1 GHz = 1/ns   1 GHz = 10^9 Hz
1 MHz = 10^6 Hz

4 binary digits = 1 hex digit

8 bit = 2 nibbles = 1 byte

Propagation delay: längste Zeit nach Änderung bis es den output beeinflusst

Contamination delay: kürzeste Zeit bis der output nicht mehr das Ergebnis des vorherigen cycles ist.

Mealy: output depends on the state and input
Moore: only on current state (can depend on old inputs)
FSM: Machine can be in a finite amount of states
Turing: Machine that reads from infinite tape and decides based on a finite table.

Minterm = Product (AND)   Maxterm = Sum (OR)
SumOfProducts: for each true (1) output, create a true statement. Then add them up.
ProductOfSums: for each false (0) output, create an expression that is false (for all inputs true.) Then take their product.
Product = AND   Sum = OR

Assembler: usually, first variable = Target
lw $t2, 4($t0)   load word at $t0+4 into $t2
sw $t2, -12($t0)   store $t2 at $t0-12   ← reg
/* $t0 contains address */   ← RAM
/* statt 4(...) auch 0xA(...) */
jal "jump and link" → sets $ra to next line (Adresse), jump back using jr
jal jumps to label
beq branch if equals ⇔ BNE

Stack is on ram ⇒ bei lw und sw ist das hinterer argument 0($sp)

One's Complement
replace 1 with 0 and 0 with 1
set sign (the MSB) to positive = 0 or negative = 1
Ex: -1 = 1111...10
two kinds of zeros

Sign/Magnitude
sign Msb = 1 => negative
Ex: 6 = 0110   -6 = 1110
Addition doesn't work

Addition with One's Complement
If there's a carry at the MSB, add it to the lSB
Ex:
  0001 0001   (17)
  1111 0111   (-8)
1 0000 1000
  ⟶ 1
=> 0001 001   (9)

Two's complement
Addition works // but zero-extension does not // use sign-extension
single 0
no End-Carry operation
Negative number = (reversed positive Number) + 1
=> Msb still the sign
Associativity holds

logical: && || !

Verilog: | OR, & AND, ^ XOR, ~ Not, 8'b101 (bitsize) = 8bit binary = 00000101 (can also have x or z)
~^ ~ XNOR
if (...) begin end
case(...) alpha: procedure ... endcase

sw $0, 0($sp) after addi $sp, $sp, -4

Testbench: Change input on clock but with delay

$v0, $v1 result; $a0 - $a3 arguments; $t0 - $t7 temporaries; $s0 - $s7 saved (if change, restore later!)
$t8 - $t9 more temp; $ra return adress; $sp stack for recursion
use sw $ra 4($sp)
and lw $t0 0($sp) for values to $t0

Tristate Buffer: Z = wie wenn abgetrennt; enable = 1 heisst out = Z

Multiplexer: if (s==0) {y=A;} else {y=B}

ABC DEF GHI ⟺ FDE IGH CAB (implemented using Tristate Buffers, 2-level logic or tree of 2:1 muxes)

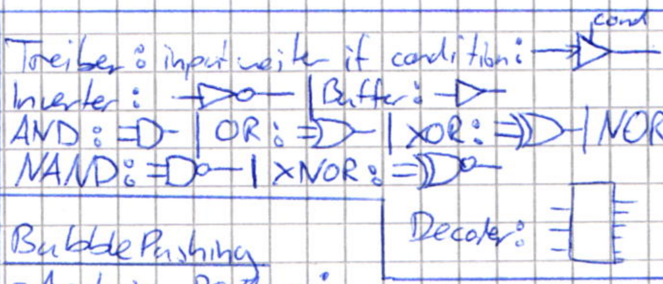
Circle don't care (X) iff it helps minimize

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

works well up to 4 vars input

Y = $\overline{S} \cdot A + S \cdot B$
(= $\overline{S} \cdot \overline{A} + S \cdot \overline{B}$) ← nicht Karnaugh map

Treiber: input wie if condition (cond)
Inverter / Buffer / AND / OR / XOR / NOR / NAND / XNOR / Decoder



Bubble Pushing
= Applying DeMorgan's
1. Change gate OR ⟺ AND
2. Add bubble where none, remove bubbles where are
Goal: Cancel many bubbles

Gray Code
2bit: 00, 01, 11, 10
reverse: 10, 11, 01, 00
prefix old with 0 and new with 1: 000, 001, 011, 010, 110, 111, 101, 100 = 3bit.

immer 0: (an Masse)
immer 1: (an Spannung)

Möglichkeit:

↑ immer 1 od. 0   Alle inputs => wie Tabelle

Sequential: has memory, outputs determined by previous and current inputs (by feeding output back to input)

Combinational: no memory, output determined by current inputs

Sequential blocking } in always block
Combinational: blocking

always@(*) is comb.

