

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

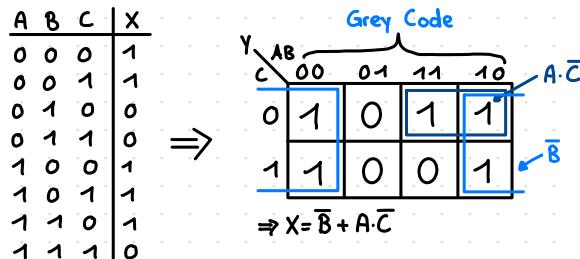
$X = (\bar{A} + B) \cdot (\bar{A} \cdot \bar{B})$ $X = (\bar{A} \cdot \bar{B}) + (A \cdot B)$

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



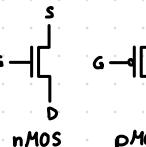
Logic Gates

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

- 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), 0000, 0001, ..., 1111

• 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 unmarked transitions
- initial state (if no reset)
- no mix of Moore/Mealy labeling
- no missing transitions

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 δ (posedge clk) and non-blocking ($=$) to model synchroners 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

dirty bit (1) is associated with each cache block. Write-back

- 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
- # of logic arithmetic and logic units (ALUs)
- size of Reorder 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)
 - $\hookrightarrow W \rightarrow D$: internal/register file forwarding
 - $\hookrightarrow M \rightarrow E$: 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.

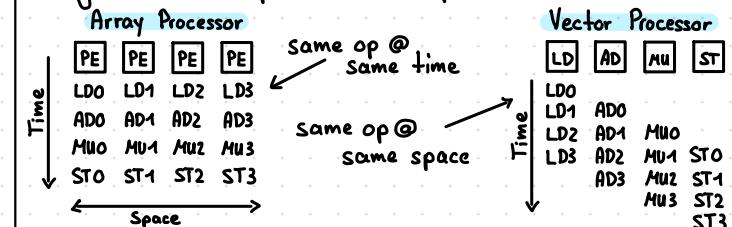
Fine Grained Multithreading

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
- \hookrightarrow 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_{cache} = Main Memory, t_{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

- 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

Cycles

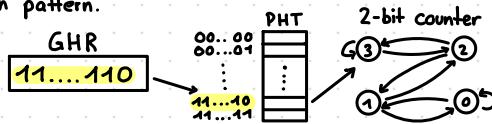
Total number of cycles can be expressed as $C = P(I - 1) + B + D$

- C = total number of cycles taken
 - P = total number of pipeline stages
 - I = total number of instructions
 - B = total number of conditional branch instructions executed
 - D = number of cycles stalled for each branch of branch

Global Branch Correlation

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

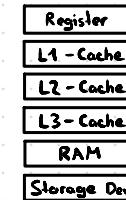
- First level: Global branch history register, keeps track of the last branch outcomes.
- Second level: Pattern history table, keeps a 2-bit counter for each pattern.



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
- #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.

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

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

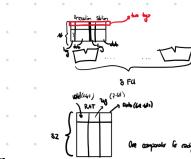
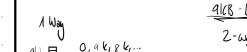
$$\text{Index Bits} = \log_2(\text{Blocks per Set})$$

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

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

Software Interlocking: NOPs

Hardware Interlocking: Detects dependencies and stalls pipeline accordingly



Utilization: $(\text{Threads} - \text{Instructions each thread is taking}) / (\text{Threads} - \text{Total Instructions taken at any point in time})$

Ways: $(\text{Total Threads}) / (\text{Threads per way})$

Min Utilization: following defined paths when in general as many parallel like the shared path, but for each other paths parallel we use 1 thread if there is a cap then the inner threads those we just not send to further threads + 0 (but needs to be in the memory)

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: $\sum(3, 5, 7, 11, 13, 15)$. $\{3, 5, 7, 11, 13, 15\} = \{0011, 0101, 0111, 1011, 1101, 1111\}$

