Lecture 01 (Introduction)

1 What is Course About?

1. Designing HPC's processors
2. We begin with COL216 review
3. Exercise climbing to 6th floor (+1 if you come from basement)
4. OOO pipelining
5. Background needed is only C programming

2 Evaluation

1. Assignments - 10 (individual) + 15 (pairs) + 20 (pairs)
2. Minor - 25
3. Major - 30
4. Absolute like grading

Lecture 02 (In Order Pipelines)

1 5 Stage Pipeline (recap)

1. Instruction fetch - IF
2. Instruction decode (+ operand fetch) - OF
3. Execute stage - EX
4. Memory access - MA
5. Register write-back - RW

This is part of GATE syllabus and we are *GATEd*

There are back connections from:

1. EX to IF - for branching
2. RW to OF - for write back

2 In-Order Pipelining

1. Multiple instructions were at different places in the pipeline
2. This leads to issues:
   i. Structural hazards
   ii. Instruction dependency
      - insert nop aka bubbles
      - forward the value using multiplexers
      - forwarding paths:
        1. RW $\rightarrow$ MA
        2. RW $\rightarrow$ EX
        3. RW $\rightarrow$ OF
        4. MA $\rightarrow$ EX
      - EPIC project tried to move this to compiler stage but didn't go anywhere
      - Load use hazard leads to impossibility when EX needs MA load
   iii. Control hazard when branching
      - predict
      - execute branch independent instructions (compiler level optimisation) - delayed branch
      - need to balance between compiler and hardware optimisations

(kids were running + trekking on stairs and not losing breath even when talking)

Lecture 03 (Out of Order Pipelines)

1 Performance Considerations

1. Performance cannot be gauged with clock speed or RAM
2. Fasteness is comparable only when given a program

1.1 Performance Equation

$$\frac{programs}{time} = \frac{programs}{instructions} \cdot \frac{instructions}{cycles} \cdot \frac{cycles}{time} = \frac{IPC \times frequency}{instruction}$$

(assumes just 1 program)

1.1.1 Instructions

Depends on compiler

1.1.2 Frequency

1. Depends on transistor technology
2. If there are more (or smaller) pipeline stages, then frequency is higher
3. $\frac{1}{frequency} = time_{cycle} = \frac{T}{k} + L$ (k pipelines, L = latch delay)
4. Adding too many stages will increase forwarding logic, thus being counter productive after a point
5. $P \propto f^3 \implies \Delta T \propto P$ - thus pipeline stages which had gone to 27 came down to somewhere between 12-15

1.1.3 IPC

1. Depends on architecture and compiler
2. Will be primary focus of the book (course as well?)
3. Ideal IPC of in-order pipeline is 1, typically < 1

1.1.3.1 CPI

$$CPI = CPI_{ideal} + \text{stall\_rate} \times \text{stall\_penalty}$$

1. Stall rate will remain constant with pipeline stages (empirical)

2. Stall penalty increases with more pipeline stages

1.1.3.2 Increasing IPC

1. Issue more than one instructions per cycle
2. Make it a superscalar processor (it is hard to do this)
    i. very complex stall and forwarding paths
    ii. of the order of $O(n^2)$ for n-issue processor
    iii. if there are branches, then our party is over

2 Out of Order Pipelining

1. Much better solution
2. Consumer is executed after producer

2.1 Basic Principle

1. Create a pool if instructions
2. Find instructions that are mutually independent and have all their operands ready
3. Execute them OOO

ILP: Instruction Level Parallelism - number of ready and independent instructions we can simultaneously execute

2.2 Understanding OOO

1. Construct a dependency graph
2. Make directed edges from consumer to producer
3. Execute all nodes with no outgoing edges
4. Larger graph is always better so that all processor units are active

2.3 Issues

1. Need to figure out instruction window size
    - typical window size is 64 to 128
2. How do we handle branching and ensure that each instruction is on the correct branch
    - need a very accurate branch predictor
    - typically 1 in 5 instructions is a branch
    - need to predict the following:
        i. if it is a branch
        ii. if it is taken or not
        iii. where is the target of the branch

2.4 Math of Branch Prediction

Number of instructions: n Number of branches: $n/5$ Probability of incorrect prediction: p Probability of at least one mistake: $P_n = 1 - (1-p)^{n/5}$

For $n = 100$, if P_n has to be at most 10, p has to be < 0.5

Highest accuracy in A1 is observed to be 97, very poor :(

Lecture 04 (Identifying and Solving Issues with OOO)

1 Nature of Dependence

1. Program Order Dependence
    - instructions dependent because of program order (kinda irrelevant)
2. Read after Write (RAW)
3. Write after Write (WAW)
4. Write after Read (WAR)
5. Control dependency

OOO processors respect only data and control dependency

1.1 Claims

1. Some dependencies are anti-dependencies (WAW, WAR)
2. They are present only because we have finite registers

1.2 Solution

$$r_x \to p_{xa}$$
$$a = avatar$$

hardware is very simple, we only have logic gates, registers and wires :)

2 Issues with Write-Back

1. These updates should appear to happen in-order
2. Interrupt or exception handler will see incorrect state otherwise - **precise exceptions**
3. Treated the same as branch failures

3 Branch Predictor

3.1 Predicting Whether Branch

1. Remember the branch status of PC

2. Store it in a Instruction Status Table (IST)
3. Indexing is done using $PC \mod n$ (n is typically 10)
4. Can have destructive interference (branch-brach or non branch-branch collision)
5. To solve this, store $32 - n$ bits or its subset (helps solve non branch-branch collision)

3.2 Predicting Whether Taken

1. Approach 1 - predict the same as last outcome
2. Approach 2 - use 2 bits to predict (saturating counter)

3.2.0.1 DisCo Stories (context: history decides what level of DisCo to be given)

DisCo levels: - Department level - Dean level - HIGH Alert

History:

1. Sir was warden and his hostel's mess secy provided food and powder - level 2; misdeed repeated - level 3; he has enjoyed sitting in all 3 levels of DisCo
2. Sir was TnP department head, some kids did no work in intern but stole accessories, given level 2 DisCo

DisCo committees are very busy throughout the year - 20-30 DisCo's per year

Lecture 05 (GHR and stuff)

1 Global History Register

Maintain state of the last n branches

2 GAp Predictor

1. Index in the table using n bits of address and k bits of GHR
2. Pattern History Table (PHT) is used to store this information

