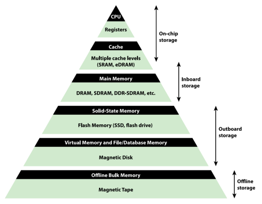
Lecture Notes
CS257 - Advanced Computer Architecture

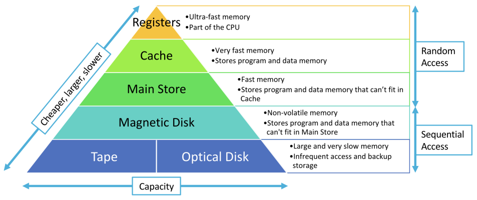
Intro | `TLP-RLP-ILP-DLP, SISD-SIMD-MISD-MIMD`

1. Levels of *parallelism* (LPs) include Data-level **DL** that operates many data items at the same time, and Task-level **TL** that operates many tasks in parallel. They include:
   • **TLP** thread-level is multithreading (*DL, TL*).
   • **RLP** request-level handles decoupled requests parallelly.
   • **ILP** instruction-level parallelism uses single thread for multiple instructions at once (*DL*).
   • **Vector Architectures, GPUs** apply single instruction to collection of data in parallel (DL).

2. **Flynn's Taxonomy**: Single instruction stream, single/multiple data stream (**SISD/SIMD**) provide instruct/data LP. Multiple instruct streams, single/multiple data streams (**MISD/MIMD**) provide thread/request LP.

3. **Performance measures**: response time, throughput. Execution time `exe_time`: wall-clock/CPU time. Speedup of X rel to Y: `exe_time`$(Y)\div$`exe_time`(X). Benchmark suites (SPEC06fp), not kernels or synthetic benchmarks.

4. 

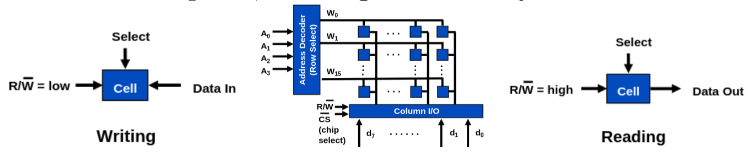

Cheaper, larger, slower
Registers
•Ultra-fast memory
•Part of the CPU
Cache
•Very fast memory
•Stores program and data memory
Main Store
•Fast memory
•Stores program and data memory that can't fit in Cache
Magnetic Disk
•Non-volatile memory
•Stores program and data memory that can't fit in Main Store
Tape
Optical Disk
•Large and very slow memory
•Infrequent access and backup storage
Random Access
Sequential Access
Capacity

5. **Spatial/temporal locality**: if memory loc referenced - likely that nearby/same loc will be referenced again soon.

RAM | `DRAM vs SRAM, DDR, read-only/mostly`

1. **Dynamic Random Access Memory (DRAM)** (main memory, 4-512MB) is volatile main memory composed of one capacitor and one transistor per bit, requiring periodic refresh cycles to retain data. Typical DRAM modules range in capacity from about 4 MB to 512 MB.

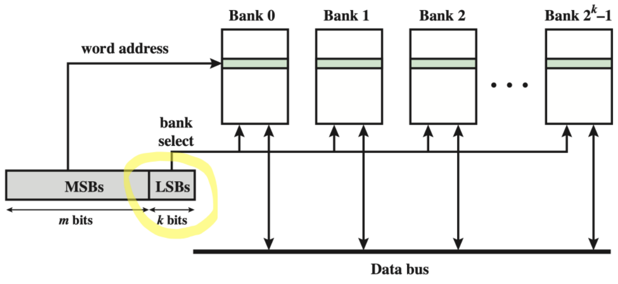
2. An M-bit $W \times B$ **DRAM block** is organised as an array of W words, each B bits wide, such that $W \times B = M$. Access requires B data pins and $\log_2 W$ address pins. To reduce decoder complexity, the address is split into row and column parts, treating the memory as a 2D matrix.



3. The `select` signal activates a specific memory cell, `r/w` determines the operation (read or write), and `data` serves as the input/output bus. This layout is most efficient when the matrix is roughly square, not excessively narrow/wide.

4. Multiplexed DRAM uses $\lceil \frac{log_2 W}{2} \rceil$ address pins.

5. **Interleaved Memory** is a grouped collection of DRAM chips comprising n independent banks that can service n requests simultaneously (memory multithreading kinda). Distr. addr among n memory units is **n-way interleaving**. Effective when num of mem banks equals or is integer multiple of num of words in cache line.

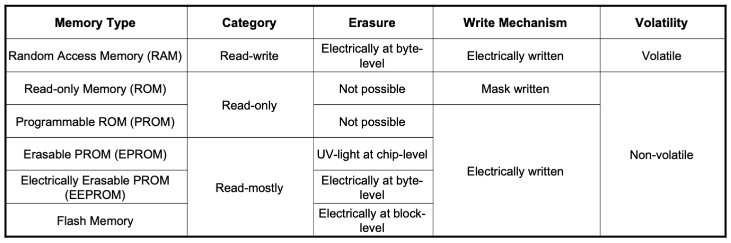


6. **SDRAM**, or **synchronous** DRAM sends data to processor at its bus clock speed to avoid wait states caused by DRAM **access time** (time before data becomes available).

7. **Double Data Rate (DDR)** SDRAM sends data to processor twice per bus clock cycle (both rising and falling edge) unlike normal SDRAM that only sends data once per rising edge. Uses higher clock rate on the bus to increase the transfer rate.

8. **Prefetch buffer** is a memory cache within DDR SDRAM, allowing to preposition multiple (e.g. 2, 4, 8,..) bits to be transferred to I/O buffer in separate pipelines at once: **2-n prefetch architecture**.

9. **Burst mode** in SDRAM eliminates the address setup time and row/column precharge time after the first access. Has **multiple-bank** internal architecture. **Mode register** specifies the burst length. Performs best when transferring large blocks of data serially.



Memory Type	Category	Erasure	Write Mechanism	Volatility
Random Access Memory (RAM)	Read-write	Electrically at byte-level	Electrically written	Volatile
Read-only Memory (ROM)	Read-only	Not possible	Mask written	Non-volatile
Programmable ROM (PROM)		Not possible	Electrically written	
Erasable PROM (EPROM)	Read-mostly	UV-light at chip-level		
Electrically Erasable PROM (EEPROM)		Electrically at byte-level		
Flash Memory		Electrically at block-level		

10. SRAM (L2 cache, 64KB-1MB) is a volatile bi-stable flip-flop, doesn't require refresh cycle. Much faster, more complex, less capacity, expensive compared to DDR.

11. **Embedded DRAM** (**eDRAM**) is intermediate between on-chip SRAM and off-chip DRAM. Usually integrated on Multi Chip Module (MCM) of an Application-Specific integrated circuit (ASIC) or microprocessor.

12. **Read-only Memory** (**ROM**) is non-volatile data storage, used in microprogramming, data is "wired onto" the chip, any error makes it unusable. To fix this, use electrically writeable **Programmable** ROM (**PROM**), but can only be written once. To fix, use read-mostly **Erasable** PROM (**EPROM**) blocks of which can be erased and rewritten using UV light. **Electrically Erasable** PROM (**EEPROM**) allows individual byte rewrite, but is very expensive and much slower write operations, less dense.

13. **Flash memory** supports high-speed block (not byte) reprogramming, 1 transistor/bit $\simeq$ EPROM $>$ EEPROM.

Virtual Memory Hierarchy, Performance

1. Can split main mem into physical and logical (**virtual**). If lookup item not found (**page fault**), "rename" its virtual address in **page table** instead of physically moving it, then reference it by the new name in physical memory.

2. **Multiple page lookup**: virtual mem is proportional to amount used, but > 1 lookup per request needed.

3. Design Principles for **mem hierarchy**: Temporal/spatial **locality**, **inclusion** (all info stored at level M_n most remote from processor, eventually copied onto M_{n-1}) and horizontal/vertical copied data **coherence**/consistency.

4. Memory hierarchy is characterised by the **cost per bit**: $C_i > C_{i+1}$, **access time** $t_{Ai} < t_{Ai+1}$, **capacity** $S_i < S_{i+1}$ at level i. Levels/layers: CPU $\leftrightarrow M_1 \leftrightarrow M_2 \leftrightarrow \cdots \leftrightarrow M_N$

5. Memory Hierarchy **Performance** depends on: Address reference patterns (order, frequency), per-level access time/cost C_i and capacity S_j, block transfer size and allocation policy (which blocks to replace).

6. Performance measure average cost per bit for combined memory $C_S = \frac{C_1 S_1 + C_2 S_2}{C_1 + C_2}$. Try to make cost C_S approach C_2 given $C_1 \gg C_2$, hence capacity $S_1 < S_2$. The bigger $S_2 - S_1$ the more can reduce relative combined costs.

7. Hit ratio $H = \frac{N_1}{N_1 + N_2}$, where N_1 is the # times the word is, N_2 - isn't in provided addr, is program dependent! Miss rate is $1 - H$.

8. Average Access time $t_{\text{avg}} = Ht_1 + (1-H)(t_1 + t_2)$. **Block transfer**: If item not at provided layer address, swap in as part of the mem block, access from the same level again. Uses slow I/O, so $t_2 \gg t_1$, $t_2 \simeq t_B$ (block transfer time).

9. **Access time ratio** between two layers $r = t_2/t_1$, **access efficiency** $e = \frac{t_1}{t_{avg}} = \frac{1}{1+(1-H)r}$ shows factor by which avg access time differs from minimum possible. For e to approach 1, H must also approach 1.

10. **Memory space** utilisation $u = \frac{S_u}{S}$ where S_u is utilised, S is all memory. Can detect wasted space with $S - S_u$, potentially signifying **empty regions**: mem blocks have different lengths, leaving "holes" (**fragmentation**), **inactive regions**: data transferred to a new block back and forth without being accessed. **System regions** occupied by mem management software.

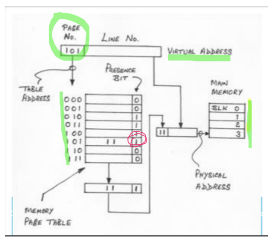
Virtual Memory | `paging vs segmentation`

1. **Virtual Memory** makes programs independent from capacity memory system, efficient memory sharing, no need for manual mem allocation. Uses mem hierarchy, simplifies loading of programs for execution (relocation).

2. To use memory efficiently in multiprogramming can use: **Swapping**: partitioning to make better use of space, and using logical vs physical address; and **Paging** vs Frames.

