Synthesis of Sequential Logic from Behavioral Code

It's all about the flip-flop.

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

Why sequential logic is so much harder than combinational logic.

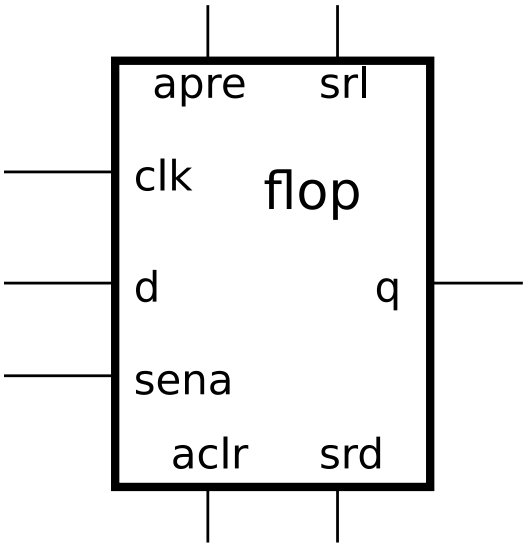
Inference: There isn't an operator that synthesizes to a flip-flop . . .
. . . as there is, say, with `+` for addition.

Logic Design: Designs are trickier . . .
. . . it's not just what will happen . . .
. . . it's also when it will happen.

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

Inference of Registers

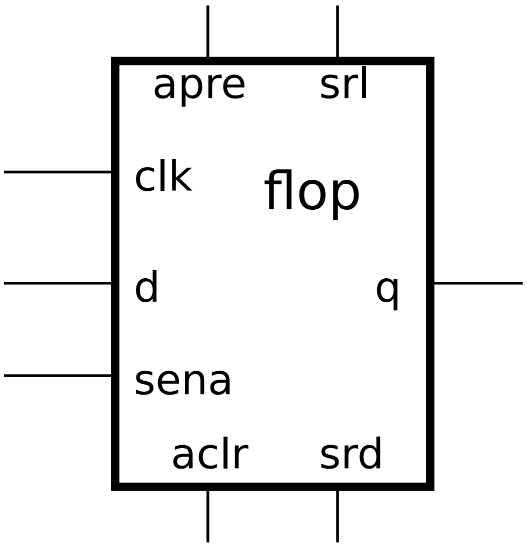
Encounter's Generic Flip-Flop: `flop`.



`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.

Encounter's Generic Flip-Flop: `flop`.



Inference and Mapping

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

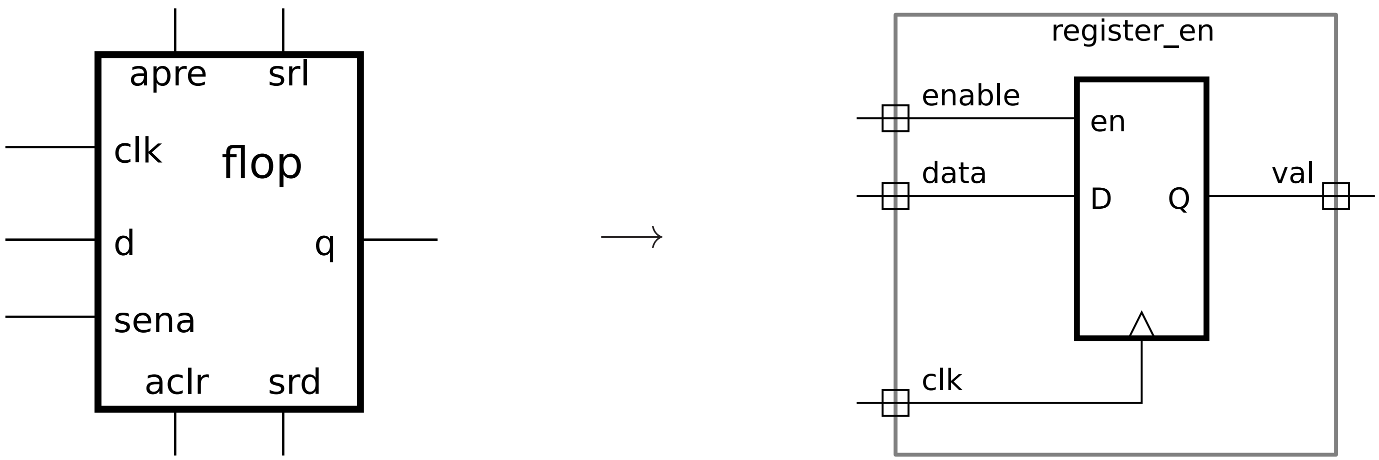
During technology mapping `flop` replaced with registers from technology library.

