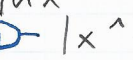
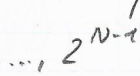
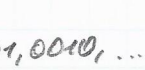
Levels of abstraction

1. Physics: Motion of electrons
2. Devices: Transistors and Terminals (measure voltage/current)
3. Analog Circuits: In which devices are assembled (amplifiers)
4. Digital Circuits: Logic gates, restrict voltage
5. Logic: More complex structures (memory)
6. Microarchitecture: Describes hardware for programmer

Gatter

- OR | $x \vee y$ | $x+y$ |  | $x \mid y$
- NOR | $\neg(x \vee y)$ | $\overline{x+y}$ |  | $\sim(a \mid b)$
- AND | $x \wedge y$ | $x \cdot y$ |  | $x \& y$
- NAND | $\neg(x \wedge y)$ | $\overline{x \cdot y}$ |  | $\sim(a \& b)$
- NOT | $\neg x$ | $\bar{x}$ |  (Buffer) | $\sim x$
- XOR | $x \neq y$ | $x \oplus y$ |  | $x \wedge y$ { 00→0, 10→1, 01→1, 11→0
- XNOR | $x = y$ | $\overline{x \oplus y}$ |  | $\sim(x \wedge y)$

Number System

- Binary: 2^N (0, 1, 2, ..., 2^{N-1}) | $2^4 \Rightarrow 0,1,2,...,15 \Leftrightarrow 0000, 0001, 0010, ..., 1111$
- Hexadecimal: 16^N (0, 1, 2, ..., 9, A, B, ..., F) | $16^1 \hat{=} 2^4$

Big-Endian: MSB ..... LSB
Little-Endian: LSB ..... MSB

Bits Bytes (=Unsigned)

- 1 Byte $\hat{=}$ 8 Bits $\hat{=} 2^8 \hat{=}$ 256 possibilities (=Unsigned: $[0/2^N - 1]$)
- 1 Word (Handeled by Microprocessors) | 32/64 Bit words
- Bit in the 1's column = least significant bit >< most significant bit

Signed numbers (positiv/negative)

2 Darstellungen für 0

- Sign/magnitude: most significant bit $\pm \frac{0}{1}$: -5_2 : 1101 / $+5_2$: 0101 | $[-2^{N-1}+1 / 2^{N-1}-1]$
- Two's Complement: $[-2^{N-1} / 2^{N-1}-1]$ | zero = 0....0 / kleinste Zahl: 1000...0 / grösste: 0111...1
  - Most significant bit ist sign-bit: 1 = negativ / 0 = positiv
  - Sign reversed: 1. invert all bits 2. add 1 to wb: $+2_{10} = 0010_2$
  - Addition/Subtraction: ± Zahl + ± Zahl (immer addieren) $-2_{10} = 1110_2$
  - When number is extended to more bits, sign bit to msb

Logic Levels: $0 - V_{OL} - V_{IL}$ - Forbidden zone - $V_{IH} - V_{OH} - V_{DD}$ (V_{IL}/V_{IH} for noise margins)

Combinational Logic Blocks (have no memory / building blocks)

- Output depends only from the input → has no memory
- Every circuit block itself is combinational and has no cyclic paths

Boolean equation variables are called literals

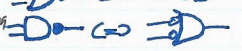
Minterm: Formel, wenn wir überall true einsetzen (And/∧/·)

Maxterm: Formel, wenn wir überall false einsetzen (OR/∨/+)

A	B	y	
0	1	1	$\bar{A} \cdot B$ ∑ Min
0	0	0	$A + B$ ∑ Max

Sum of product: Summing up each minterm, where output/y = true / $y = \bar{A}B + ... + ...$

Products of sum: Producing up each maxterm, where output/y = false / $y = (A+B) \cdot (B+C) \cdots$

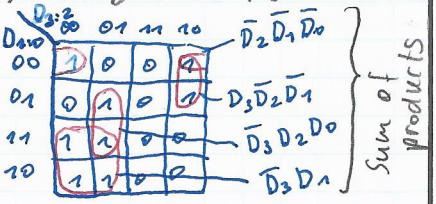
De Morgan: $y = \overline{A \cdot B} = \bar{A} + \bar{B}$ / $y = \overline{A+B} = \bar{A} \cdot \bar{B}$  (=Bubble pushing)

