University of Alabama in Huntsville

LOUIS

Honors Capstone Projects and Theses | Honors College

5-1-2022

Teaching Assembly Programming Through Video Games

Kaden Ven Gryphon

Follow this and additional works at: https://louis.uah.edu/honors-capstones

 Part of the Artificial Intelligence and Robotics Commons, and the Other Education Commons

Recommended Citation

Gryphon, Kaden Ven, "Teaching Assembly Programming Through Video Games" (2022). *Honors Capstone Projects and Theses*. 704.
https://louis.uah.edu/honors-capstones/704

This Thesis is brought to you for free and open access by the Honors College at LOUIS. It has been accepted for inclusion in Honors Capstone Projects and Theses by an authorized administrator of LOUIS.

Teaching Assembly Programming Through Video Games

by

Kaden Ven Gryphon

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Certificate

to

The Honors College

of

The University of Alabama in Huntsville

05.01.2022

Honors Capstone Director: Dr. Haeyong Chung

Kaden Gryphon 04.29.2022

Student (signature) Date

Chung 04/29/2022

Director (signature) Date

Dr. Letha Etzkorn Digitally signed by Dr. Letha Etzkorn
Date: 2022.04.29 09:52:52 -05'00'

Department Chair (signature) Date

William Wilkerson Digitally signed by William Wilkerson
Date: 2022.05.01 14:34:09 -05'00'

Honors College Dean (signature) Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

_____Kaden Gryphon_____

Student Name (printed)

Student Signature

04.29.2022

Date

Table of Contents

Copyright Permission	1
Abstract	3
Introduction	4
Chapter 1: The Compiler	5
Chapter 2: The Emulator	8
Chapter 3: The Robot	11
Chapter 4: The User Interface	12
Chapter 5: Gameplay and Levels	15
Chapter 6: Self Assessment	17
Reference List:	20
Conclusion	21
Appendix A (Level Solutions & Game Link)	22
Appendix B (List of Commands)	25

Abstract

This project aimed to create a video game that could be used to help people learn assembly programming, through a virtual robot. This game's assembly language is based off of chapter 5 of the ARM 7TDMI Data Sheet from 1995. This game starts by introducing a few ideas and commands of ARM programming before having the player complete some simple levels to reinforce their understanding. Every couple of levels is another tutorial that introduces some new commands to be used in the next levels. Evey level the player is given control of a virtual robot to program and complete a task. The task is somewhat randomized so that a general solution is required, instead of hardcoding a solution. The player is then scored on how many clock cycles and how much memory their program used. The idea being that having a tangible focused goal to use the commands on as they learn them will help the player learn ARM assembly.

Introduction

This project started because I wanted to write a compiler/emulator for some assembly code and needed an application for it. In middle and high school my favorite classes were the ones where I got to program robots. It was in a class about python programming using a turtle robot that I was first really introduced to programming. It was with these experiences and the desire to write my own compiler that this idea came.

What I Used

After looking at some options like using ASCII art in the console, or using OpenGL with C++, I decided on using the Unity game engine with C# code. I modeled the assembly language on the ARM assembly found in the 1995 version of the ARM 7TDMI data sheet chapter 5. The robot that I went with is based off my final for my first computer science class at UAH. We had to write a sorting algorithm using a robot that could move left and right, pick up and put down blocks, and compare block values.

Chapter 1: The Compiler

The first step of this project was to get a working compiler and emulator so that I knew that the project was feasible, and had an idea on how the emulator would communicate with the robot.

Parsing

The first thing I had to do to write the compiler was work on a method to parse the player's code. Most ARM commands share a similar structure.

<Command> <Rd>, <Rs>, <Rn> @comments

or

<Command> <Rd>, <Rs>, #<literal> @comments

Where Rd is the destination register, and Rs is the source register. Rn and the literal may or may not be used. When a command is parsed it is checked for grammar, like the commas and # symbols. It also counts the number of tokens. The moment the parser hits a '@' character it ends parsing the command. Using the string in <command>, the number of arguments, and if there is a literal it can be determined which command was used.

