Proposal: Subword Assignment

July 11, 2022

Author: Zachary Yedidia

Design Proposal

The goal in subword assignment is to allow users to assign individual bits of a bit-indexable data type. It is currently possible to extract a particular bit or range of bits from a UInt (for example), but that subrange cannot be assigned. Verilog allows subword assignment and so did Chisel 2. It has been requested repeatedly in Chisel 3.

This change would allow writing Chisel such as:

```
val io = IO(new Bundle {
  val x = Input(UInt(8.W))
  val y = Input(UInt(8.W))
  val en = Input(Bool())
  val out = Output(UInt(8.W))
})

io.out := x
when (io.en) {
  io.out(4, 0) := y(4, 0)
}
```

See `zyedidia/circt/subword-assignment` for the current implementation in CIRCT.

Proposed solution

This proposal suggests implementing subword assignment as an operation in FIRRTL that is lowered by an early pass in MFC. This change requires new semantics in the FIRRTL language, a lowering pass implementation in MFC, and a frontend implementation in Chisel. In order to keep the modification to FIRRTL minor, only single-bit subword assignments are supported in FIRRTL. Multi-bit subword assignment can be done by performing multiple single-bit assignments.

It is easier to use a lowering approach rather than convert directly to Verilog's subword assignment because direct conversion would require bit-level tracking of integer types to avoid combinational loops. By lowering to a vector of bits, we can automatically use the vector analysis to do the heavy lifting.

FIRRTL syntax

No changes necessary.

FIRRTL semantics

The index operator `[n]` is now also supported on values of type `UInt`, and `SInt`, only as the destination of a connection. In such a case, only the nth bit of the value is connected, and all the bits remain unmodified. The result of `v[n]` is `UInt<1>`. In order to perform `v[n] <= e`, `e` must be of type `UInt<1>`.

This only applies when `v` is of type `UInt`, or `SInt`. When used on a Vec, the index operator is a "subindex", and when used on an integer type it is a "bitindex."

Lowering algorithm

At a high-level the lowering algorithm looks for all variables that are subword-assigned. For each such variable `var`, it creates a new wire `v_var` that is a vector `UInt<1>[var.width]`. Then it makes the following transformations:

- `var <= e` becomes `v_var[i] <= bits(e, i, i)` for all `i` up to `var.width`
- `var[n] <= e` becomes `v_var[n] <= e`.
- `bits(var, hi, lo)` becomes `cat(v_var[hi], v_var[hi-1], ..., v_var[lo])`.

Then it connects the concatenation of the vector wire to the variable (`var <= cat(v_var[n], v_var[n-1], ..., v_var[0])` where `n` is `var.width-1`).

See `zyedidia/circt/subword-assignment/LowerBitindex.cpp` for the implementation.

Lowering examples

The examples use the following ports:

```
input x : UInt<4>
input y : UInt<1>
input en : UInt<1>
output out : UInt<4>
```

Example 1

```
out[0] <= y
```

transforms to

```
wire v_out : UInt<1>[4]
node out_T_0 = v_out[0]
node out_T_1 = cat(v_out[1], out_T_0)
node out_T_2 = cat(v_out[2], out_T_1)
node out_T_3 = cat(v_out[3], out_T_2)
out <= out_T_3

v_out[0] <= y
```

and generates an error saying that `out` is not fully initialized.

Example 2

```
out <= x
when en :
  out[0] <= y
```

transforms to

```
wire v_out : UInt<1>[4]
node out_T_0 = v_out[0]
node out_T_1 = cat(v_out[1], out_T_0)
node out_T_2 = cat(v_out[2], out_T_1)
node out_T_3 = cat(v_out[3], out_T_2)
out <= out_T_3

v_out[0] <= bits(x, 0, 0)
v_out[1] <= bits(x, 1, 1)
v_out[2] <= bits(x, 2, 2)
```

```
v_out[3] <= bits(x, 3, 3)

when en :
  v_out[0] <= y
```