From logic to gates:  $\bar{A}B\bar{C}$ + No connection + connection

Illegal Values: X: unknown value / Z: Entweder 0/1

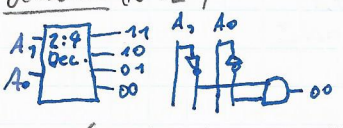
Karnaugh-Maps: (K-Maps) Eliminate complementary forms of variables (größer, besser)

- Weniger Kreise, desto besser / Jeder Kreis 1/2/4/8/2^N lang/breit/beides
- Kreise beinhalten nur 1en oder X (= Don't care)



Multiplexer

Decoder (N → 2^N)



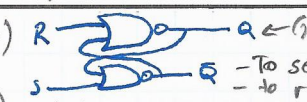
- Low → High: Rising Edge / High → Low: Falling Edge
- Propagation delay (t_{pd}): Max time from when an input changes, until the output changes
- Contamination delay (t_{cd}): minimum time from when an input changes until any output starts to change its value (value must not final sein)

Critical path (longest, and therefor slowest path)

2-Level-Logic: Only and/or

> And cyclic paths

Sequential logic blocks (has memory)



Q ← Gespeicherter Wert
- To set a bit makes it true
- to reset a bit makes it false

- SR-Latch (Set/Reset)
- D-Latch (Date Input D / clock-input Clk) - Clk = 1 = transparent = open → buffer
- D-Flip-Flop (Two back to back d-latches) - Clk = 0 = opaque = closed → blocks new data, retains old value

- D is copied to Q on the rising edge of the clock and remembers its state at all other times

Clk	D	Q
0	x	Q previous
↑	0	0
↑	1	1

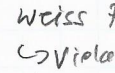
- Latch: Not edge-triggered, bistable / Flip-Flop: edge-triggered
- Enabled Flip-Flop: Has another input called EN (EN=True: Normal flip-flop (EN=false: ignores clock)
- Resettable Flip-Flop: Reset=True → output=0

Finite-State Machine: - Circuit with k registers has 2^k unique states
- Next-state-logic: holds current state and on each clock-cycle it changes to next-state
- State-register: stores the different states
- Output-logic: gives out the outputs
- Moore: Only output only depends on current state
- Mealy: Output depends on current state and input
- One-hot-encoding: a seperate bit for each state: S1: 001 / S2: 010 / S3: 100 (simpler for next)
- Binary-encoding: Not as simple to encode next-state, aber weniger Flip-Flops
- Moore has more states than Mealy

Timing: element has aperature time around clock edge, during which the input must be stable for the flip-flop (t_{setup} - rising edge - t_{hold}) ← aperature time
↳ for sequential
↳ The clock period has to be long enough for all signals to settle
↳ vor allem wegen Flip-Flop
- For asynchronous inputs from outside device (button) we need synchronizer
- Dynamic discipline: inputs of synchronous circuit must be stable during t_{setup} + t_{hold}

Frequenz: T_c between rising edge to falling edge / Frequenz: $f_c = 1/T_c$
↳ $T_c \geq t_{propagation} + t_{setup} + t_{flip\text{-}flop\ clock\ to\ Q\ propagation\ delay}$

Metastability: Wenn ich Knopf bei rising-edge drücke, weiss Flip-Flop nicht den Value
↳ Violates dynamic discipline



Parellelism:
- Spatial parallelism
  ↳ Multiple copies of hardware so that multiple tasks can be done
- Temporal parallelism / Pipelining: No additional hardware, just one register per stage
- Latency: L / Throughput: (no parallelism) 1/L / (spatial) N/L / Pipelining 1/longest stage
- Throughput: Amount of work in a given time: 1/longest stage
- Latency: Dauer eines ganzen Durchlaufs: #stages · longest stage

---

Verilog:

combinational logic
```
module sillyfunktion (input a,b,c, output y);
assign y = ~a & ~b & ~c;
endmodule
```

4 Bit Bus
```
module inv(input [3:0]a, output [3:0]y);
assign y = ~a;
endmodule
```
↑ Little Endian

Multiplexer 2:1
```
module mux2(input [3:0] d0, d1, input s, output [3:0]y);
assign y = s ? d1 : d0;
endmodule
```
↑ if/else

In Verilog, always-statement signals keep old value, until sensitivity list takes place; Always with q <= d
```
module flop(input clk, input [3:0]d, output reg [3:0]q);
always @(posedge clk)
  q <= d;
endmodule
```
Must be a reg → left

