

# Combinational Logic

## Basic Building Blocks

### Transistors

Metal (conductor), Oxide (insulator), Semiconductor  
n-type transistor: connection source ↔ drain ↔ high voltage  
p-type transistor: connection source ↔ drain ↔ low voltage

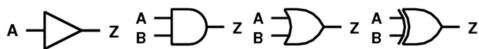


good at pulling down (electrons don't flow well)      good at pulling up (electrons flow well)

p/n-type: opposite      high voltage: 0.3V-3V  
modern system: n-&p-type → C(omplementary)MOS

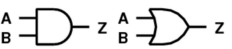
### Logic Gates

#### Buffer



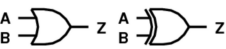
A	Z
0	0
1	1

#### AND



A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

#### OR



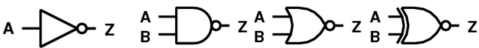
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

#### XOR



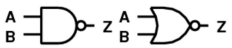
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

#### Inverter



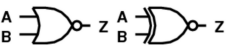
A	Z
0	1
1	0

#### NAND



A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

#### NOR



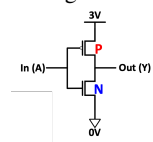
A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

#### XNOR

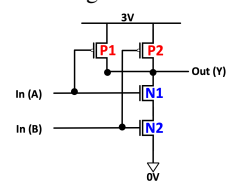


A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

### NOT-gate/inverter



### NAND-gate



AND-gate/ $Y = A \cdot B = AB$ : NAND + inverter (not impossible: p pull up and n pull down - otherwise 4) — XOR:  $1 \leftrightarrow$  odd input 1

### Latency & Power Consumption

Transistors in serial slower than transistors in parallel (pseudo nMOS to alleviate latency)  
dynamic power: charge capacitors on signal change ( $C \cdot V^2 \cdot f$ , capacitance, voltage, frequency) - static power: loss due to drain ( $(V \cdot I_{leakage})$ ) — energy = power · time

## Boolean Algebra

functional specification of logic gates

NOT:  $\bar{\bar{A}} = A$ , AND:  $A \cdot B$ , OR:  $A + B$  - ring axioms hold  
duality: switching 0/1  $\leftrightarrow$  → still valid  
idempotent law:  $X + X = X$ ,  $X \cdot X = X$ , involution law:  $\overline{(\bar{X})} = X$ , law of complementarity:  $x + \bar{x} = 1$ ,  $x \cdot \bar{x} = 0$ , commutative law:  $x + y = y + x$ ,  $x \cdot y = y \cdot x$   
DeMorgan's law:  $\overline{(X + Y + \dots)} = \bar{X} \bar{Y} \dots / \overline{(XY \dots)} = \bar{X} + \bar{Y} + \dots$

boolean algebra to simplify: better implementation  
complement, literal: input or its complement, implicant: product of literals, minterm: product that includes all inputs, maxterm: sum that includes all inputs

### standardize function representations

truth table of function unique, equations not  
S(um)O(f)P(roduts)/D(isjunct)N(ormal)F(orm): sum of minterms with value 1 - may only specify rows:  $m_3 + m_4 + \dots = \sum m(3, 4, \dots)$   
P(rodut)(O)fS(ums)/C(onjunctive)N(ormal)F(orm): product of maxterms with value 0 - may only specify rows:  $M_0 \cdot M_1 \cdot \dots = \prod M(0, 1, \dots)$   
conversion: x rows  $\sum m(1, 2, \dots) = \prod M(\dots, x-1, x)$   
inversion:  $\sum m(1, \dots) = \sum m(\dots, x) = \prod M(1, \dots)$

### logic simplification/minimization

trial & error - goal: reduce number of gates and inputs, number of transistors  
use Unity Theorem:  $F = A\bar{B} + AB = A(\bar{B} + B) = A$   
subsets of ON set w single varying variable → eliminate ("don't care", denoted X)

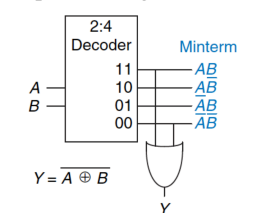
### logic completeness

Anything implemented with {AND, OR, NOT}  
NAND and NOR also logically complete

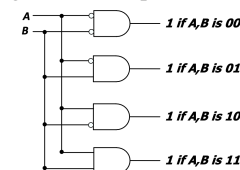
## Combinational Logic

### decoder

input pattern detector,  $n \rightarrow 2^n$  signals, input combination ↔ one output  
implement logic functions

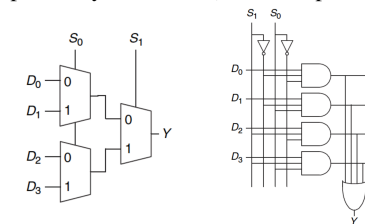


### gate-level implementation



### multiplexer

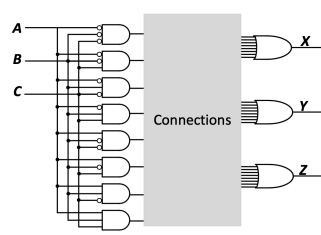
select one input as output based on  $\log_2 N$  controls (output always connected), as lookup tables (LUT) for logic



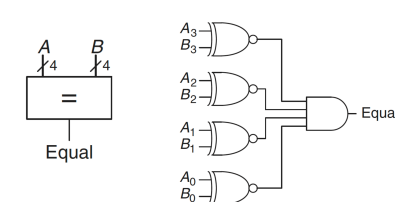
### full adder

add two bits with carry,  $S_i = A \oplus B \oplus C$ ,  $carry_{i+1} = a_i b_i + a_i \cdot carry_i + b \cdot carry_i$

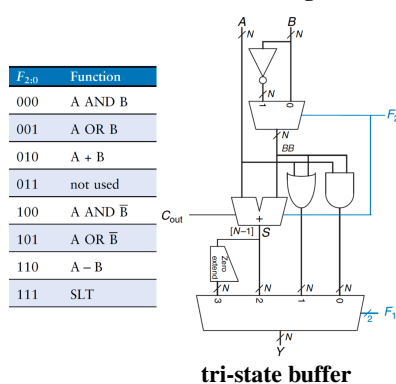
more efficient/specialized carry lookahead adder exists  
**programmable logic array**



### comparator



### arithmetic logic unit



### tri-state buffer

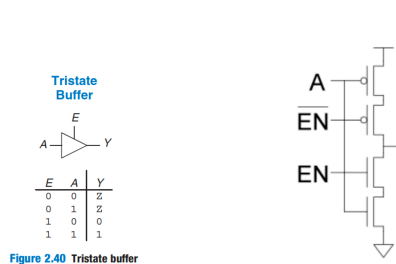
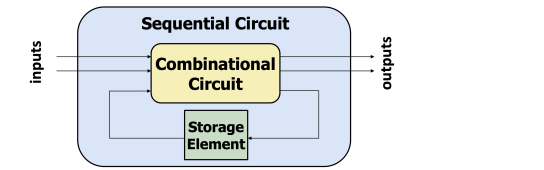


