# Vom Silizium zur Simulation und zurück
# *oder*
# Warum interessiert sich noch jemand für einen 40 Jahre alten Prozessor?

**Michael Engel**
Fakultät Elektrotechnik und Informatik
Hochschule Coburg

9. März 2017

DFG
SPP1500

# Überblick

- Simulation von Systemen und Echtzeit
- Abstraktionen: von schlampig bis pingelig... ☺
- Der 6502-Prozessor
  - Überblick über Implementierungen
  - Herausforderungen der zyklengenauen Simulation
  - Black box und white box-Ansätze
- Visual 6502: Reverse Engineering
  - Die Visual 6502-Simulation: Ausführung eines 6502-Programms
  - Vom Layout zur Netzliste...
  - ...von der Netzliste zu C-Code
  - ...und vom C-Code zum Chip: FPGA-Implementierung
- Was macht man nun damit?

# Simulation von Systemen und Echtzeit

- **Warum werden Hardwaresysteme simuliert oder emuliert?**
- Ausführen vorhandener Software, auch wenn Hardware nicht (mehr) verfügbar ist
  - Ersatz von Steuersystemen
  - Virtualisierung von Systemen
- Entwicklung von Software für Hardware, die noch nicht existiert
  - *Rapid prototyping*, *design space exploration*
- Präzise Analyse von Hardwarezuständen
  - Einblick in Zustand von Speichern und Signalen bei Nanometer-Strukturen von ICs teuer und aufwändig

# Simulation von Systemen und Echtzeit: Beispiel SNES

- **Emulation des Super Nintendo Entertainment System (SNES) aus dem Jahr 1990**
  - 65C816-Prozessorkern mit 3,58 MHz
  - 128 + 64 kB RAM
  - Maximal 512 x 478 Pixel Auflösung, 15 Bit Farbe
- Erste Emulatoren in 90er Jahren
  - Ca. 25 MHz Rechenleistung notwendig (486er!)
  - ***Fast*** alle Spiele laufen, aber Timingabweichungen von bis zu 20%

https://arstechnica.com/gaming/2011/08/accuracy-takes-power-one-mans-3ghz-quest-to-build-a-perfect-snes-emulator/

# Simulation von Systemen und Echtzeit: Beispiel SNES (2)

- **Problem der unpräzisen Emulation**
  - Spieleprogrammierer nutzen Hardware direkt
  - Nutzung von Seiteneffekten, Timing von Maschinenzyklen
- Beispiel: „Air Strike Patrol“: Schatten unter Flugzeug ist Zielhilfe
  - Durch Mid-Scanline Effekte erzeugt, aufwendig zu emulieren

Original



Unpräzise Emulation

# Abstraktionen: von „schlampig“ bis pingelig

- **Unpräzise vs. präzise Emulation**
- Präzise Emulation des SNES (in „bsnes“) benötigt 2–3 GHz Rechenleistung!
  - …und die Berücksichtigung vieler Sonderfälle, z.B. exotische Display-Modi:

Original



Unpräzise Emulation

# Abstraktionen: von „schlampig“ bis pingelig (2)

- **Wofür wird die zusätzliche Rechenleistung benötigt?**
- Hardware ist *parallel*, Software aber *sequentiell* (auf single cores)
- Signale in Hardware haben *zeitliche Abhängigkeiten*
  - Ohne Berücksichtigung: Deadlocks, Reihenfolgeprobleme
- Software-Emulator muss Abhängigkeiten durch *Synchronisation* von SW-Threads nachbilden – schlimmstenfalls ***zyklengenau!***
  - Unpräzise: nur die „wichtigsten“ – Präzise: möglichst alle
- Beispiel: ZSNES vs. bsnes-Emulator:

| # Synchronis./s | ZSNES | bsnes |
|---|---|---|
| CPU | 600.000 | 21.477.272 |
| SMP (Audioproz.) | 256.000 | 24.576.000 |
| DSP | 32.000 | 24.576.000 |
| FPU | 15.720 | 21.477.272 |
| **Summe** | **903.720** | **92.106.544** |

CPU-Basis-Takt SNES: 21,47 MHz

SMP-Takt: 24,576 MHz

# Der 6502-Prozessor

- Entwicklung von MOS Technology
  - Markteinführung 1976: US$25
- 3µm NMOS, 3510 Transistoren, 8 Bit, 1 MHz
- CPU der ersten Heimcomputer
  - Apple, Atari, Commodore, Acorn
- Heute: CPU in eingebetteten Systemen
  - CPUs und Cores verfügbar von WDC
- Aktuelle Einsatzgebiete
  - Digitale Bilderrahmen, elektron. Spielzeug
  - Armaturenbrett-Controller (Micronas)
  - Automatikgetriebe (VDO)
  - Schachcomputer

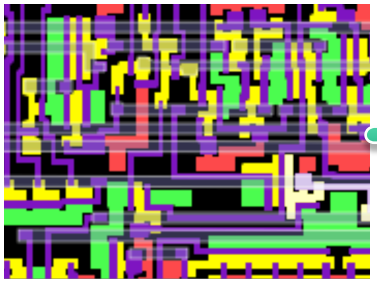


# Vergleich 6502–heute

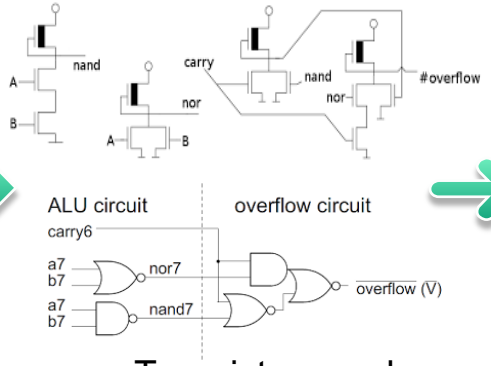
- **Welcher Aufwand ist allein für die präzise Simulation eines Prozessors notwendig?**
- MOS 6502
  - Entwickelt 1975
  - 3510 Transistoren
- Vergleich:
  - Core i7 „Ivy Bridge“ + GPU: 1,7 Milliarden (inkl. Cache)
  - Apple A10 (iPhone 7): 3,3 Milliarden (inkl. GPU+Cache)

# Emulation auf verschiedenen Abstraktionsebenen

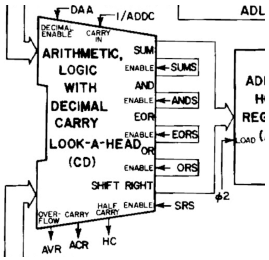
- **Verhalten einer CPU auf verschiedenen Abstraktionsebenen**
  - ISA-Ebene: Zustandsänderung in für Programmierer sichtbaren Elementen (Register) nach Befehlsausführung
  - Mikroarchitektur: Modellierung von HW-Blöcken, z.B. ALU
  - Transistor- und Gatterebene: Einzelne Logikkomponenten
  - Layout-Ebene: Abstände zwischen Transistoren, Signallaufzeiten



Halbleiter-Layout




Transistor - und Gatterebene




Mikro-architektur

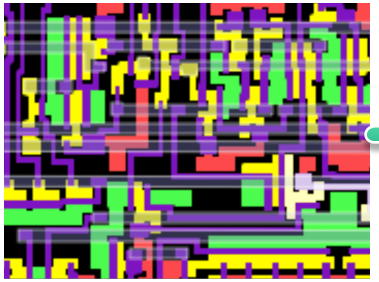

Instruction set architecture

**Schmoll, Heinig, Marwedel, Engel.** ***Improving the Fault Resilience of an H.264 Decoder using Static Analysis Methods.*** **ACM TECS, 2013**

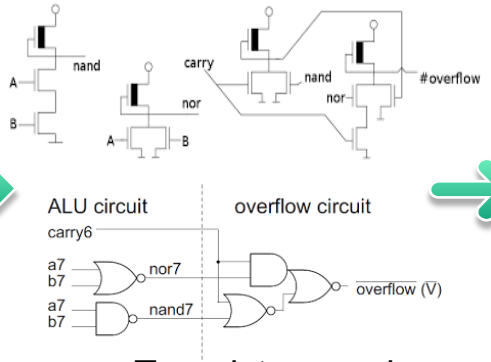
# Abstraktionsebenen: Beispiel

- **Einfaches 6502-Programm: Prüfsumme berechnen**

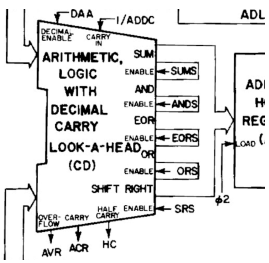
```
Checksum:         ; Sum 10 bytes pointed by $70/$71
    LDA #0        ; Set A (our checksum) to zero
    TAY           ; Also put zero in Y (the loop counter)
Loop:
    CLC           ; Clear carry (all adds are with carry)
    ADC ($70), Y  ; Add check sum
    INY           ; Y++
    CPY #10       ; is Y 10?
    BNE Loop      ; if not, loop around
    RTS           ; we're done, result in A
```



Halbleiter-Layout




Transistor - und Gatterebene




Mikro-architektur


Instruction set architecture

# Abstraktionsebenen: Emulation auf Instruktionssatz-Ebene

- **Wir betrachten nur eine Instruktion: LDA #0**

```
int emulate_cpu(int clock_count) {
    clocks_remain = clock_count;
    int pc = cpu.pc;
    int a = cpu.a;
    ...
    while (clocks_remain > 0)    {
        int opcode = read_memory(pc);
        switch (opcode)          {
            case 0xA9: // LDA immediate
                a = read_memory(pc+1);
                set_nz(a);
                pc = pc+2;
                clocks_remain = clocks_remain-2;
                break;           }
    }
    cpu.pc = pc;
    cpu.a = a;
    return clocks_remain;
}
```

Hex-Opcodes des Beispielprogramms:
**a9 00 a8 18 71 70 c8 c0 0a d0 f8 60**

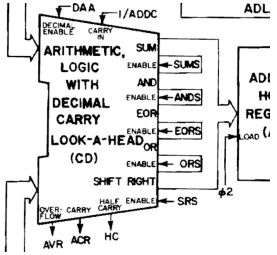
**Geänderter Zustand im Emulator:**
– Program Counter
– Akkumulator (A)
– Prozessorflag Zero (Z)
– Taktzyklenzähler (optional)

Instruction set architecture

# Abstraktionsebenen: Präzises Timing

```
#  addr  data rw Comment
 0 $0000  $a9   1  LDA #
 1 $0001  $00   1  #0
 2 $0002  $a8   1  TAY
 3 $0003  $18   1  (CLC)
 4 $0003  $18   1  CLC
 5 $0004  $71   1  (ADC)
 6 $0004  $71   1  ADC (zp),Y
 7 $0005  $70   1  $70
 8 $0070  $00   1  addrLo
 9 $0071  $00   1  addrHi
10 $0000  $a9   1  val at (addr)
11 $0006  $c8   1  INY
12 $0007  $c0   1  (CPY)
13 $0007  $c0   1  CPY #
14 $0008  $0a   1  #10
15 $0009  $d0   1  BNE
16 $000a  $f8   1  lp
17 $000b  $60   1  (RTS)
18 $0003  $18   1  CLC
```

- ISA-Ebene: Kein Timing, kein präzise simulierter Speicher…
- **Hier (für Checksummen-Programm):**
  - Einzelne Prozessorzyklen nachgebildet
  - *Halbzyklen* mit Instruktions-Prefetch
  - z.B. für präzise Emulation der Interaktion CPU-Videocontroller (C64) notwendig

