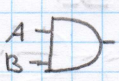
Logik

Verilog →

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

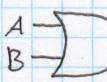
AND



A & B

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

OR



A | B

A	¬A
0	1
1	0

NOT

~A

A	B	$\overline{A\&B}$
0	0	1
0	1	1
1	0	1
1	1	0

NAND

~(A & B)

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

NOR

~(A | B)

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

XOR

A ^ B

signed magnitude 1Bit Vorzeichen

Binär

Dez. → Bin.:

5 : 2 = 2 R 1 ↑ ⇒ 101
2 : 2 = 1 R 0
1 : 2 = 0 R 1

Bin. → Dez.:

1 0 1₂
$\uparrow 2^2 \quad \uparrow 2^1 \quad \uparrow 2^0$ ⇒ $1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0$
$= 4 + 1 = 5$

$1011_2 + 11_2 = 1110_2$

$1011_2 - 111_2 = 100_2$

0+0=0	0-0=0
0+1=1	0-1=-1
1+0=1	1-0=0
1+1=10	1-1=0

Hexadezimal

0-9, A-F: A=10, B=11, ..., F=15

Dez. → Hex.:

1278 : 16 = 79 R 14 (=E) ↑
79 : 16 = 4 R 15 (=F)
4 : 16 = 0 R 4 (=4)

⇒ 4FE

Hex. → Dez.:

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

1s Complement

1. Bit ist das Vorzeichen: 0 ≙ +, 1 ≙ −

negative Zahlen: positive Darstellung invertieren:

+4 = 0100 → −4 = 1011

Achtung: +0 = −0: 0000 = 1111

−4+6 = +2:

Vorsicht beim Rechnen:

		1	0	1	1
	+	0	1	1	0
Übertrag:	1	1	1		
		0	0	0	1
	+				1
		0	0	1	0

$= 1_{10} \neq +2$

$(= 2_{10})$ ✓

2s Complement

negative Zahlen haben auch führende Null.

Dez. → 2s Com.:

1) Absoluten Betrag in Binär umwandeln
2) Wenn negative Zahl:
   a) Invertiere
   b) 1_{10} addieren (+00...1)

2s → Dez.:

1. Bit (MSB) hat den 2^x Stellenwert, nun negativ. Rest normal:

1 1 0 1 0 } ⇒ $-16 + 1\cdot 2^3 + 1\cdot 2^1 = -6$
↑ $-2^4 = -16$
0 1 0 1 0 ⇒ $1\cdot 2^3 + 1\cdot 2^1 = 10$

Addition ganz normal. Subtraktion wird als Addition angesehen: −4 − 5 = −4 + (−5). Achtung: es kann ein zusätzliches Bit im "Übertrag" haben, dieses ignorieren → nicht zwingend Overflow!

8bit: −128 ... 127, 16 Bit: −32768 ... 32767, 32bit: −2147483648 ... +2147483647.

Boolean Alg.:

- $\cdot (B \cdot C) + (B \cdot \overline{C}) = B$
- $\cdot (B + C) \cdot (B + \overline{C}) = B$
- $\cdot (B \cdot C) + (\overline{B} \cdot D) + (C \cdot D) = B \cdot C + \overline{B} \cdot D$
- $\cdot B \cdot (B + C) = B$
- $\cdot B + (B \cdot C) = B$
- $\cdot (B + C) \cdot (\overline{B} + D) \cdot (C + D) = (B + C) \cdot (\overline{B} + D)$
- (De Morgan) $\overline{B_1 \cdot B_2 \cdot B_3 \cdot \ldots} = (\overline{B_1} + \overline{B_2} + \ldots)$
- (De Morgan) $\overline{B_0 + B_1 + B_2 + \ldots} = (\overline{B_0} \cdot \overline{B_1} \cdot \ldots)$

Boolean Equations

Minterm (Sum-of-Products): Zeilen in Wahrheitstabelle, die 1 sind.

A	B	Y	minterm
0	0	0	$\overline{A}\,\overline{B}$
0	1	1	$\overline{A}B$
1	0	0	$A\overline{B}$
1	1	1	AB

⇒ $\overline{A}B + AB$

Maxterm (Product-of-Sums): Zeilen in Wahrheitstabelle, die 0 sind.

