

Basics of Computer Architecture and Operating Systems

Mainak Chaudhuri

Indian Institute of Technology Kanpur

Agenda

- Basics of computer architecture
 - Basics of the basics
 - Instruction set architecture (ISA)
 - Processor design
 - Caches and virtual memory
 - Communicating with environment
 - Performance measurement
 - Performance optimization
 - Multi-core processors
- Basics of operating systems

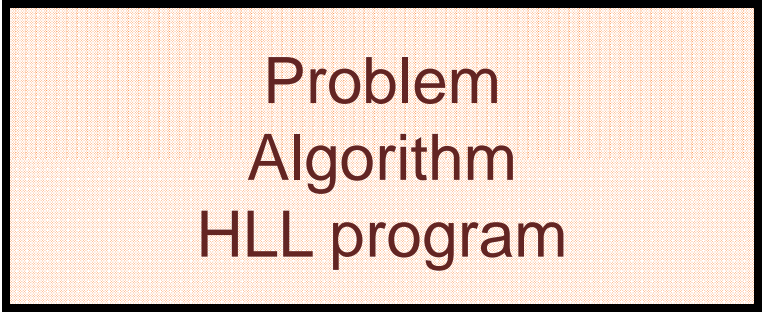
Agenda

- Basics of computer architecture
 - Basics of the basics
 - Instruction set architecture (ISA)
 - Processor design
 - Caches and virtual memory
 - Communicating with environment
 - Performance measurement
 - Performance optimization
 - Multi-core processors
- Basics of operating systems

The computing stack

The computing stack

Software



Problem
Algorithm
HLL program

The computing stack

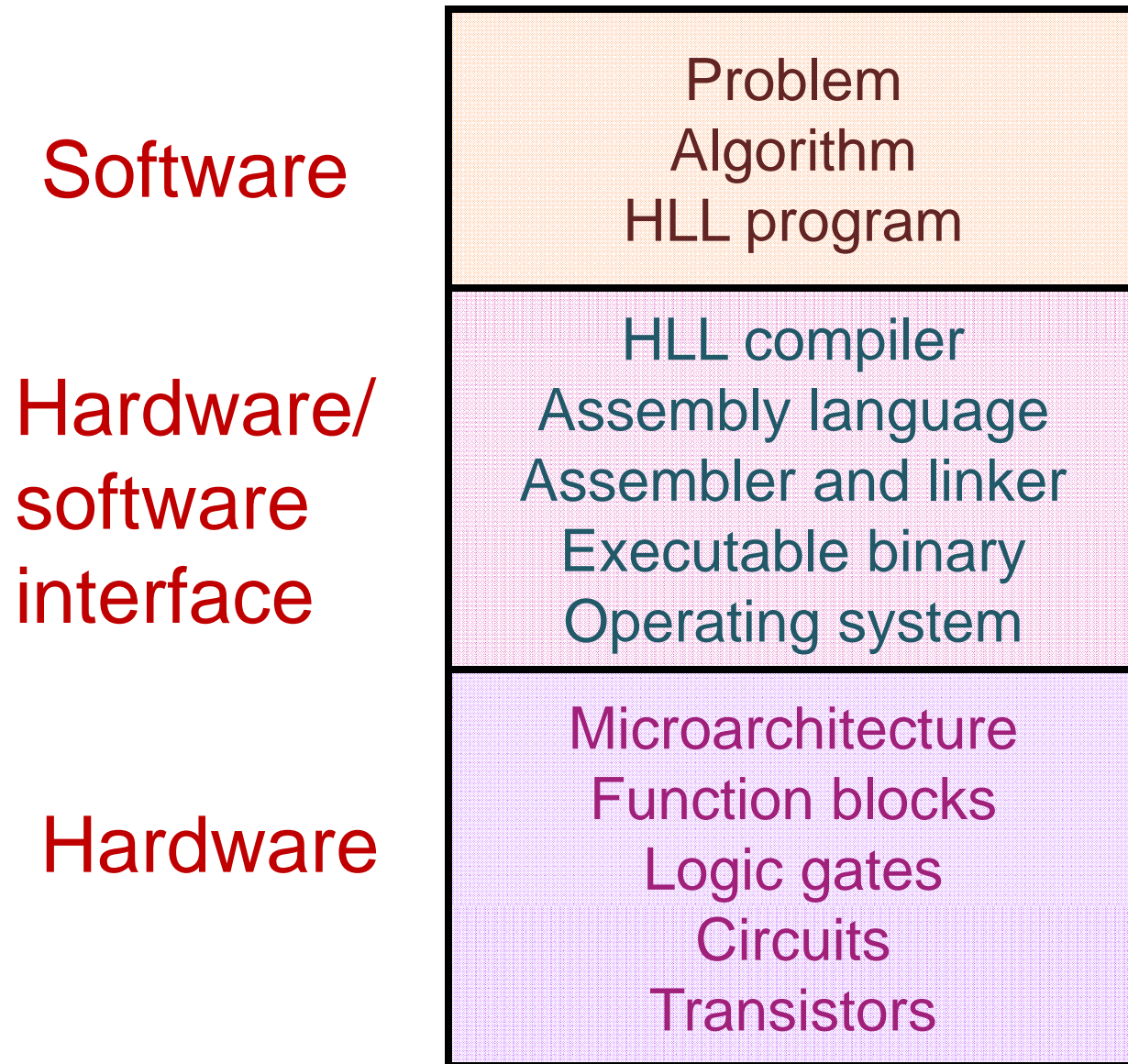
Software

Problem
Algorithm
HLL program

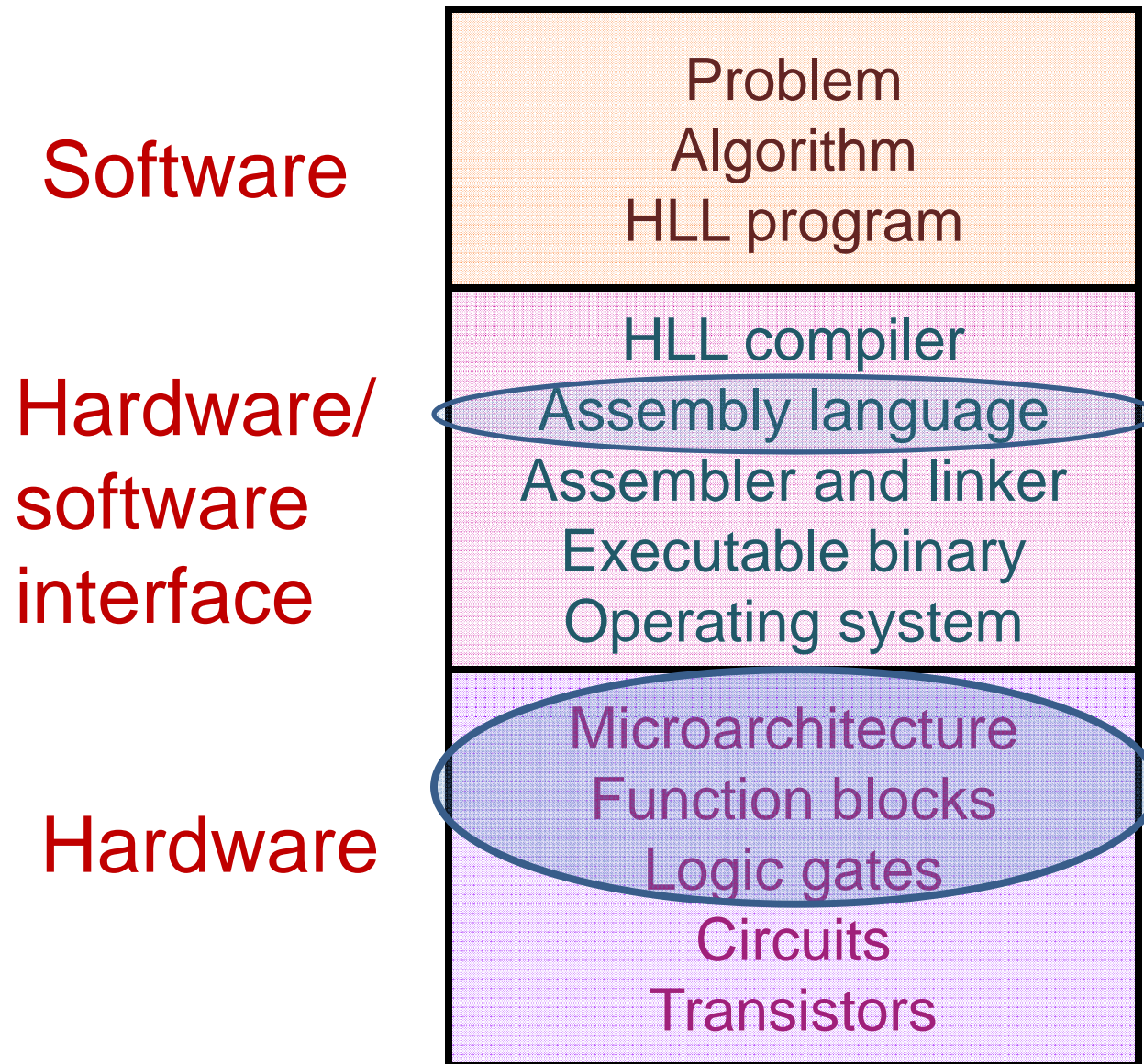
Hardware/
software
interface

HLL compiler
Assembly language
Assembler and linker
Executable binary
Operating system

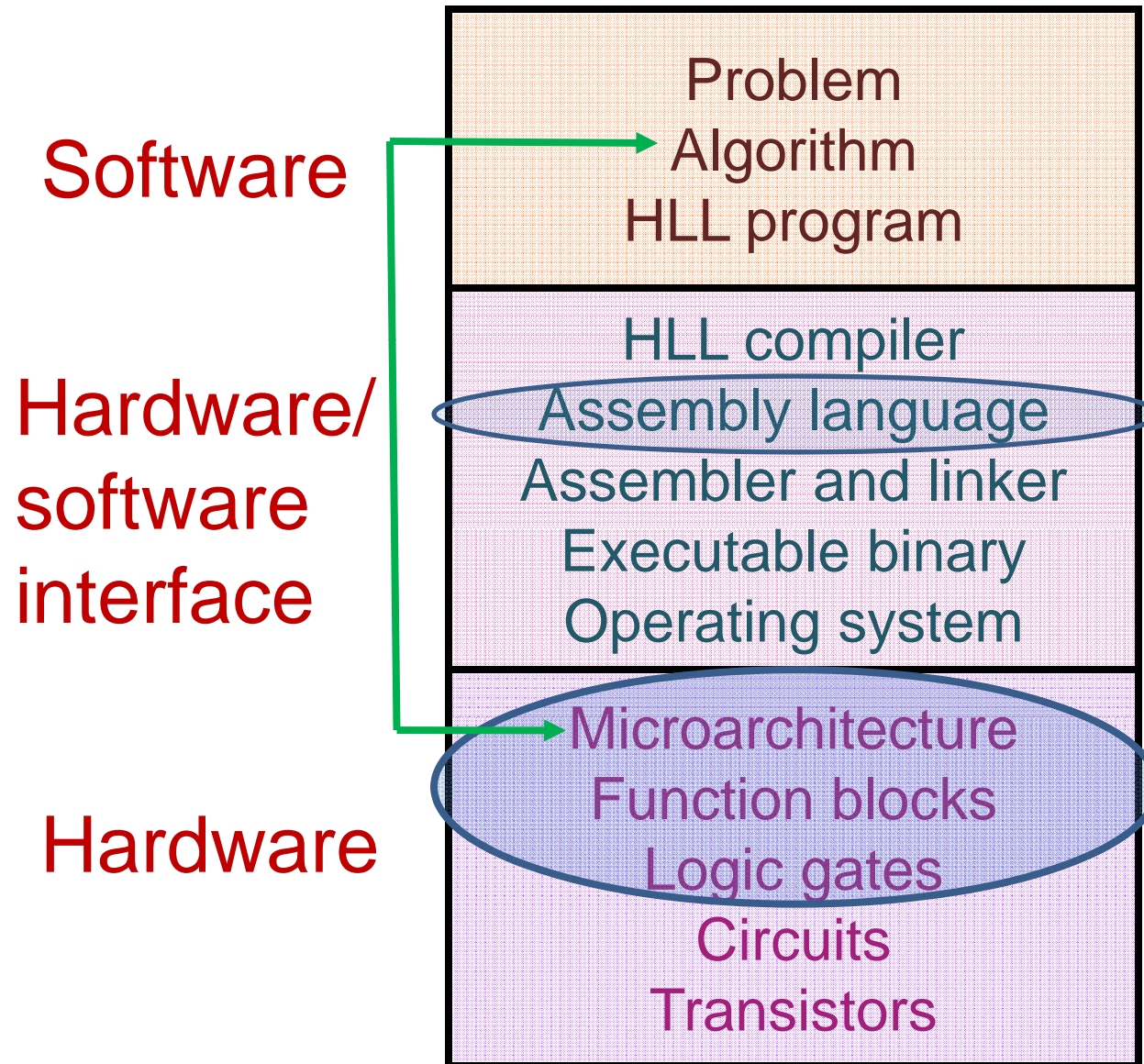
The computing stack



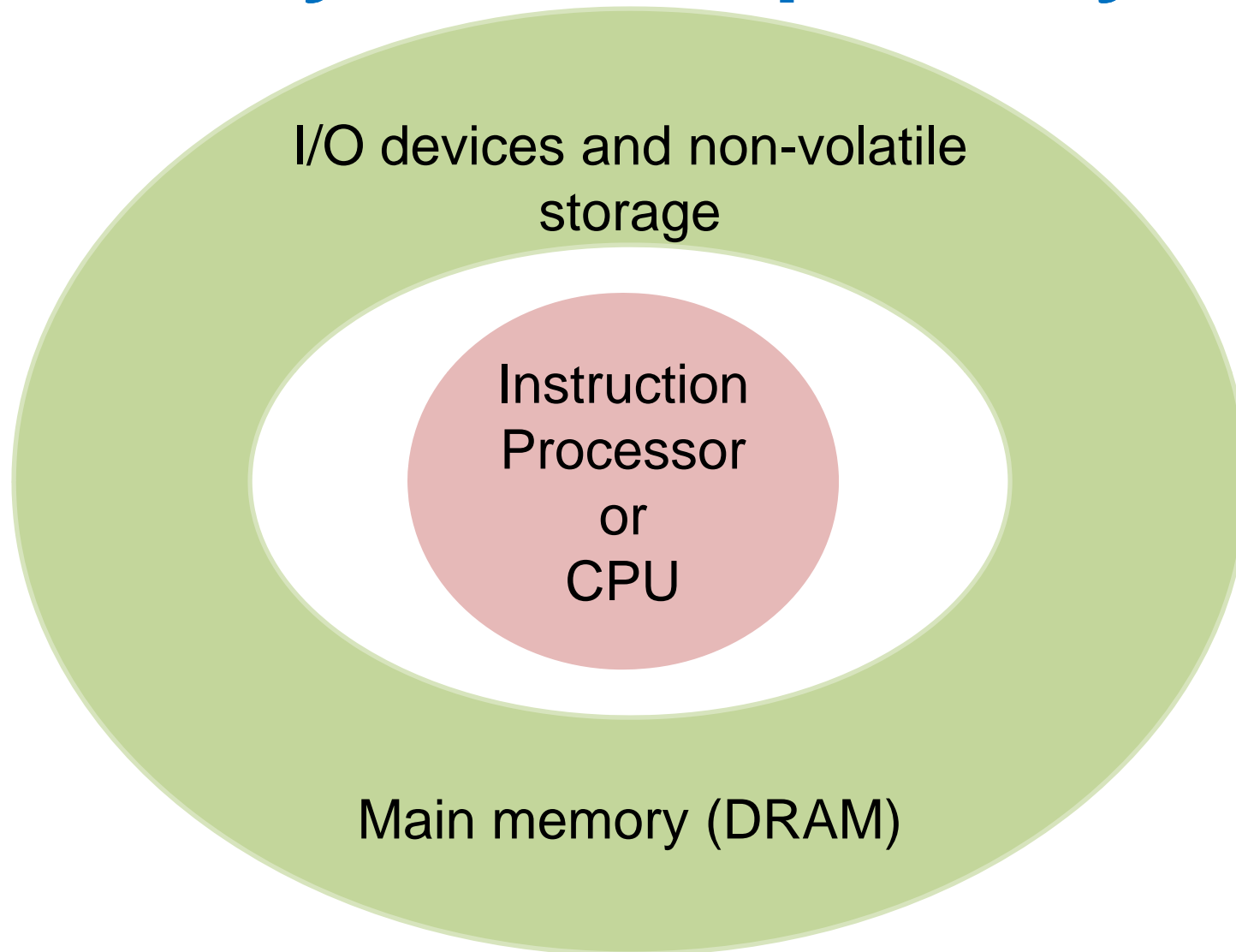
The computing stack



The computing stack



Anatomy of a computer system



Anatomy of a computer system

- The central processing unit (CPU)
 - Also known as microprocessor
 - Dictates how a task will be done, but cannot do anything on its own
 - Needs to be told what to do next in the form of a stream of “instructions”
 - These instructions are generated from a program that represents an algorithm for accomplishing the task
 - Can store intermediate/final results of a computation in main memory
 - Dynamic random access memory (DRAM); volatile
 - Can store information on persistent non-volatile storage media e.g., magnetic hard disk

Anatomy of a computer system

- Peripheral I/O devices
 - Plug-ins to the CPU for communicating with the world
 - Display (CRT, LCD, touchscreen)
 - Keyboard
 - Mouse
 - DVD reader/writer
 - Speaker
 - Microphone
 - Camera
 - Wireless communication
 - Wired Ethernet communication

Anatomy of a computer system

- How does an instruction execute inside the processor?
 - Every entity residing inside a computer has an address
 - An instruction also has an address
 - The processor maintains the address of the next instruction in a register called program counter
 - The instruction is fetched from main memory and placed in an instruction register
 - The instruction is decoded to generate the control signals for executing the instruction
 - Send to adder if this is an addition instruction

Anatomy of a computer system

- How does an instruction execute inside the processor?
 - Most instructions require source operands for execution
 - $a+b$
 - After decoding the instruction, the operands are fetched
 - Operand addresses are typically encoded in the instruction or could be implicit
 - These addresses are known after decoding the instruction
 - The instruction can now execute
 - Operands are sent to the adder

Anatomy of a computer system

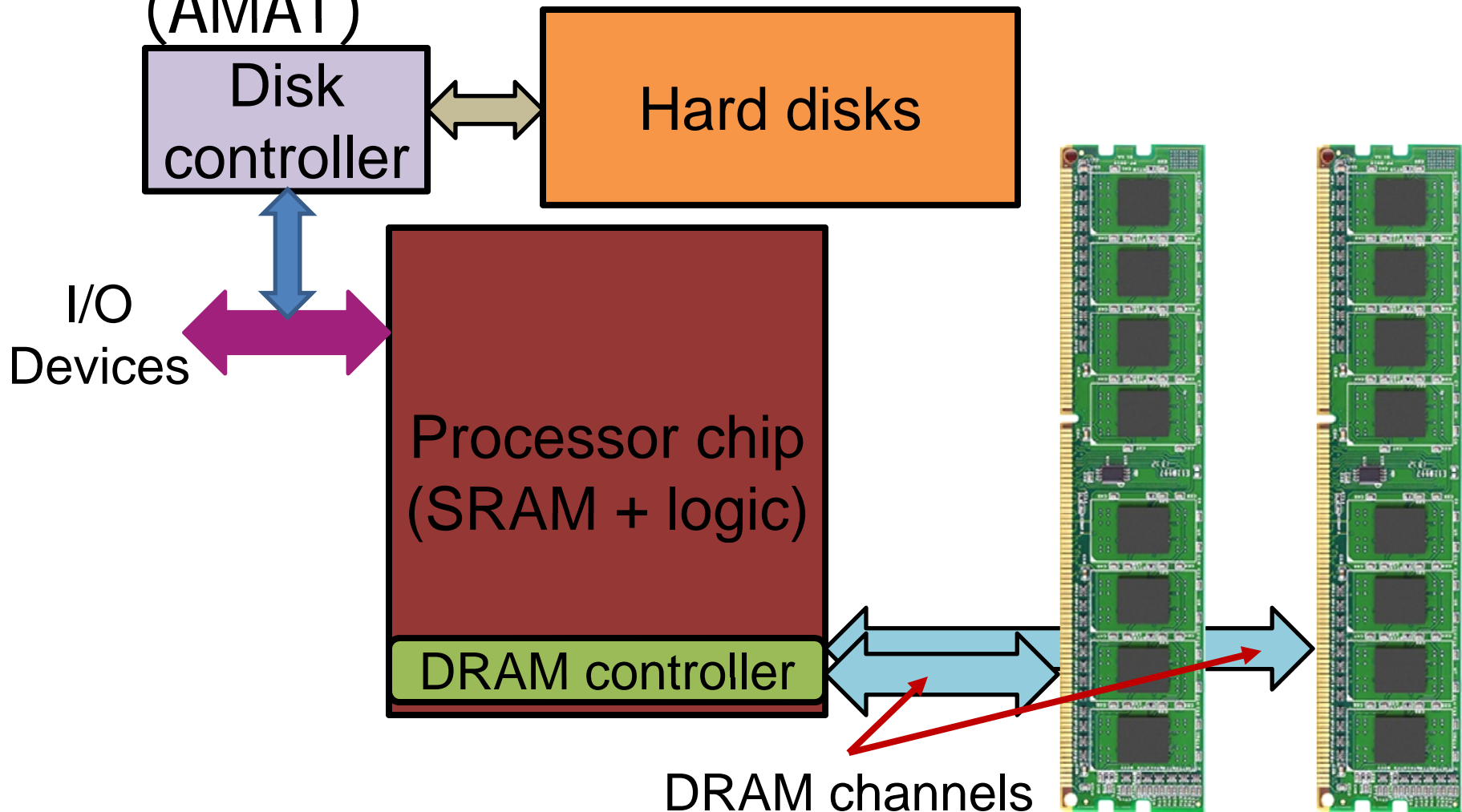
- How does an instruction execute inside the processor?
 - Most instructions generate a result
 - $c = a + b$
 - The address of the result (or destination) operand is typically encoded in the instruction or implicit
 - This address is known after decoding the instruction
 - The result is stored in the destination operand location

Anatomy of a computer system

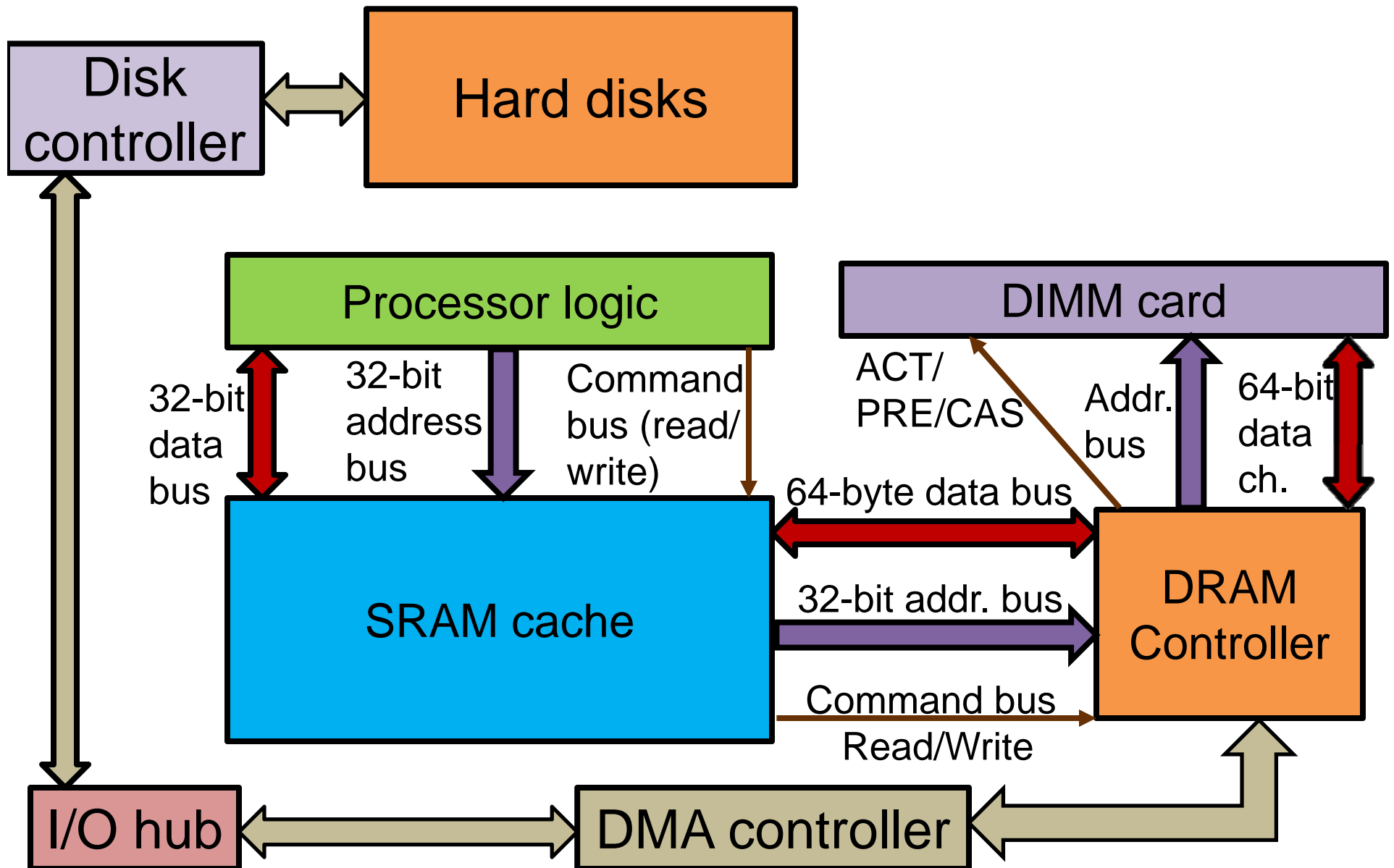
- How does an instruction execute inside the processor?
 - The execution of an instruction requires the appropriate control and data paths to be activated
- Data path is usually slow because main memory is much slower than the processor
 - Commonly used optimizations for speeding up the data path:
 - Reasonably large set of general-purpose registers inside the processor
 - Fast memory (known as cache) inside the processor

Anatomy of a computer system

- The SRAM cache on processor chip helps improve the average memory access time (AMAT)



Anatomy of a computer system



Agenda

- Basics of computer architecture
 - Basics of the basics
 - Instruction set architecture (ISA)
 - Processor design
 - Caches and virtual memory
 - Communicating with environment
 - Performance measurement
 - Performance optimization
 - Multi-core processors
- Basics of operating systems

What is ISA?

