

# 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 opposite.

**Big Endian:** größtes Byte zuerst

**Little Endian:** kleinstes Byte zuerst

$$\begin{array}{llll} 2^0 = 1 & 2^3 = 8 & 2^6 = 64 & 2^9 = 512 \\ 2^1 = 2 & 2^4 = 16 & 2^7 = 128 & 2^{10} = 1024 \quad \text{= Kilo} \\ 2^2 = 4 & 2^5 = 32 & 2^8 = 256 & 2^{11} = 1'048'576 \quad \text{= Mega} \end{array}$$

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

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

## Product of Sum:

Sum of Product:		
A	B	X
0	0	1
1	0	$\bar{A} + B$
0	1	$\bar{A} \cdot \bar{B}$
1	1	$\bar{A} + \bar{B}$

Product of Sum:		
A	B	X
0	0	0
1	0	1
0	1	0
1	1	1

$\maxterm$        $\minterm$

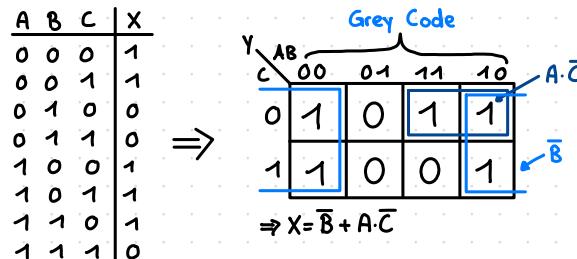
$X = (\bar{A} + B) \cdot (\bar{A} \cdot \bar{B})$        $X = (\bar{A} \cdot \bar{B}) + (A \cdot B)$

## NAND Operations

NOT:  $\overline{AA} = \overline{A}$    OR:  $\overline{AA \cdot BB} = \overline{A} + \overline{B}$    AND:  $\overline{AB} \cdot \overline{AB} = A \cdot B$

**Karnaugh Maps:** used to minimize boolean equations, they work well with up to 4 variables. Some rules:

- use fewest circles possible
- only size  $2^n \times 2^m$
- all must only contain 1's



## Logic Gates

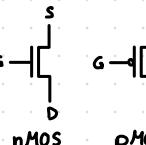
AND		NAND		OR		NOR	
A	B	A	B	A	B	A	B
0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	1
1	0	0	0	1	0	1	1
1	1	1	0	1	1	1	0

XNOR		XOR		NOT	
A	B	A	B	A	X
0	0	1	0	0	1
0	1	0	0	0	1
1	0	0	1	1	0
1	1	1	0	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.

## Correct Code:

- wires: can't be on the left of  $= / \leftarrow$  in an always block, are used to connect input and output.
- egs: 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

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

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

Decoder:

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

## Combinational / Sequential Logic

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

## 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),   
- 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  $\times 2$

# 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/Mealy 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

- Use  $\delta$  (posedge clk) and non-blocking ( $\leftarrow$ ) to model synchronic sequential logic
- Use continuous assignments to model simple combinational logic
- Use always  $\delta(*)$  and blocking (=) assignments to model more complicated combinational logic where the always statement is helpful
- Do not make assignments to the same signal in more than one always statement or continuous assig...

## ISA

## vs. Microarchitecture

- Instructions: opcodes, addressing modes, data types, instruction type and format, registers, condition codes
  - Memory: address space, alignment, addressability, 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-exception support
  - Vector instruction support
  - CPU endianness
  - Virtual page size
- size of page, protection, memory manager, mapping to physical memory, memory layout

### Performance Evaluation

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

### 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:

- Data path - relay and transform data
- Control logic - FSM that determines control signals
- + slowest stage determines cycle time

### Dataflow

In a dataflow machine, a program 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

- Pipelining
- In-order vs. Out-of-Order exec.
- Memory address 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
- size of Register Buffer in an Out-of-Order CPU

### 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
- do something else (fine-grained multithreading)
- ↳ W  $\rightarrow$  D: internal/register file forwarding
- ↳ M  $\rightarrow$  E: operand forwarding

### 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, update its value and set valid bit
  - reclaim rename tag  $\rightarrow$  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
- lock step causes stalls

### Superscalar Execution

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

- + higher IPC
- higher complexity for dependency checking  $\Rightarrow$  more hardware

### Systolic Arrays

Instead of a single processing element (PE) we have a array of PE and carefully orchestrate the data-flow between them  $\Rightarrow$  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.