Petition to make blackboards as smart boards instead of doing Leetcode

2.1 SharkTank Scam

1. Two IITD graduates in the panel were unable to catch a scam
2. Scam was related to "mid-brain activation" magic trick

2.2 Issue with GAp

1. Only stores information about last branch path
2. Need to do it for a region

3 PAp Predictor

1. Use $n1$ middle bits to select k bit GHR
2. Remaining is GAp

3.1 Better Solution

1. Instead of appending k bits for the index, mangle the bits (using xor or a better algo)
2. When using xor, predictor is called Gshare
3. Maintain multiple predictors, choose the better predictor
   i. can run both predictors
   ii. select when required

iii. wastes energy, reduces critical path length

4 Prediction and Compression

1. They are related
2. We cannot predict better than Fano's bound

5 Prediction of Branch Target

1. Use IST as Branch Target Buffer (BTB)
2. Return address can be maintained using a stack - 100% accuracy (Return Address Stack - RAS)

6 Decode Stage

6.1 CISC

1. Issue implementing OOO with CISC
2. Most CISC processors internally convert from CISC to RISC (micro OPs)
3. Some instructions map to large number of micro OPs, use microcode cache
4. Can be solved by pre-decoding the instructions when fetching into i-cache
   - 8 bits
   - start bit
   - end bit
   - functional bit
   - two-ROP
   - three-ROP

6.2 Optimizing Operations on Stack Pointer

1. Store stack pointer during decode stage
2. Directly compute memory address
3. No need to pipeline, can directly load or store
4. Need to nullify this optimisation when we have something like `ld sp 12[r1]`
   - but can store the offsets
   - introduce these offsets in future when value is loaded

6.3 Instruction Compression

1. Reduced-width instructions (thumb ISA)
   - avoid encoding complicated flags and options
   - reduce size of immediate fields

- implicit operands
- to overcome the issue of varied size instructions, have thumb instructions in groups such that total length is 4 (or 8) bytes

Lecture 06 (Instruction Fetch and Dispatch)

1 Instruction Compression

1. Maintain a dictionary for the frequently used sequence of instructions
2. Reduces code size

2 Issue

1. Need to rename registers used by each instruction
2. Different ISAs have different number of physical registers (architectural registers)
3. No separate architectural register file is maintained
4. They map to exactly one physical register at any time

2.1 Rename Stage

1. Register Alias Table (RAT)
2. Free list
3. Rename Table - entry, available bit

2.1.1 RAT

1. Value is updated when write happens to a register
2. 4 instructions are updated in one go
3. Dependency is resolved using a multiplexer

We'll finish the course by minors since "itna samay hai"

3 Dispatch

1. Renamed instructions are sent to instruction window
2. Instruction are made up of
   - valid
   - opcode
   - src tag 1 (imm1)

    - ready bit 1 (comes from available bit of rename table initially)
    - src tag 2 (imm2)
    - ready bit 2
    - dest tag
3. Instructions are chosen based on usability and resources available
4. Ready bit is 1 if register is free or can be forwarded
5. After selecting, the register files are read and the they are sent to execute unit

3.1 Wakeup

1. Once producer finishes executing, it broadcasts the tag of destination physical register
2. Each entry in IW marks its source operand as ready if the tag matches

Reminder: Physical registers are write-once, read multiple times

Lecture 07 (Tejas)

1 What is a pinfile?

There are two types of binary instrumentor:

1. Static
2. Dynamic

1.1 Static

Some dummy functions are added after some “special” instructions to analyse the trace and stuff

1.2 Dynamic

Software interrupt is called after some fixed # of instructions or similar

1.3 Information Collected

1. PC
2. Registers read/writted
3. Memory address
4. Branch taken (or not)

1.4 Working of Tejas

1. Shared memory channel between PIN and Tejas
2. Trace file exists

1.5 Tejas

1. Cycle driven
2. Event driven
3. Basically does architecture simulation
4. Maintains an event queue using a priority queue

2 Indian Industry

1. Sir and another colleague was asked to set a question paper for introductory CS by lab director (IBM)
2. After a few days sir heard angry people in cafeteria blaming people who set 100 questions
3. Bottom 30% was fired :))
4. In a famous Korean company, access card is tracked :)
5. Some companies detect screen saver

Lecture 08 ()

1 Choosing Instructions

1. We have a tree structure to choose the instruction which is “granted” permission to execute out of the instructions which “request” execution
2. The instruction which finally gets chosen is then executed
3. Each node is a “choice box” which stores information about which instruction it chose

1.1 Utilizing Resources

1.1.1 Chaining Select Units

1. One instruction is selected at a time
2. The request is xor’ed with grant bit for the next request

1.1.2 Elaborate Choices

1. Choice box selects $n > 1$ instructions
2. Needs better logic

1.1.3 Asymmetric Select Unit

1. Interesting computer science
2. Have n select units with different policies
3. In case of tie, select unit i, give priority to i^{th} input

1.2 Select Policies

1. Random
2. Oldest first (*seniority is a priority*)
3. Type of opcode
   - load instructions have higher priority
   - other priorities can be learnt based on execution pattern

2 Broadcasting the Tag

2.1 RAW Dependencies

```
add r1 r2 r3
add r4 r1 r5
```

1. We want back to back execution (consecutive cycles)
2. This implies that wakeup and select has to happen in same cycle

2.2 Optimal Pipeline

1. Rename
2. Dispatch
3. Wakeup + select
4. Register read + broadcast
5. Execute
6. Register write

We can ~~forward~~ bypass the data (*senior gives 2nd semester's notes assuming you pass*)

Sir's friend got selected in civil services. Even before training finished (and posting happened), he updated his shaadi.com profile.