A	B	Y	maxterm
0	0	0	$A + B$
0	1	1	$A + \overline{B}$
1	0	0	$\overline{A} + B$
1	1	1	$\overline{A} + \overline{B}$

⇒ $(A+B) \cdot (\overline{A}+B)$

k-Maps:

A	B	C	Y
0	0	0	0
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 ändern

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

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

X / Z

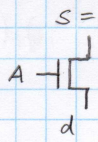
X: egal / unbekannt

Z: floating, zwischen 1 und 0

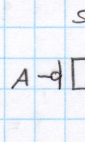
Regeln:

- in den Quadraten sind nur 1'en oder X'en
- Ziel: minimale # Quadrate und alle 1en sind in welche
- Jedes Quadrat hat 2^x 1en / X'en (also 1, 2, 4)
- Quadrate können über die Grenze sich "drehen"
- 1'en / X'en dürfen in mehreren Quadraten sein, wenn
- Bis 4 Variablen brauchbar

CMOS

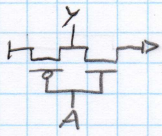
nMOS: 

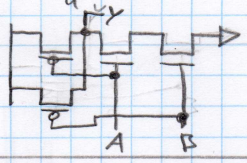
A=0 ⇒ d=0
A=1 ⇒ d=1

pMOS 

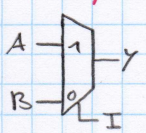
A=0 ⇒ d=1
A=1 ⇒ d=0

"⏚": Ground (=0)
"⊢" / "⎓" / "T"
Spannungsgeber (=1)

NOT-Gate: 

NAND-Gate: 

Multiplexer



wenn I=0: Y=B
wenn I=1: Y=A
I kann auch mehr als 1 Bit sein

Fixed-Point Numbersystem

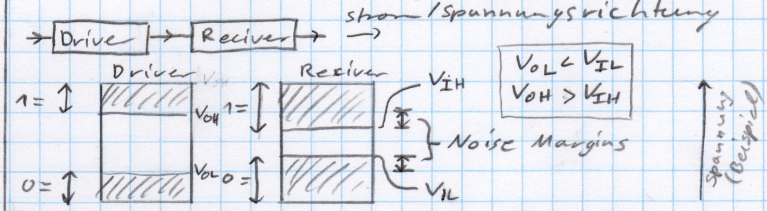
Haben ein imaginäres "Komma". Z.B.: 4 Bit Integer 4 Bit Fraction: 0110.1100 = $2^2+2^1+2^{-1}+2^{-2} = 6,75$
Für negative Zahlen kann 1s/2s complement verwendet werden.

Floating-Point Numbersystem

Entspricht der Wissenschaftliche Schreibweise von Zahlen: $\pm M \cdot B^E$.
Es hat also ein Vorzeichen (Sign, S), Basis (B), Mantissa (M), Exponent (E).
Basis ist 2 für Computer. 32 Bits haben 1 Bit für Vorzeichen, 8 Bit Exponent, 23 Bit Mantissa. In der Binäre Schreibweise wird die führende 1 der Mantissa nicht gespeichert, da sie immer da ist: $228_{10} = 11100100_2 \times 2^0 = 1.11001000 \times 2^7 \Rightarrow E = 7_{10} = 111_2$, $M = 1100100$, $S = 0$. Der Exponent muss auch negative Zahlen erlauben. Dafür wird ein Bias von 127 verwendet:
$7 + 127 = 134 \Rightarrow E = 10000110_2 \hat{=} 7$ als E
$-4 + 127 = 123 \Rightarrow E = 01111011_2 \hat{=} -4$ als E

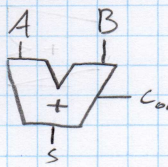
Noise Margin

Um dürfte Werte zu erhalten, muss entschieden werden welcher Bereich einer 1 und welcher einer 0 entspricht. Da durch äußere Umstände diese Werte variieren können erlaubt man eine gewisse Noise:



Adders

Half Adder

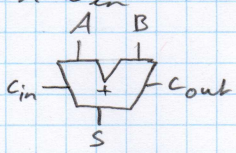


S ist das Resultat der Addition, Cout ist wahr, wenn einen Carry haben (also A=1=B ⇒ S=0, Cout=1).
Für n-Bit Addition werden zweite 1-Bit Adder in Serie geschaltet, wobei Cout zusätzlich noch zur nächst höheren Stelle addiert wird:
$A_0 + B_0 \Rightarrow C_{out} \Rightarrow A_1 + B_1 + C_{out} \ldots$

Full Adder

Hat zusätzlich noch ein C_{in}:

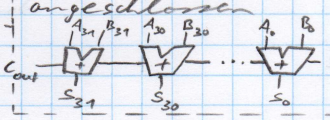
C_{in}	A	B	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Für ein n-Bit Adder werden diese 1-Bit Adder in Serie zusammen geschaltet.

Ripple-Carry Adder

n-Bit Adder bestehend aus n Full Adder. Cout wird jeweils an die nächste C_{in} angeschlossen



Der Delay wächst deutlich mit der Anzahl Bits.

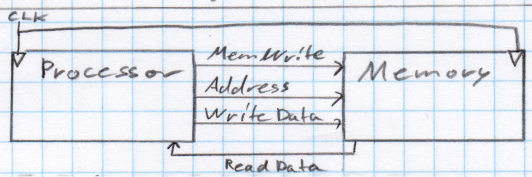
Carry-Lookahead Adder

Der n-Bit Adder wird in kleinere Blöcke aufgeteilt. Zum Beispiel ein 32 Bit in 8 4-Bit Adder. Jede Gruppe produziert zusätzlich ein "propagate" (P) und ein "generate" (G) Signal. P bedeutet, dass die Gruppe ein Carry-Out produziert (=1), wenn das Carry-In = 1 ist. G ist gegeben, wenn die Gruppe so oder so ein Cout produziert (=1). Carry-Out für i-te Spalte: $C_i = G_i + P_i \cdot C_{i-1}$
$G_{3:0} = G_3 + P_3 \cdot (G_2 + P_2 (G_1 + P_1 G_0))$, $P_{3:0} = P_3 P_2 P_1 P_0$, $C_i = G_{i:j} + P_{i:j} C_j$. Performanter Adder

Memory Performance:

Miss-Rate = #Misses / #Memory Accesses = 1 − (Hit-Rate)
Hit-Rate = #Hits / #Memory Accesses = 1 − (Miss-Rate)

Hit: Daten Anfragen, die im Cache gefunden werden
Miss: "−": Daten werden nicht im Cache gefunden → müssen im Hauptspeicher / HDD gesucht werden → sehr langsam



Memory Interface (MIPS)

- Address: Adresse
- Write Data: Data to write
- Read Data: Requested Data
- MemWrite = 0 ≙ Read
  = 1 ≙ Write

Average memory access time (AMAT) = t_{cache} + MissRate$_{cache}$ ($t_{main\,Memory}$ + MissRate$_{main\,Memory}$ · $t_{Virtual\,Memory}$)

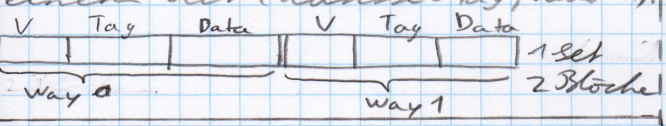
t_{xy} ≙ Zugriffszeit auf "XY"

2

Cache
besteht aus "Sets" (S), die 1-n Datenblöcke aufnehmen. Zu den Daten wird jeweils im "Tag" die Adresse zu den Daten gespeichert und ein Bit-Flag "V" für "Valid".

Direct Mapped Cache
Jede Memory-Adresse wird ein festes Set "zugewiesen". Mehrere Adressen teilen sich ein Set (wie bei Hashtabelle). Zum Beispiel werden die Bits 4:2 der Adresse verwendet um das dazugehörige Set zu finden. Set hat nur 1 Datablock

Multi-way Set Associative Cache
Jede Adresse ist immer noch einem bestimmten Set zugewiesen, kann aber in einem der N Blöcke des Sets liegen. Dabei wählt ein Multiplexer den richtigen Block aus bei einem Hit (Adresse = Tag, Valid = 1).



Fully Associative Cache
Nur 1 Set mit B Ways - also B Blöcke. Es müssen also B Tags bei einer Anfrage geprüft werden.