- = Blocking: Values are assigned immediatly/++i
- <= Non-Blocking: At the end of block/i++

Multiplexer 4:1
```
module mux4(input [3:0] d0, d1, d2, d3, input [1:0] s,
output [3:0]y);
assign y = s[1] ? (s[0] ? d3:d2) : (s[0] ? d1:d0);
```

Internal wire
```
wire p;
```

- /, //, % (Mult/Div/Mod) / <<, >> (Left, Right Shift)
- ==, != (Comparison)
- N'Bvalue (Number/Base/value) z.B. 3'b101

```
mux2 test(.d0(A); .d1(B); .S(c); .output(y));
```

- case: always @ (*) case (data)
```
0: segments = 1'b0,
1: segments = 1'b1;
default: segments = 1'b1; endcase
if: always @(*) if (a[3]) y = 2'b01;
else if (a[2]) y = 2'b00;
```

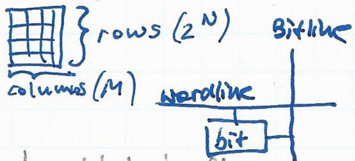
SM: #state register
```
always @(posedge clk, posedge reset)
  if (reset) state <= S0; else state <= nextstate;
//next state logic
always @(*)
  case (state)
    S0: nextstate = S1;
    S1: nextstate = S2;
    S2: nextstate = S0;
  endcase
//output logic
assign y = state
```
← moore: only states
↑ mealy: also input

```
parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b02;
```

Jonas B.

Arithmetic Circuits ← 1+1 = S=0 / Cout = 1

- Half adder: A / B / S / Cout / Full adder: A / B / S / Cout / Cin (carry Bit input)
- Magnitude Comperator: Usually done by computing A-B and looking at the sign
- Arithmetic Logic Unit (ALU) combines a variety of math/logic operations
- SLT = Set less than: A < B => Y = 1 / 1. S = A - B / 2. zero-extend unit produces N-Bit 0 Array / 3. Sign bit is input
- Shifters: ·2 / S = 0101 ·2 → 1010
- Multiplication: 0100 · 0011 = 0100 + 0100 = 1100



- Memory is organized in an array
- N Bit Address
- Each column is called a word

Memory (DRAM / SRAM / ROM) – Ram braucht immer Strom / – Rom braucht kein Strom

- FlipFlops: Very fast, parallel accessable, expensive other address in block
- SRAM: Relatively fast, only one data word at a time, less expensive
- DRAM: Slower, reading destroys content, must be rewritten, cheap, refreshed every x ms (capacitor leak)
- Volatile: loses data, when power is off (RAM) / Non-Volatile: keeps data when power off (SSD)
- Cache (static RAM) / Main Memory (Dynamic RAM) / Virtual Memory (Harddisk / SSD)

Cache misses: compulsatory (first load) / capacity / conflict:

- Temporal locality: if you used a book recently, you are likely to use it again
- Spatial locality: if you use a particular book, you are likely to use books in the same shelf
- Cache hit: when CPU requests data, that is already in the cache → quick
- Cache miss: Otherwhise CPU gets data from main memory
- Miss-rate: $\frac{\#Misses}{\#Total} = 1 -$ Hit Rate / Hit-Rate: $\frac{\#Hits}{\#Total} = 1 -$ Miss-Rate
- Average memory access time: AMAT = t_{cache} + MissRate$_{cache}$ · ($t_{MainMemory}$ + MissRate$_{MainMemory}$ ($t_{virtual\ Memory}$))
- Cache exploits temporal + spatial locality to achive a low miss rate
- Cache block: The fetched word + surrounding words for spatial locality
- Block size: number of words in a cache block
- Capacity C: contains B = C/b blocks
- Sets S: A cache is organized into S sets, each of which holds one or more blocks (each memory location is exactly mapped to one set)
- Direct mapped cache: each set contains only one block, so cache has S = B sets
- N-way set associative: each set contains N blocks, so cache has S = B/N
- Fully associative: only one set [Way 1: V | Tag | Data] [Way 2: V | Tag | Data] ...... [Way N: V | Tag | Data]
- Each set has a memory address: Tag (which of the many possible addresses is held in that set) + Set (set Number) + Byte offset
- Ein Set ist eine Art Vorfilter, dass nicht ganzer Cache angeschaut werden muss
- What data is replaced? direct-mapped: the new block for the set / fully associative: LRU (least recently used blocks)
- There are virtual addresses, that are either located in cache or in storage
  ↳ CPU translates them into physical addresses