Figure 2.40 Tri-state buffer

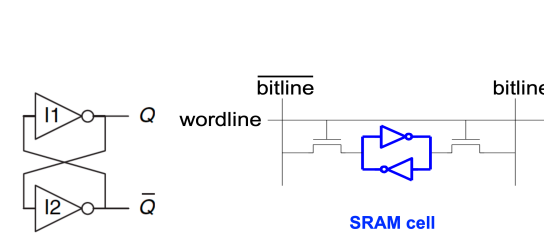
## Sequential Logic Design



memory hierarchy: Latches/Flip-Flops, SRAM, DRAM  
Latches: level-triggered (capture data during entire high of write-enable), Flip-Flop: edge-triggered, captures only on rising (clock) edge

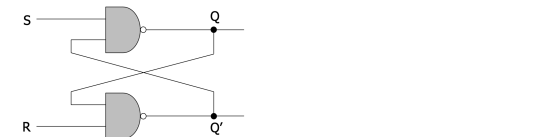
### storing information

**Cross-coupled inverter, needs control mechanism**

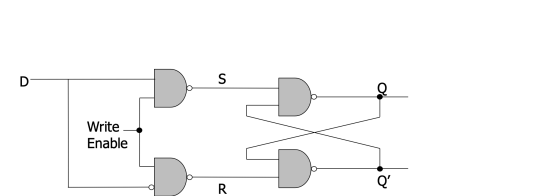


### R-S Latch

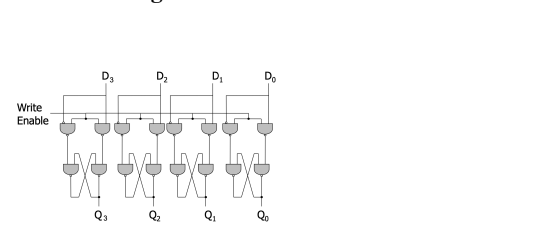
$Q$  data &  $S, R$  control -  $S = 1 = R$  quiescent (idle),  $S = 0, R = 1$  set,  $S = 1, R = 0$  reset,  $R = 0 = S$  invalid (oscillation, eventually settle unpredictably)



### Gated D Latch

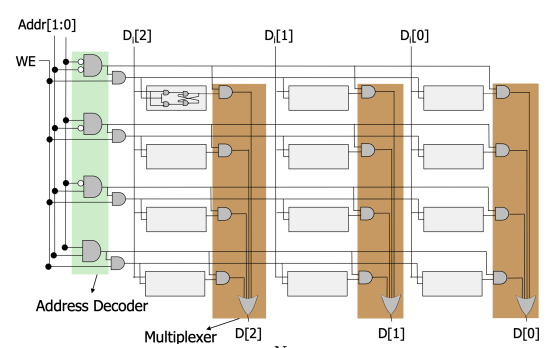


### Register with Gated D Latches



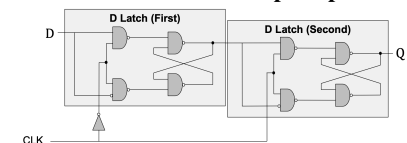
### Memory with Gated D Latch Registers

unique locations in memory, read/written with unique address,  $x$  locations have  $\log_2 x$ -bit addresses, addressability: bits in each location, address space, space of unique locations



memory boolean logic:  $2^N$  address space,  $M$ -bit addressability is  $N \rightarrow M$  function

### D Flip-Flop



### Finite State Machines State & Clock

state: snapshot, state diagram: models state machine (state change: transitions)

asynchronous: transitions at any time, synchronous: transitions when clock signals, synchronous easier, asynchronous higher efficiency

clock: signal alternating 1/0, clock cycle: rising-rising, system change only at rising edge



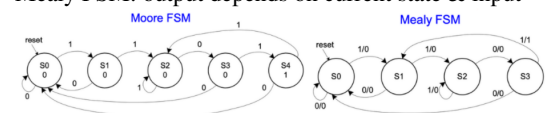
### FSM

discrete-time model, state diagram: all states & transitions — requirements: finite inputs, outputs, states, explicit state transitions, output values — three parts: next state logic (combinational), state register (sequential), output logic (combinational)

### types

Moore FSM: output depends on current state

Mealy FSM: output depends on current state & input



Mealy: less states, larger output, lower reaction time

### Designing an FSM

always reset: asynchronous (indep. from clock), synchronous (sampled at clock edge only)

1. in-/outputs, 2. states, 3. transitions

next state: transition table, encoding, equations, impl.

output: output table, encoding, equations, impl.

drawing: state register, next state l., output l.

### State Encoding

full encoding: min flip-flop, max bits:  $\log_2$  states

1-hot encoding: max flip-flop, min n.s.l., 1 bit  $\leftrightarrow$  1 states

output encoding: moore only, min output l.

## Hardware Description Languages

requirements: specify, simulate, synthesize — common: Verilog(here), VHDL

### key principles

hierarchical design: predefined gates, complexity through instantiation — minimize complexity

top-down: first top-level, identify lower-level

bottom-up: build with whats available  $\rightarrow$  complex

synthesis: HDL  $\rightarrow$  netlists — optimal solution not guaranteed, write 'nice'/easy synthesizable HDL code

simulation: verification & testing

### Verilog

module is main building block, (name, ports, functionality) — not programming language: think in hardware case-sensitive, names start w/o numbers

**expressing numbers:** 'N'Bxx', N:number of bits, B:base (b,h,d,o), xx:number (can use \_ for readability, X (invalid), Z (floating))

### Module definition

```
module example(a,b,c)
  input a;
  input [31:0] b;
  output c;
endmodule
```

```
module example( input a,
  input [31:0] b,
  output c );
  here comes functional spec.
endmodule
```

### Functional specification

use 'assign y = ...;' for continuous assignments (wires) — continuous as RHS change  $\rightarrow$  immediate LHS change

### general

```
assign y = longbus[12:5]; // slicing
assign y = {a[0],a[0]}; // concatenation
assign y = {4{a[0]}}; // duplication
```

### structural(gate-level) instantiation (only)

```
module local_name ( .port(local1), .port2
  (local2) ); // instance
module local_name ( local1, local2 ); //
  shortform, order!!!
  not g1 ( out, in );
  and g2 ( out, in1, in2 );
  or g4 ( out, in1, in2 );
```

### behavioral logical&mathematical operators, high-level

```
assign y = ~a; // not
assign y = a & b; // and
assign y = a | b; // or
```

operators (in hierarchy) bitwise: ~not, \*mult, /div, %mod, +add, -sub, <<shift(l), >>shift(r), <, <=, >, >=, ==, !=, &and, ^xor, |or, ?:ternary  
ternary operator: 'assign y = s ? d1 : d0;' (d1 if s true, else d0) — nesting allowed  
reduction operators: 'assign y = &a;' instead of 'a[a[0] & a[1] & a[2] & ...;'

### Abstraction levels

structural (low-level): tedious, better optimizations

behavioral (high-level): easier, harder optimizations

### Parameters

```
module MyModule #(parameter W_A = 8,
  parameter W_B = 16) (input [W_A-1:0]
  a, output [W_B-1:0] y);
endmodule
// instantiation:
MyModule #(16, 32) U2 (.a(ia), .y(iy));
```

### Sequential Circuits

for memory, FSMs, ... we need clock, always-block, ...

store values: declare 'reg' instead of (implicit) 'wire'

then do **procedural assignments**, RHS change  $\rightarrow$  LHS

only when called — blocking: = immediately — non-blocking: <= at the end of block all together — either

blocking or non-blocking in block

**always:** change of value in sensitivity list  $\rightarrow$  executed, if (\*) then all RHS signals — should not two always

blocks with same sens.list or same assigned signals — can annotate signal als 'posedge'/'negedge' to only trigger

on change to +/- — can do combinational logic: effective \* sens.list, all LHS always updated

**if...else, case:** only in procedural blocks

```
always @ (sensitivity list)
  begin // only if multiple statements
    p = a & b; // either this
    p <= a & b; // OR this
  end // only if multiple statements
always @ (*) begin
  case (data)
    4'b0010: b = 2'b00;
    4'b0011: b = 2'b11;
    default: b = 2'b10;
  endcase
  if (a == 4'b0001)
    // statement
  else begin
    // statements
  end
end
```

blocking: complex combinational, non-blocking: seq.

