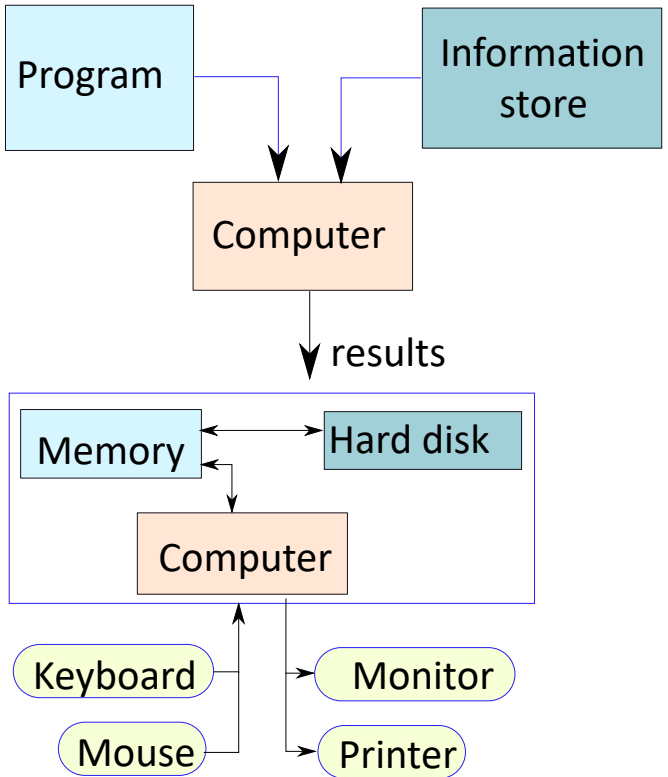
COMPUTER ARCHITECTURE

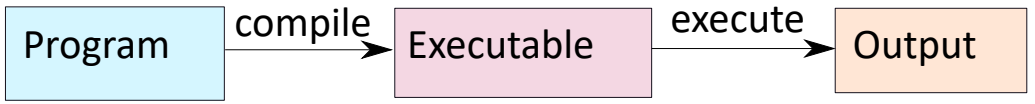
A Quick Review

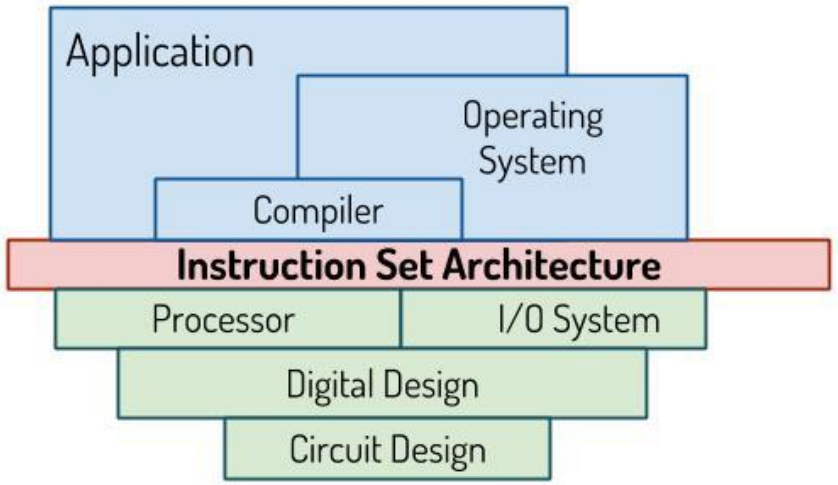
Slides Courtesy: Sarangi and Parhami Books

The Computer

- Architecture
  - The view of a computer as presented to software designers
- Organization
  - Implementation of a computer in hardware

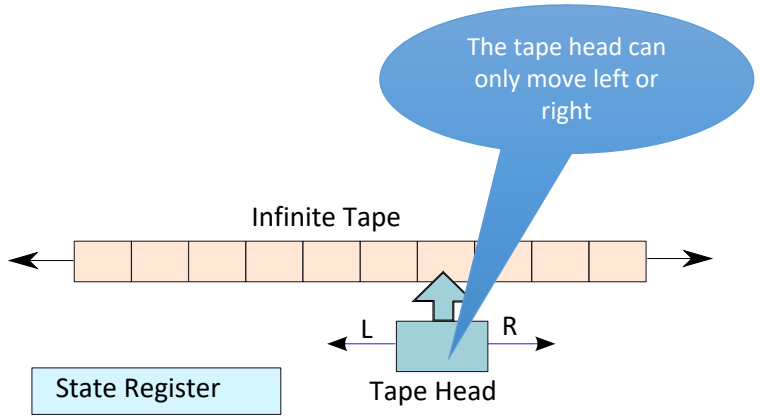


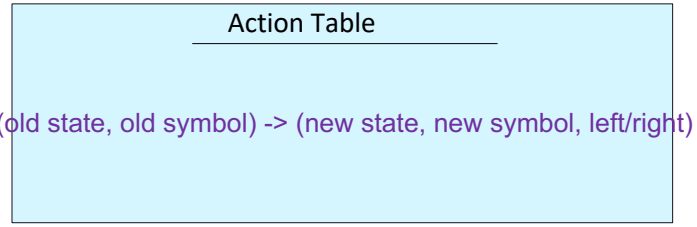


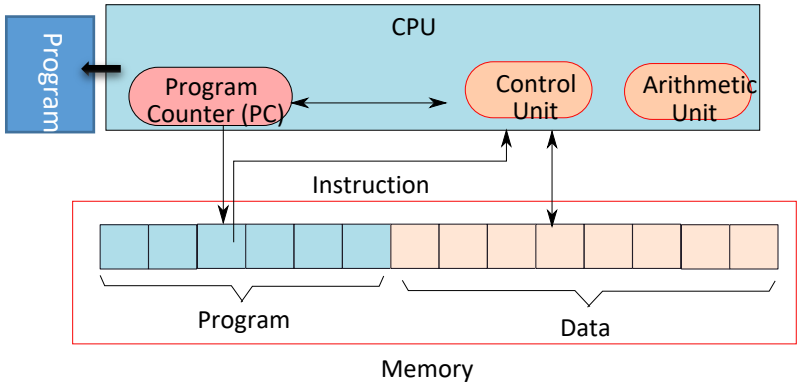


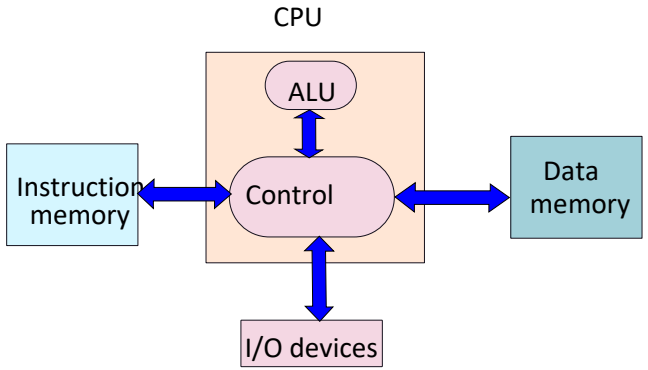
The Theory

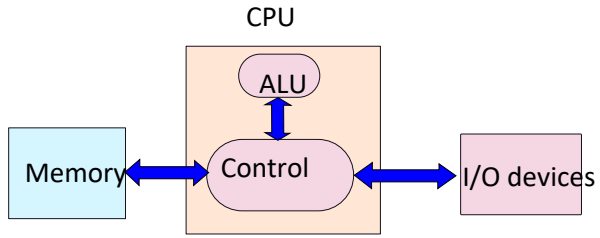
- Start at the Beginning
  - Turing Machines
    - Based on the symbol in the cell, and its current state, the Turing machine computes the transition :
      - Computes the next state
      - Overwrites the symbol in the cell (or keeps it the same)
      - Moves to the left or right by 1 cell
    - Extremely simple, and extremely powerful
- Real Architectures
  - Von Neumann
  - Harward







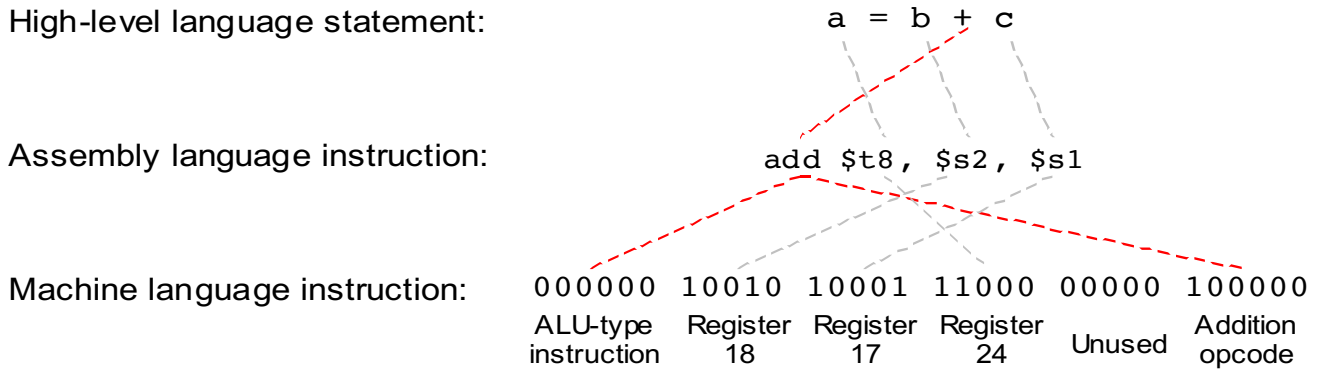


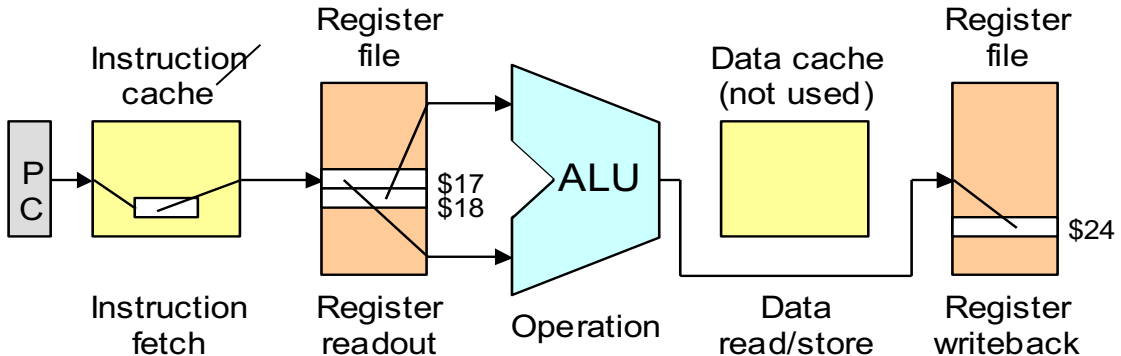


Instruction Set Architecture

- The Language of Instructions
  - Interface between Hardware and Software
    - A compiler converts a program into machine instructions in the given ISA
    - The processor executes the instructions in the ISA
      - CISC/RISC
  - Defined by Registers, Instructions and addressing Modes
    - Machine instruction typically contains
      - an opcode
      - one or more source operands
      - possibly a destination operand