3. **Address Translation**: CPU virtual address space is much greater than memory physical memory space, so when info isn't in main mem, need to swap it from secondary mem, a process can only execute in main memory.

4. **Temporal Locality**: items (data/instructions) once referenced are likely to be referenced again in near future (iterative loops, access to vars).
Spatial Locality: program often references items whose addresses are close to each other in address space (array elements).
Sequential Locality: most instructions in program are executed sequentially, with branching-out account for $<$ 30% of instructs.

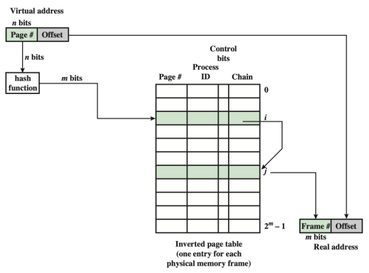


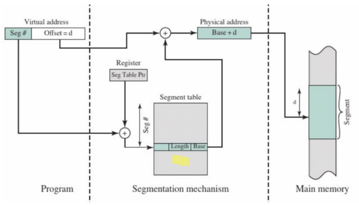
5. **Page Table**: data structure containing the translation between logical and physical addresses. Presence Bit $= 0$ signifies page fault. When virtual address space $\gg$ mem space, most entries will be empty, too many entries.
Every virtual mem reference causes **2** physical **memory accesses**: fetch page table entry, fetch data.

6. **Demand Paging**: each page of a process is brought in only when it's needed. When program references page not in memory, it triggers **page fault** which tells the OS to bring in that page.

7. **Hierarchical page Table** uses nested page tables, so page memory usage is proportional to amount of memory used by process, however need > 1 page table lookup.

8. **Inverted Page Table**: same as separate chaining hashmap: hash the page number, **probe** by indexing into page table - if encounter terminator symbol then the address isn't there, page fault, else concatenate the page frame addr with the offset(word number) to give real memory address. Weighted avg # of probes is 1.5, so desirable to have $2n$ entries for n pages in main memory.





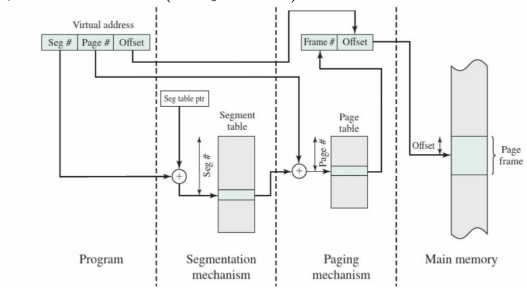
9. **Segmentation** divides memory into **segments**, or variable-length regions that represent logical groupings of instructions or data (e.g., code, stack, heap). Segments are named and can be defined by the programmer or operating system. Each segment has its own base, limit, and associated access rights.

10. Advantages offered by segmentation, not paging: simplifies growing data structures, as can be assigned to its own segment that can expand or shrink. Allows programs to be altered and recompiled independently without relinking or reloading. Lends itself to being shared among processes and to protection through assignment of access privileges.

	Page	Segment
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Difficult (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

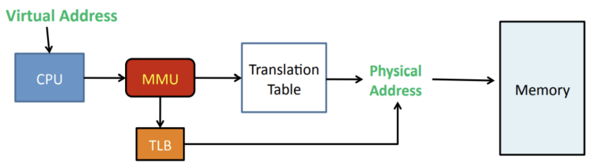
11. **Segmented Page Mapping** divides memory into variable-sized **segments**, each split into fixed-size **pages**. Reduces memory overhead of flat paging but may cause **external fragmentation**, where small unusable memory gaps remain. Mapping requires segment and page tables.

Each virtual memory reference causes **3** physical **memory accesses**: fetch segment table entry, fetch page table entry, fetch data (very slow).



12. **Memory Management Unit (MMU)** is hardware translating a virtual address into physical address. Each memory reference is passed through it.

13. **Translation Lookaside Buffer (TLB)**: cache for MMU with the most used page table entries. Offers high performance gain, usually 32-128 entries, 4 to 8-way set-associative. Switching processes is expensive because TLB has to bee flushed, but can include process id to avoid it.



14. **TLB performance**: when page address not found in TLB (TLB miss), significant overhead occurs in searching main memory page tables, even when it's in main memory.

15. **Avg. address translation time** $t_t = t_{\text{tlb}} + (1 - H_{\text{tlb}})t_{\text{mt}}$ where t_{tlb} is TLB translation time (whether hit or miss), t_{mt} is table lookup time on TLB miss, H_{tlb} is TLB hit ratio. TLB miss ratio $(1 - H_{\text{tlb}})$ is usually < 0.01.

16. **Page size** S_p impacts memory space-utilisation factor u. Large $S_p \to$ internal fragmentation, Small $S_p \to$ page tables become large, which reduces space utilisation.

17. Let S_s be average segment size in words. If $S_s > S_p$ then on average, the last page assigned to a segment will contain $S_p \div 2$ words, while the size of page table associated with each segment is $S_s \div S_p$ words (assuming each entry is single word).

    Segment memory space overhead $S = \dfrac{S_p}{2} + \dfrac{S_s}{S_p}$

    Space Utilisation $u = \dfrac{S_s}{S_s + S} = \dfrac{2S_sS_p}{S_p^2 + 2S_s(1 + S_p)}$

18. **Optimal page size** S_p^{OPT} is defined as value of S_p that maximises u or minimises S. Can differentiate:
    $\dfrac{dS}{dS_p} = \dfrac{1}{2} - \dfrac{S_s}{S_p^2} \Rightarrow S_p^{\text{OPT}} = \sqrt{2S_s}$ and $u^{\text{OPT}} = \dfrac{1}{1 + \sqrt{2 \div S_s}}$
    since S is minimum when $dS \div dS_p = 0$.

19. Given virtual address referenced A_i and next address generated A_{i+d} with distance d between the two assume efficient page replacement, then probability of A_{i+d} being in memory level M_1 is high if either:
1. $d \ll S_p$, so A_i, A_{i+d} are in the same page P, probability of these addr both being in P increases with page size.
2. $d \gg S_p$ but A_{i+d} associated with set of frequently referenced words, therefore likely to be in page $P' \neq P$, which is also in M_1. This likelihood increases with num of pages stored in M_1, so decreases with size of S_p.

20. Hit ratio H increases with small S_p, but decreases past a certain value. Prefer S_p maximising H and decreasing t_A. In large systems, values S_p yielding maximum H can be greater than S_p^{OPT}.

21. **Page Replacement Algorithms**: upon page fault mem management software swaps in the page from secondary storage, if main memory full, swap out a page beforehand.
• **Random**: pseudorandom generator provides number of page to be replaced, no consideration of principle of locality, so poor performance.
• **FIFO**: replace page that's been in main memory for longest. No additional hardware required, but no consideration of page usage, so can remove and reload too often.
• **Clock**: avoids frequent transfers. FI-Not-Used-FO has "use" bit set to 1 when page is referenced. At page fault, if the earliest page "use" is 1, reset to 0 (give it a second chance to stay in the main memory) and advance the pointer. Repeat until encounter "use"$= 0$, and replace.
• **LRU**: swap out the page not referenced for longest, or one with largest "age counter" that's cleared when page is accessed, incremented at fixed intervals. Difficult for large number of pages, tend to use approximation.
• **LRU approximation**: "use" bits instead of counters, set to 1 on each memory access. At fast intervals, OS examines all these reference bits and resets to 0 when read. Evict a page with "use" $= 0$. Record of num of times "use" bits were set to 1 gives approx usage close to LRU.
• **Working set** $w(t, T)$ at time t is collection of pages referenced during time interval $(t\text{-}T, t)$, ensuring each process keeps its working set in memory. Evict a page not used within the last T time units (process time, not absolute time), but hard to find one outside of working set. Small $T \to$ evict too soon, more page faults, large $T \to$ too many pages stay in memory, reduced efficiency.

22. **Thrashing**: too many processes in too little memory, OS keeps swapping (excessive page faults), so constant disk usage, preventing meaningful progress. Solve with good page replacement algorithms, reducing number of running processes or installing more memory.

Cache Memory

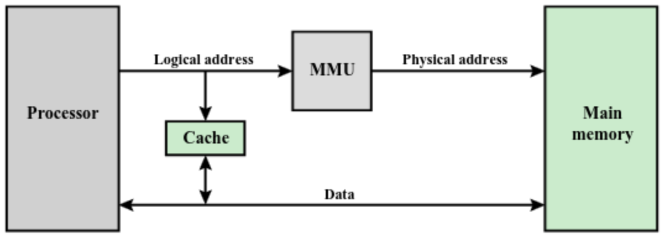
1. **Locality of Reference**: CPU can only process memory within its registers with fast access time, but low capacity. Main memory, or RAM on the other hand is slower, but has much more capacity, so CPU "stages" copy of that data in **cache** that's somewhere in-between the two.

2. **Block**: min transfer unit between cache & main memory. **Line**: cache memory portion capable of holding one block and a tag. **Tag** is portion of cache line used for addressing.

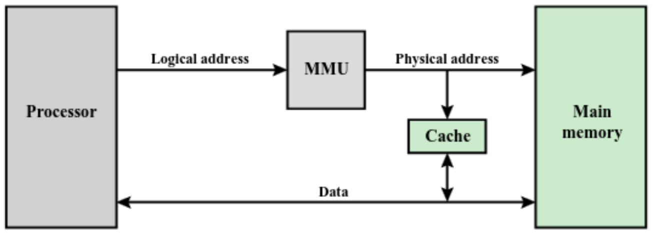
3. Main memory divided in fixed-length blocks of K words each. Cache comprises m lines of K words and a tag. Each line also includes **control bits** that indicate if line contains valid data (*valid bit*) or has been modified since being loaded (*dirty bit*) and other status information.

4. **Cache read**: receive **read address (RA)** from CPU, if cache line tag matches (**Cache Hit**), return copy of data from cache, else (**Cache Miss**) read block of data from main memory, replace victim block in cache with it, return copy of data.

5. **Virtual/logical Cache** stores data using **virtual addresses** before going through MMU, so CPU can access it faster than physical address; translation only happens on cache misses.



6. **Physical cache** uses main memory physical address, but is placed after the MMU, so have to translate at each call - slower, but avoids aliasing issues (same physical address might have multiple virtual addresses):



7. **Hit Ratio** $h = \frac{\text{\# times words are in cache}}{\text{total \# memory references}}$. **Miss Ratio** $1 - h$

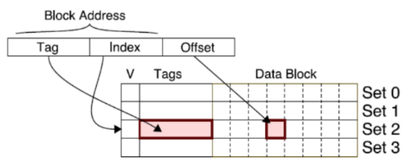
8. For **cache access time** t_c and main **memory access time** t_m, the **average access time** t_a is:
$$t_a = t_c + (1-h)t_m \implies \frac{t_a}{t_c} = 1 + (1-h)\frac{t_m}{t_c}$$
Want $t_a \div t_c$ to approach 1. In cache, $t_m \div t_c$ is low, so h of around $0.7 - 0.9$ is fine.

