Synthesis of Sequential Logic from Behavioral Code

Topics in This Set

Sequential Logic Basics, Differences with Combinational Logic

Coding of Registers

Simple example: counters.

Sequential shifter example.

Sequential v. Combinational Logic

It's all about the flip-flop.

Storage devices are the distinguishing feature . . .
. . . that differentiate combinational and sequential logic.

Combinational Logic

Outputs only depend on current inputs.

No flip-flops, registers, or other devices that have state.

Sequential Logic

Outputs depend on current **and past** input.

Has state. Typically state kept by flip-flops and/or registers.

State changes usually synchronized with a clock.

Why sequential logic is so much harder than combinational logic.

Inference: There isn't an operator that synthesizes to a flip-flop ...
... as there is, say, with + for addition.

Logic Design: Designs are trickier ...
... it's not just what will happen ...
... it's not even just when it will happen ...
... but whether *this* happens before *that* or after *that*.

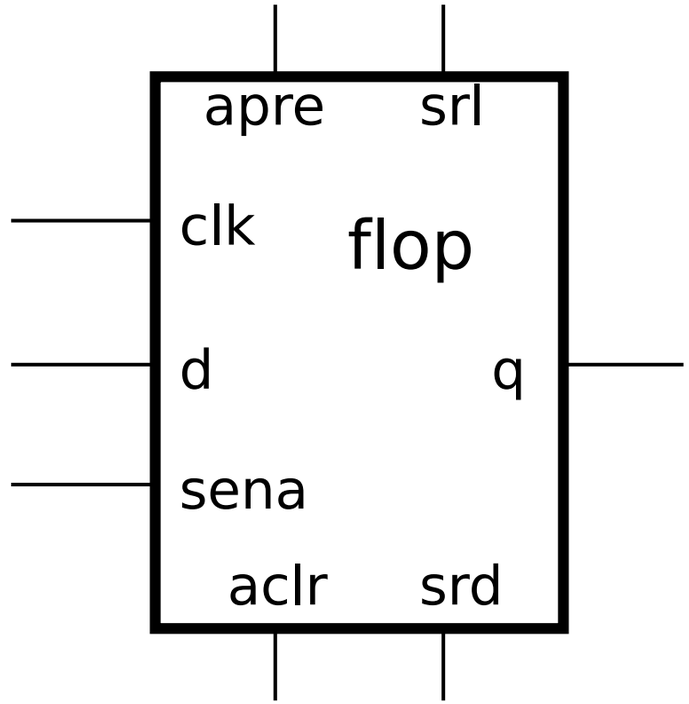
Verilog Subtleties: Those ignorant of Verilog timing may be tormented...
... with seemingly arbitrary errors or behavior.

Inference of Registers

Genus' Generic Flip-Flop: `flop`.

`flop` features:

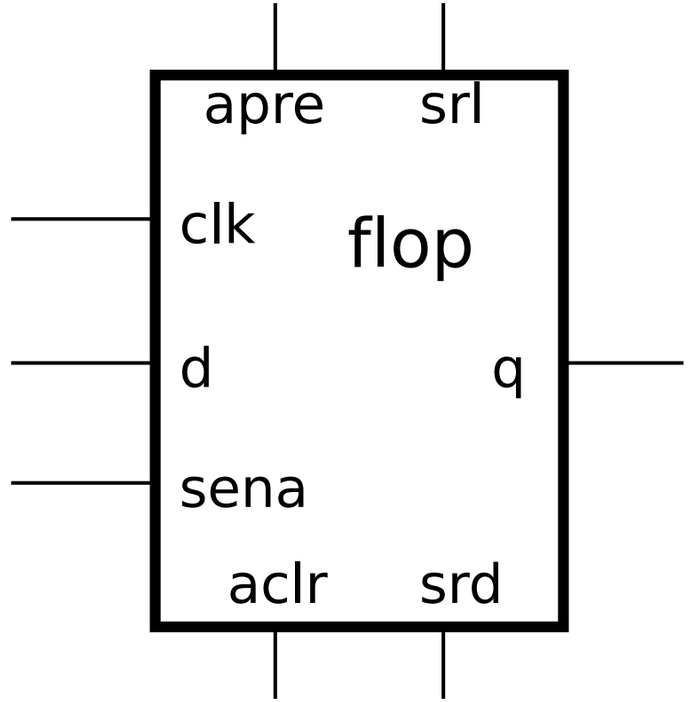
- Is positive edge triggered (`clk`).
- Has input `d` and output `q`.
- Has asynchronous preset (`apre`) and clear (`alcr`).
- Has a sync. enable (`sena`) input.



Inference and Technology Mapping

During elaboration `flop` used for all inferred edge-triggered registers.

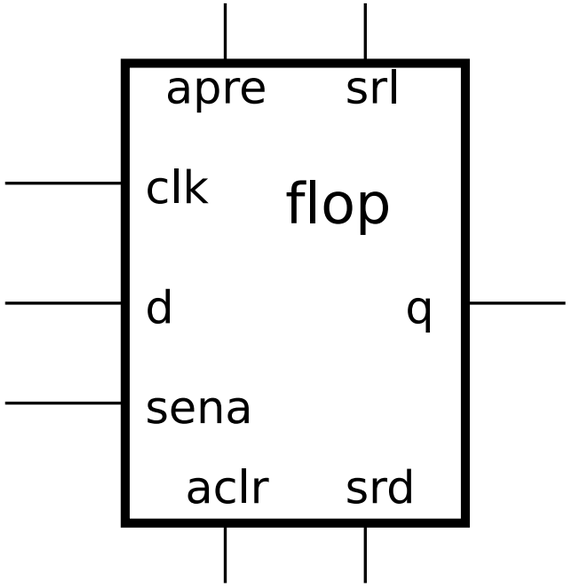
During technology mapping `flop` replaced with registers from technology library.

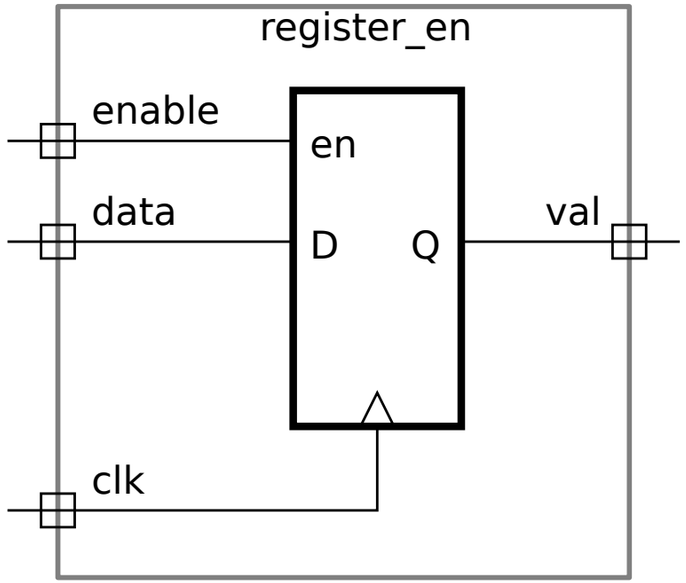


Classroom Hardware Diagrams

The term *register* will be used for one or more flip-flops.

For inferred and optimized hardware...
... will use streamlined diagrams, omitting unused inputs:





Edge-Triggered Flip-Flop Inference

Inference

Selecting a hardware component corresponding to a piece of Verilog behavioral code.

Performed by a synthesis program.

Relationship between behavioral Verilog and inferred hardware . . .
. . . is determined by the synthesis program. . .
. . . not by the Verilog standard or any other standard document.

Edge-Triggered Flip-Flop Inference Rules

These Inference Rules

Based on Cadence Genus

Reference: *Genus HDL Modeling Guide* Version 19.1, May 2019.

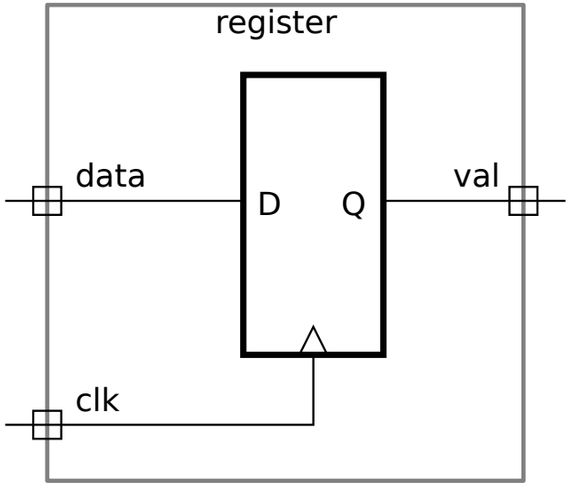
For inference of edge-triggered register `R` clocked by `clk`:

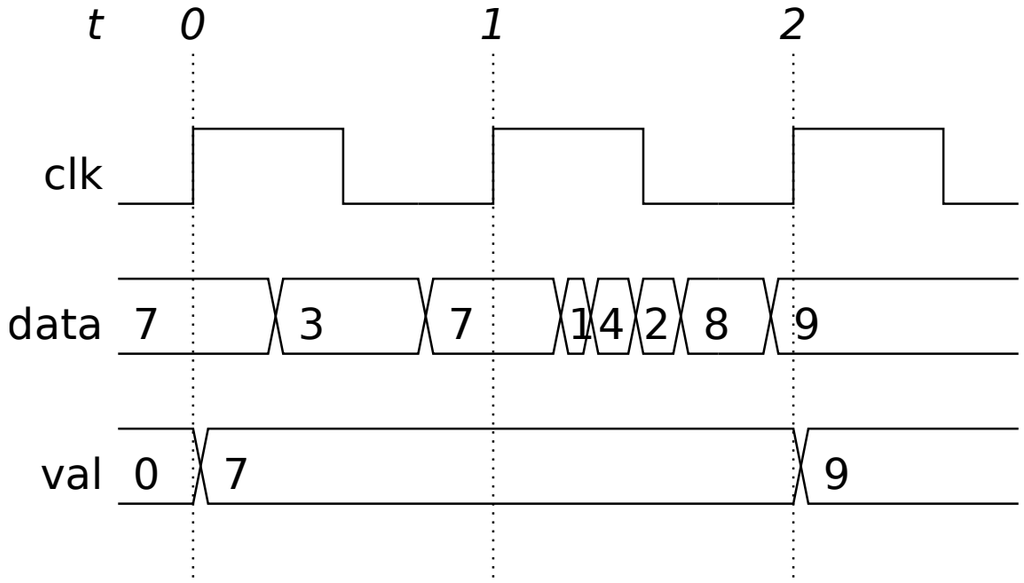
- `R` must be a variable type.
- `R` must be assigned in exactly one always block ...
  ... and must be either consistently blocking (`R=d;`) ...
  ... or consistently non-blocking (`R<=d;`).
- The always block must start with `always` or `always_ff`.
- The always must be followed by an event control of the form `@( posedge clk, ...)`.

```
module register
  #( int w = 16 )
   ( output logic [w-1:0] R,
     input uwire [w-1:0] d,
     input uwire clk );

   always_ff @( posedge clk ) R <= d;

endmodule
```

Simple Edge-Triggered Register

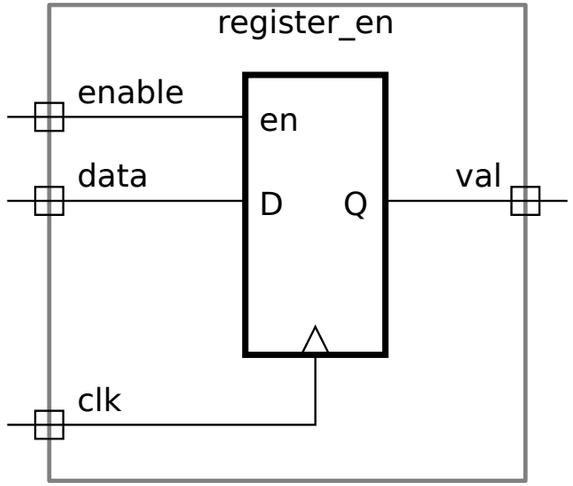
```
module register
  #( int width = 16 )
   ( output logic [width-1:0] val,
     input uwire [width-1:0] data,
     input uwire clk );

   always_ff @( posedge clk ) val <= data;
endmodule
```

