Number systems

- Amount of infos D to stor in a var with N distinct values in bits: $D = \lg_2 N$
- Powers

	1	2	4	8	16	32	64	128	256	512	1024	2048	4096			
	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{20}	2^{30}	2^{40}
	16^0				16^1				16^2				16^3			
$\sim$											10^3			10^6	10^9	10^{12}

- Systems
  - Unsigned $[0, 2^N - 1]$
  - Signed $[-2^{N-1}+1, 2^{N-1}-1]$
  - Two's C. $[-2^{N-1}, 2^{N-1}-1]$
    ↳ Take it: – invert all digits – add 1

Boolean Algebra

- Axioms
  1) Contains (at least) two elements 1, 0 where $0 \neq 1$
  2) Closed $ab \in B$ $a+B \in B$ $\forall ab \in B$
  3) Commutative $ab = ba$, $a+b = b+a$ $\forall ab \in B$
  4) Identities $a1 = a$, $a+0 = a$ $\forall a \in B \exists 1,0 \in B$
  5) Distributive $a(b+c) = ab + ac$, $a+(bc) = (a+b)\cdot(a+c)$ $\forall abc \in B$
  6) Complement $a + a' = 1$, $a \cdot a' = 0$ $\forall a \in B \exists a' \in B$
- Duality
  - Everything true for expression ⟺ true for dual
  - get dual of expression by replacing: Swap 0 and 1, Swap • and + (leave literals as they are)
- Laws

1)	$X + 0 = X$	$X \cdot 1 = X$
2)	$X + 1 = 1$	$X \cdot 0 = 0$
3)	$X + X = X$	$X \cdot X = X$
4)	$\overline{\overline{X}} = X$	—
5)	$X + \overline{X} = 1$	$X \cdot \overline{X} = 0$
6)	$X + Y = Y + X$	$X \cdot Y = Y \cdot X$
7)	$X + (Y + Z) = X + Y + Z$	$X \cdot (Y \cdot Z) = X \cdot Y \cdot Z$
8)	$X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$	$X + (Y \cdot Z) = (X+Y) \cdot (X+Z)$
9)	$(X \cdot Y) + (X \cdot \overline{Y}) = X$	$(X+Y) \cdot (X+\overline{Y}) = X$
10)	$X + (X \cdot Y) = X$	$X \cdot (X + Y) = X$
11)	$(X + \overline{Y}) \cdot Y = X \cdot Y$	$(X \cdot \overline{Y}) + Y = X + Y$
12)	$\overline{(X + Y + \dots)} = \overline{X} \cdot \overline{Y} \cdot \dots$	$\overline{(X \cdot Y \cdot \dots)} = \overline{X} + \overline{Y} + \dots$

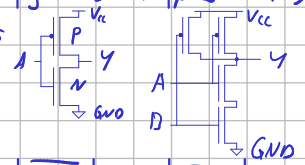
- Sum-of-Product Form: Take sum of minterms of lines evaluating to 1 of Truth Table ($A=0$ ⟺ take $\overline{A}$)
- Product-of-Sum Form: Take product of maxterms of lines evaluating to 0 of Truth Table ($A=1$ ⟺ take $\overline{A}$)

⇒ POS of F is SOP of $\overline{F}$

- Bubble-Pushing:
  - from gate output to gate input ⇒ flip AND/OR of gate
  - from circuit output to circuit input ⇒ add bubbles to all gates' input
  - from all gates input to gates output ⇒ add bubble to circuit out.
- K-Map • Help to simplify boolean equations
  - Works with max 4 variables
  - List vars in grey code at top and on left • Fill out table • Circle 1s sol. box side is power of 2, box is as large as possible, wrap around corners
    - create prime implicants of each box (ignore vars which change, negate if 0) • take sum of these minterms
  - Prevent glitches: • add circle between adjacent circles