Example

add r0, r1, r2	4 tokens no literal
add r0, r1, #4	4 tokens and literal
add r0, r1	3 tokens no literal
add r0, #4	3 tokens and literal

All three of these are add commands but they all end up being different formats. Using the number of tokens and if it has a literal the format can be detected.

Encoding

After the information from the parser is acquired the command can be encoded. This part is one of the largest contributors to the line count in this project. First it checks what the command was “add, sub, mov, cmp, etc…”. Then it checks the number of tokens. Lastly it checks if there was a literal. At this point which command was used can be figured out and the command can be converted to binary.

The binary for each command varies but it usually follows this pattern:

<bits for the command><bits for the source register><bits for the destination register>

for the ‘add’ example I will show the decoding for “add r2, r3, #5”. There are 4 tokens and one is a literal so this is the add command from format 2 in the ARM data sheet. The format 2 add binary follows this pattern:

00011<I><Op><Rn/Offset3><Rs><Rd>

Where:

I is 0 if it is using a register and 1 if it is using a literal

Op is 0 if it is a ADD operation or 1 if it is SUB operation

Rn/Offset3 is the value for the register or literal

Rs is the value for the source register

and Rd is the value for the destination register

for “add r2, r3, #5” I is 1, we are using a literal. Op is 0, we are using add. Offset3 is 101, binary for 5. Rs is 011, binary for three. And Rd is 010, binary for 2. This is appended together to create the full binary for the command “0011101010111010”.

First Pass

This game uses a two pass compiler. The first pass looks for labels and other compiler directions like ".text" and ".data" This creates a table of labels and their address in the code. This also allocates memory space for the .data section, though this ended up never being used for the game.

Second Pass

The second pass of the compiler starts compiling the commands. For simple commands as shown above the command can be converted directly to binary. For branch commands the label that is being branched to has to be searched for. The compiler looks through the table of labels found in the first pass and finds the corresponding label. Once that label is found it compares the address of the label with the address of the branch command using that difference as the offset for the branch command binary. Once all of the code is converted to binary the entire binary program is written to a text file, each line with 8 bits. The first line gives the size of the program in bytes, and line 2 and 3 is the address of the first command.

With that the program is compiled and ready to run in the emulator.

Errors

As the program is compiled any errors, such as unknown commands or registers, missing literals or registers, and unknown labels are collected. All of these errors along with their line numbers are recorded in another text file.

Chapter 2: The Emulator

Next step was to get a working emulator so that I could test that the compiler was compiling everything right, and so I could start controlling a robot.

The emulator consists of eight normal registers (r0-r7), a program counter register (PC), instruction register (IR), condition codes register (CCR), a memory address register (MAR), and a memory buffer register (MBR). It also has one kilobyte of memory. When a program is run it reads from the text file created by the compiler and loads it into memory. It then sets the PC to the first value in memory which is the starting address of the program. The main function of the emulator is the step method. This function will run through the fetch execute cycle, decode the command, and return the results. The step function returns -1 for normal commands, and a 0-7 for the robot commands.

Fetch execute

The fetch execute cycle is the process of getting the next command from memory and interpreting it. The cycle starts by moving the value of the PC to MAR. The readHalfword() funcion is then used to move the 16 bits at the address in MAR to the MBR. This value is then moved to IR to be decoded. This is a bit redundant as I could just pull the value from the memory directly, but I decided to fully implement this so that it matches how it works in hardware.

After the command is in IR it can then be decoded and executed.

Decoding

Once the fetch part of the cycle is complete the current command is in the IR register and the command now needs to be decoded. This is the reverse of the encoding step in the compiling process. First the command is and ^ to a bit mask, then the result is compared to op codes for the possible commands. Once the command is known, more masks can be applied to extract the bits that make up which registers are used or what the value of the literal is. Using this the emulator can then carry out the command and update registers as needed.

Example

Using the "001110101011010" we got in the compiler example we can now decode it. We start by extracting the op code. We are starting with "001110101011010" in the IR register. First apply the bit mask "1111111000000000" by computing IR ^ bit mask.

