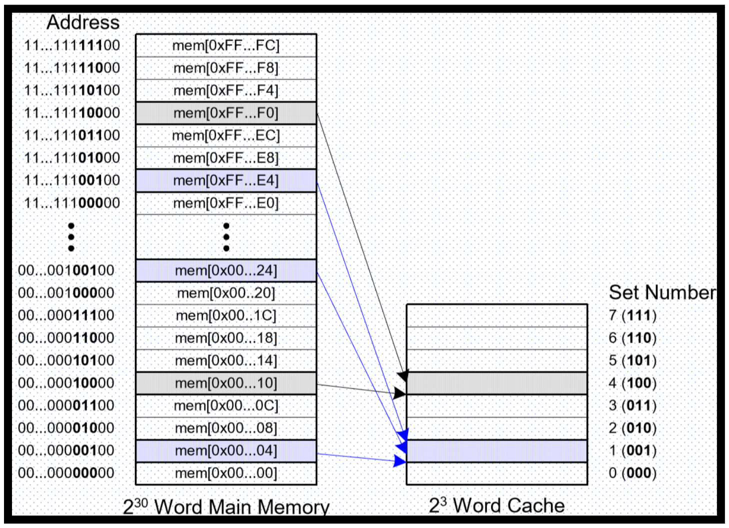
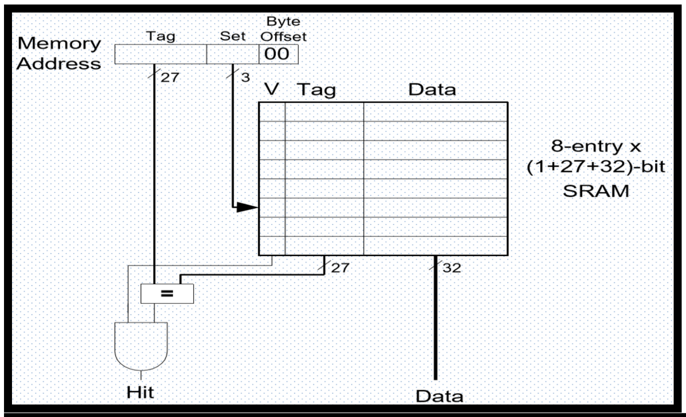
```
// ------------------------------------------------------------------------------------
// (1) Memory
// ------------------------------------------------------------------------------------

/* RECAPITULATION:
 * -------------------------------------------------------------------------------------
 * Computer Performance dependencies: { Processor_Performance, Memory_System_Performance }
 * -------------------------------------------------------------------------------------
 * Memories:
 * =========
 * SRAM -- fast but requires more space than DRAM (4-6 transistors)
 * DRAM -- sml but much slower than SRAM (factor of 5-10), val stored as charge on capacitor
 * -------------------------------------------------------------------------------------
 * Caches use SRAM for speed & technology compatibility; often hold most recently used data
 * -- Low density (6 transistor cells), high power, expensive, fast
 * -- State: content will last "forever" (until power is turned off)
 * -------------------------------------------------------------------------------------
 */

/* Memory_Hierarchy:
 * -------------------------------------------------------------------------------------
 *    ^            /\
 *    |           /  \          ____________________________________________________________
 *    |          /    \        | Tech:  | Price / GB  | Access Time (ns)   | Bandwidth (GB / s) |
 *    |         / Cache\       |=======|============|==================|=====================|
 *    |        /________\      | SRAM   |  $10,000    |     1              |        25+         |
 *    |       / Main Mem \     |________|_____________|____________________|____________________|
 *  S |      /____________\    | DRAM   |  $10        |     10 - 50        |        10          |
 *  p |     /              \   |________|_____________|____________________|____________________|
 *  e |    / Virtual Memory \  | SSD    |  $1         |     100,000        |        0.5         |
 *  e |   /__________________\ | HDD    |  $0.1       |     10,000,000     |        0.1         |
 *  d |                        |________|_____________|____________________|____________________|
 *     ------------------------>  *Note: DRAM (Main Mem) is PHYSICAL; Hard Drive is VIRTUAL Mem
 *       Capacity
 * -------------------------------------------------------------------------------------
 * Locality (Temporal/Spatial): Exploit locality to make memory accesses fast
 * ============================