Transistors and Gates

- n Mos | G, D, S | $g=0 \Rightarrow$ OFF | $g=1 \Rightarrow$ ON | pass 0 well ⇒ pull down
- p Mos | G, D, S | $g=1 \Rightarrow$ ON | $g=0 \Rightarrow$ OFF | pass 1 well ⇒ pull up
- c Mos = nMos + pMos
  ↳ not gate



- NAND
- Logic Gates

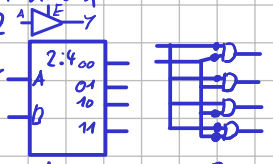
Bool	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$	$\overline{A}$
Veril.	&	~&	|	~|	^	~^	~

A	B	And	Nand	Or	Nor	Xor	XNor	Not(A)
0	0	0	1	0	1	0	1	1
0	1	0	1	1	0	1	0	
1	0	0	1	1	0	1	0	0
1	1	1	0	1	0	0	1	

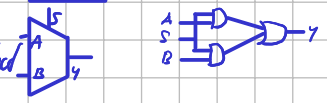
- Logical complete one: {AND, OR, Not}, {NAND}, {NOR}

Combinational Logic

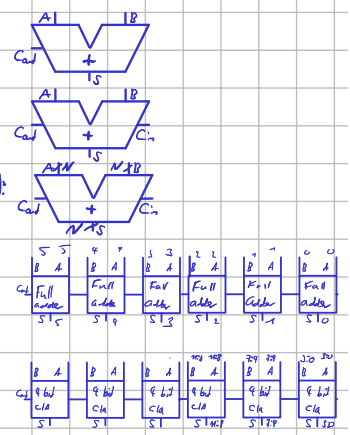
- Axioms
  1) Every circuit element is again CL
  2) Every circuit node is either: – input to circuit or – connected to output of other node
  3) Circuit contains no cycles
- Illegal Value X: Wire driven high and low (in TT means dont care)
- Floating Value Z: Wire not driven
- Tri-State Buffer: outputs 0 to Z



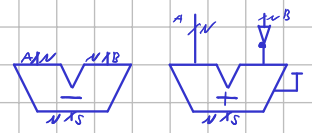
- Decoder: pattern interpreter
  - n inputs, 2^n outputs
  - one-hot output
- Multiplexer: choose input
  - n inputs, 1 output, $\lg_2 N$ select
  - connect specific inputs to outp.



- Adder
  - Half-Adder: Add two bits and output sum and carry.
  - Full-Adder: as Half-Adder with additional carry-in.
  - Carry Propagate-Adder (CPA): N bit adder
    - Ripple-Carry-Adder: chain N Fulladder – long delay
    - Carry-lookahead-Adder: detect carries early

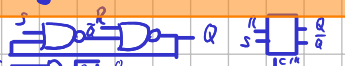
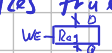
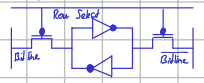
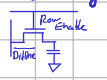


- Subtraction:
  1) Invert bits of 2nd number
  2) add 1 to 2nd number
  3) add 1st and 2nd number

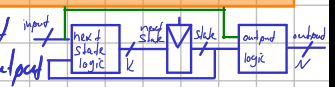


- Comparator: • equality comparator: $A == B$ • magnitude comparator: $A < B$, $A \leq B$ etc.

Sequential Logic

- R-S-Latch: 
- D-Latch:
- D-Flip-Flop:
- Enable Flip-Flop:
  - EN true ⇒ like D-FF, EN false ⇒ ignores CLK and D
- Resetable Flip-Flop:
  - Res false ⇒ like D-FF, Res true ⇒ D set to 0
  - synchronously: reset on raising edge only
  - asynchronously: reset when Res true
- Register: multiple F-F / Latches 
  - Address: $\lg_2$ [# locations] • addressability: # bits stored in each location • address space: set of unique locations
- Static RAM: 
- Dynamic RAM: 
- Latch / F-F: + Very fast + parallel access − very expensive (10+)
- SRAM: + Relatively fast − 1 data word at time − expensive (6+)
- RRAM: + cheap − slower − 1 data word at time − reading destroys content − special manufacture process

