**Running Notes 451**

Combo Circuits

* Based on truth tables
* Truth tables = 1 input, 1 output

Breadboard

* Each column of 5 holes is connected
* Each row along the top is also connected

Relay

* Presence of current = 1
* Absence of current = 0
* In CS, it is not current but rather voltage
* Current = flow while voltage = pressure

Path of least resistance always counts

**AND, OR, and NOT Gates**

* Solution to how we can build an adder that scales is not all that different from programming
* There is an algorithm to build an adder, similar to an addition algorithm

**Adders**

* Adder block 3 needs Adder block 2, and so on. So each block needs the one before to finish. This is what makes time scale differently
* Different algorithms to create adders with have different running times

**Ripple Carry Adder - XOR gate and an AND gate**

* It is possible to create a logical circuit using multiple full adders to add *N*-bit numbers. Each full adder inputs a *C*in, which is the *C*out of the previous adder. This kind of adder is called a *ripple-carry adder*, since each carry bit "ripples" to the next full adder. Note that the first (and only the first) full adder may be replaced by a half adder (under the assumption that *C*in = 0).

**Homework**

#5: <https://www.quora.com/How-do-I-draw-a-NAND-gate-using-only-NOR-gates>

#6: <https://proofwiki.org/wiki/NAND_and_NOR_are_Functionally_Complete>

#7: <https://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/05-switched/20-relays/and-nand.html>

<http://www.electro-tech-online.com/threads/relay-nand-and-and-gates-circuit.144933/>

<http://www.cs.sjsu.edu/~pearce/modules/lectures/co/logic/Gates.htm>

**Big Hammer**

Size = O(n2^n) Time = O(n)

NOT gates = 2n Time = O(1)

AND ~ n2^n Time = O(log(n))

OR ~ n2^n Time = O(n)

Time - can’t call it constant time because all of the and and or gates could have several inputs. Logarithmic due to 8 inputs = 3 gate delays, 4 gate delays would allow for 16 inputs, and so on.

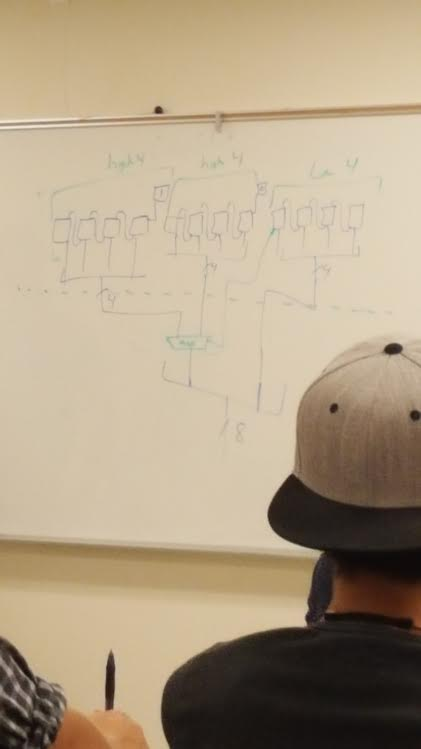
Main problem with a ripple carry adder is the “ripple-ing”. In order to leverage parallelism, what must we do to lessen the ripples?

* The faster a value is gotten, the faster the time. So instead of waiting for it to ripple down
* If you compute the formula for the carry, then calculate before getting the carry.
* Add hardware to act as if the input is a 0, and if an input is a 1. So double the hardware.

**Wednesday - 9/7/16**

Any combinatorial circuit that can be built can be made into a linear time by adding components, but now we want to make a pattern or design that will scale sub linearly.