2.3 Load Use Hazard

you won't be posted until next elections since your uncle is CM and he doesn't like you :(

1. Broadcasting is done in the first execute stage (execute is of 2 cycles)
2. More generally, if execute stage is k cycles, broadcast k cycles after select
3. Variable length is implemented using state machines - FU knows its delay (except for load/store unit)

3 When should Available Bit in Rename be set?

1. Mark it along with the broadcast for wakeup
2. Alternatively, have double broadcast: in the next cycle, update IW entry for the broadcasts in the last cycle

Lecture 09 (Load Store Queue)

1 Issue with Load Store

```
ld r1, 4[r3]
st r2, 10[r5]
```

1. Can't execute OOO
2. Both may resolve to same address
3. In case of exception, we need to make a clean cut
4. But if store has executed OOO, then memory state is corrupted
5. Loads cannot be directly sent to cache since there might be a pending store

1.1 Resolution Ideas

1. Stores need to wait
2. Need to maintain their information somewhere
3. We use a store queue for this

2 Load Store Queue

1. Allocate an entry at decode time
2. Deallocate later
3. Update entry when address is computed

2.1 Computed Address of Store

Scan entries after this entry (if we encounter a load with same address, then forward) until

1. Store to same address
2. Store with unresolved address

2.2 Computed Address of Load

Scan stores before this entry until

1. Found entry with same address - forward

2. Found entry with unresolved address - wait
3. Reached end of queue - request from memory

2.3 Actual Implementation

Have two separate circular queues for load and store

2.3.1 Load Queue Entry

1. Load address
2. Index of tail pointer in store queue when entry was added

2.3.2 Store Queue Entry

1. Store address and value
2. Index of tail pointer in load queue when entry was added

2.3.3 Basic Search

1. If there are n entries, have a n bit vector
2. $prec(i)$ = all locations before i are set to 1
3. $before(j) = \overline{prec(head)} \wedge prec(j)$ (if no wrap around)
4. $before(j) = \overline{prec(head)} \vee prec(j)$ (if wrap around)
5. $after(j) = \overline{prec(j)} \wedge \overline{map(j)} \wedge (prec(tail) \vee map(tail))$ (no wrap around)
6. $after(j) = (\overline{prec(j)} \vee prec(tail) \vee map(tail)) \wedge \overline{map(j)}$ (no wrap around)
7. To search for resolved entries before j: $before(j) \wedge (match \vee \overline{resvd})$
8. Now to choose the leftmost or rightmost entry, we can use a similar to select logic

We use a tree since it is completely parallelizable. Nix problems exist similar to P vs NP. (size n, poly(n) resources, time taken is poly(log(n)))

Lecture 10 ()

In India we have a standard: we spend a lot of money and it gets wasted. We don't keep it simple.

1 ReOrder Buffer (ROB)

1. It is basically a queue
2. Once it becomes oldest, it is guaranteed to be on the correct path
3. Stores execute when they are the oldest in ROB

1.1 Implementation

1. Contains entry for each instruction that has been fetched
2. Pipeline stalls if table is full
3. Instructions are entered in program order
4. When instruction finishes, its entry is marked ready
5. When it becomes the oldest, it is committed/retired
6. $\exists$ width which defines the number of instructions to commit per cycle

2 Branch Miss Prediction

1. Instructions newer than the branch instruction are on wrong branch
2. No memory has been modified - stores are not executed yet
3. We wait for this instruction to become the oldest
4. Once it becomes oldest, we flush the ROB and restart

3 Restoring physical registers

1. We decide when to update the free list
2. On decoding of a branch instruction, we take a snapshot of RAT
3. Free list is updated when next allocation of the register becomes oldest
   - if prediction had been unsuccessful, then we restore the mapping (when prediction became the oldest)

4 Retirement Register File/Retirement RAT

Mapping of registers when flushing

we studied in online, but exam khud se dena abhi seekhoge

5 Load/Store Instruction

1. Executed only when they are being committed
2. This ensures that they are on right path and there are no interrupts before this

6 Alternatives to RRF and RRAT

1. Take a snapshot
2. SRAM or CAM based
3. Read on your own

Lecture 11 (Alternative Approaches to Issue and Commit)

1 Address Speculation

1. Predict the memory address for a load or store (using n LSB of PC)
2. Predict the stride
3. Store last address, stride, pattern
4. Pattern bit will be set if strided access (array access)

2 Load Store Dependence Speculation

1. Predict if there is collision
2. If collision is not predicted, directly send to cache
3. Else forward values
4. Collision History Table (CHT) is used - 1 bit entry
5. This can be augmented with store load distance and load waits till there are less than D entries in LSQ

M1 has more of these tricks hence it is fast; small savings here and there leads to a lot of savings; being kanjoos :)

2.1 Store Sets

1. n LSBs map to a SSIT (Store Set Id Table)
2. This entry maps to LFST (Last Fetched Store Table)
3. Store updates LSFT (update happens during rename/decode)
4. On load store dependence, SSIT is updated

sad reality: Indians are good in theory, we don't make anything practical. "Chashma bhi imported hai, at least mera to hai; I specifically paid for it to be imported"

We made good TVs, each state had its own TV manufacturing unit

3 Load Latency Speculation

Same idea as branch predictor - hit-miss predictor

4 Value Prediction

4.1 Why are values Predictable?

1. Data redundancy
2. Bit masking
3. Constants
4. Error checking code
5. Register spill code
6. Virtual function code

4.2 Predictor

1. Last value
2. Stride based
3. Based on profiling result

We have a confidence predictor since values aren't always same.

4.3 Virtual Function Code

Each object has a virtual function table where mapping of function to PC is stored

Lecture 12 (Replay)

1 Replay

Instead of flushing the entire pipeline, we only replay the instructions which are affected because of misprediction

2 Forward Slice

The tree of preducer-consumer relation

3 Non-Selective Replay

1. Define a window of vulnerability (WV) for n cycles
2. Load should complete within these n cycles (expected)
3. If it doesn't complete, then we replay

3.1 Squashing and Reissuing

1. If there is a misprediction, all instructions in WV of dependent instruction are squashed
2. Their operands' ready bit is set to zero
3. They are reissued in order of forward slice
4. Issue remains with orphan instructions (those not in forward slice but in WV)

3.2 Implementation

1. There exists a kill wire which is set to 1 on a misspeculation
2. If the timer of the operand is non-zero, then we reset its ready bit to zero
3. Otherwise, we know that this operand will not be squashed

3.2.1 Replaying Instructions

3.2.1.1 Approach 1

1. Maintain an issue queue
2. Remove from IW if the instruction has been verified

3.2.1.2 Approach 2

1. Move the instruction to a replay queue
2. Remove it from the queue if it is verified

3.3 Orphan Instructions

1. There might exist operands that are squashed but were not in the forward slice
2. We can keep a track of squashed instructions and rebroadcast the tag of orphan instructions
3. Alternatively, we execute them when they reach the head of ROB

4 Delayed Selective Replay

1. Extend non-selective replay mechanism
2. At time of asserting kill signal, plant poison bit in destination register of load
3. Propagate the bit along bypass paths and register file
4. When instruction finishes execution, check if poison bit is set
    - if yes, squash it
    - else remove it from IW

