DDCA by dcamenisch

Binary Numbers

An N-bit binary number ranges from 0 to 2^N-1.
The rightmost bit is the most significant bit MSB.
The LSB is defined as the opposit.

Big Endian: grösstes Byte zuerst Little Endian: kleinstes Byte zuerst

$2^0=1$ $2^1=2$ $2^2=4$ $2^3=8$ $2^4=16$ $2^5=32$ $2^6=64$ $2^7=128$ $2^8=256$ $2^9=512$ $2^{10}=1024$ = Kilo $2^{20}=1'048'576$ = Mega

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

2's complement: negative numbers go by inverting every bit then adding 1, MSB used as sign flag.

Boolean Algebra

Rules: $X+X=X$ $X+\bar{X}=1$ $X\cdot(Y+Z)=(X\cdot Y)+(X\cdot Z)$
$X\cdot X=X$ $X\cdot\bar{X}=0$ $X+(Y\cdot Z)=(X+Y)\cdot(X+Z)$

De Morgan: $\overline{(X+Y+Z+..)}=\bar{X}\cdot\bar{Y}\cdot\bar{Z}\cdot...$, $\overline{(X\cdot Y\cdot Z\cdot...)}=\bar{X}+\bar{Y}+\bar{Z}+...$

Product of Sum:

A	B	X	
0	0	1	
1	0	0	$:\bar{A}+B$ (maxterm)
0	1	1	
1	1	0	$:\bar{A}+\bar{B}$

$X=(\bar{A}+B)\cdot(\bar{A}+\bar{B})$

Sum of Product:

A	B	X	
0	0	0	
1	0	1	$A\cdot\bar{B}$ (minterm)
0	1	0	
1	1	1	$A\cdot B$

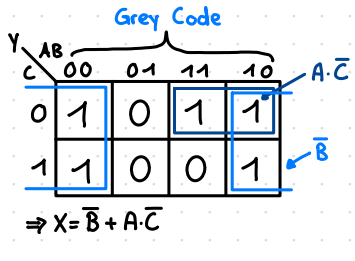
$X=(A\cdot\bar{B})+(A\cdot B)$

NAND Operations

NOT: $\overline{A\,A}$ OR: $\overline{A\,A}\cdot\overline{B\,B}$ AND: $\overline{A\,B}\cdot\overline{A\,B}$

Karnaugh Maps: used to minimize boolean equations, they work well with up to 4 variables. Some rules:
- use fewest circles possible
- only size $2^n\times2^m$
- all must only contain 1's

A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



$\Rightarrow X=\bar{B}+A\cdot\bar{C}$

Logic Gates

AND

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

NAND

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

OR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

XNOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	1

XOR

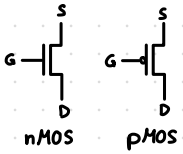
A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

NOT

A	X
0	1
1	0

Transistors

MOS transistors work like a switch there are two types of MOS transistors. pMOS conducts when G is LOW, nMOS is the opposite.



Verilog

D Flip-Flop

```
always @(posedge clk)
  begin
    q <= d;
  end
```

↳ synchronous, if it should be asynchronous add rst to sensitivity list.
→ if d and q are multiple bits wide, the code implements multiple flip-flops.

Full-Adder

```
assign sum = a^b^c_in;
assign c_out = (a&b)|(a&c_in)|(b&c_in);
```

Counter

```
always @* begin
  count_next = count;
  count_next += 1;
end
always @(posedge clk)
  count <= count_next;
```

Correct Code:

- wires: can't be on the left of =/<= in an always block, are used to connect input and output.
- regs: can't be connected to output port of a module, can't be used in input port declaration, only on the left in always blocks.
- I/O: check if names match and if all ports are assigned.
- not multiple assignments to the same signal
- names can't start with numbers: ~~2good~~
- no module recursion

Combinational vs. Sequential:

Comb: all left hand signals get assigned, all inputs are in sensitivity list, all outputs are assigned

Building Blocks

MUX: 
select one input, $\log_2(n)$-bit control signal

Decoder: 
n inputs, 2^n outputs, one output is 1 depending on they input pattern

PLA: array of AND gates, followed by an array of OR gates

Tri-State-Buffer:  enables gating of signals onto wire

Combinational / Sequential Logic

Comb:
- no memory
- no cyclic paths
- combines inputs to get output
- signals are assigned in every possible condition
ex. MUX
- all inputs in sensitive list

Seq:
- has memory
  - depends on prior inputs
ex. Flip-Flops (register)

Finite State Machine

Goes through different state, where each state depends on the prev. state and the input.

