

# Digitaltechnik Zusammenfassung 2009 <sup>1</sup>

Revision 26

Stefan Heule

13. Juli 2016









<sup>1</sup>Licence: Creative Commons Attribution-Share Alike 3.0 Unported (<http://creativecommons.org/licenses/by-sa/3.0/>)

# 1 Kombinatorische Schaltungen

A combinational circuit outputs, only depend on it's inputs therefore it is memoryless, a sequential circuit output's depend both on it's current and old inputs, it has memory. A combinational circuit has the following properties:

- Every circuit element is combinational.
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

## 1.1 Gatter

	Logik	Algebra	Schaltbild	Verilog
OR	$x \vee y$	$x + y$		<code>x    y</code>
NOR	$\neg(x \vee y)$	$\overline{x + y}$		
AND	$x \wedge y$	$x \cdot y$		<code>x &amp;&amp; y</code>
NAND	$\neg(x \wedge y)$	$\overline{x \cdot y}$		
NOT	$\neg x$	$\bar{x}$		<code>!x</code>
XOR	$x \neq y$	$x \oplus y$		<code>x ^ y</code>
XNOR	$x = y$	$\overline{x \oplus y}$		<code>x == y</code>
	$x \rightarrow y$	$x \leq y$		<code>!x    y</code>

## 1.2 Wahrheitstabellen

A literal can have the value of 1, 0, x if it is a illegal value(contention) or z if it's is floating(not yet defined).

a	b	$\neg a$	$a \wedge b$	$a \vee b$	$a \rightarrow b$	$a \leftrightarrow b$	$a \oplus b$
1	1	0	1	1	1	1	0
1	0	0	0	1	0	0	1
0	1	1	0	1	1	0	1
0	0	1	0	0	1	1	0

## 1.3 Normalformen

- DNF:** Disjunktive Normalform(sum of products)(Formel mit nur  $\vee$ ,  $\wedge$  und  $\neg$  wo alle mit  $\vee$  verbunden sind). In der Wahrheitstabelle können die "1-Zeilen" verodert werden.

( $0 \rightarrow \neg x, 1 \rightarrow x$ )

- Literal:** negierte oder unnegierte Variable
- Produktterm:** Konjunktion von Literalen
- Minterm:** holds all literals with and's
- Maxterm:** holds all literals with or's

- KNF:** Konjunktive Normalform(alle mit  $\wedge$  verbunden sind). In der Wahrheitstabelle können die "0-Zeilen" verundet werden.

( $0 \rightarrow x, 1 \rightarrow \neg x$ )

## 1.4 Umformungen

$$\begin{aligned}
 a \oplus b &= (a \wedge \neg b) \vee (\neg a \wedge b) = \bar{a}b + a\bar{b} \\
 &= (a \vee b) \wedge (\neg a \vee \neg b) = (a + b)(\bar{a} + \bar{b}) \\
 &= (a \vee b) \wedge \neg(a \wedge b) = (a + b)(\overline{ab})
 \end{aligned}$$

$$P \neg A + PA = P$$

## 1.5 Bubble pushing

You can use bubble pushing to make a figure more readable. To do this push the bubbles from the output to the input. The rules of bubble pushing are:

- Pushing bubbles backward (from the output) or forward (from the inputs) changes the body of the gate from AND to OR or vice versa
- Pushing a bubble from the output back to the inputs puts bubbles on all gate inputs.
- Pushing bubbles on all gate inputs forward toward the output puts a bubble on the output

## 1.6 Frequenzberechnung(sequential circuits)

$$\begin{aligned}
 \text{Zykluszeit: } t_z &\geq t_s + t_p + t_l + t_{skew} \\
 \text{propagation delay: } t_p &\leq t_z - (t_l + t_s + t_{skew}) \\
 \text{propagation delay: } t_l &\geq t_{hold} + t_{skew} - t_{ccq}
 \end{aligned}$$

$$\text{Max. Frequenz: } f_{max} = \frac{1}{t_z} = \frac{1}{t_s + t_p + t_l}$$

$t_s$  : Setupzeit

$t_p$  : Propagierungsdelay der Flipflops

$t_l$  : Längster Pfad

$t_{skew}$  : Difference between times clock takes to reach

$t_{ccq}$  : Time in the flip-flop

The propagation delay is the sum of the times threw each logic unit in the longest path.

The contamination delay is the sum of the times threw each logic unit in the shortest path.

If we look a the result in between these two delays, the result might change value, this is called a glitch. If the contamination delay is too large the clock period can be increased. Synchronous sequential circuits have a timing specification including the clock-to-Q propagation and contamination delays, tpcq and tccq, and the setup and hold times, tsetup and thold. For correct operation, their inputs must be stable during an aperture time that starts a setup time before the rising edge of the clock and ends a hold time after the rising edge of the clock. The minimum cycle time, Tc, of the system is equal to the propagation delay, tpd, through the combinational logic plus tpcq tsetup of the register. For

correct operation, the contamination delay through the register and combinational logic must be greater than thold. Despite the common misconception to the contrary, hold time does not affect the cycle time.

## 1.7 Rechenregeln

Kommutativität	$x \wedge y \equiv y \wedge x$ $x \vee y \equiv y \vee x$
Assoziativität	$x \wedge (y \wedge z) \equiv (x \wedge y) \wedge z$ $x \vee (y \vee z) \equiv (x \vee y) \vee z$
Distributivität	$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$ $x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$
De'Morgan	$\neg(x \wedge y) \equiv \neg x \vee \neg y$ $\neg(x \vee y) \equiv \neg x \wedge \neg y$
Idempotenz	$x \wedge x \equiv x$ $x \vee x \equiv x$
Controlling Value	$x \wedge 0 \equiv 0$ $x \vee 1 \equiv 1$
Neutraler Wert	$x \wedge 1 \equiv x$ $x \vee 0 \equiv x$
Doppelte Negation	$\neg(\neg x) \equiv x$
Abschwächung	$x \wedge y \equiv x$ wenn $x \Rightarrow y$ $y$ schwächer als $x$
Verstärkung	$x \vee y \equiv x$ wenn $y \Rightarrow x$ $y$ stärker als $x$
Konsensus	$x \cdot y + \bar{x} \cdot z + y \cdot z \equiv x \cdot y + \bar{x} \cdot z$
Shannon-Expansion	$e \equiv x \wedge e[1/x] \vee \neg x \wedge e[0/x]$

where  $e$  is a function and  $e[1/x]$  is the function where you replace  $x$  by 1. Shannon's expansion tells us that any function can be written with a multiplexer(because a double multiplexer is written with  $\neg$  the selector  $\wedge$  the first function  $\vee$  the selector  $\wedge$ ).

## 1.8 Bindung der Operatoren

stärker  $\neg \wedge \vee \rightarrow \oplus \leftrightarrow$  schwächer ( $\oplus$ =xor)

# 2 Sequential circuits

The outputs of sequential logic depend on both current and prior input values. Hence, sequential logic has memory. The sequential system remembers its prior inputs in information which is called the state of the system, which is stored in state variables. We are going to build synchronous sequential circuits consisting of combinational logic and banks of flip-flops containing the state of the circuit, final state machines.

## 2.1 Memory

A bistable element is an element that has two phases, and holds one bit of info.

**2.1.0.1 latches** There are two types of latches, the sr latch which can store the data, but is a bit weird and the d latch which can store data, it also has an input clock, when clock is 1, the latch shows its new updated value, when clock is 0 the latch shows its old value.

