Numbering Systems

Hexadecimal: A=10, B=11, C=12, D=13, E=14, F=15. E.g.: $2ED = 2\cdot16^2 + E\cdot16^1 + D\cdot16^0$

Binary: Least significant bit (LSB): Bit in 1 column, most significant bit (MSB): Bit at the other end. $2^{10} = 1024 \approx 10^3 \rightarrow$ kilo, $2^{20} = 1'048'576 \approx 10^6 \rightarrow$ mega, $2^{30} = 1'073'741'824 \approx 10^9 \rightarrow$ giga

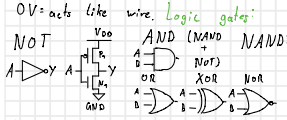
Data types:

Unsigned: Only positive numbers, normal binary representation. **Unsigned:** Positive/negative, multiple schemes: **Sign/magnitude number:** MSB for sign (0=positive, 1=negative), remaining bits for absolute value. **Two complement:** Identical to unsigned, but MSB has value of -2^{N-1} (instead of 2^{N-1}). Reversing sign: Invert all bits and add 1. Normal addition works.

Combinational circuits

A combinational circuit's outputs depend only on the current values of the input. **Transistors:** n-type:

If the gate is supplied with a high voltage, connection from source to drain acts like a wire. With 0V, connection is broke. p-type: High voltage = Broken connection, 0V = acts like wire. **Logic gates:**



Boolean equations: A minterm is a product („AND") containing all of the inputs to the function. A maxterm is a sum („OR") containing all of the functions (also min/maxterm when the complement of an input is contained). **Sum-of-products form:** In this form, we take the minterm of each true entry in the truth table (when input=0 the complement, otherwise the input => A=0, B=0 becomes $\bar{A}\bar{B}$) and sum („OR") them. **Product-of-sums form:** Each row of a truth table corresponds to a maxterm that is false for that row (e.g. A+B for A=0, B=0 or $\bar{A}+B$ for A=1, B=0). In the pos form, we take that maxterm for each false row and „AND" them. **Boolean algebra:** The most important theorems are:

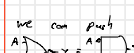
1) $B\cdot C = C\cdot B$ / $B+C = C+B$ 2) $(B\cdot C)\cdot D = B\cdot(C\cdot D)$ / $(B+C)+D = B+(C+D)$

3) $(B\cdot C)+(B\cdot D) = B\cdot(C+D)$ / $(B+C)\cdot(B+D) = B+(C\cdot D)$ 4) $B\cdot(B+C) = B$ / $B+B\cdot C = B$ 5) $(B\cdot C)+(B\cdot\bar{C}) = B$ / $(B+C)\cdot(B+\bar{C}) = B$ 6)

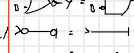
$(B\cdot C)+(\bar{B}\cdot D)+(C\cdot D) = B\cdot C+\bar{B}\cdot D$ / $(B+C)\cdot(\bar{B}+D)\cdot(C+D) = (B+C)\cdot(\bar{B}+D)$

7) $\overline{B_0\cdot B_1\cdot B_2\cdot\ldots} = (\bar{B}_0+\bar{B}_1+\bar{B}_2+\ldots)$ / $\overline{B_0+B_1+B_2+\ldots} = (\bar{B}_0\cdot\bar{B}_1\cdot\bar{B}_2\cdot\ldots)$

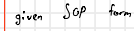
The last (De Morgan's Law) is important for conversions between different logic functions, e.g. $A+B+C = \overline{\overline{A+B+C}} = (\bar{A}\cdot\bar{B}\cdot\bar{C})$ or $\overline{(X+Y)} = \bar{X}\bar{Y}$ or $\overline{(X\bar{Y})} = \bar{X}+Y$. **Bubble pushing** uses this in circuit. If we push bubbles backwards (forwards), AND changes to OR and vice-versa. A bubble on the output becomes one on all inputs and if we have one on all inputs, we can push it to the output. E.g.:



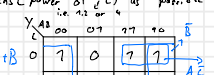
or



Two bubbles on a wire cancel each other, i.e.



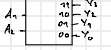
We can easily build a circuit from a given SOP form by introducing an AND-gate for every minterm and combining all results of the AND-gates with an OR-gate (for POS we do the contrary). **Illegal / Floating values:** X indicates an unknown/illegal value (driven by HIGH and LOW). Z means a floating value (neither HIGH nor LOW) and can be 0, 1 or anything between. **Karnaugh maps:** In a K-map, we write the possible values in gray code order (00, 01, 10, 11) and try to circle as few rectangular blocks (power of 2) as possible (i.e. 1, 2, 4 or 8) and then read the minterms.



Wrapping around is allowed. $Y = A\bar{C} + \bar{B}$ Don't cares (X) can be used as 0 or 1.

Building blocks: Multiplexers: Choose an output among several possible inputs based on select signal(s).

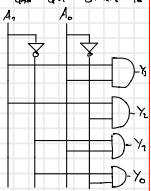
Decoder: Asserts exactly one of its 2^N outputs based on N inputs:



Lookup tables: Multiplexers can be used to implement logic by connecting all truth values to HIGH and all others to LOW (e.g. 8:1 multiplexer for 3-input logic)

(2:1 Multiplexer)

2:4 decoder:

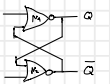


Programmable logic array: 2^N AND-Gates for every input combination with programmable connections to OR-gates.