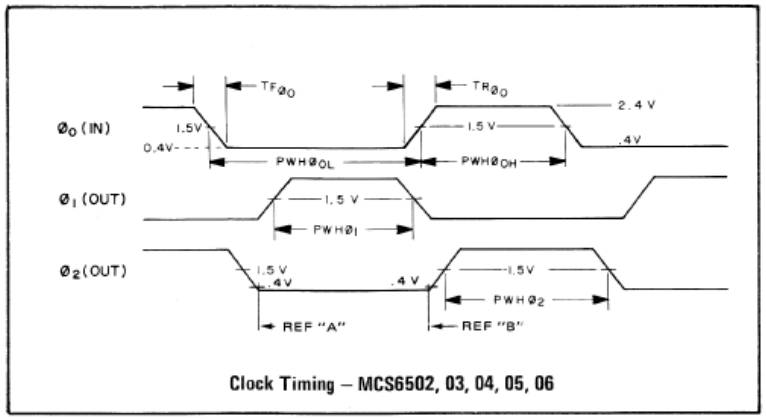



Mikro-architektur

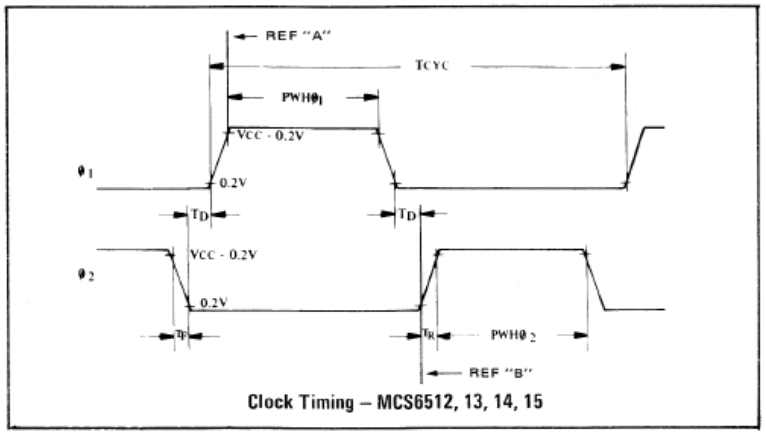
# Abstraktionsebenen: Präzises Timing (2)

- Timing-Diagramme aus Datenblatt der CPU: exakte Zeitparameter

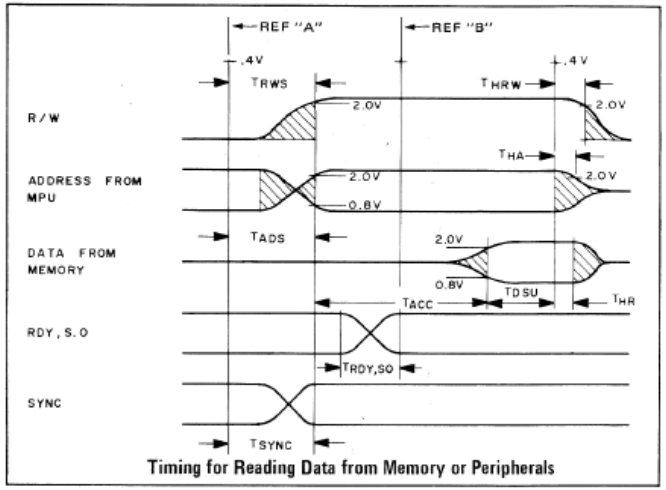
**COMMON CHARACTERISTICS**



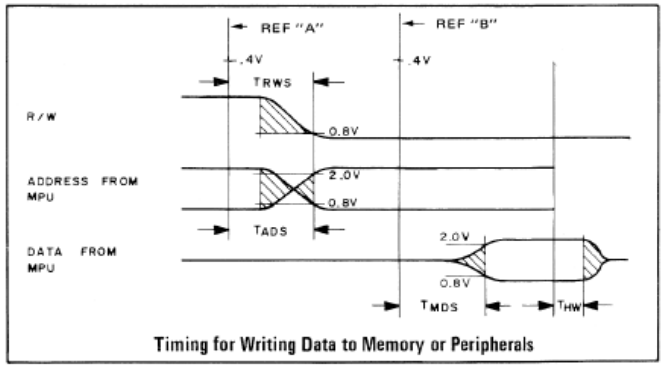

Clock Timing – MCS6502, 03, 04, 05, 06




Clock Timing – MCS6512, 13, 14, 15

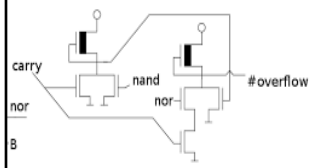


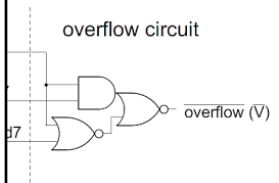

Timing for Reading Data from Memory or Peripherals




Timing for Writing Data to Memory or Peripherals

Note: "REF." means Reference Points on clocks.




Transistor - und Gatterebene

# Emulation: „black box“ gegen „white box“

- **Wie kann eine hochpräzise Emulation realisiert werden?**
- **Black box:** Innenleben der CPU ist nicht bekannt
- Rekonstruktion des Verhaltens allein auf Basis öffentlich zugänglicher Informationen
  - Datenblätter, Patente, ...
  - Ausführen von Software und Messen von Timings usw.
- **Vorteil:**
  - Juristisch schwer angreifbar (aber Patentrechte können betroffen sein)
- **Nachteil:**
  - Fehlerhafte oder unvollständige Datenblätter
  - Aufwändige Analysen erforderlich

# Beispiel „black box“-6502: ag_6502-Core

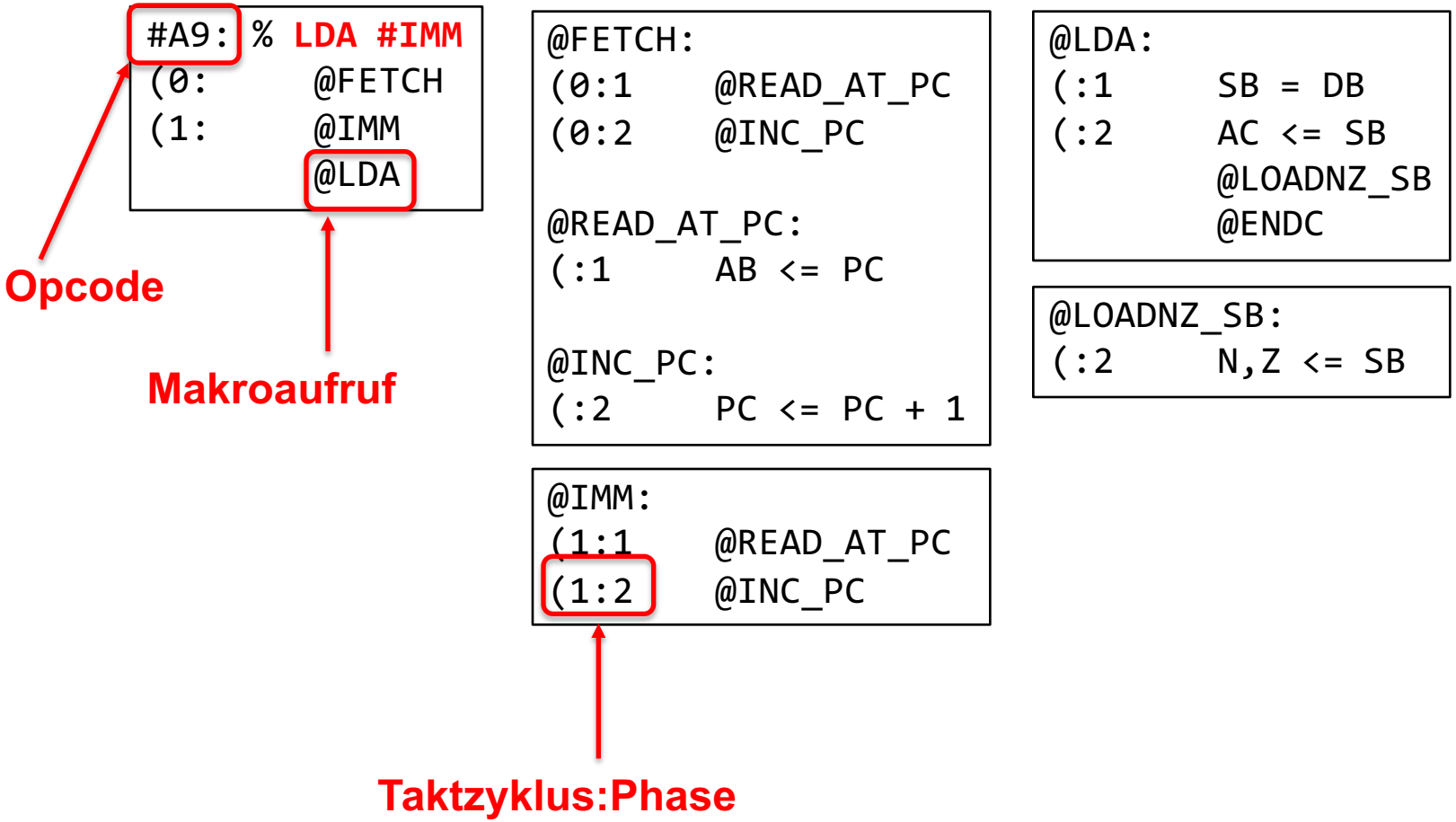
- **Zyklengenaue Hardware-Implementierung des 6502**
  - http://opencores.org/project,ag_6502
- Beschreibungssprache für Timingverhalten und Semantik des 6502, wird in Verilog-Code übersetzt
- Basisstruktur (Datenpfad) der CPU (Register, ALU usw.) von Hand in Verilog modelliert

- Timing-Angaben aus Datenblatt (hier vom 65C816)

| Address Mode | Note | Cycle | VPB | MLB | VDA (14) | VPA (14) | Address Bus (15) | Data Bus | RWB |
|---|---|---|---|---|---|---|---|---|---|
| 18.Immediate #<br>ADC, AND, BIT, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, REP, SEC, SEP<br>14 OpCodes, 2 and 3 bytes, 2 and 3 cycles | (1)(6) | 1<br>2<br>2a | 1<br>1<br>1 | 1<br>1<br>1 | 1<br>0<br>0 | 1<br>1<br>1 | PBR,PC<br>PBR,PC+1<br>PBR,PC+2 | OpCode<br>IDL<br>IDH | 1<br>1<br>1 |