* Ripple carry adder - want the carry ins faster. If all of the carry inputs are known right away this would greatly help the performance
* 8 bit adder with a ripple
  + What if you cut this in half, so 4 adders on each side. Then you hard code a 0 or 1 carry into the second half. Now you can work in parallel on the two 4-bit halves
  + We then add a multiplexer (mux) that will make the decision from the initial lower 4 bits on if it should send an actual 1 or 0. The decision is made before the final output is given
  + This solution is almost twice as fast as the initial ripple carry adder



* The time of this is then O(log n) because of the constant division by 2.
* Hardware wise, there are (1.5) n full adders needed if you do one pattern iteration (i.e. dividing by 2). Every time the pattern is applied the amount of full adders needed increases by 50%
* The complexity of this is actually O(n\*1.5 log^n) which translates to somewhere in between O(n) and O(n^2)

**Monday - 9/12**

How can we build a device that remembers something from a logic gate point of view?

This feedback loop can remember if it’s in state 0 or state 1.

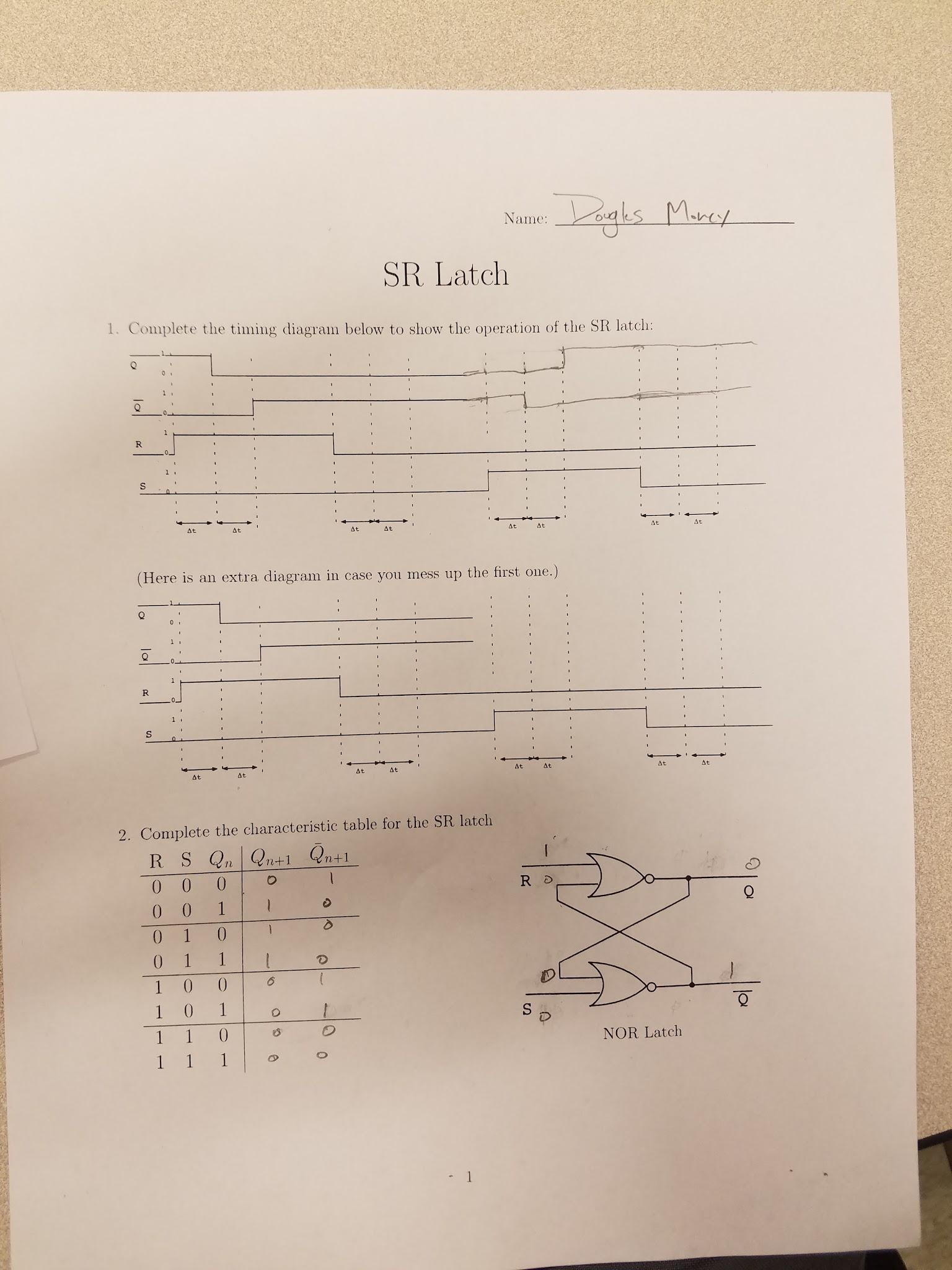
What gate should go in the middle of this sequential circuit?