**2.1.0.2 flip-flop** a D flip-flop is composed of two back to back D latches. It has an input D and an output Q. The D flip-flop copies D to Q on the rising edge of the clock, and remembers its state at all other times.

An enabled flip-flop has another input, EN (enable), when EN is TRUE, the enabled flip-flop behaves like an ordinary D

flip-flop. When EN is FALSE, the enabled flip-flop ignores the clock and retains its state.

A resettable flip-flop adds another input called RESET. When RESET is FALSE, the resettable flip-flop behaves like an ordinary D flip-flop. When RESET is TRUE, the resettable flip-flop ignores D and resets the output to 0. Such flip-flops may be synchronously or asynchronously resettable. Synchronously resettable flip-flops reset themselves only on the rising edge of CLK. Asynchronously resettable flip-flops reset themselves as soon as RESET becomes TRUE, independent of CLK.

**2.1.0.3 registers** , a register has  $n$  flip-flops that all have the same CLK, so that all the bits are updated at the same time.

Remember that a D latch is level-sensitive, whereas a D flip-flop is edge-triggered. The D latch is transparent when CLK 1, allowing the input D to flow through to the output Q. The D flip-flop copies D to Q on the rising edge of CLK. At all other times, latches and flip-flops retain their old state.

## 2.2 synchronous sequential circuit

This circuit is composed of combinational logic and registers that store the state of the system. The state changes at the clock edge, it's synchronised to the clock.

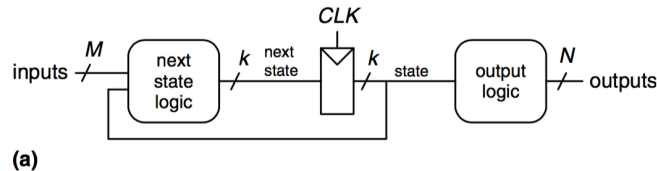
A synchronous sequential circuit has:

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register.

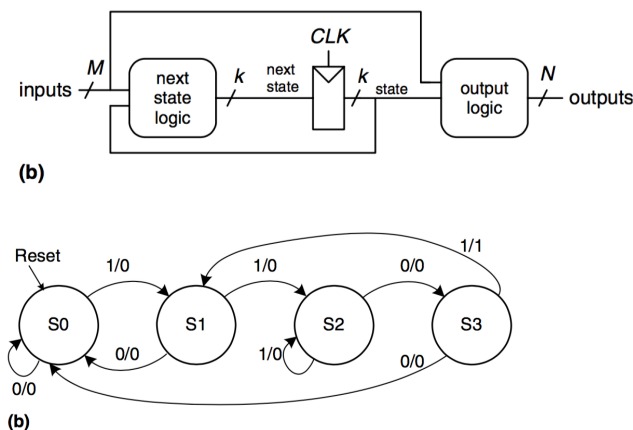
Flip-flops, finite state machines and pipelines are synchronous sequential circuit.

## 2.3 Finite state machines

A finite state machine with  $k$  registers can be in one of  $2^k$  unique states. There are two types of FSM's. The Moore machines, where the output depends on the current state of the machine.:



And mealy machines, where the outputs depend on both the current state and the current inputs.



The second picture are the transition state diagrams, since the mealy machine also depends on the input the arcs also contain the output.

To encode the different states you can use binary encoding, where each state is represented by a binary number. Or one hot encoding where each state is represented by a bit (ex 100, 010 and 001). One hot takes more place but requires fewer gates. To design a FSM:

- i. Identify the inputs and outputs.
- ii. Sketch a state transition diagram.
- iii. For a Moore machine:
  - (a) Write a state transition table.
  - (b) Write an output table.
- iv. For a mealy machine:
  - (a) Write a combined state transition and output table.
- v. Select state encodings—your selection affects the hardware design.
- vi. Write Boolean equations for the next state and output logic.
- vii. Sketch the circuit schematic.

**2.3.0.1 Dynamic discipline** , There is a time during which you cannot write a value to a flip-flop because it is during the clock cycle. For the circuit to sample its input correctly, the input must have stabilized at least some setup time, before the rising edge of the clock and must remain stable for at least some hold time, after the rising edge of the clock. The sum of the setup and hold times is called the aperture time of the circuit, because it is the total time for which the input must remain stable.

The dynamic discipline states, that the inputs of a synchronous sequential circuit must be stable during the setup and hold aperture time around the clock edge.

**2.3.0.2 Metastable State** , when the state of the flip-flop is undefined, illegal. The flip-flop will only stay like this for some time and then change to a valid state. Metastable states are due to asynchronous input from users which we can't avoid.

**2.3.0.3 Synchronizer** A synchronizer takes an asynchronous input and a clock and with a high probability returns a synchronous output.

## 2.4 Parallelism

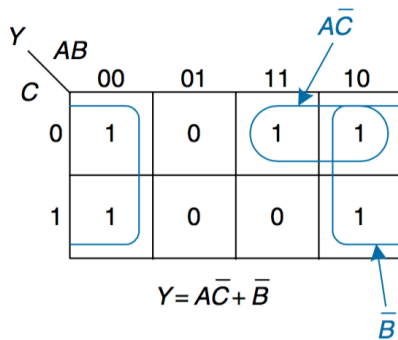
**2.4.0.1 Latency** , is the time required for a piece of information to pass from start to end of the system

**2.4.0.2 Throughput** , number of pieces of information the system produces per unit of time.

**2.4.0.3 Pipelining a circuit** , you can pipeline a circuit by adding registers into the circuit which will let you divide the calculations, like pipelining in PP. Adding a pipeline stage improves throughput at the expense of some latency.

### 3 Formel-Minimierung

#### 3.1 Karnaugh-Maps



You circle the biggest combination of one's possible with max length 2, and take the variables that don't change, karnaugh graphs work for functions with 4 or less variables.

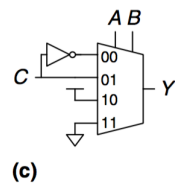
#### 3.2 Multiplexer

You can also write logic with a multiplexer. The selector of the multiplexer are your literals, the input of the multiplexer are one's for the input combination of your function, and zeros for the rest. To make the multiplexer smaller you can use a decoder.

**3.2.0.1 decoder** , in a decoder you make output of the multiplexer depend on a chosen literal, the High, to build a decoder it's easiest to use the truth table:

A	B	C	Y	A	B	Y
0	0	0	1	0	0	$\bar{C}$
0	0	1	0	0	1	C
0	1	0	0	1	0	1
0	1	1	1	1	1	0
1	0	0	1			
1	0	1	1			
1	1	0	0			
1	1	1	0			

(a) (b)



## 4 Assembler

### 4.1 Binärzahlen

0	0000	0	8	1000	8	$2^0 = 1$	$2^8 = 256$
1	0001	1	9	1001	9	$2^1 = 2$	$2^9 = 512$
2	0010	2	10	1010	a	$2^2 = 4$	$2^{10} = 1024$
3	0011	3	11	1011	b	$2^3 = 8$	$2^{11} = 2048$
4	0100	4	12	1100	c	$2^4 = 16$	$2^{12} = 4096$
5	0101	5	13	1101	d	$2^5 = 32$	$2^{13} = 8192$
6	0110	6	14	1110	e	$2^6 = 64$	$2^{14} = 16384$
7	0111	7	15	1111	f	$2^7 = 128$	$2^{15} = 32768$

### 4.2 Instructions

$a = b + c$       add a, b, c

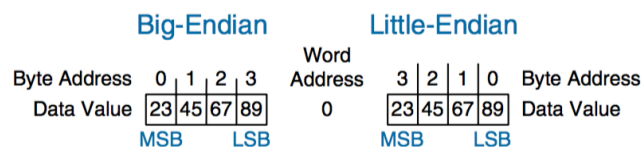
$a = b - c$       sub a, b, c

$\$s0-\$s7$        $\$t0-\$t9$       are memory in the SPRAM register, t are temporary(local) and s are saved.