# Beispiel „black box“-6502: ag_6502-Core (2)

- **Implementierung von „LDA immediate“ im ag_6502:**



# Beispiel „black box“-6502: ag_6502-Core (3)

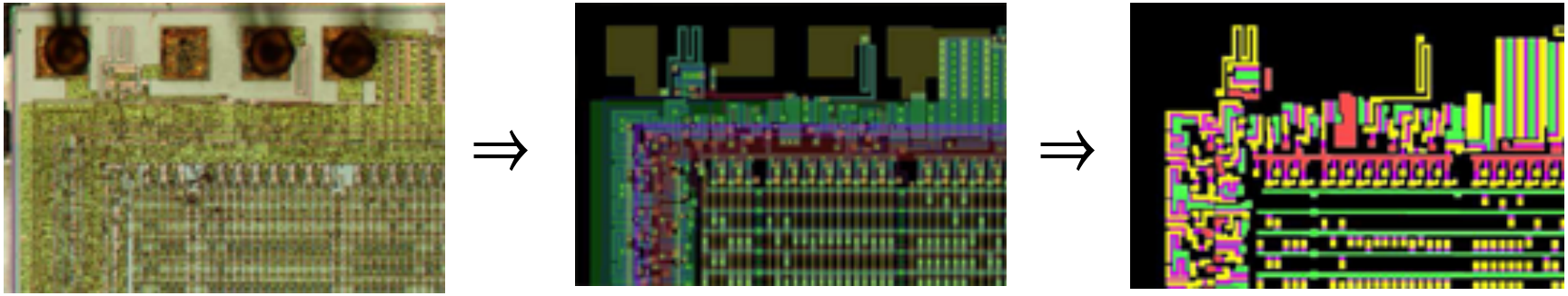
- **Übersetzung der Beschreibung in Schaltung auf Gatterebene**
- Beispiel Steuersignal „Byte von Datenbus an ALU leiten“:

```
//    action: DB <= ALU:
assign E_DB__ALU =
   ({L[0],L[1],L[2],L[3],L[4],L[6],L[7],L[8],L[9],L[10]} == 10'b0110011110)
|| (({L[0],L[1],L[2],L[6],L[7],L[10]} == 6'b011111)
    && ((!L[9] && ((!L[3] && (({L[4],L[8]} == 2'b00) || L[4]))
|| (L[3] && (({L[4],L[8]} == 2'b00) || L[8]))))
|| ({L[3],L[4],L[8],L[9]} == 4'b1101)));
```

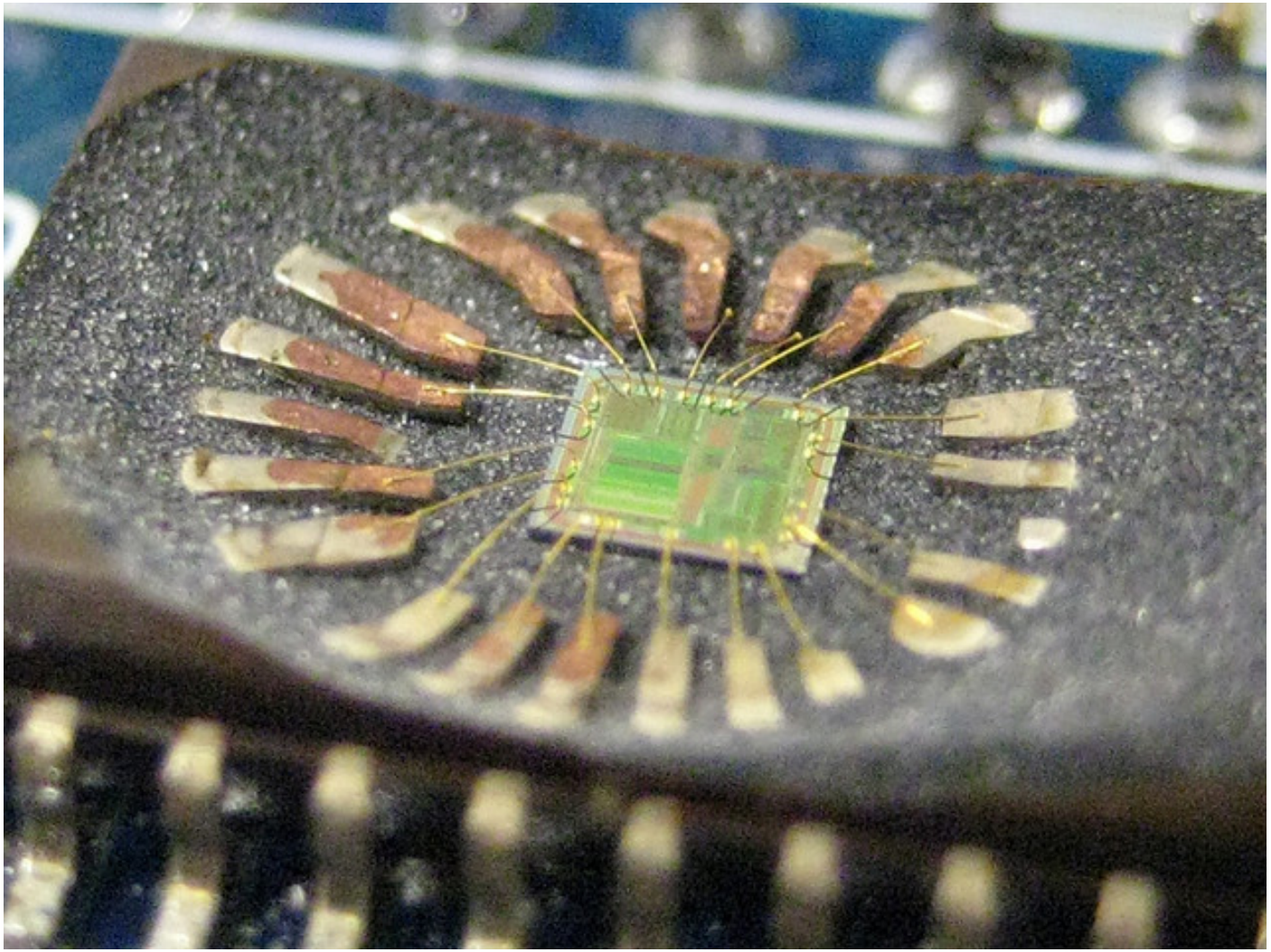
**Logikformel für ein einzelnes Steuerbit!**

# White box: Reverse Engineering

- Wenn keine Konstruktionsunterlagen vorliegen:
  - Analyse der Strukturen des realen CPU-Chips
  - Übliche Vorgehensweise der DDR-Halbleiterentwicklung… ☺
- **Methode:**
  - „Decapping“ des Chips (mit Säure)
  - Die-Mikroskopaufnahme ⇝ Rekonstruktion Transistormodell
  - Schwierig für aktuelle Chips wegen Strukturbreiten < 100nm
  - Machbar für den 6502 (3 µm Strukturbreiten)



# Decapping

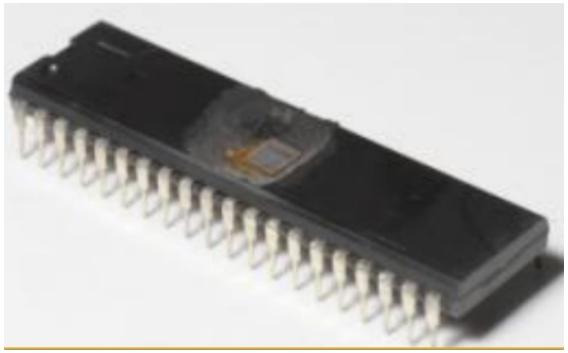


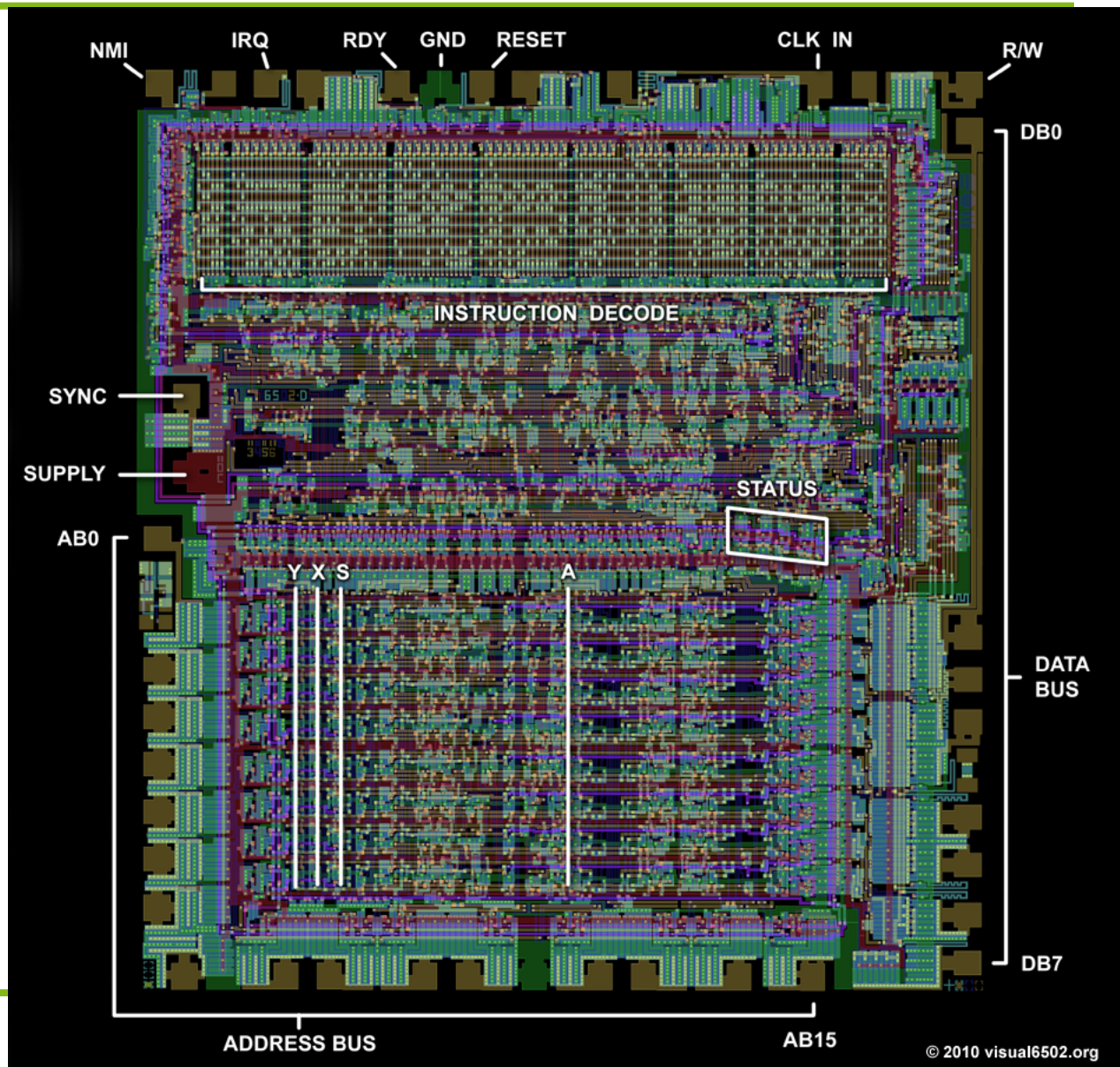
# Layout-Modell des 6502

- „visual6502“: Modell auf Transistorebene
- NMOS-Transistoren
- Netzliste
- Semantische Annotation