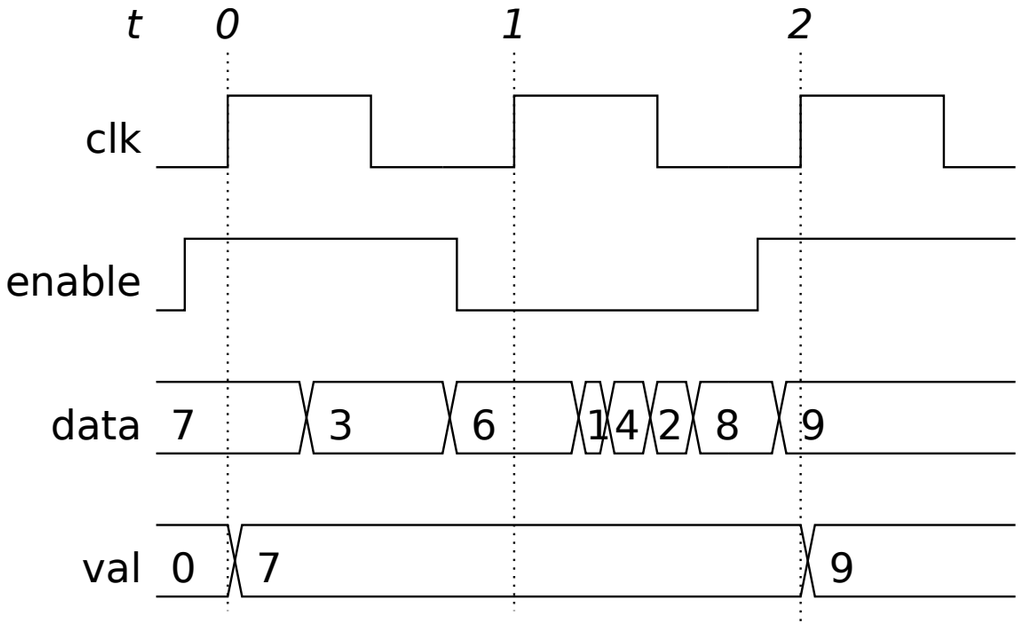




Register with Enable

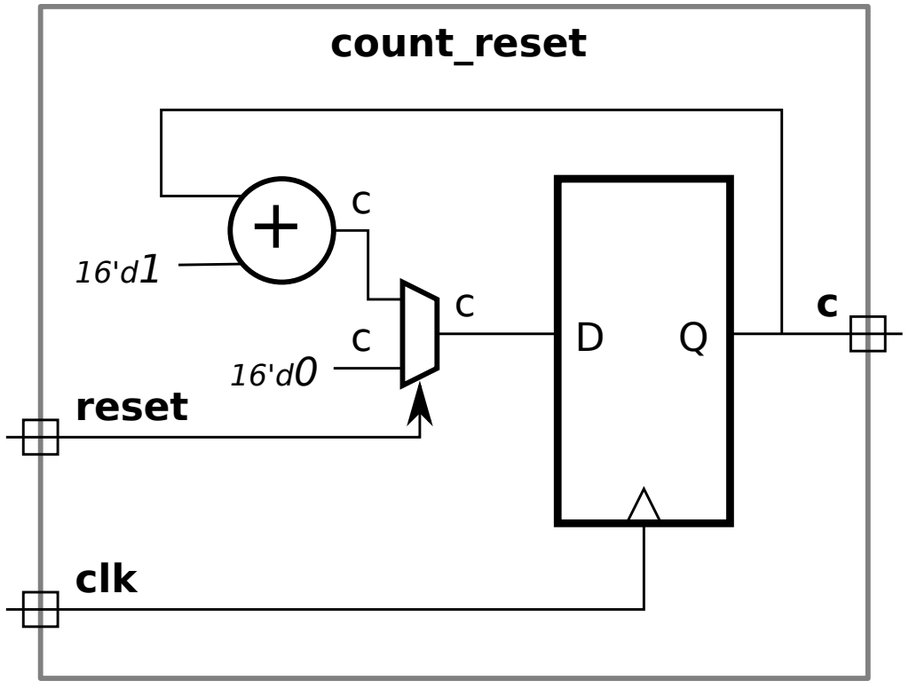
```
module register_en
  #( int width = 16 )
   ( output logic [width-1:0] val,
     input uwire enable,
     input uwire [width-1:0] data,
     input uwire clk );

   always_ff @( posedge clk )
     if ( enable ) val <= data;

endmodule
```





Clock with Reset

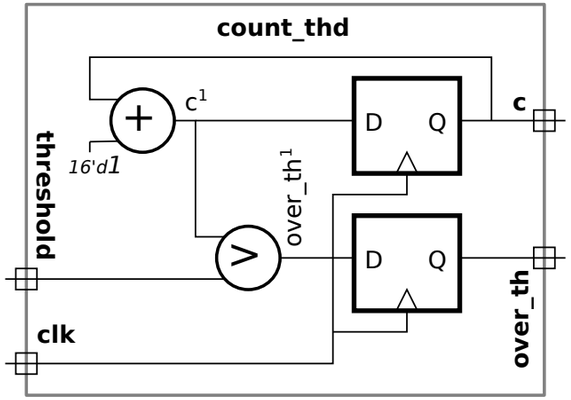
Note multiple `c` values.

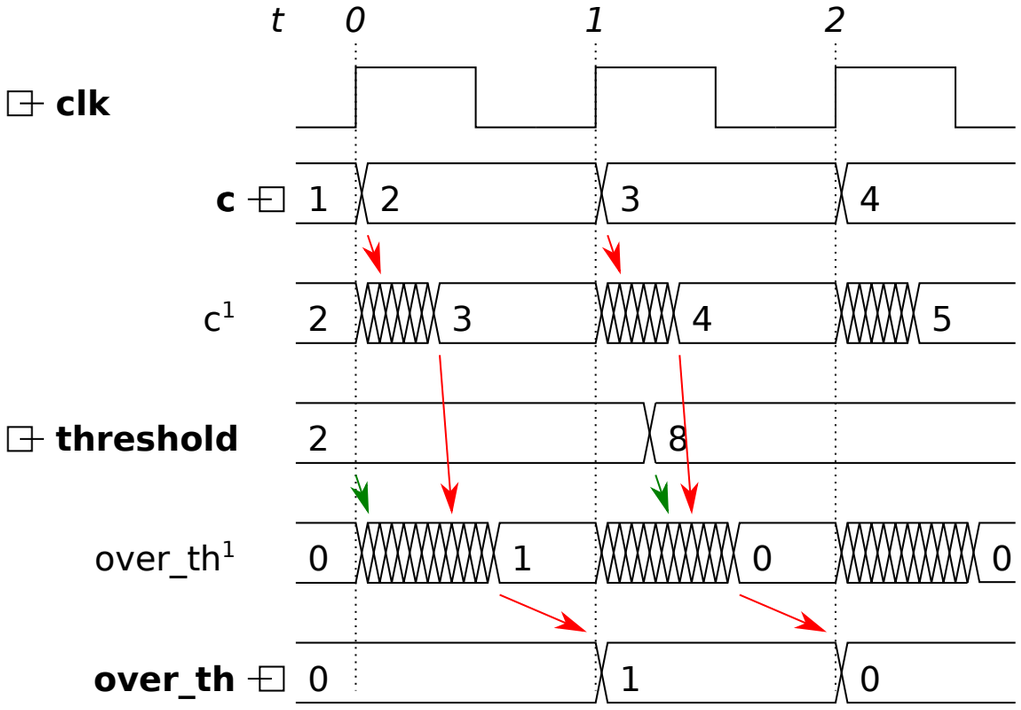



```
module count_reset
  #( int bits = 16 )
   ( output logic [bits-1:0] c,
     input uwire reset,
     input uwire clk );

   always_ff @( posedge clk ) if ( reset ) c <= 0; else c <= c + 1;

endmodule
```

Threshold Output

```
module count_thd
  #( int bits = 16 )
   ( output logic [bits-1:0] c,
     output logic over_th,
     input uwire [bits-1:0] threshold,
     input uwire clk );

   always_ff @( posedge clk )
     begin
        c = c + 1;
        over_th =  c > threshold;
     end
endmodule
```

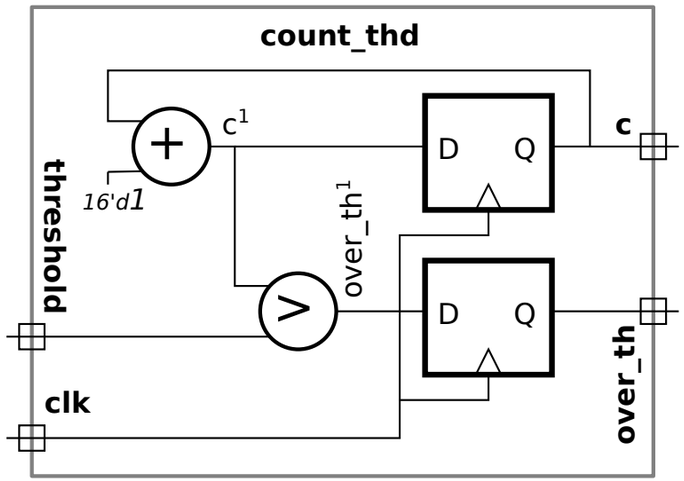


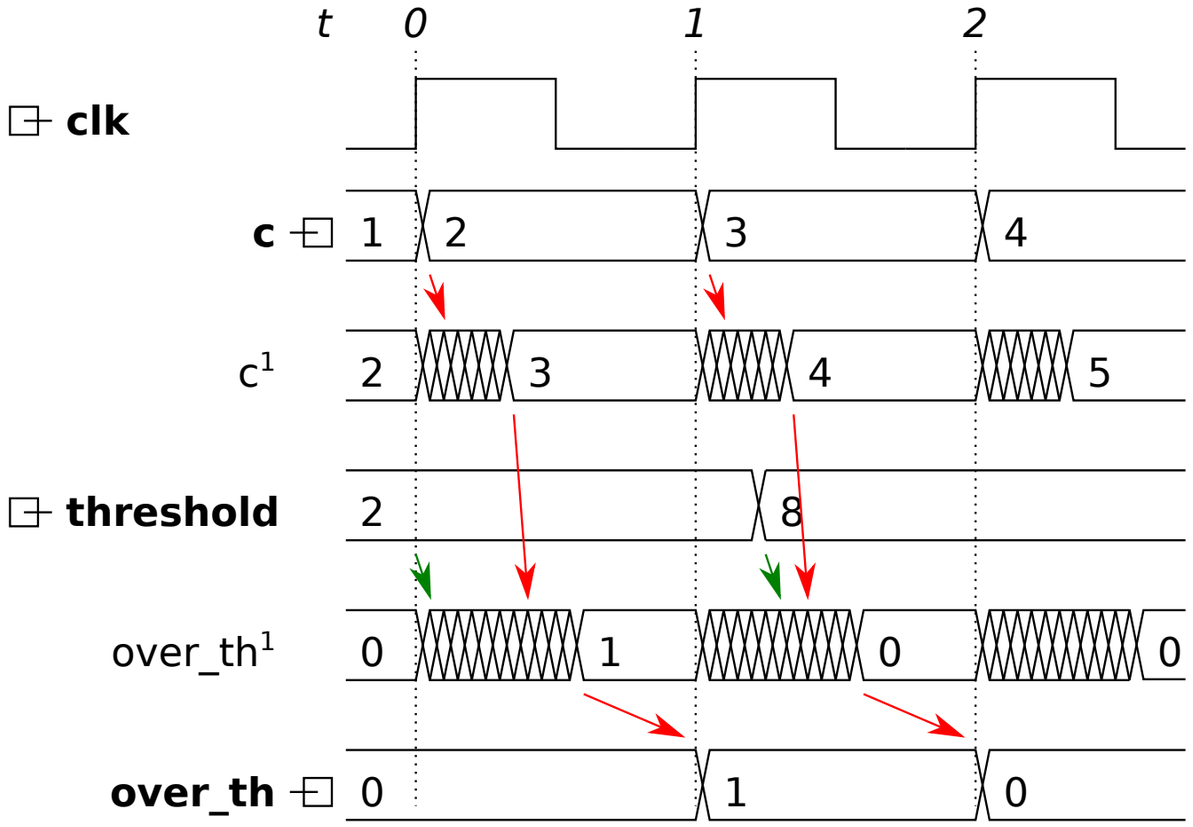


Two Issues:

Critical path through adder/comparison unit.