### Timing

add delay with #time, linked to statement, separate within procedural block (always, initial)

```
'timescale 1ns/1ps
...
assign #9 z2 = a;
```

## Timing and Verification

clock to fast  $\rightarrow$  failure — transistors: time to switch (electron speed limit, capacitance) — env. impacted

### Timing in Combinational Circuits

contamination delay  $t_{cd}$ : until output starts changing — propagation delay  $t_{pd}$ : until output settles

compute from shortest and longest/critical path

**glitches:** input change  $\rightarrow$  two output transitions — Karnaugh maps: moving between p.implicants: indication — neglect in synchronous circuits: fixing expensive

### Timing in Sequential Circuits (registers)

clock: period/cycle time  $T_c$

### Input

setup time  $t_{setup}$ :  $\Delta$ time stable before sampling

hold time  $t_{hold}$ :  $\Delta$ time stable after sampling

aperture time  $t_a := t_{setup} + t_{hold}$

### Output

contamination delay  $t_{ccq}$ : earliest  $Q$  change after clock

propagation delay  $t_{pcq}$ :  $\Delta$ time after clock that  $Q$  settles

### Constraints

setup:  $T_c > t_{pcq} + t_{pd} + t_{setup}$  = seq. overhead+work

— low  $t_{pd}$ : high speed, high  $t_{pd}$ : lower efficiency

hold:  $t_{ccq} + t_{cd} > t_{hold} \rightarrow$  minimum  $t_{cd}$

### Clock Skew

$:= \Delta$ time two clock edges — minimize! —  $t_{skew}$ : worst

setup:  $T_c > t_{pcq} + t_{pd} + t_{setup} + t_{skew} = t_{pcq} + t_{pd} + t_{setup, effective}$

— hold:  $t_{cd} + t_{ccq} > t_{hold} + t_{skew} = t_{hold, effective}$

$\Rightarrow$  critical path design (minimize!), balanced design

### Verification & Testing

difficult today: complex — do on-line testing (while in use), important:

S(ilent)D(ata)C(orruption)/C(orrup)E(xecution)E(rrors)

pre-silicon testing: formal, HDL sim., circuit sim.(here)

high-level > low-level: high-level funct. verif.&low-level timing, power

### Functional Verification & Testing

operates correctly — D(evice)U(nder)T(est) — input:

test patterns, output compared to reference values — simple: man./man., self-checking: man./autom., autom.:./

(in-&output generation / error checking) — self-checking: testvectors (in-/output in file) possible

brute-forcing NEVER feasible

```
module testbench3();
  reg clk, reset; // internal
  reg a, b, c, yexpected; // testvector
  wire y; // output of circuit
  reg [31:0] vectornum, errors; // track
  reg [3:0] testvectors[10000:0]; // array
  test dut (.a(a), .b(b), .c(c), .y(y) );
  always begin // no sensitivity list
    clk = 1; #5; clk = 0; #5; // 10ns
    period
  end
  initial begin
    $readmemb("example.tv", testvectors);
    vectornum = 0; errors = 0;
    reset = 1; #27; reset = 0;
  end
  always @(posedge clk) begin
    {a, b, c, yexpected} = testvectors[
      vectornum];
  end
  always @(negedge clk) begin
    if (~reset) begin
      if (y != yexpected) begin
        $display("Error: inputs = %b", {a
```

```
, b, c));
$display(" outputs = %b (%b exp)
", y, yexpected);
errors = errors + 1;
end
vectornum = vectornum + 1;
if (testvectors[vectornum] === 4'bx
) begin
$display("%d tests completed with
%d errors", vectornum,
errors);
$finish;
end end end module
```

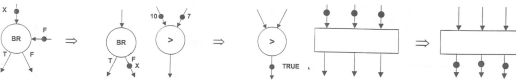
**Timing Verification & Testing**  
check: timing constraints recognized — high-level sim.  
with #time — circuit-level after synth.

**Von Neumann Model**  
V.N.M: fundamental computing model: mem-  
ory(data,instructions),processing unit,input, out-  
put,control unit — instructions in linear memory array,  
sequential instruction processing

**Von Neumann Model**  
register file,memory,IP exposed to programmer  
**Memory**  
grouped: bytes (8 bits), words (8,16,32,...) — address  
space — addressability: byte(MIPS,LC-3b), word(LC-  
3) — capacity=address space-addressability — en-  
dianess: big(most-significant addr. last),little(least-  
sig.addr.last)  
read/write with M(emory)D(ata)R(egister)&M(A)ddressR

**Processing Unit**  
comprises many F(unctional)U(nits) — ALUs/FUs pro-  
cess words — register file close to compute (word size)  
**Control unit**  
conducts step-by-step execution of instructions with  
I(nstruction)R(egister),P(rogram)C(ounter)/I(nstruction)  
P(ointer) — IP increment by multiple bytes/word/...

**Dataflow**  
other execution models exist — instructions fetched/ex-  
ecuted in dataflow order/when operands ready — no pro-  
gram counter: notifies receivers, instruction executed  
when all received (tokens) — inherently more parallel



tradeoffs (from von Neumann difference) — partly inte-  
grated internally (usually not exposed to programmer)

**Instruction Set Architectures (ISA)**  
ISA: instruction set (opcodes, data types, addressing  
modes, length), memory (address space, addressability),  
register file (size, count) ⇔ everything programmer  
**instructions** opcode + operands — three instruc-  
tion types below — opcodes: different amounts ⇔  
hw/sw complexity — operative: do computation, data-  
movement: load/store memory, control: change control

flow  
**data types** one/several data types: 2's complement inte-  
gers, unsigned integers, floating point numbers  
**semantic gap** hw/sw translator for ISA/semantic change

**Addressing Modes**  
specify location of operands: LC-3 all those, MIPS not  
indirect but pseudo-direct  
**PC-relative:** 'memory[ PC(incremented) +  
signext(offset)]' — limit: target close to PC  
**indirect:** 'memory[ memory[ PC(incremented) +  
signext(offset)]]' — can access everywhere  
**base+offset:** 'memory[ baseregister + signext(offset)]'  
— can access everywhere  
**immediate:** hardcoded value  
**register:** NOT memory, just specify register address

**Instructions MIPS**  
r-type, i-type (immediate), j-type (jump) (, f-type (float))  
— r: additional func bits for operation (opcode 0)  
**operative instructions**  
r-type: op,src,src,dst,shift,operation