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

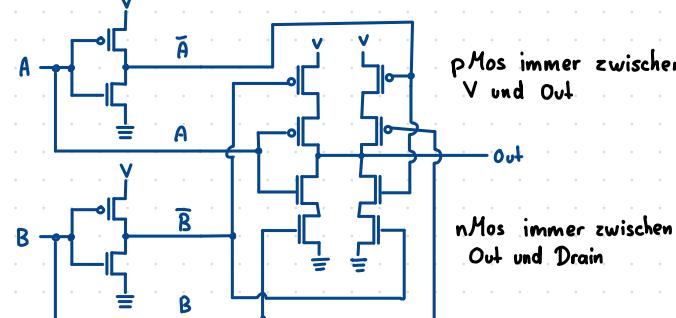
Ex. Convert the following equation to only contain NANDs.

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

Ex. Convert the following equation to only contain NANDs.

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

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
```

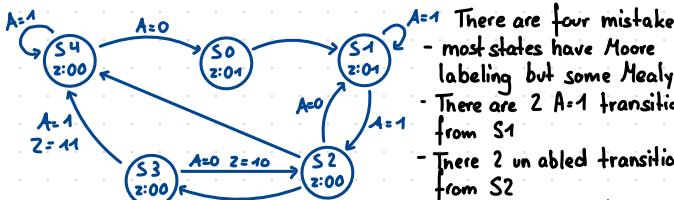
The code implements 2 D-FlipFlops, each works with a asynchronous active low reset.

Ex. Is this code syntactically correct?

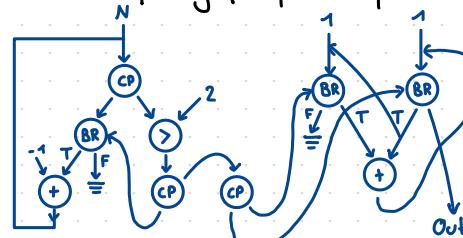
```
module fulladd (input a, b, c, output reg s, c_out);
  assign s = a ^ b;
  assign c_out = (a & b) | (a & c) | (b & c); we use assign therefore these have to be wires
endmodule
```

```
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.



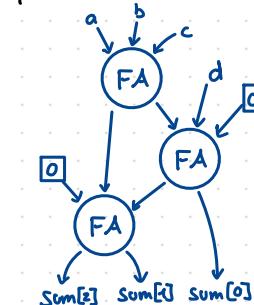
Ex. Draw a data flow graph for the fibonacci function.



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

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 $\Rightarrow P+I=40, D=3$

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

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

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

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

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:

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

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

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

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

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

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	14	L-1	

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

VLD	[0]	50		
	[31]	50		1 + 50 + 7 + 50 + 4 + 16 + 1 = 129
	[30]	7	50	=> 129 cycles
VLD	[0]	11	50	
	[31]	50		
	[30]	7	50	
VADD		4		
VMUL		16		
VRSHFA		14		

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 $\frac{1}{4}$ 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 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 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.

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

ABAHBGHHAEHDHGCGCABHDECCBADEF

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.

ABAHBGHHAEHDHGCGCABHDECCBADEF

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 address space is 1MB, page size 2KB and block size 8B. What is the size of the data store?

$$\text{Tag store} = 2^9 \cdot (2^4 + 5) = 15 \times 2^9 \Rightarrow i = 9 \quad t = 5$$

$$\text{Data store} = 2^9 \times 8 \times 2^3 = 8\text{KB}$$

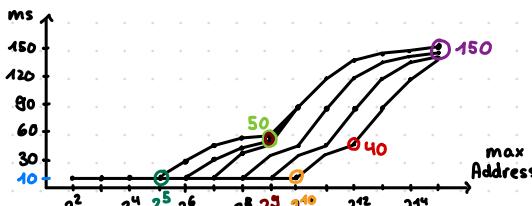
How many bits of the virtual index come from the VPN?