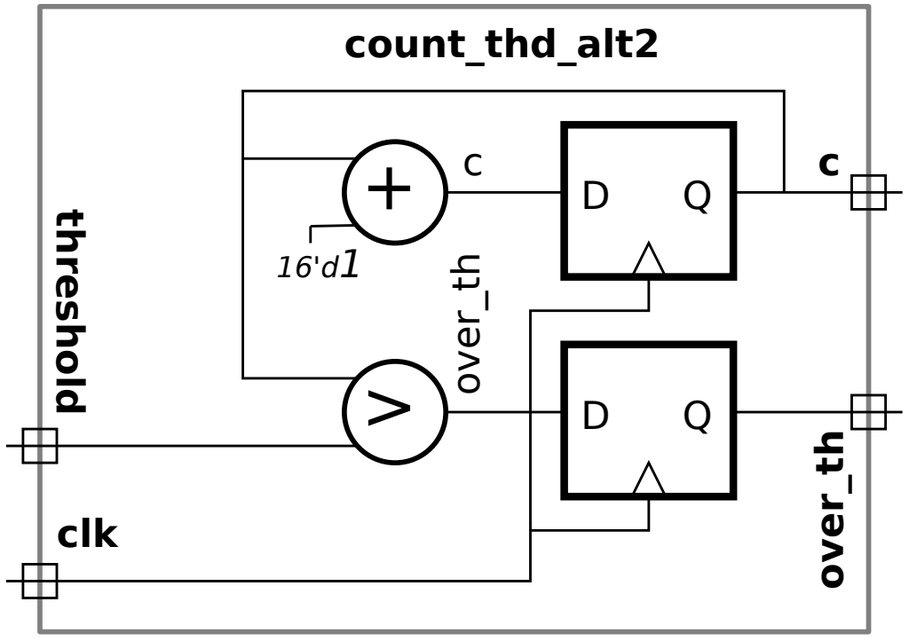
Do we really want a flip-flop for `over_th`?





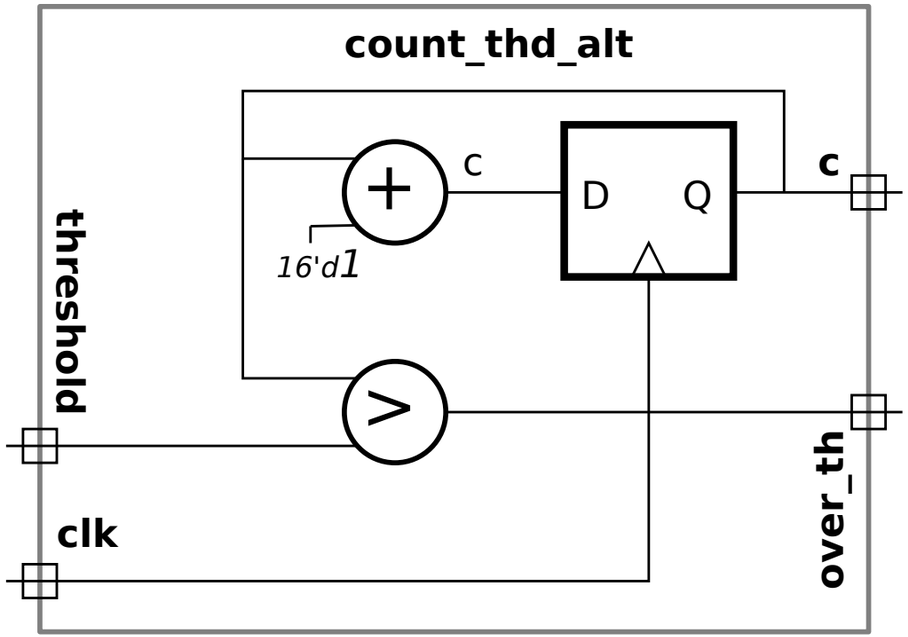
Fix critical path issue.

```
module count_thd_alt2
  #( int bits = 16 )
   ( output logic [bits-1:0] c,
     output logic over_th,
     input uwire [bits-1:0] threshold,
     input uwire clk );

   always_ff @( posedge clk )
     begin
        over_th = c > threshold;
        c = c + 1;
     end

endmodule
```



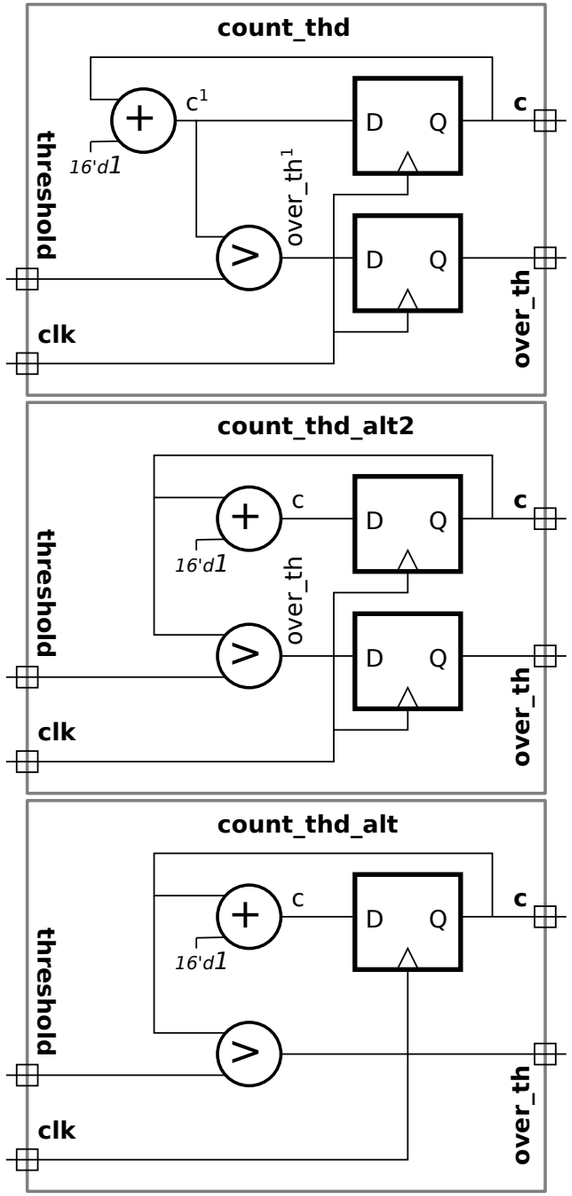
React any time to threshold, not just at positive edge.

```
module count_thd_alt
  #( int bits = 16 )
   ( output logic [bits-1:0] c,
     output logic over_th,
     input uwire [bits-1:0] threshold,
     input uwire clk );

   always_ff @( posedge clk ) c <= c + 1;

   always_comb over_th = c > threshold;

endmodule
```



Comparison

```
module count_thd #( int bits = 16 )
   ( output logic [bits-1:0] c, output logic over_th,
     input uwire [bits-1:0] threshold, input uwire clk );
   always_ff @( posedge clk ) begin
        c = c + 1;
        over_th =  c > threshold;
     end
endmodule

module count_thd_alt2 #( int bits = 16 )
   ( output logic [bits-1:0] c, output logic over_th,
     input uwire [bits-1:0] threshold, input uwire clk );
   always_ff @( posedge clk ) begin
        over_th = c > threshold;
        c = c + 1;
     end
endmodule

module count_thd_alt #( int bits = 16 )
   ( output logic [bits-1:0] c, output logic over_th,
     input uwire [bits-1:0] threshold, input uwire clk );

   always_ff @( posedge clk ) c <= c + 1;
   always_comb over_th = c > threshold;
endmodule
```

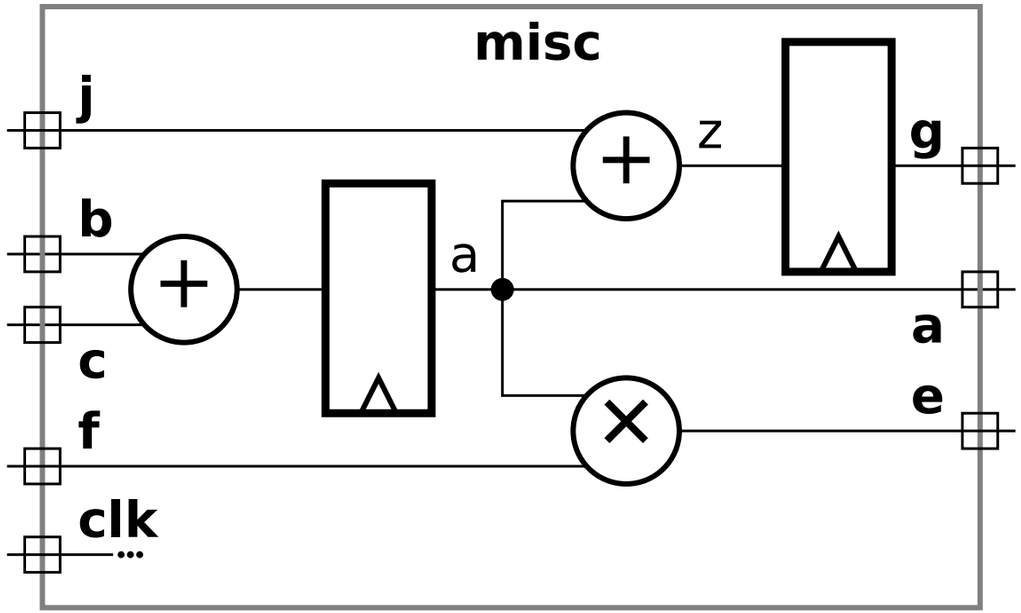


Show inferred logic for the following:

```
module misc #( int n = 8 )
   ( output logic [n-1:0] a, g, e,
     input uwire [n-1:0] b, c, j, f,
     input uwire clk );

   logic [n-1:0] z;

   always_ff @( posedge clk ) begin
      a <= b + c;   // Note: nonblocking assignment.
      z = a + j;
      g = z;
   end

   always_comb begin
      e = a * f;
   end

endmodule
```

Show inferred logic for the following: (solution)

```
module misc #( int n = 8 )
   ( output logic [n-1:0] a, g, e,
     input uwire [n-1:0] b, c, j, f,
     input uwire clk );

   logic [n-1:0] z;

   always_ff @( posedge clk ) begin
      a <= b + c;   // Note: nonblocking assignment.
      z = a + j;
      g = z;
   end

   always_comb begin
      e = a * f;
   end

endmodule
```

