CS 61C Notes 2

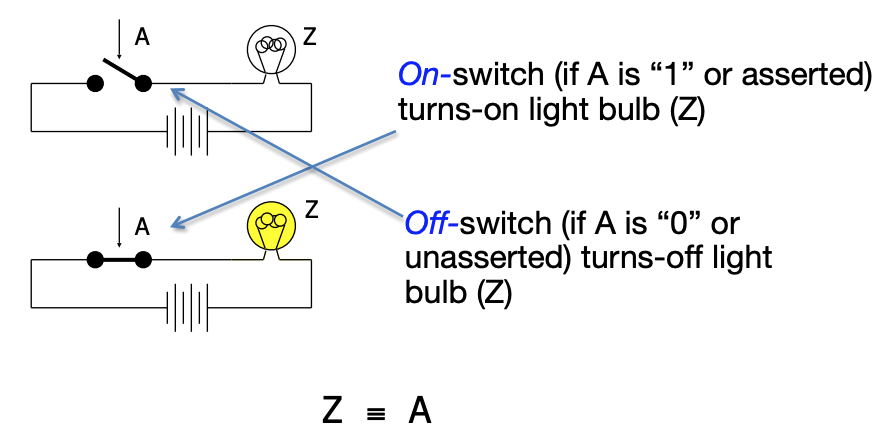
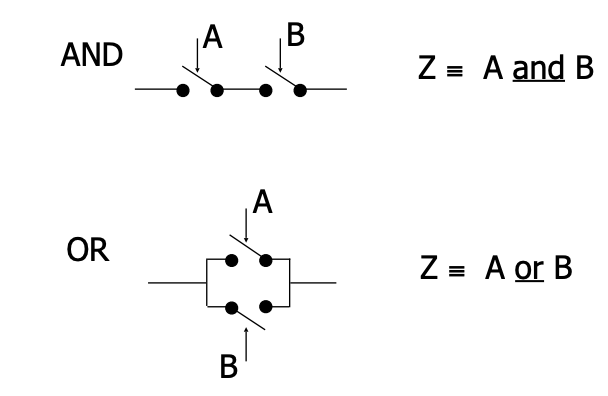
LEC 09: Hardware Design

Synchronous Digital Sustems:

* **Synchronous**
  + All operations coordinated by a central clock
  + “Heartbeat” of the system!
  + **Asynchronous** systems much much much much harder to design & debug
* **Digital**:
  + Represent all values by discrete values
  + Two binary digits: 1 and 0, or True and False
  + Electrical signals are treated as 1’s and 0’s
    - High/low voltage for true/false, 1/0

Switches: Basic Element of Physical Implementations:

* Simple Circuit design: Compose switches into Booleans:

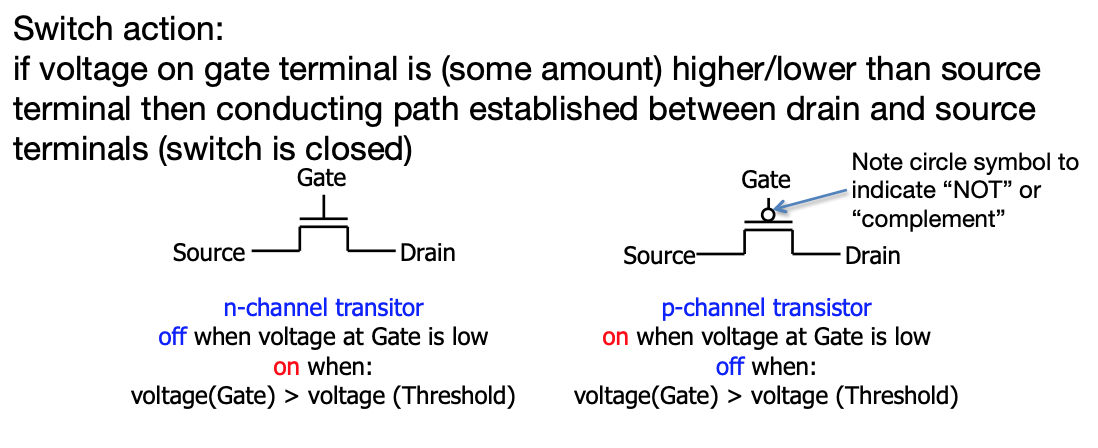


Transistors:

* High voltage (Vdd) represents 1, or true
* Low voltage (0 Volt or Ground) represents 0, or false
* Pick a midpoint voltage to decide if a 0 or a 1
  + Voltage greater than midpoint = 1
  + Voltage less than midpoint = 0
  + This removes noise as signals propagate – a big advantage of digital systems over analog systems

We Use CMOS Transistors:

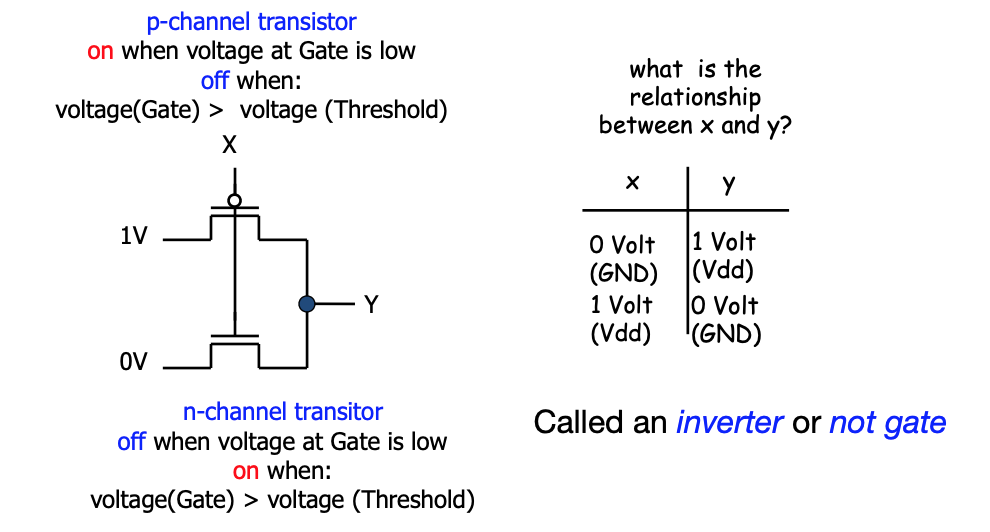
* **MOS**: Metal-Oxide on Semiconductor
  + Describes how the transistors are constructed
  + These are "Field Effect Transistors" (MOSFETs)
* **C** for complementary: use **pairs** of normally-on and normally-off switches
* CMOS transistors act as **voltage-controlled switches**
* **Three** terminals: **source**, **gate**, and **drain**

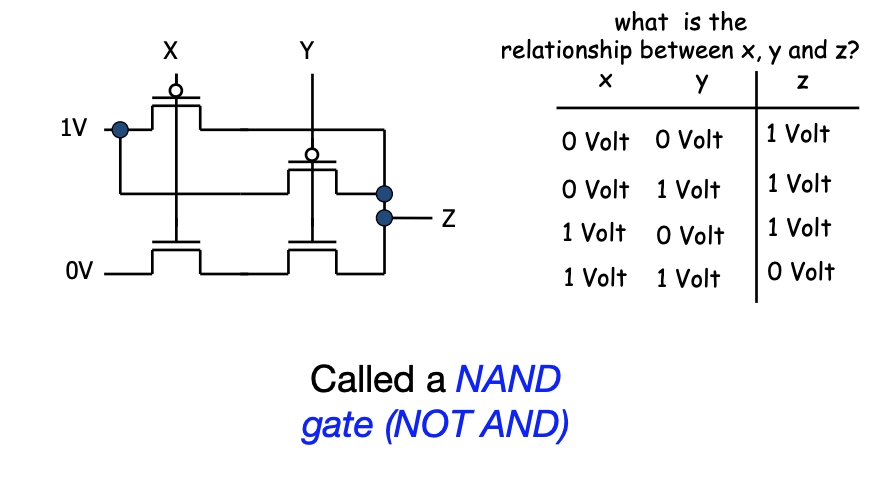


CMOS Circuit Rules:

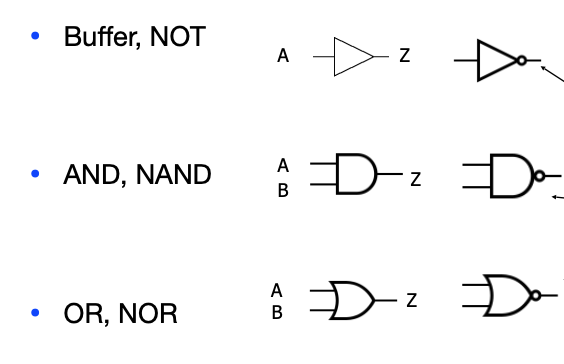
Don’t pass weak values => Use Complementary Pairs

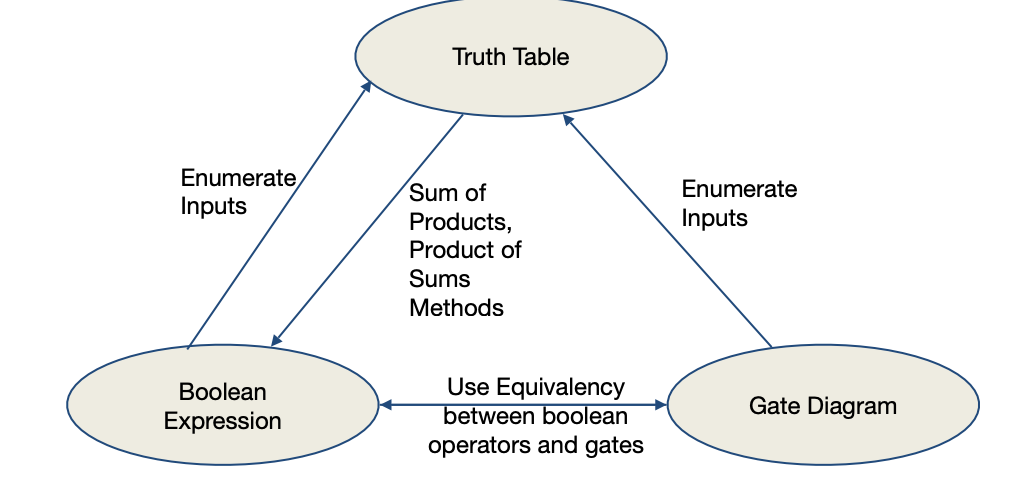
* N-type transistors pass weak 1’s (Vdd - Vth)
* N-type transistors pass strong 0’s (ground)
* Use N-type transistors only to pass 0’s (N for negative)
* Converse for P-type transistors:
  + Pass weak 0s, strong 1s
  + Pass weak 0’s (Vth), strong 1’s (Vdd)
  + Use P-type transistors only to pass 1’s (P for positive)
* Use pairs of N-type and P-type to get strong values

