Equation Sheet (Y3S1) – Computer Systems Architecture

1. Processor Performance & Quantitative performance

$$speedup = \frac{t_{exec\ B}}{t_{exec\ A}} = \frac{perf_A}{perf_B}$$

$$speedup\ (\%) = (speedup - 1) \times 100\ (-ve \text{ is speedown})$$

$$overall\ speedup = \frac{t_{exec\ (without\ feature)}}{t_{exec\ (w/\ feature)}} = \frac{1}{(1-f) + \frac{f}{S}} \text{ (Amdahl's Law)}$$

$$t_{exec} = n_{inst} \times n_{clk\ per\ instr} \times t_{clk} = n_{inst} \times CPI \times t_{clk} \text{ (Iron Law)}$$

2. Memory Hierarchy

2.1. Cache Memory

$$capacity = n_{sector/track} \times size_{sector} \times n_{heads} \times n_{cylinders}$$

$$sustained\ data\ rate = n_{sectors/track} \times size_{sector} \times rotation\ rate$$

$$MAT = t_{seek} + t_{head\ switch} + t_{rotational\ latency} + t_{sector\ R/W}$$

$$h = \frac{n_{L1}}{n_{total}} \qquad T_{av} = h \times t_{L1} + (1-h) \times (t_{L1} + t_{L2})$$

$$e = \frac{t_{L1}}{t_{av}} = \frac{1}{1 + (1-h) \times \frac{t_{L2}}{t_{L1}}} \qquad (t_{av} \approx t_{L1}\ when\ h = 1)$$

$$\therefore T_{av} = h_1 \times t_{L1} \quad + \quad (1-h_1) \times h_2 \times (t_{L1} + t_{L2}) \quad + \quad (1-h_1) \times (1-h_2) \times h_3 \times (t_{L1} + t_{L2} + t_{L3})$$

2.2. Virtual Memory

(none)

3. Instruction Pipelining

$$t_k = n + (k-1) \qquad S_k = \frac{t_{non\ pipelined}}{t_{k\ pipeline}} = \frac{nk}{n + (k-1)}$$

$$S_k = \frac{k}{1 + n_{stalls\ per\ instr}}$$

4. Input/Output

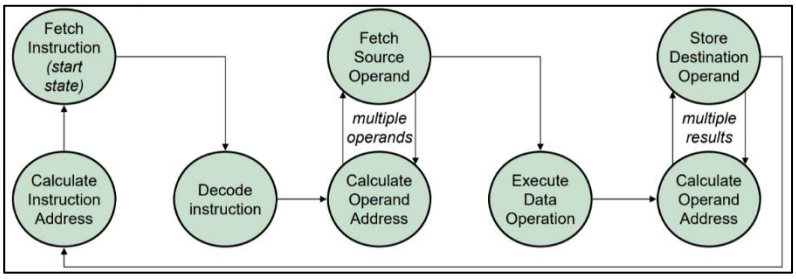
(none)

1. Processor Performance & Quantitative performance

Machine instructions: determines the **operations** undertaken by a processor

Instruction set: group of instructions composed of **all the machine instructions** a processor could execute

Instruction execution cycle: **execution structure** of instructions in processors

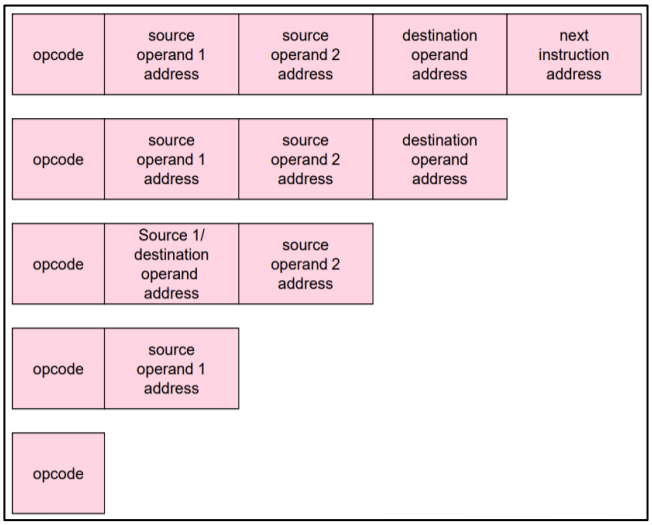


Operation Code (opcode): specifies **operation to be performed** (MOV, ADD…)

Source Operand /Destination Operand /Next Instruction reference: provides **memory location** of one or more **operands** / **next instruction** needed in the next operation

Instruction composed of (5): **number of operands** (0, 1, 2, 3), **operand storage** (memory or processor register), **operand address specification** (addressing modes), **data type** (byte, int, float), **operations** (add, sub, mul)

Architecture classification (4): done by addressing type (0 to 4), typically **load/store**, general purpose register (**GPR**), accumulator (**acc**), **stack**.



0-address: stack pointer register (**stack**)

1-address: accumulator register (**acc**)

2-address: shared source/destination operand (**GPR**)

3-address: program counter (PC) register (**load/store**)

4-address: (*out of scope*)

- Load/Store Architecture
  - 3-address add R_1, R_2, R_3 — $R_1 = R_2 + R_3$
  - 2-address load R_1, A — R_1 = mem[A]
  - 2-address store A, R_1 — mem[A] = R_1
- General Purpose Register (GPR) Architecture
  - 2-address add R_1, R_2 — $R_1 = R_1 + R_2$
  - 2-address add R_1, A — $R_1 = R_1$ + mem[A]
  - 3-address add R_1, A, B — R_1 = mem[A] + mem[B]
- Accumulator Architecture
  - 1-address add A — acc = acc + mem[A]
  - 1-address load A — acc = mem[A]
  - 1-address store A — mem[A] = acc
- Stack Architecture
  - 1-address push A — mem[++tos] = mem[A]
  - 1-address pop A — mem[A] = mem[tos--]
  - 0-address sub — mem[tos] = mem[tos--] - mem[tos]

(practice needed - see tutorial 1)

Addressing mode (7): specify address of operand in either register or memory for access in an optimum way

Immediate	ADD R_1, $3	; $R_1 = R_1 + 3$
Register Direct	ADD R_1, R_2	; $R_1 = R_1 + R_2$
Register Indirect	ADD R_1, (R_2)	; $R_1 = R_1$ + mem[R_2]
Memory Direct	ADD R_1, ($100)	; $R_1 = R_1$ + mem[100]
Indexed	ADD R_1, (R_2, R_3)	; $R_1 = R_1$ + mem[R_2 + R_3]
Displacement	ADD R_1, (R_2, $24)	; $R_1 = R_1$ + mem[R_2 + 24]
Scaled	ADD R_1, (R_2, R_3, $4)	; $R_1 = R_1$ + mem[R_2 + (R_3 * 4)]