001110101011010 ^ 1111111000000000 = 0011101000000000

This result is decoded to be the add command with two registers and a literal. Now that this is known we can then apply the rest of the bit masks to get the registers and literal.

literal: 001110101011010 ^ 0000000111000000 = 0000000101000000

shift right 6: 0000000000000101 = 3

Rs: 001110101011010 ^ 0000000000111000 = 0000000000011000

shift right 3: 0000000000000011 = 3

Rd: 001110101011010 ^ 0000000000000111 = 0000000000000010

shift right 0: 0000000000000010 = 2

This can be put together to decode the command as "add r2, r3, #5". The value 5 is added to register 3 and the results are written to register 2. After that the cycle is complete and -1 is returned.

Robot commands

The player needs a way to control the robot with the code. Format 17 is traditionally for system interrupts, but for this game I used format 17 for commands that control the robot. While the compiler and emulator is set up to handle eight possible functions I am only using three.

“botmove” returns system interrupt 0

“botgrab” returns system interrupt 1

“botlook” returns system interrupt 2

The emulator does not actually do anything to handle the behavior of these functions, it simply returns the interrupt value to the robot so that the robot knows when to perform these actions.

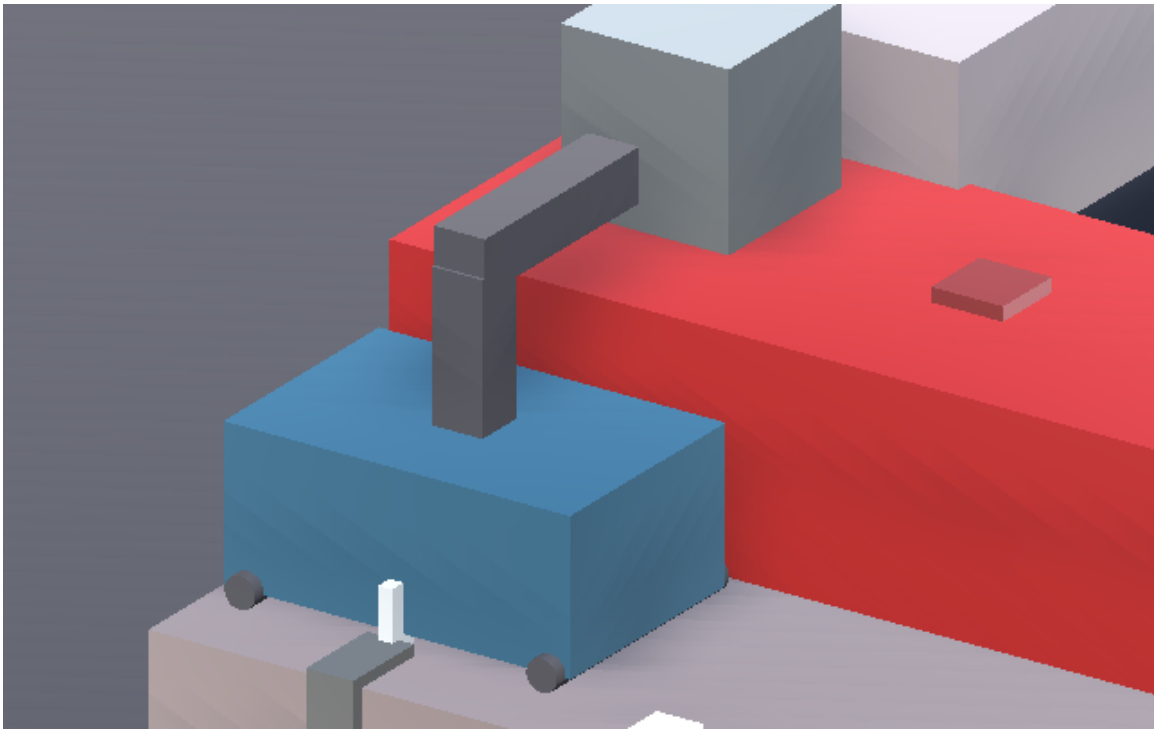

Figure 1: The robot with a block in front of it

Chapter 3: The Robot

The robot is based on my CS 121 assignment I mentioned in the introduction. I decided on a robot that could move left and right, pick up and set down blocks, and read the letter on the block. There would be a shelf in front of the robot that would have blocks on them that the robot could then swap around.