Assembly MIPS-Lanuage (has 32 registers, that holds commonly used operands) $0 = 0

```
- add a, b, c   // a=b+c   / addi $s0, $s0, 4
- sub a, b, c   // a=b-c   / addi $s0, $s0, -4
- lw $s3, 1($0)   // load memory word 1 into $s3   (1 = offset)
- sw $s7, 5($0)   // write $s7 to memory word 5   (5 = word No.)
- sll $t0, $s1, ...   // ·2^N
- srl $t0, $s1, 4   // :2^N   $t0 = 2^N · $s1
- Branch: beq $s0, $s1, target   // $s0 == $s1 -> target
          bne $s0, $s1, target   // $s0 ≠ $s1 -> target
- Jump: j target
- For-Loop: addi $t0, $s0, 128
    while:            <- if $t0==128 -> done
        beq $s0, $t0, done
        addi $s0, $s0, 1
        j while
    done:
```

- R-Type: op, rs, rt, rd, shamt, funct (Register) (rs = source, rd = destination, shamt = only for shift)
  ↳ add / sub
- I-Type: op, rs, rt, imm (immediate) (rs = source, rt = dest, imm = number)
  ↳ addi / lw / sw
- J-Type: op, address (jump)
- Procedures: call: jump and link (= jal)
  method: jump register (= jr)
  arguments: $a0 – $a3
  return value: $v0
- Stack: Um während Method-Call keine Daten zu überschreiben ($sp: Stack pointer)

```
store:    addi $sp, $sp, -4   (Stack geht nach unten)
          sw $R16, $sp
retrieve: lw $R16, $sp
          addi $sp, $sp, 4
```

Microarchitecture

Execution time = (# instructions) · $\left(\frac{\text{cycles}}{\text{instruction}}\right)$ · $\left(\frac{\text{seconds}}{\text{cycle}}\right)$

Single Cycle

- Each instruction is executed in a single cycle
- Clock cycle ist abhängig von der langsamsten Operation (Critical path)

1. Read one instruction / 2. Decode what it means / 3. Find operands from register or memory
4. Perform operation / 5. write back data / 6. Go to next instruction

- Program counter (PC) tells us, where to find the next instruction
  ↳ Needs to be incremented +4 during each cycle
- Register file: read write from register
- Arithmetic logic unit (ALU): performs operation
- Control unit: setzt alle Funktionssignale: MemtoReg / MemWrite / Branch / AluSRC / RegDst / RegWrite + AluOP
- CPI = Cycles per Instruction = 1 (single cycle)
- Disatvantages: Clock Cycle: slowest instruction (LW) / 3 Adders / seperate instruction / data memory

Transfer from memory to register?
Writing in Memory back?
Branch?
Immediate or Register
Is destination in register write back to register?

The von Neumann Model

- Memory is unified between instructions / data
- instructions stored in a linear memory array -> darum immer PC + 4

Multicycle Processor

1. Fetch: fetch instruction from memory
2. Decode: the instruction is being encoded
3. Execute: ALU
4. Memory access
5. Write Back results

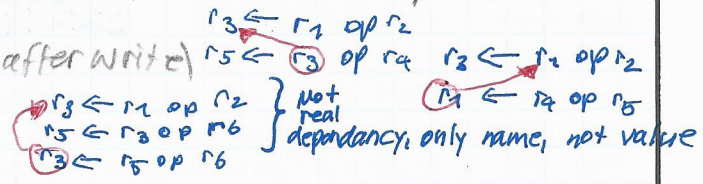
} 1 phase per cycle
↳ finite state machine

- Execution time = = #instructions · CPI · T_c
- What if memory takes longer than one cycle to complete
  - Asynchronous: we wait for ready bit from memory-controller
  - Synchronous: we wait exactly x-cycles, because we know, how long a mem access needs

Pipelining „Assembly line production"

Same pipeline stages

- Ideal: Identical operations / uniform suboperations / independant operations
- Data dependances
  - Flow dependance (true dependance, read after write)
  - output dependance (write after write)
  - Anti dependance (write after read)