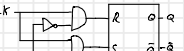
Timing: Propagation delay: t_{pd} is the maximal time from when an input changes until (the outputs) reach their final value. **Contamination delay:** Minimum time from when an input changes until any output starts to change its value. (Half) **Glitches:** When a single input transition causes multiple output transitions. Occurs because a shorter path first changes the output and later the longer path changes it back to the correct output. Can be resolved by adding gates (in the K-map -> Circle between boundaries).

Sequential circuits

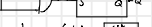
The output of sequential logic depends on both current and previous input values (i.e. has memory). **Building blocks:** **SR Latch:** Has a Set(S) and Reset(R) input.



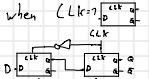
When S=R=0, remembers the old value. When S=1, R=0 => Q=1, when S=0, R=1 => Q=0. When S=R=1, Q and $\bar{Q}$ is 0. **D Latch:**



Next state is controlled by D (D=1 => Q=1, D=0 => Q=0 and only asserted when CLK=1).

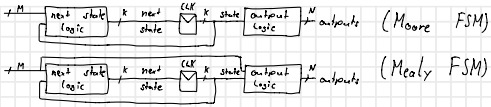


D flip-flop: A D flip-flop copies D to Q on the rising edge of the clock and remembers the value otherwise. Built from two latches.



Register: An N-bit register is a bank of N flip-flops that share a common CLK so they are updated at the same time.

Enabled/resettable flip-flops: Flip-flops with ENABLE (AND-ed with CLK) and RESET ($\overline{RESET}$ AND-ed with D) signals. **Circuits:** In a synchronous sequential circuit, state transitions occur at the rising edge of the clock. **Timing:** The setup time (of a storage element) is the time before the clock edge that data must be stable. The hold time is the time after the clock edge that data must be stable. Aperture time = setup + hold time. Storage elements have a propagation delay, too. Therefore we have for the clock time $T_c \geq t_{pcq} + t_{pd} + t_{setup}$ where t_{pcq} and t_{pd} are the prop. delays of the storage element respectively circuit. t_{pcq} / t_{setup} is the sequencing overhead (= wasted work"). Because of the hold time, we need to have a minimum propagation delay: $t_{cd} > t_{hold} - t_{ccq}$ where t_{cd} / t_{ccq} are the combinational delays of the circuit / storage element. **Finite state machines:** An FSM is a discrete-time model of a stateful system and pictorially shows the set of all possible states that a system can be in and how the system transitions from one state to another. An FSM has 5 elements: 1.) A finite number of states, 2.) A finite number of external inputs 3.) A finite number of external outputs 4.) An explicit specification of all state transitions 5.) An explicit specification of what determines each external output value. An FSM consists of next state logic (combinational circuit), state register(s) and output logic. The output is determined by the current state and the inputs. In **Moore machines** the output depends only on the current state, whereas in **Mealy machines** the output depends on the current state and the inputs.

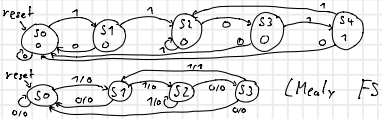


(Moore FSM)

(Mealy FSM)

Based on a FSM state transition table (containing current state, inputs and next state) one can derive the formulas for the next state logic. Based on a FSM output table (containing current state and outputs) one can derive the output logic.

State encodings: In the fully encoded scheme, we just enumerate the states and therefore need $\log_2(\#states)$ bits. In the 1-Hot encoded scheme we use a bit for every state (e.g. 0001, 0010, 0100, 1000 for 4 states). In the output encoded scheme (only for Moore machines), we encode the output in the state. Example FSM: Output 1 when „1101" encountered:

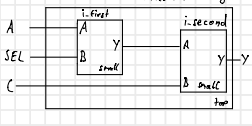


(Moore FSM)

(Mealy FSM)

Hardware description Languages / Verilog

Verilog works with modules that can instantiate sub-modules, that can instantiate sub-modules, etc... Until we have basic building blocks). A module has a name and ports with a direction (input/output) and a name. Multi-bit inputs/outputs (busses) are declared like this: `input [3:0] ai` (we could also use `[0:3]`, but `[3:0]` is recommended). Verilog is case sensitive, names cannot start with a number and white-space is ignored. Bit manipulation: It is possible to assign partial busses (e.g. `assign b=a[2:5];`) and concatenating is done by `{}`: `assign y= {a[2], a[7], a[0], a[0]};` If we want to assign multiple copies: `assign y= {4 {a[0]}};` Structural HDL: In structural HDL, the module body contains gate-level description of the circuit. Each module contains other modules (instances) that are connected. E.g.:




```
module top (input A, input SEL,
            input C, output Y);
  wire n1;
  small i_first (.A(A), .B(SEL), .Y(n1));
  small i_second (.A(n1), .B(C), .Y(Y));
endmodule
```

Verilog supports basic logic gates (and, or, not) that are instantiated like modules, e.g. `and g1(y1, d0, ns);`. Behavorial HDL: In behavioral HDL, the module body contains functional description of the circuit. It contains logical/mathematical operators: `&` (AND), `|` (OR), `^` (XOR), `~(a & b)` (NAND), `~(a | b)` (NOR). Furthermore, reduction operators exist, e.g. `assign y=&ai` for bit-wise AND on a bus a. The ternary operator `?` can be used for conditional assignments: `assign y= s? d1:d0;` For comparison, `==` resp. `!=` is used. Expressing numbers: Numbers are expressed as `N'Bxx` where N is the nr. of bits, B the base (b for binary, h for hexadecimal, d for decimal and o for octal) and xx the number. For readability, underscores can be used as seperators. E.g.: `8'b0000_0001`. We can also use `Z` (floating) or `X` (don't care, sim only).

