actual programs. However, these applications run within a larger system context (i.e., with an operating system and possibly other programs).

- **Operating System Influence**:

The operating system typically increases the cache miss rate of applications by around 20% because it introduces additional processes and data, which can evict or replace cache data the application might need.

Cache Environment: Effects of System, Transactions, and Multiprogramming

- **Warm Cache (Multiprogrammed Environment)**:
  - Multiple programs reside in memory, and control switches between them after each has executed a set number of instructions (denoted Q).
  - When a program resumes execution, it finds only some of its previously cached data still in the cache.
  - Some of its cache data will have been replaced due to the other programs' activities, resulting in higher miss rates. This is known as a **warm cache**: some data from the previous session remains, but much of it is replaced.
- **Effect**: Increased miss rate when a program resumes because it has to reload some of its data.

Cache Environment: Effects of System, Transactions, and Multiprogramming

- **Cold Cache (Transaction Processing)**:
- In this environment, shorter programs (transactions) run to completion without interruption, often in the presence of a few supporting programs and the operating system.
- Each transaction typically runs through its full instruction sequence (completing Q instructions) and is not resumed later. This scenario is called a **cold cache** because each transaction starts with no relevant data in the cache, leading to higher miss rates initially as the cache must be loaded from scratch for each new transaction.
- **Effect**: The cache starts "cold" with each transaction, resulting in higher miss rates until the cache fills with relevant data.

Interconnect

- To discuss the challenges and options involved in connecting various components within a System-on-Chip (SOC) design, specifically focusing on interconnect architectures.

SOC Design & Integration of IP Cores

- SOC designs often involve combining various pre-designed and verified components (or intellectual property [IP] cores).

- System integrators aim to reuse existing designs as much as possible to save on costs and reduce risks.

- However, a crucial challenge is determining the best way to connect these IP cores.

Interconnect Architectures

Unlike traditional computer buses, SOC interconnect architectures offer several methods. Two major types mentioned are:

- **Bus Architecture**: A simpler form of interconnect designed specifically for SOC, which is discussed in comparison with other methods.

- **Network-on-Chip (NOC)**: This is a more complex interconnect that uses switches (instead of a simple bus) to connect IP cores.
- Unlike a direct or custom connection, NOC uses standard switches that can be reused for various IP cores.

Basic Structure and Connectivity

- **Bus Architecture**:
- In a bus architecture, all units (e.g., processors, memory, I/O devices) connect to a single shared communication line.
- Only one unit can communicate over the bus at a time, leading to potential bottlenecks as the system scales.
- Typical in simpler SOCs or systems with fewer components, where limited data needs to be moved between units.
- **NOC (Network-on-Chip)**:
- NOC is a network-like structure on a chip, where multiple switches or routers connect different components (nodes).
- Units communicate through network packets, allowing multiple, parallel communication paths.
- Ideal for complex SOCs with many cores and components, as it scales better with increased communication needs.

Switch-Based Interconnects in NOC

In NOC, the connection happens through switches. The switch can take different forms:

- **Crossbar**: A type of switch that allows any unit to connect directly with any other.

- **Directly Linked Interconnect**: Units connect directly to each other without intermediaries.

- **Multistage Switching Network**: A more complex setup that routes data through multiple switching points.

Overview of System-on-Chip (SOC) interconnect architectures

SOC Module Components: an SOC module containing various IP blocks (pre-designed and verified components). These typically include:

- **Processors**: The main computing units.
- **On-Chip Memory**: For cache, data, and instruction storage.
- **Application-Specific IP Blocks**: These are specialized processors or units, like graphics processors, video codecs, and network controllers, that perform specific functions within the SOC.

