1) Numbers:

	14	13	12	11	10	9	8	7	6	5	4	3	2	1
2 hoch 0 = 1	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1 = 2															1
2 = 4														1	0
3 = 8													1	1	0
4 = 16												1	0	0	0
5 = 32											1	0	0	0	0
6 = 64										1	0	0	0	0	0
7 = 128									1	0	0	0	0	0	0
8 = 256								1	0	0	0	0	0	0	0
9 = 512							1	0	0	0	0	0	0	0	0
10 = 1024						1	0	0	0	0	0	0	0	0	0

11 = 2048, 12 = 4096, 13 = 8192, 14 = 16384, 15 = 32768, 16 = 65536, 17 = 131072, 18 = 262144, 19 = 524288, 20 = 1048576

21 = 2097152, 22 = 4194304, 23 = 8388608, 24 = 16777216, 25 = 33554432, 26 = 67108864, 27 = 134217728, 28 = 268435456, 29 = 536870912, 30 = 1073741824

Hex: $ABC = A \cdot 16^2 + B \cdot 16 + C$

$16^0 = 1$, $16^1 = 16$, $16^2 = 256$

$16^3 = 4096$, $16^4 = 65536$, $16^5 = 2^{20}$

0-9, A=10, B=11, C=12, D=13

E=14, F=15

Unsigned: nur positiv
↳ Range: $0, 2^n$

Sign-Magnitude: erstes bit 0=+, 1=-
↳ vorteil: -(+) nur ein bit unterschied
↳ nachteil: 2 null, schwierige (addition / subtraktion)
↳ Range: $-2^{n-1}+1 \dots 2^{n-1}-1$

Ones' complement: bitwise not
↳ Range: $-(2^{n-1}-1) \dots 2^{n-1}-1$
↳ Nachteil: 2 null → 000 und 111 → braucht carry für addition

Twos complement: -> + same circuit for adding + and - numbers
↳ Range: $-(2^{n-1}), 2^{n-1}-1$ nur 1 null, einfache addition
↳ 8 bit: -128 bis 127
00000000 = 0, 00000001 = 1, 01111111 = 127, 11111111 = -1, 10000000 = -128, 11111110 = -2

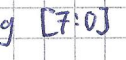
2) Gates & Verilog:

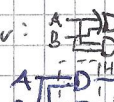
AND	OR	NOT	NAND	XOR	NOR	XNOR
&	|	~	~&	^	~|	~^

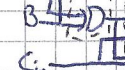
a	b	OR	AND	NOT a	NAND	XOR	NOR	XNOR
0	0	0	0	1	1	0	1	1
0	1	1	0	1	1	1	0	0
1	0	1	0	0	1	1	0	0
1	1	1	1	0	0	0	0	1

NAND - Gates	Target	NOR - Gates
	NOT	
	AND	
	OR	
	XOR	

Verilog: reg [7:0] name [0:31] //32 element, 8 bit wide
Memory

Half-Adder: 
$S = (A \hat{} B)$
$Cout = (A \& B)$

Full-Adder: 
$Cout = (A\&B) | ((A \hat{} B) \& C_{in})$
$AB + AC_{in} + BC_{in}$
$S = C_{in} \hat{} (A \hat{} B)$
$A \oplus B \oplus C$

Multiplexer (MUX): 

S	Y
0	D_0
1	D_1

Combinational Logic (CL)

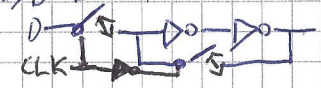
↳ Memory less ↳ Output determined by current input
↳ no cycles ↳ every Element is CL ↳ z.B. Full Adder

Sequential Logic

↳ Has Memory ↳ Output determend by current & past input values
↳ z.B. FSM oder Pipeline

↳ Asynchron: moves if circuit finished
↳ Synchron: moves on CLK

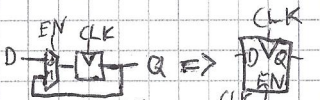
Memory:
↳ Transparent: input propagates to output
↳ Latch/Opaque: output is stored
↳ D-Latch



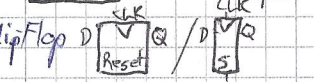
↳ Flip-Flop: combines two latches as master/slave

↳ Registers (array): multiple parallel Flip-Flops

↳ Enabled Flip-Flops



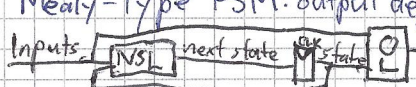
↳ Resetable/Setable-FlipFlop
↳ reset? Q=0 / ↳ set? Q=1



3) Finite State Machine (FSM):

