Writing an Assembler for the i281 CPU

Jack Morrison

Department of Computer Engineering, Iowa State University

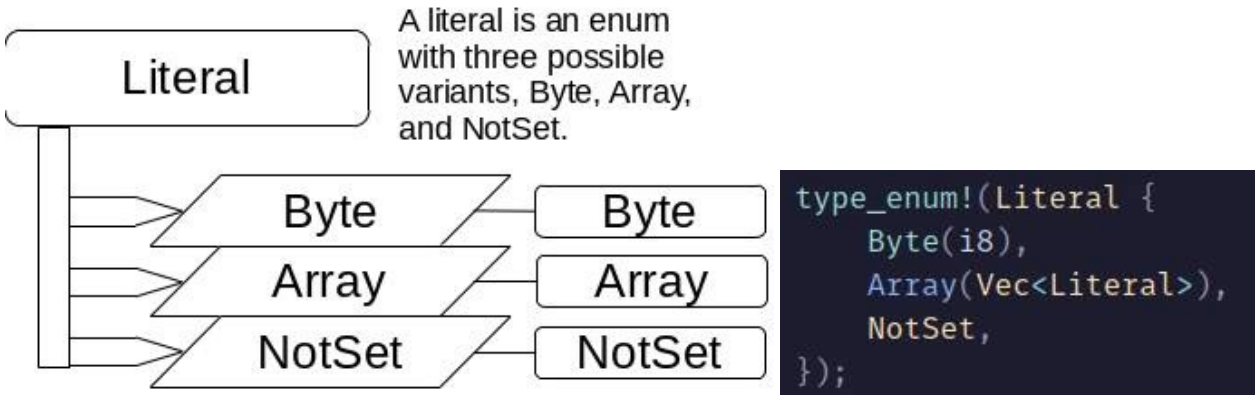
CPR E 281: Digital Logic

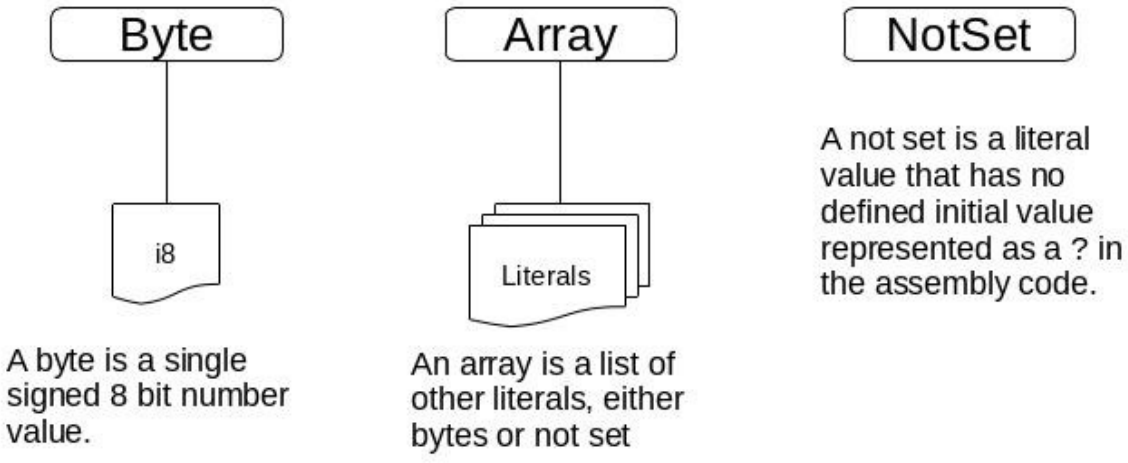
Dr. Alexander Stoytchev

December 8, 2022

The main goal of my final project was to create from scratch an assembler for the i281 assembly instruction set. I could have set about acomplishing this many ways, however the approach I took was to split the compiler / assembler into 3 major parts. These are the topic of this paper the abstract syntax tree (AST) and accompanying parser, intermediate representation (IR) and static analyzer, and finally the verilog compiler. These are the main pieces that make up the complete compiler tool along with some other minor components. These where the pieces I decided to break the problem into and from there build the final product. The language that this compiler is written in is rust a systems programming language designed for memory safety and speed. This was my choice of language because I like writing programs in it, but also because rust has many benefits for writing large projects. With all that in mind the first thing I did was start work on the AST.

The abstract syntax tree (AST) is something that I knew existed before beginning work on this project but had little knowledge on how to implement. To start I began studying other ASTs my main inspiration actually came from the rust syn crate (crates in rust are packages). Syn is huge as it is designed to cover the entire rust programming language, and I only need to implement a small assembly language so I looked at how syn represented different data structures. I started by creating primitive types: Ident, keyword, opcode, oper, register, and literal. These are typically a single logical item. An ident being an identifier of some kind, oper being either an addition or subbtration operator, and a literal being either a single byte or an array of bytes or not set. Literals are the most complex of the primitive types but I would still classify it as primitive because while it has many types it still only consists of one logical piece. Below is a diagram of how a literal is designed, and an image of the code that defines the literal.