Assignments: `assign` is used for continual assignments i.e. value of LHS is updated when RHS changes. E.g. `assign y=ai;` It is equivalent to `always@(*)`. In an `always @(sensitivity list)` block, whenever the event in the sensitivity list occurs, the statement is executed. `posedge` defines a rising edge (transition from 0 to 1) and is used with an always block for sequential circuits. `always @(posedge clk, negedge reset)`

```
if (reset==0) q<=0
else          q<=d
```

E.g. a D Flip-Flop with asynchronous reset:

When the clock is not rising, the value is preserved. Variables that are assigned in always blocks need to be declared as regs.

An always block defines combinational logic if all outputs are continuously updated i.e.: 1) All right-hand side signals are in the sensitivity list (shorthand: `always @(*)` 2) All left-hand signals get assigned in every possible combination of if..else / case blocks. `always @(*)`

We can use if...else (only!) in always blocks or case-statements like in the example on the right.

```
case(data)
  4'd0: q= 1'b1
  ⋮
  default: q= 1'b0;
endcase
```

With non-blocking (`<=`) assignments, all assignments are made at the end of the block (in parallel!). With blocking (`=`), each assignment is made immediately and the process blocks (sequential!).

FSM in Verilog: `module div3FSM (input clk, input reset, output q);`

In a typical FSM in Verilog, we assign the next state in a sequential always block and implement the next state logic in an always @(*) (combinational) block. It is important to note that combinational always blocks should always have a default case (so that no memory is synthesized for the other cases).

```
always @(posedge clk, posedge reset)
  if(reset)   state <= S0;
  else        state <= nextstate;
always @(*)
  case (state) // S0: nextstate=S1;
               // S1: nextstate=S2; // S2: nextstate=S0;
    default: nextstate=S0; // endcase
  assign q = (state == S0);
endmodule
```

Testing: We differentiate simple (hardcoded inputs, manual verification), self-checking (hardcoded inputs, automatic verification with print statements) and automatic testbenches (inputs generated and run through „golden model"). For testing/simulation, we can use `#N` (delay by N time units) and `$display("...");`

Computer models

Von Neumann Model: The Von Neumann Model consists of 5 parts: 1) Memory: The memory stores data and programs. It contains bits that are grouped into bytes (8 bits) and words (e.g. 8, 16, 32, 64). Memory can be byte- or word-addressable. Each data word has a unique address and the total number of addresses is the address space. How the bytes in a word are addressed depends on the endianness. In little endian, the LSB has address 0 whereas in Big Endian the MSB has address 0.

LE:

3	2	1	0

BE:

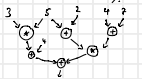
MSB			LSB
0	1	2	3

(example with a 32-bit word)

To access memory there are the Memory Address Register (MAR) and the Memory Data Register (MDR). To read, the MAR is loaded with the address and data is placed in the MDR. To write, the MAR is loaded with the address and the MDR with the data and the Write Enable signal is activated. 2) Processing unit: Can consist of many functional units, normally ALU (Arithmetic and Logic Unit). The ALU processes words and has registers (fast memory, one register contains normally one word). 3/4) Input/output: Keyboard, monitor, mouse,... 5) Control unit: The control unit conducts the step-by-step process of executing a program. It has an Instruction Register (IR) that contains the instruction and a Instruction Pointer (IP) / Program counter (PC) that contains the address of the next instruction to execute. Instruction cycle: An instruction goes to the following 6 phases (not all have 6 phases): 1) Fetch: Obtains the instruction from memory and places it in IR. 2) Decode: Identifies the instruction i.e. FSM next state decided based on opcode. 3) Evaluate addresses: Computes the address of the memory location that is needed to process the instruction. 4) Fetch operands: Obtains the source operands (memory / registers) needed to process the instruction. 5) Execute: Executes the instruction (control instructions change the PC here). 6) Store result: Writes to the designated destination.

Addressing modes: An addressing mode is a mechanism for specifying where an operand is located. There are 6 addressing modes: 1) Immediate / Literal (operand is in bits of the instruction) 2) Register 3) PC-relative memory 4) Indirect memory (calculated address is address of location that contains address) 5) Base+offset (a base and an offset is specified) 6) Pseudo-direct addressing (e.g. MIPS: Upper four bits of PC concatenated with direct address). Instruction types: 1) Operate instructions (execute in the ALU) 2) Data movement instructions 3) Control flow instructions

Data flow: In data flow, availability of data determines the order of execution. A data flow node fires when its sources are ready. Programs are represented as data flow graphs.



Data flow is very good at exploiting regular parallelism, only real dependencies constrain processing.

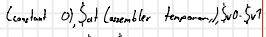
But it has no precise state semantics (difficult debugging), sometimes too much parallelism and high bookkeeping overhead (tag matching, data storage).