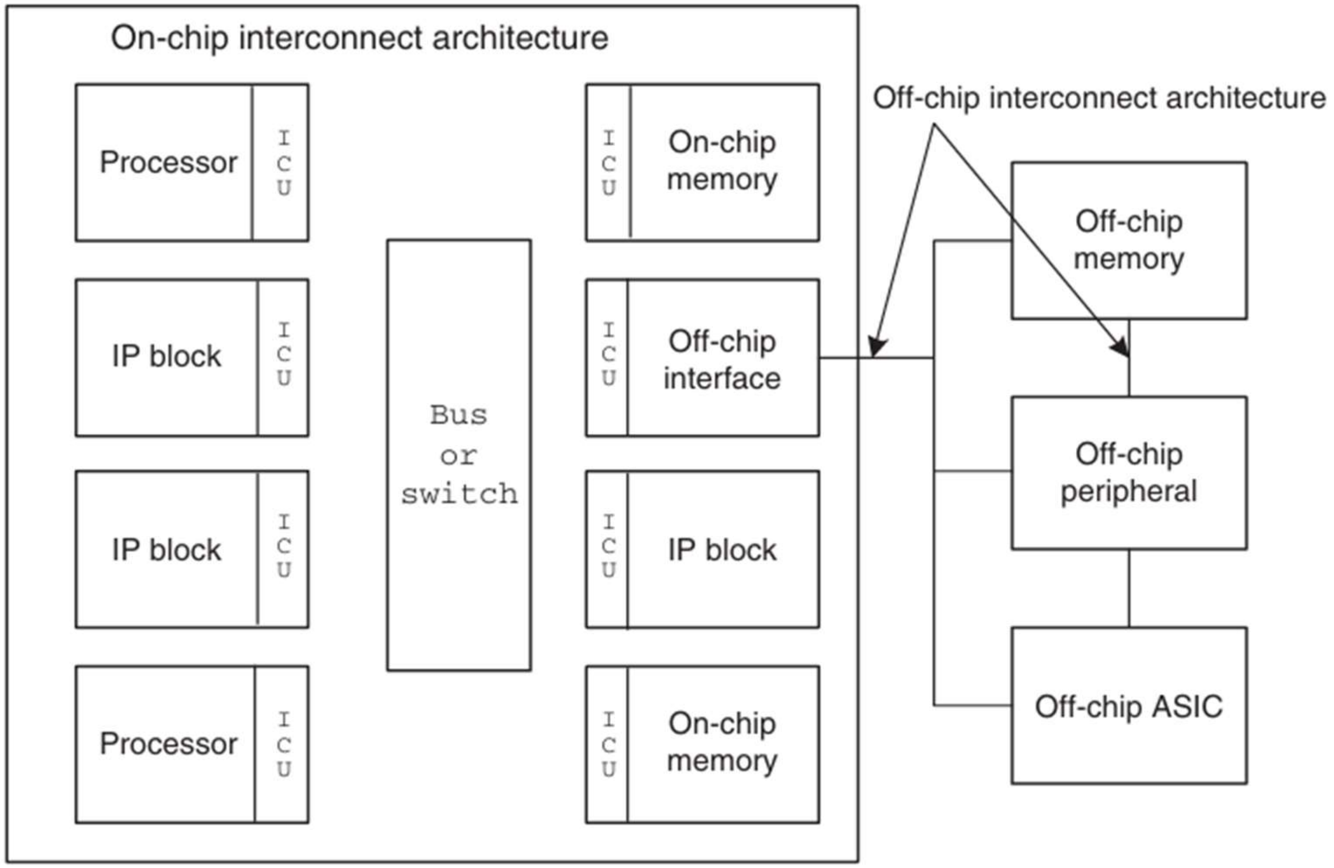
System-on-Chip (SOC) interconnect architectures

- **Communication via Interconnect**: The IP blocks within the SOC need to communicate with each other.
- They do so through an interconnect, accessed through an Interconnect Interface Unit (ICU). The ICU standardizes the communication protocol, ensuring compatibility among different SOC modules.

- **External Connections**: Outside the SOC, off-chip components (such as external memory, peripheral devices, and storage) are connected to the SOC.

The overall performance and cost depend on the efficiency of both on-chip (within the SOC) and off-chip (external) interconnects.