Instruction types (3): **data transfer** (load, mov, store, input, output), **data operation** (arithmetic: add, sub, mul, div, inc, logic: &&, ||, >>, character manimpulation: compare, search), **program control** (jump, branch, call, return, interrupt)

Iron Law: **simpler instructions** (more instructions, fewer clk per instruction) || **complex instruction** (few instructions, larger clk per instruction) → expensive hardware implementation = fewer clock cycles (&VV)

$$t_{exec} = n_{inst} \times n_{clk\ per\ instr} \times t_{clk} = n_{inst} \times CPI \times t_{clk}$$

Microprogramming: technique that imposes an interpreter into hardware rather than software (microcode is a layer of hardware level instructions which allows cheap CPU to execute complex instructions of an expensive CPU)

Complex Instruction Set Computer (CISC): took over from "simple instruction set" 1960s, CISC resulted in **faster program execution** as **more was fit in one instruction**, but these **instructions took a bit longer** (faster software dev time).

Reduced Instruction Set Computer (RISC): took over from CISC 1980s, by avoiding the use of microprogramming for **faster program execution** as **less was fit in one instruction**, but these **instructions took less long**. All instructions directly executed by hardware (can be **pipelined** with short clk cycles), **lots of registers** on-chip, **load/store architecture** (most ops register-to-register with some rare load/store), **simple addressing mode**, **simple instruction format**.

CISC vs RISC: CISC **eases task of compiler** as processor instructions resemble high-level languages or HLL (however it **is hard to exploit** CISC as compiler must find cases to exactly fit HLL statements & **optimising general code is difficult**) and improves instruction efficiency to **reduce program size** (RISC state **memory is inexpensive** so program size is no issue, RISC can have shorter programs than CISC, RISC directly executed in **hardware and do not need interpretation so faster**)

Register windowing: **optimisation technique for reducing memory traffic on procedure calls** (several banks of registers are used and a new one is allocated on each procedure call). **Register file organised into windows**, behaves as **a small fast buffer** holding variables most likely to be used (as cache but faster, although **less capable of dynamic adapting** as compiler less good than cache replacement scheme). Register windows **not necessarily efficient** as not all procedures will need all available registers. Bad for **global variables** as not very dynamic, good for moving data between procedures.

ARM: best **low cost, low power, high performance** (mobiles, cameras…) using **3-5 cycle pipeline** (fetch, decode, execute, data, write back) & **load store**.

Trends: **processor performance & memory capacity increasing rapidly** but **memory performance failed to keep up** → solutions: improve performance of memory (hard), transfer more data between processor on memory via wider busses, optimise memory hierarchy (cache), pipeline instructions.

Speedup: performance measure to describe how much faster a computer (A) is from another (B).

$$speedup = \frac{t_{exec\ B}}{t_{exec\ A}} = \frac{perf_A}{perf_B}$$

$$speedup\ (\%) = (speedup - 1) \times 100\ (-ve \text{ is speedown})$$

Benchmarking: using special standardised **program for measuring speedup** (compiled & executed on any computer such that there is **no bias** to a particular architecture & should exercise significant hardware resources for solving typical/representative work)

Standard Performance Evaluation Corporation (SPEC): **benchmarking program distributor** keeping up to pace with current tech (SPEC95, SPEC2000, SPEC CPU2006 V1.2 = SPECint2006 & SPECfp2006, SPEC CPU2017. SPECint2006: **12 programs** (OO database, chess, word processor, C compiler, data compressor…). SPECfp2006: **14 programs** (computational chemistry, 3D graphics, seismic wave propagation simulator, image recognition, fluid dynamics…). Also output metrics as a **geometric mean** of **base & peak times** with **reference to 300MHz** Sun Ultra system.

Guidelines for design & analysis of architecture (4): Iron Law, Amdahl's Law, Locality of Reference, Parallelism - used for cost/performance trade-off.

- Amdahl's Law: improvement gained from using a faster mode of execution is limited by the fraction of time the faster mode can be used (i.e. improving rare tasks less useful than common tasks). Equation depends on two things: fraction of original task that can be converted to take advantage of new feature (f) & improvement gained by faster execution mode (S)

$$overall\ speedup = \frac{t_{exec\ (without\ feature)}}{t_{exec\ (w/\ feature)}} = \frac{1}{(1-f) + \frac{f}{S}}$$

- CPI: clock cycles per instruction used in **Iron Law**

$$CPI = \frac{n_{clk\ cycles}}{n_{instr}}$$

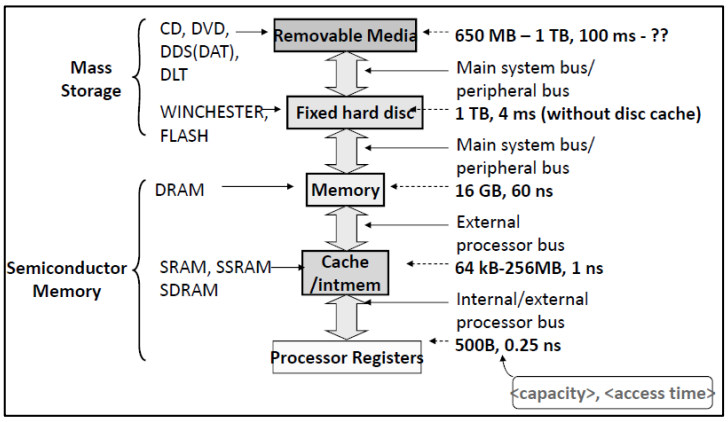
- Locality of reference: **reuse recent data & instructions** (two types: temporal & special locality - 90% t in 10% code) **Temporal** = reference in near future memory items from past, **Spatial** = reference portion of mem close to last mem ref

- Parallelism (3): **system level** (use multiple processors, multiple drives), **processor level** (pipelining), **detailed DSD level** (set associative caches use multiple banks of memory || ALUs use carry-look ahead which uses parallelism for computing)

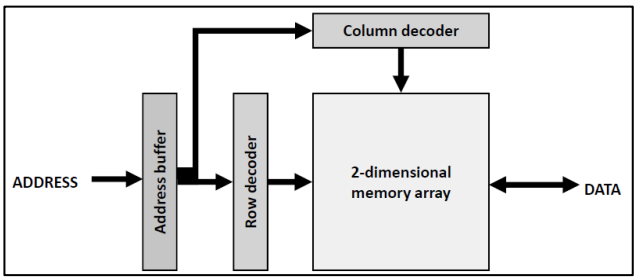
2. Memory Hierarchy

2.1. Cache Memory

Throughput: **requirement by processor** for **fast access to data** (i.e. 3GHz processor requires memory with access time of 0.33ns). As **MAT increases with increasing memory size**, different *types* of memory are required (**memory hierarchy**). Constant **trade-off with size** (larger smaller cost per bit & slower), **speed** (faster, greater cost per bit) & **price**.

