Motivation

- CIRCT’s main backend emits System Verilog
- Output may include interfaces for human consumption, may be inspected during verification, etc.
- Complicated types lead to clutter
  - Often show up multiple times
  - Can make the output unreadable
- “This is quickly getting ridiculous”
  - John Demme

```verilog
interface IValidReady_Struct;

logic valid;
logic ready;

struct packed {
  logic encrypted;
  logic [3:0] compressionLevel;
  logic [31:0][7:0] blob;
} data;

endinterface
```
Motivation

- CIRCT’s main backend emits System Verilog
- Output may include interfaces for human consumption, may be inspected during verification, etc.
- Complicated types lead to clutter
  - Often show up multiple times
  - Can make the output unreadable
- “This is quickly getting ridiculous”
  - John Demme

```verilog
module encode(
    input clk,
    input valid,
    input struct packed {
        logic encrypted;
        logic [3:0] compressionLevel;
        logic [31:0][7:0] blob;
    } data);

    ...

endmodule
```
Goals

- Use System Verilog typedef statements to give names to complicated types
- Refer to the declared type in interfaces, modules, etc.
- Reason about declared types
  - Resolve name conflicts
  - Collect all type declarations to a separate output
  - Etc.

```verilog
typedef struct packed { ... } MyStruct;

module encode(
  input MyStruct data,
  ...
)
endmodule

interface IValidReady_Struct;
  MyStruct data;
...
endinterface
Aside - MLIR type aliases

!MyStruct = type !hw.struct<...>

hw.module(%arg0: !MyStruct) {
  ...
}

- **Supported** in printed IR syntax
- Sort of what we’re looking for
- But the !MyStruct name is gone after the IR is parsed
What we’ve tried - Attempt 1

- No problem, let’s just add a new typedef op.
- Initial prototype here: [PR 645](#)
  - System Verilog exporter registered the name of every typedef’ed type
  - Used the name when it encountered a type
  - What if you want different names for the same type in different places?
- Abandoned, and considered some alternatives...

```verilog
svtypedef "MyStruct" : !hw.struct<...>

hw.module(%arg0: !hw.struct<...>) {
  ...
}

typedef struct packed { ... } MyStruct;

module encode(
  input MyStruct data,
  ...
)
endmodule
```
What we’ve tried - Attempt 1 alternatives

1. Specify which typedef to use as an attribute on the using op
   - Pro: explicit typedef op in IR
   - Pro: doesn’t mess with type system
   - Con: requires attribute preservation
   - Con: requires knowing which ops print the type

2. Add a wrapper type that references the typedef
   - Pro: explicit typedef op in IR
   - Pro: no duplication of types
   - Con: need to resolve reference to know type
   - Con: need to see through type wrappers

3. Add a wrapper type that names a type and makes the typedef implicit
   - Pro: type is known locally
   - Pro: simple to apply a name to any type
   - Con: typedef is implicit in IR
   - Con: duplicated types throughout IR
   - Con: need to see through type wrappers

```swift
hw.struct_get %0["field"] : !hw.struct<...>,
    { typealias = "MyStruct" }

!hw.typealias@MyStruct

!hw.typealias"MyStruct", !hw.struct<...>>
```
What we’ve tried - Attempt 2

Tried option 3 next, since it seemed simple to implement
Quickly realized the implicit typedef makes life hard for System Verilog exporter, and anything else that wants to reason about these types
Will discuss “seeing through” type alias in a bit
What we’ve tried - Attempt 3

- Tried option 2 next
- Explicit typedef op is no problem
  - Landed in PR 1029
- But how do we get at the referenced typedef during parsing and verification?
  - Ops need to know the input types in their verifiers, e.g. to ensure a Struct access is indeed accessing a Struct
  - This may be possible, or possible with a small tweak to OpAsmParser

```plaintext
sv.typedef @MyStruct : !hw.struct<...>
hw.module(%arg0: !hw.typealias:@MyStruct) {
  ...
}

typedef struct packed { ... } MyStruct;
module encode(
  input MyStruct data,
  ...
) endmodule
```
What we’ve tried - Attempt 3

- Ended up putting the inner type into the type alias, so now we have both option 2 and option 3…
  - Implemented in [PR 1076](#) and [PR 1077](#)
- Still need to “see through” type alias

```
sv.typedef @MyStruct : !hw.struct<...>
hw.module(%arg0: !hw.typealias<@MyStruct,
           !hw.struct<...>>) { 
    ...
}
===>
typedef struct packed { ... } MyStruct;
module encode(
  input MyStruct data,
  ...
endmodule
```
What we’ve tried - “seeing through” type aliases

- What happens when an op expecting a Struct sees a type alias for a Struct?
- Have prototyped Clang-style helpers to get canonical types
- Have prototyped a parallel set of type_isa and type_cast “operators”
- Experiments in PR 1143

```cpp
operandType.isa<StructType>()
===>
operandType.cast<HWType>().isStructType()

operandType.isa<StructType>()
===>
type_isa<StructType>(operandType)
```
What we’ve tried - where to put typedefs in the IR?

- If they are Symbols, don’t want them conflicting with other Symbols, like module names.
- Current approach is to have a special type declaration op, which is a Symbol itself (with a specific, reserved name), and holds the typedefs in a single block.
- Also considered putting them as attributes on the top-level ModuleOp.
  - Is this a bad idea?

```c
sv typedefs @__TYPEDEFS {
  sv typedef @MyStruct : !hw.struct<...>
}

module {
  typedefs = {
    "MyStruct": !hw.struct<...>
  }
}
```
What we’re hoping to achieve now

- Want to get back to option 2, where type alias can refer to typedef op without also duplicating the type
  - Can/should we resolve the symbol reference during parsing and verification?
  - Is there a better way?

- Want to figure out the best place to put the typedefs in the IR
  - Especially if they must be parsed before any type aliases

- Want to standardize on how the type alias is canonicalized
  - Either of the approaches outlined, or something else we haven’t considered

sv.typedef @MyStruct : !hw.struct<...>

hw.module(%arg0: !hw.typealias<@MyStruct>) {
  ...
}

==> 
typedef struct packed { ... } MyStruct;

module encode(
  input MyStruct data,
  ...
)
endmodule
Thoughts?