Index	block
9	3

What is the physical address space?

$$\begin{aligned} \text{Page offset} &= 11 \text{ Bits} & \text{Page Tag} &= 5 \text{ Bits} \\ \Rightarrow 2^{11} \cdot 2^5 &= 2^{16} = 64\text{KB} \end{aligned}$$

Ex. Fill in the blanks:



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

Ex. Give a 4-way assoc. write back cache with a $2^{11} \times 89$ bit tag store, a 9 bit replacement policy, 64B blocks. It is virtually indexed, physically tagged 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?

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

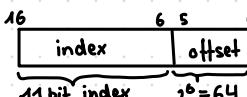
What is the size of the data store?

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

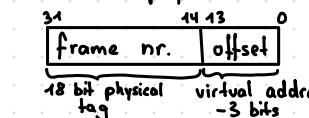
How many bits in the PPN overlap with the index bits in the virtual address?

3, since data can be present in up to 2^{3-8} sets.

Draw the virtual address:



Draw the physical address:



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

What is the virtual address space?

$$2^{\text{VPN}} \cdot \text{Page Offset} = 2^{20} \cdot 2^{14} = 1024 \cdot 2^{14} \text{ from } 1024 \cdot 2$$

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

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]
```

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

B: int[100] a;

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

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

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 $\Pr[B1.T \& B2.T] = 1/2$ the probability for B3 to be taken is $1/3$, resulting in $1/2 \cdot 1/3 = 1/6$ probability to increase and $1/2 \cdot (1-1/3) = 1/3$ to decrease, therefore B3 contributes $1/6 - 1/3 = -1/6$.

B4: $\Pr[B2.T \& B3.T] = 1/6 \Rightarrow 1/6 \cdot 1 = 1/6$

B1: $\Pr[B3.T \& B4.T] = 1/6 \Rightarrow 1/6 \cdot 1 = 1/6$

B2: $\Pr[B4.T \& B1.T] = 1/6 \Rightarrow 1/6 \cdot 1/2 = 1/12 = 0$

Resulting in a total of $1/6$ per iteration. Therefore after 120 iterations, the expected value is 20.

Fibonacci Code in Assembly

```
int fibfib(n){  
    int a=0;  
    int b=1;  
    int c=a+b;  
    while (n>0){  
        c=a+b;  
        a=b;  
        b=c;  
        n--;  
    }  
    return c;  
}
```