Example 3

```
out <= x
out[2] <= y
```

transforms to

```
wire v_out : UInt<1>[4]
node out_T_0 = v_out[0]
node out_T_1 = cat(v_out[1], out_T_0)
node out_T_2 = cat(v_out[2], out_T_1)
node out_T_3 = cat(v_out[3], out_T_2)
out <= out_T_3

v_out[0] <= bits(x, 0, 0)
v_out[1] <= bits(x, 1, 1)
v_out[2] <= bits(x, 2, 2)
v_out[3] <= bits(x, 3, 3)
v_out[0] <= y
```

Example 4

```
out <= x
out[0] <= bits(out, 1, 1)
```

transforms to

```
wire v_out : UInt<1>[4]
node out_T_0 = v_out[0]
node out_T_1 = cat(v_out[1], out_T_0)
node out_T_2 = cat(v_out[2], out_T_1)
node out_T_3 = cat(v_out[3], out_T_2)
out <= out_T_3

v_out[0] <= bits(x, 0, 0)
v_out[1] <= bits(x, 1, 1)
v_out[2] <= bits(x, 2, 2)
v_out[3] <= bits(x, 3, 3)
v_out[0] <= v_out[1]
```

Example 5

Example 5 uses additional ports:

```
input en_2 : UInt<1>
```

```
out <= x
when en :
  out[0] <= y
  when en_2 :
    out[1] <= y
    out[2] <= y
else :
  out[1] <= y
out[3] <= y
```

transforms to

```
wire v_out : UInt<1>[4]
node out_T_0 = v_out[0]
node out_T_1 = cat(v_out[1], out_T_0)
node out_T_2 = cat(v_out[2], out_T_1)
node out_T_3 = cat(v_out[3], out_T_2)
out <= out_T_3

v_out[0] <= bits(x, 0, 0)
v_out[1] <= bits(x, 1, 1)
v_out[2] <= bits(x, 2, 2)
v_out[3] <= bits(x, 3, 3)

when en :
  v_out[0] <= y
  when en_2:
    v_out[1] <= y
    v_out[2] <= y
else :
  v_out[1] <= y
v_out[3] <= y
```

Example 6

Example 6 uses different ports.

This example is from the Chisel issue tracker for subword assignment: `chipsalliance/chisel3#2541`.

```
input request : UInt<3>
output grant : UInt<3>
output not_granted : UInt<2>

grant[0] <= bits(request, 0, 0)
not_granted[0] <= not(bits(grant, 0, 0))
grant[1] <= and(bits(request, 1, 1), bits(not_granted, 0, 0))
not_granted[1] <= and(not(bits(grant, 1, 1)), bits(not_granted, 0, 0))
grant[2] <= and(bits(request, 2, 2), bits(not_granted, 1, 1))
```

transforms to

```
wire v_grant : UInt<1>[3]
node grant_T_0 = v_grant[0]
node grant_T_1 = cat(v_grant[1], grant_T_0)
node grant_T_2 = cat(v_grant[2], grant_T_1)
grant <= grant_T_2

wire v_not_granted : UInt<1>[2]
node not_granted_T_0 = v_not_granted[0]
node not_granted_T_1 = cat(v_not_granted[1], not_granted_T_0)
not_granted <= not_granted_T_1

v_grant[0] <= bits(request, 0, 0)
v_not_granted[0] <= not(v_grant[0])
v_grant[1] <= and(bits(request, 1, 1), v_not_granted[0])
v_not_granted[1] <= and(not(v_grant[1]), v_not_granted[0])
v_grant[2] <= and(bits(request, 2, 2), v_not_granted[1])
```

Note: combinational loops

Here is example 4 shown again:

```
out <= x
out[0] <= bits(out, 1, 1)
```

In this case a combinational loop is avoided because the `bits` operation is transformed into a subindex on the generated vector wire, and therefore the compiler can determine that `out[0]` does not affect the output of `bits(out, 1, 1)`. In order to prevent combinational loops in general, every operator must be transformed into an equivalent operation on bit vectors. This proposal only implements the special case of `bits`. Thus the following would cause a combinational loop:

```
input x : UInt<4>
input y : UInt<4>
output out : UInt<4>

out <= x
out[0] <= bits(or(out, y), 1, 1)
```