0	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

i-type: op,src/dst(base),dst/src(register),immediate/literal

**data movement instructions**  
i-type: 'sw \$s3, 8(\$s0)' (base+offset), 'sw rt, imm(rs)'  
— 'lw ...'  
lui (load immediate): 'lui \$s0, 255'/'lui rt, imm', 'ori  
\$s1, \$s0, 42'/'ori rt, rs, imm'  
MIPS byte-addressable: word x with offset 4x (32bit)  
**control instructions**  
i-type: BRANCHES 'beq rs, rt, imm' (rs==rt → PC =  
PC(incremented) + signext(offset) \* 4) — JUMP j-type

2	target
6 bits	26 bits

'j target' (PC = PC(incremented)[31:28] — targer \* 4),  
'jr register', 'jal ...' (function calls)  
**LC-3**

**operative instructions**  
ADD, AND (no subtraction)

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

OP	DR	SR1	0	00	SR2
4 bits	3 bits	3 bits			3 bits

NOT: only consider OP, DR, SR1, everything else 1 —  
'NOT R3, R5'  
immediate: 'ADD R1, R4, #-2'

**data movement instructions**  
word-addressability-LD, LDR, LDI, LEA, ST, STR, STI  
LD/ST: 'memory[PC(incremented) + signext(offset)]' -  
'LD R2, 0x...'

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

OP	DR/SR	PCoffset9
4 bits	3 bits	9 bits

LDI (indirect): 'memory[memory[PC(incremented) +  
signext(offset)]]'  
LDR: 'LDR R1, R2, 0x1D', 'memory[BaseR + offset]'

OP	DR	BaseR	offset6
6	1	2	0x1D

LEA: as LD but 'PC(incremented) + signext(offset)'  
(w/o memory)  
**control instructions** BRANCH  
update to register: N(egative), Z(ero), P(ositive) condi-  
tion codes set — BRn, BRz, BRp, BRzp, ... (if one true)

0000	n	z	p	PCoffset9
4 bits				9 bits

JUMP 'JMP BaseR'

1100	000	BaseR	000000
4 bits	3 bits		

TRAP: OS call (8bit vec)  
**Functions**  
calling: jal (MIPS), jsr (LC-3) — returning: jr (MIPS),  
ret (LC-3) — arguments: \$a0-\$a3 (MIPS) (stack \$sp if  
more) — return value \$v0 (MIPS) — only \$t regs not  
guaranteed to be preserved (MIPS)

**Microarchitecture/uarch**  
uarch: hw internal (must not confirm ISA) — uarch can  
change w/o ISA — uarch follows design points

**Basics**  
build system (hw(there) or hw with software translation)  
— instruction proces.: A(rchitectural)S(tate)→AS' —  
define finite state machine det. AS' — state trans.  
atomic for program.  
components: datapath (deal w./transform data signals,  
FU/ALU, reg./mem — hw for flow of data), control  
logic (hw for control signals, what datapath hw should  
do) — control signals: combinational vs. microcoded

**Performance Analysis**  
C(ycles)P(er)I(nstruction) · C(lock)C(ycle)T(ime) =  
instruction ex.time — #instructions·average CPI·clock  
cycle time=program ex.time — CPI: constant(single),  
varying/shorter(multi)  
single: 'longest' instr./critical path det. CCT — deter-  
mine by cons. all instr.types (control logic matters too)

**Single-cycle uarch**  
T instruction/clock cycle — slowest instruction bot. neck  
many disadvantages: slowest is bot. neck, resource du-  
plication, complex implement., diff.2optimize

**Multi-cycle uarch**  
T instruction/multiple clock cycle — only exe. req. IPCs  
— CCT not det. by 'slowest' instr. (CCT indep. in-  
str.proc.time) — crit. path design, bread&butter, bal-  
anced design — allows for waiting (for memory, ...) adds  
seq. reg. overhead  
AS→step1→step2→...→AS' — design FSM: (mult.)

states/IPC stage — states def. by control signals  
control logic implemented as FSM

**Pipelining**  
introduce concurrency to multi-cycle — fully separate  
stages: diff. instr. in diff. stages — easy to start with  
single cycle as resource layout more suitable  
divide IPC into distinct stages — req. suff. res./stage —  
diff. intr./stage  
steps seq. dependent, consid. dep. between instr., con-  
sid. imbalance between stages  
have lead-in/-out, full utilization middle

**Pipelines**  
ideal: no inter-stage overhead, uniform partitioning  
uniform partitioning: S=seq. delay/stage,  
time/stage= $\frac{T}{k} + S$ , R cost/register, cost=G+Rk

**Pipelining instruction processing**  
simple 5 IPC stage stages — imbalanced stages: lower  
speedup — all must proceed all stages (wasted time)  
pipeline registers at level between stages (latch on trans.)  
control signals: same as single-cycle, must latch too/  
keep in phase with data — option 1: generate once,  
buffer until consumed — option 2: buffer instruction  
parts, generate on demand  
problems: external (wasted stages)/internal (unbalanced  
staged) fragmentation, stalls (dependencies)

**Stalls & Dependencies**  
causes: contention, dependencies, long-latencies  
stalling: disable PC&IF, keep instr.in stages (add we  
signal to pipeline registers f,d,e), synchronous reset ex.  
pipeline register, bubbles

**Data Dependencies**  
flow (true), anti (write after read), output (write after  
write)  
anti/output: false, always same stage  
flow: detect&wait, detect&bypass, detect&eliminate,  
detect&move, predict, sth.else — 1st half write, 2nd read  
interlocking (detection, correctness guarantee): sw  
(static scheduling)/hw, (bubbles, find indep.instr.) — sw  
difficult: unknowns→profiling  
1.hw scoreboarding: valid bit/register, stall when invalid  
2.hw comb.dep.detection logic  
**detect&wait** stop up-stream stages, drain down-stream  
**detect&bypass** forward data as soon as available, allow  
dependent instr. to execute until data needed, add bypass  
paths — still stalling when no bypass possible (latency)  
**detect&eliminate** compiler, reorder indep.instr., nops  
**fine-grained multithreading** instructions in pipeline  
from different indep. threads — no: add. logic, waste  
cycles — disadv.: low single-thread, hw, contention,  
need threads — employed in GPUs/accelerators

**Control Dependencies**  
next instruction depends on control flow instr., next ad-  
dress maybe not directly available  
option branch prediction: speculative execution, flush  
pipeline on wrong prediction — flushing: misprediction  
penalty  
option early branch resolution: check resolution in de-  
code, additional logic/hw, less flushing/branch mispre-  
diction penalty

**Remarks**  
combination difficult (timing)



**Precise exceptions**  
 idea: indep. instr. exec. on diff. hw (division, add, ...) — older instruction finishes with error after younger, younger's writeback sequentially invalid  
 exception: program-internal problem (prog. context) — interrupt: external event (syst. context)  
 upon(excpt. immed., intrpt delayed): stop, save, handle, return — precise AS, flush younger instr., save PC/registers, redirect to handler  
 precise: when handling, AS consistent (prev. retired, later !retired)