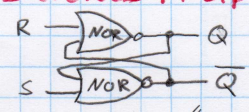
Überblick Direct Mapped Cache braucht am wenigsten Hardware. Dann Multi-Way und am Meisten braucht der Fully Ass.. Die # Misses bzw. Konflikten ist gerade anders rum. Fully Asso. eher für sehr kleine Caches

Die Blockgrössen sind meist mehr als 1 Word. Dadurch wird benachbarter Speicher auch in den Cache geladen. Chancen sind gross, dass dieser benötigt wird!

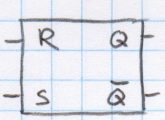
Memory Arrays
Bei einer N-Bit Adresse hat es 2^N Zeile und bei M-Bit Datengrösse M Spalten.

RAM: "volatile": flüchtig
ROM: "nonvolatile": nicht flüchtig

Latches / Flip-Flops
SR Latch

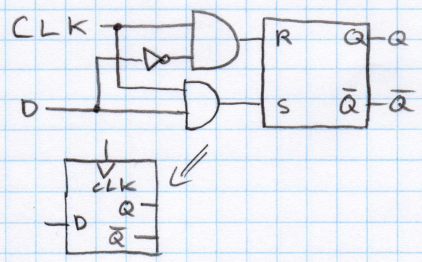


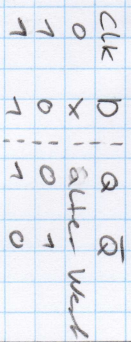
R: setzt es auf 0 = Q (reset)
S: setzt es auf 1 = Q (set)



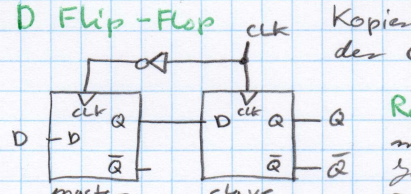
S	R	Q	$\bar{Q}$
0	0	Q_{vorher}	bzw. $\bar{Q}_{vorher}$
0	1	0	1
1	0	1	0
1	1	0	0

D Latch





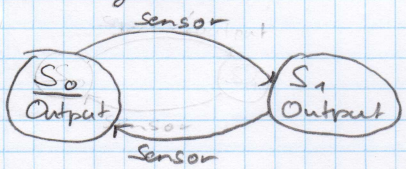
D Flip-Flop
Kopiert D nach Q beim Ansteigen des CLK.



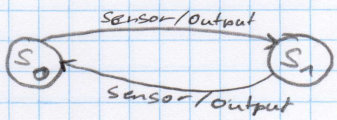
Register werden aus mehreren D Flip-Flops gebaut. Jedes speichert 1 Bit des Registers

Finite State Machines (FSM)
Hat M inputs, N outputs und k Bits. Es hat also $\leq 2^k$ einzigartige Zustände.

Moore machines
Output basiert nur auf dem aktuellen Zustand.



Mealy machines
Output basiert auf aktuellem Zustand und dem aktuellen Input



Zusätzliche Tabellen

State Transition Table

State	Inputs A	Inputs B	Next state
S0	0	0	S0
S1	0	1	S0
S0	1	0	S1

→ oder mit State encoding

00	0	0	00
01	0	1	00
00	1	0	01

↑ Bei Moore ohne Output (wie hier). Bei Mealy noch extra Spalte mit Output!

State Encoding (State Binär)

State	Encoding
S0	00
S1	01
S2	10

Output Encoding

Output	Encoding
grün	00
rot	01
blau	10

Output Table

State	Outputs O_1	Outputs O_2
00	1	0
01	0	1
10	1	1

Vorgehen
a) Inputs / Outputs identifizieren

b) State Transition Diagram Skizzieren ($\frac{0}{00} \rightarrow 0$)

c) Bei Moore:
- State Transition Tabelle machen
- Output Tabelle

c) Kombinierte Tabelle machen (State Transition und Output)