G. James, B. Silverman, B.Silverman:
***Visualizing a Classic CPU in Action:***

***The 6502***
Proc. of SIGGRAPH 2010



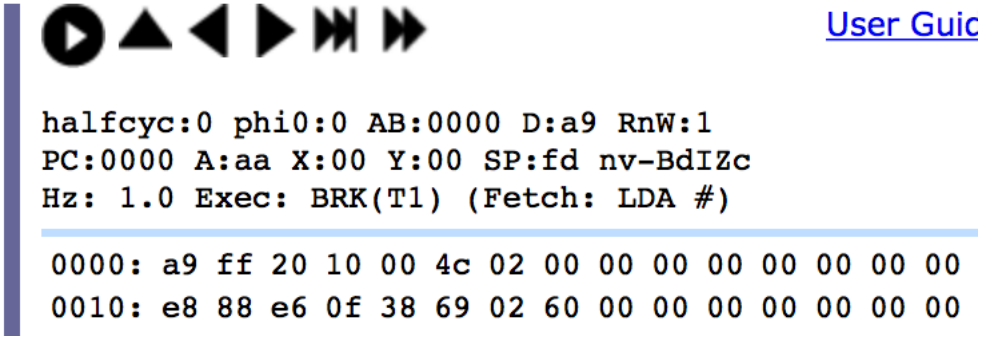


# Demo Visual 6502

- ...mit unserem Programm zur Prüfsummenberechnung

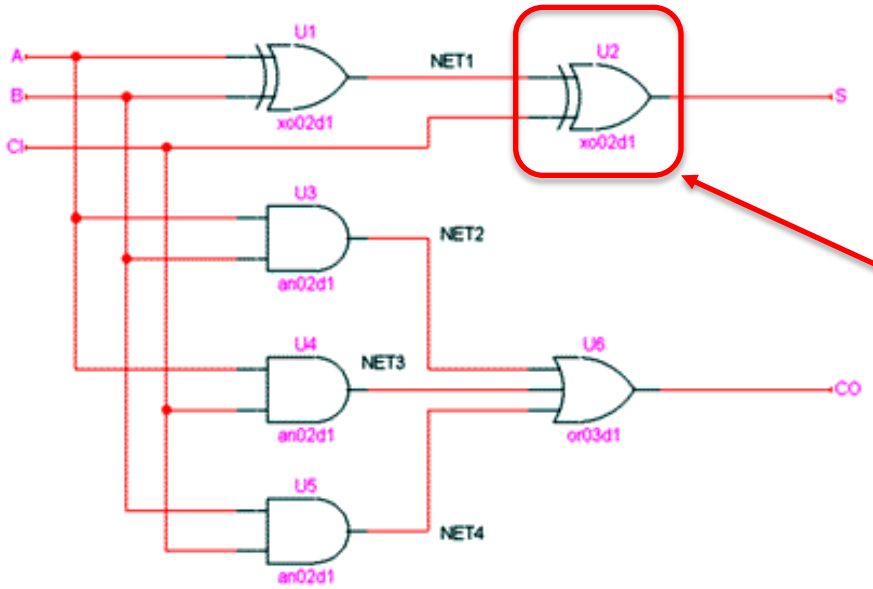
Hex-Opcodes des Beispielprogramms:
**a9 00 a8 18 71 70 c8 c0 0a d0 f8 60**

- Eingabe der Hexcodes im „advanced mode“ möglich:



# Basis des visual 6502: Reverse Engineering der Netzliste

- Netzliste = Liste von Tupeln, die Verbindungen zwischen Komponenten angeben („Leitungen“)
- Beispiel für Volladdierer auf Gatterebene:

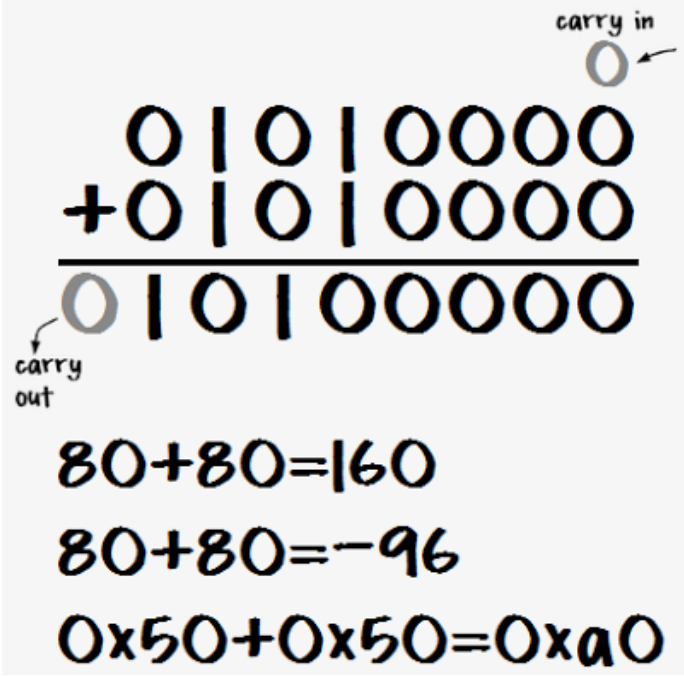



```
// full_adder Module
// Verilog Netlist Level Description
// (using "generic_library" cells)
module full_adder (a, b, ci, s, co);
    input a, b, ci;
    output s, co;
    wire NET1, NET2, NET3, NET4;

    xo02d1 U1( NET1, a, b );
    xo02d1 U2( s , NET1, ci );
    an02d1 U3( NET2, a, b );
    an02d1 U4( NET3, a, ci );
    an02d1 U5( NET4, b, ci );
    or03d1 U6( co, NET2, NET3, NET4 );
endmodule
```

# Beispiel: von der Arithmetik zu Transistor-Netzliste

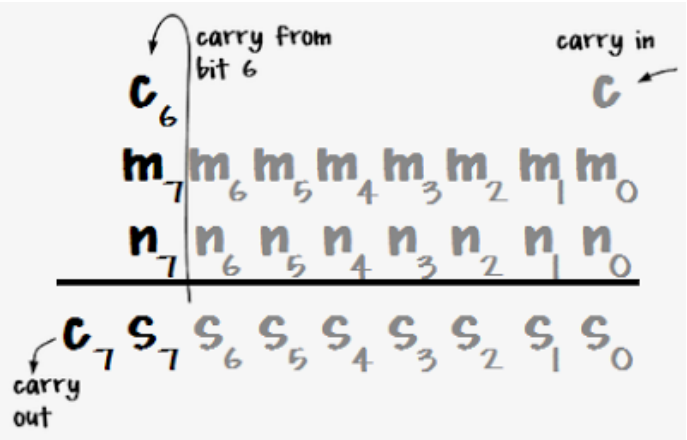
- Beispiel: „Overflow“-Flag (V) des 6502
- V wird gesetzt, wenn Ergebnis einer *vorzeichenbehafteten* Addition oder Subtraktion nicht in den Akkumulator (8 Bit) passt
- Beispiel:
  - 80+80=160
  - Passt in 8 Bit, kein Carry
- **Aber:**
  - MSB (Bit 7) ist gesetzt
  - **Ergebnis kann falsch als vorzeichenbehaftete Zahl –96 (Zweierkomplement) interpretiert werden!**
- **V-Flag zeigt diesen Zustand an**




http://www.righto.com/2012/12/the-6502-overflow-flag-explained.html

# Beispiel: von der Arithmetik zu Transistor-Netzliste (2)

- Verschiedene Zustände können V setzen:




| Inputs | | | Outputs | | | | Example | | |
|---|---|---|---|---|---|---|---|---|---|
| M7 | N7 | C6 | C7 | S7 | V | Carry / Overflow | Hex | Unsigned | Signed |
| 0 | 0 | 0 | 0 | 0 | 0 | No unsigned carry or signed overflow | 0x50+0x10=0x60 | 80+16=96 | 80+16=96 |
| 0 | 0 | 1 | 0 | 1 | 1 | No unsigned carry but signed overflow | 0x50+0x50=0xa0 | 80+80=160 | 80+80=-96 |
| 0 | 1 | 0 | 0 | 1 | 0 | No unsigned carry or signed overflow | 0x50+0x90=0xe0 | 80+144=224 | 80+-112=-32 |
| 0 | 1 | 1 | 1 | 0 | 0 | Unsigned carry, but no signed overflow | 0x50+0xd0=0x120 | 80+208=288 | 80+-48=32 |
| 1 | 0 | 0 | 0 | 1 | 0 | No unsigned carry or signed overflow | 0xd0+0x10=0xe0 | 208+16=224 | -48+16=-32 |
| 1 | 0 | 1 | 1 | 0 | 0 | Unsigned carry but no signed overflow | 0xd0+0x50=0x120 | 208+80=288 | -48+80=32 |
| 1 | 1 | 0 | 1 | 0 | 1 | Unsigned carry and signed overflow | 0xd0+0x90=0x160 | 208+144=352 | -48+-112=96 |
| 1 | 1 | 1 | 1 | 1 | 0 | Unsigned carry, but no signed overflow | 0xd0+0xd0=0x1a0 | 208+208=416 | -48+-48=-96 |

# Beispiel: von der Arithmetik zu Transistor-Netzliste (3)

- Logikrepräsentation der Erzeugung des V-Bits (rote Zeilen):

**Einfach:**

```
V = C6 xor C7
```

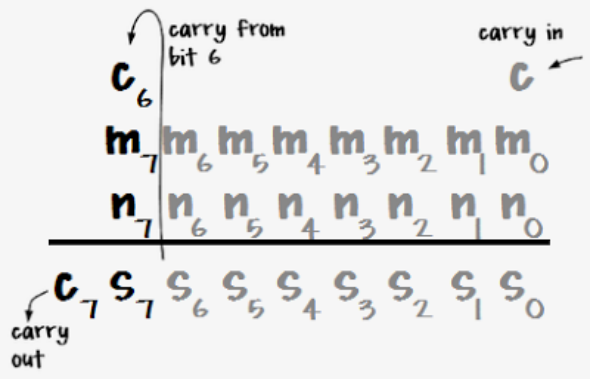
**Ohne XOR:**

```
V = (!M7 & !N7 & C6) | (M7 & N7 & !C6)
```

**Implementierung im 6502 (de Morgan-Regeln):**

$$V = \text{not } (((M_7 \text{ nor } N_7) \text{ and } C_6) \text{ nor } ((M_7 \text{ nand } N_7) \text{ nor } C_6))$$

| Inputs | | | Outputs | | | | Example | | |
|---|---|---|---|---|---|---|---|---|---|
| M7 | N7 | C6 | C7 | S7 | V | Carry / Overflow | Hex | Unsigned | Signed |
| 0 | 0 | 0 | 0 | 0 | 0 | No unsigned carry or signed overflow | 0x50+0x10=0x60 | 80+16=96 | 80+16=96 |
| 0 | 0 | 1 | 0 | 1 | 1 | No unsigned carry but signed overflow | 0x50+0x50=0xa0 | 80+80=160 | 80+80=-96 |
| 0 | 1 | 0 | 0 | 1 | 0 | No unsigned carry or signed overflow | 0x50+0x90=0xe0 | 80+144=224 | 80+-112=-32 |
| 0 | 1 | 1 | 1 | 0 | 0 | Unsigned carry, but no signed overflow | 0x50+0xd0=0x120 | 80+208=288 | 80+-48=32 |
| 1 | 0 | 0 | 0 | 1 | 0 | No unsigned carry or signed overflow | 0xd0+0x10=0xe0 | 208+16=224 | -48+16=-32 |
| 1 | 0 | 1 | 1 | 0 | 0 | Unsigned carry but no signed overflow | 0xd0+0x50=0x120 | 208+80=288 | -48+80=32 |
| 1 | 1 | 0 | 1 | 0 | 1 | Unsigned carry and signed overflow | 0xd0+0x90=0x160 | 208+144=352 | -48+-112=96 |
| 1 | 1 | 1 | 1 | 1 | 0 | Unsigned carry, but no signed overflow | 0xd0+0xd0=0x1a0 | 208+208=416 | -48+-48=-96 |