**ensuring precise exceptions**  
 single-/multi-cycle: easy — instr. for getting exception in handler (mfc0, MIPS)  
 pipelined: difficult — all worst case latency bad — approaches: reorder buffer (here), history buffer, future register file, checkpointing  
 R(e)O(rder)B(uffer): buffer out-of-order results, retire in-order — circular buffer — valid flag, destination register id, destination register value, store address, store data, pc, valid bits for registers control bits, exceptions decode: reserve in ROB, complete: write to ROB, ROB: oldest w/o exception complete: retire, restore: flushing ROB  
 dep. instr.: bypass paths, read from ROB, read from reg.file — ROB requires expensive search/content addressability, avoid w indirection — reg.file stores valid flag, ref.to ROB (access reg.file), maybe R(egister)A(lias)T(able)

## Out-of-Order Execution

out-of-order dispatch, out-of-order completion, in-order retirement — internal dataflow execution  
 non-ready instructions in reservation stations, operands of reservation station entries ready: dispatch  
 complete parallel execution: long-latency tolerance  
 hw requirements: consumer-producer link (renaming/-tagging), reservation stations, track readiness/broadcast operands, dispatch (schedule), complete, retire

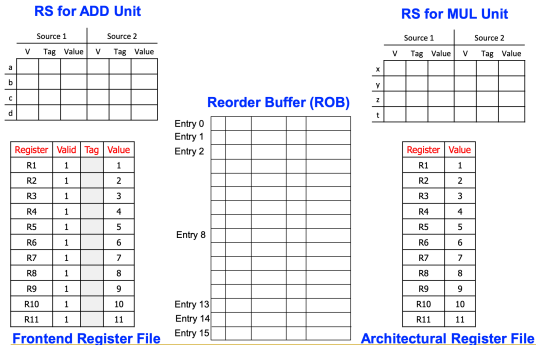
**Tomasulo's Algorithm**  
 initial OoO execution at IBM by T. — no precise exceptions/ROB initially  
 decode: add w renamed operands to reservation station, add own output to RAT (stall if no res.station available) — in res.station: monitor C(ommon)D(ata)B(us) for source operands/tags — ready: dispatch to FU — FU completion: broadcast tag & value on CDB, register file also listens, free name tag  
 req. busses from each FU to all registers/table entries

**How it works**  
 de-/allocate reservation stations, global/FU reservation station, ROB/res.stations/reg.file, broadcast simultaneously/serial/...

**Avoiding Value Replication**  
 values replicated in tables: waste — introduce P(hysical)R(egister)F(ile) (centralized value storage) — references to PRF from ROB, ...  
 works because tags/references smaller than values

**Remarks**  
 internal dataflow graph limited to instruction window (determined by number of reservation stations) — exploits irregular parallelism¶

**Precise Exceptions**



**Loads & Stores**  
 static/dynamic dependencies, small/large state space, (in)visible to others  
 cor1: renaming useless during execute, cor2: dep. determined after (partial) exec., cor3: ready, dep. unknown if others with unknown addr.  
**scheduling** approaches — conservative: wait until certainty — aggressive: assume independence — intelligent — latter 2 req. recovery  
 find dependency — opt1: all prev. committed — opt2: buffer of pending stores, check for match  
**reordering** must buffer stores for seq. semantics — dependency: access buffer on load  
 L(oad)Q(ueue) & S(tore)Q(ueue), both in one queue possible — complex store-to-load forwarding logic to search for value

## Dataflow & Superscalar Execution

**Dataflow** — may directly mapped to hw — diff. on today's hw — diff. with today's concepts(excpt.,debug,...)  
**Superscalar Execution**  
 OoO-exec. horizontal parallelism — sup.scalar: vertical parallelism — also parallel fetch, decode, retire:  $N$ -wide sup.scalar  $\Leftrightarrow N/\text{cycle}$   
 req.: dep.checking between  $N$  instr., duplicate datapaths  
 dependencies: detect & wait, detect & bypass, detect & eliminate, detect & move, speculative, FGM

## Branch Prediction

always control dependencies — 15-25% control flow —  $N$  branch resolution latency:  $N \cdot W$  (sup.scalar) waste — want prediction during fetch (so that ready for next) types (direction, possibilities, when): conditional (unknown, 2, execution), unconditional (always, 1, decode), call ("), return (always, many, execution), indirect ("")  
 goal: keep pipeline correct full: stall, prediction, branch delay slot, FGM, predicate execution, multipath execution

**Static (compile-time) Branch Prediction**  
 always (not) taken, BTfN, profile, program analysis — req. ISA support (metadata/-bits) — reduce: penalty, mispredictions  
**Always (not) taken** taken:  $\approx 60 - 70\%$  accuracy — sw/hw: increase probability (code layout) — penalty  $\leftarrow$  bubble/flush count  
**BTfN**: backward taken (loops), forward not taken

**Profile**: likely direction from profile run, accuracy depends on representitiveness of profile run  
**Program Analysis**: use heuristics (take loop, not take fp comparisons, not take leq), misprediction 20 % 1993  
**Programmer**: language pragmas, programmer hints

**Dynamic (runtime) Branch Prediction**  
**BTB**: Branch Target Buffer, store branch target, access with PC later, content hints whether taken  
**Last Time Predictor**: predict branch as last time, bit/branch, store in B(ranch)H(istory)T(able), corresponds to small FSM, loop:  $\frac{N-2}{N}$ , good long, bad short  
**N-bit Counter**:  $N$  instead of 1 bit for counting, less volatile/add hysteresis, 2: strongly/weakly (not) taken, 80-90% accuracy on average  
**Two-level Prediction**: use P(attern)H(istory)T(able)/PAT, index with history, get  $N$ -bit prediction counter  
*global*: G(lobal)H(istory)R(egister), 3-bit GHR reasonable, improved with GHRxorPC for indexing  
*local*: L(ocal)HR/BHR for each branch, one PHT  
*general*: BHR: G(lobal)/S(et of branches)/P(branch), PHT: G,S,P, PHT counters: A(adaptive), S(tatic)  
**loop branch** counts loop iterations, compares to limit  
**perceptron** simple ML, single neuron: weight\*x+bias > 0, 1 perceptron/branch, weight $\leftrightarrow$ bit on GHR

IF  $\text{sign}(y_{out}) \neq t$  or  $|y_{out}| \leq 0$   
 FOR  $i=0..n$ :  
 $w_i = w_i + tx_i$

**hybrid history-length predictor**  
**Hybrid** multiple predictors, select best for branch  
**Branch Confidence Estimation**  
 useful for hybrid/choosing among predictors, simple: track past (in)correct at branches — index table with PCxorGHR & reduction on past correctness

**Branch Delay**  
 delayed branch execution,  $N$  instructions after branch always executed — perfect branch resolution, diff. to find instructions for delay slots (esp. cond. branches) with squashing: not execute delay slots when not taken

## V(ery)L(ong)I(nstruction)W(word) Architectures

similar to superscalar, but compiler already packs in instr. bundles — true VLIW: instr. in bundle indep. (req. hw understanding), sometimes even indep. between seq. — lockstep execution  
 bundle has structure: maybe instr. type linked to position  
**Philosophy** similar to RISC: compiler most hard work, hw simple (faster, energy efficient) — low power  $\neq$  low energy  
**tradeoffs**: simple hw, compiler indep instr., recompilation for new uarch req., lockstep: stalling of indep. instr.  
**trace scheduling (indep. of VLIW mostly)** - basic block: single entry/exit point, after branch reordering for optimization — super block: combine frequent basic blocks to one single-entry/multiple-exit blocks, enables aggressive optimizations for superblock/common case  
**ISA translation** through hw/sw, to get better tradeoff