Overview of System-on-Chip (SOC) interconnect architectures



A simplified block diagram of an SOC module in a system context

Overview of System-on-Chip (SOC) interconnect architectures

Choosing a suitable interconnect architecture requires the understanding of a number of system level issues and specifications

- **Communication Bandwidth**: This is the amount of data transferred per second between modules. Higher bandwidth is needed for high-throughput applications.
- **Communication Latency**: This measures the delay between a module's request for data and the response. Latency can significantly impact applications like real-time communication but may have minimal effect on video streaming.
- **Master and Slave Roles**: In SOC, a "master" (like a processor) initiates communication, while a "slave" (like memory) responds. SOC designs usually contain multiple masters and slaves.
- **Concurrency Requirement**: The number of independent simultaneous communication channels operating in parallel. Usually, additional channels improve system bandwidth.

Overview of System-on-Chip (SOC) interconnect architectures

- **Packet or Bus Transaction**: Information exchange differs based on the interconnect architecture:
- **Bus**: Uses addresses, control bits (read/write), and data.
- **NOC**: Uses packets with headers (address, control info) and payload (data).
- **ICU (Interconnect Interface Unit)**: Manages protocol and physical transactions. It may support complex functions like buffering and out-of-order transactions. In buses, a protocol translation unit is called a "bus wrapper"; in NOC, it manages packet transport.
- **Multiple Clock Domains**: Different IP modules may run at distinct clock rates, creating separate timing regions called clock domains. This requires careful handling to avoid timing and synchronization issues.

Overview of System-on-Chip (SOC) interconnect architectures

- **Exploring Interconnect Requirements**:
  - Designers must consider these specifications—bandwidth, latency, concurrency, and clock domains—when selecting an architecture. Examples of interconnect architectures include:

- **Avalon Bus**: Used in Altera field-programmable gate arrays (FPGAs).
- **Wishbone Interconnect**: For open-source cores.
- **AXI4-Stream Interface**: For FPGA applications.

BUS: BASIC ARCHITECTURE

Role of Bus Architecture in Performance:

- The bus architecture, which connects components within a computer system, plays a crucial role in determining system performance.
- A poorly designed bus can slow down the transfer of instructions and data between the processor, memory, and peripherals, creating a bottleneck that affects the entire system's efficiency.

BUS: BASIC ARCHITECTURE

Popular Bus Standards: Over the years, many bus standards have been developed to improve connectivity and efficiency in systems, including:

- **VME Bus** and **Intel Multibus-II** for microprocessors and other systems.

- **ISA**, **EISA**, **PCI**, and **PCI Express** for personal computers. These standards are designed for connecting integrated circuits (ICs) on a printed circuit board (PCB) or across multiple PCBs in larger systems.

BUS: BASIC ARCHITECTURE

Limitations for SOC Technology: Traditional bus standards have limitations when applied to SOC (System-on-Chip) designs:

- **Limited Signal Count**: Due to constraints in pin count on IC packages or PCB connectors, which is costly to increase.
- **Performance Constraints**: The bus speed can be limited by factors such as high capacitive load on each bus signal, connector resistance, and electromagnetic interference from fast-switching signals.
- **Power and Space Savings**: On-chip buses in SOCs allow for smaller, less power-intensive drivers, saving both area and power.

Arbitration and Protocols

- **Arbitration**: Since the bus is essentially shared wires, only one unit can use it at a time.
- Arbitration is the process that ensures orderly bus access. The bus master is the unit that initiates communication, while slave units passively respond.
  - **Bus Master**: A master (like a processor in an SOC) initiates communication, controlling data flow and addressing specific slaves.
  - **Bus Slave**: Slaves (like memory or I/O devices) respond to master requests.
- **Centralized Arbitration Unit**: In simpler systems, a centralized arbitration unit manages bus ownership, following a specific protocol to grant access to requesting units.

Bus Protocol

A bus protocol is the set of rules governing communication on the bus. It specifies:

- **Data Type and Order**: The type of data sent and the order in which it’s sent.
- **Completion Signaling**: How the sender signals it has finished transmitting.
- **Data Compression**: Any data compression method used, if applicable.
- **Acknowledgment**: How the receiver acknowledges successful data receipt.
- **Arbitration and Error Checking**: Procedures for resolving bus contention (when multiple units request the bus simultaneously) and prioritizing requests, as well as the type of error-checking mechanisms.

