BROWN

Low Power VLSI System Design

Lecture 10: Low Power Memory Design

Prof. R. Iris Bahar
EN2912
October 11, 2017

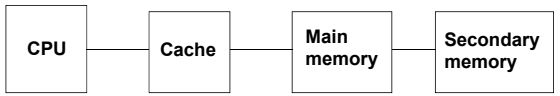
BROWN

SRAMs to Memory

- Last lecture focused on the SRAM cell and the 2D or 3D memory architecture built from these cells
- Several techniques can be used to reduce power
  - Reduce the bit line voltage swing
  - Isolate memory cells from the bit lines after sensing
  - Use pulsed word lines
  - Block addressing
  - Sleep transistors
- ***How to these techniques fit into the "bigger picture" of memory design?***

BROWN

Memory Hierarchy



Registers

Levels L1, L2, … (hardware implementation, SRAMs)

Virtual memory

(software implementation)

Hierarchical organization makes memory look large and fast

Memory access is checked in fast caches first before resorting to slow memory at lower levels.

BROWN

Locality

- Locality is a principle that makes having a memory hierarchy a good idea
- If an item is referenced,
  - **temporal locality:** it will tend to be referenced again soon
  - **spatial locality:** nearby items will tend to be referenced soon.
- Why does code have locality?
  - loops
  - instructions accessed sequentially
  - arrays, records

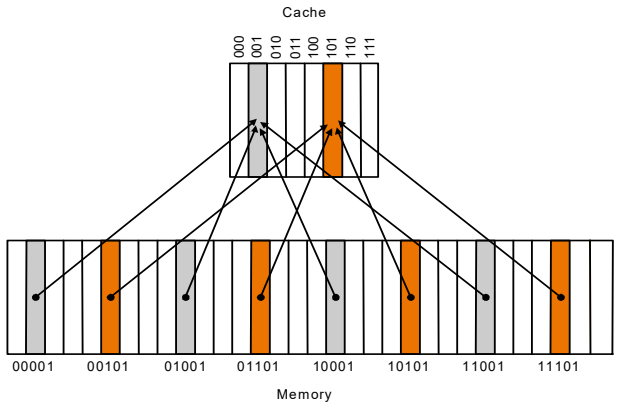
BROWN

Direct Mapped Cache

- Simple approach: Direct mapped
  - block size is one word
  - every main memory location can be mapped to exactly one cache location
  - lots of words in the main memory share a single location in the cache
- How is the address composed for the cache?
  - cache address is identical with lower bits in the main memory address
  - tag (higher address bits) differentiates between competing main memory words
- We are taking advantage of temporal locality.

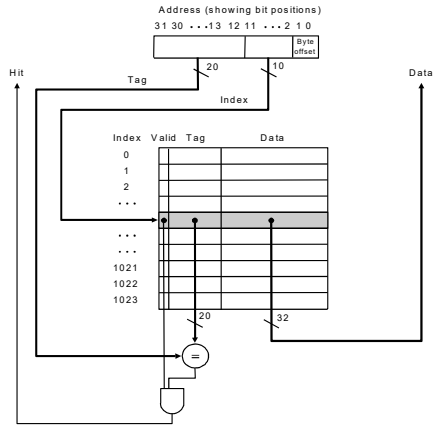
BROWN

Direct Mapped Cache Example



BROWN

Cache Access



BROWN

Flexible Placement of Blocks

- Direct mapped cache
  - a memory block can go exactly in one place in the cache
  - use the tag to identify the referenced word
  - easy to implement, but rigid placement can cause high miss rate
- Fully associative cache
  - a memory block can be placed in *any* location in the cache
  - search all entries in the cache in parallel
  - requires a comparator associated with each cache entry
- Set-associative cache
  - a memory block can be placed in a fixed number of locations
  - n locations: n-way set-associative cache
  - a block is mapped to any of n locations in a set
  - Requires searching all locations of the set