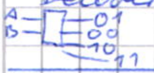
Combinational Rules:

Each circuit Element is itself combinational

Every node (wire) is either an input of the circuit or connects to exactly one output terminal of a circuit element.

No cyclic Paths, every Path visits each Node max 1

Decoders: Only one output is 1, Outputs = minterms



Verilog

module test(a,b,c,y);
input a; input c;
input b; output y;
endmodule

⟺

module test (input a, input b, output y);
endmodule

input [31:0] a; = 32-bit input bus

initial begin, similar to always, need block end

/*Case Sensitive!, Names cannot start with numbers*/
//Whitespace is ignored

Instantiation: test my_test (.a(beta), .b(...)); (outer name; inner name)

Synthesizers cannot guarantee optimal synthetisation

"Assign" precedes "=" to model dataflow but not instead of always block

&a ANDs all a[:]

Ternary (if) can be nested: assign y = S[1] ? (S[0] ? a : b) : (S[2] ? c : b);

More Operators: {a[1], a[2]} concat
% mod   <<<, >>> arithmetic shift
<<, >> shift   ~& NAND

#5; delays

Parameter in module at top: module max #(parameter width=8) (default)

call: max max1 (...);
max #(12) max2 (...);
max #(.width(12)) max3 (...);

Timing can be modeled but not synthesized

Registers: Multiple Flipflops to store more than 1 bit

when CLK=0
L1 transparent
L2 opaque
D passes through N_1
when CLK=1
L2 transparent
L1 opaque
D passes to Q

rules of synchronous sequential circuit

Each thing is either a register or a combinational circuit. At least one is a register. All regs recieve the same clock signal. Every cyclic Path contains at least one register.

Common synchronous sequential: FSMs, Pipelines

FSM consists of:
- state reg: Store current state, load next state at clock edge, sequential circuit
- next state logic: Combinational, determines next state
- output logic: combinational, determines output

can have one of a finite possible states

Words: Implicant: product (AND) of literals
Minterm: product (AND) That includes all input vars
Maxterm: sum (OR) that includes all Input variables

X: don't care, in Verilog also don't know
-: in VHDL don't know
Z: No specific value, "floating"
Ex: buffer with unsatisfied condition

Reasons for Contamination Delay ≠ Propagation delay
- Different rising & falling Delays
- Multiple in- and outputs, some faster
- Circuits slow down when hot, speed up when cold

Glitch

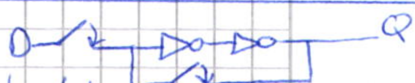
One variable input changes differently timed output => temporarily wrong output

Happens when in Karnaugh map switches to different rectangle ("prime implicant")

Fix: add redundant rectangle and thus circuit parts over that border.

Can't get rid of all glitches because some happen on simultaneous transitions on multiple inputs

D Latch



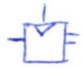
To change state, break loop and set new input, then close loop. Called "transparent mode" if loop broken, input propagates to Q

so-called bi-stable circuit.

Opposite of transparent is "Latch mode"

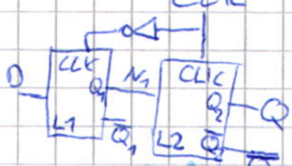
Clock 1 => transparent
Clock 0 => Latch mode

FlipFlop



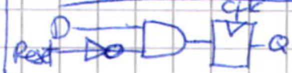
on posedge clk, samples D
Q stays D until next posedge

Two back-to-back Latches



Enabled Flipflops have an EN-input to set write to Enabled

Resettable Flipflop



synchronous

asynchronous requires changing internally

settable

Q=1 if S=1
else normal Q=D

Onehot-encoding: 0010 or 0100 or 1000 or 0001
more flipflops needed

binary encoding: 01 00 10 11
more nextstate and output logic needed

Designing FSM:
1. Identify in- and outputs
2. Sketch transition diagram
3. write state transition table
4. select state encodings
5. moore/mealy: rewrite state transition table with selected encodings
6. only moore: write output table
7. write boolean equation for nextstate and output logics
8. sketch circuit schematic