# Beispiel: von der Arithmetik zu Transistor-Netzliste (4)

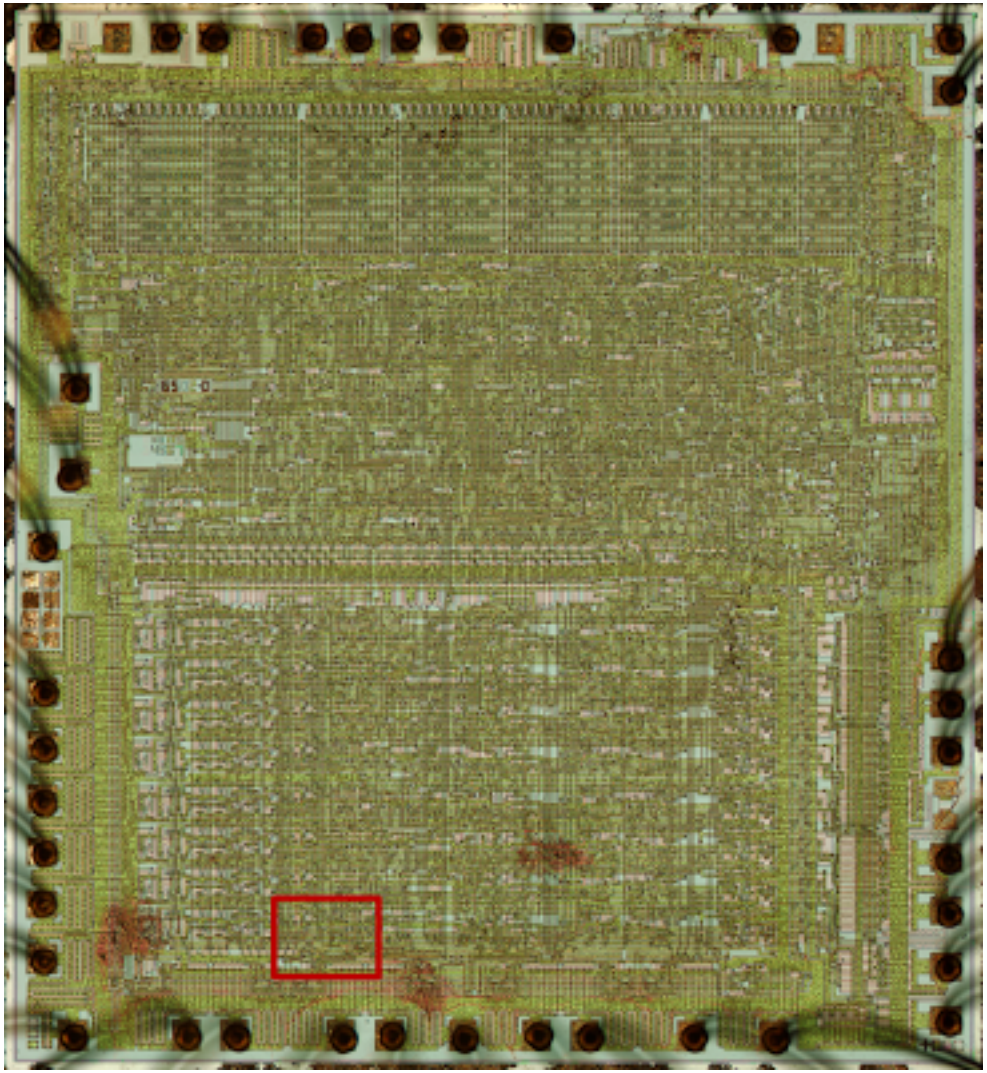
**Implementierung im 6502:**

```
V = not (((M7 nor N7) and C6)
    nor ((M7 nand N7) nor C6))
```

**Reverse Engineering:**

Rot markierter Bereich auf dem Foto des 6502-Die

**…was steckt da drin?**



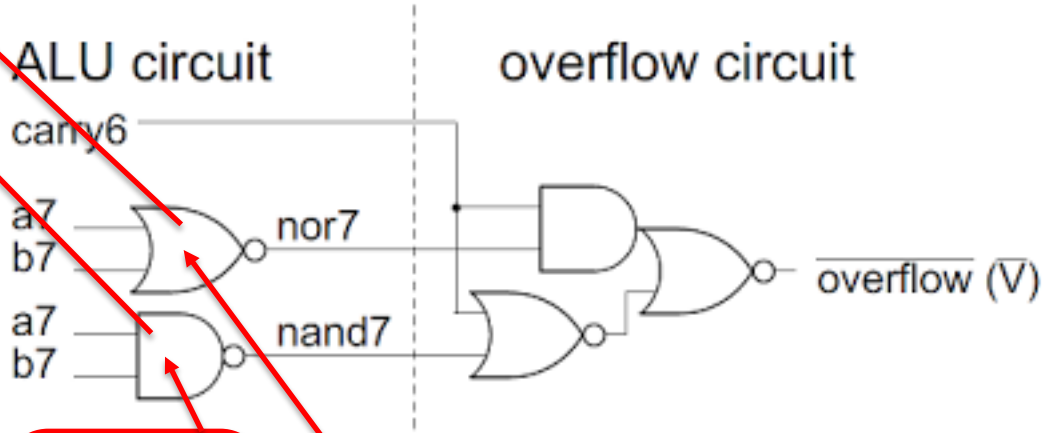
http://www.righto.com/2013/01/a-small-part-of-6502-chip-explained.html

# Beispiel: von der Arithmetik zu Transistor-Netzliste (5)

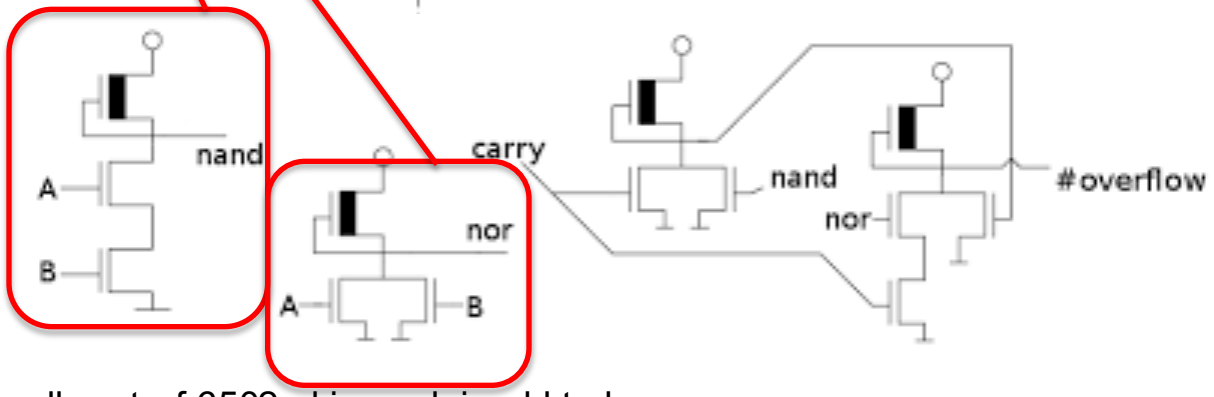
**Implementierung im 6502:**

```
V = not (((M7 nor N7) and C6)
    nor ((M7 nand N7) nor C6))
```

**In Gatterdarstellung:**




**Als Transistorschaltung:**

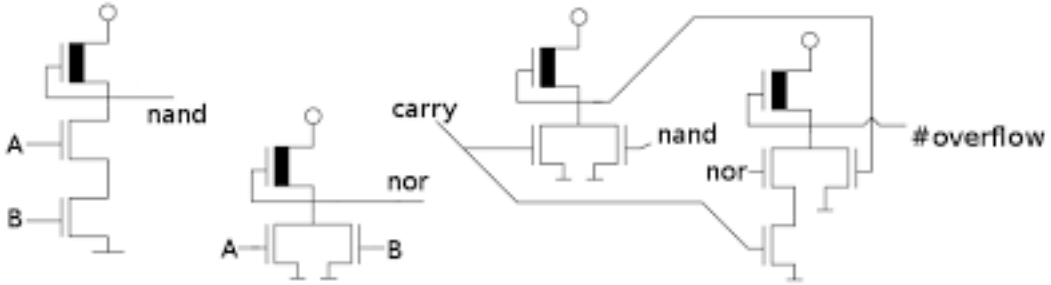



http://www.righto.com/2013/01/a-small-part-of-6502-chip-explained.html

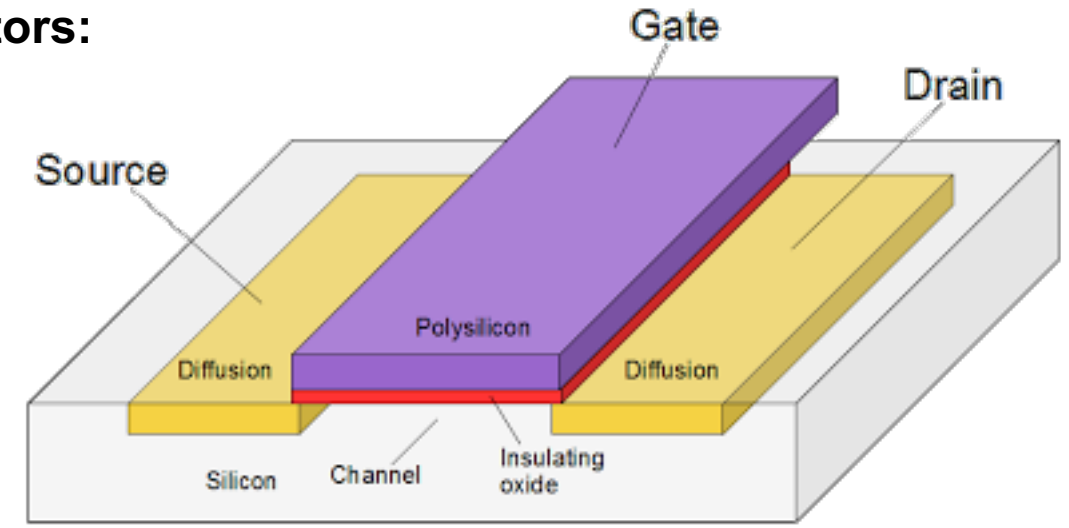
# Beispiel:
# von der Arithmetik zu Transistor-Netzliste (6)

