Lab 5: Building a Simple Computer Simulator for Robotics Applications

EE 234: Section 2

Michael Mentele

Partner: Jonathan Quissenberry

October 16, 2012

**Introduction**

In this lab we built a robotic machine language (ROBOMAL) simulator to familiarize ourselves with fetch/decode/execute cycles and load/store architectures. This will also provide a code base for later labs pertaining to robot control. The table below (taken from the lab assignment) describes the op-code and operands each command will accept.

|  |  |  |
| --- | --- | --- |
| **Operation Code** | | **Description** |
| *Data Transfer Instructions:* | | |
| READ | 10 | Reads PORTE 7:0 and stores it into a specific data memory cell. |
| WRITE | 11 | Writes to PORTE 7:0 from a specific data memory cell. |
| LOAD | 12 | Loads a word from a specific data memory cell into s0 |
| STORE | 13 | Stores a word from s0 into a specific data memory cell. |
| *Arithmetic Instructions:* | | |
| ADD | 20 | Adds a word from a cell in data memory to s0. The result is stored in s0. |
| SUBTRACT | 21 | Subtracts a word from a cell in data memory from s0. The result is stored in s0. |
| MULTIPLY | 22 | Multiplies the word in s0 by a word in a specific data memory cell. The result is stored in s5:s0. |
| *Branch Instructions:* | | |
| BRANCH | 30 | Branch to a specific address in data memory. |
| BRANCHEQ | 31 | Branch to a specific address in data memory if s0 is zero. |
| BRANCHNE | 32 | Branch to a specific address in data memory if s0 is not zero. |
| HALT | 33 | End of the program, robot stops. |
| *Robot Control Instructions (RCI):* | | |
| LEFT | 40 | Turn the robot left some specified number of degrees between [0:99]. |
| RIGHT | 41 | Turn the robot right some specified number of degrees between [0:99]. |
| FORWARD | 42 | Move the robot forward at slow (00), medium (01), or fast speed (10). |
| BACKWARD | 43 | Move the robot backward at slow (00), medium (01), or fast speed (10). |
| BRAKE | 44 | Slow the robot down for some number of seconds between [0:99]. |

We will be using the peripheral LEDs to indicate several of the RCI commands however not all commands can be simulated in this way or even completed as yet. Instead they were verified using MPLABs GUI.

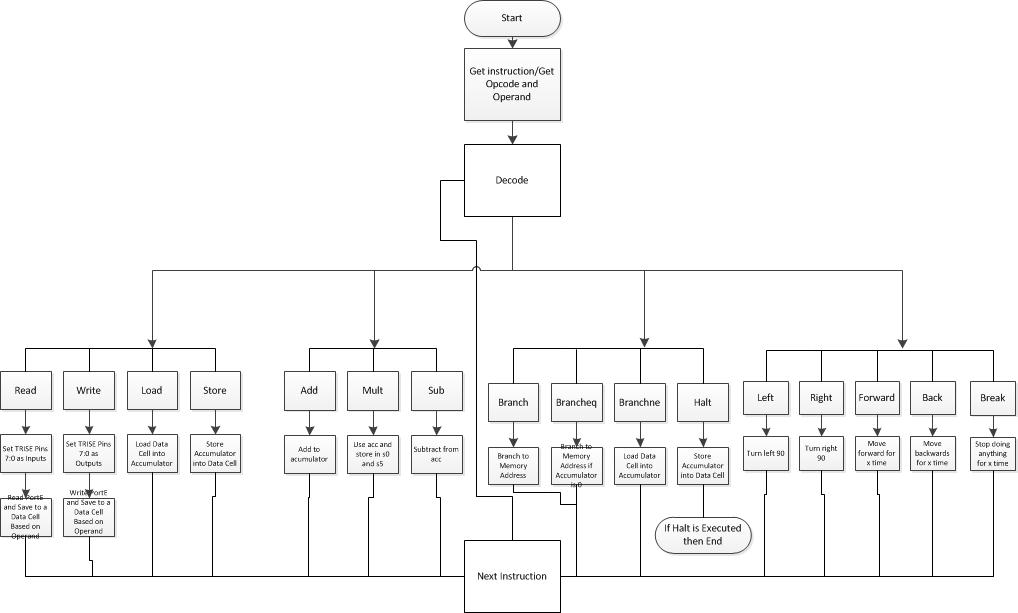
**Software**

**Design**

When we enter the code it begins by specifying several programs in data memory. For this lab I created five programs. The first was from the pre-lab and the other four simply ran through all the commands from each section. For example for the “data type” intructions the program “datatest” went through READ, WRITE, LOAD, and STORE respectively ending with a HALT command. The other programs were similar but with the respective command types. In addition all programs end with a HALT command.

The variables in the programs are placed at the end of the programs because our simulator would attempt to decode a variable when it came across it in “program memory.”

The memory is set up to use relative locations. Therefore you need to specify the number of cells from the command in question to the required. The other way to do this would be to keep track of the beginning of your address in memory this will be discussed as an alternative in the conclusion.