As seen in figure 1, the design of the robot is simple. It is just a box with wheels and an arm sticking out of the top. The robot also has a marker on its side to help show precisely where the robot is located. At the end of the arm there is a box collider that is used to detect and grab blocks. Figure 2 shows the robot holding a block.

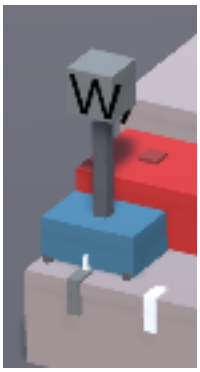

Figure 2: Robot holding block

The blocks are normally white but when the robot is holding one, or the robot arm is able to grab the block, the color is changed to gray to highlight the block. This same things happen to the markers on the floor. As the robot moves left and right, the white markers turn gray when the robot is directly above them. This helps show exactly where the robot is, and what the robot can grab even with the camera at this angled perspective.

To test the robot I used a simple script that would use the “A” and “D” keys to move left and right, and the “space” key to grab a block. After I was sure the robot worked I could begin to set up the user input, where the player could write and compile code to control the robot.

Chapter 4: The User Interface

The user interface (UI) consists of a few main parts. The robot view, the code input, the registers view, the console, and the control buttons. This is shown in figure 3 below.

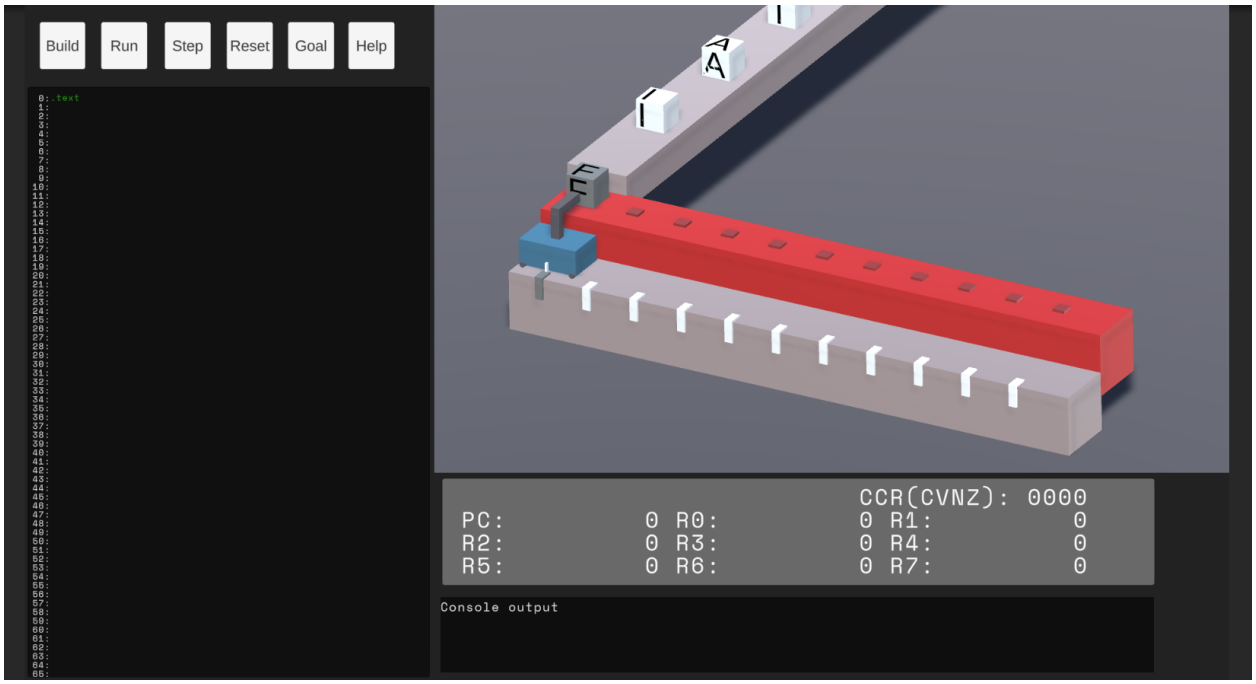

Figure 3: The UI

The Robot View

The robot view consists of the floor that the robot sits on, with the position markers. The shelf the blocks sit on, with spots that the blocks will snap into. The conveyor belt that feeds new blocks to the first spot on the shelf. And the robot itself.