**Implementierung im 6502:**

```
V = not (((M7 nor N7) and C6)
    nor ((M7 nand N7) nor C6))
```

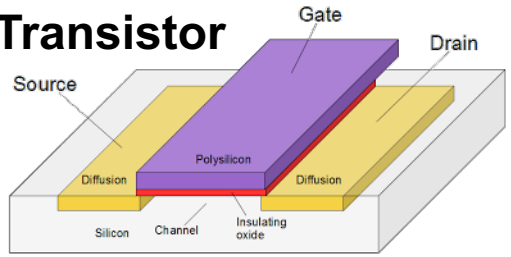


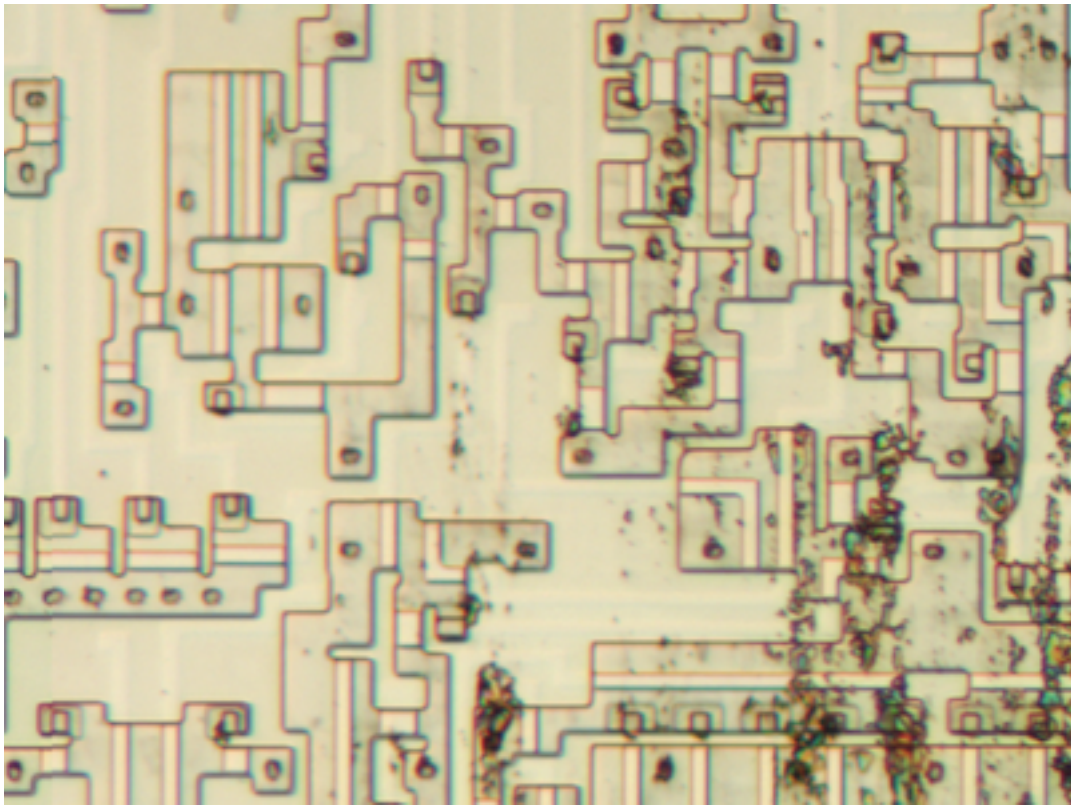

**Aufbau eines MOS-Transistors:**



# Beispiel:
# von der Arithmetik zu Transistor-Netzliste (7)

**MOS-Transistor**



**Die-Foto:**

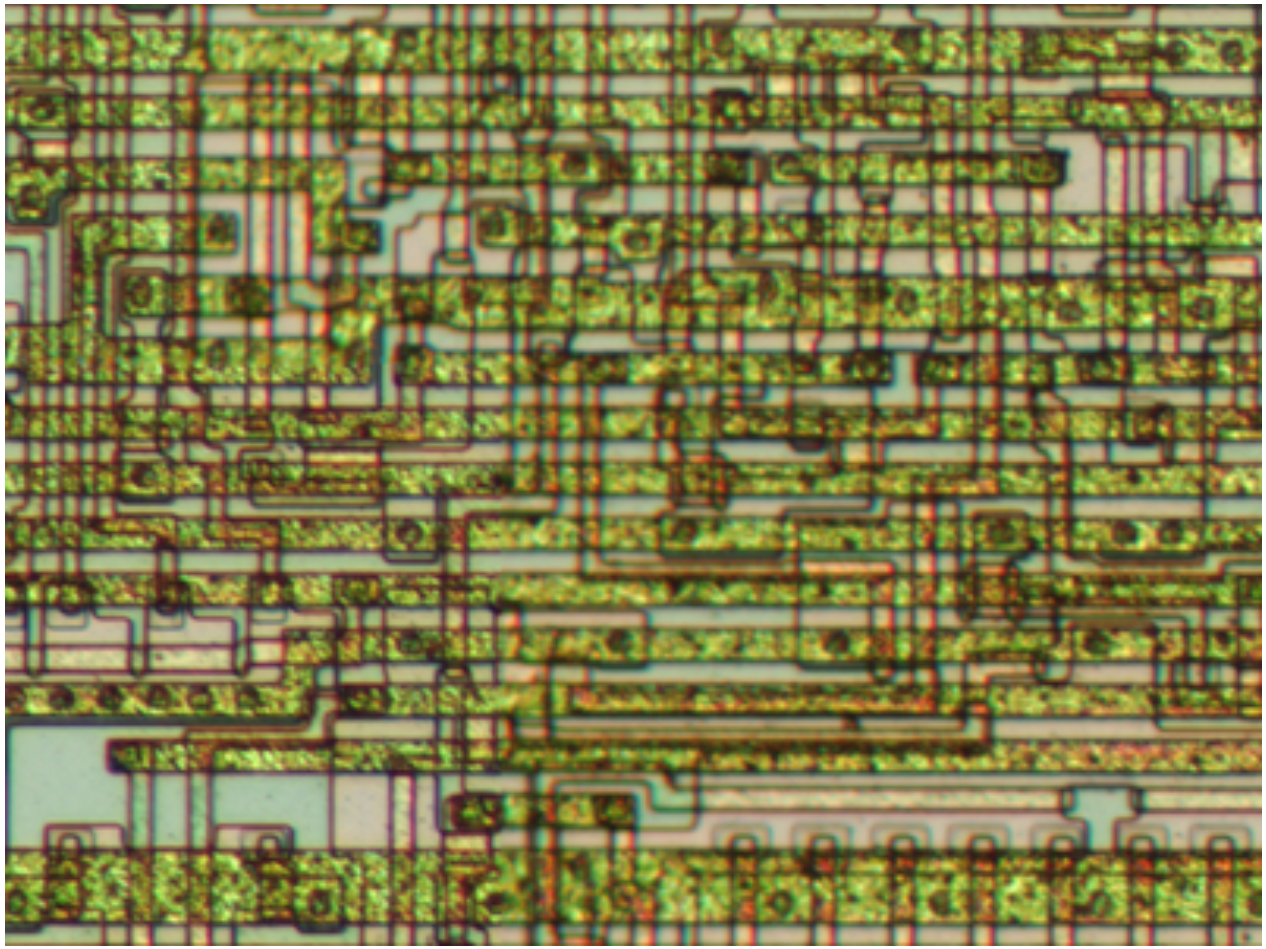
– dunkler graue Regionen:
n+ diffusion areas, leitend (dopiert)

– weiße Streifen:
Trennen n+ Regionen, sind die Gates der Transistoren

– graue Quadrate:
Vias, verbinden vertikal zu anderen Ebenen des Chips

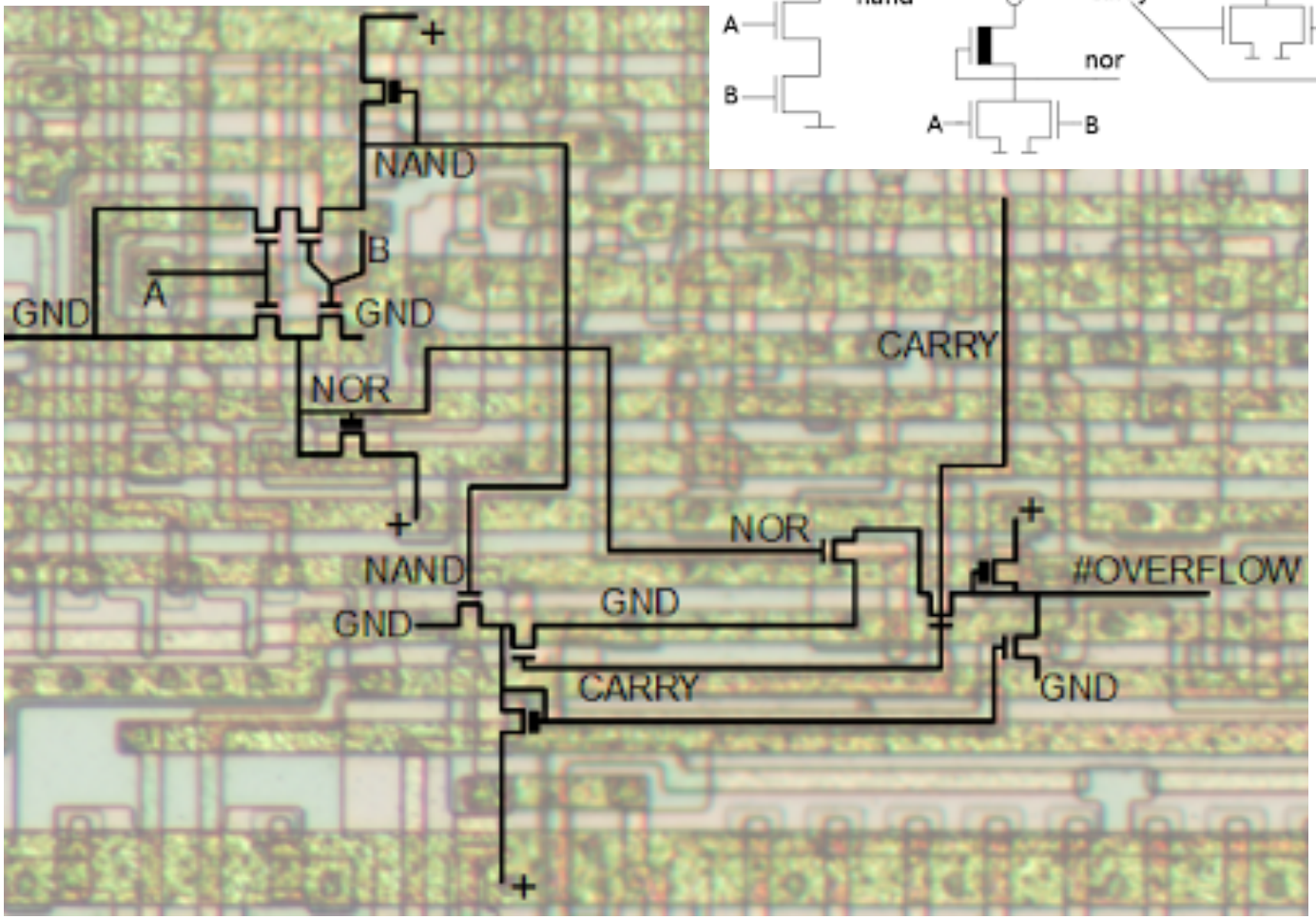
# Beispiel:
# von der Arithmetik zu Transistor-Netzliste (8)

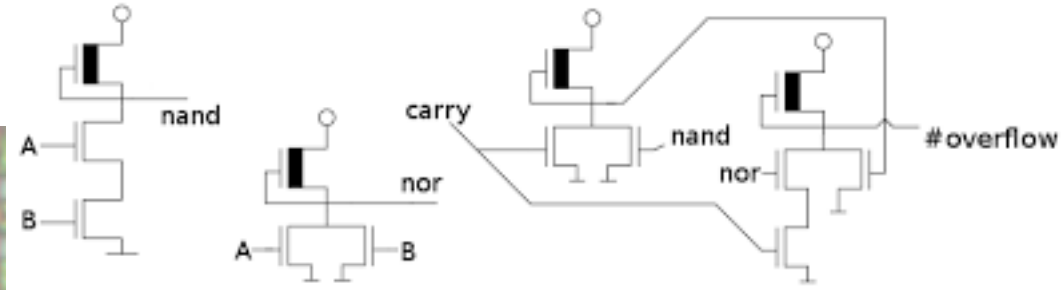
**Die-Foto mit allen Layern:** grüne Streifen = Metall-Layer, “Leiterbahnen”