- Operations of a computer are done through instructions
 - One instruction does one operation
- Set of instructions supported by a computer is the instruction set of the computer
- Organization of the computer defined by the instruction set is the instruction set architecture (ISA) of the computer
- The guiding principle in designing ISA is that the implementation of the ISA should be simple
- The programming language that uses instructions of a computer is called an assembly language

Computer operations

- Every computer must be able to do arithmetic operations
 - Add, subtract, multiply, divide
 - The variables operated on and the result variables are called operands
 - The operands used in an operation are called source operands or read operands
 - The operands used for storing the result of an operation are called destination operands or write operands
 - $c = a + b$: a and b are source or read operands, c is the destination or write operand
 - Most computers allow at most two sources and one destination per instruction

Computer operations

- Consider the MIPS instruction “add a, b, c”
 - MIPS is the name of a processor family
 - We will understand what MIPS stands for later in the course
 - “add” is the name of the instruction which defines the operation to be done when this instruction is encountered by the computer
 - Among the three operands (a, b, c), the one right after “add” is the destination operand
 - Usually the destination operand is mentioned right after the operation name
 - Other two operands (b, c) are source operands
 - Add b and c, put the result in a

Operands

- So far we have assumed that instruction operands are same as HLL program variables
 - This assumption is incorrect
 - Each hardware operation reads its source operands from some storage and writes its destination operand to some storage
 - This storage can be either a register from the register file inside the processor or a memory location
 - By memory we usually mean RAM of the computer (DRAM)
 - Anything that is accessed from DRAM is also copied into SRAM cache inside the processor replacing something if the SRAM cache is already full
 - So, at any point in time, the SRAM cache stores a subset of the DRAM contents
 - The data in a memory location can be found in SRAM cache or in DRAM (DRAM is accessed only if SRAM cache does not have the data)

Operands

- Translating an HLL program to an assembly language program involves two basic steps
 - Mapping HLL operators to computer instructions
 - Exact instruction names differ from one computer to another, but the operations done by the instructions are largely same
 - Mapping HLL operands (or variables) to computer instruction operands
 - Requires deciding which variable is stored where and when
 - Every variable gets a fixed memory location
 - However, this forces every computer instruction to operate on memory operands only and obviates the need for a register file
 - But accessing a register is faster than accessing memory (faster than even SRAM cache)

Operands

- Mapping variables to operands
 - Fast access to registers motivates mapping variables to register operands
 - Simplicity of design also motivates restricting the operands of certain instructions to registers only
 - Consistent latency of these operations (recall that accessing memory can have variable latency)
 - In MIPS, many instructions (including arithmetic) allow only register operands
 - In x86, both register and memory operands are allowed
 - Makes instruction latency variable
 - Width of register file dictates the width of a register operand (usually 32 or 64 bits)
 - Dictates the width of datapath used in computation

Operands

- Mapping variables to operands
 - Since taller register file is slower, the number of registers must be restricted
 - Since each variable is given a unique memory location to store its value, it must be copied to a register if an instruction wants to use it as a register operand
 - In MIPS, arithmetic and many other instructions only allow register operands
 - Not all variables of a program can be allocated in registers at the same time due to restricted number of registers
 - Variables are allocated in and de-allocated from registers as the program progresses (filled from and spilled into memory)

Operands

- Mapping variables to operands
 - Register allocation of variables is the compiler's responsibility
 - Goal is to minimize the number of fills and spills because memory access is inefficient
 - Example: assume four registers r1, r2, r3, r4

Consider the C statements:

`a=b + c; // Allocate a, b, c to r1, r2, r3`

`d=e + f; // Allocate d to r4, how to allocate e and f?`

`a=a + d;`

`b=a + e;`

Operands

- Mapping variables to operands

Consider the C statements:

`a=b + c; // Allocate a, b, c to r1, r2, r3`

`d=e + f; // Allocate d to r4, how to allocate e and f?`

`a=a + d;`

`b=a + e;`

- Assembly language translation (not exactly MIPS)

`load r2, addr_b #fill b`

`load r3, addr_c #fill c`

`add r1, r2, r3 #a = b + c`

`load r2, addr_e #fill e`

`load r3, addr_f #fill f`

`add r4, r2, r3 #d = e + f`

`add r1, r1, r4 #a= a + d`

`add r3, r1, r2 #b=a + e`

`store r1, addr_a #spill a (note changed syntax)`

`store r3, addr_b #spill b`

`store r4, addr_d #spill d`

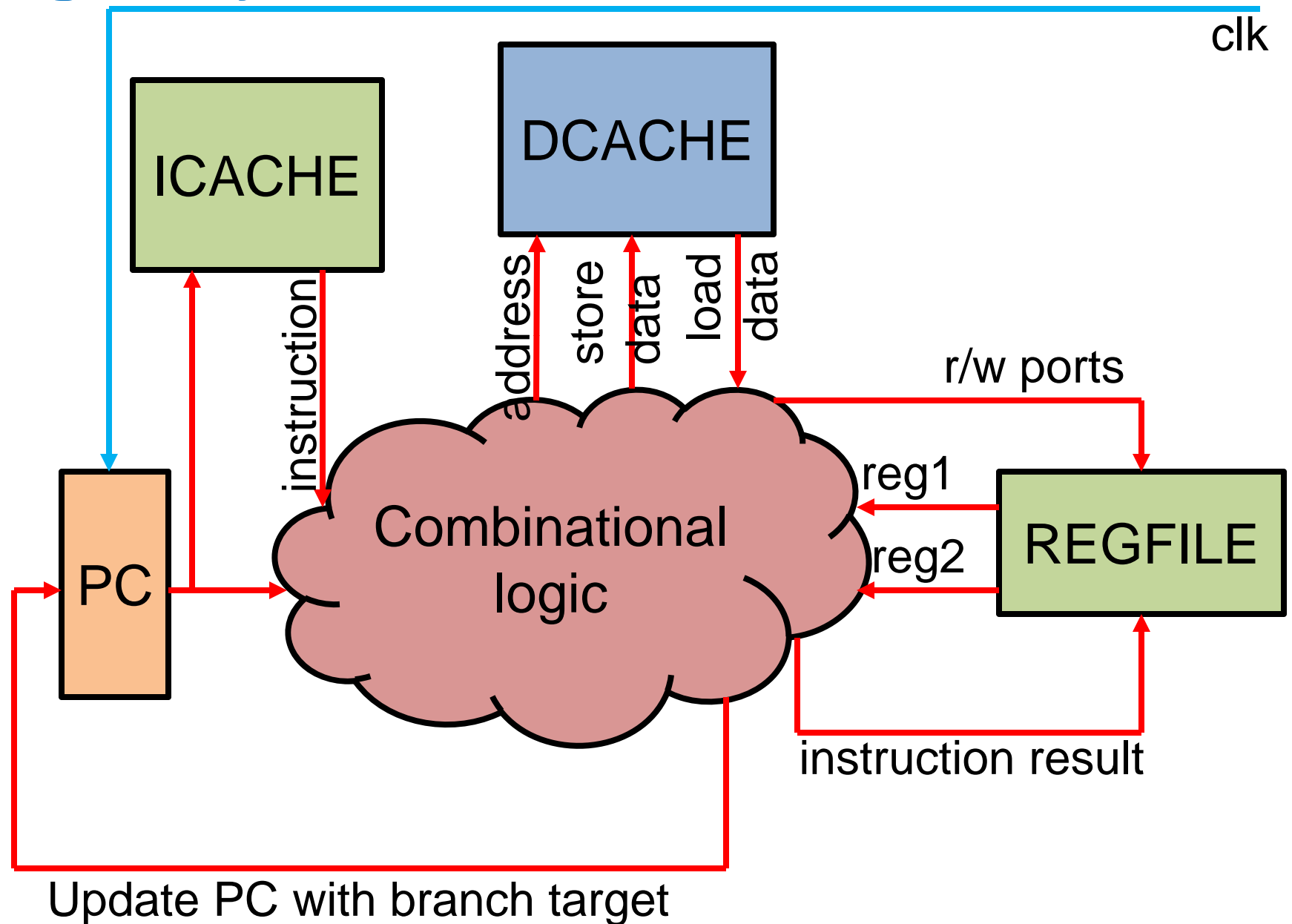
Agenda

- Basics of computer architecture
 - Basics of the basics
 - Instruction set architecture (ISA)
- Processor design
 - Caches and virtual memory
 - Communicating with environment
 - Performance measurement
 - Performance optimization
 - Multi-core processors
- Basics of operating systems

Abstract model of computer

- Each instruction undergoes five stages
 - Stage 0 (IF): fetch the instruction pointed to by program counter from memory
 - Stage 1 (ID/RF): decode the instruction to extract various fields and read source register operands
 - Stage 2 (EX): execute the instruction in ALU; compute address of load/store instructions; update program counter to branch target (if this instruction is a control transfer instruction)
 - Stage 3 (MEM): access memory if load/store instruction; use address computed in stage 2
 - Stage 4 (WB): write result back to destination register if the instruction produces a result

Single-cycle instruction execution



Multi-cycle instruction execution

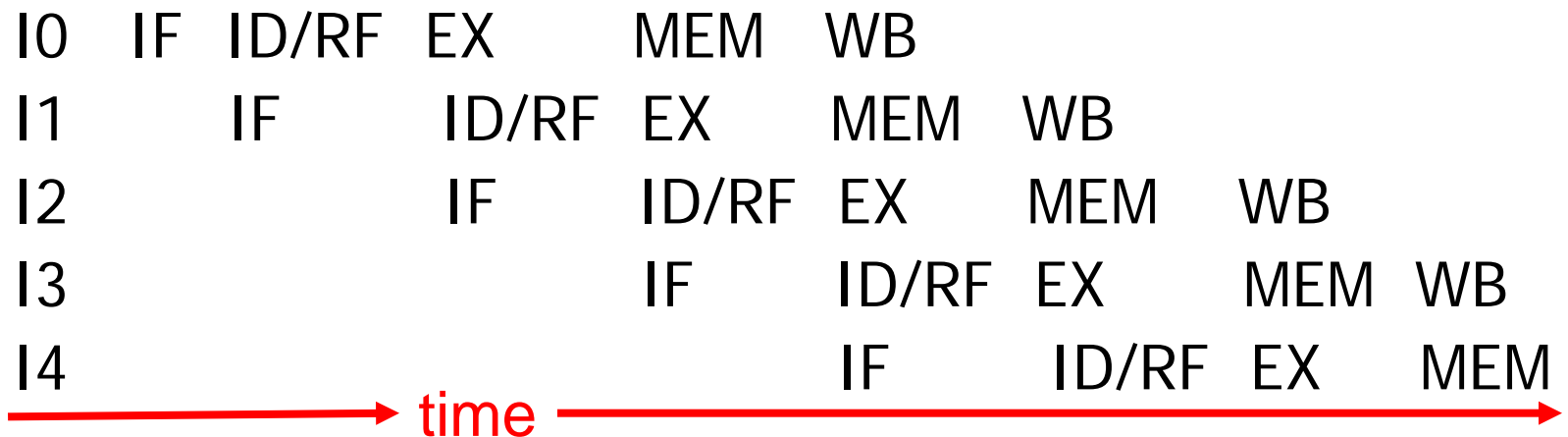
- Each stage takes one cycle to execute
 - Processor is now a five state FSM and the memory elements are sequential
 - Need flip-flops at stage boundaries and each stage is a combinational logic
 - Clock cycle time = latency of the longest stage

Pipelined instruction execution

- Observations from multi-cycle design
 - In the second cycle, I know if it is a branch; if not, start fetching the next instruction?
 - When the ALU is doing an addition (say), the decoder is sitting idle; can we use it for some other instruction?
 - In summary, exactly one stage is active at any point in time: wastes hardware resources
- Form a pipeline
 - Process five instructions in parallel
 - Each instruction is in a different stage of processing (called pipe stage)

Pipelined instruction execution

- Individual instruction latency is five cycles, but ideally can finish one instruction every cycle after the pipeline is filled up
 - Ideal CPI of 1.0 at the clock frequency of multi-cycle design
 - Execution time is ideally one-fifth of the multi-cycle design
 - Instruction throughput improves five times (number of instructions completed in a given time)

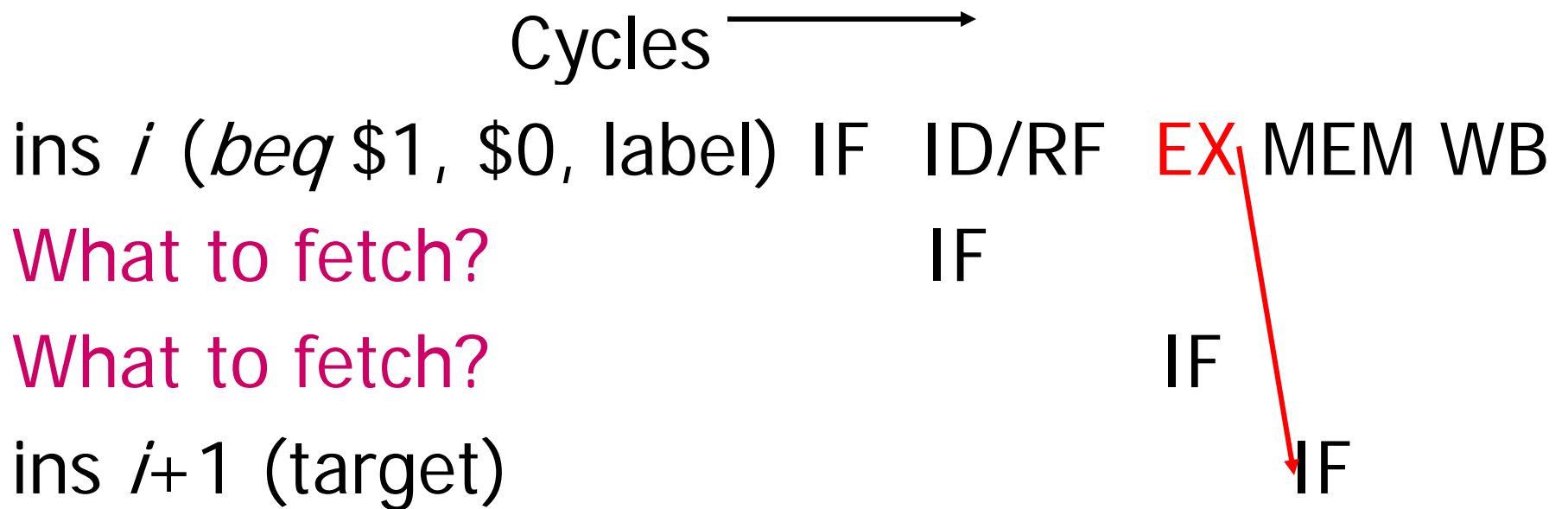


Pipelined instruction execution

- Gains
 - Extracting parallelism from a sequential instruction stream: known as instruction-level parallelism (ILP)
 - Can complete one instruction every cycle (ideally)
- Loss
 - Each pipe stage may get lengthened a little bit due to control overhead (skew time, setup time, propagation delay): limits the gain due to pipelining
 - Each instruction may take slightly longer for this reason
 - Bigger aggregate memory bandwidth: icache and dcache may miss in the same cycle
 - Pipeline hazards

Pipeline hazard: control

- Branches pose a problem



- Two pipeline **bubbles**: increases average CPI

Branch prediction

- Today all processors rely on branch predictors that observe the behavior of individual branches and learn to predict their future behavior
 - Makes it possible to infer the next instruction's PC even before the branch executes
 - Pipeline must be flushed on a wrong prediction

Pipeline hazard: data

- Pipelining disturbs the sequential thought-process
 - Data dependencies among instructions start to show up

add \$1, \$2, \$3

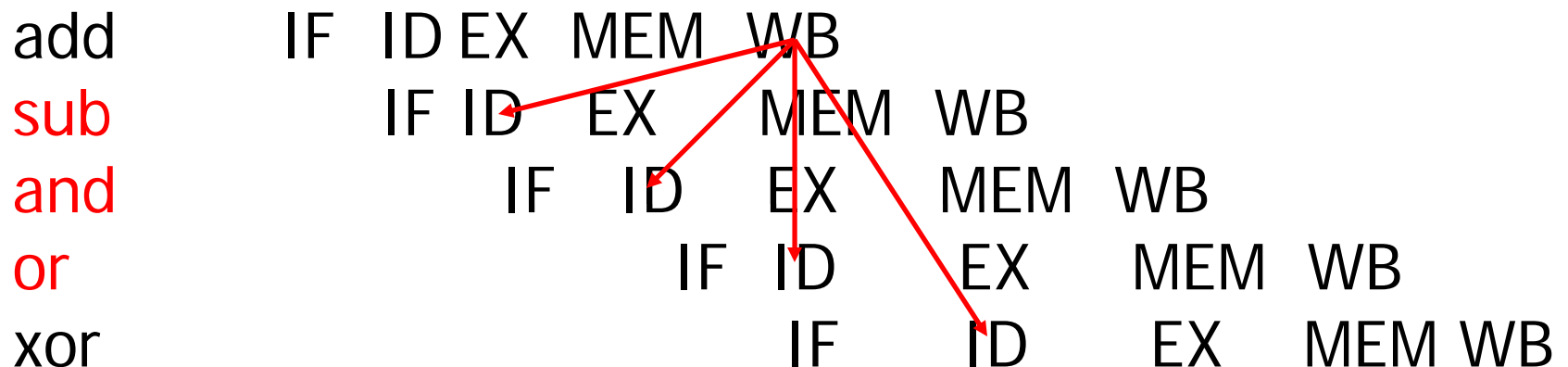
sub \$4, \$1, \$5

and \$6, \$1, \$7

or \$8, \$1, \$9

xor \$10, \$1, \$11

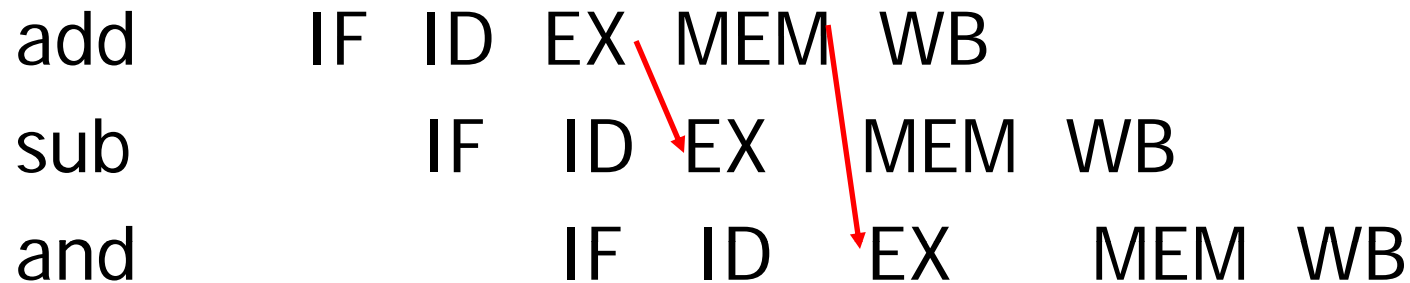
- Result of add is needed by all instructions (RAW hazard: read after write hazard)



Pipeline hazard: data

- How to avoid increasing CPI?

- Can we forward the correct value just in time?

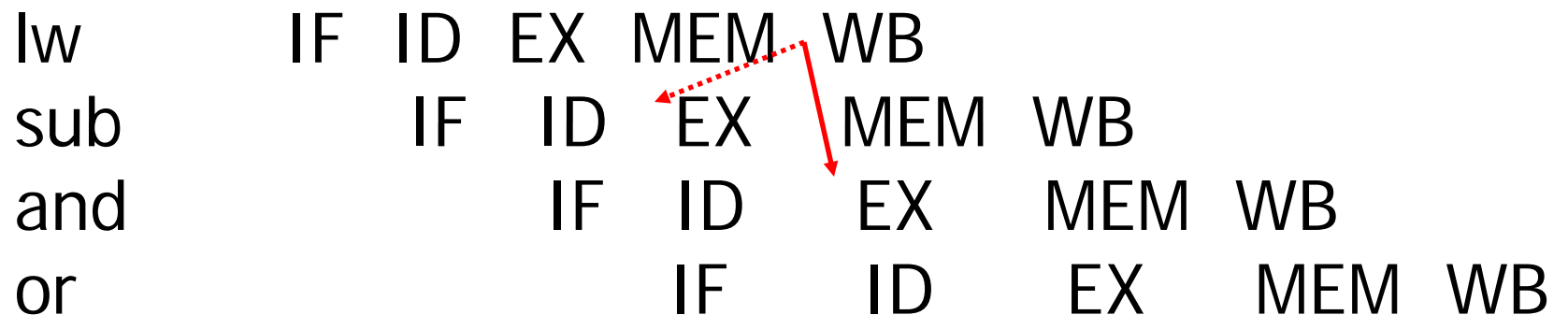


- Read wrong value in ID/RF, but bypassed value overrides it (need a multiplexor for each ALU input to choose between the RF value and the bypassed value)

Pipeline hazard: data

- Can we always avoid stalling?

```
lw    $1, 0($2)
sub   $4, $1, $5
and   $6, $1, $7
or    $8, $1, $9
```

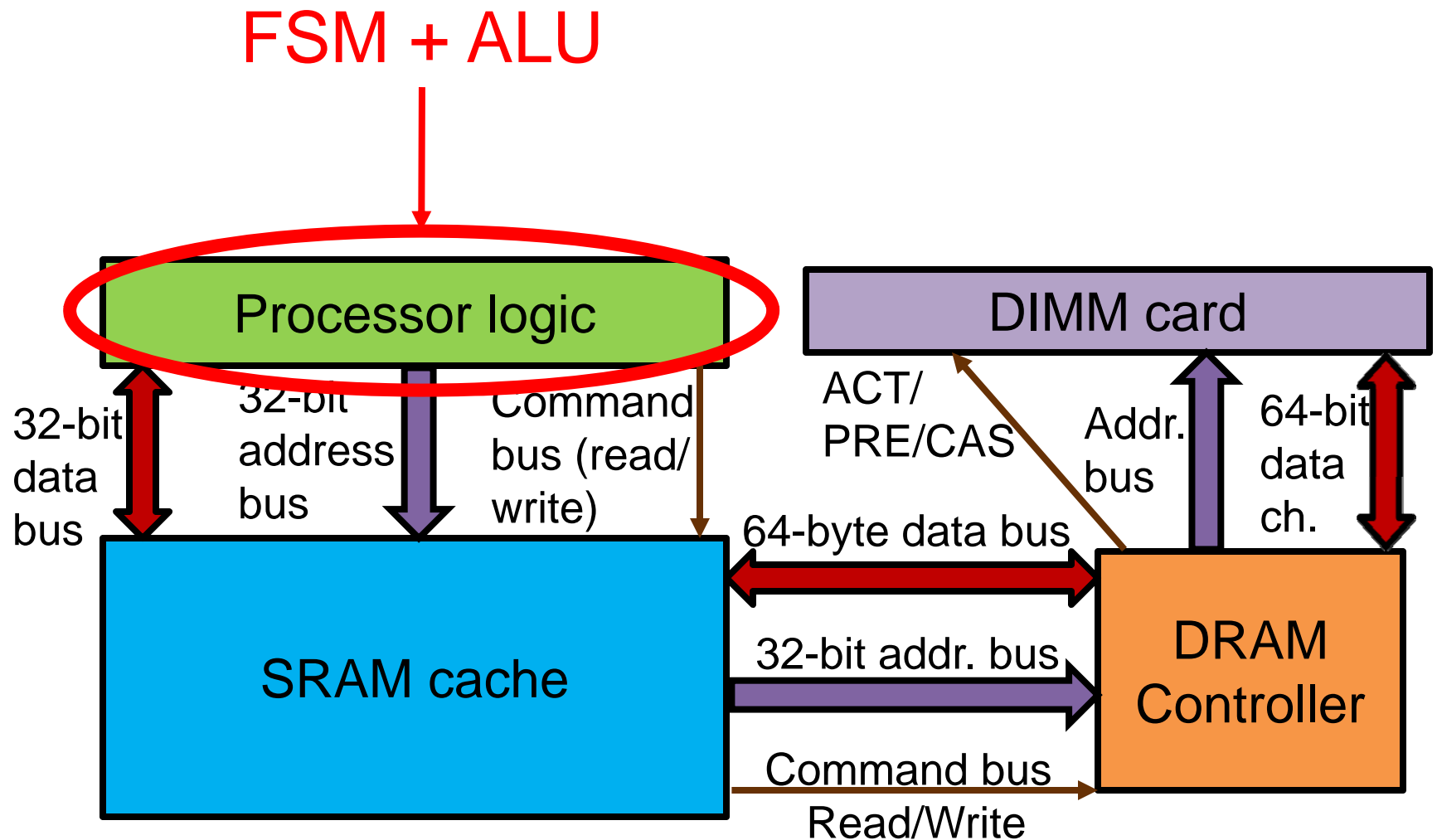


- Need some time travel (backwards)! Not yet feasible!!
- One option: hardware *pipeline interlock* to stall the *sub* by a cycle

Pipelined instruction execution

- Hazards can cause pipeline stalls and introduce bubbles in the pipeline depending on the pipeline organization
- Overall speedup of pipelining over multi-cycle non-pipelined implementation = $\text{number of pipe stages} / (1 + \text{average stall cycles per instruction})$

Abstract model of computer



ALU architecture

- ALU is responsible for executing the core of the instructions
 - Everything except load/store instructions
- ALU takes two inputs for most instructions and produces a result that may or may not get written to a destination register
 - For example, control transfer instructions do not write to a general-purpose register, but writes to the program counter
- ALUs are of two kinds: integer and floating-point
 - The floating-point ALU is often referred to as the floating-point unit (FPU)

Agenda

➤ Basics of computer architecture

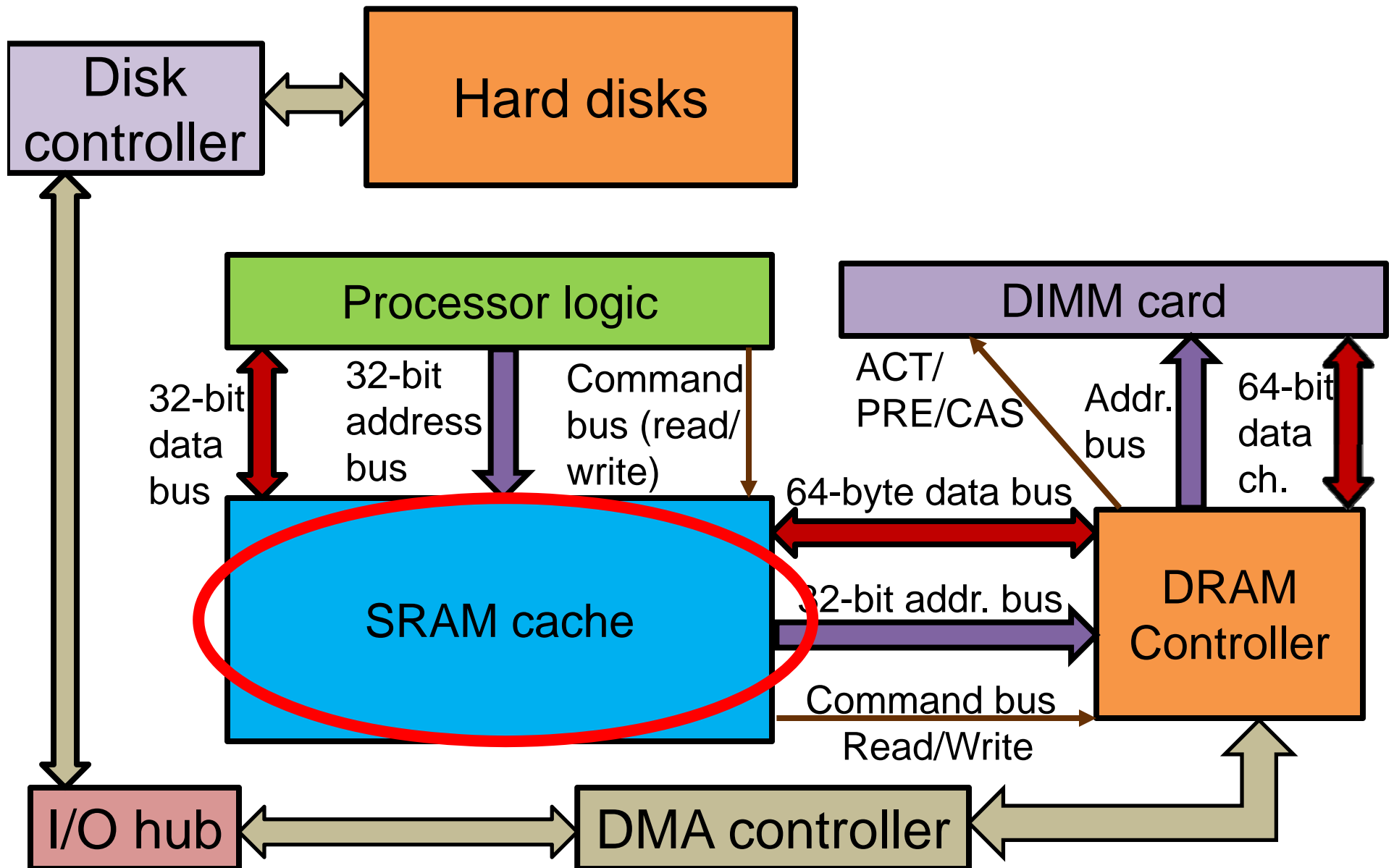
- Basics of the basics
- Instruction set architecture (ISA)
- Processor design

➤ Caches and virtual memory

- Communicating with environment
- Performance measurement
- Performance optimization
- Multi-core processors

• Basics of operating systems

Abstract model of computer



Locality principles

- Principles of locality exhibited by programs
 - Code and data accessed now are likely to be accessed again in near-future
 - Any interesting program would have loops and/or recursions
 - Justifies why code and data accesses may be repeated
 - Example: reuse of rows of A and columns of B when multiplying matrices A and B
 - Known as temporal locality
 - Code and data allocated close to the code and data being accessed now are likely to be accessed in near-future
 - Sequential code access
 - Sequential data access (e.g., walking over an array)
 - Known as spatial locality

Memory and storage hierarchy

- Locality principles imply an important corollary
 - Programs usually work on a small portions of code and data at a time
 - The code and data needed over a time window of length t could be a subset of the code and data needed over a bigger time window of length t'
 - Think about nested loops
- This corollary is exploited to build a hierarchy of memory and storage structures
 - Keep most recently used code and data close to the processor because this is needed now
 - Keep increasingly larger supersets of code and data gradually away from the processor

Memory and storage hierarchy

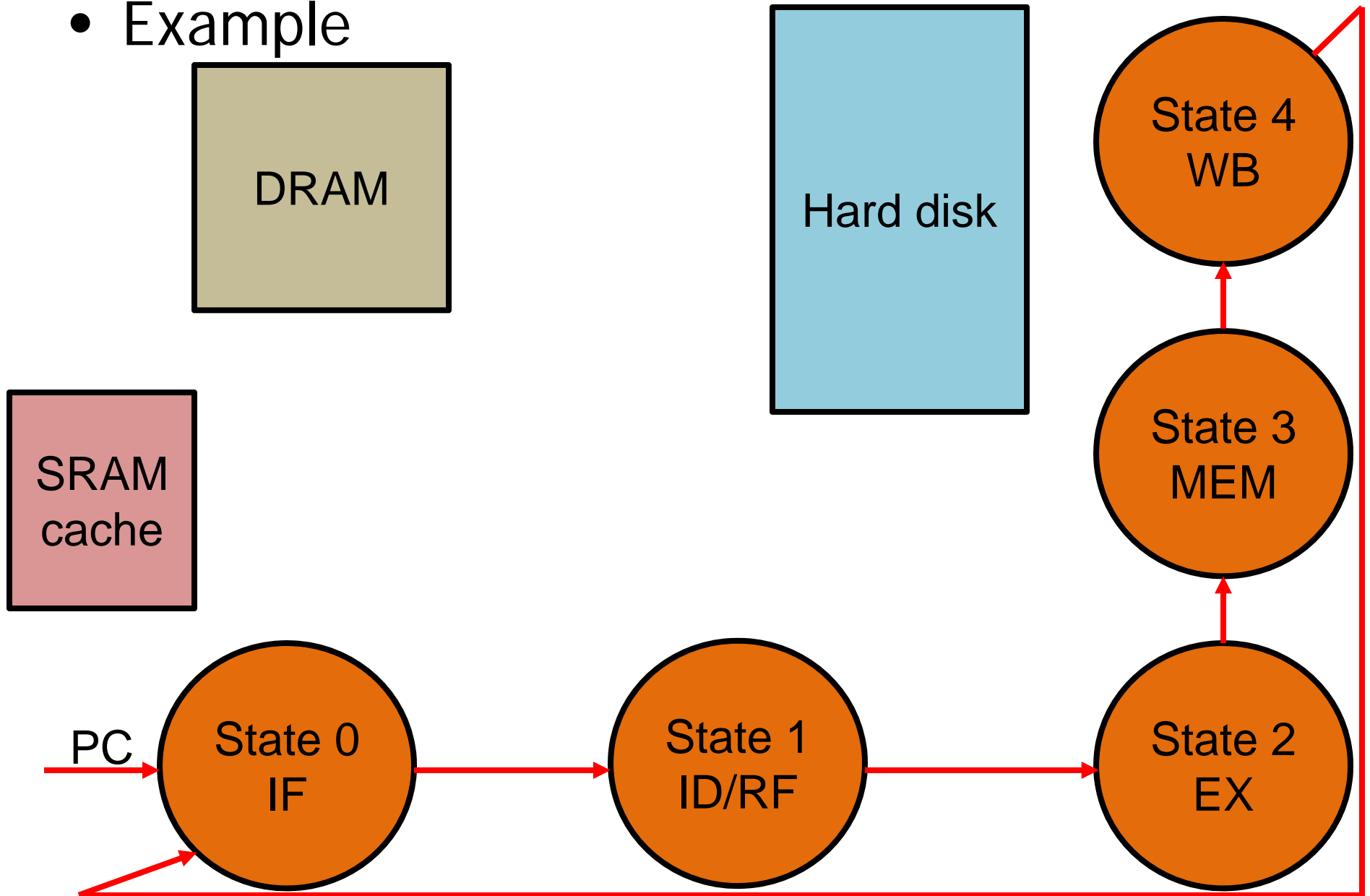
- Why not keep everything in a large on-chip SRAM?
 - Expensive and slow
- Memory and storage parts are usually arranged in a hierarchy
 - SRAM caches are closest to the processor logic, smallest in size, and fastest
 - Total on-chip cache is usually few tens of MBs
 - DRAM is outside processor chip, much larger in size, much slower than SRAM caches
 - Tens to hundreds of GBs
 - Hard disk holds everything, non-volatile, very large, very slow
 - Tens to hundreds of TBs