CMOS Networks:



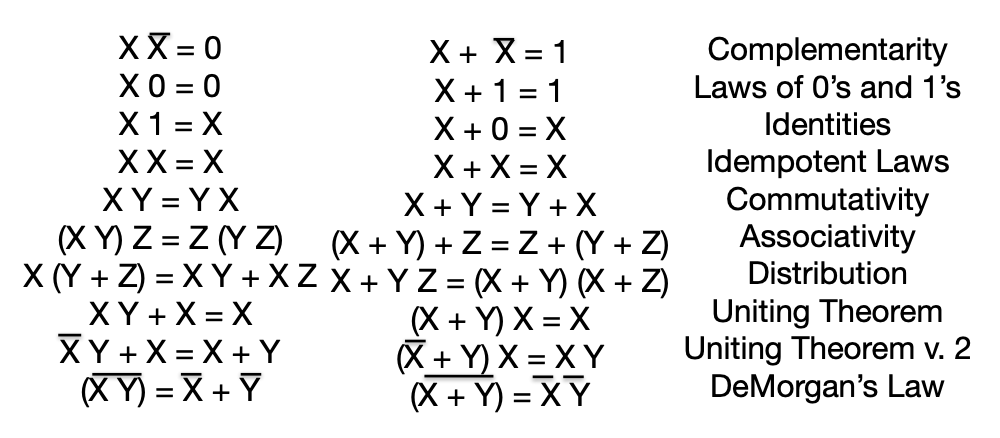
Combinatorial Logic Symbols (Logic Gates):



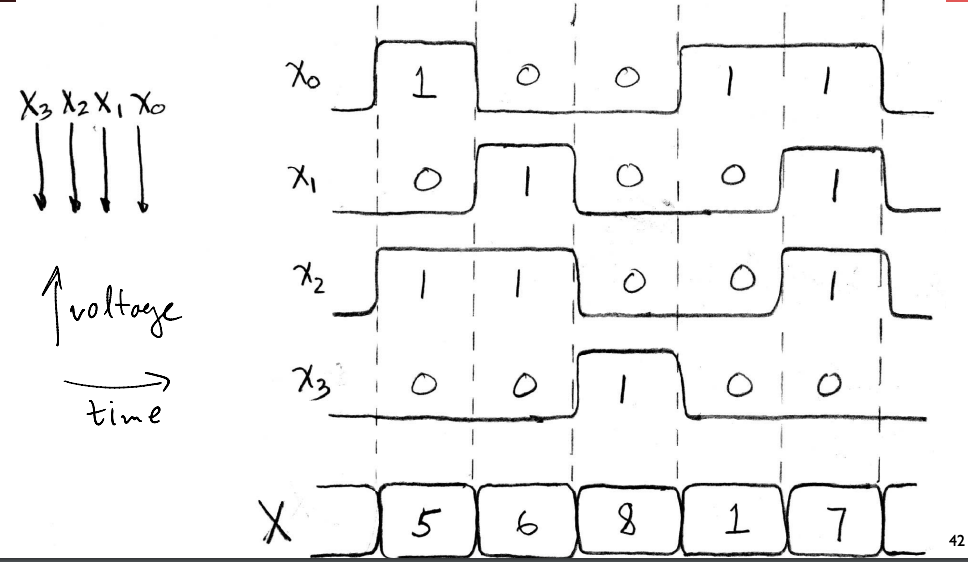
Boolean Algebra:

* Use plus “+” for **OR**
  + “logical sum”
* Use product for **AND** (a•b or implied via ab)
  + “logical product”
* “Hat” to mean complement (**NOT**)

Laws of Boolean Algebra:



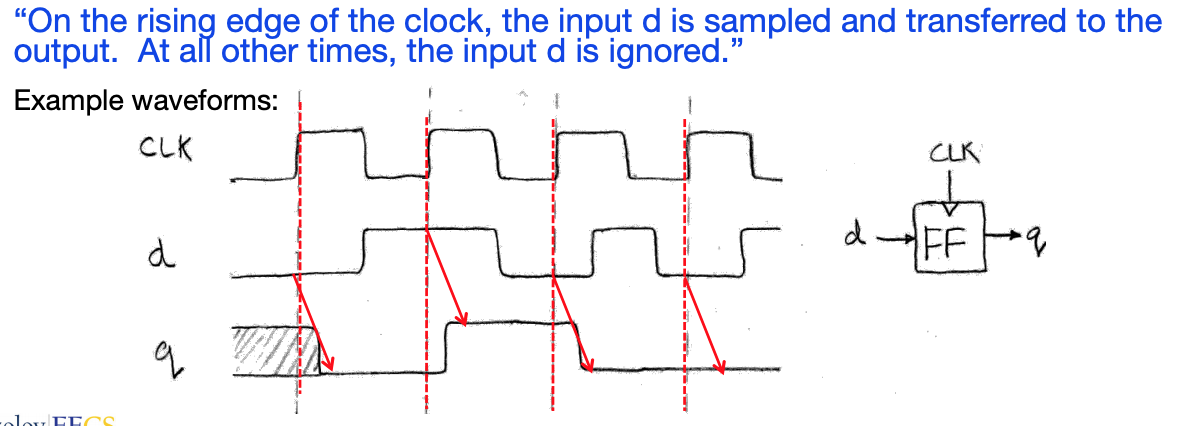
Signals and Waveforms: Showing time and grouping:



2 Types of Circuits:

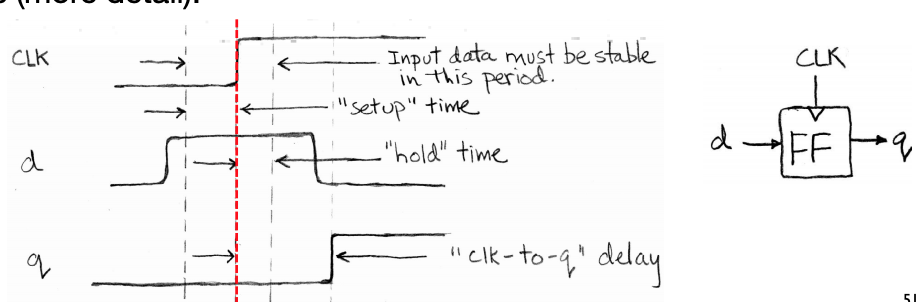
* **Combinational Logic (CL) circuits**
  + Output is a function of the inputs only, not the history of its execution
    - E.g., circuits to add A, B (ALUs)
* **Sequential Logic (SL)** 
  + Circuits that “remember” or store information
  + aka “State Elements”
    - E.g., memories and registers (Registers)

Register Internals:

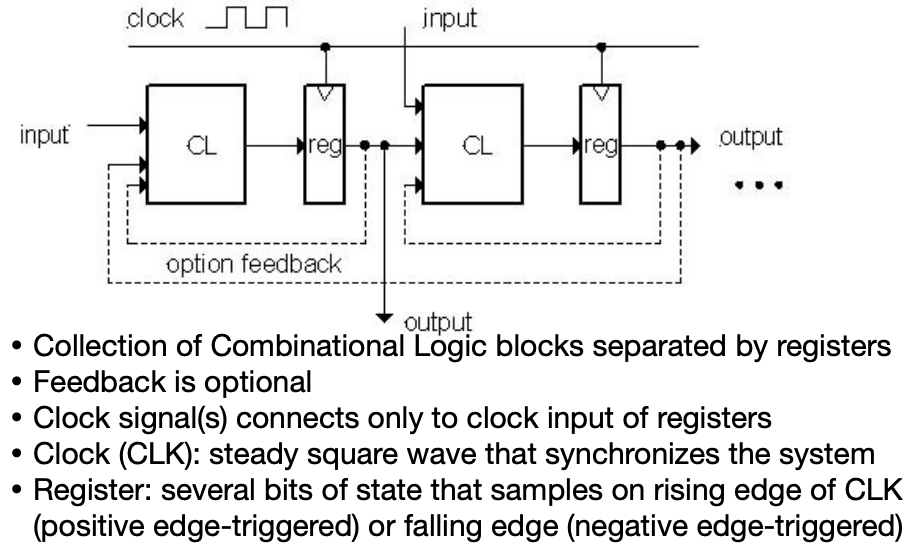
* n instances of a **“Flip-Flop”**
* **Flip-flop** name because the output flips and flops between 0 and 1
* D is “data input”, Q is “data output”
* Also called “D-type Flip-Flop”
  + When clock goes from low to high, copy the input to the output
* Edge-triggered d-type flip-flop
  + This one is “positive edge-triggered”

Setup Time and Hold TIme:

* **Set up time:** when the input must be stable before the edge of the CLK
* **Hold time:** when the input must be stable after the edge of the CLK
* **CLK-to-Q” Delay:** how long it takes the output to change, measured from the edge of the CLK



Model For Synchronous Systems:



LEC 10: More Digital Circuits

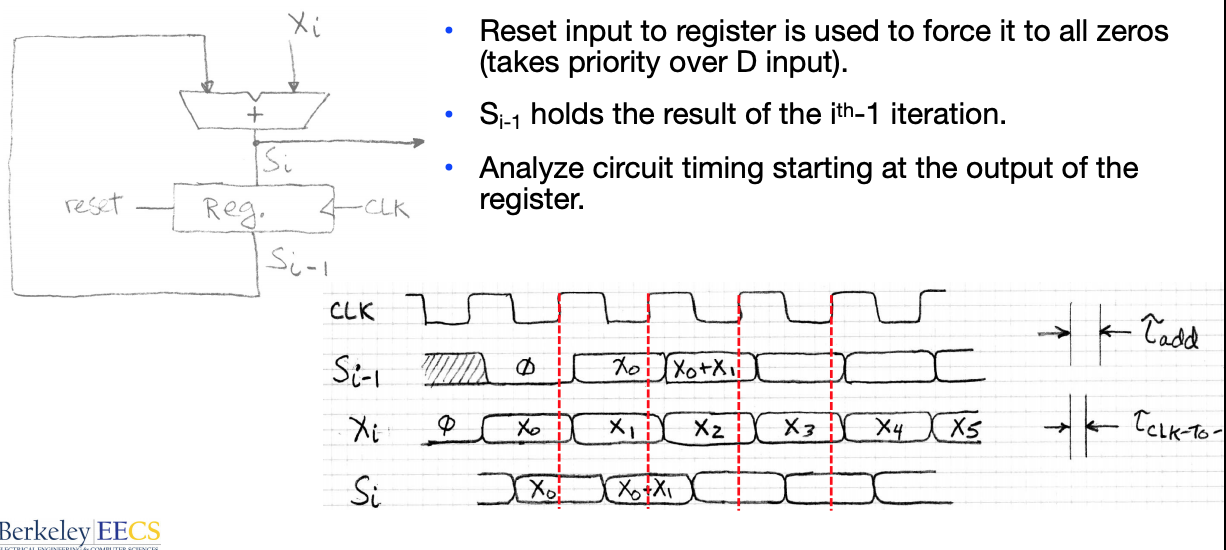
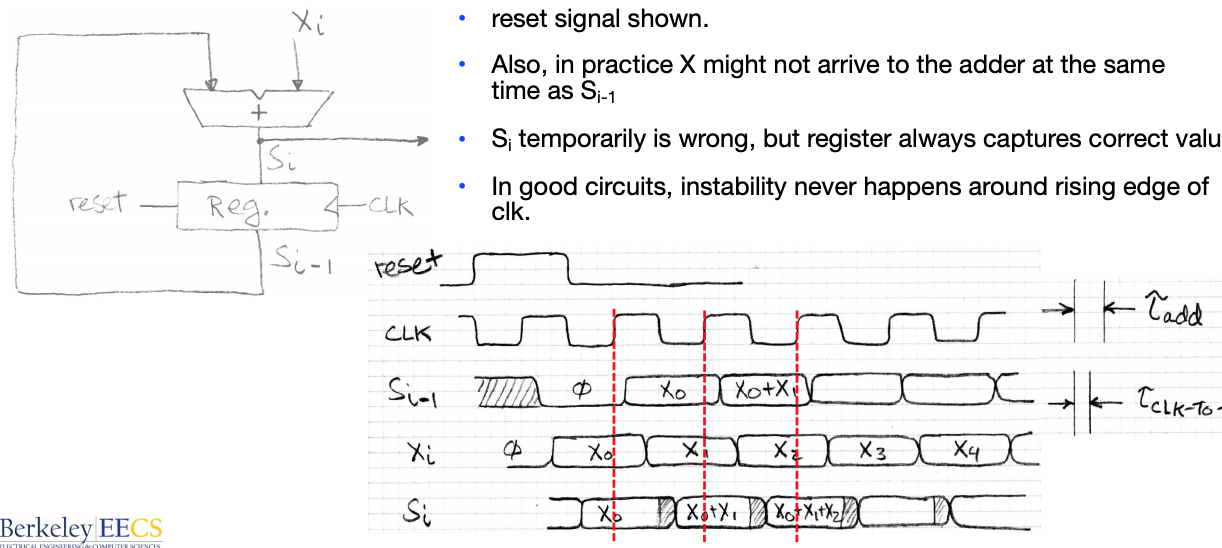
Uses for State Elements (Sequential Logic Circuits):

* Place to store values for later re-use:
  + Register files (like x1-x31 in RISC-V)
  + Memory (caches and main memory)
* Help control flow of information between combinational logic blocks
  + State elements hold up the movement of information at input to combinational logic blocks to allow for orderly passage

How to build a Flop Flop?

Two Latches Example:

* When clk is high...
  + D -> Q
* When clk is low...
  + Q stays with whatever it was
* Chain 2 latches together to create a flip-flop
* Setup time:
  + Need to propagate D to Q on the first latch
* Hold time:
  + Need to make sure the first latch doesn't change before the clock fully switches
* Clk->Q time:
  + Time needed to go through the second latch

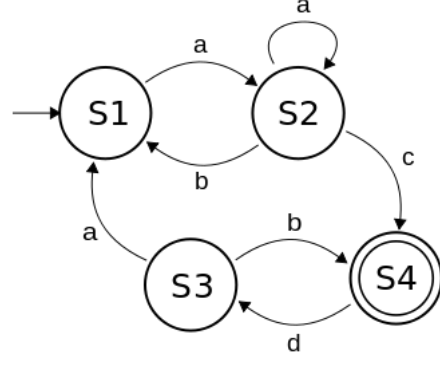
Accumulator Timing:

Problems With Clocking:

* The clock period **must be** longer than the critical path
  + Period = Max Delay = CLK-to-Q Delay + CL Delay + Setup Time
  + Otherwise, you will get the wrong answers
  + But it can be even longer than that
* Critical path:
  + clk->q time
    - Necessary to get the output of the registers
  + **worst case** combinational logic delay
  + **Setup time** for the next register
* Must meet all of these to be correct

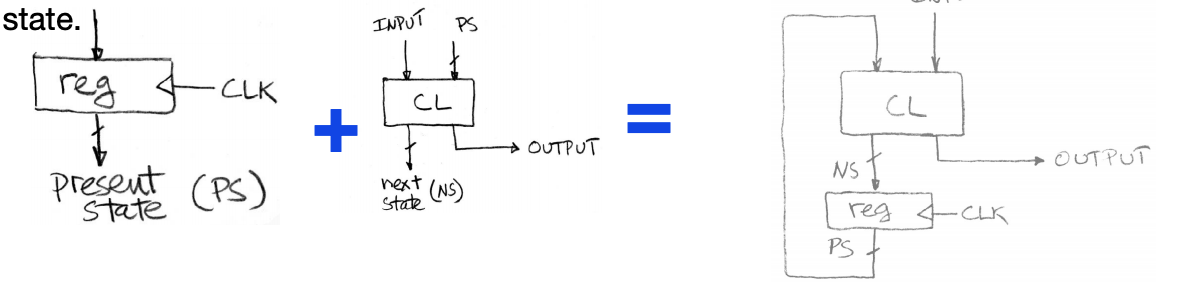
Hold Time Violations:

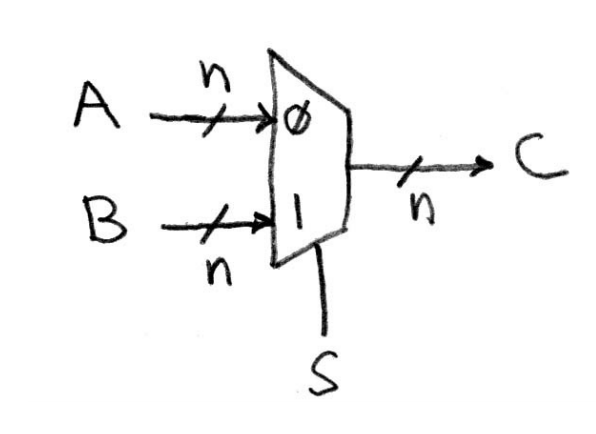
* An alternate problem can occur...
  + Clk->Q + **best case** combinational delay < Hold time...
* What happens?
  + Clk->Q + data propagates...
  + And now you don't hold the input to the flip flop long enough
* Solution:
  + Add delay on the best-case path (e.g. two inverters)



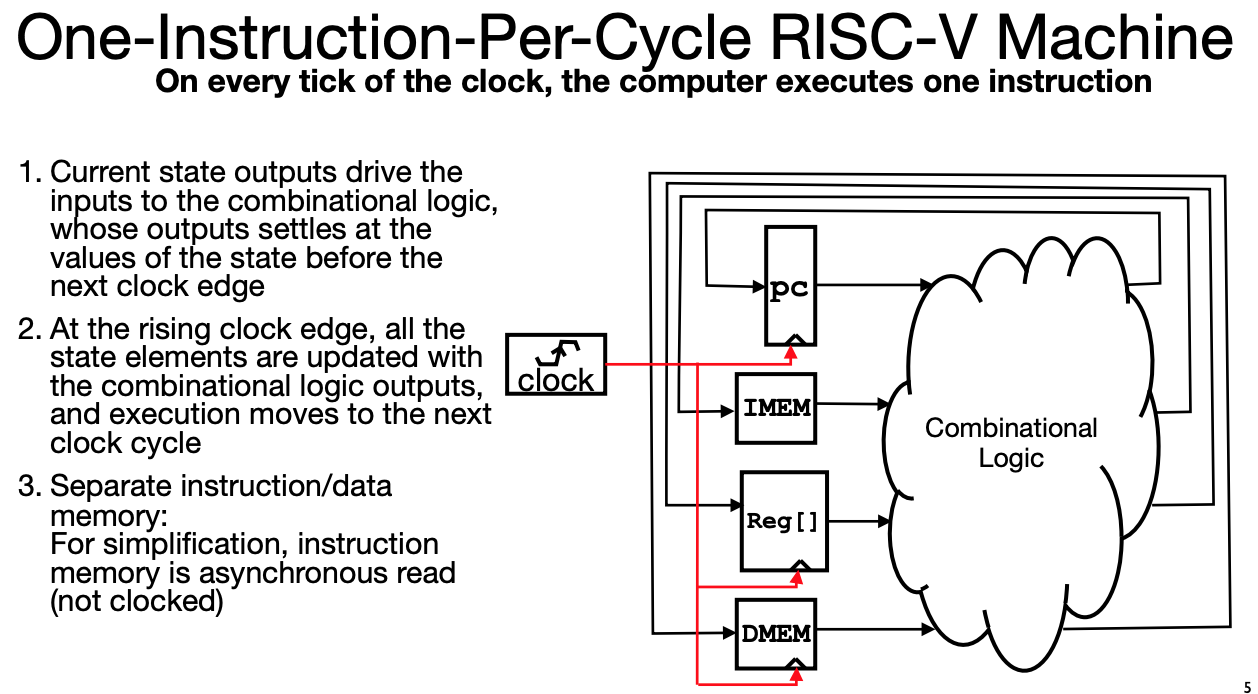
Finite State Machines (FSM):

* A convenient way to conceptualize computation over time
* We start at a state and given an input, we follow some edge to another (or the same) state
* The function can be represented with a “state transition diagram”.
* With combinational logic and registers, any FSM can be implemented in hardware.



