BOOLEAN LOGIC

Theorems:

$X \cdot Y + X \cdot \overline{Y} = X$ $\quad (X + Y) \cdot (X + \overline{Y}) = X$

$X + X \cdot Y = X$ $\quad X \cdot (X + Y) = X$

$(X + \overline{Y}) \cdot Y = X \cdot Y$ $\quad (X + \overline{Y}) + Y = X + Y$

De Morgan: $\overline{(X + Y + Z + \dots)} = \overline{X} \cdot \overline{Y} \cdot \overline{Z} \cdot \dots$

$\overline{(X \cdot Y \cdot Z \cdot \dots)} = \overline{X} + \overline{Y} + \overline{Z} + \dots$

Minterm: product (AND) that includes all input vars

Maxterm: sum (OR) that includes all input variables

SUM-OF-PRODUCTS: Search all TRUE combinations and add (OR) the minterm of each line

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	0

$Y = \overline{A}\,\overline{B} + A\overline{B} = \sum m(0,2)$

$Y = (A + \overline{B}) \cdot (\overline{A} + \overline{B}) = \prod M(1,3)$

PRODUCT-OF-SUMS: Search all FALSE combinations and multiply (AND) the inverted maxterms.

Karnaugh Maps:
- use fewest squares of only 1s
- squares are of size 2^x
- squares as large as possible

C \ AB	00	01	11	10
0	1	0	1	1 → $A\overline{C}$
1	1	0	0	1 → $\overline{B}$

$\Rightarrow Y = \overline{B} + A\overline{C}$

Represent 0x cafe1234

Big Endian:

0	1	2	3
ca	fe	01	23

Little Endian:

3	2	1	0
ca	fe	01	23

Logic Gates (NOT ⊸)

OR

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

AND

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

NAND

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

XNOR

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

XOR

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

NOR

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Example: use only NAND:

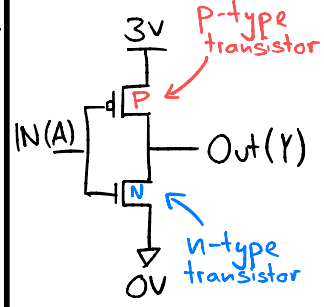
$F = \overline{(A+B \cdot C) + \overline{C}} = \overline{\overline{(A+B \cdot C) + \overline{C}}}$

$= \overline{\overline{(A+B \cdot C) \cdot C}} = \overline{\overline{\overline{(A+B \cdot C)} \cdot C}}$

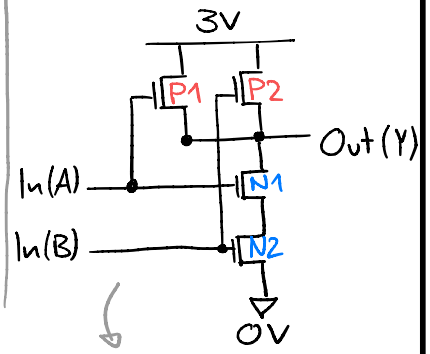
$= \overline{\overline{(\overline{A} \cdot \overline{(B \cdot C)}) \cdot C}} = \overline{\overline{(\overline{A \cdot A}) \cdot (\overline{B \cdot C})) \cdot C}}$

CMOS TRANSISTORS

Inverter



NAND gate:



Never pull up and down at the same time!

A	B	P1	P2	N1	N2	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

VERILOG

Operators:

(use this with integers)

`!` logical negation — `~` bitwise negation

↳ any non-zero val → 0, only 0 → 1

`&` bitw./red. AND — `|` bitw./red. OR

`~&` reduction NAND — `~|` reduction NOR

`^` bitw./red. XOR — `~^`/`^~` bitw./red. XNOR

`||` logical OR — `&&` logical AND

↳ 1 iff either input is true or non-zero

`<<` shift left — `>>` shift right

`==` logical equality — `!=` logical inequality

`===` case equality — `!==` case inequality

COMBINATIONAL VS. SEQUENTIAL

↳ All outputs are assigned in every possible way. Also, all inputs are in sensitivity list

PERFORMANCE ANALYSIS

- CPI: cycles per instruction
- IPC: instructions per cycle
- MHz: frequency, 10^6 cycles per second
- MIPS: million instructions / second $\frac{\text{MHz}}{\text{CPI}}$