## Systolic Array Architectures

in ASICs, accelerates certain tasks/exploits regular parallelism, highly concurrent, balanced compute & I/O  
 use regular array of P(rocessing)E(lements), collective compute, max. compute/data, PEs may be structured many-dimensional, PEs local memory & execute  
 needs indep. data, carefully input data, buffer output flexibility possible through weights in PEs, may be changed on the fly, PEs may have own data/instr. memory  
**Convolutions** as in CNN, CV, filtering, ... — can be implemented in systolic arrays (via matrix multiply), Google TPU, Cerebras WSE  
**examples/use** warp computer: 10 programmable processors in linear array, programmable, extends general purpose computer

## Decoupled Access and Execute

mitigate Tomasulo's complexity — separate memory access & execute — separate memory/execute instruction streams, execute in separate parts, communication via FIFO queues — req. synchronization (branches, ...) can run ahead of each other/some OoO without reservation stations etc., compiler support, branch req. synchronization (loop unrolling to reduce)  
 compiler instr. splitting vs. dynamic  
 not done on ISA-level usually, but hw-internal approx. for latency tolerance

## S(single)I(nstruction)M(ultiple)D(data) Architectuers

exploits regular parallelism, same operation done on multiple data — SIS(ingle)D, M(ultiple)ISD, MIMD — MISD generalization of systolic arrays  
 SIMD: same instruction to multiple processing engines (with diff. data) — I(nstruction)L(evel)P(arallelism) — amortization of fetch overhead — data parallel programming model  
**Vector Register** holds  $N$   $M$ -bit values, entire vector/register — also control registers: V(ector)LEN(gh) (upper bound  $M$ ), VSTR(ide), VMASK (set with vector test instr.): limits proc. to certain vec. elements  
 operate on vector instead of scalars here — must: load-/store vectors, operate on diff. VLENs, vec. elements may be stored apart in memory (def. by VSTR) as row-/column in matrix  
**Vector Processors** instr. on mult. data in consecutive time steps — pipelined execution on hw, operation/element in consecutive cycles — very deep pipelines possible (indep., !control flow, easy (pre)loading) — VFUs can also be deeply pipelined  
**Array Processors** instr. on mult. data in parallel (multiple PEs)  
**combination** array & vector combined in practice — pipeline with multiple resource instances — may partition vector register: partition linked to specific VFUs (via lanes) — may overlap execution of multiple vector instructions

**Memory (vector)**  
 memory (bandwidth) easily bottleneck — load/store req. mult. memory accesses — get high throughput(1) with banking **memory banking** divided, banks accessed independently, share address/data busses (reduce pins),

start/complete 1 bank access/cycle —  $N$  concurrent access to  $N$  banks possible (relatively prime best) — sustain throughput when “#banks  $\geq$  #bankLatency” — bank conflicts: more banks, more ports/bank, better data layout, better address mapping to banks

### Chaining (vector)

forwarding from one VFU to next VFU as soon as result available — still memory limits (banking, ports, ...)

### Remarks

**stripmining** enable vectors longer than vector register entry length — do multiple iterations (diff. VLEN last)  
**scatter/gather** vectors might not be stored in strided fashion, index vector defines references/indirection, can scatter/gather with index vector + base address  
**conditionals in loops** can use VMASK if not want to execute operation on specific vector elements, encode condition in bitmasks/VMASK (predicate execution) — hw simple: execute all, turn of write — hw density-time: scan VMASK, only execute necessary  
**automatic code vectorization** compilers may unroll loops to vector operations etc.  
**modern ISAs** modern systems not full ISAs, have extensions (packed arithmetics)

## GPU Architectures

works like SIMD underneath (execution model)/MIMD/SIMT(head), programmed using multiple threads (programming model)/SP(rogram)MD: thread for each independent execution, threads run same program  
threads executing same instruction grouped into warps (wavefront) by hw automatically, executed via SIMD at once (thread executes on SIMD line), each tread uses id to identify its task  
SIMT advtgs.: threads separate (don't need GPU), flexible warp grouping (supports branches), long latency tolerance  
each warp (thread even) indep.: no interlocking, no dep. checking, fine-grain multithread warps → easy to keep pipeline full  
A GPU comprises many S(treaming)M(ultiprocessors), SMs comprise many SP(rocessors) themselves. Each SM may execute multiple warps in a pipelined fashion. One instruction is executed as a SIMD instruction so that each thread is executed on a SP in the end. SIMD line ↔ SP+ — blocks: group of threads, can cooperate/share data — grid: comprises blocks, execute all the same kernel  
dynamic warp regrouping (branch), with many threads efficient, can't move threads between lanes (separate registers, FUs)  
underutilization: branch divergence, long latency ops. — two-level warp scheduling of smaller warp groups (one latency blocks ready, others can continue) — large warps, break into sub-warps  
**terminology** generic, NVIDIA, AMD — vector length, warp size, wavefront size — pipelined FU/scalar pipeline, streaming processor/CUDA core, — SIMD function unit/SIMD pipeline, group of  $N$  streaming processors, Vector ALU — GPU core, streaming multiprocessor, compute unit

## Memory

SRAM: on-chip/in-core, DRAM: 'typical', storage

### Importance

most area is memory — memory is main bottleneck — started 3D-stacking

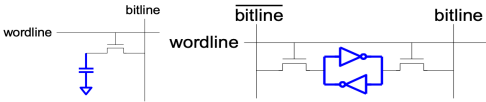
**wordload** ML, AI, genomics, databases, datacenter → near/in-memory computing or FPGAs  
**performance** memory stalls  $\approx$  70% stalls  
**energy**  
**reliability/security**

### Fundamentals

**virtual/physical** system maps virtual memory addresses to physical memory — programmer sees 'infinite' memory/translation invisible — last PART — physical here

**ideal memory** 0 latency,  $\infty$  capacity, 0 cost,  $\infty$  bandwidth, 0 energy

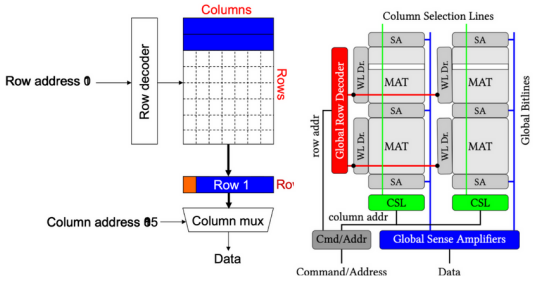
**memory arrays** flip-flops/latches: fast, expensive (bit ↔ 10s transistors) — S(tatic)RAM: relatively fast, volatile, expensive (6+) — D(ynamic)RAM: slower, refresh, volatile, special manufacturing, cheap (1trans.+1capacitor) — storage: much slower, non-volatile, very cheap  
memory: 2-dim. array of bit cells — 1 bit or byte or .../cell — rows(=:depth) and columns(=:width)



reading: sensing amplifiers — detect signal change — SRAM good: complementary — reset bitlines beforehand  
in practice: multiple words/row → squared memory (useful for caching)  
dram: capacitor leaks, refreshing — sram: used on-chip  
simple scaling difficult: latencies

