ROM and Assembly

Yeah so we actually made our own programming language

Akilesh Praveen — CMSC398E

UMD

April 16, 2020

Agenda

1. Announcements
   - Projects 5, 6, and 7
2. ROM
   - The Problem We Want to Solve
   - Our Application of ROM
   - Structure of our ROM
   - Interpretation of our Programs

Announcements

Projects 5, 6, 7

- Projects 5, 6, and 7 are now released on Piazza
- Relevant instructional material is/will be linked
- They can be done in **any order**, but I would suggest doing them in order (5, then 6, then 7)
- The next 3 lectures (including this one) are designed to help you understand the projects in that order, so stay tuned for that

Intro + Background

Intro

- We've built the ALU; the brains of the operation
- Now we need a few more things to take this from just a calculator circuit to an actual computer
  - Ways to **store** programs
  - Ways to **interpret** those programs
  - Ways to **execute** those programs
  - Ways to **store data** for those programs while they're executing
- We're going to use the digital logic circuit theory to build circuits to address all of these! (Projects 5, 6, and 7)

Intro

- Ways to **store** programs - **ROM** *(Project 5)*
- Ways to **interpret** those programs - **389E Assembly** *(Project 5)*
- Ways to **execute** those programs - **Program Counter** *(Project 6)*
- Ways to **store data** for those programs while they're executing - **RAM** *(Project 7)*
- Today, we'll be talking about the first two points- **storing and interpreting**.

The Problem We Want to Solve

The Problem We Want to Solve

- Before we architect a solution, let's understand the problem we need to solve.
- We need a **way to store our programs**
- We need a **way to interpret our programs**
- Let's start with some standard questions to help guide us there
  - How exactly will we represent our programs?
  - How much space will we need?
- Once we figure out how we'll represent our programs, we can easily develop constraints in order to store them. Once they're stored, we can tackle how we'd help our computer to interpret them.

The Problem We Want to Solve

- How will we represent our programs?
  - So far, we've designed `ADD`, `SUB`, `MULT`, and other 3-bit operations. Where else have we seen basic operations like this? *(Think back to 216!)*
  - Let's take a page out of the first programmers' books and say that we're going to write some stripped down version of assembly.
  - For this project, we will be working with a 14-bit version of Assembly created by the Divine for use in CMSC389E
  - In other words, we need **14 bits** to be able to store each instruction in CMSC389E Assembly

The Problem We Want to Solve

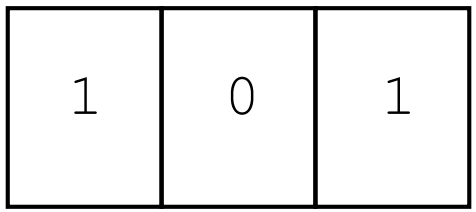
- How much space will we need?
  - That's a problem. We only have a 3-bit computer, yet our 'Assembly' will definitely take more than just 3 bits to store. (14)
  - We're going to use a sneaky workaround to get around this issue
  - Most computers store their programs in writable memory, so that they can be rewritten by the CPU, but we can drop that functionality for now
  - We need a way to store data larger than 3 bits so that we can fully realize our Assembly dreams. However, our computer can only manipulate 3 bits at once.
  - **What if** we didn't need to be able to manipulate those bits? Would that change our size constraint?

So then, Why ROM?

Why ROM?

- ROM is looking like a pretty good candidate for this
- **Question**: Why is ROM the optimal choice for this?
- Remember: our computer has a 3-bit architecture
- We didn't pick read-only because it was optimal, we picked it because it's the *easiest to build*
- There isn't much instruction you would be able to store in 3-bit sized memory!

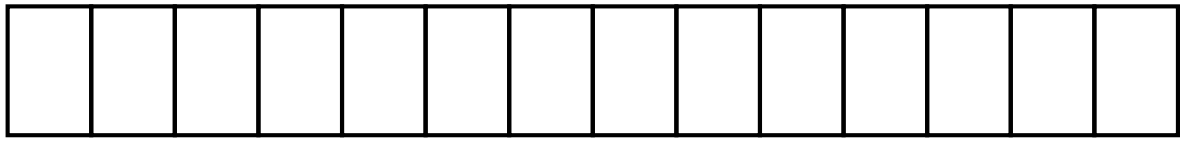
Example:



The Big Idea

- **BIG IDEA:** Our computer can manipulate only 3 bits at a time, and we want more than that for our programs. So, let's make it so that our computer won't have to ever manipulate the bits in our programs!
  - **PRO:** We can store a lot more data!
  - **CON:** We'll have to manually edit our 'computer' to change the programs
- This is no big problem for us though, as all we have to do is mess with a little redstone here and there if we want to 'edit' our programs.