BINARY NUMBERS

$2^1 = 2$ $\quad 2^2 = 4$ $\quad 2^3 = 8$ $\quad 2^4 = 16$

$2^5 = 32$ $\quad 2^6 = 64$ $\quad 2^7 = 128$ $\quad 2^8 = 256$

$2^9 = 512$ $\quad 2^{10} = 1024$ $\quad 2^{11} = 2048$ $\quad 2^{12} = 4096$

$2^{13} = 8192$ $\quad 2^{14} = 16384$ $\quad 2^{17} = 32'768$

$2^{18} = 65'536$ $\quad 2^{19} = 131'072$ $\quad 2^{20} = 262'144$

SI PREFIXES

kilo	10^3	2^{10}
mega	10^6	2^{20}
giga	10^9	2^{30}
tera	10^{12}	2^{40}
peta	10^{15}	2^{50}
exa	10^{18}	2^{60}
zetta	10^{21}	2^{70}
yotta	10^{24}	2^{80}

NUMBERS

Unsigned: only positive, 0 to $2^n - 1$

Signed: -2^{n-1} to $2^{n-1} - 1$, MSB is sign flag

2s Complement: invert each bit and add 1 to the LSB. MSB is sign flag

FINITE STATE MACHINES

Moore: output only dependent on current state, requires more states, synchronous output, output placed on states

Mealy: output depends on current state and current inputs, fewer states needed, output placed on transitions

Design a FSM: 1) identify inputs/outputs
2) sketch state transition diagram
3) write state transition table
4) write output table
5) select state encoding
6) write boolean equations for next state and output logic
7) sketch circuit

State **encodings**:
- binary: $\log_2$(# states) bits needed
  reduce # flip-flops to hold state
- one-hot: # states bits needed
  reduce next-state logic
- output: reduces output logic

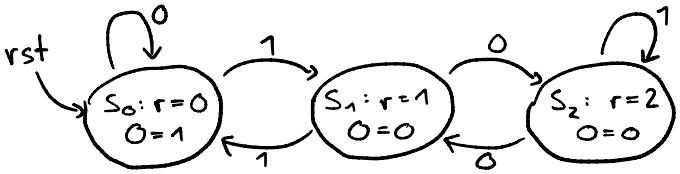
A correct **state-diagram** has:
- reset line
- only one transition per input
- no missing transitions
- no unmarked transitions
- initial state (if no reset)
- no mix of Moore-Mealy labelling
  output in bubble ↙ ↘ output on transition

Area of FSMs:
- # of FFs = (# bits for state enc.) × 2
- # of logic gates: count in the next-state logic

Find next-state & output boolean equations
1) Assign state encodings and outputs
2) Create table of the form

S[0]	S[1]	S[2]	T_A	T_B	N[0]	N[1]	N[2]	O[0]	O[1]
0	0	0	x	1	0	0	1	0	0
0	0	0	x	0	0	1	0	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	1	1	1	0	0	0	1	1	1

3) Find equations for N[x] with sum-of-products:
$N[2] = (S[0] \cdot S[1] \cdot S[2] \cdot T_A) + (\overline{S[0]} \cdot \overline{S[1]} \cdot \overline{S[2]} \cdot T_B)$
4) Find equations for O[x] with sum-of-products
Example: output 1 ⇔ input stream % 3 = 0



MIPS ISA (4B instruction size)

r0	$zero	always 0
r1, r26, r27	$at, $k0, $k1	reserved for assembler
r2-r3	$v0-$v1	results
r4-r7	$a0-$a3	arguments
r8-r15	$t0-$t7	temps, not saved
r16-r23	$s0-$s7	saved contents
r24-r25	$t8-$t9	temps, not saved
r28	$gp	global pointer
r29	$sp	stack pointer
r30	$fp	frame pointer
r31	$ra	return address

64-bit integer subtraction:
- A[63:32] in $4, A[31:0] in $5
- B[63:32] in $6, A[31:0] in $7