Name	Number	Use
$\$0$	0	the constant value 0
$\$at$	1	assembler temporary
$\$v0-\$v1$	2-3	procedure return values
$\$a0-\$a3$	4-7	procedure arguments
$\$t0-\$t7$	8-15	temporary variables
$\$s0-\$s7$	16-23	saved variables
$\$t8-\$t9$	24-25	temporary variables
$\$k0-\$k1$	26-27	operating system (OS) temporaries
$\$gp$	28	global pointer
$\$sp$	29	stack pointer
$\$fp$	30	frame pointer
$\$ra$	31	procedure return address

lw  $\$s3, 1(\$0)$       #load memory word ( $0+1 = 1$ ) into s3

sw  $\$s3, 3(\$0)$       #write s3 into memory word 3



Each word in memory consists of 4 bytes, therefore you have to store in memory by a multiple of 4.

addi  $\$s1, \$s0, -12$       you use addi to add constants to two register operands, there is no subi so use negative numbers instead.

You can use addi to initialise a constant ex      addi  $\$s0, \$0, 0x4f3c$       initialisis  $\$s0$  to  $0x4f3c$

#### 4.2.1 Intructions types

There are three instruction types R-type, I-type, and J-type. R-type instructions operate on three registers(add, sub). I-type instructions operate on two registers and a 16-bit immediate(addi, lw, sw). J-type (jump) instructions operate on one 26-bit immediate.

Each instruction is stored in a 32 bit word. So a program consists of 32 bit numbers representing the instructions.

#### 4.2.2 Logic instructions

R-type:

and  $s1, s2, s3$       writes two s1 the bytes that are both 1 in s2 and s3

there is also or and nor

I-type: andi, ori and xori      they work the same as R-type but put a fix value in hexadecimal instead of last operand.

### 4.2.3 Shifts

Are R-type: `sll(v)` shift left logical, `srl(v)` shift right logical and `sra(v)` shift right arithmetic. Shifting a value left by  $n$  is like multiplying it with  $2^n$ . shifting a value right by  $n$  is like dividing it with  $2^n$ . arithmetic shifts and places one's instead of zeros.

### 4.2.4 mult in div

`mult $s0, $s1` multiplies `s0` with `s1` the 32 most significant bits are placed in `hi` and 32 least in `lo`.

`div $s0, $s1` the quotient is placed in `lo` and the remainder in `hi`

### 4.2.5 branching

if else is executed with branching and `beq(branch if equal)` and `bne(branch if not equal)` instructions.

#### MIPS Assembly Code

```
addi $s0, $0, 4      # $s0 = 0 + 4 = 4
addi $s1, $0, 1      # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2      # $s1 = 1 << 2 = 4
beq  $s0, $s1, target # $s0 == $s1, so branch is taken
addi $s1, $s1, 1      # not executed
sub  $s1, $s1, $s0     # not executed

target:
add  $s1, $s1, $s0     # $s1 = 4 + 4 = 8
```

#### MIPS Assembly Code

```
addi $s0, $0, 4      # $s0 = 0 + 4 = 4
addi $s1, $0, 1      # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2      # $s1 = 1 << 2 = 4
bne  $s0, $s1, target # $s0 == $s1, so branch is not taken
addi $s1, $s1, 1      # $s1 = 4 + 1 = 5
sub  $s1, $s1, $s0     # $s1 = 5 - 4 = 1

target:
add  $s1, $s1, $s0     # $s1 = 1 + 4 = 5
```

If it's true it executes the target, else it executes the code after and then the target.

### 4.2.6 jump

There are 3 jump instruction `j`, `jr` and `jal`. `j` just jumps to the specified label ex

`j target` jumps to target

`jr` jumps to the address held in a register and `jal` is similar to `j` but is used by procedures to save a return address.

### 4.2.7 if else case

You can make if else statements using `beq` and `bne`:

#### High-Level Code

```
if (i == j)
    f = g + h;

f = f - i;
```

#### MIPS Assembly Code

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
bne $s3, $s4, L1      # if i != j, skip if block
add $s0, $s1, $s2      # if block: f = g + h
L1:
sub $s0, $s0, $s3      # f = f - i
```



High-Level Code	MIPS Assembly Code
<pre> if (i == j)     f = g + h;  else     f = f - i; </pre>	<pre> # \$s0 = f, \$s1 = g, \$s2 = h, \$s3 = i, \$s4 = j     bne \$s3, \$s4, else      # if i != j, branch to else     add \$s0, \$s1, \$s2      # if block: f = g + h     j    L2                # skip past the else block else:     sub \$s0, \$s0, \$s3      # else block: f = f - i L2: </pre>

You can implement a case by using multiple if else statements.

High-Level Code	MIPS Assembly Code
<pre> switch (amount) {     case 20: fee = 2; break;      case 50: fee = 3; break;      case 100: fee = 5; break;      default: fee = 0; }  // equivalent function using if/else statements if      (amount == 20) fee = 2; else if (amount == 50) fee = 3; else if (amount == 100) fee = 5; else      fee = 0; </pre>	<pre> # \$s0 = amount, \$s1 = fee  case20:     addi \$t0, \$0, 20      # \$t0 = 20     bne \$s0, \$t0, case50  # i == 20? if not,                         # skip to case50     addi \$s1, \$0, 2      # if so, fee = 2     j    done            # and break out of case  case50:     addi \$t0, \$0, 50      # \$t0 = 50     bne \$s0, \$t0, case100 # i == 50? if not,                         # skip to case100     addi \$s1, \$0, 3      # if so, fee = 3     j    done            # and break out of case  case100:     addi \$t0, \$0, 100     # \$t0 = 100     bne \$s0, \$t0, default # i == 100? if not,                         # skip to default     addi \$s1, \$0, 5      # if so, fee = 5     j    done            # and break out of case  default:     add  \$s1, \$0, \$0      # charge = 0  done: </pre>

#### 4.2.8 Loops

For loops you can use the same principal as for the case but if a condition is met it will skip to the same section.

High-Level Code	MIPS Assembly Code
<pre> int pow = 1; int x   = 0;  while (pow != 128) {     pow = pow * 2;     x = x + 1; } </pre>	<pre> # \$s0 = pow, \$s1 = x     addi \$s0, \$0, 1      # pow = 1     addi \$s1, \$0, 0      # x = 0      addi \$t0, \$0, 128    # t0 = 128 for comparison while:     beq  \$s0, \$t0, done  # if pow == 128, exit while     sll  \$s0, \$s0, 1     # pow = pow * 2     addi \$s1, \$s1, 1     # x = x + 1     j    while done: </pre>

High-Level Code	MIPS Assembly Code
<pre> int sum = 0;  for (i = 0; i != 10; i = i + 1) {     sum = sum + i; } </pre>	<pre> # \$s0 = i, \$s1 = sum     add  \$s1, \$0, \$0     # sum = 0     addi \$s0, \$0, 0      # i = 0     addi \$t0, \$0, 10     # \$t0 = 10  for:     beq  \$s0, \$t0, done  # if i == 10, branch to done     add  \$s1, \$s1, \$s0   # sum = sum + i     addi \$s0, \$s0, 1     # increment i     j    for done: </pre>

### 4.2.9 magnitude comparison

To use a smaller than:

`slt rd, rs, rt` sets `rd` to 1 when `rs` < `rt`. Otherwise, `rd` is 0.

### 4.2.10 Arrays

A array is basically a address in memory which you increment by 4 to reach the other array values.

#### High-Level Code