The Code Input

The left side of the screen is where the player writes their code. The lines are numbered to help with fixing errors. The full list of implemented commands is in Appendix B.

The Register View

The registers can be seen listed below the robot view in the gray box. It has all eight normal registers, the PC, the CCR, and it will show the decoded command above PC (not visible in figure 3). This decode command is interesting as the player can see exactly what the computer is doing. When the player writes a branch command "beq loop2" the label "loop2" is converted to a memory offset. So when the game gets to the branch command the decoded command above the PC will show something like "beq -6" as the computer does not know what the labels are, just what offset is stored with that branch command.

Something similar happens with the "mov Rd, Rs" command. This command does not exist in any of the formats. Instead the compiler is turning this into "add Rd, Rs, #0" using an add command where zero is added to the source then stored in the destination. These inconsistencies with how the decoder reads the player's code, match the experience with using the real debuggers for ARM assembly. When players tested the game this was something that they commented on, that the decoded command was not alway the same as what they wrote.

The Console

The console is below the registers and this is where the error messages are displayed. An example of an error message is, "ERROR LINE 1: UNKNOWN REG."

The Control Buttons

The buttons above the code input are used by the player to access various functions.

Build- This takes the players code and compiles it. It saves the code to a text file. Any errors are then written to the console. Random starting values for registers or blocks will be set too.

Run- This loads the program from the text file and starts running it.

Step- This runs the next command then stops. This will interrupt the program if the player has already hit run.

Reset-This will put the robot back to its starting position, reset any blocks and clear all the registers. Random starting values for registers or blocks will be set too.

Goal- This button brings up a pop up with this level's goal on it.

Help- This button brings up a pop up with a list of commands that have been taught, as well as giving the player the option to return to the earlier tutorials.