Bus Bridge

- A **bus bridge** connects two separate buses, which may use different protocols or have different operational characteristics. It plays a critical role in managing communication across these buses in the following ways:
- **Protocol Conversion**: If two buses use different communication standards, the **bridge adapts** and **converts** data formats to enable smooth data exchange between them.
- **Segmentation for Traffic Management**: The bridge segments the buses, allowing each to manage its own traffic.
- This segmentation allows both buses to operate simultaneously, enhancing concurrency and enabling faster parallel data transactions.
- **Memory Buffering for Write Posting**: Often, a bridge includes memory buffers. When a master device on one bus sends data to a slave device on another bus, the data is first stored in the buffer.
- This lets the master continue with other tasks without waiting for the data to be fully written on the other bus, effectively increasing system speed and efficiency by avoiding delays.

Physical Bus Structure

- The **physical structure** of a bus—such as the number of wire paths and the time taken per cycle—affects how data is transmitted and how devices access the bus.

- **Arbitration**: Since multiple devices may need to use the bus simultaneously, **arbitration** is needed to decide which device gains control in each cycle.
- Arbitration mechanisms manage this by organizing a **request cycle** (where device requests are prioritized) and an **acknowledge cycle** (where the highest-priority request is granted access).
- **Simple vs. Complex Arbiters**:

Simple vs. Complex Arbiters

Simple Arbiters:

- These use basic request and acknowledgment cycles to decide bus access.

Complex Arbiters:

- These include extra control lines and logic, allowing each device to monitor bus status and priority, and to understand when the bus will become available without extra delay.

- By doing so, arbitration is completed without adding extra cycles, thus optimizing data transaction speed.

Types of buses

- **Unified vs. Split Buses and Single Transaction vs. Tenured Buses**:
- **Unified Bus**: In a unified bus, both the address (which indicates where data should go) and data (the actual information being transferred) share the same physical bus. In this setup:
  - First, the address is sent over the bus.
  - This is followed by one or more cycles where the actual data is transmitted.
- **Split Bus**: Here, there are two separate buses—one dedicated to address transmission and the other to data.

  This separation allows the address and data to be transmitted simultaneously, potentially speeding up transactions because they don't have to share the same line.

Types of buses

Single Transaction Bus:

- This type of bus is occupied for the entire duration of a transaction. Only one transaction can occupy the bus until it completes, meaning no other devices can use the bus during this time.

Tenured Bus:

- A tenured bus is only occupied during the necessary cycles for transmitting addresses or data.
- This bus allows transactions to be managed more flexibly by buffering messages.
- With separate address and data transactions, it can handle **multiple messages** or transactions in parallel.
- This separation reduces bus **occupancy time**, increasing efficiency by freeing the bus for other transactions sooner.

Bus Examples

- There are many possible bus designs with varying combinations of physical bus widths and arbitration protocols.
- The examples below consider some obvious possibilities. Suppose the bus has a transmission delay of one processor cycle, and the memory (or shared cache) has **a four - cycle access delay** after an initial address and requires an additional cycle for each sequential data access.

(a) Simple Bus, Single Transaction

(b) Bus with Arbitration Support

(c) Tenured Split Bus, 4 Bytes Wide

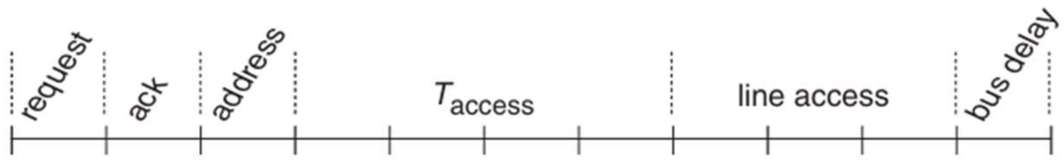
(d) Tenured Split Bus, 16 Bytes Wide, One-Cycle Bus Transaction Time

Bus Examples

(a) Simple Bus, Single Transaction

- **Structure**: A single transaction bus with a simple request/acknowledge (ack) arbitration mechanism.
- This bus has a physical width of 4 bytes, meaning it can transfer 4 bytes at a time.
- **Operation**: The bus transaction latency (total time) is 11 cycles:
  - The first word from memory is transmitted at the end of T_{access}.
  - The final word is transmitted at the end of the line access.
- **Arbitration**: The final bus cycle resets the arbiter (the logic managing device access to the bus).