Moore: Output depends only on current state
Mealy: Output depends on the current state and the input

State Encodings:
- Binary Encoding (minimizes flip-flops), 00, 01, 10, 11
- One-Hot (max. flip-flops, min. next state logic), 0001 0010 0100 1000
- Output (min. output logic)

Designing a FSM

- identify inputs/outputs
- state transition diagram
- write state transition & output table
- write boolean equations for next state

Area of FSM

- # FF = # bits for state x2
- # logic gates = count next-state and output logic

Correctness of state diagram

- reset-line
- not multiple transitions for the same input
- no missing transitions
- no unmarked transitions
- initial state (if no reset)
- no mix of Moore/Measly labeling

MIPS

J-Type: Jump / Branch Instructions
R-Type: Register for all operands (OP=0)
I-Type: Instructions with an immediate/constant value

The caller calls a function, while the callee gets called.
The caller needs to take care of the temporary registers \$t0 - \$t9, while the callee needs to save and restore the preserved registers \$s0 - \$s7.

ISA and Microarchitecture

The ISA is the interface between software and hardware ("what the programmer sees").

The microarchitecture specifies the underlying implementation that actually executes the instructions.

Blocking and Non-Blocking Assignment Guidelines

1) Use always @(posedge clk) and non-blocking (<=) to model synchronous sequential logic
2) Use continuous assignments to model simple combinational logic
3) Use always @(*) and blocking (=) assignments to model more complicated combinational logic where the always statement is helpful
4) Do not make assignments to the same signal in more than one always statement or continuous assignment.

ISA vs. Microarchitecture

ISA

- Instructions: opcodes, adressing modes, data types, instruction type and format, registers, condition codes
- Memory: adress space, alignment, adressability, virtual memory management
- Call, interrupt and exception handling
- I/O: memory mapped vs. instructions
- Power & Thermal management
- Multiprocessing / Multithreading support
- Access control, priority and privilege
- Memory-mapped location of exception vectors
- Function of each bit in a programmable branch prediction register
- Order of execution of loads and stores in multi-core CPU
- Program counter width
- Hardware FP-exeption support
- Vector instruction support
- CPU endianness
- Virtual page size
- SIMD instruction support
- Support for branch prediction hints conveyed by compiler
- Ability to flush a cache line using OS

Microarchitecture

- Pipelining
- In-order vs. Out-of-Order exec.
- Memory adress scheduling policy
- Speculative execution
- Superscalar processing
- Clock gating
- Caching: levels, size, associativity, replacement policies
- Error correction
- Physical structure
- Instruction latency
- Physical memory page size
- Instruction issue width
- reservation stage capacity
- # pipeline stages
- latency of branch miss prediction
- fetch width of superscalar CPUs
- # non-programmable CPU registers
- # of integer arithmetic and logic units (ALUs)
- Size of Reoder Buffer in an Out-of-Order CPU

Performance Evaluation

- CPI: cycles per instruction
- IPC: instruction per cycle
- MHz: frequency, 10^6 cycles/sec.
- MIPS: million instructions / sec. = MHz/CPI
- Time = # instr. · CPI · $\frac{1}{Hz}$
- Speedup = oldTime/newTime
- higher MHz ⇏ higher MIPS, IPS could be lower
- higher MIPS ⇏ less time, could need more instructions

Single-Cycle Machines

Each instruction takes a single clock cycle and all state updates are made at the end of the cycle.

- − slowest instruction determines cycle time
- + easy to build

Multi-Cycle Machines

Instruction processing is broken into multiple stages/cycles, state updates happen during execution and architectural updates at the end. Instruction processing consists of two components:

- Datapath – relay and transform data
- Control logic – FSM that determins control signals
- + slowest stage determines cycle time

Dataflow

In a dataflow machine, a programm consists of dataflow nodes. A node fires (executes) when all its inputs are ready.

write-through
data written to cache block is simultaneously written to main memory

write-back
dirty-bit (D) is associated with each cache block. Written back to main memory when evicted from cache

Pipelining

The idea is to process multiple instructions at once by keeping each stage occupied. In reality there are a few problems:

- Resource contention, can be fixed by duplication, increased throughput or detection and stalling
- Long latency operations
- Data dependencies, there are flow (read after write), output (write after write) and anti (write after read) dependencies. The last two exist due to a limited amount of registers.

Handling flow dependencies

- stall
- eliminate at software level
- predict values
- data forwarding
  - ↳ W → D: internal / register file forwarding
  - ↳ M → E_1: operand forwarding
- do something else (fine-grained multithreading)

Pipeline Stages

- Fetch: CPU reads instructions from instruction memory
- Decode: CPU reads source operands from register file and decodes instruction to control signals
- Execute: CPU performs a computation with the ALU
- Memory: CPU reads/writes data memory
- Writeback: CPU writes result to register file

Interlocking & Scoreboarding

Detection of data dependencies to ensure correct execution.

Out-of-Order Execution

Idea to move dependent instructions out of the way of independent ones. Reservation stage as rest are for dependent instructions.

Reorder Buffer

Complete instructions OoO but reorder them before making results visible to architectural state.

Tomasulo's Algorithm

Implementation of OoO-Execution. Uses register renaming to eliminate output and anti-dependencies. It further uses reservation stations for individual operations.