### DRAM Subsystem

channel → rank → bank → subarrays → mats **channel** each channel has a memory controller, one interface to compute unit — everything to one DIMM  
**DIMM** dual in-line memory module, physical module/organizational structure, rank on front and back — everything to one rank  
**rank** has 8 chips on it (ex) — distributed to chips  
**chip** has 8 banks (ex) — everything to one bank  
**bank** has 32 rows with 8 bytes (ex) — accessed in independently in parallel, each bank is a memory array — some buffers used/a type of cache — row address (part of address) to get row, sense amplifiers put in row buffer, column address (part of address) to get data, write back (read is destructive) — in practice: subarrays with local row buffer in array, row decoder forwards part of address to local decoder — MAT: usually refers to one row in a subarray, but may divide row into two subrows to not have to active the entire row, depending on selected column — read dominated by wordline, bitline drives



### DRAM Operation

decode row address, drive word-lines, selected bit-cells drive bitlines, differential sensing, decode column address, select subset of row, send to output, (rewrite as destructive), precharge bit-lines  
cache blocks (blocks of memory) distributed across chips, accessing takes  $\frac{\text{amount}}{\text{rankAccess}}$ , dist. across chips

### Emerging technologies

P(hase)C(hange)M(emory), resistive memory, state (crystalline/high reflexivity/low resistivity vs amorphous/low reflexivity/high resistivity) changed using heat — higher density, lower cost, non-volatile, no refresh, slower, endurance problem, higher-energy  
SST-MRAM (magnet based, polarity)  
Memristors: atomic structure  
Flash, SSD common — doubtful for past two decades (flash durability problems)

### Hierarchy & Caches

hierarchy: storage (off-chip), DRAM (off- or on-chip), SRAM (usually on-chip) as L3 cache, L2 cache, L1 cache (in-core), register file — bigger → slower, faster → expensive — cheaper with time  
deal with opposing goals: levels of hierarchy/caches — due to locality (past predicts future) appears fast & large — temporal locality: likely to use same data reference again, spatial locality: likely to operate on related/close data  
manual cache management: programmer/compiler must do all, still for embedded, GPUs, accelerators (typically scratchpad today, addition) — automatic: hardware manages across all levels, transparent to programmer, hides uarch details — today: usually only register file manual  
extension: remote memory with remote nodes on low-latency network  
sharing/separating data/instruction caches — utilization vs. quality of service — l1 split, l2+ shared  
**Latency** intrinsic access time  $t_i :=$  get local data/learn miss — perceived access time  $T_i := t_i +$  getting data from higher level if miss — hit rate  $h_i := \frac{\#hits}{\#hits + \#misses}$  — miss rate  $m_i := \frac{\#misses}{\#hits + \#misses}$  —  $T_i = t_i + m_i T_{i+1}$  — A(verage)M(emory)A(ccess)T(ime): usually lower better  
optimize —  $m_i$  low: increase  $C_i$  → increase  $t_i$ , better cache management — lower  $T_{i+1}$ : make faster (expensive), intermediate hierarchy level

### Cache Design

cache: memorizes used/produced data — cache comprises cache blocks — cache block stores block of data from higher level — block: subsequent set of addresses with metadata (source, access patterns) — memory split

into fix blocks, cache can store blocks  
questions: placement, replacement, granularity of management, write policy, instruction/data  
**storing cache blocks**

have tag (valid, tag) and data stores (data), indexed by index bits  
+associativity ↔ +hit rates, -speed, costly hw — return for +associativity → +hit rates diminishes  
**direct-mapped** block only in one cache block — cache block determined by index bits of address — index cache with index bits, compare tag (remaining address part), miss or index data store — contention (conflict miss) if two blocks with same index bits in cache  
**set-associative** block in certain set of cache blocks, less conflict misses —  $n$ -way/degree  $n$ :  $n$  cache blocks/set — more expensive hw: compare all cache blocks in set — bigger tag, smaller index  
**full-associativity** any block in any cache block — one set — very expensive: many comparators

### Cache Management

cache full & new block: conflict miss — must evict before add — define priority, who to evict — insertion (new priorities), promotion (change priorities), replacement (who evict?) — eviction/replacement policy  
**L(east)R(ecently)U(sed)** evict lru, difficulty: track order — minimize processing (+memory):  $\lceil \log_2 n \rceil$  bits/block (position) — minimize memory (+latency):  $\lceil \log_2(n!) \rceil$  bits (combination ↔ order) — true LRU expensive: not in modern systems  
**locality approx.:** not M(ost)R(ecently)U(sed) — hierarchical LRU (expl. MRU groups, LRU in groups) — victim-victimNext  
**Random** sometimes better than LRU/MRU — set trashing if working set > associativity → Random — set sampling: compare performance repeatedly, choose then  
**Belady's OPT** replace block furthest referenced — provably minimizes miss rate — v.diff. to impl. — min. miss rate  $\neq$  min exec. time (diff. latencies, overlapping)

### Multilevel Management

higher level: +size, +associativity, +latency, tag/date serially — access higher levels in parallel (speculative) vs. serially on miss — faster vs. energy — latter: different policies as different access patterns  
data in all/only one cache?, bypass when loading?, evicted put where? — inclusive: block in inner also in outer (coherency), exclusive, inner never in outer (space utilization), non-inclusive: may/may not

### Writes

write through: write to higher level caches as soon as data is modified (+simple, +coherency, -bandwidth) — write back: write back when evicted (combine, +bandwidth, -dirty bit) — dirty bit (in tag store): indicates modified  
allocate cache block for write on miss? — not if overwritten — *subblocks* with indivd. dirty bits: load not entirely overwritten

### Performance

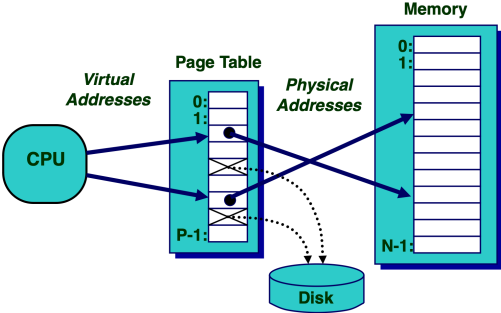
cache size (working set) for temporal locality — block size for spatial locality - for large blocks: critical word first (forward immed.), subblocks (load indep. supply faster) — associativity (-misses, +latency, +cost) — replacement policy  
miss/hit rates — compulsory miss: first block reference → +block size, prefetching — capacity miss: all misses



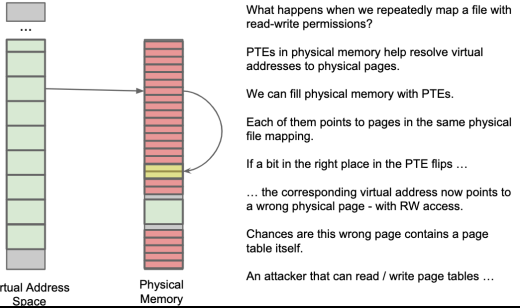
in fully-associative, opt. replacement, same capacity → software management of working set — conflict miss: all other misses (different organization could avoid) → +cache size, +associativity (rand. indexing, software eviction hints, victim cache) *improve performance* — -miss rate: associativity, better policies, sw opt. — -miss latency/cost: multi-lvl. caches, critical word first, subblocking, better replacement policy, non-blocking cache, sw opt. — -hit latency/cost  
software opt.: restructuring data access patterns (*loop interchange*, loop fusion, array merging), restructuring data layout (data structure separation/merging, exmpl. outsource rarely-accessed fields of objects as reference), data reuse/blocking/tiling (expl. matrix multiply)