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **Q** |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

So a NOR gate is needed due to the desired output. This is an SR latch.

When someone comes along and flips the R switch (so R is now 1) the next thing that will happen will be a 1 and a 0 going to the first NOR gate, so Q is output as a 0. After that, the same will have to pass for Q-bar, therefore it will take Q-bar somewhere between 0 and 2 (delta) t seconds to change.

**SR LATCH**



**Clock -** Signal that alternates between 1 and 0. When 0, don’t allow anything to change (i.e. steady input). When 1, things can change.

**Lab 3**

1. Remember how to wire up switches - every switch needs a resistor
2. For the D input clock use the push button switch rather. They switch left to right, so put 5v connection on one side and the resistor on the other side.
3. To push a 1 into your chip, hook the preset to 0. The preset and clear are active low

**Friday - 9/16/16**

* When you make an adder with a sequential circuit you can hand it 2 bits at every clock pulse in order to add bigger numbers

**Word Size -** first step in designing a CPU is to determine your word size

* Comfortable chunk of information in the machine
* If word size is 32 bit, then generally things are moved around with 32 bits at a time

Next thing to consider is the set of instructions and designing those

A **decoder** is used to enable or disable the particular registers that either need to be written or left alone.

**Monday - 9/19**

1st 6 bits in a 32 bit (or 64) instruction is the OpCode which tells the CPU what exactly it will be doing next.

* Program counter - special purpose register. Holds the address of a register being run at the given moment
* Every instruction gives a prompts a new value to be entered into the program counter

What influences how many registers you might want?

* More registers = more data storage without using RAM

**Instruction Set Design is as important as CPU design**

An accumulator also used to be used which acted as a register to hold intermediate values in addition

* This was used a long time ago, when the price of hardware was higher this was easier to do
* Using an accumulator allows the removal of the second multiplexer needed for an input to the ALU

Instruction sizes are also very important in this, which goes along with instruction set design and why it is so important.

Instructions with 0 parameters - this means implicit parameters are used instead of explicitly given parameters

**Big Picture Lesson -** Things will go away and look like they are obsolete, but these ideas come back a lot of time and play another part.

**Chapter 6 - Design Principles**

1. Regularity Promotes Simplicity
2. Smaller is Faster
3. Make the common case fast

Byte Addressable vs. Word Addressable

* Consider a 4GB address space

Byte Addressable - Each byte has its own address

Word Addressable - One address points to a row of bytes, instead of one byte.

**MIPS is byte addressable**. But this doesn’t say anything about the actual memory implementation system

**JLS is word addressable**

**Lb - load byte** (Ex. lb $t0, 0xABCD4321) nominally a 43 bit instruction, so we must break it down.

What MIPS does is something very similar to addi.

Could do this: lui $at, 0xABCD

Addi $at, $at, 4321

Lw $t0, $at

What’s the problem with doing this? Doing this for addi makes the common case fast.

The data that is coming for your program is generally 32 bits. Almost all of your loads would have to be broken up into multiple pieces, which is going to make the common case fast.

The problem is the 43 bit instruction is being broken down into 3 instructions and this causes many more steps, which slows down processing the instruction.

