Hex to Dec: \( X Y Z \times 10^2 + (x \cdot 10^1) + (y \cdot 10^0) + (z \cdot 10^0) \)

Dec to Hex: Zahlen durch 16 teilen + Rest entspricht der i-ten Stelle, i++; Ergebnis wieder durch 16, so lange bis Rest = 0.

Hex: \( 0 \ldots 9, A (\equiv 10), B (\equiv 11), C (\equiv 12), D (\equiv 13), E (\equiv 14), F (\equiv 15) \)

Ones Complement: Invert the number \( +1 \) becomes 0, 0 becomes 1

Two's Complement: From ones complement, add 1: MSB = 1: negative number, MSB = 0: positive number or 0.

<table>
<thead>
<tr>
<th>NOT</th>
<th>AND</th>
<th>NAND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg a )</td>
<td>( a \land b )</td>
<td>( \neg(a \lor b) )</td>
<td>( a \lor b )</td>
</tr>
</tbody>
</table>

\[ \text{Operators:} \]

\[ + = \text{AND} \]

\[ = = \text{OR} \]

<< or >>: Shift left or shift right

<<< or >>>: Signed shift left or signed shift right

\( y = s ? d1 : d0 \rightarrow \text{if}(s) \text{ then } y = d1, \text{ else } y = d0 \)

module function (input a, input b, output s) ... endmodule

always @(*); always @posedge clk) + a when signals change

\( + = \text{blocking}, \text{ value is assigned immediately} \)

\( - = \text{non-blocking, value is assigned at end of block} \)

wire \( a \rightarrow a \) simple wire \#5: Wait 5 ns

wire [5:0] a \rightarrow a 6-bit bus

case(data); \( xyz \times 1 \rightarrow \) default: \( a \times c = 0 \) (only in always)
Registers:

- $0: Constant 0
- $0-$r7: Procedure return values
- $0-$a3: Procedure argument values
- $t0-$t7: Temporary Registers
- $s0-$s7: Saved Variables
- $a: Return address
- $sp: Stack pointer

R-Type: Register Operands
I-Type: Immediate Type Operands
J-Type: Jump Type Operands

Procedures:
- Call: passes arguments to callee, ja to set return address
- Calli: performs procedure, returns result to caller, jr to return

Stack: Memory used to temporary save data, LIFO, grows down from higher address to lower address. Use $sp to use the stack:

```
add $sp, $sp, -12 # allocate space for 3 registers
sw $s0, 8($sp) # save $s0 to first free stack place
```

Multi-cycle:

- 3 cycles: beq, j ; 4 cycles: R-Type, sw, addi; 5 cycles: lw

Microinstruction:
- Control Signals associated with the current state
- Microsequencing: Act of transitioning from one state to another, determining the next state and its Microinstruction
- Control Store: Stores Control Signals for every possible state
- Microsequencer: Determines the next Microinstruction
- Allows a simple design to do powerful computation, enables easy extensibility of ISA, enables update of machine behavior

Pipelining:

- Idea: Divide the instruction processing cycle into different stages of processing, to use hardware better / faster.
- Steady state: Pipeline is full and used to its full potential
- Speedup: Optimal = 1 / (T/l + s) for l stages, where s=trash delay & T= time for non-pipelined Version

Data dependencies that can occur:
- Flow dependence (read after write): True dependence, always obey
- Output dependence (write after write): Exist because of limited set of registers, can be solved by renaming registers
- Anti dependence (write after read): same as Output dependence

Von-Neumann Model: Instruction & Data memory is unified, sequential instruction processing
Detect and eliminate
- Predict the value, execute speculatively and verify
- Do something else → Fire-grained multithreading

Fire-grained Multithreading
Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently

Data Forwarding / Bypassing: Add additional dependence check logic and data forwarding paths to apply the processor value to the consumer right after the value is available → less stalling / increased hardware cost

Stalls: Condition where the pipeline steps/pauses, can be caused by Resource Contention, Dependencies, Long-Latency Operations

Scoreboarding: Central Place keeps track of dependencies. Only if all are solved the operation may execute, else it stalls

Interrupts: External to the running thread, can be handled when convenient

Exceptions: Internal to the running thread, must be handled when detected.