Instruction Set Architecture (ISA)

The ISA is the interface between what the software commands and what the hardware carries out. It specifies:

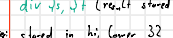
1) The memory organization: The address space (LC-3: 2^{16}, MIPS: 2^{32}), the addressability (LC-3: 16 bits, MIPS: 32 bits) and if memory is word- or byte-addressable (LC-3: word-addressable, MIPS: byte-addressable). 2) The register set: R0 to R7 in LC-R, in MIPS: $0 (constant 0), $at (assembler temporary), $v0-$v1 (function return value), $a0-$a3 (function arguments), $t0-$t9 (temporary variables), $s0-$s7 (saved variables), $k0-$k1 (OS temporaries), $gp (global pointer), $sp (stack pointer), $fp (frame pointer) and $ra (function return address).

3) The instruction set: Opcodes, data types and addressing modes. **LC-3:** LC-3 (Little Computer 3) is an educational ISA. **Opcodes:** Instructions are 16 bits wide and have 4-bit opcodes (4 MSBs).



ADD has opcode 0001 and the instruction looks like the example (bits 5-3 always 0). LC-R has three single bit registers for condition codes: N(negative), Z(zero) and P(positive). They can be used in branches (e.g. BRz ⇒ Branch if zero). **Data types:** LC-3 only supports 2's complement integer. **Addressing modes:** Immediate, register, PC-relative, indirect and base+offset (no pseudo-direct!).

MIPS: MIPS (Microprocessors without interlocked pipeline stages) is a commercial RISC-ISA. **Opcodes:** MIPS uses 32-bit instructions with 6-bit opcodes. We differentiate between three instruction types: 1) **R-type instructions:** R-type instructions operate on three registers. All R-type instructions have an opcode of 0 and the operation is specified by the funct field. rs/rt are the source registers and rd is the destination register. shamt is only used in shift operations (otherwise 0) and specifies the shift amount. 2) **I-type instructions:** I-type instructions use two register operands and one immediate operand. Examples are addi, lw or sw. In lw/sw, we specify the destination register, a source register + an offset. E.g. (lw $t2, 32($0) or sw $s7, 4($t7)). 3) **J-type instructions:** The only J-type instruction are j (unconditional jump) and jal (jump and link: jump and store return address in $ra). The 4 most significant bits of the target are taken from the program counter and the two least significant bits are set to 0.



Data types: MIPS supports 2's complement integers, unsigned integers and floating point numbers. **Addressing modes:** Immediate, register, PC-relative, indirect, base+offset and pseudo-direct addressing. **Programming:** The most important MIPS instructions are: add $d, $s, $t / addi $d, $s, i / and $d, $s, $t / andi $d, $s, i / div $s, $t (result stored in lo, remainder in hi) / mult $s, $t (upper 32 bits stored in hi, lower 32 in lo) / nor $d, $s, $t (A NOR $0 = NOT A) / or $d, $s, i / sll $d, $s, a (shift left) / slt $d, $s, $t ($d = ($s < $t)) / beq $s, $t, label / bne $s, $t, label / j label / jal label / lb $d, i($s) (load byte, $d = SE(MEM[$s+i])) / lw $d, i($s) (load word)) / sb $t, i($s) / sw $t, i($s) / mthi $d ($d = hi) / mtlo $d ($d = lo) / mthi $s (hi = $s) / mtlo $s (lo = $s). **Generating constants:** To assign 32-bit constants, we use lui and ori. E.g. for 0x6d5e4f3c: lui $s0, 0x6d5e, ori $s0, $s0, 0x4f3c.

Programming constructs: If...else, while, for loops and arrays in MIPS

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

```
# $s0=i, $s1=g, $s2=h, $s3=j, $s4=f
    bne $s3, $s4, else
    add $s0, $s1, $s2
    j L2
else:
    sub $s0, $s0, $s3
L2:
```

```
int pow=1;
int x=0;
while(pow!=128){
  pow=pow*2;
  x=x+1
}
```

```
# $s0=pow, $s1=x
    addi $s0, $0, 1
    addi $s1, $0, 0
    addi $t0, $0, 128
while:
    beq $s0, $t0, done
    sll $s0, $s0, 1
    addi $s1, $s1, 1
    j while
done:
```

```
int sum=0;
for(i=0;i!=100;i=i+8){
  sum=sum+i;
}
```

```
# $s0=i, $s1=sum
    add $s1, $0, $0
    addi $s0, $0, 0
    addi $t0, $0, 100
for:
    beq $s0, $t0, done
    add $s1, $s1, $s0
    addi $s0, $s0, 8
    j for
done:
```

```
int array[5];
array[0]=array[0]+8;
array[2]=array[2]+6;
```

```
# $s0=base address
    lui $s0, 0x1000
    ori $s0, 0x7000
    lw $t1, 0($s0)
    sll $t1, $t1, 3
    sw $t1, 0($s0)
    lw $t1, 4($s0)
    ...
```

Function calls: In MIPS, the calling function (caller) passes up to 4 arguments in registers $a0-$a3 and the called function (callee) passes the return value in registers $v0-$v1. The caller stores the return address in $ra (via jal) and the callee returns to this location. The callee must leave the saved registers, $ra and the stack unmodified. Further arguments are passed on the stack.

The stack: The stack is memory that is used to save local variables within a function. The stack pointer $sp points to the top of the stack and the stack grows down (expands to lower memory addresses when a program needs more space). Functions have to restore the stack before returning.

Microarchitecture

The microarchitecture specifies how the underlying implementation actually executes instructions. It has to obey the semantics of the ISA when making the results visible to the programmer. Anything done in hardware without exposure to software is microarchitecture.

We differentiate the datapath (hardware elements that deal with/transform data) and control path (hardware that determine control signal(s)). Fundamental design principles are: 1) Critical path design (reduce the maximum combinational logic delay) 2) Bread and butter / common case design 3) Balanced design (Design to eliminate bottlenecks; balance hardware for work!)