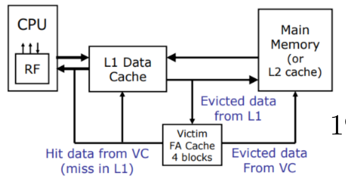
9. **Block Placement**: number of cache lines m is smaller than # of main memory blocks, so need to map each block j onto a line. Can do so using following mappings:
• **Direct Mapping**: each block mapped into one possible cache line. Mapping: $i = j \mod m$ for line number i.
• **Set Associative**: a block can be placed in a restricted number of places comprising **sets**, number of which $i = j \mod v$ and $m = v \times k$ where k is num of lines in each set.
• **Fully Associative**: permit each block to be loaded into any line. **Cache control Logic (CCL)** interprets mem address as Tag-Word field. To see if block in cache, CCL simultaneously examines every line's tag for match.

Cache set number i, main memory block number j, number of lines in the cache m, number of sets v and number of lines in each set k.

10. **Cache Memory Address**: divided into a *block address* (tag + index or set bits) and a *block offset* (word). The offset selects the word within the block; the index selects the cache line (direct mapping) or the set (set-associative); and the tag is compared against stored tags to detect a hit. (Offsets are not used for comparison, since they always match by definition.)

11. **Direct-Mapped Block Identification**: use the index to select one cache line; compare the stored tag with the address tag; then check the valid bit. Simple, low-cost, but has conflict misses as each block has fixed location.



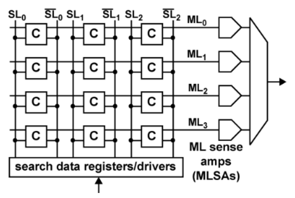


12. **Victim Cache**: a small (4–16 lines), fully-associative buffer placed between the direct-mapped L1 cache and the next level. It holds evicted L1 lines to reduce conflict misses, so the store mutually exclusive data.

13. **Set-Associative Block Identification**: compute set index $i = j \mod v$; compare the tag against all k lines in that set and check each valid bit; on a miss, invoke a replacement policy (e.g. LRU) to choose which line to evict. Balances hardware complexity and conflict-miss reduction.

14. **Fully-Associative Block Identification**: treat the entire cache as a single set (no index field); compare the tag against all m lines in parallel and check valid bits; on a miss, use a replacement policy to evict any line. Not determined by size of line field in CPU address, so can be any size. High flexibility but requires complex hardware.

15. **Content addressable memory (CAM)** is made out of SRAM cells, but more expensive and holds much less data. CAM searches for the provided tag, returning the address where the match is found (and for some architectures the associated data word) in a single clock cycle (extremely fast). Commonly used in TLB and victim caches.



16. **Way-prediction cache** guesses which "way" in a set-associative cache holds the data. If the guess is right, you get a hit in a single cycle with lower power; if it's wrong, fall back to the full associative check at a small extra cost. Higher set associativity usually has higher hit ratio.

17. **Cache replacement Policies** are needed for (set) associative mapping block replacement. Must be implemented in hardware. Most effective usage-based algorithm is LRU, others include: random, FIFO. Direct mapping only has one line per block, so no choice is possible.

18. **LRU**: the "age" counter associated with each line incremented at regular intervals and reset when line is referenced. Can modify to consider that counters have fixed number of bits and only the relative age is required. On hit, reset hit line to 0 and increment all counters with a smaller count. On miss when the cache is full, replace the line with max. count, reset it to 0 and increment all other counters. Largest count is the least recently used line. Need $\lceil \log_2 n \rceil$-bit counter for n-way associativity.

19. **Cache Write Policies** must maintain consistency with main memory. **Write-through** updates both the cache and lower-level memory on every write, ensuring coherency but generating high memory traffic. **Write-back** writes only to the cache and sets a *dirty bit*; on eviction, dirty blocks are written back to main memory, reducing overall writes but leaving memory stale in between. This requires I/O operations to be routed through the cache controller, adding hardware complexity.

20. **Write miss** at cache level: **write allocate**, the block containing the word to be written is fetched from main memory (or next level cache) into the cache and processor proceeds with the write cycle (*write through*). **No write allocate**: that block is modified in main memory and not loaded into cache (*write back*). But can be used in both.

21. **Cache coherency** in multilevel cache ensures the data is consistent between different levels so that CPU sees the latest updates.

22. Ensure cache consistency with the following.
• **Bus watching with write through**: when data is

changed, each cache "listens in" on the system bus, and either update their data or invalidate it: simple, but needs everyone on the same bus and using write through.
• **Hardware transparency**: additional hardware ensures all updates to main memory via cache are reflected in all caches (**inclusive policy**).
• **Nonchacheable memory**: shared main memory portion that's never cached; all accesses result in cache misses.

23. **Split cache** (L1) can improve performance: **Instruction Cache** is reach only (by CPU), so no need to write back to main memory when block overwritten. **Data cache**: less predictable access, larger size, read/write happens often. Usually used for L1 cache - others use **unified cache**.

Cache Optimisation

1. **Avg mem access time** = hit_time+(miss_rate×miss_penalty)
• Reduce **miss rate**: larger block and cache sizes, higher associativity. • Reduce **miss penalty**: multilevel caches and give reads priority over writes. • Reduce **hit time**: avoid address translation when indexing the cache.

2. **Cache misses:** compulsory, capacity, conflict and coherency (in multiprocessors). Hit/miss rates don't reflect true performance as they don't factor cost of cache miss.

3. **Larger block sizes** (`-MissRate,+MissPenalty`): spatial locality might reduce miss rate, but increases miss penalty and the number of capacity and conflict misses.

4. **Larger caches** (`-MissRate,+HitTime`) can cause longer hit times and incrased power consumption.

5. **Higher associativity** (`-MissRate,+HitTime`) reduces the number of conflict misses, but can cause longer hit times and increased power consumption.

6. **Multilevel caches** (`-MissPenalty`) reduces miss penalty, L1 keeps pace with clock cycle times, while L2 and higher serve to reduce number of main memory accesses. Now the **average multilevel access time** with L1, L2 is:
$\text{HitTime}_{L1}+\text{MissRate}_{L1}\times(\text{HitTime}_{L2}+\text{MissRate}_{L2}\times\text{MissPenalty}_{L2})$

7. **Prioritising read misses over writes** (`-MissPenalty`): a small write buffer holds pending stores so a read miss can bypass earlier writes, avoiding read-after-write hazards and, when no address conflict and the bus is free, reducing miss penalty with little hardware cost.

8. **Avoiding address translation during cache indexing** (`-HitTime`) uses page offset p (like virtual memory) to index cache. **Virtually indexed, Physically Tagged (VIPT)** caches stores at most p bits. Increase associativity to make cache larger with same page size, size of index follows: $2^{\text{index}} = \frac{\text{cache size}}{\text{block size}\times\text{set assoc.}}$.

Code Optimisation and Refactoring

1. Best/worst case scenario: 100% system utilisation/one subsystem has bottleneck. Performance can be memory or compute bound.

2. **Peak FLOPs**: max achievable FLOPs for given machine, **Program efficiency**: $\text{FLOPs}_{\text{achieved}} \div \text{FLOPs}_{\text{peak}} \times 100$. $\text{FLOPs}_{\text{peak}} = \text{clockSpeed} \cdot \text{cores} \cdot \text{flop/cycle}$.

3. **Cache line** is sequential data chunk loaded into cache. Cache miss can be **compulsory**: data never loaded into cache before, **capacity**: data was evicted due to capacity, **conflict**: expected location already occupied.

4. **Optimisation speedup**: unoptimised time/optim time. Have **algorithmic** optim, **code refactoring** (make compiler apply efficient single-core optimisations) and **parallelisation**. Compiler optim is controlled by flags.

5. **Pipelining** is technique exploiting parallelism to execute concurrently and increase throughput, but need to write code that supports async execution as dependencies may slow the system down by preventing concurrency. **Fetch and execute**: instruct → fetch → instruct → execute.

6. **Read after Write (RAW)** hazard:

```
int a=5; int b=a*2 // pipeline may try to fetch 'a'
    before assigning a value to it
```

7. **Loop dependency**: later loop iterations depend on earlier ones' result, so can't parallelise the execution.

```
for (i=1;i<N;i++) arr[i]=arr[i-1]+1; // dependent
```

Dependency detection: for each variable x in a loop: if x is read and never written: no dependency. If there be any access to x in other iterations than the current - there's a dependency.

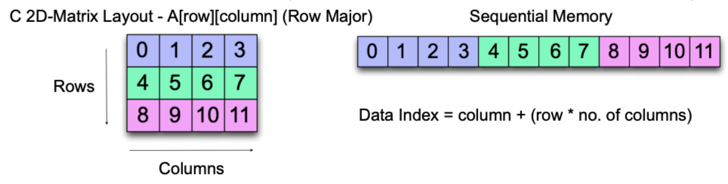
8. **Loop peeling**: some dependency-inducing task extracted from loop, e.g. `arr[0]=0`, now can parallelise.

```
arr[0] = special_case(); // Handle separately
for (i=1;i<N;i++) arr[i]=arr[i-1]+1; // now indep.
```

9. **Pointer Aliasing**: multiple pointers point to the same or overlapping memory location. Use `restrict` if not aliased to ensure unique memory location pointer.

```
void foo(int *a, int *b) {*a += *b;}
// If a and b alias-can't parallelise, use restrict
void foo(int *restrict a, int *restrict b) {*a+=*b;}
```

10. **Loop interchange**: modify order of memory access by switching order of loops (traversing table cols before rows).