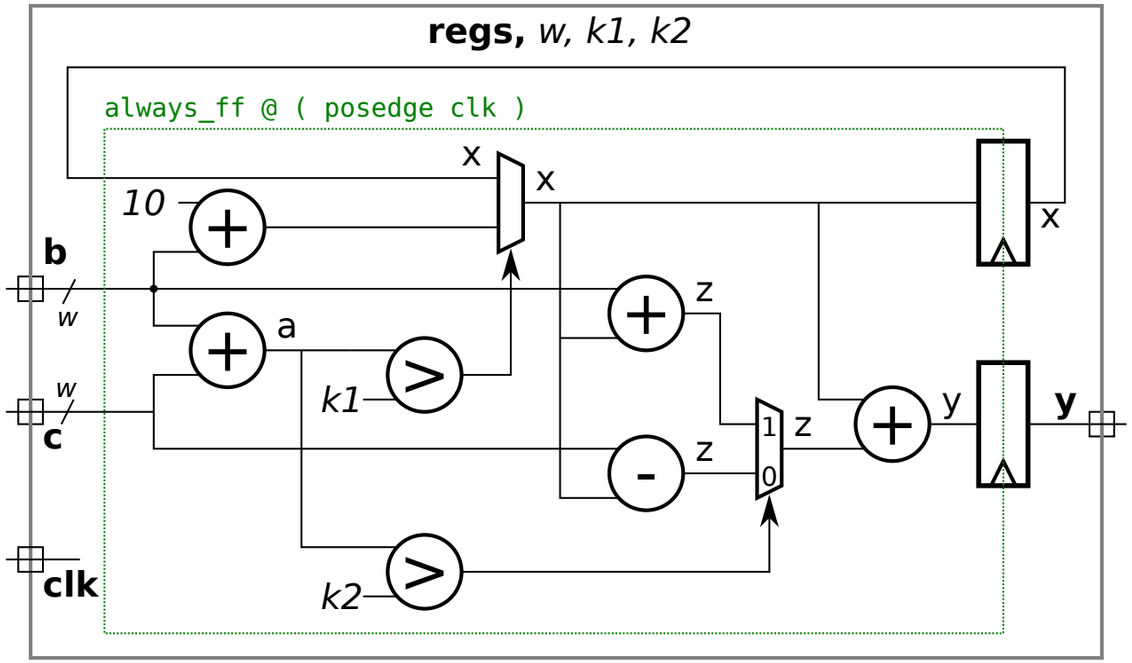


Show inferred logic for the following:

```
module regs
  #( int w = 10, int k1 = 20, int k2 = 30 )
   ( output logic [w-1:0] y,
     input logic [w-1:0] b, c,
     input uwire clk );

   logic [w-1:0] a, x, z;

   always_ff @( posedge clk ) begin

      a = b + c;
      if ( a > k1 ) x = b + 10;
      if ( a > k2 ) z = b + x; else z = c - x;
      y = x + z;

   end

endmodule
```

Show inferred logic for the following: (solution)

```
module regs
  #( int w = 10, int k1 = 20, int k2 = 30 )
   ( output logic [w-1:0] y,
     input logic [w-1:0] b, c,
     input uwire clk );

   logic [w-1:0] a, x, z;

   always_ff @( posedge clk ) begin

      a = b + c;
      if ( a > k1 ) x = b + 10;
      if ( a > k2 ) z = b + x; else z = c - x;
      y = x + z;

   end

endmodule
```

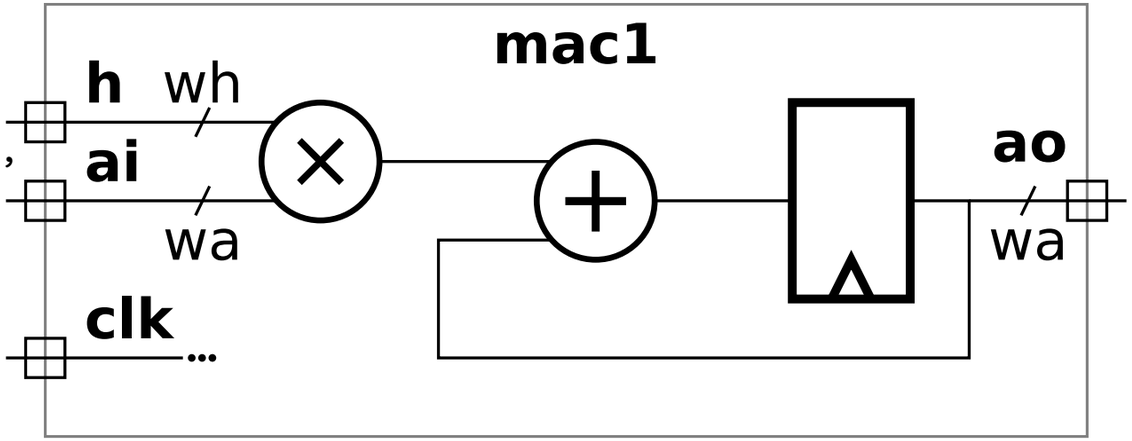


Consider the following similar multiply/accumulate modules:

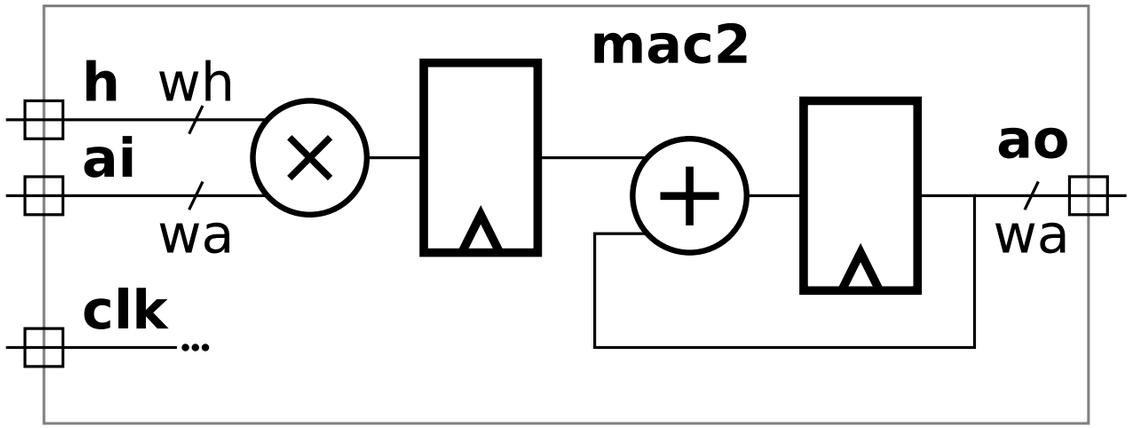
```
module mac1 #( int wa = 32, wh = 16 )
   ( output logic [wa-1:0] ao,
     input uwire [wh-1:0] h, input uwire [wa-1:0] ai,
     input uwire clk );

   always_ff @( posedge clk ) ao <= h * ai + ao;
endmodule

module mac2 #( int wh = 4, wa = 3 )
   ( output logic [wa-1:0] ao,
     input uwire [wh-1:0] h,  input uwire [wa-1:0] ai,
     input uwire clk );

   logic [wa-1:0] p;

   always_ff @( posedge clk ) begin
      p <= h * ai;
      ao <= p + ao;
   end
endmodule
```

Consider the following similar multiply/accumulate modules:

```
module mac1 #( int wa = 32, wh = 16 )
   ( output logic [wa-1:0] ao,
     input uwire [wh-1:0] h, input uwire [wa-1:0] ai,
     input uwire clk );

   always_ff @( posedge clk ) ao <= h * ai + ao;
endmodule
```




```
module mac2 #( int wh = 4, wa = 3 )
   ( output logic [wa-1:0] ao,
     input uwire [wh-1:0] h,  input uwire [wa-1:0] ai,
     input uwire clk );

   logic [wa-1:0] p;
   always_ff @( posedge clk ) begin
      p <= h * ai;
      ao <= p + ao;
   end
endmodule
```



Example: Sequential Shifter

Remember: We can build a w-bit logarithmic shifter using ...
... $\lceil \lg w \rceil$ 2^i-bit fixed-amount shifters and 2-input muxen ...
... for $i \in \{2^0, 2^1, 2^2, \ldots, 2^{\lceil \lg w \rceil - 1}\}$.

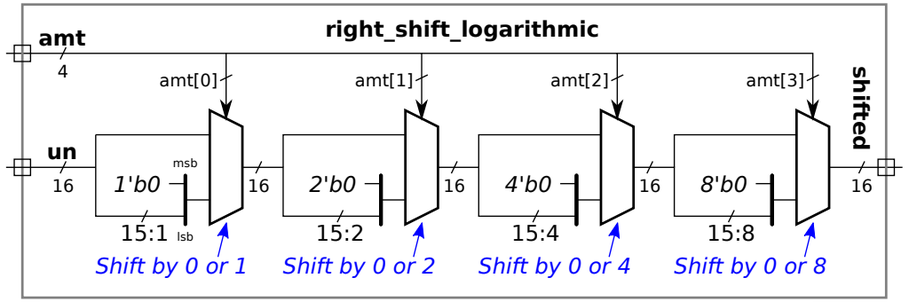
Why not use one fixed shifter and use it up to $w - 1$ times?

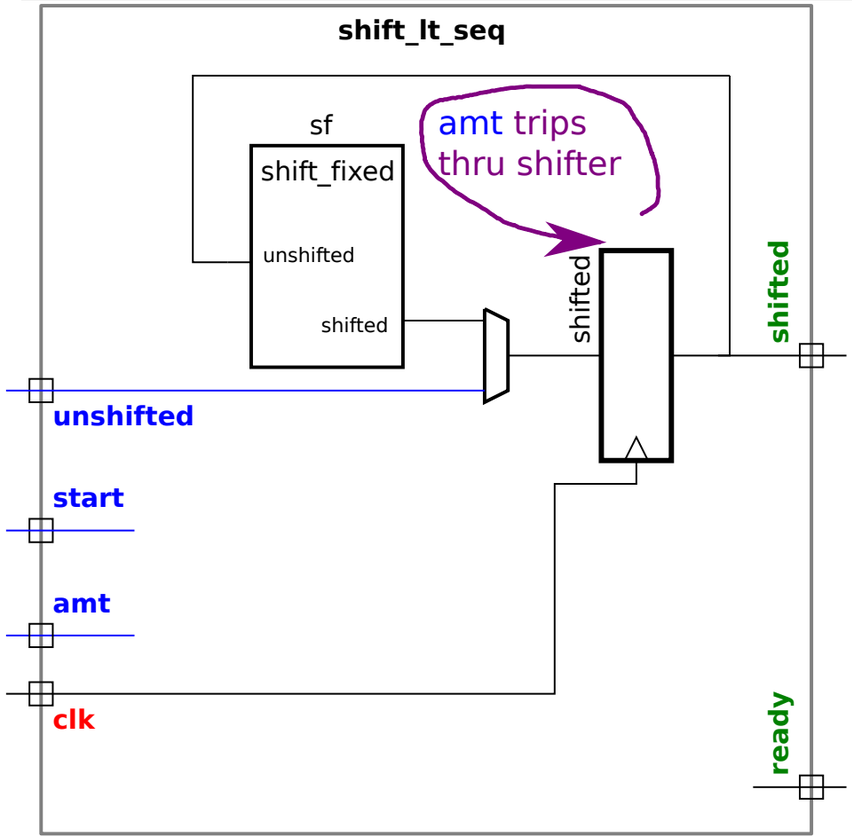
Why not use fewer than $\lceil \lg w \rceil$ shifters and muxen ...
... but use them multiple times?

We'll start with one fixed shifter.

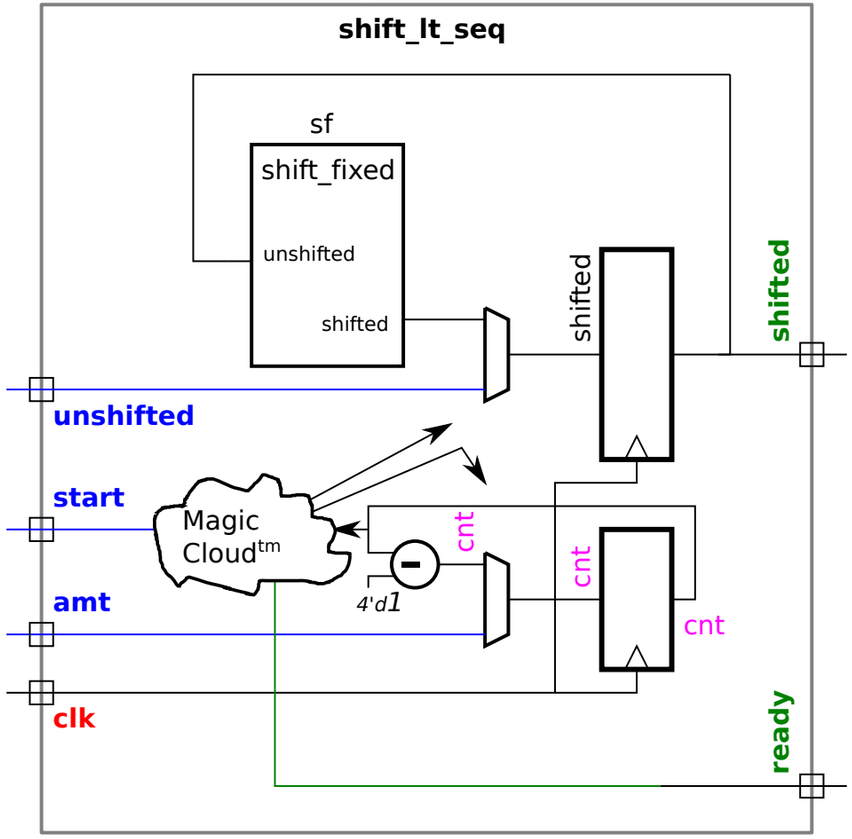
Idea sketch for sequential shifter.

