Instruction Pipelining

Lecture 13

Instruction level Pipelining

- Pipelining**:**
  - An implementation technique where multiple instructions are overlapped in execution.
  - The computer pipeline is divided in **stages**.
    - Each stage completes a part of an instruction in parallel.
  - Pipelining does not decrease the time for individual instruction execution.
    - Instead, it increases instruction throughput.
  - The **throughput** of the instruction pipeline is determined by how often an instruction exits the pipeline.

The speedup from pipelining

- The speedup in Pipelining: ideally equal to the number of pipeline stages.
  - A four-stage pipeline yields a four-fold speedup in the completion rate versus single-cycle,
  - This speedup is possible because:
    - The more pipeline stages in a processor, the more instructions the processor can work on simultaneously and the more instructions it can complete in a given period of time.
- If the stages of a pipeline are not balanced and one stage is slower than another, the entire throughput of the pipeline is affected.

Characteristics of Pipelining

- In terms of a pipeline within a CPU:
  - Each instruction is broken up into different stages
  - Ideally if each stage is balanced (all stages are ready to start at the same time and take an equal amount of time to execute)
  - The implementation of pipelining has the effect of reducing the average instruction time, therefore reducing the average CPI.
    - EX: If each instruction in a microprocessor takes 5 clock cycles (unpipelined) and we have a 4 stage pipeline, the ideal average CPI with the pipeline will be 1.25 .

Speedup for one instruction

- For one instruction, there is no speedup?
  - The speedup comes with the *parallel* execution of multiple instructions.
  - While the first instruction is decoding, the second can be fetched;
  - while the first instruction is performing the ALU instruction, the second can be decoding, and the third can be fetched, etc.

Speedup for one instruction

- Suppose one instruction takes four cycles to execute in a nonpipelined CPU:
  - one cycle to fetch the instruction, one cycle to decode the instruction, one cycle to perform the ALU operation, and one cycle to store the result.
- In a CPU with a 4-stage pipeline, that instruction still takes four cycles to execute

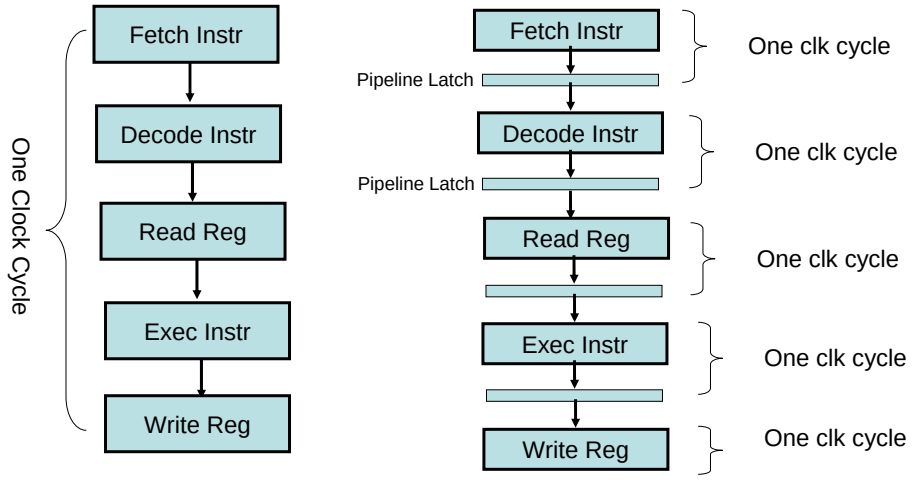
Pipeline Latency & Throughput

- Pipeline throughput:
  - Instructions completed per second.
    - the number of instructions that the processor completes *each clock cycle*
- Pipeline latency:
  - How long does it take to execute a single instruction in the pipeline.
    - The amount of time that a single instruction takes to pass through the pipeline.
- While pipelining can reduce the a processor's cycle time and increase the instruction throughput, it increases the latency of the processor

Instruction-Level Pipelining

- Some CPUs divide the fetch-decode-execute cycle into smaller steps.
  - These smaller steps can often be executed in parallel to increase throughput.
  - Such parallel execution is called *instruction-level parallelism*.
- This term is sometimes abbreviated *ILP* in the literature.