Moore-Type FSM: output depends only on state

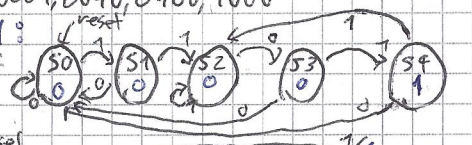
Mealy-Type FSM: output depends on current state and inputs



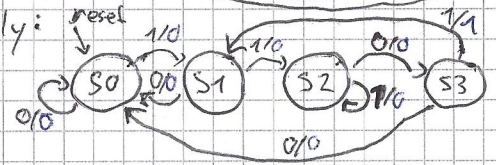
State encoding: (z.B. 4 states)
↳ normal: 00, 01, 10, 11
↳ One-Hot: 0001, 0010, 0100, 1000

Read: 1101:

Moore FSM:



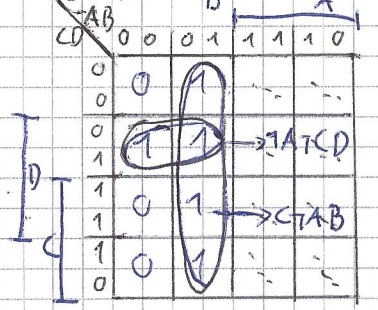
Mealy:



Things to keep in mind with FSM:
1) All states need identifying name or labels
2) There need to be a reset
3) Every state has to be Moore or Mealy, no mixing!
4) With Mealy every transition need an output, with Moore every state
5) Each value can only correspond to one transition
6) Every Transition needs an associated value
7) Every possible input needs one transition
8) If the input doesn't mater, assign X to it.

15-937-717 pwacker
Pascal Wacker 1.6

Karnaugh-Veitch-Diagramm (Grey code -> only one bit change)
↳ Grouping: - rectangular
- # is 2^x, $x \in \mathbb{N}_0$



A	B	C	D	E
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1

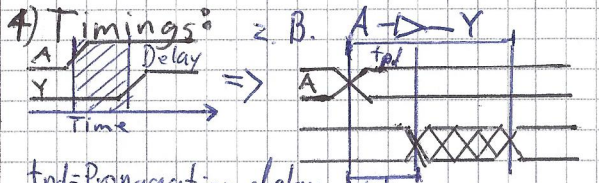
and so on...

$E = \neg A \& \neg C \mid \neg A \& D \mid \neg A \& (B \vee C \vee D)$

Minterm: Product (AND), true stays, fals is inverted
Maxterm: Sum (OR), false stays, true is inverted
Sum-of-Products: OR-ing minterms that are true
Product-of-Sums: AND-ing maxterms that are false

4) Timings:

z.B. A -> Y

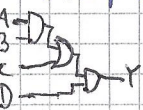


t_{pd} = Propagation delay
↳ max delay between input & output

t_{cd} = Contamination delay
↳ min delay between input & output

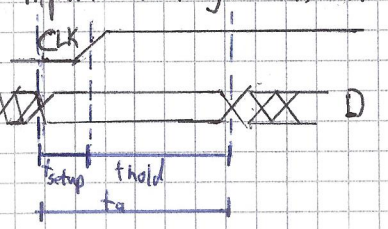
Caused by:
↳ Capacity & Resistance in circuit
↳ Speed of Light limitation

Reason for different t_{pd} and t_{cd}:
↳ different rising & falling delay
↳ Multiple inputs and outputs (some faster)
↳ Temperature (slow when hot, fast when cold)



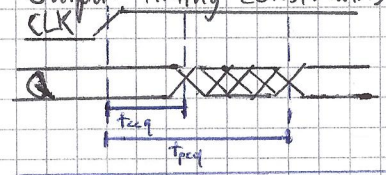
Critical Path (t_{pd}) = $2 \cdot t_{pd\_And} + t_{pd\_Or}$ (A,B) -> (Y)
Shortest Path (t_{cd}) = t_{cd_And} (D) -> (Y)

Input timing constraints



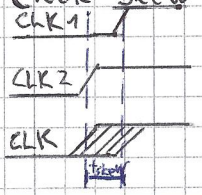
t_{setup}: time before clock edge where data must be stable
t_{hold}: time after clock edge where data must be stable
t_a: Aperture: time around clock edge, where time must be stable $t_a = t_{setup} + t_{hold}$

Output timing constraints



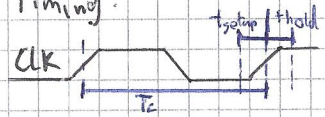
t_{ccq}: Contamination Delay
↳ time after clock edge output Q might be unstable
t_{pcq}: Propagation Delay
↳ time after clock edge output Q guaranteed to be stable