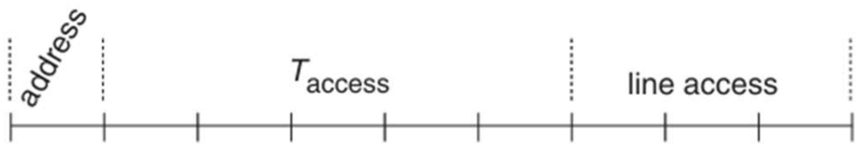
- T_{access} is the time required to access the first word from memory after the address is issued, and line access is the time required to access the remaining words.



Bus Examples

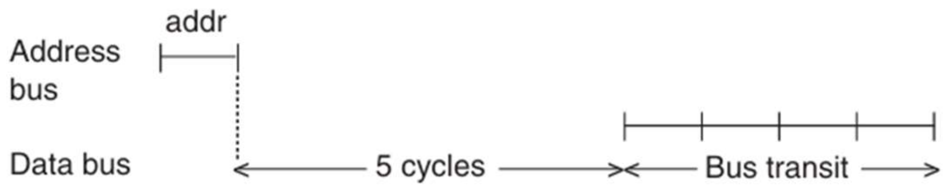
(b) Bus with Arbitration Support

- **Structure**: Similar to the simple bus in terms of width (4 bytes) and integrated address/data transmission, but it includes a more advanced arbiter.
- **Operation**:
  - An extra cycle (**five instead of four**) is needed to move the address from the bus receiver to memory.
  - The request and ack cycles overlap with bus processing, eliminating the need for an additional reset cycle at the end.
  - Total latency is reduced to 10 cycles.



Bus Examples

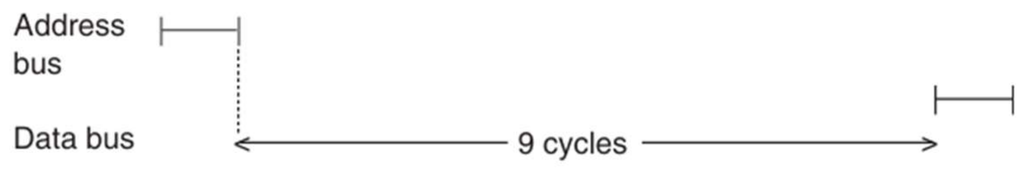
- **Tenured Split Bus, 4 Bytes Wide**
- **Structure**: This is a tenured bus, meaning it only occupies the bus during necessary cycles. It has separate address and data buses (split bus) and a physical width of 4 bytes.
- **Operation**:
  - The requested line is fetched into a buffer for five cycles, then transmitted in four cycles.
  - Total latency remains at 10 cycles, but the bus is only occupied for four of these cycles, freeing up more time for other transactions.
  - The address bus is occupied for just one cycle out of the 10, enhancing overall communication performance.



Bus Examples

(d) Tenured Split Bus, 16 Bytes Wide, One-Cycle Bus Transaction Time

- **Structure**: A tenured split bus with a physical width of 16 bytes, allowing it to handle larger data segments per cycle.
- **Operation**:
  - The 16-byte cache line is fetched by memory in multiple cycles, but once fetched, it is transmitted across the bus in just one cycle.
  - Although transaction latency remains at 10 cycles, the address and data buses are each used for only one cycle, leaving additional cycles available for other tasks.
  - An optional additional cycle allows the bus to re-access, though it may not be necessary.



Bus Examples

- In both tenured cases, **bus bandwidth exceeds memory bandwidth**. For instance, in case (d), while the bus is busy for only one cycle, the memory is busy for seven cycles.
- This disparity creates a **memory bottleneck**, meaning system performance is limited by how fast memory can provide data, not by the bus capacity.

SOC STANDARD BUSES

- **AMBA** and **CoreConnect** are popular bus standards used in SoCs.
- The **AMBA (Advanced Microcontroller Bus Architecture)** developed by ARM, which is widely used in SoCs with ARM processors
- The **CoreConnect** bus developed by IBM, which has been used in Xilinx's Virtex FPGA platforms.
- These standards define the communication protocols between components like processors, memory, and peripherals within a chip.