Pass value through shifter `amt` times.





Idea sketch for sequential shifter.



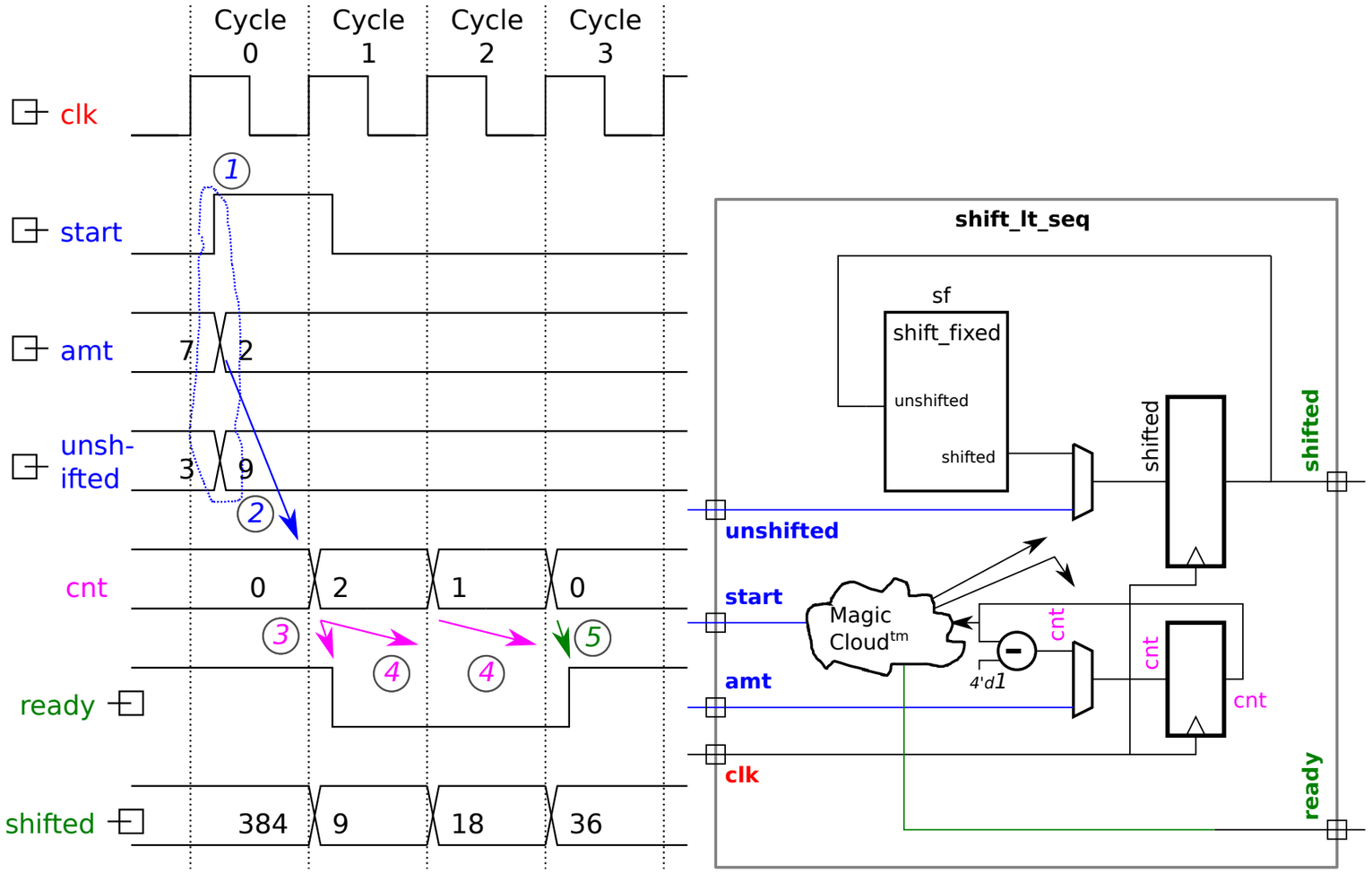
Use register `cnt` to count number of times.

Timing.

1: External device provides inputs.

Inputs assumed to be available...

... early in clock cycle.

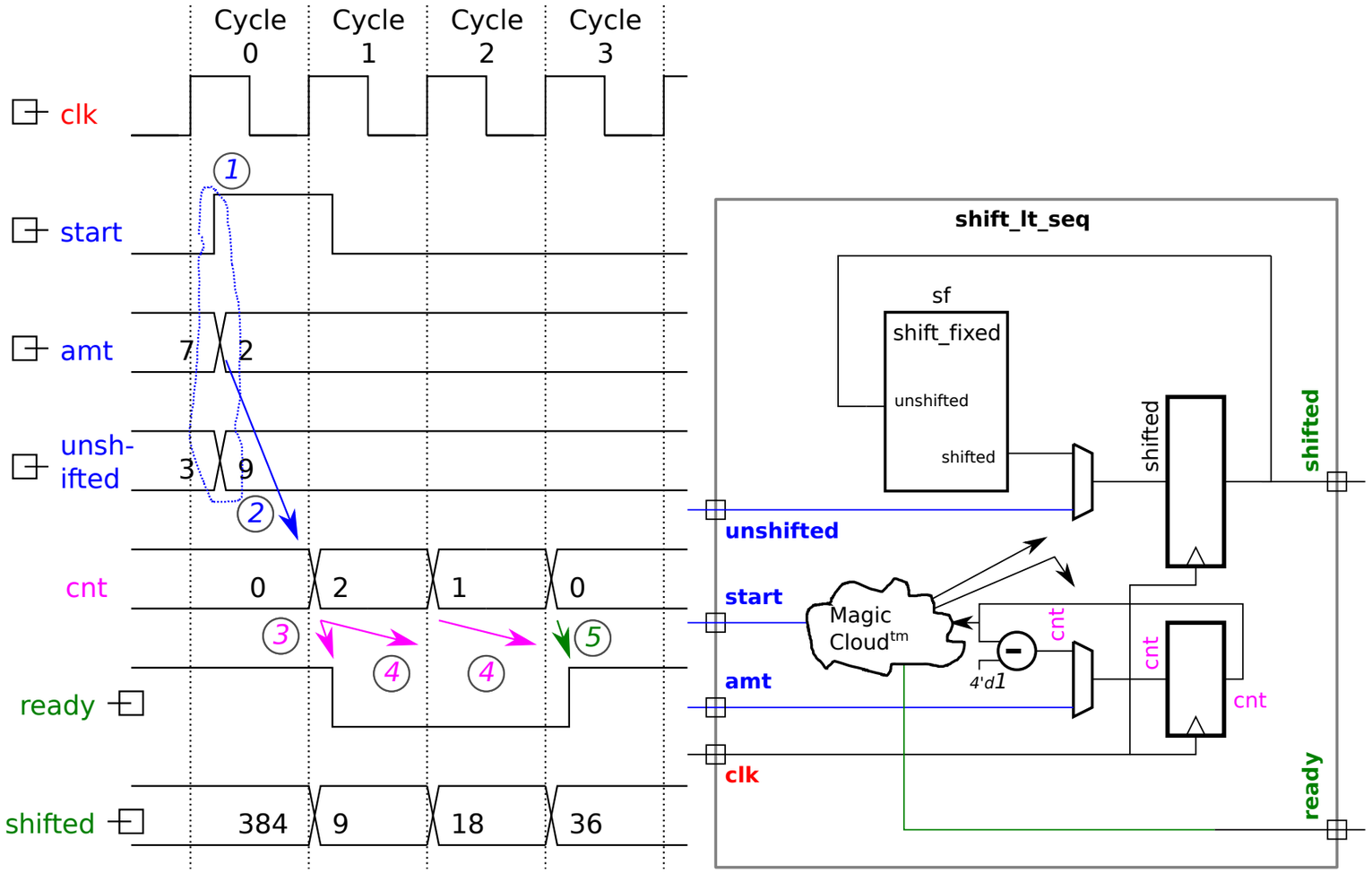


Timing.

2: At positive edge:

`cnt` initialized to `amt`.

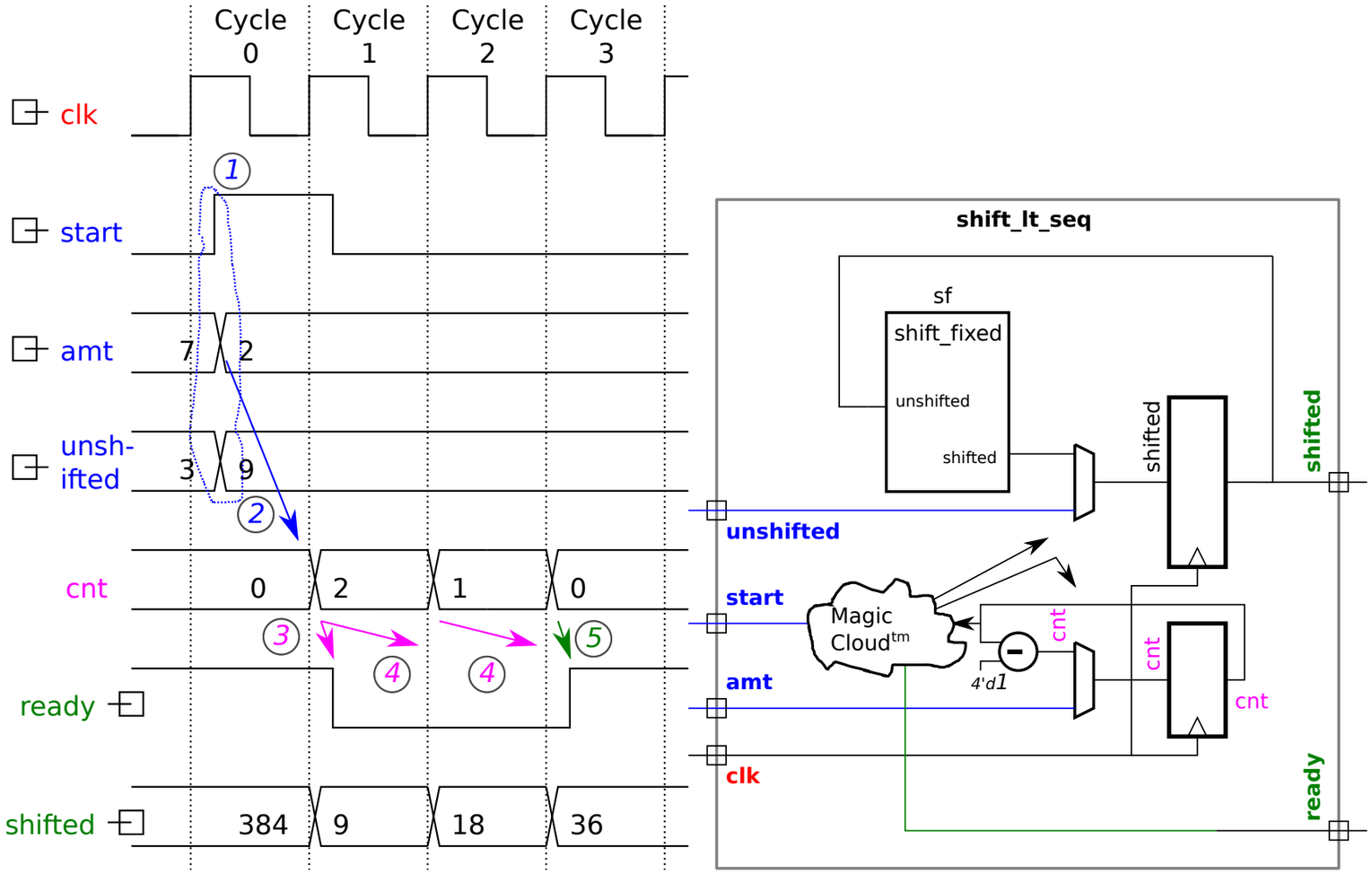
`shifted` initialized to `unshifted`.



