RAM

Really Awesome Memory

Akilesh Praveen — CMSC398E

UMD

April 29, 2020

Agenda

1. Announcements
   - Projects 5, 6, and 7
2. Intro + Background
3. RAM as a Concept
4. RAM in Minecraft
5. Implementation Details
6. Final Project
7. Demo

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)
- We already did a lecture on Project 5 and 6, today we'll be talking about **Project 7**

Intro

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 ways to store data for these programs, using registers of **RAM**.

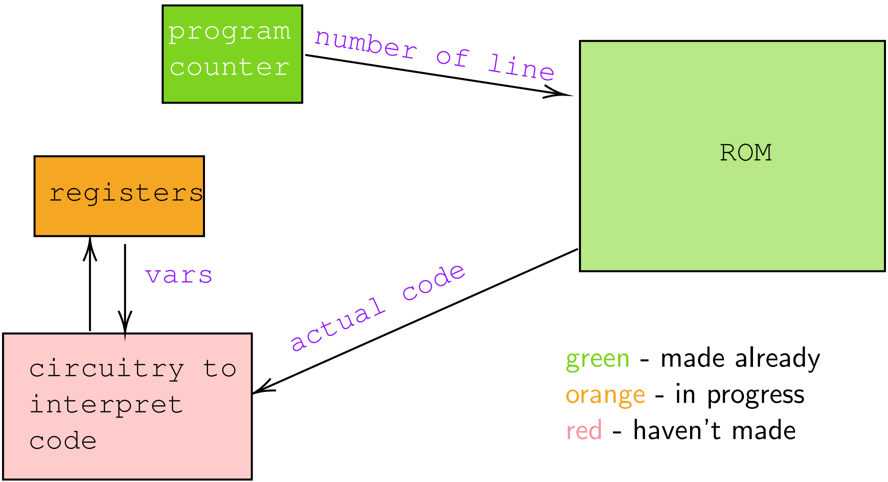
Approach

- Let's figure out our approach to this.
- Right now, we have circuits in place to iterate through the lines of code we're written and provide us with the exact instruction for every line
- Now, while our interpreter circuit (which we have not yet created) is actually executing that code, where will it store the values it calculates?
- We need **fast-access**, **modifiable** memory that we can use to store the values that we calculate.

Approach

- In other words, we need **registers** for our computer.
  - Aside: What are registers exactly? Think back to 216 and Assembly
- At the end of the day, these requirements are pretty much filled out by RAM
- First, we're going to look at the concept of RAM as a whole, and figure out why it's the perfect choice for us here
- Then, we'll talk about implementation in Minecraft

Overview



green - made already
orange - in progress
red - haven't made

RAM as a Concept

What is RAM?

- Known as **Random Access Memory**
- The quickest memory in the computer that we can pull from and write to
    - *What a coincidence, that's just what we need!*
- As such, this is the computer architecture structure that we've decided that our programs will be reading and writing to, reliably quickly, *while they execute*

What is RAM?

- Keep in mind that we're not just making this stuff up, we're following a general recipe (more on this in the final lecture!)
- Also, Keep in mind that for our case, *RAM* and *Registers* will be used interchangeably
- So, what do the computer gods say about how we should build our RAM?

What is RAM?

- **Volatility:** RAM is only useful when the circuit is powered on, and will lose all the data it's storing when the computer powers off. (For better or for worse)
- **Access Equality:** It takes the same amount of time to access any address of RAM

What is RAM?

- **Volatility:** RAM is only useful when the circuit is powered on, and will lose all the data it's storing when the computer powers off. (For better or for worse)
- **Access Equality:** It takes the same amount of time to access any address of RAM
  - How do we achieve this access equality?

What is RAM?

