Group 103
Joshua Blanck

This project can be run similarly to Lab 8, as it uses ANTLR. The project was developed using the docker image of ANTLR.
i. e. In an environment where ANTLR is installed, run
1. antlr4 MIPS.g4
2. javac *.java
3. grun MIPS start (optional flags) < (input file)
A sample input file showcasing the project is provided, called MIPS.in.
For reference on valid instructions and their formats, as well as register names, the file MIPS reference.pdf is included.
Valid memory references start with 'm' and have a number less than 127, e.g. m0, m55, m127 are valid

CheckList
In MIPS.g4;
Variables simulating a MIPS processor (in the @members block):

rs, rd, rt, imm16 - Instruction arguments, represent the datapath
acc - The 64-bit accumulator as used for Multiply and Divide operations
reg - a 32-int array representing the register file
mem - a 128-int array representing data memory
regNames - a hash map that maps register names to reg indices
regNames initializations

Lexer and Parser Rules

REGNUM – Rule that gives a valid register name
MEMADDR – valid memory addresses
IMM16POS and IMM16NEG – valid ranges for a 16-bit immediate argument, negative values include -32768 and so had to be distinct from positive
COMMENT – Assembly comment, starts with #, goes until end of line
start – reads in instructions or comments until end of file
rtype – structure of an r-type instruction : rd, rs, rt, sets rd, rs, rt to proper indices
itype – structure of an i-type instruction: rd, rs, imm16, sets rd, rs and imm16
instruction – valid assembly instructions for this parser

Arithmetic Operations

add – adds contents of rs and rt, stores in rd
addi – adds contents of rs and the imm16, stores in rd
sub – subtracts contents of rt from rs, stores in rd
subi – subtracts imm16 from of rs, stores in rd – not an actual MIPS instruction, but included for symmetry
lui – loads the imm16 value into the high order bits of rd

Print Operation

ptype – structure for a print instruction: register or memory
print – given a register name or memory name, prints the contents thereof – not a MIPS instruction, included here to see how the values change

Load and Store Operations

ltype – structure for Load instructions: rd, imm16(rs)
stype – structure for Store instructions: rs, imm16(rt)
LW – uses contents of rs as a memory address, adds the offset imm16 and loads the memory data at that address into rd
SW – uses contents of rt as a memory address, adds the offset imm16 and stores the data of rs into that memory address

Multiply and Divide Operations

mdtype – structure for a multiply/divide operation: rs, rt

div – stores rs/rt in the low bits of the accumulator, and rs % rt in the high bits

madd – multiplies rs * rt, and adds that value to the accumulator

msub – multiplies rs*rt and subtracts that value from the accumulator

mul – r-type multiply, multiplies rs*rt and stores into rd. Truncates to 32 bits if overflow occurs

mult – multiplies rs*rt and stores into accumulator

Accumulator Access Operations

mvtotype – structure for move into instructions: rd

mvfrtype – structure for move from instructions: rs

mfhi – moves the 32 high order bits of accumulator into rs

mfhi – moves the 32 low order bits into rs

mthi – move the contents of rd into the high order bits of the accumulator

mfhi – moves the contents of rd into the low order bits

Other

Both all capital and all lowercase instructions are valid (e.g. add and ADD are the same instruction)

MIPS Parser Using ANTLR

Joshua Blanck
Group 103

Table of Contents

Overview

This project is a MIPS parser using ANTLR. What this means is, running the grammar on an input file of MIPS instructions will simulate a MIPS environment. The parser generates a 32-bit register file and a 128-bit memory segment. As the parser reads in the instructions, it executes them as if it were a MIPS processor. Due to the limitations of ANTLR as a parser, Branch instructions could not be implemented.

This project forced me to go more in-depth into ANTLR as a lexer/parser. Some of the more complex pieces included register file access and interaction between 32-bit and 64-bit values.

My Bloom's Taxonomy Section is fairly short, as I believe this project to be fairly simple to understand.

New & Complex

Since we had covered ANTLR relatively lightly in class, the majority of this project was new, mostly because I had to gain a deeper understanding of Lexer and Parser rules. Specifically, how lexer and parser rules differ, how fragments are understood, and how the union of the three creates a working parser. Since we had a light introduction, we never covered models/patterns of lexer/parser writing, which forced me to develop my own: the instruction looks for its name, uses the structure (r-type/i-type/etc) to get the arguments, and then executes the instruction.

Further, while not exactly new to me, MIPS instructions were not covered in class, and so qualify as "new". MIPS is an instruction set architecture, commonly called an Assembly Language. It serves as the interface between processor hardware and software. MIPS specifically is the ISA most commonly taught in our COM S and CPR E courses; COM S 321, and CPR E 288 and 381to name a few, so it is the ISA that myself, and likely most other COM S/S E/CPR E majors are most familiar with.

The most complex part of this project, or rather what presented the most difficulty, was figuring out how the register file ought to be indexed and accessed. Initially, register names would be input as 'r[0-31]' and the parser would truncate the r and take the number as the index. Once I decided to include the alternate register names, (e.g. t0, s0, zero) I decided a HashMap with key-value pairings would be the optimal solution. The HashMap mapped the valid register names to the index, for both the simple (r[0-31]) or alternate names (e.g. r8 and t0 refer to the same register)

The other piece that presented some difficulty was in implementing the Multiply/Divide and the Accumulator Access Operations. The difficulty in this resulted from the MIPS implementation of the operations. Since register values are all 32-bit (int), multiplying them together can result in values of size >32. MIPS has a 64-bit accumulator value for Multiply/Divide instructions, which I emulated using a long value, acc. The actual complexity came in the Access Operators, since they require the acc to have high or low order bits extracted or have the high or low order bits assigned to. This was resolved be making liberal use of bitwise operators, bit shift operators, and many casts to long or int, as needed.

Bloom's Taxonomy

After completing this project, I believe I have gained a deeper understanding of how lexers and parsers interact and operate. The lexer reads through the file and tokenizes the parts it recognizes, whereas a parser recognizes patterns of tokens, and can take some action depending on the pattern. Understanding this cleared up some confusion on when to use a lexer or a parser rule, for example, which after some searching seemed to be a common issue.

Understanding the parser also led me to the conclusion that implementing Branch/Jump instructions would be impossible. The parser operates by iterating sequentially through the file, and Branch or Jump instructions, by definition, force non-sequential behavior in a program. With ANTLR, there is no way to force the parser to 'go back' some number of lines and resume parsing. To allow for proper implementation of Branch/Jump, I would need to write a compiler in its entirety, as a lexer and parser are only parts of a proper compiler.

Overall, I think of this project as kind of silly: It takes in Assembly language, which is lexed and parsed by Java by way of ANTLR, which then executes the instructions in Java, which is compiled down into Java bytecode, then executed by the JRE, which takes the Java bytecode and converts it to machine-specific code. Or put simply, the input code starts at low level, is brought to a very high level, is brought down to a still high level, and then back to low level.