Please see the flowchart below for an overview of the ROBOMAL architecture. You can see that the fetch/decode/execute cycle in executed in a typical fashion. You first *load* the instruction from program memory. Then you decode the instruction and use the op-code to navigate to the correct subroutine and use the operand to perform the correct processing in that subroutine. After execution your store your values (if any) back into program memory.

**Instruction types**

Data Transfer Instructions:

For these instructions you are always referring to some data memory cell. Much of the code here is recycled for each subroutine. Generally you will just be interacting with memory.

Arithmetic Instructions:

You will perform arithmetic operations based on the accumulator and variables in memory.

Branch Instructions:

These instructions allow you to jump to different memory cells in program memory. This is useful for creating program loops or specifying conditional behavior.

Robot Control Instructions:

This is the part that performs physical interaction of the car with the outside world eventually. These instructions make the robot move but in the case of this program do modify variables in memory.

**Important Registers**

S0 => Accumulator Register

S1 => Instruction Counter

S2 => Instruction Register

S3 => Operation Code

S4 => Operand

S5 => Program Memory Location

S6 => Accumulator Overflow

**Test Procedure and Results**

**Methodology**

For this simulation device we tested functionality using breakpoints and the various views in MPLAB such as CPU registers, memory, and peripheral memory. For this project the only sources of error would be if the program was written in an unanticipated fashion. We realized this was not accounted for in that the code is not 100% explicit so that if you do something “wrong” it doesn’t know to jump to some kind of error loop. The solution to this will be discussed in the conclusion.

**Procedure**

1. Test Correct Operation
   1. Test each possible case
      1. Does it fetch?
      2. Does it decode correctly?
      3. Does it navigate to each subroutine correctly (opcode)?
      4. Does it access or use the correct operations depending on the operands?
2. Attempt to Break Current Build
   1. Program corruption
      1. Subroutines?
      2. Programs?

**Results**

Once you specify your program(s) and specify which one to run there is no further human interface. Therefore the only possible source of error is dependent on our inputs (programs) to the system (code/subroutines). When programs were corrupt or had unanticipated values that navigated to, for example, cells that didn’t exist the program behaved in a way that made sense; they didn’t do any meaningful work. The solution to how to identify and avoid getting “stuck” in loops or other problems is discussed in the conclusion.

**Answers to Questions**

1.Explain how the operation code of each instruction indicates the category of instruction to which it belongs.

The first digit of the op-code designates the instruction category. So 1X would be a data instruction, 2X would be arithmetic, 3X would be branching, and 4X would be the RCI instructions.

2. Describe how your design would change if instruction were 16-bits instead of 32-bits.

If the instructions sizes were 16-bits then the normal bitwise operations for decoding instructions would require less shifting. Our accumulator would have less versatility because of the smaller size this would mean we can perform smaller integer operations for arithmetic instructions. It would simply change how programs are read and manipulated.

3. Imagine that you were going to control a robotic car. Expand on the ROBO-MAL language and design five other instructions that would be helpful in controlling the robotic car. These do NOT have to be robot control instructions; they may be data transfer, arithmetic, or branch instructions. Also, provide the operation code for the instructions. Feel free to be creative!!!

|  |  |  |
| --- | --- | --- |
| Name | Instructions | Description |
| Division | 23 | Divides the accumulator by a specific data cell from program memory. |
| Variable Reset | 14 | Zero out a specific data cell from memory. |
| Reset Accumulator | 15 | Set the accumulator to 0. |
| LED Sequences (Turn Signals/Brake Lights/etc.) | 45 | Illuminate LEDs corresponding to the direction you are turning. |
| Variable Rerun | 16 | Restart the program an amount of times depending on a value stored at a specific memory cell. |

**Conclusion**

The project worked how I expected for the most part. One thing that could have been implemented is some kind of safety net for when the program is written incorrectly. A simple way to do this would have been to add a branch statement after my “do” loop where we read the opcode and choose a subroutine where if none of them were branched to it would automatically end the program.

In the future this code base can be used to add more instructions and programs too. We will use this to work with a DC motor and other sensors which will likely need additional instructions.

**Appendix**

Pre-Lab:

|  |  |  |
| --- | --- | --- |
| Memory Cell | Instruction or Data Value | Description |
| 00 | 4202 | FORWARD at medium speed |
| 01 | 3300 | HALT program, robot stops |
| 02 | 4190 | RIGHT 90 degrees |
| 03 | 4202 | FORWARD at medium speed |
| 04 | 3300 | HALT program, robot stops |
| 05 | 4190 | RIGHT 90 degrees |
| 06 | 4202 | FORWARD at medium speed |
| 07 | 3300 | HALT program, robot stops |
| 08 | 4190 | RIGHT 90 degrees |
| 09 | 4202 | FORWARD at medium speed |
| 10 | 3300 | HALT program, robot stops |
| 11 | 4190 | RIGHT 90 degrees |