 *
 * Temporal_Locality:
 * -- Locality in time (recently used data likely to be used again soon)
 * -- Exploit by keeping recently accessed data in the higher levels of memory hierarchy
 *
 * Spatial_Locality:
 * -- Locality in space (used data likely to be used alongside nearby data)
 * -- Exploit by bringing data nearby accessed data into higher levels of memory hierarchy
 * -------------------------------------------------------------------------------------
 * Amerage_Memory_Access_Time (AMAT) : avg time for processor to access data
 * Hit  : data found in that level of memory hierarchy
 * Miss : data not found, goto next level of memory
 *
 * Hit_Rate   : ( Hit_Cnt  / MemoryAccess_Cnt )
 * Miss_Rate  : ( Miss_Cnt / MemoryAccess_Cnt ) = 1 - Hit_Rate
 * AMAT       : t_cache + MR_cache ( t_MM )
 * -------------------------------------------------------------------------------------
 * AMDAHL'S_LAW: The effort spent increasing performance of a subsystem is wasted unless subsystem
 * affects a large percentage of the overall performance
 * -------------------------------------------------------------------------------------
 * EXAMPLE_1: Processor w/ 2-levels of hierarchy: cache & main memory
 * ==========
 * -- Program has 2,000 loads & stores, and has 1,250 of these data values in cache
 * -- t_cache  = 1   cycle
 * -- t_MM     = 100 cycles
 *
 * What is are the hit & miss rates for the cache, and what is the AMAT of the program ?
 *
 * Hit_Rate   = 1250 / 2000          = 0.625
 * Miss_Rate  = 750  / 2000          = 0.375
 * AMAT       = 1    + 0.375( 100 ) = 38.5 cycles
 * -------------------------------------------------------------------------------------
 */
```

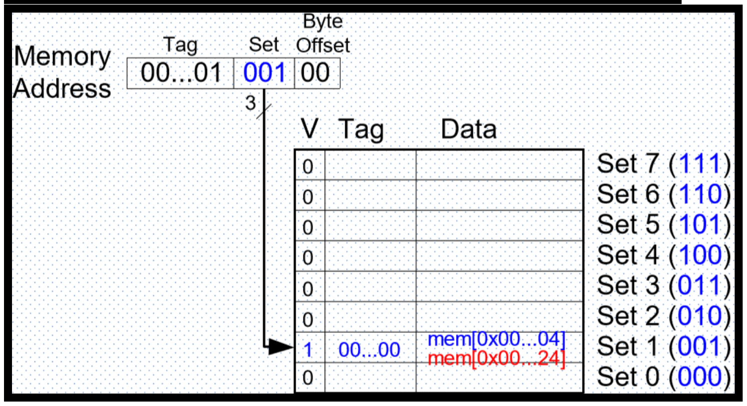
```
// ---------------------------------------------------------------------------------
// (2) Cache
// ---------------------------------------------------------------------------------

/* RECAPITULATION:
 * ---------------------------------------------------------------------------------
 * Cache ideally anticipates needed data & puts it in cache, but it is impossible to predict puture
 *
 * Past2Predict_Future: Temporal & Spatial Locality used to anticipate data usage to place in cache
 * ---------------------------------------------------------------------------------
 * DEFINITIONS:
 * ============
 * Temporal_Locality              : cp recently accessed data into cache
 * Spatial_Locality               : cp neighboring data into cache
 * Cache_Line                     : a memory block that is transferred to a memory cache