Memory and storage hierarchy

- Hierarchical organization allows very fast access to a small subset of code and data needed now from the SRAM cache
- Later this code and data can be exchanged to bring something else from DRAM
 - SRAM caches have finite capacity, so something must be replaced to bring something new if the cache is already full
- Also, code and data in DRAM can be swapped with something else from hard disk on demand
 - Less frequent than exchange between SRAM and DRAM

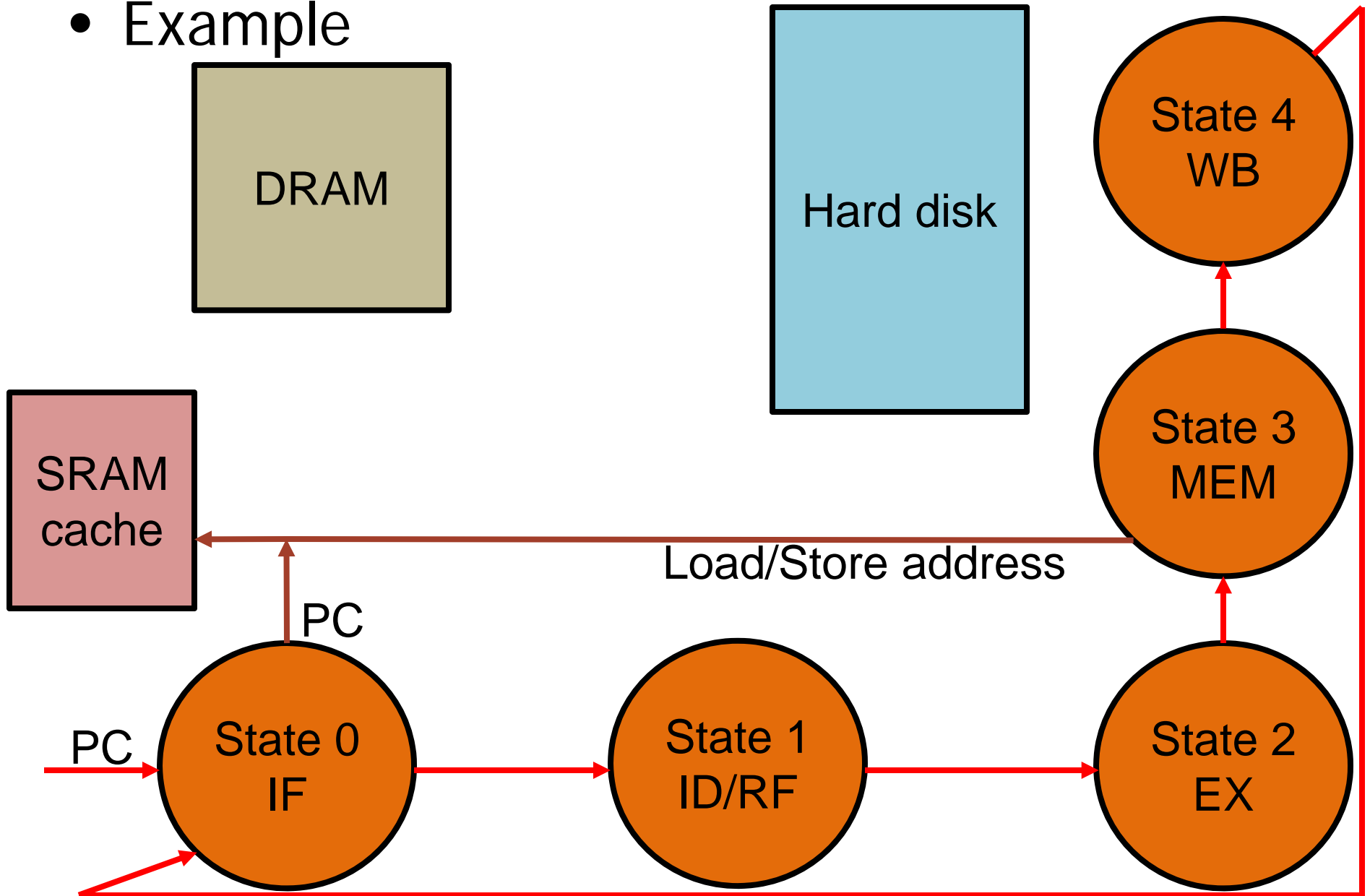
Memory and storage hierarchy

- Example



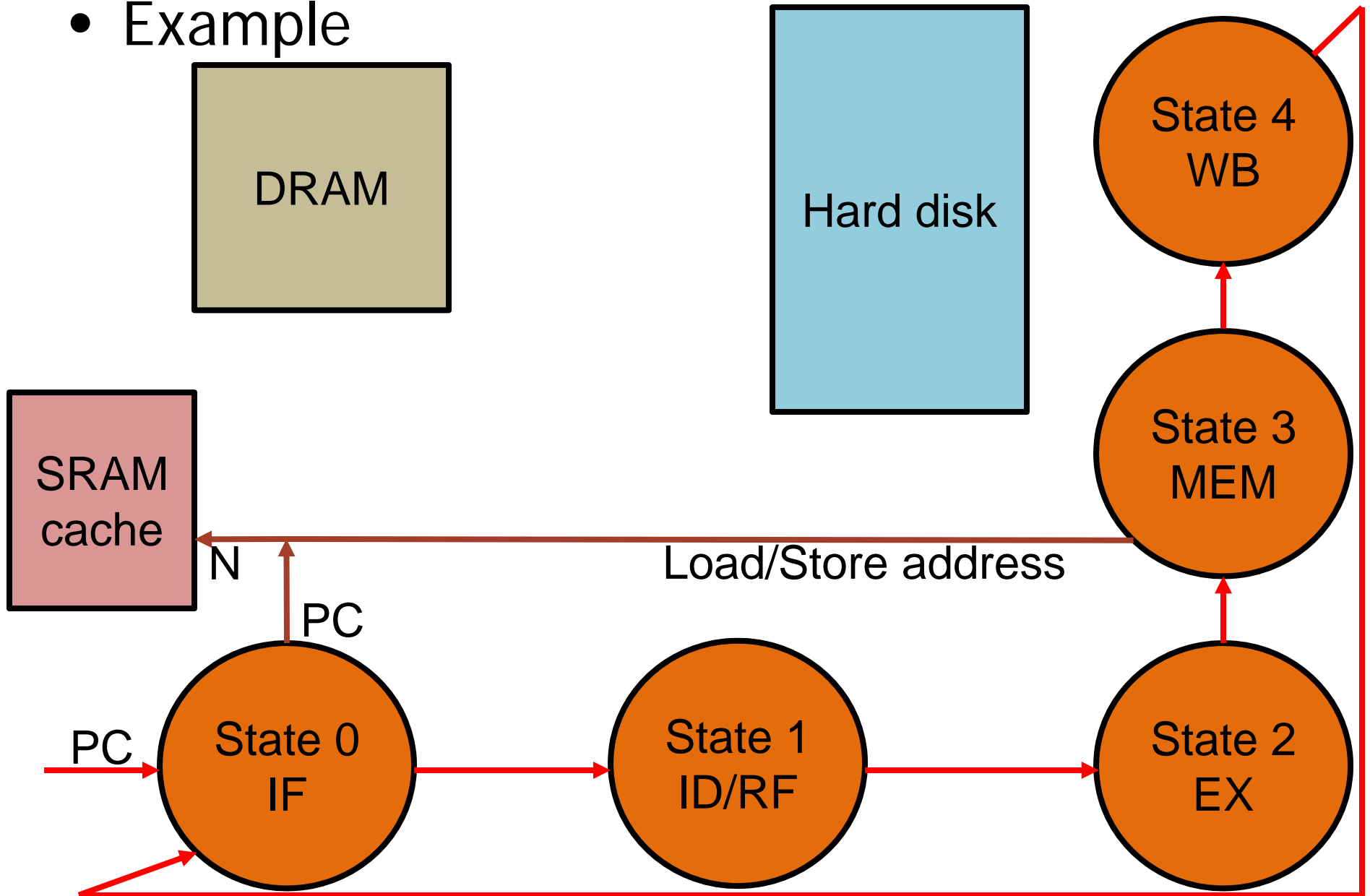
Memory and storage hierarchy

- Example



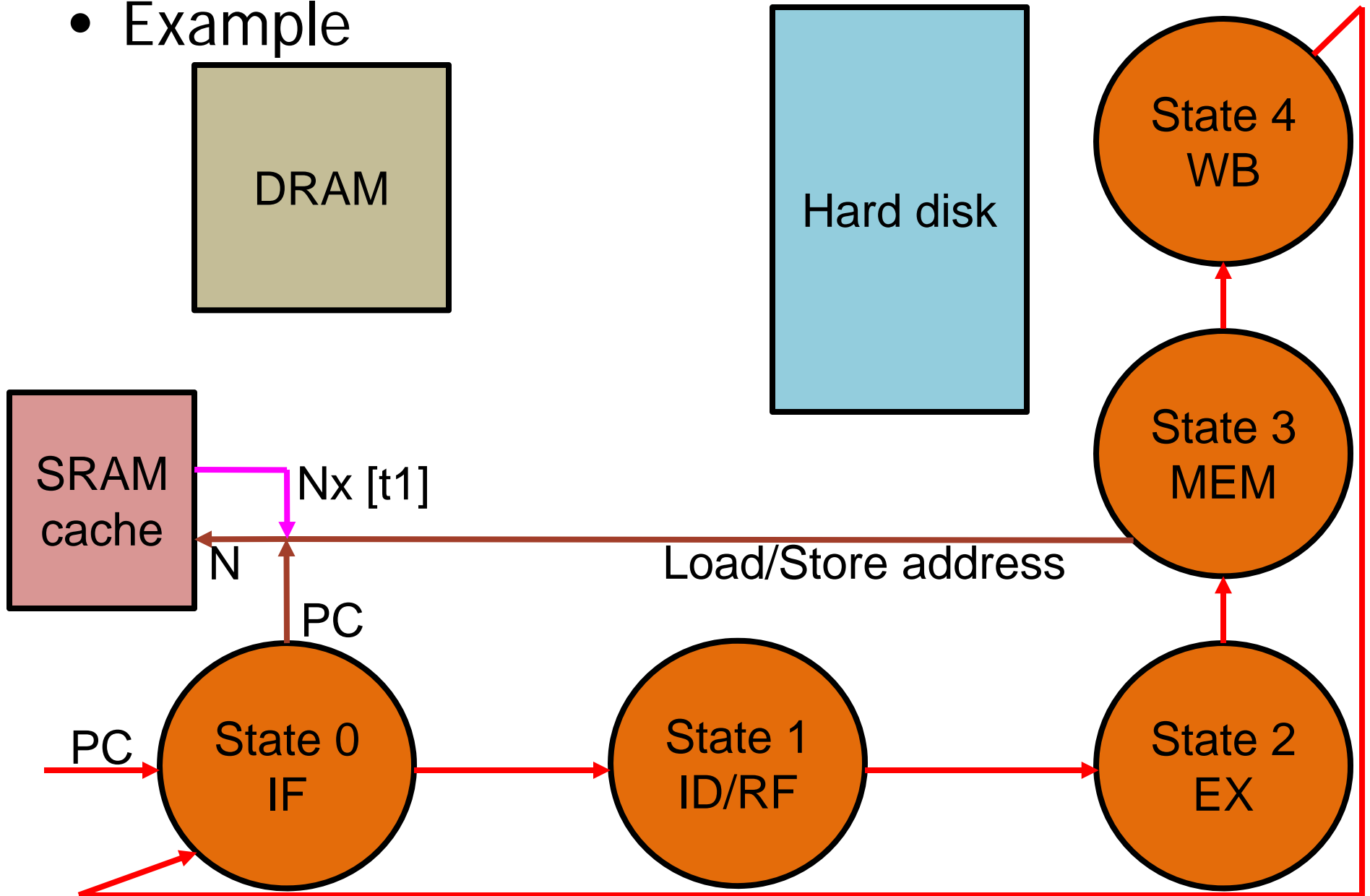
Memory and storage hierarchy

- Example



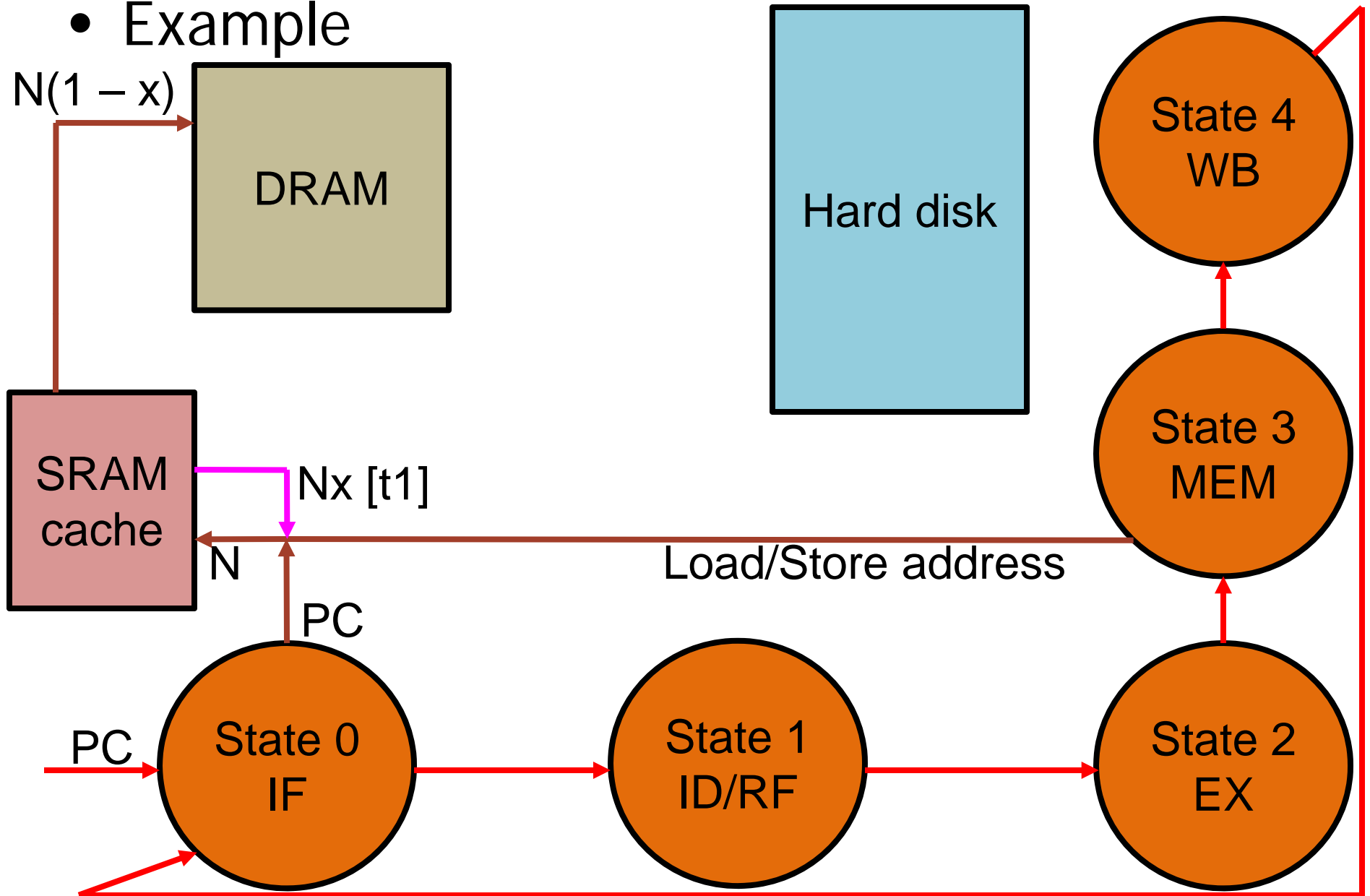
Memory and storage hierarchy

- Example



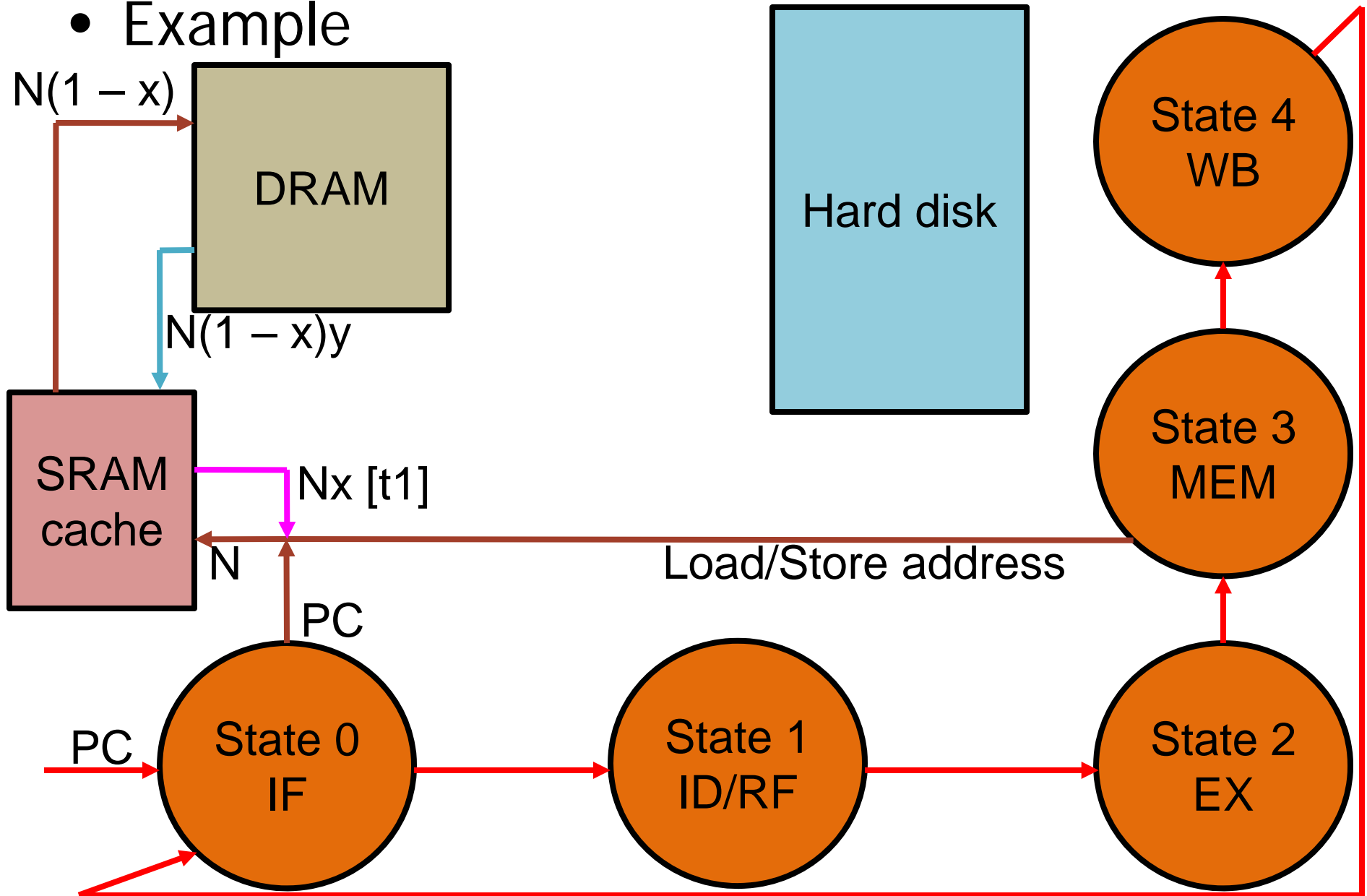
Memory and storage hierarchy

- Example



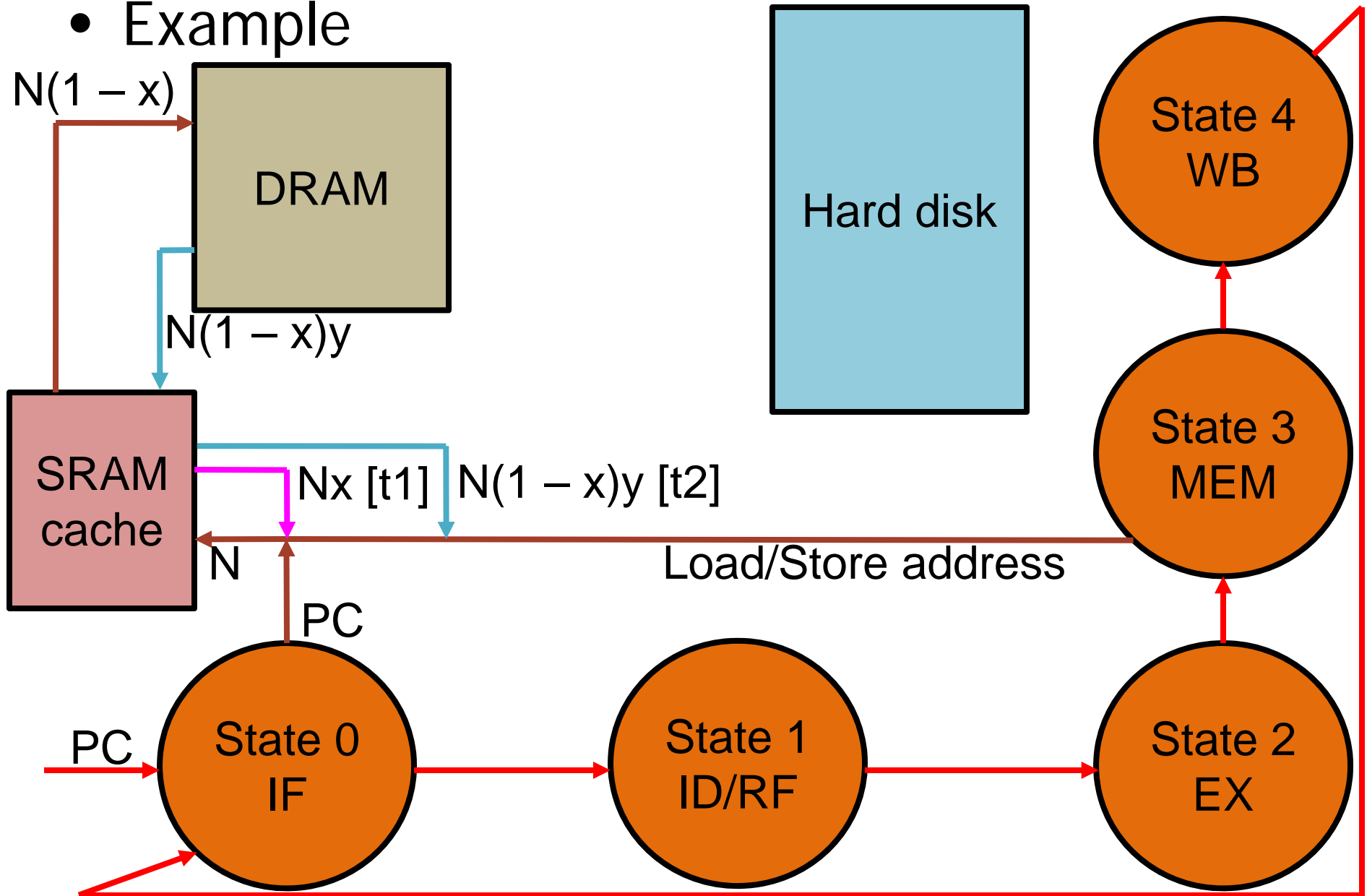
Memory and storage hierarchy

- Example



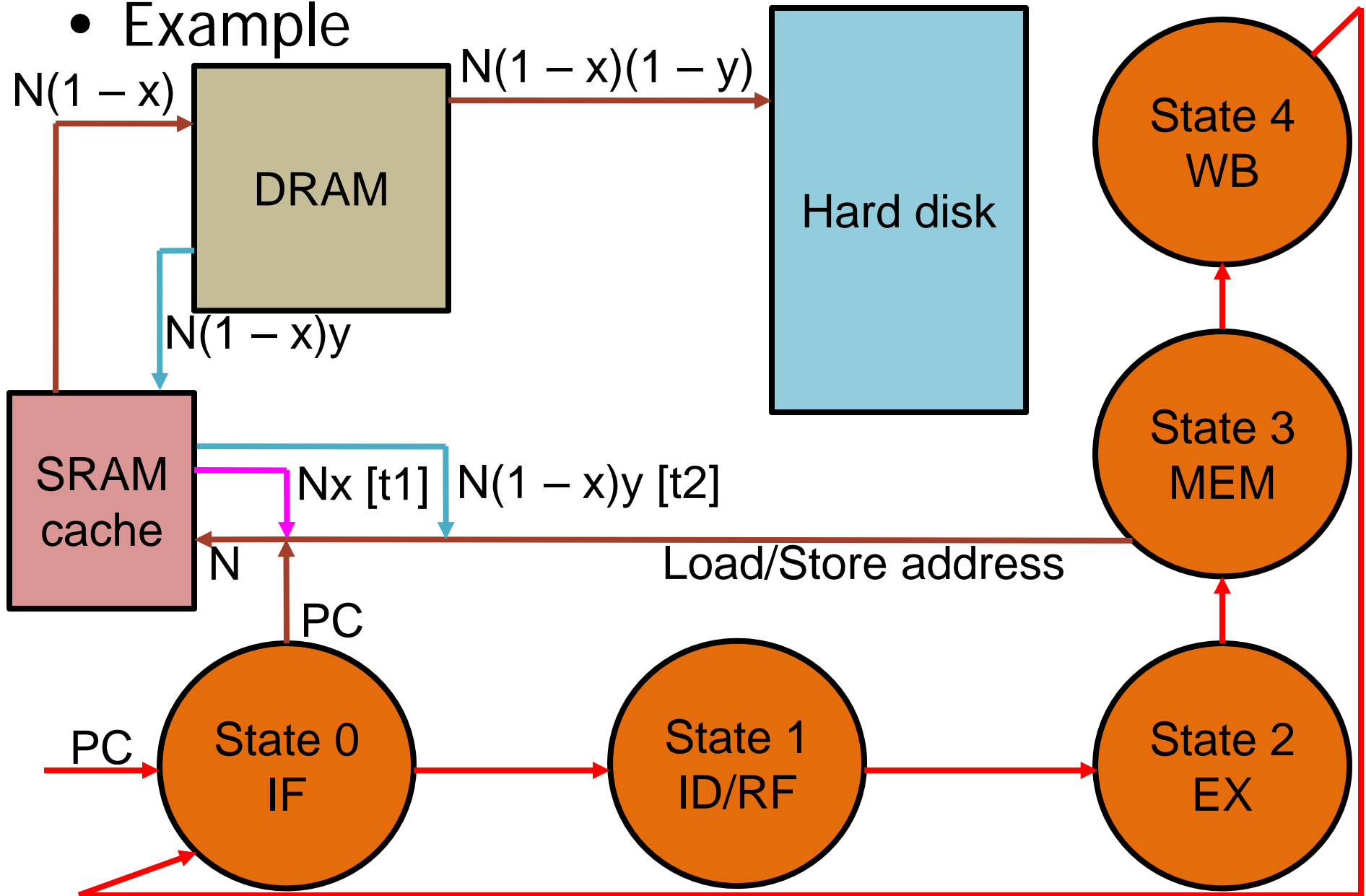
Memory and storage hierarchy

- Example



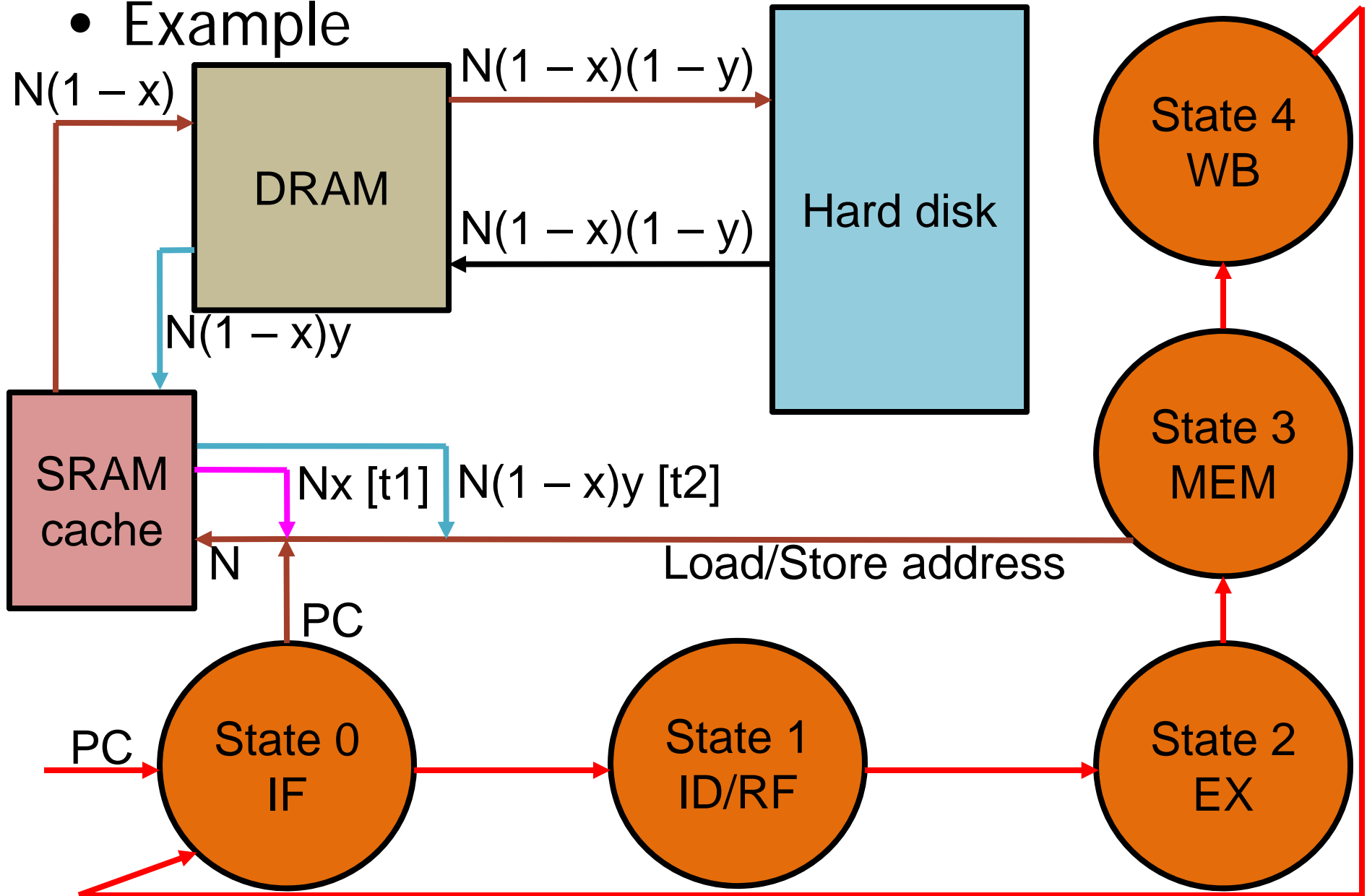
Memory and storage hierarchy

- Example



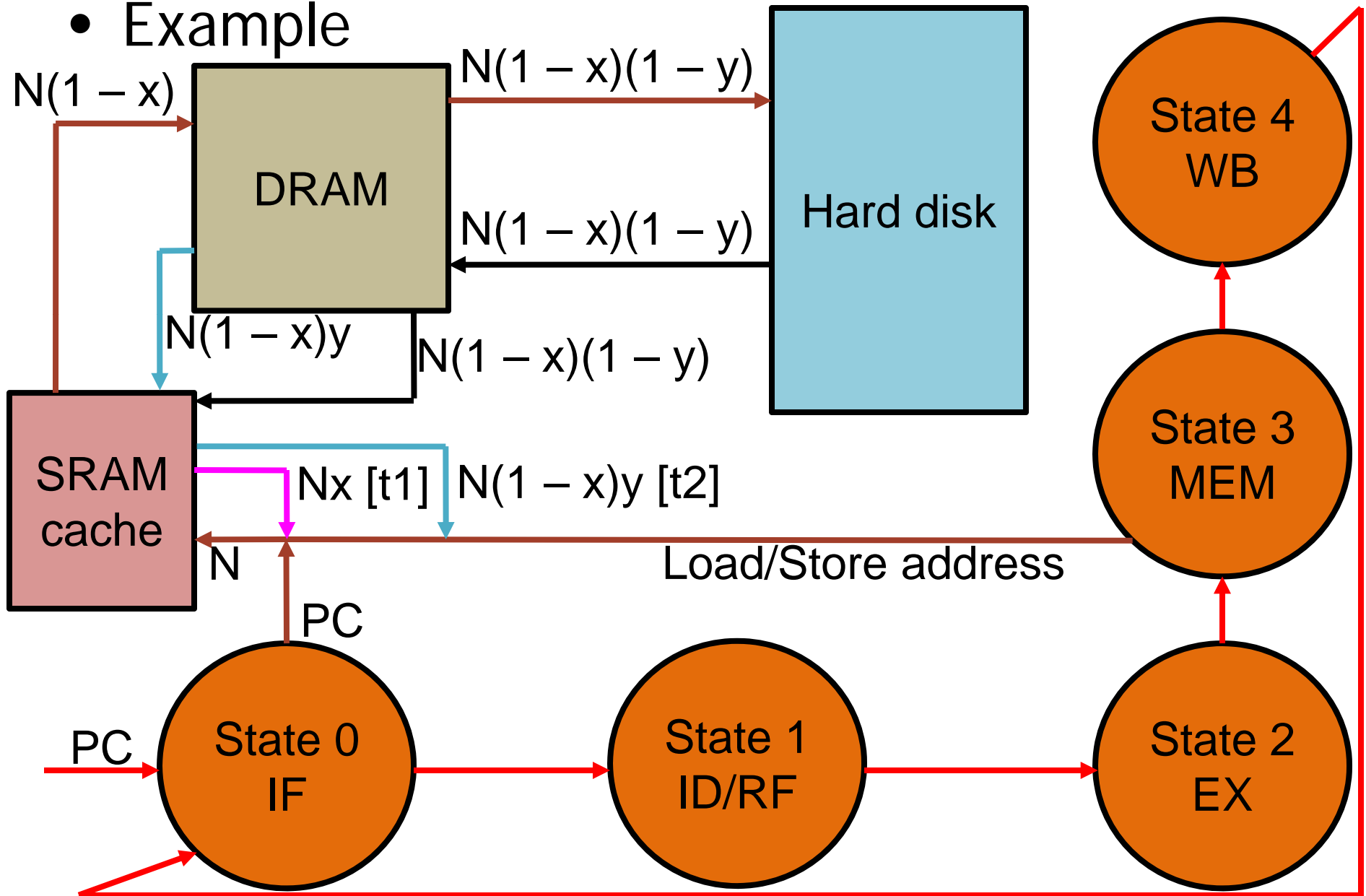
Memory and storage hierarchy

- Example



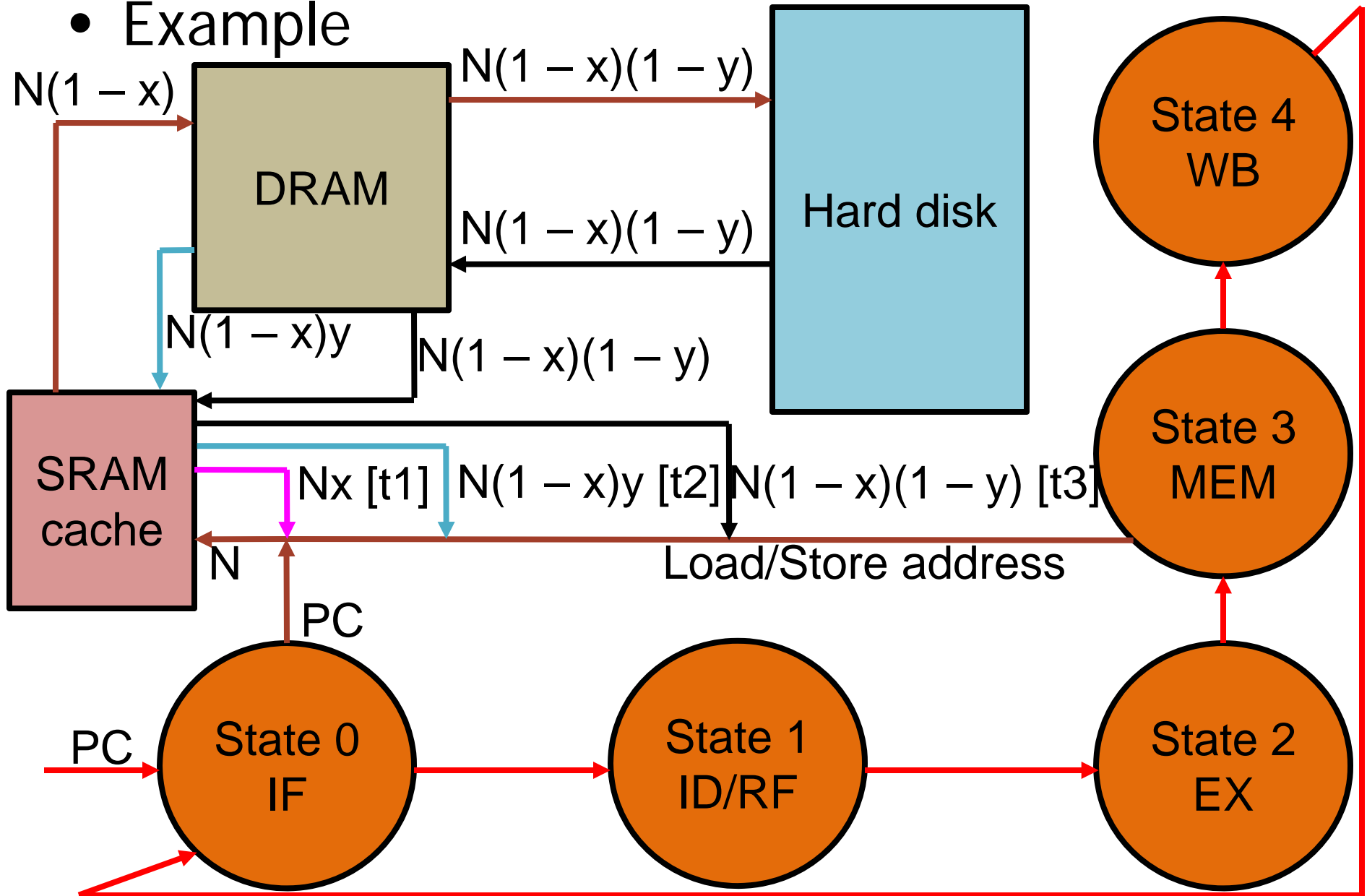
Memory and storage hierarchy

- Example



Memory and storage hierarchy

- Example



Memory and storage hierarchy

- Example
 - Suppose a program's load/store instructions and instruction fetcher generate N memory accesses
 - Nx accesses find the requested data in on-chip SRAM cache ($x < 1$)
 - $N(1 - x)y$ accesses find the requested data in DRAM ($y < 1$)
 - Remaining accesses fetch data from hard disk
 - An access to SRAM cache requires time t_1 (hit)
 - An access to DRAM requires time t_2 (cache miss)
 - An access to hard disk requires time t_3
 - Average access time = $(Nxt_1 + N(1 - x)yt_2 + N(1 - x)(1 - y)t_3)/N$
 - Since $t_1 \ll t_2 \ll t_3$, as x and/or y increase(s), the average access time goes down

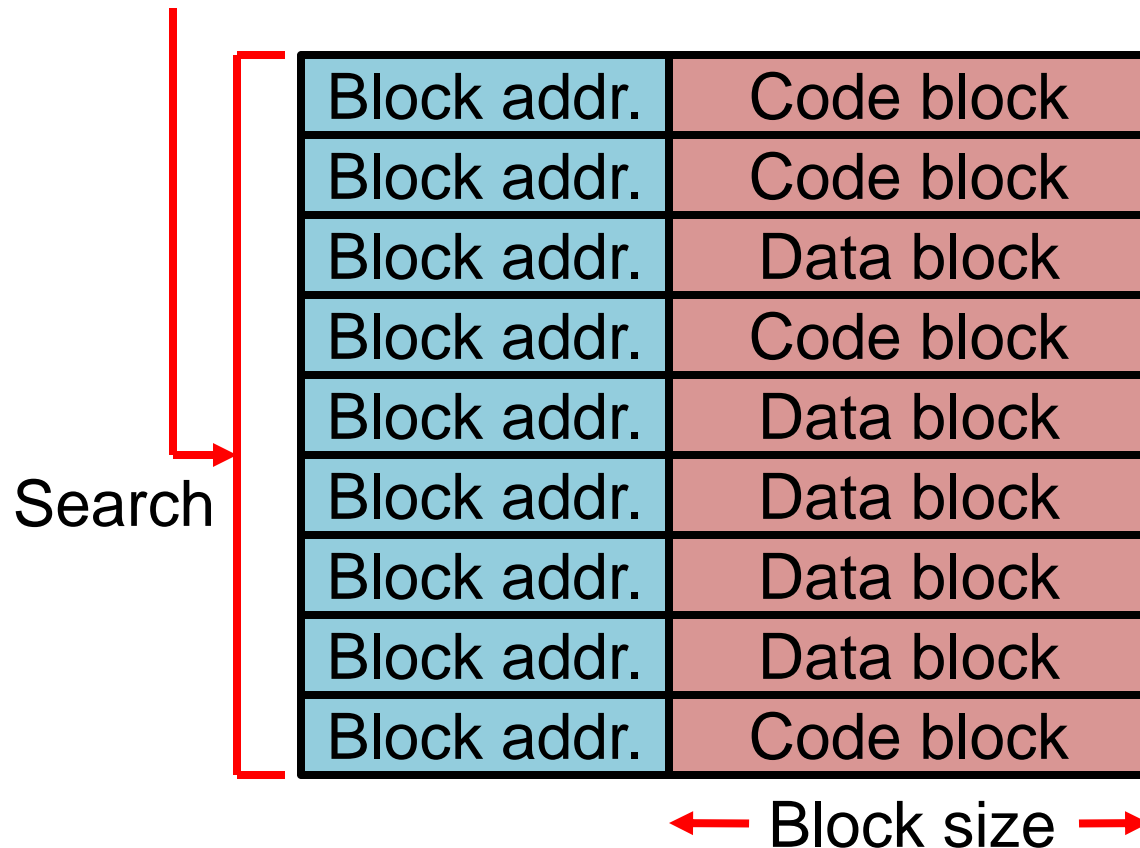
Basics of SRAM cache

- Cache is looked up in two events
 - State 0 of FSM generates an access for instruction using program counter as the address