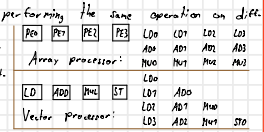
Single-cycle machines: In a single-cycle machine, each instruction takes a single clock cycle. The slowest instruction determines the cycle time. All control signals are generated in the same clock cycle as the one during which data signals are operated on. Performance is measured by {# of instructions} x {clock cycle time}. The control logic is implemented by combinational logic. **Multi-cycle machines:** In multi-cycle machines, each instruction takes as many clock cycles as it needs to take. We have many programmer-invisible states and therefore multiple state transitions per instruction. This is beneficial with regards to the design principles. But we need to store intermediate results which increases hardware costs (additional registers) and register setup/hold overhead is paid multiple times for an instruction. Performance is measured by {# of instructions} x {Average CPI} x {clock cycle time}. In a multi-cycle machine, the instruction cycle is implemented as a FSM that cycles through the states and eventually returns to the fetch state. A state is defined by the control signals asserted in it and control signals for the next state are determined in the current state. There are more control signals than in a single-cycle machine because we also need to control the additional registers. **Microprogrammed multi-cycle architecture:** In this architecture the control signals associated with the current state are called microinstructions. The act of sequencing from one state to another is called microsequencing. A control store stores control signals for every possible state and a microsequencer decides which control signals will be used in the next cycle (i.e. the next state). The microsequencer does this based on the current state, the current instruction, whether the condition of a branch is met and whether the memory operation is completing in the current cycle. Microprogrammed control allows easy extensibility of the ISA and enables update of machine behavior.

Parallelism / performance improvement

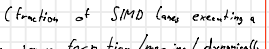
Pipelining: In a pipeline, we divide the instruction processing into „distinct" stages of processing (pipeline stages). We process a different instruction in each stage. The latency of a pipeline/instruction is the time it takes to execute that instruction. The throughput (how many instructions per second) is

determined by the longest stage i.e. $\frac{1}{max(stage)}$. The control signals must be pipelined along with the data so that they remain synchronized (or we could also carry the instruction down the pipeline and decode „locally" in the pipeline stages). **Pipeline stalls:** A stall is a condition when the pipeline stops moving and can have multiple causes: 1) **Resource contention:** Happens when instructions in two pipeline stages need the same resource (eg. the register file). We can eliminate the cause of contention (duplicate resource / increase throughput, e.g. multiple ports or separate caches) or detect the resource contention and stall one of the stages. 2) **Dependencies / Hazards:** We differentiate between data and control dependencies. **Data dependencies:** There are three types of data dependencies: Flow dependence (read after write), output dependence (write after write) and anti dependence (write after read). Only flow dependencies are „true" dependencies, the other exist due to the limited number of architectural registers. **Handling flow dependencies:** There are 5 fundamental ways to handle them. For some of them, we need ways to detect the dependency: **Dependency detection:** 1) **Scoreboarding:** Each register has a valid bit, an instruction that is writing to the register resets the valid bit and an instruction in Decode stage checks if all its source / destination registers are valid. 2) **Combinational dependence logic:** Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction being decoded. To handle the dependency, we can do the following: 1) **Detect and wait:** Simply wait until the value is available. 2) **Detect and forward / bypass:** Forward the result value to the dependent instruction as soon as the value is available (needs additional data paths and a „hazard unit" that controls them). 3) **Detect and eliminate:** Try to detect the dependencies at compile-time and insert enough NOPs or move independent instructions up. 4) **Predict the value, execute „speculatively" and verify** 5) **Do something else (fine-grained multithreading):** Execute instruction from a different thread. **Handling output / anti dependencies:** They can be handled with Register Renaming and a Reorder Buffer. **Control dependencies:** A control dependency is a data dependency on the Program Counter. The problem is to determine which instruction should be fetched next. The next fetch address after a control flow instruction is not determined after N cycles in a pipelined processor. If we are fetching W instructions per cycle, a branch misprediction leads to NxW wasted instruction slots (= Branch misprediction penalty"). There are multiple solutions for handling control dependencies: 1) **Stall the pipeline until we know the next fetch address** 2) **Guess the next fetch address (branch prediction)** 3) **Employ delayed branching (branch delay slot):** N instructions (delay slots) that come after the branch are always executed. The branch must be independent of these instructions. 4) **Do something else (fine-grained multithreading)** 5) **Eliminate control flow instructions (predicated execution):** There are special condition registers where we can evaluate predicates and an instruction is only executed when one or more condition registers are true. Instead of `if (a==5) {b=4;} else {b=3;}` we would have: `CMPEQ condition,a,5; CMOV condition,b,4; CMOV !condition,b,3`. Branches are eliminated and control dependencies become data dependencies. Instructions with false predicates become NOPs. 6) **Fetch from both possible paths (multipath execution):** Useful for hard-to-predict branches, but each path requires its own context and paths to follow can become exponential. **Branch prediction:** A good branch predictor is very important. If we have 20 pipeline stages, a 5 wide fetch, 20% branches and 90% accuracy, to fetch 500 instruction it takes: 100 + 20 + 10 = 300 cycles (an IPC of 500/300 instead of 500/100 with 100% accuracy). The CPI for branches is calculated like: [1 + (prob. wrong guess) * (branch misprediction penalty)]. To reduce the branch misprediction penalty, we can resolve branches early. Branch prediction needs to predict 3 things at fetch stage: 1) Whether the fetched instruction is a branch 2) The branch direction 3) The target address. For 1 and 3 we use a branch target buffer that stores the target address computed last time together with the program counter. For 2, there are multiple ways (predict direction): 1) **Compile-time (static) prediction:** The simplest way is to predict always not-taken. The compiler can lay out code such that „not-taken" is more likely. Accuracy is quite poor (30-40%). Better is always-taken prediction (60-70%) because backwards branches (Loops) are usually taken. Backward taken, forward not taken (BTFN) combines these approaches. **Profile-based prediction:** The compiler determines likely direction for each branch using a profile run and encodes that direction as a hint. Accuracy depends on branch behavior and profile. **Program-based prediction:** Heuristics based on program analysis are used to determine the predicted direction (e.g. „error-check" → usually not taken). **Programmer-based prediction:** The programmer provides the predicted direction with progmas (hints). 2) **Run time (dynamic):** The idea of dynamic branch prediction is to predict branches based on dynamic information (collected at runtime) and requires additional hardware. **Last time predictor:** A single bit per branch is stored in the BTB that indicates if the branch was taken last time. For loops accuracy is (N-2)/N (first/last wrong). We can also use two bits to track the history and only change prediction on two consecutive misses. Accuracy for a loop is (N-1)/N and for many programs 85-90%. **Global branch correlation:** Recently executed branch outcomes are correlated with the outcome of the next branch (e.g. `if (cond1 and cond2) → first not taken, second also not taken`). Therefore we have a global history register (N bits with 1=taken 0=not taken) and a pattern history table (2^N entries with 2 bits per entry; one entry for every possible global history value). The stored 2 bits in the pattern history table work like the 2-bit (last time predictor (4 states). This can be improved further if we hash the global history value with the Branch PC and use this value for the PHT lookup. **Local branch correlation:** In this approach, we have a per-branch history register and a per branch PHT. The prediction is made based on the outcome of the branch the last time the same local history was encountered. This works well for e.g. small loops, because in a „3-times loop" after the local history „0111" the branch will always be not taken (was taken 3 times → loop is over). **Other branch predictors:** Loop branch detector/predictor, perceptron predictor (perceptron „learns" correlation between branch history/outcome) or hybrid history prediction. **Hybrid branch predictors:** They use more than one predictors and have a „meta-predictor" that selects the predictor based on various criteria (confidence, warmup time, ...). **Precise exceptions:** Because not all instructions take the same amount of time, we can have multiple functional units that take different numbers of cycles and can let independent functional units start execution on a different functional unit before a previous instruction finishes execution. But this breaks sequential semantics of the ISA and the architectural state should be consistent (precise) when an exception/interrupt is ready to be handled. This means that all previous instructions and no later should be retired. Precise exceptions aid software debugging, enables recovery from exceptions and restartable processes. **Reorder buffer (ROB):** The idea is that instructions are completed out of order, but they are reordered before making them visible. The ROB stores everything required to correctly reorder operations, update the architectural state and handle exceptions/interrupts (therefore stores valid bit(s), register id/values, memory addr/values, PC, exception info, ...).