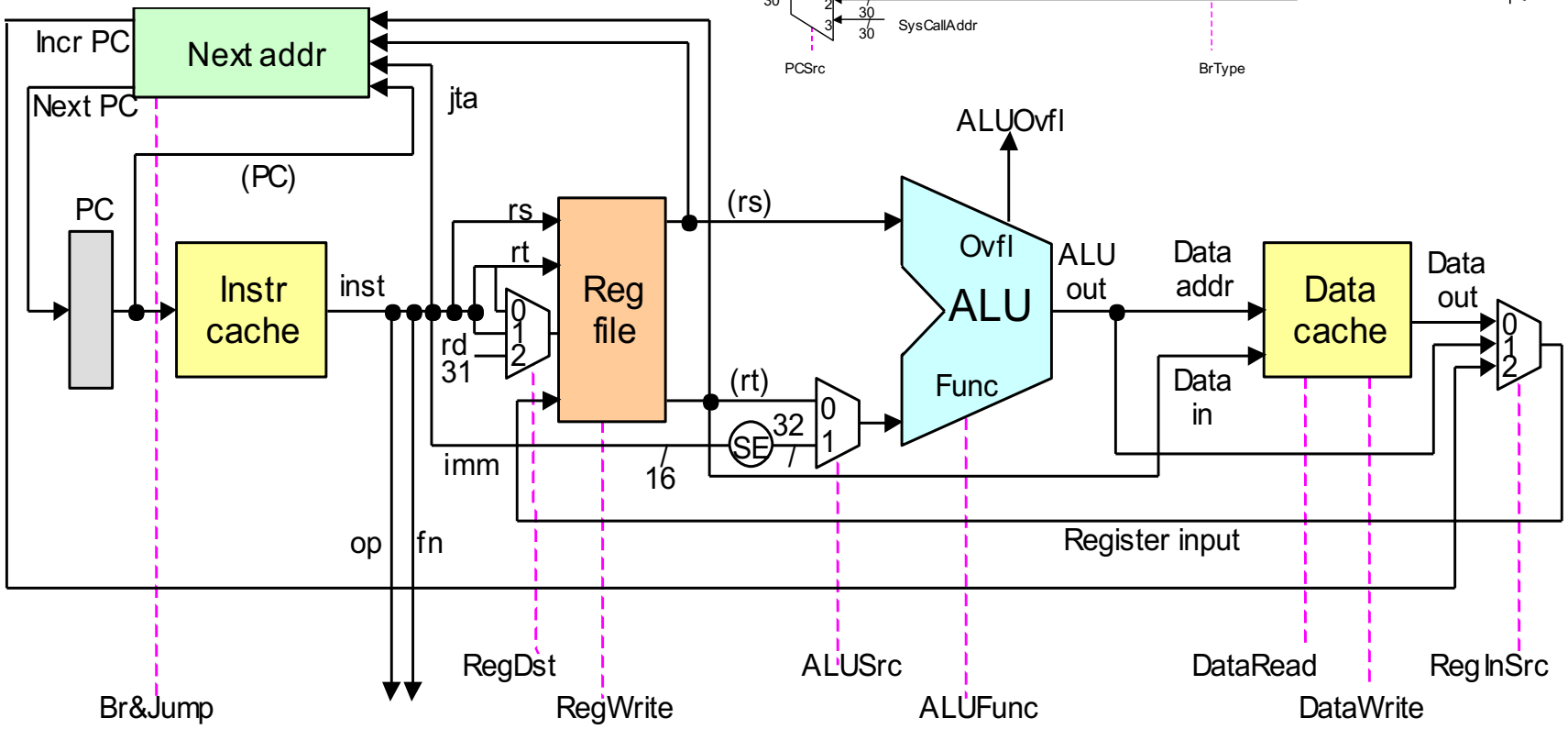
```
addi   $t0,$s0,61  # set $t0 to ($s0)+61
xori   $t0,$s0,0x00ff# set $t0 to ($s0)⊕ 0x00ff
lw     $t0,40($s3)  # load mem[40+($s3)] in $t0
sw     $t0,A($s3)  # store ($t0) in mem[A+($s3)]
j      verify       # go to mem loc named "verify"
```

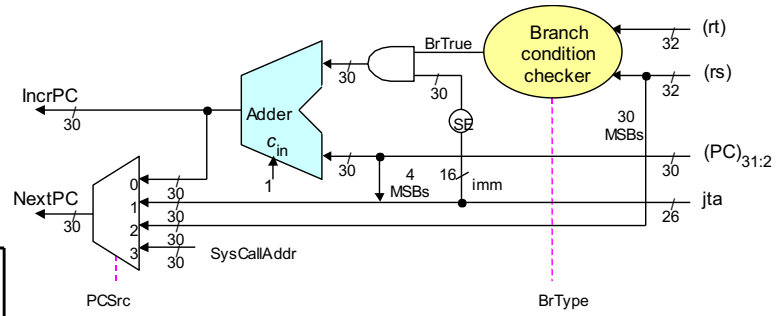


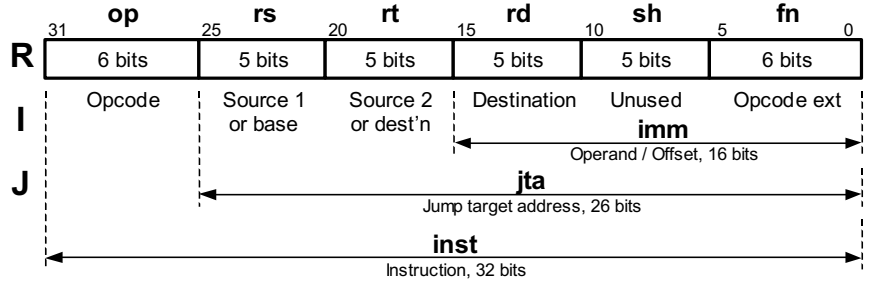


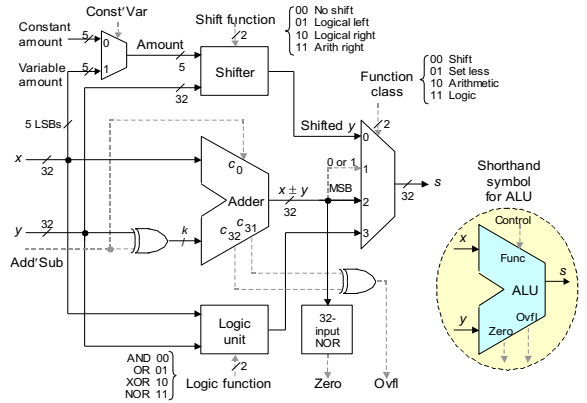
Single Cycle MicroArchitecture

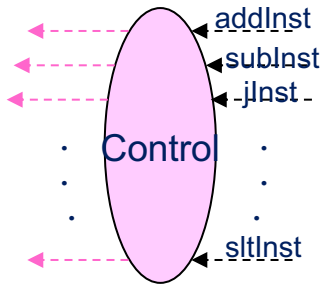
- Abstract view of the instruction execution unit