C 2D-Matrix Layout - A[row][column] (Row Major)
Rows
Columns
0 1 2 3
4 5 6 7
8 9 10 11
Sequential Memory
0 1 2 3 4 5 6 7 8 9 10 11
Data Index = column + (row * no. of columns)

Non-sequential access uses stride of row length, if it fits into cache, little performance impact, if not - significant.

```
for (row = 0; row < N; row++)     // can switch
    for (col = 0; col < M; col++) // these two
        process_block(row, col);
```

11. **Loop Blocking**: restructure loops to process data in small blocks of size B instead of sequentially, improving cache locality and reducing memory access latency.

```
for (i = 0; i < N; i += B) // note the block size B
    for (j = 0; j < M; j += B) // like a moving kernel
        process_block(i, j); // insted of a cell
```

12. **Loop fusion**: merge multiple same-range loops into one s.t. no dependencies are broken. Useful when high overhead of loop conditional checks, poor temporal locality/expensive intermediate inter-loop result storage.

```
for (i = 0; i < N; i++) { processA(i); processB(i); }
```

13. **Loop fission**: improve locality by splitting loops comprising many unrelated operations (opposite of fusion). Useful if poor temporal locality of memory accesses between loop iterations or registers spilling into cache.

```
for (i = 0; i < N; i++) processA(i); // handle
for (i = 0; i < N; i++) processB(i); // separately
```

14. **Loop unrolling**: expand loop body to do multiple iterations. Mustn't break inter-loop dependencies. Useful when high overhead of loop conditional checks or multiple arrays storing intermediate values between loop iterations.

```
for (i = 0; i < N; i += 4) // unroll factor = 4
    process(arr[i]); process(arr[i+1]);
    process(arr[i+2]); process(arr[i+3]);
```

In C can use `#pragma unroll(`n`)`

15. **Loop Pipelining**: reorder independent operations across iters within a loop to enable instruct pipelining. **Prolog** and **epilog** represent portions of pipeline leading up to the full overlap. The middle can be pipelined.

```
for (i = 0; i < N; i++)
    a=load(i); b=compute(a); // load then compute
    store(i, b); // store last to optimise
```

Vectorisation

1. **SISD**: classic serial application, applies one instruction to one data stream at a time. **SIMD**: one instruction can be applied to multipe data streams at the same time. **MISD** is rarely encountered, and **MIMD**: different instructions can be applied across different data sets in parallel.

2. **Vector instructions** utilise SIMD to apply a single instruction across all data elements simultaneously. Vector registers contain multiple data elements at a time.
$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} + \begin{bmatrix}4 & 5 & 6\end{bmatrix} = \begin{bmatrix}5 & 7 & 9\end{bmatrix}$$
Useful in image overlaying (chroma keying)

3. **Auto-vectorisation**: compiler implemented, less control but easy: `#pragma ivdep / simd / vector always`

4. **Vectorisation Intrinsics**: map to underlying assembly instructions, more control but difficult, can cover cases that compiler can't.

5. **In-line assembly** instructions written directly in code, offer most control but difficult to implement, and less portable (depends on system).

6. # parallelised operations = register width ÷ datatype size. **Streaming SIMD Extensions (SSE)** has register width of 16 bytes, so can hold 4 data elements of size 4 bytes, or 2 of size 8 bytes. **Advanced Vector Extensions (AVX)** are twice the register width of SSE. **Altivec** is old and **GPUs** are basically large SIMD units.

7. **Memory Alignment**: memory block aligned to a value m if byte offset is divisible by m - typically a power of 2 (**alignment boundary**), e.g. 16/32/64 bytes, depending on processor. Aligned 12|45|67; Unaligned 1|234|56|7

8. To use aligned loads and stores, memory alignment must be at least the size of vector register, for optimal performance - at least the size of cache line.

9. <`immintrin.h`> header provides access to SSE intrinsics. `__m128` is 128bit single precision (float) unit vector, `__m128d` is double. Can allocate memory with requested alignment with `_mm_malloc(vector, alignment`, and free it with `_mm_free`. Add/multiply with `_mm_add_ps` or `_mm_mul_ps`.

10. SSE can't selectively run different operations in parallel, so avoid branching - precompute results and select correct value from each vector of results using a mask, but slow (two branches means twice as much computation), so best to eliminate them.

11. **SSE Shuffle**: rearrange elements within SSE vector register or duplicate its parts: `_mm_shuffle_ps(vec_a,vec_b,i)`, for control value i, or macro `_MM_SHUFFLE(2,1,0)` where leftmost entry is the highest index of loaded array.

Threading

1. Compiler can't auto-thread code, need to implement it.

2. **Thread** is a single flow of control within a process. Threaded program can run code blocks in parallel asynchronously (MIMD). Usually have one thread per core, each with private memory accessible only by itself and global memory shared by all threads.

3. **Race condition** is when final result is dependent on order of execution. Parallel algorithm must end with deterministic result despite nondeterministic thread ordering.

4. **OpenMP**, or Open Specifications for MultiProcessing, is an API used to explicitly direct multithreaded shared memory parallelism. Explicit parallelism: programmer has full control over parallelisation.

5. OpenMP doesn't specify parallel I/O, it's up to the programmer. Threads can "cache" their data and not required to be consistent with real memory all the time.

```
#include <omp.h>
// #pragma omp parallel [clause list]
#pragma omp parallel if (is_parallel == 1)
    num_threads(8) shared (var_b) private (var_a)
    firstprivate (var_c) default (none) {
  // parallel section executed by all threads
  // all threads join master thread and disband
}
```

   Clause list may include: **conditional parallelisation** `"if (bool expr)"`, **degree of concurrency** `num_threads(n)` and **data scoping**: `private`, `firstprivate`, `shared` taking variable list as parameters, and `default` taking one of the above (`shared` or `none`).

6. **Number of threads** is determined in following order:
   1. Evaluation of the "if" clause.
   2. Setting of the `num_threads()` clause.
   3. Use of `omp_set_num_threads()` function.
   4. Setting of `OMP_NUM_THREAD` environment table.
   5. Implementation default: typically # of cores on a node.

   Threads are numbered from 0 (master thread) to $N-1$.

7. `omp_(set/get)_num_threads(n)` sets/gets the maximum # threads. `omp_get_thread_num()` gets current thread id.

8. **Flow control**: `parallel`, `barrier`, `critical`, `atomic` deals with parallel execution, blocking and synchronisation of threads, thread creation and preventing violation of dependencies and race conditions.

9. **False Sharing:** Occurs when threads update nearby variables in the same cache line, causing slowdowns. Avoid by aligning data or using thread-local variables.

10. • `#pragma omp` **`parallel`** {...} signifies start of parallel threaded region, bounded by "{}".
    • `#pragma omp` **`barrier`** prevents any one thread from continuing until all threads have reached the barrier and synchronised. Helps prevent violation of dependencies, but is expensive, doesn't scale well.
    • `#pragma omp` **`critical`** {...} serialises a portion of a parallel region: only one thread executing at a time, others are blocked. Prevents race conditions, but is slow.
    • `#pragma omp` **`atomic`** allows immediately following update to execute atomically, preventing race conditions from multiple writing threads, faster than `critical`. Only works with $+, *, -, /, \&, \hat{}, |, <, >$, not nested functions.

11. **Variable scope** `private`, `shared`, `default`, `reduction`, `firstprivate`: each thread has their own private scope and access to shared memory. Variables declared in parallel region are private to a thread, those outside must have a scope type declared if used inside the region.

12. • `#pragma omp` **`private(var1,..)`**: separate variable instnace per thread, doesn't retain original value, useful when original value doesn't matter, e.g. loop indices.
    • `#pragma omp` **`firstprivate(var1,..)`**: like `private`, but copies original value. Use when original value matters, e.g. constant calculated outside the loop.
    • `#pragma omp` **`shared(var1,..)`**: uses shared global memory, accessible by all threads. Used when multiple threads need to access same data, but may cause race condition.

    • `#pragma omp` **`default(shared|none)`**: sets the default variable scope for all unspecified outer variables used within the parallel region. `shared` (OpenMP default) makes them shared, while `none` forces you to declare every variable's scope, otherwise raising compile error.
    • `#pragma omp` **`reduction(op:var)`**: performs a reduction on `var` with operator `op` (e.g. `+`, `*`, `max`). OpenMP creates a private copy per thread, applies `op` locally, then combines all partial results into the shared `var` at the end, avoiding race conditions without explicit locks.

13. **Workload Decomposition**: `for`, `schedule` distributes (ideally evenly) work between all threads.

14. • `#pragma omp` **`parallel for`** shares iterations of a loop across threads.
    • `#pragma omp` **`parallel schedule(type, chunksize)`** balances overhead and data distribution. `type` can be:
    >> **`static`**, dividing loop iterations into blocks of chunksize iterations and assigning to threads in round-robin fashion.
    >> **`dynamic`**, assigning such new blocks to a thread each time it completes a block.
    >> **`guided`**: like dynamic, subsequent chunks decrease size.
    Default chunk size is 1.

Processor Organisation

1. In each **instruction (fetch-decode-execute) cycle**, the CPU fetches instruct from memory using PC (which is then incremented to point to next instruct). Then loads instruct into IR, decodes it to determine operation and operands; executes the instruct updating CPU state. Cycle repeats continuously, often in ≥ 1 CPU **clock cycle**.

2. CPU components include:
Arithmetic Logic Unit (ALU) math/logic operations, **Internal Processor Bus** between registers and ALU. **Control Unit (CU)** decodes program instructions and handles logistics for their execution, **Registers** storage.

3. **User-visible registers** used by low level programmers to minimise references to main memory. Includes *data, address, index, segment pointer and condition code* registers.

4. **Control and status registers**: used by CU to maintain processor operation and privileged OS programs to control program execution. Includes *PC, IR, MAR, MBR.*

5. **Instruction Set (IS)** is a collection of instructions that processor can execute. Has binary **opcode** specifying the operation (e.g., ADD R1, R2). **source** and **result operand** specifying inputs to the operation and where its result goes. Finally, **next instruction reference**: where to fetch next instruction.

6. **Opcode mnemonics**: `ADD, SUB, MUL, DIV, LOAD, STOR`.

7. Instruction types include:
data processing: arithmetic/logic ops (e.g. `ADD, SUB`),
data storage: move data to/from regisers/mem (`LD, ST`)
data movement: I/O transfer (e.g. `IN, OUT`)
control: test/branch instructions (e.g. `BEQZ, JR`).

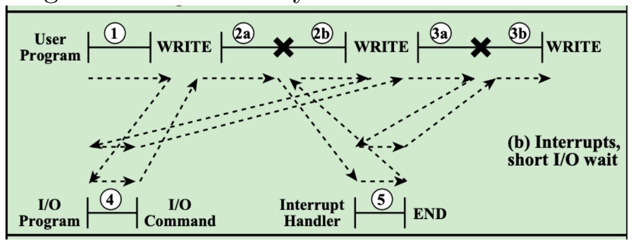
8. **Instruction Addressing Modes:**

Mode	`ADD R4, R3, <..>`	`R4 = R3 + <..>`
Register (reg)	`R2`	R2
Immediate	`#5`	5
Displacement	`100(R1)`	Mem[100 + R1]
Reg Indirect	`(R1)`	Mem[R1]
Absolute	`(0x475)`	Mem[0x475]
Mem Indirect	`@(R1)`	Mem[Mem[R1]]
PC-relative	`100(PC)`	Mem[100 + PC]
Scaled	`100(R1)[R5]`	Mem[100+R1+R5$\times$4]

9. Datatypes: ***Binary Integer*** (8/16/32/64)-bit, ***Floating Point*** (32/40/64/80)-bit, ***Address***(16/24/32/48/64)-bit. Can also have ***vector data***.