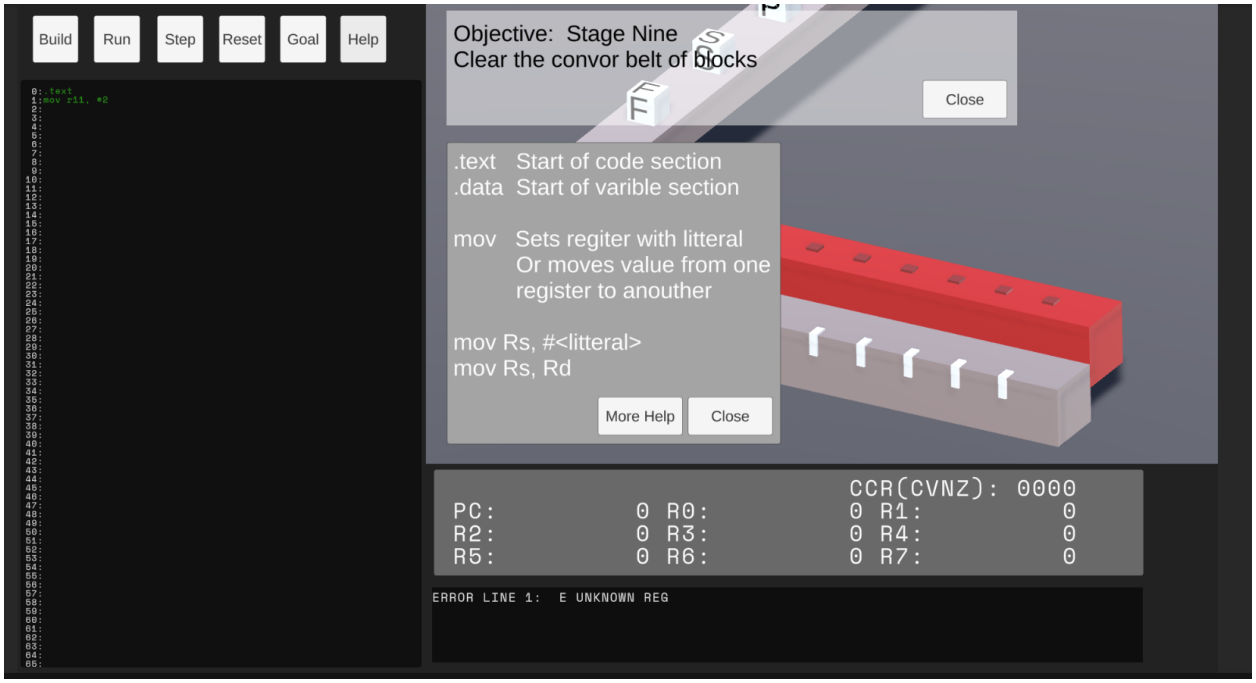

Figure 4: the goal and help pop ups.

Chapter 5: Gameplay and Levels

This game is broken up into ten levels with a tutorial between every couple of levels. Each tutorial introduces a few new commands, and their use. Then the levels after that have the player use the newly learned commands. I will use the first set of tutorials, and levels as an example.