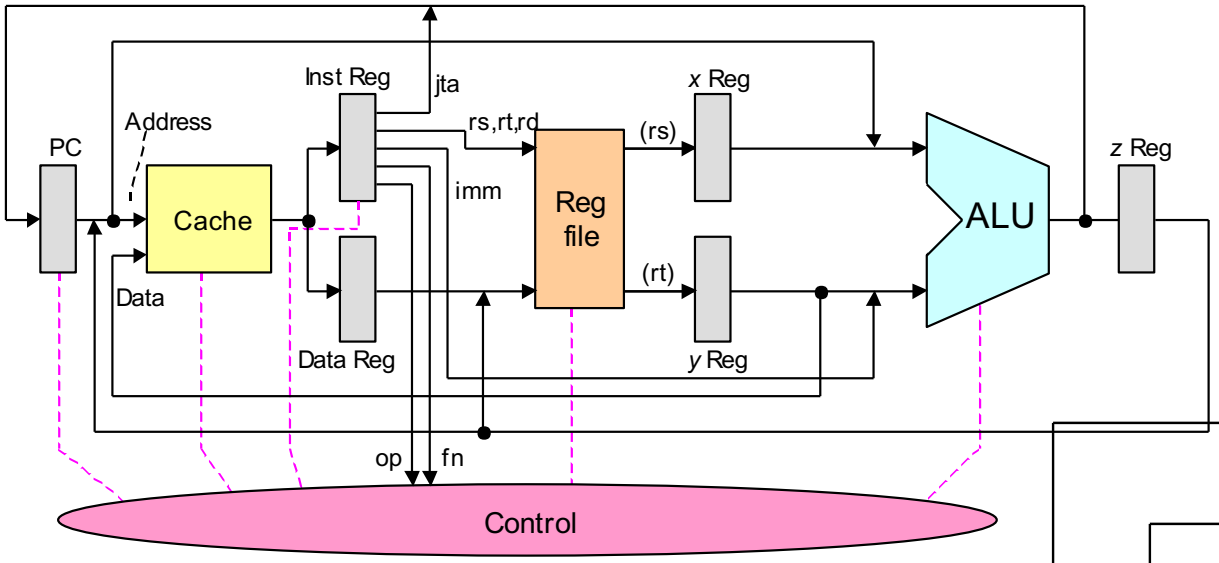


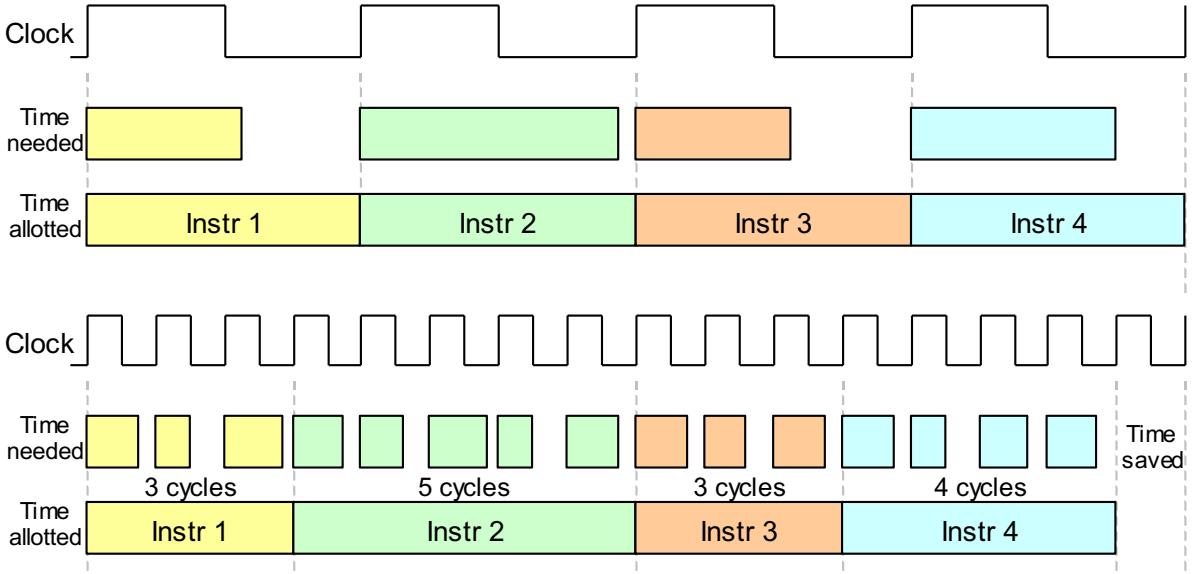


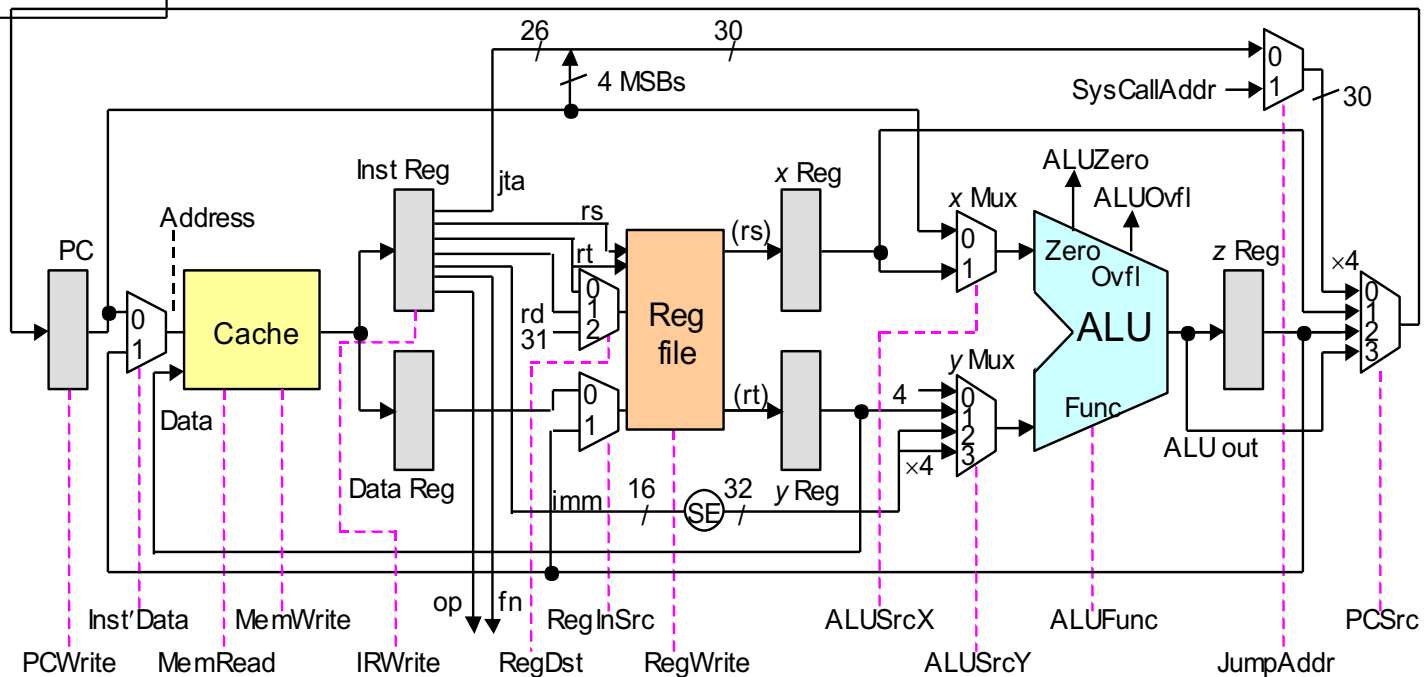
Instruction fetch **Reg access / decode** **ALU operation** **Data access** **Reg write- back**