⇒
```
subu $3, $5, $7
sltu $2, $5, $7
add  $2, $6, $2
sub  $2, $4, $2
```

```
fib:
    addi  $sp, $sp, -16      allocate stack space
    sw    $16, 0($sp)        save r16
    add   $16, $4, $0        r16 for arg n
    sw    $17, 4($sp)        save r17
    add   $17, $0, $0        r17 for arg a=0
    sw    $18, 8($sp)        save r18
    addi  $18, $0, 1         r18 for arg b=1
    sw    $31, 12($sp)       save return address
    add   $2, $17, $18       c=a+b
branch:
    slti  $3, $16, 2         use r3 as temp
    bne   $3, $0, done       branch if n=0
    add   $2, $17, $18       c=a+b
    add   $17, $18, $0       a=b
    add   $18, $2, $0        b=c
    addi  $16, $16, -1       n=n-1
    j branch
done:
    lw    $31, 12($sp)       } restore registers
    lw    $18, 8($sp)        }
    lw    $17, 4($sp)        }
    lw    $16, 0($sp)        }
    addi  $sp, $sp, 16       restore stack pointer
    jr    $31                return to caller
```

```
REP MOVSB                    beq  $1, $0, AfterLoop
↳ 3 registers:               CopyLoop:
  ECX, ESI, EDI                lb   $4, 0($2)
↳ repeat ECX-times             sb   $4, 0($3)
  copy 1 byte from             addiu $2, $2, 1
  address ESI to EDI           addiu $3, $3, 1
                               addiu $1, $1, -1
Example: ECX → $1,             bne  $1, $0, CopyLoop
ESI → $2, EDI → $3           AfterLoop:
```

ISA VS MICROARCHITECTURE

ISA: interface that is exposed to software
changes affect compiler and programmer

- Instructions: opcodes, addressing modes, data types, instruction types, registers, condition codes
- Memory: address space, addressability, alignment, virtual memory management
- Call, interrupt and exception handling
- Access control, priority and privilege
- Input-Output: memory-mapped vs. instruction
- Power and thermal management
- Multithreading and multiprocessor support
- Memory-mapped location of exception vectors
- Function of each bit in a programmable branch-predictor configuration register
- Order of load / stores in multi-core CPU
- Memory addressing modes available for arithmetic operations
- Program counter width
- General purpose register $29 is stack pointer
- Hardware floating point exception support
- Vector instruction support
- Memory-mapped I/O port address
- CPU endianness
- Virtual page size
- No condition codes
- 5-bit immediate can be used in ADD
- # general-purpose registers
- size of addressable memory

Microarchitecture: specifies how the underlying implementation actually executes the instructions. Changes are invisible to programmer and compiler

- pipelining (e.g. number of stages)
- in-order vs. out-of-order execution
- memory access scheduling policy
- speculative execution
- superscalar processing
- clock gating
- caching: levels, size, associativity, replacement
- error correction
- physical structure of registers
- ALU has no subtract unit
- # cycles to execute ADD instruction
- register file has 1 input & 1 output port
- latency of branch misprediction
- physical memory page size
- bit-width of interface CPU ↔ L1 cache
- # cache sets at each level of cache hierarchy
- max. reservation station capacity
- ALU critical path
- Instruction issue width
- # read/write ports in physical reg. file
- # ALUs
- # instructions fetched per clock cycle
- # prefetches of hardware prefetcher
- Reorder Buffer size

PIPELINING

- Fetch: CPU reads instr. from instr. memory
- Decode: CPU reads source operands from register file and decodes the instruction to produce the control signal
- Execute: computation with ALU
- Memory: reads or writes to/from memory
- Writeback: write result to register file

Pipeline stalls happen: • resource contention
• data / control flow dependencies
• multi-cycle operations

Handle **flow dependencies:** • detect and wait
• detect and forward data to dependent instr.
• detect and eliminate in software (↔ no hardware checks)
• detect and move out of way for indep. instr.
• predict, execute speculatively, verify
• do something else (fine-grained multithreading)

Reorder Buffer: complete instructions out-of-order, but reorder them before making results visible. Access register file first, and point to ROB entry which will contain result if reg not valid.

Valid	Destination Reg. ID	Dest. Reg. Value	Store-Addr.	Store-Data	PC	Val. Bits	Exception?

Handle WAR/WAW dependences:
register renaming (register ID → ROB entry ID)

```
                          1 2 3 4 5 6 7 8 9 10 11 13 15 17
1 MUL R5, R6, R7          F D E1 E2 E3 E4 M W
2 ADD R4, R6, R7          F D E1 E2 E3 - M W
3 ADD R5, R5, R6            F D - - E1 E2 E3 M W
4 MUL R4, R7, R7            F - - D E1 E2 E3 E4 M W
5 ADD R6, R7, R5                  F D - E1 E2 E3 M W
6 ADD R3, R0, R6                  F - D - - E1 E2 E3 M W
7 ADD R7, R1, R4                        F - - D E1 E2 E3 M W
```

ADD: 3 cycles, MUL: 4 cycles

min **read/write ports** reg file: 2×read, 1×write

data forwarding: cycles 7 and 10

internal forwarding in reg file: cycle 13 (decode & write at same time)

optimize: remove ② (WAW), move ④ up (stalls for D), switch ⑥ and ⑦ (no RAW dependency)

TOMASULO'S ALGORITHM

Implement dynamic scheduling with register renaming to eliminate output (WAW) and anti-dependences (WAR).