```
int array [5];

array[0] = array[0] * 8;

array[1] = array[1] * 8;
```

#### MIPS Assembly Code

```
# $s0 = base address of array
lui    $s0, 0x1000    # $s0 = 0x10000000
ori    $s0, $s0, 0x7000 # $s0 = 0x10007000

lw     $t1, 0($s0)    # $t1 = array[0]
sll    $t1, $t1, 3    # $t1 = $t1 << 3 = $t1 * 8
sw     $t1, 0($s0)    # array[0] = $t1

lw     $t1, 4($s0)    # $t1 = array[1]
sll    $t1, $t1, 3    # $t1 = $t1 << 3 = $t1 * 8
sw     $t1, 4($s0)    # array[1] = $t1
```

### 4.2.11 Functions

To call a procedure `jal` is used and `jr $ra` is used to return from a procedure. Procedures use `$a0-$a3` as input arguments and `$v0-$v1` as return values.

```
int main ()
{
    int y;

    ...

    y = diffofsums (2, 3, 4, 5);

    ...
}

int diffofsums (int f, int g, int h, int i)
{
    int result;

    result = (f + g) - (h + i);
    return result;
}
```

```
main:
    ...
    addi $a0, $0, 2    # argument 0 = 2
    addi $a1, $0, 3    # argument 1 = 3
    addi $a2, $0, 4    # argument 2 = 4
    addi $a3, $0, 5    # argument 3 = 5
    jal diffofsums     # call procedure
    add $s0, $v0, $0    # y = returned value
    ...

# $s0 = result
diffofsums:
    add $t0, $a0, $a1   # $t0 = f + g
    add $t1, $a2, $a3   # $t1 = h + i
    sub $s0, $t0, $t1   # result = (f + g) - (h + i)
    add $v0, $s0, $0    # put return value in $v0
    jr  $ra             # return to caller
```

The implementation of `diffofsums` is not great because it changes the value of `s0` which it shouldn't. To ameliorate this `diffofsums` should only use `t` variables or restore the value of `s0` after the function using a stack.

### 4.2.12 stacks

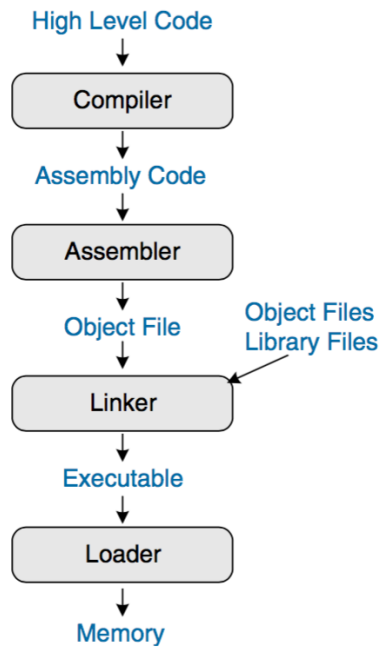
Stacks are a LIFO queue. Better version of `diffofsums` using a stack:

```
# $s0 = result
diffofsums:
    addi $sp, $sp, -4    # make space on stack to store one register
    sw   $s0, 0($sp)     # save $s0 on stack
    add  $t0, $a0, $a1    # $t0 = f + g
    add  $t1, $a2, $a3    # $t1 = h + i
    sub  $s0, $t0, $t1    # result = (f + g) - (h + i)
    add  $v0, $s0, $0     # put return value in $v0
    lw   $s0, 0($sp)     # restore $s0 from stack
    addi $sp, $sp, 4     # deallocate stack space
    jr   $ra             # return to caller
```

Preserved	Nonpreserved
Saved registers: \$s0-\$s7	Temporary registers: \$t0-\$t9
Return address: \$ra	Argument registers: \$a0-\$a3
Stack pointer: \$sp	Return value registers: \$v0-\$v1
Stack above the stack pointer	Stack below the stack pointer

In recursive function calls you have to restore also non preserved registers.  
If more than 4 input variables are needed you can use the stack.  
global variables are \$gp

#### 4.2.13 the Program just for interest



## 5 Verilog

### 5.1 Operatoren

+, -, *	Arithmetic operators
/	Integer division (fractional part truncated)
%	Modulo (takes sign of the first operand)
**	Exponent
-	Negation (2's complement)
~	Bitwise NOT (1's complement)
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
^^, ^^	Bitwise XNOR
<<, >>	Logical shift (padding with 0)
<<<, >>>	Arithmetic shift (padding with leftmost bit when shifting right)
?:	Conditional
{ a, b }	Concatenation
{ num { a } }	Replication

### 5.2 Vergleiche

>, <, >=, <=	Relational operators (0, 1 or x)
==, ==	Logical equality (0, 1 or x)
===, !=	Case equality (0 or 1)

### 5.3 Konstanten

Angabe mit [**<width in bits>**]'<base><number>

'b	(binär, 2)
'o	(oktal, 8)
'd	(dezimal, 10)
'h	(hexadezimal, 16)

z.B. -4'd3 (1101), 4'b11 (0011), 'h08FF (16 bit hex)

### 5.4 Module

In Verilog bestehen die Modelle aus Modulen, wobei jedes Modul ein Interface besitzt, welches die Inputs und Outputs definiert. Ein Modul kann dann instanziiert werden, wobei ihm ein eindeutiger Name zugewiesen wird.

```
// modul definieren
module abc(input a,b, input [3:0] data, output out);
    wire x;
    assign x = (!a || b) ^ a;
    assign out = x ;
endmodule
```

```
endmodule

// module instanzieren
module main(..)
    abc A1(a,x,data,myvar);
endmodule
```

### 5.5 always

Mit **always** lassen sich Endlosschleifen ausdrücken. Zusätzlich kann die Ausführung auf positive/negative Flanken einer bestimmten Variable eingeschränkt werden.