All previous instructions should be retired/committed, and no late instruction should be retired (= finish execution and update arch.

Precise Exceptions: Writeback is stalled until it’s ready to write back if it had been executed in a sequential way (previous instructions all wrote back)

Imprecise Exceptions: Writeback happens right after execute stage is over, can lead to hazards when exceptions occur!

Out-of-Order Processor:
Execute instructions in any order, reorder them after executing

Ready RAT: Complete instructions C0, but reorder them before making results visible to architectural state, preserving the Non-Terminable Model.

Tomasulo's Algorithm: Has Register Alias Table (RAT), Functional Units (FU) and Reservation Stations (RS). RAT & RS have valid bit, tag and value fields for each entry.
1) Operation is decoded, operands are put into RS. If data from RAT is valid, it’s used, else its tag is put into the RS.
2) If all operands in the RAT are available for an operation, the operation is passed to the FU and executed
3) When result of Operation is available, it’s broadcasted to the RS, so that operations waiting for it can be executed. Value is also written to the RAT, now valid again

Other Ways of Execution

VLIV: Very long instruction word, one instruction word contains several instructions that are executed on several FU.

SIMD: Some Instruction, Multiple data

Array Processor: Instructions operate on multiple data elements at the same time using different spaces

Vector Processor: Instructions operate on multiple data elements at the same space, but in consecutive time steps

+ No dependencies within a Vector

+ Each instruction generates a lot of work → lower int. Fetch bandwidth
+ Highly regular memory access pattern
+ No need to explicitly code loops
+ Works only if parallelism is regular
+ Memory bandwidth can become a bottleneck

Each vector data register holds \( N \times M \)-bit values.

**Control Registers:** VLEN, VSTR, VMASK (Can be used for conditional programs)

**Vector FUs** are deeply pipelined and allow fast clock cycles.

A loop is vectorizable if each iteration is independent of others.

**Vector chaining:** Data forwarding from one FU to another (page 5).

A SIMD Processor has multiple lanes which operate in parallel.

**Memory Banking** is used to resolve long memory latency.

All memory banks have a shared data bus and a shared address bus.

Each bank \( 0, \) bank \( 1, \) bank \( 2 \), etc → throughput of \( 1 \) element / cycle if done correctly.

Number of banks \( \geq \) Memory latency

\[ \text{Next address} = \text{Previous Address} + \text{stride} \]

**GPU:**

\# Warps = \# Threads / \# Threads per Warp

The instruction pipeline operates like a SIMD pipeline, but the programming is done using threads.

→ SIMD not exposed to the programmer (SIMT)

SIMD = Single Sequential instruction stream of SIMD instructions

SIMT = Multiple instruction streams of scalar instructions

- Can treat each thread separately

**Warp/Wavefront**

A group of threads executing the same instruction.

- Essentially a SIMD operation formed by hardware.

**Branching in Warps:** Predicate Execution. GPU uses a mask similar to VMASK to only operate on correct data. Is done by the GPU in contrast to Vector machine where programmer needs to do this.

**Improving SIMD utilization:** Find individual threads that are at the same PC and group them together into a single warp dynamically.

---

**Programming Model vs Execution Model**

**Programming Model** refers to how the programmer expresses the code, e.g., Sequential (Von Neumann), Data Parallel (SIMD), Dataflow, Multithreaded (MIMD, SPMD)

**Execution Model** refers to how the hardware executes the code underneath, e.g., Out-of-order, Vector Processor, Array Processor, Dataflow Processor, etc.

*Don’t need to match, a Sequential Program can be executed on a Out-of-Order Processor!*

**Memory:**

Flip-Flops (or latches)

+ Very fast, parallel access

- Expensive (one bit costs 20+ transistors)
Static RAM (SRAM)
- Relatively fast, only one data word at a time
- Less expensive (still 6 transistors!)