Moore/Mealy:

alwaysBlock

assign is not used
possible to use both blocking and nonblocking
from "always" Assigned variables need to be reg:
module flop(input clk, input [3:0]d, output reg [3:0]q)
negedge, posedge
"always" needs begin ... end
if ... then ... else and case(expression) are possible
result is combinational if defined for all cases
expression: statement
default: statement
casez ← check for don't cares

use non-blocking to model synchronous sequential

Reg — Combinational — Reg

Setup time for D: $T_c \geq t_{pcq} + t_{pd} + t_{setup}$ (reg, propagation delay, per gate)

Hold time for D: $t_{hold} < t_{ccq} + t_{cd}$ (contamination, per gate)

for setup time: $T_c = \frac{1}{f_c}$

Fix hold time: add buffers to short paths

Latency: time for 1 Token
Throughput: Tokens per time
is different e.g. in pipeline

Metastability

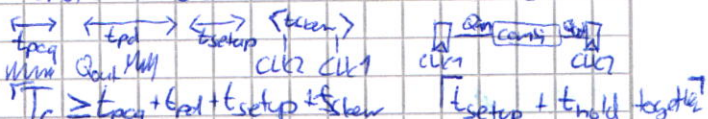
T_0/T_c = probability of metastate. So usually if you wait long enough, the metastate will go away in a flipflop

Synchronizer minimizes t_c chance but not to 0: It is built using two back-to-back Flipflops. As normal $P_{fail} = \left(\frac{T_0}{T_c}\right) \cdot e^{\frac{t}{\tau}}$, two Flipflops make $P_{fail2} = \frac{T_0}{T_c} \cdot e^{t - \frac{T_c - t_{setup}}{\tau}}$

If asynch input changes N/second
$\Rightarrow P_{synchronizer} = P_{fail2} \cdot N$ $MTBF = \frac{1}{(P_{fail2} \cdot N)}$ mean time between failures

Setup time with clock skew

Worst case: Clock skew → CLK2 is before CLK1



$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$

$t_{setup} + t_{hold}$ together is called aperture time

Hold time with clock skew

worst case: CLK2 later than CLK1

$t_{hold} + t_{skew} \leq t_{ccq} + t_{cd}$

This whole stuff is called "Dynamic Discipline"

Carry Save Adders reduce 3 N-bit inputs to 2 N-bit inputs with a single gate delay

Multiplier like Decimal multiplication
1. to find partial products
2. to reduce using Carry Save Adders
3. to sum using Carry Propagate

Adder can be programmable to subtract
Can implement Comparator by adding Complement.

Fast Adders aka Lookahead Carry adder
Calculates simultaneously if Carry will propagate to next Group if exists(carry)
⇒ can tell next group early that there will be a carry

Negator: TwosComplement(A) $\overline{A} = \overline{A} + 1$

Arithmetic Shift: Not based on Adder.
Like Shift, but on rightshift fills in the old MSB. >>> Arithm, >> normal

Rotator: Bits that fall off right enter Left again
ROR or ROL

Shift-Register Serial input, shift one entry in and one out. Also enables
Serial in → parallel out
Add a Multiplexer to switch between parallel in and serial in
⇒ can act as parallel→serial

ALU Example:

$F_{2:0}$	
000	A&B
001	A|B
010	A+B
011	
100	A&~B
101	A|~B
110	A-B
111	SLT

Fast Adder
options:
CarrySkip
CarryIncrement
CarrySelect
CarryLookahead

Memory: SRAM, DRAM (cache, main, virtual)

Memory Array
only access m bits (a "word") of the same time ⇒ more compact

Decoder: Activates one row

Transistors configured as switches

NOTE: ROM is much denser than RW

Memory
Flipflops or Latches: very fast, parallel access. Expensive (20+ transistors per bit)
Static RAM: relatively fast, but only one data word at the time. Less expensive (one bit = 6 transistors)
Dynamic RAM: slow, reading destroys content (needs refresh), only one D-word at the time, needs special process, cheaper
Harddisk/Flash: Much slower, nonvolatile*, low per-bit-cost (no transistors)
*stays without power

Fixed-Point: 4 digits, then 3, use 2^n to calculate value (has to be consent)

Signed Fixed Point: use sign or two's complement

Rounding modes: up, down, towards zero, to nearest

Floating point

① $\pm 1.01001 \cdot B^E$ => [1 | 8 | 23] (Sign: positive = 0, Exponent, Mantissa: filled in from left to right)

② Because first Bit is always 1, don't need that one in Mantissa: ~~1~~0100100000...

