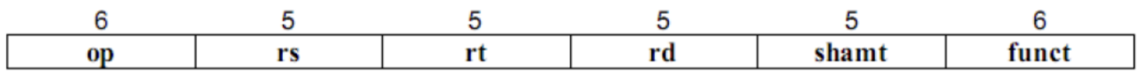
Luke Reicherter
KU ID: 3135060
EECS 510
Final Project

The Formal Language

The idea of this language was inspired by the content covered in EECS 330 (Embedded Systems) and EECS 645 (Computer Systems Architecture). In these classes, we studied the complete path from high level source code to low level assembly code, and further into the hardware level execution model. The MIPS (Microprocessor without Interlocked Pipeline Stages) instruction set architecture was the main focus for these processes within both courses, as the clean and easy to understand format made it easy to demonstrate compilation, decoding, and execution steps. The goal of my project was to show that a real world instruction set architecture, specifically the MIPS R-Type format, can be modeled using a single Turing Machine. To do this, I implemented a machine in Python within a class named “RTypeTM”. This class houses a dictionary that contains every transition in the format of: (Current State, Read Character): (Write Character, Direction, Next State).

In the MIPS architecture, R-Type instructions perform operations on two specified registers containing binary numbers, and store the output in a third selected register. R-Type instructions start with a five bit opcode, followed by a five bit rs field (source register), a five bit rt field (target register), a five bit shamt field (shift amount), and a six bit funct field (operation). My project demonstrates how this type of MIPS instruction can be mapped onto a Turing Machine by using a simplified address system and limited function types.



6	5	5	5	5	6
op	rs	rt	rd	shamt	funct

Constructing the Tape + Alphabet:
Symbol Alphabet: Σ = {0, 1, #, A, B, C, D, E, X, Y, Z, :, +, _, d}

Creating a valid tape involves making a R-Type MIPS instruction followed by a consistent register format. The steps to make a valid tape are listed below, append the string from each step to the end of the string from the previous step. An example string after each step will be shown, as well as the meaning of the string for the MIPS instruction section. DO NOT include quotes in the input.

Step 1: Opcode

- Always “000000” for R-Type instructions
- Example String So Far: “000000”
- Meaning: R-Type Instruction

Step 2: RS Field

- The RS Field contains the address of the register for the first operand in the equation
- Select one of the following five bit sequences, each corresponding to a register address:
    - For register A: “00001”
    - For register B: “00010”
    - For register C: “00100”
    - For register D: “01000”
    - For register E: “10000”
- Example String So Far: “00000000001”
- Meaning: R-Type Instruction with the first operand contained in register A

Step 3: RT Field

- The RT Field contains the address of the register for the second operand in the equation
- Select one of the following five bit sequences, each corresponding to a register address:
    - For register A: “00001”
    - For register B: “00010”
    - For register C: “00100”
    - For register D: “01000”
    - For register E: “10000”
- Example String So Far: “0000000000100010”
- Meaning: R-Type Instruction with the first operand contained in register A, the second operand contained in register B

Step 4: RD Field

- The RD Field contains the address of the register where the output of the equation will be stored

- Select one of the following five bit sequences, each corresponding to a register address:
    - For register A: “00001”
    - For register B: “00010”
    - For register C: “00100”
    - For register D: “01000”
    - For register E: “10000”
- Example String So Far: “000000000010001000100”
- Meaning: R-Type Instruction with the first operand contained in register A, the second operand contained in register B, and destination register C

Step 5: Shamt Field

- Since the only allowed operations in this machine are add and subtract, the Shamt field is unused
- Always “00000”
- Example String So Far: “00000000001000100010000000”
- Meaning: R-Type Instruction with the first operand contained in register A, the second operand contained in register B, and destination register C. No shift amount.

Step 6: Funct Field

- Used to select the operation
    - For addition “000010”
    - For subtraction “000001”
- Example String So Far: “00000000001000100010000000000010”
- Meaning: R-Type Instruction with the first operand contained in register A, the second operand contained in register B, and destination register C. No shift amount. Using addition, the equation becomes: Value in register A + Value in register B, with the output stored in register C.

Step 7: First Hashtag

- Used to separate instruction section from the register section
- Always “#”
- Example String So Far: “00000000001000100010000000000010#”

Step 8: Registers

- Contains the registers where the values are stored
- Registers can contain any amount of bits, as long as the sizes remain consistent between registers.
- NOTE: All testing has been done with a standard eight bits in each register. Increasing or decreasing this amount may lead to errors, but should work in most cases
- Replace Z's with 1's or 0's
    - "A:ZZZZZZZZB:ZZZZZZZZC:ZZZZZZZZD:ZZZZZZZZE:ZZZZZZZZ"
- Example String So Far:
  "00000000001000100010000000000010#A:00110011B:11001100C:00001000D:11000000E:11111111"

Step 9: Last Hashtag

- Used to separate register section from the workspace
- Always "#"
- Example String So Far:
  "00000000001000100010000000000010#A:00110011B:11001100C:00001000D:11000000E:11111111#"

Step 10: Running the machine

- Upon starting the machine, input the generated string from the previous steps
- After output is generated, only steps 1 - 6 need to be repeated for the input
- DO NOT repeat hashtag or register steps for subsequent inputs
- This simulates consecutive instruction inputs with consistent register values
- Example Input:
  "00000000001000100010000000000010#A:00110011B:11001100C:00001000D:11000000E:11111111#"
- Example Output:
  Finish
  String is valid
  Final tape output: A:00110011B:11001100C:11111111D:11000000E:11111111
  Input the next instruction to compute on same registers:

Other Notes:

- Negative binary number outputs are not guaranteed to work

Grammar

Due to the size of the alphabet, and the amount of states, the unrestricted grammar is quite large (15+ pages!!!). A complete printout of the grammar is available in a separate file within the assignment folder. To generate the grammar, I used a python function that translates the transition dictionary into a list of rules used to generate valid strings.

Visual Automaton

The visual automata is contained within the assignment folder. Matplotlib.pyplot and networkx were used to generate the graph automatically. Due to the size of the automata, some overlapping states and overall graph messiness can occur. To regenerate the graph, uncomment the tm.tm_to_graph( ) line.

Test Cases

Below is a list of test cases and outputs:

Test 1:

- Input:
  “0000000000100010000010000000000010#A:00111110B:00010101C:11111111D:00000100E:00100000#”
- Output:
  Finish
  String is valid
  Final tape output: A:01010011B:00010101C:11111111D:00000100E:00100000
  Input the next instruction to compute on same registers:
- Input: ”00000010000100001000000000000010”
- Output:
  Finish
  String is valid
  Final tape output: A:01010011B:00010101C:11111111D:00000100E:01000000
  Input the next instruction to compute on same registers:

Test 2:

- Input: 10001001010101001010101001010101010010101001010101

- Output:

  op0

  String is not valid

Test 3:

- Input:

  0000001000000100000010000000000001#A:01010011B:00010101C:11111111D:0000010
  0E:01000000#
- Output:

  Finish

  String is valid

  Final tape output: A:01010011B:00010101C:00111100D:00000100E:01000000