1. If reservation station is available:
   - instr. + renamed operands inserted into reservation station
   - rename destination register

   Else stall
2. While in reservation station:
   - watch common data bus for tag of sources
   - if tag seen grab value
   - if both operands are valid inst. ready for dispatch
3. Dispatch instr. to functional unit
4. After instr. finishes:
   - put tagged value onto common data bus
   - if register file contains tag, upate its value and set valid bit
   - reclaim rename tag → no valid copy of tag in the system

VLIW

The idea is that the compiler finds independent instructions and statically schedules them into single VLIW instructions.

Lock step execution: if one instruction stalls, the whole VLIW stalls

- + simple hardware
- + no dependency checking
- + no instruction distribution
- − compiler needs to find N independent instructions per cycle (complex)
- − lock step causes stalls

Superscalar Execution

Idea is to fetch/decode/... multiple instructions per cycle.

- + higher IPC
- − higher complexity for dependency checking ⇒ more hardware

Systolic Arrays

Instead of a single processing element (PE) we have a array of PE and carefully orchestrate the dataflow between them ⇒ Maximize computation done on a single element.

Difference from pipelining: Array structure is non-linear and multi-dimensional. PE connections can be multi-directional with different speeds. PEs can have local memory and execute kernels.

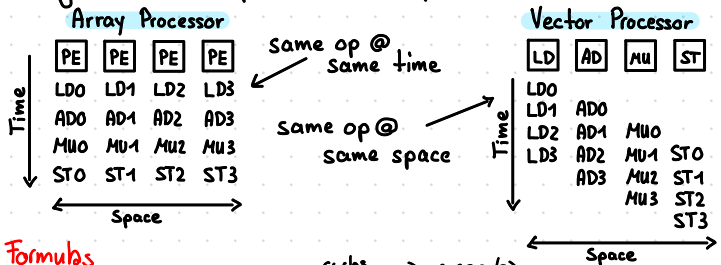
Fine Grained Multithreading

Hardware has multiple thread contexts (PC + reg) and each cycle the fetch engine fetches from a different thread.

- + no dependencies
- + no branch prediction
- + improved throughput, latency, tolerance, utilization
- − extra hardware
- − reduced single thread performance
- − resource contention
  - ↳ dependency checking between threads

SIMD

Single instruction operates on multiple data.



Formulas

$$\text{Execution time} = (\#\text{ instructions})\left(\frac{\text{cycles}}{\text{instructions}}\right)\left(\frac{\text{seconds}}{\text{cycle}}\right)$$

CPI = Cycle per Instr.

$$\text{Miss Rate} = \frac{\text{Number Of Misses}}{\text{Number Of Total Memory Accesses}} = 1 - \text{Hit Rate}$$

$$\text{Average Memory Access Time (AMAT)} = t_{cache} + MR_{cache}(t_{MM} + MR_{MM}\, t_{VM})$$

↳ MM = Main Memory, VM = virtual memory, MR = Miss Rates

Number of set bits: $\log_2 S$

Remaining tag bits indicate memory address of the data stored in a given cache set

Two least significant bits → Byte offset

Vector Processing

Performs operation on a whole array. This is only possible if the operations on each element are independent from each other.

The data gets stored in vector registers. Vector chaining describes the vector version of data forwarding. It allows a operation to start as soon as an individual element is ready. The stride is the distance of the vector elements in memory. If a vector is to long it can be split into multiple vectors (strip mining).

+ a lot of work per instruction
+ regular memory access pattern
+ no need for loops
- works only if parallelism is regular, else it is very inefficient

GPU

GPUs are SIMD engines but programmed using threads (SPMD). A set of threads executing the same program are grouped into a warp.

Dynamic Warp merging: Merge threads executing the same instr. after branch divergence. This forms new warps from the warps waiting.

Delayed Branching

Means that some instructions after a branch are executed regardless of which way the branch goes. A compiler can instructions in such a delay slot if they don't influence the branch itself, else they are filled with NOPs.

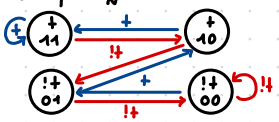
Branch Prediction

A technique used to predict the next adress after a branch. If the prediction is wrong, we have to flush the pipeline (misprediction penalty).

Prediction Direction Schemes

static
- always (not)-taken (30-40%) 60-70% accuracy
- BTFN: backwards taken forwards not taken (good with loops)

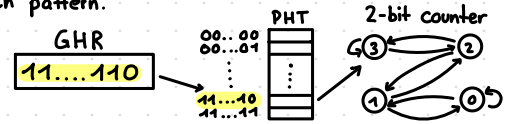
dynamic
- Last time predictor: single bit stored in BTB (branch target buffer) indicates last direction. Loop accuracy = $\frac{N-2}{N}$.
- 2-bit counter based prediction:
- Local: no interference between different branches
- Global: single counter for all branches



Global Branch Correlation

The idea is that recently executed branch outcomes are correlated with the outcome of the next branch.

- First level: Global branch history register, keeps track of the last branch outcomes.
- Second level: Pattern history table, keeps a 2-bit counter for each pattern.

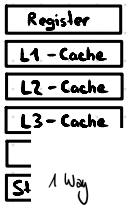


Memory Hierarchy

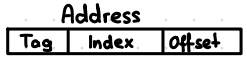
Memory Array: stores data, address selection logic selects row, readout circuitry reads data.