- **Volatility:** RAM is only useful when the circuit is powered on, and will lose all the data it's storing when the computer powers off. (For better or for worse)
- **Access Equality:** It takes the same amount of time to access any address of RAM
  - How do we achieve this access equality?
  - By leveraging a **mux** + **demux** combination, we can efficiently access one of hundreds, thousands, or more memory addresses in a cluster of RAM

Aside: Types of RAM (computer world stuff)

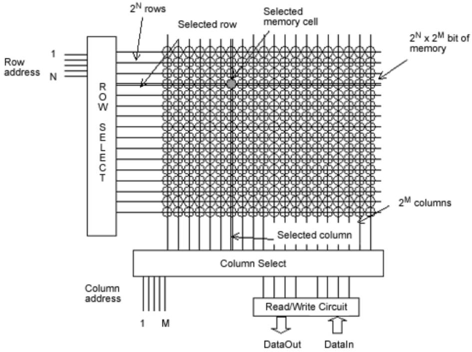
- There are two types of RAM in the computer world: **SRAM** and **DRAM**
  - Static RAM and Dynamic RAM
  - Static RAM is just circuits, whereas Dynamic RAM is circuits plus a capacitor
- SRAM is generally faster than DRAM, which is why it is mainly employed in the cache
- DRAM takes up less space, though, (seems counterintuitive, but it's true)
- As such, we use DRAM in the actual RAM chips on our computers (DDR4, DDR5, etc)

Aside: Types of RAM (computer world stuff)

- Here in CMSC389E, we don't have to worry about silly things like power consumption or storage bandwidth, because our computer is about as smart as a Saguaro Cactus
- As such, we don't need to worry about differentiating between RAM architectures or even building a cache
- Instead, we just need a reliable way for the programs we're executing to be able to read and write to memory while they're running

RAM Diagram

- Here's how RAM would look in a computer
- As we go over this, try and relate these components to digital logic components you've learned about already
  - (hint: you know everything you need to know to build this)



RAM in Minecraft

What makes RAM different for us?

- Previously on CMSC389E, you worked with ROM, which was *read-only*
- You've also seen the background for basic latches
    - Remember, latches are just augmented flip-flops. Look back to the 'memory' lecture if you need a refresher.
- We also talked about using a form of latch-based storage to keep track of the numbers we were counting for our program counter

What makes RAM different for us?

- Think of this next problem we're tackling as a further exploration of consistent value storage, as a beautiful combination of the last two projects you worked on
- We're going to store data in a similar way to what you've seen in the ROM (using a neat decoder-based technique to **read** from it)
- We're going to use latches in an intelligent way (similar to what you saw in the program counter project) in order to **write** data

What makes RAM different for us?

- Before, you essentially created a singular 'register' that could be added to and subtracted from every cycle. Now, our goal is to do **two things**
- Remove any extraneous bits from that logical circuit
- Cut that down to a more compact, modular form (we want to put a bunch of RAM together!)

So What do We Need?

- Let's synthesize this information into an actionable set of requirements
- We need a circuit that **stores data** for us, that we can easily **write to** and **read from**

So What do We Need?

- Let's synthesize this information into an actionable set of requirements
- We need a circuit that **stores data** for us, that we can easily **write to** and **read from**
- We need to be able to access all the **numerous** addresses in that circuit as **efficiently as possible**

So What do We Need?

- Let's synthesize this information into an actionable set of requirements
- We need a circuit that **stores data** for us, that we can easily **write to** and **read from**
- We need to be able to access all the **numerous** addresses in that circuit as **efficiently as possible**
  - Optimally, we want to request and address in binary, and be able to perform a R/W operation on that particular register

So What do We Need?

- Let's synthesize this information into an actionable set of requirements
- We need a circuit that **stores data** for us, that we can easily **write to** and **read from**
- We need to be able to access all the **numerous** addresses in that circuit as **efficiently as possible**
    - Optimally, we want to request and address in binary, and be able to perform a R/W operation on that particular register
- Take a moment to think about which components we'll need

So What do We Need?

- Let's list out the circuits that we've talked about in order to get this done.

latches/flip flops

logic gates

demultiplexer

multiplexer

adder

encoder

decoder

multiplier

So What do We Need?