Design: Final Thoughts

- In order to represent an instruction, we want to keep it as simple as possible.
- Remember, in order to **store** our Assembly-based instructions, we need 14 bits for each instruction.
  14 bits = one line of instruction
- 
- **Conclusion:** We need individual lines of read-only memory, with 14 bits in each line. This will allow us to properly **store** the programs for our computer.

Is space the reason why people use ROM?

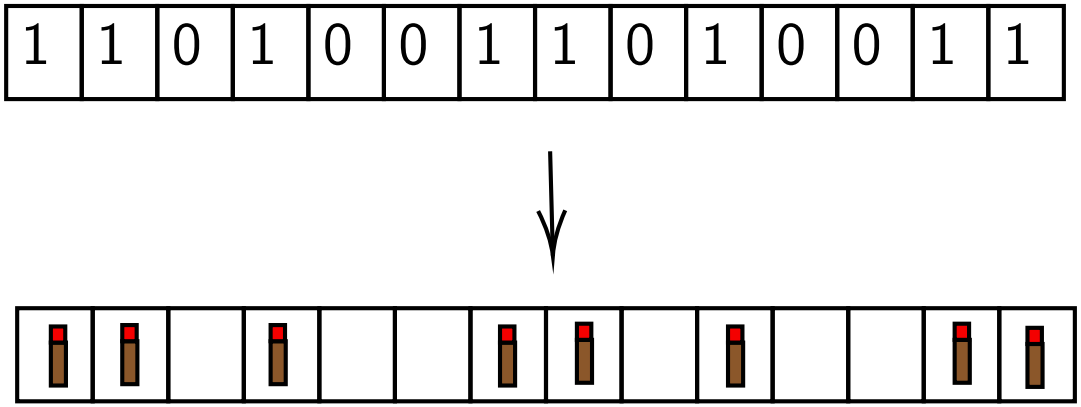
- Absolutely not.
- Most functional computers don't have this (absurdly small) space requirement, and are perfectly able to store programs in ROM and RAM alike, as their systems are capable of manipulating more than 3 bits at a time.
- However, this doesn't mean ROM is totally useless.
- Some programs are better off staying in ROM rather than RAM
- **Example:** The reason your computer still turns on after you wipe the disks is because the BIOS is (in most cases) a program that's been stored in ROM on your hardware.

The Structure of Our ROM

Storage In Minecraft

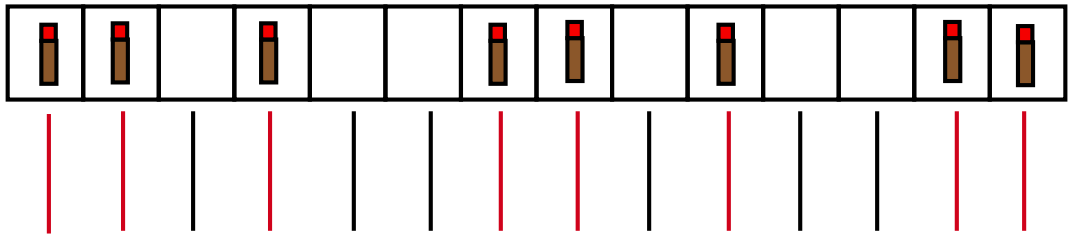
ROM in Minecraft

- Now that our memory only has to be read-only, we're set. Remember that simple read-only memory circuit we looked at? This will be even easier.
- By eliminating the need to **set** and **reset** our memory, we've reduced it down to a highly simple concept.
- In other words, a 1 can be as simple as a redstone torch being in a spot, and a 0 can be as simple as a redstone torch not being in that spot.



ROM in Minecraft

- We can figure out what each line is saying by running some redstone wire under these blocks, in order to pick up the output of the torches.

