Logik / Gates

AND ⇒ A & B | NAND ⇒ ~(A & B)

OR ⇒ A | B | NOR ⇒ ~(A | B)

NOT → ~A | XOR ⇒ A ^ B

Dezimal

Dez. → Bin.:

5₁₀ : 2 = 2 R 1 ↑
2 : 2 = 1 R 0
1 : 2 = 0 R 1
⇒ 5₁₀ = 101₂

Bin. → Dez.:

$101_2 = 1\cdot2^2 + 0\cdot2^1 + 1\cdot2^0 = 5_{10}$

1011 + 11 = 1110	1011 − 110 = 101
	−1

$2^8 = 256$
$2^{16} = 65'536$
$2^{32} = 4'294'967'296$

Sign / Magnitude: 1. Bit Vorzeichen, $[-2^{n-1}+1, +2^{n-1}-1]$

Hexadezimal

0, 1, 2, …, 9, A, B, C, D, E, F
(10) (11) (12) (13) (14) (15)

Dez. → Hex.:

1278 : 16 = 79 R 14 = E
79 : 16 = 4 R 15 = F
4 : 16 = 0 R 4 = 4
⇒ $1278_{10} = 4FE_{16}$

Hex. → Dez.:

$4FE_{16} = 4\cdot16^2 + 15\cdot16^1 + 14\cdot16^0 = 1278_{10}$

2s Complement: $[-2^{n-1}, 2^{n-1}-1]$

1. Bit (MSB) ist Indikator für negative Zahl.

Dez. → 2s Comp.:
1) absoluten Wert in Bin. umwandeln
2) wenn negative Zahl:
(a) invertieren (b) 1_{10} addieren

2s Comp. → Dez.
ganz normal, nur dass 1. Bit den Wert -2^n hat!

Bei Addition kann ein zusätzliches Bit entstehen → nicht zwingend Overflow!
Overflow detection: • Addition positiver Zahlen gibt negative (MSB = 1 statt 0), • Addition negativer Zahlen gibt positive Zahl (MSB = 0 statt 1)

CMOS

nMOS: S=1; A=0 ⇒ d=0; A=1 ⇒ d=1
pMOS: S=1; A=0 ⇒ d=1; A=1 ⇒ d=0

Ground (= 0)

⊢, ⏚, ⊤ Spannungspegel (= 1)

NOT: A, Y

NAND: A, B, Y

MHz Umrechnungen

MHz = 1/s, kHz = 1/ms, GHz = 1/ns, MHz = 1/ms

Bsps: $T_c = 50\,ns \Rightarrow 1/50ns = 20\,MHz = 1/50\,GHz$
$T_c = 5\,ns \Rightarrow 1/5ns = 1/5\,GHz = 0{,}2\,GHz$

Boolean Equations

Minterm / Sum-of-Products

A	B	Y	SOP
0	0	0	$\bar{A}\bar{B}$
0	1	1	$\bar{A}B$
1	0	0	$A\bar{B}$
1	1	1	AB

⇒ $\bar{A}B + AB$

Maxterm / Product-of-Sums

A	B	Y	POS
0	0	0	$A+B$
0	1	1	$A+\bar{B}$
1	0	0	$\bar{A}+B$
1	1	1	$\bar{A}+\bar{B}$

⇒ $(A+B)\cdot(\bar{A}+B)$

X / Z: X: egal / unbekannt (1 oder 0)

Z: floating (irgendwo zwischen 1 und 0)

K-Map

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

immer nur 1 Bit ändert sich!

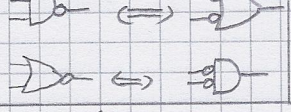
C \ AB	00	01	11	10
0	1	0	1	1
1	1	0	0	1

→ $A\bar{C}$, $\bar{B}$

⇒ $A\bar{C} + \bar{B}$

- Ziel: minimale # Quadrate
- im Quadrat sind nur 1'en und X'en erlaubt
- Jedes Quadrat hat 2^n 1'en / X'en (→ 1, 2, 4, …)
- 1'en und x'en dürfen in mehreren Quadraten sein
- Bis 4 Variabeln benutzbar

Bubble-Pushing



Multiplexer:

A, B, C, D → 00, 01, 10, 11 → I

I: Input, wird gegen 00, 01, 10, 11 geprüft

Delays:

t_{pd}: Propagation delay. Zeit vom Moment der Input-veränderung bis zum annehmen des finalen Output Wertes. Critical Path dafür nehmen

t_{cd}: Contamination delay. Zeit vom Moment der Input veränderung bis zur ersten Änderung irgendeines Outputs. Kürzesten Pfad dafür nehmen!