BROWN

Types of Cache Misses

- Compulsory misses: happens the first time a memory word is accessed
  - the misses for an infinite cache
- Capacity misses: happens because the program touched many other words before re-touching the same word
  - the misses for a fully-associative cache
- Conflict misses: happens because two words map to the same location in the cache
  - the misses generated while moving from a fully-associative to a direct-mapped cache

BROWN

Locating a Block

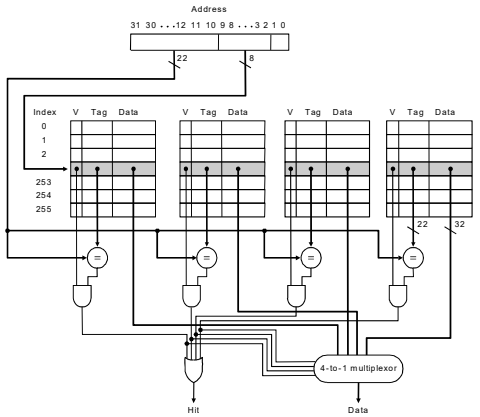
- Address portions

tag	index	block offset

  - Index selects the set.
  - Tag chooses the the block by comparison.
  - Block offset is the address of the data within the block.
- The costs of an associative cache
  - comparators and multiplexers
  - time for comparison and selection

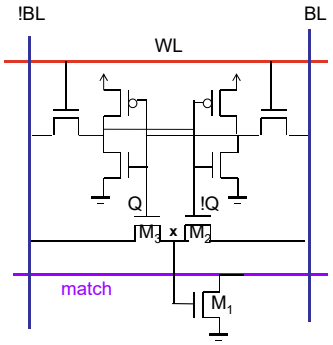
BROWN

4-Way Set-Associative Cache



BROWN

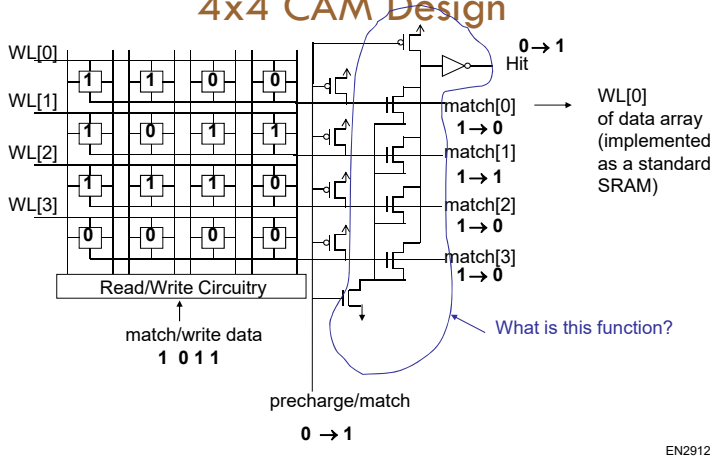
9-T CAM Cell



- Writes progress as in a standard SRAM cell
- Compares the stored data (Q and !Q) to the bit line data
  - Precharged match line tied to all cells in a row
  - If Q and BL match, **x** is discharged through M_2 or M_3 and thus M_1 is OFF keeping the match line high
  - Else if Q and BL don't match, **x** is charged to $V_{DD} - V_T$ and the match line discharges

BROWN

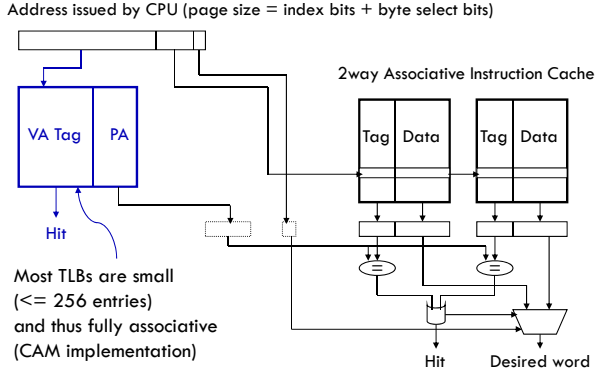
4x4 CAM Design



BROWN

Content addressable memories

- Memories addressed by their content. Typical applications include cache tags, translation lookaside buffers (TLBs), and instruction queues.