 - State 3 of FSM generates an access for data using the address computed in state 2
 - Requested block is searched in the cache
 - Needs to store the address along with each block
 - Block address is the search key
 - This searching time can be very large if done sequentially
 - A parallel search would require a large number of comparators (equal to number of blocks in the cache)
 - Would consume a lot of power and area

Basics of SRAM cache

- Start with a simple design

Address $\gg \log_2(\text{Block size})$



Address is either the PC or the load/store address

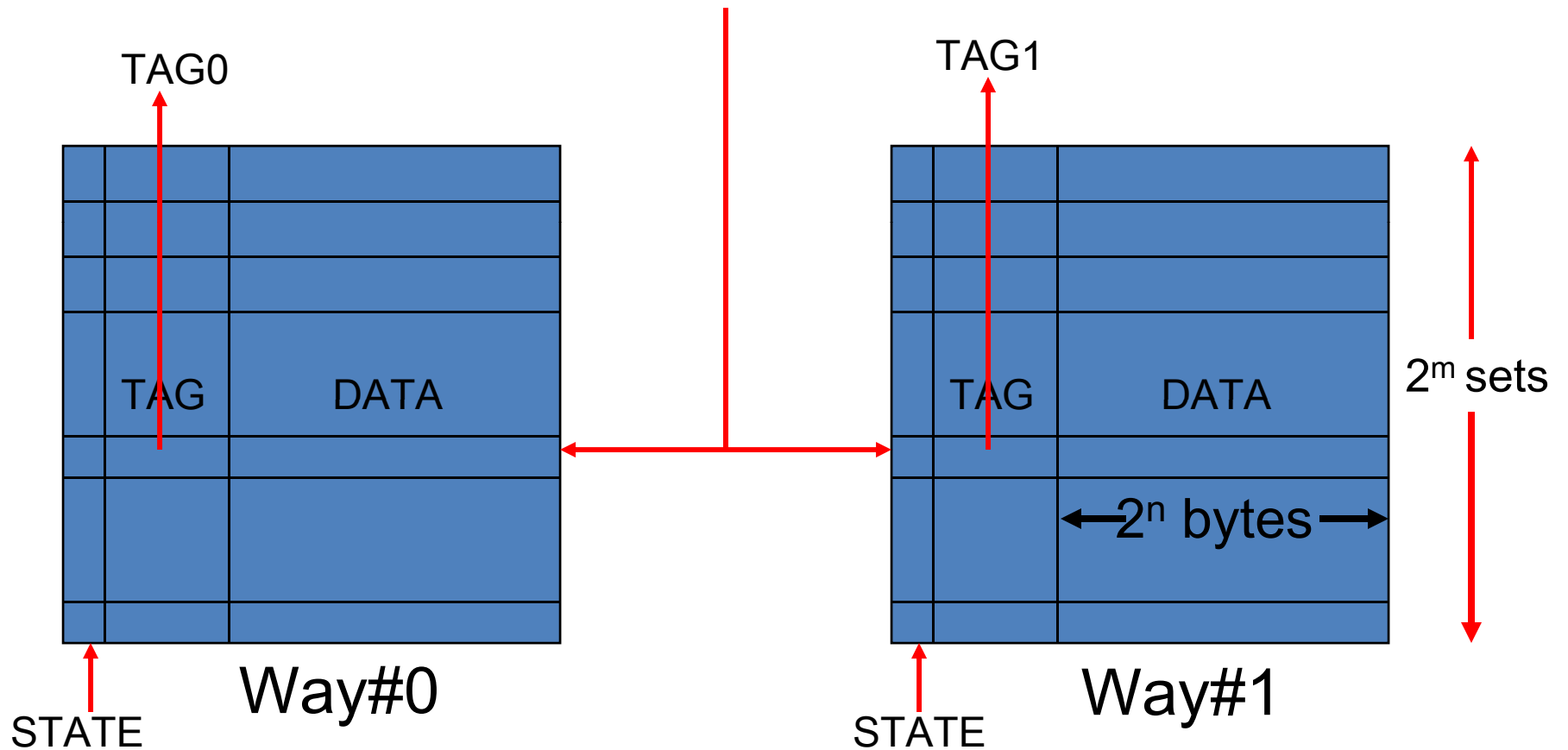
Basics of SRAM cache

- If the looked up block is found in the cache, it is called a cache hit; otherwise it is a cache miss
 - Cache miss requests are forwarded to the DRAM controller for further handling
 - Eventually the DRAM controller will respond with the requested block and it will be allocated in the cache
 - What if the cache is full?
 - Needs to replace a block
 - Which block to replace?
 - Maybe the block that is not used recently (least-recently-used or LRU replacement algorithm)
 - Maybe a random block (random replacement algorithm)
 - LRU replacement requires keeping track of time of access
 - Random replacement requires a random number generator

Basics of SRAM cache

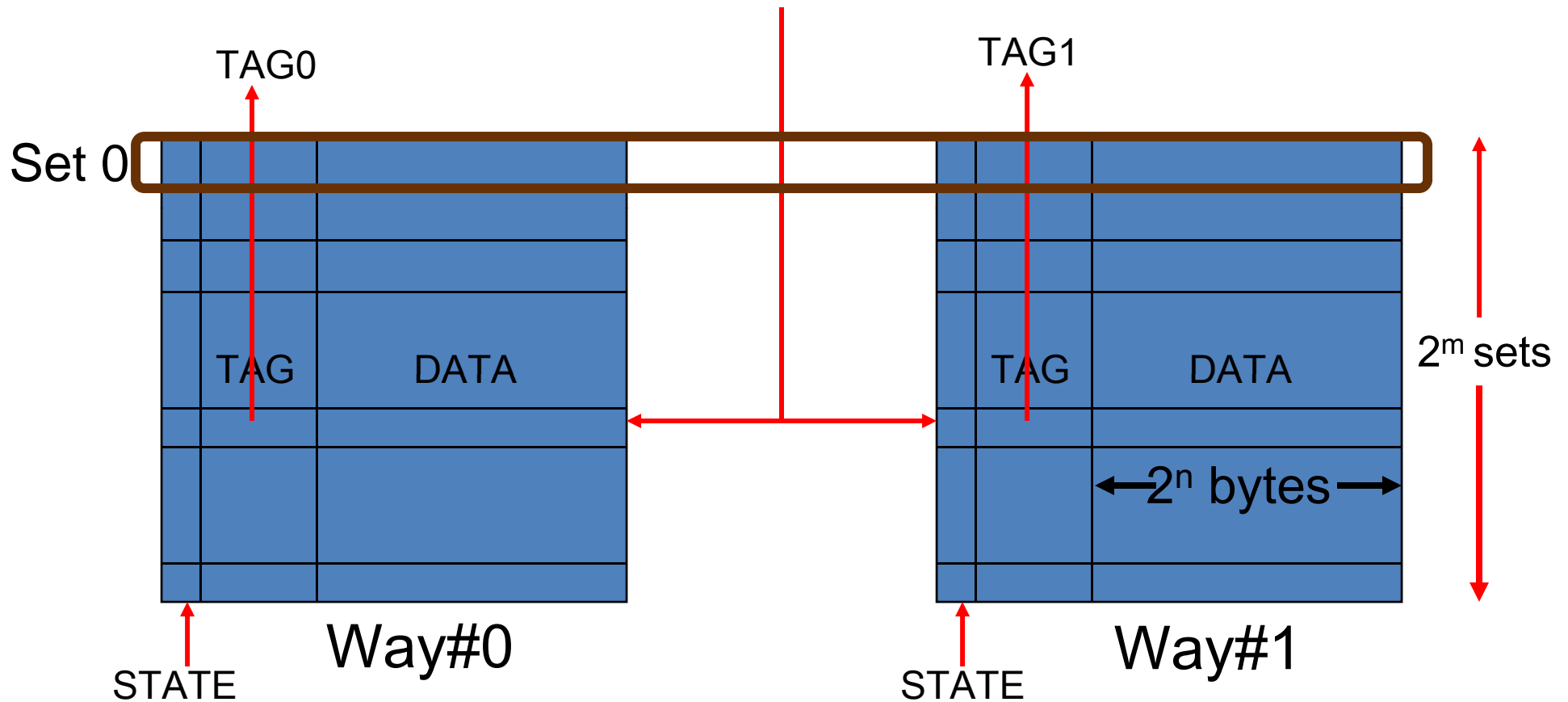
- Cache hits need to be much faster than cache misses to be useful
- To optimize the search time of a cache block, caches are typically organized as hash tables
 - Each hash element has a block, a block address or tag, and a few state bits (e.g., valid/invalid)
 - The blocks in a cache are logically divided into disjoint sets (these are hash buckets)
 - Each set can have a maximum number of valid blocks
 - This maximum number is known as the associativity of the cache
 - For example, a 16 KB cache with 64-byte blocks can have 32 sets each with associativity 8

2-way set associative cache



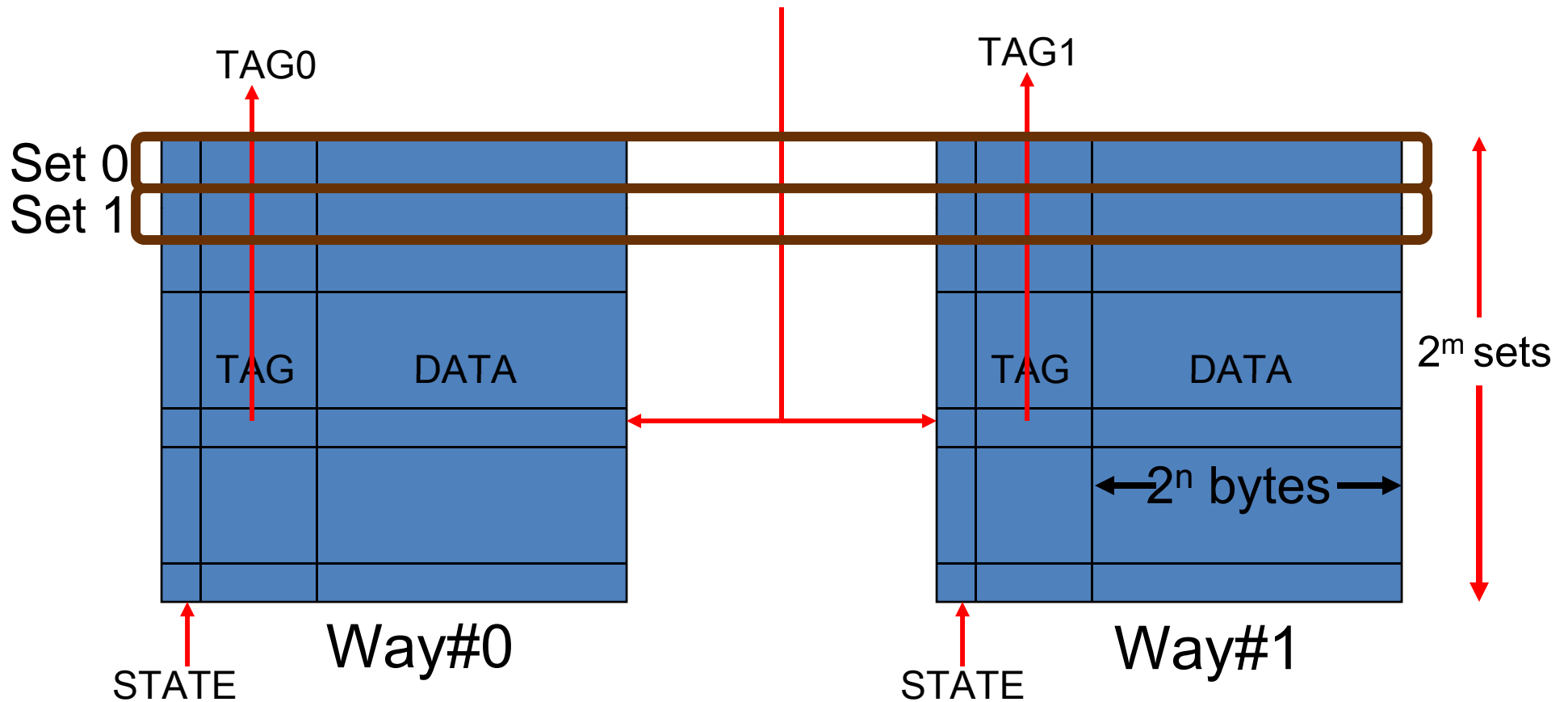
Block address = {TAG, INDEX}; TAG bits can uniquely identify a block in a set

2-way set associative cache



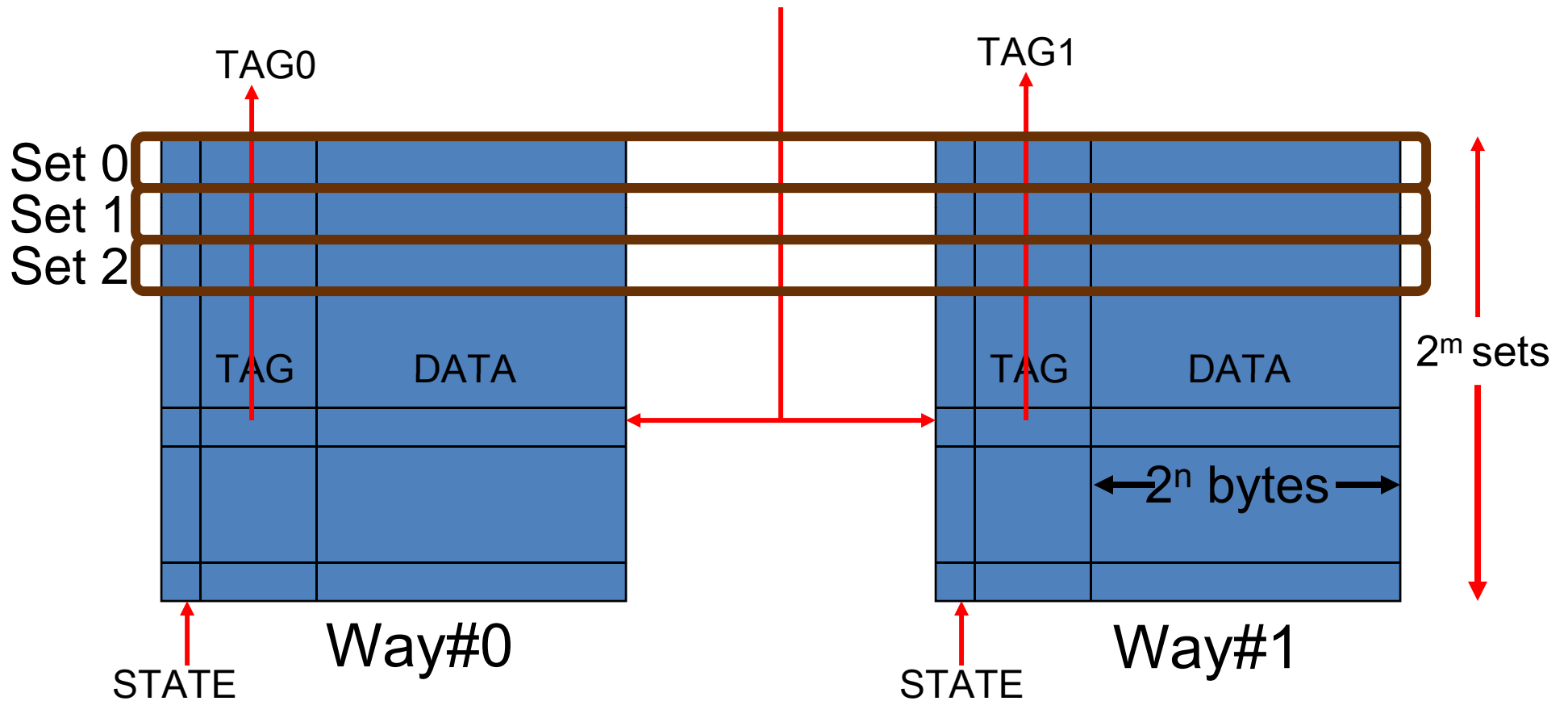
Block address = {TAG, INDEX}; TAG bits can uniquely identify a block in a set