③ Biased: 8 bits → 2^7 and use minus 1 => ($2^7 - 1$ + actual exponent)

Now, Floating point numbers can be compared while encoded to tell which is larger. Two's complement would've made that harder. Comparison can be done with fixed-point hardware.

To interpret, subtract the bias again: for [-126, 127] 127

double precision for [-1022, 1023] 1023

quad-precision: [-16382, 16383] 16383

0 = x 00000000...

∞ = 0 11111111 00...

-∞ = 1 11111111 00...

NaN = x 11111111 non-zero

Problems: Overflow and underflow are possible.

Addition:

1. Extract stuff, add leading 1. to Mantissa
2. compare exponents ($E_1 - E_2$ = shift)
3. shift smaller Mantissa if exponents differ
4. Add/subtract (based on sign) Mantissas
5. Normalize Mantissa & adjust exponent
6. Round if fits not in 23 bits
7. Package up again

in Hex: Just transform like nothing special

in ARM-Processor: 32 32-bit regs for Floats ($f0-$f31). Double precision stored using 2 regs (not in MIPS)

Big-Endian vs Little Endian (bits still same order in byte): Numerate bytes starting from MSB vs LSB

F-Type instruction

op	cop	ft	fs	fd	funct
6 bits	5	5	5	5	6

op = 010001 (17)

cop = 16 for single prec, 17 for double

fs, ft = source operands

fd = destination

R-Type

op	rs	rt	rd	shamt	funct
6	5	5	5	5	6
0	source regs		destination	shift amount or 0	

I-Type

op	rs	rt	imm
6	5	5 (destination)	16

2^{32} bit = 2^{30} byte = 2^{20} kbyte = 2^{10} Mbyte = 1 GB

J-Type

op	addr
6	26

In MIPS, Programs typically start at PC = 0x00400000

jr $s0 jumps to address

for less than ifs, use slt followed by beq with 0

A Word has 4 bytes

Stack at $sp resizes (itself?) to need

Make space on sp: addi $sp, $sp, -4

write $s0 on it: sw $s0, 0($sp)

deallocate: addi $sp, $sp, 4

Program is stored on ROM; 2^{32} bits in Mem → 32-bit addr

Stack above $sp must be preserved

Stack is stored on top, normal data on bottom of Dynamic Data

Below that is static data and text

Lui/ori: initialize array to addr

```
lui $s0, 0x1234 #upper $s0
ori $s0, $s0, 0x8000 #lower $s0
#addr is 0x12348000
lw $t1 0($s0) #first item
lw $t1 4($s0) #second item
#array has size 32 bit
```

Lui followed by ori simply stores address to array in $s0. Because that addr is > 16 bit

At Each adress is 16 bit space

Mips Datapath

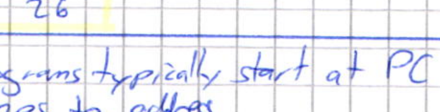
control signals:

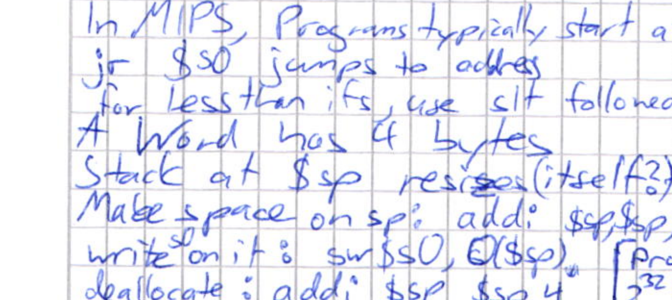
- ALU Inputs: rt or immediate
- Write Addr of regfile: rd or rt
- Write Data in of regfile: ALU out or Data Memory out
- Write Enable of regfile: not always a register write
- Write Enable of memory: Only when writing memory (sw)

=>

Type	AluOp / Funct	Op	Reg Write	Branch	Reg Dst	Alu Src	Mem Write	Jump	Mem to Reg
R	funct	0	1	0	1	0	0	0	0
lw	add	35	1	0	0	1	0	0	1
sw	add	43	0	0	X	1	1	0	X
beq	sub	4	0	1	X	0	0	0	X
J	X	3	0	X	X	X	0	1	X
addi		16	1	0	0	1	0		0