 * ---------------------------------------------------------------------------------
 * VARIABLES:
 * ==========
 * Capacity                 ( C )  : Number of data bytes in cache
 * Block_Size               ( b )  : Bytes of data brought into cache at once ( b = log_2 ( B ) )
 * Numberof_Blocks          ( B )  : Number of blocks in cache (B = C / b)
 * Degree_of_Associativity ( N )  : Number of blocks in a set
 * Numberof_Sets            ( S )  : Each mem addr maps to 1 cache set (S = B / N )
 * Set_Index                ( s )  : ( s = log_2 ( S ) )
 * ---------------------------------------------------------------------------------
 */

/* MAPPING:
 * ---------------------------------------------------------------------------------
 * Caches are organized into S sets, and caches are categorized by the number of blocks in a set N
 *
 * Direct_Mapped            : 1 block per set
 * N-Way_Set_Associative    : N blocks per set
 * Fully_Associative        : All cache blocks are within 1 set
 * ---------------------------------------------------------------------------------
 * Format of addr mapping for direct & set-associative mapping:
 *
 *  ___________________________________________
 * |_____Physical_Memory_Word_Address__________|
 * |__Block_Address________|___Block_Offset____|
 * |__Tag____|__Set_Index__|___Block_Offset____|
 * ---------------------------------------------------------------------------------
 * Main Disadvantage of Direct Mapping:
 * The fact that each block in the cache has a fixed location means that the hit ratio will be low
 * ( lrg amount of cache misses ). This is due to having multiple processes accessing blocks mapped
 * to the same line ( same index in phys addr but having different tag values ), which causes the
 * blocks to be swapped in and out of cache since they cannot both reside in cache simultaneously.
 * ---------------------------------------------------------------------------------
 */
```



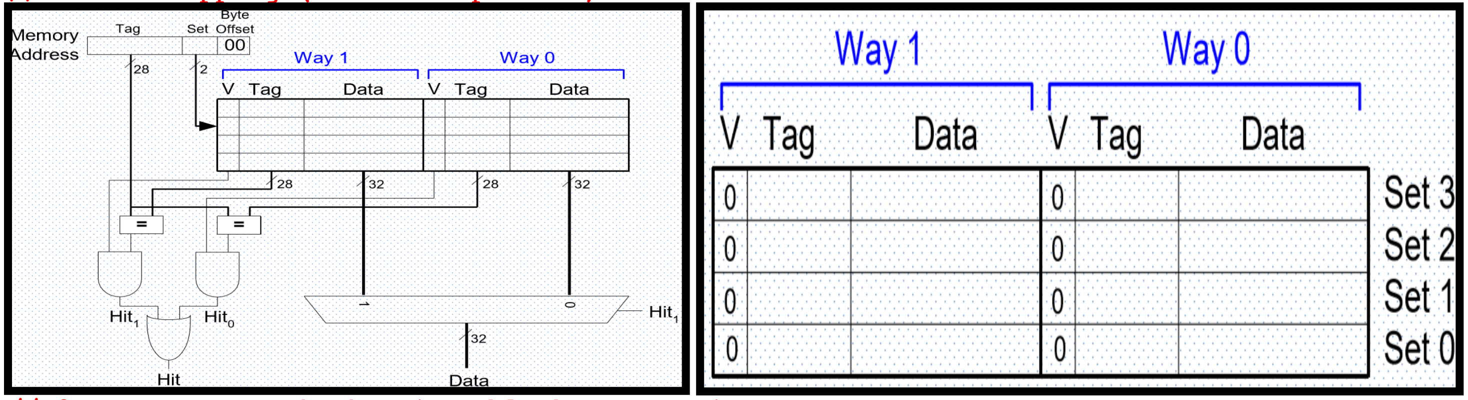
// Cache: 2^30 Word Main Mem Directly

```
/* EX: Direct Mapped Cache to 8 Sets
 * -------------------------------------------
 * C = 8 words (capacity)
 * b = 1 word (blk size)
 * -------------------------------------------
 * Thus, num of blocks, B = 8
 */

/* Other cache notes:
 * -------------------------------------------
 * Cache is too small to hold all data needed
 * -- If full, access to X will evict Y
 * -- Capacity_Miss when accessing Y again
 *
 * Q : How to choose Y to minimize chance of
 *     needing it again ?