BROWN

Issues for Set-Associative Caches

- Set-associative caches have a significant HW overhead
- Tag lookup is more complicated
- The CPU would like the data as soon as possible
  - For direct mapped caches, there is only one choice of which data to send
  - What about a set-associative cache?
- Can you send the data to the CPU before the tag has been checked?
- **What about power concerns?**

BROWN

Saving Power by Understanding Memory Needs of Application

- Depending on the applications, cache misses may be predominantly capacity, conflicts, or compulsory
- Depending on the application, the working set may be large or small
- ***IDEA***: Allocate only enough caching resources as application demands
  - Unused portions are gated and powered off
  - Reconfigure associativity to meet power/performance needs
- Can save both dynamic and static (leakage) power

➡ How do you know what the resource needs of your application are?

Saving Power by Understanding Data Access Behavior

- General nature of cache line usage:
  - cache lines typically have a flurry of frequent use when first brought into the cache, and then have a period of "dead time" before being evicted.
- What's the cost of holding these unused lines in the cache?
- ***IDEA:*** Turn off cache lines that are no longer useful
  - Shut off power supply to cache line → save leakage energy
- How do we know when a cache line is no longer useful?
  - Keep track of the last time it was accessed
- What factors do we have to consider to implement this scheme?
  - What happens to data stored in the cache line?
  - What if our "usefulness estimator" is wrong?

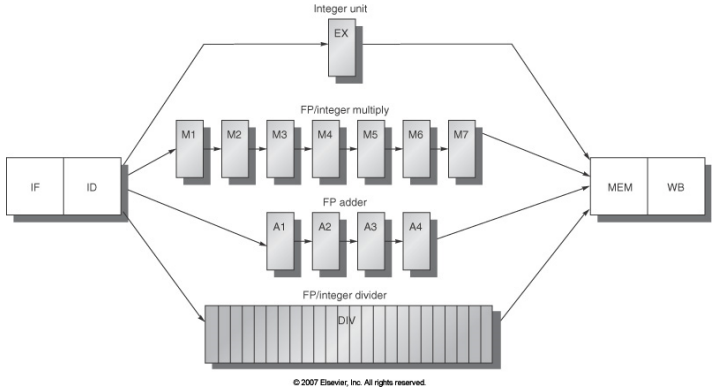
Saving Power by Understanding Cache Locality Behavior

- General nature of accesses within a cache:
  - during a fixed period of time the activity in a cache is only centered on a small subset of the lines.
- Not all lines need to be accessed right away
- ***IDEA:*** Keep the "hot lines" in active mode and the "cold lines" in a standby state
  - Standby state does not erase current data
- What is the cost of re-accessing cold lines?
- How does this compare in terms of performance and power to gating off Vdd?

Saving Power at the Architecture Level

- The cache is a major source of power dissipation for a microprocessor
- But it's not the only source
- What about other resource structures within a microprocess?

Microprocessor Pipeline



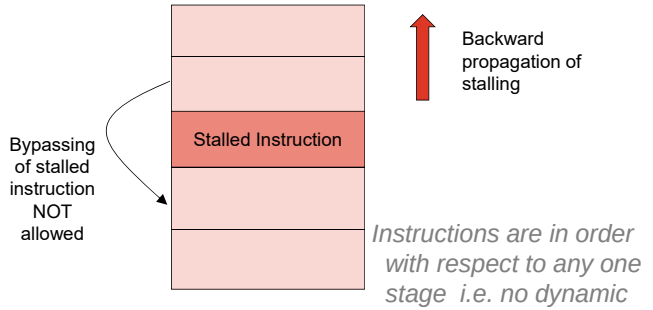
BROWN

Instruction Level Parallelism

- **Instruction-level parallelism:** overlap among instructions: Due to pipelining or multiple instruction execution
- *What determines the degree of ILP?*
  - dependences: property of the program
  - hazards: property of the pipeline
- Upper Bound on single Pipeline Throughput
  Limited by IPC = 1
- Inefficient Unification Into Single Pipeline
  Long latency for each instruction
- Performance Lost Due to Rigid Pipeline
  Unnecessary stalls