PC: 32-bit reg



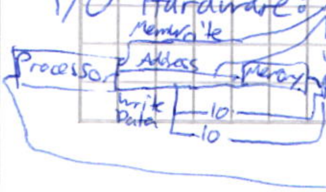


Control Unit: opcode in → all control signals out

Funct → Alu Decoder ← AluOp

↳ AluControl

I/O Hardware: Address Decoder



I/O Registers hold values written to IO

ReadData Multiplexer

Program executes in $N \times CPI \times T_{Clock}$

Faster: CISC (Less instructions for same program)

Use better Compilers ↑

RISC (Less cycles per instruction)

Multiple units in parallel ↑

Pipelining, newer technology (increase clock frequency)

Average Memory Access Time $= t_{cache} + MR_{cache}(t_{main} + MR_{main}(t_{virtual\,m}))$ ← miss rate

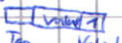
Capacity C, Block size b, Number of blocks $B = C/b$
Degree of associativity (N), Number of sets $S = B/N$ | $N = B/S$
Each mem. addr addresses exactly one set in cache

"Direct Mapped Cache" => 1 block/set
"N-way set associative" => N blocks/set
"Fully associative" => all blocks in one set

Direct Mapping

address: 

Block in cache = "set"

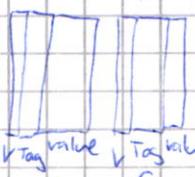
=> cache 

at cacheblock determined by mod

31-11 Tag | 10-5 Slot | 4-0 offset

N-way "associative cache"

N blocks/set



Fully associative

Expensive to build

Larger Blocks => less compulsory misses on first load
"Capacity Miss": Full but want to store + in cache ← reduced by bigger cache
"Compulsory Miss": Cache empty ← reduced by bigger Block size but bigger Blocks increase
"Conflict Miss": Data maps to same location but is different one than here ← reduced by greater associativity

Least recently used data in set (or unvalid) gets replaced
Block in cache <=> "page" in Virtual Memory

Ex: System
VM $2GB = 2^{31}B$
Physical $2^{27}B$
Page Size $2^{12}B$

Organisation
31 bit virtual addr.
27 bit phys. addr.
12 bit Page offset
Virtual Pages $= 2^{31}/2^{12}$
Phys Pages $= 2^{27}/2^{12}$

Page Table to locate addr in physical mem
Entry for each virtual page
Each entry has a "valid bit": whether is (or in HDD) on physical
and the physical page number
Ex: virtual 0x5F20 → find 0x0001 in table
→ append offset → physical 0x1F20
Each Process has it's own Pagetable

Translation Lookaside Buffer
Small cache of recent page translations, reduces for most loads/stores # accesses from 2 to 1. Usually < 512 entries. Fully associative

Single Cycle
speed determined by longest path
Two Mems: Instructions & Data
Three Adders: ALU, PC, Branch
Simpler instructions might be faster than longest path (1 cycle)

Multi Cycle
CPI = Average
need only one Memory, less Adders => less space
⊖ sequencing overhead

Carry skip Adder
set propagation criteria $p = a \text{ XOR } b$
=> if C_{in}, it'll propagate,
& p_{3-4} => if equals 1 => last C_{out} = first C_{in}
else calculate last C_{out}

Carry Lookahead
$C_0 = C_i$
$C_1 = G_0 + P_0 C_i$
$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_i$
$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_i$
$G = A \cdot B$, $P = A + B$

Carry-Increment-Adder
Assume $C_{in} = 0$
later increment if C_{in} would've been 1

Carry-Select Adder
Parallel 2 adders, one assumes 0, the other 1

Parallel-Prefix Adders
all have pre- and postprocessing
is a model

Ripple Carry Adder
Most efficient
Others: BK offers good compromise
SK has high fan-out
KS and HC are fast but costly for routing

Pipelining
Data Hazard: Data not computed yet
write happens in first half of cycle
read in second
- Fix Data hazards: insert nops at compile time
- rearrange code at compile time
- Forward data at runtime (sufficient for RAW but not for lw, which reads until end of Mem stage)
- Stall the processor at runtime

lw takes "2" cycles => stalling if read directly after
Stalling: Enabled by using EN-switches for fetch and decode, and a synchronous reset
=> Decode and Fetch hold their old values
Execute is flushed

Forwarding: If dest. reg == src reg forward data
If both Mem and Writeback are same dest. notion, give Mem priority