Pipeline Vs non-pipeline processor



Instruction flow in a pipelined processor

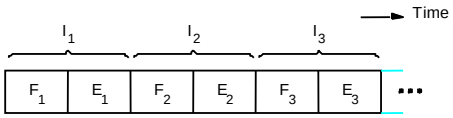
Cycle

	1	2	3	4	5	6	7
IF	Instr 1	Instr 2	Instr 3				
ID							
RR							
EX							
WB							

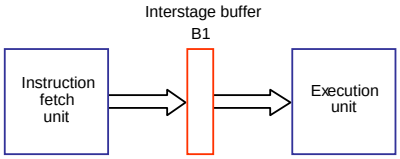
- Cycle time of pipeline processors
  - Dependent on four factors
    - The cycle time of unpipelined processor
    - The number of pipeline stages
    - Division of stages
    - Latency of pipeline latches
      - Cycle Time $_{pipelined}$ = (cycle time $_{unpipelined}$ / number of stages) + latency

Use the Idea of Pipelining in a Computer

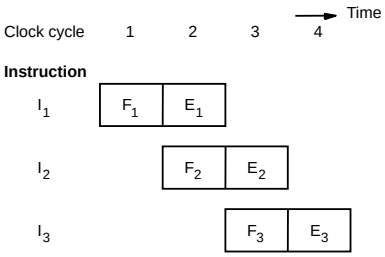
Fetch + Execution



(a) Sequential execution



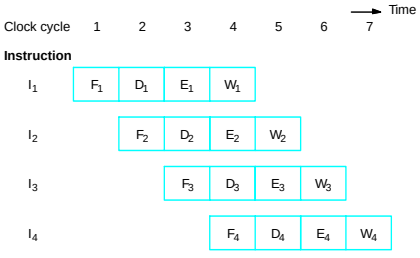
(b) Hardware organization



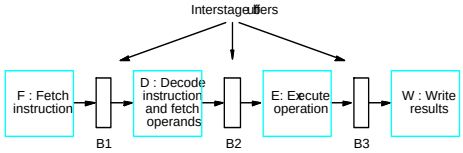
(c) Pipelined execution

Use the Idea of Pipelining in a Computer

Fetch + Decode
+ Execution + Write



(a) Instruction execution divided into four steps



(b) Hardware organization

IPL - Example

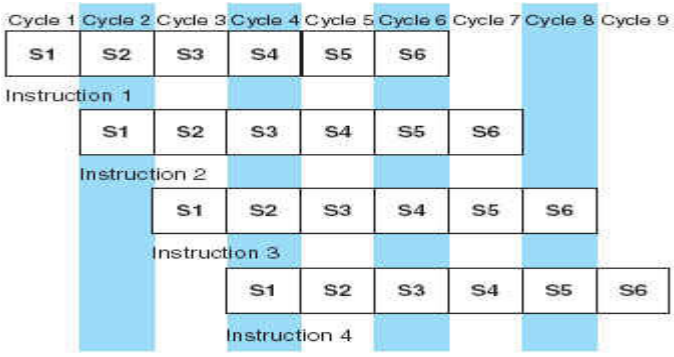
- Suppose a fetch-decode-execute cycle were broken into the following smaller steps:

1. Fetch instruction.	4. Fetch operands.
2. Decode opcode.	5. Execute instruction.
3. Calculate effective address of operands.	6. Store result.

- Suppose we have a six-stage pipeline.
  - S1 fetches the instruction,
  - S2 decodes it,
  - S3 determines the address of the operands,
  - S4 fetches operands,
  - S5 executes the instruction, and
  - S6 stores the result.

Instruction-Level Pipelining

- For every clock cycle, one small step is carried out, and the stages are overlapped.



S1. Fetch instruction.	S4. Fetch operands.
S2. Decode opcode.	S5. Execute.
S3. Calculate effective address of operands.	S6. Store result.