...optional, wenn verlangt:

d) State Encoding wählen

e) Boolsche Gleichungen für nächsten State und Output schreiben

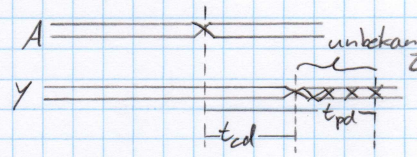
f) Schaltplan zeichnen

3

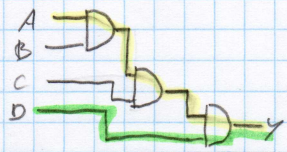
Timing

Big-Endian:	MSB: 0. Byte
Little-Endian:	MSB: n. Byte

- propagation delay (t_{pd}): maximale Zeit, bis der Output auf den Inputwechsel reagiert
- contamination delay (t_{cd}): minimale Zeit, bis irgend ein Output sich ändert



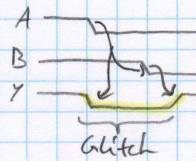
Für eine komplexere Schaltung sucht man für t_{pd} den zeitlich längsten Pfad. Für t_{cd} den zeitlich kürzesten.



wenn D = 1→0 und D = 0→1, wenn Rest schon 1 ist.

$\Rightarrow t_{pd} = 3t_{pd\_and}$
$t_{cd} = t_{cd\_and}$

Glitches

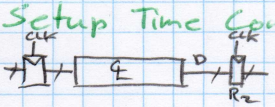


Wenn man t_{pd} wartet, hat Problem. Y nimmt für kurze Zeit wegen der Delays einen unerwarteten Wert an.

System Timing

T_c ist die Zeit zwischen zwei "sizing edges" der "clock". $f_c = 1/T_c$ ist die Frequenz. $T_c \geq t_{pd}$ + [setup zeit]

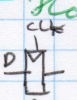
Setup Time Constraint



D muss fertig sein bevor R_2 von clk aktiviert wird

$\Rightarrow T_c \geq t_{pd} + t_{setup}$

Hold Time Constraint

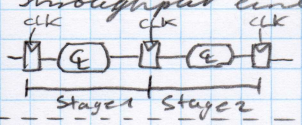


t_{Hold} ist nötig, da sich D eine gewisse Zeit nicht ändern darf, damit R den Wert annehmen kann

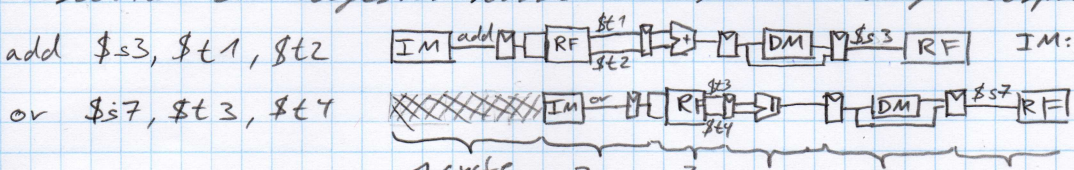
Parallelism

Tokens: Gruppe von Eingaben, die verarbeitet werden zu Ausgabe. **Throughput**: Anzahl Tokens, die verarbeitet werden können pro Zeiteinheit. Mehrere Tokens gleichzeitig zu verarbeiten heißt Parallität.

Spatial parallelism: Kopien der Hardware verarbeiten mehrere Tokens gleichzeitig. **Temporal parallism**: Aufteilen in Tasks, diese zum Teil gleichzeitig abarbeiten auf gleicher Hardware => auch als "Pipelining" bezeichnet. Dazu werden Register eingefügt zwischen Teilstücke der Schaltung. Diese speichern die Zwischenergebnisse für nächste Clk-Zyklus. Sei L das längste Stück, dann ist der Throughput einer Pipeline-Konstruktion 1/L.



MIPS 5 werden 5 Stages eingeführt: Fetch, Decode, Execute, Memory, Writeback. **Fetch**: Instruktion aus Memory lesen | **Decode**: Control-Signal aus Instruktion erzeugen | **Execution**: Ausführen mit der ALU | **Memory**: Lesen/Schreiben vom/ins Memory | **Writeback**: Resultat ins Register schreiben wenn nötig. Beispiel:



Hazard: wenn eine Instruktion auf eine vorherigen basiert, die noch nicht so weit ist im pipelined Processor. Mit **Forwarding** können Hazards gelöst werden bei dem der Source eine Destination ist einer vorher (noch laufenden) getakteten Instruktion. Durch eine Hazard-Detection-Unit wird so ein Fall erkannt. Durch Multiplexer werden die Source-Daten nicht aus dem Memory geladen, sondern den richtige Register mit dem Zwischenresultat der vorhergehende Instruktion. Funktioniert nicht immer! Dann wird ein **Stall** gemacht. Die Instruktion bleibt im aktuelle Abschnitt, bis die Daten verfügbar sind.