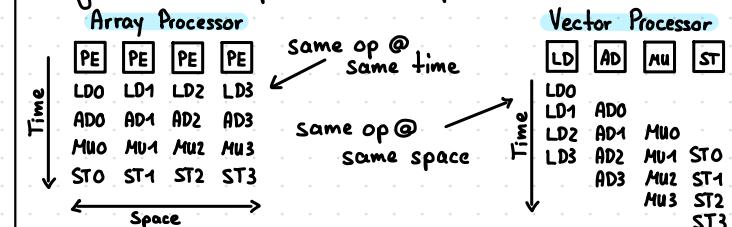
### Fine Grained Multithreading

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

$$\text{Execution time} = (\# \text{instructions}) \left( \frac{\text{cycles}}{\text{instructions}} \right) \left( \frac{\text{seconds}}{\text{cycle}} \right) \quad (\text{CPI} = \text{cycles per instr.})$$

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

$$\text{Average Memory Access Time} = t_{\text{cache}} + \text{MR cache} (t_{\text{main}} + \text{MR main})$$

$t_{\text{cache}}$  = Main Memory,  $t_{\text{main}}$  = Virtual Memory, MR = Miss Rate

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  $\rightarrow$  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 too long it can be split into multiple vectors (strip mining).

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

## 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 address after a branch. If the prediction is wrong, we have to flush the pipeline (misprediction penalty).

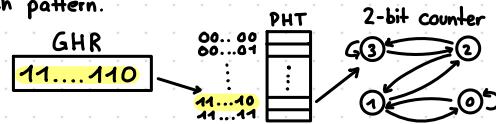
## Prediction Direction Schemes

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

## Global Branch Correlation

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

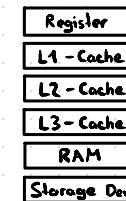
- First level: Global branch history register, keeps track of the last branch outcomes.
- Second level: Pattern history table, keeps a 2-bit counter for each pattern.



## 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 address 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 = Welches 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  $n$

$$\begin{aligned} \# \text{blocks: } B &= C / b \\ \# \text{sets: } S &= B / n \end{aligned}$$

## 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.

Writethrough: write to all levels immediately, simpler but bandwidth intensive.

## Classification of Misses

Compulsory Miss: first reference 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

$$\text{bits of storage} = \text{associativity} \cdot (\text{tag} + \text{dirty} + \text{valid}) + \text{LRU}$$

## 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 cache misses instead of stalling. Therefore it can detect potential cache misses earlier.

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

## Virtual Memory

Much larger than physical memory. Virtual address space is divided in pages, while physical address space is divided into frames. Page Table stores mapping: Virtual → Physical together with a valid bit (and more meta data).

$$\# \text{Virtual Pages} = \frac{\text{Virtual Address}}{\text{Page Size}}$$

$$\# \text{Physical Pages} = \frac{\text{Physical Address}}{\text{Page Size}}$$

$$\text{VA: } \frac{\text{Virtual Page Number}}{\text{Page Offset}}$$

$$\text{PA: } \frac{\text{Page Table}}{\text{P. Page Number}}$$

same a physical tag

$$\text{Physical Address Space} = \frac{\text{PPN}}{\text{Page Offset}}$$

Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Index/Tag	Virtual Page Number

Multi-level page tables: keeps PT size small

Memory protection: different PT for each program

Translation Lookaside Buffer TLB: cache PT entries to speed up address translation

$$\text{Cache capacity} = (\text{Block Size in Bytes}) \cdot (\text{Blocks per Set}) \cdot (\text{Number of Sets})$$

$$\text{Index Bits} = \log_2(\text{Blocks per Set})$$

$$\text{Block Offset Bits} = \log_2(\text{Block Size in Bytes})$$

$$\text{Tag Bits} = (\text{Address Bits}) - (\text{Index Bits}) - (\text{Block Offset Bits})$$

Software Interlocking: NOPs

Hardware Interlocking: Detects dependencies and stalls pipeline accordingly

## 1 Way

0, 1, k, k...

1, k, k, k...

2, 1, 1, 1, k...

3, 1, 1, 1, k...

4, 1, 1, 1, k...

5, 1, 1, 1, k...

6, 1, 1, 1, k...

7, 1, 1, 1, k...

8, 1, 1, 1, k...

9, 1, 1, 1, k...

## 2-way

0, 1, k, k...

1, k, k, k...

2, 1, 1, 1, k...

3, 1, 1, 1, k...

4, 1, 1, 1, k...

5, 1, 1, 1, k...

6, 1, 1, 1, k...