Memory banking: Multiple memory units with a common data and adress bus, helps to resolve long latency. Units can be accessed individually.

Locality: temporal = access to same address in short time
spatial = access to nearby address



Blocks & Addressing cache:
- memory is divided into fixed-size blocks
- each block maps to a location in the cache (index bits)
- for a cache hit the tags need to match



Offset = Byte im Block
Index = Zeile in Tag Store
Tag = Welcher Block im Set

Associativity:
- multiple blocks have the same index → conflict misses
- n-way associative allows n-Blocks with same index
- 1-way → direct mapped no index → fully associative

Cache Performance:
- cache size, total data c
- block size b
- associativity n
- #blocks: B = c / b
- #sets: S = B / n

Replacement Policies:
- FIFO, first-in-first-out
- LRU, least-recently-used
- Random

Handling Writes

Writeback: write to lower levels when the block is evicted, needs a dirty bit.

Write through: write to all levels immediatly, simpler but bandwith intensive.

Classification of Misses

Compulsory Miss: first refrence is always a miss (prefetching)

Capacity Miss: cache is too small

Conflict Miss: all other misses (more associativity)

Improvement Ideas
- reduce miss rate
- reduce miss latency or cost
- reduce hit latency or cost

bits of storage = associativity · (#sets · (tag + dirty + valid) + LRU)

Prefetching

The idea is to improve cache performance by preloading data to avoid misses. There are different techniques to prefetching, some are software based while others hardware dependent.

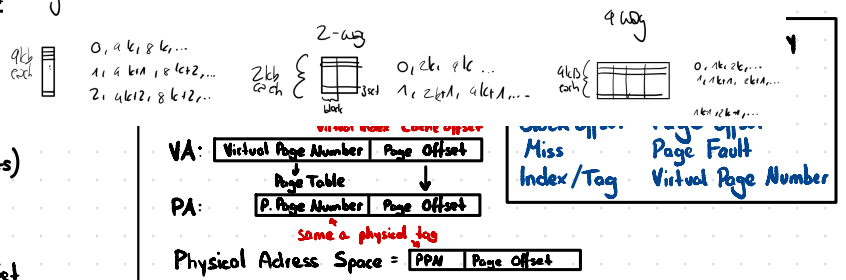
Stride prefetcher: prefetches cache block in a pattern with a certain stride (if stride = 0, next block prefetching)

Runahead execution: allows the processor to pre-process instr. during cach misses instead of stalling. Therefore it can detect potential cache misses earlier.

- accurracy = #pref. used / #pref total
- coverage = #acc. predicted / #total acc.

Virtual Memory

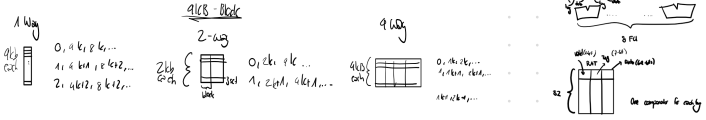
Much larger than physical memory. Virtual adress space is divided in pages, while physical adress space is divided al



Multi-level page tables: keeps PT size small
Memory protection: different PT for each programm
Translation Lookasid Buffer TLB: cache PT entries to speed up adress translation

Cache capacity = (Block Size in Bytes) · (Blocks per Set) · (Number of Sets)
Index Bits = $\log_2$ (Blocks per Set)
Block Offset Bits = $\log_2$ (Block Size in Bytes)
Tag Bits = (Address Bits) − (Index Bits) − (Block Offset Bits)

Software Interlocking: NOPs
Hardware Interlocking: Detects dependencies and stalls pipeline accordingly



Ex. Find the simplest sum-of-products form for this equation: $F = B + (A+\bar{C})\cdot(\bar{A}+B+\bar{C})$

$F = B + A\bar{A} + A\bar{B} + A\bar{C} + \bar{C}\bar{A} + \bar{C}\bar{B} + \bar{C}$
$= B + A + \bar{C}$

Ex. Simplify the following min-terms: $\Sigma(3,5,7,11,13,15)$.

$\{3,5,7,11,13,15\} = \{0011, 0101, 0111, 1011, 1101, 1111\}$

$F = \bar{A}\bar{B}CD + \bar{A}B\bar{C}D + \bar{A}BCD + A\bar{B}CD + AB\bar{C}D + ABCD$
$= CD\cdot(\bar{A}\bar{B} + \bar{A}B + A\bar{B} + AB) + BD\cdot(\bar{A}\bar{C} + A\bar{C})$
$= CD + BD\bar{C}$
$= D(B+C)$

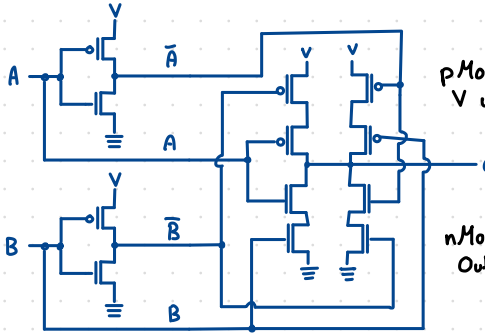
Ex. Convert the following equation to only contain NANDs.

