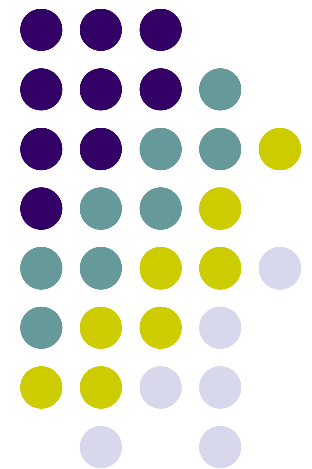
Pipelining, Instruction Level Parallelism and Memory in Processors

Advanced Topics

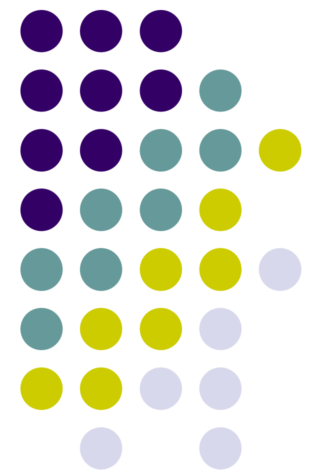
ICOM 4215

Computer Architecture and Organization

Fall 2010

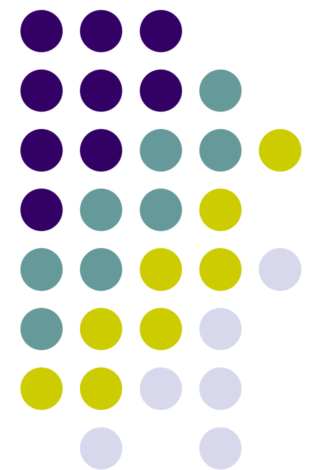


NOTE: The material for this lecture was taken from several different sources. They are listed in the corresponding sections

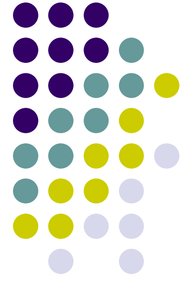


Pipelining

From the Hennessy and Patterson Book: Computer Organization and Design: the hardware software interface, 3rd edition

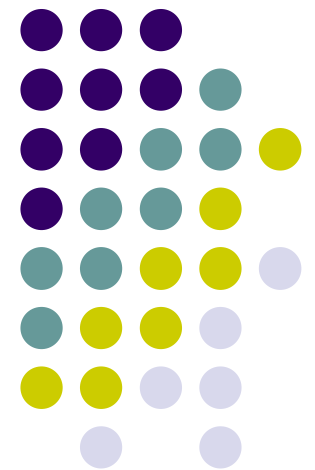


Overview

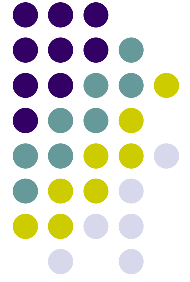


- Pipelining is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

Basic Concepts

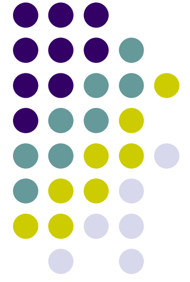


Making the Execution of Programs Faster

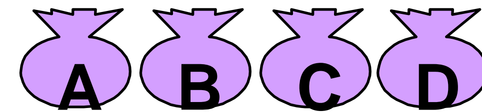


- Use faster circuit technology to build the processor and the main memory.
- Arrange the hardware so that more than one operation can be performed at the same time.
- In the latter way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

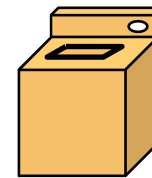
Traditional Pipeline Concept



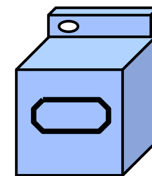
- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold



- Washer takes 30 minutes

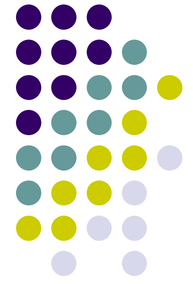


- Dryer takes 40 minutes



- “Folder” takes 20 minutes

Traditional Pipeline Concept



6 PM 7 8 9 10 11 Midnight

Time

30 40 20 30 40 20 30 40 20 30 40 20

A

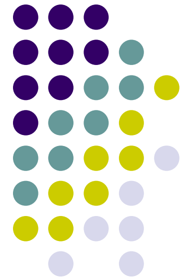
B

C

D

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Traditional Pipeline Concept