```
;b:  
add $sp, -16 //alloc stack space  
dec $sp //leave R16  
add $sp, 16 //alloc for arg  
add $sp, 16 //move R16  
add $sp, 16 //move R17  
add $sp, 16 //move R18  
add $sp, 16 //move R19  
add $sp, 16 //move R20  
add $sp, 16 //move R21  
add $sp, 16 //move R22  
add $sp, 16 //move R23  
add $sp, 16 //move R24  
add $sp, 16 //move R25  
add $sp, 16 //move R26  
add $sp, 16 //move R27  
add $sp, 16 //move R28  
add $sp, 16 //move R29  
add $sp, 16 //move R30  
add $sp, 16 //move R31  
add $sp, 16 //move R32  
add $sp, 16 //move R33  
add $sp, 16 //move R34  
add $sp, 16 //move R35  
add $sp, 16 //move R36  
add $sp, 16 //move R37  
add $sp, 16 //move R38  
add $sp, 16 //move R39  
add $sp, 16 //move R40  
add $sp, 16 //move R41  
add $sp, 16 //move R42  
add $sp, 16 //move R43  
add $sp, 16 //move R44  
add $sp, 16 //move R45  
add $sp, 16 //move R46  
add $sp, 16 //move R47  
add $sp, 16 //move R48  
add $sp, 16 //move R49  
add $sp, 16 //move R50  
add $sp, 16 //move R51  
add $sp, 16 //move R52  
add $sp, 16 //move R53  
add $sp, 16 //move R54  
add $sp, 16 //move R55  
add $sp, 16 //move R56  
add $sp, 16 //move R57  
add $sp, 16 //move R58  
add $sp, 16 //move R59  
add $sp, 16 //move R60  
add $sp, 16 //move R61  
add $sp, 16 //move R62  
add $sp, 16 //move R63  
add $sp, 16 //move R64  
add $sp, 16 //move R65  
add $sp, 16 //move R66  
add $sp, 16 //move R67  
add $sp, 16 //move R68  
add $sp, 16 //move R69  
add $sp, 16 //move R70  
add $sp, 16 //move R71  
add $sp, 16 //move R72  
add $sp, 16 //move R73  
add $sp, 16 //move R74  
add $sp, 16 //move R75  
add $sp, 16 //move R76  
add $sp, 16 //move R77  
add $sp, 16 //move R78  
add $sp, 16 //move R79  
add $sp, 16 //move R80  
add $sp, 16 //move R81  
add $sp, 16 //move R82  
add $sp, 16 //move R83  
add $sp, 16 //move R84  
add $sp, 16 //move R85  
add $sp, 16 //move R86  
add $sp, 16 //move R87  
add $sp, 16 //move R88  
add $sp, 16 //move R89  
add $sp, 16 //move R90  
add $sp, 16 //move R91  
add $sp, 16 //move R92  
add $sp, 16 //move R93  
add $sp, 16 //move R94  
add $sp, 16 //move R95  
add $sp, 16 //move R96  
add $sp, 16 //move R97  
add $sp, 16 //move R98  
add $sp, 16 //move R99  
add $sp, 16 //move R100  
add $sp, 16 //move R101  
add $sp, 16 //move R102  
add $sp, 16 //move R103  
add $sp, 16 //move R104  
add $sp, 16 //move R105  
add $sp, 16 //move R106  
add $sp, 16 //move R107  
add $sp, 16 //move R108  
add $sp, 16 //move R109  
add $sp, 16 //move R110  
add $sp, 16 //move R111  
add $sp, 16 //move R112  
add $sp, 16 //move R113  
add $sp, 16 //move R114  
add $sp, 16 //move R115  
add $sp, 16 //move R116  
add $sp, 16 //move R117  
add $sp, 16 //move R118  
add $sp, 16 //move R119  
add $sp, 16 //move R120  
add $sp, 16 //move R121  
add $sp, 16 //move R122  
add $sp, 16 //move R123  
add $sp, 16 //move R124  
add $sp, 16 //move R125  
add $sp, 16 //move R126  
add $sp, 16 //move R127  
add $sp, 16 //move R128  
add $sp, 16 //move R129  
add $sp, 16 //move R130  
add $sp, 16 //move R131  
add $sp, 16 //move R132  
add $sp, 16 //move R133  
add $sp, 16 //move R134  
add $sp, 16 //move R135  
add $sp, 16 //move R136  