Even though the `or` operation performs independently on all bits, the analysis engine operates on a word level. Since the narrowing is performed after the `or` operation, the compiler doesn't realize that bits 0, 2, and 3 of `out` don't affect bit 1 of `or(out, y)`. Since the compiler thinks that `out[0]` may have affected bit 1 of `or(out, y)`, connecting that bit to `out[0]` causes a combinational loop.

This proposal's implementation only performs special-casing to solve this for `bits`, which is the most common cause of combinational loops in subword assignment.

The solution in this case would be to swap the order of the `or` and the `bits`:

```
out <= x
out[0] <= or(bits(out, 1, 1), bits(y, 1, 1))
```

The full transformation that fixes the previous example (i.e., a special-case that performs the `or` operation on each bit individually) would be

```
input x : UInt<4>
input y : UInt<4>
output out : UInt<4>

out <= x

wire tmp : UInt<4>
tmp[0] <= or(bits(out, 0, 0), bits(y, 0, 0))
tmp[1] <= or(bits(out, 1, 1), bits(y, 1, 1))
tmp[2] <= or(bits(out, 2, 2), bits(y, 2, 2))
tmp[3] <= or(bits(out, 3, 3), bits(y, 3, 3))
out[0] <= bits(tmp, 1, 1)
```

then this can be transformed using the existing algorithm.

MFC backend

The implementation in MFC creates a new operation `BitindexOp`, similar to `SubindexOp` but operating on integer types instead of vector types. A pass called `LowerBitindex` will lower all occurrences of `BitindexOp` according to the algorithm described above. This pass should be run early in the pipeline.

Chisel frontend

Some changes to Chisel will be needed to support subword assignment. The syntax

```
v(3, 0) := e
```

will generate the following FIRRTL

```
v[3] <= bits(e, 3, 3)
v[2] <= bits(e, 2, 2)
v[1] <= bits(e, 1, 1)
v[0] <= bits(e, 0, 0)
```

Alternative solutions

Full Chisel implementation

Can implement the transform in Chisel, but this adds a lot of complexity to Chisel, which isn't architected for performing individual transforms like this.

Pros

- Doesn't require any change to the FIRRTL spec or MFC.

Cons

- Adds a lot of complexity to Chisel.
- Cannot be updated in the future to produce cleaner Verilog.

Multi-bit subword assignment

Pros

- Generates smaller FIRRTL.
- Allows for generating cleaner Verilog that directly uses Verilog's multi-bit subword assignment support.

Cons

- Introduces new syntax into the FIRRTL spec (grammar must be changed).
  - `x[3:0] <= e`?
  - `bits(x, 3, 0) <= e`?
  - `x <[3:0] e`?
- Introduces new questions/inconsistencies in the FIRRTL spec:
  - Should it also be possible to do multi-item assignment of Vecs?
  - Should it be possible to do multi-bit subword lookup or only assignment?
    - Syntax: `x <= e[3:0]`?
    - Should this also be possible for Vecs?

Alternative lowering algorithm

- Lower directly to Verilog subword assignment instead of transforming.
  - Pro: produces cleaner Verilog.
  - Con: requires tracking bit-level information on integer types to prevent combinational loops.
- We also considered a different renaming algorithm but it wasn't as good because it didn't handle combinational loops at all.

Future Directions

- Multi-bit subword assignment.
- Lowering directly to Verilog subword assignment.
- Variable-indexed subword assignment.
  - The lowering algorithm in this proposal should work without changes for this.

  - Since subindexes and subaccesses are separate, implementing this would involve creating a bitaccess operation, and lowering that to a subaccess instead of a subindex during the lowering pass.