6 PM 7 8 9 10 11 Midnight

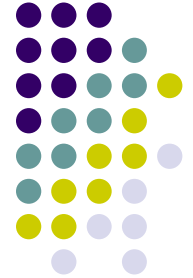
Time

30 40 40 40 40 20

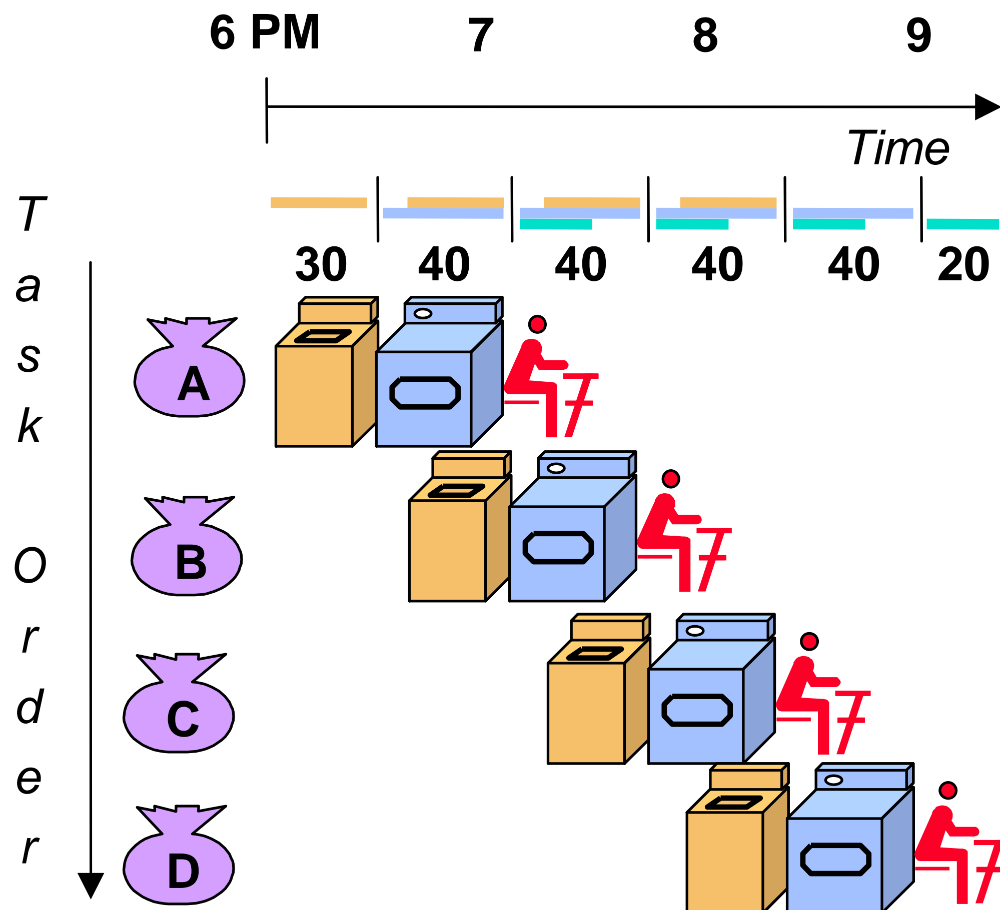
Task Order

A B C D

- Pipelined laundry takes 3.5 hours for 4 loads

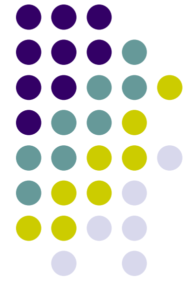


Traditional Pipeline Concept

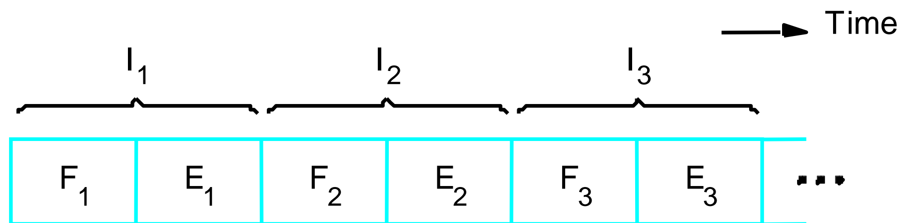


- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences

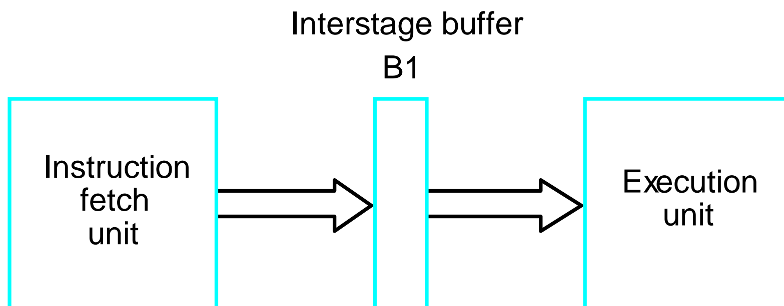
Use the Idea of Pipelining in a Computer



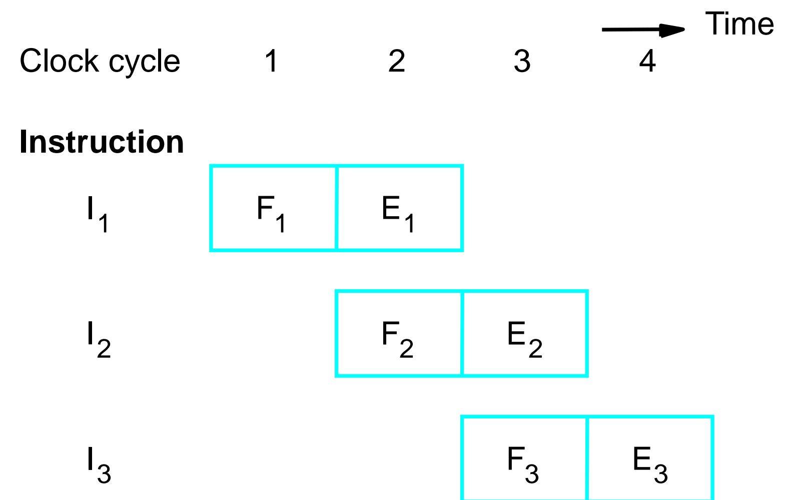
Fetch + Execution



(a) Sequential execution



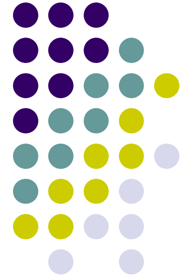
(b) Hardware organization



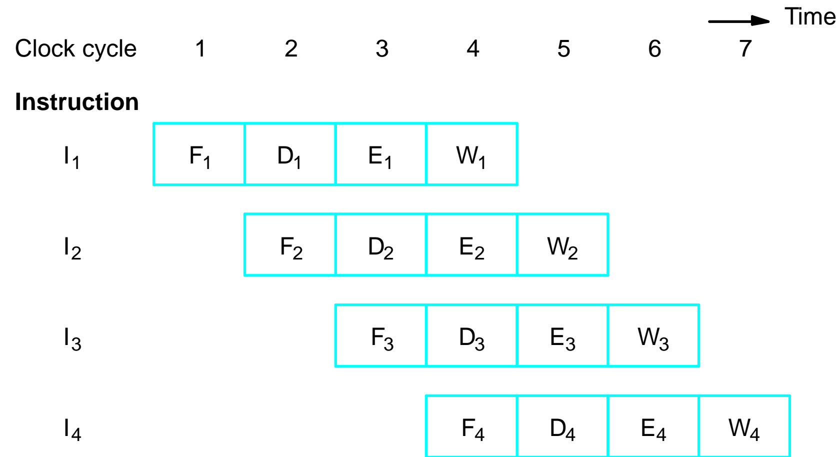
(c) Pipelined execution

Figure 8.1. Basic idea of instruction pipelining.