AMBA (Advanced Microcontroller Bus Architecture)

AMBA, created in 1997, originated from ARM processors and is now widely used in various embedded systems.

It structures the internal SoC communication through a hierarchical bus architecture with two main bus types:

- **AHB (Advanced High-Performance Bus)** and
- **APB (Advanced Peripheral Bus)**.

AMBA (Advanced Microcontroller Bus Architecture)

- **AHB** (Advanced High-Performance Bus) is the high-speed bus in the AMBA architecture. It connects high-performance components like processors, memory, and DMA (Direct Memory Access) controllers.
- **High speed and bandwidth**: Uses separate lines for address, read, and write operations, supporting wide data widths (starting from 32 bits and scalable to 1024 bits).
- **Concurrent operations**: Multiple masters and slaves can operate simultaneously on the bus.
- **Burst mode**: Allows efficient, high-speed data transfers.
- **Split transactions**: Enables ongoing transactions without blocking others, improving system efficiency.
- **Single clock edge synchronization**: Simplifies system design.

AMBA (Advanced Microcontroller Bus Architecture)

- **APB (**Advanced Peripheral Bus) is designed for low-performance peripherals, optimized for simplicity and minimal power usage.

- APB connects slower peripheral components and reduces the overall complexity of the system interface.

- **ASB (Advanced System Bus)**: A predecessor to AHB, ASB was designed for simpler systems with less demanding performance needs.

Goals of the AMBA Design

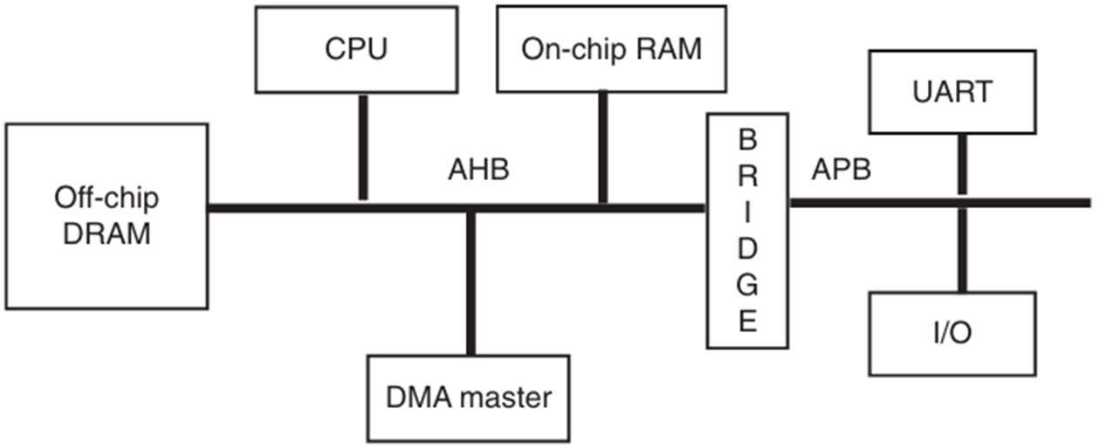
- **Modular Design and Reusability**: Encourages partitioned, reusable components and prevents inefficient designs by guiding designers with a modular, flexible bus structure.

- **Standardized Interface and Clocking Protocol**: AMBA's structured interface and clocking facilitate high performance and easy integration.

- **Low Power Support**: AMBA's hierarchy is energy-efficient, especially in the APB, aligning well with ARM's low-power processors.

- **On-Chip Test Access**: Offers an optional test feature that utilizes the bus for testing modules on the chip.

A typical AMBA bus - based system



A typical system using the AMBA bus architecture.

The AHB forms the system backbone bus on which the ARM processor, the high - bandwidth memory interface and random - access memory (RAM), and the DMA devices reside.

The interface between the AHB bus and the slower APB bus is through **a bus bridge module**.

AHB Bus Implementation

- The AHB protocol enables a **multimaster system** where multiple master devices (e.g., processors or DMAs) can control the bus.