Clock Skew: Difference in clock between two registers



t_{skew}: Clock Skew

Timing:



- Clock period time: T_c
- Clock frequency: $f_c = 1/T_c$
  ↳ f_c measured in Hertz (cycles per second) 1MHz = 10^6 Hz, 1GHz = 10^9 Hz

$T_c \geq t_{pcq} + t_{pd} + t_{setup}$
$t_{hold} \leq t_{ccq} + t_{cd}$

$t_{pd} \leq T_c - (t_{pcq} + t_{setup})$
$t_{cd} > t_{hold} - t_{ccq}$

5) MIPS: (RISC-Type Microcontroller)

R-Type: Register Type, add rd, rs, rt ⇒ rd = rs + rt

Op	rs	rt	rd	shamt	func
6bit	5bit	5bit	5bit	5bit	6bit
0 for R	source		destination	shift amount	what function

I-Type: Immediate Type, addi rt rs imm; lw rt imm(rs)

Op	rs	rt	imm
6bit	5bit	5bit	16bit, two complement immediate

↳ constants = immediates (da)

J-Type: Jump Type

Top	addr
6bit	26bit

j und jal sind einzige j-type

Memory: byte addressable memory

Word Address	Data					Word
0x4	F2	F1	AC	07	Word 1	
0000 0000	AB	CD	EF	78	Word 0	
	MSB 0	1	2	LSB 3	← Big-Endian: MSB → LSB	
	3	2	1	0	← Small-Endian: LSB → MSB	

lw $s3, 4($0)
↳ F2 F1 AC 07
sw $t4, 0x0($0)
↳ write $t4 to Word 0

Name	Register Nr.	Usage	Preserved ↳ Callee-saved	Non Preserved ↳ Caller-saved
$0	0	constant val 0		
$at	1	assembler temp		
$v0-$v1	2-3	return values		X
$a0-$a3	4-7	arguments		X
$t0-$t7	8-15	temp		X
$s0-$s7	16-23	saved vars	X	
$t8-$t9	24-25	temps		X
$k0-$k1	26-27	OS temp		
$gp	28	global pointer		
$sp	29	stack pointer	X	
$fp	30	frame pointer		
$ra	31	return address	X	

MIPS PC starts at 0x0040 0000

+Stack above $sp, stack below $sp

MIPS-Memory Layout:

max. 2^{32} = 4GB from 0x0000 0000 to 0xFFFF FFFF

In Memory: - Instructions (text)
- Data (Global/Static) and Dynamic (allocated in program)

Reserved	0xFFFF FFFC
	0x8000 0000
↓ Stack	0x7FFF FFFC
Dynamic Data	
↑ Heap	0x1001 0000
	0x1000 FFFC
Static Data	0x1000 0000
	0x0FFF FFFC
Text ↑	0x0040 0000
	0x003F FFFC
Reserved	0x0000 0000

```
addi $sp, $sp, -8 # allocate 2 reg
sw   $ra, 4($sp)  # save $ra
sw   $s0, 0($sp)  # save $s0
# do stuff
lw   $s0, 0($sp)  # restore $s0
lw   $ra, 4($sp)  # restore $ra
addi $sp, $sp, 8  # deallocate space
```

6) Processor:

- Single Cycle:
  ↳ Each instr. takes 1 Cycle
  ↳ All state updates at end of cycle
  ↳ Cycle time defined by slowest instr. (slow)
  ↳ long clock cycle time

- Multi Cycle
  ↳ Each instr. broken in multiple cycles
  ↳ State update during cycle, architectural state update at end of instruction.
  ↳ Cycle broken in stages, cycle time defined by slowest instruction (fast)

Both follow "von Neumann" model → Instr. & Data Memory is one
→ sequential instr. processing

Microinstr.: Controlsignal associated with current state
Microsequence: transition from one state to another, determin next state and its Microinstr
Controlstore: stores control signals for all possible states
Microsequencer: Determins next Microinstr. enables easy extensibility of ISA and updates

- Pipelining: use Hardware better
  ↳ Steady State: Pipeline is full
  ↳ Speedup: Optimal = 1/(CT/k + S) k = # stages, S = latch delay
  ↳ Cost increase → additional HW T = non pipelined

Instruction Cycle

- ↳ Fetch
- ↳ Decode
- ↳ Evaluate Addr.
- ↳ Fetch Operands
- ↳ Execute
- ↳ Store Results

1) Instr. Fetch (IF)
2) Instr. Decode & Registr. Operands Fetch (ID/RF)
3) Execute/Evaluate Mem Adr. (EX/AG)
4) Mem Operand Fetch (MEM)
5) Store/Write back Result (WB)

Problems:

- Resource contention:
  ↳ Instr in multiple stages need same resource
  ↳ solution:
    ↳ more resources (multiple ports)
    ↳ stall one instr.
- Data-Dependencies
  ↳ Flow dependences (true dependency → read after write)
  ↳ Output dependences (write after write)
  ↳ Anti dependences (write after read)
- Solution (mostly to flow): → detect & wait, oder detect & forward/bypass
  → detect & eliminate
  → predict & verify (or time graded multith.)