2-way set associative cache



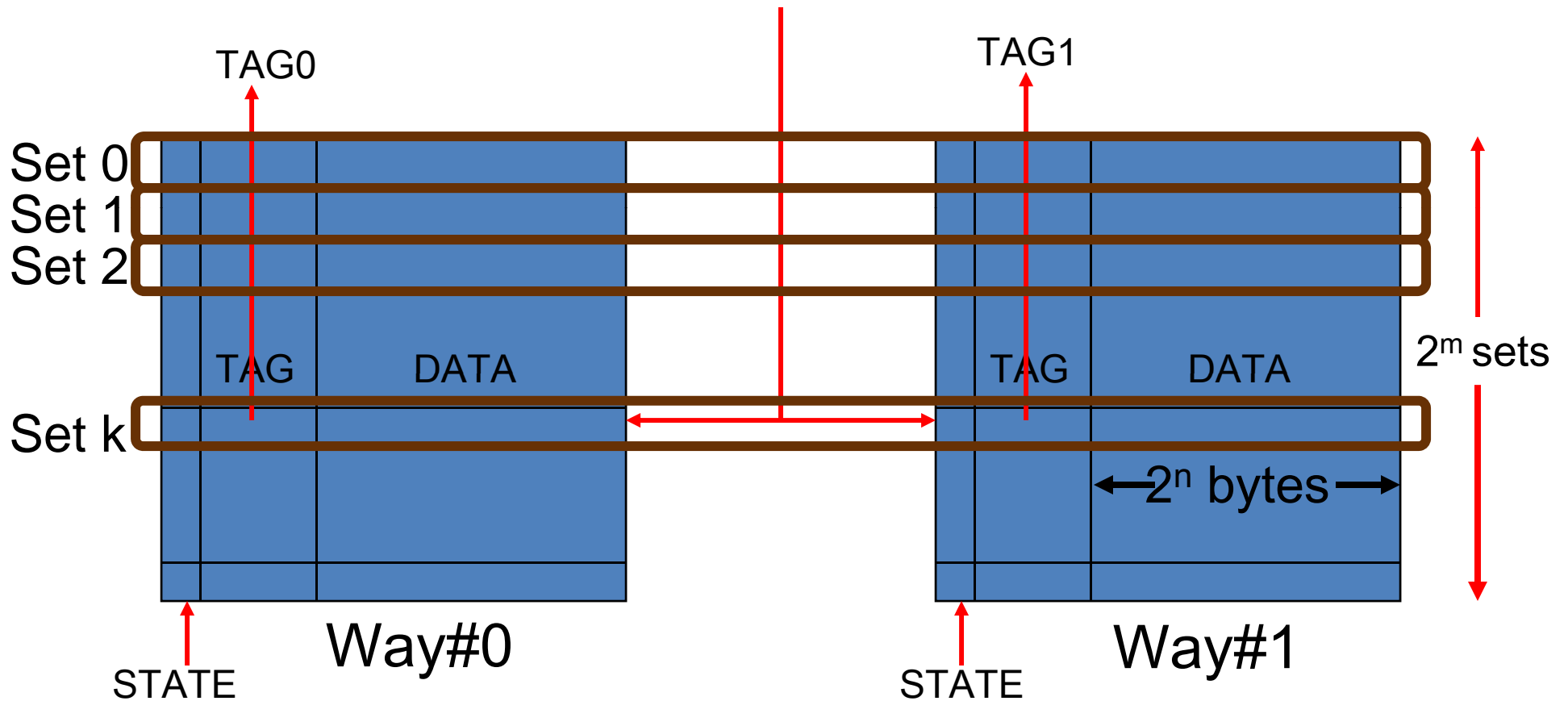
Block address = {TAG, INDEX}; TAG bits can uniquely identify a block in a set

2-way set associative cache



Block address = {TAG, INDEX}; TAG bits can uniquely identify a block in a set

2-way set associative cache



Block address = {TAG, INDEX}; TAG bits can uniquely identify a block in a set

Basics of SRAM cache

- A cache with associativity A is often called an A -way set-associative cache
- To look up a cache with an address
 - The set index is first determined by passing the address through a hash function
 - Within the set, the block addresses are searched in parallel for the target address
 - Number of comparators is equal to the associativity of the cache (which is usually small)

Basics of SRAM cache

- Observations
 - If associativity is one, number of sets is equal to the number of blocks in cache
 - A given block address has a unique location in cache
 - Known as direct-mapped cache (a given block address directly maps to a unique location in cache)
 - Simple design, but may suffer from large number of collisions between blocks (known as conflicts)
 - Block addresses $0, N, 2N, 3N, \dots$ all map to set 0 if there are N blocks in the cache
 - Can increase the number of cache misses compared to a set-associative design
 - This is the minimum possible associativity

Basics of SRAM cache

- Observations
 - If associativity is equal to the number of blocks in the cache, then the number of sets is one
 - Need to search all blocks in the cache during a lookup
 - Known as fully associative cache (usually small in size)
 - A given block can be placed anywhere in the cache
 - Significantly reduces conflicts
 - This is the maximum possible associativity for a given cache capacity
 - $\text{Cache capacity} = \text{no. of blocks} \times \text{block size} = \text{no. of sets} \times \text{no. of ways} \times \text{block size}$

Cache performance

- Definitions
 - Hit rate = number of hits / number of accesses
 - Bigger the better
 - Miss rate = number of misses / number of accesses = $1 - \text{hit rate}$
 - Smaller the better
 - Average memory access time (AMAT) = hit/miss detection time + miss rate x miss penalty
 - Decreasing any of the three terms would improve AMAT
 - Miss rate x miss penalty is often referred to as the memory stall time
 - Overall execution time = CPU time + memory stall time

Multi-level cache hierarchy

- All commercial processors have two or three levels of SRAM cache on chip today
 - L1 is the closest to the processor and L2, L3, ... are further away, get gradually bigger and slower
 - The levels are designed such that the latency of fetching a block from the last level SRAM cache is still much smaller than fetching from DRAM
 - Typical sizes: L1: < 100 KB, L2: < 1 MB, L3: 2-32 MB
 - Associativity typically increases with increasing level
 - Block size may increase or may remain constant
 - Typical round-trip latencies could be L1: <1 ns, L2: ~5 ns, L3: ~10 ns, DRAM: 50-100 ns
 - Missing in last-level cache can be highly detrimental for performance

Multi-level cache hierarchy

- Processor FSM and logic injects code/data requests to L1 instruction/data cache
- L1 cache hits are returned immediately to the processor; L1 cache misses are forwarded to the L2 cache
- L2 cache hits are returned to the L1 cache, which forwards the requested bytes to the processor and also fills the block in L1 cache
 - Future accesses can be satisfied from the L1 cache until the block is replaced from L1 cache
- L2 cache misses are forwarded to L3 cache or DRAM depending on the number of SRAM cache levels

Multi-level cache hierarchy

- Updated AMAT equation
 - Average memory access time (AMAT) = L1 hit/miss detection time + L1 miss rate x L2 hit/miss detection time + L1 miss rate x L2 miss rate x L3 hit/miss detection time + L1 miss rate x L2 miss rate x L3 miss rate x DRAM fetch latency

Why virtual memory?

- With a 32-bit address you can access 4 GB of physical memory (you will never get the full memory though)
 - Seems enough for most day-to-day applications
 - But there are important applications that have much bigger memory footprint: databases, scientific apps operating on large matrices etc.
 - Even if your application fits entirely in physical memory it seems unfair to load the full image at startup
 - Just takes away memory from other processes, but probably doesn't need the full image at any point of time during execution: hurts multiprogramming
- Need to provide an illusion of bigger memory: Virtual Memory (VM)

Virtual memory

- Need an address to access virtual memory
 - Virtual Address (VA)
- Assume a 32-bit VA
 - Every process sees a 4 GB of virtual memory
 - This is much better than a 4 GB physical memory shared between multiprogrammed processes
 - The size of VA is really fixed by the processor data path width
 - Large virtual and physical memory is very important in commercial server market: need to run large databases

Addressing VM

- There are primarily three ways to address VM
 - Paging, Segmentation, Segmented paging
 - We will focus on flat paging only
- Paged VM
 - The entire VM is divided into small units called **pages**
 - Virtual pages are loaded into **physical page frames** as and when needed (**demand paging**)
 - Thus the physical memory is also divided into equal sized **page frames**
 - The processor generates virtual addresses
 - But memory is physically addressed: need a **VA to PA translation**

VA to PA translation

- The VA generated by the processor is divided into two parts:
 - Page offset and Virtual page number (VPN)
 - Assume a 4 KB page: within a 32-bit VA, lower 12 bits will be page offset (offset within a page) and the remaining 20 bits are VPN (hence 1 M virtual pages total)
 - The page offset remains unchanged in the translation
 - Need to translate VPN to a physical page frame number (PPFN)
 - This translation is held in a **page table** resident in memory: so first we need to access this page table
 - How to get the address of the page table?

VA to PA translation

- Accessing the page table
 - The **Page table base register (PTBR)** contains the starting physical address of the page table
 - PTBR is normally accessible in the kernel mode only
 - Assume each entry in page table is 32 bits (4 bytes)
 - Thus the required page table address is
$$\text{PTBR} + (\text{VPN} \ll 2)$$
 - Access memory at this address to get 32 bits of data from the page table entry (PTE)
 - These 32 bits contain many things: a valid bit, the much needed PPFN (may be 20 bits for a 4 GB physical memory), access permissions (read, write, execute), a dirty/modified bit etc.

Page fault

- The valid bit within the 32 bits tells you if the translation is valid
- If this bit is reset that means the page is not resident in memory: results in a page fault
- In case of a page fault the kernel needs to bring in the page to memory from disk
- The disk address is normally provided by the page table entry (different interpretation of 31 bits)
- Also kernel needs to allocate a new physical page frame for this virtual page
- If all frames are occupied it invokes a page replacement policy

VA to PA translation

- Page faults take a long time: order of ms
 - Need a good page replacement policy
- Once the page fault finishes, the page table entry is updated with the new VPN to PPFN mapping
- Of course, if the valid bit was set, you get the PPFN right away without taking a page fault
- Finally, PPFN is concatenated with the page offset to get the final PA

PPFN	Offset
------	--------
- Processor now can issue a memory request with this PA to get the necessary data
- Really two memory accesses are needed
- Can we improve on this?

TLB

- Why can't we cache the most recently used translations?
 - Translation Look-aside Buffers (TLB)
 - Small set of registers (normally fully associative)
 - Each entry has two parts: the tag which is simply VPN and the corresponding PTE
 - The tag may also contain a process id
 - On a TLB hit you just get the translation in one cycle (may take slightly longer depending on the design)
 - On a TLB miss you may need to access memory to load the PTE in TLB (heavily optimized today)
 - Normally there are two TLBs: instruction and data

Memory op latency

- L1 hit: ~ 1 ns
- L2 hit: ~ 5 ns
- L3 hit: ~ 10 - 15 ns
- Main memory: ~ 70 ns DRAM access time + bus transfer etc. = ~ 110 - 120 ns
- If a load misses in all caches it will eventually come to the head of the ROB and block instruction retirement (in-order retirement is a must)
- Gradually, the pipeline backs up, processor runs out of resources such as ROB entries and physical registers
- Ultimately, the fetcher stalls: **severely limits ILP**

Agenda

➤ Basics of computer architecture

- Basics of the basics
- Instruction set architecture (ISA)
- Processor design
- Caches and virtual memory

➤ Communicating with environment

- Performance measurement
- Performance optimization
- Multi-core processors

• Basics of operating systems

I/O devices

- A computer that can only execute instructions for computing and accessing memory is not very useful
 - There is no way for the environment to give inputs to the computer or examine outputs of computation
- A typical computer interfaces with a large number of I/O devices
 - Keyboard, display, mouse, speaker, microphone, hard disk, printer, USB devices, etc.
 - Need a mechanism to communicate with these devices
 - How to read a key punch or how to display

I/O devices

- A typical I/O device has a set of command registers and a set of data registers
 - These are assigned unique addresses and can be read or written to through load or store instructions
 - Known as memory-mapped I/O registers
 - To the computer, these appear as memory locations
 - The only difference is that they are not in DRAM, but in I/O devices
 - For example, printing something on the printer involves storing the data to be printed to the printer data registers and an appropriate command to the printer command registers

I/O devices

- Certain I/O commands send responses back to the computer
 - A keyboard read needs to be conveyed to the computer
 - Completion of a disk read needs to be conveyed to the computer
 - Any read operation must be communicated to the computer
 - To detect completion of a disk or keyboard read, one possibility is to continuously poll a register of the disk controller or the keyboard controller
 - Wastes computer's time (computer could do something else during this time)

I/O devices

- Polling works only if the computer is aware that it will receive some response from a certain I/O device
 - Certain responses are accidental (e.g., ctrl+C to terminate a program or a mouse click)
 - In such cases, the computer did not know beforehand and was not polling the memory-mapped register
- An efficient solution that covers all cases is implemented using interrupts
 - These are signals sent by the I/O devices to the computer
 - For example, a key punch generates an interrupt

Interrupts

- Interrupts stop the normal instruction processing of a computer and make it execute an interrupt handler function
 - Interrupts can be generated by hardware (e.g., I/O devices) or software (e.g., exceptions and system calls)
 - There are two ways to implement interrupt or exception handlers
 - Vectored interrupts or exceptions
 - Cause-based interrupt handling (non-vectored)

Direct memory access (DMA)

- Copying large amounts of data from input devices to memory may take a significant amount of time
 - Reading files from disk
- Same applies to copying from memory to output devices
 - Write to files on disk
- Occupying the computer during this time wastes computer's resources
 - Could compute something useful during this time
- Direct memory access (DMA) frees up the computer during data copying from/to I/O

Direct memory access (DMA)

- Computer initializes a specialized hardware called DMA controller by setting up the copy address range and the number of bytes to be copied
- DMA controller does the actual copy operation
 - DMA controller when copying from an input device sends the data to the DRAM controller for writing
 - DMA controller when copying to an output device sends read requests to the DRAM controller
 - When the copying completes, the DMA controller sends an interrupt to the computer to notify about the DMA completion

Exceptions

- Exceptions refer to situations where the running program exhibits unexpected behavior
 - Arithmetic overflow, divide by zero, fetching and decoding an illegal opcode, accessing an illegitimate address
 - In such situations, the PC of the offending instruction is saved in a register called exception PC (EPC) and the program counter is changed to point to a location that has a “trap” instruction
 - The trap instruction allows the computer to enter the operating system which further invokes the appropriate exception handler after examining the cause register

Exceptions

- Some exceptions are restartable while some exceptions are not
 - Restartable exceptions return to normal execution of the program after the exception handler completes
 - Example: non-availability of code/data in memory (causes a page fault exception)
 - Non-restartable exceptions typically lead to termination of the running program with an appropriate message printed on display
 - Arithmetic exceptions, illegal opcode, crossing legitimate memory boundary, etc.

System calls

- System calls are pseudo-function calls to request access to certain hardware/software resources of the computer system
 - Reading from a file on disk, reading from keyboard, writing to display, writing to a file, allocating dynamic memory, etc. involve system calls
 - Some computers refer to system calls as software interrupts
 - MIPS ISA has the syscall instruction for this purpose
 - Before invoking the syscall instruction, few registers need to be set up with appropriate information

syscall instruction

- Every system call has a number to indicate the purpose of the system call
 - Reading from a file, writing to a file, allocating dynamic memory all have different system call numbers

Agenda

➤ Basics of computer architecture

- Basics of the basics
- Instruction set architecture (ISA)
- Processor design
- Caches and virtual memory
- Communicating with environment

➤ Performance measurement

- Performance optimization
- Multi-core processors

• Basics of operating systems

Why measure performance?

- Primarily two reasons
 - To measure how good a computer is
 - To understand why a computer is or is not performing as expected
- Performance of a computer is a function of the performance of individual programs that run on the computer
- A program's performance is best measured as the reciprocal of its execution time
- Equivalently, two computers' performance on the same program can be compared by looking at the reciprocal of respective execution times

CPI equation

- How do we measure execution time? Which are the determinant factors?
- Assume that we want to calculate the execution time of a program
 - Execution time = Clock cycles to execute the program \times clock cycle time
 - Executed clock cycles = number of executed instructions \times average cycles per instruction
 - Execution time = instruction count \times CPI \times cycle time
 - Cycle time is also same as reciprocal of frequency (in appropriate unit)
 - Execution time equally depends on three components

Amdahl's law

- Look for portions of program that takes large amount of time to execute
 - Allocate resources and design time proportionate to execution time
 - As x increases the achieved speedup goes up for a fixed y ; as y increases, speedup remains limited by x
 - Amdahl's law is usually used to compare design alternatives i.e. which design would bring more performance

Amdahl's law

- Amdahl's law can be used to derive upper bound on achievable speedup in a parallel computer
 - Suppose a sequential program takes time t to run on a single processor
 - A fraction s of this time is spent in executing inherently sequential portions of the program
 - The remaining time can be perfectly parallelized on arbitrary number of processors
 - Maximum achievable speedup = $t / (s*t + (1 - s)*t / P)$ which is $1 / (s + (1 - s) / P)$ on P processors
 - In the limit, speedup gets capped at $1 / s$
 - Even if s is 0.05, speedup cannot be more than 20
 - To get very large speedup, s should be tiny

Benchmarks

- Want to compare two processors by measuring their performance
 - Need some standardized set of programs
 - These are called benchmark programs
 - Different types of computers have different focus: needs different set of benchmarks
 - Performance metric for desktop/workstation is execution time (also known as response time)
 - Performance metric for servers is throughput (number of jobs done per unit time with a limit on response time per job)
 - Performance metric for embedded processors is also execution time with an emphasis on deadline and power consumption

Agenda

➤ Basics of computer architecture

- Basics of the basics
- Instruction set architecture (ISA)
- Processor design
- Caches and virtual memory
- Communicating with environment
- Performance measurement

➤ Performance optimization

- Multi-core processors

• Basics of operating systems

Moore's law

- Number of transistors in a given area doubles every 18 to 24 months
 - Made possible by improved technology enabling shrinking of transistors
 - More transistors means more smartness in the chip and hence, more performance
 - Could design more sophisticated ALUs, for example
 - Moore's law has played a vital role in performance improvement of computers
 - Negative side: switching more transistors every clock cycle increases power consumption leading to more heating (needs sophisticated cooling solutions)
 - Moore's law is said to have "slowed down" these days because of this reason

Out-of-order Execution

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Out-of-order Execution

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Out-of-order Execution

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Out-of-order Execution

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Out-of-order Execution

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Out-of-order Execution

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Out-of-order Execution

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Out-of-order Execution

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Out-of-order Execution

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20

Results must become visible in-order



Multiple Issue

```
load r2, addr  
add r3, r2, r1  
xor r10, r5, r3  
sub r9, r10, r1  
addi r29, r29, 0xffff  
sll r29, r29, 2  
mul r20, r20
```



Multiple Issue

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Multiple Issue

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Multiple Issue

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Multiple Issue

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20



Multiple Issue

Cache miss

load r2, addr

add r3, r2, r1

xor r10, r5, r3

sub r9, r10, r1

addi r29, r29, 0xffff

sll r29, r29, 2

mul r20, r20

Results must become visible in-order



Out-of-order Multiple Issue

- Some hardware nightmares
 - Complex issue logic to discover independent instructions
 - Increased pressure on cache
 - Impact of a cache miss is much bigger now in terms of lost opportunity
 - Various speculative techniques are in place to “ignore” the slow and stupid memory
 - Increased impact of control dependence
 - Must feed the processor with multiple correct instructions every cycle
 - One cycle of bubble means lost opportunity of multiple instructions
 - Complex logic to verify



MLP

- Need memory-level parallelism (MLP)
 - Simply speaking, need to mutually overlap several memory operations
- Step 1: Non-blocking cache
 - Allow multiple outstanding cache misses
 - Mutually overlap multiple cache misses
 - Supported by all microprocessors today
- Step 2: Out-of-order load issue
 - Issue loads out of program order (address is not known at the time of issue)
 - How do you know the load didn't issue before a store to the same address? Issuing stores must check for this memory-order violation

Out-of-order loads

sw 0(r7), r6

... /* other instructions */

lw r2, 80(r20)

- Assume that the load issues before the store because r20 gets ready before r6 or r7
- The load accesses the store buffer (used for holding already executed store values before they are committed to the cache at retirement)
- If it misses in the store buffer it looks up the caches and, say, gets the value somewhere
- After several cycles the store issues and it turns out that 0(r7) == 80(r20) or they overlap; now what?

Load/store ordering

- Out-of-order load issue relies on **speculative memory disambiguation**
 - Assumes that there will be no conflicting store
 - If the speculation is correct, you have issued the load much earlier and you have allowed the dependents to also execute much earlier
 - If there is a conflicting store, you have to squash the load and all the dependents that have consumed the load value and re-execute them systematically
 - Turns out that the speculation is correct most of the time
 - To further minimize the load squash, microprocessors use simple memory dependence predictors (predicts if a load is going to conflict with a pending store based on that load's or load/store pairs' past behavior)

MLP and memory wall

- Today microprocessors try to hide cache misses by initiating early prefetches:
 - Hardware prefetchers try to predict next several load addresses and initiate cache line prefetch if they are not already in the cache
 - All processors today also support prefetch instructions; so you can specify in your program when to prefetch what: this gives much better control compared to a hardware prefetcher
- Researchers have worked on load value prediction
- Even after doing all these, memory latency remains the biggest bottleneck
- Today microprocessors are trying to overcome one single wall: the **memory wall**

Agenda

➤ Basics of computer architecture

- Basics of the basics
- Instruction set architecture (ISA)
- Processor design
- Caches and virtual memory
- Communicating with environment
- Performance measurement
- Performance optimization

➤ Multi-core processors

- Basics of operating systems

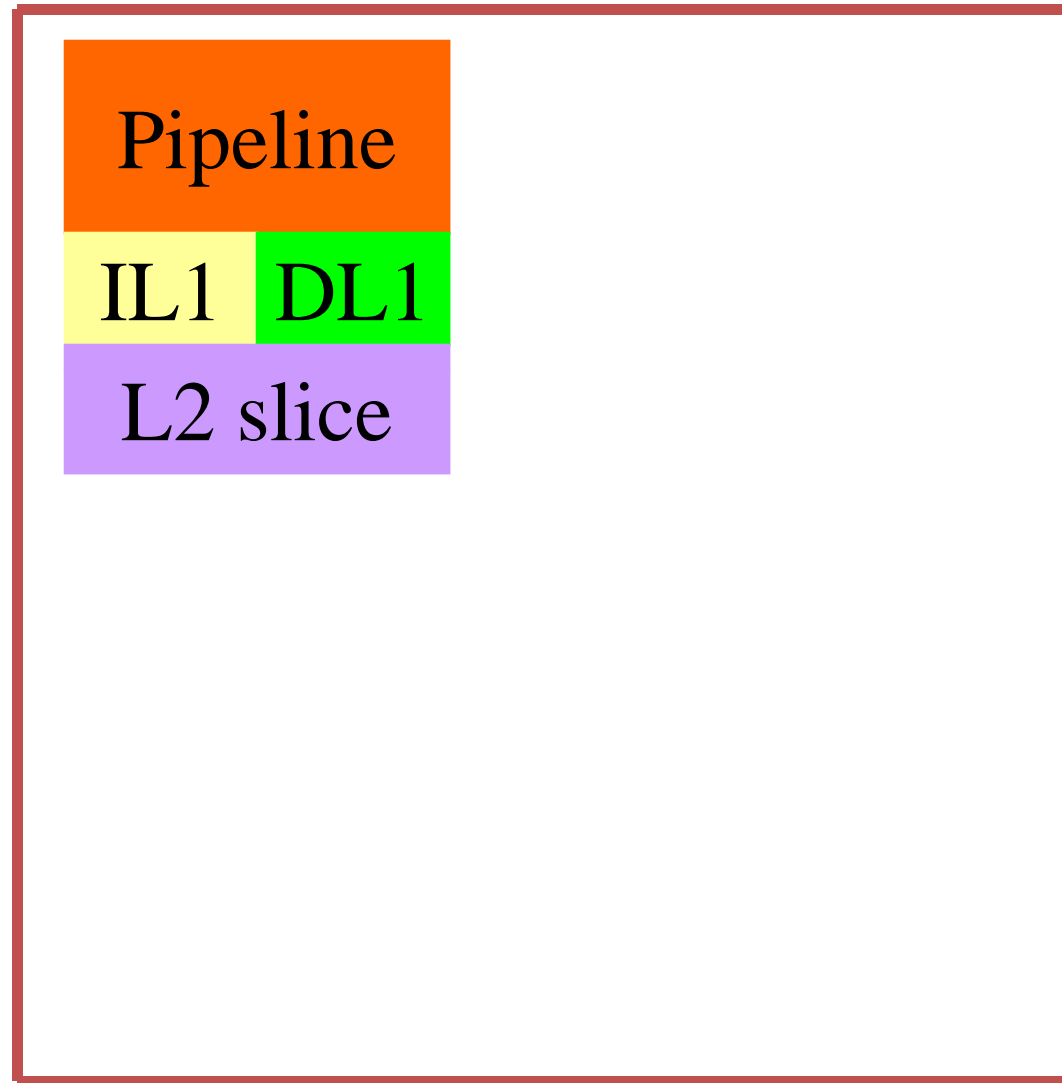
Multi-core

- Put a few reasonably complex processors or many simple processors on the chip
 - Each processor has its own primary cache and pipeline
 - Often a processor is called a core
 - Often called a chip-multiprocessor (CMP)
- Did we use the transistors properly?
 - Depends on if you can keep the cores busy
 - Introduces the concept of thread-level parallelism (TLP)

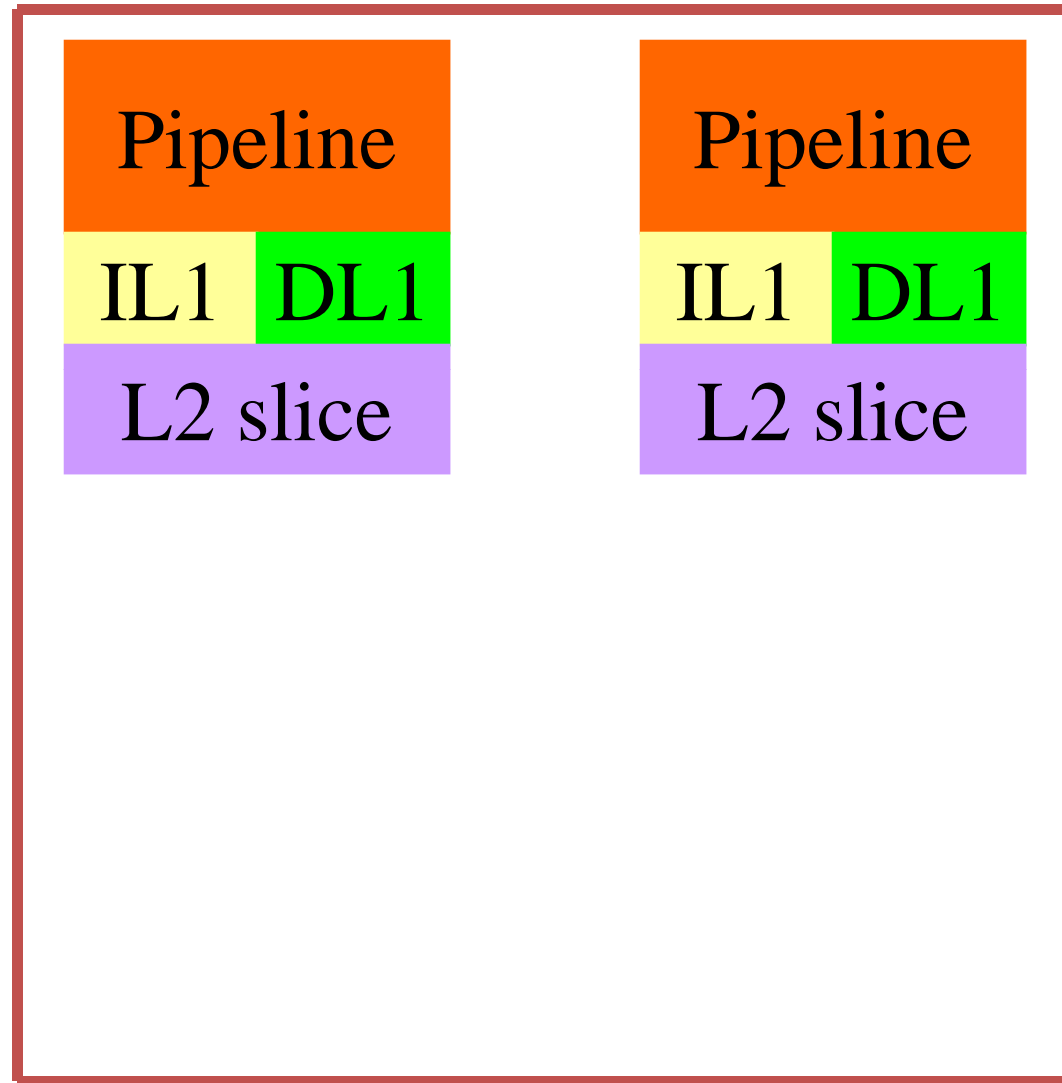
Tiled CMP (Hypothetical Floor-plan)