$F = (\overline{A\bar{B}} + C) + AC = \overline{(\overline{A\bar{B}} + C)\cdot\overline{AC}} = \overline{(A\bar{B}\cdot\bar{C})\cdot\overline{AC}}$
$= \overline{A\bar{B}\cdot\overline{AC}} = \overline{\overline{A\bar{B}}\cdot\overline{A\bar{B}}\cdot\overline{AC}}$

Ex. Convert the following equation to only contain NANDs.

$F = \overline{(A+B\bar{C})} + \bar{C} = \overline{\overline{(A+B\bar{C})}+\bar{C}} = \overline{(A+B\bar{C})\cdot C} = \overline{\overline{(A+B\bar{C})\cdot C}}$
$= \overline{\overline{(A\cdot\overline{B\bar{C}})\cdot C}} = \overline{\overline{(A\cdot A)\cdot(B\cdot C)\cdot C}}$

Ex. Draw a XOR-Gate with transistors.



pMos immer zwischen V und Out

nMos immer zwischen Out und Drain

Ex. Sequential or Combinational circuit?

```
module one (input clk, input a, input b, output reg [1:0] q);
  always @(*)
    if (b)
      q <= 2'b01;
    else if (a)
      q <= 2'b10;
endmodule
```

This code results in a sequential circuit because a latch is required to store old values of q if both conditions are not satisfied.

Ex. Is this code a correct multiplexer?

```
module four(input sel, input [1:0] data, output reg z);
  always @(sel)
    if (sel)
      z = data[1];
    else
      z = data[0];
endmodule
```

No, the input data is missing in the sensitivity list. A update would not be reflected to the output z.

Ex. Does this result in a D-FlipFlop with a synchronous active-low reset?

```
module mem (input clk, input reset, input [1:0] d, output reg [1:0] q);
  always @(posedge clk or negedge reset)
    if (!reset) q <= 0;
    else q <= d;
endmodule
```

The code implements 2 D-FlipFlops, each works with a asynchronous active low reset.

Ex. Is this code syntactically correct?

```
module fulladd (input a, b, c, output reg s, c_out);
  assign s = a^b;
  assign c_out = (a&b) | (a&c) & (b&c);
endmodule
```

we use assign therefore these have to be wires

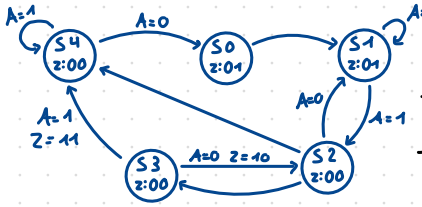
```
module top (input wire [5:0] instr, input wire op, output z);
  reg [1:0] r1, r2;
  wire [3:0] w1, w2;
  fulladd FA1 (.a(instr[0]), .b(instr[1]), .c(instr[2]),
               .c_out(r1[1]), .z(r1[0]));
  fulladd FA2 (.a(instr[3]), .b(instr[4]), .c(instr[5]),
               .c_out(r2[1]), .z(r2[0]));
  assign z = r1 | op;
  assign w1 = r1 + 1;
  assign w2 = r2 << 1;
  assign op = r1^r2;
endmodule
```

should be wires

no .z port

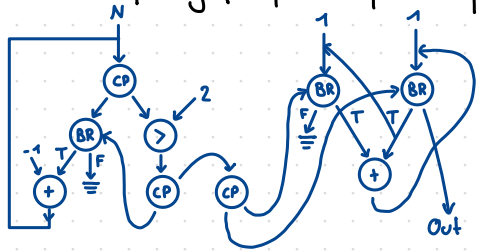
multiple drivers

Ex. List all the mistakes in this diagram.



There are four mistakes:
- most states have Moore labeling but some Mealy
- There are 2 A=1 transitions from S1
- There 2 unabled transitions from S2
- There is no reset

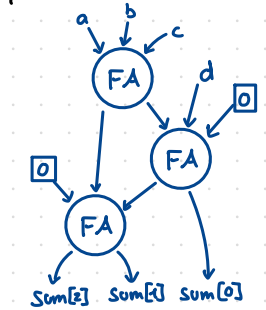
Ex. Draw a data flow graph for the fibonacci function.



Ex. Which designs are compatible with each other?

superscalar – in-order
superscalar – out-of-order
single cycle – branch prediction
reservation station – microprogramming
fine-grained multithreading – single core
precise exceptions – out-of-order retirement
branch prediction – fine-grained multithreading
fine-grained multithreading – pipelining
Tomasulo's algorithm – in-order
direct mapped cache – LRU replacement

Ex. Draw the dataflow graph for a four 1-bit addition, you can use Full Adder nodes.



Ex. Any $n \geq 3$ 1-bit addition can be implemented only using Full Adders. Fill out the table.

n	#required FAs	n	#required FAs
3	1	6	4
4	3	7	4
5	3	8	7

Ex. Two programms A, B run on the same machine, both have the same # memory requests, but A needs to stall way more. Why could this be?