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 Encounter RTL Compiler.

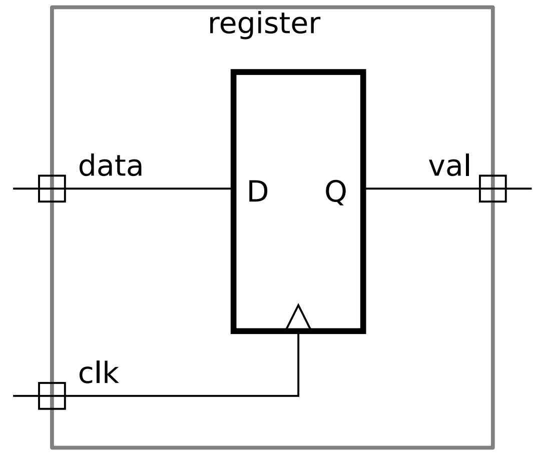
Reference: *HDL Modeling in Encounter RTL Compiler* 14.2 April 2015.

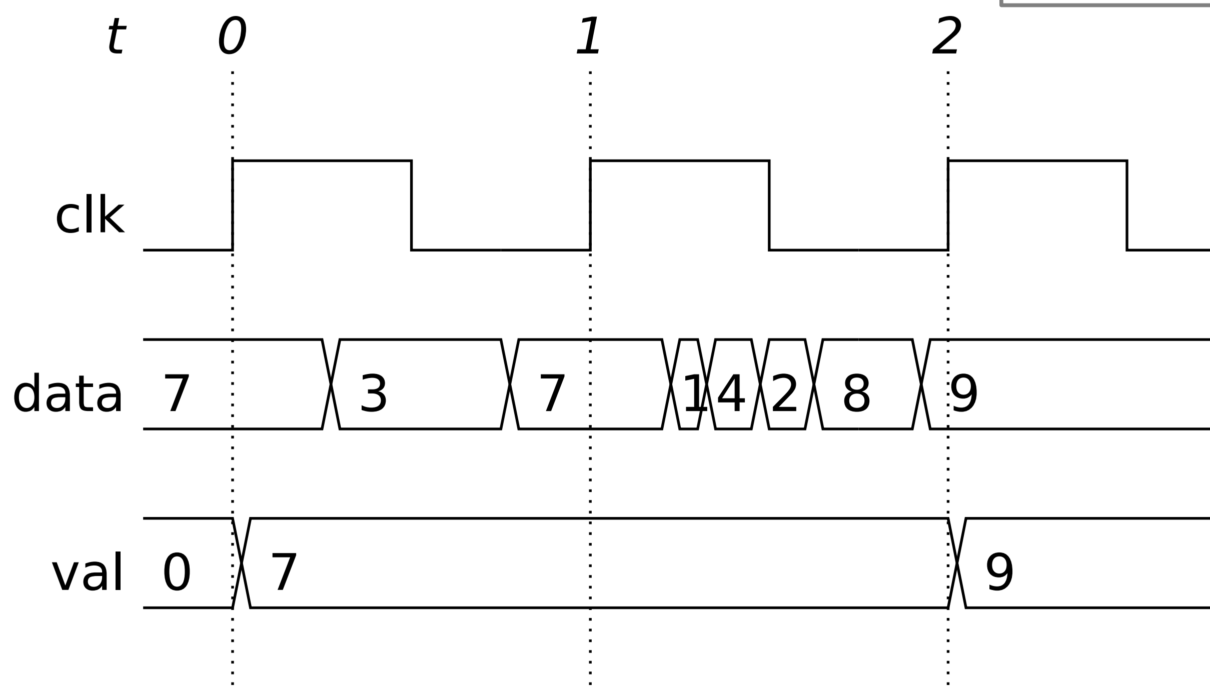
For inference of edge-triggered register `R` clocked by `clk`:

- `R` must be a variable type.
- `R` must be assigned in exactly one always block . . .
  . . . and must be consistently blocking (`R=x;`) or non-blocking (`R<=x;`).
- The always block must start with `always` or `always_ff`.
- The always used must be followed with `@( posedge clk, ...)`.