 * A : Least Recently Used (LRU) Replacement
 * -- Evict least recently used block in a set
 * -------------------------------------------
```

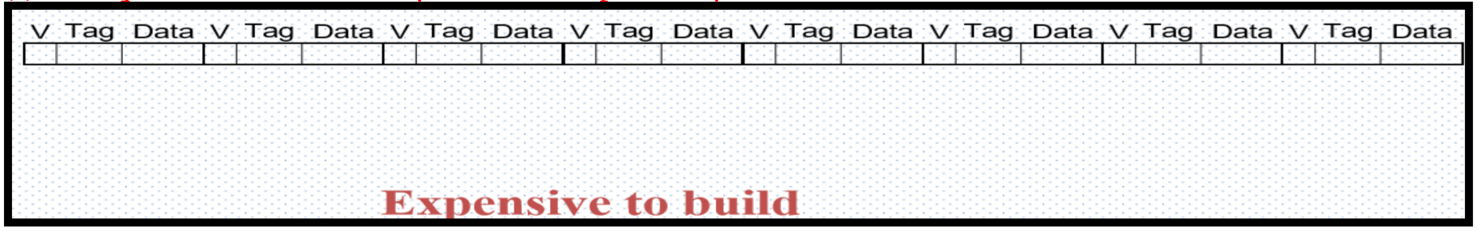




// Direct Mapping (one block per set)



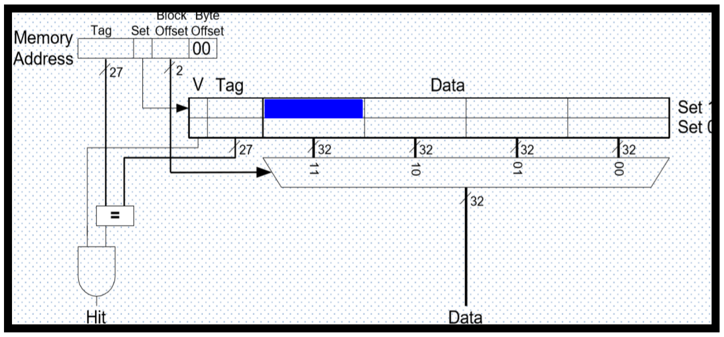
// 2-Way Set Associative (two blocks per set)

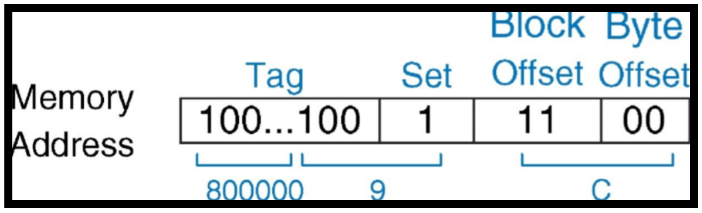


V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data
Expensive to build

// Fully Associative (all cache blocks are within one set)

```
// -----------------------------------------------------------------------------------------------
// EX: Increasing block size, b = 4 words --> Number of blocks, B = 2 (C/b = 8/4 =2)
// -----------------------------------------------------------------------------------------------
```

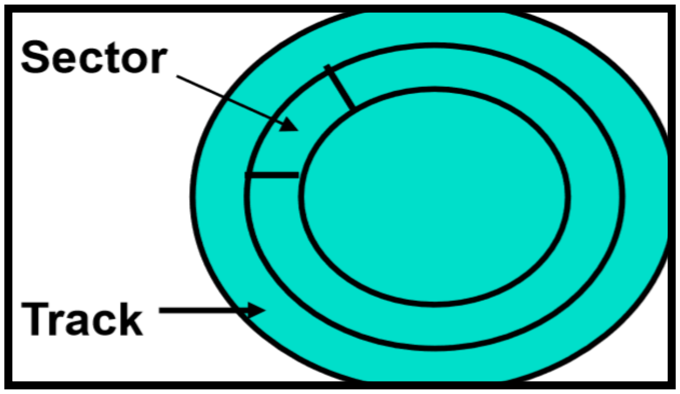


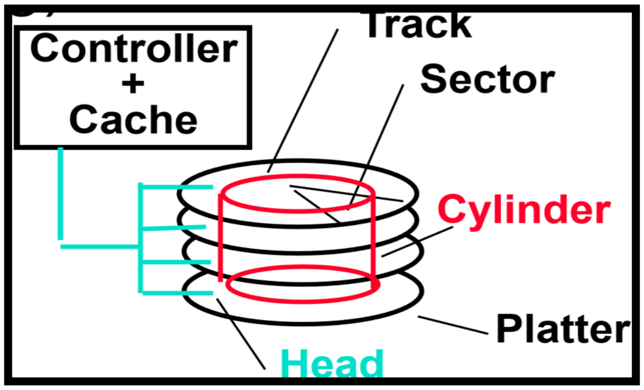


// Direct Mapping (all cache blocks (4 words each) are within one set)

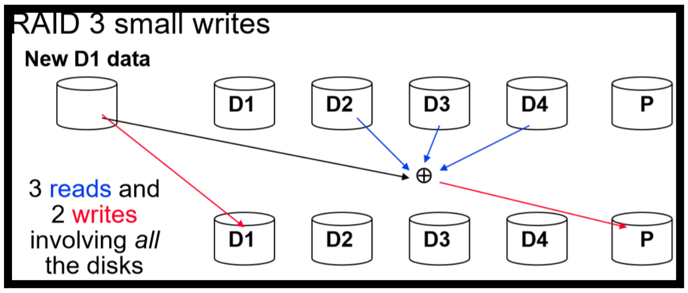
```
/* -----------------------------------------------------------------------------------------------
 * Types of misses (3):
 * (1) Compulsory    : First time data accessed
 * (2) Capacity      : Cache is too small to hold all data of interest
 * (3) Conflict      : Data of interest maps to same location in cache
 * Miss_penalty      : time it takes to retrieve a block from lower level of hierarchy
 * -- Bigger blocks reduce compulsory misses but increase conflict misses
 * -- Multilevel caches will have lower miss rates but longer access times
 * Typical levels    : Modern day PCs have L1, L2, L3 cache
 * -----------------------------------------------------------------------------------------------
 */
```

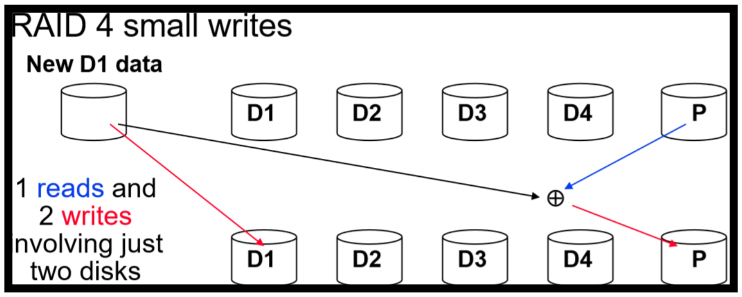
```
// ---------------------------------------------------------------------------------------
// (3) Disks_&_RAIDs
// ---------------------------------------------------------------------------------------

// MAJOR_COMPUTER_COMPONENTS : { (1) Processor (control, datapath), (2) Memory, (3) Devices (I/O) }
// MEMORY (current focus)     : { Secondary Memory (Disk), Main Memory, Cache }

/* MAGNETIC_DISKS:
 * ------------------------------------------------------------------------------------------------
 * PURPOSE                                           \ /     -- Sectors of a track
 * =======                                        ( ( O ) )  -- Track of a platter (magnetic disk)
 * -- Long term, nonvolatile storage
 * -- Lowest level in the memory hierarchy (slow, large, inexpensive)
 *
 * General_Structure:
 * -- Rotating platter coated w/ a magnetic surface
 * -- Moveable r/w head to access the information on the disk
 * -- Cylinder refers to all the tracks under the head at a given point on all surfaces
 *
 * Typical_Numbers:
 * -- 1 to 4 (1 or 2 surface) platters per disk of 1" - 3.5"
 * -- Rotational speeds of 5,400 to 15,000 RPM
 * -- 10,000 to 50,000 tracks per surface
 * -- 100 to 500 sectors per track (smallest unit that may be read / written ( typically 512B ) )
 * ------------------------------------------------------------------------------------------------
 * DISK R/W COMPONENTS
 * ===================
 * (1) Seek_Time           : Time to position the head over the proper track (3 to 14 ms avg)