FSM

- Moore: current state ⇒ output 
- Mealy: current state + input ⇒ output
- Binary State Enc: + Fewer bits to represent (state Reg.)
- One-Hot State Enc: + Simpler next-state & output logic
- Output Encoding: + Simple output logic
- Transition Table: • Current State (Encoded) • Inputs (with don't care)
  - Next State (encoded) ⇒ Read of next-state logic equation

Verilog

- Structural: gate level description, connection of modules
- Functional: functional description, logic & math operators
- Numbers: N'BXX N: # bits, B: base (b,h,d,o), (XX) value in base
- Non-blocking ⇐: assignment made at end of block
- Blocking =: assignment made immediately
- Signals within always block must be reg, all other wire
- Code in always block is combinational, iff all outputs are always assigned: • all RHS variables are in the sensitive list (including conditional vars) • all LHS variables are in all conditions assigned
- Verification: • Functional • Timing
- initial: run once at start • # n: wait n ns

Timing

- Sequential
  - Contamination delay t_{cd}: delay till Y starts changing
  - Propagation delay t_{pd}: delay till Y finishes changing
  - longest Path: t_{pd}
  - shortest Path: t_{cd}
- Combinational
  - Setup time t_{setup}: time before clockedge when input must stable
  - Hold time t_{hold}: time after clockedge where input must stable
  - Aperture time t_a: time around clockedge where input must stable
    - $t_a = t_{setup} + t_{hold}$
  - Contamination delay clock-to-q t_{ccq}: time after clockedge where output starts to change
  - Propagation delay clock-to-q t_{pcq}: time after clockedge where output is stable
  - Cycle time Tc: Speed of clock
  - $T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{screw}$
  - Sequencing overhead $t_{pcq} + t_{setup} + t_{screw}$: wasted time
  - Hold time constrain: $t_{hold} + t_{screw} \leq t_{ccq} + t_{cd}$

Von Neumann Model

- Memory:
  - stores Data & Programs
  - addressability: way how data is addressed
  - address space: # addresses
  - Big Endian: MSB on the left, LSB on the right
  - Little Endian: MSB on the right, LSB on the left
  - Two registers are required: − Memory Address Register (MAR) − Memory Data Register (MDR)
  - Read: 1) Load MAR with the address 2) Data is placed in the MDR
  - Write: 1) Load MAR with the address and MDR with the data 2) activate Write Enable
- Processing Unit: • processes data of a specific word-length. E.g. ALU • Contains registers for temp data of word length
- Input / Output (Peripherals)
- Controll unit:
  - Step-by-Step execution of a program
  - Instruction Register (IR): keep track of instruction which is currently being executed
  - Program Counter (CP) / Instruction Pointer (IP): Register containing the address of the next instruction

Dataflow Model

- Instruction is fetched and executed in data flow order
- when operands are ready, there is no IP
- ordering is specified by dependencies
- Program consists of dataflow nodes, which is fetched and executed when all inputs are ready

Instructions

- **Instruction**: Word in the language of a computer
  - transform AS (Arch. State) to AS'
- **Instruction Set Architecture (ISA)**: specifies
  - Memory Organisation • Register Set • Instruction set
- ISA can be implemented in SW (and then compiled) or in HW
- **Instruction Cycle**: (not all six phases are always used)

IF • **Fetch**: load instruction from memory into IR
  1) load MAR with content of PC and increment PC
  2) Place instruction in MDR
  3) Load IR with the contents of MDR
  - common to every instruction type

ID • **Decode**: identify instruction with a 4-16 Decoder by looking at opcode at IR[15:12]
  - **Evaluate**: Compute address of memory location which is needed to execute the instruction
    - adds register and immediate

MEM • **Fetch Operands**: fetch source operands required to process the current instruction
  - Either from Memory or Register