# Beispiel:
# von der Arithmetik zu Transistor-Netzliste (9)

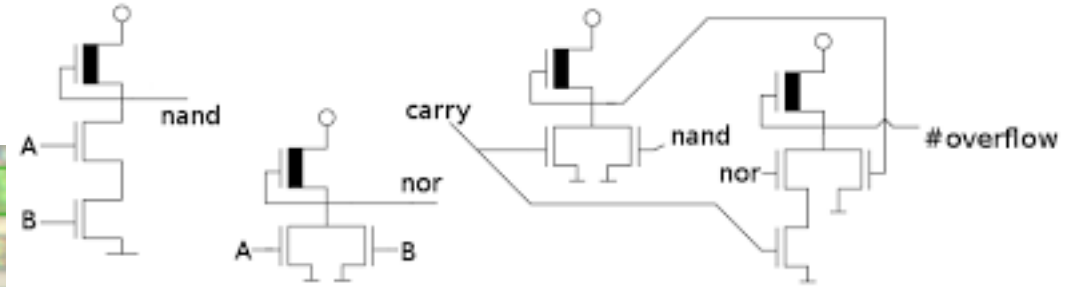
**Annotiertes Die-Foto**:

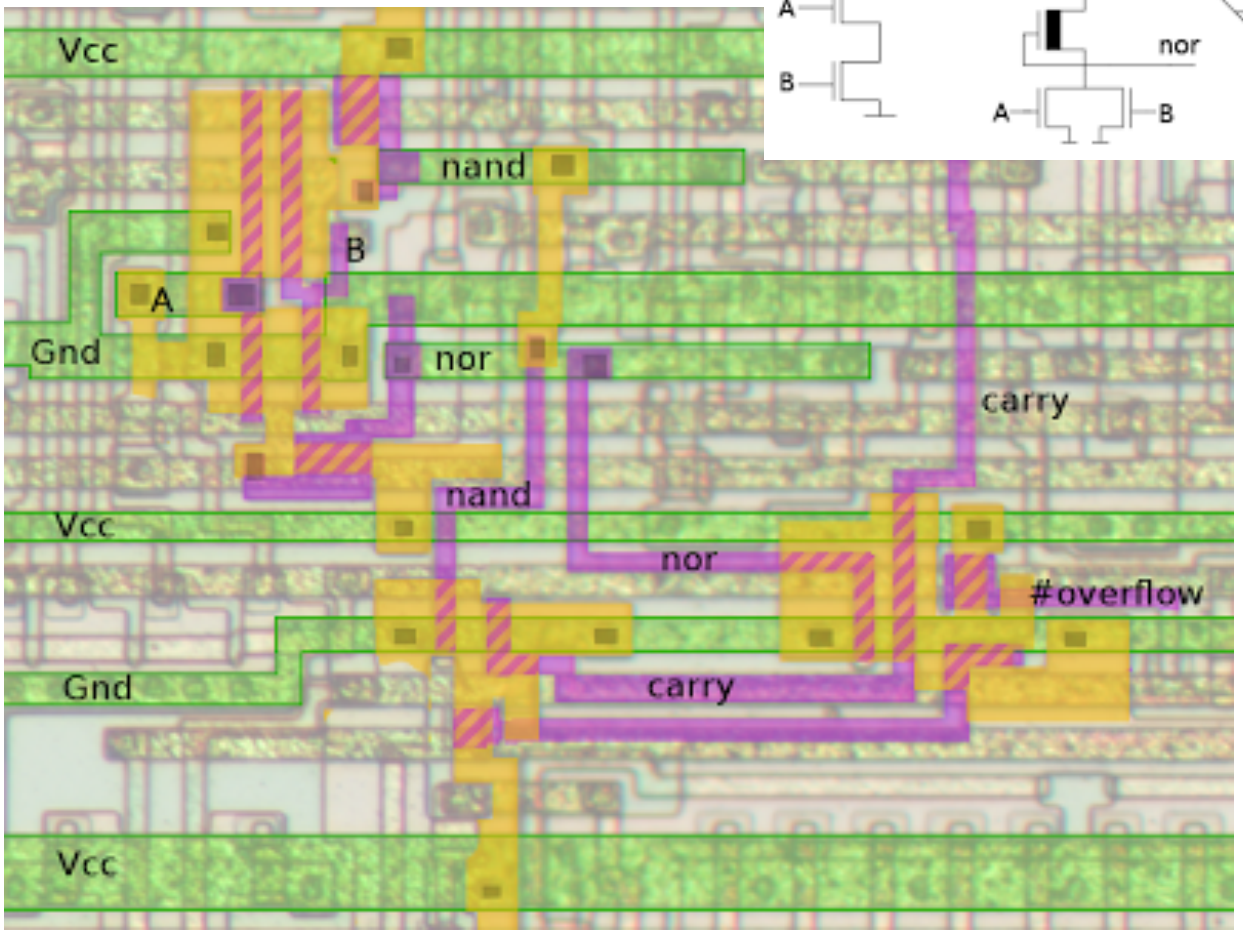


# Beispiel:
# von der Arithmetik zu Transistor-Netzliste (10)

**Übersichtlicher (Rechteck-Diagramm):**




**Farbschema im visual6502 ist anders:**

– Diffusion: grün, gelb, orange und rot
– Metal layer ist weißlich
– Vcc und GND fehlen
– Polysilizium ist violett
– Transistoren sind nicht explizit dargestellt

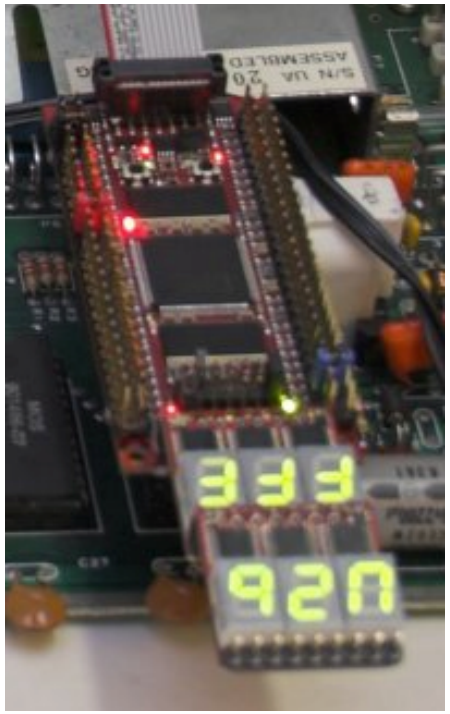
# Hardware-Entwurf in den 1970er-Jahren

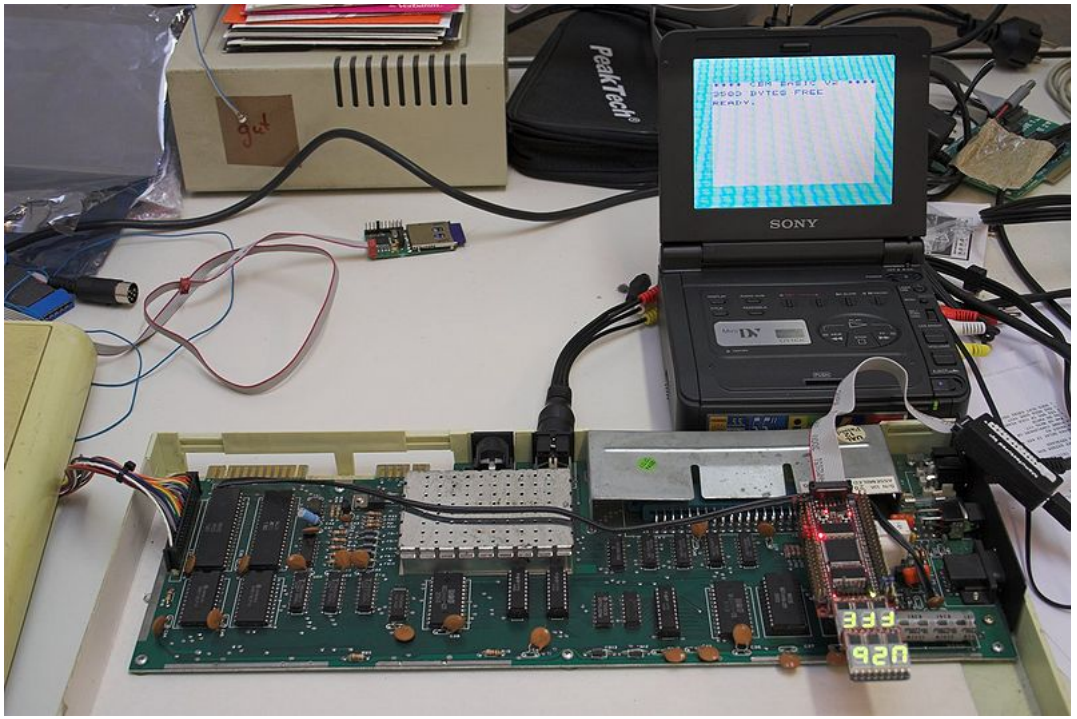
- Transistorschaltpläne (Rechtecke) von Hand auf Folie (Rubylith) gezeichnet und fotografisch verkleinert
- Damit Belichtung des Chips, Entwicklung durch fotografische Verfahren → Masken zum Dotieren des Siliziums

Das Entwickler-Team des 6502

# Von der Netzliste zurück zur Hardware

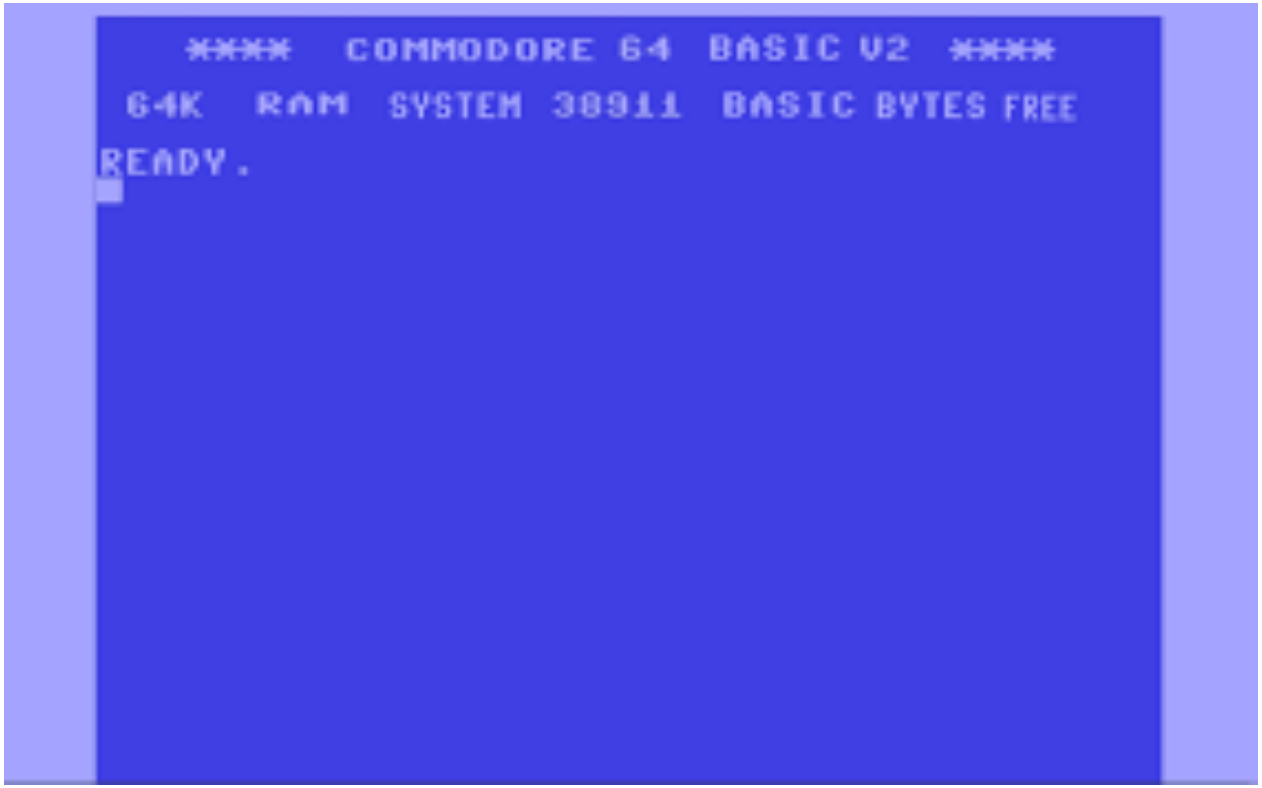
- Transistormodell-Netzliste: von Javascript nach...
  - C-Simulation „perfect6502“ (Michael Steil)
  - Verilog ⇝ FPGA-Implementierung (Xilinx 3S500E)
  - Verwendung in echten Rechnern: C64, VC20, Apple II





# Demo perfect6502: Commodore BASIC

- Ausführung des C64-BASIC-Interpreters mit perfect6502
- ...als Prozess in Unix!