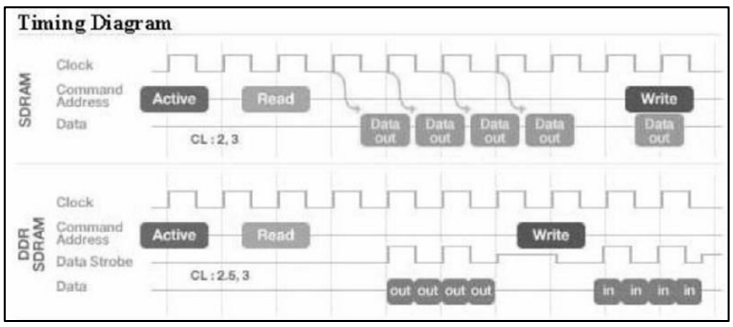


Memory types (2): **volatile** (static transistor/FF SRAM, dynamic trans/capacitor DRAM) & **non-volatile** (ROM, EPROM, NOR/NAND Flash…)

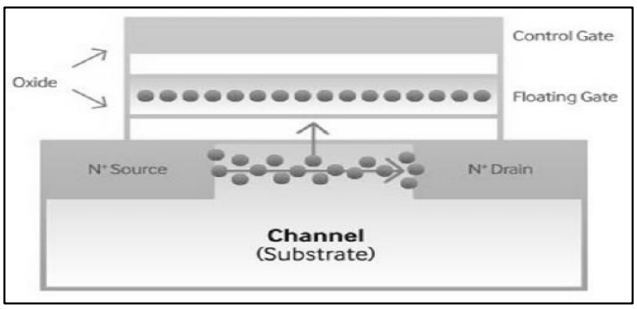


DRAM memory organisation: **2D memory** arranged in an N×N square array by splitting address lines in half for row/column sel.

DDR RAM: **Class of SDRAM** (synchronous dynamic) up to DDR4 (allows for higher transfer rates with stricter timings).



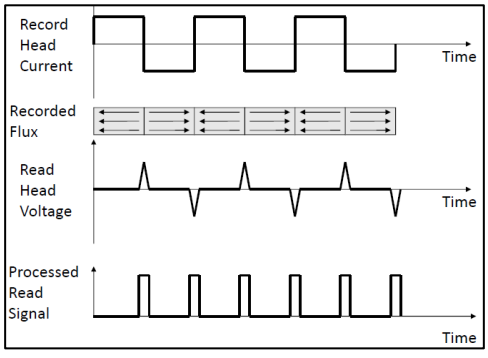
Flash memory organisation: floating gate used to allow trapped electrons to set transistor to 'low' state. Charging floating gate is done through electron tunnelling.



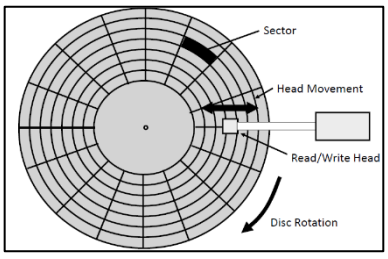
Types of flash (2): **NAND** (fast PROGRAM, fast ERASE, slow random access, slow byte PROGRAM - used for disks & sequential data) & **NOR** (fast random access, fast byte PROGRAM, slow PROGRAM, slow ERASE, replace EPROM).

Types of mass memory storage (4): **magnetic** (tape, HDD), **magneto-optical** (MO), **optical** (CD/CD-R/CD-RW, DVD/DVD-RAM) & **microelectronic** (SSD)

HDD: uses a **magnetic head** to **change state of magnetic coating** (encode)



- Disc formatting: minimises flux reversals (greater packing density), sufficient flux reversal needed though for clock reads.



- Sector formatting: organised as follows **[GAP][ID][GAP][DATA]** (allows for margin of error for slight difference in rotation rate & time for read/write head to switch between read/write operations (i.e. read the ID field before write). ID field format is constructed as follows - **[ID address mark] [Track Num] [Head Num] [Sector Num] [Sector Leng] [CRC1] [CRC2]**. Data field is as follows **[Data address mark] [User data] [CRC1] [CRC2]**. Only **one head active** at a time.

$$capacity = n_{sector/track} \times size_{sector} \times n_{heads} \times n_{cylinders}$$

$$sustained\ data\ rate = n_{sectors/track} \times size_{sector} \times rotation\ rate$$

$$MAT = t_{seek} + t_{head\ switch} + t_{cylinder\ switch} + t_{rotational\ latency} + t_{sector\ R/W}$$

$$t_{rotational\ latency} = \frac{1}{2 \times rotation\ rate} \qquad t_{sector\ R/W} = \frac{1}{rotation\ rate \times n_{sectors/track}}$$

Sustained data rate: rate of R/W data for a prolonged time with head/platter changes (contrast: interfaceDR - instantaneous)

Zoned bit recording: **outer cylinders have larger circumference** than inner, fixed number of sectors per track causes less **densely packed outer sectors** → adjust **sectors per track to maximise packing density** (capacity & data rate increased)

Cylinder skewing: **solved the issue of cylinder switch** taking finite time so disc needs to make 1 additional rotation for the first sector to be R/W if sectors of cylinders are aligned. Does so by **reordering** the sectors on each track. (+**head skewing**)

Program locality: **exploited by caches for faster average MAT**. Programs tend to **reference parts of their address space** (working set) **local in time and place**. When a block from main memory is loaded into **cache it also brings with it lots of useful nearby data**. Caching decreases program execution time by up to ×10.

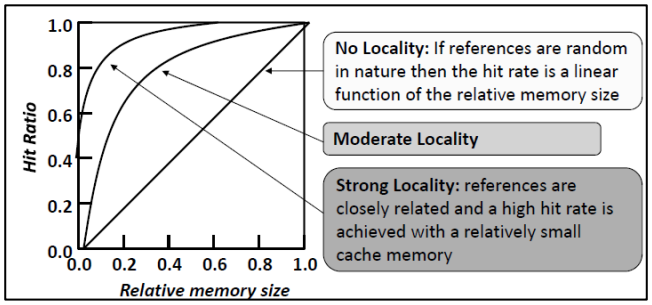
Latency: delay due to R/W (equivalent to MAT). Cache SRAM latency 1 to 10ns, Main Memory DRAM latency 50-100ns.

Cache hit/miss: when processor attempts R/W, checks if data item is in cache there, if yes is a **cache hit** else **cache miss**.