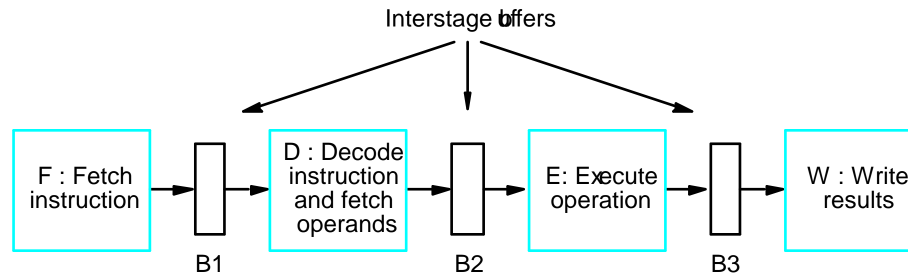
Use the Idea of Pipelining in a Computer



Fetch + Decode
+ Execution + Write



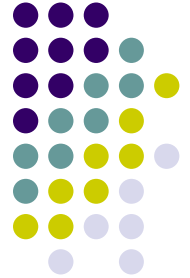
(a) Instruction execution divided into four steps



(b) Hardware organization

Figure 8.2. A 4-stage pipeline.

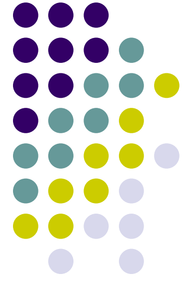
Ideal Pipelining



Cycle: Instr:	1	2	3	4	5	6	7	8	9	1 0	1 1	1 2	1 3
i	F	D	E	W									
i+1		F	D	E	W								
i+2			F	D	E	W							
i+3				F	D	E	W						
i+4					F	D	E	W					

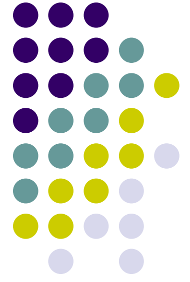
Taken from: Lecture notes based on set created by Mark Hill and John P. Shen
Updated by Mikko Lipasti, ECE/CS 552: Chapter 6: Pipelining

Pipeline Performance



- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
- Unfortunately, this is not true.

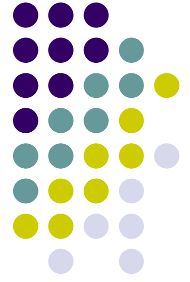
Role of Cache Memory



- Each pipeline stage is expected to complete in one clock cycle.
  - The clock period should be long enough to let the slowest pipeline stage to complete.
  - Faster stages can only wait for the slowest one to complete.
- Since main **memory** is **very slow** compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.
- Fortunately, we have cache.

(We'll talk about memory in a moment)

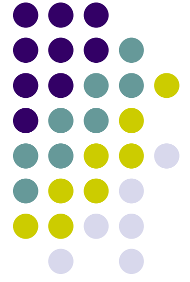
Pipelining Idealisms



- Uniform subcomputations
  - Can pipeline into stages with equal delay
- Identical computations
  - Can fill pipeline with identical work
- Independent computations
  - No relationships between work units
- Are these practical?
  - No, but can get close enough to get significant speedup

Taken from: Lecture notes based on set created by Mark Hill and John P. Shen
Updated by Mikko Lipasti, ECE/CS 552: Chapter 6: Pipelining

Pipeline Performance

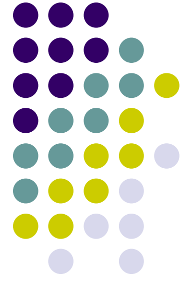


Time →

Instruction \ Clock cycle	1	2	3	4	5	6	7	8	9
I_1	F_1	D_1	E_1	W_1					
I_2		F_2	D_2	E_2	E_2	E_2	W_2		
I_3			F_3	D_3			E_3	W_3	
I_4				F_4			D_4	E_4	W_4
I_5							F_5	D_5	E_5

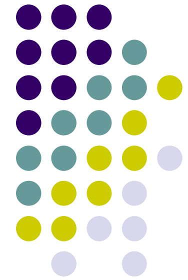
Figure 8.3 Effect of an execution operation taking more than one clock cycle

Pipeline Performance

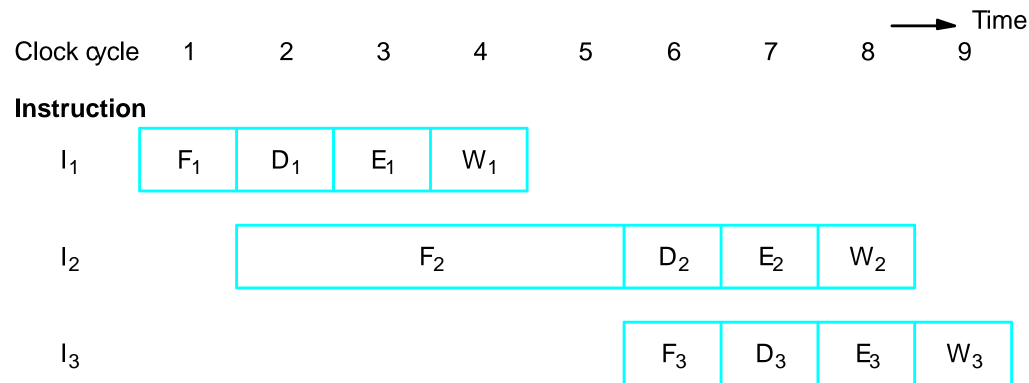


- The previous pipeline is said to have been stalled for two clock cycles.
- Any condition that causes a pipeline to stall is called a hazard.
- Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.
- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.
- Structural hazard – the situation when two instructions require the use of a given hardware resource at the same time.

Pipeline Performance



Instruction hazard



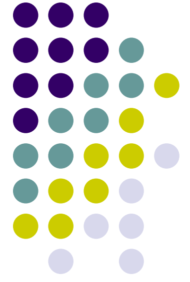
(a) Instruction execution steps in successive clock cycles

(b) Function performed by each processor stage in successive clock cycles

Idle periods – stalls (bubbles)

Figure 8.4. Pipeline stall caused by a cache miss in F2.

Pipeline Performance



Structural hazard

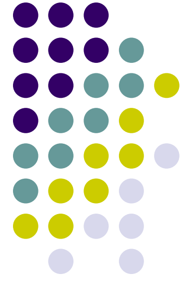
Load X(R1), R2

Time →

Clock cycle	1	2	3	4	5	6	7
Instruction							
I_1	F_1	D_1	E_1	W_1			
I_2 (Load)		F_2	D_2	E_2	M_2	W_2	
I_3			F_3	D_3	E_3		W_3
I_4				F_4	D_4		E_4
I_5						F_5	D_5

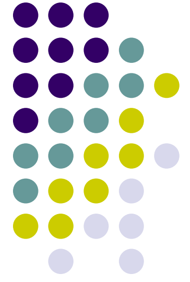
Figure 8.5. Effect of a Load instruction on pipeline timing.