**What MIPS does** to speed this up is put the offset into place.

We can do even better than this, though, by doing something like this

Lui $at, 0xABCD

Lw $t0, 0x4321($at)

Lw $t1, 0x4315($at)

Lw $t2, 0x4300($at)

This is doing a load in faster steps

Sw - opcode (6), rs, rt, and imm value

One of the registers in a base register, and one is the data register.

The rs register is the base register because we are already using the data path for rt for something else. The immediate value must be used with the base register in order to complete the instruction, this is why it must go into the ALU with the data (imm value)

By setting it up this way, we have both the ALU and data memory in our critical path. If it were set up the other way (doing lb in 3 steps), the data memory and ALU would be in parallel and not both of them would be on the critical path. With this system, the CPU would run at about 1 GHz, but with the 3 instruction (more instruction) system, the CPU would run at about 1.15 GHz. So the question is, fewer faster instructions being done at one time, or a lot of slow instructions.

This is the discussion regarding CISC vs RISC instruction sets.

We can grab the 26 bits out of our instruction that are left after the opcode. They are then shifted left by 2, and turned around and put back into the program counter. Then a control wire will go from our control unit, back to the program counter. If this value is 1, then it will be executed. A shift left by 2 is essentially a multiplication of 4. This ensures that a jump will keep you within the text segment.

Pseudo Instructions - increase regularity for the CPU which then promotes simplicity.

e

This cpu’s slowest path for the clock is 925 picoseconds.

If we used the ALU and the data memory in parallel, then the total longest time would become 725 picoseconds

CISC vs RISC: RISC = simple and fast as possible, but may mean more instructions

CISC = many complex instructions (way of doing things until about the late 90’s)

**10/3/2016**

Looking at the time it takes the single cycle CPU to go through 1 cycle

Program counter takes time 30

The memory both take time 250

MUX’s take 25

Registers take 150

Register setup takes 20

Control unit takes 100 (faster than register file)

Time of a CPU

Time = n\*p (number of instructions \* time to run each)

Number of instructions means the number issued to the CPU, not the exact number in code. The number actually executing.

**10/10/2016**

* Half of our instructions are sitting around doing nothing
* The more we keep the transistors busy, the more work that can be done with the same amount of hardware

One idea:

* Clock doesn’t have to tick at a fixed rate
  + This isn’t how machines are built, clocks just cannot tick at different rates

What if we let instructions take more than one cycle? We have 3 different adders in our CPU, this is because the branch instruction needs 3 adders.

A multi-cycle CPU with micro operations, the CPU can be used to do multi things that you want to happen at different times. You’re not stuck doing things in a certain order.

On the other hand, a CISC style, so taking say 10 micro instructions and pushing them into 1 instruction, makes many changes. You’re not having to do some of the overhead, such as increment the PC every time, etc.

The other things is variable length instruction and multiple addressing modes

Variable length - hard to handle in a multi cycle in a MIPS style processor, but these make much more sense in something like Intel architecture

Addressing modes - also not very nice to be done in MIPS, but again can be handled much easier in other modes such as Intel architecture

For a complex, multi-cycle processor, a micro code is used. If opcode ‘8’ comes in, then you go to line ‘8’ in the micro code lookup table for the appropriate info. Or

Microcode is reprogrammable code that turns the Control Unit into almost a mini CPU itself. The microcode is what truly handles the controller.

**Conclusion**

Multi-cycle allow for more programmability.

It’s not perfect, though. A lot of things are working against this.