Flow (RAW): $r_3 \leftarrow r_1$ op r_2; $r_5 \leftarrow r_3$ op r_4
Output (WAW): $r_3 \leftarrow r_1$ op r_2; $r_4 \leftarrow r_5$ op r_2; $r_3 \leftarrow r_6$ op r_7
Anti (WAR): $r_3 \leftarrow r_1$ op r_2; $r_1 \leftarrow r_4$ op r_5

Fine grained-Multithreading (Switch to other thread every cycle)
+ (no) logic needed for control & data dependencies within thread
+ no need for branch prediction
+ improve system throughput, latency tolerance, utilisation
- single thread performance suffers
- extra logic/HW for thread context
- Resource contention between threads (caches/Memory)
- dependency checking logic between threads (load/store)

Dependency Detection

- scoreboard → each register has valid flag (otherwise stall)
  + simple - stall all types not only flow
- combinational → check instr. in later stages for RAW
  + no stall for WAW/WAR - complex/expensiv logic
- Forwarding/Bypassing → shortcut from Mem/WB, mem has pri

Control Dependencies: (special data dependency)
BEQ allways execute next sequential (assumeme branch not tk
↳ branch prediction in 4th stage, if taken flush pipeline

Interupts: external to thread, handle when convinient
Exception: internal to thread, handle when detected
↳ All prev instructions should be commited, no later
Precise Exception: WB is stalled to apear in sequential order
Imprecice Exception: WB happens directly after execution → hazzard on exception

Out-of-Order Processor:

Execute instr. as you please, reorder them after execution using a ROB (Reorder Buffer)

7) Tomasulo's Algorithm:

Uses Register Alias Table (RAT), Functional Units (FU) and Reservation Stations (RS), RAT & RS have valid flag and value fields

1) Decode operation, put operants in RS, if data in RAT is valid use it, otherwise store tag
2) If all operants of RS are available, pass to FU
3) After execution brodcast value from FU to RS and RAT

8) Other Execution models:

- VLIW: Very Long Instruction Word:
  ↳ one instruction word contains several instructions which are executed on seperate FU
- SIMD: Single Instruction Multiple Data
  ↳ Array Processor: Instruction operates on multiple data elements of the array at once
  ↳ Vector Processor: Instruction operates on multiple data elements at the same space in consecutive time steps
  + no dependencies within vector
  + each instr. generates lot of work → lower instr. fetch bandwith
  + highly regular memory access
  + no need for loops
  - works only if parallelism is regular
  - Memory bandwith as bottle neck

Control Registers: VLEN, VSTR, VMASK (used for conditions)
Vector FU: deeply piplined & fast CLK, loops are vectorizable if independent.
Vector Chaining: Forwarding for Vectors

GPU:

- Instruction Pipeline operates like SIMD pipeline, but programming is using threads

SIMT (Single Instruction Multiple Threads)
↳ Multiple instruction streams of scalar instructions
→ can treat each thread seperately
→ can group threads into wrapps at will, to maximice SIMD throughput.
↳ Warp / wave front: A group of threads executing the same instruction → basically SIMD by HW
↳ Branching in Warp: Prediction by GPU using mask (similar to VMASK) to only operate on correct data. Done by HW, not like vectors done by SW Programmer!
↳ Improving SIMD utilisation: find threads at same PC and groupe them.

Programming- vs. Execution Model

Programming Model: how Programmer expresses Code: Sequential (von Neumann), Data Parallel (SIMD), Data Flow, Multithreads (MIMD, SPMD)
Execution Model: how HW executes: Out-of-Order, Vector-/Array-Processor, Dataflow Processor
→ no need to match, ex: sequential on O-o-O Processor

9) Memory:

- Flip-Flops and Latches:
  + Fast, parallel access — Expensiv (20+ transistors/bit)
- Static RAM (SRAM):
  + Relative Fast, only one data word at a time
  - still expensive (6 transistors/bit)
- Dynamic RAM (DRAM)
  + cheap — slower, reading destroys content (refresh), one data word, need special process
  1: ↓ : 0
- SSD, HDD, usw. + cheaper - slower