Simple Register

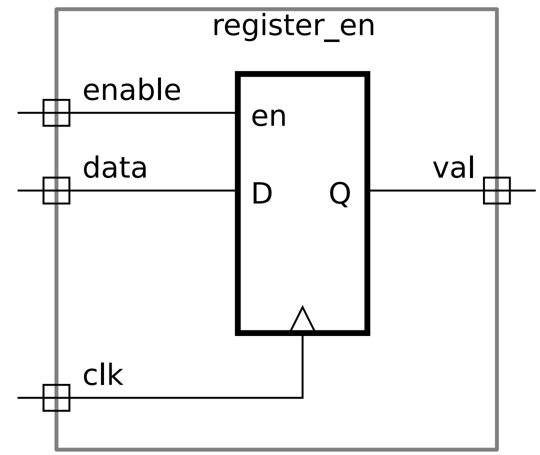
```
module register
  #( int width = 16 )
   ( output logic [width-1:0] val,
     input wire [width-1:0] data,
     input wire clk );

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

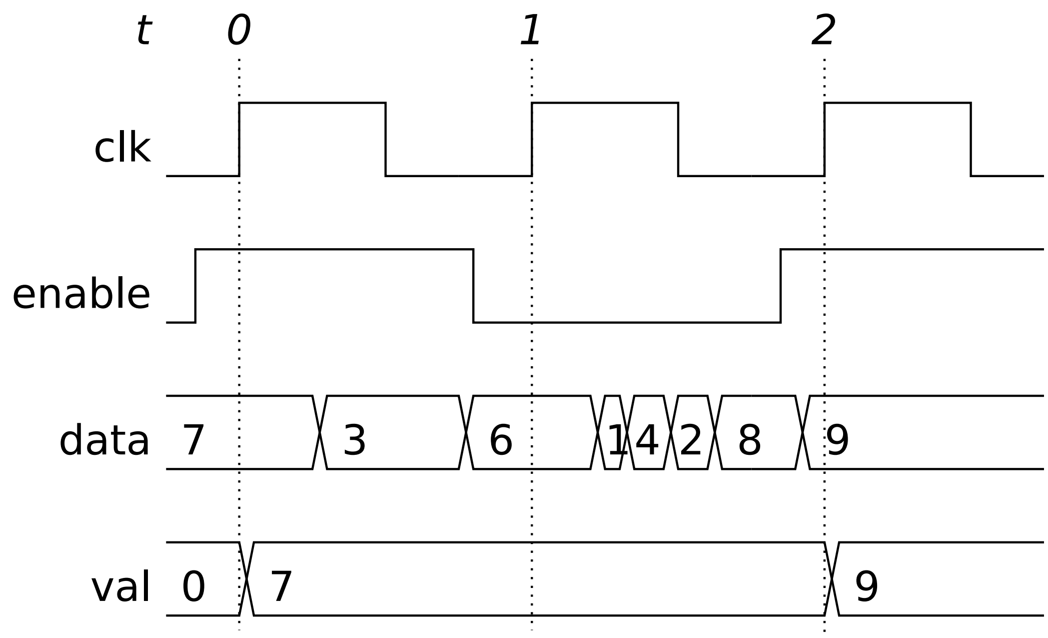




Register with Enable

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

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

endmodule
```