Dynamic RAM (DRAM)
- Cheap (one bit is 1 transistor)
- Slower, reading destroys content (refresh), one data word, needs special process charging a small capacitor

Temporal Locality: Locality in time, keep recently accessed data in higher levels of memory
Spatial Locality: Locality in space, bring nearby data into higher levels of memory too

Hit: Is found in that level of memory
Miss: Is not found, must go to next level of memory

Hit rate: \( \# \text{ Misses} / \# \text{ Memory accesses} \)

Average Memory Access Time (AMAT): \( t_{\text{cache}} + MR_{\text{cache}} \times t_{\text{memory}} \)

Capacity \( C \): Cache size \( \times \) number of bytes stored in cache

Block size \( b \): bytes of data stored and brought into cache together

Number of blocks \( B = C / b \): \# of blocks in the cache

Degree of associativity \( N \): \# blocks per set of cache

Number of sets \( S = B / N \): Set is one or more memory blocks that map to some memory address. Inside sets, the tag is used to find the correct cache entry.

Memory Address: TAG | INDEX | OFFSET

### Cache

<table>
<thead>
<tr>
<th># Sets</th>
<th>Assoc</th>
<th>( B = 2^N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Mapped</td>
<td>( S = B )</td>
<td>1</td>
</tr>
<tr>
<td>Fully Associative</td>
<td>1</td>
<td>( N = B )</td>
</tr>
<tr>
<td>N-Way Set Associative</td>
<td>( S = N )</td>
<td>( N )</td>
</tr>
</tbody>
</table>

Bigger Block size: Several data units per block, reduces compulsory misses but increases conflict misses.

Compulsory miss: Cache data accessed for first time \( \to \) empty
Conflict miss: Data of interest maps to some location as other

Virtual Memory:

Analogies to cache: Block \( \to \) Page; Miss \( \to \) Page Fault

Tag = Virtual Page Number

Translating Addresses: Page Table; has entry for each virtual page. Is a look-up table between the virtual and the physical address space.

Virtual Memory: Bigger, Address space \( = \) RAM + HDD

Physical Memory: Memory in RAM, smaller

Translating: Take Virtual page number, search entry, get smaller physical address.

- Similar to how cache works, higher level of memory in cache \( \to \) Physical Memory.
Dataflow Model (of a computer)

Instructions are fetched and executed in data flow order:
- i.e. when its operands are ready
- i.e. there is no instruction pointer
- Instruction ordering specified by data flow dependence, each instruction specifies who should receive the result; an instruction can execute when it received all operands

Many instructions execute at the same time, inherently more parallel

Vector Chaining

If a Vector Processor supports Chaining, the data is forwarded as soon as it completes a stage \( t \) time per step is only added once, and not \( (t + \text{VEN}-1) \), \( \text{VEN}-1 \) is only added once, at the last steps

\( \Rightarrow \) Treat individual instructions as "delays."

Karnaugh Maps

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

1) Fill out map with truth values
2) Group 1 into groups where
   \# Elements = \( 2^k \) for \( k \in \mathbb{N} \)
3) AND all variables that don't change in the group.

Systolic Arrays

Idea: Replace a single processing element (PE) with a regular array of PEs and carefully orchestrate flow of data such that it collectively transforms input data

\( \Rightarrow \) Similar to blood flow in the human body

- Pipeline: Individual PEs, Array structure can be non-linear
  & multidimensional; PE connections can be multidirectional;
  PEs can have local memory and execute kernels

SIMD Utilization

SIMD Utilization = part of the SIMD pipeline that is kept busy by warps.

\( \neq 1 \) (\( \neq 100 \% \)): Some branches are taken / not taken, which leads to some instructions not being executed.

Total instr. = Threads in a Warp \* Instructions per thread (e.g. on if instr. is executed \( n \) times if the warp has \( n \) threads in it.)