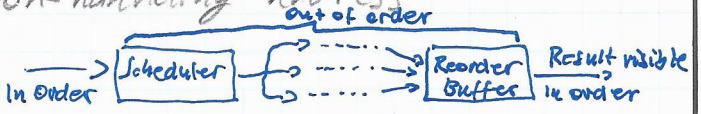


- How to handle data dependencies?
  1. detect and wait until value is available (stalling)
  2. detect and forward / bypass data to dependent instructions
     ↳ Compiler ändert Reihenfolge, indem er unabhängige Instruktionen zwischen Abhängige gibt
  3. Detect and eliminate dependancy at the software level
  4. fine grained multithreading: switch to another thread every cycle -> no dependencies
- Detection of dependencies: scoreboarding: jedes Register hat eigenes valid bit, das resettet wird, wenn ein thread ins Register schreiben will, spätere instructions wissen das dann
  5. data forewarding: sobald Ergebnis berechnet wurde, wird es direkt an abhängige, spätere Instruktion weitergeleitet, und erst dann im Register gespeichert
- Control dependancies / Branch prediction -> if prediction is wrong -> Pipeline gets flushed
  - Branch is not determined until 4. stage of pipeline
  1. Always predict, the branch is not taken
  2. Backward branches are usually taken (loops)
  3. Look at history, whether branch was previously taken

Hardware/software Scheduling
- Software: static scheduling, fixed, known before runtime
  - compiler orders, hardware executes them in order
  - software doesn't know source-registers
    - ↳ for example, multiplication with 0, software doesn't know, but hardware does, an copy directly 0
  - Everything, that is determined at run-time is bad
  - Profiling: Compiler testet mit mehreren Inputs
- Hardware: dynamic-scheduling, out-of-order

Exceptions vs. Interrupts
- Exception: caused by internal to running thread
- Interrupt: caused by external to running thread (mouse/keyboard)
- Handle exceptions at the same time
- Handle interrupts, when all instructions are finished in pipeline
- Precise Exception: wenn sie geworfen wird, müssen alle vorherigen Instruktions abgeschlossen sein, und keine mehr angefangen worden
- When Exception is detected:
  1. All previous instructions are finished
  2. Flush all younger exceptions
  3. Save PC and registers
  4. Redirect fetch address to exception-handeling-address
  5. Exception is handeled
  6. Resumes previous PC-address

↳ ohne precise exceptions ist Debugging unmöglich



Reorder buffer
- Complete instructions out-of-order, but reorder them before making results visible
- Value gets written in reorder-buffer, and he reorders results and writes them back
- Dependencies: Remapping: Wenn ich irgendwo hinschreiben muss, marke ich Pointer von Memory to reorder-buffer

Superscalar-Execution
- Fetch, decode, execute, retire multiple instructions per cycle
- Need to add extra hardware

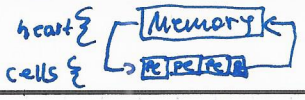
Combinations
- Scalar >< Superscalar + - Inorder >< out-of-order

Very-long-instruction-word
- Consists of multiple, independent instructions packed together in one instruction

SIMD-Processing (=Single instruction operates on multiple data)
- Eine Implementation von SIMD sind Vector-Prozessoren (GPU) (=Arrayprozessor)
- Beispiel: Matrixmultiplikation, same operation performed on many data elements
- Advantages: - Weniger branches (loops fallen weg)
  - Weniger fetch, denn eine Instruktion für viele Daten
  - Keine Abhängigkeiten
- Eine operation muss SIMD-Voraussetzungen erfüllen
- Memory of Vector-GPU is divided into banks, that can be accessed independently

Vector-Chaining: The same as forwarding, simply for SIMD-instructions

GPU
- Each thread executes the same code, but on a different data piece
- Warp: a set of threads that execute the same instruction at the same PC-count
- Jeder thread im warp durchgeht alle 6-Stages
  - ↳ Also "Single-instruction-on-single-data" in jeder Einheit im Warp, aber zusammen dann SIMD



Systolic arrays

- Replace a single processing element with an array of processing-elements
- Difference from pipelining
  - These are individual things to be done on the data
  - Array structure can be non-linear and multi-dimensional
- Advantage: Specialized, improved efficiency, simple design, high concurrency
- Disadvantage: not general -> specialized
- Taken further: pipeline parallelism and stream processing