```
always @([posedge | negedge] var) begin .. end
```

### Zeitauflösung/Delays

'timescale <Zeiteinheit>/<Auflösung>

Wird im Header eines Moduls angegeben. z.B. 1ns/100ps bedeutet dass der Delay-Operator #1 für 1ns verzögert und die Simulation in 100ps Zeitschritten läuft.

### Zuweisungsoperatoren

**a = b** *blocking assignment*: Der Wert der Zielvariable wird sofort aktualisiert.

**a <= b** *non-blocking assignment*: Die rechte Seite wird sofort ausgewertet, die Zuweisung erfolgt jedoch erst im nächsten Zeitschritt (nicht Flanke/Takt!), was eine parallele Ausführung solcher Befehle ermöglicht. Die NBA's bewirken keine Verzögerung im Timingdiagramm (ausser wenn die Auflösung gleich der Zeiteinheit ist, was nicht sehr sinnvoll ist).

**assign a = b** *continuous assignment*: Weist der Zielvariable sofort den Wert des Ausdrucks zu, wenn immer dieser ändert. Ausserhalb von always und initial Blöcken!

### Weitere Konstrukte

<b>reg</b> a,b;	Lokale 1-Bit Variablen (Register)
<b>reg</b> [3:0] d;	Lokale 4-Bit Variablen
<b>wire</b> a	Draht, verbindet z.B. Ein-/Ausgänge von Modulen
<b>assign</b> #3 a = b & c	Zuweisung (Delay: 3 Einheiten)

## 6 mips Processor

A computer architecture is defined by its instruction set and architectural state. The architectural state for the MIPS processor consists of the program counter and the 32 registers. We only consider the following instruction set:

R-type arithmetic/logic instructions: add, sub, and, or, slt.

Memory instructions: lw, sw

Branches: beq

After having built the microarchitectures with these instructions we extend them with addi and j.

We will divide our microarchitectures into two interacting parts: the datapath and the control. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. MIPS is a 32-bit architecture, so we will use a 32-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

It is often good to separate the memory into two, one containing the instructions and the other the data.

The microprocessor is built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, the processor can be viewed as a giant finite state machine, or as a collection of simpler interacting state machines.

### 6.1 Execution time

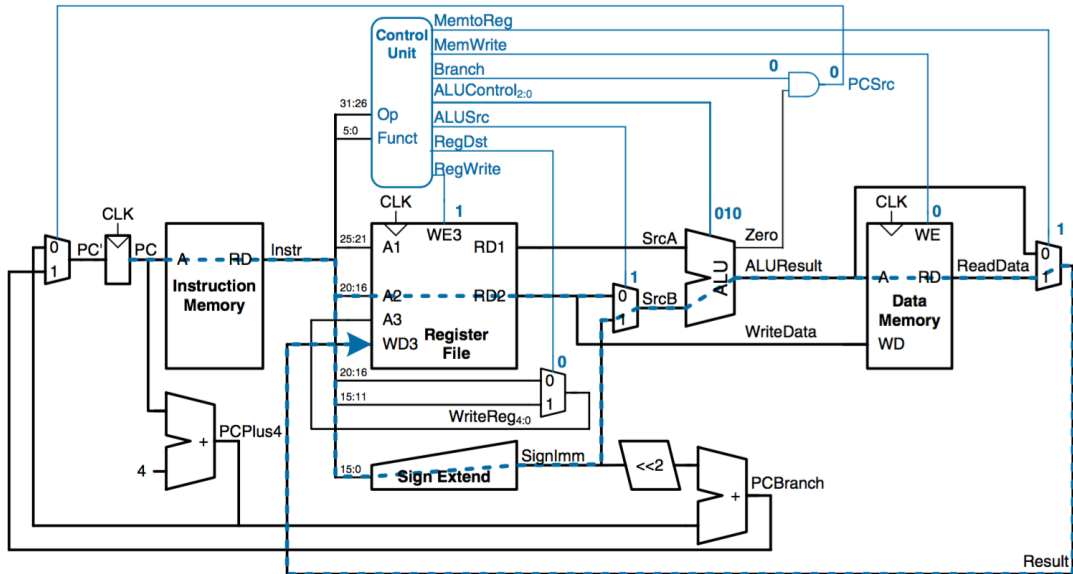
The execution time for a program is given by:

$$T = (\#instructions) * \left( \frac{cycles}{instruction} \right) \left( \frac{seconds}{cycle} \right) \quad (1)$$

### 6.2 Different architectures

#### 6.2.1 Singel-cycle

Executes an entire instruction in one cycle. It is easy to explain and has a simple control unit. Because it completes the operation in one cycle, it does not require any nonarchitectural state. However, the cycle time is limited by the slowest instruction.



##### 6.2.1.1 Performance

$$T_c = T_{pcqpc} + 2 * t_{mem} + t_{RFread} + 2 * t_{mux} + t_{ALU} + t_{RFsetup} \quad (2)$$

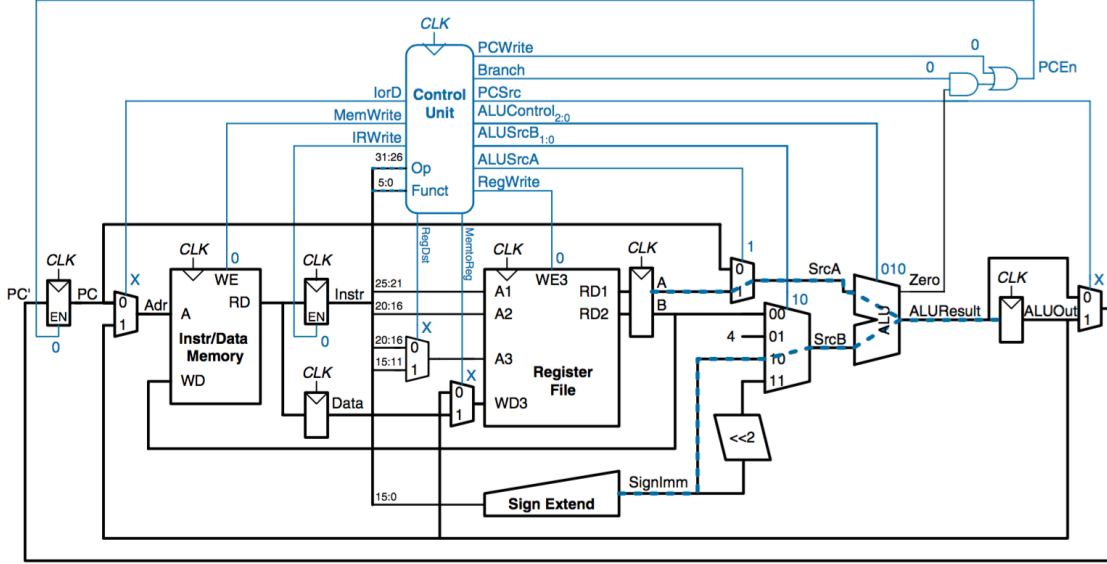
##### 6.2.1.2 Problems

The single cycle has 3 major problems:

- requires a clock cycle long enough to support the slowest instruction (lw)
- requires 3 adders which are expensive
- Has two memories one for instructions and one for data. Most computers have one.

## 6.2.2 Multicycle

executes instructions in a series of shorter cycles. In each short step, the processor can read or write the memory or register file or use the ALU. Simpler instructions execute in fewer cycles than complicated ones. Moreover, the multicycle microarchitecture reduces the hardware cost by reusing expensive hardware blocks such as adders and memories (has one adder and one memory). The multicycle microprocessor accomplishes this by adding several nonarchitectural registers to hold intermediate results. The multicycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles. The controller produces different signals on different steps during execution of a single instruction, so it is now a finite state machine rather than combinational logic.



**6.2.2.1 Performance** The multicycle processor requires three cycles for beq and j instructions, four cycles for sw, addi, and R-type instructions, and five cycles for lw instructions. The CPI depends on the relative likelihood that each instruction is used.

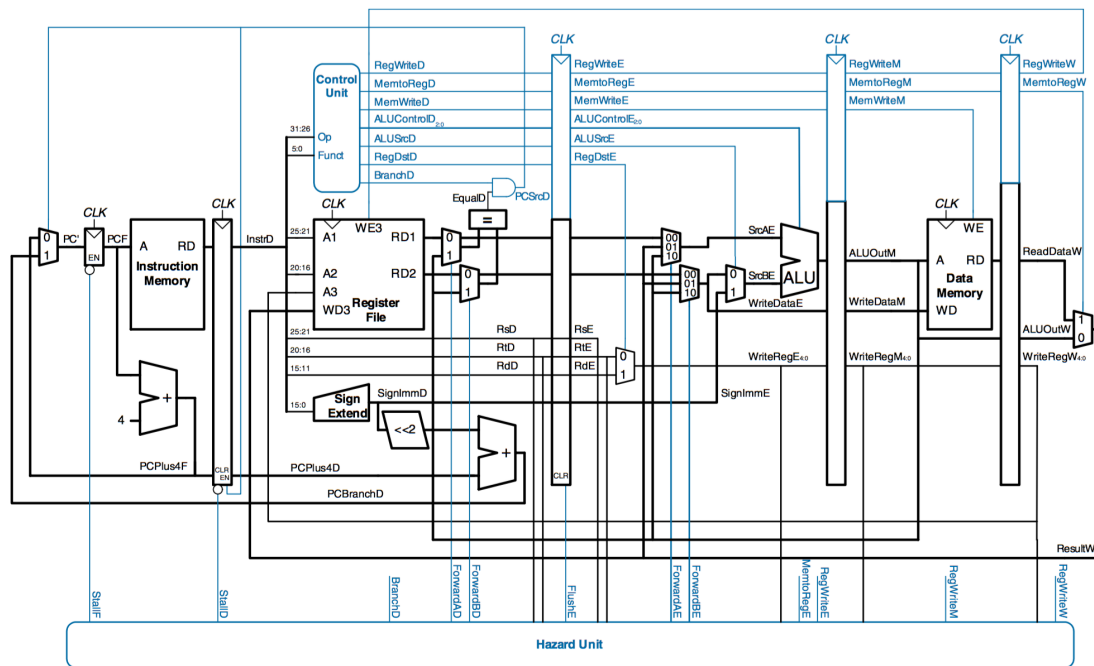
$$T_c = T_{pcq} + t_{mux} + \text{MAX}(t_{ALU} + t_{mux}, t_{mem}) + t_{setup} \quad (3)$$