 *
 * (2) Rotational_Latency  : Time to Wait for sector to rotate under head (1/2 of 1/RPM convert2ms)
 *                           -- 0.5 / 5400 RPM = 5.6 ms to 0.5 / 15000 RPM = 2.0 ms
 *                           -- Usually largest component of the access time
 *
 * (3) Transfer_Time       : Time to transfer block of bits (one or more sectors)
 *                           under the head to the disk controller's cache (30 to 80 MB/s avg rates)
 *                           -- Disk controller's "cache" uses spatial locality in disk accesses
 *                           -- Cache transfer rates are much fasater (e.g., 320 MB/s)
 *
 * (4) Controller_Time     : Time to perform disk I/O access overhead by disk controller (<.2ms avg)
 * ------------------------------------------------------------------------------------------------
 * DEPENDABILITY,_RELIABILITY,_AVAILABILITY
 * ========================================
 * MTTF : Reliability          -- Measured by the mean time to failure ( MTTF )
 * MTTR : Service Interruption -- measured by the mean time to repair  ( MTTR )
 *
 * Availability =  MTTF / ( MTTF + MTTR )
 *
 * Increase MTTF by either:
 * -- improving the quality of the components, or
 * -- designing the system to continue operating in the presence of faulty components
 *
 * Fault_Avoidance : Preventing fault occurrence by construction
 * Fault_Tolerance : Correcting ot bypassing faulty components (HW) via redundancy
 * ------------------------------------------------------------------------------------------------
 */
```





```
/* REDUNDANT_ARRAY_OF_INDEPENDENT_DISKS (RAIDs): Disk_Arrays
 * -----------------------------------------------------------------------------------------------
 * Arrays of small & inexpensive disks increase potential throughput by having many disk drives
 * -- Data spread over multiple disks, allows for multiple accesses made to several disks at a time
 * Reliability is lower than a single disk
 * Availability can be improved by adding redundant disks (RAID)
 *
 * MTTF : Mean time to failure of disks is tens of years
 * MTTR : Mean time to repair is in the order of hours
 * -----------------------------------------------------------------------------------------------
 * RAID_Level_0: Stripping; No redundancy                ___________________________________________
 *                                                      |__(_blk1_)__(_blk2_)__(_blk3_)__(_blk4_)__|
 * -- Mult sml disks, opposed to 1 lrg disk; no cost diff
 * -- Failure of one or more disks is more likely as the number of disks in sys increases
 *
 * Stiping:
 * -- Spreading the blocks over multiple disks, meaning multiple blocks may be accessed in parallel
 * -- Which greatly increases performance --> A 4-disk sys has 4x the throughput of a 1-disk sys
 * -----------------------------------------------------------------------------------------------
 * RAID_Level_1: Redundancy via Mirroring
 *
 *  ________________________________________________________________________________________
 * |________________Data_Disks:__________________|______________Redundant_(check)_Data__________|
 * |_(_blk1.1_)_(_blk1.2_)_(_blk1.3_)_(_blk1.4_)_|_(_blk1.1_)_(_blk1.2_)_(_blk1.3_)_(_blk1.4_)_|
 *