Main memory structure: 2^n **bytes** divided into fixed size **blocks** with K bytes per block ($size = 2^n \times K$). **Block size = line**

Cache memory structure: 2^m **bytes** divided into fixed size **lines** with K bytes per line ($size = 2^n \times K$). Also **contains tag**.

Cache hit rate: **number of cache hits** compared to **total memory accesses**. If program has **high locality**, **even small caches** will yield **hit ratios larger than 0.75**.



Hit ratio (h): number of hits in the lowest cache level L1 compared to total number of accesses

$$h = \frac{n_{L1}}{n_{total}} \qquad T_{av} = h \times t_{L1} + (1-h) \times (t_{L1} + t_{L2})$$

Access efficiency (e): measure of **how close average access time** is to the **L1 access time** (t_{T1}).

Typically, $\frac{t_{L2}}{t_{L1}} = 10 \therefore h = 0.9$ & $e = 0.5$

$$e = \frac{t_{L1}}{t_{av}} = \frac{1}{1 + (1-h) \times \frac{t_{L2}}{t_{L1}}} \qquad (t_{av} \approx t_{L1}\ when\ h = 1)$$

Block placement/replacement policy: goes to allocated spot dictated by memory mapping / go to place of the assigned spot

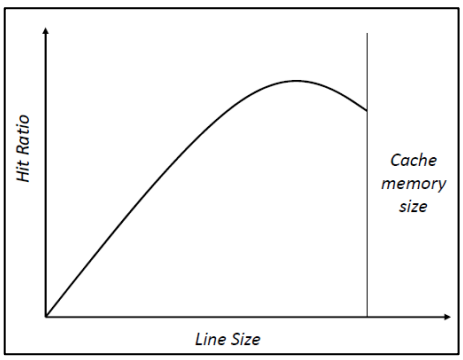
Block identification policy: uses tag for the assigned spot. Number of bits for tag depends on address mapping mechanism.

Memory mapping mechanism (3): **direct mapped** (given block placed in only one given line, only one tag comparison is made, small simple hardware overhead, low performance) || **fully associative** (block placed in any preferably unused cache line, all tags in cache compared simultaneously, high overhead as replacement algorithm needed, high performance) || **set associative** (block placed in one of a subset of cache memory lines, all tags in set compared simultaneously, good trade-off)

Types of cache misses (3): **compulsory cache miss** (first access is a block not in cache so must be a miss), **capacity cache miss** (if cache can't contain all blocks of program, capacity misses due to block replacement), **conflict cache miss** (if set associative or direct mapped, conflict miss occurs if a block is discarder and later retrieved if too many blocks map to its set)

Block replacement strategies (4): only relevant for associative caches - least recently used (LRU), first in first out (FIFO), least frequently used (LFU), pseudo random or round robin (RR)

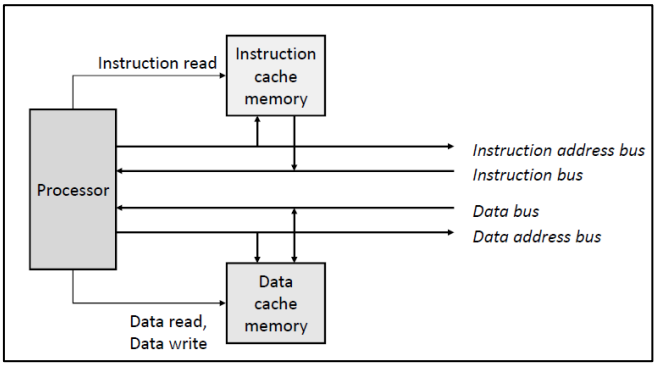
Write policy (2): **write-allocate** (load block into cache then write to cache, usually uses **write back** - all write operations made to cache memory and only when a written-to-line is replaced then it is written to main memory so main can be invalid for long) || **write-no-allocate** (write to main memory directly without loading into cache through intermediate write buffer, usually uses **write through** - all write operations made to main memory and cache memory so main is always valid for a performance compromise)



Line size: **as it increases hit ratio increases as locality is exploited, trade-off as if too big reduces lines that can fit** so **useless data is stored**

(practice needed - see tutorial 4)

Cache splitting (2): **unified cache** (stores both data and instructions) || **split cache** (L1D for data, L1P for instructions - modern approach as for set memory unified has lower CHR, allows for pipelining while unified does not as instruction fetched from cache while data written to cache)

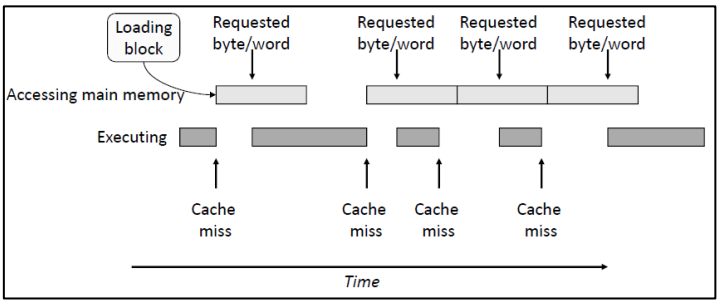


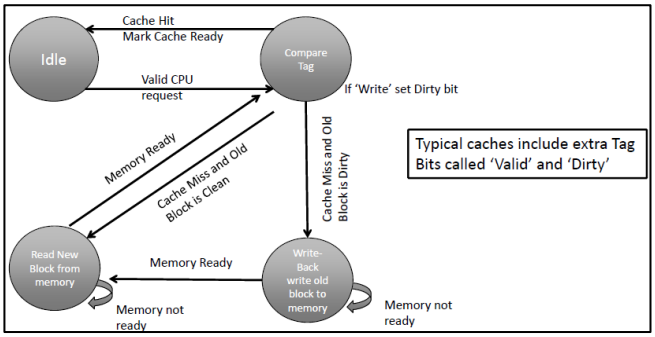
Number of cache levels: typically L1, L2 (fits on-chip) with L3 (must be off-chip). On-chip cache **significantly reduces external processor activities** & **faster access time** than off-chip. *Update: now up to L4 (victim cache for replaced data)*

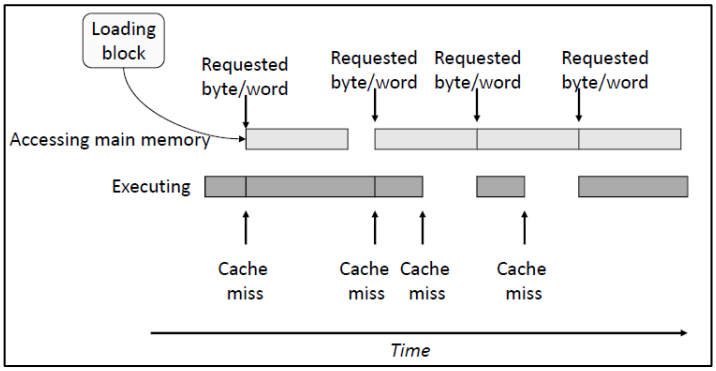
$$\therefore T_{av} = h_1 \times t_{L1} \quad + \quad (1-h_1) \times h_2 \times (t_{L1} + t_{L2}) \quad + \quad (1-h_1) \times (1-h_2) \times h_3 \times (t_{L1} + t_{L2} + t_{L3})$$

Improve cache performance (3): minimise $T_{av} = hit\ time + (miss\ rate \times miss\ penalty)$ via decrease miss rate || miss penalty || reduce hit time.