Pipeline performance

- **Under ideal conditions**
  - **Time per instruction =**Time per instruction on non-pipelined machine / Number of pipe stages
  - Speedup = No of stages
- BUT
  - Stages are not balanced
    - Overhead (latches imposes propagation delay)
- Consider 10ns clock and 4 instructions, 4 cycle are required for ALU(40%) and Branch(20%) operation and 5 cycle for Memory operation(40%)
  - Overhead due to clock skew is 1ns
  - What is the speedup due to pipelining ?
    - Avg Instruction Execution Time = 10 [(.4 +.2) 4 + .4 x 5] = 44 ns
    - Average Time per instruction for pipeline machine = 11 ns
    - Speed up = 44/11 = 4

Pipelining

- Suppose we have a *k*-stage pipeline:
  - Assume:
    - clock cycle time is *tp* (it takes *tp* time per stage)
    - we have *n* instructions (T*asks*) to process
  - Task 1 (*T*1) requires ***k* x *tp*** time to complete
    - The remaining ***n* - 1** tasks emerge from the pipeline one per cycle,
    - So the total time for these tasks of $(n - 1)tp$
- Therefore, to complete *n* tasks using a *k*-stage pipeline requires:
  - $(k \times tp) + (n - 1)tp = (k + n - 1)tp$

Pipelining

- Let's calculate the speedup while using a pipeline
  - Without a pipeline:
  - The time required is $n\ t_n$ cycles,
    - where $t_n = k$ x tp.
  - Therefore, the speedup (time without a pipeline divided by the time using a pipeline) is:

$$\text{speedup } S = \frac{nt_n}{(k+n-1)t_p}$$

The theoretical speedup, *k*, is the number of stages in the pipeline.

Speedup example

- A nonpipelined system takes 200ns to process a task.
- The same task can be processed in a 5-segment pipeline (k)with a clock cycle of 40ns (tp).
  - What is the speedup ratio of the pipeline for 200 tasks (n).
  - What is the maximum speedup that could be achieved with the pipeline unit over the nonpipelined unit?
- SpeedUp = (200ns x 200)/((5+200-1)(40ns)) = 40000/8160 = 4.91
  - Max SpeedUp = 5

Real world Example

- The 8086 – 80486 are single-stage pipeline architectures
- The Pentium had two parallel five-stage pipelines, called the U pipe and the V pipe, to execute instructions.
  - Stages for these pipelines include Prefetch, Instruction Decode, Address Generation, Execute, and Write Back.
- The Pentium II increased the number of stages to 12,
  - Stages Prefetch, Length Decode, Instruction Decode, Rename/Resource Allocation, UOP Scheduling/Dispatch, Execution, Write Back, and Retirement.
- The Pentium III increased the stages to 14.
- The Pentium IV to 24.

Cache Memories

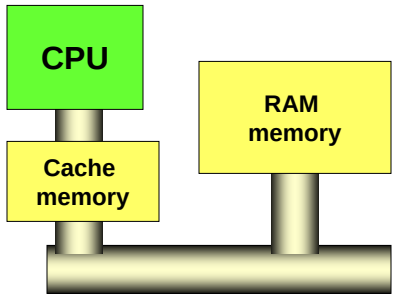
- Cache:
  - Small, Fast SRAM based memory, sit between large memory and the CPU
    - May be located on CPU chip or module
  - Contains the copy of the portion of main memory
    - Holds the operands and instructions most likely to be used by the CPU
  - Temporary store for often used instructions
    - Level 1 cache is built within the CPU (internal)
    - Level 2 cache may be on chip or nearby (external)
  - Faster for CPU to access than main memory
- The cache is placed both physically closer and logically closer to the CPU than the main memory.