A could have a lot of row buffer conflicts, while B has a lot of row buffer hits.

Ex. If a processor executes more IPS, does a program always finish faster?

No, the number of instructions for a program could be different for different processors.

Ex. If a program runs on a processor with a higher frequency, does this imply that it executes more IPS?

No, a processor with a lower frequency can have a much higher number of IPC.

Ex. Write a MIPS 64-bit substraction (2s-complement) where {$4 $5} - {$6 $7}.

```
subu   $3, $5, $7
sltu   $2, $5, $7
add    $2, $6, $2
sub    $2, $4, $2
```

Ex. A machine with 5 pipeline stages uses delay slots to handle control dependences. Jump and branch are resolved during execution stage. How many delay slots are needed?

2, since we can fill them during fetch and decode of the jump / branch instruction.

Can we modify the pipeline to reduce the number of delay slots?

Yes, if we move the resolution of the jump/branch target to the decode stage, we only need one delay slot.

Ex. How many delay slots are needed for the following implementations?
In-order with branch resolving during 4th stage: 3
OoO with 64 reservation stages, branch resolving during 2nd cycle of branch execution and 15 stages befor the execution stage: Don't know

Ex. Given the following microbenchmark for a pipelined machine. Calculate #dynamic instructions executed, # pipeline stages and #cycles of stall caused by branch instruction.

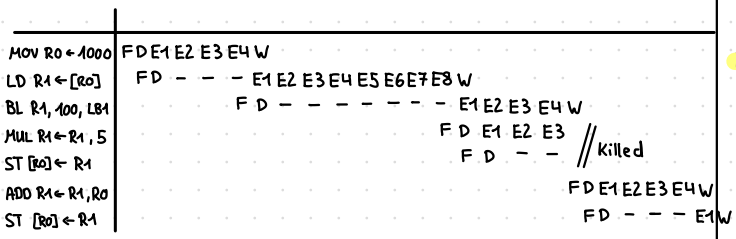
		Initial R1	#Cycles
LOOP1:			
	SUB R1, R1, #1	4	51
	BGT R1, LOOP1	8	63
LOOP2:		16	87
	B LOOP2	all runs execute the same	#dynamic instr.

Let: C = #cycles
P = #stages
I = #dynamic instr.
B = #branch instr.
D = #cycles stall / branch

$C = P + I - 1 + B \cdot D$

$51 = P + I - 1 + 4D$
$63 = P + I - 1 + 8D$
$87 = P + I - 1 + 16D$
$\Rightarrow P + I = 40,\ D = 3$

Ex. Given a scalar processor with in-order fetch, out-of-order dispatch and in-order retirement. It has 4 pipeline stages, and 2 reservation stations (one for each type). If the following program gets executed, answer the questions?



Cache hit latency? 1 cycle, the last ST instr. is a hit.
Cache miss latency? 8 cycles, the first LD instr. misses.
Cache line size? Unknown
#entries in each reservation station? ALU at least 2, MU unknown
#ALUs? if pipelined at least 1, else at least 2.
Is the ALU pipelined? If there is only 1 ALU yes, else unknown
Does the processor have branch prediction? Yes, because there are instr. that get killed.
At which stage do branches get resolved? At the end of E4, because in the next cycle the previously fetched instr. get killed.

Ex. Given Tomasulos Algorithm with: 8 functional units with their own tag/data bus, 32x64 bit registers, 16 reservation stations p.r. functional unit and 2 source register per reservation station, calculate:

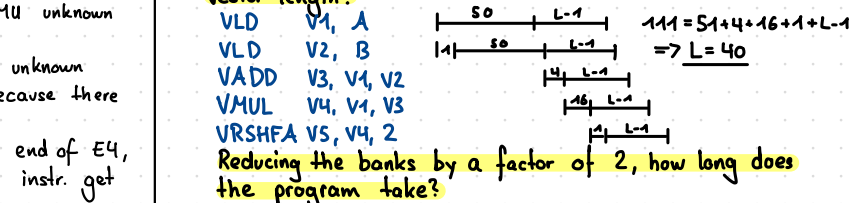
#tag comparator / reservation station entry = $2 \times 8 = 16$
#tag comparators = $16 \times 16 \times 8 + 32 \times 8 = 2304$
min. tag size = $\log(16 \times 8) = 7$
min. size of register alias table = $32 \times (7 + 64 + 1) = 2304$
min. total size of tag store = $32 \times 7 + 8 \times 16 \times 2 \times 7 = 2016$

Ex. Comparing a VLIW and a in-order superscalar processor with the same machine width and frequency. For a program A, the VLIW machine is much faster, why could this be?
The superscalar proc. is in-order, requiring bubbles in the pipeline, while the VLIW proc. can reorder instr.
For some other program B, the VLIW is slower, why could this be?
VLIW needs NOPs, while the superscalar proc. doesn't. These NOPs can lead to lower I-cache hit rate and higher fetch bandwith.