BROWN

Stalls in an Inorder Pipeline



Instructions are in order with respect to any one stage i.e. no dynamic reordering

BROWN

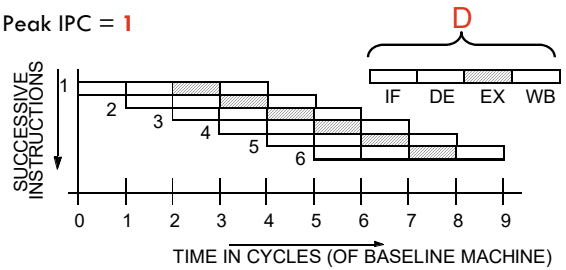
Architectures for Instruction-Level Parallelism

Scalar Pipeline (baseline)

Pipeline Depth = **D**

Operation Latency = **1**

Peak IPC = **1**



BROWN

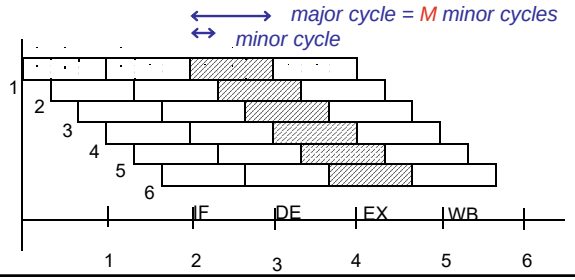
Superpipelined Machine

Superpipelined Execution

Instruction Parallelism = *DxM*

Operation Latency = *M minor cycles*

Peak IPC = *1 per minor cycle* *(M per baseline cycle)*



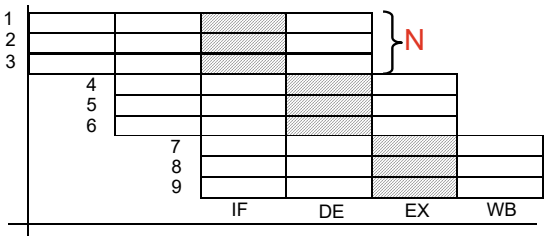
Superscalar Machines

Superscalar (Pipelined) Execution

Instruction Parallelism = *DxN*

Operation Latency = *1 baseline cycles*

Peak IPC = *N per baseline cycle*

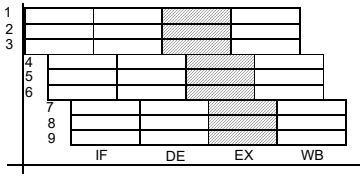


Limitations of Inorder Pipelines

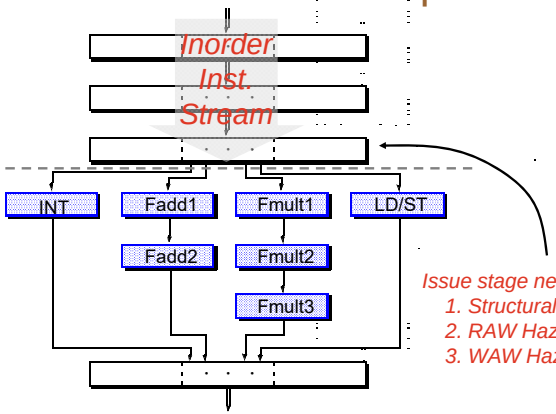
- Clocks per Instruction (CPI) of inorder pipelines degrades very sharply if the machine parallelism is increased beyond a certain point, *i.e. when NxM approaches average distance between dependent instructions*
- Forwarding is no longer effective