Reservation stations: holds the operands of an instruction. Instructions are executed out-of-order, when the operands are ready

OP	RS 1 (0 if ready	RS 2	Val 1	Val 2	Imm. / Addr.	Busy

reservation station ID — only one of these has a value

When a value is computed, it is streamed on the common data bus to all RS and reg file.

1. insert instr. & data (source tag or value) into a free reservation station
2. while in reservation station, watch common data bus. If tag seen, grab value. If all operands available, mark as ready for dispatch
3. dispatch instr. to functional unit when ready
4. after FU completed: put tagged value on CDB, reg file has tag for RS that writes to it. If see tag, grab value.

Example: draw RAT and RS after cycle 7:

```
MUL  R1, R2 → R3      F D E1 E2 E3 E4 E5
ADD  R3, R4 → R5        F D  -  -  -  -
ADD  R2, R6 → R7          F  D  E1 E2 E3
ADD  R8, R9 → R10           F  D  E1 E2
MUL  R7, R10 → R11             F  D  -
ADD  R5, R11 → R5              F  D
```

Register Alias Table

Reg	Valid	Tag	Value
R1	1	-	1
R2	1	-	2
R3	0	x	-
R4	1	-	4
R5	0	d (a)	-
R6	1	-	6
R7	0	b	-
R8	1	-	8
R9	1	-	9
R10	0	c	-
R11	0	y	-

Soure 1 + Source 2

	V	Tag	Val	V	Tag	Val
a	0	x		1	~	4
b	1	~	2	1	~	6
c	1	~	8	1	~	9
d	0	a		0	y	

Soure 1 * Source 2

	V	Tag	Val	V	Tag	Val
x	1	~	1	1	~	2
y	0	b		0	c	
z						
+						

# of tag comparators per RS = #FU * # source registers
↳ if every FU has separate data bus (usually 2)

total # tag comparators =
#FU * #RS per FU * # tag comp. per RS
+ #FU * # architectural registers

min size of tag = #FU * # RS per FU

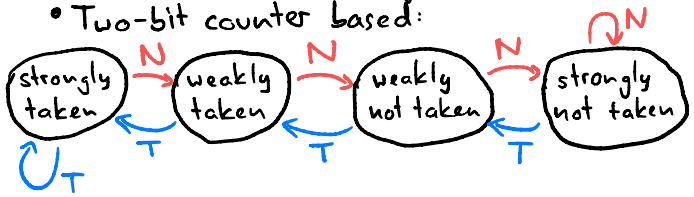
min size of RAT = (data + tag + valid) * # registers

min size of tag storage = tag size * # registers
+ tag size * #RS per FU * #FU * # source registers

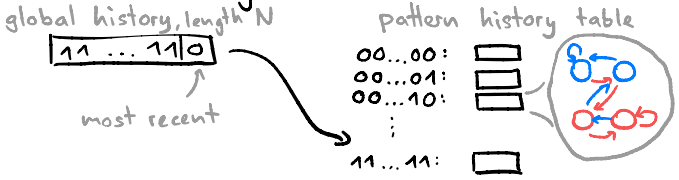
BRANCH PREDICTION

Static prediction direction schemes:
- Always not-taken
- Always taken
- Backward taken, forward not-taken

Dynamic prediction direction schemes:
- Last-time: single bit per branch, stored in branch target buffer (BTB)
- Two-bit counter based:



- Local: PC-based, for each branch
- Global: counter shared among all branches
- Two-level global:



⇒ associate branch outcomes with "global T/NT history" of all branches

- Two-level local: have a per-branch history. choose based on last time with the same local branch history

Example: pipeline w/ multiple stages, each stage completes in 1 cycle. stalls only when conditional branch instruction. Consider:

```
LOOP1:
   SUB  R1, R1, #1     //R1=R1-1
   BGT  R1, LOOP1      //Branch if R1>0
LOOP2:
   B    LOOP2          //Branch until killed
```