Memory Banking: Used to resolve long memory latency. All "n" memory banks share one data bus and one address bus.
Fetch from bank 0, bank 1, ..., bank n-1
↳ throughput of 1 Element/Cycle if done correctly
↳ Number of banks ≥ Memory latency (usually 2^x banks)
↳ Next bank address = previous bank address + stride

Cache: - Temporal locality, keeps recent data ~~close~~ in smaller but faster memory (higher level) → closer to processor
- Spatial locality, keeps close data in higher level caches (like next word, next and prev entry)

- Hit: entry found in cache
- Miss: entry not found in cache
  ↳ Compulsory: first time data is accessed
  ↳ Capacity: cache to small to hold data (kick wit LRU)
  ↳ Conflict: data of interest maps to same address
  ↳ Miss penalty: time it takes to retrive data of miss
- Miss rate: #misses/#total access = 1 − Hitrate (#hit/#total)
- Average Memory Access Time (AMAT)
  ↳ $AMAT = t_{cache} + MissRate \cdot MissPenalty$ (MM = RAM, VM = HD)
  ↳ $t_{cache} + MR_{cache}(t_{MainMemory} + MR_{MM}(t_{VirtualMemory}))$

Example: 2'000 loads, 1250 in cache — HitRate = 1250/2000 = 0.625, MissRate = 750/2000 = 0.375

Example: $t_{cache} = 1$ cycle, $t_{MM} = 100$ cycles, was ist AMAT
$AMAT = t_{cache} + MR_{cache} \cdot t_{MM} = (1 + 0.375 \cdot 100) = 38.5$ cycles

- Capacity (C): number of data bytes a cache stores
- Block size (b): bytes of data brought into cache at once
- Number of Blocks (B): number of blocks in cache: B = C/b
- Degree of associativity (N): number of blocks in a set
- Number of sets (S): each memory address maps to exactly one cache set: S = B/N

(1) Direct mapped: 1 block per set — no conflict but expensive
(2) N-way set associative: N blocks per set
(3) Fully associative: all blocks are in single set

(1): N = 1, S = B
(2): 1 < N < B, S = B/N
(3): N = B, S = 1

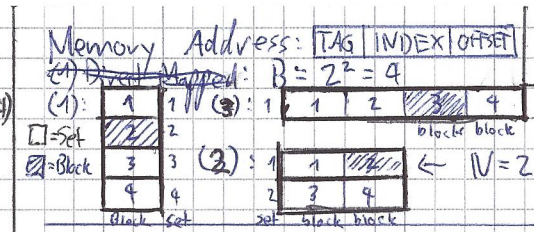
Associativity reduces conflict misses
Larger blocks reduces compulsory misses (spatial locality)

→ z.B. Full associated LRU cache can't prevent.

Memory Address: | TAG | INDEX | OFFSET |

15-937-717 Pascal Wacker 3/6

(1) Direct Mapped: B = 2^2 = 4



Virtual Memory:

Wie cache, mit: Block = Page, Miss = Page Fault, Tag = Virtual Page Number

Translating Addresses: Page Table is a Lookup-Table (LUT) holding an entry for every virtual Page. LUT between virtual and physical address space

10) Some numbers:

10^6 = MHz, 10^9 = GHz

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

z.B. AF → 1010 1111 → A F

- in 2 complement:
z.B. -11 in 2compl.
= 1111 0101
((X+1)·-1) bin ·XOR
+1
-10
10 → 0000 1010
→ 0→1, 1→0
→ 1111 0101 = -11
for 1 complements, leave the "+1" step!

Example Cache:

(1) 0x1000 4000 Memory Word 4 bytes (mem byte addressable)
(2) 0x1000 4020 Cache Type: Direct Mapped
(3) 0x1000 4004 Cache Size: 8 Words
(4) 0x1000 4028 Cache block size: 1 Word
(5) 0x1000 4024 Cache Access time: t_{cache} = 2 Cycles
(6) 0x1000 4010 Main Memory Access time: t_{MM} = 40 Cycles
(7) 0x1000 400C C = 8 Words · 4 bytes = 32, b = 1 Word = 4 bytes
(8) 0x1000 402C B = C/b = 32/4 = 8, N = 1, S = B = 8
(9) 0x1000 4040 S0 ↔ S = 8 $\log_2(S) = \log_2(8) = 3$ → Index = 3 bit
(10) 0x1000 4000 S1
(11) 0x1000 400C S2
(12) 0x1000 402C S3
(13) 0x1000 4020 S4
(14) 0x1000 4024 S5
(15) 0x1000 4054 S6
(16) 0x1000 4030 S7