EX/MA • **Execute**: Eg. in ALU

WB • **Store Result**: write result to Memory or register

- **Addressing Modes**: where are the operands located
  - Immediate / literal: directly in instruction (LC-3, Mips) • Register (LC-3, Mips) • PC-relative: add immediate to PC (for Memory access only) (LC-3, MIPS) • Indirect: Mem[Mem[PC+ offset]] (LC-3) • Base + Offset (LC-3, Mips) • Pseudo-Direct (Mips)
- Program is executed in sequence, except when changed using control instructions
- Instruction Processing Unit consists of:
  - Datapath: HW elements that deal and transform data
  - Control logic: HW elements that determine control signals
- **Analysis**: Execution Time of
  - 1 instruction: CPI x (1/f)
  - 1 program: N x ØCPI x (1/f)
  - # instructions /s: f / CPI
- **R-Type**: Opcode 6 | RS 5 | RT 5 | RD 5 | Shift 5 | Func 5
- **I-Type**: Opcode 6 | RS 5 | RT 5 | Immediate 16
- **J-Type**: Opcode 6 | Pseudo-Address 26

Microarchitecture

- Implementation of ISA, not exposed to SW
- Architecture = ISA + Microarchitecture
- Defines how the underlaying implementation executes instructions
- **Architectural State**: • visible by programmer • memory • registers • PC

Single-Cycle CPU

- Process instruction in a single cycle: CPI = 1
- State update at end of instruction
- Slowest instruction determines clock cycle T_c
- higher HW cost (2 Memory, 3 ALU)

Multi-Cycle CPU

- Single instruction is split into multiple stages
- CPI different for each instruction
- State update can be during instruction
- Architectural State update at end of instruction
+ Slowest stage determines clock cycle T_c
- Control signals for the next cycle can be created in the current cycle (more parallelism)
- Need to store the intermediate results in each cycle
- HW overhead + overhead for fetch/Store
- Some HW is idle during certain stages

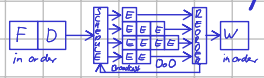
Pipelining

- use idle HW ⇒ increase instruction processing throughput
- Slowest step decides the throughput
- Split Single-cycle machine in different stages and add pipeline registers ⇒ extra overhead
- feedback to earlier stages (write to mem) are challenging
- control signal are also pipelined to stay in sync
- **Stall**: condition where pipeline stops moving
  - hold back values in pipeline registers, and insert nop
  - Flush stalled stage to prevent wrong data
  - all following pipelines are stalled too.
- Instructions in two pipeline stages require the same HW. Solve: 1) duplicate HW 2) Stall one pipeline
- **Forwarding**:
  - from last E to first E of dep. instruction
  - from M to D of dep. instruction
- **Dependancies (Control and data)** dictate instruction ordering
  - **Flow dep.** (Read after Write) } dep. on value
    - Solve by: - Detect and wait till value available - Detect and forward data - Detect and Eliminate on SW level - Predict the needed value and verify - Other ...
  - **Output dep.** (Write after Write) } • dep. on name, not value
  - **Anti dep.** (Write after Read) } • easier to handle
    - Solve: write to reg in one cycle and in prog. order
- **Detect Dependancies**:
  - **Score board**: each register has a valid bit, which is set when an instruction is writing. In decode stage it is checked if all registers are writable ⇒ 1) Yes - No deps 2) No - Stall
    + Simple - Stall for all types of dependancies
  - **Combinational dep. check logic**: logic which checks if a later stage is writing to a register of the instruction currently beeing decoded ⇒ 1) yes - stall pipeline 2) No - no flow dep. + no stall on anti and output dep. - complex - complexer pipeline ⇒ comp. logic
- **Control dep.** What is the PC in the next cycle?
  1) PC is non-control-flow instruction ⇒ PC' next seq instr
  2) PC is control-flow instruction ⇒ • stall • always take
    - additional early-detect logic • predict based on history