SR-Latch

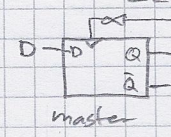
R, S, Q, $\bar{Q}$

R=1 ⇒ Q=0 (Reset); S=1 ⇒ Q=1 (set)

R	S	Q
0	0	Qvorher
0	1	1
1	X	0

D FLip-Flop

Kopiert D nach Q beim Ansteigen der CLK



Shifters

neu

Logic shift: 1100 >>2 = 0011, 1100 <<2 = 0000

Arithmetic shift:

1100 1 >>2 = 1111 0

0100 1 >>2 = 0001 0

1100 1 <<2 = 0010 0

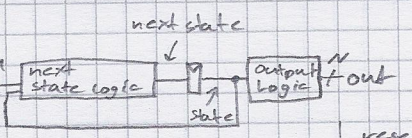
D-Latch

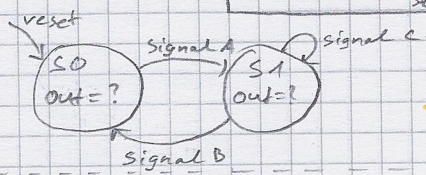
CLK, D, R, S, Q, $\bar{Q}$

CLK	D	Q
0	X	Qvorher
1	0	0
1	1	1

FSM

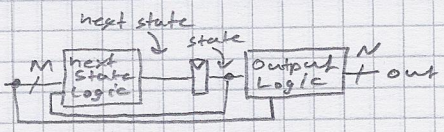
Moore:

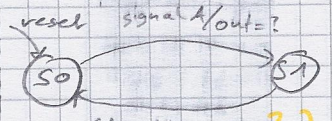




2.)

Mealy:





2.)

1.) Inputs (Signale), Outputs definieren / analysieren
2.) State Transition Diagramm skizzieren
3.) State Transition Table 3a
- State Encoding Table
- Output Encoding Table } 3b
4.) (State Transition Table mit den encodeten Werten)
5.) Output Table
6.) (Boolsche Gleichungen für next state und Output)
7.) Schaltplan

allgemeine Tabellen

Output Encoding:

Output	Encoding
red	0 0
blue	1 0
⋮	⋮

State Encoding Table:

State	Encoding
S0	0 0
S1	1 0
⋮	⋮

3b

Moore Tabellen:

Output Table:

State	Outputs out_1	Outputs out_2
S0	1	0
S1	0	1
S2	1	1
⋮	⋮	⋮

State Transition Table:

State	Input a_1	Input a_2	Input a_3	next state
S0	X	X	1	S0
S0	1	X	X	S1
⋮	⋮	⋮	⋮	⋮

3a

Mealy Tabellen:

State	Inputs a_1	Inputs a_2
S0	1	X
S0	X	1
S1	1	1
⋮	⋮	⋮

State Transition + Output Table: 3a

next state	Outputs o_1	Outputs o_2
S0	1	1
S1	0	1
S0	1	0
⋮	⋮	⋮

Timing

shortest Path ↖ ↗ critical Path

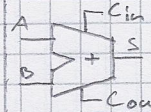
- "clock-to-Q contamination / propagation": Zeit nach CLK rise bis Output beginnt sich zu ändern / finalen Wert erreicht
- "setup time" (t_{setup}): Zeit vor CLK rise während welcher alle Inputs stabil sein müssen
- "hold time" (t_{hold}): Zeit nach CLK rise während welcher die Inputs stabil sein müssen
- "aperture time": $t_{setup} + t_{hold}$: Zeit während der Inputs stabil sein müssen
- "clock period" / "cycle time": T_c: Zeit zwischen zwei CLK rising
- "clock frequency": $\frac{1}{T_c}$, $1\,Hz = 1$ Cycle pro Sekunde

Timing Analysis

1) Minimales T_c berechnen:
   a) kritischen Pfad finden
   b) T_c ist grösser gleich der Summe aus:
      (i) clock-to-Q propagation (Zeit bis Inputs aus Flip-Flops gelesen wurden)
      (ii) propagation delay (Zeit, die die logischen Gates benötigen)
      (iii) setup time (Zeit bis Output gespeichert ist)

2) Prüfen auf "hold time violations":
   a) kürzesten Pfad betrachten
   b) T_c wie bei 1) für diesen Pfad berechnen, statt propagation jeweils contamination bei (i) und (ii)
   c) prüfen ob T_c < hold time vom "output speicher". Wenn ja, so ist die Schaltung instabil! Oft lösbar, in dem Buffer eingebaut werden für die kürzeren Pfade

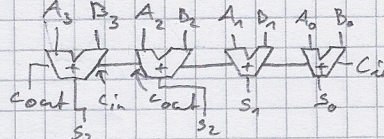
Full Adder



("Half Adder hat kein C_{in})

N-bit Adder

Ripple-Carry Adder: Aneinander gereihte Full Adder, wobei C_{in} ans nächste C_{out} angeschlossen ist.