Assembly: Instruction Types: Alle 32 bit lang

R-Type: register-type. `op (6 bit) | rs (5 bit) | rt (5 bit) | rd (5 bit) | shamt (5 bit) | funct (6 bit)` op: operation code, rs + rt: source register, rd: destination reg., shamt: für shift operationen (amount of shift), für nicht shift operationen = 0, funct: gehört, wie opcode, zur Instruktion. op + func → operation "ID"

I-Type: immediate-type. `op (6 bit) | rs (5 bit) | rt (5 bit) | imm (16 bit)` op: operation code, rs: source register, imm: "source": direkter Wert als Quelle, rt: destination register oder als weiterer source register

J-Type: jump-type. `op (6 bit) | addr (26 bit)` op: opcode, addr: adresse

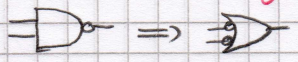
Examples: add $s0, $s1, $s2 R-Type

($s0 ← target, $s1 ← source 1, $s2 ← source 2)

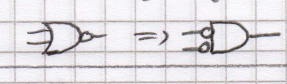
addi $s0, $s1, 5 I-Type

Stack: Last-in-first-out queue. add: "push", get: "pop". Jede Funktion muss vor dem Ende ihren allozierten Speicher wieder aufräumen. MIPS-Stack wächst nach unten. $sp zeigt auf den zuletzt allozierten Speicher. **Schritte einer Prozedur:** 1) Platz auf Stack besorgen für Register, die zu speichern sind 2) Registerwerte auf Stack speichern. 3) Unterprozedur ausführen mit hilfe der Register 4) originalwerte vom Stack wiederherstellen 5) Platz freigeben auf Stack.

Bubble Pushing: vom Output anfangen und zum Eingang gehen



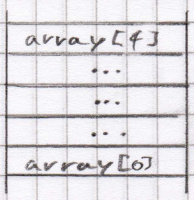
o—o => — (not not)



Shifters **Logic shifter:** 11001 >>2 = 00110, 11001 <<2 = 00100

Arithmetic shifter: 11001 >>> 2 = 11110, 11001 <<2 = 00100

Array auf stack



load `array[0]` to $t1. $s0 ist baseadresse des Arrays:

`lw $t1, 0($s0)` . `array[2]` in $t2 laden:

`lw $t2, 8($s0)`

↳ 2 · 4bit

Register bei Routinen: Perserved: Aufgerufene muss diese sich merken (auf Stack legen) Nonperserved: Aufrufenden muss diese sich merken

Verilog:

Modul:

```
module [name] (a, b, c, y):
    input a;
    input b;
    input c;
    output y;

    //code

endmodule
```

Ranges (Bus): [start : end]

```
input [31:0] a; //a[31],...,a[0]
output [15:8] y; //y[15],...,y[8]
```

Numbers:

[N]'[b][xx]: N: # bits

b: Basis:

→ 4'b1001 → '1001'
→ 4'd 5 → '0101'

"b": Binary
"h": Hex
"d": Dezimal
"o": Oktal

xx: Wert

Always:

```
always @ (Liste für Reaktionsauslöser)
    begin
        //code
    end
```

↳ "posedge clk" (clk Parameter/Input des Moduls!)

"negedge clk"

↳ posedge: 0→1, negedge: 1→0

"clk" immer: 0→1 und 1→0

//code: Signale zugewiesen in einem Alwaysblock haben "reg" Keyword und Zuweisung ist "<=": output reg y;

im always Block: y <= 1;

If

```
If [Bedingung then
    //code
else
    //code
end
```

Case

```
case (input)
    0: //code;
    1: //code;
    //...
    default: //code;
endcase
```

Program Execution Time = (#Instructions) · (cycles/instruction) (seconds/cycle)
= #Instructions · CPT · T_c

Single-cycle: $T_c = t_{pcq\_pc} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$

CPI = 1

Multi-cycle: $T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$, CPI > 1