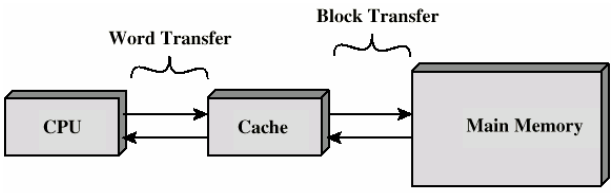
Locality

- PRINCIPAL OF LOCALITY:
  - The tendency to reference data items that are near other recently referenced data items, or that were recently referenced themselves.
- TEMPORAL LOCALITY:
  - Memory location that is referenced once is likely to be **referenced multiple times** in near future.
- SPATIAL LOCALITY:
  - Memory location that is referenced once, then the program is likely to be **reference a nearby memory location** in near future.
- Principle of locality helped to speed up main memory access by introducing cache memory that hold blocks of the most recently referenced instructions and data items.

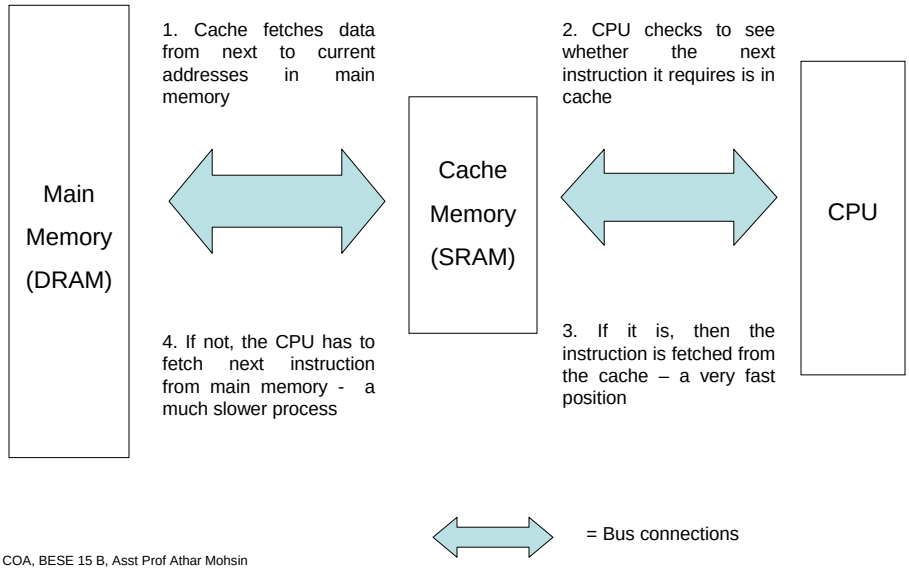
Cache Memory

- This memory sits VERY close to the CPU so that it can be accessed quickly
- Cache is:
  - Faster
  - More expensive
  - Smaller
  - Holds fewer "things" than RAM memory