- **Precise Exceptions/Interrupts**: Architectural state must stay consistent: • previous instructions should retire (finish and update architectural state) • no later instruction should retire
  ⇒ Problematic when pipeline stages take diff. amount of time
- **Reorder Buffer (ROB)**: complete instructions out-of-order but reorder before making visible to arch. state
  1) decoded instruction saves space in ROB 2) when finished, it writes value to reserved space 3) when reserved space sequentially oldest, submit to arch. state
  • Flow dependency ⇒ forward value from ROB
- **ROB** contains all info about decoded but not retired instruct.
  • implemented as circular queue (content addressable Mem.)
  • if value in ROB, take it from there, else from RF
  • Improved version: access RF and check if value is valid: 1) No – RF has address to value in ROB (⇒ easy access via address) 2) Yes – take it and store address of ROB entry in register. Set valid to no
  ⇒ eliminates output and anti dependencies
  + solve precise exceptions + solve false deps. – ROB access slow

Out-of-Order Execution

- Move dep. instructions out of the way of indep.
  1) Link consumer of value to producer via register renaming.
  2) Buffer instructions till ready to be executed in reservation station. 3) Check readiness of instructions' values 4) Dispatch instruction to functional unit (FU) when all values ready
- **Tomasulo's Algorithm**



  1) Res. Station full ⇒ stall
  2) Check reg. alias table if operand reg. are valid: • valid – copy value to res. table and set as valid • invalid – copy tag from reg. alias table to res. table and set as invalid
  3) Set dest. reg. in reg. alias table to invalid and ad the res. table id as the tag (renaming)
  4) When execution finished, broadcast tag and value to reg. alias table and res. table, update valid and value

- **Precise exception**: Have a seperate RF which is visible to the arch. state and updated in prog. order
  • On exception, Flush all newer instructions in ROB and res. table and copy ARF to RAT
- **Instruction window**: decoded but not retired instructions
- Memory access is problematic: • when to schedule? • are there load/store deps.? • solve: – conservative: stall until address computed – agressive: assume there is no dep. – Intelligent: predict dep.
  • Maintain LQ and SQ to see if there is a dependant load/store instruction in the instruction window
  ⇒ expensive structure, complicated search required

Superscalar Execution

- Process N instructions per cycle (in-order or OoO)
  + High IPC – more HW – complex dep. check

Handle Control Dependances

- Keep pipeline full with correct sequence of instructions
- * **Stall**: wait until dep. resolved
- * **Branch Prediction**: Predict PC'
- **Branch target buffer BTB** stores target address for cond. branch
  ⇒ given PC, we can check if instruction is a cond. branch and if it was already executed: get address
- Predicted fetch stage requires: 1) whether fetched instruction is a branch (BTB has address ⇒ it's a branch) 2) Branch direction 3) Branch target address (BTB)
- **Compile time**: – not dynamic
  • always not taken/always taken: + simple – low accuracy
  • Backward taken, forward not + simple – mid accuracy
  • Profiling (Compiler gives a hint via hint bit) + more accurate – requires hint bit – depends on profile
  • Program-based (analyze opcode, loop etc.) + no profiling – may not be representative – compiler analysis
  • Programmer-based (hint from programmer) + no profiling/analysis + programmer knows code

- **Run time**: + based on history – more complex
  • Last-time predictor (single bit indicating last decision)
  • Two-bit counter (4 states) + better prediction – more HW
  • global Branch Correlation (store T/NT history of all branches in global History Reg. (GHR). Use GHR to index Pattern History Table (PHT) which records the outcome seen, using last-time-predictor for this certain GHR pattern.
  ⇒ 2 level history (GHR + PHT)
  • gshare predictor: GHR hashed with branch PC + more info + better utilisation of 2-bit counter – latency
  • Local Branch History. Use GHR for single branch to detect end of loop and PHT to capture last decision
  • Hybrid (combine multiple predictors) + better accuracy + reduce warmup time – requires meta controller – latency
  • Perceptron. learn correlation between branch history register bits and branch outcome + better accuracy – hard to implement – learn only linear functions
  • Multiple history length: look back different far
  • Confidence Estimation. How sure is prediction
- * **Branch Delay slot**: Skip branch and start executing instructions which must certainly be executed + keep pipeline full – hard to fill delay slot – ties ISA to HW
- * **Predicate Execution** (Eliminate Control-flow instructions): Combine predicate operations ⇒ single branch instruction instead of multiple (control dep. to data dep.)
  + Fewer branches (⇒ fewer mispredicts/stalls) – possible unnecessary work (1. predicate falls, no need to check follow.)
- * **Multipath Execution** (fetch both paths if possible):
  + May improve performance + No ISA change – Nested branches ⇒ exponential work – Each branch requires seperate context – Wasted work if paths merge