4-Bit Adder, wobei A_3, B_3, S_3 MSB sind. Es läuft also von links nach rechts. Delay wächst direkt mit der Anzahl Bits: $n \cdot t_{adder} = t$

$t_{CLA} = t_{pg} + t_{pg\_block} + \frac{n}{k} t_{AND\_or} + k \cdot t_{Full\_Adder}$

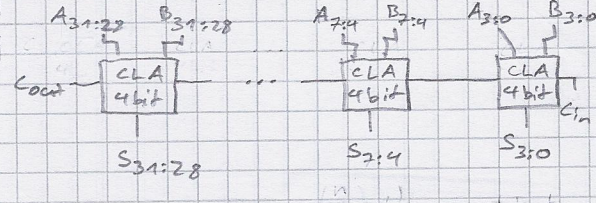
t_{pg}: t_p von 2-Input OR, N: #Bit, k: #Bit pro Block

t_{pg_block}: t_p von 3 2-Input OR + 3 2Input AND | t_{AND_or}: Delay vom $C_{in} \to C_{out}$ zwischen zwei Blöcken

t_{Full_adder}: Delay vom Full Adder

Carry-Lookahead Adder (CLA)

Die Adder werden in Blöcke unterteilt, z.B. 4-Bit Blöcke (= 4 Full Adder, 32 bit hätte 8 × 4-Bit Blöcke). Jeder Block generiert ein "generate" oder "propagate" Signal (= 0 oder 1). Ist "generate" gesetzt, so erzeugt der Block ein Carry out unabhängig von Carry in. Ist "propagate" gesetzt, so verhindert der Block ein Carry in nicht, sondern gibt es weiter.



Prefix Adder

Die "propagate" und "generate" Signale werden zuerst für 2er, dann für 4er, 8er, ... Gruppen berechnet. Bis für jede Spalte die Signale bekannt sind. $t_{PA} = t_{pg} + \log_2(N)\, t_{pg\_prefix} + t_{xor}$. t_{pg_prefix}: t_{pg} von einem Prefix-Block.

Multicycle Processor

- braucht nur einen Adder
- zerlegt Instruktionen in kleine, atomare Schritte
- Pro Cycle kann aus dem Memory gelesen werden, in dieses geschrieben oder die ALU verwendet werden. Einfache Operationen können so auch schneller verarbeitet werden.

$$T_c = t_{pcq} + t_{mux} + \max\{t_{ALU} + t_{mux};\ t_{mem}\} + t_{setup}$$

Single Cycle Processor

1 Instruktion pro Cycle,

$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + 2t_{mux} + t_{ALU} + t_{RFsetup}$

t_{RF}...: register file, t_{pcq_PC}: t_p vom PC Register. → Critical Path: Lw. → clock-to-Q

Pipelined Processor

Prozessor wird in fünf Schritte unterteilt, die jeweils gleichzeitig ablaufen:

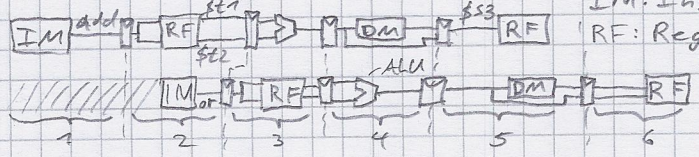
- Fetch: Instruktion aus Memory lesen
- Decode: Instruktion dekodieren und Control Signale setzen
- Execute: Ausführen mit Hilfe der ALU
- Memory: lesen/schreiben aus/ins Mem
- Writeback: in Register schreiben

Hazards: Entsteht, wenn eine Instruktion auf Resultate einer vorherigen wartet, die noch nicht so weit ist. "RAW": Read After Write (es wird zu früh aus den Register gelesen).

"data hazard": Register lesen, bevor es geschrieben wurde. "control Hazard": eine Instruktion wird vom Fetch-Schritt gelesen bevor entschieden ist, was die nächste Instruktion überhaupt ist. "Forwarding" kommt zur Einsatz, wenn der Source der Operation die Destination der vorherigen Instruktion ist. Die Hazard-Detection-Unit erkennt dies und ein Multiplexer dafür sorgt, dass die Daten aus dem Speicher des Zwischenresultats gelesen werden, statt der Register. Funktioniert nicht immer, dann wird ein "stall" generiert. "Stall" Die Instruktion wird angehalten, bis die Daten bereit sind.