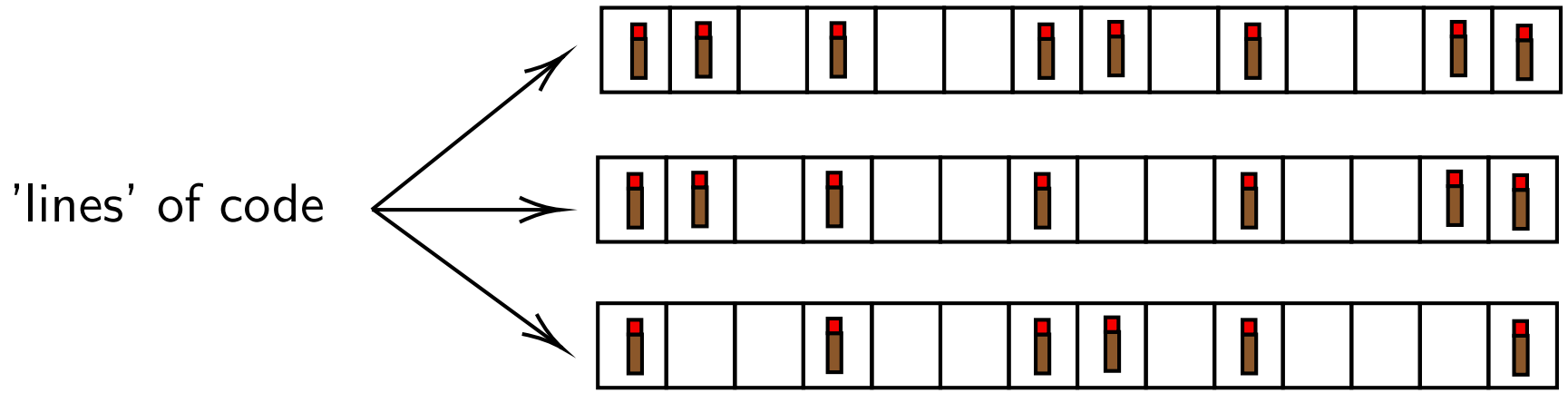


ROM in Minecraft

- Now we have an idea of how to write a 'line' of our CMSC389E assembly
- Forget about the actual rules of the assembly for now, let's just be pleased with the fact that we've figured out how to represent a line of 0's and 1's within Minecraft
- Now here's another question. This is just one line. How can we represent different lines?

ROM in Minecraft

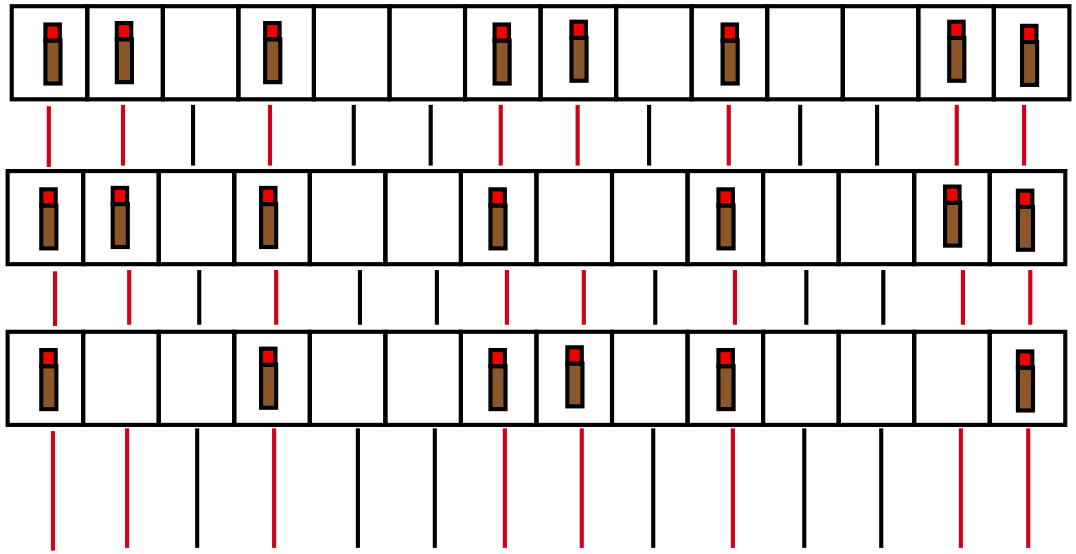
- Now we have an idea of how to write a 'line' of our CMSC389E assembly
- Forget about the actual rules of the assembly for now, let's just be pleased with the fact that we've figured out how to represent a line of 0's and 1's within Minecraft
- Now here's another question. This is just one line. How can we represent different lines?
- **Answer: It's quite simple. We make more 'lines'**



Interpretation in Minecraft

ROM in Minecraft

- Now that we can **store** multiple lines, we have a new problem
- Let's say we wanted to interpret this code, line by line. **How can we detect what each 'line' is telling us to do?**



- We can see right away that simply running wires underneath every line, using a **bus**, won't work at all. We'll just get all the 'on' bits at once!