Very Long Instruction Word

- Compiler packs indep. instr. into bundle which is fetched and executed concurrently
- Multiple functional units
- Lock step: 1 VLIW op. stalls ⇒ all VLIW ops. stall

+ simple HW thanks to: • no dynamic scheduling • no HW dep. check • no instruction distribution to funct. units

− Compiler needs to find N indep. instructions − Recompilation required when N changes − Lock step

Systolic Arrays

- Array of PE instead of single PE (or even multidimens.)
- allow "on-the-fly" computation
- PE can have own data (e.g. weights), data mem. and instr. mem.

+ multiple use of fetched data (better use of mem. bandwidth)

+ High concurrency + Regular design + Specialized

− specialized − bad at exploiting irregular parallelism

Decoupled Access Execution (DAE)

- Separate access and execute streams
- Communicate via ISA-visible queues
- Compiler generates these streams and synchronizes them via control flow instructions

+ one stream can run ahead + Limited OoO execution without wakeup/select complexity − Compiler support

− Synchronisation between A and E required − multiple instruction streams

Fine-Grained Multithreading

- For data & control dependances
- Switch thread context (PC + regs) each cycle
- No instruction is fetched from same context till fetched instruction resolves

+ no HW to handle Data and control deps. + Improve throughput

− extra HW for multiple contexts + thread select logic

− single thread performance − dep. check between threads

SIMD

- Single instruction operates on multiple data

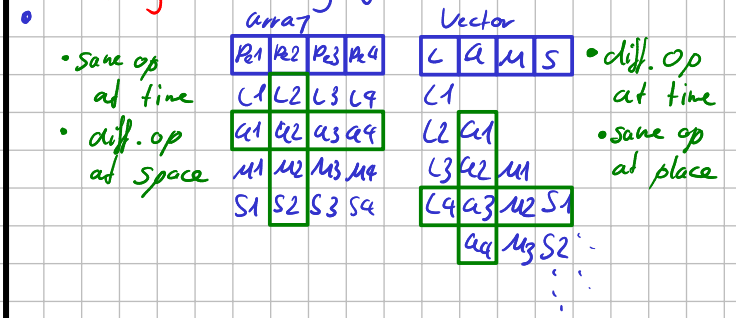
* Array processor: Instr. operates on multiple data elements at the same time using different spaces
  - have multiple PE

* Vector processor: Instr. operates on multiple data elements in consecutive time steps using the same space

- have specialised units (e.g. LD, ADD, ST etc.)
- operate on vectors (of different length)

+ no deps. between vectors + each instruction generates lot of work + regular mem. access pattern − works only if parallelism is regular − Mem. bandwidth can be bottleneck

- Vector data reg. holds N M-bit values
- Vector control reg.: VLEN (Vec. length ≤ N), VSTR (Vec. stride; dist. of data entries in mem), VMASK (indicate on which elem. to op. on)
- Deep pipeline (⇒ faster clock)
- Memory Bank: Mem. divided in blocks which are accessed indep.
  - Share addr. and data bus but have sep. MDR & MAR
  - Sustain N data access if each go do separate bank
  - 1 bank access takes 1 cycle
  - Bank Conflict: access data from same bank. solve by:
    - more banks − better data layout − better addr.−bank mapping
- Loop vectorizable if each iteration is independent
- Vector chaining: Forward data from func. unit to func unit
  - Only 1 mem access at the time (Multiple Mem. ports solve this problem)
- Special ISA support required
- Chaining: Forwarding from Vec. PE to next one