The microbenchmark has following results:

Initial R1 value	Number of cycles taken
4	51
8	63
16	87

⇒ $C = P + I - 1 + B \cdot D$ (B: # of conditional branch instructions; D: # of cycles stalled per conditional branch)

C: total # of cycles; P: # pipeline stages; I: total # of dynamic instr. executed

⇒ $51 = P + I - 1 + 4 \cdot D$
$63 = P + I - 1 + 8 \cdot D$
$87 = P + I - 1 + 16 \cdot D$

↳ stall per cond. branch: D = 3 cycles
pipeline stages + # dynamic instructions: P + I = 40

Example:
```
for(i=0; i<10^6; i++)      //B1
    if(i%3==0)             //B2
```

⇒ 2-bit GHR, PHT with 2-bit saturating counter. Initial values for 100% mispredict in first 5 iterations:

⇒ T T T N T N T T T N

NN NT TT TN

PHT Entry	Value
T T	01
T N	00
N T	01
N N	00 or 01

Example:
```
for(i=0; i<N; i++)          //B1 (has local correlation)
    if(a[i]%2==0)           //B2 \
    if(a[i]%3==0)           //B3  > globally correlated
    if(a[i]%6==0)           //B4 /
```
↳ B4 taken ⟺ B2 & B3 t.

2-bit GHR. PHT with 4 entries: TT, TN, NT, NN
PHTE inc. when taken. Calc. PHTE of TT after 120 it.

B3: B1_T & B2_T = $\frac{1}{2}$ ⇒ INC = $\frac{1}{2}\cdot\frac{1}{3} = \frac{1}{6}$, DEC = $\frac{1}{2}\cdot\frac{2}{3} = \frac{1}{3}$

B4: B2_T & B3_T = $\frac{1}{6}$ ⇒ INC = $\frac{1}{6}\cdot 1 = \frac{1}{6}$, DEC = 0

B1: B3_T & B4_T = $\frac{1}{6}$ ⇒ INC = $\frac{1}{6}\cdot 1 = \frac{1}{6}$, DEC = 0

B2: B4_T & B1_T = $\frac{1}{6}$ ⇒ INC = $\frac{1}{6}\cdot\frac{1}{2} = \frac{1}{12}$, DEC = $\frac{1}{6}\cdot\frac{1}{2} = \frac{1}{12}$

chance of B2_T if B4_T and B1_T

Total: $(\frac{1}{6} - \frac{1}{3}) + \frac{1}{6} + \frac{1}{6} + (\frac{1}{12} - \frac{1}{12}) = \frac{1}{6}$ ⇒ $120 \cdot \frac{1}{6} = 20$

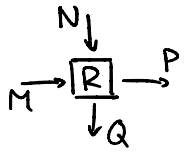
SYSTOLIC ARRAYS

⇒ Array of PEs instead of single PE

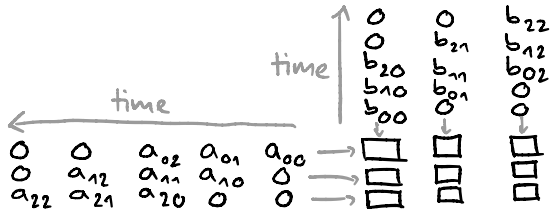
Example matrix multiplication:

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

Processing Element:



P=M, Q=N, R= R+(M·N)



VECTOR PROCESSING

Memory banks: split up memory, so many data elements can be fetched concurrently.

Stride: constant distance that separates elements that are adjacent in a vector register.

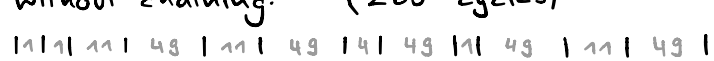
The data for vector instructions is stored in vector registers, which holds multiple "vectors".

