Digitaltechnik FS 2017

Hex to Dec: $XYZ_{16} = (x \cdot 16^2) + (y \cdot 16^1) + (z \cdot 16^0)$

Dec to Hex: $Zahl_{10}$ durch 16 teilen → Rest entspricht der i-ten Stelle, i++, Ergebnis wieder durch 16, so lange bis Rest = 0.

Hex: 0 ... 9, A(=10), B(=11), C(=12), D(=13), E(=14), F(=15)

Ones Complement: Invert the number → 1 becomes 0, 0 becomes 1

Twos Complement: Form ones complement, add 1. MSB = 1: negative number, MSB = 0: positive Number or 0.

NOT	AND	NAND	OR
~a	a & b	~(a & b)	a | b
			
NOR	**XOR**	Operators:	
~(a | b)	a ^ b	• = AND	
		+ = OR	

- << or >>: Shift left or shift right
- <<< or >>>: Signed shift left or signed shift right
- y = s ? d1 : d0 → if(s) then y = d1, else y = d0
- module function (input a, input b, output s) ... endmodule
- always @(*); always @(posedge clk) → do when signals change
- = blocking, value is assigned immediately
- <= non-blocking, value is assigned at end of block
- wire a → a simple wire • #5: Wait 5 ns
- wire [5:0] a → a 6-bit bus
- case(data); xyz: abc = 1; default: abc = 0 (only in always)

Logic

- minterm: Product (AND), true stays, false is inverted
- maxterm: Sum (OR), false stays, true is inverted
- Sum-of-Products: OR-ing the minterms for which the output is true
- Product-of-sums: AND-ing the maxterms for which the output is false

Combinatorial: Memoryless, Output defined by current input, no cycles

Sequential: Has memory, output defined by input & previous state

FSM: Next state is determined by current state & inputs

Moore: Outputs depend only on current state

Mealy: Outputs depend on current state & inputs

Timing

Propagation delay t_{pd}: max delay from input to output

Contamination delay t_{cd}: min delay from input to output

setup time t_{setup}: time before the clock edge that the data must be stable

Hold time t_{hold}: time after the clock edge that the data must be stable

Aperture time: $t_{setup} + t_{Hold}$

After contamination delay the data might change, but it's stable after the propagation delay

Clock period: T_c, time between rising Edges. $F_c = 1 / T_c$ is the frequency of the circuit.

MIPS Assembly:

- operation, destination, source1, source2 (e.g. add $a, $b, $c)

Big-Endian: byte numbers start at the big end (MSB)

Little-Endian: byte numbers start at the little end (LSB)

Registers:

$0: Constant zero

$v0-$v1: procedure return values

$a0-$a3: Procedure argument values

$t0-$t7: Temporary Registers

$s0-$s7: Saved Variables

$ra: Return address

$sp: Stack pointer

R-Type: Register Operands

I-Type: Immediate Type Operands

J-Type: Jump Type Operands

Procedures:

- Caller, passes arguments to callee, jal to set return address
- Callee, 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:

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

Single cycle: Perform each instruction in one cycle. Constant CPI, no pipelining, slowest OP slows down whole CPU.

Multi-Cycle: Perform each instruction in multiple cycles → possibility to use pipelining

Von-Neumann Model: Instruction & Data memory is unified, sequential instruction processing

Multicycle Speed: 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 = $1 / (T/k + S)$ for k stages, where S = latch delay & T = time for non-pipelined version

→ Cost increases too because additional hardware is needed!

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

How to handle dependencies: (Flow dependences especially):

- Detect and wait
- Detect and forward / bypass the data

- Detect and eliminate
- Predict the value, execute speculatively and verify
- Do something else → Fine grained multithreading

Fine 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 supply the produces value to the consumer right after the value is available
→ less stalling / increased hardware cost

Stalls: Condition where the pipeline stops/pauses, can be caused by Resource Contention, Dependences, 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 / commited, and no later instruction should be retied (= finish execution and update arch. State)

Precise Exceptions: Writeback is stalled until it's ready to write back if it had been executed in a sequential way (previas 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

Reorder Buffer: Complete instructions OoO, but reorder them before making results visible to architectural state, preserving the Von-Neuman Model.

Tomasulos 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

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

SIMD: Same 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 instr. fetch bandwith

+ Highly regular memory access pattern
+ No need to explicitly code loops
-- Works only if parallelism is regular
- Memory bandwith can become a bottleneck
Each vector data register holds N M-bit values
Control Registers: VLEN, VSTR, VMASK (can be used for conditional programs.
Vector FU are deply 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 6)
A SIMD Processor has multiple lanes which operate in parallel

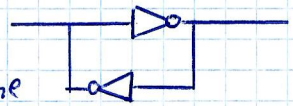
Memory Banking is used to resolve long memory latency
All n memory banks have a shared data bus and a shared address bus.
Fetch from bank 0, bank 1, bank 2, etc → throughput of 1 element / cycle if done correctly
Number of banks >= Memory latency
Next address = previous Address + stride

GPUs # 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 programme (SIMT)
SIMD = Single Sequential instruction stream of SIMD instructions
SIMT = Multiple instruction streams of scalar instructions
+ Can treat each thread sparately
+ Can group threads into Warps flexibly to maximize benefits of SIMD processing
Warp / Wavefront A group of threads executing the same instruct
⇒ 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 contrary 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)

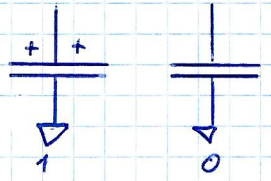
Static RAM (SRAM)



+ 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

Cache

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

Miss rate: # Misses / # Memory accesses

Average Memory Access Time (AMAT): $t_{cache} + MR_{cache} * t_{memory}$

Capacity C: Cache size → number of bytes stored in cache

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

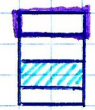
Number of blocks (B = C/b): # of Blocks in the cache

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 same memory address. Inside sets, the tag is used to find the correct cache entry.

Memory Address:

TAG	INDEX	OFFSET

	# sets	Assoc.	$B = 2^2 = 4$
Direct Mapped	$S = B$	1	 Set / Block
Fully Associative	1	$N = B$	
N-Way Set Assoc.	$S = \frac{B}{N}$	N	 $S = \frac{4}{2} = 2$

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

Compulsory miss: Cache data accessed for first time → empty

Conflict miss: Data of interest maps to same location as other

Virtual Memory:

Analogies to cache: Block = Page; Miss = Page Fault;
Tag = Virtual Page Number

Translating Addresses: Page Table; has entry for each virtual page. Is a Lookup-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 → 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 recive the result; an instruction can execute when it recieved all operands

⇒ Many instructions execute at the same time, inherently more parallel

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

⇒ Similar to blood flow in the human body

vs. Pipelining: 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.

If not 1 (= 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. an if instr. is executed n times if the warp has n threads in it.)

Vector Chaining

If a Vector Processor supports Chaining, the data is forwarded as soon as it completes a stage → time per step is only added once, and not (time * VLEN-1)! VLEN-1 is only added once, at the last steps

⇒ Treat individual instructions as „delays".

Karnaugh Maps

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

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