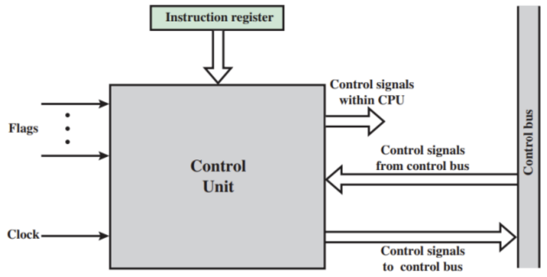
10. IS Architecture **(ISA)** encoding: **fixed length**: easy to decode (RISC, ARM), **variable length**: saves space (CISC, `x86`) and **very long instruction word (VLIW)**: multiple instructions in one fixed-length bundle (HP/ST).

11. **Indirect Cycle** introduces **interrupts** and **indirections** to account for indirect memory addressing potentially requiring additional memory accesses.



Interrupt classes include **program**: for errors in execution (e.g. segfault), **timer**: for periodic OS tasks, **I/O**: for I/O events or requests, or **hardware failure**: for crashes due to power or memory parity error.

12. **Control Unit** ***sequences micro μ-operations*** for processor and ***generates control signals*** that execute them. Control signals open and close logic gates, resulting in data transfer to/from registers and operation of ALU.


Instruction register
Control signals within CPU
Flags
Control Unit
Control signals from control bus
Control bus
Clock
Control signals to control bus

• Input: **clock**: for timing, **IR**: for opcode and addressing mode of current instruct (determining micro-operations to perform during execute cycle); **flags**: to determine status of processor and outcome of previous ALU operations; and **control signals from control bus**.

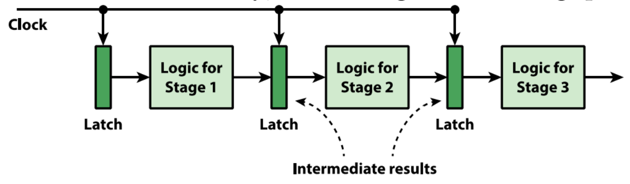
• Output: Control signals 1) **within processor**: to either move data between registers or activate specific ALU functions; and control signals 2) **to control bus**: to memory or to I/O modules.

13. **Hardwired CU** (RISC): implemented using digital logic circuits that directly transform input signals into output control signals. It uses a **sequencer**, driven by the clock, to step through control states for each instruction.
Pros: Fast
Cons: complex hardware: difficult to design and test. Inflexible: hard to add new instructs, long design time.

14. **Microprogrammed CU** (CISC): Uses a microprogram stored in memory to generate control signals for each instruction. Each instruction is broken down into a sequence of micro-operations executed via routines.
Pros: Easy reusable hardware design, supports complex instructions. Flexible: easy to add new instructs by reprogramming microprogram memory.
Cons: Slower than hardwired CU due to memory lookup.

Performance Pipelining

1. **Pipelining** exploits parallelism by having many operations execute concurrently, increasing data throughput.



2. Different ***pipe stages/segments*** complete different parts different instructions in parallel.
Throughput: how often an instruct exits the pipeline.
Processor cycle: time between moving an instruct one step down the pipeline.
Proc. cycle length = slowest pipe stage (1 clock cycle).
Balance length of each pipeline stage, in ideal conditions:
$$\text{Time per instruct} = \frac{\text{time per instruct on unpipelined machine}}{\text{number of pipe stages}}$$

3. **5-stage pipeline**: `IF` ⇒ `ID` ⇒ `EX` ⇒ `MEM` ⇒ `WB`: instruct fetch, decode, execution, memory access and write-back.
   - **`IF`**: send PC to memory, fetching current instruct. PC+=4, since each instruct is 4 bytes
   - **`ID`**: Decode the instruction and read the source registers specified in it (in parallel if RISC). Also computes possible branch target address (for branch instructs).
   - **`EX`**: ALU performs the actual computation using operands prepared in previous stage, e.g. arithmetic or logical operations, effective address calculation, or condition evaluation. The stage may also involve a bit shifter, or a multi-cycle multiplier or divider.
   - **`MEM`**: memory read from temporary buffer (load) or write data from the register to memory (store) using the effective address from EX.
   - **`WB`**: write result to register file (from ALU or memory).

4. **Performance Issues**: pipelining increases throughput, but not speed of a single instruction.
• **Stage imbalance**: slowest stage limits the clock speed.
• **Pipelining overhead**: delay from pipeline registers and maximum delay between stages (**clock skew**).
• **Pipeline latency**: instruction duration limits how deep the pipeline can go. Pipeline **depth** = # stages
Goal: keep pipeline ***correct, moving, full****.*

5. **Cycle time**: advance set of instructs one stage in pipeline:
$$\tau = \max_{1\le i\le k}[\tau_i] + d = \tau_m + d$$
for time delay on circuitry in i^{th} stage τ_i, max stage delay τ_m, # stages k and time delay d of latch advancing signals and data between stages.

6. **Total time** for k-stage pipeline to execute n instructs:
$$T_{k,n} = [k + (n-1)]\tau$$
The first instruction takes k cycles, and each of the remaining $n-1$ instructions completes in 1 cycle.

7. **Pipelining Speedup factor**:
$$S_k \frac{T_{1,n}}{T_{k,n}} = \frac{\text{no pipelining}}{\text{pipelining}} = \frac{nk\tau}{[k+(n-1)]\tau}$$

8. **Pipeline Hazards** occur when pipeline or its portion must stall as conditions don't permit continued execution (***pipeline bubble***). Conflicts: *resource, control, data.*

9. **Resource Conflicts (*structural hazard*)**: ≥ 2 instructs in pipeline need same resource, so serialise (slow).
• ***Schedule***: Eliminate contention: separate instruct and data caches, instruct buffers, or multi-ported memory.
• ***Stall***: Detect contention, stall one of the stages, inserting a *pipeline bubble* (a delay carrying no useful work).
• ***Duplicate***: Add more hardware s.t. each instruct can access independent resources in parallel.

10. **Control Conflicts (*procedural dependency*)**: Occur when the pipeline cannot determine the next instruction due to a branch or jump.
• ***Multiple Streams***: Fetch both paths by duplicating pipeline front-end. Costly due to register/memory contention and risk of fetching incorrect path.
• ***Prefetch Branch Target***: Prefetch both the branch target and next sequential instruction. If the branch is taken, the target is ready; otherwise, discard.
• ***Loop Buffer***: Small, high-speed memory storing recently fetched sequential instructions. Helps avoid memory access delay and is effective for loops.
• ***Branch Prediction***: Guess if a branch will be taken. *Static* methods ignore history, prediction taken always/never/by opcode. *Dynamic* use execution history for accuracy, use taken/not taken switch or branch history table.

11. **Data Hazards (*data dependency)***: conflict in access of operand location. `R3=..; R4=R3+1`, so have to stall `R4`, otherwise it will get wrong old value of `R3`. Solve using:
Schedule: programmer avoids hazardous instruction order (manual). **Stall**: hardware freezes earlier stages until the hazard clears. **Bypass**: forward results directly to dependent stages before previous instruction finishes. **Speculate**: assume no hazard; if wrong, discard and restart.

12. Instruct i occurs before j, both share register x; have:
Read After Write (RAW): j reads x before i writes x.
Write After Read (WAR): i reads x, then j writes x.
Write After Write (WAW): i writes x, then j writes x.
These don't occur in 5-stage pipeline, unless it allows to write in more than one pipe stage or proceed when a previous one is stalled. RAW is **true data dependency**.

$r_3 \leftarrow r_1$; $r_5 \leftarrow r_3$ (**RAW**)
13. $r_3 \leftarrow r_1$ $r_1 \leftarrow r_4$ (**WAR**)
$r_3 \leftarrow r_1$ $r_3 \leftarrow r_6$ (**WAW**)

WAR, WAW (**output dep.**) are dependencies on a register name, not a value.

14. Speedup comprises **fraction enhanced** ≤ 1 that can achieve **speedup enhanced** > 1. **Ahmdal's law**:

$$\text{Speedup}_{\text{overall}} = \frac{\text{time}_{\text{old}}}{\text{time}_{\text{new}}} = \frac{1}{(1 - \text{fraction}_{\text{enhanced}}) + \frac{\text{fraction}_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}}}$$

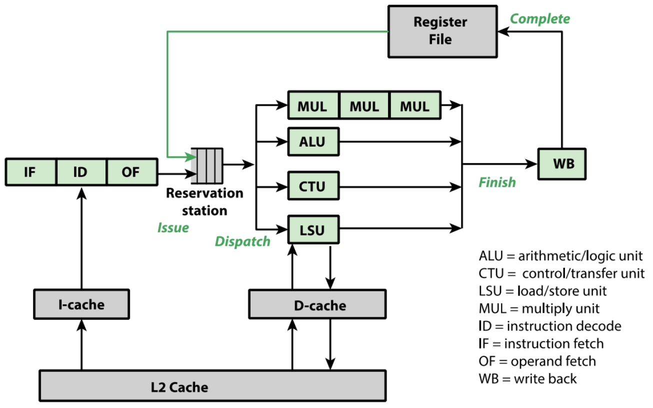
15. **Pipeline enhancements**:

1) **Data forwarding (bypassing)**: forward result value from memory or writeback stage to dependent instruction as soon as the value is available.

2) Separate $L1$ cache into I and D caches, removing conflict between `IF` and `EX` stages.

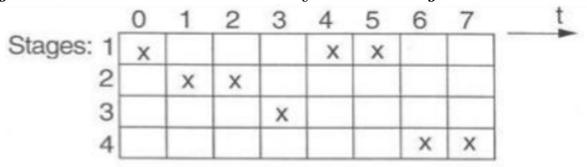
3) Dedicated execution units have different delays, allowing for more flexible pipelining.

4) Reservation station: buffer holding operations and operands for EX unit until operands are available.