7, 1, 1, 1, k...

8, 1, 1, 1, k...

9, 1, 1, 1, k...

## 4-way

0, 1, k, k...

1, k, k, k...

2, 1, 1, 1, k...

3, 1, 1, 1, k...

4, 1, 1, 1, k...

5, 1, 1, 1, k...

6, 1, 1, 1, k...

7, 1, 1, 1, k...

8, 1, 1, 1, k...

9, 1, 1, 1, k...

following the same path (except if the thread with the same path is in the 1st way as well, then the thread in the 2nd way is the winner and the thread in the 3rd way is the loser)

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

$$\begin{aligned} F &= B + A\bar{A} + A\bar{B} + A\bar{C} + \bar{C}\bar{A} + \bar{C}\bar{B} + \bar{C} \\ &= B + A + \bar{C} \end{aligned}$$

Ex. Simplify the following min-terms:  $\sum(3, 5, 7, 11, 13, 15)$ .  $\{3, 5, 7, 11, 13, 15\} = \{0011, 0101, 0111, 1011, 1101, 1111\}$

$$\begin{aligned} F &= \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}CD + \bar{A}BC\bar{D} + ABCD \\ &= CD \cdot (\bar{A}\bar{B} + \bar{A}\bar{B} + \bar{A}\bar{B} + AB) + BD \cdot (\bar{A}\bar{C} + A\bar{C}) \\ &= CD + BD\bar{C} \\ &= D(B + C) \end{aligned}$$

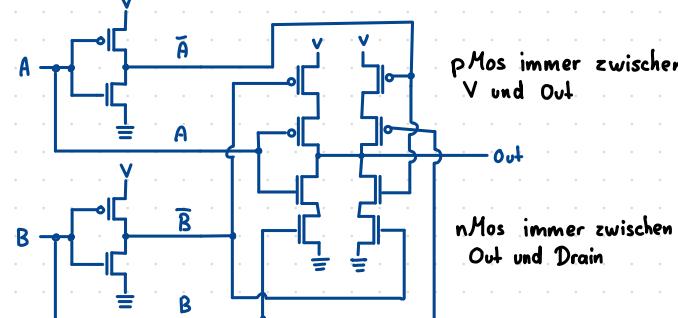
Ex. Convert the following equation to only contain NANDs.

$$\begin{aligned} F &= (\bar{A}\bar{B} + C) + AC = (\bar{A}\bar{B} + C) \cdot \bar{AC} = (\bar{A}\bar{B} \cdot \bar{C}) \cdot \bar{AC} \\ &= AB \cdot \bar{AC} = \overline{\overline{AB} \cdot \overline{AC}} \end{aligned}$$

Ex. Convert the following equation to only contain NANDs.

$$\begin{aligned} F &= (\bar{A} + BC) + \bar{C} = (\bar{A} + BC) \cdot \bar{C} = (\bar{A} + BC) \cdot C \\ &= (\overline{A \cdot BC}) \cdot C = (\overline{A} \cdot \overline{B} \cdot \overline{C}) \cdot C \end{aligned}$$

Ex. Draw a XOR-Gate with transistors.



Ex. Sequential or Combinational circuit?

```
module one (input clk, input o, 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 @(*)
    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 @(*)
    if (!reset) q <= 0;
    else q <= d;
  endmodule
```

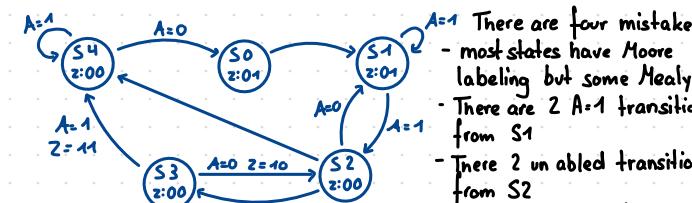
The code implements 2 D-FlipFlops, each works with a asynchronous active low reset.

Ex. Is this code syntactically correct?

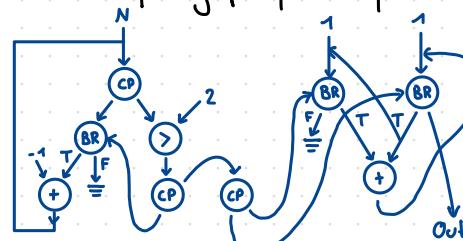
```
module fulladd (input a, b, c, output reg s, c.out);
  assign s = a ^ b;
  assign c.out = (a & b) | (a & c) | (b & c); we use assign therefore these have to be wires
endmodule
```

```
module top (input wire [5:0] instr, input wire op, output z);
  reg [1:0] r1, r2; should be wires
  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; multiple drivers
endmodule
```

Ex. List all the mistakes in this diagram.