(1)	S0	Compulsory	(14)	S1	Hit
(2)	S0	Conflict	(15)	S1	Conf
(3)	S1	Comp	(16)	S0	Conf
(4)	S2	Comp	# Hits = 2		
(5)	S1	Conf	# Comp = 4		
(6)	S0	Hit	# Conf = 10		
(7)	S3	Comp			
(8)	S3	Conf			
(9)	S0	Conf			
(10)	S0	Conf			
(11)	S3	Conf			
(12)	S3	Conf			
(13)	S0	Conf			

14 Miss

34 Tag	5 | Index	2 | Offset (MIPS)

|Offset| = $\lceil \log_2(b) \rceil$, |Index| = $\lceil \log_2(S) \rceil$
Set 0: 0x00, 0x20, ... |Tag| = 32 − |Offset| − |Index|
Set 1: 0x04, 0x24, ...
Set 2: 0x08, 0x28, ...
Set 3: 0x0C, 0x2C, ...
Set 4: 0x10, 0x30, ...
Set 5: 0x14, 0x34, ...
Set 6: 0x18, 0x38, ...
Set 7: 0x1C, 0x3C, ...

Solutions: 2-Way ← associated
- Block Size: 4 Words: 56
- Increasing Cache capacity to 16 Words: only words used in combination

Time: 2 Hit · t_{cache} + 14 Miss · (t_{cache} + t_{MM})
= 2·2 + 14·(40+2) = 592 Cycles

VLIW (Very Long Instruction Word)

↳Advantages: + No need for dynamic scheduling HW → simpler HW
+ No need for dependency checkin within VLIW Instr. → simple HW for multiple instr. issue + no renaming
+ No need for instruction alignment/distribution after fetch to different FU → simple HW

↳Disadvantages: − Compiler needs to find N independent operations per cycle (if not insert NOP in VLIW instruction) → parallelism loss & code size increase
− Recompilation required when execution with (N) or instr. latency or FU changes (unlike super scalar processing)
− Lockstep execution causes independent operations to stall → No instr. can progress until slowest instr. completes

---

SISD: Single Instruction Single Data
SIMD: Single Instruction Multiple Data
↳ Array Processor, Vector Processor
MISD: Multiple Instruction Single Data
↳ Closest form: Systolic Array Processor, Streaming Processor
MIMD: Multiple Instruction Multiple Data (Multiple Instruction Streams)
↳ Multiprocessor, Multithreaded Processor

Array Vs. Vector Processor:
Array: mehrere FU, jede bearbeitet ein Array element.
Vector: eine FU, elemente werden eines nach dem anderen durchgeschickt, spart [F][D]!

Wann muss [D] nach [Stall] erneut ausgeführt werden? Only if not 0-0-0!, 0-0-0 never again (OOO!)
Falls Processor Resource Contention, ex: add waits for an adder to become free → again [D]
If Data Dependency no need for [D] again! (except no forwarding, then [D] again (Back EX 4))

In-Order Pipeline mit ROB (Reorder Buffer)
[F]: Fetch
[D]: Decode, Access regfile/ROB, allocate entry in ROB, check if can execute and if so dispatch
[E]: Execute Instr., Instr. can execute 0-o-0!
[R]: Completion, Write Result to ROB
[W]: Retirement/Commit: check for exceptions
↳ if none write result to architectural register File or memory
↳ else handle, flush pipeline, start at exception

Verilog:

```
module mux2 (input [1:0] i; output s, input z);
  assign s= (z)? i[1]:i[0];
endmodule
module one (input [3:0] data, input sel1, input sel2, output z);
  wire [1:0] temp;

  mux2 i0 (data[1:0], sel1, temp[0]);
  mux2 i1 (data[3:2], sel1, temp[1]);
  mux2 final (temp, sel2, z);
endmodule
```

Error: input connected with output! better use a connect by name

```
module two (input [3:0] a, input [0:3] b, output reg [3:0] z);
  always @ (*)
    if (a == 0)
      z = {b[0], b[1], b[2], b[3]};
    else
      z = a+b
  endmodule
```

Syntax: OK, could be written as "assign z = a+b", mixing of [3:0] and [0:3]. is okay but strange

```
module three (clk, rst, a, b, c, z);
  input a, b, c, clk, rst; output z; reg z;
  always @ (*)
    begin
      q <= a^b;
      if (c) q <= ~(a^b);
    end
  always @ (negedge clk)
    if (rst) z = 1'b0;
    else     z = q;
endmodule
```