add $sp, 16 //move R137  
add $sp, 16 //move R138  
add $sp, 16 //move R139  
add $sp, 16 //move R140  
add $sp, 16 //move R141  
add $sp, 16 //move R142  
add $sp, 16 //move R143  
add $sp, 16 //move R144  
add $sp, 16 //move R145  
add $sp, 16 //move R146  
add $sp, 16 //move R147  
add $sp, 16 //move R148  
add $sp, 16 //move R149  
add $sp, 16 //move R150  
add $sp, 16 //move R151  
add $sp, 16 //move R152  
add $sp, 16 //move R153  
add $sp, 16 //move R154  
add $sp, 16 //move R155  
add $sp, 16 //move R156  
add $sp, 16 //move R157  
add $sp, 16 //move R158  
add $sp, 16 //move R159  
add $sp, 16 //move R160  
add $sp, 16 //move R161  
add $sp, 16 //move R162  
add $sp, 16 //move R163  
add $sp, 16 //move R164  
add $sp, 16 //move R165  
add $sp, 16 //move R166  
add $sp, 16 //move R167  
add $sp, 16 //move R168  
add $sp, 16 //move R169  
add $sp, 16 //move R170  
add $sp, 16 //move R171  
add $sp, 16 //move R172  
add $sp, 16 //move R173  
add $sp, 16 //move R174  
add $sp, 16 //move R175  
add $sp, 16 //move R176  
add $sp, 16 //move R177  
add $sp, 16 //move R178  
add $sp, 16 //move R179  
add $sp, 16 //move R180  
add $sp, 16 //move R181  
add $sp, 16 //move R182  
add $sp, 16 //move R183  
add $sp, 16 //move R184  
add $sp, 16 //move R185  
add $sp, 16 //move R186  
add $sp, 16 //move R187  
add $sp, 16 //move R188  
add $sp, 16 //move R189  
add $sp, 16 //move R190  
add $sp, 16 //move R191  
add $sp, 16 //move R192  
add $sp, 16 //move R193  
add $sp, 16 //move R194  
add $sp, 16 //move R195  
add $sp, 16 //move R196  
add $sp, 16 //move R197  
add $sp, 16 //move R198  
add $sp, 16 //move R199  
add $sp, 16 //move R200  
add $sp, 16 //move R201  
add $sp, 16 //move R202  
add $sp, 16 //move R203  
add $sp, 16 //move R204  
add $sp, 16 //move R205  
add $sp, 16 //move R206  
add $sp, 16 //move R207  
add $sp, 16 //move R208  
add $sp, 16 //move R209  
add $sp, 16 //move R210  
add $sp, 16 //move R211  
add $sp, 16 //move R212  
add $sp, 16 //move R213  
add $sp, 16 //move R214  
add $sp, 16 //move R215  
add $sp, 16 //move R216  
add $sp, 16 //move R217  
add $sp, 16 //move R218  
add $sp, 16 //move R219  
add $sp, 16 //move R220  
add $sp, 16 //move R221  
add $sp, 16 //move R222  
add $sp, 16 //move R223  
add $sp, 16 //move R224  
add $sp, 16 //move R225  
add $sp, 16 //move R226  
add $sp, 16 //move R227  
add $sp, 16 //move R228  
add $sp, 16 //move R229  
add $sp, 16 //move R230  
add $sp, 16 //move R231  
add $sp, 16 //move R232  
add $sp, 16 //move R233  
add $sp, 16 //move R234  
add $sp, 16 //move R235  
add $sp, 16 //move R236  
add $sp, 16 //move R237  
add $sp, 16 //move R238  
add $sp, 16 //move R239  
add $sp, 16 //move R240  
add $sp, 16 //move R241  
add $sp, 16 //move R242  
add $sp, 16 //move R243  
add $sp, 16 //move R244  
add $sp, 16 //move R245  
add $sp, 16 //move R246  
add $sp, 16 //move R247  
add $sp, 16 //move R248  
add $sp, 16 //move R249  
add $sp, 16 //move R250  
add $sp, 16 //move R251  
add $sp, 16 //move R252  
add $sp, 16 //move R253  
add $sp, 16 //move R254  
add $sp, 16 //move R255  
add $sp, 16 //move R256  
add $sp, 16 //move R257  