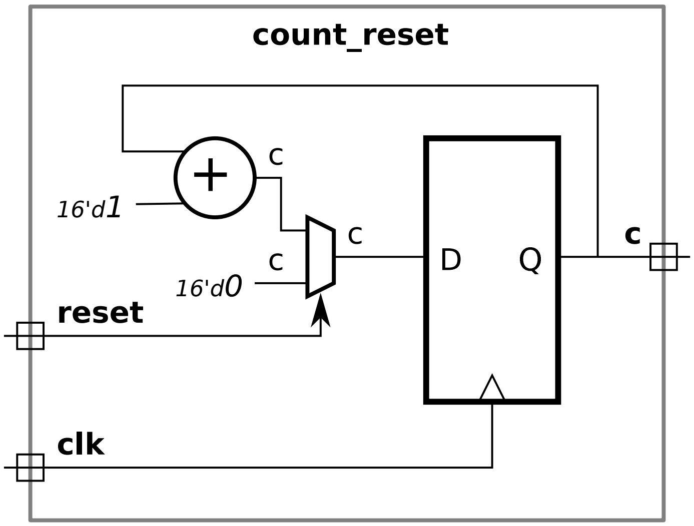




LSU EE 4755 Lecture Transparency. Formatted 12:12, 26 October 2016 from lsli-syn-seq.

Clock with Reset

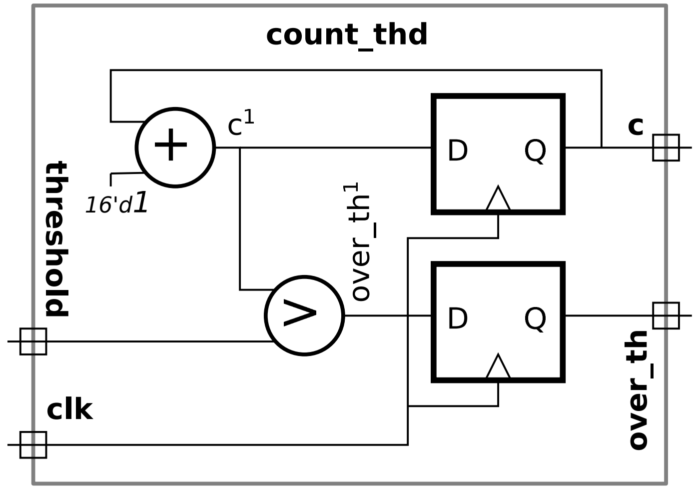
Note multiple C values.