**Advanced Caching**  
**M(emory)L(evel)P(arallelism)** memory access may have long latency — impact determined by: latency overlapped?, latency tolerance?, evicting longer-to-refetch block?  
MLP: prefer parallel misses to better utilize stall — min. miss rate, thus, not opt. — prefer elimination of isolated miss, higher-latency misses  
**Caches in Multi-Core Systems** shared vs. private, QoS & predictability, allocation/thread, shared/limited bandwidth/space  
*shared caches* +utilization, -comm. latencies, +shared mem.program. model/+coherency, +contention, -single thread performance, -QoS, -consistency, +capacity, -speed — cache management to mitigate  
**Cache Coherence** private caches, shared memory comm., cache incoherence → cache coherency protocol broadcast based: invalidate/update elsewhere on write directory-based cache coherence: directory stores what caches have, caches consult directory — exmpl.: store  $P + 1$  bits/cache block, 1/P indicates contains, 1 exclusive bit — read: set process bit, invalidate exclusivity before (directory + exclusive thread) — write: invalidate all other, exclusive to this  
**Prefetching**  
reduce miss rate & latency (maybe eliminate misses) — speculative, works with predictable patterns — nos misprediction penalty  
**What address** address prediction algorithm — prefetch accuracy:=  $\frac{\text{used prefetches}}{\text{sent prefetches}}$  — patterns, compiler/programmer input — stride, repeating variable stride, rep. pat-

tern,  
**When** (early, late, on-time) affects timeliness — earlier or aggressive → timely  
**Where to place** caches (today), separate buffer — role in replacement policy (equal to demand fetched, ...)  
**Where prefetcher** all memory-lvls. (today)  
**How** sw prefetching: prefetch instructions — hw pref.: finds access patterns, prefetches automatically — execution based: 'thread'/instr. flow from program to prefetch (sw or hw) — (cooperative, hybrid)  
**Prefetchers**  
**Stride** record stride, stable  $N$ : predict next  $M$  — per-instr. or per-memory-region — need last address referenced, stride, (confidence) — stream prefetching (hw for  $N = 1$ )  
**complex prefetchers** multiple regular strides (multi-stride detection common today), linked data structure traversal, indirect array access, multiple data structures concurrently — bootstrapping: table various histories  
**self-optimizing memory (Pythia)** get state (memory request features), prefetch, get reward (optimize) — features: PC, branch PC, last 3 PCs, cacheline address, physical page number ,  $\Delta$ two cacheline addresses — prefetch as offset — reward: usefulness (accuracy, timeliness), system-level (memory bandwidth, cache pollution, energy) — outperforms current solutions in always  
**hybrid hw**  
**multi-core** prefetch shared data (avoid coherency misses), important as shared/limited res. — throttle prefetching: conflicts, contention (bus, RAM)  
**execution based** pre-execute program part, initiation loads of data — separate core, FGMT, same thread context/during cache misses — word for branches too  
*runahead execution* ideally OoO w large instr. window (expensive) — upon oldest instr. long-latency: runahead mode: speculative pre-execute (without stalling), generates misses/prefetches — upon long-latency return: flush, resume — no add. stalling, accuracy, instr. prefetch, train other prefetchers, limited by branch prediction, limited by MLP, -energy — best in hybrid  
**Performance**  
accuracy:=  $\frac{\text{used prefetches}}{\text{sent prefetches}}$  — coverage:=  $\frac{\text{prefetched misses}}{\text{all misses}}$  —  
timeliness:=  $\frac{\text{on-time prefetches}}{\text{used prefetches}}$  — bandwidth consumption (best during idle) — cache pollution correlated

**Virtual Memory**  
multiple programs without interference, authorization, unbound capacity — difficulties only physical: limited size, multiple programs, ISA  $\nleftrightarrow$  physical memory, programmer burden, data relocation, sharing data processes individual illusion or large address space (in-direction to physical address space) — hw & system cooperatively manage mapping — part of ISA — benefit: program uarch indep.  
**Basics**  
physical memory as (fully-associative) cache for disk (block-page, block offset-page offset, miss-page fault, index-virtual apge number) — virtual/linear addresses  $\xrightarrow{\text{translation}}$  physical/real addresses — mapping page-based (page:=block of memory, multiple KBs), frame:=page unit in memory — mapping via LUT/page-table (reference to memory or disk) managed by OS/hw, first reference arbitrary to memory — loading from disk/evicting from memory necessary by virtual memory system — virtual memory system: M(emory)M(angement)U(nit) + OS  


page table — PTE(ntry): valid bit, PPN, replacement bits, dirty bits, protection bits, disk, metadata — stored in physical memory, P(age)T(able)B(ase)R(egister)  
**Address Translation**  
virtual address = V(irtual)P(age)N(umber) + Page Offset — offset unchanged, VPN→P(hysical)PN  
valid bit set: produce page — not set: OS page fault exception handler initiates I/O controller to load via D(irect)M(emory)A(ccess)

eviction: CLOCK algorithm — circular buffer, references all PTEs with physical addresses, pointer to last examined page — evict: traverse, evict first with 0, switch all 1s to 0s — ARC another algorithm (considers access frequency)  
**Multi-Level Paging**  
one PT/thread: huge! — multiple levels instead — VPN split into identifier for each table hierarchy, continuously index into lower-level tables —  $N$ -level page table,  $N$  page table accesses — only first-level PT must be in memory  
**T(ranslation)L(ookaside)B(uffer)** Cache to reduce long-latency of multi-level accesses — maybe only one access to TLB if hit — 16-512 entries in practice, 90-99% hit rates — consider TLB as L1 cache, memory as L2 cache  
**Memory Protection**  
memory protection/isolation between processes if correct mapping by OS — shared physical address for shared memory — page table stores protection bits (access rights: read, write, execute, kernel) — access control concurrent to translation — multi-level paging: multi-level access (ISA specifies access from bit combination) — if access violates bits, Access Protection Exception — x86: protection rings 0 (kernel/OS-only)-1/2(OS service)-3(user application)  
**RowHammer** evades protection by exploiting hardware failure — induce bit-flips in neighboring rows by reading a row repeatedly — can gain kernel privileges  


What happens when we repeatedly map a file with read-write permissions?  
PTEs in physical memory help resolve virtual addresses to physical pages.  
We can fill physical memory with PTEs.  
Each of them points to pages in the same physical file mapping.  
If a bit in the right place in the PTE flips ...  
... the corresponding virtual address now points to a wrong physical page - with RW access.  
Chances are this wrong page contains a page table itself.  
An attacker that can read / write page tables ...  
**Final Considerations**  
can do hashed page tables — special treatment of virtualization and guest OSs: additional translation (guest virtual, host virtual/guest physical, host physical)