Lecture 13 (Replay and Simpler OOO)

1 Token Based Selective Replay

1. 90% of the misses are accounted by 10% of the instructions
2. Each instruction that is predicted to have a miss gets a **free token**
3. The id of the token is stored in the instruction packet
4. Token is a n bit vector if there are n tokens
5. Tokens are propagated similar to poison bits

1.1 After Execution

1. If the token head completed execution in expected number of cycles, broadcast the token id and the operands can turn the bit off
2. Else, token id is broadcasted to signal a replay

1.2 Misprediction

If an instruction that is predicted to not miss has a miss, we flush the pipeline when the instruction reaches head of ROB

2 Simpler OOO Design

1. Physical Register File (PRF) -> ARF
    - have a dedicated ARF to store the committed state
    - enhance the ROB to store uncommitted values
    - rename stage points to either ARF or ROB depending on where the latest value is stored

2.1 Pros

1. Recovery from misspeculation is easy
2. We do not need a free list

2.2 Cons

Values are stored at multiple places

Hardware development has 60% of people who test and verify since correctness is a hard requirement, 25% backend team which handles hardware constraints of the design, 10% of the designers of the chipset (frontend), remaining 5% are the architects

Sir is 16 years post PhD; his friends are now becoming architects

3 Compiler Based Optimizations

1. Constant folding - storing constants into variables instead of computing them
2. Strength reduction - convert multiplication and division to shifts and adds
3. Common subexpression elimination
4. Dead code elimination
5. Silent stores - repititive stores (which are not needed)

3.1 Loop Based Optimizations

1. Loop invariant based code motion - invariant moved outside the loop
2. Induction variable based optimization - multiplication changed to addition with some initialization
3. Loop fusion
4. Loop unrolling

Lecture 14 (Compiler Optimizations and Memory System)

1 Software Pipelining

1. Execute independent instructions at the same time and dependent instructions somewhat later
2. Helps in reducing the waiting time for dependent instruction completion
3. Can use different loop iterators to ensure that this can happen - private iteration

2 Caches

basic overview

Lecture 15 (More on Caches)

escalator » elevator since: open space, can use even if not working, can be used even in emergency

Virat Kolhi's response time is different from Sarangi sir's. Yesterday Afghanistan ke saath century, next time Somalia ke saath

UK will have a new King

in interview, the interviewer is angry; food and tea sucks; they have pressure; they want quick and small answers

for interview, shave and go; no need to wear shirt and coat and tie; no need to do Leetcode and Codechef - they don't help

1 Stuff Discussed

1. Context switching
2. Virtual memory mapping
3. Associative cache
4. Optimizing cache
    - maintain a Victim Cache to overcome associativity (5 blocks needed but 4 way associative)
    - prefetching
    - critical load first
5. Types of cache misses
    - Capacity misses - prefetching
    - Conflict misses - VC
    - Cold misses - prefetching

Lecture 16 (SRAM and CAM)

1 SRAM Array

1.1 SRAM Cell

1. Cross-coupled inverter pair
2. Can be implemented in CMOS as well

1.2 Array

1. Have row address and column address
2. Column is chosen using a mux
3. Row address is used to enable the word line for the entire row

1.2.1 Two Lines

1. We have two outputs BL and BL' for each cell
2. Instead of waiting for the capacitor to charge up, we measure the difference
3. *similar to tarazu - we are that old!*
4. Also helps eliminate noise

1.2.2 Precharging Trick

We precharge both lines to 0.5V

2 CAM Array

2.1 CAM Cell

1. SRAM Cell on top
2. Surrounded by A' and A lines
3. Match line is below SRAM cell
4. T1, T2 in series on left; T3, T4 in series on right
5. When $Q = A$, match line has > 0 voltage

2.2 Array

1. All WLs are set to 1
2. A_i's are set to the tag address
3. Match is set to a non-zero value for all match lines
4. In case of match, a priority encoder will have the correct index

3 CACTI

1. Finds optimal number of banks to have minimal access time, area, power
2. Cuts array into sub-arrays (= bank)

3.1 P-Complete

Given $O(n)$ processors, solve problem in $O(poly(log(n))$

1. DFS
2. Given a circuit, give its output
3. Linear programming

Lecture 17 (Cache Optimizations)

1 Pipelined Cache

1.1 Tasks Involved - Read

1. Tag array access || Data array access
2. Tag comparison
3. Data Block selection

1.2 Tasks Involved - Write

1. Access tag array
2. Tag comparison
3. Data write

2 Non-Blocking Cache

1. Cache miss should not stall later accesses
2. Miss Status Holding Register is used
3. Maintains read/write bit, word address, tag, store value
4. Miss queue is maintained in FIFO order
5. Can minimize the number of miss requests sent to lower levels

3 Skewed Associative Cache

4 Cuckoo hashes are used

3.1 Cuckoo Hashing

1. Have two hash functions
2. Check either of the two locations for lookup or deletion
3. For insertion, if collision, remove old entry and insert it in the other one

4 Way Prediction

1. Reading all tags and comparing is inefficient
2. Predict the way
3. Compare that first
4. Else search all

4.1 Technique

1. Steps: read registers, compute address, check for LSQ forwarding, access d-cache
2. Meanwhile compute $r1 \oplus 12$ (instruction = ld r2, 12[r1]) and get way via the way predictor
3. Prediction available when instruction at d-cache

5 Loop Tiling

pappu code for matrix multiplication; this is used 99% of the time

1. Issue with row-major vs column-major - row better for first matrix, column for second matrix
2. Can instead have tiling in the loop
3. Smaller blocks are multiplied
4. Can store these rectangles in cache

6 Virtually Indexed Physically Tagged (VIPT) Caches

1. We have been ignoring address translation
2. Can use the offset for index and byte offset
3. Page ID and frame ID are like the tag, so this translation can be done in parallel
4. This limits the number of sets in cache

Lecture 18 (Trace Cache)

i-cache is more important than d-cache since instruction fetch happens in order, hence it is more performance sensitive.