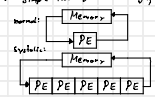
When an instruction is decoded it reserves the next ROB entry. When it completes, it writes results to the ROB and when the oldest instruction in the ROB has completed without exceptions, the architectural state is updated. **Register renaming:** A register has a valid bit and if the register is invalid, it stores the address of the ROB entry that (will) contain the register value (=indirection). When a later operation needs a value in the ROB, it can get it like this. **Out of order execution:** Renaming with ROB eliminates false data dependencies but a true data dependency stalls dispatch of younger instructions into functional units. The idea of OoO-Execution/Dynamic scheduling is to move dependent instructions out of the way for independent ones (into reservation stations). When all source values are ready, the instruction fires. To enable OoO-Execution, there is a register alias table (RAT) that associates tags with values and is used for renaming (→eliminating output/anti dependencies). **Tomasulos algorithm:** 1) If the reservation station is available: instruction + renamed operands (source tag/value) inserted into reservation station, else stall 2) While in reservation station, each instruction: a) Watches common data bus (CDB) for tag of its sources b) When tag seen, grab value for the source and keep it in the reserved station c) When both operands available, dispatch instruction to the functional unit. 3) After instruction finishes in the functional unit: a) Arbitrate for CDB b) Put tagged value onto CBD (tag broadcast) c) If the tag in the register file matches the broadcast tag, write broadcast value into register (and set valid bit). d) Reclaim rename tag **RAT:** The RAT has an entry for each register with tag value and a valid bit. When a rename tag is claimed, the reservation station entry ID can be used (but not necessary). **With precise exceptions:** To enable precise exceptions, a „2 hump" approach is used where the first hump is the scheduler (reservation stations) and the second hump the reorder buffer. Besides the RAT, there is a an architectural register file that is only updated when an instruction retires (from the reorder buffer i.e. in order). **Memory:** Register dependencies are known statically whereas memory dependencies are determined dynamically and memory state is much larger than register state. A problem is that a younger load can have its address ready before an older store (known as the memory disambiguation problem). There are three approaches to the problem: 1) Conservative: Stall the load until all previous addresses are computed. 2) Aggressive: Assume load is independent 3) Intelligent: Predict if the load is dependent. **Data Forwarding:** Memory has to be updated in program order. Therefore there are Load Queues (LQ) and Store Queues (SQ). A load searches the SQ and a store the LQ. This „store-to-load forwarding logic" is implemented as Content adressable memory. **Superscalar execution:** In N-wide superscalar execution, N instructions per cycle are fetched, decoded, executed and retired. There needs to be additional hardware resources and hardware for dependency checking between concurrently-fetched instructions. Ideal IPC is N, but because of dependencies IPC is usually lower. **Very Long Instruction Word (VLIW):** The idea of VLIW is that software (compiler) packs independent instructions in a longer instruction bundle that is executed concurrently. The instructions (in contrast to SIMD) can be typically complicated. In traditional VLIW, instructions in a bundle are statically aligned to be fed into the functional units. VLIW results in simple hardware because there is no need for dynamic scheduling, dependency checking within a VLIW instruction and instruction alignment/distribution to different functional units. But the compiler needs to find N independent operations per cycle, recompilation is required, when N, instruction latencies or functional units change and all instructions in a bundle wait for the slowest to complete. **Fine-grained Multi-threading:** In FGMT, the hardware has multiple thread contexts (PC/registers) and in each cycle, the fetch engine fetches from a different thread. There is no need for dependency checking between instructions (only one instruction in pipeline from a single thread), no need for branch prediction and system throughput/utilization is improved. But single thread performance is reduced, extra hardware is needed, there can be resource contention between threads in cache/memory and some dependency checking logic between threads remain (load/store). **SIMD:** In the „Single instruction multiple data" approach, parallelism arises from performing the same operation on different pieces of data. In an array processor, an instruction operates on multiple data elements at the same time whereas in a vector processor, an instruction operates on multiple data elements in consecutive time steps using the same space. **Vector processors:** A vector processor is a processor that operates on vectors rather than scalars. He has vector data registers that hold N M-bit values and the control registers VLEN, VSTR and VMASK. VLEN indicates the vector length i.e. the nr. of elements stored in a vector register (max. value: N). VMASK indicates which elements of a vector to operate on and is set by vector test instructions. Elements in memory are separated from each other by a constant distance (stride). The VSTR register controls this distance. **Memory banking:** Memory is divided into banks that can be accessed independently; banks share address and data buses (but can also have multiple). We can sustain N parallel accesses if all N go to different banks. Banks can be word-interleaved (consecutive words are stored in consecutive banks) which is ideal for stride=1. Strides should be relatively prime to each other to avoid bank conflicts. But when e.g. the column of a matrix stored in row major format (consecutive elements in a row are laid out consecutive in memory) is accessed, this is not always the case. To minimize bank conflicts, we can have more banks, a better data layout or another mapping of addresses to banks (e.g. randomized). **Vector chaining:** Vector chaining is data forwarding from one vector functional unit to another (e.g. elements of a vector load are forwarded to a vector add although the load hasn't finished i.e. loaded the whole vector). **Vector stripmining:** If there are more elements than the capacity of the vector register, the loop has to be split so that each iteration works on the max. amount. The last iteration normally does not operate on the max. amount except the item count is divisible by the register capacity. **Gather/scatter:** If we have loops with indirect accesses (e.g. A[B[i]]), an index vector is used that is added to the base address to generate the addresses. **Masked vector instructions:** VMASK register is a bit mask determining which data elements should be acted upon. A simple implementation of masked instructions executes all N operations, but turns off result write-back according to the mask. A density-time implementation scans the mask and only executes elements with non-zero masks. **GPUs:** A GPU is a SIMD (SIMT) machine but is programmed using threads (SPMD: Single Program Multiple Data). Each thread executes the same code but operates on different piece of data and each thread has its own context. A set of threads executing the same instruction are dynamically grouped into a warp by the hardware. A warp is therefore a set of threads that execute the same instruction (i.e. at the same PC). SIMT is an execution model of multiple