Pipeline Performance



- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- We need to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.

Program Data Dependences



- True dependence (RAW)
  - j cannot execute until i produces its result

$$Wr(i) \cap Rd(j) \neq \phi$$

- Anti-dependence (WAR)
  - j cannot write its result until i has read its sources

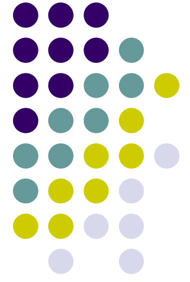
$$Rd(i) \cap Wr(j) \neq \phi$$

- Output dependence (WAW)
  - j cannot write its result until i has written its result

$$Wr(i) \cap Wr(j) \neq \phi$$

Taken from: Lecture notes based on set created by Mark Hill and John P. Shen
Updated by Mikko Lipasti, ECE/CS 552: Chapter 6: Pipelining

Data Hazards



- We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially.
- Hazard occurs

  $A \leftarrow 3 + A$

  $B \leftarrow 4 \times A$
- No hazard

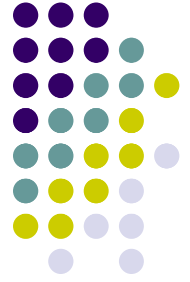
  $A \leftarrow 5 \times C$

  $B \leftarrow 20 + C$
- When two operations depend on each other, they must be executed sequentially in the correct order.
- Another example:

  Mul R2, R3, R4

  Add R5, R4, R6

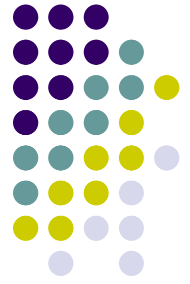
Control Dependences



- Conditional branches
  - Branch must execute to determine which instruction to fetch next
  - A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction.
  - The decision to branch cannot be made until the execution of that instruction has been completed.
  - Branch instructions represent about 20% of the dynamic instruction count of most programs.

Taken from: Lecture notes based on set created by Mark Hill and John P. Shen
Updated by Mikko Lipasti, ECE/CS 552: Chapter 6: Pipelining

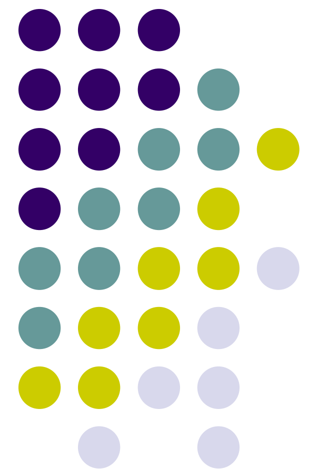
Resolution of Pipeline Hazards



- Pipeline hazards
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard resolution
  - Static: compiler/programmer guarantees correctness
  - Dynamic: hardware performs checks at runtime
- Pipeline interlock
  - Hardware mechanism for dynamic hazard resolution
  - Must detect and enforce dependences at runtime

Taken from: Lecture notes based on set created by Mark Hill and John P. Shen
Updated by Mikko Lipasti, ECE/CS 552: Chapter 6: Pipelining

Instruction Level Parallelism



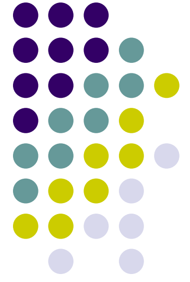
Taken from

Soner Onder

Michigan Technological University, Houghton MI and Notes from Hennessy and Patterson's book

http://www.eecs.berkeley.edu/~pattrsn

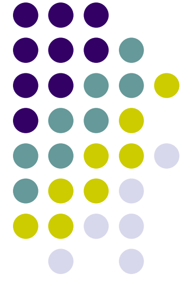
Instruction Level Parallelism



- Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance

- 2 approaches to exploit ILP:
  1) Rely on hardware to help discover and exploit the parallelism dynamically, and
  2) Rely on software technology to find parallelism, statically at compile-time

Forms of parallelism



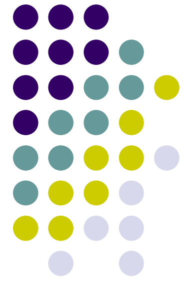
- Process-level
  - How do we exploit it? What are the challenges?
- Thread-level
  - How do we exploit it? What are the challenges?
- Loop-level
  - What is really loop level parallelism? What percentage of a program's time is spent inside loops?
- Instruction-level
  - Lowest level

Coarse grain

Human intervention?

Fine Grain

Instruction-level parallelism (ILP)



- Briefly, ability to execute more than one instruction simultaneously.
- In order to achieve this goal, we should not have dependencies among instructions which are executing in parallel:

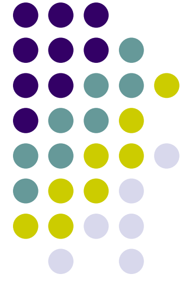
  - H/W terminology Data hazards

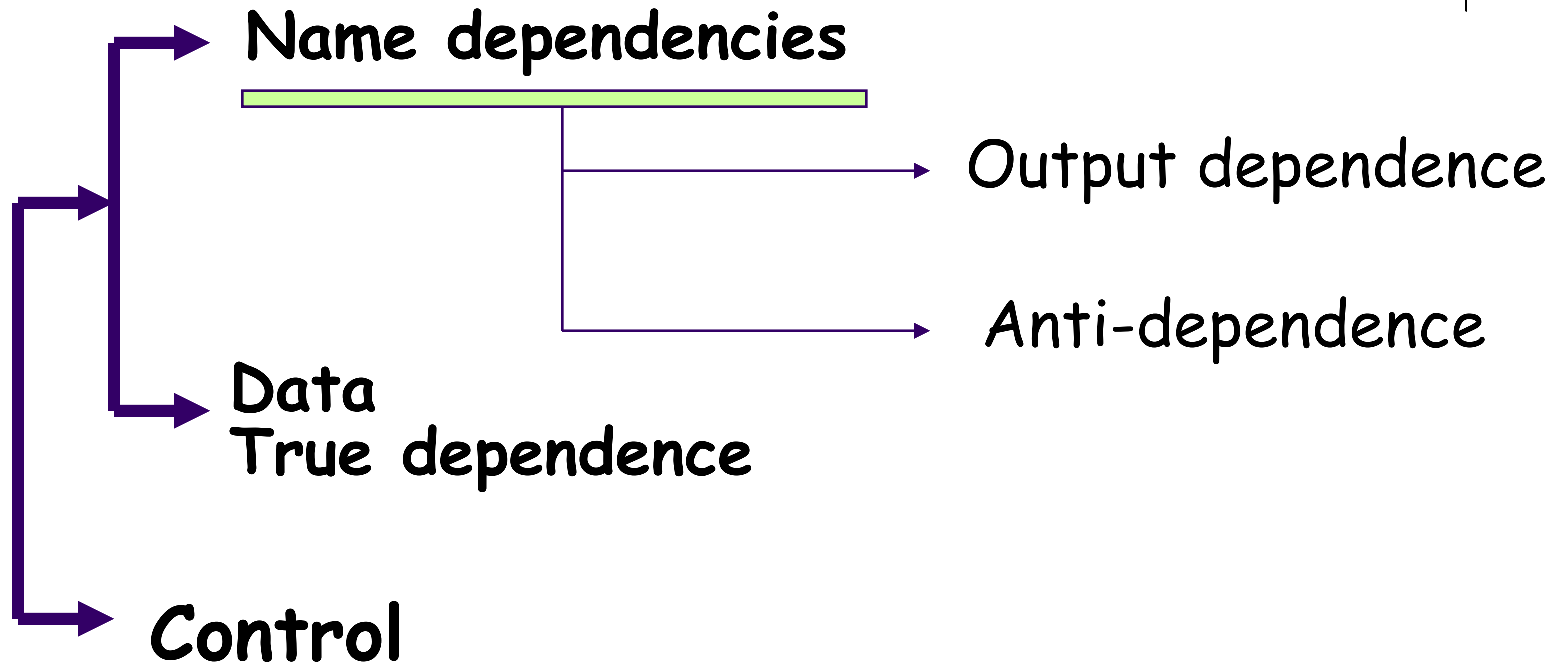
    (I.e. RAW WAR WAW)

  - S/W terminology Data dependencies

    Name & true dependencies

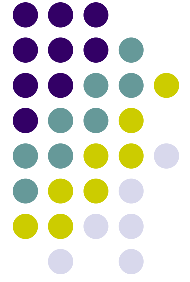
Dependencies





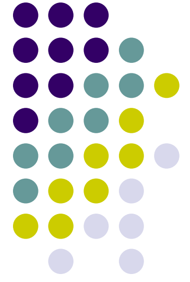
Do you remember the hazards they may lead to?

Increasing ILP



- Techniques
  - Loop unrolling
  - Static Branch Prediction
    - Compiler
  - Dynamic Branch Prediction
    - At runtime
  - Dynamic Scheduling – Tomasulo's Algorithm
    - Register renaming

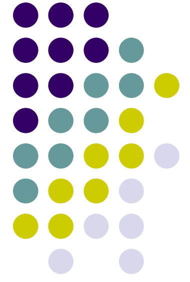
Memory Hierarchy



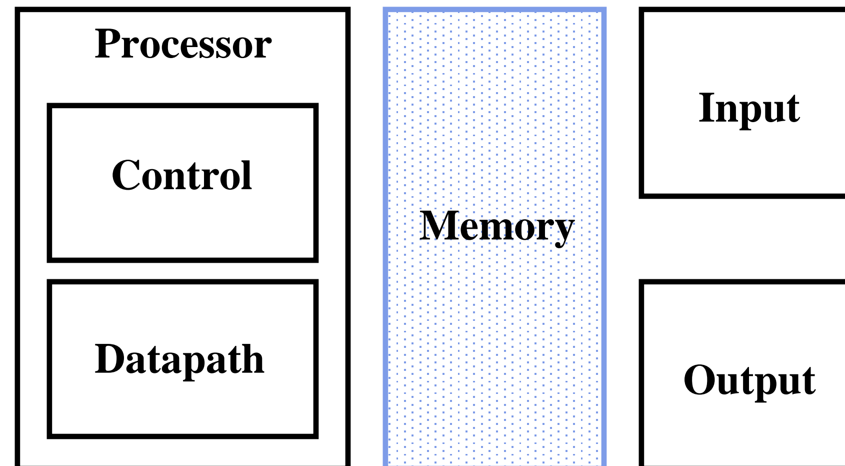
The following sources are used for preparing the slides on memory:

- Lecture 14 from the course Computer architecture ECE 201 by Professor Mike Schulte.
- Lecture 4 from William Stallings, *Computer Organization and Architecture*, Prentice Hall; 6th edition, July 15, 2002.
- Lecture 6 from the course Systems Architectures II by Professors Jeremy R. Johnson and Anatole D. Ruslanov
- Some of figures are from Computer Organization and Design: The Hardware/Software Approach, Third Edition, by David Patterson and John Hennessy, are copyrighted material (COPYRIGHT 2004 MORGAN KAUFMANN PUBLISHERS, INC. ALL RIGHTS RESERVED).

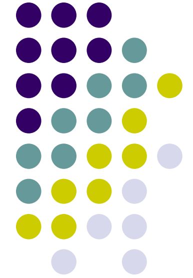
The Big Picture



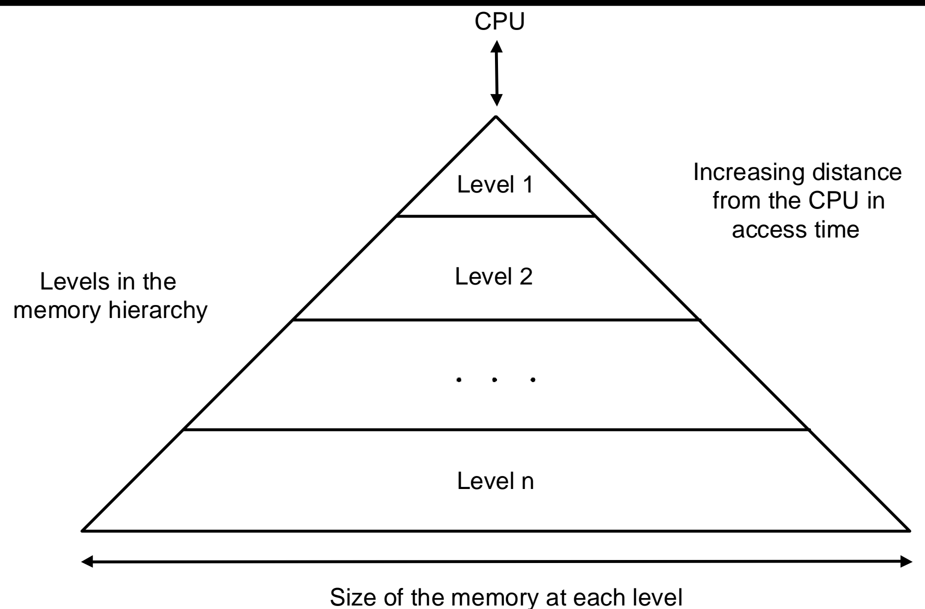
- The Five Classic Components of a Computer
- Memory is usually implemented as:
  - Dynamic Random Access Memory (DRAM) - for main memory
  - Static Random Access Memory (SRAM) - for cache