1 Trace Cache

1. Basic blocks are defined - single point of entry and exit
2. Have a cache that can store such traces
3. If trace is accurate, prediction is not needed

1.1 Approach

1. Trace consists of multiple cache lines
2. Linked list of cache lines
3. We store the decoded micro ops

1.2 Design

1. Tag array
2. Data array
3. Controller
4. Fill buffer

1.2.1 Tag Array

1. Tag
2. Valid
3. Type
4. Next way
5. Prev way
6. NLIP - address of next CISC instruction
7. micro IP - index into the table of micro ops

1.3 Storage

1. Store trace segments in consecutive sets
2. The way number is stored in next set
3. Set # is used to determine the max size of linked list

1.4 Rules

1. Never distribute micro ops across cache lines
2. Terminate a data line if more branch micro ops than a threshold
3. Terminate trace if
    - We encounter indirect branch
    - Interrupt or branch misprediction notification
    - Maximum length reached

Lecture 19 (Instruction and Data Prefetching)

1 Instruction Prefetching

1.1 Next Line Prefetching

Spatial locality

1.2 Markov Prefetching

1. High correlation between consecutive misses
2. Given that X incurs a miss, predict the line which will incur a next miss
3. Can have n history as well

1.3 Call Graph Prefetching

1. Function call is predictable
2. Predict and prefetch the function

1.3.1 Hardware Approach Call Graph History Cache (CGHC)

1. Each entry contains a list of functions to be executed
2. Index is 1 initially
3. On returning from func1, prefetch func2

2 Data Prefetching

2.1 Stride Based Prefetching

1. Reference Prediction Table (RPT)
2. Each entry is made up of: instruction tag, previous address, stride, state

2.2 Extension

1. Decide when to prefetch
2. Needs to be dynamic

3. Depends on code in the loop

2.3 Pointer Chasing

1. No visible hardware pattern
2. Can insert code to actually prefetch node->next
3. `gcc_intrinsics.h` exists for this

exists a term called black belt programmer

2.4 Runahead Mode

1. Misses in L1 (especially L2) lead to stalls since IW and ROB fill up
2. Idea is to do some work during stall period

2.4.1 Implementation

1. Return a a junk value for the miss
2. Restart execution with junk value - add INV (invalid) bit
3. This is useful since we will prefetch data and train predictors
4. Once miss returns, flush instructions and re-execute instructions
5. For data requests during this time, maintain a runahead L1 cache

2.4.1.1 Runahead L1

1. If store is INV because of data (not because of address), prefetch this address
2. Can maintain additional state of INV and INV store

2.4.1.2 Loading Value

1. Try forwarding in LSQ
2. Else, access runahead cache
3. If miss, try accessing from L1
4. Else, load from L2 (prefetching)

2.5 Helper Threads

1. Spawn threads to execute backward slice of load
2. Backward slice determines the address of load
3. If resources available, saves time

Lecture 20 (On Chip Network)

1 Layout of Memory Chip

1. Checkerboard - core and cache bank alternate
2. Rim layout - cache inside, core outside

2 Router

1. Each node has a router that communicates on its behalf
2. Has a data buffer to send information

2.1 Connection between Routers

1. $D = kl^2$, l is length of wire
2. Delay increases a lot as length increases

2.1.1 Buffered Wires

1. Add buffers to reduce delay
2. For optimal number of buffers, we get minimum delay as $D = 2\sqrt{kd}l - d$, where d is delay of buffer

3 Multi Layer Interconnects

Alternates between horizontal and vertical between different vertical layers

4 Interconnect in Silicon Chips

1. Bus approach - fails trivially
2. Tiling - Network on Chip

4.1 NoC

1. Arrange a set of nodes as tile

2. Router sends and receives all messages for its tile
3. Router also performs forwarding

5 Parameters

5.1 Bisection Bandwidth

1. Number of links that need to be snapped to divide NoC into two equal parts
2. Gives idea of *path diversity*

5.2 Diameter

Maximum optimal distance between any two pair of nodes

Note: Sir knows Delhi roads quite well

6 Topologies

1. Chain
2. Ring
3. Fat tree
4. Mesh
5. Torus
6. Folded torus
7. Hypercube
8. Clos Network

6.1 Hybercube

1. Recursive structure
2. If it has N nodes,
   - diameter = $\sqrt{N}$
   - bisection bandwidth = $N/2$

6.2 Clos Network

1. nr inputs and outputs
2. Inner layer has a $(m \times m)r$ switches
3. Performs a permutation
4. If $m \geq n$, we can reconnect unused input to unused output by rearranging
5. If $m \geq 2n - 1$, we can reconnect without rearrangement

6.2.1 Butterfly Network

Uses only 2×2 switches and $\log(n)$ layers

7 Message Transmission - Hierarchy of Messages Sent

1. Message
2. Packet - head flit, body flits, tail flit
3. Flit - flow control digit (typically 8 or 16 bytes)
4. Phit - physical digit

8 Flow Control

1. We can't drop flits unlike in network transmission
2. Sender needs to have idea about free space at receiver's end

8.1 Credit Based Flow Control

1. Sender (A) maintains an estimate of number of free buffers at receiver (B)
2. If A thinks B has enough free space, only then it sends

8.1.1 Assumptions

1. Routers are clock synchronized
2. Time is measured in number of cycles
3. We first receive a message, process it and add it to buffer
4. Status messages are 1 phit each

8.1.2 Formulation

$$t_D = t_{ph} + t_f + 2t_{pr}$$

$$t_{ph} = \text{ time taken for receiving status message (credit) - phit}$$

$$t_f = \text{ time to send flit}$$

$$t_{pr} = \text{ processing time}$$

Lecture 22 ()

man working in 9 companies during COVID time, police involved

1 Credit Based Flow Control (contd)

1. If t_D/t_f buffers are available, then A will never stall
2. But need to send 2x messages

2 On-Off Flow Control

Only send credit when number of free buffers falls below N_{off} or rises above N_{on}

2.1 Analysis

1. $N_{off} \geq t_D/t_f$ (can be determined using risky period - both flits and credit in transit)
2. $N - N_{on}$ should be greater than a threshold for efficiency: $N \geq 2t_D/t_f$

3 Circuit Switching

1. Reserve path from source to destination
2. On reservation, send message
3. After sending message, clear buffer

trunk calling still happens in Ethiopia

3.1 Space Time Diagram

1. Probing time: K cycles
2. Response time: K cycles
3. Sending time: $K + L/B - 1$
4. Total time $= 3K + L/B - 1$

3.2 Pros and Cons

1. Good for bulk transfer
2. Terrible for single transfer
3. Locks up resources

4 Packet based Flow Control - Store and Forward

1. Receive entire packet at next router, then forward
2. Takes $K * L/B$ cycles

5 Virtual Cut Through (VCT)

1. Don't wait for entire packet to come
2. Takes $K + L/B - 1$ cycles
3. But need to ensure enough space for entire packet