## 6.2.3 Pipeline

applies pipelining to the single-cycle microarchitecture. It therefore can execute several instructions simultaneously, improving the throughput significantly. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires nonarchitectural pipeline registers. The added logic and registers are worthwhile; all commercial high-performance processors use pipelining today.

We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is almost five times faster. Hence, the latency of each instruction is ideally unchanged, but the throughput is ideally five times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Pipelining introduces some overhead, so the throughput will not be quite as high as we might ideally desire.

Reading and writing the memory and register file and using the ALU typically constitute the biggest delays in the processor. We choose five pipeline stages so that each stage involves exactly one of these slow steps. Specifically, we call the five stages Fetch, Decode, Execute, Memory, and Writeback. They are similar to the five steps that the multicycle processor used to perform lw. In the Fetch stage, the processor reads the instruction from instruction memory. In the Decode stage, the processor reads the source operands from the register file and decodes the instruction to produce the control signals. In the Execute stage, the processor performs a computation with the ALU. In the Memory stage, the processor reads or writes data memory. Finally, in the Writeback stage, the processor writes the result to the register file, when applicable.



**6.2.3.1 Hazards** Hazards are classified as data hazards or control hazards. A data hazard occurs when an instruction tries to read a register that has not yet been written back by a previous instruction. A control hazard occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. This can kind of be fixed with the hazard detection unit (by forwarding). They can also be solved by stalling the pipeline. Stalling a stage is performed by disabling the pipeline register, so that the contents do not change. When a stage is stalled, all previous stages must also be stalled, so that no subsequent instructions are lost. Stalls degrade performance.

**6.2.3.2 Performance** CPI should ideally be 1, but a stall or a flush waste a cycle.

=

## 7 Memory

Computer performance depends on the speed of the memory and the processor. DRAM (dynamic ram) is 10 to 100 times slower than processors.

Computer memories are the combination of a small fast cheap memory and a slow large cheap memory. The cache is SRAM (static) (main memory). A cache hit is when memory is retrieved from the SRAM or it's a cache miss. The third level of memory is the hard disk (virtual memory).

cache is physical memory. main memory. virtual.

### 7.0.0.1 Average memory access time (AMAT)

$$AMAT = t_{cache} + MR_{cache}(t_{MM} + (MR_{MM}t_{VM})) \quad (4)$$

MR = miss rate

If a cache misses the data found will be copied to the cache.

Two rules for what is in the cache: spatial and temporal locality. Spatial, means that when a word is copied to the cache it can also copy similar words. Temporal, means that if there is a cache miss that data will be copied to the cache because it might be searched for again.

## 7.1 Caches

Caches are organized as two-dimensional arrays. The rows are called sets, and the columns are called ways. Each entry in the array consists of a data block and its associated valid and tag bits. Caches are characterized by:

- i. capacity C
- ii. block size B and number of blocks,  $B = C/b$
- iii. number of blocks in a set N

## 7.2 Types of cache

**7.2.0.1 Direct mapped cache** Each set in the cache contains one block (one data per address). Block one of main memory maps to the set 1 of the cache and so on until the last set in the cache, then the next block in MM maps to the first set in the cache again.

**7.2.0.2 N-way set** Each set contains N blocks. Have lower miss rates, but are slower and more expensive.

**7.2.0.3 fully associative** is a cache with only one set (B-way set).

If a set is full when new data must be loaded, most of the time LRU (least recently used) block is deleted.

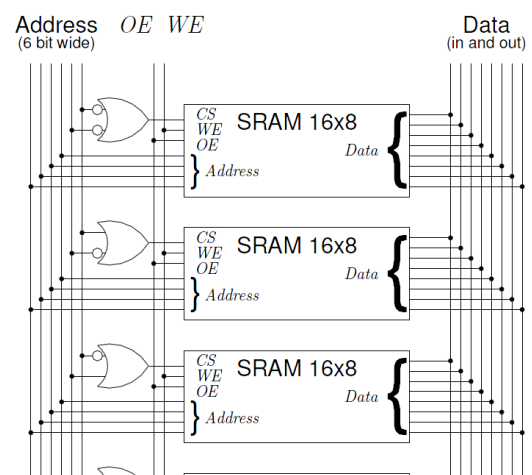
## 7.3 Multiple level caches

Most systems use multiple levels of caches. Each level is also from SRAM but bigger and therefore slower.