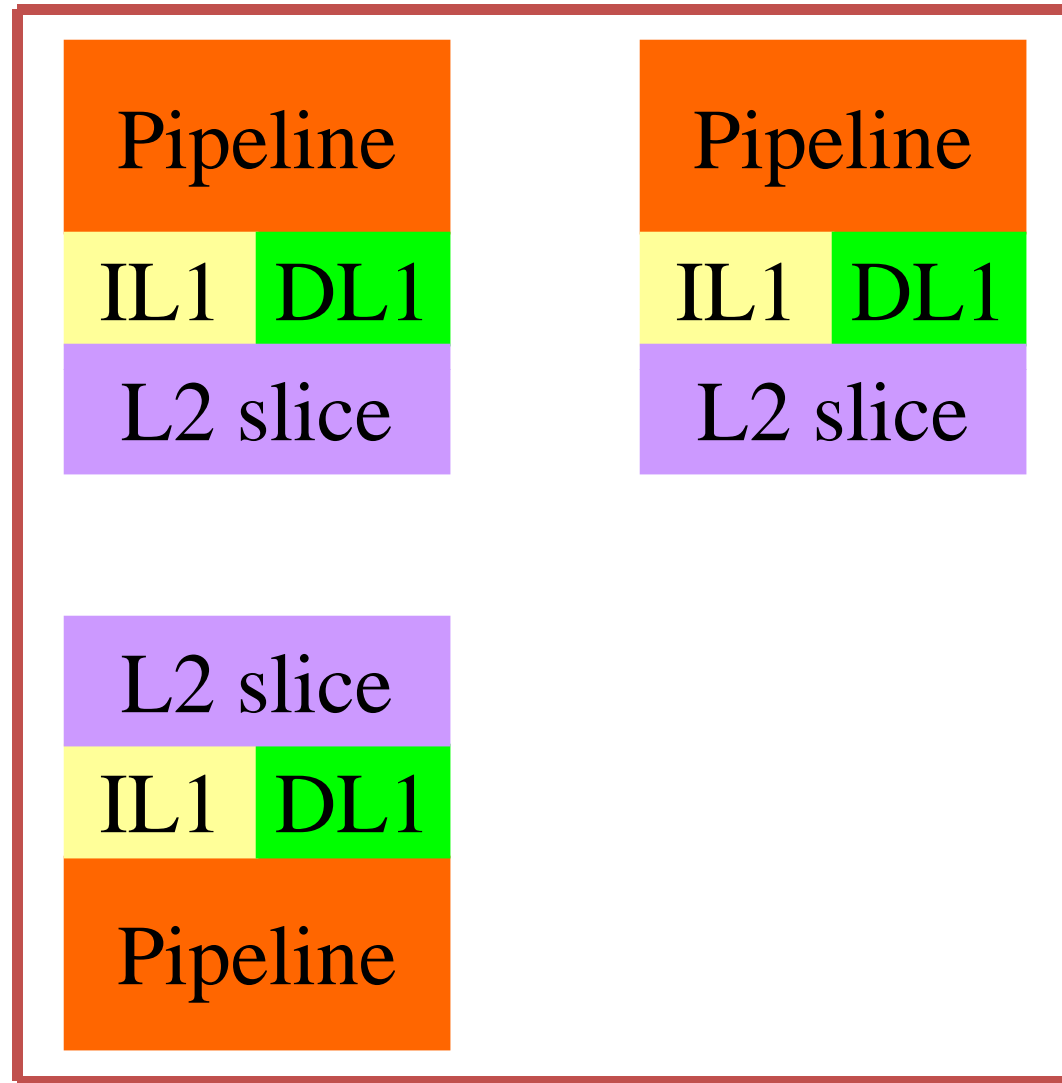
Tiled CMP (Hypothetical Floor-plan)



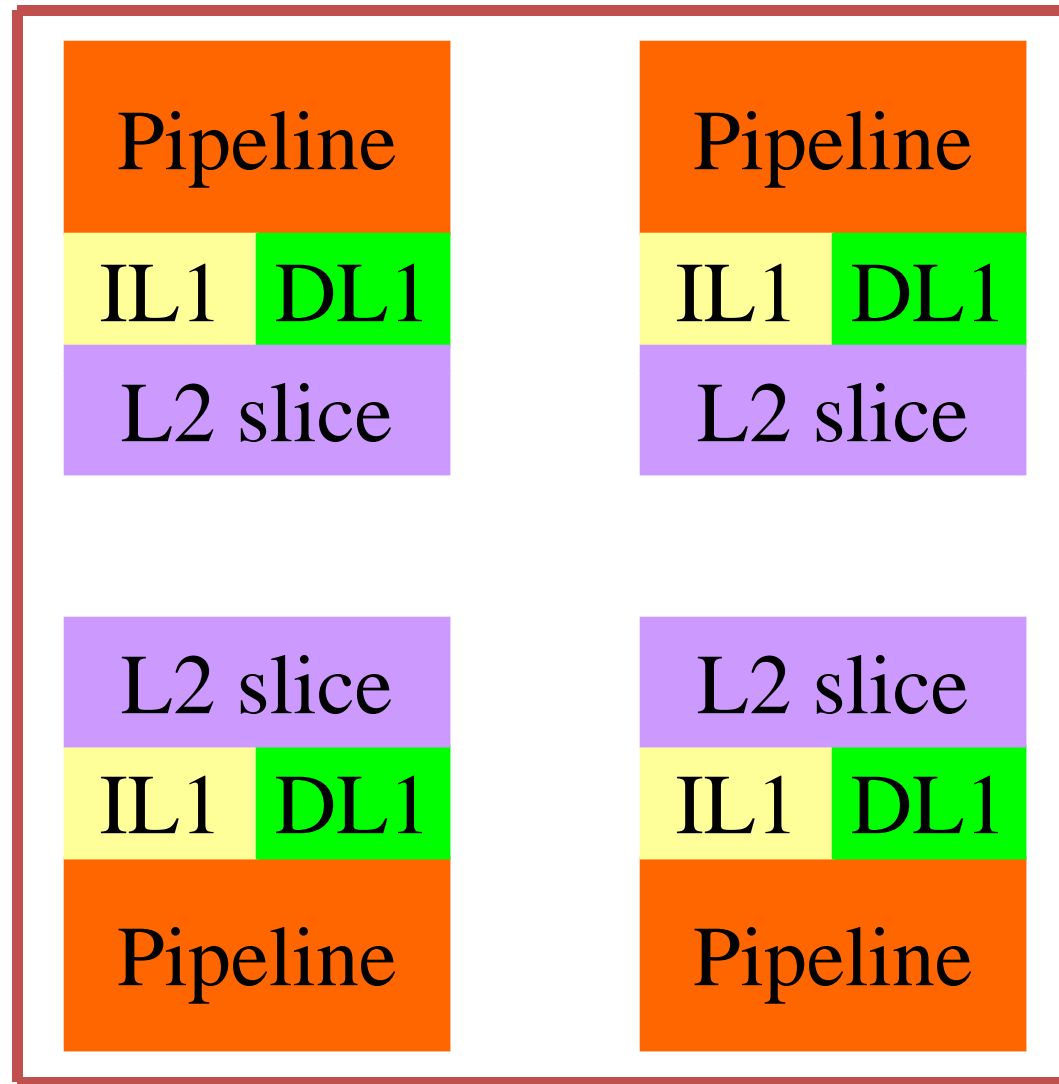
Tiled CMP (Hypothetical Floor-plan)



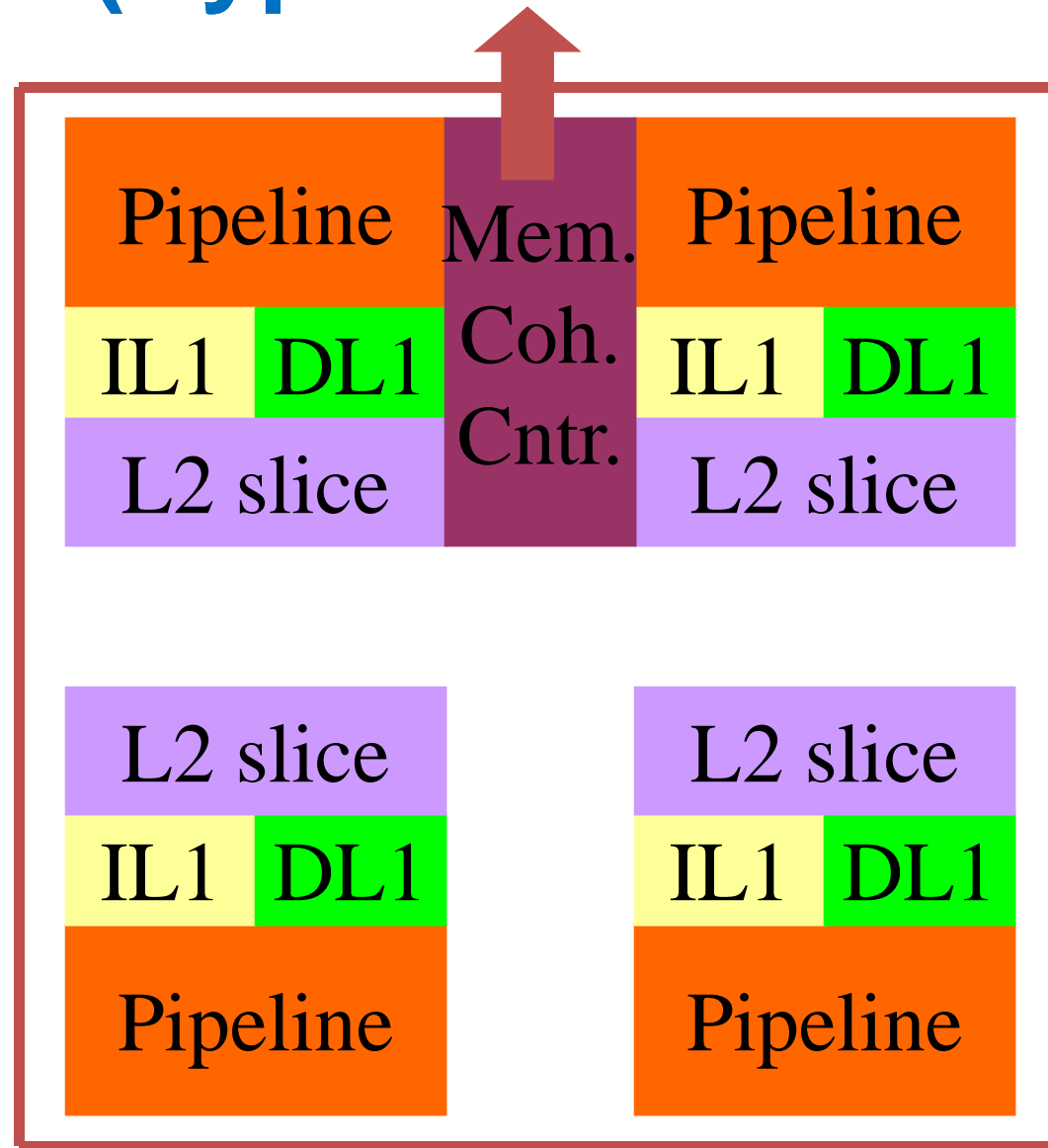
Tiled CMP (Hypothetical Floor-plan)



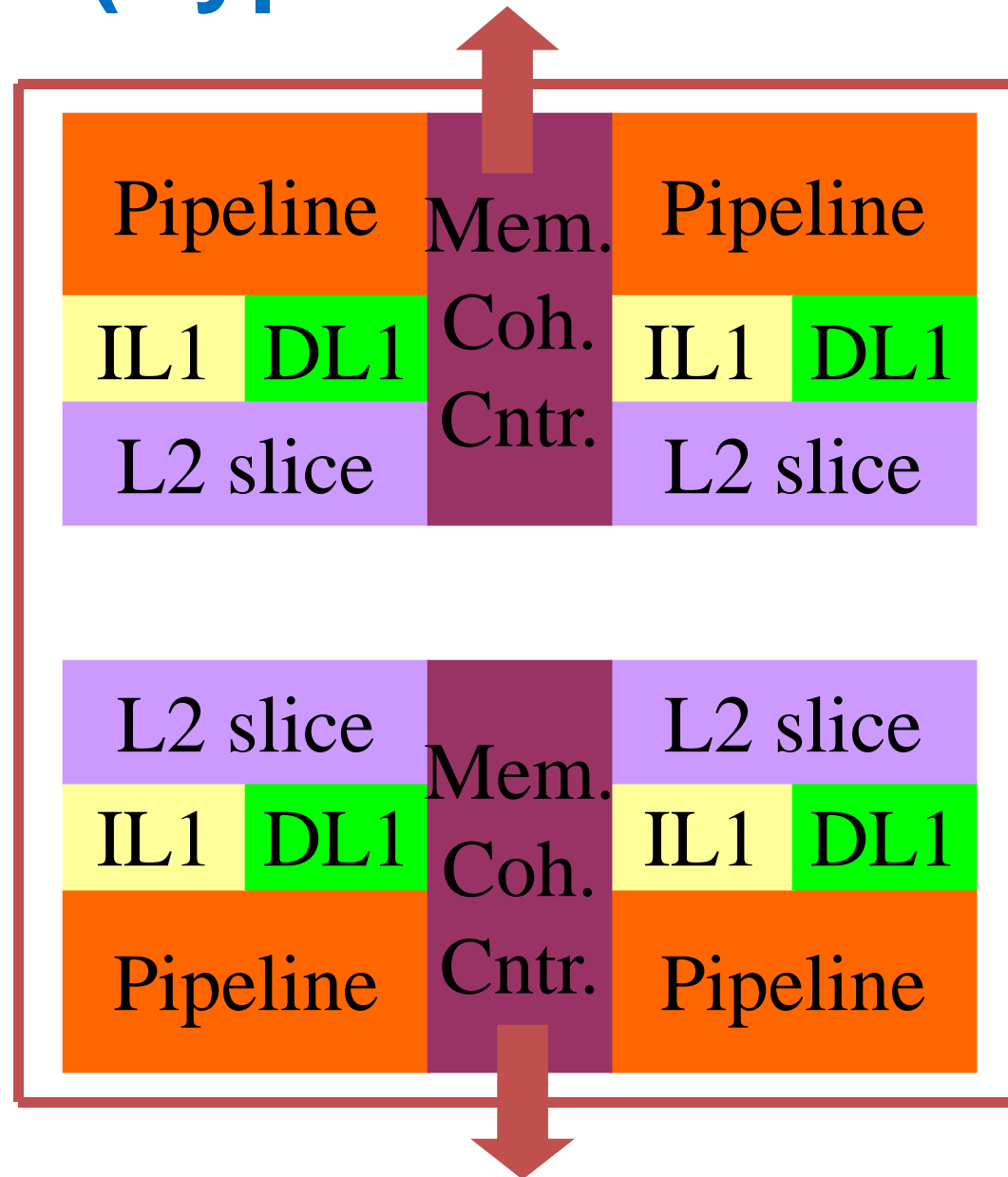
Tiled CMP (Hypothetical Floor-plan)



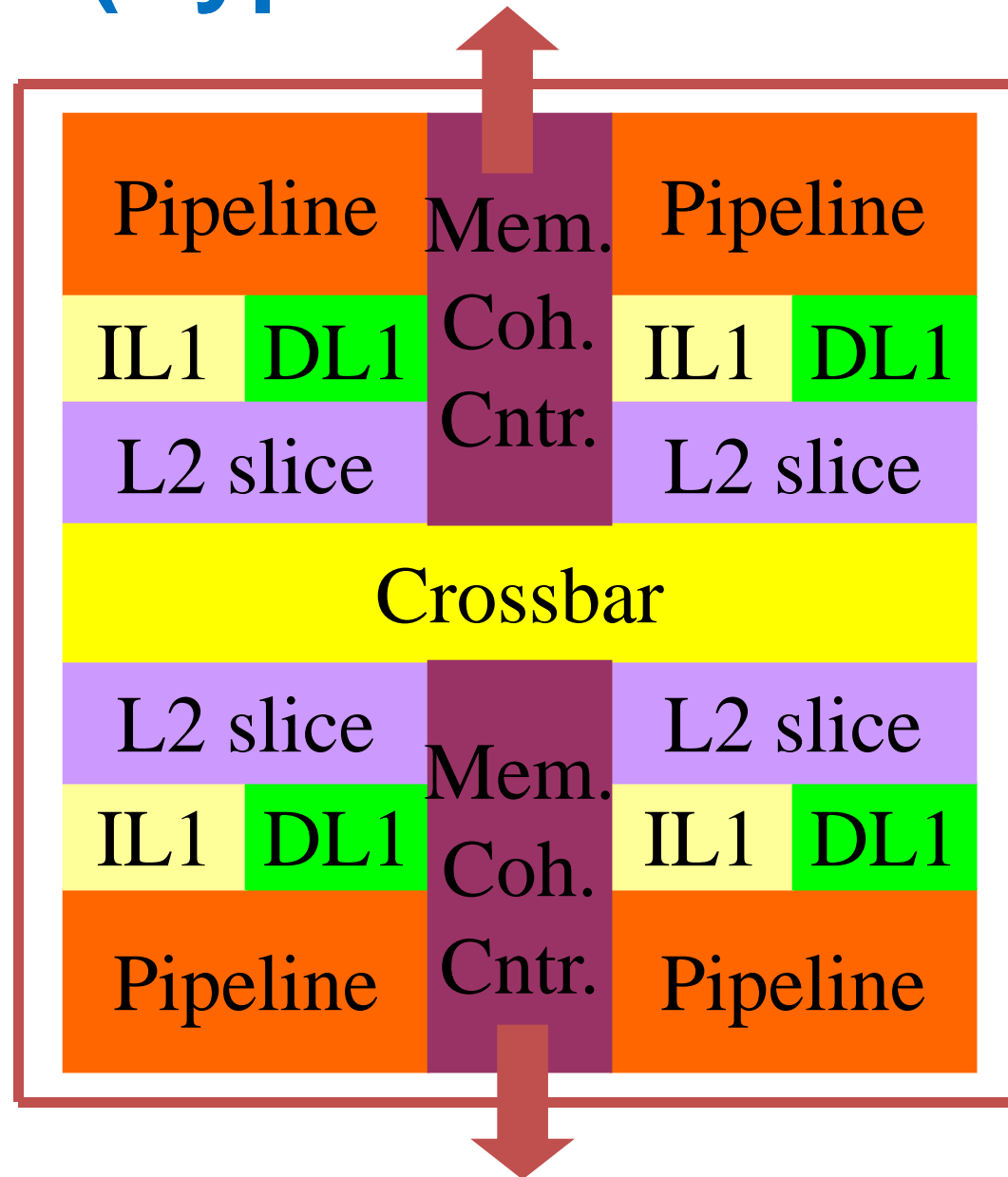
Tiled CMP (Hypothetical Floor-plan)



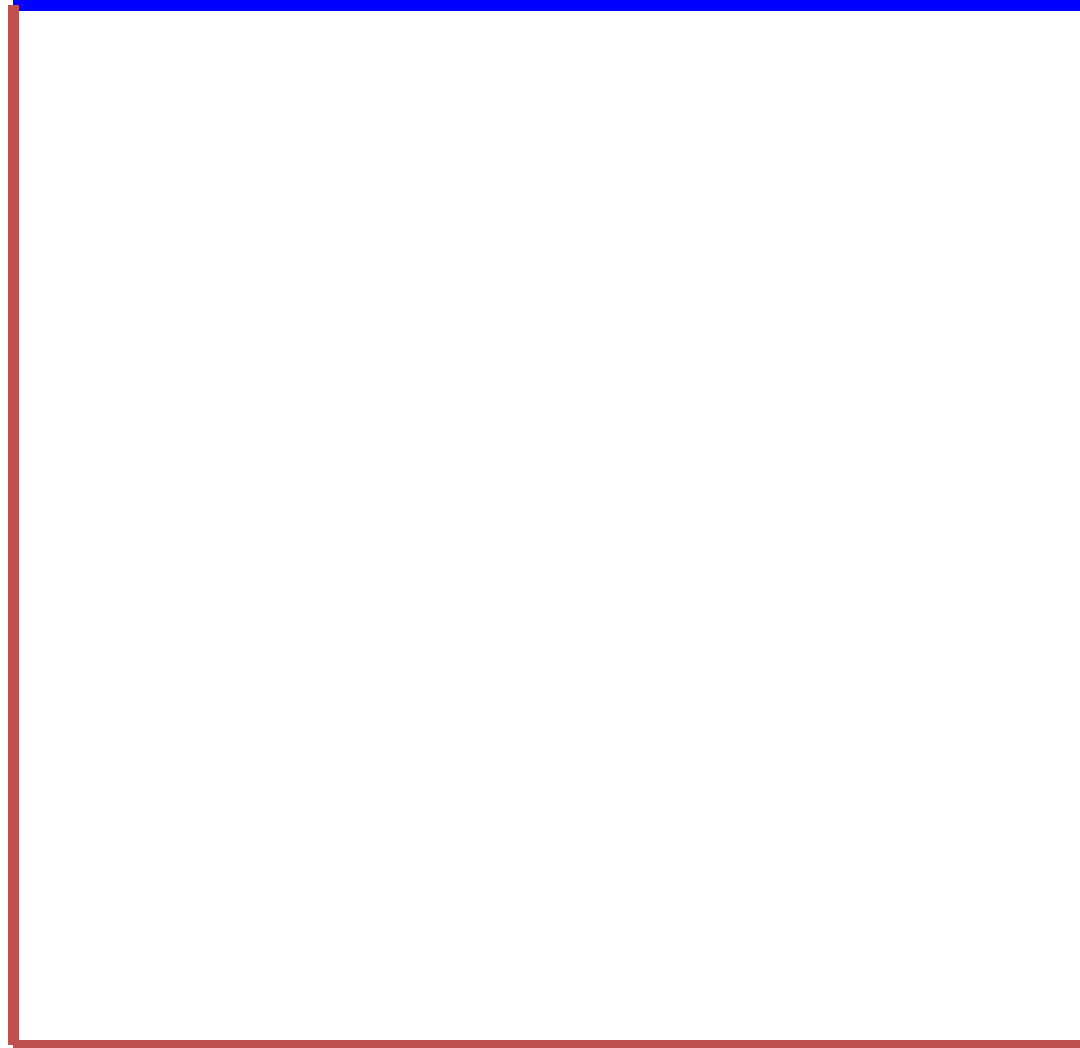
Tiled CMP (Hypothetical Floor-plan)



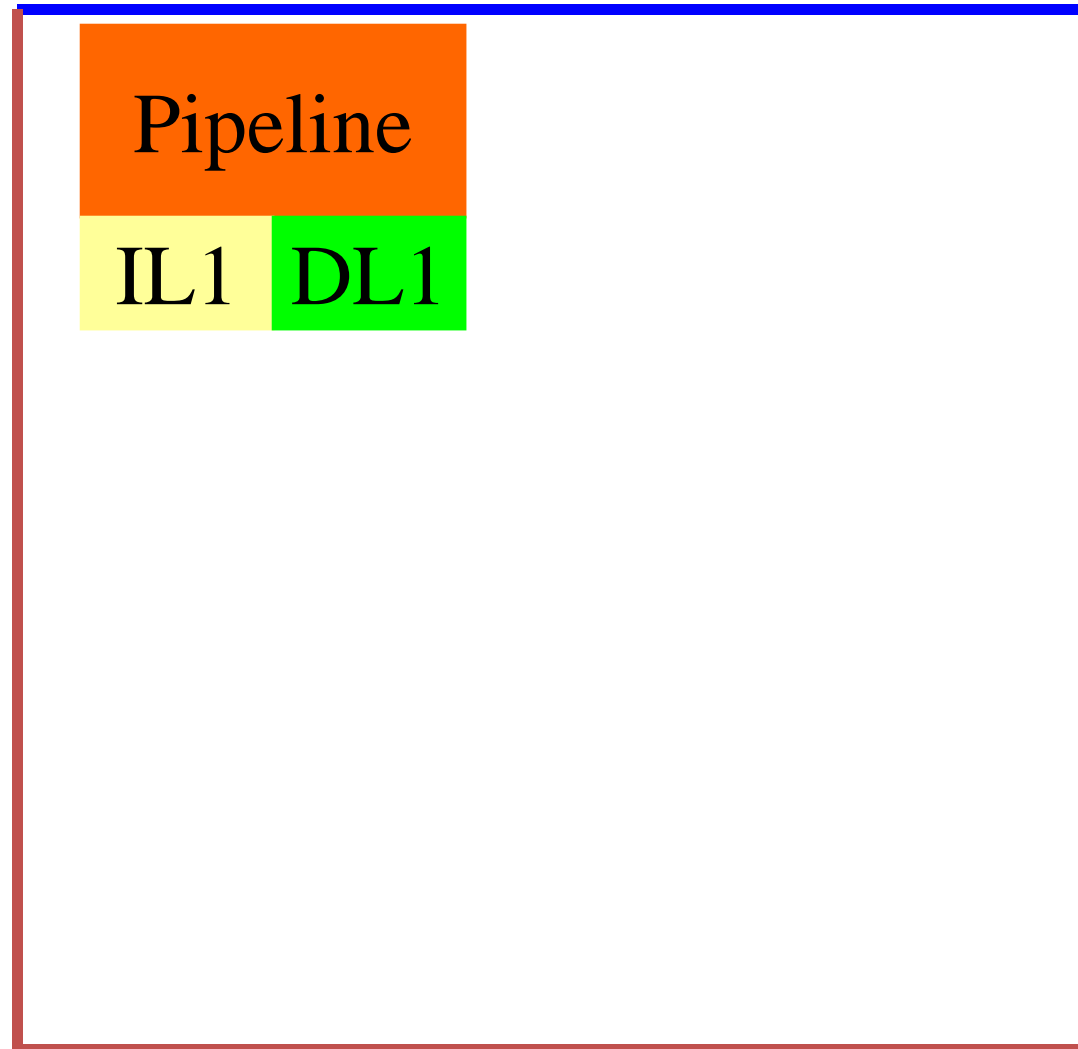
Tiled CMP (Hypothetical Floor-plan)