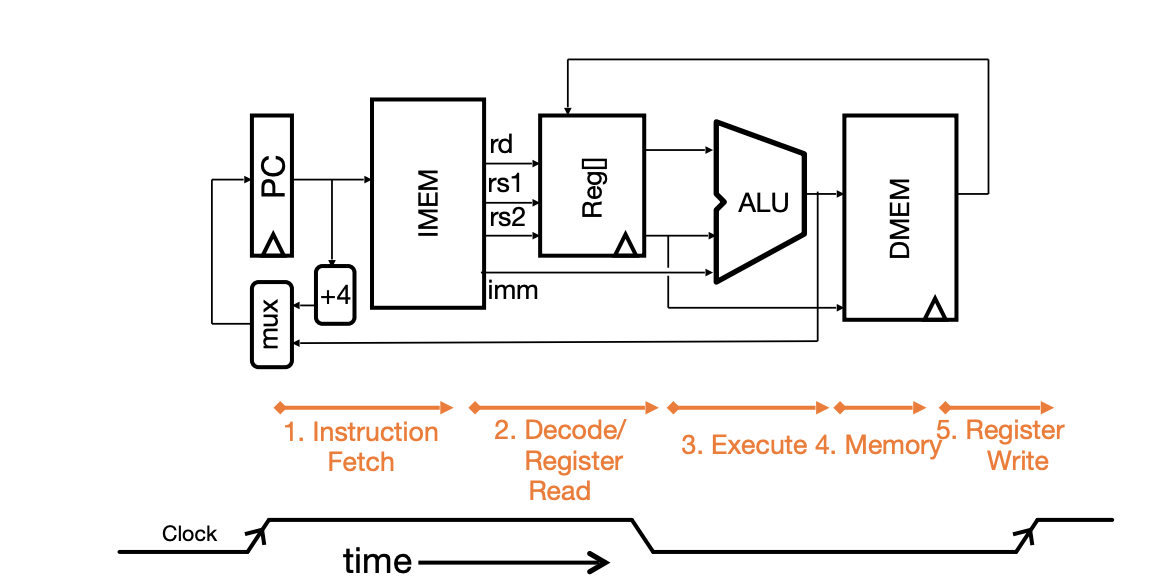
Building Standard Functional Units:

* **Data Multiplexers (Mux)**
  + If 0, output A
  + If 1, output B
  + 2-1, n-bit-wide Mux
  + Using this, we can build the Arithmetic and Logical Unit (ALU)

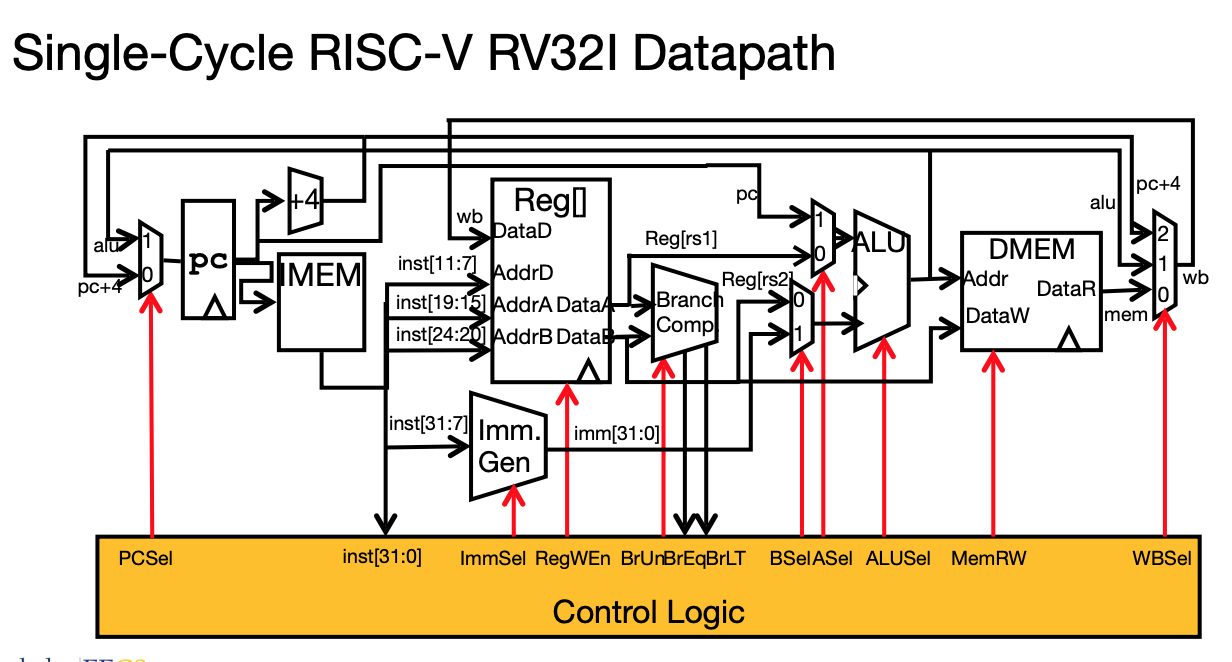
LEC 11: RISC-V Processor Data path

Each instruction reads and updates this state during execution:

* **Registers (x0, x31)**
  + First register read specified by rs1 field in instruction
  + Second register read specified by rs2 field in instruction
  + Write register (destination) specified by rd field in instruction
  + x0 is always 0 (writes to Reg[0]are ignored)
* **Program Counter**
  + Holds address of current instruction
* **Memory (MEM)** 
  + Holds both instructions & data, in one 32-bit byte-addressed memory space
  + We’ll use separate memories for instructions (IMEM) and data (DMEM)
    - Later replace these with instruction and data caches
  + Instructions are read (fetched) from instruction memory (assume IMEM read-only)
  + Load/store instructions access data memory

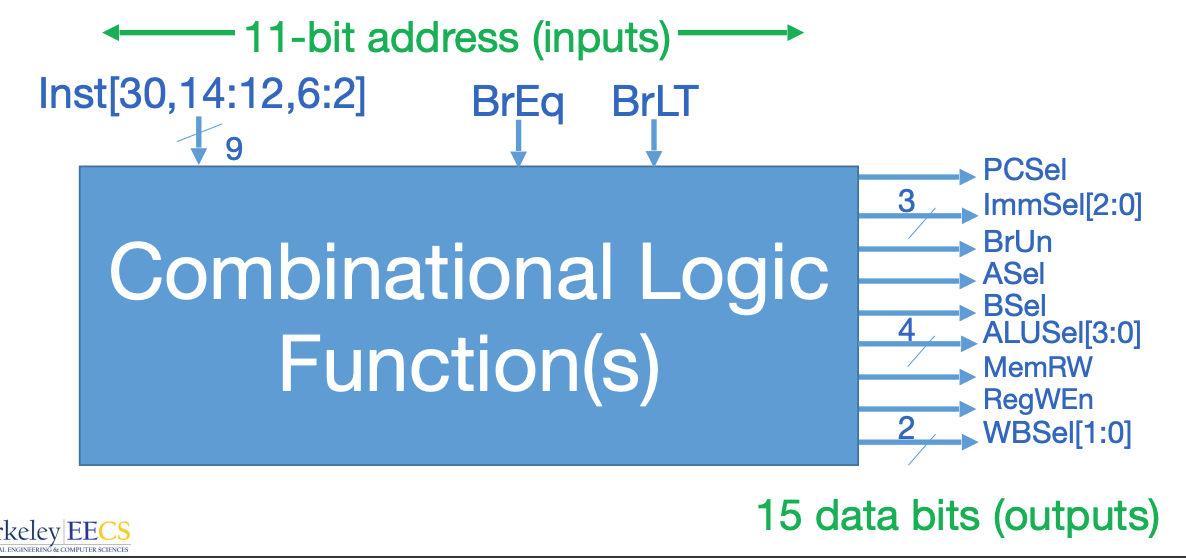


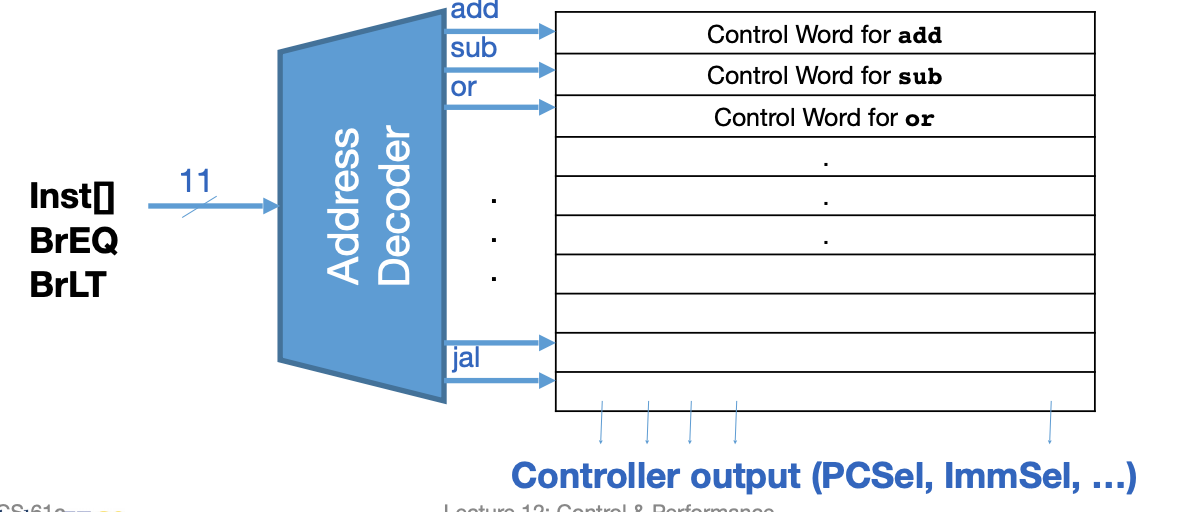
All instructions:



* Universal datapath
  + Capable of executing all RISC-V instructions in one cycle each
  + Not all units (hardware) used by all instructions
* 5 Phases of execution
  + IF, ID, EX, MEM, WB
  + Not all instructions are active in all phases
* Controller specifies how to execute instructions
  + what new instructions can be added with just most control?