Techniques to minimise miss penalty (2): when cache miss occurs, entire block being loaded stalls (pauses) processor. Can be helped by **early restart** (resume execution as soon as requested byte of block is loaded in cache - works for instruction cache it's sequential but not so for data caches) || **requested word first / critical word first** (block load starts at address of the requested byte/word first followed by remainder of block with wrap around - faster than early restart but still does not solve penalty due to quick succession of cache misses).



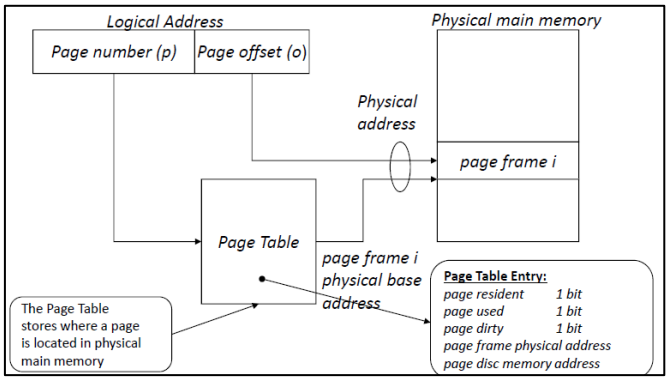




2.2. Virtual Memory (*CS)

Virtual memory: technique used by contemporary processors to **address more memory locations than exist in physical main memory** to bypass the issue of a **program being larger than main memory**. OS **swaps** programs and data in and out of secondary storage (HDD) as required. Each process has its own logical address space. Supports memory protection.

Memory Management Unit (MMU): virtual memory like cache but instead of moving between cache and main, move between main and swap file on HDD. Managed by MMU.



Virtual memory implementations (2): **demand paging** (fixed sized blocks) || **segmentation** (variable sized blocks)

- Page fault: when **reference made to page not currently in main memory** (**LRU** and **FIFO** used for **page replacement** in main memory). Each page has a **dirty bit** in **page table entry** to indicate contents have been **changed since** it was **loaded from disk**. **Write back** always used as miss penalty is very harsh (millions of clock cycles).

- Thrashing: occurs when a program generates page faults frequently

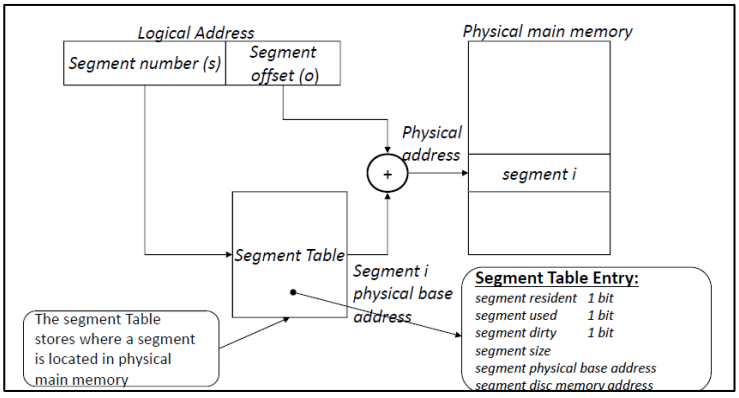
- Average memory wastage in the last page: $n/2$ bytes for a page size of n bytes

- Page directory: each program has its own pages thus page directory used to identify sets of pages for a program

- Segmented VM: **visible** to programmer unlike paging (which is handled by OS). Segments are **dynamic in size**. **Segment fault** occurs when reference to a segment is made that is not currently in physical main memory. Allows multitasking OS to have several programs resident in physical main memory simultaneously.

External fragmentation: occurs as holes occur between segments (holes are fit either by **periodic compacting**, **best fit** || **first fit**).

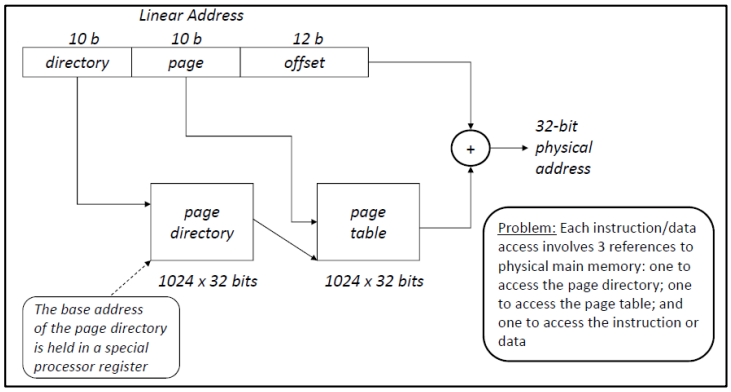
-	Paging	Segmentation
Visible to prog?	No	Yes
Block replacing	Trivial	Hard
Memory η	Internal p. frag	External p. frag
Disk traffic η	High	Low
Primary reason for invention	Simulate large memories	Provide multipl address spaces



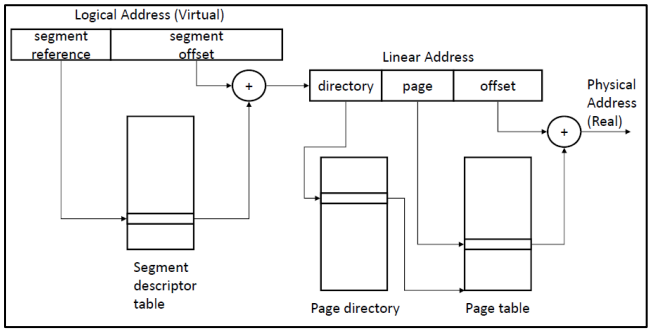
Multiplexed Information and Computing Service (MULTICS): early OS to **combine segmentation with paging in linked list**. **Protection bits** added to the segment or page rather than individual locations. OS may allow **more than one process** to **access the same segment**.