Timing.

3: Early in Cycle 1:

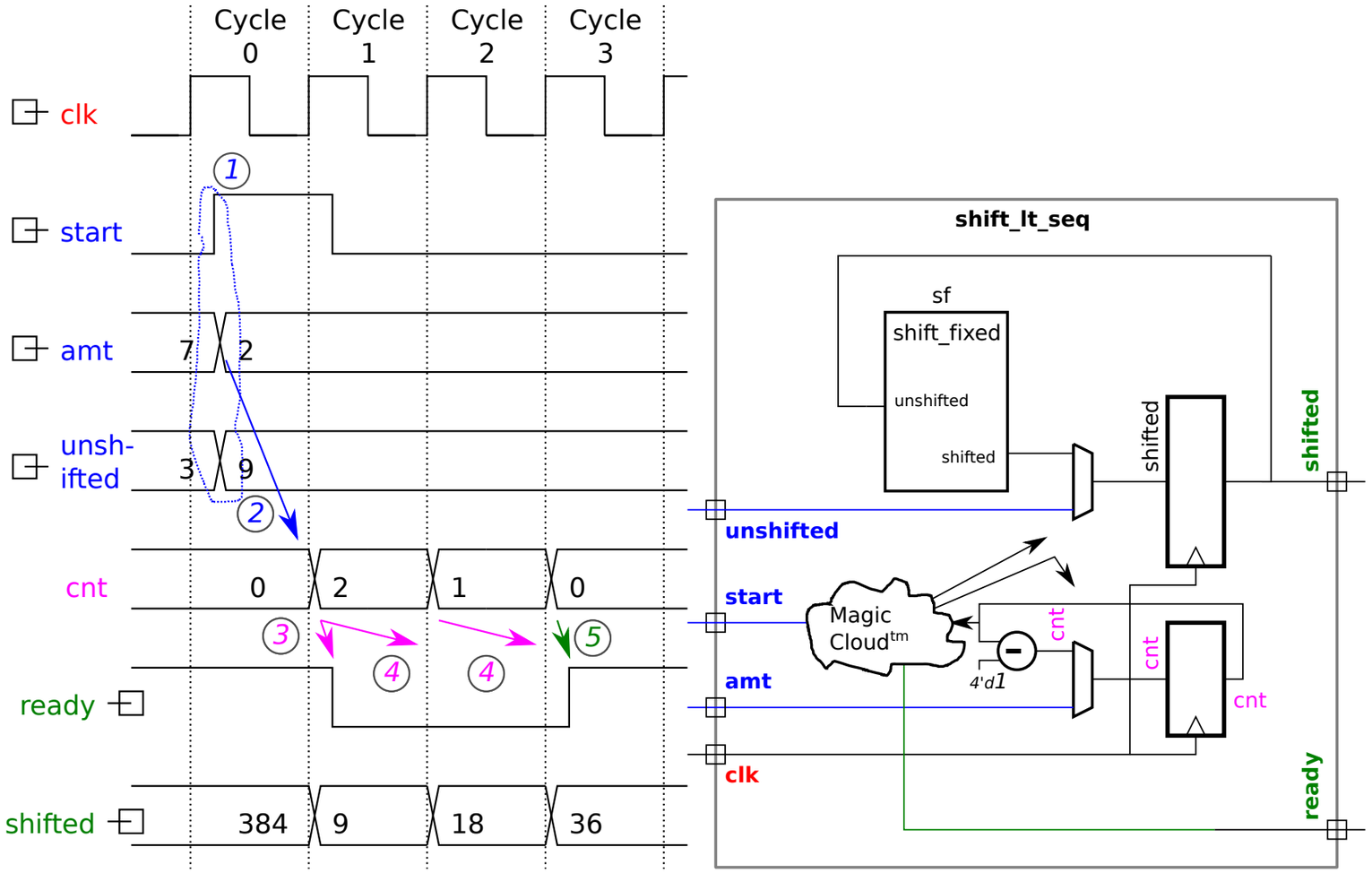
`ready` goes to zero.



Timing.

4: During cycles 1 and 2:

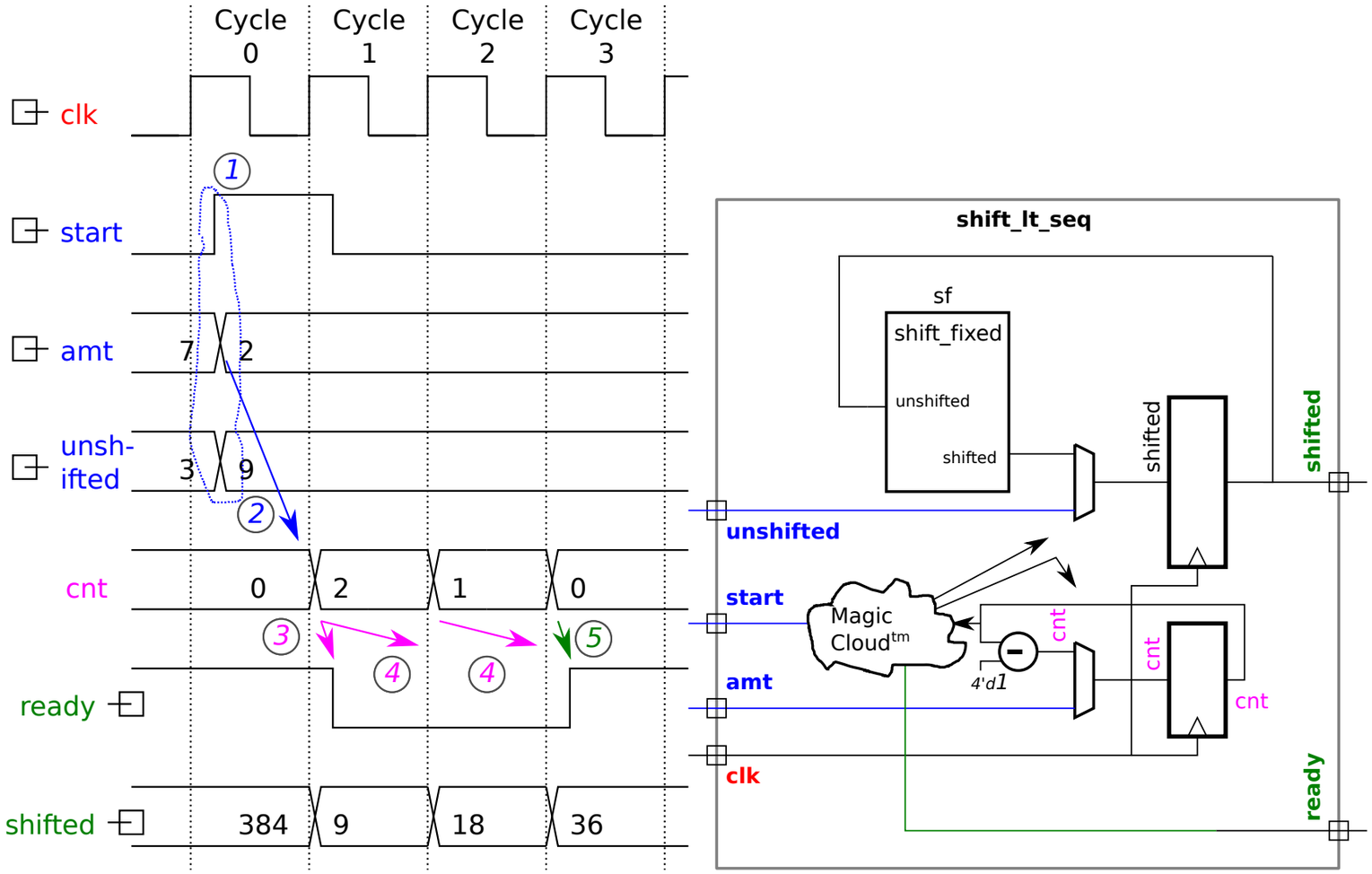
New value of count is computed, "shift" performed.



Timing.

5: Beginning of cycle 3:

Ready signal set to 1.



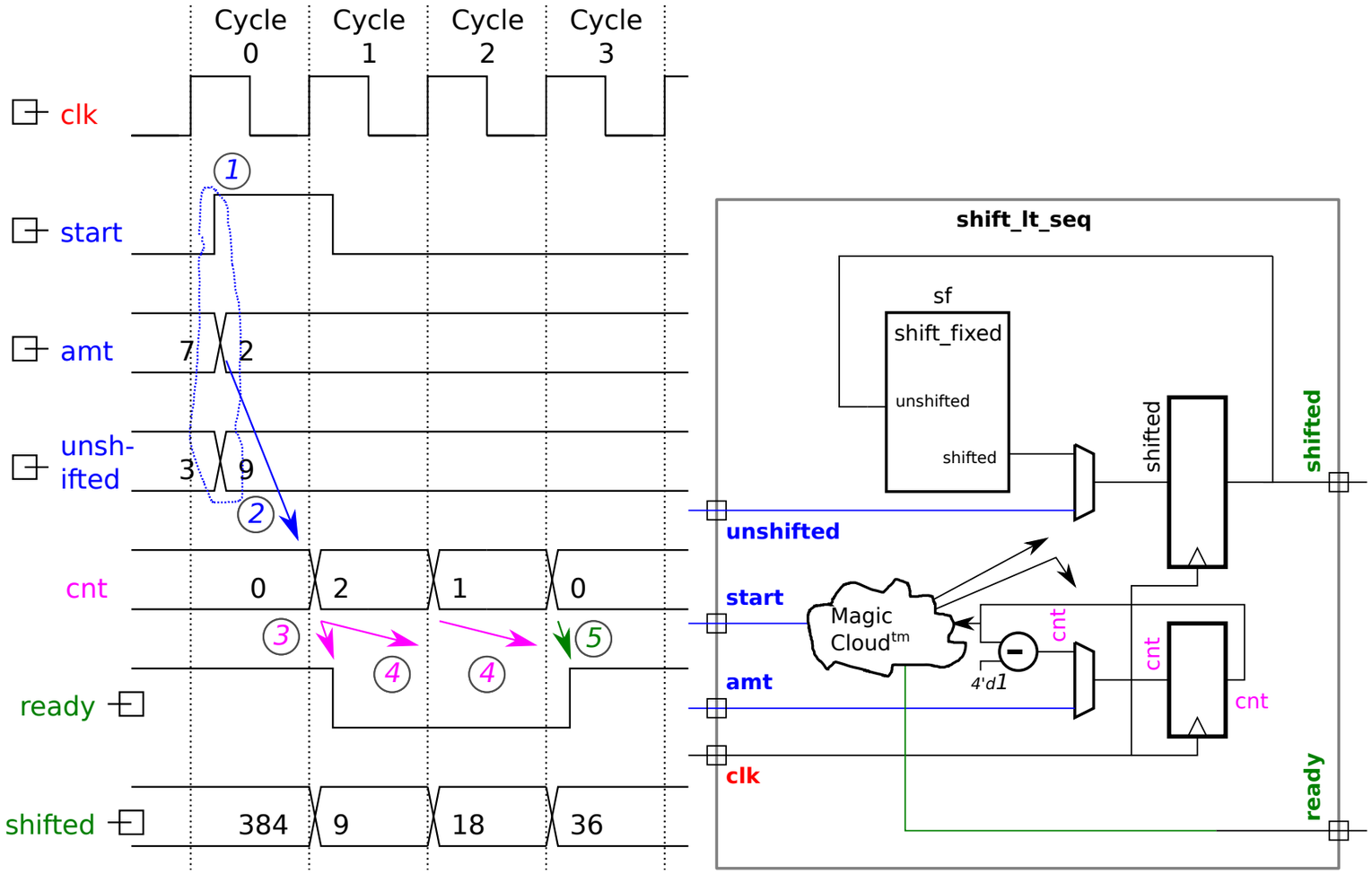
Notes about behavior.