IM: Instruction Memory
RF: Register File
DM: Data Memory

add $s3, $t1, $t2
or $s7, $t3, $t4



$$T_c = \max\{t_{pcq} + t_{mem} + t_{setup};\ 2(t_{RFread} + t_{mux} + t_{eq} + t_{XNO} + t_{mux} + t_{setup});\ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup};\ t_{pcq} + t_{memwrite} + t_{setup};\ 2(t_{pcq} + t_{mux} + t_{RFwrite})\}$$

Memory Systems

"miss rate" = #misses / #memory access = 1 − hit rate

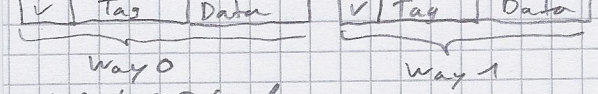
"hit rate" = #hits / #memory access = 1 − miss rate

"average memory access time" (AMAT) $= t_{cache} + MR_{cache}(t_{mm} + MR_{mm}\, t_{vm})$

t_{cache}: access time cache, MR: Miss Rates, t_{mm}: access time main memory, t_{vm}: access time virtual Memory (HDD)

Cache besteht aus "Sets" (S), die 1-N Datenblöcken (B) haben. Zusätzlich kommt noch der "Tag" hinzu, der die Adresse des aktuellen Inhalt enthält und ein Bit-Tag "Valid", das angibt ob die Daten gültig sind.

Multi-way set Associative Cache
Jede Adresse ist einem Set zugewiesen, kann aber in einem der N Blöcke des Sets liegen.



Direct Mapped Cache Jeder Memory Adresse wird ein fester Set zugewiesen. Mehrere Adressen teilen sich also das gleiche Set. Z.B.: werden die Bits 4:2 als Cache Adresse angesehen (ID des Sets). Jedes Set hat 1 Block

Fully Associative Cache Nur 1 Set mit B Blöcken. Es müssen also B Tags geprüft werden bei jeder Anfrage

Direct Mapped braucht am wenigste Hardware. Dann der Fully Associative und darauf folgt der Multi-way Set Associative. Bei der Anzahl Konflikte gerade anders rum.

Die Blöcke sind meist größer als 1 Word, damit benachbarte Speicher gleich mit kommt

Virtual Memory

Cache	Virtual Memory
Block	Page
Block size	Page size
Miss	Page fault
Tag	Virtual Page number

Page Table wird für die Übersetzung virtueller adressen zu physischen adressen (Mem) genutzt. Jeder Eintrag (Zeile) besteht aus einem "valid" Bit und der "physical page number". Es gibt für jede "virtual page number" einen solchen Eintrag. Ist "valid" = 1, so liegen die Daten im Memory, sonst auf der Platte (Page Fault).

Address Translation Der Computer muss eine virtuelle Adresse übersetzen um die Daten im Memory zu finden oder einen Page fault zu verursachen und die Daten von der HDD laden. Die höherwertigen Bits geben die "Page number" an, die niedrigwertigen Bits geben den "Page Offset" an (da Wort innerhalb der Page). Für die Übersetzung der "virtual page number" (19 bit) zur "physical page number" (15 bit) werden Page Tables und TLBs verwendet.

Translation lookaside Buffer (TLB) ist ein kleiner Cache, der die Page Table Einträge enthält, die in der Umgebung des letzten Zugriffs liegen. CPU schaut somit zuerst hier in TLB nach, bevor er die Page Table anspricht, die im Memory liegt. Hit rate meist 99%.

Terminologie "physical Memory": Memory, also RAM

Assembly Instruction Types R-Type: „register type". |op (6bit)|rs (5bit)|rt (5bit)|rd (5bit)|shamt (5bit)|funct (6bit)|, op = 0x0 für R-Type, funct gibt Operation an, rs + rt source register, rd destination register, shamt ist die Anzahl Shifts, die bei Shift-Operationen gemacht werden soll, shamt für restliche Operationen = 0x0. add $s0, $t1, $s2: $0 destination (rd), $t1 source (rs) $s2 source (rt). I-Type: "immediate-type". |op (6bit)|rs (5bit)|rt (5bit)|imm (16 bit)|. rt kann sowohl source, als auch destination Register sein. Imm hat den konstanten Wert drin und op gibt die Operation an. J-Type: „jump type": |op (6bit)|addr (26bit)|. op gibt Operation an und addr ist die Adresse.