GPU

- Pipline like SIMD pipeline, programming using threads
- Thread executes same instruction but on multiple data
- Warp: Dynamic set of thread executing the same instruction
  - # warps = ⌈# threads / warp size⌉
- SIMT: Multiple instruction streams of scalar instructions
  - Can treat thread separably
  - Can group threads into warps flexibly
- Instruction inside warp can be multithreaded
- Threads devided into blocks, blocks into warps
- Threads of a single warp can take different branches
  - Branch divergence: if taken branches lead to different execution paths
  - Can be redistributed into new warps
- Utilisation: # busy lines / # active threads (for 1 program)

Memory

- Memory Array: N address bits, M data bits: • 2^N rows M columns • Depth: # rows / words • Width: # columns / size word • Array size: $2^N \times M$ (ideally $2^n \times 2^m$, $n \approx m$)
- Interleaving/Banking: divide large array in smaller blocks which can be accessed independently
- Temporal locality: Same data will be accessed again in near future
  - Store accessed data in cache
- Spacial locality: nearby data will be accessed in near future
  - prefetch and store data in cache
- Cache lvl analysis: i^{th} lvl has
  - intrinsic access time t_i and perceived access time T_i, $T_i > t_i$
  - hit rate h_i with acc. time t_i | miss rate m_i with acc. time $t_i + T_{i+1}$
  - $T_i = h_i t_i + m_i \cdot (t_i + T_{i+1}) = t_i + m_i T_{i+1}$
- Direct Mapped Cache: each block maps to loc. in cache determined by index bits of the address
  - Access: 1. index into tag & data store using index bits of addr. 2) check valid bit in tag store 3) compare tag bits in addr. with tag bits of tag store
  - reduce conflict misses by splitting each table into 2, 4, ...
    - + Higher hit rate − longer access time − HW cost
- Cache replacement policy

- Update next lvl cache when modified:
  - Write-through: up on modification + simple + consistency between levels – more bandwidth intensitive
  - write-back: update next lvl cache when block is evicted + combine multiple writes to same block – bit required to mark the block as modified
- allocate cache block on each write miss?
  - Yes: + combine writes + simpler (write miss similar to read miss) – transfer of whole cache block
  - No: + requires less space
- Subblocked Caches: valid & dirty bits for subblocks + do not have to transfer full blocks – complex
- Separate / unified Instruction and data caches
  - Unified: + dynamic sharing of space – data / instructions trash eachother – physical placement
- Cache size: total data capacity
  - Working Set: data referenced in given time interval
- Block size: data associated with address tag
- Associativity: # blocks represented by the same index
- 3 Typ misses: • compulsory • capacity • conflict
- Resource Sharing: allow multiple contexts to use certain HW + utilisation + efficiency + reduce communication latency – performance
- Capacity (C): # data bytes a cache stores
- Block size (b): # bytes of data brought to cache at once
- # Blocks ($B = C/b$)
- Deg. of associativity (N): # blocks in set
- # sets ($S = B/N$): each mem. addr. maps to exactly one set
- Cache Coherence: How to update cache when shared
  - Snoopy bus: bus on which changes are broadcasted
  - Directory based: mediator gives permissions to requests
  - SW based
- Virtual Memory: memory as seen by programmer
- Page size: amount of mem. transferred from HDD to DRAM at once
- Page table: lookup table virt. mem. addr. to physical mem. addr.
  - Indexed by Virt. Page Number bits of Virt. mem. addr.
- translation increases # mem. acces to >2
- Translation Lookup table (TLB) caches accesses
  - Very high hitrate
- Memory Management Unit (MMU): HW components (Page table registers, TLBs, fetch logic etc.)
- clock page replacement algo: • circular list of physical frames • pointer points to last-examined frame • replace pointed frame is invalid, else set invalid and continue
- Each process has own pagetable