Pipelining

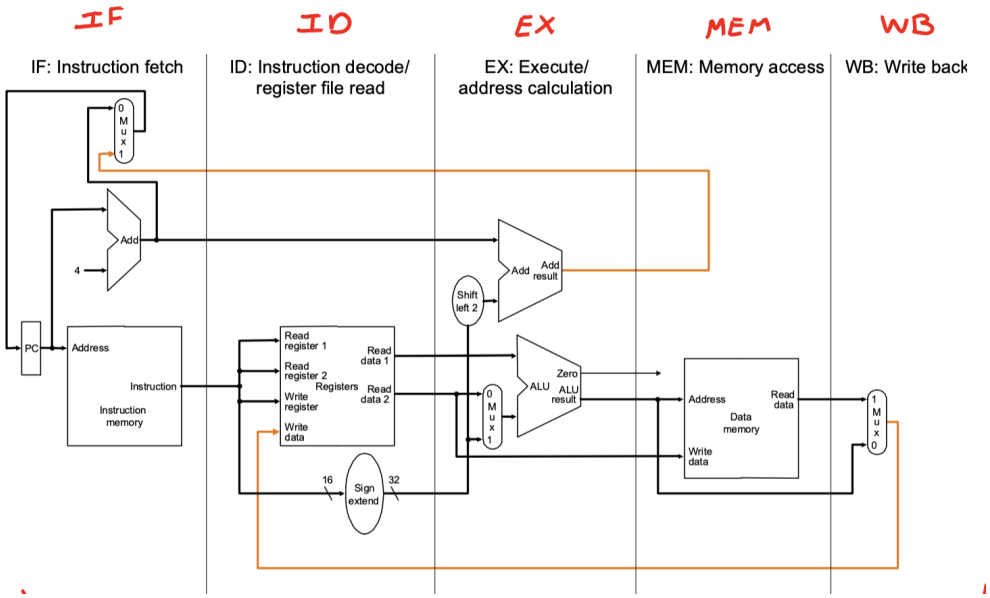
- What is pipelining?

  Pipelining is a technique used in CPU design to *improve instruction processing*. It allows the CPU to overlap the execution of multiple instructions, breaking them down into smaller stages and processing them concurrently. This increases the overall efficiency of the CPU by reducing the time wasted during instruction execution.

  a. It starts a new instance after every clock
  b. The time to complete a given instance does not change
  c. Multiple instances working at the same time
  d. An instance completes every clock

  *single cycle = 1000 ns vs pipeline = 208ns

- Understand the pipeline datapath (5 Stages)



1. IF (Instruction Fetch)
   a. The PC increases to get the next instruction from the instruction memory
   b. The instruction is accessed in the instruction memory at the address specified by the PC
   c. The fetched instruction is stored in the instruction register (IR)
   d. The PC is updated to point to the next instruction.

2. ID (Instruction Decode)

a. The instruction stored in the instruction register (IR) is decoded to determine the operation to be performed and the operands involved.
b. The register file is accessed to read the values of the source registers required for the instruction.

3. EX (Execute)
    a. The Arithmetic Logic Unit (ALU) performs the required arithmetic or logical operation on the operands.
    b. This stage also handles branch instructions; the target address for the branch is calculated based on the condition specified in the instruction.

4. MEM (Memory access)
    a. If the instruction involves memory access (lw/sw), the calculated memory address is sent to the data memory, and data is read from or written to that address.

5. WB (Write-back)
    a. The results of the previous stage (either the ALU computation or data read from memory) are written back to the register file.
    b. The register file is updated with the new values, which can be used by subsequent instructions.

* The pipeline stages have registers in between them for: data storage, synchronization, flow control, time control, and hazard control,

- <u>Compute CPI for pipeline processors</u>
  CPI = Total Cycles / Average Instruction Count

1. Example Problem #1: Consider a pipeline processor with four stages: IF, ID, EX, and WB. The critical path length is four cycles, and during a workload, 500 instructions are executed. However, due to data hazards, there are 200 pipeline stalls during the execution of the workload.