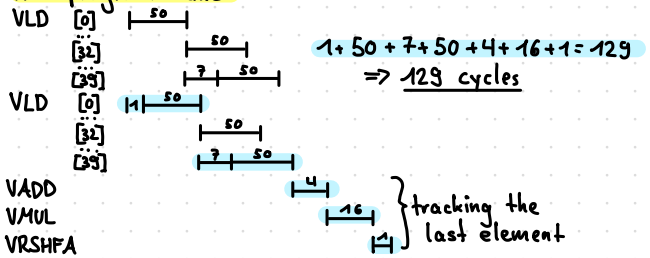
Ex. Which of the following are goals of VLIW?
i. Simplify code compilation
ii. Simplify application development
(iii.) Reduce overall hardware complexity
(iv.) Simplify hardware dependence checking
v. Reduce processor fetch width

Ex. Given a vector proc. with these fully interleaved / pipelined instr. VLD/VST 50 cycles, VADD 4 cycles, VMUL 16 cycles, VDIV 32 cycles and VRSHFA 1 cycle. Assume: in-order pipeline, chaining between functional units, first element bank 0, 8KB row buffer / bank, 64 bit vector elements, each memory bank has 2 ports and there are 2 load/store units. What is the minimum (power of 2) # banks so memory accesses never stall?
64 banks, because access latency is 50ms and 64 the next power of 2.

Executing this program takes 111 cycles, what is the vector length?



Reducing the banks by a factor of 2, how long does the program take?



Now the #banks get reduced further and it takes 279 cycles. How many banks are there?

$279 = 1 + 16 + 4 + 1 + 7 + \left\lceil \frac{40}{x} \right\rceil \cdot 50$
$\Rightarrow 5 = \lceil 40/x \rceil \Rightarrow x = 8$ memory banks

In a new version 4 vector proc. share the same memory with 4 times the banks. However the execution is slower than if each program ran on a single proc. with 1/4 banks, why could this be?
Row buffer conflicts as all cores interleave their vectors across all banks.
How can this be fixed?
Partition the memory mappings, or use better memory scheduling.

Ex. Consider the following warps, how can dynamic warp formation be used?

$X = \{1\,0\,0\,1\,0\,1\,1\,1\}$ $X' = \{1\,0\,0\,1\,0\,1\,1\,1\}$
$Y = \{1\,0\,0\,0\,1\,0\,0\,1\}$ $\Rightarrow$ $Y' = \{1\,1\,0\,0\,1\,0\,0\,1\}$
$Z = \{0\,1\,0\,0\,0\,0\,0\,0\}$ $Z' = \{0\,0\,0\,0\,0\,0\,0\,0\}$

There are several answers, but notice that X, Y can't be merged.

Ex. How effective is a 16KB, 4-way assocative cache with 8B instructions?
Not effective, since the block size is 4B, each instruction needs two accesses. Further it can't exploit spatial locality.

Ex. Given the following access pattern and hit rate for a cache determine it's characteristics.

	Addresses Accessed						must miss		Hit rate
1.	0	4	8	16	64	128			1/2
2.	31	8192	63	16384	4096	8192	64	16384	5/8
3.	32768	0	129	1024	3072	8192		hit	1/3

Cache block size: 8, 16, 32, 64 or 128 B
From ① we can see that only 32 or 64 are possible. From ② we can see that 63 must be a hit and therefore it can only be 64B.
Cache Associativity: 1, 2, 4 or 8 way
Combining this with the possible cache sizes of 4 or 8KB we can see that 1 and 2 way would cause too much misses in ② and 8 way would cause another miss in ③. therefore it must be 4 way.
Cache size: 4 or 8 KB
In ③ the access to 0 is a miss and therefore 8192 should be a hit, but with 4KB, 1024 and 3072 would map to the same set and therefore it couldn't be a hit.
So cache size must be 8KB.
Replacement Policy: LRU or FIFO
For 8192 to hit in ③ it must be LRU.

Ex. Given a one level cache with 128 B and block size 32 B. Using LRU, the following blocks are accessed:

A B A H B G H H A E H D H G C C G C A B H D E C C B A D E F

In a direct mapped cache which blocks are in the same set?

A/B H/D G/C E/F

For a fully associative cache, write down the misses.

A B A H B G H H A E H D H G C C G C A B H D E C C B A D E F

Ex. Given a 2-way assoc. write back cache with LRU and a $2^9 \times 15$ bit tag store. It is virtually indexed, physically tagged. The virtual adress space is 1 MB, page size 2 KB and block size 8 B. What is the size of the data store?

Tag store $= 2^i \cdot (2 + t + 5) = 15 \times 2^9 \Rightarrow i = 9 \quad t = 5$

Data store $= 2^9 \times 8 \times 2 = \underline{8KB}$

How many bits of the virtual index come from the VPN?

1	Index (9)	block (3)

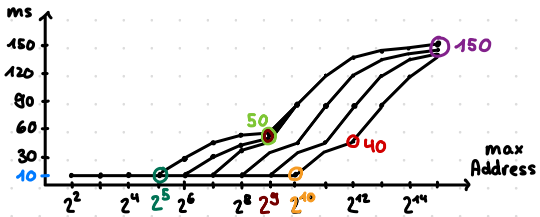
(11 bit page off.) $\Rightarrow$ 1 Bit

What is the physical address space?