## 7.4 Misses

The first request to a cache block is called a compulsory miss, because the block must be read from memory regardless of the cache design. Capacity misses occur when the cache is too small to hold all concurrently used data. Conflict misses are caused when several addresses map to the same set and evict blocks that are still needed.

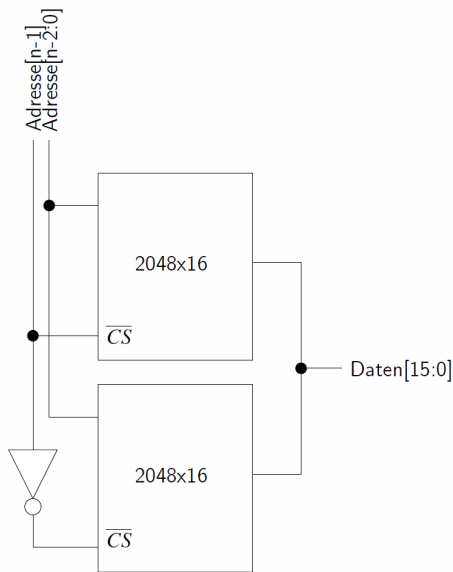
## 7.5 Kaskadierung von RAM-Chips





ram loses it's memory when the power is turned off rom doesn't.

### 7.6 64 KBit (4096x16)

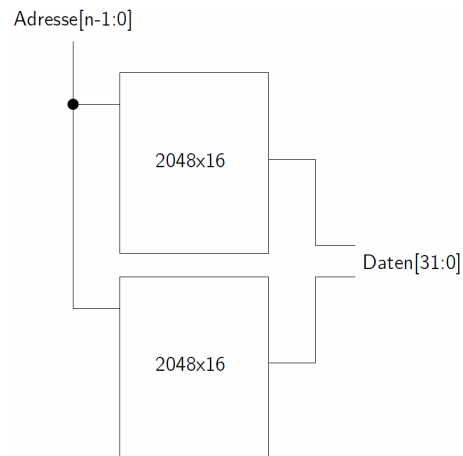


## 8 CMOS

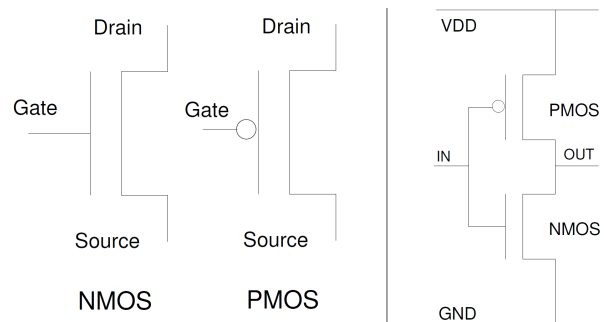
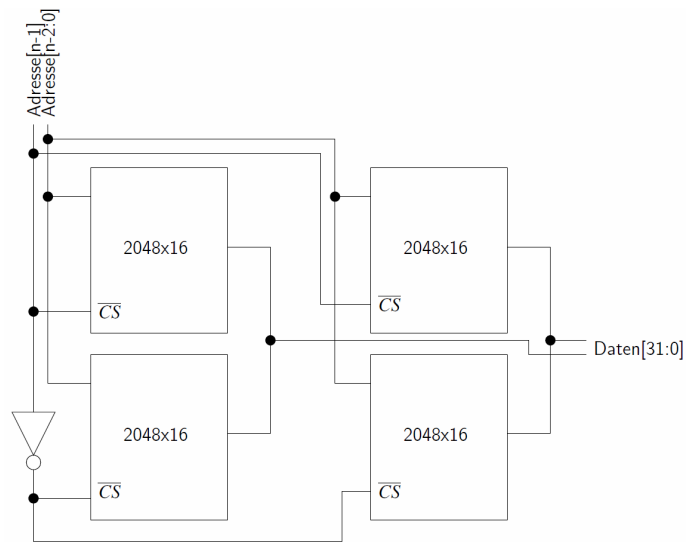
Es gibt zwei Schalttypen, NMOS und PMOS, beide werden aus Halbleitertransistoren aufgebaut. Dabei kommt dotiertes Silizium zum Einsatz, entweder p-dotiert (mit einem Überschuss an positiver Ladung) oder n-dotiert (zu viel negative Ladung).

PMOS und NMOS Schalter dürfen nicht beliebig verbaut werden, ein NMOS-Transistor funktioniert nur zwischen out und GND, während ein PMOS-Transistor nur zwischen VDD und out eingesetzt werden darf.

### 7.7 64 KBit (2048x32)



### 7.8 128 KBit (4096x32)



## 9 Binärzahlen (2er-Komplement)

00000000	0	0x00	01000000	64	0x40	11111111	-1	255	0xff	10111111	-65	191	0xbf
00000001	1	0x01	01000001	65	0x41	11111110	-2	254	0xfe	10111110	-66	190	0xbe
00000010	2	0x02	01000010	66	0x42	11111101	-3	253	0xfd	10111101	-67	189	0xbd
00000011	3	0x03	01000011	67	0x43	11111100	-4	252	0xfc	10111100	-68	188	0xbc
00000100	4	0x04	01000100	68	0x44	11111011	-5	251	0xfb	10111011	-69	187	0xbb
00000101	5	0x05	01000101	69	0x45	11111010	-6	250	0xfa	10111010	-70	186	0xba
00000110	6	0x06	01000110	70	0x46	11111001	-7	249	0xf9	10111001	-71	185	0xb9
00000111	7	0x07	01000111	71	0x47	11111000	-8	248	0xf8	10111000	-72	184	0xb8
00001000	8	0x08	01001000	72	0x48	11110111	-9	247	0xf7	10110111	-73	183	0xb7
00001001	9	0x09	01001001	73	0x49	11110110	-10	246	0xf6	10110110	-74	182	0xb6
00001010	10	0x0a	01001010	74	0x4a	11110101	-11	245	0xf5	10110101	-75	181	0xb5
00001011	11	0x0b	01001011	75	0x4b	11110100	-12	244	0xf4	10110100	-76	180	0xb4
00001100	12	0x0c	01001100	76	0x4c	11110011	-13	243	0xf3	10110011	-77	179	0xb3
00001101	13	0x0d	01001101	77	0x4d	11110010	-14	242	0xf2	10110010	-78	178	0xb2
00001110	14	0x0e	01001110	78	0x4e	11110001	-15	241	0xf1	10110001	-79	177	0xb1
00001111	15	0x0f	01001111	79	0x4f	11110000	-16	240	0xf0	10110000	-80	176	0xb0
00010000	16	0x10	01010000	80	0x50	11101111	-17	239	0xef	10101111	-81	175	0xaf
00010001	17	0x11	01010001	81	0x51	11101110	-18	238	0xee	10101110	-82	174	0xae
00010010	18	0x12	01010010	82	0x52	11101101	-19	237	0xed	10101101	-83	173	0xad
00010011	19	0x13	01010011	83	0x53	11101100	-20	236	0xec	10101100	-84	172	0xac
00010100	20	0x14	01010100	84	0x54	11101011	-21	235	0xeb	10101011	-85	171	0xab
00010101	21	0x15	01010101	85	0x55	11101010	-22	234	0xea	10101010	-86	170	0xaa
00010110	22	0x16	01010110	86	0x56	11101001	-23	233	0xe9	10101001	-87	169	0xa9
00010111	23	0x17	01010111	87	0x57	11101000	-24	232	0xe8	10101000	-88	168	0xa8
00011000	24	0x18	01011000	88	0x58	11100111	-25	231	0xe7	10100111	-89	167	0xa7
00011001	25	0x19	01011001	89	0x59	11100110	-26	230	0xe6	10100110	-90	166	0xa6
00011010	26	0x1a	01011010	90	0x5a	11100101	-27	229	0xe5	10100101	-91	165	0xa5
00011011	27	0x1b	01011011	91	0x5b	11100100	-28	228	0xe4	10100100	-92	164	0xa4
00011100	28	0x1c	01011100	92	0x5c	11100011	-29	227	0xe3	10100011	-93	163	0xa3
00011101	29	0x1d	01011101	93	0x5d	11100010	-30	226	0xe2	10100010	-94	162	0xa2
00011110	30	0x1e	01011110	94	0x5e	11100001	-31	225	0xe1	10100001	-95	161	0xa1
00011111	31	0x1f	01011111	95	0x5f	11100000	-32	224	0xe0	10100000	-96	160	0xa0
00100000	32	0x20	01100000	96	0x60	11011111	-33	223	0xdf	10011111	-97	159	0x9f
00100001	33	0x21	01100001	97	0x61	11011110	-34	222	0xde	10011110	-98	158	0x9e
00100010	34	0x22	01100010	98	0x62	11011101	-35	221	0xdd	10011101	-99	157	0x9d
00100011	35	0x23	01100011	99	0x63	11011100	-36	220	0xdc	10011100	-100	156	0x9c
00100100	36	0x24	01100100	100	0x64	11011011	-37	219	0xdb	10011011	-101	155	0x9b
00100101	37	0x25	01100101	101	0x65	11011010	-38	218	0xda	10011010	-102	154	0x9a
00100110	38	0x26	01100110	102	0x66	11011001	-39	217	0xd9	10011001	-103	153	0x99
00100111	39	0x27	01100111	103	0x67	11011000	-40	216	0xd8	10011000	-104	152	0x98
00101000	40	0x28	01101000	104	0x68	11010111	-41	215	0xd7	10010111	-105	151	0x97