Modes of Virtual memory (4): **unsegmented-unpaged** (logical same as physical, useful for low complexity high performance controller), **unsegmented-paged** (memory viewed as paged linear AS - protection and memory management at page level), **segmented-unpaged** (memory viewed as collection of logical AS - protection down to the single byte), **segmented-paged** (logical memory partitions subject to access control & paging is used to manage allocation of memory within partitions, used by Unix/Win OS).

Address architecture: usually VA has 16-bit segment reference & 32-bit offset reference with 2 bits for protection & 1-bit for segment local/global.

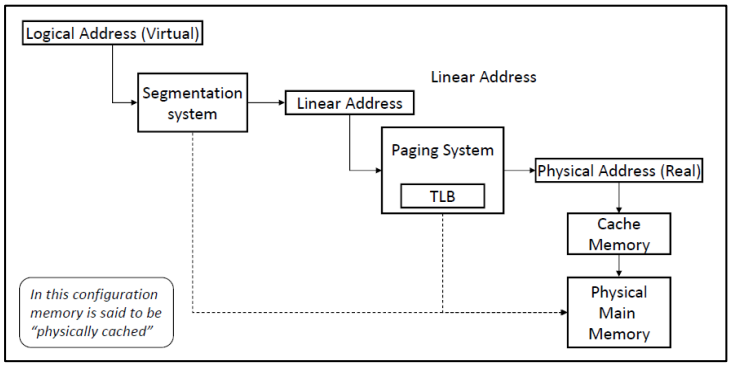


Protection (2): 2 bits used - **privilege level** (0 OS only, 1 system calls, 2 libraries, 3 user programs) & **access attribute** (in data segments specifies if read only or read-write access allowed, for code segments specifies read-only or read-execute).



(practice needed - see tutorial 5)

Translation lookaside buffer (TLB): used for special cache memory for page table entries (locality of reference states most VA accesses will be to recently used pages)



3. Instruction Pipelining

Pipelining: implementation technique where **multiple sequential instructions are overlapped in execution**. Takes advantage of **parallelism** (key method for faster processors). Each stage is called a **pipe stage** (undertakes instruction fetch || instruction decode || operand fetch…) and these stages connected to form a **pipe**. **Throughput** determined by frequency at which instructions are executed (all stages must be ready to proceed at the same time). Time between moving an instruction one step down the pipeline is a **processor cycle** (usually 1 - 2 clock cycles.

Pipelining advantages: **reduces average execution time** (can be viewed as **reducing CPI**), **not visible** to programmer.

DLX processor instruction cycles (5): each takes **1 clock cycle** - instruction fetch **IF** (get instruction to memloc pointed to by internal PC into IR & increment PC by 4), instruction decode **ID** (instruction in IR decoded so processor knows what operation to perform), instruction execute **EX** (e.g. data operation), memory access **MA** (load, store || branch instructions only), write back **WB** (load, register-register ALU || register-immediate instructions only).

Unpipelined: branch = 4 (IF, ID, EX, MA), store = 4 (IF, ID, EX, MA), load = 5 (IF, ID, EX, MA, WB), arithmetic/logic = 4 (IF, ID, EX, WB)
$\therefore CPI = f_{load} \times t_{load} + (1 - f_{load}) \times t_{others}$ [*here is 4.1*]

Pipelined: New instruction started every clock cycle so CPI = 1

Hardware resources utilised: IF (MM), ID (R), EX (ALU), MA (MM), WB (R)

	Clock Cycle Number								
1	2	3	4	5	6	7	8	9	10
IF	ID	EX	MA	WB					
	IF	ID	EX	MA	WB				
		IF	ID	EX	MA	WB			
			IF	ID	EX	MA	WB		
				IF	ID	EX	MA	WB	
					IF	ID	EX	MA	WB

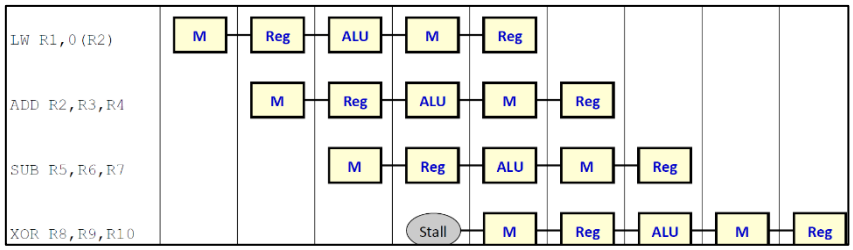
Performance issues: pipelining **increases instruction throughput** but **not execution time** of each instruction. Pipeline usually **requires overhead for pipeline control**. Branching may be an issue.

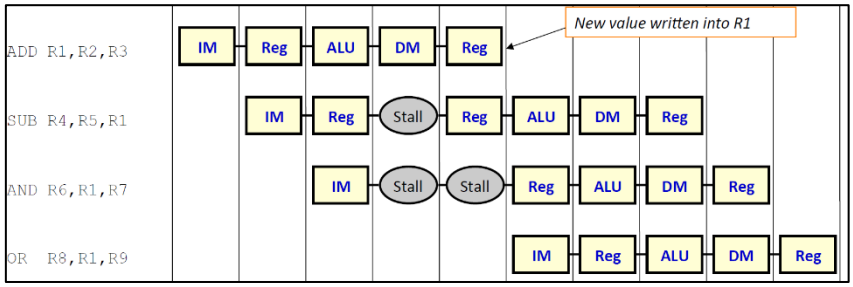
Speedup of pipeline: if t_p is time to advance a set of instructions one stage (*1 for DLX*), k is number of pipeline stages (*5 for DLX*), n instructions with no branches, t_k total number of clock cycles to execute a sequence of n instructions.

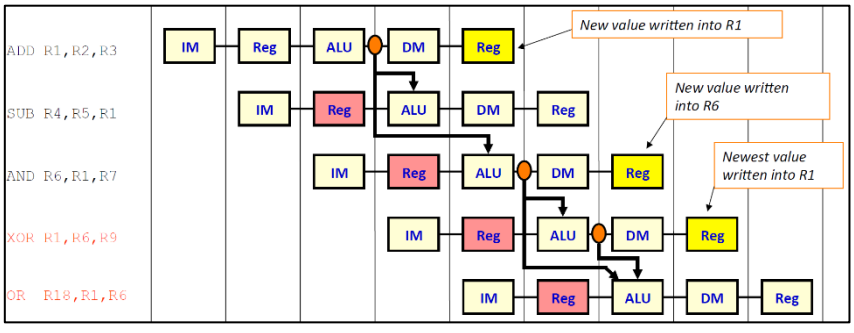
$$t_k = n + (k - 1) \qquad S_k = \frac{t_{non\ pipelined}}{t_{k\ pipeline}} = \frac{nk}{n + (k - 1)}$$

Scaling: speedup does not scale linearly as pipeline stages cannot increase indefinitely (pipeline control overhead increases clock cycle time, dependencies causing hazards - *typically pipelines of up to 31 stages*)

Pipeline hazards (3): **structural hazards** (finite hardware resources cannot be used by many instructions at once), **data hazards** (instruction depends on result of previous instruction that is still in pipeline), **control hazards** (branch instructions change the PC in an unpredictable way).