Start signal must be stable at positive edge.

Inputs required to be available early in clock cycle.

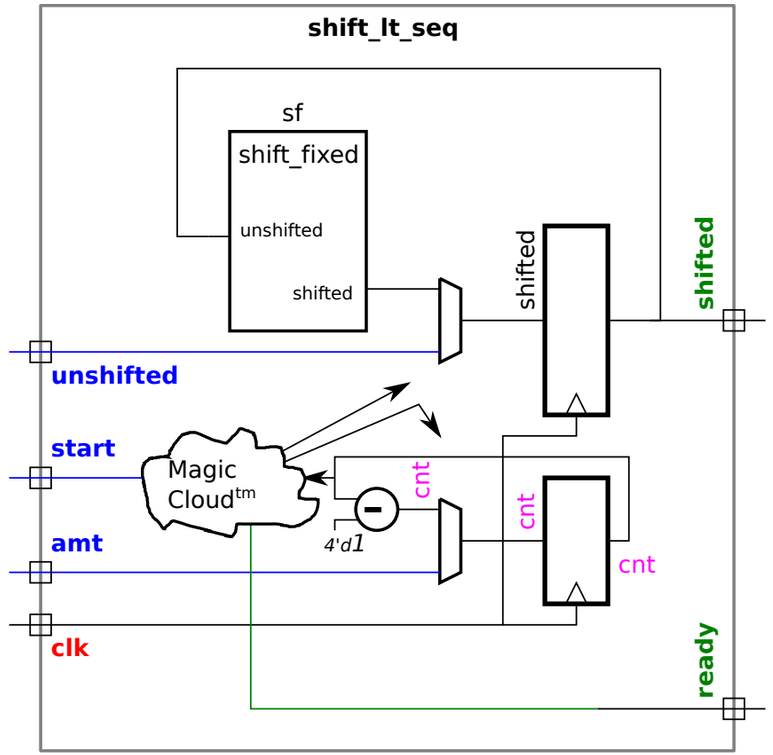
Result available at beginning of clock cycle.

Ready signal available early in clock cycle.



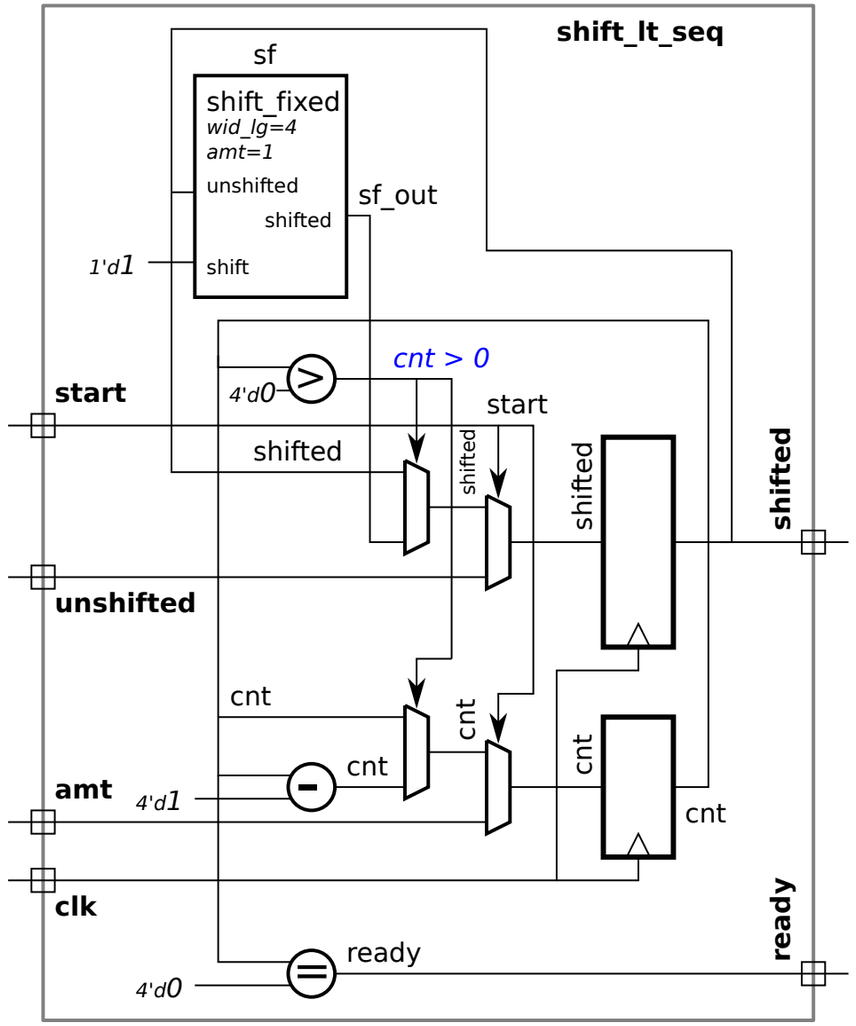
Sequential Shifter Verilog

```
module shift_lt_seq #( int c = 4, int w = 1 << c )
   ( output logic [w-1:0] shifted,       output uwire ready,
     input uwire [w-1:0] unshifted,      input uwire [c-1:0] amt,
     input uwire start, clk );

   uwire [w-1:0] sf_out;
   shift_fixed #(c,1) sf( sf_out, shifted, 1'b1 );  // Fixed Shifter

   logic [c-1:0] cnt;

   always_ff @( posedge clk )
      if ( start == 1 ) begin

         shifted = unshifted;          // Load a new item to shift ...
         cnt = amt;                    // .. and initialize amount.

      end else if ( cnt > 0 ) begin

         shifted = sf_out;             // Shift by one more bit ..
         cnt--;                        // .. and update count.

      end

   assign ready = cnt == 0;            // Set ready to 1 when count is zero.

endmodule
```

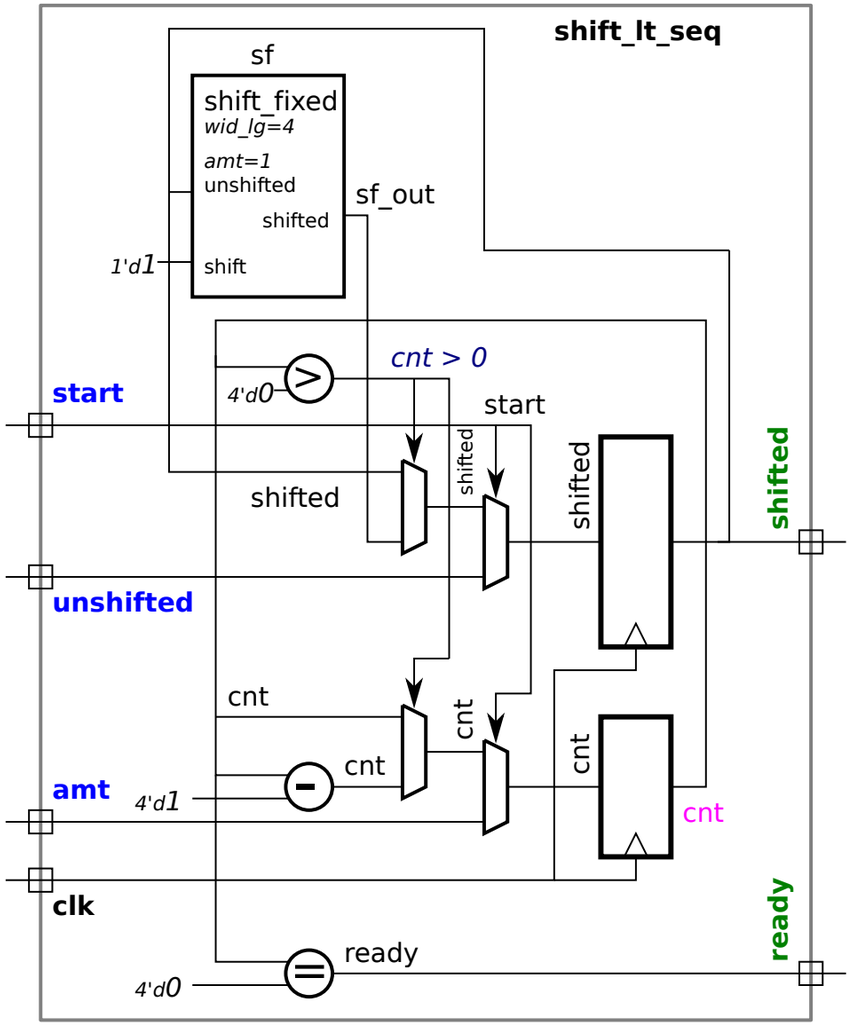


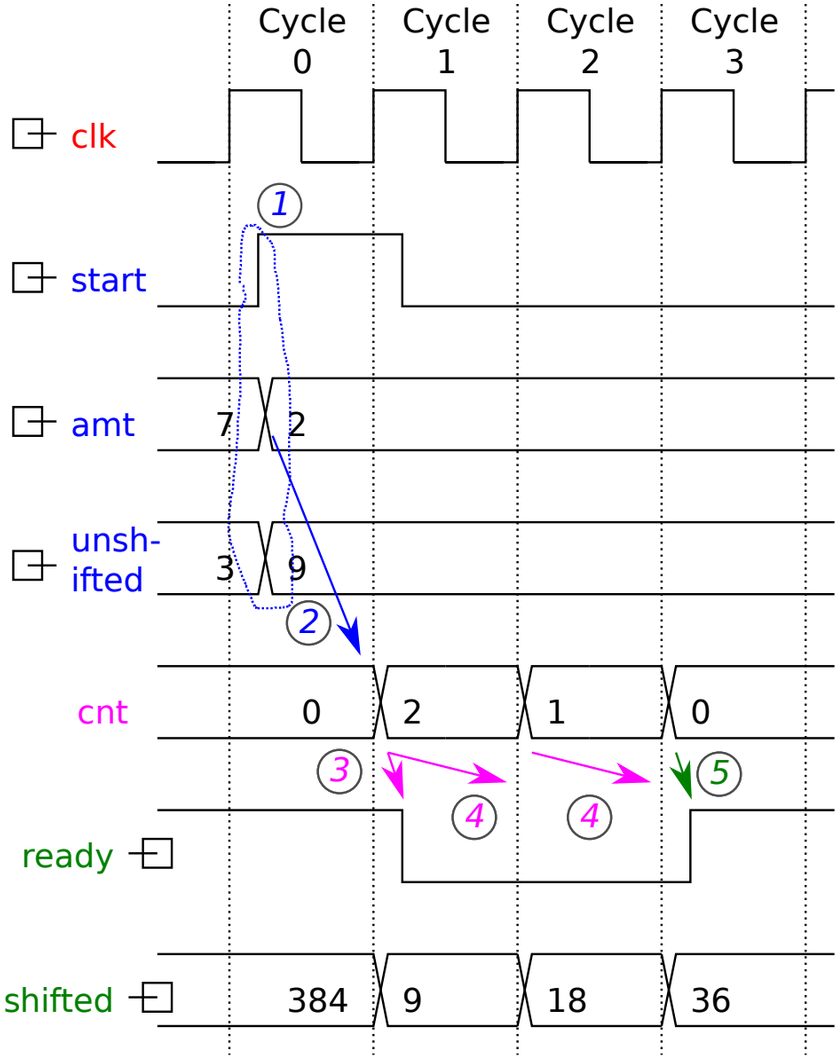
Inferred Hardware, No Optimization

```
module shift_lt_seq
  #( int c = 4, int w = 1 << c )
   ( output logic [w-1:0] shifted,
     output uwire ready,
     input uwire [w-1:0] unshifted,
     input uwire [c-1:0] amt,
     input uwire start, clk );

   uwire [w-1:0] sf_out;
   shift_fixed #(c,1) sf(sf_out,shifted,1'b1);

   logic [c-1:0] cnt;

   always_ff @( posedge clk )
      if ( start == 1 ) begin

         shifted = unshifted;
         cnt = amt;

      end else if ( cnt > 0 ) begin

         shifted = sf_out;
         cnt--;

      end else begin  shifted = shifted;  cnt = cnt;  end

   assign ready = cnt == 0;

endmodule
```