Syntax: OK, strange to use nonblocking in always...

```
module four (input [2:0] sel, output reg [5:0] z);
  case (sel)
    0: z = 6'b00_0000;
    1: z = 6'b00_0001;
    ...
    default: z = 6'b11_1111;
  endcase
endmodule
```

Error: case statement is a sequential statement and as such need to wrap in always block!
Note a case does always need a default

```
module fsm (input in, input clk, input rst, output
            reg detected);
  reg [1:0] next_state, state;
  parameter INIT = 2'b11; parameter ONE=2'b00
  always @ (*)
    begin
      next_state <= present;
      detected <= 1'b0;
      case (present)
        INIT: next_state <= in? INIT : ONE;
        ONE: if (in)
               next_state <= INIT;
             else
               begin
                 next_state <= ONE;
                 detected <= 1'b1;
               end
        default: next_state <= present
      endcase
    end
  always @ (posedge clk, posedge rst)
    if (rst) present <= INIT;
    else     present <= next_state
  endmodule
```

OK: Mealy type FSM

```
module sram (input [15:0] din, input [13:0] addr,
  input we, input clk, output [15:0] dout);
  // stuff
endmodule
```

each address stores 16 bit ⇒ 16/8 = 2 bytes. 14 addr. which are $2^{14} = 2^{10} \cdot 2^4 = 1024 \cdot 16 \approx 16k$ addresses
in total 16k · 2 bytes = 32k Bytes storage

Systolic Arrays:

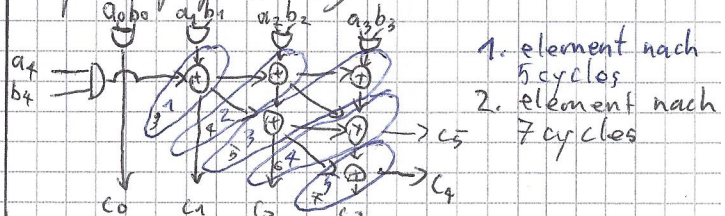
Goal: design accelerator that has:
- simple, regular design (keep # unique parts small and regular)
- high concurrent → fast
- balanced computation and I/O (memory) bandwidth

Idea: Replace single processing element (PE) with regular array of PEs and carefully orchestrate flow of data between the PE
- the PE collectively transform Data

Benefit: Maximizes computation done on a single piece of data element brought from MEM

Advantages: + Multiple use of data items → reduce need for fetching/refetching
+ high concurrent + regular design

Disadvantage: - not good at exploiting irregular parallelism
- special purpose, not generic



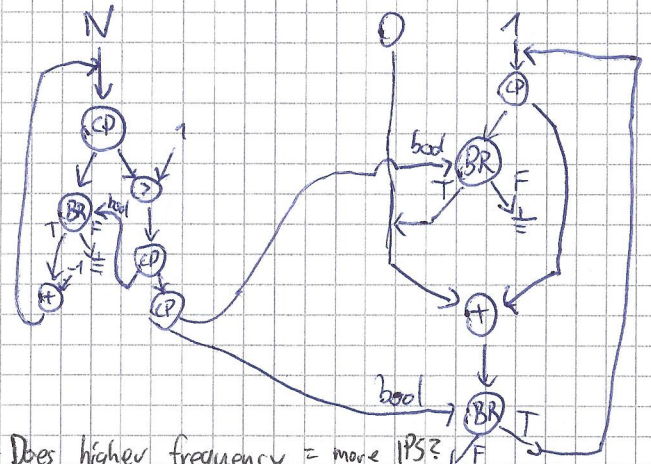
1. element nach 5 cycles
2. element nach 7 cycles

Practical Excersice 1:

```
fib: addi $sp, $sp, 16   // allocate space
     sw   $16, 0($sp)    // save r16
     add  $16, $4, $0
     sw   $17, 4($sp)
     add  $17, $0, $0
     sw   $18, 8($sp)
     addi $18, $0, 1
     sw   $31, 12($sp)
     add  $2,  $17, $18
branch: slti $3, $16, 2
     bne  $3, $0, done
     add  $2, $17, $18
     add  $17, $18, $0
     add  $18, $2, $0
     addi $16, $16, -1
     j branch
done: lw  $31, 12($sp)
     lw   $18, 8($sp)
     lw   $17, 4($sp)
     lw   $16, 0($sp)
     addi $sp, $sp, 16
     jr $31
```

```
int fib(int n) {
  int a = 0;
  int b = 1;
  int c = a+b;
  while (n > 1) {
    c = a + b;
    a = b;
    b = c;
    n--
  }
  return c;
}
```