 * -- 2x num of disks as RAID_0 (e.g., 4-disk RAID_0 --> 8-disk RAID_1, w/ 2 sets of 4 data disks )
 * -- numRedundantDisks = numDataDisks (i.e., twice the cost of one big disk)
 * -- Writes must be made to both sets of disks
 * -- If one disk fails, the sys goes to the "mirror" for data
 * -----------------------------------------------------------------------------------------------
 * RAID_Level_3: Bit-Interleaved Parity
 *
 *  ________________________________________________________________
 * |________________Data_Disks:____________|__(odd)_Bit_Parity_Disk__|
 * |    (_1_)    (_0_)    (_1_)    (_0_)   |         (_1_)           |
 * |   blk1,b0  blk1,b1  blk1,b2  blk1,b3  |                         |
 * |____________________Disk_Fails_________|_________________________|
 *
 * -- N number of disks in a protection group
 * -- numRedundantDisks = 1x numProtectionGroups (i.e., twice the cost of one big disk)
 * -- Writes require writing the new data to the data disk as well as computing the parity
 * -- Computing the parity involves reading the other disks so that the parity disk may be updated
 *
 * -- Can tolerate limited disk failure, since the data can be reconstructed
 * -- Reads require reading all the operational data disks as well as the parity disk
 * -- in order to calculate the missing data that was stored on the failed disk
 * -----------------------------------------------------------------------------------------------
 * RAID_Level_4: Block-Interleaved Parity
 *
 *  ____________________________________________________________
 * |________________Data_Disks:____________|__Block_Parity_Disk__|
 * |__(_blk1_)_(_blk2_)_(_blk3_)_(_blk4_) _|________(____)_______|
 *
 * -- The parity is stored as blocks associated with sets of data blocks
 * -- 4x the throughput (stripping)
 * -- numRedundantDisks = 1x numProtectionGroups
 * -- Supports "small reads" & "small writes" (going to only a few data disk in a protection group)
 * -- Watching which bits change when writing, need only change corresponging bits on parity disk
 * -- Parity disk must be updated on every write, so it is a bottleneck for back-to-back writes
 * -- Can tolerate limited disk failure, since the data can be reconstructed
 * -----------------------------------------------------------------------------------------------
 */
```





```
/* OpSys_NOTES:
 * -----------------------------------------------------------------------------------------------
 * RAID-0 (striping): Spliting array of blocks across multiple disks (via RR)
 * pros: faster r/w speeds                    ||          cons: no data redundancy
 * -----------------------------------------------------------------------------------------------
 * RAID-1 (mirroring): Duplicating full chunks of data onto multiple disks
 * pros: redundancy of data (not striping)    ||          cons: slower (double the writing)
 * -----------------------------------------------------------------------------------------------
 * Single-Disk vs. Two-Disk RAID-0 :
 * =================================
 * 2Disk_RAID0: higher chance of data loss, no redundancy, very fast for r/w
 * Single_Disk: better data integrity, slower due to only single disk for r/w
 * -----------------------------------------------------------------------------------------------