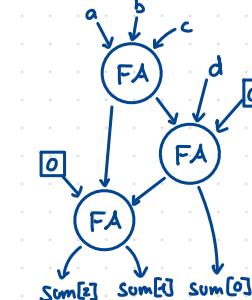
Ex. Draw a data flow graph for the fibonacci function.



Ex. Which designs are compatible with each other?

superscalar - in-order precise exceptions - out-of-order retirement  
superscalar - out-of-order branch prediction - fine-grained multithreading  
single cycle - branch prediction fine-grained multithreading - pipelining  
reservation station - microprogramming Tomasulo's algorithm - in-order  
fine-grained multithreading - single core 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 programs 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 subtraction (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  
 000 with 64 reservation stages, branch resolving during 2nd cycle of branch execution and 16 stages before 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.

LOOP1:              Initial R1    #Cycles

SUB R1, R1, #1	4	51
BGT R1, LOOP1	8	63
	16	87

LOOP2:              B LOOP2    all runs execute the same #dynamic instr.

Let:  $C = \# \text{cycles}$

$P = \# \text{stages}$

$I = \# \text{dynamic instr.}$

$B = \# \text{branch instr.}$

$D = \# \text{cycles stall / branch} \Rightarrow P + I = 40, D = 3$

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

$$51 = P + I - 1 + 4D$$

$$63 = P + I - 1 + 8D$$

$$87 = P + I - 1 + 16D$$

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?

```

MOV R0<-1000 FDE1E2E3E4W
LD R1<[R0] FD - - E1E2E3E4E5E6E7E8W
BL R1,100,LB1 FD - - - - E1E2E3E4W
MUL R1<R1,5 FD E1E2E3 // killed
ST [R0]< R1 FDE1E2E3E4W
ADD R1<R1,R0 FD - - - E1W
ST [R0]< R1

```

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 Tomasulo's Algorithm with: 8 functional units with their own tag/data bus, 32x64 bit registers, 16 reservation stations per functional unit and 2 source register per reservation station, calculate:

$$\# \text{tag comparator / reservation station entry} = 2 \times 8 = 16$$

$$\# \text{tag comparators} = 16 \times 16 \times 8 + 32 \times 8 = 2304$$

$$\min. \text{tag size} = \log(16 \times 8) = 7$$

$$\min. \text{size of register alias table} = 32 \times (7 + 64 + 1) = 2304$$

$$\min. \text{total size of tag store} = 32 \times 7 + 8 \times 16 \times 2 \times 7 = 2048$$

Ex. Comparing a VLIW and an 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 re-order 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 L1-cache hit rate and higher fetch bandwidth.

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 is the next power of 2.

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

VLD	V1, A	50	L-1	111 = 51 + 4 + 16 + 1 + L-1
VLD	V2, B	50	L-1	=> L = 40
VADD	V3, V1, V2	4	L-1	
VMUL	V4, V1, V3	16	L-1	
VRSHFA	V5, V4, 2	16	L-1	

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

VLD	[0]	50		
	[31]	50		
	[30]	7	50	1 + 50 + 7 + 50 + 4 + 16 + 1 = 129
	[29]	7	50	=> 129 cycles
VLD	[0]	50		
	[31]	50		
	[30]	7	50	
	[29]	7	50	
VADD		4		
VMUL		16		
VRSHFA		16		

tracking the last element

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

$$279 = 1 + 16 + 4 + 1 + 7 + \lceil \frac{1}{4} \rceil \cdot 50$$

$$\Rightarrow 5 = \lceil \frac{1}{4} \rceil \cdot 7 \Rightarrow x = 8 \text{ memory banks}$$

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?

$$X = \{1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\}$$

$$Y = \{1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\}$$

$$Z = \{0\ 1\ 0\ 0\ 0\ 0\ 0\}$$

$$X' = \{1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\}$$

$$Y' = \{1\ 1\ 0\ 0\ 1\ 0\ 0\ 1\}$$

$$Z' = \{0\ 0\ 0\ 0\ 0\ 0\ 0\}$$

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

Ex. How effective is a 16KB, 4-way associative 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 its 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. 32368 0 123 1024 3072 8192 1	hit	1/3

Cache block size: 8, 16, 32, 64 or 128B

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.