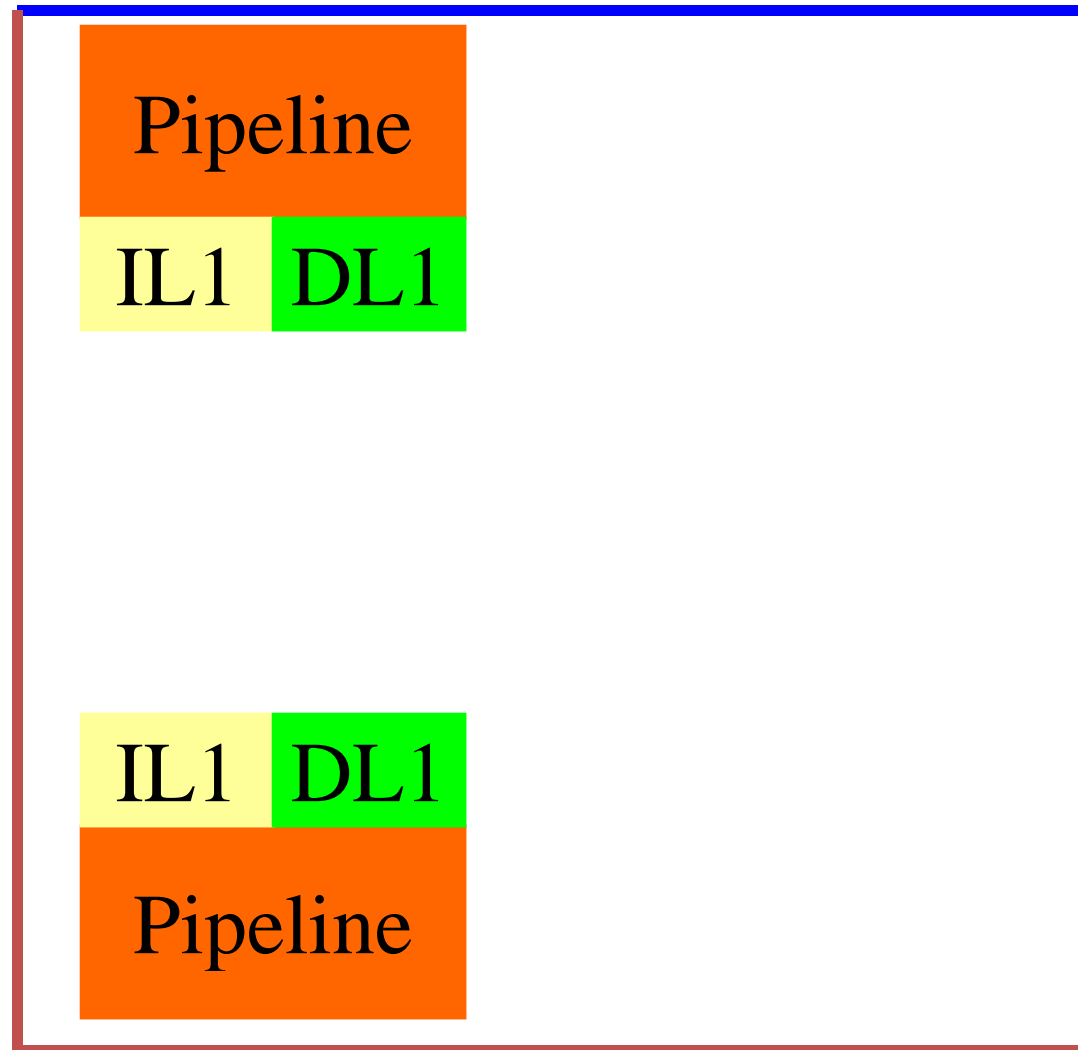
Shared Cache CMP



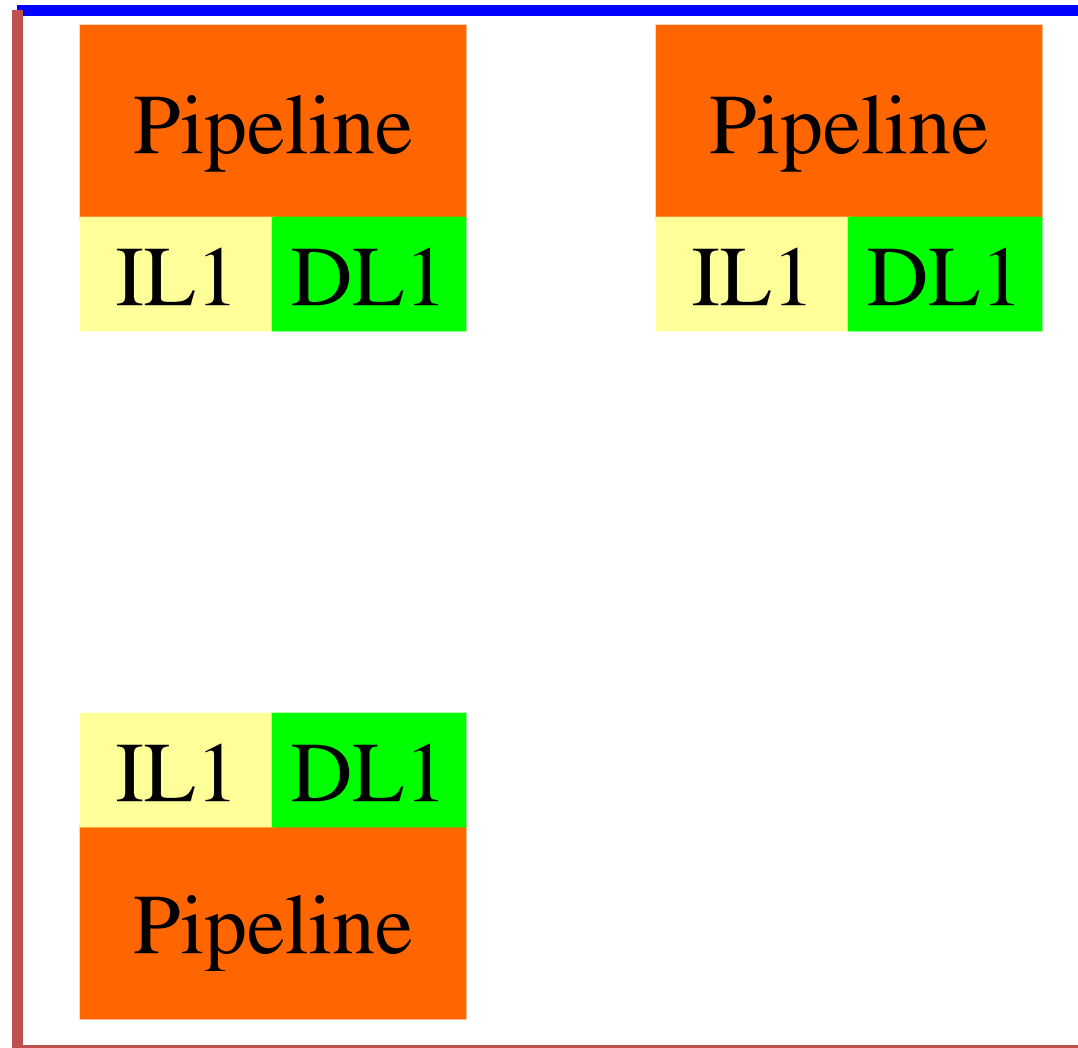
Shared Cache CMP



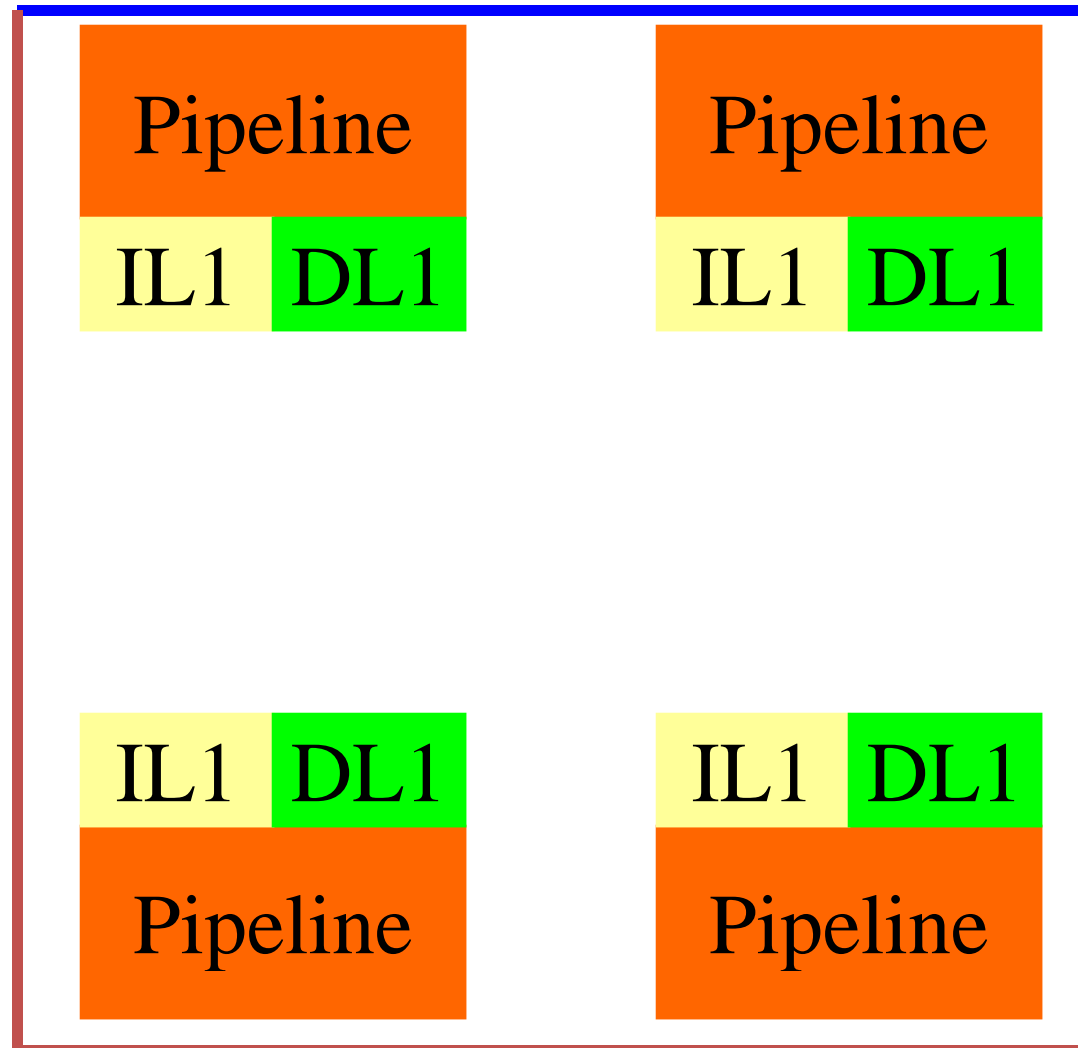
Shared Cache CMP



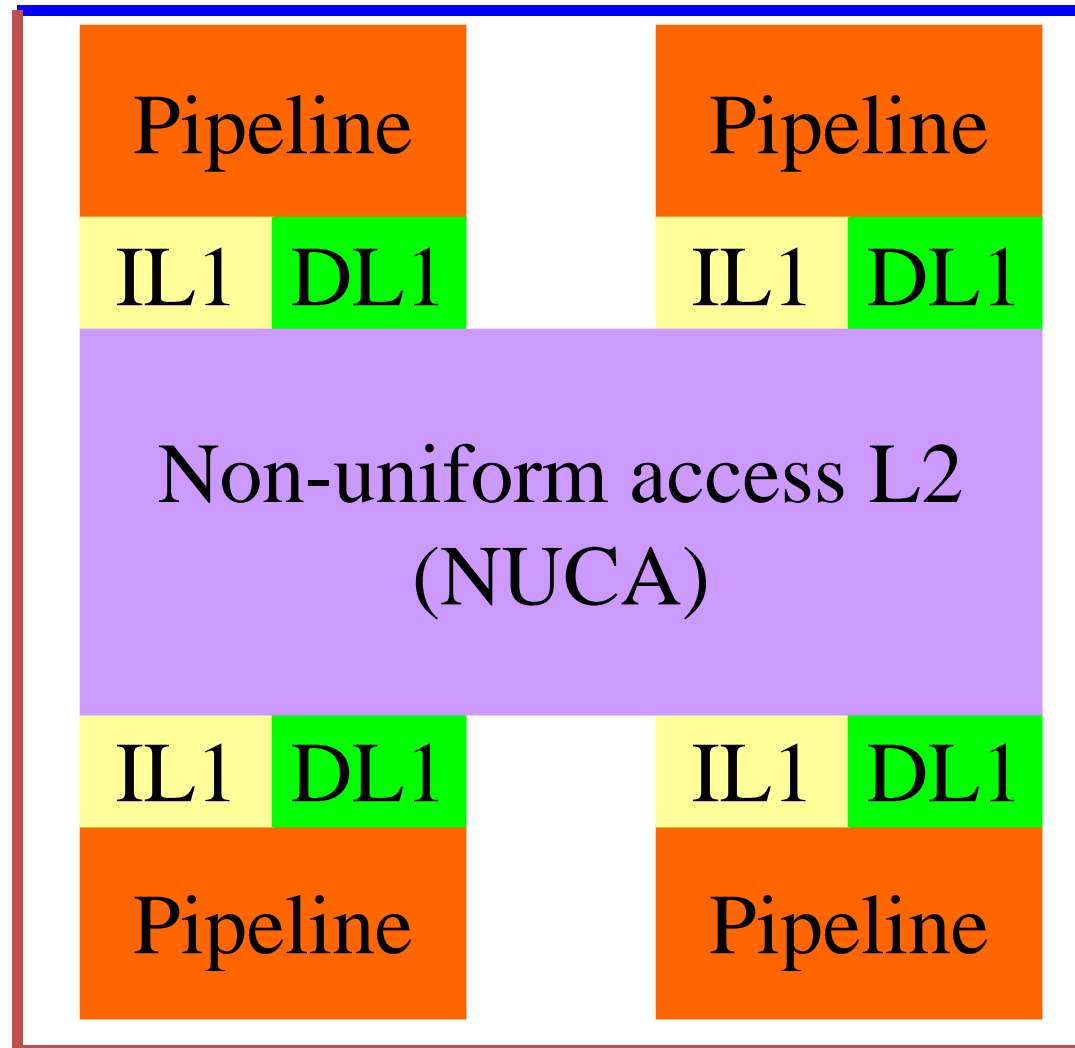
Shared Cache CMP



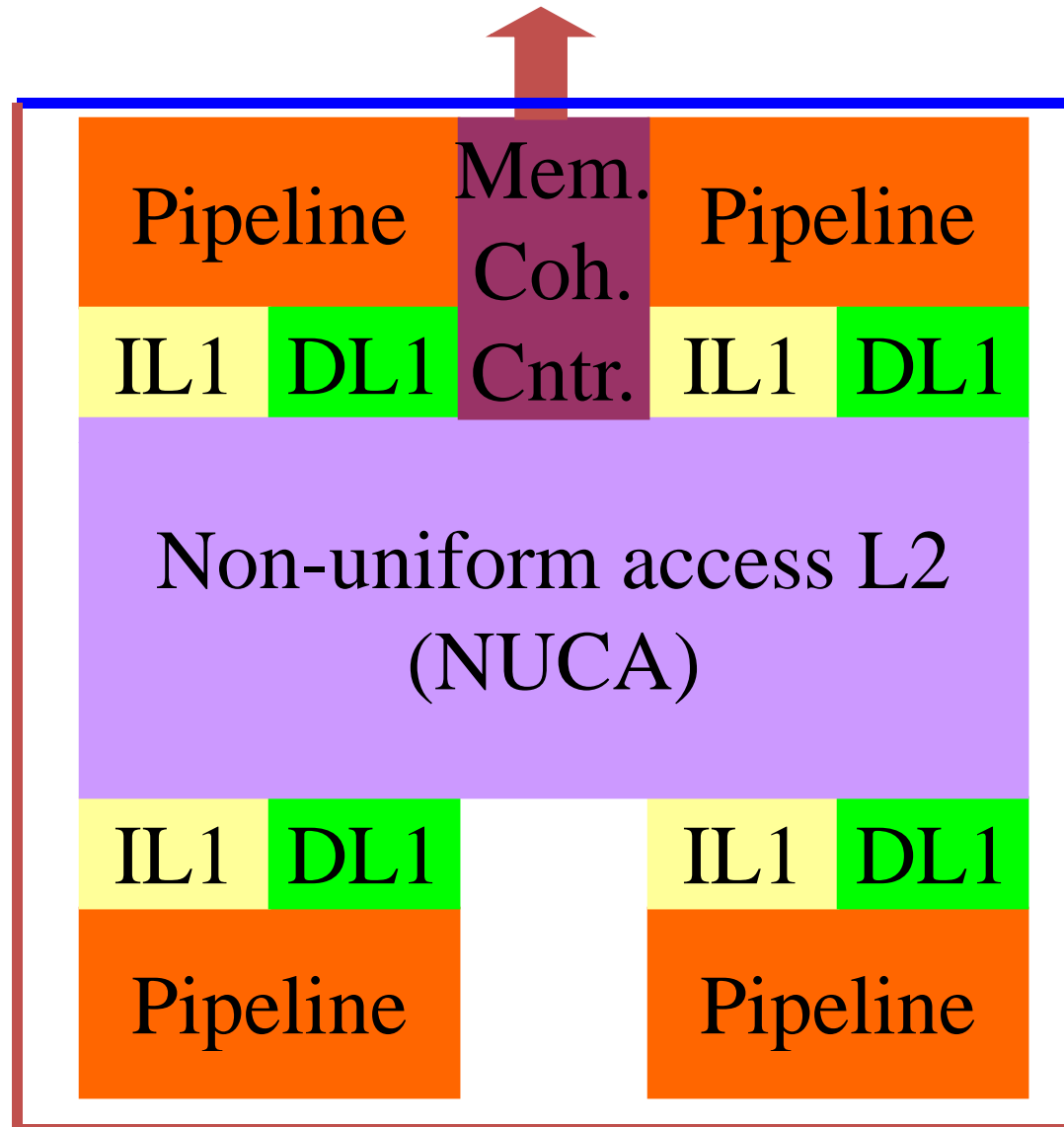
Shared Cache CMP



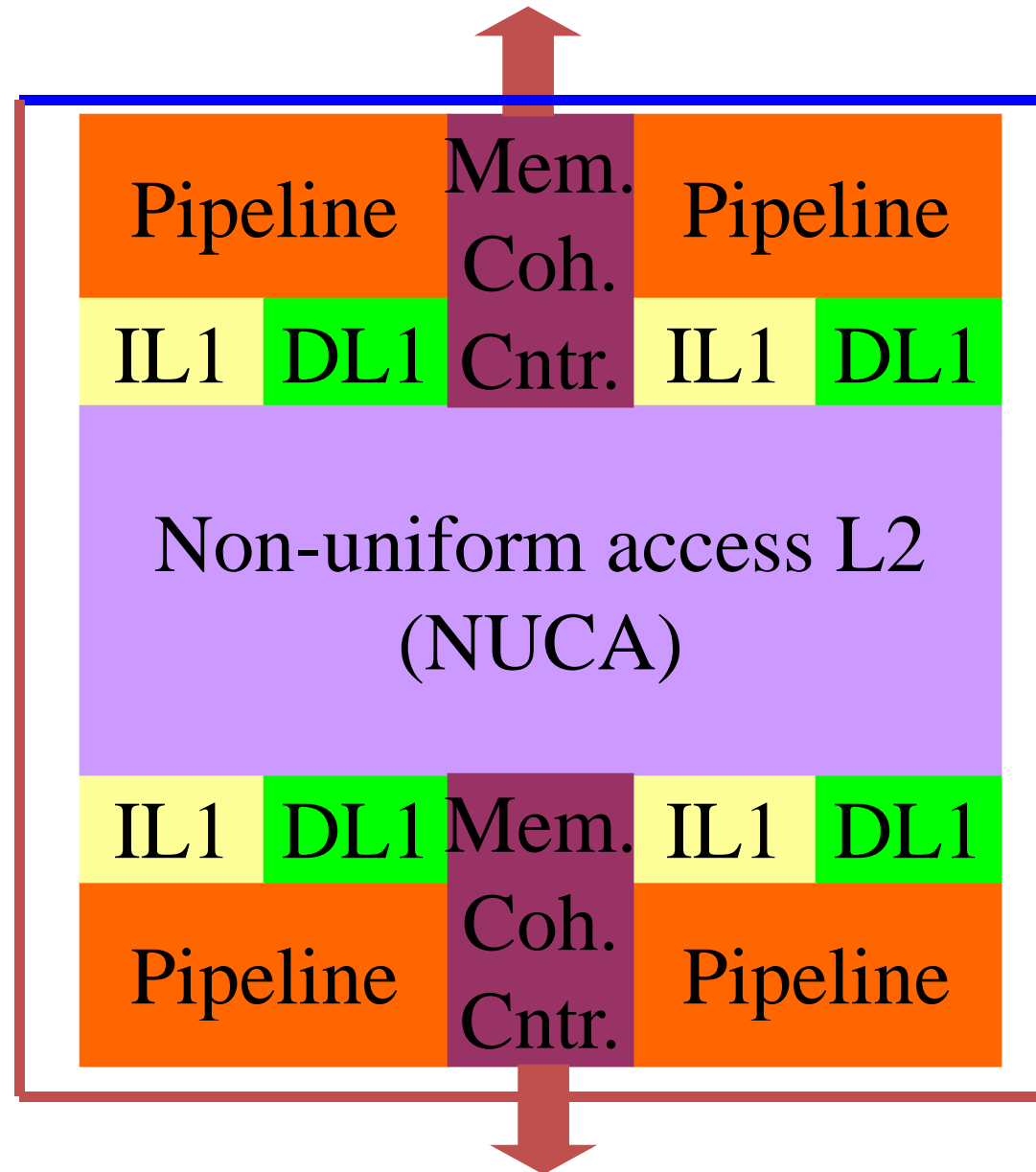
Shared Cache CMP



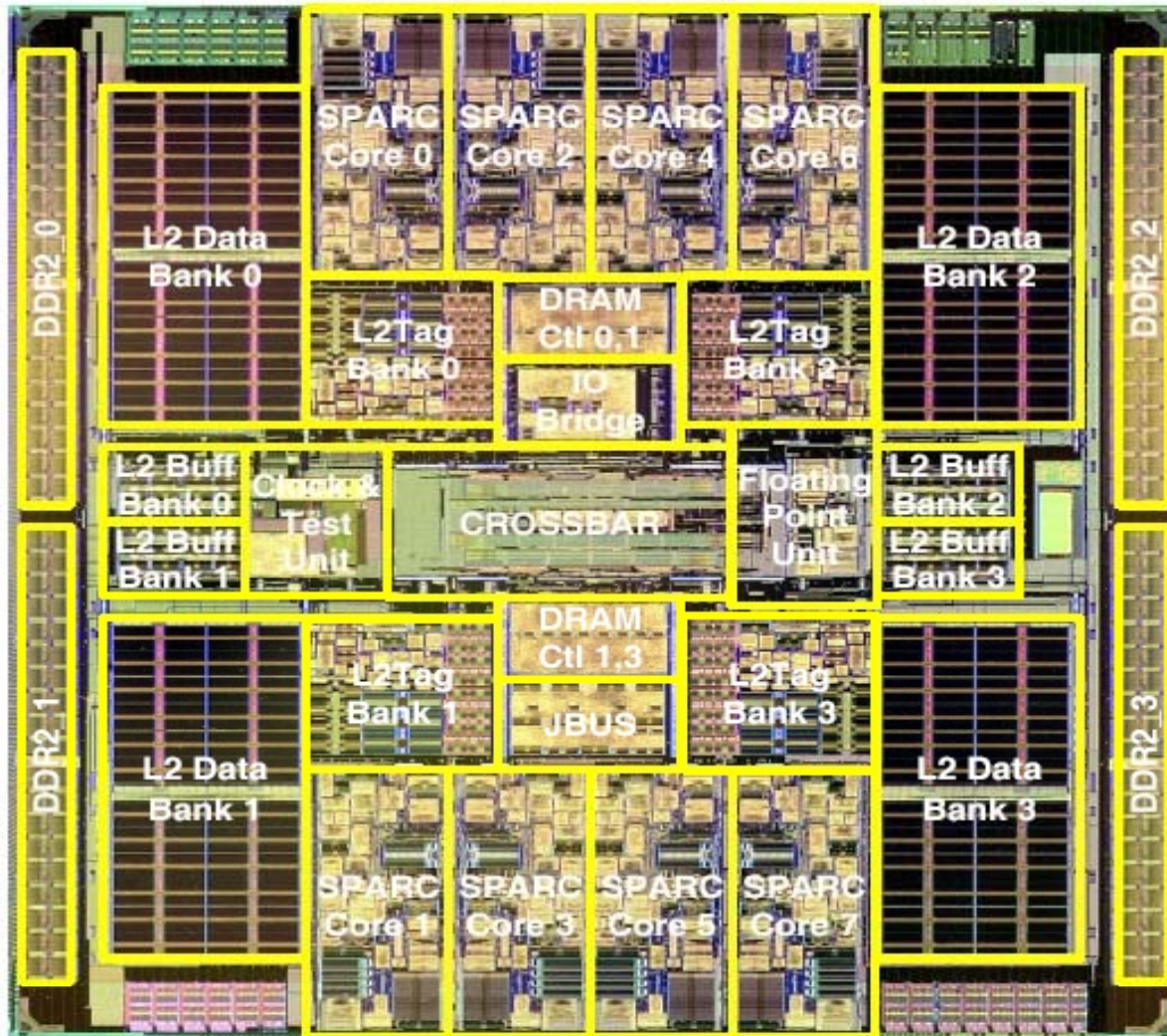
Shared Cache CMP



Shared Cache CMP



Niagara Floor-plan



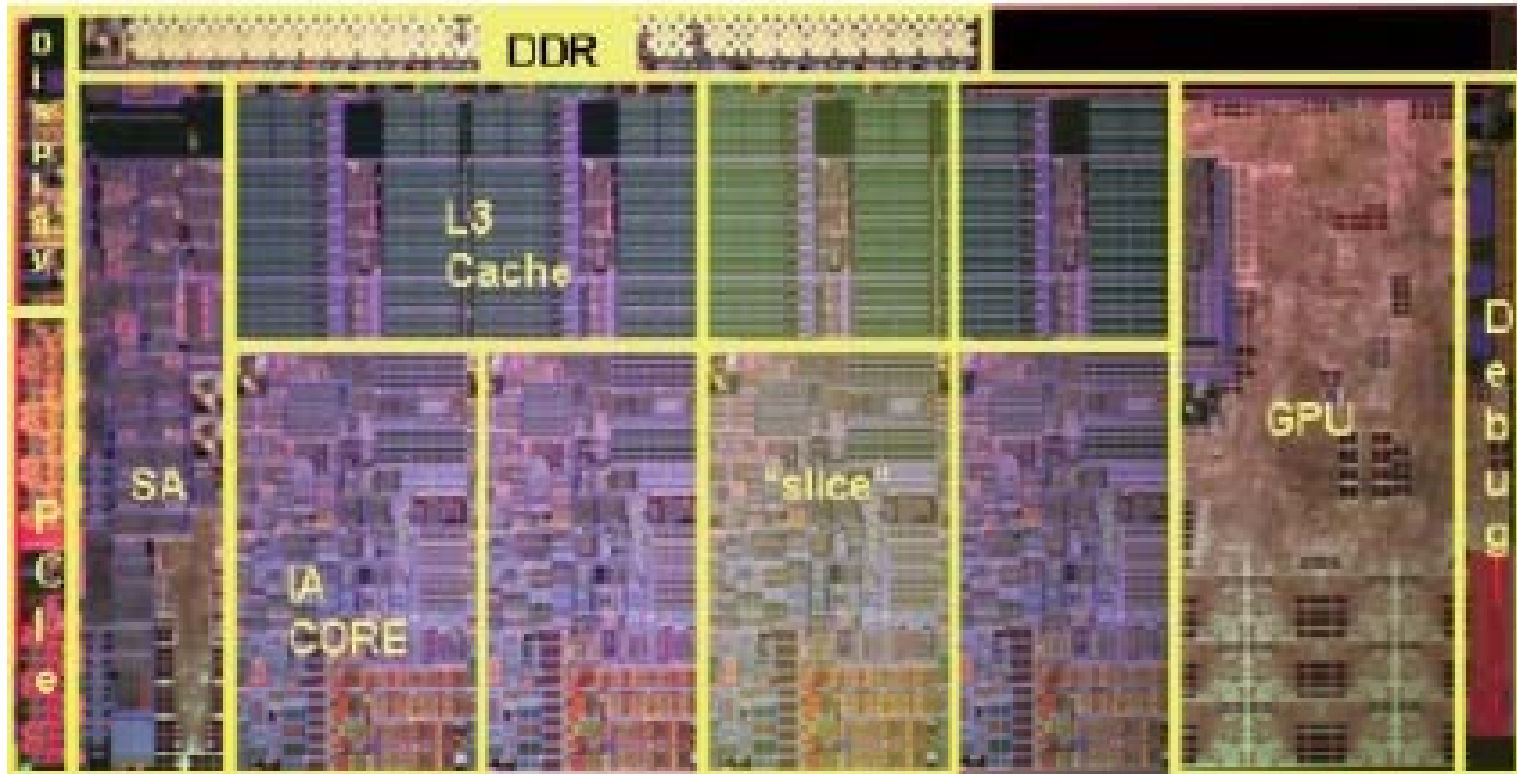
Features:

- Eight 64b Multithreaded SPARC Cores
- Shared 3MB L2 Cache
- 16KB ICache per Core
- 8KB DCache per Core
- Four 144b DDR-2 DRAM Interfaces (400 MTs)
- 3.2GB/s JBUS I/O
- Crypto: Public Key (RSA)
- Extensive RAS

Technology:

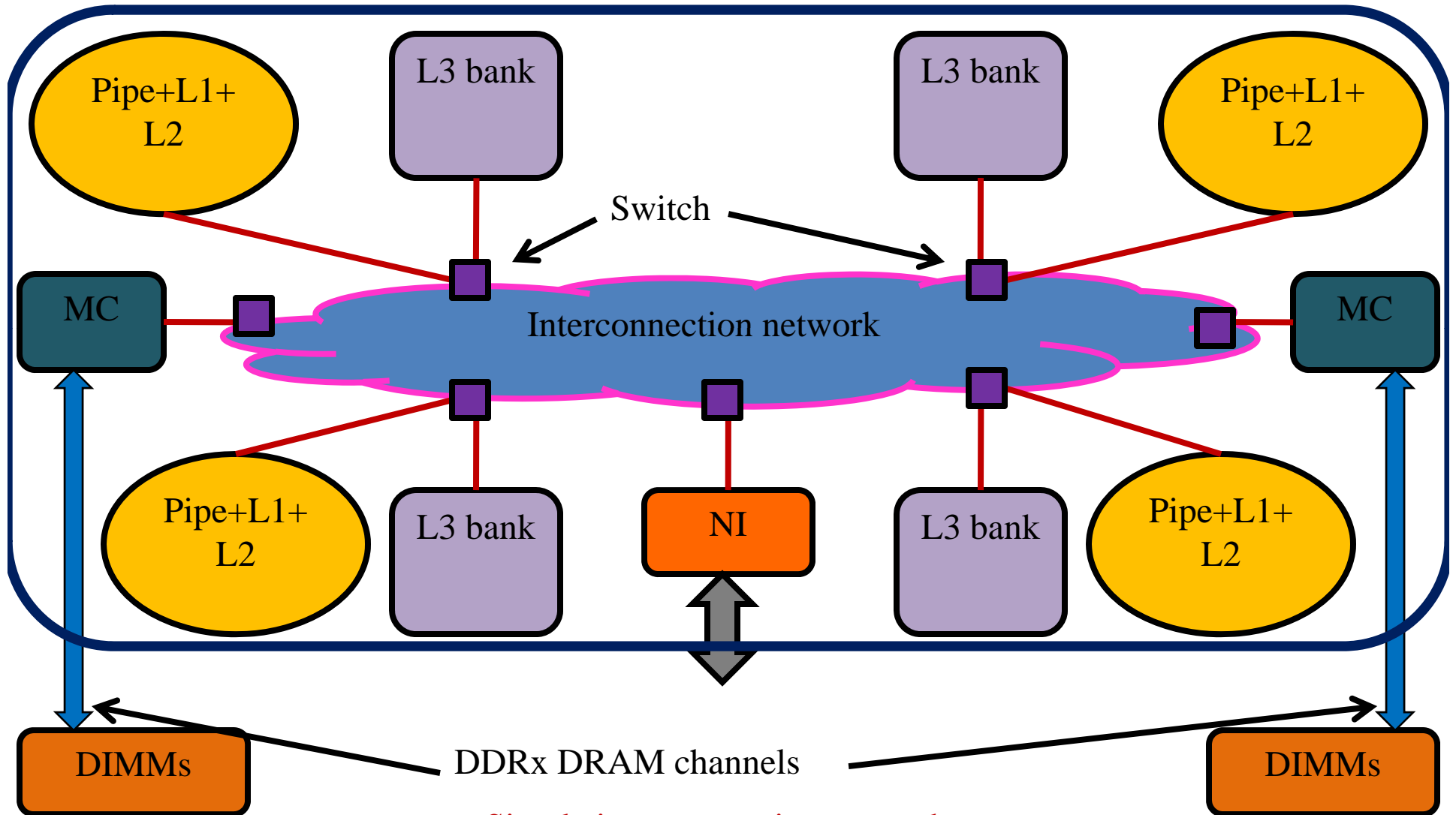
- 90nm CMOS Process
- 9LM Copper Interconnect
- Power: 63 Watts @ 1.2GHz
- Die Size: 378mm²
- 279M Transistors
- Package: Flip-chip ceramic LGA (1933 pins)

Quad-core Sandy Bridge



Sandy Bridge die photo (courtesy of ISSCC).