Multi Cycle MicroArchitecture

- Improve Performance

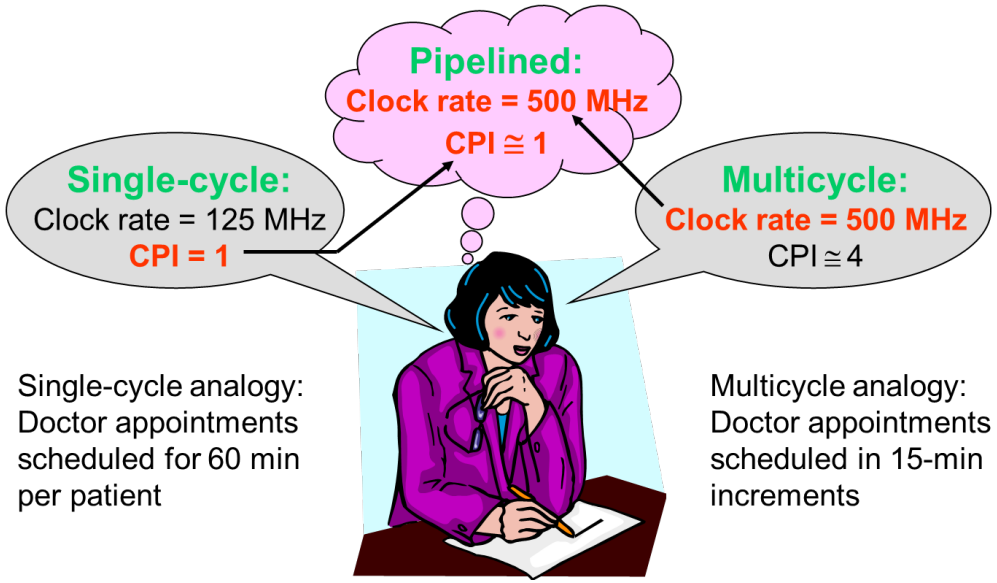


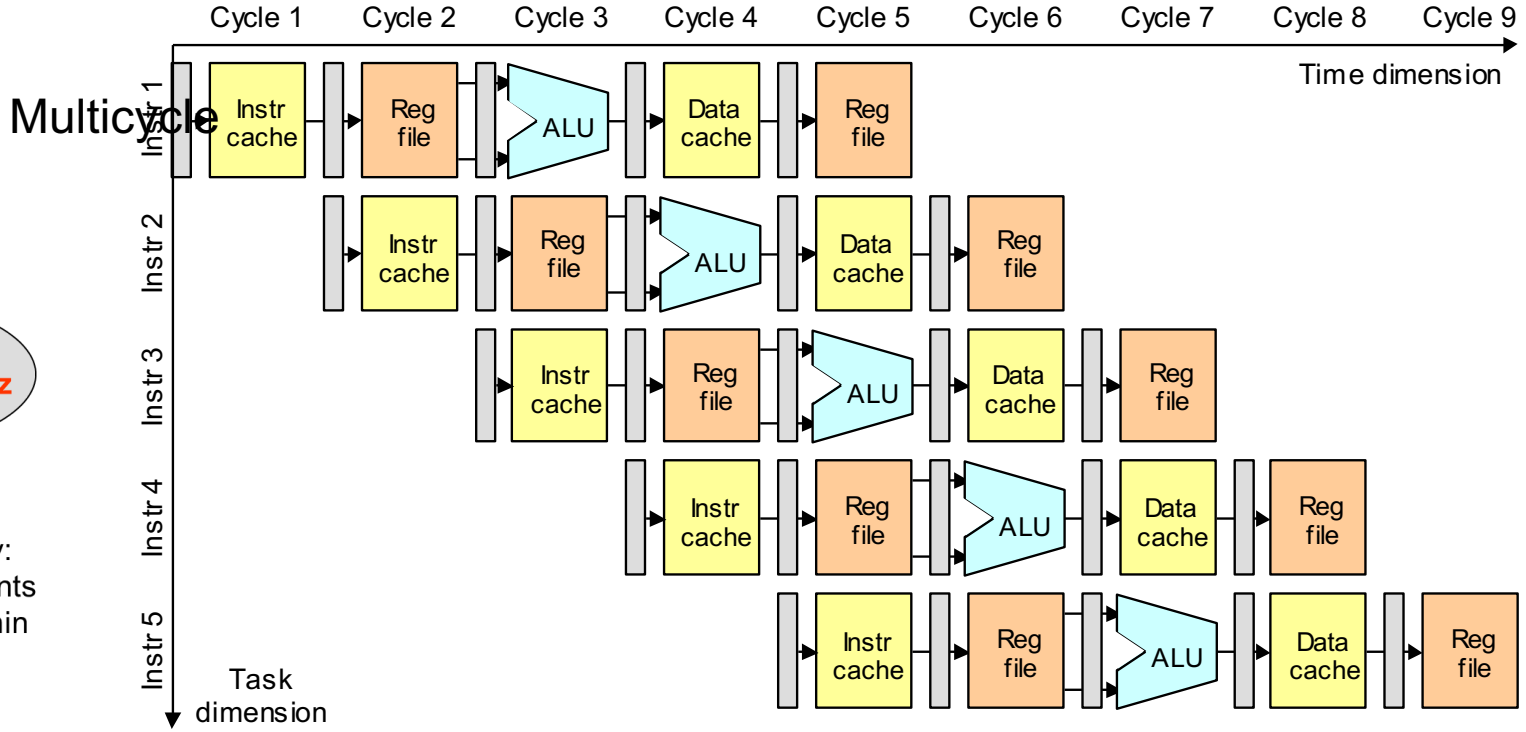


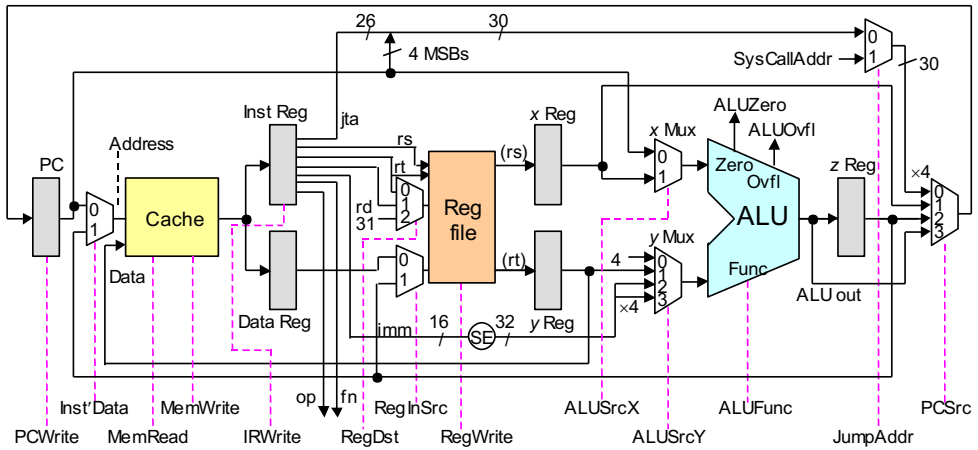


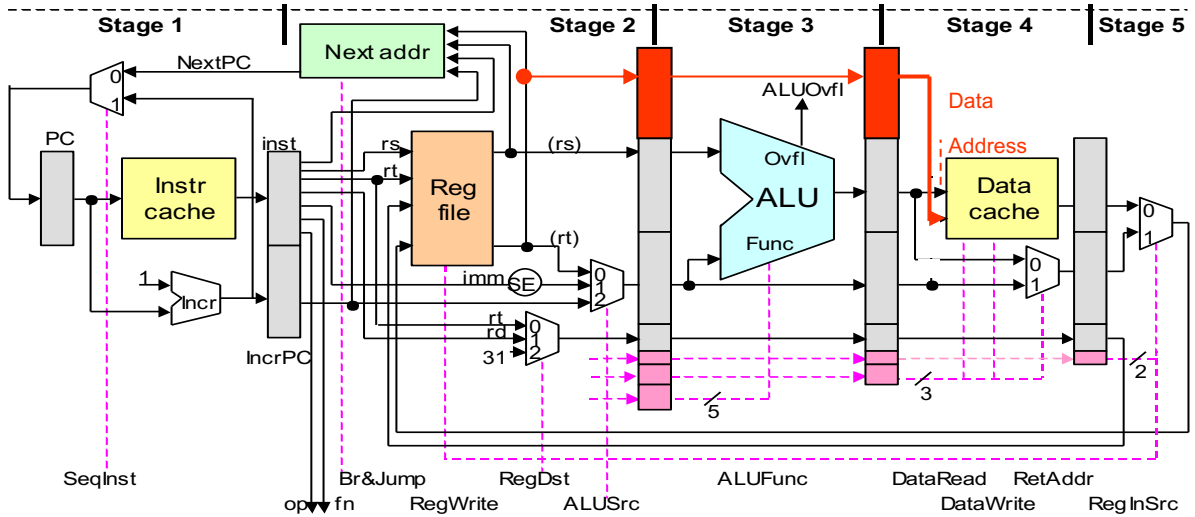
Pipelined MicroArchitecture

- CPI + High Clock Rates









Performance Equation

- How do we Evaluate

Performance = 1 / Execution time simplified to 1 / CPU execution time

CPU execution time = Instructions × CPI / (Clock rate)

Performance = Clock rate / (Instructions × CPI)

Instruction access	2 ns
Register read	1 ns
ALU operation	2 ns
Data cache access	2 ns
Register write	1 ns
Total	8 ns

Single-cycle clock = 125 MHz

R-type	44%	6 ns
Load	24%	8 ns
Store	12%	7 ns
Branch	18%	5 ns
Jump	2%	3 ns

Weighted mean ≅ 6.36 ns

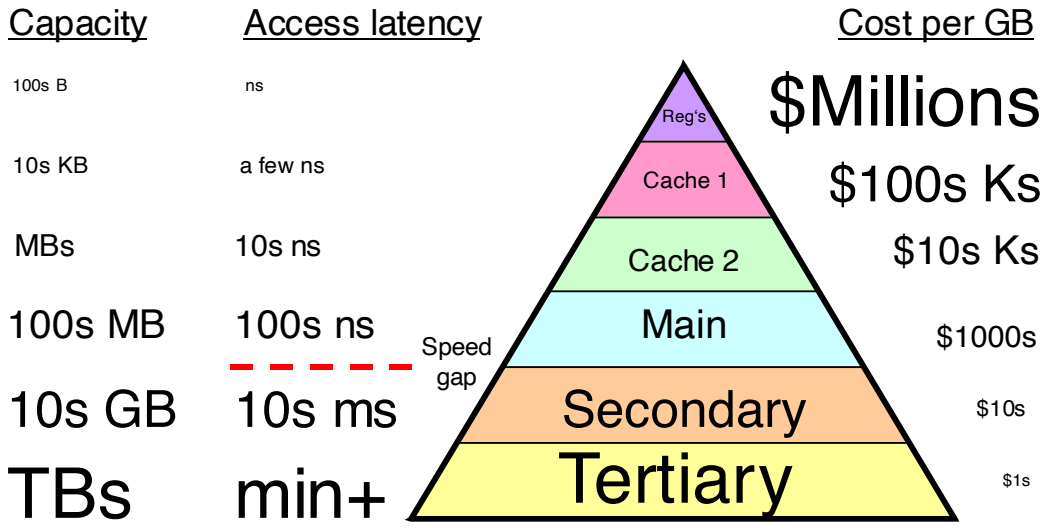
R-type	44%	4 cycles
Load	24%	5 cycles
Store	12%	4 cycles
Branch	18%	3 cycles
Jump	2%	3 cycles

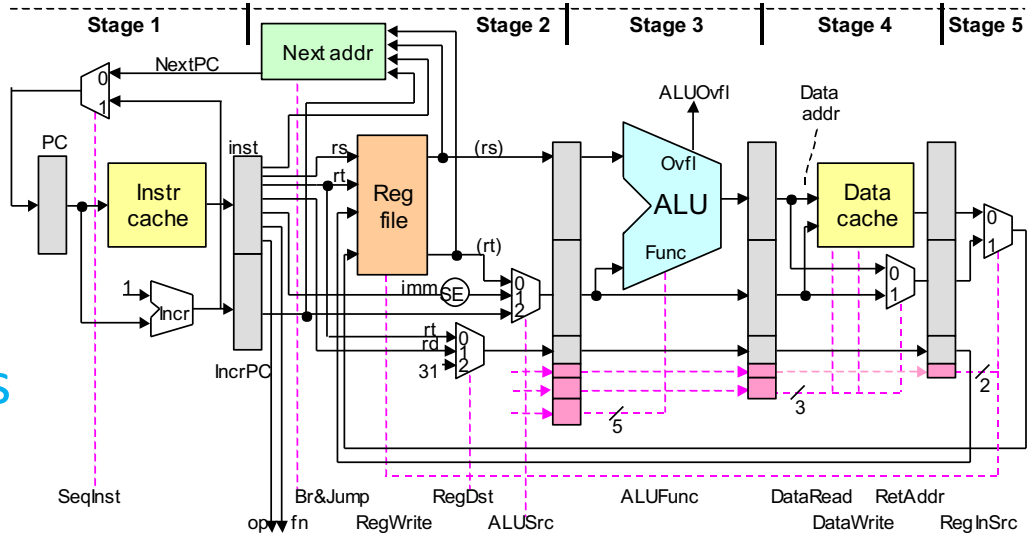
Multi-cycle clock = 500 MHz

	Contribution to CPI
R-type	0.44×4 = 1.76
Load	0.24×5 = 1.20
Store	0.12×4 = 0.48
Branch	0.18×3 = 0.54
Jump	0.02×3 = 0.06
	Average CPI ≅ 4.04

Need for a Memory Hierarchy

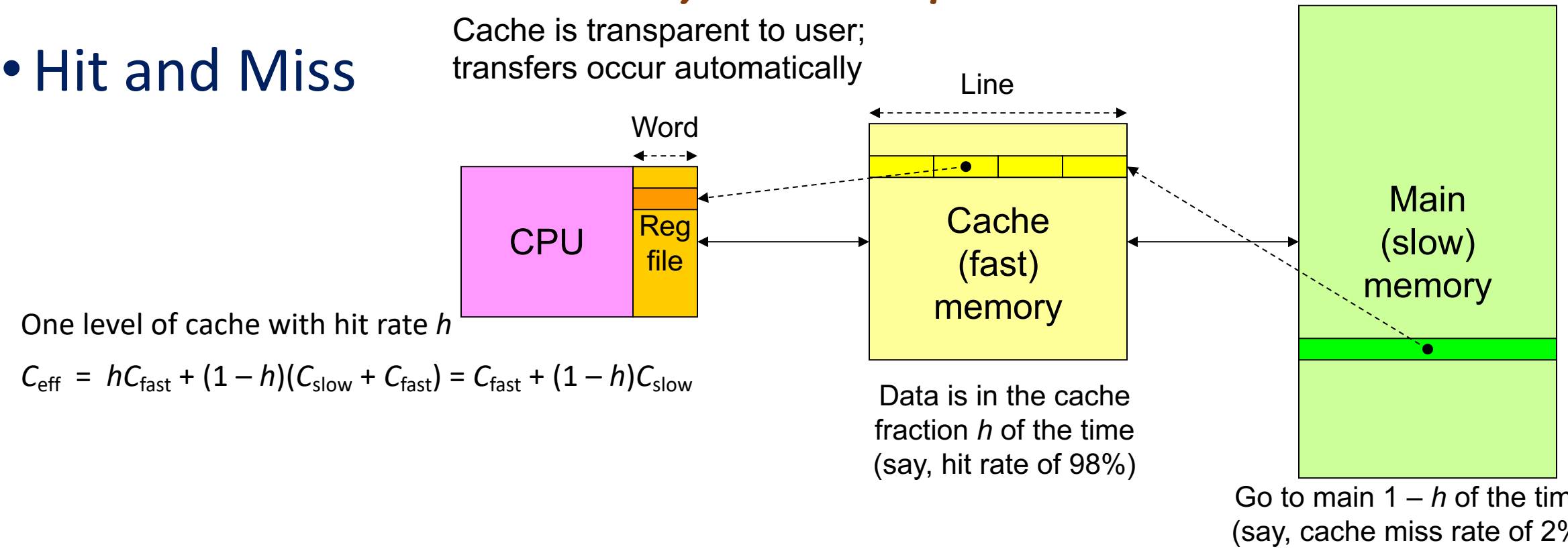
- The widening speed gap between CPU and main memory
  - Processor operations take of the order of 1 ns
  - Memory access requires 10s or even 100s of ns
- Memory bandwidth limits the instruction execution rate
  - Each instruction executed involves at least one memory access
    - Hence, a few to 100s of MIPS is the best that can be achieved
  - A fast buffer memory can help bridge the CPU-memory gap
    - The fastest memories are expensive and thus not very large
    - A second (third?) intermediate cache level is thus often used