add $sp, 16 //move R258  
add $sp, 16 //move R259  
add $sp, 16 //move R260  
add $sp, 16 //move R261  
add $sp, 16 //move R262  
add $sp, 16 //move R263  
add $sp, 16 //move R264  
add $sp, 16 //move R265  
add $sp, 16 //move R266  
add $sp, 16 //move R267  
add $sp, 16 //move R268  
add $sp, 16 //move R269  
add $sp, 16 //move R270  
add $sp, 16 //move R271  
add $sp, 16 //move R272  
add $sp, 16 //move R273  
add $sp, 16 //move R274  
add $sp, 16 //move R275  
add $sp, 16 //move R276  
add $sp, 16 //move R277  
add $sp, 16 //move R278  
add $sp, 16 //move R279  
add $sp, 16 //move R280  
add $sp, 16 //move R281  
add $sp, 16 //move R282  
add $sp, 16 //move R283  
add $sp, 16 //move R284  
add $sp, 16 //move R285  
add $sp, 16 //move R286  
add $sp, 16 //move R287  
add $sp, 16 //move R288  
add $sp, 16 //move R289  
add $sp, 16 //move R290  
add $sp, 16 //move R291  
add $sp, 16 //move R292  
add $sp, 16 //move R293  
add $sp, 16 //move R294  
add $sp, 16 //move R295  
add $sp, 16 //move R296  
add $sp, 16 //move R297  
add $sp, 16 //move R298  
add $sp, 16 //move R299  
add $sp, 16 //move R300  
add $sp, 16 //move R301  
add $sp, 16 //move R302  
add $sp, 16 //move R303  
add $sp, 16 //move R304  
add $sp, 16 //move R305  
add $sp, 16 //move R306  
add $sp, 16 //move R307  
add $sp, 16 //move R308  
add $sp, 16 //move R309  
add $sp, 16 //move R310  
add $sp, 16 //move R311  
add $sp, 16 //move R312  
add $sp, 16 //move R313  
add $sp, 16 //move R314  
add $sp, 16 //move R315  
add $sp, 16 //move R316  
add $sp, 16 //move R317  
add $sp, 16 //move R318  
add $sp, 16 //move R319  
add $sp, 16 //move R320  
add $sp, 16 //move R321  
add $sp, 16 //move R322  
add $sp, 16 //move R323  
add $sp, 16 //move R324  
add $sp, 16 //move R325  
add $sp, 16 //move R326  
add $sp, 16 //move R327  
add $sp, 16 //move R328  
add $sp, 16 //move R329  
add $sp, 16 //move R330  
add $sp, 16 //move R331  
add $sp, 16 //move R332  
add $sp, 16 //move R333  
add $sp, 16 //move R334  
add $sp, 16 //move R335  
add $sp, 16 //move R336  
add $sp, 16 //move R337  
add $sp, 16 //move R338  
add $sp, 16 //move R339  
add $sp, 16 //move R340  
add $sp, 16 //move R341  
add $sp, 16 //move R342  
add $sp, 16 //move R343  
add $sp, 16 //move R344  
add $sp, 16 //move R345  
add $sp, 16 //move R346  
add $sp, 16 //move R347  
add $sp, 16 //move R348  
add $sp, 16 //move R349  
add $sp, 16 //move R350  
add $sp, 16 //move R351  
add $sp, 16 //move R352  
add $sp, 16 //move R353  
add $sp, 16 //move R354  
add $sp, 16 //move R355  
add $sp, 16 //move R356  
add $sp, 16 //move R357  
add $sp, 16 //move R358  
add $sp, 16 //move R359  
add $sp, 16 //move R360  
add $sp, 16 //move R361  
add $sp, 16 //move R362  
add $sp, 16 //move R363  
add $sp, 16 //move R364  
add $sp, 16 //move R365  
add $sp, 16 //move R366  
add $sp, 16 //move R367  
add $sp, 16 //move R368  
add $sp, 16 //move R369  
add $sp, 16 //move R370  
add $sp, 16 //move R371  
add $sp, 16 //move R372  
add $sp, 16 //move R373  
add $sp, 16 //move R374  
add $sp, 16 //move R375  
add $sp, 16 //move R376  
add $sp, 16 //move R377  
add $sp, 16 //move R378  