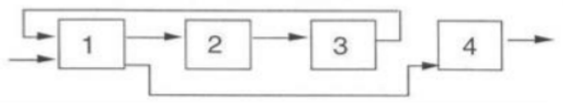


Pipeline Design

1. **Reservation table**: resource (pipeline stage) rows and time unit (clock cycle) columns. Cross "×" at row i, column j means station i is busy at time j.



Stages:	0	1	2	3	4	5	6	7
1	x				x	x		
2		x	x					
3				x				
4							x	x



2. But (i), (ii) indicate that pipelines may not accept initiations at start of every clock period, else **collisions** occur.

(i) Stages operate for ≥ 1 time period, "×"'s in adjacent columns of same row.
(ii) **Feedback**: ≥ 1 "×" in a row, non-adjacent columns.

3. **Latency**: # clock cycles between two initiations.
Average Latency (AL): avg # clock periods between initiations over repeating cycle.
Minimum AL (MAL) is smallest possible latency among all possible sequences of initiations.

4. **Collision** is an attempt by ≥ 2 initiations to use same pipeline resource at same time.

5. **Collision vector** $C_1, ..C_n$ where $C_i = 1$ is forbidden latency: initiating instruct i time units after preceding instruct causes collision. $C_i = 0$: permitted latency. $C_0 = 1$

6. **Initial collision vector** is pipeline state after first initiation. Compute by shifting the reservation to the right by 1 table, and check conflicts against itself. Can also compute forbidden latencies (subtract stages within same resource), and mark them 1. **SEE LEC 09 SLIDE 15**

7. **Latency cycle**: latency sequence repeating same subsequence indefinitely. Constant cycle has 1 latency value.

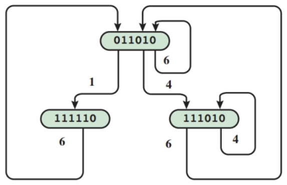
8. **Scheduling strategy (state diagram)**:

1. Shift collision vector left by 1, inserting rightmost 0. Each shift corresponds to increase in latency by 1.

2. Continue shifting p times until a **0 is shifted out from the leftmost bit**. Then p is a permissible latency.

3. Bitwise OR initial and each shifted register, producing a state node, connect nodes with edges labelled p.

4. Repeat this from each initial and newly created state.



9. **Simple cycles** visit each state once (e.g. $1 \rightsquigarrow 6$). A **greedy cycle** chooses minimum possible latency edge at each state, usually gives a good average latency, though not necessarily the minimum achievable (MAL). For example, $4 \rightsquigarrow 4$ is a greedy cycle starting from state 111010.

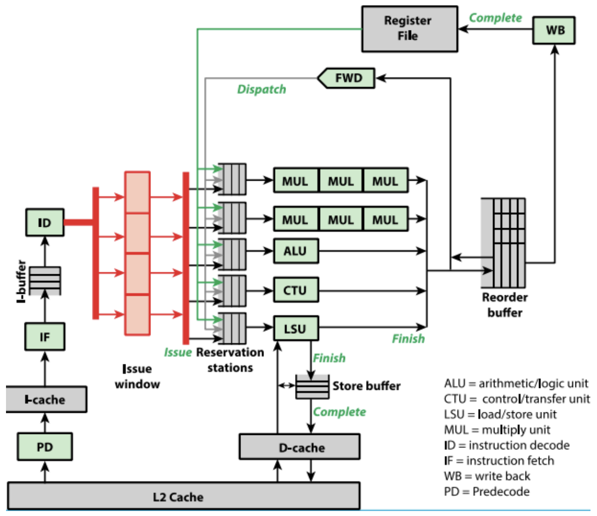
10. Max "×"'s in row $\leq$ MAL $\leq$ greedy cycle AL $\leq$ # 1s in initial collision vector + 1

11. **Reduce latency** by inserting delays D in the pipeline to expand reservation table, reducing chance of collisions. Typically, lower bound of delays $\leq$ any fixed latency.

Superscalar processors

1. **Superscalar**: fetch-decoding multiple instruct in parallel. Can reduce state-ready **CPI** (clock pulses per instruction) to less than 1. Applicable to RISC and CISC.

2. **Issue rate** is # instructs issued per instruction cycle. For superscalar, max issue rate is the width of issue window, or **degree of superscalar** σ. Stages following issue window are not duplicate, so called **non-uniform superscalar**.



3. Superscalar **Performance model IF ⇒ DR ⇒ EX ⇒ WB** has fetch, decode, execute, write stages; each is duplicated. Have s stages (pipeline length), N instructs per program fragment, and degree of σ. Then number of clocks required to execute program segment is:

Aligned	$s + (N/\sigma) - 1$
Non-aligned	$s + (N/\sigma)$

4. Probability of alignment is $1/\sigma$, so weighted avg # clocks:

$$\textbf{Avg clocks} = \frac{1}{\sigma}\left(s + \frac{N}{\sigma} - 1\right) + \left(1 - \frac{1}{\sigma}\right)\left(s + \frac{N}{\sigma}\right) = \boxed{s + \frac{N-1}{\sigma}}$$

$$\textbf{Superscalar CPI} = \frac{s + (N-1)/\sigma}{N} = \boxed{\frac{1}{\sigma} + \frac{1}{N}(S - \frac{1}{\sigma})}$$

$$\textbf{Speedup } S = \frac{\text{non-supersc. clocks}}{\text{superscalar clocks}} \frac{s + N - 1}{s + (N-1)/\sigma} = \boxed{\frac{\sigma(s + N - 1)}{\sigma s + N - 1}}$$

As $N \to \infty$, speedup $\to \sigma$, and as $\sigma \to \infty$, then speedup $\to 1 + (N-1)/s$, so superscalar speedup is limited by local parallelism of the program. Speedup increases linearly with σ until instruction level parallelism (ILP) limits further increase. ILP typically ranges $2 - 4$, so impractical to have σ too large, instead save on chip area.

5. **ILP** refers to degree to which instructs can be parallelised. Limited by true data, procedural, output and anti dependencies, and resource conflicts.

6. **Superpipelining** divides pipeline into more smaller stages to clock to higher frequency.

7. **ILP vs Machine Parallelism:** ILP overlaps independent instructions, limited by data/procedural dependencies. Machine parallelism is hardware's ability to exploit ILP, constrained by number of parallel pipelines. High ILP is wasted if machine parallelism is low.

8. **Instruction issue** is the process of initiating instruct execution in processor's functional units (move from decode to first execute stage of pipeline). **Instruction issue policy**: rules applied or protocol used to issue instructions.

9. Reorder independent instruct sequences to schedule in parallel, but requires processor lookahead to know order in which instructs are fetched, executed, and committed. Each requires 2 `DR`, 3 `EX` and 2 `WB` units running in parallel.

10. **In-order Issue, In-order Completion**: Instruction issue stalls if `EX` has a conflict or takes multiple cycles. Processor decodes instructions only up to the point of dependency or conflict; otherwise, stalls.

11. **In-order Issue, Out-of-order Completion**: Any number of instructions may complete out-of-order, limited by machine parallelism across `EX` units. Helps when instructions take multiple cycles. Stalled by resource conflicts, true data dependencies (RAW), and procedural dependencies. Introduces output dependencies (WAW).

12. **Out-of-order Issue, Out-of-order Completion**: Post-conflict instruct may be independent and execute earlier for improved performance. Decouples `DR` and `EX` pipeline stages: processor continues to fetch-decode into an instruction window. When `EX` unit is free, it selects a ready instruct with no conflicts or prohibiting dependencies. More instructs available for issuing, reducing chance of pipeline stall, but introduces anti-dependency (WAR).

13. **Register Renaming**: avoids anti- and output-dependencies (storage conflicts) by dynamically allocating hardware registers. When an instruction executes, it's assigned a new register, and subsequent source operands are renamed to refer to the correct version, ensuring correctness and avoiding false dependencies.

14. Instruct register reference `r#`, and hardware register `hw#`.

```
    I1: ADD r1, r2, r3   ⇒   ADD hw5, hw2, hw3
    I2: MUL r1, r4, r5   ⇒   MUL hw6, hw4, hw5
```

Avoids RAW error.

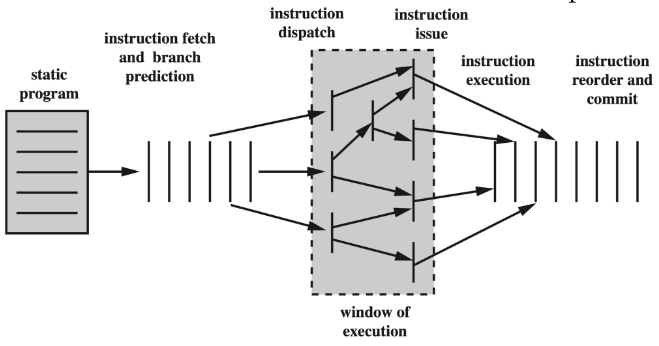
15. Branching can hinder pipeline performance. Prefetching has 2 pipeline stages between prefetch and execute: microprocessor prefetches next instruct after branch instruct, and branch target. Produces two-cycle delay when branch is taken, needs improvement.

16. **Delayed Branch strategy**: processor calculates branch results before prefetching any unusable instructions. Instruct following branch always executed, keeping pipeline full while new instructs are fetched. But ≥ 1 instructs must execute in delay slot, hard because of inter-instruct dependencies. So now very useful.

17. **Branch Prediction**, **speculative execution** like before

18. **Superscalar Execution**: static linear program transformed by branch prediction into dynamic instruct stream, which is dispatched to execution window. Each instruction is now structured by its data dependencies, and executed according to hardware resource availability. Finally, instructs are **committed/retired**, or put back in-order.

    Speculative execution may cause instructions to be completed and later discarded if the branch is mispredicted.



Data Level Parallelism

1. **Data Parallelism**: same operation applied to different data items in parallel (e.g. dot product), unlike thread parallelism where diff. instructs execute on diff. threads.

2. **SIMD Processing**: single instruct operates on multiple data elements using multiple processing elements. **Array processor**: multiple spaces in parallel. **Vector processor**: same space, executed over time.

3. **Vector Processors**: operate on 1D arrays of data (vectors) instead of scalars. Each vector instruct operates on all elements in consecutive cycles. Requires vector registers, vector length (VLEN), and stride (VSTR) registers.

    • **Execution**: pipeline stages operate on different vector elements in parallel, deeper pipelines. No intra-vector dependencies/control flow, so no interlocking needed.

    • **Limitations**: memory bandwidth is bottleneck if compute/memory operation balance not maintained, data improperly mapped to memroy banks.

