Verilog: always @ Blocks

Chris Fletcher
UC Berkeley
Version 0.2008.9.4

September 5, 2008

1 Introduction

Sections 1.1 to 1.6 discuss always@ blocks in Verilog, and when to use the two major flavors of always@ block, namely the always@(*) and always@(posedge Clock) block.

1.1 always@ Blocks

always@ blocks are used to describe events that should happen under certain conditions. always@ blocks are always followed by a set of parentheses, a begin, some code, and an end. Program 1 shows a skeleton always@ block.

Program 1 The skeleton of an always@ block

```
1 always @( ... sensitivity list ... ) begin
2     ... elements ...
3 end
```

In Program 1, the sensitivity list is discussed in greater detail in Section 1.5. The contents of the always@ block, namely elements describe elements that should be set when the sensitivity list is "satisfied." For now, just know that when the sensitivity list is "satisfied," the elements inside the always@ block are set/updated. They are not otherwise.

Elements in an always@ block are set/updated in sequentially and in parallel, depending on the type of assignment used. There are two types of assignments: <= (non-blocking) and = (blocking).

1.2 <= (non-blocking) Assignments

Non-blocking assignments happen in parallel. In other words, if an always@ block contains multiple <= assignments, which are literally written in Verilog sequentially, you should think of all of the assignments being set at **exactly** the same time. For example, consider Program 2.

Program 2 <= assignments inside of an always@ block

```
4 always @( ... sensitivity list ... ) begin
5     B <= A;
6     C <= B;
7     D <= C;
8 end
```

Program 2 specifies a circuit that reads "when the sensitivity list is satisfied, B gets A's value, C gets B's **old** value, and D gets C's **old** value." The key here is that C gets B's **old** value, etc (read: **think OLD**

value!. This ensures that C is not set to A, as A is B's **new** value, as of the always@ block's execution. Non-blocking assignments are used when specifying sequential[1] logic (see Section 1.4).

1.3 = (blocking) Assignments

Blocking assignments happen sequentially. In other words, if an always@ block contains multiple = assignments, you should think of the assignments being set one after another. For example, consider Program 3.

Program 3 = assignments inside of an always@ block

```
1 always @( ... sensitivity list ... ) begin
2     B = A;
3     C = B;
4     D = C;
5 end
```

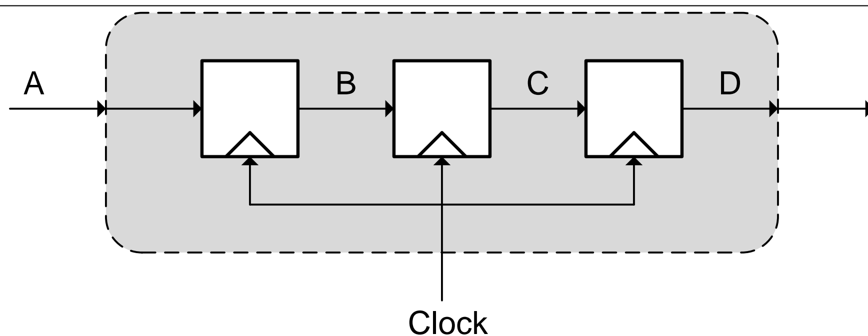
Program 3 specifies a circuit that reads "when the sensitivity list is satisfied, B gets A, C gets B, and D gets C." But, by the time C gets B, B has been set to A. Likewise, by the time D gets C, C has been set to B, which, as we stated above, has been set to A. This always@ block turns B, C, and D into A. Blocking assignments are used when specifying combinational logic (see Section 1.5).

1.4 always@(posedge Clock) Blocks

always@(posedge Clock) ("always at the **positive edge** of the clock") or always@(negedge Clock) ("always at the **negative edge** of the clock") blocks are used to describe **Sequential Logic**, or Registers. Only <= (non-blocking) assignments should be used in an always@(posedge Clock) block. Never use = (blocking) assignments in always@(posedge Clock) blocks. Only use always@(posedge Clock) blocks when you want to infer an element(s) that changes its value at the positive or negative edge of the clock.

For example, consider Figure 1, a recreation of Program 2 that uses posedge Clock as its sensitivity list. Figure 1 is also known as a shift register. The completed always@ block is shown in Program 4.

Figure 1 A shift register



1.5 always@(*) Blocks

always@(*) blocks are used to describe **Combinational Logic**, or Logic Gates. Only = (blocking) assignments should be used in an always@(*) block. Never use <= (non-blocking) assignments in always@(*) blocks. Only use always@(*) block when you want to infer an element(s) that changes its value as soon as one or more of its inputs change.

[1] This point might be confusing. We said that non-blocking statements happen in parallel. Yet, they are useful for specifying *sequential* logic? In digital design, sequential logic doesn't refer to things happening in parallel or a sequence, as we have been discussing, but rather to logic that has *state*.