Adress Modes register-only, immediate, base addressing: Dazu da die Operanden zu lesen / schreiben. PC-relative, pseudo-direct: Modes um den PC zu schreiben.
Register-only addressing: benutzt Register für die Operande. R-Type Instructions sind von diesem Typ. **Immediate addressing**: Benutzen eine 16-bit Konstante mit Register (addi z.B). I-Type instructions meist von diesem Typ. **Base addressing**: die effektive Adresse berekt aus dem Inhalt des Werts in einem Register plus dem 16-bit Offset in immediate Feld.
PC-relative addressing: conditional branch instructions benutzen dies. Der neue PC besteht aus der Summe des PC Werts und dem signed value vom immediate Feld. **Pseudo-direct addressing**: Jede Adresse an die gesprungen werden kann ist "word-aligned". Die zwei letzten Werte sind also 00, die nächsten 26 Bit kommen aus dem "imm"-Feld. Die höchsten 4 Bit kommen vom PC selbst.

Stack / Procedure calls **Stack**: FIFO, push (insert), pot (remove + get), $sp current stack position that is valid (used). Geht von höherer Adresse zu und "wächst" "nach unten". **Procedure**: Eine Prozedur speichert zuerst die Register auf den Stack. Dabei gibt es folgt vor: • Platz auf Stack machen, • Werte der Register speichern, • Prozedur-Code ausführen, • gespeicherte der Register • Register wieder herstellen mit Hilfe des Stacks • Stack Speicher freigeben. "Stack frame" Stück des Stacks, die eine Prozedur für sich beansprucht. **Register Usage**: Eine Prozedur muss die "preserved" Register speichern und wieder herstellen. Die nonpreserved darf sie nicht benutzen / verändern. Der aufrufende Code ist also verantwortlich die nonpreserved Register zu speichern. **Preserved**: $s[0-7], $ra, $sp **Nonpreserved**: $t[0-9], $a[0-3], $v[0,1]. Der Stack über $sp ist preserved, darunter nicht (nonpreserved).

Combinational logic: Output basiert nur auf aktuellem Input

Sequential logic: Output basiert auf aktuellem und vorherigen Input

Memory adressierung word-addressable: Jedes data-word hat eine eigene Adresse. byte-addressable: Jedes Byte (8 Bit) hat seine eigene Adresse. Beim Mips ist 1 word 32-Bit lang, besteht also aus 4 · 8-Bit. Dennoch ist jede word adresse ein Vielfaches von 4 (0, 4, 8, 12, 16, ...). Die Adresse des words entspricht der Adresse des 1. Bytes dieses words. lw $s7, 8($0) lädt die Daten von 0+8=8 nach $s7.

Verilog

```
module [Name] (
    input clk,
    input [3:0] a,
    output reg [3:0] q);
    // code
endmodule
```

a[3:0] : a[3] MSB, a[0] LSB, a = 4 Bit
a[0:3] : a[0] MSB, a[3] LSB, a = 4 Bit

wire a; "lokale" Variable, für Zuweisung "assign" gebrauchen.

assign c = b | c; ist "continues assignment", ändert sich b oder c, so wird a neu evaluiert

posedge: 0→1, negedge: 1→0

always @([sensitivity list]) wird immer ausgeführt, wenn sich eines der Werte der "sensitivity list" ändern. Alle linken Seiten von "<=" und "=" in einem always-Block müssen "reg" sein. "reg" ≠ Flip-Flop!

```
begin
    // code   [ clk, reset nicht vergessen ]
end
```

"<=" ist ein "non-blocking" assignment [- hat es posedge/negedge so nicht "<=" verwenden]

Regeln:
- "<=" und always-Block für synchrone sequentielle Logik
- continues assignment ohne always-Block für kombinatorische
- nie in verschiedenen always-Blöcken auf gleiche Element schreiben
- nie mehrere continues assignments für die gleiche linke Seite.

[N]'[b][xx]: [N]: # Bits, [b]: Basis (b→binär, h→Hex, d→Dez., o→okt.), [xx] Wert in besagter Basis. Wandelt es dann in Binär um.

```
If [Bedingung] then
    begin
        // code
    end
Else
    begin
        // code
    end
end
```

```
case ([check]):
    0: // code;
    1: // code;
    ...
    default: // cod;
endcase
```

assign y = a ? t : f;
"true" "false"

Flip-Flops:

```
reg q;
always @(posedge clk)
    q <= d
```

```
reg q;
always @(posedge clk, posedge reset)
    if(reset) q <= 0;
    else q <= d
```

```
reg q;
always @(posedge clk, posedge reset,
    posedge set)
    if(reset) q <= 0
    else
        if(set) q <= 1
        else q <= d
```