Inferred Hardware, No Optimization



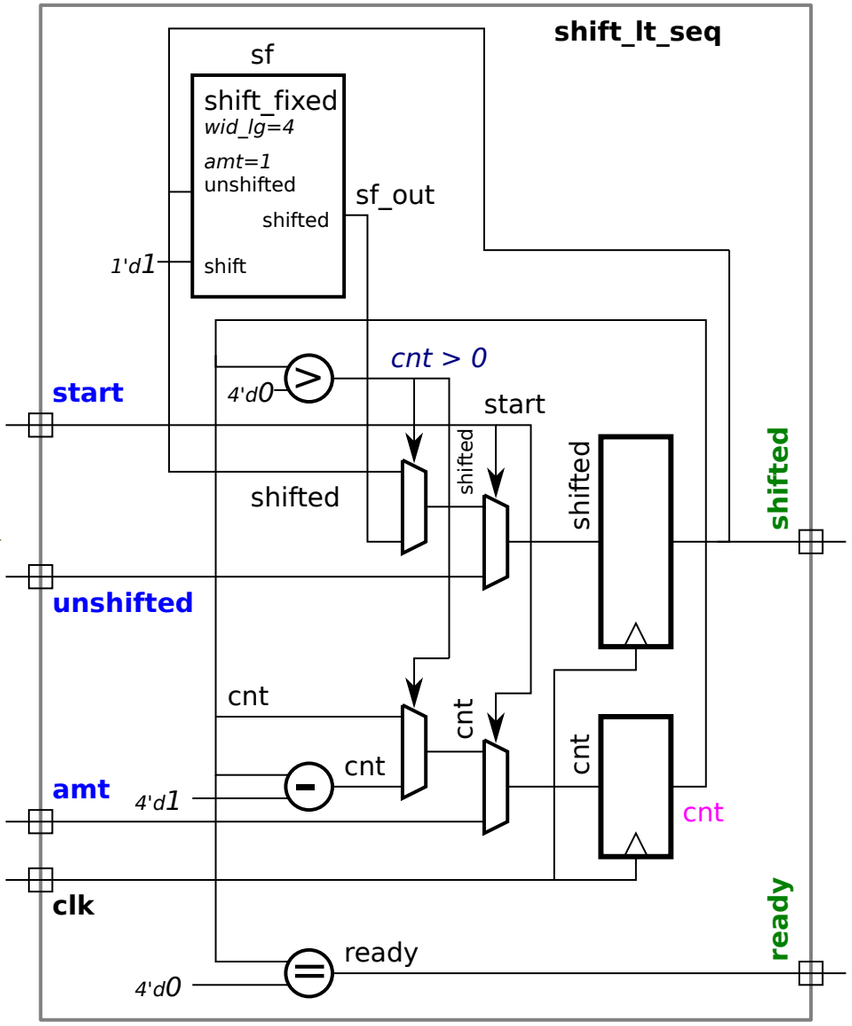


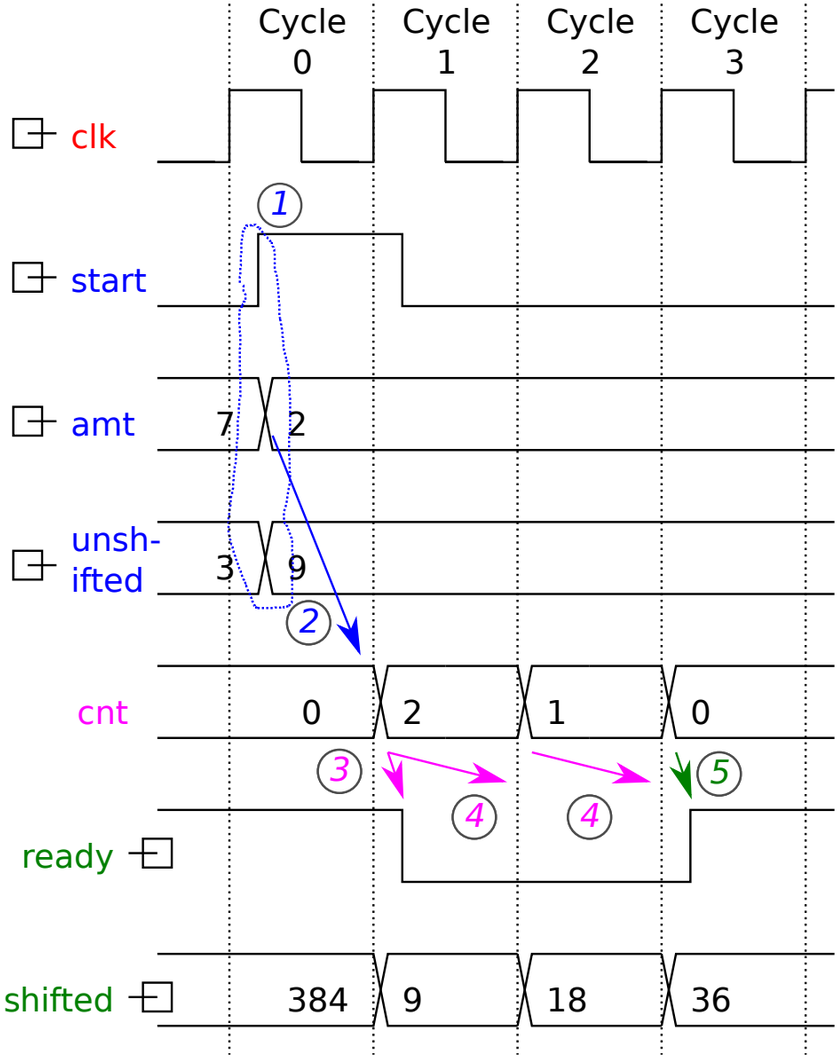
Pay Attention To

Setup delay: inputs to registers.

Operation delay: register to register.

Output delay: generation of the ready signal.





Streamlining and Optimization

Streamline hardware illustration to make it readable.

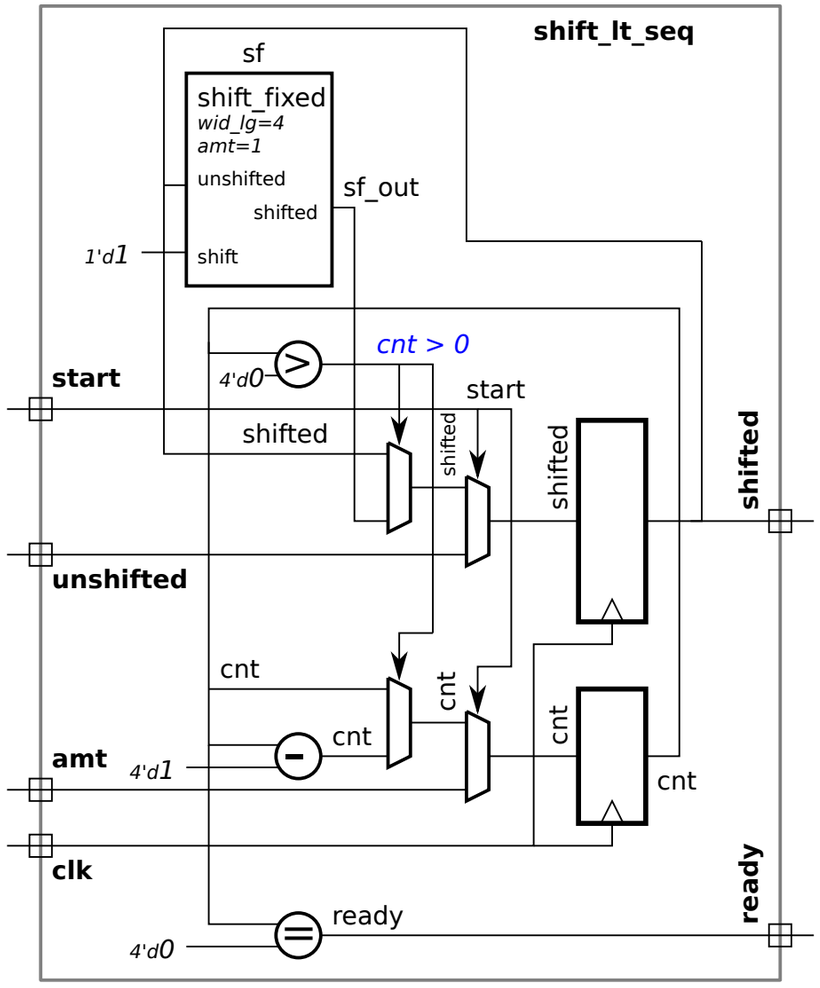
Include optimizations we hope synthesis program will make.

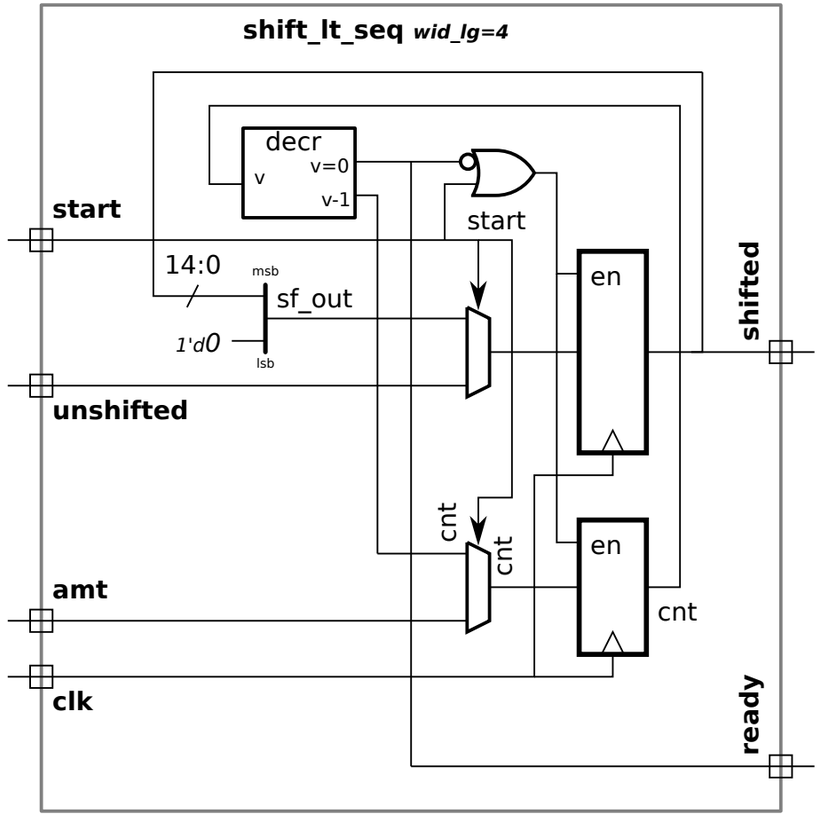
Optimization Opportunities

Use an enable for registers.

Shifter is just a bit renaming plus one zero.

The three operations on `cnt`, $c > 0$, $c - 1$, and $c == 0$ ...
... can all be done by the same logic.


shift_lt_seq
sf
shift_fixed
wid_lg=4
amt=1
unshifted
shifted
shift
sf_out
1'd1
cnt > 0
4'd0
start
shifted
shifted
shifted
shifted
unshifted
cnt
cnt
cnt
cnt
amt
4'd1
cnt
clk
ready
ready
4'd0


shift_lt_seq wid_lg=4
decr
v
v=0
v-1
start
start
14:0
msb
sf_out
1'd0
lsb
en
shifted
shifted
unshifted
cnt
cnt
en
amt
cnt
clk
ready

Cost Analysis

Cost of Optimized Hardware

Shifted and Cnt Registers (with enable): $10(w + \lg w)\,\mathrm{u_c}$.

Shifted and Cnt Muxen: $3(w + \lg w)\,\mathrm{u_c}$.

Decrement (cost of BHAs): $3 \lg w\,\mathrm{u_c}$.

The OR: $1\,\mathrm{u_c}$.

Total: $[13w + 16 \lg w + 1]\,\mathrm{u_c}$.

Cost of Logarithmic Shifter

The $\lg w$ multiplexors: $3w \lg w\,\mathrm{u_c}$.

Log shifter becomes more expensive at $w = 35$.