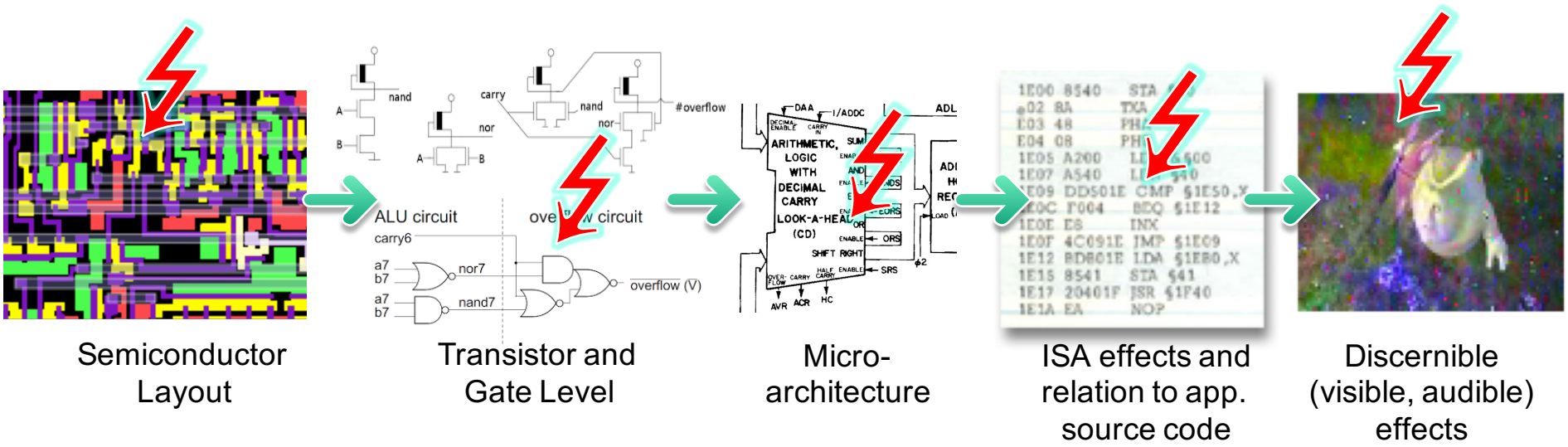


# Systemsimulation des 6502

- **Open Source-Implementierungen: Zusammenarbeit vieler Autoren**
- **Die wichtigsten URLs zum Weiterlesen:**
- http://visual6502.org – Javascript-Simulation und viel mehr...
- https://github.com/mist64/perfect6502 – C-Realisierung
- https://github.com/pmonta/FPGA-netlist-tools
  Peter Monta‘s netlist tools
- https://github.com/ikorb/FPGA-netlist-tools
  Ingo Korbs erweiterte Version
- http://visual6502.org/wiki/index.php?title=6502_-_simulating_in_real_time_on_an_FPGA – Überblick mit mehr Fotos
- https://media.ccc.de/v/27c3-4159-en-reverse_engineering_mos_6502
  – Michael Steils Vortrag auf dem 27C3 zum Reverse Engineering
- http://www.righto.com/2013/01/a-small-part-of-6502-chip-explained.html
  – Ken Shirriffs Blog zu Reverse Engineering
- http://www.downloads.reactivemicro.com/Public/Electronics/CPU/6502%20Schematic.pdf – Transistor-Level 6502-Schaltpläne

# Was machen wir damit?

- **Forschung zu Fehlertoleranz:**
  - Untersuchung der Auswirkungen von Fehlern auf Hardwareebene auf die Programmausführung
  - Simulation in Echtzeit auf realem System möglich
  - Untersuchung der Fehlerfortpflanzung von Layout bis zu den Auswirkungen auf die Anwendung




Semiconductor Layout | Transistor and Gate Level | Micro-architecture | ISA effects and relation to app. source code | Discernible (visible, audible) effects

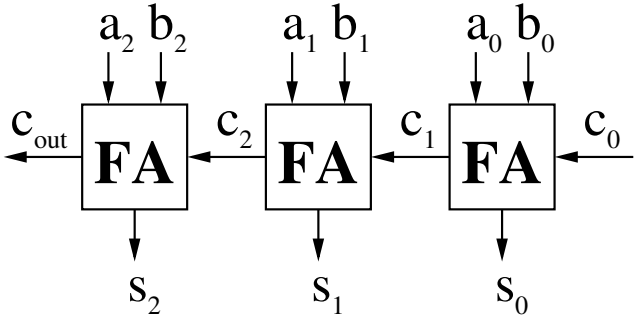
**Schmoll, Heinig, Marwedel, Engel.** ***Improving the Fault Resilience of an H.264 Decoder using Static Analysis Methods.*** **ACM TECS, 2013**

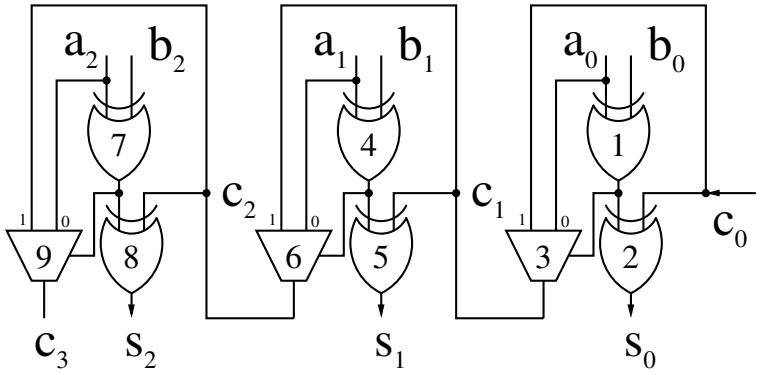
# Ausblick: Probabilistische Systeme oder „was machen wir noch damit?“

Energieeinsparung durch selektive, feingranulare Reduktion der Versorgungsspannung auf Bit-Ebene

- Generelle Reduktion der Versorgungsspannung erhöht Fehlerwahrscheinlichkeit für alle Bits gleichermaßen
- Selektive Reduktion erlaubt **kontrollierte** Erhöhung der Fehlerwahrscheinlichkeit, z.B. bei niederwertigen Bits
- Kooperation mit Nanyang Technological University Singapur
  - Prof. Vincent J. Mooney III
  - Probabilistische CMOS-Schaltungen und –Systeme
  - Fehlermodelle für probabilistische Logik

# Verzögerungen bei Ripple-Carry-Addierer

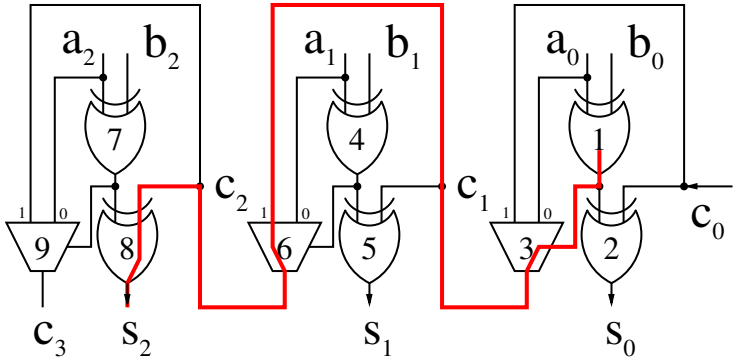



Gatterverzögerungen für
90 nm-Technologie, 1,2 V:

| Gatter | Verzögerung |
|---|---|
| XOR | 33,3 ps |
| MUX | 30,5 ps |

Worst Case: 127,6 ps
=> 7,8 GHz max. Taktfrequenz




Worst-Case-Verzögerung

# Probabilistischer Ripple-Carry-Addierer

Energieverbrauch proportional zu Quadrat der Versorgunsspannung

XOR-Gatterverzögerungen
für 90 nm-Technologie:

| Spannung [V] | Energie [pJ] | Verzögerung [ps] |
|---|---|---|
| 0,8 | 8,63 | 46,9 |
| 1,0 | 14,30 | 37,9 |
| 1,2 | 21,65 | 33,3 |

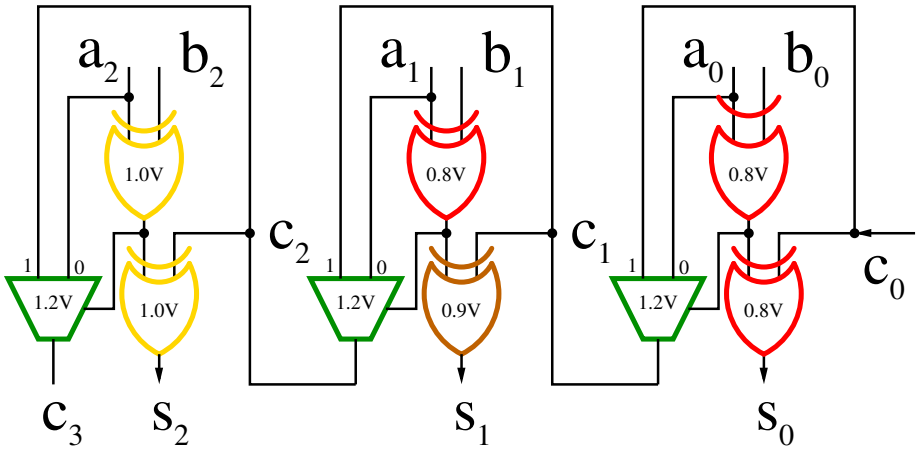
**Energieeinsparung:**

⇒ Maximale Taktfrequenz sinkt

⇒ Worst-Case-Verzögerung zu lang für ursprüngliche Frequenz

Verwendung von
**Biased Voltage Scaling** [xxx]:

Selektive Kontrolle der Versorgungs-Spannung minimiert Wahrscheinlichkeit des Überschreitens der Worst-Case-Verzögerung:



# Der berühmteste 6502-Benutzer?

# Danke...