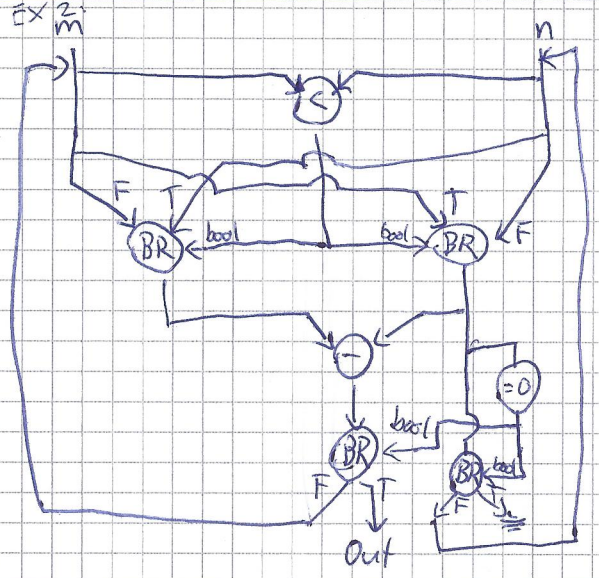
Dataflow version:



- Does higher frequency = more IPS?
  ↳ no, maybe a slower CPU has higher IPC. IPC = Instr. Per Cycle, IPS = Instructions per Second
- Does higher IPC = finish programm faster?
  ↳ No, maybe higher IPC because smaller instr → total # of instr. could be different

ISA A: 10 IPC, 500 MHz ? what is performance
ISA B: 2 IPC, 600 MHz / in Milion Instr. Per Second?
A: $10 \frac{\text{instr}}{\text{cycle}} \cdot 500'000'000 \frac{\text{cycle}}{\text{second}} = 5000$ MIPS
B: $2 \frac{1}{2} \cdot 600'000'000 \frac{2}{5} = 1200$ MIPS
We don't know which one is beter, since compiled program could use 100 instr on ISA B und 100'000 on ISA A
Difference ISA / Microarchitecture.
ISA = exposed to software, MA transparent to compiler/programmer
- Machine has no subtract instr. ISA
- ALU doesn't have subtract unit Microarchitecture
- Machine doesn't have condition codes, ISA

EX 2:



GCC/GGT
↳ Euklid Algo

Practical EX 3:
As pipline depth increases, the latency for a single instr. increases → each additional pipeline latch adds processing overhead

In a superscalar processor, the hardware detects the data dependencies between concurrently fetched instr. In contrast in a VLIW processor, the compiler does

TLB (Transaction Lookaside Buffer) caches: Page table entries

For a vector length of N we need N FU in an array processor and 1 in a vector processor

Warps: # threads / (# threads / warp) ex:
Warp1 = 64 threads = 2^6 1'024'768 threads = 2^{20}
Warps = $2^{20} / 2^6 = 2^{(20-6)} = 2^{14}$

8 FU, each with own broadcast bus
32 x 64-bit architectal registers
16 reservation station entries per FU
Each RS has 2 source registers

15-937-717 | Pascal Wacker 5/6

#of tag comparators per RS: 8 · 2 (8 FU, 2 source registers)

#of total comperators: $8 \cdot 16 \cdot 8 \cdot 2 + 32 \cdot 8$ (8 FU, 16 RS/FU, 2 source registers, 32 registers, 8 FU)

min size of tag: $\log(16 \cdot 8) = 7$ (16 RS/FU, 8 FU)

min size of RAT / frontend Register File:
↳ 72 · 32 (72 = 64 bit data + 7 bit tag + 1 bit valid), 32 Registers

min possible size of tag storage in Machine:
$7 \cdot 32 + 7 \cdot 16 \cdot 8 \cdot 2$ (32 register, 16 RS/FU, 8 FU, 2 source registers)

$2^0 = 1$			
1: 2	11: 2048	21: 2'097'152	
2: 4	12: 4096	22: 4'194'304	
3: 8	13: 8192	23: 8'388'608	
4: 16	14: 16384	24: 16'777'216	
5: 32	15: 32768	25: 33'554'432	
6: 64	16: 65536	26: 67'108'864	
7: 128	17: 131'072	27: 134'217'728	
8: 256	18: 262'144	28: 268'435'456	
9: 512	19: 524'288	29: 536'870'912	
10: 1024	20: 1048'576	30: 1'073'741'824	

→ Interlocking
Scoreboarding: each reg in reg file with valid bit
- instr. writing to reg → reset valid bit
- instr. decode check for valid bit
bit not stall
+ simple - stalls for all dependences
Combinational Dependency Checking Logic
if instr. in later stage writes to any source
↳ stall
+ only stalls flow - complex (specially with deep pi
Forwarding / Bypass
+ less staling, skiping Reg file