Floor of today's chips



Simple interconnection networks:
Linear bus, ring, 2D mesh (usually bidirectional)

Cache Coherence

Cache coherence

- Nothing unique to multiprocessors
 - Even uniprocessor computers need to worry about cache coherence
 - For sequential programs we expect a memory location to return the latest value written
 - For concurrent programs running on multiple threads or processes on a single processor we expect the same model to hold because all threads see the same cache hierarchy (same as shared L1 cache)
 - For multiprocessors there remains a danger of using a stale value: hardware must ensure that cached values are **coherent** across the system and they satisfy programmers' intuitive memory model

Cache coherence: Example

- Assume a write-through cache
 - P0: reads x from memory, puts it in its cache, and gets the value 5
 - P1: reads x from memory, puts it in its cache, and gets the value 5
 - P1: writes x=7, updates its cached value and memory value
 - P0: reads x from its cache and gets the value 5
 - P2: reads x from memory, puts it in its cache, and gets the value 7 (now the system is completely incoherent)
 - P2: writes x=10, updates its cached value and memory value

Cache coherence: Example

- Consider the same example with a writeback cache
 - P0 has a cached value 5, P1 has 7, P2 has 10, memory has 5 (since caches are not write through)
 - The state of the line in P1 and P2 is M while the line in P0 is clean
 - Eviction of the line from P1 and P2 will issue writebacks while eviction of the line from P0 will not issue a writeback (clean lines do not need writeback)
 - Suppose P2 evicts the line first, and then P1
 - Final memory value is 7: **we lost the store x=10 from P2**

What went wrong?

- For write through cache
 - The memory value may be correct if the writes are correctly ordered
 - But the system allowed a store to proceed when there is already a cached copy
 - Lesson learned: must invalidate all cached copies before allowing a store to proceed
- Writeback cache
 - Problem is even more complicated: stores are no longer visible to memory immediately
 - Writeback order is important
 - Lesson learned: do not allow more than one copy of a cache line in M state

Protocol implementations

- Must invalidate all cached copies before allowing a store to proceed
 - Need to know where the cached copies are
 - Solution1: Never mind! Just tell everyone that you are going to do a store
 - Leads to broadcast snoopy protocols
 - Popular with small-scale bus-based CMPs and SMPs
 - AMD Opteron implements it on a distributed network (the Hammer protocol)
 - Solution2: Keep track of the sharers and invalidate them when needed
 - Where and how is this information stored?
 - Leads to directory-based scalable protocols

Synchronization

Synchronization types

- Mutual exclusion
 - Synchronize entry into critical sections
 - Normally done with locks
- Point-to-point synchronization
 - Tell a set of processors (normally set cardinality is one) that they can proceed
 - Normally done with flags
- Global synchronization
 - Bring every processor to sync
 - Wait at a point until everyone is there
 - Normally done with barriers

Synchronization

- Normally a two-part process: acquire and release; acquire can be broken into two parts: intent and wait
 - **Intent**: express intent to synchronize (i.e. contend for the lock, arrive at a barrier)
 - **Wait**: wait for your turn to synchronization (i.e. wait until you get the lock)
 - **Release**: proceed past synchronization and enable other contenders to synchronize
- Waiting algorithms do not depend on the type of synchronization

Waiting algorithms

- Busy wait (common in multiprocessors)
 - Waiting processes repeatedly poll a location (implemented as a load in a loop)
 - Releasing process sets the location appropriately
 - May cause network or bus transactions
- Block
 - Waiting processes are de-scheduled
 - Frees up processor cycles for doing something else
- Busy waiting is better if
 - De-scheduling and re-scheduling take longer than busy waiting
 - No other active process
 - Does not work for single processor
- Hybrid policies: busy wait for some time and then block

Implementation

- Popular trend
 - Architects offer some simple atomic primitives
 - Library writers use these primitives to implement synchronization algorithms
 - Normally hardware primitives for acquire and possibly release are provided
 - Hard to offer hardware solutions for waiting
 - Also hardwired waiting may not offer that much of flexibility

Software locks

- Bakery algorithm

Shared: choosing[P] = FALSE, ticket[P] = 0;

Acquire: choosing[i] = TRUE; ticket[i] = max(ticket[0],...,ticket[P-1]) + 1;
choosing[i] = FALSE;

for j = 0 to P-1

 while (choosing[j]);

 while (ticket[j] && ((ticket[j], j) < (ticket[i], i)));

endfor

Release: ticket[i] = 0;

- Does it work for multiprocessors?
 - Performance issues related to coherence?
- Too much overhead: need faster and simpler lock algorithms
 - Need some hardware support

Hardware support

- Start with a simple software lock

Shared: lock = 0;

Acquire: while (lock); lock = 1;

Release or Unlock: lock = 0;

- Assembly translation

Lock: lw register, lock_addr /* register is any processor register */

 bnez register, Lock

 addi register, register, 0x1

 sw register, lock_addr

Unlock: xor register, register, register

 sw register, lock_addr

- Does it work?
 - What went wrong?
 - We wanted the **read-modify-write** sequence to be **atomic**

Atomic exchange

- We can fix this if we have an atomic exchange instruction

```
        addi register, r0, 0x1                /* r0 is hardwired to 0 */
Lock:   xchg register, lock_addr             /* An atomic load and store */
        bnez register, Lock
Unlock remains unchanged
```

- Various processors support this type of instruction
 - Intel x86 has *xchg*, Sun UltraSPARC has *ldstub* (load-store-unsigned byte), UltraSPARC also has *swap*

Compare & swap

- More sophisticated: compare & swap
 - Takes three operands: reg1, reg2, memory address
 - Compares the value in reg1 with contents of address and if they are equal swaps the contents of reg2 and address

Compare & swap

```
addi reg1, r0, 0x0      /* reg1 has 0x0 */
```

```
addi reg2, r0, 0x1      /* reg2 has 0x1 */
```

```
Lock:  compare & swap reg1, reg2, lock_addr
```

```
      bnez reg2, Lock
```

Barrier

- High-level classification of barriers
 - Hardware and software barriers
- Two types of software barriers
 - Centralized barrier: every processor polls a single count
 - Distributed tree barrier: shows much better scalability

Memory Consistency Models

Memory consistency models

- A parallel program can have multiple possible outputs

T0: $x=1$; T1: $y=x$; print y ;

What does T1 print if the initial value of x is 0?

- Coherence protocol is not enough to completely specify the output(s) of a parallel program
 - Coherence protocol only provides the foundation to reason about the legal outcomes of accesses to the **same** memory location
 - Memory consistency models tell us the possible outcomes arising from legal ordering of accesses to **all** memory locations

Memory consistency models

- Without any specified ordering constraints, all possible outputs could be legal
T0: A=1; print B; T1: B=1; print A;
What do T0 and T1 print if the initial values of A and B are zero?
- Given a memory consistency model, only a subset of outcomes is possible and it is important for a programmer to know this subset
 - A shared memory machine advertises the supported memory consistency model; it is a “contract” with the writers of parallel applications and system software

Memory consistency models

- Usually, the programmer wants one specific output from a correct program
 - Suppose the programmer wants T1 to print 1 in the following program
T0: x=1; T1: y=x; print y;
 - It may seem that synchronization can guarantee such an output (flag=0 initially)
T0: x=1; flag=1; T1: while(!flag); y=x; print y;
 - What if flag=1 is made visible to other processes before x=1 becomes visible?
 - Either by compiler re-ordering or hardware re-ordering (latter must still honor precise exception)
 - This is quite possible given that the two stores are going to two different addresses

Memory consistency models

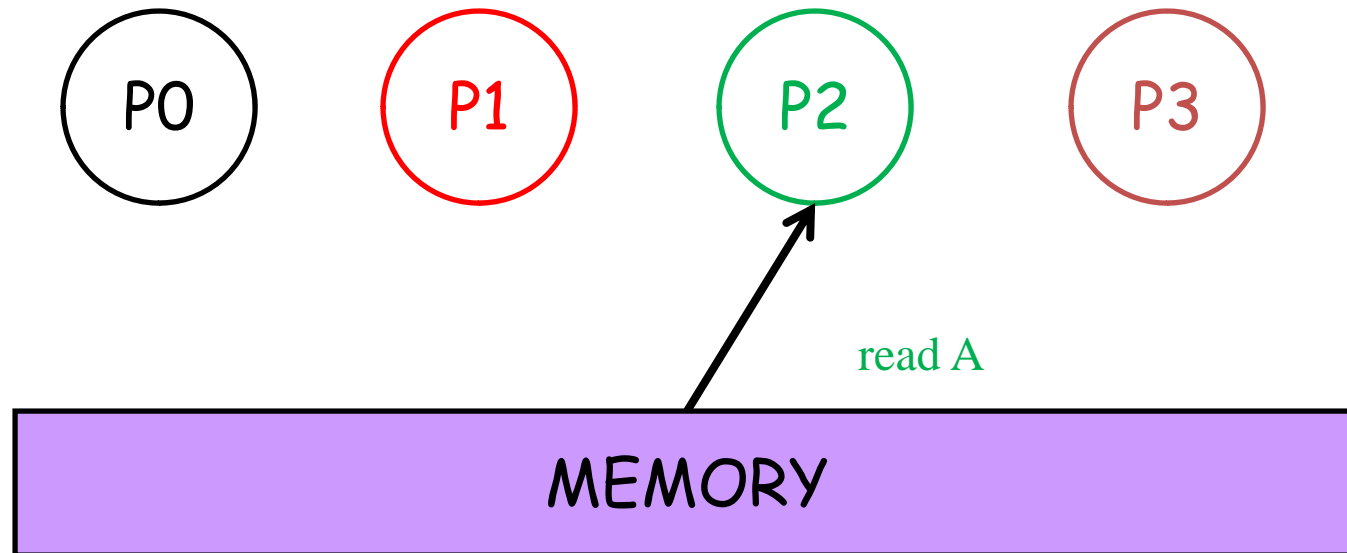
- A **memory consistency model** is a set of rules that specify the set of allowed orderings between all memory accesses
- A multiprocessor normally advertises the supported memory consistency model
 - This essentially tells the programmer what the possible correct outcome(s) of a program could be when run on that machine
 - A multiprocessor is said to implement a memory consistency model when it adheres to the set of ordering rules specified by that model

Sequential consistency

- Many memory consistency models exist
 - Each model represents a unique point in the three-dimensional space spanned by ease of programming, implementation complexity, and performance/energy
 - Sequential consistency (SC) is the most intuitive one and we will focus on it first
- Legal SC orders
 - Achieved by interleaving accesses from different processors
 - The accesses from the same processor are presented to the memory system in program order

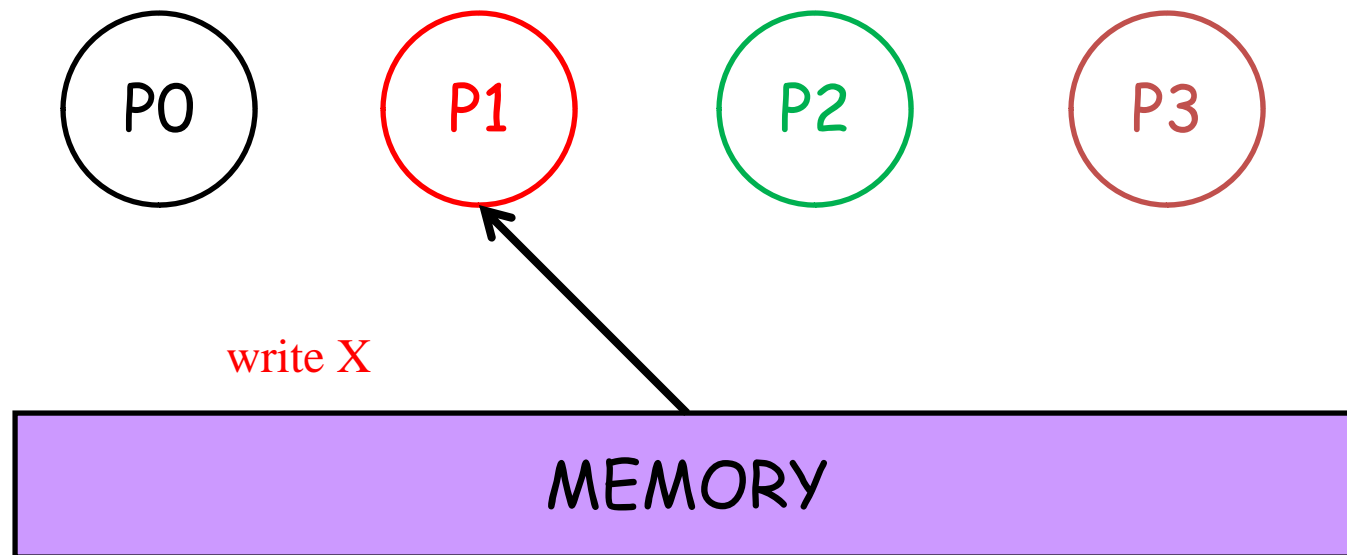
Sequential consistency

- A legal SC ordering is equivalent to the total order obtained by a randomly moving switch connecting the processors to memory
 - Picks the next access from a randomly chosen processor



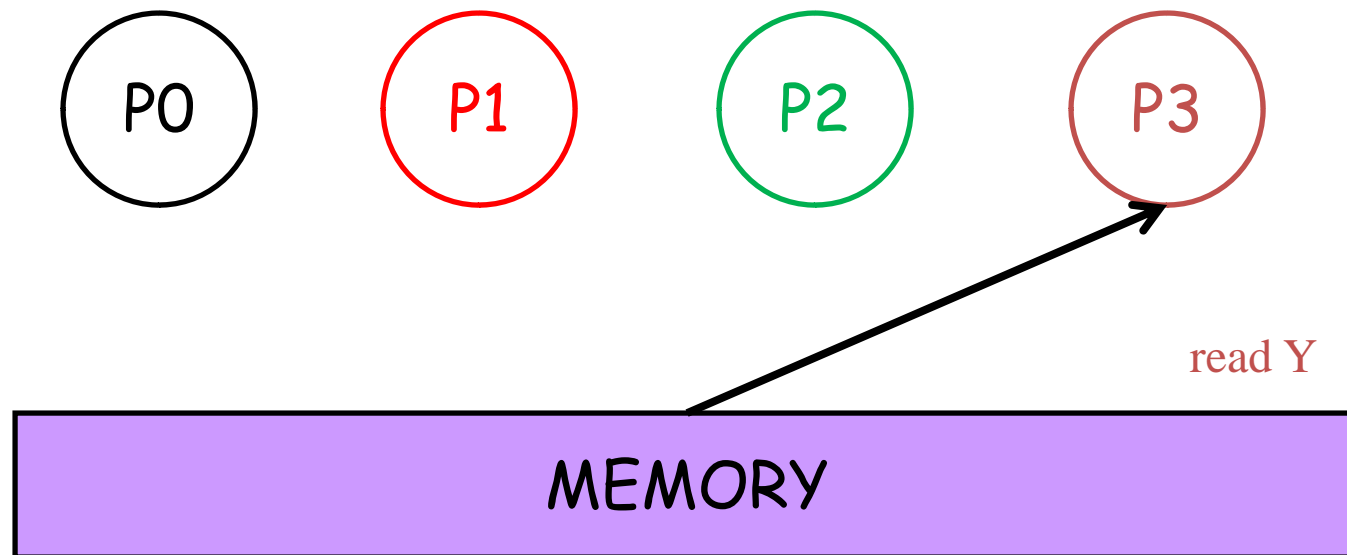
Sequential consistency

- A legal SC ordering is equivalent to the total order obtained by a randomly moving switch connecting the processors to memory
 - Picks the next access from a randomly chosen processor



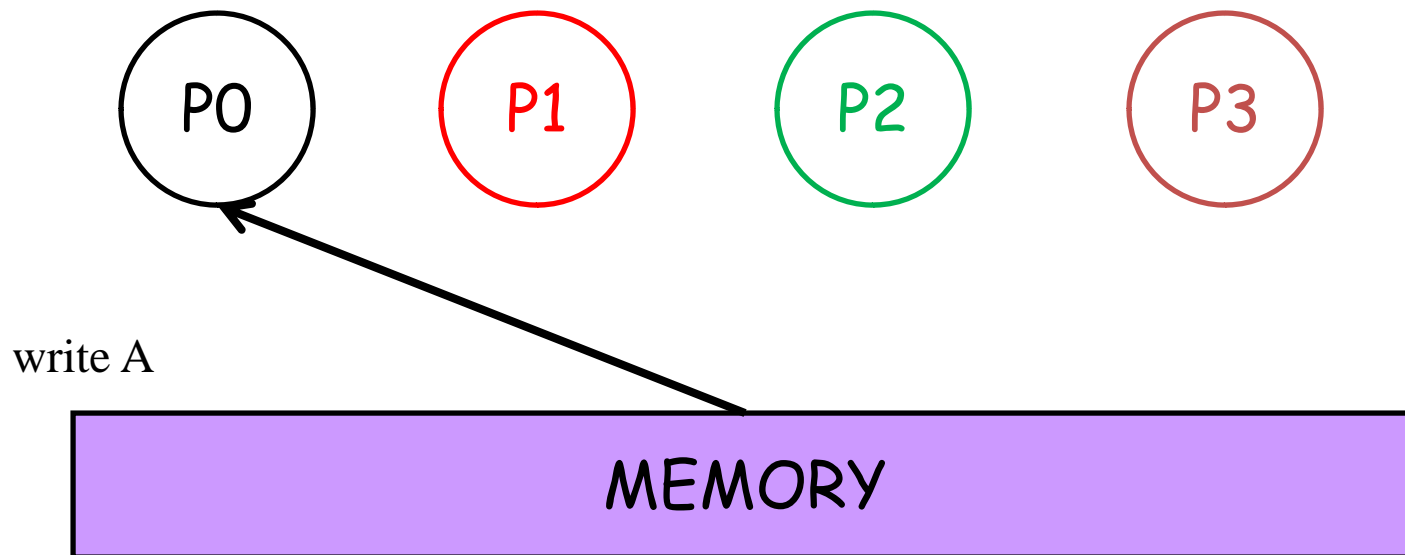
Sequential consistency

- A legal SC ordering is equivalent to the total order obtained by a randomly moving switch connecting the processors to memory
 - Picks the next access from a randomly chosen processor



Sequential consistency

- A legal SC ordering is equivalent to the total order obtained by a randomly moving switch connecting the processors to memory
 - Picks the next access from a randomly chosen processor



Total order: read A, write X, read Y, write A

What is program order?

- Any legal re-ordering is allowed
- The program order is the order of instructions from a sequential piece of code where programmer's intuition is preserved
 - The order **must** produce the result a programmer expects
- Can out-of-order execution violate program order?
 - No. All microprocessors commit instructions in-order and that is where the state becomes visible
 - For modern microprocessors the program order is really the commit order

Example

P0: $x=8$; $u=y$; $v=9$;

P1: $r=5$; $y=4$; $t=v$;

Total order: $x=8$; $u=y$; $r=5$; $y=4$; $t=v$; $v=9$;

Another legal total order:

$x=8$; $r=5$; $y=4$; $u=y$; $v=9$; $t=v$;

- All such total orders are allowed by sequential consistency
 - A multiprocessor implementing sequential consistency can produce an output conforming to any of these total orders
 - All these possible outputs are correct for a sequentially consistent multiprocessor

Relaxed models

- Implementing SC requires complex hardware and verification effort
 - But such violations are rare
 - Many processors today relax the consistency model to get rid of complex hardware and achieve some extra performance at the cost of making program reasoning complex
 - T0: A=1; B=1; flag=1; T1: while (!flag); print A; print B;
 - Can re-order the stores to A and B without any problem
 - Cannot compromise on precise exception, however
 - SC is too restrictive; relaxing it does not always violate programmers' intuition

Relaxed models

- Three attributes of a relaxed model
 - System specification: which orders are preserved and which are not; if all program orders are not preserved what support is provided (software and hardware) to enforce a particular order that the programmer wishes (often SC-compliant order is required)
 - Programmer's interface: set of rules, if followed, will lead to an execution as expected by the programmer; specified in terms of high-level language annotations and labels
 - Translation mechanism: how to translate programmer's annotations to hardware actions

Total store order (TSO)

- One of the many relaxed models
- TSO allows a load to bypass and commit earlier than an older incomplete store
 - A blocked store at the head of the ROB can be removed (but remains in a FIFO store buffer) and subsequent instructions are allowed to commit bypassing the blocked store
 - Motivation: can hide latency of store operations
 - This is the only allowed re-ordering
 - Only a load can bypass an older store
 - A load cannot bypass an older load; a store cannot bypass an older load; a store cannot bypass an older store

Total store order (TSO)

- TSO allows a load to bypass and commit earlier than an older incomplete store
 - Programmer's intuition is preserved in most cases, but not always
 - P0: A=1; flag=1; P1: while (!flag); print A; [same as SC]
 - P0: A=1; B=1; P1: print B; print A; [same as SC]
 - P0: A=1; P1: while (!A); B=1; P2: while (!B); print A; [same as SC]
 - P0: A=1; print B; P1: B=1; print A; [violates SC]
 - Implemented in many Sun (Oracle) UltraSPARC microprocessors

Agenda

- Basics of computer architecture
 - Basics of the basics
 - Instruction set architecture (ISA)
 - Processor design
 - Caches and virtual memory
 - Communicating with environment
 - Performance measurement
 - Performance optimization
 - Multi-core processors
- Basics of operating systems

What is an operating system

- A piece of software application
- Resource manager of any computing system
 - Schedules resources like CPU, memory, hard disks, and other I/O devices

Summary of OS functionalities

- Two broad categories
 - Functionalities to improve performance
 - Job scheduling, context switch, memory management
 - Functionalities to improve ease of use
 - File systems, I/O, security
- Two modes of operations
 - User mode and kernel mode
 - A system call from a user program leads to a switch to kernel mode
 - Kernel mode allows unrestricted access to hardware including execution of privileged instructions

Summary of OS functionalities

- Two modes of operations
 - User mode avoids catastrophic failures
 - Isolated virtual address space for each process in user mode
 - Isolated execution of each process
 - No direct access to any hardware device
 - CPU needs to support a mode bit as part of the machine status word or processor status word
 - Switching modes is expensive
 - Needs to save and restore user and kernel register states

Summary of OS functionalities

- System boots up in kernel mode
 - Kernel of the OS is loaded by the bootstrap loader from a fixed location in the disk
 - Only when a user program is scheduled to run on the CPU, does the mode bit switch to user mode
- System calls invoke system call handlers
 - Locations of the handlers are found in an interrupt vector table residing in the low memory
 - Hardware interrupts are handled in the same way
 - System call arguments are passed in registers and/or in memory (by passing a pointer in a register); what are these arguments?

Summary of OS functionalities

- OS does several book-keeping tasks periodically
 - Job scheduling is an important example
 - Implemented by setting up a timer register which is decremented on every processor clock tick
 - When the timer register expires, a hardware interrupt is generated
 - The interrupt handler services the periodic tasks one by one and sets up the timer register again

Summary of OS functionalities

- Four basic OS modules
 - Process management, memory management, storage management, protection
 - Process management involves
 - Creation, deletion, scheduling of processes
 - Offering support for communication between processes
 - Synchronizing communicating processes
 - Handling deadlocks
 - Memory management involves
 - Handling memory allocation and de-allocation requests from user and kernel mode processes

Summary of OS functionalities

- Four basic OS modules
 - Storage management involves
 - Implementing a virtual environment called file system
 - Protection cross-cuts all the three modules
 - Controls accesses to the resources managed by the OS
 - Usually the OS kernel is assumed to be trusted
 - A stricter protection model requires hardware-supported security

Processes and Threads

What constitutes a process

- A process is an executable in action
 - An executable is a passive entity and a process is an active instantiation of the executable
 - A new process gets created when an executable starts running
 - At any point in time, there may be several processes inside the system (all may not be executing)
 - Multiple processes may execute the same program
 - Multiple open shells, multiple internet browsers
 - Process states: created, ready, executing, waiting/sleeping, terminated

What constitutes a process

- Handling multiple processes is necessary to maximize hardware resource utilization
 - User executables run as user processes
 - Kernel executables run as kernel processes
 - Hardware cannot distinguish between these without looking at the mode bit
- Each process has its own text, data, and stack regions
 - Stack and heap grow in opposite directions
 - Intermediate values are maintained in processor registers; stack pointer and the program counter are two special registers

What constitutes a process

- Anything that is needed to reconstruct the state of a process is in the process context
 - Processor registers
 - Text, data, stack regions in memory
 - Various kernel data structures such as list of open files and their seek pointers, etc.
 - The process control block (PCB) is a kernel data structure that maintains all information pertaining to the process context including the current state of the process

Process control block

- The PCB entries start off as invalid for a newly created process
- The process enters the ready queue by linking its PCB to the tail of the queue
 - Ready queue is a linked list of PCBs of processes that are ready to execute
- On transition from executing to sleep state
 - The PCB is used to remember the process context by maintaining
 - a pointer to the memory region holding the saved registers and other context information
 - a pointer to the process region table

Process control block

- On transition from executing to sleep state
 - The PCB is linked at the tail of the waiting list of PCBs for the particular device or event
- On transition from ready to executing state
 - Context is restored from PCB
 - Known as a context switch
- The PCB of a non-executing process is either in the waiting queue of some device/event or in the ready queue

Processes and threads

- A process has a single thread of control
 - This thread refers to the sequence of instructions executed by the process
 - A fork() call generates a new thread of control
 - Is it a thread or a process? Often used interchangeably
- Historically, a thread refers to a piece of code in execution and is a part of a parent process
 - As a result, lighter-weight than a full process

Processes and threads

- Multiprocessing: executing independent or communicating processes simultaneously (or in a time-shared manner)
- Multithreading: executing possibly dependent and/or communicating parts of the same process simultaneously in different threads of control
- Multiprogramming: executing independent programs in different processes simultaneously
- Process scheduler is usually not aware of the dependencies, if any, between the processes in the ready queue

Process creation

- A process is always created by another process
 - Created process is the child of the creating process
- The system boots up as a process
 - *init* process in UNIX, *schd* process in Solaris
 - This is the root of all processes
 - Every process gets a unique integer ID known as the process ID or pid. The root process has pid zero.
 - In UNIX, the system boot process has pid zero and *init* has pid one.
- A process can be created by calling `fork()`
 - Child pid is returned to parent, zero is returned to child. A negative return value indicates error in UNIX.

Process Scheduling

Overview of process scheduling

- Every process in the ready queue can request CPU cycles for execution
 - An algorithm must select one process per processor for execution; this is the scheduling algorithm
 - Two types of process schedulers: short-term and medium-term schedulers
- A short-term scheduler selects one process from the ready queue for execution
 - Invoked whenever the currently executing process enters the sleep state or encounters a timer interrupt or terminates
 - Decides the overall utilization of the CPU

Overview of process scheduling

- Two types of processes
 - CPU-bound: These processes mostly compute i.e., major portion of life spent in ready or executing state
 - I/O-bound: These processes spend significant amount of time in I/O i.e., major portion of life spent in sleep state
 - The time spent computing between two consecutive I/O operations is known as a CPU burst
 - A CPU burst of a process can get executed through multiple transitions between the ready and executing states culminating into a sleep state when the CPU burst completes
 - The CPU burst of a process does not include the time spent waiting in the ready queue

Overview of process scheduling

- Primary goal of a short-term scheduler
 - Every time a scheduling event occurs, it should pick a process so that
 - all I/O bursts are overlapped by CPU bursts
 - the CPU is busy all the time executing the CPU burst of some process (this may still not lead to 100% CPU utilization)
 - the scheduled process can execute for the full allocated scheduling quantum (minimizes the chance of too frequent context switches); this is usually hard to guarantee
- Scheduling quality plays an important role in determining the degree of multiprogramming

Overview of process scheduling

- Medium-term scheduler
 - If the system is running low on resources (e.g., memory), some processes may have to be swapped out from memory to disk and later swapped in when needed
 - The medium-term scheduler selects the processes to be swapped out from memory and swapped in from disk
 - Not very critical to overall performance as long as a currently running process is not swapped out
 - Invoked whenever the resource usage goes above or falls below a threshold (usually infrequent)

Overview of process scheduling

- Long-term scheduler
 - Found in old computers where jobs (or processes) used to be submitted in batches
 - The long-term scheduler decides which of the submitted jobs will be loaded in memory and entered in the ready queue
 - Invoked whenever a job completes so that a new job can be loaded (even less frequent than the medium-term scheduler)

General scheduling mechanism

- Process scheduling is the activity of selecting the process that will run next on the CPU
- A scheduler saves the context of the currently running process, selects a process from the ready queue, restores the context of selected process

General scheduling mechanism

- The scheduler can be invoked in four possible circumstances
 - The currently running process goes on a long-latency system call i.e. transitions from running to sleep state
 - A new process is created or a process completes a long-latency system call i.e., a process transitions from created to ready or from sleep to ready
 - The currently running process terminates
 - The currently running process receives a timer interrupt
 - The first and the third cases lead to non-preemptive or co-operative scheduling
 - The remaining two cases lead to pre-emptive scheduling

Goals of process scheduling

- A scheduling algorithm can target a subset of the following
 - Maximize throughput: rate at which processes complete
 - Minimize turnaround time
 - Average, maximum, standard deviation?
 - Minimize waiting time in the ready queue
 - Direct measure of scheduler efficiency
 - Average, maximum, standard deviation?
 - Minimize response time
 - How long a process takes to produce the first result
 - Important for interactive systems
 - Average, maximum, standard deviation?

Thank you