instruction streams of scalar instructions. It has two major advantages: Each Thread can get treated separately and threads can get grouped into warps flexibly. Warps can be interleaved on the same pipeline (Fine grained multithreading of warps). The thread ID is used to index and access different data elements. **Branch divergence:** Occurs when threads inside warps branch to different execution paths and reduces SIMD utilization (fraction of SIMD lanes executing a useful operation). By dynamic warp formation/merging (dynamically merge threads executing the same instruction (after branch divergence)), SIMD utilization can be improved.



GPU programming: In CUDA/OpenCL programming model, the GPU executes kernels. Threads are arranged in blocks (within a block, shared memory and synchronization) and blocks can be arranged into grids. Hardware is free to schedule thread blocks and thread blocks can be organized into 1D, 2D or 3D arrays. **Performance considerations: Memory access:** To hide latency, FGMT is used. Furthermore, we want to have coalesced memory accesses (concurrent threads access nearby memory locations so they ideally access the same cache line). CPUs prefer Array of Structures (struct foo {float a; float b;} A[8];) whereas GPUs prefer Structure of Arrays (struct foo {float a[8]; float b[8];} A;). By dividing the input into tiles that are loaded into shared memory, we can take advantage of data reuse. Shared memory is banked, therefore we need to pay attention that we don't introduce bank conflicts within a warp (good data layout, padding, randomized mapping or hash functions). **SIMD utilization:** Can be improved by divergence-free mappings. **Atomic conflicts:** Intra-warp conflict degree is the nr. of threads in a warp that update the same memory position. Can be improved by privatization (row sub-block calculation that gets merged). **Data transfers between CPU/GPU:** There are synchronous and asynchronous data transfers. By overlapping communication and computation (dividing 1 stream into multiple ones), performance can be improved. **Systolic Arrays:** The goal of systolic arrays is to design an accelerator that has simple and regular design, high parallelism and balanced computation and I/O performance. A single processing element (PE) is replaced with an array of PEs. Systolic arrays can be multi-dimensional. Data input needs to be carefully orchestrated. Systolic arrays are not good at exploiting irregular parallelism and relatively special purpose (need SW/programmer support).