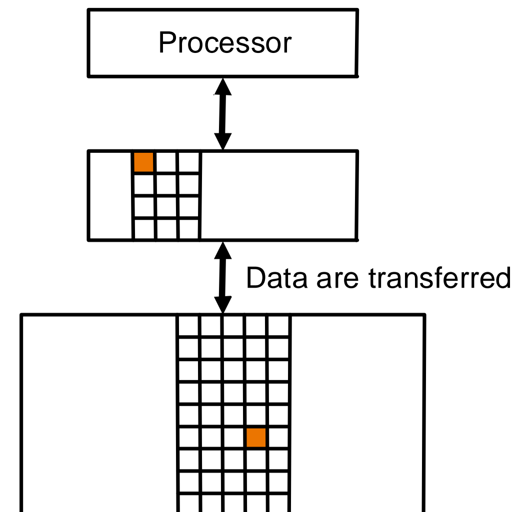


Memory Hierarchy

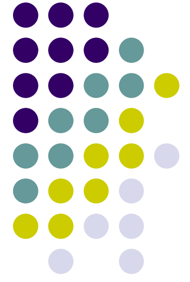


Memory technology	Typical access time	$ per GB in 2004
SRAM	0.5–5 ns	$4000–$10,000
DRAM	50–70 ns	$100–$200
Magnetic disk	5,000,000–20,000,000 ns	$0.50–$2



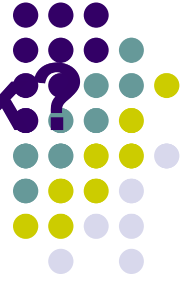


Memory



- SRAM:
  - Value is stored on a pair of inverting gates
  - Very fast but takes up more space than DRAM (4 to 6 transistors)
- DRAM:
  - Value is stored as a charge on capacitor (must be refreshed)
  - Very small but slower than SRAM (factor of 5 to 10)

Memory Hierarchy: How Does it Work?

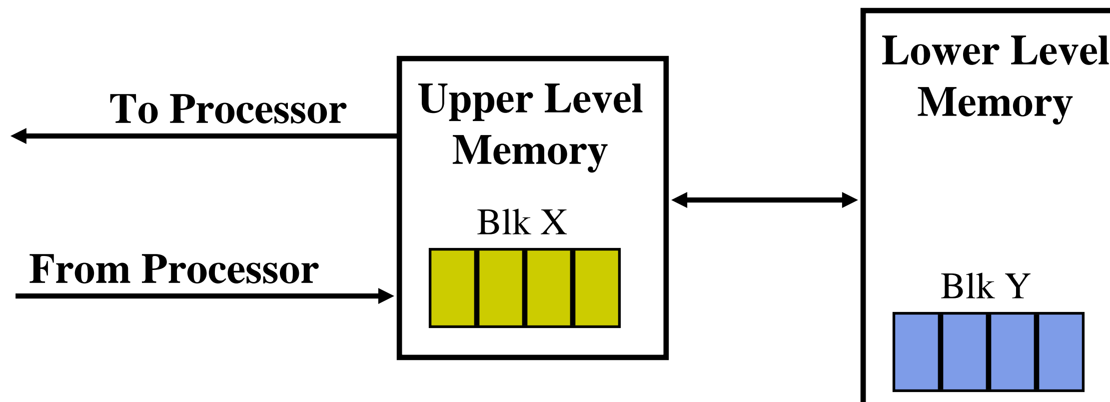


- Temporal Locality (Locality in Time):

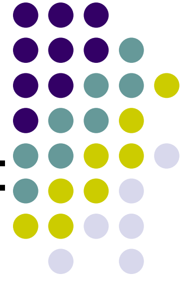
  => Keep most recently accessed data items closer to the processor

- Spatial Locality (Locality in Space):

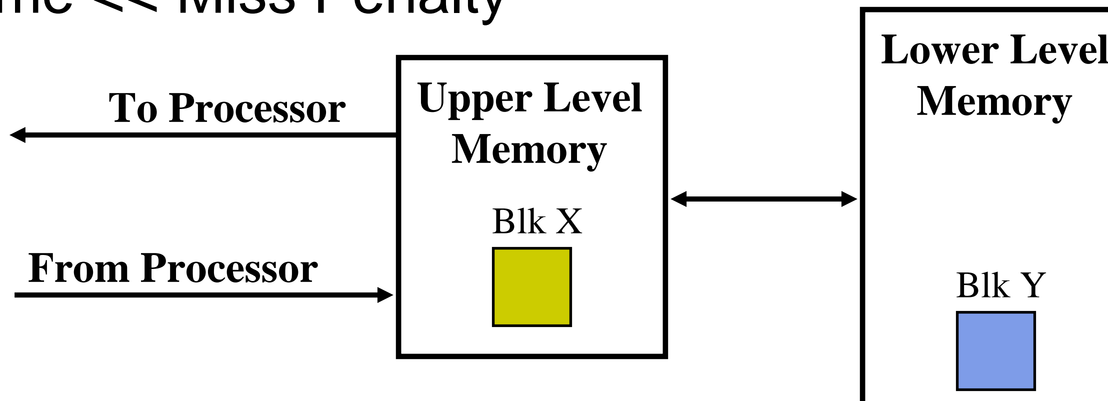
  => Move blocks consists of contiguous words to the upper levels



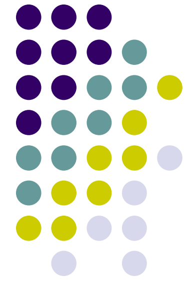
Memory Hierarchy: Terminology



- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of
    RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level +
    Time to deliver the block the processor
- Hit Time << Miss Penalty



Memory Hierarchy of a Modern Computer System



- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.