Control Logic Truth Table (incomplete)

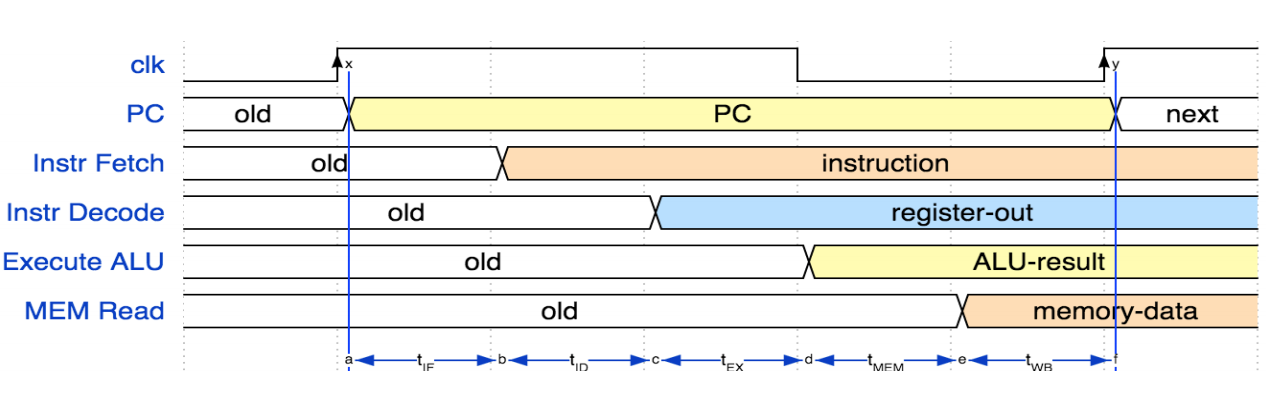




ROM:

* “Read-Only Memory”
* Regular structure
* Can be easily reprogrammed to:
  + fix errors
  + add instructions

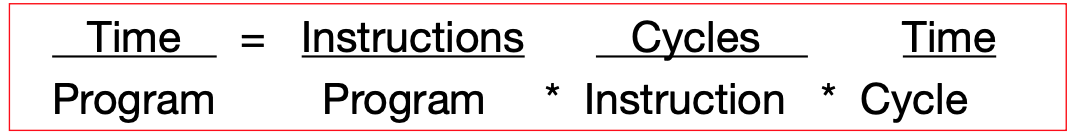
Approximate Instruction timing:



Critical Path:



Performance Measures:

* **Latency:**
  + Program execution time (time to update display)
* **Throughput:**
  + Number of server requests handled per hour
* **Energy Per Task:**
  + E.g. how many movies you can watch per battery charge or energy bill per datacenter
  + Note: Power is not a good measure, since low-power CPU might run for a long time to complete one task consuming more energy than faster computer running at higher power for a shorter time
* **“Iron Law” of Processor Performance:**

Instructions Per Program:

* Determined by
  + Task
  + Algorithm, e.g. O(N2) vs O(N)
  + Programming language
  + Compiler
  + Instruction Set Architecture (ISA)
* Input

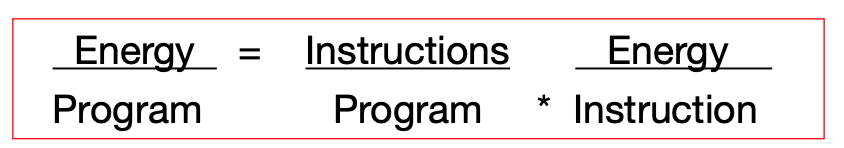
Average Clock Cycles per Instruction (CPI):

* Determined by
  + ISA (CISC versus RISC)
  + Processor implementation (or microarchitecture)
    - E.g. for “our” single-cycle RISC-V design, CPI = 1
  + Superscalar processors, CPI < 1 (next lecture)

Time per Cycle (1/Frequency):

* Determined by
  + Processor microarchitecture (determines critical path through logic gates)
  + Technology (e.g. 14nm versus 28nm)
  + Power budget (lower voltages reduce transistor speed)

Energy per Task:



Energy “Iron Law”:

* Energy efficiency (e.g., instructions/Joule) is key metric in all computing devices
* For power-constrained systems (e.g., 20MW datacenter), need better energy efficiency to get more performance at same power
* For energy-constrained systems (e.g., 1W phone), need better energy efficiency to prolong battery life 39
* Performance = Power \* Energy Efficiency

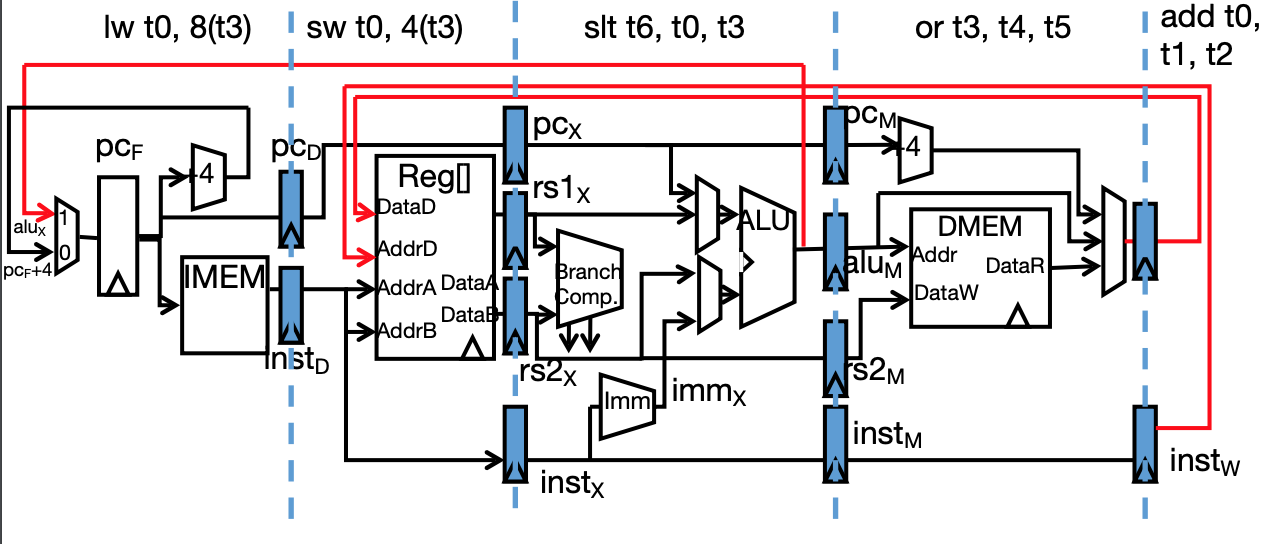
(Tasks/Second) (Joules/Second) (Tasks/Joule)

Pipelining:

* Pipelining can **improve** **throughput** but can **never improve latency**
  + Stays the same **sequentially** but things are done **simultaneously**

LEC 13: Pipelining RISC-V

Latency:

* Pipelining makes latency worse
* 2 Sources:
  + Unbalanced pipeline stages
  + Setup & clk🡪q time for the pipeline registers

Pipeline Registers:

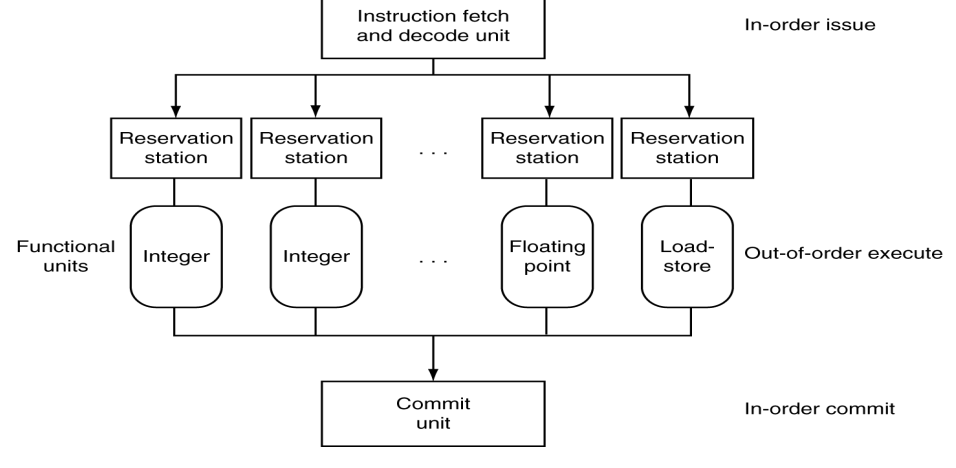
* Need registers between stages to hold information produced in previous cycle

Pipelining Hazards (conflict for use of a resource):

* **Structural hazard:**
  + **Problem**:
    - Two or more instructions in the pipeline compete for access to a single physical resource
  + **Solution 1:** 
    - Instructions take it in turns to use resource, some instructions have to stall
  + **Solution 2:** 
    - Add more hardware to machine
    - Can always solve a structural hazard by adding more hardware (caches)
* **Data hazard** 
  + Data dependency between instructions
  + Need to wait for previous instruction to complete its data read/write
  + **Solution 1: Stalling**
    - Redo the instruction
      * Reduces performance but may be required to get correct results
    - Compiler can rearrange code or insert NOPs
  + **Solution 2: Forwarding (aka Bypassing)**
    - Use result when it is computed
      * Don’t wait for it to be stored in a register
      * Requires extra connections in the datapath (detect need for forwarding)
    - Forwarding with R-Type Inst:
      * We forward with two muxes just after RS1x and RS2x pipeline registers
        + The output of the read registers
      * Select either the register, the output of the ALUm register, or the output of the MEM/WB register
    - Forwarding with lw:
      * Slot after a load is called a load delay slot
        + If that instruction uses the result of the load, then the hardware will stall for one cycle
        + Equivalent to inserting an explicit nop in the slot
        + except the latter uses more code space
        + Performance loss
      * Idea:
        + Put unrelated instruction into load delay slot • No performance loss!
* **Control hazard** 
  + Flow of execution depends on previous instruction
    - E.g. Branch and jump
  + **Solution:**
    - If branch not taken, then instructions fetched sequentially after branch are correct
    - If branch or jump taken, then need to kill insts by converting to NOPs
  + **Reducing Branch Penalties:**
    - Every taken branch in simple pipeline costs 2 dead cycles
    - To improve performance, use “branch prediction” to guess which way branch will go earlier in pipeline
    - Only flush pipeline if branch prediction was incorrect

Increasing Processor Performance:

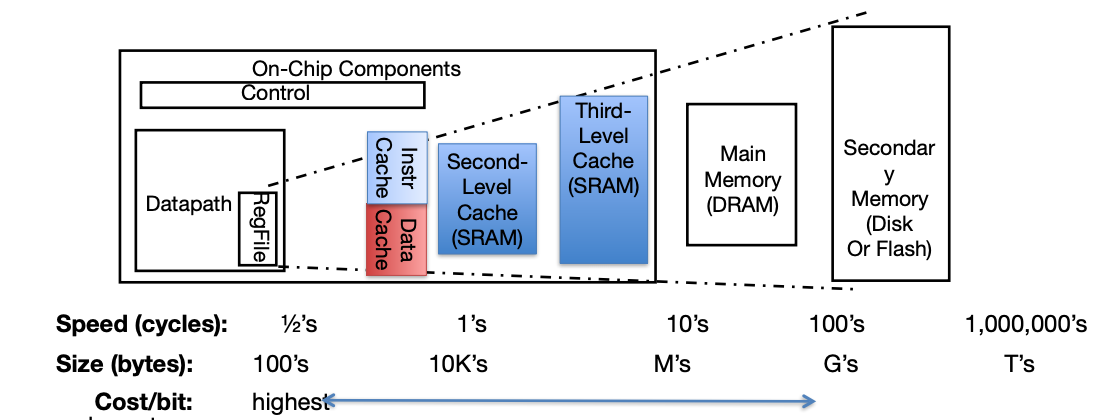
1. Clock rate
   1. Limited by technology and power dissipation
2. Pipelining
   1. “Overlap” instruction execution
   2. Deeper pipeline: 5 => 10 => 15 stages
   3. Less work per stage ! shorter clock cycle
   4. But more potential for hazards (CPI > 1)
3. Multi-issue “superscalar” processor

Superscalar Processor:

* Multiple issue “superscalar”
  + Replicate pipeline stages => multiple pipelines
  + Start multiple instructions per clock cycle
* “Out-of-Order” execution
  + Reorder instructions dynamically in hardware to reduce impact of hazards

LEC 14: Caches

Big Idea: **Memory Hierarchy**



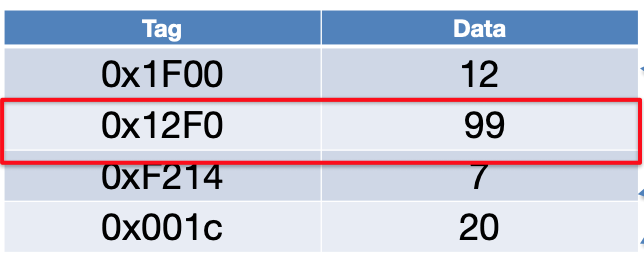
* Cache is designed to give **illusion** of speed of fastest memory with the size of the largest memory

Big Idea: **Locality**

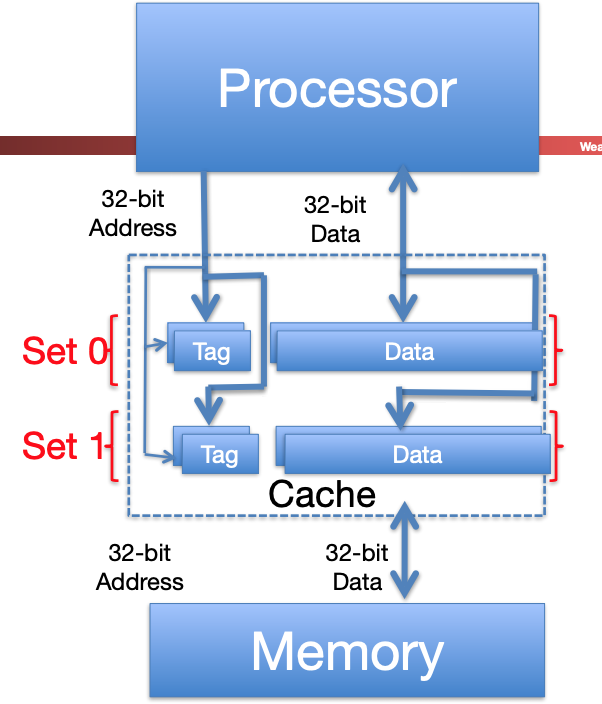
* **Principle:**
  + Programs access small portion of address space at any instant of time (**spatial** locality) and repeatedly access that portion (**temporal** locality)
* **Temporal Locality (locality in time)**
  + If a memory location is referenced, then it will tend to be referenced again soon
* **Spatial Locality (locality in space**)
  + If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
* Good for memory reference:
  + N-loop iterations
  + Stack acceses/pops
  + Data accesses (vectors)
* Bad for locality
  + Pointer chasing (i.e. linked lists, trees, etc.)

Memory Access with cache:

* Example: Processor issues address 0x12F0 to Cache, say 0x12F0 holds value 99

1. Cache checks to see if it has a copy of data at the address
   * + If finds a match (**Hit)**: cache reads 99, sends to processor
     + No match (**miss)**: cache sends address to Memory
       - Memory reads the address and sends 99 to cache
       - Cache replaces word which can store 0x12F0 with 99
       - Cache sends 99 to processor
2. Processor loads 99 into register

* **Cache Tags**
  + Need way to tell if have copy of location in memory so that can decide on hit or miss
  + On cache miss, put memory address of block as “tag” of cache block
* **Cache Replacement**
  + Must evict one block to make room for a new cached block

Hardware Cost of Cache:

* Need to compare every tag to the Processor address
* Comparators are expensive
* Optimization: use 2 “sets” of data with a total of only 2 comparators
* 1 Address bit selects which set (ex: even and odd set)
* Compare only tags from selected set
* Generalize to more sets

Processor Address Fields used by Cache Controller:

* **Block Offset**: Byte address within block
* **Set Index**: Selects which set
* **Tag**: Remaining portion of processor address (32 bits total)
* **Size of Index** = log2 (number of sets)
* **Size of Tag** = Address size – Size of Index – log2 (number of bytes/block)

One More Detail: Valid Bit

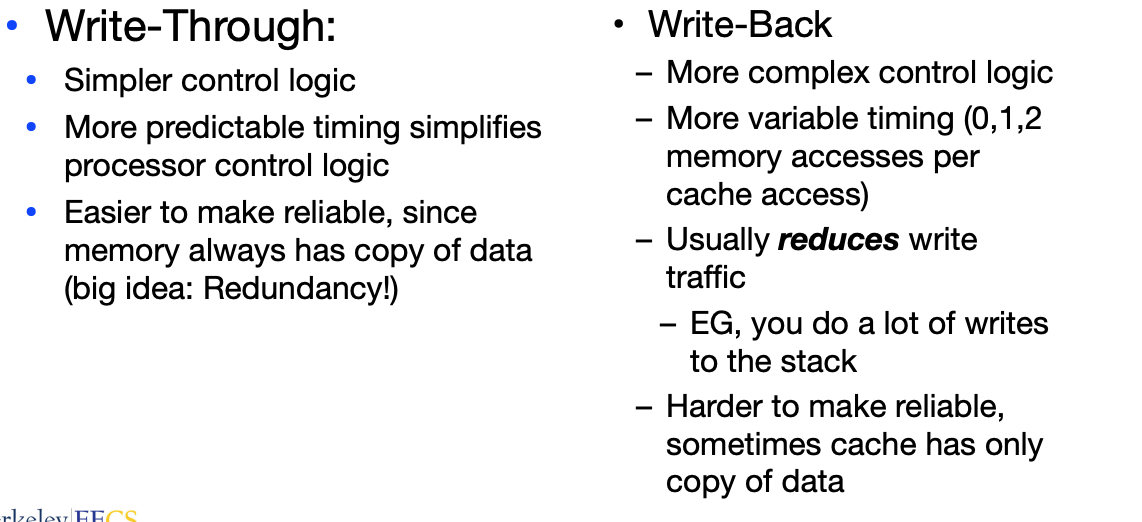
* When start a new program, cache does not have valid information for this program
* Need an indicator whether this tag entry is valid for this program
* Add a “valid bit” to the cache tag entry
  + 0 => cache miss, even if by chance, address = tag
  + 1 => cache hit, if processor address = tag

Handling Stores with Write-Through:

* Store instructions write to memory, changing values
* Need to make sure cache and memory have same values on writes: 2 policies

1. **Write-Through Cache:**
   * Write cache and write through the cache to memory
     + Every write eventually gets to memory
     + But its slow, doesn’t take advantage of temporal locality
2. **Write-Back Cache:**
   * Store/cache hit:
     + Write data in cache only and set dirty bit
     + Memory has stale value
   * Store/cache miss:
     + Read data from memory, then update and set dirty bit
     + “Write allocate” policy
     + On any miss, write back evicted block, only if dirty. Update cache with new block and clear dirty bit

Write Through vs Write back:



Common Combinations:

* Write through and no write allocate
* Write back with write allocate

Cache (Performance) Terms:

* **Hit rate:** fraction of accesses that hit in the cache
* **Miss rate**: 1 – Hit rate
* **Miss penalty:** time to replace a block from lower level in memory hierarchy to cache
* **Hit time:** time to access cache memory (including tag comparison)
* **Average Memory Access Time (AMAT)**
  + Average time to access memory considering both hits and misses in the cache
  + **AMAT = Time for a hit + Miss rate \* Miss penalty**

LEC 15: Caches #2

More Terms:

* **Cache level:** 
  + The order in the memory hierarchy:
    - L1$ is closest to the processor
  + L1 caches may only hold data (Data-cache, D$) or instructions (Instruction Cache, I$)
    - Most L2+ caches are "unified", can hold both instructions and data •
* **Cache capacity:** 
  + The total # of bytes in the cache
* **Cache line or cache block:** 
  + A single entry in the cache
* **Cache block size:** 
  + The number of bytes in each cache line
* **Number of cache lines:**
  + Cache capacity / block size:
* **Cache associativity:** 
  + The number of possible cache lines a given address may exist in.
  + Also the number of comparison operations needed to check for an element in the cache
  + **Direct mapped**:
    - A data element can only be in one possible location (N=1)
  + **N-way set associative**: A data element can be in one of N possible positions •
  + **Fully associative**: A data element can be at any location in the cache.
    - Associativity == # of lines
* **Total # of cache lines** == capacity of cache/line size
* **Total # of lines in a set** == # of cache lines / associativity
* Address is divided into |**TAG**|**INDEX**|**OFFSET**|
  + **Offset**:
    - The lowest bits of the memory address which say where data exists within the cache line.
    - It is log2(line/black size)
    - So for a cache with 64B blocks it is 6 bits
  + **Index**:
    - The portion of the address which says where in the cache an address may be stored
    - Takes log2(# of cache lines / associativity) bits
    - So for a 4 way associative cache with 512 lines it is 7 bits
  + **Tag**:
    - The portion of the address which must be stored in the cache to check if a location matches
    - # of bits of address - (# of bits for index + # of bits for offset)
    - So with 64b addresses it is 51b...
* **Eviction:** 
  + The process of removing an entry from the cache
* **Write Back:** 
  + A cache which only writes data up the hierarchy when a cache line is evicted
  + Instead set a **dirty bit** on cache entries
* **Write Through:**
  + A cache which always writes to memory
* **Write Allocate**:
  + If writing to memory **not in the cache** fetch it first
* **No Write Allocate**:
  + Just write to memory without a fetch