ROM in Minecraft

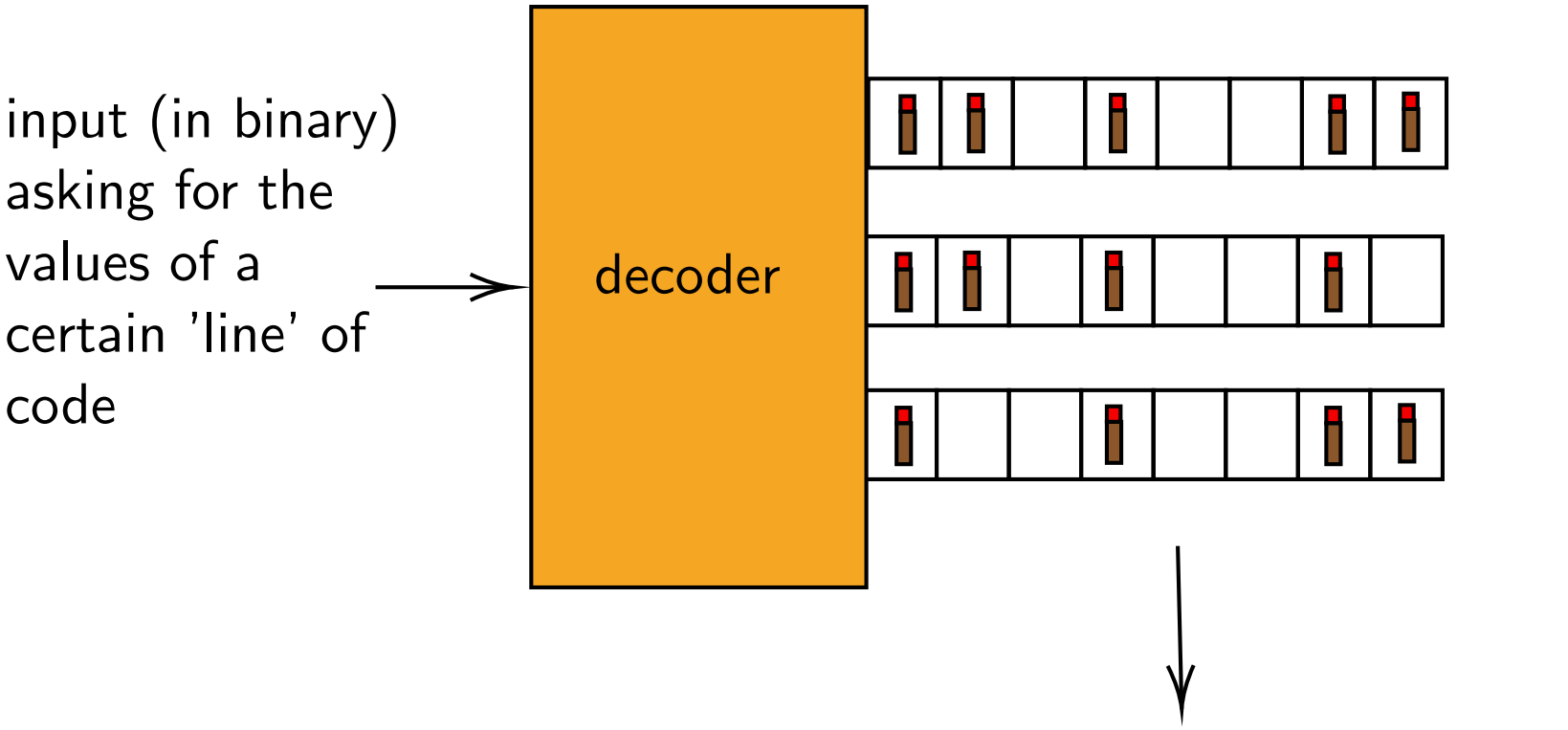
- We need a way to get the data from one of these lines at a time. Is this reminding you of a circuit you've built before?

- Think about where else you've seen long parallel rows of blocks with torches!

ROM in Minecraft

- That's right- you can use a decoder!
- Specifically, we want a binary to decimal decoder.
- We'll explain why in a minute.