Processor

Control

Datapath

Registers

On-Chip Cache

Second Level Cache (SRAM)

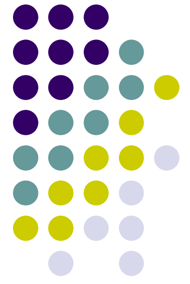
Main Memory (DRAM)

Secondary Storage (Disk)

Tertiary Storage (Tape)

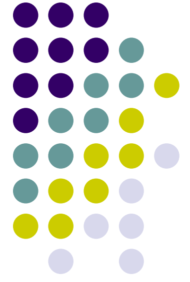
Speed (ns):	1s	10s	100s	10,000,000s (10s ms)	10,000,000,000s (10s sec)
Size (bytes):	100s	Ks	Ms	Gs	Ts

General Principles of Memory

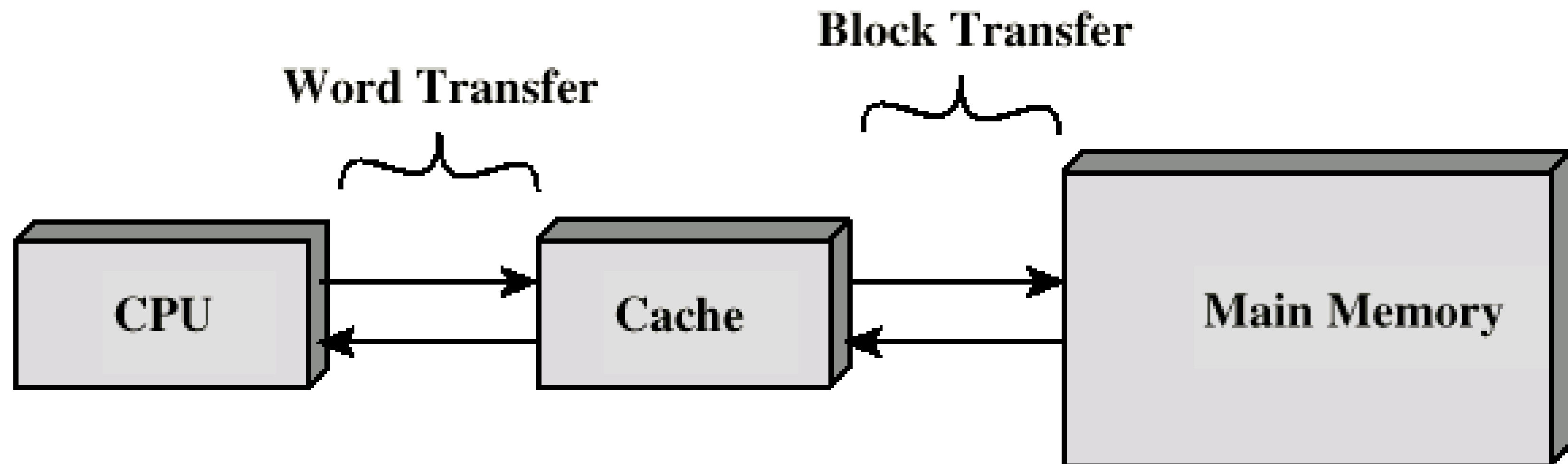


- Locality
  - *Temporal Locality*: referenced memory is likely to be referenced again soon (e.g. code within a loop)
  - *Spatial Locality*: memory close to referenced memory is likely to be referenced soon (e.g., data in a sequentially access array)
- Locality + smaller HW is faster = memory hierarchy
  - *Levels*: each smaller, faster, more expensive/byte than level below
  - *Inclusive*: data found in upper level also found in the lower level

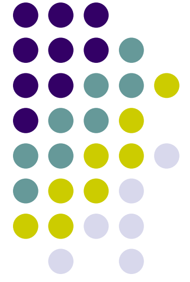
Cache



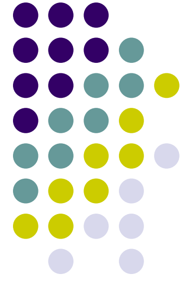
- Small amount of fast memory
- Sits between normal main memory and CPU
- May be located on CPU chip or module



Cache operation - overview



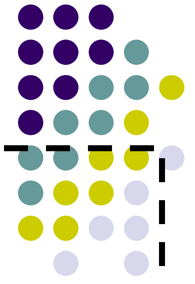
- CPU requests contents of memory location
- Check cache for this data
- If present, get from cache (fast)
- If not present, read required block from main memory to cache
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot



Cache Design

- Size
- Mapping Function
- Replacement Algorithm
- Write Policy
- Block Size
- Number of Caches

Relationship of Caches and Pipeline



Memory

I-$

D-$

Next PC

Next SEQ PC

Adder

4

Address

Memory

IF/ID

Adder

MUX

Zero?