5.1 Solution

1. Flow control at flit level
2. But more issues

Lecture 22 (Flow Control and Routing)

1 Wormhole Flow Control

1. Virtual cut through but flow control is at flit level
2. If head is stuck, then stalling happens again - Head of Line blocking

2 Virtual Channels

Have multiple queues (lanes) instead of a single queue

3 Issues in Routing

1. Deadlock
2. Livelock
3. Starvation (superset of deadlock)

4 Generic Solution

Introduce ageing - give priority to oldest packets

5 Understanding Deadlocks - Resource Dependence Graph (RDG)

1. Arrow from current location (what resource it is holding) to packet
2. Arrow from packet to what it wants
3. This deadlock can also happen with virtual channels

Lecture 23 (Routing)

koi bhi chiz ratna bilkul bura nahi hai, quote me on this - SR Sarangi

1 Turn Graph

1. Both CDGs and RDGs lose orientation information of the channels
2. Consider any path C in the channel graph
3. The TG contains all the nodes and channels belonging to C
4. This preserves the orientation of the channels
5. TG does not contain any other channels
6. We insert new nodes called channel node in middle of each edge/channel
7. Cycle in channel nodes of TG $\iff$ cycle in CDG

1.1 Properties

1. Every edge in the CDG translates to either a set of collinear nodes or turns in the TG
2. Every cycle in CDG is a cycle in TG
3. Every cycle in the CDG can be translated to a sequence of straight paths and turns in the corresponding TG

1.1.1 Aim

From above we get that ensuring that there are no cycles in routing algorithm ensures that there will be no deadlock

2 Cycle-Free Routing Algorithms

2.1 Dimension-Ordered Routing

1. First move along one axis, then the other and so on
2. However, no path diversity
3. Cannot handle congestion

2.2 Oblivious Routing (Valiant's Algorithm)

1. Select a random point P
2. Perform Dimension-Ordered routing from A to P and then from P to B
3. Low congestion because of high path diversity
4. However, the routes can be very long

2.3 Minimally Oblivious Routing

1. Restrict the domain of P around B
2. Reduces path diversity

2.4 Adaptive Routing

1. Out of all possible turns, use maximal subset of turns such that they can never form a cycle
2. For each cycle, remove one turn

2.4.1 Examples

1. West-first
2. North-last
3. Negative-first

2.5 Data Line based Routing

1. Explained for 2 VCs
2. Assume that each VC has its own deadlock free channel
3. Inject packet into VC0
4. If it crosses the "date line", it moves to VC1

3 Causes for Deadlock

1. MutEx
2. Circular wait
3. No preemption
4. Hold and wait

Lecture 24 (Route Computation)

1 Routing Table

1. Each node maintains a routing table and specifies the possible next hops based on final destination
2. Make a choice out of possibilities depending on congestion information
3. Take into account the delay incurred in sending flits the last time the channel was used

2 Allocate Switch Ports

Can design a $m \times n$ switch for input vs output

2.1 Combine Smaller Switches

1. For a $m \times n$ switch, latency is $m + n$ and area is $m \times n$
2. For 10×5, we can combine as:
   i. 5 switches of 2×1 followed by 5×5 switch - better
   ii. 2 switches of 5×2 follows by 4×5 switch

2.2 Dimension Sliced Switch

Used for X-Y routing, saves on area

Startup in switch industry will work really well. They are very expensive. SIT ke liye planning was done by faculties

3 Allocation and Arbitration

1. Arbiter chooses one out of N requests for resource allocation
2. Allocator creates one to one mapping between N requests and M resources - bipartite matching

3.1 Round Robin Arbiter

Combinational logic that performs round robin

3.2 Matrix Arbiter

1. If given agent is not interested, it sets entries in its row to 0 and in column to 1
2. In every cycle, request is granted to the agent who has 1 in all entries of its row
3. Once agent i is done servicing, it sets all entries in its row to 0 and all in column to 1

3.3 Separable Allocator

1. First column selects resource
2. Second column selects agent
3. Does not give maximal matching

3.4 Wavefront Allocator

1. Start by giving each diagonal element a row and column token
2. Each round, row token moves 1 step to left, column moves 1 step down (with wraparound)
3. If agent i is interested in resource j, it grabs both row and column token when it receives them
4. If some i, j is chosen, then
    i. no other agent can request for resource j
    ii. agent i cannot request for any other resource
5. This ensures maximal matching

4 Router Pipeline

1. Buffer Write
2. Route Computation
3. VC Allocation
4. Switch Allocation
5. Switch Allocation

5 Lookahead Routing

1. Compute route for next hop
2. Send routing decision along with packet
3. Removes route computation from critical path

Lecture 25 (Optimizing NoC and Performance Analysis)

1 Bypassing

1. If router queues are empty
2. Attempt to directly traverse the switch

2 Speculative VC Allocation

1. Allocate switch and VC simultaneously
2. Saves another stage if VC is found
3. Else, resort to conventional methods

3 Late VC Selection

1. Maintain a queue of free VSc with each outgoing link
2. When head flit traverses the switch, assign it a VC from the queue
3. If free VC is not available, cancel the process and restart the conventional process

4 Non-Uniform Cache (NUCA)

It is better to access data from adjacent cache lines since data traversal time is lesser

4.1 Static NUCA

1. Map cache blocks to cache banks
2. Have mapping as tag ID | bank ID | set ID | byte

4.2 Dynamic NUCA

1. Define a bank set - columns of banks
2. Home bank is the closest bank in the set from the core
3. For searching, we follow one of the three strategies
    i. Sequential

   ii. Two-way
   iii. Broadcast
4. On a hit, we move the block closer to the home bank
5. For eviction, instead of moving it to lower level, we move it away from home bank

5 Performance Aspects

1. Architectural Simulator
2. Synthetic Traffic Based Simulator

5.1 Synthetic Traffic Generation

1. Random traffic
2. Bit-complement - $(D_x, D_y) = (\bar{S_x}, \bar{S_y})$
3. Transpose - $(D_x, D_y) = (S_y, S_x)$
4. Bit-reverse
5. Bit-rotation - left or right shift
6. Shuffle - similar to left shift
7. Tornado - translation of coordinates along some line