Pipeline speedup including stalls:

$$S_k = \frac{k}{1 + n_{stalls\ per\ instr}}$$

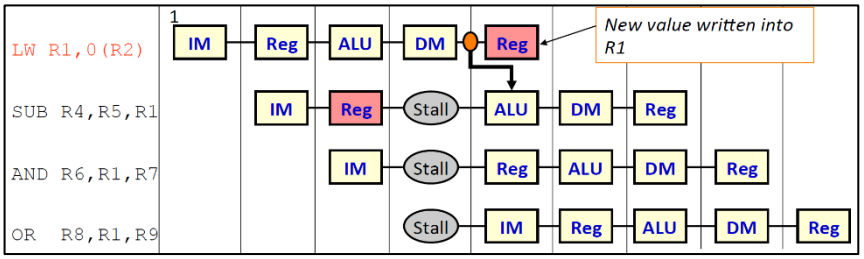
- Structural hazard avoidance: **split cache** so processor can access both memories at once

- Data hazard avoidance: **forwarding** (a.k.a. **bypassing**) allows ALU to feed data back to its input before transfer to memory.

(practice needed - see tutorial)

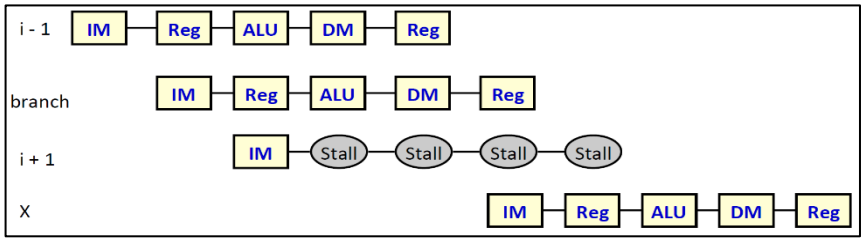
Data hazard classification (3): **RAW** (read after write - B reads before A writes so B gets old val), **WAW** (write after write - A writes before B writes so value updated incorrectly), **WAR** (write after read - B writes before read by A so A incorrectly gets new value)

Pipeline interlock: LW instruction has inherent delay as data memory is only read at clock cycle 4 so forwarding can only happen thereafter - requires a new hardware feature called **pipeline interlock** (detects LW data hazard and stalls pipeline until hazard is cleared).



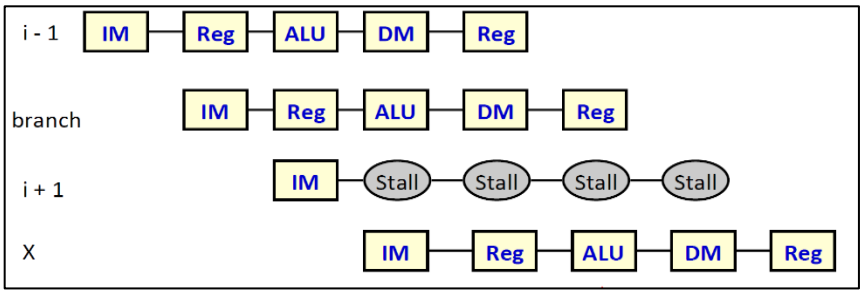
Compiler Instruction reordering: compiler generates code that avoids pipeline stalls by reordering logic.

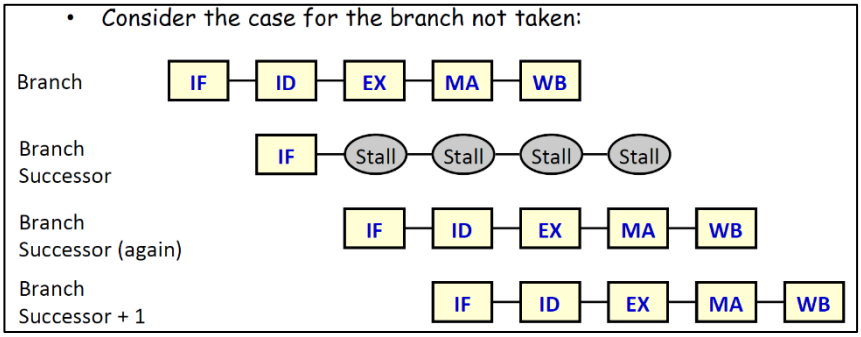
- Control hazard avoidance: control hazards cause **much larger performance loss** (typically) as **conditional branch update** value of PC **after MA stage (DM)** so **pipeline must stall**. Branch works by reading value of resulting register



Solutions (2): **determine whether or not branch is taken earlier** in pipeline, **compute target address earlier** in pipeline (achieved via additional hardware adder)

→ compute branch condition at end of ID (Reg1) stage
→ computer branch target address at end of ID (Reg1) stage

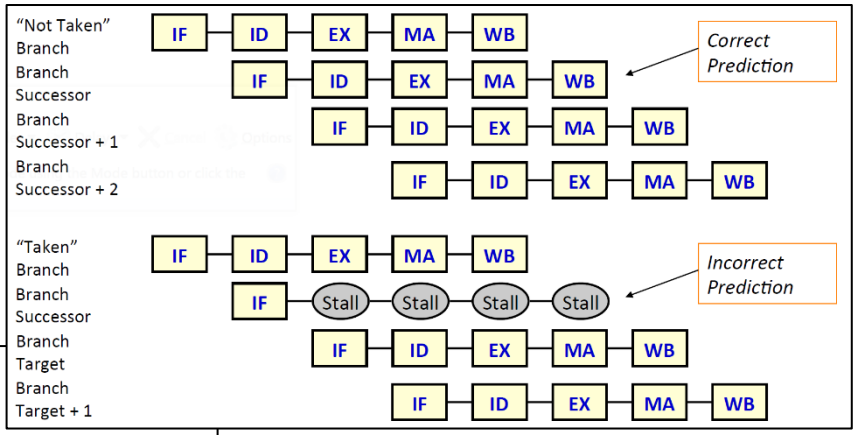


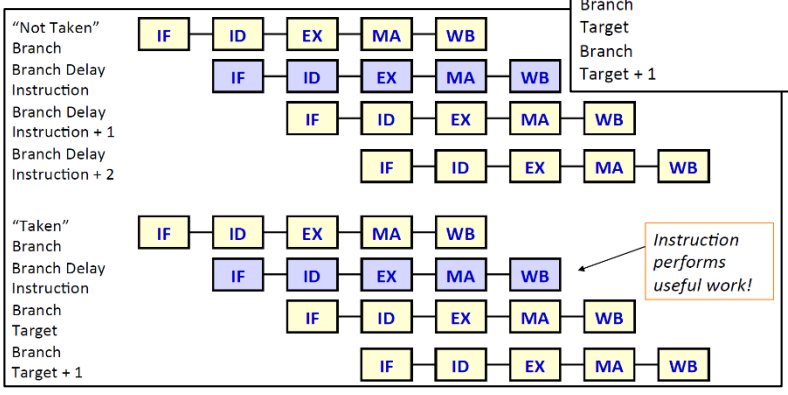


Cases: branch taken (bit redundant as you need to wait anyway) or not.