    To Compute CPI:

    a. Determine the number of pipeline stages: 4 stages.
    b. Identify the critical path: 4 cycles.

c. Calculate the total number of cycles: The total number of cycles is the sum of the critical path cycles and the pipeline stalls. Total Cycles = Critical Path + Pipeline Stalls = 4 cycles + 200 stalls = 204 cycles.
d. Determine the average instruction count: 500 instructions.
e. Compute the CPI: CPI = Total Cycles / Average Instruction Count = 204 cycles / 500 instructions = 0.408 CPI.

2. Example Problem #2: Suppose you have a pipeline processor with five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). The critical path in this pipeline is three cycles. During a workload, 100 instructions are executed. There are no pipeline stalls.

   To Compute CPI:

   a. Determine the number of pipeline stages: 5 stages.
   b. Identify the critical path: 3 cycles.
   c. Calculate the total number of cycles: Since there are no pipeline stalls, the total number of cycles is the same as the critical path, which is 3 cycles.
   d. Determine the average instruction count: 100 instructions.
   e. Compute the CPI: CPI = Total Cycles / Average Instruction Count = 3 cycles / 100 instructions = 0.03 CPI.

- <u>Branch hazards & Squashing</u>
  1. **Control Hazards**
     a. Occurs =
        i. the *branch's target address is determined later* in the pipeline (meaning where the program will jump to if the condition is met)
        ii. this could mean earlier stages may have already tried to fetch or process things using the incorrect address
        iii. *BASICALLY mis -matches between what is fetched and what needs to be executed*
     b. Solution = *stalling* the pipeline until the branch condition is evaluated and the target address is known (delaying until known)
  2. **Data Hazards**
     a. Occurs =
        i. if a branch instruction *depends on the result of a previous instruction* that has not yet completed execution

ii. THINK loading then immediately using it in a branch comparison statement after (Load R1, [R2] then Branch if R1 > 0)

b. Solution = involves *stalling* the pipeline or introducing *bypassing* mechanisms to ensure that the correct data is available when needed by the branch instruction

3. Squashing

a. Technique =

i. A technique used to handle branch hazards; it involves *discarding the instructions in the pipeline that have been fetched* after a branch instruction whose outcome has been determined (also known as flushing or canceling)

ii. THINK when a branch instruction is executing, if it is true the instruction already fetched needs to be ignored or discarded since it has a new instruction address

b. Steps =

i. Determine branch outcome destination (is the branch taken or not)

ii. Determine the direction of branch taken (if taken, the instructions following it after in the pipe need to be discarded)

iii. Flushing to the pipeline (instructions are removed from the pipeline)

iv. Fetching correct instructions

NOTE: squashing ensures that the pipeline correctly handles the change in control flow caused by the branch instruction… but introduces performance penalty since it discards wasted cycles

*** Squash instantiated before being thrown out vs Stall which is just being thrown out

- <u>Data hazards</u>

NOTE: A pipeline only works if it is read and written, at the very latest, in the same cycle.

1. Overview

a. Occurs = When data is not in the register file/memory, instead stuck in a pipeline register

b. Solution = Forwarding/Bypassing; copy the data from the pipeline register and replace the old value from the register file/memory (decided through a Forwarding Unit)

*** it *cannot* fix user-after load, only stalling can do this

2. **EX Hazards (Read-After-Write Hazards)**:
    a. Occurs =
        i. the instruction following depends on the result of the current instruction which is still being executed
        ii. basically, the result of the *previous instruction is not yet available* in the register file or a destination register
        iii. THINK when an instruction writes to a register/memory location, but the next instruction needs to read from the same register/memory location
    b. Solution =
        i. *stalling* the pipeline by inserting a "no-operation" instruction until the data becomes available in the register file
        ii. *forwarding/bypassing* the data from the executing instruction to the subsequent instruction

3. **MEM Hazards (Write-After-Read Hazards, Read/Write):**
    a. Occurs =
        i. the instruction following depends on the result of the current instruction which is still being loaded/stored
        ii. basically, the result of the *previous instruction is not yet available* in the register file or a destination register
        iii. THINK when a load instruction reads/writes data from memory, but the next instruction needs to read/write from the same memory location
    b. Solution =
        i. involves *stalling* the pipeline by inserting "no-operation" instructions until the memory operation completes and the data becomes available
        ii. *forwarding/bypassing* the data from the memory unit to the subsequent instruction