 * Single-Disk vs. Two-Disk RAID-1 :
 * =================================
 * 2Disk_RAID1: fault tolerance (redundancy) but halves storage, slower r/w
 * Single_Disk: faster for r/w, equivalent storage space, zero data backup
 * -----------------------------------------------------------------------------------------------
 */
```

```
// ---------------------------------------------------------------------------
// (4) CPU_Performance
// ---------------------------------------------------------------------------

/* CPU:
 * -------------------------------------------------------------------------------------------------
 * Definitions:
 * ============
 * Response_Time (latency) -- Time between start & completion of a task (aka execution time)
 * Throughput              -- Total amount of work done in a given time period
 *
 * Clock: Used as a computer performance measurement
 * Ticks: Used to indicate start times of activities
 * -------------------------------------------------------------------------------------------------
 * Equations:
 * ==========
 * Clock_Rate ( frequency )        = cycles per second ( 1Hz = 1 cycle / sec )
 * Cycle_Time ( time btwn ticks )  = seconds per cycle (time between ticks )
 * CPU_ExecTime                    = CPU_clockcycles x ClockCycle_time
 *                                 = CPU_ClockCycles / Clock_Rate
 * -------------------------------------------------------------------------------------------------
 * Execution time normal reported in cycles, not seconds:   seconds       cycles       seconds
 *                                                          -------   =   -------  x   --------
 *                                                          program       program      cycle
 * -------------------------------------------------------------------------------------------------
 * HW_Performance_Improvement_Methods:
 * ===================================
 * (1) Reducing length of the clock cycle, or
 * (2) Reducing number of clock cycles required for a program
 * -------------------------------------------------------------------------------------------------
 */

/* CLOCK_CYCLES_PER_INSTRUCTION ( CPI ):
 * -------------------------------------------------------------------------------------------------
 * -- Average number of clock cycles that each instruction takes to execute
 * -- Provides a way to compare 2-different implmentations of the same instruction set architecture
 * -------------------------------------------------------------------------------------------------
 * Equations:
 * ==========
 * CPU_ClockCycles = Program_Instructions x Avg_ClockCycles_Per_Instruction
 * -------------------------------------------------------------------------------------------------
 */

/* RECAPITULATION:
 * -------------------------------------------------------------------------------------------------
 * Basic performance equation in terms of instruction count, CPI, & clock cycle time
 *
 * CPU_Time = Instruction_Cnt x CPI x ClockCycle_Time
 *          = Instruction_Cnt x CPI / Clock_Rate
 *
 * These formulas are key as they separate the three key factors that affect performance
 * -------------------------------------------------------------------------------------------------