00101001	41	0x29	01101001	105	0x69	11010110	-42	214	0xd6	10010110	-106	150	0x96
00101010	42	0x2a	01101010	106	0x6a	11010101	-43	213	0xd5	10010101	-107	149	0x95
00101011	43	0x2b	01101011	107	0x6b	11010100	-44	212	0xd4	10010100	-108	148	0x94
00101100	44	0x2c	01101100	108	0x6c	11010011	-45	211	0xd3	10010011	-109	147	0x93
00101101	45	0x2d	01101101	109	0x6d	11010010	-46	210	0xd2	10010010	-110	146	0x92
00101110	46	0x2e	01101110	110	0x6e	11010001	-47	209	0xd1	10010001	-111	145	0x91
00101111	47	0x2f	01101111	111	0x6f	11010000	-48	208	0xd0	10010000	-112	144	0x90
00110000	48	0x30	01110000	112	0x70	11001111	-49	207	0xcf	10001111	-113	143	0x8f
00110001	49	0x31	01110001	113	0x71	11001110	-50	206	0xce	10001110	-114	142	0x8e
00110010	50	0x32	01110010	114	0x72	11001101	-51	205	0xcd	10001101	-115	141	0x8d
00110011	51	0x33	01110011	115	0x73	11001100	-52	204	0xcc	10001100	-116	140	0x8c
00110100	52	0x34	01110100	116	0x74	11001011	-53	203	0xcb	10001011	-117	139	0x8b
00110101	53	0x35	01110101	117	0x75	11001010	-54	202	0xca	10001010	-118	138	0x8a
00110110	54	0x36	01110110	118	0x76	11001001	-55	201	0xc9	10001001	-119	137	0x89
00110111	55	0x37	01110111	119	0x77	11001000	-56	200	0xc8	10001000	-120	136	0x88
00111000	56	0x38	01111000	120	0x78	11000111	-57	199	0xc7	10000111	-121	135	0x87
00111001	57	0x39	01111001	121	0x79	11000110	-58	198	0xc6	10000110	-122	134	0x86
00111010	58	0x3a	01111010	122	0x7a	11000101	-59	197	0xc5	10000101	-123	133	0x85
00111011	59	0x3b	01111011	123	0x7b	11000100	-60	196	0xc4	10000100	-124	132	0x84
00111100	60	0x3c	01111100	124	0x7c	11000011	-61	195	0xc3	10000011	-125	131	0x83
00111101	61	0x3d	01111101	125	0x7d	11000010	-62	194	0xc2	10000010	-126	130	0x82
00111110	62	0x3e	01111110	126	0x7e	11000001	-63	193	0xc1	10000001	-127	129	0x81
00111111	63	0x3f	01111111	127	0x7f	11000000	-64	192	0xc0	10000000	-128	128	0x80

# 10 Arithmetik

## 10.1 1er-Komplement

$$\langle s_n d_{n-1} \dots d_0 \rangle_{1er} := \begin{cases} \sum_{i=0}^{n-1} d_i \cdot 2^i & s_n = 0 \\ - \sum_{i=0}^{n-1} (1 - d_i) \cdot 2^i & s_n = 1 \end{cases}$$

## 10.2 Sign magnitude

Uses the most significant bit to show if value is positive or negative, 0 positive and 1 negative. But addition doesn't work and two zeros

## 10.3 2er-Komplement

To go from one to the other you invert the bits and add 1. Watch out because doing addition, if there are not enough

## 10.5 Number systems

**10.5.0.1 Fixed point numbers** , are a representation of a fraction, they have a point somewhere in the number and after the point the exponent becomes negative. Two's complement works with fixed point numbers.

$$01101100 = 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75 \quad (5)$$

**10.5.0.2 Floating point** , are like a scientific representation, they have a mantissa(M), a base(B) and an exponent(E). 32 bits are used, 1 for the sign, 23 for the mantissa and 8 for the exponent.

## 10.6 Operations

### 10.6.1 Halbaddierer

$$\begin{aligned} s &\equiv (a + b) \bmod 2 && \equiv a \oplus b \\ o &\equiv (a + b) \div 2 && \equiv a \wedge b \end{aligned}$$

### 10.6.2 Volladdierer

$$\begin{aligned} s &\equiv (a + b + i) \bmod 2 && \equiv a \oplus b \oplus i \\ o &\equiv (a + b + i) \div 2 && \equiv a \cdot b + a \cdot i + b \cdot i \end{aligned}$$

### 10.6.3 Überlauf

$$[a] + [b] \notin \{-2^{n-1}, \dots, 2^{n-1} - 1\} \iff c_{n-1} \oplus c_n$$

### 10.6.5 Booth Recoding - Multiplikation zweier Binärzahlen ( $A \cdot B$ )

i. Wähle kürzere der beiden Zahlen als A, wähle P gleich lang wie B

ii. Erstelle Tabelle  $P_{m,\dots,0} \mid A_{n,\dots,0} \mid A_{-1} = 0$

iii. Wende folgende *Regeln* an, starte bei  $i = 0$ , Tue bei jedem Schritt:  $i := i + 1$ :

bits the addition can overflow making a negative number where there shouldn't be one.

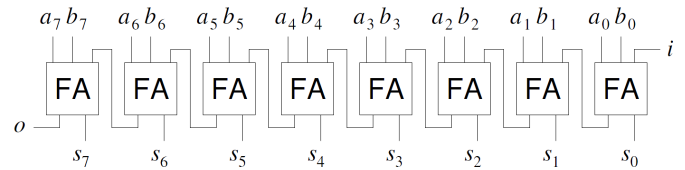
$$\begin{aligned} \llbracket s_n d_{n-1} \dots d_0 \rrbracket &= \begin{cases} \sum_{i=0}^{n-1} d_i \cdot 2^i & s_n = 0 \\ -(1 + \sum_{i=0}^{n-1} (1 - d_i) \cdot 2^i) & s_n = 1 \end{cases} \\ &= s_n \cdot -2^n + \sum_{i=0}^{n-1} d_i \cdot 2^i \end{aligned}$$

## 10.4 Lemma

$$2^n = 1 + \sum_{i=0}^{n-1} 1 \cdot 2^i$$

### 10.6.4 Ripple-Carry Addierer

Einfacher Addierer mit  $O(n)$  Zeit und Platz.



$A_i$	$A_{i-1}$	ToDo
0	0	-
0	1	addiere B zu P
1	0	addiere -B zu P
1	1	-

iv. Shifte arithmetisch nach rechts (d.h. Vorzeichen nachschieben)

v. Wiederhole (3) und (4)  $n$  (Länge A) mal

