This week, I did two major things about the XSD reader generator (uxsdcxx!). The first was to improve the generated code's performance by ~50% by integrating a fast lexer and moving to flat pointers from std::vectors and alike. The second one was to implement enough of XSD 1.0 to generate a reader out of the fpga_architecture.xsd file.

A week ago, uxsdcxx's rr_graph reader used to take 15 seconds to load a certain Artix 7 rr_graph file into its generated structures. I thought std::stoi and friends (which take a std::string argument and throw an exception on invalid input) were causing this, so I swapped them for faster std::strtol calls. (PugiXML's as_int() family of functions wouldn't work because they have no way to check for errors.) That reduced the running time to 13.7 seconds.

Given that PugiXML takes ~4 seconds (on my computer) to load the rr_graph file into its own structs, it didn't make much sense to take 9.7 seconds to validate and copy the structs. I looked at two things for potential speedups: using POD structures as mithro suggested and using a faster lexer to identify nodes and attributes (which, uh, also mithro suggested but I was thinking about it too!!)

Using PODs meant converting this structure:

```
struct t_rr_nodes {
    std::vector<t_node> node_list;
};
```

to this:

```
struct t_rr_nodes {
    int num_node;
    t_node * node_list;
};
```

Then where would the t_nodes reside in memory? In an arena:

```
int g_num_node = 0;
t_node *node_arena;
```

However, for this to work, we need to traverse the DOM once more to count the <node> elements and allocate enough memory for them. Two traversals would be too expensive.

Then I turned to lexer optimizations. I translated Julian Klode's triehash.pl to Python and adapted it for use with uxsdcxx. I chose it because while *trie*s take up a lot of space, they are simple and work well with small words like node and attribute names.

Triehash has an optimization taking advantage of fast unaligned 64-bit reads on amd64 processors. Since I'm doing just the grunt work, I translated only the amd64-optimized part for now and opened an [issue] for translating the char-by-char trie.

The lexers were converted from this:

```
inline enum_node_type lex_node_type(const char *in){
        if(strcmp(in, "CHANX") == 0){
                return enum_node_type::CHANX;
        }
        else if(strcmp(in, "CHANY") == 0){
                return enum_node_type::CHANY;
        }
        else if(strcmp(in, "SOURCE") == 0){
                return enum_node_type::SOURCE;
        }
        else if(strcmp(in, "SINK") == 0){
                return enum_node_type::SINK;
        }
        else if(strcmp(in, "OPIN") == 0){
                return enum_node_type::OPIN;
        }
        else if(strcmp(in, "IPIN") == 0){
                return enum_node_type::IPIN;
        }
        else throw std::runtime_error("Found unrecognized enum value " + std::string(in) + "of
enum_node_type.");
}
```

to this:

```
inline enum_node_type lex_node_type(const char *in){
    unsigned int len = strlen(in);
    switch(len){
    case 4:
        switch(*((triehash_uu32*)&in[0])){
        case onechar('I', 0, 32) | onechar('P', 8, 32) | onechar('I', 16, 32) | onechar('N', 24, 32):
            return enum_node_type::IPIN;
            break;
        case onechar('O', 0, 32) | onechar('P', 8, 32) | onechar('I', 16, 32) | onechar('N', 24, 32):
            return enum_node_type::OPIN;
            break;
        case onechar('S', 0, 32) | onechar('I', 8, 32) | onechar('N', 16, 32) | onechar('K', 24, 32):
            return enum_node_type::SINK;
            break;
        default: break;
        }
        break;
    case 5:
                [...]
```

While the trie-based lexer looks cluttered, it's a lot faster. The running time dropped to ~9.4 seconds, which is a 30% improvement.

Now, since the lexer was optimized, it was possible to traverse the tree twice. For this task, I divided the reader functions into two classes: count_foo and load_foo. count_foo class of functions only traverse the node tree (they don't check attributes), validate the model groups, and increment the global counter when they see an element with an arena to be allocated.

I moved model group validation to count functions, because, well, they can do it. Attribute validation is left to the load functions because it's still expensive to check every attribute twice.

Here is the function counting nodes:

```
int gstate_t_rr_nodes[2][1] = {
    {0},
    {0},
};
void count_rr_nodes(const pugi::xml_node &root){
    int next, state=1;
    for(pugi::xml_node node = root.first_child(); node; node = node.next_sibling()){
        gtok_t_rr_nodes in = glex_t_rr_nodes(node.name());
        next = gstate_t_rr_nodes[state][(int)in];
        if(next == -1)
            dfa_error(gtok_lookup_t_rr_nodes[(int)in], gstate_t_rr_nodes[state],
                      gtok_lookup_t_rr_nodes, 1);
        state = next;
        switch(in){
        case gtok_t_rr_nodes::NODE:
            count_node(node);
            g_num_node++;
            break;
        default: break; /* Not possible. */
        }
    }
    if(state != 0)
        dfa_error("end of input", gstate_t_rr_nodes[state], gtok_lookup_t_rr_nodes, 1);
}
```

Nothing much- it makes sure that the node element occurs at least once, and ticks a counter every time it sees a node.