    • **Advantages**: *No dependencies within a vector*: efficient pipelining and regular data-level parallelism; *Each instruction does a lot of work*: reduced fetch bandwidth; *Regular memory access*: easy vector prefetching, memory interleaving; *Loops can be implicit*: fewer branches

    • **Disadvantages**: Only works well with regular (SIMD) parallelism; Inefficient for irregular tasks (e.g., pointer-based search), performance improvement limited by vectorisability of code.

4. Each **vector register** holds up to N values, each M bits wide. `VLEN` specifies the number of elements to operate on (`VLEN` $\leq N$). `VMASK` is a bitmask indicating which elements to operate on, typically set by vector test instructions, e.g., `VMASK[i] = (`V_k` == 0)`. Also has vector stride `VSTR`.

5. **Vector Functional Units**: use deeply pipelined hardware to execute operations, each stage of which handles one vector element per cycle. Such vectors are independent, so control logic is simple; high throughput and efficient parallel execution. Has fast clock cycle.

6. Need to load/store multiple elements, separated from each other by a constant (usually 1) distance (**stride**). Load elements in consecutive cycles if can load one per cycle. To handle those taking longer than 1 cycle, need to bank the memory and interleave elements across banks.

7. **Memory Banking**: memory divided into independently accessible banks that share address and data buses (minimising pin cost). Start, complete one bank access each cycle and sustain N parallel accesses to different banks.

8. $\text{address}_{\text{next}} = \text{address}_{\text{prev}} + \text{stride}$. If stride=1 and consecutive elements interleaved across banks, number of which is $\geq$ bank latency, then can sustain 1 element per cycle throughput.

9. **Vector Chaining**: data forwarding from one vector functional unit to another to increase operation speed.

10. **SIMD Array Processors**: consist of a grid of identical **processing elements (PEs)**, each with its own local memory (1D or 2D mesh). A single control unit broadcasts the same instruction to all PEs, which execute it in lockstep on their own local data. Suited for data-parallel tasks like matrix operations, where each PE handles a portion of the data independently.

11. **(i) Vector processors** (*temporal parallelism*) streams 1 instruct to many data elements through a deeply pipelined functional unit, processing one element per cycle.
    (ii) Array processors (*spatial parallelism*) use many processing elements to execute the same instruction simultaneously, each on its own data.

    (i) VP parallelise over time, (ii) AP parallelise over space.

GPU

1. **Execution Model (Hardware)**: how hardware executes code: out-of-order execution, vector, array, dataflow processors, multiprocessor, multithreaded processor.

2. **Programming Model (Software)**: how the programmer expresses the code:

1. **Sequential (SISD)**: plain scalar code, one instruction on one piece of data at a time.
2. **Data Parallel (SIMD)**: programmer explicitly writes code to apply the same operation across multiple data elements in parallel.
3. **Multithreaded** (MIMD/SPMD): compiler generates threads to execute each iteration, each doing same thing on different data. Executed on MIMD or SIMT.
4. **Dataflow**: instructions fire as soon as their operands are ready, not in program order—ideal for fine-grained, irregular parallelism.

3. **Graphics Processing Unit (GPU)** is a SIMT Machine that uses sets of threads executing same instruction dynamically grouped into **warps**, executed by **Single Instruction Multiple Thread (SIMT)** model.

   Pros: SIMT can treat each thread separately, so execute independently, hence MIMD processing; warp grouping is flexible, so maximise benefits of SIMD processing.

4. **Warp** is a set of threads executing the same instruction (same program counter) on different data. Follows **Single Program Multiple Data (SPMD)** model. Not exposed to GPU programmers.

5. **Grid** is the code that independently executes on GPU comprising **thread blocks** (sets of threads) indexed by *block ID*. Different blocks can't communicate directly, must use atomic memory operations in global memory. Threads in a block can synchronise and share data via low-latency shared memory (max 512 threads per block).

6. Same instruct in different threads uses **thread id** to index and access different data elements. GPU hardware handles thread management, not OS.

7. CPU runs **sequential or modestly parallel** code, while GPU executes **massively parallel** kernels as blocks of threads. Parallel kernel of device: `KernelA<<<nBlk, nThr>>>(args);`, for `nBlk` # blocks and `nThr` # threads.

8. **GPU Streaming Multiprocessor (SM)** is the core unit containing multiple **Streaming Processors (SPs)** (CUDA cores) arranged as SIMD lanes. A **Warp Scheduler** picks a ready warp (SIMD thread), and the **Dispatch Units** issue its instruction simultaneously to all SPs, with each SP executing one lane. The scheduler can switch warps each cycle since threads are independent, hiding latency across many SIMD pipelines.

9. **Branching**: in SIMT, selected threads can be activated or deactivated s.t. instructs and data are processed only on active threads, while the local data remain unchanged on inactive threads.

10. Hardware is free to schedule thread blocks. Each block can execute in any order relative to other blocks.

11. **Warp-Level FGMT**, or fine-grained multithreading has one instruction per thread in pipeline at once (no interlocking); interleaves warp execution to hide latencies. Register values of all threads stay in register file, enables long latency tolerance of millions of pixels for operations such as memory access.

Symmetric Multproc & Cache Coherence

1. **Symmetic Multiprocessor (SMP)**: standalone computer with: ≥ 2 similar capacity and same memory access time processors, all performing same (symmetric) functions; share memory, I/O; internally connected by bus (passive medium). Controlled by integrated OS.

2. **SMP Organisation**: bus has `control, address, data` lines. **Direct Memory Access (DMA)** to I/O with:

   • ***Addressing***: distinguish modules on bus to determine src, dest of data.
   • ***Arbitration***: any I/O module can temporarily function as "master".
   • ***Time-sharing***: when one module is controlling the bus, others are locked out.

3. **Bus organisation** should be ***simple***, ***flexible***: easy to add processors to the bus; ***reliable***: device failure shouldn't propagate to whole system.

   Disadvantages: all memory references pass through common bus, limiting performance by bus cycle time. Fix by each processor having cache memory, reducing # bus accesses, but leads to *cache coherence* problems.

4. **Cache coherence**: if a word is altered in one cache, it should invalidate or update a word in another cache, so other processes must be notified of every change.

   • **Software solutions**: need less hardware, detect problems at compile time (faster), but takes conservative decisions, so inefficient cache utilisation.

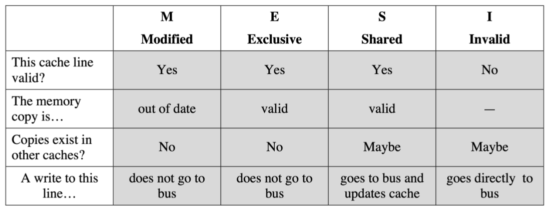
   • **Hardware solutions** (***cache coherence protocols***): runtime problem detection. Only deal with the problem when it happens, so better cache utilisation (faster). Transparent to programmer and compiler, so simpler.

5. **Directory Protocol**: a main memory directory records which caches hold copies of which shared block. On a write, consult directory to invalidate or update those caches, avoiding broad- casts. Scales to large, hierarchical interconnects but can become a central bottleneck and adds directory-storage overhead.

6. **Snoopy Protocol** cache controllers listen (snoop) on shared bus to detect reads/writes to lines it holds. On a write, controllers broadcast updates (***write-update***) or invalidate their copies (***write-invalidate***). It's simple on a bus-based multiprocessor but generates extra bus traffic and doesn't scale well to complex networks.

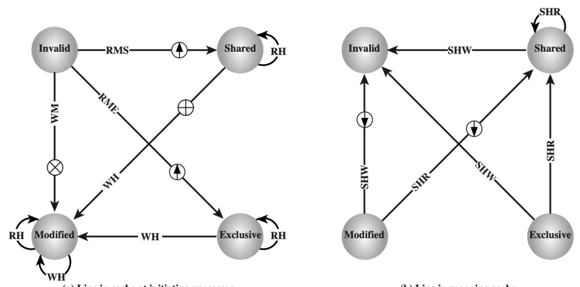
7. **Write-update**: multiple readers/writers, word to be updated distributed to all others on shared line, all update.

8. **Write-invalidate (MESI)**: multiple readers, one writer, invalidate all other caches on the line upon write, marking lines ***modified, exclusive, shared, invalid***; cache has 2 status bits for those. Uses snoop bus arrangement and write-back cache policy. Useful for multicore processors.



	M Modified	E Exclusive	S Shared	I Invalid
This cache line valid?	Yes	Yes	Yes	No
The memory copy is…	out of date	valid	valid	—
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line…	does not go to bus	does not go to bus	goes to bus and updates cache	goes directly to bus

RH	Read hit
RMS	Read miss, shared
RME	Read miss, exclusive
WH	Write hit
WM	Write miss
SHR	Snoop hit on read
SHW	Snoop hit on write or read-with-intent-to-modify
⊕︎↓	Dirty line copyback
⊕	Invalidate transaction
⊗	Read-with-intent-to-modify
⊕︎↑	Cache line fill



(a) Line in cache at initiating processor (b) Line in snooping cache

9. **MESIF (Intel)**: extends MESI with a *Forward* (F) state (special shared line owner). On a read miss only the F cache responds (avoiding multiple sharers replying or memory access), improving scalability in distributed/coherent systems.

10. **MOESI (AMD)**: adds an *Owned* (O) state to MESI—granting exclusive write rights while still supplying data to others—so that dirty lines can be forwarded directly from cache to cache without first writing back to main memory.

11. **Multiprocessor OS design considerations**: simultaneous concurrent processes, scheduling, synchronisation, memory management, reliability and fault tolerance.

Multithreading, Multicore Systems

1. **Process**: active program in memory with: ***resource ownership***, virtual address space to hold the data and ***scheduling*** to optimise execution order, interleaved using PCB stack **process switch** managed by OS.

2. **Thread**: dispatchable unit of work within a process. **Thread switch** is cheaper than a process switch since only the thread's context (not the entire process state) must be saved and restored.

3. **Multitasking**: OS-based switching between processes (or threads) via context switch. **Multithreading**: hardware-based partitioning of the instruction stream into multiple concurrent threads.

4. **Hardware Multithreading**: have multiple thread contexts in a single processor.

   Pros: memory, dependency and branching *latency tolerance* through *better resource and hardware utilisation*; improved system throughput (reduced context switch penalty) using thread-level parallelism and better superscalar/out-of-order (OoO) processor utilisation.

   Cons: require multiple thread contexts implemented in hardware (area, power, latency, cost) and reduced single-thread performance from resource sharing & contention; switching penalty.