Tutorial One

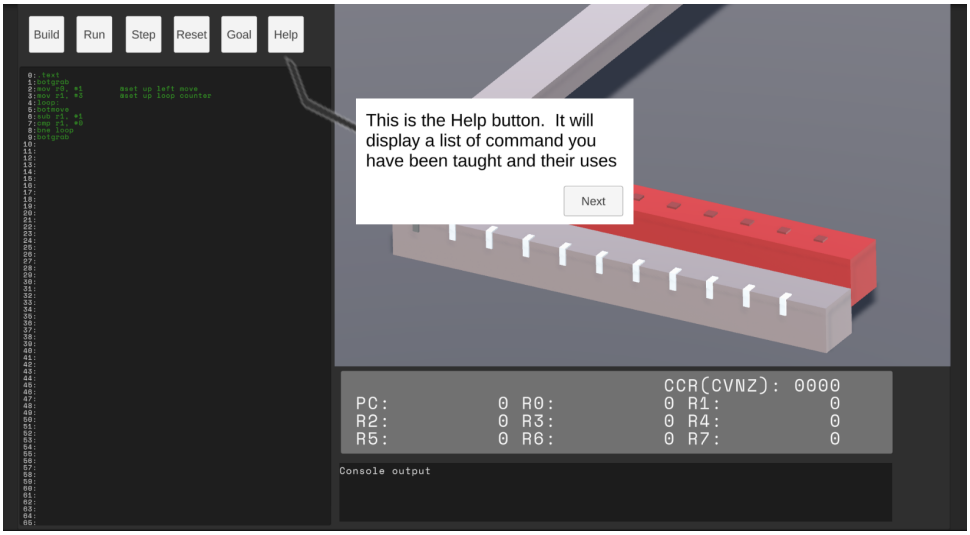

Figure 5: The UI tutorial

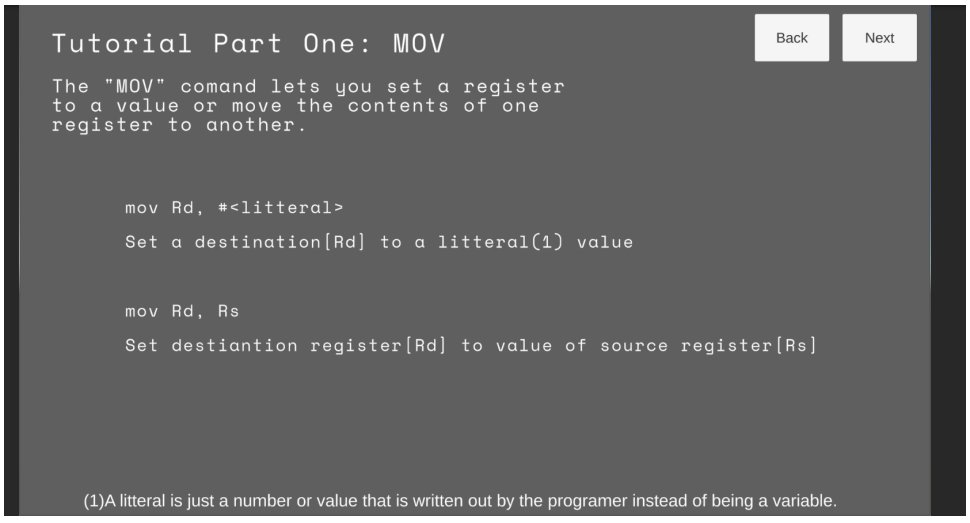

Figure 6: Tutorial one MOV

The game starts with introducing the player to the various elements of the UI. Then it introduces the player to some of the most fundamental ARM commands. During the tutorials the song “Storm Keep” plays, this was written by me.

After the tutorials the first level starts which asks the player to use the mov command to move the value from register two to register five. Durning levels the song “Robot Morning Grove” plays, also written by me. When the player’s program reaches its end the game checks if the player has completed the level's goal. If the player has not completed the goal, a lose screen pops up asking the player to try again. Some of the loose screens have hints on how to beat the level. If the player completes the goal the win screen pops up and gives the player their score. It tells the player how many clock cycles their program used, and how much memory the program used. The win screen then gives the player the option to go to the next level or try another solution for the current level.

A later tutorial level set teaches the player about the branch commands and how to use them to create loops. The tutorial covers their use, and the syntax while giving an example of a loop. Then the level asks the user to move the robot to the right the number of times shown in register three. As every time the build button is pressed the value in register three is randomized; the user must create a loop that uses the value in register three dynamically, instead of hardcoding a static solution.

After all ten levels is a free play sandbox, where there is no given goal, but the player is encouraged to try to sort all the blocks as a challenge. The full list of levels and their solutions can be found in Appendix A.

Chapter 6: Self Assessment

I will be honest, some parts of this project are well thought out and designed in a robust modular way. And others are just barely held together with pixie dust and dreams.

Pros

One of the biggest areas I think I did well in was matching the real world workings of writing and compiling assembly code. The player's code does actually get compiled to ones and zeros, and the emulator has a proper fetch execute cycle. Things like the decoded command above the PC showing that branch commands just store the offset and do not know what the labels are, or that mov Rd, Rs get converted to add, where not planned. The fact that I was matching the real world process caused these odd quirks that happen in real world assembly programming to show up in my game.

I also think I did a good job with the back end design for how the robot is controlled. The robot just reads what the emulator cpu state is in order to be controlled. It does not directly influence the emulation process. This means that I could create an entire new robot, something like a turtle bot, and reuse the entire emulator code with relative ease. I would not have to change any of the emulator code, I would just have to write a new interface that would read the cpu state to control the new turtle bot. I feel I did a good job designing the robots and emulator in such a way that it is very modular and not too difficult to change the level, or robot around without rewriting everything.

I think that my project succeeded in its goal. It can be used to teach assembly. While I was only able to have one person test it, I think that it had good results. I had my roommate play the game with little to no interference from me. My roommate has little to no programming experience, and absolutely no assembly experience. And while They went into the game saying that they had no chance of understanding how any of it worked, after just an hour or two They had made it through nearly all of the levels. And from listening to their monologue as They played, They had a pretty good understanding of the basic assembly ideas. At the end of it I could then put a simple assembly program in front of Them and They could read it and tell me what it would do.

Cons

One of the biggest cons of this project is that it is fairly buggy. I have tried to fix most of them, but the fixes are just patch work. All of the short term bug fixes built up on each other to make it very difficult to edit the code. This really shows in the compiler. The code for the compiler is pretty sloppy. The compiler sometimes fails to compile and I have not been able to figure out why. But with how much of a mess that part of the code is I can not easily debug it. After the experience I gained finishing this project I think I could do a much better job of writing the compiler. I would have liked to completely rewrite the compiler, but I did not have the time to spend redoing that much work.