1. Doesn’t do a perfect job on the waste in a single clock cycle.
   1. I.e. for loading the op into instruction memory, the rest of the CPU is sitting idle while this happens
2. The amount of time it takes to process different pieces are not the same, and your clock cycle must be as slow as your slowest instruction. Ex: Instruction memory takes 250 units, while ALU only takes 200 units. Then there is a waste of 50 units during the ALU processing cycle.
3. Multi Cycle timing formula: t = n \* p \* CPI
   1. Also must take into effect the mix of instructions, so the CPI is a weighted average
   2. Say lw is only used 25% of the time, sw 10%, etc. Then this is not just a basic average

Best your clock period can do is: **(P/R) + 50 (this is because we added the register file to the critical path)**

Branch instruction in a pipeline:control hazard. What are we supposed to be running? This is a hazard caused by a change in flow control.

Third hazard: Structural Hazard (not very important) not double booking your hardware. There are an issue, but it is solvable by throwing more hardware at it.

How long does it take an n instruction program to run (in cycles) in a k pipelining program?

Once the pipeline is full, every time the clock ticks another instruction will finish, which this takes n cycles. It takes k - 1 cycles to load up the pipeline. Therefore, it works out to be n + (k-1). THIS IS ONLY IF THE PIPELINE STAYS FULL.

This describes the number of cycles taken with a full pipeline. When doing actually performance calculations, leave the k-1 off. There are billions and billions of instructions. CPI here is on average, how many stalls is this responsible for.

**10/14/2016 - SINGLE CYCLE and PIPELINED DATAPATH**

**Forwarding**

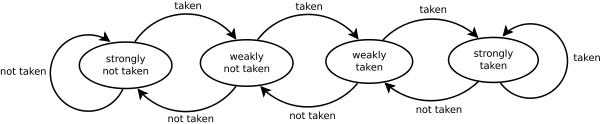
* Memory happens later in the cycle of the CPU

**10/19/2016**

* BTB (Branch Table Buffer) looks up the prediction if the branch is taken or not.
  + Full 32 bit value
  + BTB runs as the same time as the inst. Mem.
  + Then the inst. Mem does checks if the op == beq
  + If it does, then it will do the predicted branch
  + If not, it will add 4 to the PC and continue
  + A hash table is used for the BTB (kind of)
    - Needs to hash a 32 bit key into a 12 bit slot
    - Throws away 20 bits that aren’t wanted
    - High bits do not change much, so the mod takes/keeps the low bits

Using 2 bits to predict

* Must watch for patterns within bits
* 1 pattern that might be looked for would be “strongly taken” → leads to being taken
* The opposite would be “strongly not taken” → leads to being not taken
* Weakly taken → still will lead to a taken. Then if it is taken, our state changes back to strongly taken
* The same can be configured for weakly not taken and strongly not taken



**Super Pipelining**

* Software scheduled - the compiler is in charge of doing the scheduling
  + VLIW - very long instruction word
  + Simpler - faster, cheaper, smaller
  + Complex compiler
    - You’d need a different exe for every different VLIW machine
* Hardware scheduled - Control unit
  + Superscalar
  + Tune to specific application
  + Single binary

Software - you can’t predict a branch, etc.

Software is more static in its reaction, while hardware is more dynamic on the fly.

**Cache hits vs. misses**

Again, hardware can react here a little bit quicker and more dynamically, while software cannot.

Out of order issue - Hardware reorders the code on the fly to optimize the code, or to lessen the chance of hazards

**WAR data hazards** - sometimes called “fake” or “anti dependency”

* This is because you can change the register that is being written into to a different register
* This then completely takes away the problem of using the same register for a write and read
* Only have this if you have out of order issue
* Most CPU’s don’t allow out of order completion
* Other big mess with out of order completion - what if you raise an exception with an instruction that you moved something in front of?

**Scoreboarding and/or Tomasulo algorithms -** are used to solve this problem

* May be a good idea to look at for end of semester pres.

**Memory**

**Cache**

* Blocks come and go in groups
* Bytes are mapped in round robin
* Things that live next to each other in memory live next to each other in cache
* Every spot in the cache has (mem. Size / cache size) possibilities that could reside there.
* Things that are at least a cache size apart may conflict

Cold miss - cache miss because the cache just hasn’t been set up yet

Conflict miss -