Page Offset = 11 Bits Page Tag = 5 Bits

$\Rightarrow 2^{11} \cdot 2^5 = 2^{16} = 64KB$

Ex. Fill in the blanks?



	L1	L2	L3	DRAM
line size	N/A	X	X	N/A
cache assoc.	$2^{10}/1024^*$ = 1	$2^{12}/1024^*$ = 4	X	X
cache size	$2^5 = 32$	$2^9 = 512$	X	X
access latency	10ms	40ms	X	150 − 50 = 100ms

*max stride

Ex. Give a 4-way assoc. write back cache with a $2^{11} * 89$ bit tag store, a 9 bit replacement policy, 64 B blocks. It is virtually indexed, physically taged and data from a given adr. can be in up to 8 sets. It uses a 2 level page table with each 1024 entries. How many bits of the virtual address are for the set?

Tag Store $= 2^{11} * 89 \rightarrow 11$ Bits

What is the size of the data store?

$2^{11} \cdot 4 \cdot 64B = 512$ KB

How many bits in the PPN overlap with the index bits in the virtual adress?

3, since data can be present in up to $2^3 = 8$ sets.

Draw the virtual address:

index (16–6)	offset (5–0)

11 bit index, $2^6 = 64$

Draw the physical address:

frame nr. (31–14)	offset (13–0)

18 bit physical tag; virtual address − 3 bits

What is the page size? $2^{14} = 16KB$

What is the virtual address space?

$2^{VPN} \cdot 2^{Page\ Offset} = 2^{20} \cdot 2^{14} = 16\,GB$ (VPN: 2 level; 20: from 1024 · 2)

What is the physical address space? $2^{32} = 4\,GB$

Ex. What is the prefetch accuracy and coverage for A, B using a stride prefetcher?

A:
```
int[100] a;
sum = 0;
for (i = 0; i < 1000; i += 4)
    sum += a[i]
```

B:
```
int[100] a;
sum = 0;
for (i = 0; i < 1000; i *= 4)
    sum += a[i]
```

Accuracy: A: $\frac{248}{249}$ B: 0 Coverage: A: $\frac{248}{250}$ B: 0

Ex. Given this code explain which branches correlate locally / globally.

```
for (int i = 0; i < N; i++) {   //B1
    val = array[i];
    if (val % 2 == 0)   //B2
        sum += val
    if (val % 3 == 0)   //B3
        sum += val
    if (val % 6 == 0)   //B4
        sum += val
```

Locally: only B1, since for B2, B3, B4 the previous value does not matter.

Globally: B4 is correlated with B3 and B2. If one B4 is taken, B2 and B3 are also taken.

Ex. For the same code, calculate the expected value for the PHTE taken-taken after 120 iterations.

W.l.o.g. we take a look at the numbers 1-6. For a single iteration we have 4 chances to increment the PHTE.

- B3: Given Pr[B1.T & B2.T] = 1 · ½ the probability for B3 to be taken is ⅓, resulting in 1 · ½ · ⅓ = ⅙ probability to increase and 1 · ½ · (1 − ⅓) = ⅓ to decrease, therefore B3 contributes ⅙ − ⅓ = −⅙.
- B4: Pr[B2.T & B3.T] = ⅙ ⇒ ⅙ · 1 = ⅙
- B1: Pr[B4.T & B3.T] = ⅙ ⇒ ⅙ · 1 = ⅙
- B2: Pr[B4.T & B1.T] = ⅙ ⇒ ⅙ · ½ − ⅙ · ½ = 0

Resulting in a total of ⅙ per iteration. Therefore after 120 iterations, the expected value is $\underline{\underline{20}}$.

Fibonacci Code in Assembly

```
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;
}
```

```
fib:
addi $sp, $sp, -16    //allocate stack space
sw   $16, 0($sp)      //save r16
add  $16, $4, $0      //r16 for arg n
sw   $17, 4($sp)      //save r17
add  $17, $0, $0      //r17 for a, init to 0
sw   $18, 8($sp)      // save r18
addi $18, $0, 1       // r18 for b, init to 1
sw   $31, 12($sp)     // save return address
add  $2, $17, $18     // c = a+b 3rd saved on stack, not important

branch:
slti $3, $16, 2       // use r3 as temp
bne  $3, $0, done
add  $2, $17, $18     // c=a+b
add  $17, $18, $0     // a=b
add  $18, $2, $0      // b=c
addi $16, $16, -1     // n=n-1
j    branch

done
lw   $31, 12($sp)     //restore r31
lw   $18, 8($sp)      //restore r18
lw   $17, 4($sp)      //restore r17
lw   $16, 0($sp)      //restore r16
addi $sp, $sp, 16     //restore stack pointer
jr   $31              //return to caller
```

Callee saved: other callee can change registers

Caller saved: callee cannot change them