Cache Replacement Policies (Eviction):

* Random Replacement
* Least-Recently Used

Source of Cache Misses:

* **Compulsory (cold start, first reference):** 
  + 1st access to a block, not a lot you can do about it.
  + If running billions of instructions, compulsory misses are insignificant
* **Capacity (cache isn’t big enough):**
  + Cache cannot contain all blocks accessed by the program
    - Misses that would not occur with infinite cache
* **Conflict (collision):**
  + Multiple memory locations mapped to same cache set
  + Misses that would not occur with ideal **fully associative cache of the same size**

Improving Cache Performace:

* AMAT = Time for a hit + Miss rate \* miss penalty
  + Reduce the time to hit in the cache (smaller cache)
  + Reduce the miss rate (bigger caches, longer cache lines)
  + Reduce the miss penalty (use multiple cache levels)

Local vs. Global Miss Rates:

* Local miss rate
  + the fraction of references to one level of a cache that miss
* Global miss rate
  + the fraction of references that miss in all levels of a multilevel cache

LEC 16: More Caches

Breaking Caches: Capacity

* Up until the test exceeds the cache capacity (everything’s fine)
* But once sizeof(array) > cache size... Then things break down and you start getting misses
* Which increases the loop time (AMAT)
* So where the size breaks capacity, is the size of the cache

Breaking Caches: Spatial Locality

* Spatial locality breaks down if only a single entry in each cache line is ever accessed
* So worst-case behavior occurs when each line is only accessed in one location
* Combined with where the capacity break occurs, you now know the line-size

Un-breaking Caches: Associativity

* If your array is 2x the cache capacity... But you are striding at >= 2\*block size...
* You aren't using all the cache entries
  + So by definition, all your misses are no longer capacity misses but conflict misses!

Victim Caches:

* Conflict misses are a pain, so in addition to the main cache, have a very small (16-64 entry) **fully associative** victim cache
* Whenever we evict a cache entry, put it in the victim cache
* Now on cache misses, check the victim cache first. If it is in the victim cache, you can just reload it from there

Another Cache: Branch Predictor

* Branches and jumps are both PC relative
  + For jal we can quickly look at the instruction and compute the new location for the PC if we can predict right
* **Idea:** Branches have temporal locality
  + For loops
  + Rare conditionals (if)

Bloom Filters:

* A single-sided error set with two operations:
  1. **bloomfilter.Insert(A)**
  + Inserts A into the set
  1. **bloomfilter.Present(B)**
  + Checks if B is in the set
* **Idea:** Treat the memory as a large array of bits, and set the bits based on the hash of A

Related Cache: Return Target Location

* On RISC-V, you call a function with jal or jalr (object oriented) with the return set to ra • And you return with a jalr with the source register as ra
* So lets maintain a small stack in hardware
* **Result:** we should **always** correctly predict a function return address

Final Related Cache: Branch Target Buffer

* Function calls using jal we will never mispredict on RISC-V
* On a jalr, which writes to ra rather than x0
  + Look in a small cache for the address to predict to based on current PC
  + When evaluating the jump, set the value in this cache to the address used

Multiple Processors:

* **Reading Memory:**
  + No problem, each processor just caches the data independently
  + No issue with multi processors reading the same thing
* **Writing:**
  + Need **coherency**: Writes from 1 processor **must** be reflected in memory that other processors read after some short period of time
  + **Goal** is to guarantee if processor A writes to memory location L, **within time T,** all other processors will see the updated data
  + **Idea: Broadcasting messages**
    - Each process (or more precisely its cache) can send and receive messages
      * Requests are broadcast
      * Replies may or may not be broadcast
    - **Broadcasting writes**
      * Processor A wants to write to physical location L for the first time...
      * First do a write-allocate...
      * It then sets the dirty bit on the cache
      * And broadcasts a message that "I am writing to L"
      * All other processors which receive the message
        + Do I have address L cached?
        + If no: Do nothing
        + If yes: invalidate the entry in the cache
    - **Broadcasting Reads:**
      * If there is a miss in the upper level of the cache in the processor...
      * Broadcast a read request: "Hey, does anyone have location L?"
      * If nobody has written to this location...
        + The memory controller/common cache just does a fetch and returns it: Just like any other cache miss
      * If a processor has this location with the dirty bit set...
        + It flushes the entry (writing the value to memory) and clears the dirty bit
        + Then says new value to the requesting processor

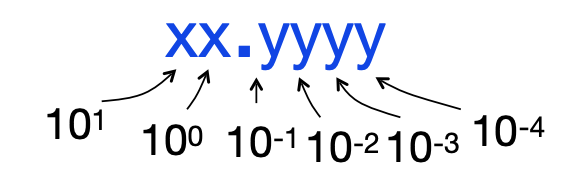
New Miss Type for Multi Procesors: Coherence

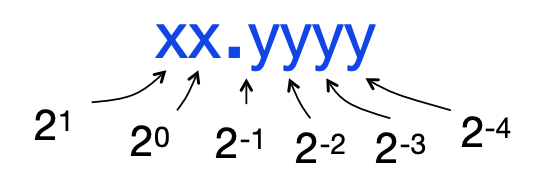
* A coherence miss occurs when two processes want to access the same data
* Coherence misses are caused only by writes, not reads

Virtual Memory:

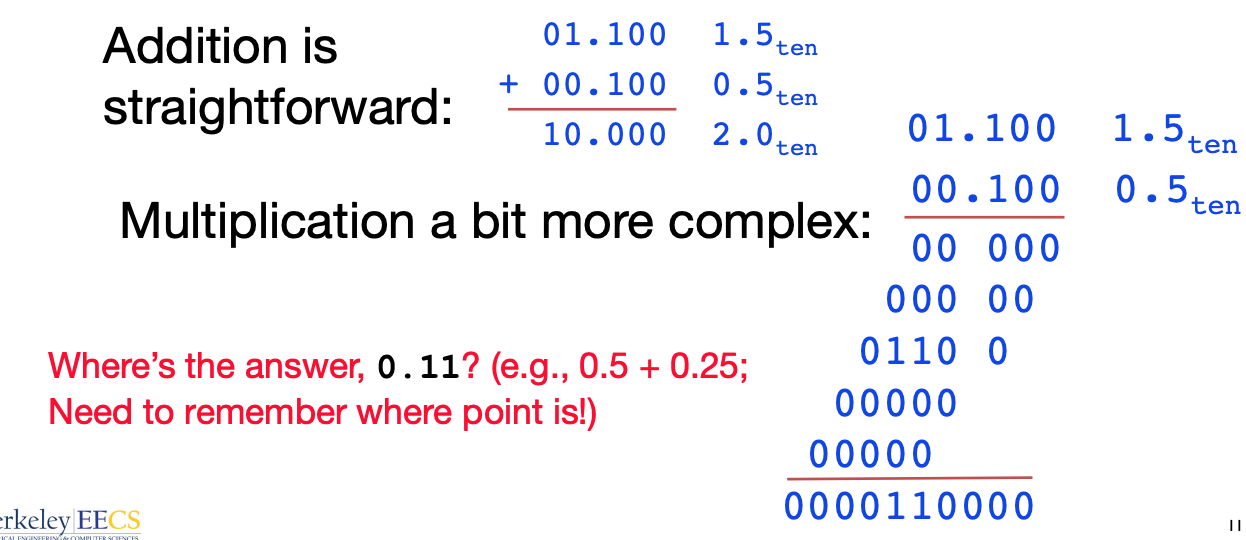
* **Idea:** 
  + Virtual memory can copy pages between RAM and disk
  + Main memory thus acts as a cache for the disk
* Properties:
  + **Associativity:** Fully Associative
  + **Replacement Policy:** Full LRU
  + **Block Size:** 4kB or more
  + **Hit time:** call it 0
  + **Miss penalty:**
    - Latency to get a block from disk: 1ms or so for an SSD (1,000,000 clock cycles)
* Implications:
  + As long as you don’t really use it, Virtual Memory is great
    - As long as your **working set** fits in physical memory, virtual memory is great at handling a little extra
  + But as soon as your working set exceeds physical memory, performance starts to drop exponentially
    - System starts **thrashing:** Repeatedly needing to copy data to and from disk

LEC 17: Performance and Floating-Point Arithmetic

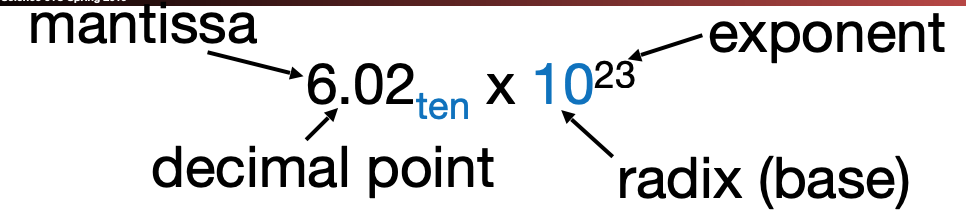
Representation of decimal (base 10) first: Binary Point like decimal point signifies boundary between integer and fractional parts

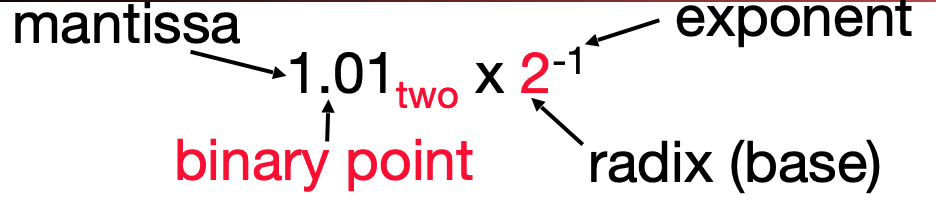


Addition and Multiplication with Fixed Point:

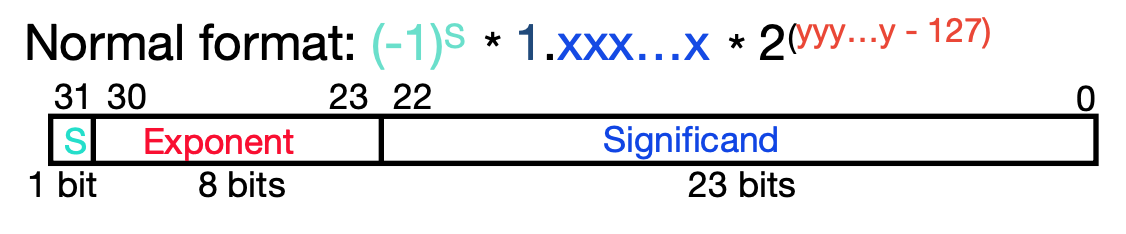


* Moving from a fixed binary point to “float” the binary point to make most effective use of limited bits
* Computer arithmetic that supports this is called **floating point** because it represents numbers where the binary point is not fixed, as it is for integers
* Variables are **float**
  + **double** for double precision

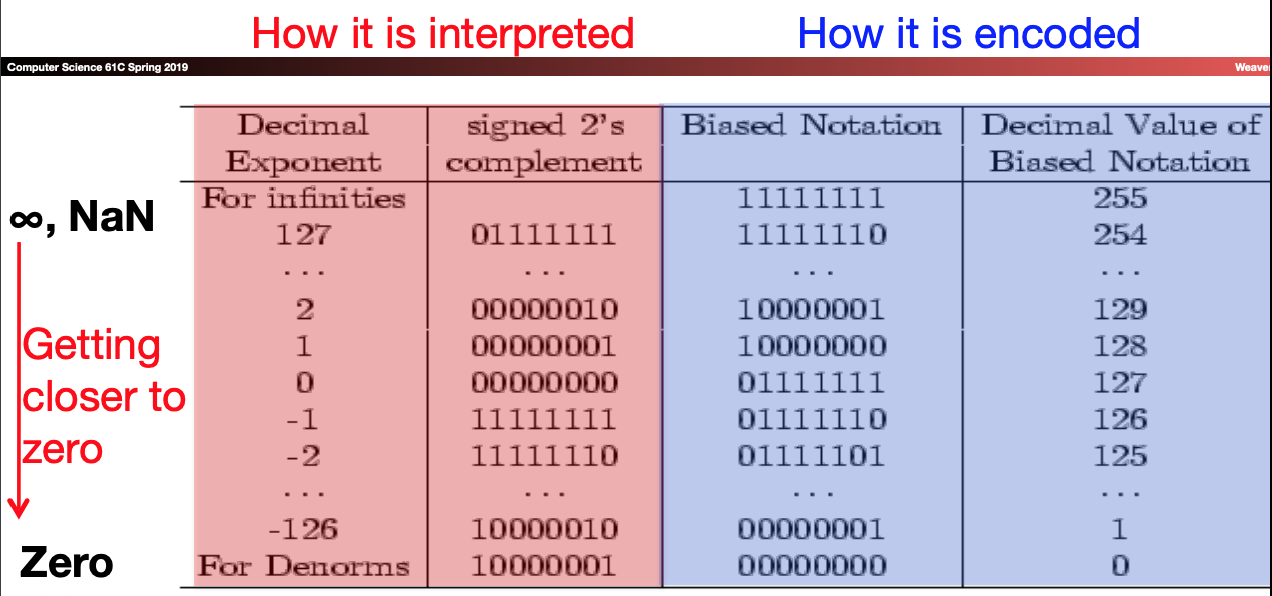
Scientific notation (decimal) Scientific notation (binary)



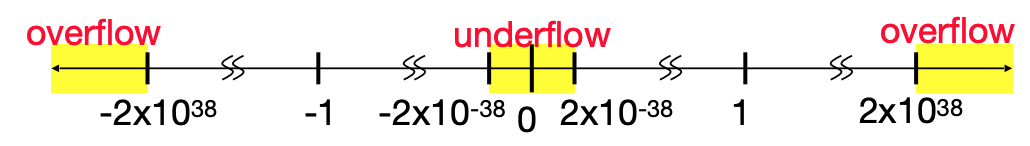
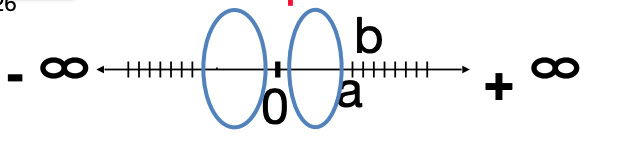
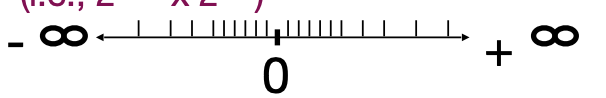
IEE 754 Floating-Point Standard:

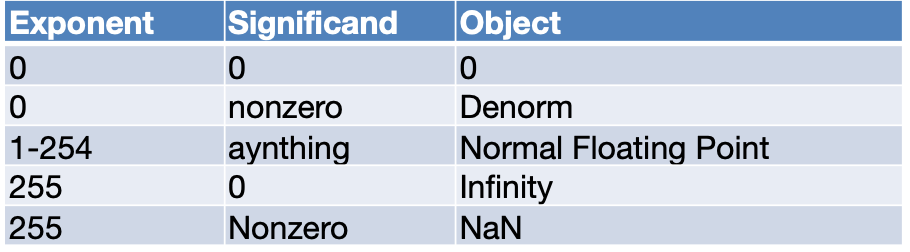
* Standard arithmetic for reals for all computers
* Keep as much precision as possible, help programmer with errors in real arithmetic
  + +∞, -∞, Not-A-Number (NaN), exponent overflow, exponent underflow, +/- zero
* For “single precision”, a 32 bit-word
  + 1 bit for **sign (s)** of floating point number
  + 8 bits for **exponent (E)**
  + 23 bits for **fraction (F)**
  + Can represent approximately numbers in range 2.0 \* 10^(-38) to 2.0 \* 10^(38)
* Uses **biased exponent** representation b/c wanted FP numbers to be used even if no FP hardware and wanted bigger (integer) exponent fields to represent bigger numbers
  + 2’s complement poses a problem because negative numbers look bigger

Bias Notation (exponent = stored value – 27)



Floating-Point Representation:

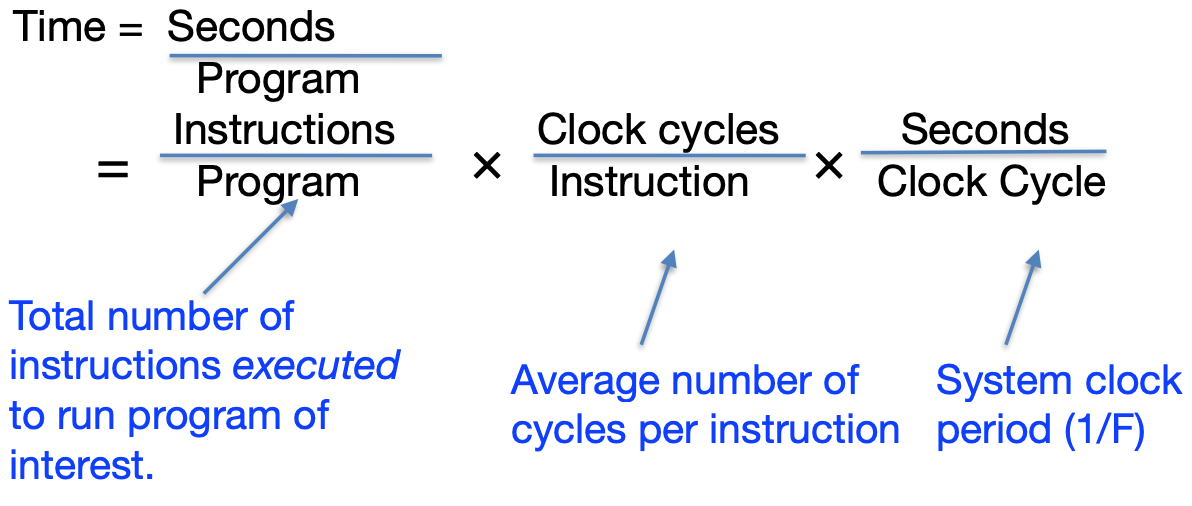
* What if result is too large?
  + **Overflow 🡪** Exponent larger than represented in 8-bit Exponent field
* What if result is too small?
  + **Underflow 🡪** Negative exponent larger than represented in 8-it exponent field
* What about bigger or smaller numbers?
  + IEEE 754 Floating-Point **Double Precision Standard (64 bits)**
    - 1 bit for **sign (s)** of floating-point number
    - 11 bits for **exponent (E)**
    - 52 bits for **fraction (F**) (get 1 extra bit of precision if leading 1 is implicit)
* **Representation for 0:**
  + **Exponent** all zeroes, **Significand** all zeroes
  + **Sign** can be both (+0, -0)
    - +0: 0 00000000 00000000000000000000000
    - -0: 1 00000000 00000000000000000000000
* **Representation for ± ∞**
  + In FP, divide by 0 should produce ± ∞, not overflow
  + Why?
    - OK to do further computations with ∞, (X/0 > Y may be valid comparison)
  + Most positive exponent reseved for ∞, Significand all **zeroes**
* **Representation for Not-a-Number (NaN)**
  + Exponent = 255, Significand nonzero
  + Why is this useful?
    - Hope NaNs help with debugging
    - Contaminate: op(NaN, X) = NaN
    - Can use the significand to identify which (**quiet NaNs** and **signaling NaNs)**
* **Representation for Denormalized number:**
  + **Problem**: there’s a gap among representable numbers around 0
    - Smallest representable positive number:
      * a = 1.0… 2 \* 2^-126 = 2^-126
    - Second smallest representable positive number:
      * b = 2^-126 + 2^-149
  + Solution:
    - **Denormalized number:** no (implied) leading 1, **implicit exponent = -126**
    - Smallest representable positive number:
      * a = 2-149 (i.e., 2-126 x 2-23)
    - Second-smallest representable positive number
      * b = 2-148 (i.e., 2-126 x 2-22)

**Numbers Summary:**

Review: Performance:

* **Latency (or response time or execution time)** 
  + Time to complete one task
* **Bandwidth (or throughput)**
  + Tasks completed per unit time

Review: CPU Latency Performance Equation



What affects each component?