Vector chaining is data forwarding with vectors. It allows processing of individual elements as soon as they are ready

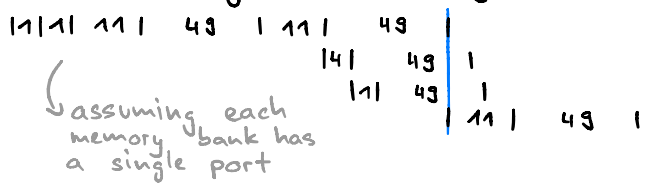
Example: i=0...49 C[i]=(A[i]+B[i])/2

MOVI	VLEN = 50	1
MOVI	VSTR = 1	1
VLD	V0 = A	11 + VLEN - 1
VLD	V1 = B	11 + VLEN - 1
VADD	V2 = V0+V1	4 + VLEN - 1
VSHFR	V3 = V2 >> 1	1 + VLEN - 1
VST	C = V3	11 + VLEN - 1

Without chaining: (285 cycles)



With chaining: (182 cycles)



If # data elements > vector length: break loops and execute multiple times with (different) VLEN

banks should be at least equal to the number of cycles of VLD and VST if stride=1. Stride and # banks should be relatively prime if stride>1.

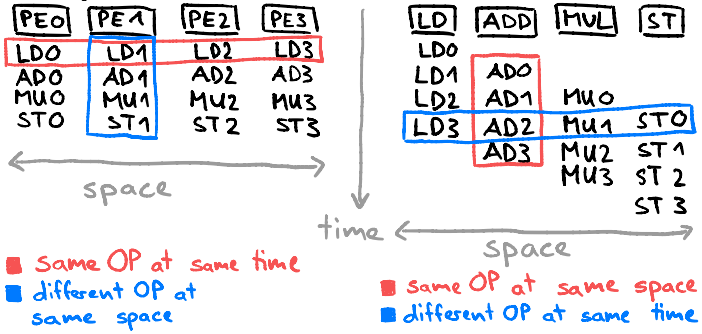
SIMD PROCESSING

Warp: a set of threads that execute the same instr. at same PC on different data

⇒ # warps = (# threads)/(# threads per warp)

SIMD utilization: fraction of SIMD lanes executing a useful operation

Array Processor vs. Vector Processor



- Same OP at same time
- different OP at same space
- same OP at same space
- different OP at same time

Examples

Tracing the cache: (fully-assoc. / 4 blocks)

Req	A	B	A	H	B	G	H	H	A	E	H	D	...
M?	x	x		x	x	x				x		x	
MRU	A	B	A	H	B	G	H	H	A	E	H	D	
		A	B	A	H	B	G	G	H	A	E	H	
			B	A	H	B	B	G	H	A	E		
LRU				A	A	A	B	G	G	A			

⇒ Cache miss rate = $\frac{\#M}{\#Req}$

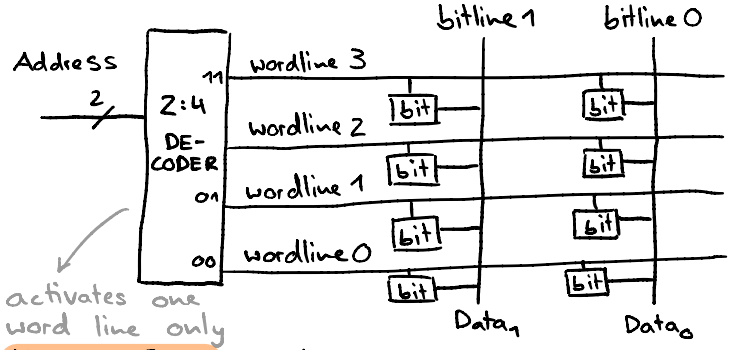
Cache Reverse Engineering

Addresses Accessed								Hit Rate
0	16	24	25	1k	255	1100	305	2/8
31	65k	65k	131k	262k	8	305	1060	3/8
262k	65k	4						2/3

- Block size: seq1: 8B: {0}, {16}, {24,25}, {255}, ... 16B: {0}, {16,24,25}, {255}, {305}, ... ⇒ 2 hits!
- Associativity: seq2: 3 hits w/ block size 16B ⇒ 31, 65k, 305 hit ⇒ 8 must miss, replaced by blocks mapping to set 0 (65k, 131k, 262k) ⇒ 2-way
- Replacement: seq3: with LRU only 262k would hit ⇒ with FIFO, 4 also hits
- Cache size: we know block size 16B, 2-way. ⇒ 4 bits for indexing byte in block ⇒ 7 bits if size=4kB, 8 bits if size=8kB ⇒ 2048 maps to set 0 ⇔ size is 4kB