⇒ must stall more often

» *Pipeline may never be full due to frequent dependency stalls!!*



In-order Issue into Diversified Pipelines

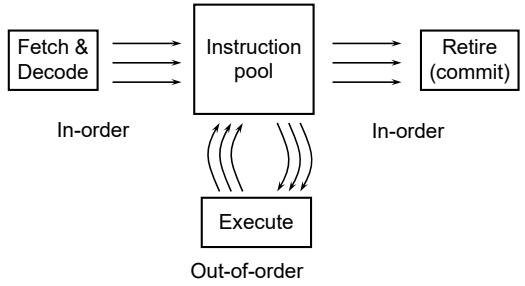


Issue stage needs to check:

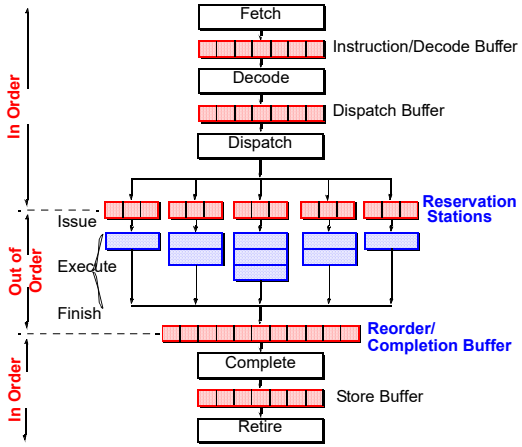
1. *Structural Dependence*
2. *RAW Hazard*
3. *WAW Hazard*

Out of Order Execution - General Scheme

Many high performance processors use out of order execution. Most of them are doing the fetching and the retirement *IN ORDER*, but it executes in *OUT OF ORDER*



A Modern(ish) Superscalar Processor



Out of Order Execution

- Execute instr. based on "data flow" rather than program order.
- Basic idea:
  - The fetch is done fast enough to "fill-out" a window of instructions.
  - Of all instructions in the window, look for those ready to execute:
    - All the data the instructions are dependent on, are ready
    - Resources are available.
- As soon as the instruction is executed it needs to signal to all the instructions which are depend on it that the input is ready.
  - Triggers "wake-up" and "instruction select" for next cycle
- Advantages:
  - Help exploit Instruction Level Parallelism (ILP)
  - Help cover latencies (e.g., cache miss, divide)

The Cost of Speculation

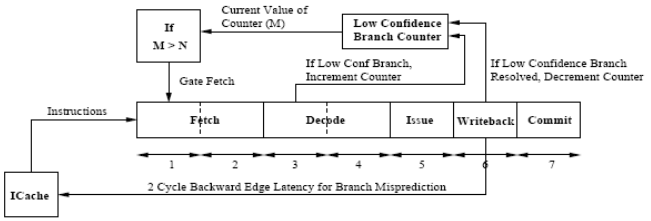
- How does the processor find enough "ready" instructions?
  - Look beyond branch boundaries
- Modern processor have fairly sophisticated branch prediction mechanisms (in HW and SW)
- Front end fills instruction window with instructions down path of predicted branch
- What happens if the branch prediction is wrong?
- What is the cost in terms of performance and power of keeping these "wrong path" instructions in the instruction window?

Reducing Mis-speculated Instructions to Save Power

- Executing wrong path instruction is a waste of power
  - Does nothing to improve effective IPC either
- ***IDEA:*** if branch prediction becomes too speculative, don't bother continuing to fetch instruction past branches
  - Fetch unit stops reading new instructions from the cache
  - Instruction window does not take new instructions
  - Instruction execution rate may slow, but only until predicted branches have been resolved
- How do we know if an instruction flow has become "too speculative"?
  - What do we need to monitor?

BROWN

Pipeline Gating



- Low-confidence branch counter records # of unresolved branches that reported as low-confidence.
- The processor ceases instruction fetch if there are more than *N* unresolved low-confident branches in the pipeline.