ROM in Minecraft



ROM in Minecraft

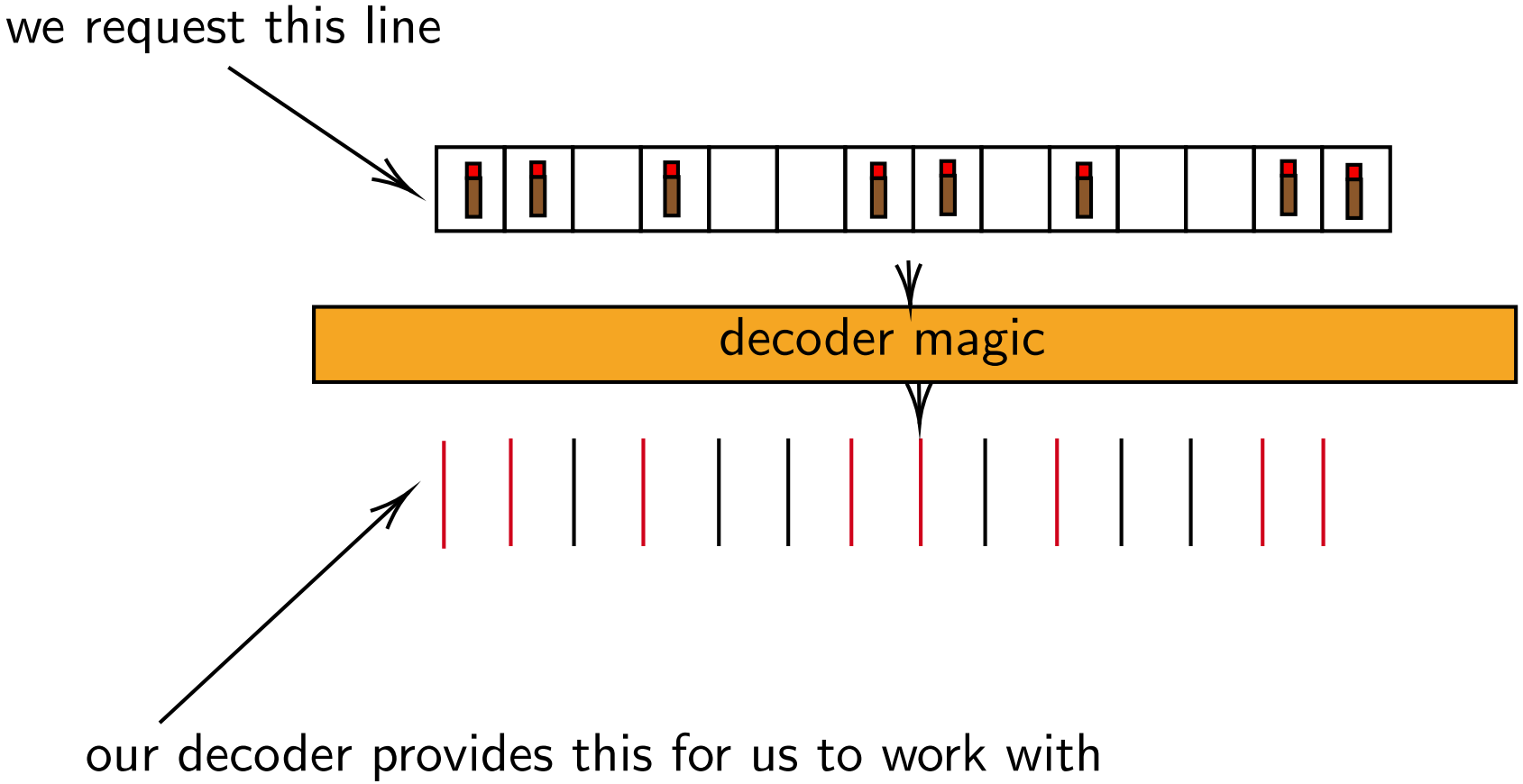
- Why a binary to decimal decoder, though?
  - You don't know this yet, but it's highly convenient for us to request the line number in **binary**.
  - Our circuit basically says: Here's the line number we're currently at (in binary). Can you get me the data stored in that line of code, please?

Interpreting The 'Assembly'

- Now, we can safely say that we know how to do two things.
  - First, we can store multiple lines of 'CMSC389E Assembly Code' using redstone torches and blocks
  - Second, we can use a binary to decimal decoder to, given a line number, produce the 'line' of code that is stored in our ROM.
- This is excellent news, because now all we need is to **interpret** those lines of code, and we're home free.

Interpreting The 'Assembly'

- Here's what we've got so far. We ask for a line of code, and we get **14 parallel signals**, representing 14 bits of 389E Assembly



Interpreting The 'Assembly'

- Okay, so how will we actually 'interpret' these 14 parallel signals of CMSC389E Assembly?
- In other words, what will we actually *do* with the signals we have?
- That is where our journey ends for now. (The answer lies in the project specification for P5)
- I'll talk more about the actual work you have to do to interpret the Assembly lines using an actual circuit once we finish the rest of the pieces of the puzzle (Program Counter + Registers) as well.