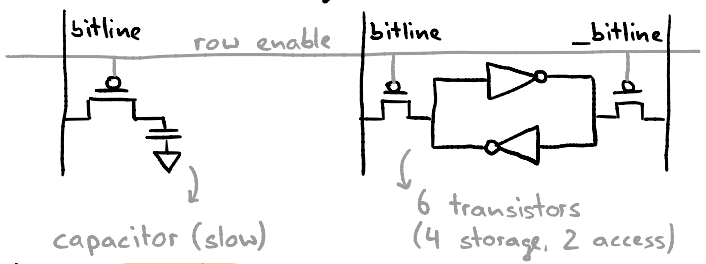
CACHE & MEMORY

Memory Array:



Memory Bank: divides a large array into multiple banks which can be accessed indep. Has rows (many) and columns (fewer) that have distinct addresses. A row buffer stores the most recently accessed row (simple cache)

DRAM	vs.	SRAM
• slow, but cheap		• fast, but expensive
• requires refresh		• no refresh required
• used in main memory		• used in caches



Memory **locality:**

- temporal: program accesses same memory location many times in small window of time
- spatial: program accesses nearby memory location in a window of time

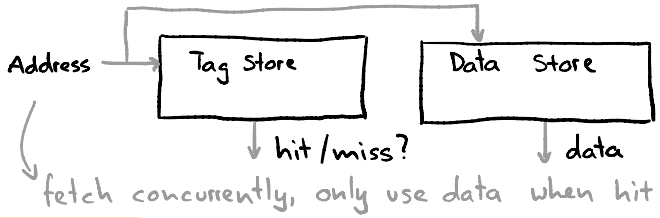
Memory is divided into blocks: | Data (64B) | Tag |

The block address maps to a potential address in cache (determined by index bits):

8-bit-address: | 2b | 3bits | 3 bits |

tag | index | byte in block

If the stored tag is valid & matches the tag of the requested block: cache hit



Associativity:

- fully-associative: every memory chunk can be mapped to any cache block
  ↳ | 5 bits | 3 bits | → no index, only 5 bit tag
  ↳ location needs to be calculated w/ logic
- direct-mapped: a memory chunk can only be mapped to a single cache block
- set-associative: allows same index multiple times (n-way associative)
  ↳ 2-way-associative: | 3 bits | 2 b | 3 bits |
  tag ← increased; index | byte in block
  points to a set instead of single block

Replacement Policy:

- FIFO: first in, first out
- LRU: least recently used
- Not MRU: not most recently used

↳ for every program a different replacement strategy might be optimal

Cache performance:

- cache size (total data capacity excl. tags) C
- block size (data associatet w/ address tag) $b = 2^{offset}$
- associativity (how many blocks can be represented in same index/set?) N
- number of blocks: $B = C/b$
- number of sets: $S = B/N$
- tag store size: $2^{index} \cdot ([associativity] \cdot (tag + valid + dirty) + [LRU])$
- data store size: $2^{index} \cdot [associativity] \cdot [block\ size]$

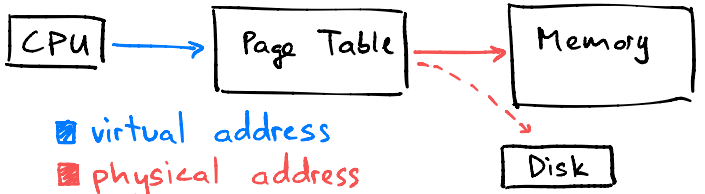
PREFETCHING

Performance:
- Accuracy: $\frac{used}{sent}$ prefetches
- Coverage: prefetched misses / all misses
- Timeliness: $\frac{on\text{-}time}{used}$ prefetches

Runahead execution: with small instruction window, pre-execute instructions if data has to be fetched from memory → prefetch

Stride prefetches: if there is a constant stride between memory accesses → use to predict

VIRTUAL MEMORY



Virtual Address: | Page table number | Page table offset | Page offset |

↳ uses 2 levels of page tables

Page table entry: | V | Physical Page Number |

⇒ page in virt. memory is block in cache

Physical Address: | Physical page number | Page offset | ← same

Translation Lookaside Buffer (TLB):

Small cache of recently used PTEs (i.e. recently used Virtual-to-Physical translations)

Page faults: if a page is not in physical memory but on disk: which to replace?
⇒ clock page replacement: pointer to last accessed frame, set R bit in PTE if accessed. when replace, look for first PTE with R bit not set (and unset the traversed PTEs)

Physical adress space: $2^{[page\ offset]+[physical\ tag]}$