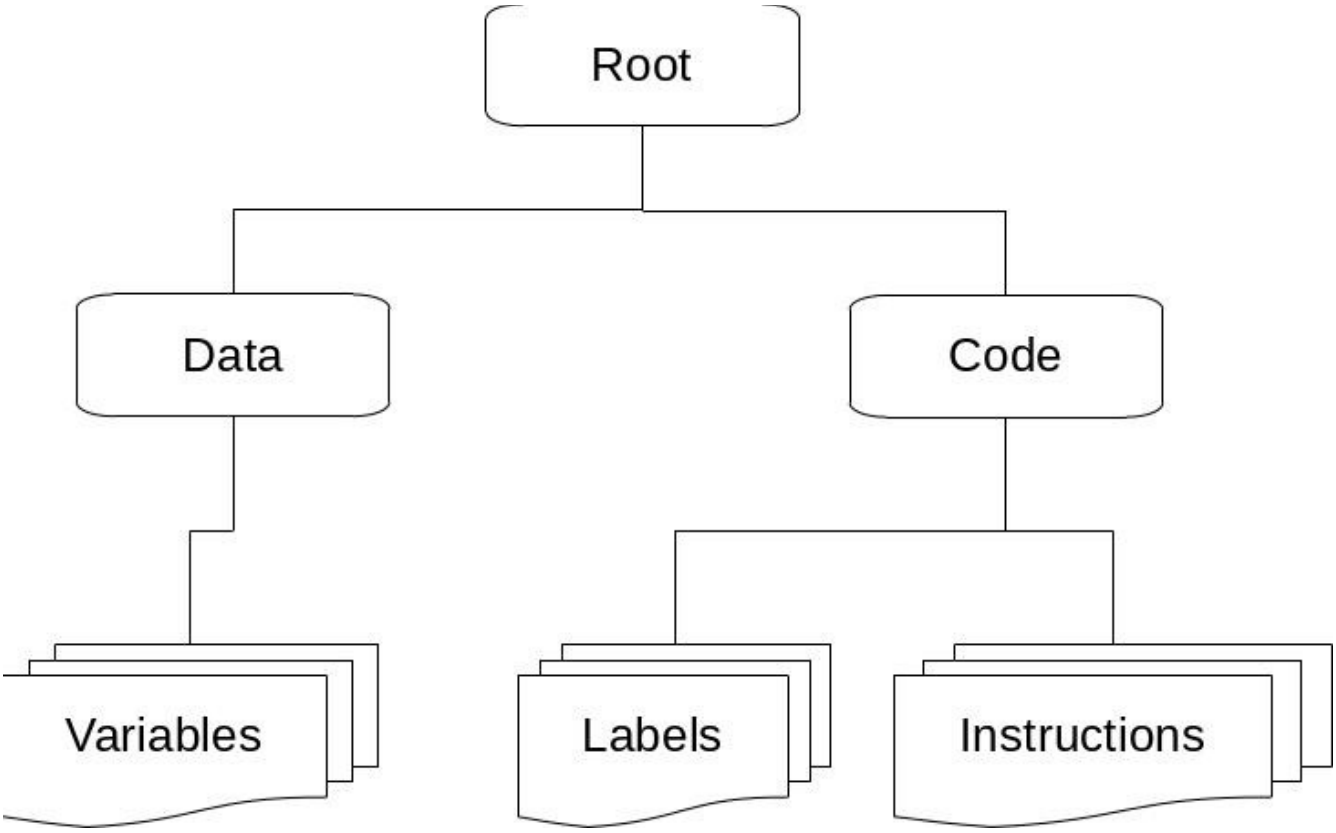
That type_enum! thing is a rust macro it is used to take in rust syntax and generate extra code from it. In this case it creates three things as well as the literal enum. Each of the things inside the curly braces is a variant of the enum the macro creates a new struct for each of them and gives what is inside the parenthesis to it as struct fields. So the Byte(i8) creates a new struct named Byte with a field 0 that has type i8 which is a signed 8 bit integer. The Array is interesting as it contains a vector of literals meaning it can contain a list of Bytes and NotSets. It can also technically contain arrays, however the parser will never insert an array into an array. The macro does a lot more than this but here is an idea of what the the macro generates in terms of structure.

```rust
pub enum Literal {
    Byte(Byte),
    Array(Array),
    NotSet,
}

pub struct Byte(i8);
pub struct Array(Vec<Literal>);
pub struct NotSet;
```

The macro also implements useful traits and other functionality that would expand into several more lines of code for each variant.

The AST has some primitive types but I still had to figure out how to structure the overall layout of the assembly into code. The way I decided to do this was to look at the two main segments of the assembly the data block and the code block. These two blocks are the two largest pieces of any assembly file, and are the first splitting point. Then the data segment contains a list of variables and the code segment contains a list of labels and instructions. Below is a diagram of the root of the AST and the two segments.



A variable consists of an ident which is the name of the variable and the literal value that it is assigned. Labels also consist of an ident which is the name of the label and a code address that is the instruction index starting at zero for the first instruction. An Instruction is an enum with one variant for each uniquely functioning instruction.