5. **Fine-grained multithreading** (*cycle by cycle*): switch to another thread every cycle s.t. no 2 instructions from the thread are in pipeline concurrently. Improved pipeline utilisation, tolerates control/data dependency latencies by overlapping latency with useful work from other threads.

   Pros: no intra-thread dependency or branch-prediction logic, idle/bubble cycles reused for other threads, and overall higher pipeline utilization and latency tolerance.

   Cons: extra hardware complexity (multiple contexts and thread-selection logic), reduced single-thread throughput (one fetch per N cycles), increased cache/memory contention, and remaining cross-thread dependency checks.

6. **Coarse-grained multithreading**: switch to a different hardware context when a thread is stalled due to cache miss, synchronisation event of floating point operations.

   Pros: very low hardware complexity (no per-cycle thread-selection or branch/dependency logic), and switches happen infrequently.

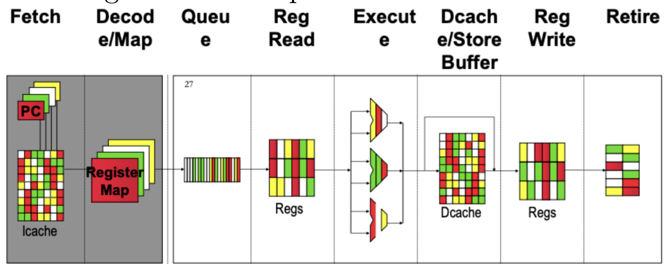
   Cons: each switch incurs a full pipeline flush (dead cycles) or complex hardware to save pipeline state, single-thread throughput drops to $1/N$ of the pipeline bandwidth.

7. Finer the granularity the less constrained the programmer is in parallelising a program, but the more significant part of the execution is taken by threading system overhead.

8. Functional Unit (FU) utilisation: data dependencies reduce FU utilisation in pipeined processors.

• **Superscalar** and out-of-order machines: FU is underutilised because of vertical and horizontal loss (wasted space in pipeline).
• **Predicated Execution**: convert control dependencies into data dependencies, improving utilisation but potentially discarding some results.
• **Chip Multiprocessor**: partition FUs across cores, still limited single-thread performance and FU utilisation.
• **Coarse-grained MT**: instructs from only one thread can be issued during any cycle; uses blocked MT.
• **Fine-grained MT**: still low single-thread performance and FU utilisation due to intra-thread dependencies.
• **Simulatenous Multithreading**: utilise FUs with independent operations from same or different threads.

9. **Simultaneous Multithreading (SMT)**: most common implementation, extends fine-grained multithreading on top of a multiple-issue, dynamically scheduled processor. Uses thread-level parallelism to improve FU utilisation by allowing instructions from multiple threads to issue in the same cycle. Enabled by register renaming and dynamic scheduling to resolve dependencies.



10. **Chip multiprocessing (multicore)**: entire processor (**core**) replicated on single chip, each such processor handles separate threads. Chip logic area is used effectively without increased pipeline design complexity.

11. **Interleaved, blocked MT** are ***concurrent***: better resource utilisation as delay event penalties avoided). **SMT, multicore** are ***parallel***: replicated execution resources, increased performance through parallelism.

12. **Pollack's rule**: performance $\propto \sqrt{\text{die area}}$ (*proportional*). Multicore has potential near-linear improvement, but only if software can take advantage.

13. **Multicore organisations** depend on: # on-chip cores; # levels and amount of cache memory shared. In general:

    a) Dedicated L1 cache, b) dedicated L2 cache, c) shared L2 cache and d) shared L3 cache.

14. **Shared higher-level cache** reduces overall miss rates (*constructive inference*), doesn't replicate data in multiple cores, amount of cache allocated to each core is dynamic. Interprocessor communication is easy (*shared memory*), moves cache coherency problem to lower cache levels.

15. Each core may employ superscalar or SMT. But, SMT with 4 cores each supporting 4 threads appears to OS the same as a processor with 16 cores, hence easier for software to fully exploit parallel resources.

16. Effective applications for multicore processors:
    • **Native MT**: *thread-level* parallelism, have small number of highly threaded processes.
    • **Multi-process**: *process-level* parallelism, have many single-threaded processes.
    • **Java**: embrace threading in fundamental way; JVM is multi-threaded process.
    • **Multi-instance**: virtualising technology provides some degree of isolation (secure) to application instances.

17. **HSA (Heterogeneous System Architecture)**: unified CPU–GPU architecture with shared virtual memory space, coherent cache policy, and unified programming interface. Enables CPUs and GPUs to access the same data and coordinate efficiently, supporting hybrid programs that exploit both CPU serial power and GPU parallelism. OS/hardware manage memory paging and coherence transparently.

18. **Uniform Memory Access (UMA)**: all processors have access to all main memory parts using loads and stores. Access time to all regions for different processors is the same. Standard for SMP, but doesn't scale well, limits # cores that can be put together, architecture dependant.

19. **Nonuniform Memory Access (NUMA)**: a single address space visible to all CPUs, access to remote memory via `LOAD` and `STORE` instructs, but slower than local memory. Allows for transparent system-wide memory.

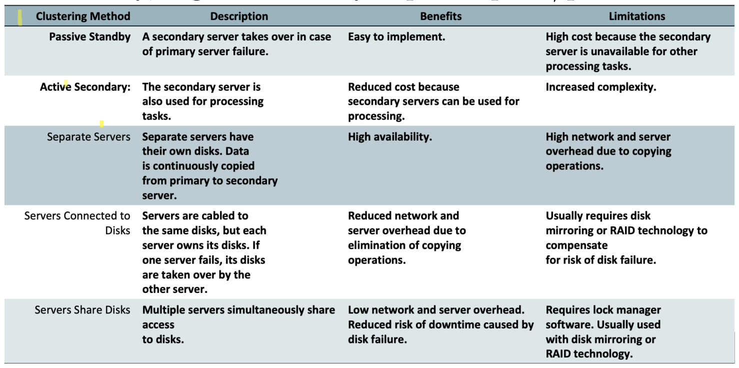
20. **Cache-coherent NUMA (CC-NUMA)** maintains CC among all caches in all processors. Caching helps with remote data access, but brings back CC issue. Can fix using bus snooping, but too expensive if many CPUs, so use **directory-based protocol**: associate each node with directory for its RAM blocks, DB stating in which cache block is located and what is its state.

    Pros: performant at parallelism levels higher than SMP. No major software changes. Avoid performance drop from remote memory access with good L1/L2 cache design, software with good temporal locality, and virtual memory systems that migrate pages closer to where they're used.

    Cons: not transparent, OS must manage page/process allocation and load balancing explicitly. Failure of a node is hard to recover from, raising availability concerns.

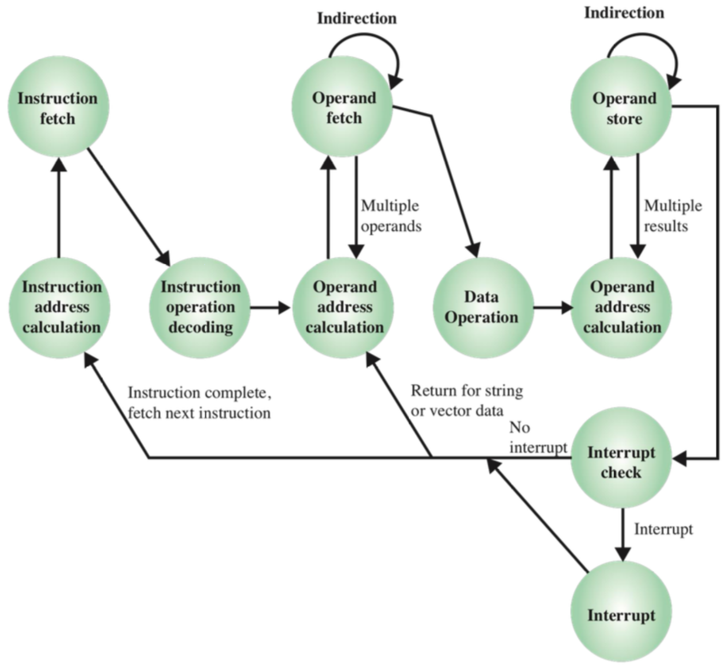
21. **Clusters**: group of interconnected computers (**nodes**) working together as a unified computing resource that can create the illusion of being one machine.

22. Clusters provide high incremental performance, absolute scalability, high availability superior price/performance.



Clustering Method	Description	Benefits	Limitations
Passive Standby	A secondary server takes over in case of primary server failure.	Easy to implement.	High cost because the secondary server is unavailable for other processing tasks.
Active Secondary:	The secondary server is also used for processing tasks.	Reduced cost because secondary servers can be used for processing.	Increased complexity.
Separate Servers	Separate servers have their own disks. Data is continuously copied from primary to secondary server.	High availability.	High network and server overhead due to copying operations.
Servers Connected to Disks	Servers are cabled to the same disks, but each server owns its disks. If one server fails, its disks are taken over by the other server.	Reduced network and server overhead due to elimination of copying operations.	Usually requires disk mirroring or RAID technology to compensate for risk of disk failure.
Servers Share Disks	Multiple servers simultaneously share access to disks.	Low network and server overhead. Reduced risk of downtime caused by disk failure.	Requires lock manager software. Usually used with disk mirroring or RAID technology.

23. Revisited: instruction execution cycle in a CPU:


Indirection
Indirection
Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data
No interrupt
Interrupt check
Interrupt
Interrupt

24. **Asymmetry**: enables specialisation, finding balance between purely general and special purpose approaches.

    Pros: Can enable optimisation of multiple metrics, adaptation to workload behaviours or special-purpose benefits with general-purpose usability/flexibility.

    Cons: Higher overhead in management (scheduling onto assymetric components) and more complexity in design, verification than symmetric design. Overhead in switching between multiple components can lead to degradation.

25. **Best if**: serialised code section $\rightarrow$ one powerful large core; parallel code section $\rightarrow$ many weak small cores. Small cores much more energy and area efficient than large cores. Large cores are power-inefficient: e.g., $2\times$ performance for $4\times$ area (power).

26. **Asymmetric Chip Multiprocessor (ACMP)** provides one large core and many small cores. Accelerates serial part with large core (2 units), executes parallel part on small and large cores for high throughput (12+2 units).

Feature	**Large Core**	**Small Core**
Instruction Order	Out-of-order	In-order
Fetch Width	4-Wide fetch	2-Narrow fetch
Pipeline Depth	Deeper pipeline	Shallow pipeline
Branch Prediction	Aggressive (hybrid)	Simple
Functional Units	Multiple	Few
Trace Cache	Present	Absent
Memory Speculation	Supported	Not supported