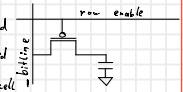


Decoupled Access/Execute (DAE): The idea is to decouple operand access and execution via two separate instruction streams that communicate via ISA-visible queues. The compiler generates both instruction streams and synchronizes the two upon control flow instructions. The execute stream can run ahead the access stream and vice versa. This enables limited out-of-order execution without the wakeup/select complexity. But it needs compiler support and branch instructions need sync. between A and E.

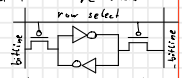
Memory

A memory array is a two-dimensional array of bit cells with 2^M rows (depth: number of words) and N columns (=size of word). When reading, the address decoder activates one wordline. Large arrays are divided into multiple banks that can be accessed independently.

DRAM: Consists of 1 capacitor and 1 access transistor. Capacitor charge indicates stored value. The capacitor leaks, which causes the cell to lose charge over time. Therefore they need to be refreshed.



SRAM: Two cross coupled inverters store a single bit. 4 transistors for storage and 2 for access needed.



DRAM vs. SRAM: DRAM has slower access, higher density, lower cost, requires refresh and manufacturing requires putting capacitor and logic together. The SRAM manufacturing process is compatible with the logic process. **Memory hierarchy:** The ideal memory has zero access time, zero cost, infinite capacity and bandwidth. These requirements oppose each other. Therefore we have multiple levels of storage (that get bigger and slower). **Memory locality:** Caching makes use of two types of locality: 1) **Temporal Locality:** A program tends to reference the same memory location many times → store recently accessed data in cache (fast memory). 2) **Spatial Locality:** A program tends to reference a cluster of memory locations at a time (e.g. instruction memory/arrays) → divide memory into equal size blocks and fetch the accessed block in its entirety. The perceived access time for a memory hierarchy level is: $T_i = t_i + m_i \cdot T_{i+1}$ where m_i is the miss rate of the level. **Cache:** A cache consists of a tag store and a data store. **Direct-mapped cache:** N bits are used to index into the data store (size: 2^N) and the remaining bits as tag. A block can only go to one location. **Associative cache:** In the case of a 3 bit index: Instead of having one column of 8, we have two columns of 4 blocks (2-way associative). Only 2 bits are used for index, rest is tag. **Full associativity:** A block can be placed in any cache location, whole address is tag (→ more comparators needed). **Replacement policy:** In associative caches, there are multiple ways to decide which blocks to replace: Random, FIFO, LRU (least recently used), Not MRU,... **Handling Writes:** Tag store can have a "dirty bit" indicating the block was modified. When we use a write back policy, the modified data gets written to the next level when the block is evicted (in contrast to write through policies). **Critical word:** When we use large blocks we can divide them into sub-blocks and load the requested word first. **Cache misses:** 1) Compulsory miss: first access, always a miss; 2) Capacity miss: Cache too small, would happen even in a fully-associative cache with optimal replacement policy. 3) Conflict miss: all other. **Cache coherence:** We need to ensure that all processors see consistent values. **Virtual memory:** The programmer sees virtual memory and can assume its „infinite". System (SW+HW) maps virtual memory to physical memory. This enables relocation, protection/isolation and sharing. Virtual address space is divided into pages and physical into frames. A virtual page is mapped to a physical frame on a location in disk. If an accessed page is not in memory, system brings it into a frame and adjusts the mapping (demand paging). A page table stores the mapping. A virtual address consists of a virtual page number (VPN) and a page offset. The VPN gets translated to a physical page number and is concatenated with the page offset, resulting in the physical address. Page table can become very large and can result in two memory accesses for one store/load. Therefore, a Translation Lookaside Buffer (TLB) is used that caches the most recently used translations in hardware. Each process has its own page table.

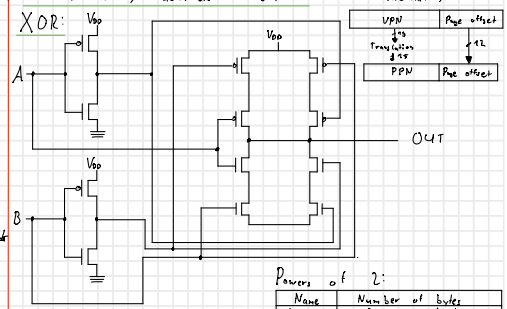
Appendix:

Example mem. address cache structure:

tag	index	byte to word
2 bits	3 bits	3 bits

(where the cache must be)

Virtual memory address translation:



XOR:

Powers of 2:

Name	Number of bytes	
KiB	$2^{10} = 1024$	($\approx 10^3$)
MiB	$2^{20} = 1'048'576$	($\approx 10^6$)
GiB	2^{30}	($\approx 10^9$)
TiB	2^{40}	($\approx 10^{12}$)
PiB	2^{50}	($\approx 10^{15}$)
EiB	2^{60}	($\approx 10^{18}$)