add $sp, 16 //move R379  
add $sp, 16 //move R380  
add $sp, 16 //move R381  
add $sp, 16 //move R382  
add $sp, 16 //move R383  
add $sp, 16 //move R384  
add $sp, 16 //move R385  
add $sp, 16 //move R386  
add $sp, 16 //move R387  
add $sp, 16 //move R388  
add $sp, 16 //move R389  
add $sp, 16 //move R390  
add $sp, 16 //move R391  
add $sp, 16 //move R392  
add $sp, 16 //move R393  
add $sp, 16 //move R394  
add $sp, 16 //move R395  
add $sp, 16 //move R396  
add $sp, 16 //move R397  
add $sp, 16 //move R398  
add $sp, 16 //move R399  
add $sp, 16 //move R400  
add $sp, 16 //move R401  
add $sp, 16 //move R402  
add $sp, 16 //move R403  
add $sp, 16 //move R404  
add $sp, 16 //move R405  
add $sp, 16 //move R406  
add $sp, 16 //move R407  
add $sp, 16 //move R408  
add $sp, 16 //move R409  
add $sp, 16 //move R410  
add $sp, 16 //move R411  
add $sp, 16 //move R412  
add $sp, 16 //move R413  
add $sp, 16 //move R414  
add $sp, 16 //move R415  
add $sp, 16 //move R416  
add $sp, 16 //move R417  
add $sp, 16 //move R418  
add $sp, 16 //move R419  
add $sp, 16 //move R420  
add $sp, 16 //move R421  
add $sp, 16 //move R422  
add $sp, 16 //move R423  
add $sp, 16 //move R424  
add $sp, 16 //move R425  
add $sp, 16 //move R426  
add $sp, 16 //move R427  
add $sp, 16 //move R428  
add $sp, 16 //move R429  
add $sp, 16 //move R430  
add $sp, 16 //move R431  
add $sp, 16 //move R432  
add $sp, 16 //move R433  
add $sp, 16 //move R434  
add $sp, 16 //move R435  
add $sp, 16 //move R436  
add $sp, 16 //move R437  
add $sp, 16 //move R438  
add $sp, 16 //move R439  
add $sp, 16 //move R440  
add $sp, 16 //move R441  
add $sp, 16 //move R442  
add $sp, 16 //move R443  
add $sp, 16 //move R444  
add $sp, 16 //move R445  
add $sp, 16 //move R446  
add $sp, 16 //move R447  
add $sp, 16 //move R448  
add $sp, 16 //move R449  
add $sp, 16 //move R450  
add $sp, 16 //move R451  
add $sp, 16 //move R452  
add $sp, 16 //move R453  
add $sp, 16 //move R454  
add $sp, 16 //move R455  
add $sp, 16 //move R456  
add $sp, 16 //move R457  
add $sp, 16 //move R458  
add $sp, 16 //move R459  
add $sp, 16 //move R460  
add $sp, 16 //move R461  
add $sp, 16 //move R462  
add $sp, 16 //move R463  
add $sp, 16 //move R464  
add $sp, 16 //move R465  
add $sp, 16 //move R466  
add $sp, 16 //move R467  
add $sp, 16 //move R468  
add $sp, 16 //move R469  
add $sp, 16 //move R470  
add $sp, 16 //move R471  
add $sp, 16 //move R472  
add $sp, 16 //move R473  
add $sp, 16 //move R474  
add $sp, 16 //move R475  
add $sp, 16 //move R476  
add $sp, 16 //move R477  
add $sp, 16 //move R478  
add $sp, 16 //move R479  
add $sp, 16 //move R480  
add $sp, 16 //move R481  
add $sp, 16 //move R482  
add $sp, 16 //move R483  
add $sp, 16 //move R484  
add $sp, 16 //move R485  
add $sp, 16 //move R486  
add $sp, 16 //move R487  
add $sp, 16 //move R488  
add $sp, 16 //move R489  
add $sp, 16 //move R490  
add $sp, 16 //move R491  
add $sp, 16 //move R492  
add $sp, 16 //move R493  
add $sp, 16 //move R494  
add $sp, 16 //move R495  
add $sp, 16 //move R496  
add $sp, 16 //move R497  
add $sp, 16 //move R498  
add $sp, 16 //
```