- Forwarding & Stalls
    1. Forwarding (Bypassing)
        a. When the result of an instruction in an earlier pipeline stage (stored in a pipe register) to be *directly forwarded to the following instruction* that requires it (bypassing the need to wait for the result to be stored in the register file)

b. Typically involves multiplexers and additional control logic that detects hazards and determines when and where to forward the data
c. There are different forwarding paths based on the specific hazard types:
    i. EX to ID: The execution stage to the decode stage
    ii. WB to ID: The writing stage to the decode stage
    iii. MEM to ID: The memory stage to the decode stage
    iv. MEM to EX: The memory stage to the execute stage

2. Stalls
    a. A *"no-operation" (NOP) instruction inserted* into the pipeline causing a delay of instructions so the correct one can be executed
    b. Used when the required data is not yet available for an instruction (due to a data or control hazards arise)
    c. NOTE: Stalls can *decrease* pipeline performance as they introduce delays and reduce the pipeline's efficiency

- <u>Branch predictors (More Control Hazard Techniques)</u>

Overview: predict the outcome of branch instructions before the actual outcome is determined, allowing the pipeline to continue fetching and executing instructions speculatively based on the prediction

1. Local Branch Predictors
    a. Pattern-based prediction
    b. Focuses on predicting the outcome of a branch instruction based on its own historical behavior; it maintains a pattern history table (PHT) that tracks the recent outcomes of the branch instruction.

2. Correlation Branch Predictors
    a. Global branch prediction
    b. Takes into account the relationship between multiple branches to improve prediction accuracy by finding correlations and patterns among them; maintain a global history table (GHT) that stores the combined history of multiple branches

3. Tournament Branch Predictors
    a. Combines the strengths of the both local and correlation branch predictors
    b. Employs multiple predictors and uses a selector/predictor chooser to determine which prediction to use; monitors the accuracy of the two predictors and selects the prediction most accurate

Example Problem # 1:
A program is composed of 25% loads, 25% stores, 30% ALU inst, and 20% branches. Assume 30% of the loads are followed by an instruction that uses the load result. Assume branches are resolved in the ID stage and 40% of the branches are taken (next instruction is always fetched).

What is the average CPI for the program?
lw = .25 (.30 + (2 cycles for stall) + .70 (1))
sw = .25 (1)
alu = .30 (1)
branches = .20 (.40 + (2 cycles for squash) + 60 (1))

CPI = lw + sw + alu + branches
CPI = 1.155 CPU cycles

Example Problem #2:
A program is composed of the following instruction mix: 25% loads, 25% stores, 30% ALU inst and 20% branches. Assume 30% of the loads are followed by an instruction that uses the load result. Branches are resolved in the ID stage. 40% of the branches are taken, and the next instruction is always fetched.

1. Identify process percentages (seen in problem given)

2. Determine CPI Contributions to each process
   lw = Since 30% of the loads are followed by an instruction that uses the load result, there will be dependencies or data hazards. We'll assume a CPI contribution of 2 cycles for each load.

   sw = Stores typically have a CPI contribution of 1 cycle since they don't typically have dependencies.

   alu = Assume a CPI contribution of 1 cycle for ALU instructions

   branches = Since branches are resolved in the ID stage and 40% of the branches are taken, we'll assume a CPI contribution of 2 cycles for each branch

3. Calculate Weight
   lw = 25% * 2 cycles = 0.5 cycles
   sw = 25% * 1 cycle = 0.25 cycles
   alu = 30% * 1 cycle = 0.3 cycles
   branches = 20% * 2 cycles = 0.4 cycles

4. Calculate Overall CPU
   CPI = lw + sw + alu + branches
   CPI = 0.5 cycles + 0.25 cycles + 0.3 cycles + 0.4 cycles
   CPI = 1.45 CPU cycles

Caches

- What is caching?

  The use of a cache memory hierarchy to improve the performance of memory accesses (CPU vs Memory access speed). They are small, high-speed memory units that *store frequently accessed data or instructions*, providing faster access compared to accessing the main memory.