Cache Memory –The Operative Terms

- Hit and Miss

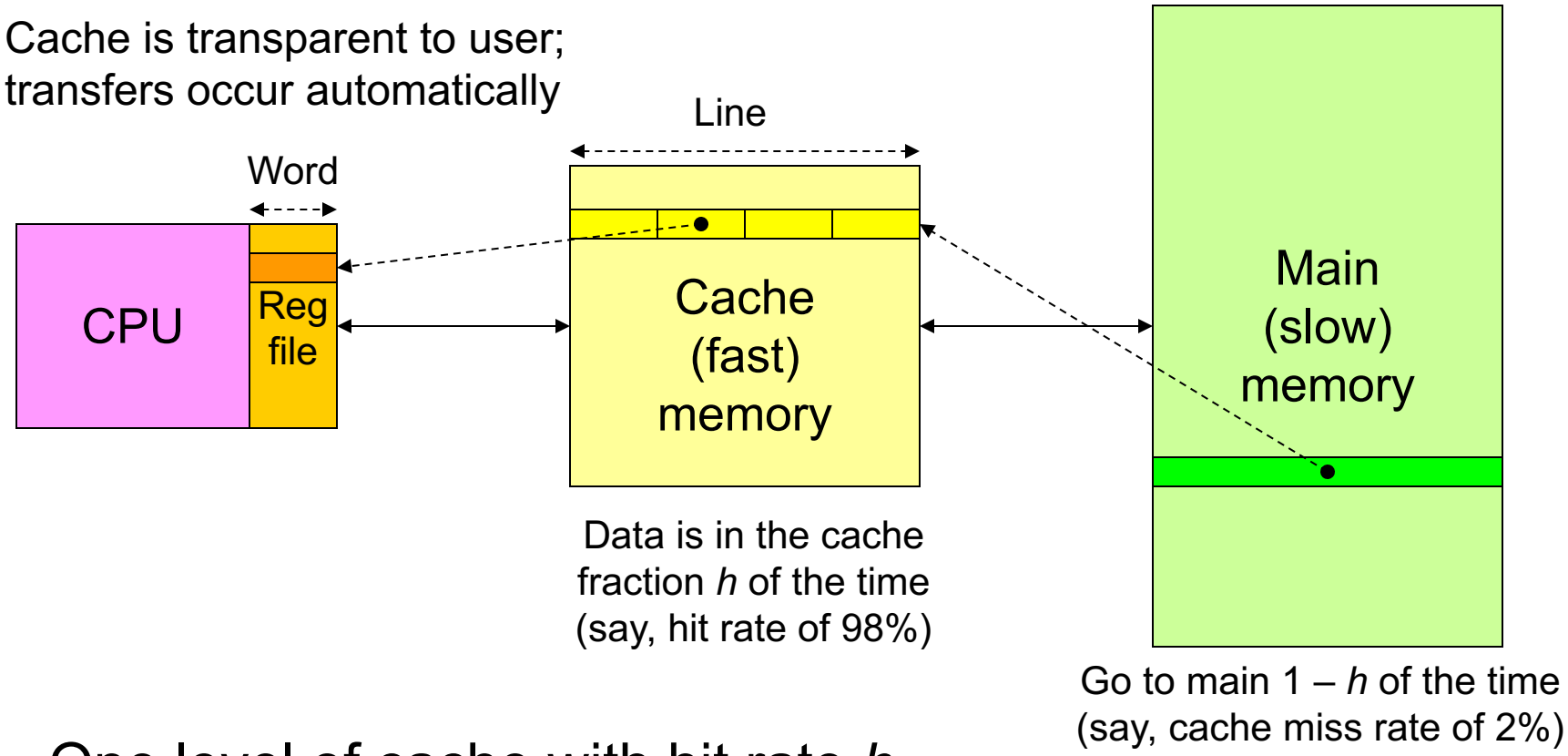


Compulsory misses: With *on-demand fetching*, first access to any item is a miss. Some "compulsory" misses can be avoided by prefetching.

Capacity misses: We have to oust some items to make room for others. This leads to misses that are not incurred with an infinitely large cache.

Conflict misses: Occasionally, there is free room, or space occupied by useless data, but the mapping/placement scheme forces us to displace useful items to bring in other items. This may lead to misses in future.

Cache, Hit/Miss Rate, and Effective Access Time



One level of cache with hit rate h

$$C_{eff} = hC_{fast} + (1 - h)(C_{slow} + C_{fast}) = C_{fast} + (1 - h)C_{slow}$$

Cache Memory Design Parameters

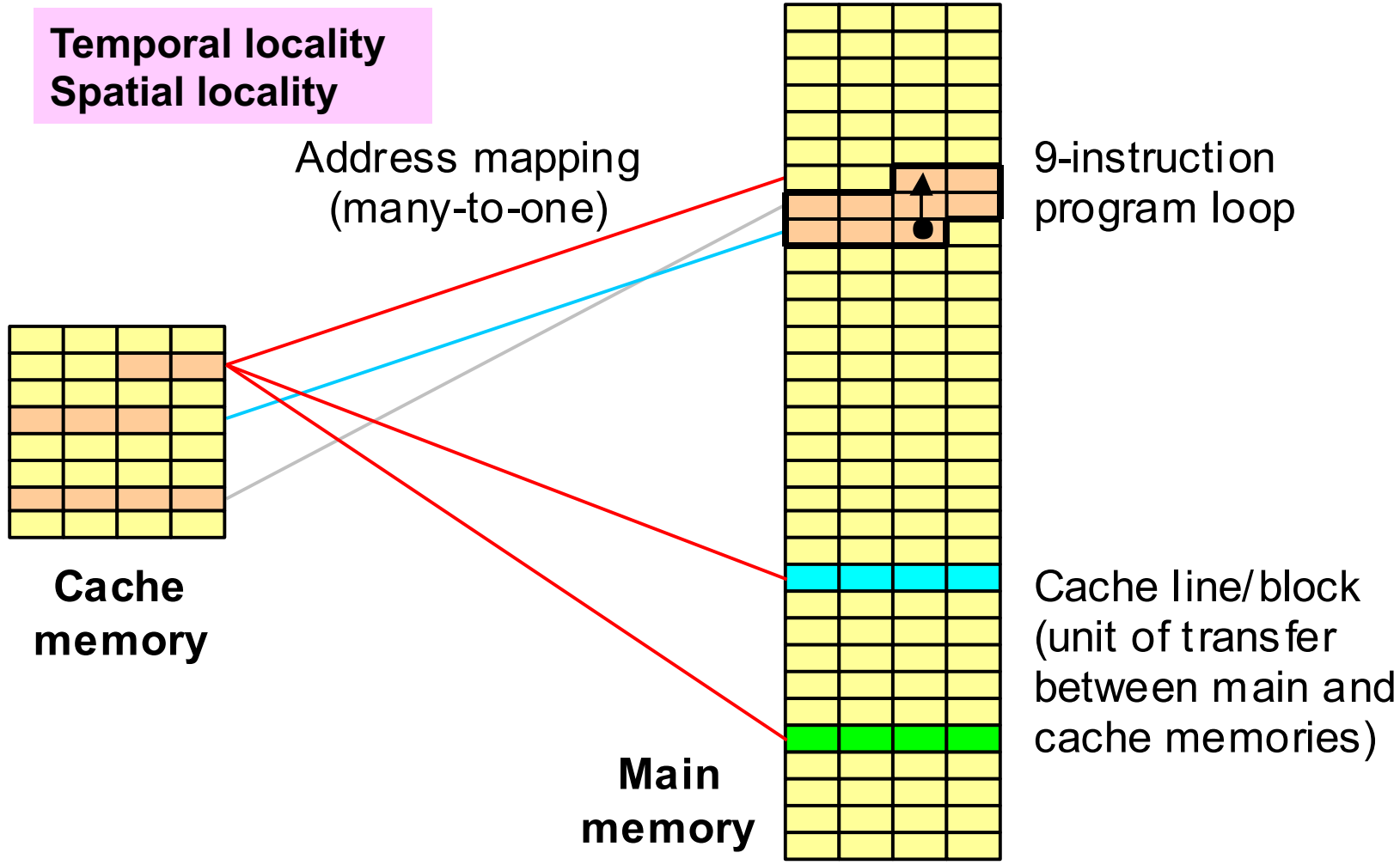
- *Cache size*
  - In bytes or words
    - A larger cache can hold more of the program's useful data but is more costly and likely to be slower.
- *Block* or *cache-line size*
  - Unit of data transfer between cache and main
    - With a larger cache line, more data is brought in cache with each miss.
      - This can improve the hit rate but also may bring low-utility data in.
- *Placement policy*.
  - Determining where an incoming cache line is stored
    - More flexible policies imply higher hardware cost and may or may not have performance benefits (due to more complex data location).
- *Replacement policy*.
  - Which of several existing cache blocks should be overwritten.
    - Typical policies: choosing a random or the least recently used block.
- *Write policy*.
  - *write-through*
    - Updates to cache words are immediately forwarded to main
  - *write-back* or *copy-back*
    - modified blocks are copied back to main if and when they must be replaced

Locality of Reference

- Why does cache work?