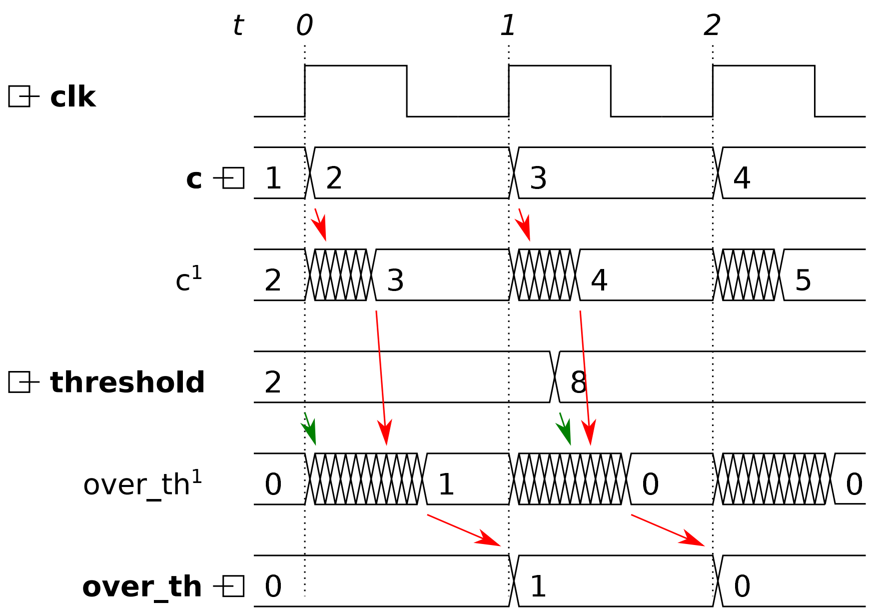



```
module count_reset
  #( int bits = 16 )
   ( output logic [bits-1:0] c,
     input wire reset,
     input wire clk );

   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 wire [bits-1:0] threshold,
     input wire clk );

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

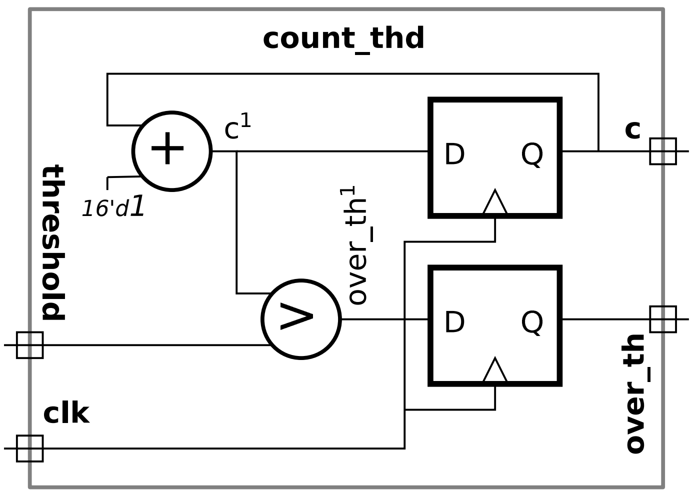


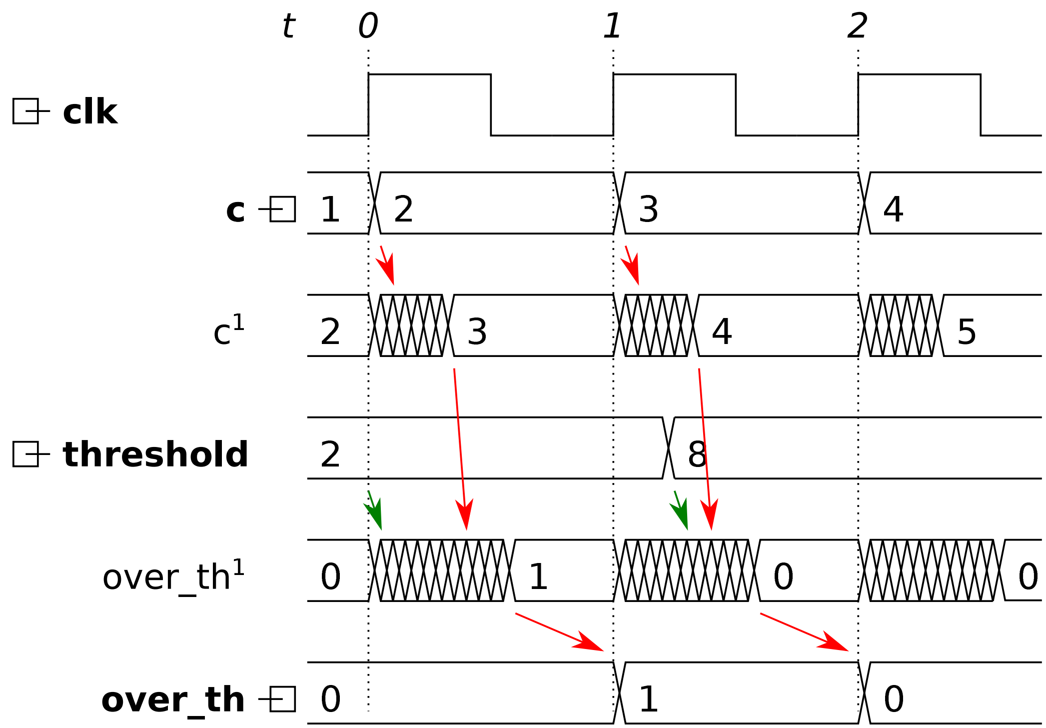


Two Issues:

Critical path through adder/comparison unit.

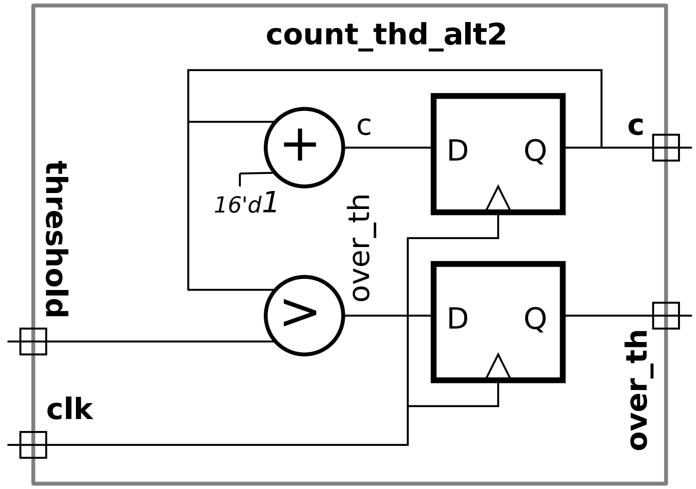
Do we really want a flip-flop for `over_th`?





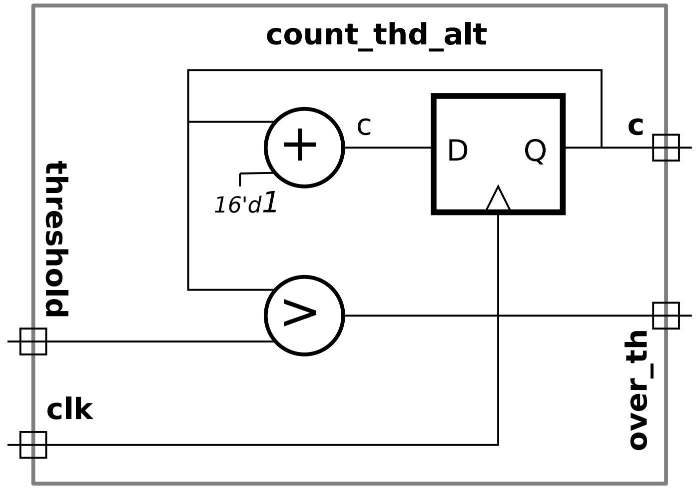
Fix critical path issue.

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

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

   always_comb over_th = c > threshold;

endmodule
```