Program 4 A shift register, using <= assignments inside of an `always@(posedge Clock)` block

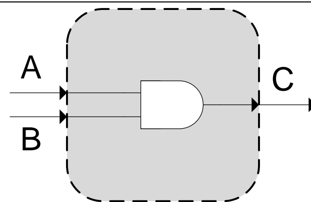
```
1 always @(posedge Clock) begin
2     B <= A;
3     C <= B;
4     D <= C;
5 end
```

Always use '*' (star) for your **sensitivity list** in `always@( * )` blocks. The sensitivity list specifies which signals should trigger the elements inside the `always@` block to be updated. For example, given 3 wires `A`, `B` and `C`, we can create an `and` gate through Program 5, and shown graphically in Figure 2.

Program 5 An `and` gate inside of an `always@( * )` block

```
1 always @(A or B) begin
2     C = A & B;
3 end
```

Figure 2 The `and` gate produced by Program 5 (this is a normal `and` gate!)



Program 5 specifies that "when A or B change values, update the value of every element inside the `always@( * )` block. In this case, the only element inside the `always@( * )` block is `C`, which in this case is assigned the `and` of `A` and `B`. A very common bug is to introduce an **incomplete sensitivity list**. See Program 6 for two examples of incomplete sensitivity lists.

Program 6 An `and` gate with an incomplete sensitivity list (**this is incorrect!**)

```
1 always @(A) begin
2     C = A & B;
3 end
```

```
1 always @(B) begin
2     C = A & B;
3 end
```

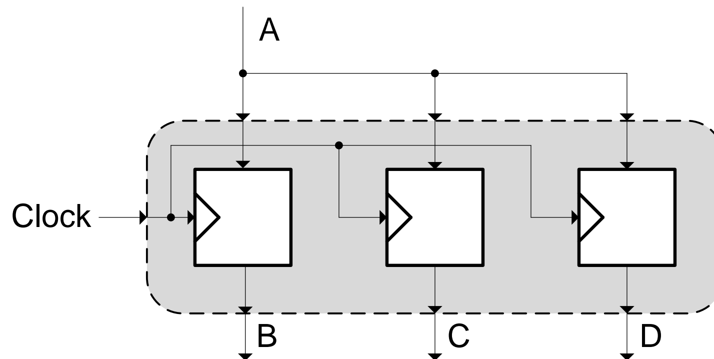
In Program 6, the first example produces an `and` gate that only updates its output `C` when `A` changes. If `B` changes, but `A` does not change, `C` does not change because the `always@(A)` block isn't executed. Likewise, the second example produces an `and` gate that doesn't react to a change in `A`. **Incomplete sensitivity lists are almost NEVER what you want!** They introduce very hard-to-find bugs. As such, we use `always@( * )`. The '*' is shorthand for `always@(A or B)` in our examples. In other words, '*' sets the sensitivity list to any values that can have an impact on a value(s) determined by the `always@( * )` block. '*' provides a bug-free shorthand for creating complete sensitivity lists.

1.6 Pitfalls

You might be wondering what happens if you don't follow the conventions set forth in Sections 1.4 and 1.5. The following are some easy-to-make mistakes in Verilog that can have a dramatic [and undesired] effect on a circuit.

1. Consider the shift register from Figure 1. If you place `=` assignments inside of an `always@(posedge Clock)` block to produce the shift register, you instead get the parallel registers shown in Figure 3 and Program 7. You might also get one register, whose output is tied to `B`, `C` and `D`. Both possible outcomes are equivelent. These circuit make sense, but don't create shift registers! (As shift registers are common construct, we assume that you wanted to create a shift register)

Figure 3 Parallel registers



Program 7 Parallel registers, using `=` assignments inside of an `always@(posedge Clock)` block

```
1 always @(posedge Clock) begin
2     B = A;
3     C = B;
4     D = C;
5 end
```

2. The opposite example (shown in Program 8), where we place `<=` assignments inside of `always@( * )` is less pronounced. In this case, just consider what type of circuit you want to create: do you want all statements to be executed in parallel or in 'sequence' (see Section 1.2 and 1.3)? In the `always@( * )`, the distinction between `<=` and `=` is sometimes very subtle, as the point of `always@ ( * )` is to trigger at indefinite times (unlike the very definite `posedge Clock`). We recommend `=` in conjunction with `always@( * )` to establish good convention (as `=` was originally meant to be associated with combinational logic).

Program 8 `<=` assignments inside of `always@( * )` blocks

```
1 always @( * ) begin
2     B <= A;
3     C <= B;
4     D <= C;
5 end
```

3. Consider the case of incompletely specified sensitivity lists. An incompletely specified sensitivity list, as discussed in Section 1.5, will create an `always@` block that doesn't always set/update its elements when it should. In truth, synthesis tools will often know what you mean if you provide an incomplete sensitivity list, and pretend that your sensitivity list was complete. This is **not** the

case with simulation tools (like ModelSim), however. ModelSim will not correct your sensitivity list bugs, and your simulations will be plagued with odd errors. Furthermore, the synthesis tools catching your errors is not guarenteed. An easy way to avoid these potential problems is to use `always@( * )` as opposed to `always@(Input1 or Input 2 or ...)`.