- Here are some circuits that come to mind

latches/flip flops

logic gates

demultiplexer

multiplexer

adder

encoder

decoder

multiplier

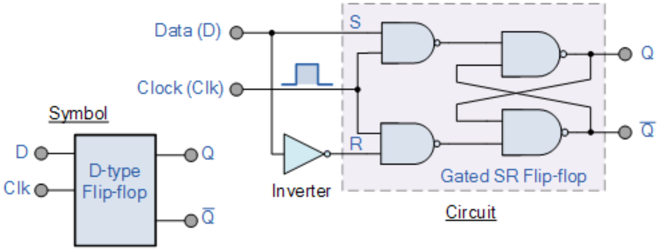
Implementation Details

Implementation Details

- Ok, so we've got a general idea of how we want to build our RAM
- We've seen a picture of how it's done on computers
- We also know the general components that we want to be employing in this implementation
- **Big Question:** How will *we* implement it?

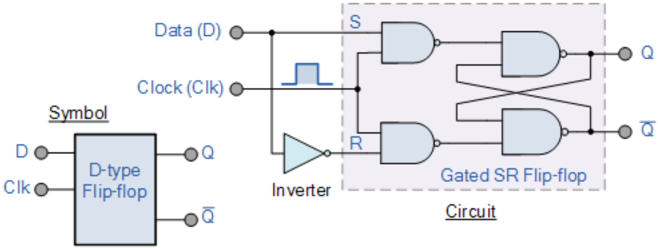
D Flip Flop

- We're implementing SRAM (No need for capacitor here)
- We'll be using D Flip Flops
  - The D in this case stands for data
- Think of a D Flip Flop as an augmented version of the SR Flip Flops we learned about earlier



D Flip Flop

- It's basically our old Flip Flop, with an added special feature- we're making sure the same signal isn't fed in from both S and R
- If this is confusing for you, just understand that it's a very cool way to use flip flops (latches) to store data



D Flip Flops

- Just like before, each Flip Flop can be seen as one **bit** of memory
  - That is, each of these fancy circuits can store either a 1 or a 0

D Flip Flops

- Just like before, each Flip Flop can be seen as one **bit** of memory
  - That is, each of these fancy circuits can store either a 1 or a 0
- We're going to need a lot more of these!

D Flip Flops

- Just like before, each Flip Flop can be seen as one **bit** of memory
  - That is, each of these fancy circuits can store either a 1 or a 0
- We're going to need a lot more of these!
- Okay, let's assume we now have a ton of these bad boys, all laid out in some sort of array or matrix.
- How would we efficiently be able to access the ones we want?

D Flip Flops

- Just like before, each Flip Flop can be seen as one **bit** of memory
  - That is, each of these fancy circuits can store either a 1 or a 0
- We're going to need a lot more of these!
- Okay, let's assume we now have a ton of these bad boys, all laid out in some sort of array or matrix.
- How would we efficiently be able to access the ones we want?
  - **Hint:** Remember when we talked about efficient memory addressing in a previous lecture?

MUX & DEMUX

- To get the specific memory address we want, we will use a demux
- We're going to use an array for simplicity's sake
- Plus, there's another very cool reason we want to build these RAM blocks in order...

Sequential Access

- What if we want to pull more than just a singular bit from RAM?
- In fact, this is a common problem in the real world- that's why RAM has **block sizes**
- If your block size is 4, and you want to pull the block starting at n, all you need to do is use a counter
    - Then you'll be able to pull from n, $n+1$, $n+2$, and $n+3$
- **Aside:** Most computers distinguish clearly between RAM and registers, but we're going to splay the definition a little, just for our convenience

General Architecture

- **Aside**: We're straying from real world architecture choices a bit, but don't forget that this is still a game :)
- We don't have to concern ourselves with the nuances of electronics, nor do we have the same spatial limitations of the real world
- What we do have is limited time and workload expectation, so that's what we're forced to work with
- **TL;DR** we made things easier when making certain architectural choices. You are still building a functional computer, albeit a *very* simple one.

Final Project

Demo