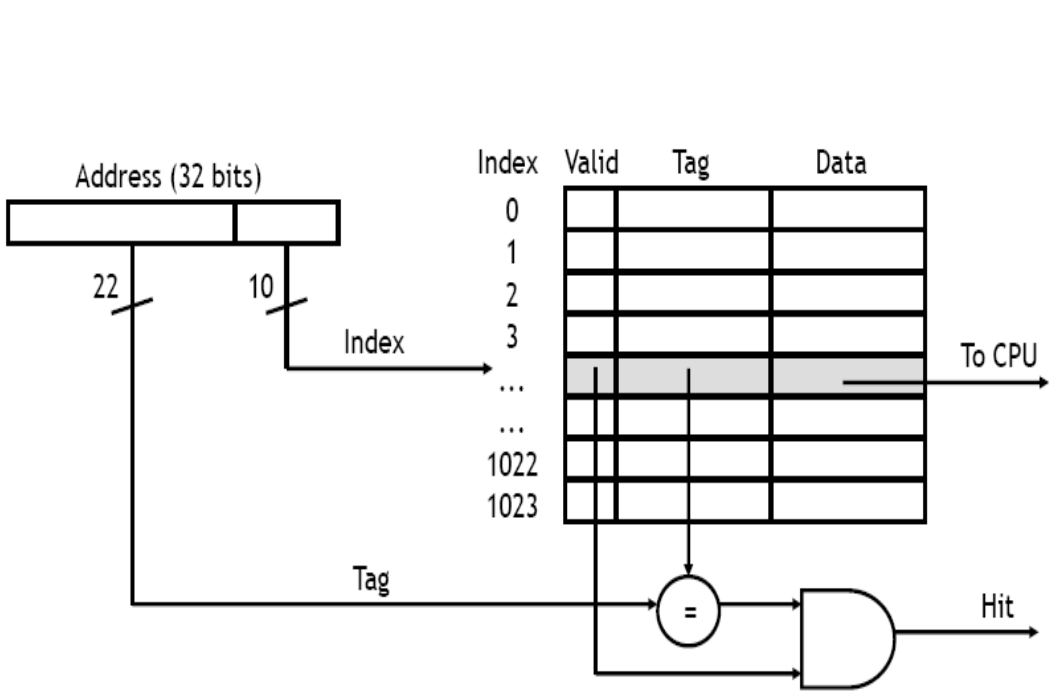
```
sum= 0;
for (i= 0; i< MAX; i++)
    sum= sum+ f(i);
```

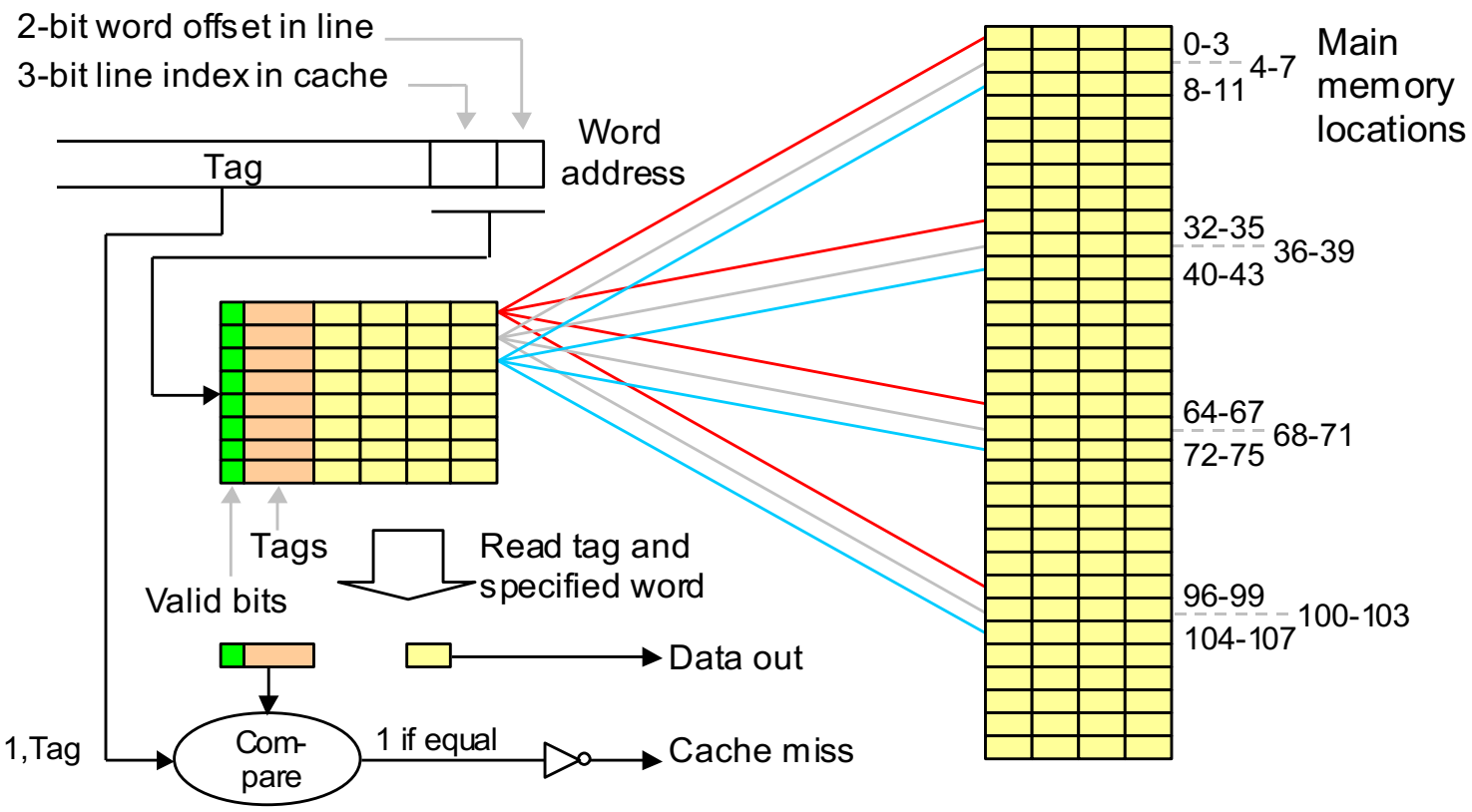
```
sum= 0;
for (i= 0; i< MAX; i++)
    sum= sum+ a(i);
```



Direct Mapped Cache

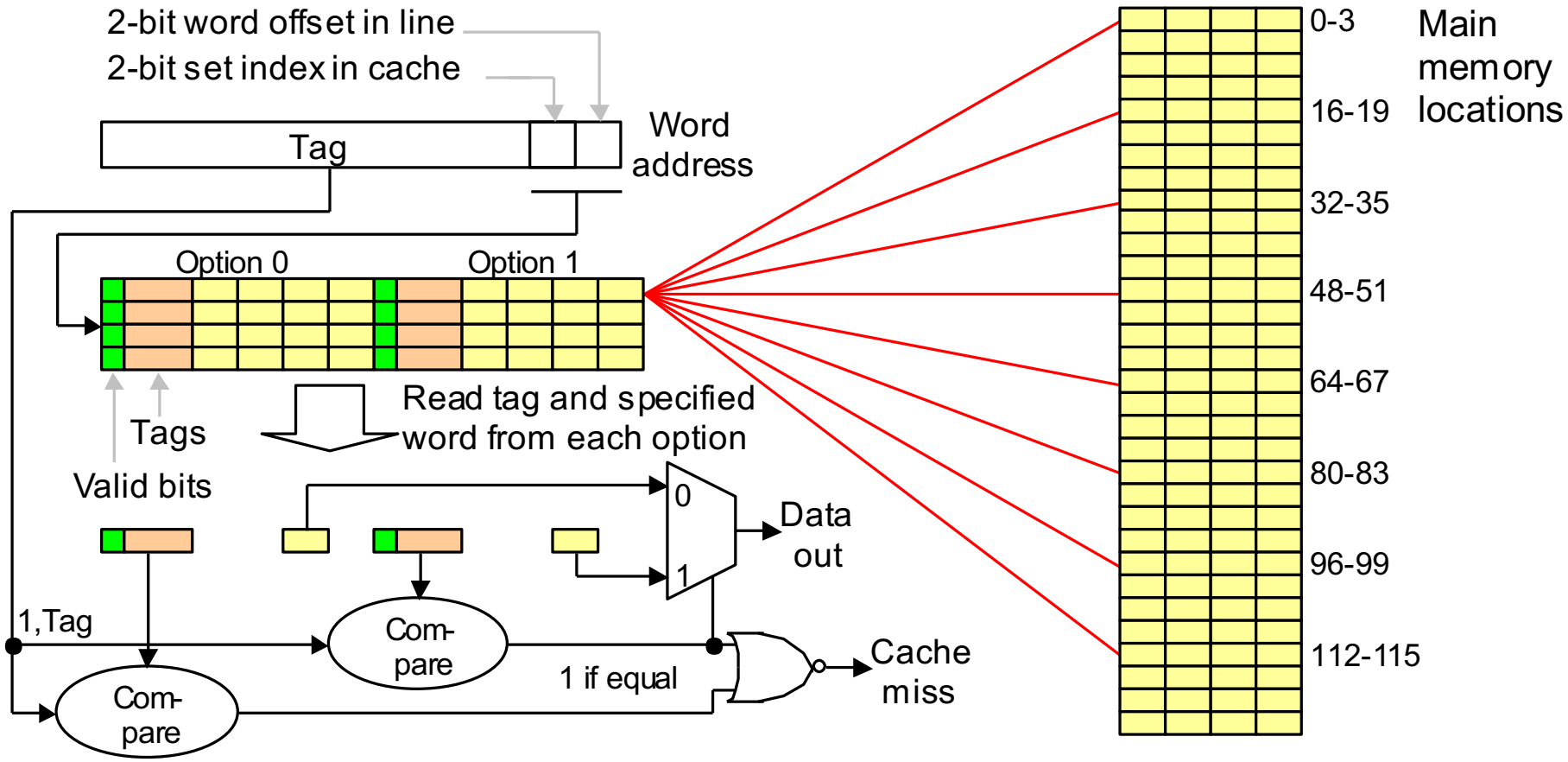
- Caches are divided into blocks, which may be of various sizes
  - Each main memory address maps to exactly one cache block
    - i mod 2^k