The operation of cache memory

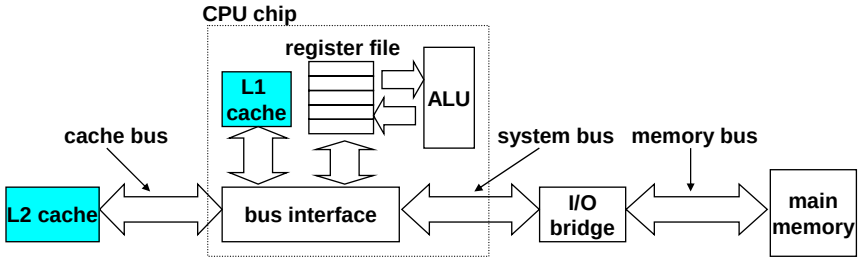


Memory Hierarchy Basis

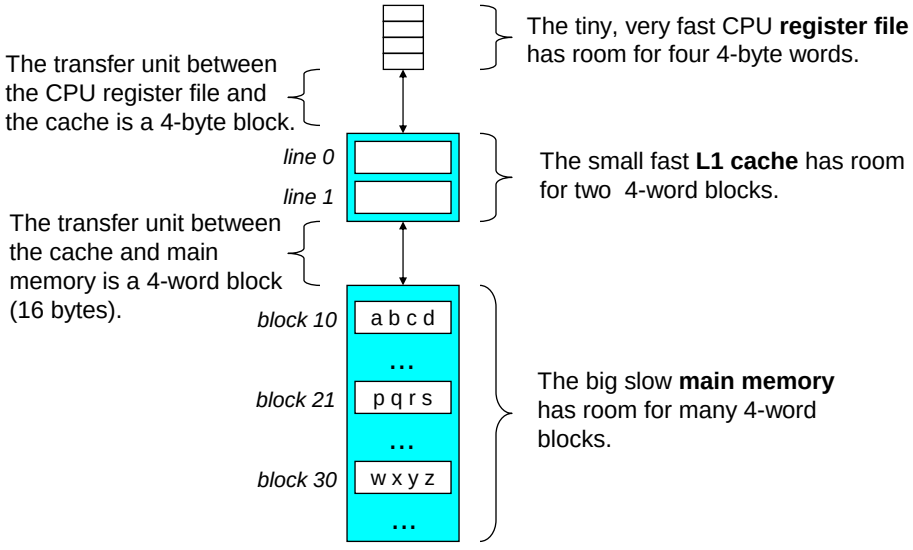
- Disk contains everything.
- When Processor needs something, bring it into all higher levels of memory.
- Cache contains copies of data in memory that are being used.
- Memory contains copies of data on disk that are being used.
- Entire idea is based on Temporal Locality:
  - if we use it now, we'll want to use it again soon

Cache Memories

- Hold frequently accessed blocks of main memory
  - CPU looks first for data in L1, then in L2, then in main memory.
- Typical bus structure:



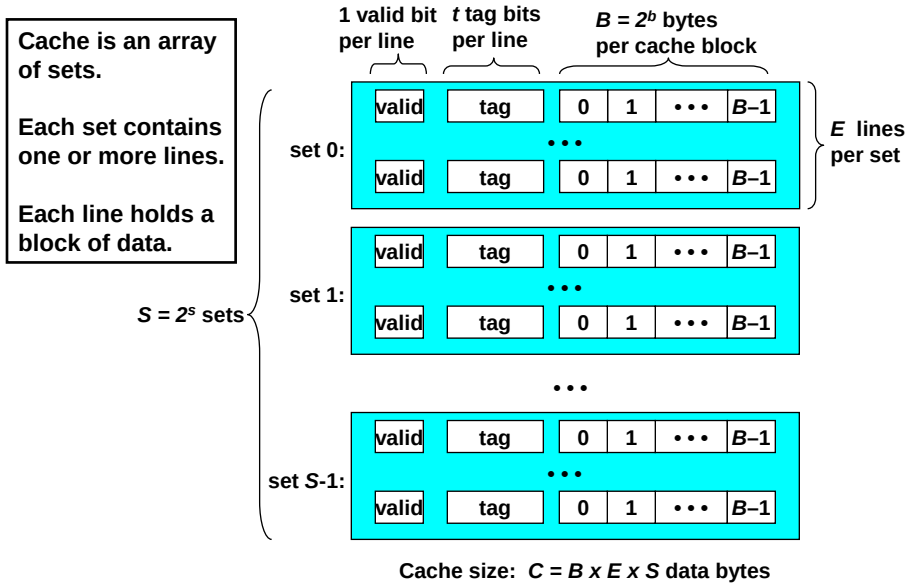
Inserting an L1 Cache Between the CPU and Main Memory



Cache Terminologies

- Cache Line/Block
  - Cache memory is subdivided into cache lines
  - Cache Lines / Blocks:
    - The smallest unit of memory than can be transferred between the main memory and the cache.
- Tag / Index
  - Every address field consists of two primary parts:
    - A dynamic (tag) which contains the higher address bits, and
    - A static (index) which contains the lower address bits
      - The Dynamic Tag may be modified during run-time while the Static Tag one is fixed.

General Org of a Cache Memory



Cache Terminologies

- Valid / Dirty bit
  - a valid bit is needed to indicate whether or not the slot holds a line that belongs to the program being executed
  - a dirty bit keeps track of whether or not a line has been modified while it is in the cache.
- Cache Hit:
  - A request to read from memory, which can satisfy from the cache without using the main memory.
- Cache Miss:
  - A request to read from memory, which cannot be satisfied from the cache, for which the main memory has to be consulted.