  Caching works based on the principle of *locality*, which is that programs tend to use the same items repeatedly. There are two primary types of locality:
  1. Temporal Locality (Time): if an item is just accessed, it is likely to be accessed again in the near future
  2. Spatial Locality (Space): if an item is just accessed, a neighboring item is likely to be accessed in the near future

- How to Find Data (including byte offset, block offset, index, tag)
  1. Determine the cache configuration: First, you need to know the cache configuration; including the total cache size, block size, and amount of sets
  2. Extract the Address Components:
     a. **Byte Offset**: Determines the position of the data within a cache block or line
        SOLVE:
        S = line size in bytes / 4 bytes per word
        Block Offset = log2(S) bits

     b. **Block Offset**: Identifies the specific block or line within a set
        SOLVE:
        S = line size in bytes / 4 bytes per word
        Block Offset = log2(S) bits

     c. **Index**: Determines which set in the cache the data belongs to
        SOLVE:
        S = total cache size in bytes / line size in bytes size

Index = log2(S) bits

d. **Tag**: Represents the remaining bits of the address after extracting the byte offset, block offset, and index. It uniquely identifies the memory block that is being stored in the cache.
SOLVE:
Tag = 32 address - index - block offset - byte offset

3. Calculate the components (as stated above)
4. Use the index to locate the cache set and compare the tag
    a. If they match, it's a HIT
    b. If the do not match, it's a MISS
5. IF A HIT, access the data on the cache line

- <u>How to Compute Hits and Misses</u>
    1. Follow the same steps above
    2. If it is a hit, increase the total hit counter. If it is a miss, increase the total miss counter.
    3. Repeat the process for each memory access
    4. Calculate
        a. Hit rate = (Number of cache hits) / (Total number of memory accesses)
        b. Miss rate = (Number of cache misses) / (Total number of memory accesses)

- <u>Cache Associativity</u>

    Refers to the organization and structure of cache memory. It determines how cache blocks are mapped to cache sets.

    1. Direct-Mapped Cache:
        a. Each memory block is mapped to exactly one specific cache set.
        b. This means that each cache set can hold only one block, resulting in a *one-to-one mapping* between memory blocks and cache sets.
    2. Set-Associative Cache:
        a. Each memory block can be mapped to a specific set of cache locations. The cache is divided into multiple sets, and each set can hold multiple blocks.
        b. This means that, although it is similar to the direct-mapped, it can *store multiple blocks within each set*.
    3. Fully-Associative Cache:

a. Each memory block can be placed in any location within the cache.
b. This means there is *no specific mapping* between memory blocks and cache sets. Each cache location can hold one block, and any block can be stored in any location within the cache.

NOTE: High flexibility, but requires more complex hardware for efficient searching and replacement algorithms.

- Cache Policies
  1. Write -Through:
     a. Every write operation updates both the cache and the corresponding location in the main memory simultaneously.
     b. This ensures that the data and the memory remain consistent. HOWEVER, it is *extremely slow* by creating memory traffic

  2. Write - Back:
     Writes are initially performed only in the cache, only updating the location in the main memory when the cache block is evicted or replaced.

  3. LRU (Least Recently Used): a replacement policy that replace the one accessed the longest ago (this algorithm is calculated through time stamps)

- Bit Use

  They provide additional information about the state and content of the cache block, helping manage cache state, handle cache misses, and maintain data consistency in the cache hierarchy

  1. Valid Bit:
     a. A flag indicating whether the cache block contains valid data.
     b. It is set to 1 (true) if the block is valid/holds a valid copy of data or set to 0 (false) if the block does not contain valid data.
     c. PRIMARILY used to handle *cache misses*. When a cache miss occurs and a new block is fetched from memory, the valid bit is set to 1 to indicate that the block now holds valid data.
  2. Dirty Bit:
     a. A flag indicating whether the block has been modified (written to) since it was last fetched from memory.

b. It is set to 1 (true) if the block is dirty/its contents have been changed/updated or set to 0 (false) if the block has not been modified since it was fetched.
c. PRIMARILY used for *write-back cache policies*. When a dirty block needs to be replaced in the cache due to capacity constraints or cache eviction, the dirty bit is checked.