Set Associative Cache

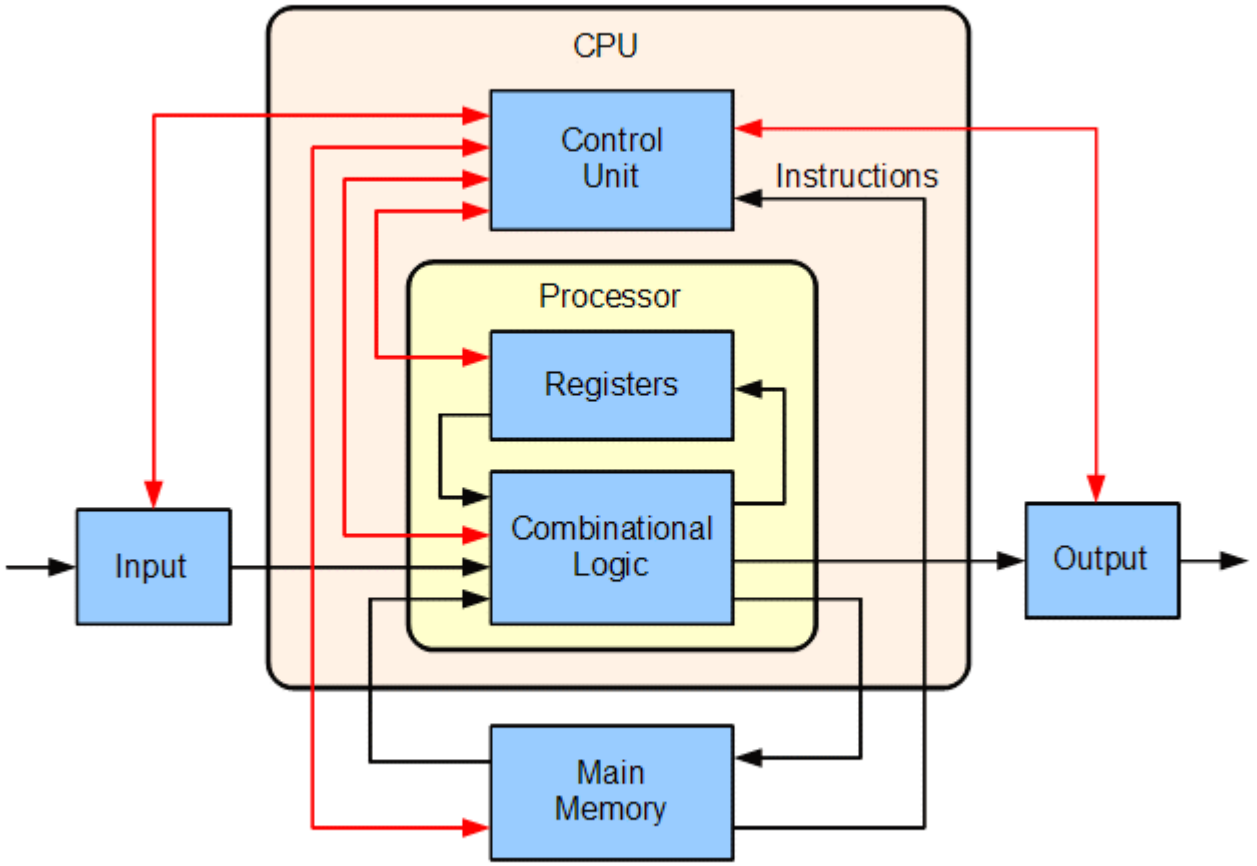
- Fully associative
  - Data to be stored in any unused cache block
- Two-way set-associative cache
  - Holds 32 words of data within 4-word lines and 2-line sets.



The Write Problem

- Architecture
  - Split cache: separate instruction and data caches (L1)
  - Unified cache: holds instructions and data (L1, L2, L3)
- What happens on a write
  - Write-through slows down the cache to allow main to catch up
  - Write-back or copy-back is less problematic, but still hurts performance due to two main memory accesses in some cases
    - Provide write buffers for the cache so that it does not have to wait for main memory to catch up

Simulator –gem5



Introduction

- Gem5 is the fusion of two projects
  - GEMS
    - detailed and flexible memory system model
    - Includes support for multiple cache coherence protocols and interconnect models
    - developed @ The University of Wisconsin Madison
  - M5
    - Highly configurable simulation framework to support multiple ISAs, and diverse CPU models
    - developed @ The University of Michigan
- System simulator
  - Good support of complex components interactions (OS / CPU / Caches / Devices / …)
  - Accuracy depends on the model completeness
  - Lot of components available out-of-the-box (CPUs, memories, I/Os, …)

gem5

- Good for
  - Architectural exploration
    - Gem5 provides a fast and easy framework to interconnect hardware components and evaluate them !
    - Hardware/software performance evaluation ?
    - Gem5 has a good support of various ISA and allows for realistic HW/SW performance evaluation
- Not Good for
  - Hardware/software verification ?
    - RTL functional verification is much more accurate !
  - Software development and verification ?
    - Faster technologies are available through binary-translation (e.g. QEMU, OVP)

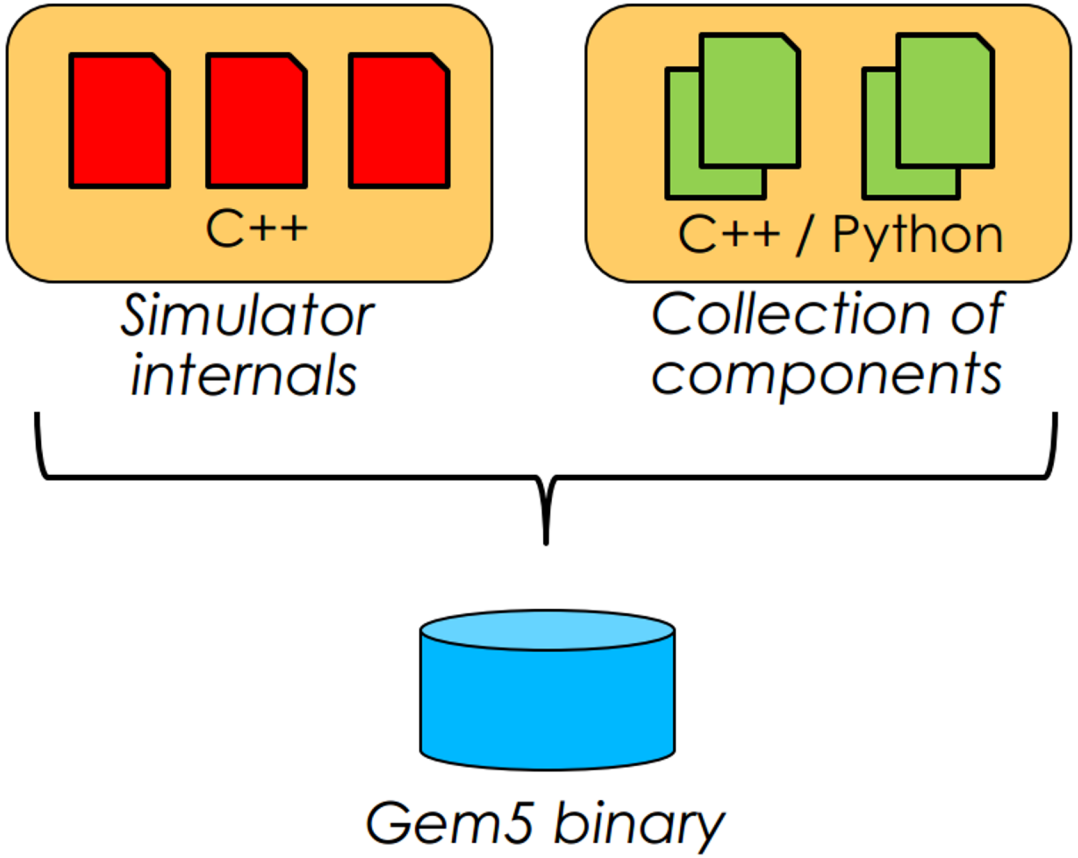
Simulation modes

- Full-system (FS)
  - Models bare-metal hardware
    - Includes the various specified devices, caches, …
  - Boots an entire OS from scratch
    - Gem5 can boot Linux (several variants) or Android out of-the-box

- Syscall Emulation (SE)
  - Runs a single static application
  - System calls are emulated or forwarded to the host OS
  - Lot of simplifications (address translation, scheduling, no pthread …)