Pipelining v2

pipelining is faster than normal or multicycling

Control Hazards: not yet known what instruction
e.g. cause of beq
⇒ predict and flush if wrong. Backwards (loops) are usually taken
if wrong guess → penalty of flushing
Load/Branch take 2 CPI if
directly read after or if wrong guess, else 1
⇒ calculate CPI of processor/program using Probability

Pipeline: fetch → decode → execute → memory → writeback
↑ use regfile and only half a clock cycle to complete

$T_c = \max\{ t_{pcq} + t_{mem} + t_{setup},$ fetch
$2(t_{ctreadd} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}),$ decod
$t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup},$ execute
$t_{pcq} + t_{memwrite} + t_{setup},$ memory
$2(t_{pcq} + t_{mux} + t_{RFwrite})\}$ writeback

Symmetric Multiprocessing

Multiple cores, shared Mem
Contrary to Async Multiprocessing
Clusters: each core has own memory
Async: shared memory, each core a different Task.

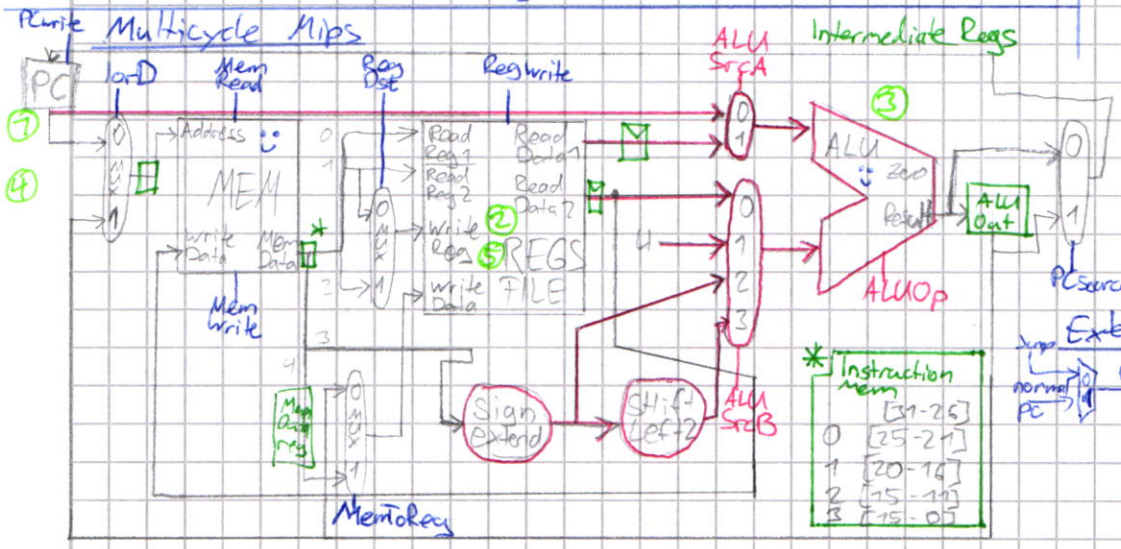
Revisit Multicycle

IF = Instruction fetch, MEMory,
ID = Decode, EXecute, WriteBack

beq	IF	ID	EX		
R	IF	ID	EX		
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

second cycle

Multicycle Mips



Cycles:

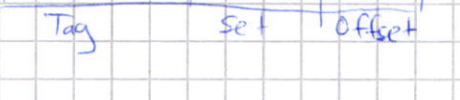
1. fetch & PC increment
2. read src from REGFILE
3. Perform ALU comp
4. read or write memory
5. store data back to reg

Byte Caching (Associative):

Block size $= 2^b$ eg. 4 ⇒ offset = b = 2

#sets $= 2^a$ eg. 4 ⇒ a = set = 2

Tag = Restliche bytes der Adresse

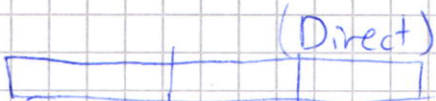


(Direct):

offset funktioniert gleich

Set: wenn a sets, $a = 2^b$, dann sind esdie b bits d. Addr.

Tag: rest



Float single: 1 | 8 | 23 — $\pm M \cdot B^E$ — Mantisse ersten bit ignorieren weil immer 1
sign Exp. mant — Exponent 127 = $2^8 - 1$ hinzufügen
double 1 | 11 | 52 — bias 1023 — bias
sign 1 negative