And, here is the function loading nodes:

```
void load_rr_nodes(const pugi::xml_node &root, t_rr_nodes *out){
    out->node_list = &node_arena[g_num_node];
    for(pugi::xml_node node = root.first_child(); node; node = node.next_sibling()){
        gtok_t_rr_nodes in = glex_t_rr_nodes(node.name());
        switch(in){
        case gtok_t_rr_nodes::NODE:
            load_node(node, &node_arena[g_num_node]);
```

```
            g_num_node++;
            out->num_node++;
            break;
        default: break; /* Not possible. */
        }
    }
    if(root.first_attribute())
        throw std::runtime_error("Unexpected attribute in <rr_nodes>.");
}
```

Here, it traverses the node list again but without validating. Instead, it uses the global node counter(which is zeroed after allocation) as a marker and loads the rr_nodes into the corresponding offset in the node arena.

Also see the attribute validation in action- no attributes are expected in <rr_nodes> so it errors out if it finds one.

Moving away from vectors and counting/loading this way reduced the running time to 7.7 seconds, which I found to be good enough.

In fact, my branch of VPR using the generated reader and copying the generated structs over to VPR structs ended up being ~1 second faster than the current one :) I expect it to fall back, though, after I start checking errno after calls to std::strtol.

The second thing I did was to generate an arch.xml reader (370 KB) from fpga_architecture.xsd. For this, I had to implement xs:alls and xs:unions. xs:lists were creating trouble since they require allocation but they are found in attributes. I ended up just loading them as strings for the time being.

xs:alls are XSD 1.0 alls and they are implemented in the same way with attribute validation:

```
void count_node(const pugi::xml_node &root){
    std::bitset<4> gstate = 0;
    for(pugi::xml_node node = root.first_child(); node; node = node.next_sibling()){
        gtok_t_node in = glex_t_node(node.name());
        if(gstate[(int)in] == 0) gstate[(int)in] = 1;
        else throw std::runtime_error("Duplicate element " + std::string(node.name()) + " in <node>.");
        switch(in){
        case gtok_t_node::LOC:
            count_node_loc(node);
            break;
        case gtok_t_node::TIMING:
            count_node_timing(node);
            break;
        case gtok_t_node::SEGMENT:
            count_node_segment(node);
```

```
            break;
        case gtok_t_node::METADATA:
            count_metadata(node);
            break;
        default: break; /* Not possible. */
        }
    }
    std::bitset<4> test_state = gstate | std::bitset<4>(0b1110);
    if(!test_state.all()) all_error(test_state, gtok_lookup_t_node);
}
```

To load xs:unions, we simply try to load the value into the member types until one of them fits:

```
[...]
        case atok_t_clock::BUFFER_SIZE:
            out->buffer_size.tag = type_tag::ENUM_BUF_SIZE;
            out->buffer_size.as_enum_buf_size = lex_buf_size(attr.value(), false);
            if(out->buffer_size.as_enum_buf_size != enum_buf_size::UXSD_INVALID)
                break;
            out->buffer_size.tag = type_tag::DOUBLE;
            out->buffer_size.as_double = std::strtod(attr.value(), NULL);
            if(errno == 0)
                break;
            throw std::runtime_error("Couldn't load a suitable value into union buf_size.");
            break;
[...]
```

This is the code to load a value, which is a union type of the string “auto” and a double, into clock->buffer_size. It’s loaded into a tagged union:

```
struct union_buf_size {
    type_tag tag;
    union {
        enum_buf_size as_enum_buf_size;
        double as_double;
    };
};
```

I didn’t experiment much with the arch.xml reader, but I found a schema error with it right away. It found metadata in pb_type_inputs, a child element which is not specified in the schema.

It also revealed a problem- the reader errored out on the “xmlns:xi” attribute of the root element. It didn’t expect an attribute there. I wish there was an easy way to skip attributes starting with xml (which likely doesn’t contain data for us), but it looks like we will either ignore unexpected attributes or add “xml” prefix checks everywhere.

Next week, I'll do some polishing on the generator. While it can generate correct and decently performing code, it's not very straightforward to do something useful with the code. I'll also try to use it with VPR to see how would it replace the hand-written arch file parser.