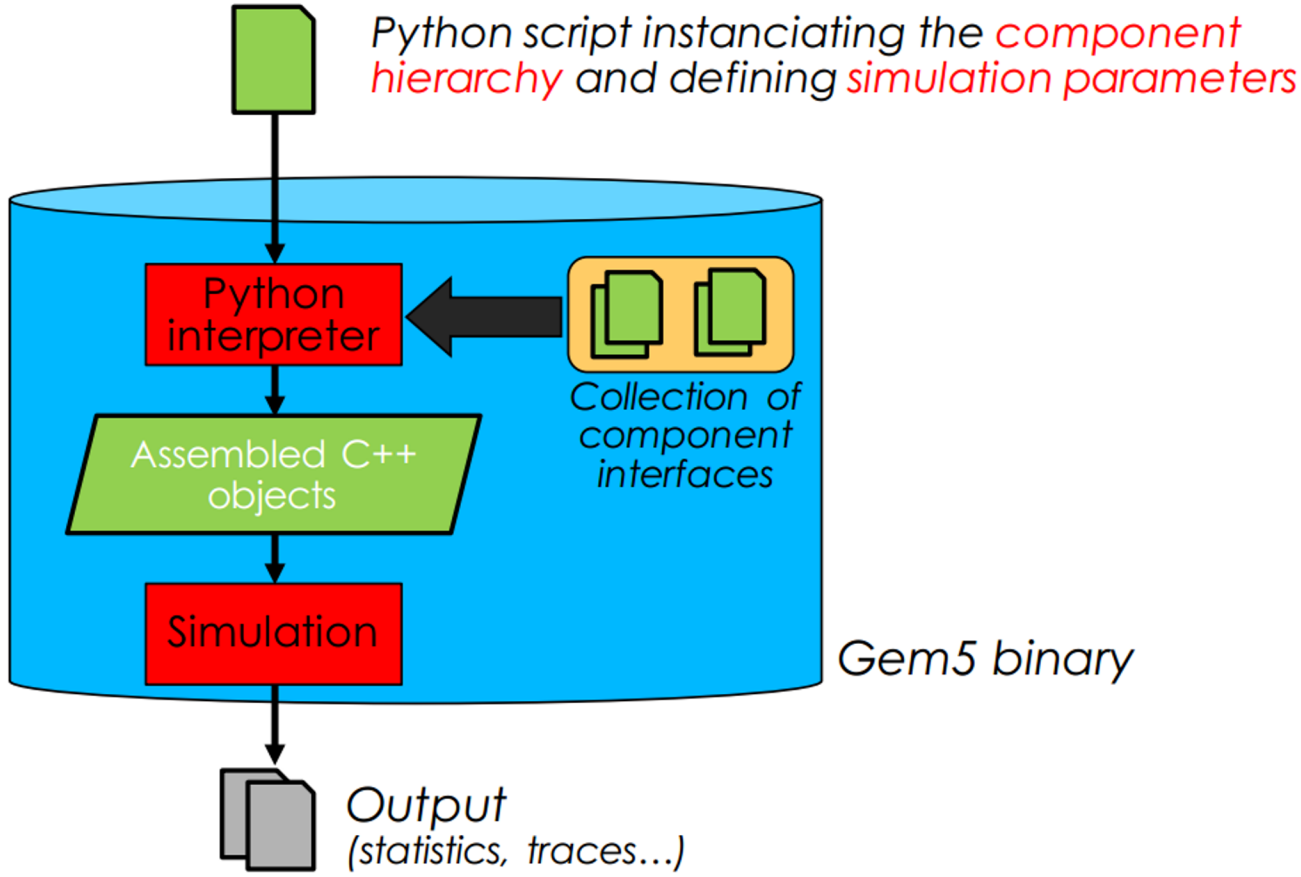
Behind the scene of a simulation

Compilation of the simulator



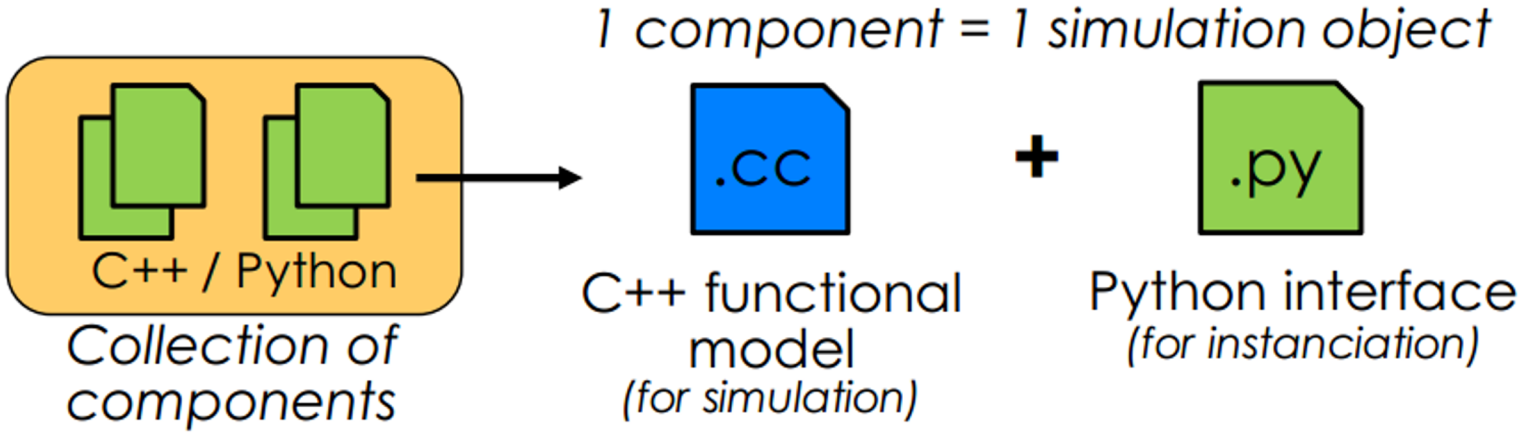
Behind the scene of a simulation

Compilation of the simulator

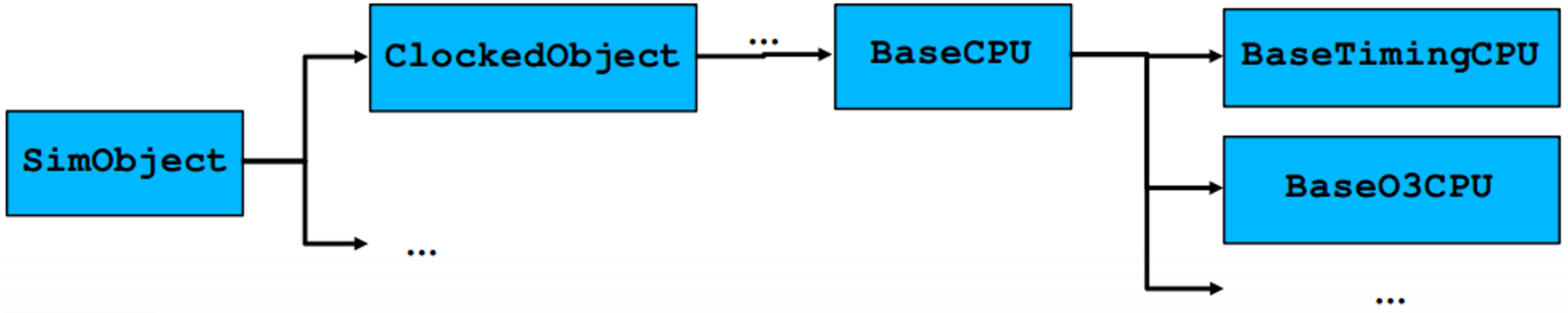


What's under the hood ?

Simulation objects



- SimObjects follow a strict C++ class hierarchy for easier extension with code reuse



What's under the hood ?
Events

- Gem5 is event-driven
  - Discrete event timing model

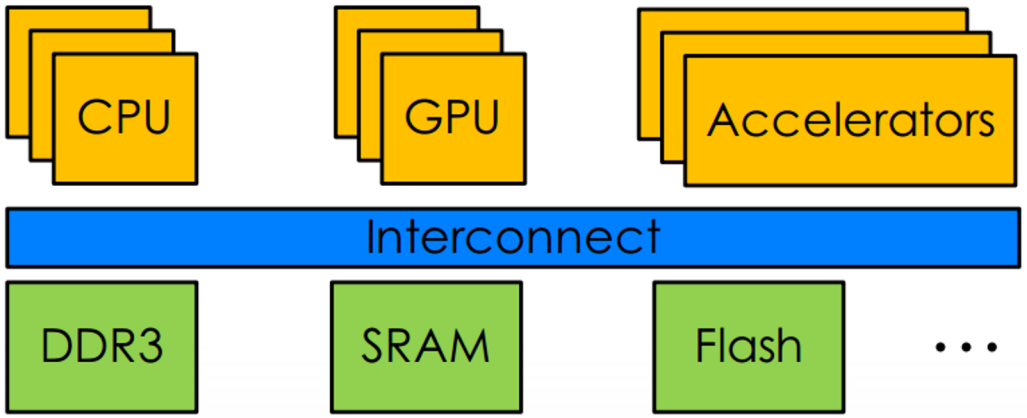
- Simulation objects schedule events for the next cycle after a specific time elapsed

What's under the hood ?
CPU Models

- Supports: Alpha, ARM, MIPS, Power, SPARC, and x86
- Configurable CPU models : Supports 3 CPU models
  - Simple Atomic/Timing
    - Fast CPU model
  - InOrder
    - Detailed pipelined in-order CPU model
  - O3
    - Detailed pipelined out-of-order CPU model
- Supports a domain specific language to represent ISA details

What's under the hood ?
Memory System

- Models a system running heterogeneous applications
  - running on heterogeneous processing tiles
  - using heterogeneous memories and interconnect



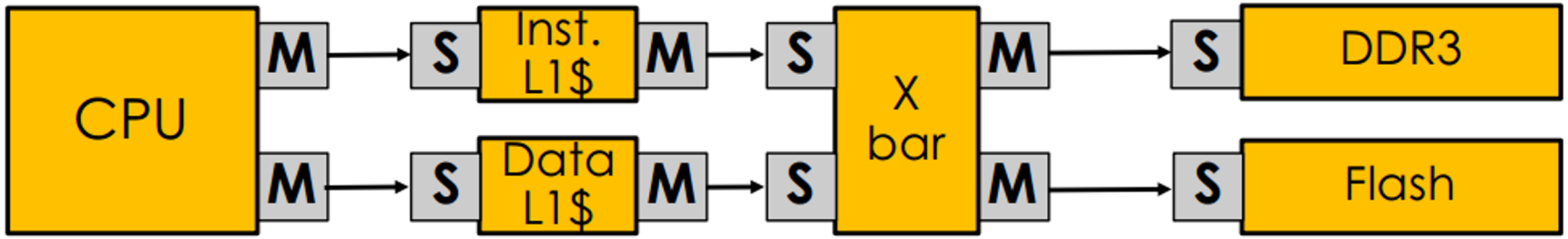
What's under the hood ?
Memory System

- Two memory systems in Gem5
  - Classic
    - All components instantiated in a hierarchy along with CPUs, etc.
    - Only MOESI coherence protocol
    - Fast, but less detailed than Ruby
  - Ruby
    - Detailed simulation model of various cache hierarchies
    - Various cache coherence protocols (MESI, MOESI, …)
    - Interconnection networks

What's under the hood ?
Memory ports

- Memory ports are present on every MemObject
  - They model physical memory connections
  - You interconnect them during the hierarchy instantiation
    - E.g. a CPU data bus to a L1 cache
    - 1 master port always connects to 1 slave port

- Data is exchanged atomicly as packets



What's under the hood ?
Memory ports

- 3 types of transport interfaces for packets
  - Functional
    - Instantaneous in a single function call
    - Caches and memories are updated at once
  - Atomic
    - Instantaneous
    - Approximate latency, but no contention nor delay
  - Timing
    - Transaction split into multiple phases
    - Models all timing in the memory system

Nice feature

- Checkpointing
  - Snapshot the relevant system state and restore it later
- Fast-forward
  - Idea is to start the simulation in atomic mode and switch over to detailed mode for important simulation period