Lecture 26 (Multi-Core Systems)

apparently working memory is important

1 Shared Memory vs Message Passing

1.1 Shared Memory

1. Easy to program
2. Issues with scalability
3. Code is portable across machines

1.2 Message Passing

1. Hard to program
2. Scalable
3. Code may not be portable

2 Amdahl's Law

$$T_{par} = T_{seq} \times \left(f_{seq} + \frac{1 - f_{par}}{P} \right)$$

$$S = \frac{1}{f_{seq} + \frac{1-f_{par}}{P}}$$

3 Gustafson-Barsis's Law

New workload: $W_{new} = f_{seq}W + (1 - f_{par})PW$

Sequential time: $T_{seq} = \alpha W_{new}$

Parallel time: $T_{par} = \alpha(f_{seq}W + (1 - f_{par})PW/P)$

Speed-up: $S = f_{seq} + (1 - f_{seq})P$

4 Design Space of Multiprocessors

1. SISD - Single Instruction Single Data
2. SIMD - Single Instruction Multiple Data
3. MISD - Multiple Instruction Single Data
4. MIMD - Multiple Instruction Multiple Data
    i. SPMD - Single Processor Multiple Data
    ii. MPMD - Multiple Processor Multiple Data

5 Hardware Threads

1. Cores with large issue widths have wasted issue slots
2. Run multiple processes simultaneously on the same core
3. Maintain a hardware thread ID
4. Need separate ROBs (and some other hardware implementation details)

6 Coarse-Grained Multithreading

1. Run instruction for each thread for k cycles
2. Separate program counters, ROBs and retirement register files per thread
3. Thread ID is tagged with each entry
4. Switch in case of high-latency event

7 Fine-Grained Multithreading

1. Small k
2. Switch happens in case of low-latency event as well

8 Simultaneous Multithreading

1. Dynamically split the issue slots between threads
2. Heuristics
    - fairness
    - instruction criticality
    - thread criticality
    - instruction type
3. Hyperthreading is SMT but static partitioning

9 Issues with Large Caches

1. Access times can be very slow

2. Parallel accesses will make it slower
3. Have distributed caches - “shared private cache”

Lecture 27 (Memory Consistency Models)

1 Valid Outcomes

1. Different situations can have different outcomes for the same program
2. Thus, we have a set of valid outcomes for a single program
3. Every processor (and memory system) has a set of specifications that specify the allowed outcomes/behaviours

2 Memory Consistency vs Coherence

1. Coherence deals with a single memory location
2. Consistency is about what are the allowed values during execution of program

3 Observer

1. Each observer has a different PoV
2. Completion time might be anytime between start and end time (or even later)

4 Sequential Execution vs Legal Sequential Execution

1. Sequential is simply ordered
2. Legal is when every read reads the latest write
3. Observer at memory location sees a legal sequential execution

5 Atomicity

1. Each operation has a single global completion time
2. Can generate a sequential execution if this property satisfies

$P|T$: all operations issues by thread T in order

$P|T \equiv S|T$: one to one mapping from parallel to sequential execution

$P \equiv S$: for all T, $P|T \equiv S|T$

6 Sequential Consistency

1. Atomicity leads to sequential consistency
2. Additionally we assume that program order (within thread) is preserved
3. SC is gold standard (kinda)

math is to state the obvious by scaring you

6.1 Per Location SC

1. Provides the illusion of a single memory location even if we have a distributed cache
2. SC is difficult to implement but PLSC is needed

6.2 Why not SC?

1. Loads will need to be issued at commit time
2. Benefits of OOO and LSQ will go away
3. For high performance SC needs to be sacrificed

7 Non-Atomic Writes

1. Writes might be seen at different times by different threads
2. This cannot be in SC
3. However, this can be in PLSC and we are happy
4. We allow non-atomic writes but PLSC should be preserved

Lecture 28 ()

1 PLSC and Coherence

1. If accesses to a single location preserve program order, then we have no issues
2. However, we might have non-atomic writes because of other addresses
3. Accesses for a single address are atomic

2 Ordering between Accesses to same Variable

(observer is at the core)

1. Read $\rightarrow$ Read: no difference
2. Write $\rightarrow$ Read: this may be global since writes might reach later for other cores (since atomicity need not exist)
3. Write $\rightarrow$ Write: needs to be global
4. Read $\rightarrow$ Write: this becomes global since writes are ordered

3 Axioms of Coherence

1. Write Serialization: Writes to the same location are globally ordered
2. Write Propagation: A write is eventually seen by all the threads

4 `fence`

1. Store instruction completes when all threads can read the new value
2. Instruction that ensures that all instructions prior to it complete execution
3. It is a memory barrier

5 Execution Witness

1. Parallel Execution $\rightarrow$ Execution Witness $\rightarrow$ Sequential Execution
2. If we can create SE that obeys the memory model, then the execution is valid
3. EW is a graph with nodes as instructions

4. Edges can be local and global
5. These edges are happens-before edges (hb)

Lecture 29 (Execution Witness)

1 Program Order (po) Edge

1. Edges between reads and writes (both directions)
2. Edges between instruction and synch (both directions)

2 Read From (rf) Edge

1. Write to read edge
2. Can be external and internal
3. If writes are atomic, then `rfe` edges are global
4. `rfi` is not global because of LSQ forwarding

3 Write Serialization (ws) Edge

1. Originates from PLSC and coherence
2. Is always global

small is beautiful - not sir talking about class size

4 From Read (fr) Edge

1. Read to write edge
2. It is global

5 Synchronization Edge (so) Edge

1. Exist for synchronization variables
2. All updates to such variables are synch

6 Summary

1. `ws`, `fr` and `so` are always global
2. `ppo` $\subset$ `po` be global
3. `grf` $\subset$ `rf` be global
4. `grf` $=$ `rf` and `ppo` $=$ `po` will hold in case of SC
5. `ghb` $=$ `ppo` $\cup$ `ws` $\cup$ `fr` $\cup$ `grf`
6. Subset is decided according to memory model

7 Cycles in EW

1. We perform a toposort to get the sequential execution
2. If sort is not possible, then we have a cycle

8 Access Graph

1. `up` edges between accesses to same location in same thread
2. `PLSC` $\equiv$ `up` $\cup$ `ws` $\cup$ `fr` $\cup$ `grf`

9 Data and Control Dependence

1. We add causal dependencies (`dep` edges) for if (and similar) statements
2. Otherwise we might have “thin air read”
3. `causal graph $\equiv$`rf`$\cup$`gpo`$\cup$`dep‘

Lecture 30 (Cache Coherence)