Scheme 1: **predict every branch as "not taken"** (static) - processor continues to fetch as normal (although this only slight improvement as most the time branch is taken - from statistical analysis)

Scheme 2: **delayed branch** (static) - sequential successor instruction of a branch contained in branch delay slot, instruction in delayed branch slot is **always** executed irrespective of branch taken or not, branch if taken is delayed until after instruction that succeeds it is executed (branch delay of 1 always used, delayed branch does not cause latency & compilers used to make successor instructions valid & useful)



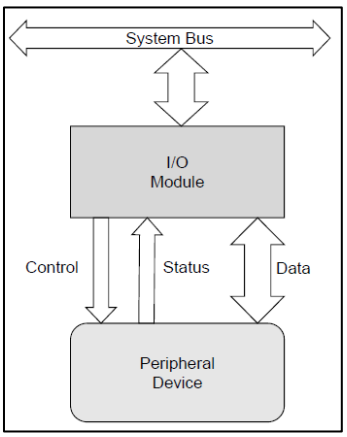


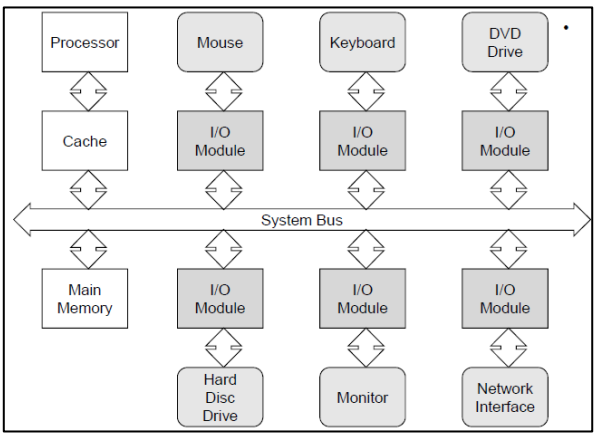
Scheme 3: **dynamic branch prediction** - record branch history statistics to attempt to improve accuracy of prediction (using additional circuitry & single bit taken / not taken bit & two bits for results of last two instances of branches). If 2 history bits wrong, change prediction decision.

If branch taken, target instruction cannot be fetched until branch target address calculated → solved by initiating IF as soon as branch decision is made & storing it in a branch target buffer (small cache with a table containing (3) - **address** of branch instruction, **history bits** recording prediction state & **address of associated target instruction**.

(practice needed - see tutorial L9)

4. Input/Output (*CS)





IO module: **devices interfacing with the system bus** and **control one or more peripheral devices** (e.g. SATA disc drives, mSATA for SSD, VGA/HDMI/Displayport…)

Peripheral device classification (3): **human readable** (keyboard, monitor, mouse, mic), **machine readable** (HDD, scanner…) & **communications** (LAN, WAN…)

Control signals: determine **function** of **peripheral devices** (send data to IO module / accept data from IO module / report status / perform specified action)

Data signals: set of **bits (serial or parallel) to be sent or received**

Status signals: indicates **form of peripheral device** (ready to receive data from IO module, data available, error)

Data rate: amount of **data (in B) transferred per second** within computer or between a peripheral device and the computer

Data rate margin: extent to which **transfer capacity exceeds data rate** (data rate limited by a **bottleneck** at which data rate margin is 0)

IO techniques (3): **programmed IO** (processor executes instructions that give direct control of IO including device status, sending read or write commands & transferring data - uses **busy/waiting** a.k.a. **polling**), **interrupt driven** (processor initialises IO, continues to execute other instructions until interrupted by IO module when peripheral device is read - uses IACKs & context switches), **direct memory access DMA** (processor instructs DMA controller to perform a data transfer with a start address & number of B & mode of transfer and leaves DMA to perform transfer **without any further processor involvement**).

IO technique drawbacks: **PIO is faster** than IDIO as IDIO requires context switches but **ID is more efficient** long run, for PIO and IDIO, IO data transfer rate limited by frequency with which processor can test and service peripheral device so **for large volumes DMA is most efficient** but requires an additionally small processor (DMAC) which sends and interrupt when data transfer is successful/failed.

IO addressing modes (2): **memory mapped IO** (processor has single AS/commands for both memory and IO modules, processor treats control status and data of IO modules as memory locations), **isolated IO** (a.k.a. IO mapped or port mapped - processor has separate AS for memory locations and IO modules, specific IO commands)

Circular buffer: two buffer pointers BufEnd & BufStart incremented as appropriate when data enters or leaves (empty when BufStart = BufEnd, full when BufEnd+1 modulo Buf_Size = BufStart.

Bus masters: processors/devices capable of taking control of the system bus of another processor (DMAC uses **bus request BR** signal to the **bus arbiter** which responds with a **bus grant** signal)

DMA transfer modes (3): **burst mode** (simplest - entire transfer completed without a break and will keep system bus idle for a while if peripheral device cannot keep up with speed of system bus, not very efficient), **cycle stealing** (single data transfer initiated for each DMA request made by peripheral - useful for peripherals slower than system bus, efficient as other bus masters use TDM to still access system bus) || **transparent DMA** (DMA transfers performed independently of system bus and require dual-port memory - truly concurrent with activity on bus, DMAC uses a separate DMA transfer bus)

DMA transfer types (2): **two cycle transfer** (for each word transferred in response to a DMAREQ first system bus cycle DMAC reads word from source and stores in internal register, second bus cycle sends word to memory or other IO module - not efficient, flexible as memory-memory supported) || **one cycle transfer** (for each word transferred in response to a DMAREQ DMAC places memory address on system bus and issues DMAACK to select IO module - efficient as reading source takes place in same bus cycle, less flexible since IO module not addressed so memory-memory impossible)

(practice needed - see tutorial L10)