- Instead of **tristate buffers** (often used in PCB systems), AHB uses a central **multiplexer** for signal routing, enhancing performance and lowering power consumption.
- The central multiplexer approach simplifies the design by removing the need for each component to have its **own tristate buffer.** Instead, all address and control signals from masters are routed through the multiplexer.

- A **central arbiter** decides which master's address and control signals are sent to the slaves, while a **decoder** selects the appropriate read data and response signals from the relevant slave during a transaction.
- This approach improves efficiency and reduces power usage compared to traditional tristate setups.

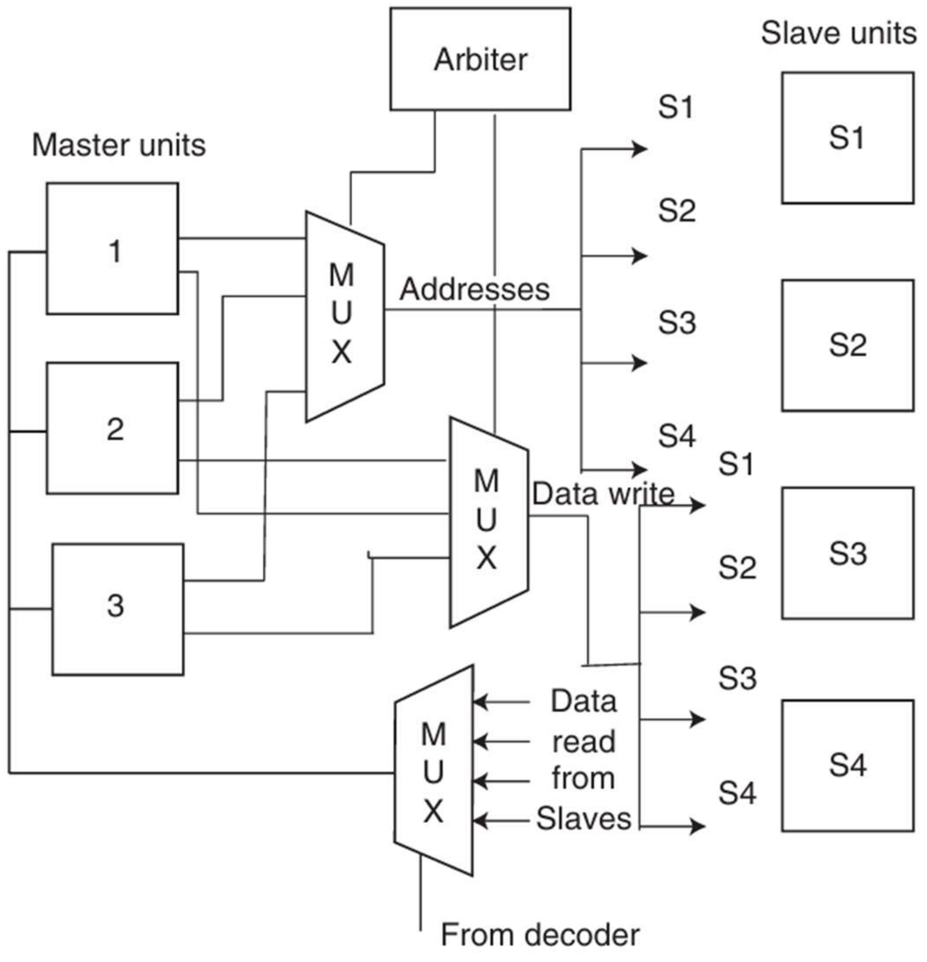
AHB Bus Implementation

- **Bus Master Obtains Access to the Bus.**
  - This process begins with the master asserting a request signal to the arbiter.
  - If more than one master simultaneously requests the control of the bus, the arbiter determines which of the requesting masters will be granted the use of the bus.

Bus Master Initiates Transfer.

- A granted bus master drives the address and control signals with the address, direction, and width of the transfer.
- A write data bus operation moves data from the master to a slave, while a read data bus operation moves data from a slave to the master.

Multiplexor (MUX) interconnection for a three masters/four slaves system


Arbiter
Slave units
Master units
1
2
3
M U X
Addresses
M U X
Data write
M U X
Data read from Slaves
From decoder
S1
S2
S3
S4
S1
S2
S3
S4
S1
S2
S3
S4