1 Write Update Protocol

1.1 States

1. M - modified
2. S - shared
3. I - invalid

1.2 Event/Message Notification

1. Rd - Read request
2. Wr - Write request
3. Evict - Evict the block
4. Wb - Write back data to the lower level
5. Update - Update the copy of the block

1.3 Message Types

1. RdX - Generate a read miss message. Send it on the bus/NoC if required
2. WrX -Generate a write miss message. Send it on the bus/NoC if required
3. WrX.u - Get permission to write to a block that is already present in the cache
4. Broadcast - Broadcast a write on the bus
5. Send - Send a copy of the block to the requesting cache

1.4 Snoopy Protocol

1. All messages are broadcast messages
2. Since they are sent on bus, all caches can read
3. Easy to design but not scalable

1.5 State Machine

1.6 Events Received from the Bus

1. Only one sister cache can use the bus at any time, global order of writes is preserved

2. If bus master disallows starvation, then all writes will complete

2 Write Invalidate Protocol (MSI Protocol)

1. In M state, only one cache can contain a copy of the block
2. Multiple cache can have the block in S state as read-only

2.1 State Machine

Such state machines are called “Distributed State Machines” or “Actor Models”

3 MESI Protocol

1. Introduce a new state E - exclusive
2. Avoid modify request on bus when exclusive owner

4 Important Questions

1. Who supplies data if sister sends read miss or write miss?
    - Caches who have a copy of cache, arbitrate for the bus. Whoever gets the access first, sends the data. Others snoop this and cacnel their request.
    - This leads to overhead
2. Do we need to write-back on $M \rightarrow S$ transition?
    - If we can avoid it, then we can optimize performance and power

5 MOESI Protocol

1. Introcude O - owner state
2. This state will send data for any read miss requests
3. It write-backs on eviction
4. We also introduce some temporary states in case there is no owner

Lecture 31 (Directory Protocol)

knowing English and following Indian accent are two different things

1 Directory Protocol

1. When we do not have a bus
2. Have a dedicated structure called directory
3. It co-ordinates the actions of coherence protocol
4. Sends and receives messages to/from caches and lower levels

1.1 Directory Entry

1. State
2. Block address
3. List of shareres

1.2 Design of Directory

1. RdX - locate a sharer and fetch the block
2. WrX and WrX.u - ask all sharers to invalidate their lines and give exclusive rights to write requestor
3. Evict - delete the sharere from entry

1.3 Issues

1. Need a very large directory
2. Directory may also become a point of contention

1.4 Resolution Ideas

1. Distributed directories - split the physical address space, resolves contention issues
2. Directory as cache - if entry is evicted from directory, invalidate all sharers

1.5 How to Maintain List of Shareres

1. Fully mapped scheme - inefficient
2. Maintain a bit for a set of caches - snoopy protocol inside this set
3. Partially mapped scheme - store id of only k sharers, if more than k, broadcast

2 Memory Models

2.1 Write-to-Read Order

1. `rfi` is not global
2. Because of LSQ forwarding and write buffers

2.2 Non-Atomic Writes

1. `rfe` is not global
2. Because of local tiles of caches

2.3 Write-to-Write Order

1. This is not allowed even if writes are atomic
2. But is violated in case of non-blocking caches

2.4 Read-to-Read Order

1. Violated because of OOO loads in LSQ

2.5 Read-to-Write Order

1. Is maintained in OOO machines in general
2. Violated when we have speculative writes

2.6 Can `rfi` be relaxed in SC?

1. Answer is yes
2. Proof is in book

Lecture 32 (Locking and Stuff)

1 xchg Instruction

1. Read memory, modify memory, write register - RMW instruction
2. Get exclusive access with write permission for memory address
3. Perform the RMW operation
4. But do not respond to any other requests from local or other caches or directory while this operation is in progress
5. Respond after execution

2 Spinlock

```
.lock:
    mov r1, 1
    xchg r1, 0[r2]
    cmp r1, 0
    bne .lock
    ret
```

```
.unlock:
    mov r1, 0
    xchg r1, 0[r2]
    ret
```

3 Test and Exchange Lock

```
.lock:
    mov r1, 1

.test:
    ld r2, 0[r0]
    cmp r1, 0
    bne .test
```

```
xchg r1, 0[r0]
cmp r1, 0
bne .test
ret
```

4 Atomic Operations

1. Test and set - `tas r1, 0[r0]`
2. Fetch and increment - `fai r1, 0[r0]`
3. Fetch and add - `faa r1, r2, 0[r0]`
4. Compare and Set - `cas r1, r2, r3, 0[r0]`
5. Load linked, store conditional - `ll r1, 0[r0]` and `sc r3, r2, 0[r0]` (store only if value not modified since `ll`)

5 Eliminating Starvation

1. Request T finds another request R that is waiting for a long time
2. T decides to help R
3. This is an altruistic algorithm

6 Consensus Number

Maximum number of threads that can solve a problem using a wait-free algorithm

Lecture 33 ()

1 When to use Locks

When multiple threads access same memory location and perform conflicting and concurrent accesses.

2 Data Race

Pair of conflicting and concurrent accesses to the same regular variable constitute a data race.

- SC does not imply data race freedom
- Data race freedom implies SC

2.1 Theorem

If we have a data race in a program (any memory model, any EW), we can construct an SC execution that also has a data race. Contrapositive is more important: if we cannot construct an SC EW that has a data race, then the program is data race free.

Had I known I could have made money by professional anti-vaxxing, I would have done it as well

Lecture 34 (Secure Processors)

1 Confusion

1. If a single bit in the key is changed, then most or all of the ciphertext bits should be affected
2. This ensures that the key and ct are not related

2 Diffusion

1. If we change a single bit in the plaintext, then half the bits in ct should change
2. Prevents related message attacks

3 Rounds in AES

1. Write the 16 bytes as a 4×4 matrix
2. Replace each of them using a lookup table S-box
3. Left rotate the i^{th} row by i positions
4. Take the four bytes in each column and modular multiply it with a matrix
5. Compute a bitwise XOR with the round key

4 Generating Round Key

1. Rotate word
2. Substitute word
3. XORWord - $B_0B_1B_2B_3 \rightarrow$
    - $B_0 = RC[i]$
    - $RC[0] = 1$, $RC[i] = 2 \cdot RC[i-1]$

5 AES Algo

1. First round - only the XOR with round key is performed
2. Final round - mix columns is skipped