The other main bug that the project has is that the win detection does not seem to always work when running the game on other computers. On mine all the levels work, but when my roommate was playing the game on his computer some of the levels would say that he failed even though he had a correct solution. I did not have time to try and fix this, so I added the right control key to skip the level in case the player gets stuck in one of those bugs. I honestly do not even know how I would debug this, as every time I want to test a fix I would have to build the entire project, send it to a different computer and test it there.

While the game does accomplish its goal of teaching assembly I would have liked to have improved on that. I would have liked to add more features to give the player direct feedback on how they were doing and what their errors were.

Reference List

"ARM 7TDMI." Advanced RISC Machines Ltd (ARM), August 1995.

Conclusion

Overall I think that this project was a success. I had a lot of fun working out the implementation of the compiler and emulator, and it is extremely satisfying to see the robot respond to the assembly code you write. And with my roommates experience with the game I know I was able to teach at least one person a bit about assembly programming. I gained a lot of experience making this and look forward to applying that experience to my future projects.

Appendix A: Level Solutions

Link to game hosted on Google Drive:

https://drive.google.com/file/d/1vQ4Ef6o9fRi_mxaYFBisNENqAJGFUFpd/view?usp=sharing

Stage One: Set r2 to 5

```
.text
mov r2, #5
```

Stage Two: Move the value from r3 to r6

```
.text
mov r6, r3
```

Stage Three: Move the robot to the right

```
.text
mov r0, #1
botmove
```

Stage Four: Add r2 and r3 together and store results in r0

```
.text
add r0, r2, r3
```

Stage Five: Send the robot off the right side of the shelf

```
.text
mov r0, #1
loop:
botmove
b loop
```

Stage Six: Move the robot to the right the number of times stored in r3

```
.text
mov r0, r3
loop:
botmove
sub r0, #1
bne loop
```

Stage Seven: Move the block one space to the right

```
.text
```

```
mov r0, #1
grab:
botgrab
cmp r1, #1
bne grab
botmove
botgrab
```

Stage Eight: Move the block to the position given in r2

```
.text
mov r0, r2
grab:
botgrab
cmp r1, #1
bne grab
loop:
botmove
sub r0, #1
bne loop
botgrab
```

Stage Nine: Clear the conveyor of all 5 blocks

```
.text
mov r2, #1 @distance to travel
mov r3, #5 @loop counter for blocks
grab:
botgrab
cmp r1, #1
bne grab
moveRight:
mov r0, r2
rightloop:
botmove
sub r0, #1
bne rightloop
botgrab
moveLeft:
sub r0, r0, r2
leftloop:
botmove
```

```
    add r0, #1
    bne leftloop
    add r2, #1
    sub r3, #1
    bne grab
```

Stage Ten: Place to two blocks in order

```
    .text
    mov r2, #1 @distance to travel
    grab:
    botgrab
    cmp r1, #1
    bne grab
    botlook
    mov r4, r1
    mov r0, #1
    botmove
    botgrab
    mov r0, #0
    botmove
    grab2:
    botgrab
    cmp r1, #1
    bne grab2
    botlook
    cmp r1, r4
    blt less
    mov r0, #1
    botmove
    botmove
    botgrab
    b end
    less:
    botgrab
    end:
    mov r1, #0
```

Appendix B: List of Commands

- mov Rd, #<literal>
- mov Rd, Rs
- botmove — if r0 is positive move right, else move left
- add Rd, <literal>
- add Rd, Rs, <literal>
- add Rd, Rs1, Rs2
- sub Rd, #<literal>
- sub Rd, Rs, <literal>
- sub Rd, Rs1, Rs2
- b — unconditional branch
- cmp Rs, #<literal>
- cmp Rs, Rd
- beq — branch if equal
- bne — branch if not equal
- botgrab — set r1 to 1 if block grabbed, or 2 if block placed
- botlook — set r1 to value of block held
- blt — branch if less than