RS1

RS2

Reg File

Sign Extend

Imm

RD

ID/EX

MUX

ALU

RD

EX/MEM

Data Memory

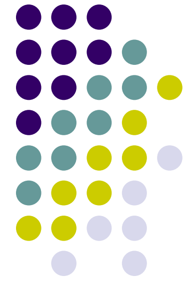
RD

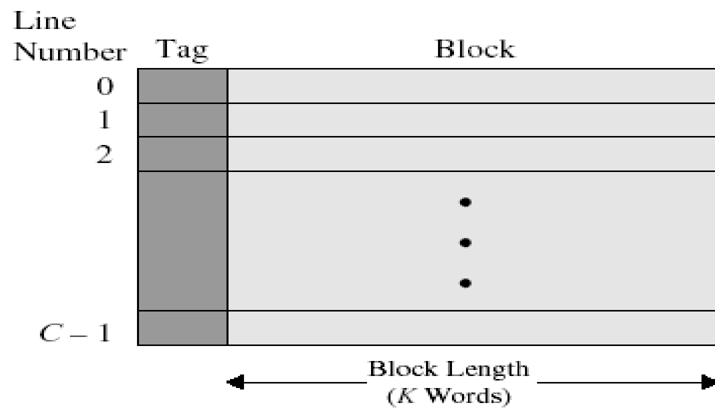
MEM/WB

MUX

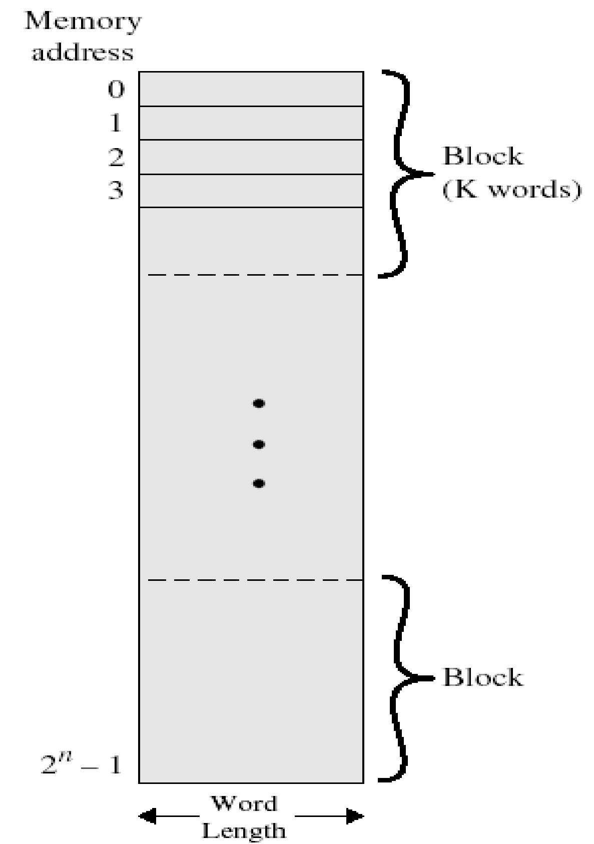
WB Data

Cache/memory structure



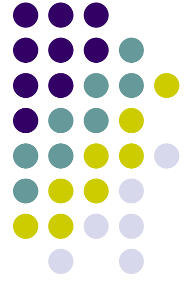


(a) Cache



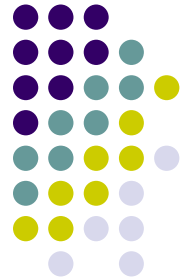
(b) Main memory

Block Placement



- Direct Mapped: Each block has only one place that it can appear in the cache.
- Fully associative: Each block can be placed anywhere in the cache.
- Set associative: Each block can be placed in a restricted set of places in the cache.
    - If there are n blocks in a set, the cache placement is called n-way set associative

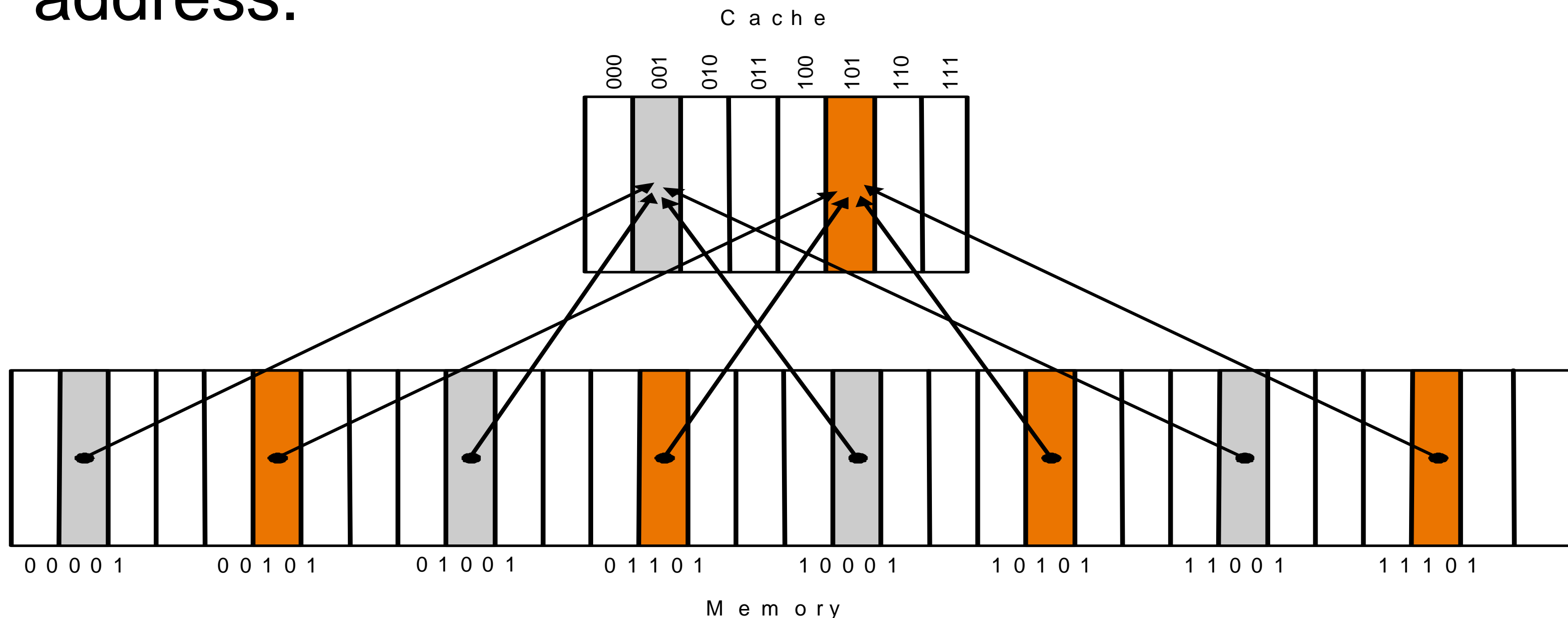
Example: Direct Mapped Cache



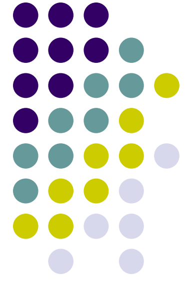
- Mapping: memory mapped to one location in cache:

  (Block address) mod (Number of blocks in cache)

- Number of blocks is typically a power of two, i.e., cache location obtained from low-order bits of address.



Summary



- Pipelines
  - Increase throughput of instructions
  - May have stalls
- ILP
  - Exploit multiple units and pipeline
  - Avoid dependencies
- Memory Hierarchy and cache
  - Support ILP and pipelines