4. Lastly, a very subtle point which perhaps has the potential to cause the most frustration is **latch generation**. If you don't assign **every** element that **can** be assigned inside an `always@( * )` block **every** time that `always@( * )` block is executed, a latch (similar to a register but much harder to work with in FPGAs) will be inferred for that element. This is **never** what you want and is a terrible place for bugs. As this is subtle, it is somewhat hard to visualize. Consider Program 9.

Program 9 An `always@( * )` block that will generate a latch for `C`

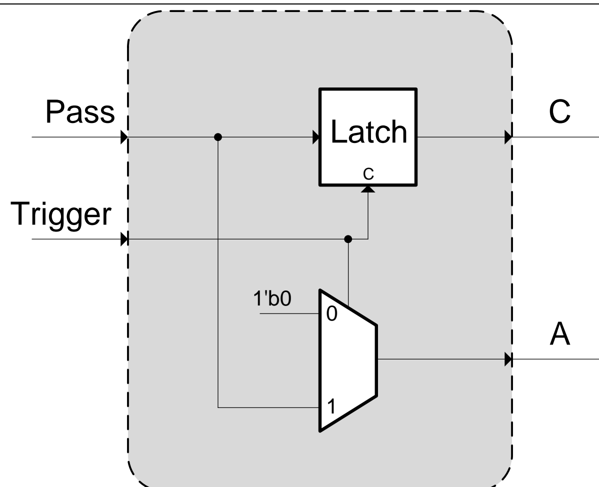
```
1 wire Trigger, Pass;
2 reg A, C;
3
4 always @( * ) begin
5     A = 1'b0;
6     if (Trigger) begin
7         A = Pass;
8         C = Pass;
9     end
10 end
```

In Program 9, `A` and `C` are both assigned in at least one place inside the `always@` block.

`A` is always assigned at least once. This is because the first line of the `always@` block specifies a default value for `A`. This is a perfectly valid assignment. It ensures that `A` is always assigned with each execution of the `always@` block.

`C` on the other hand is **not** always assigned. When `Trigger = 1'b1`, the `if` statement 'executes' and both `A` and `C` get set. If `Trigger = 1'b0`, however, the `if` is skipped. `A` is safe, as it was given a default value on the first line of the `always@` block. `C` on the other hand, doesn't get assigned at all when this happens. As such, a latch is inferred for `C`. The erroneous circuit depicted in Program 9 is shown in Figure 4.

Figure 4 The circuit generated by Program 9 (this is an erroneous circuit!)



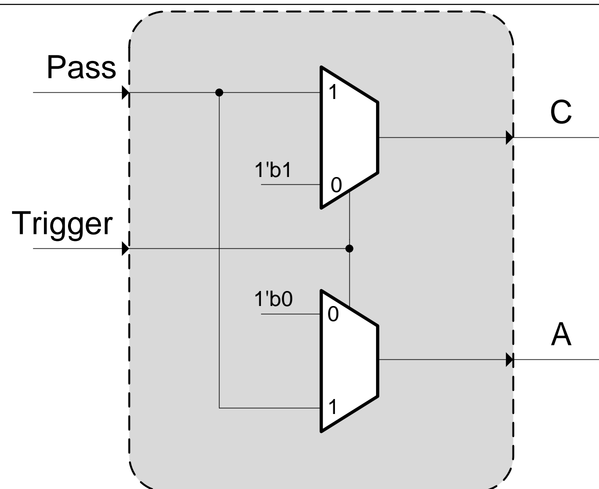
To fix this problem, we must make sure that `C` gets set every time the `always@` block is 'executed.' A simple way to force this is to add another default value, depicted in Program 10 and shown in Figure 5.

Program 10 An `always@( * )` block that will not generate latches

```
wire Trigger, Pass;
reg A, C;

always @( * ) begin
    A = 1'b0;
    C = 1'b1;
    if (Trigger) begin
        A = Pass;
        C = Pass;
    end
end
```

Figure 5 The circuit generated by Program 10 (this is correct!)



Default values are an easy way to avoid latch generation, however, will sometimes break the logic in a design. As such, other ways of ensuring that each value always gets set are going to be worth looking into. Typically, they involve proper use of the Verilog `else` statement, and other flow constructs.

Know that setting a `reg` to itself is not an acceptable way to ensure that the `reg` always gets set. For example, `C = C;` enjected into the top of the `always@( * )` block in Program 9 will not supress latch generation. In every 'execution' of an `always@( * )` block, each value that is assigned in **at least one place** must be assigned to a non-trivial value during **every** 'execution' of the `always@( * )` block.