 * Performance is specific to a particular program
 * Total execution time is a consistent summary of performance
 *
 * Increased performance of a given architecture is done by:
 * -- Increasing Clock_Rate (without adverse CPI affects)
 * -- Improving Processor_Organization that will lower CPI
 * -- Enhancing Compiler that will lower CPI &/or instruction count
 * -------------------------------------------------------------------------------------------------
 */
```

```
/* PERFORMANCE_&_EXECUTION_TIME_EX:
 * ----------------------------------------------------------------------------------------------------
 * Replacing processor w/ faster version
 * -- Increases response time && Increases throughput
 * ----------------------------------------------------------------------------------------------------
 * Add multiple processors to a system that already uses multiple processors for separate tasks
 * -- Increases throughput, but no one task will get work done fast (no increased response time)
 * ----------------------------------------------------------------------------------------------------
 * Performance & execution time relationship in computer X
 * =======================================================
 * Performance_X = 1 / Execution_Time_X
 *
 * For computers X & Y:
 * ===================
 * Performance_X / Performance_Y = n <==> "X is n times faster than Y"
 *
 * Performance_X  =  Execution_Time_Y  =  n
 * -------------  =  ----------------
 * Performance_Y  =  Execution_Time_X
 * ----------------------------------------------------------------------------------------------------
 * Problem_1: Which machine is faster?
 * ==========
 * machine_A runs a program in 10 secs
 * machine_B runs the dame program in 15 secs
 *
 * A is faster
 * ----------------------------------------------------------------------------------------------------
 * Problem_2: How much faster is machine_A than machine_B?
 * ==========
 * Execution_Time_B  =  15  =  n = 1.5
 * ----------------     ---
 * Execution_Time_A  =  10
 *
 * Machine_A is 1.5 times faster than machine_B
 * ----------------------------------------------------------------------------------------------------
 */

/* CLOCK_EX:
 * ----------------------------------------------------------------------------------------------------
 * 200 Mhz Clk has a cycle time of 1 / (200 * 10^6 ) = 5*10^(-9) = 5 nanoseconds
 * ----------------------------------------------------------------------------------------------------
 * NOTE: Different instructions take different amounts of time on different machines
 * NOTE: Mult requires more time than Add; Floating Point operations require more than integer ops
 * ----------------------------------------------------------------------------------------------------
 * CPI_EX_1: 2 implementations of the same instruction set architecture (ISA).
 *
 * Machine A has a clock cycle time of 10 ns and a CPI of 2.0 for a given program
 * Machine B has a clock cycle time of 20 ns and a CPI of 1.2 for that same program
 *
 * Which machine is faster? By how much?
 *
 * machine_A runs is 1.2x faster than machine_B
 * ----------------------------------------------------------------------------------------------------
 */
```