Example: Sequential Shifter

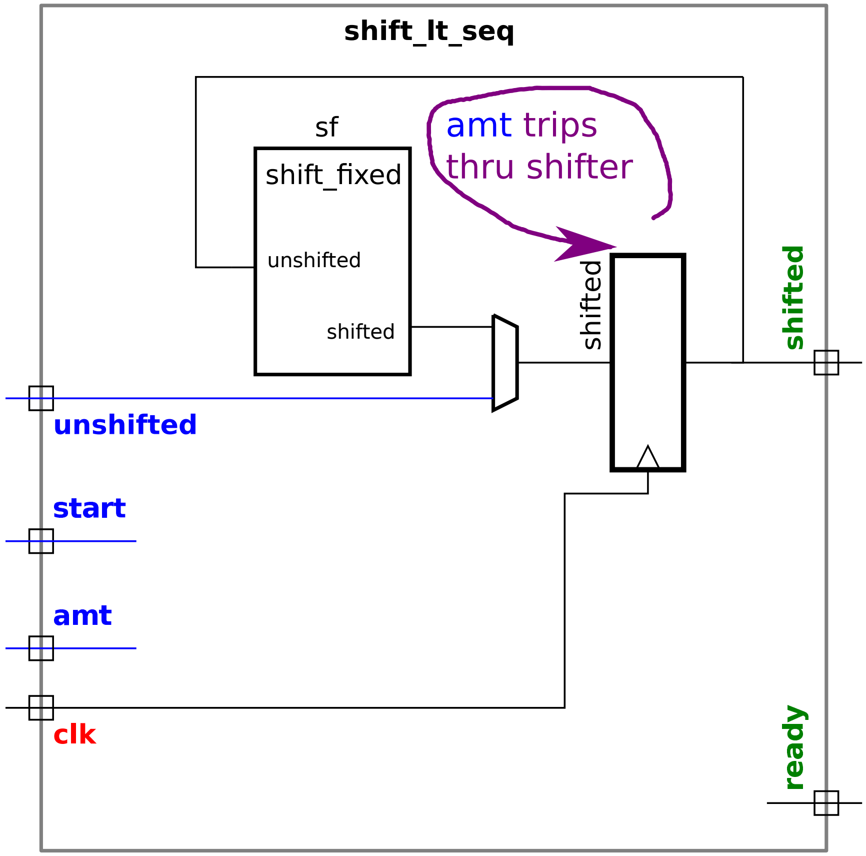
Remember: We can build an n-bit shifter using $\lceil \log_2 n \rceil$ 2^i-bit shifters and 2-input muxen.

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

Why not use $< \lceil \log_2 n \rceil$ shifters and muxen but use them multiple times?

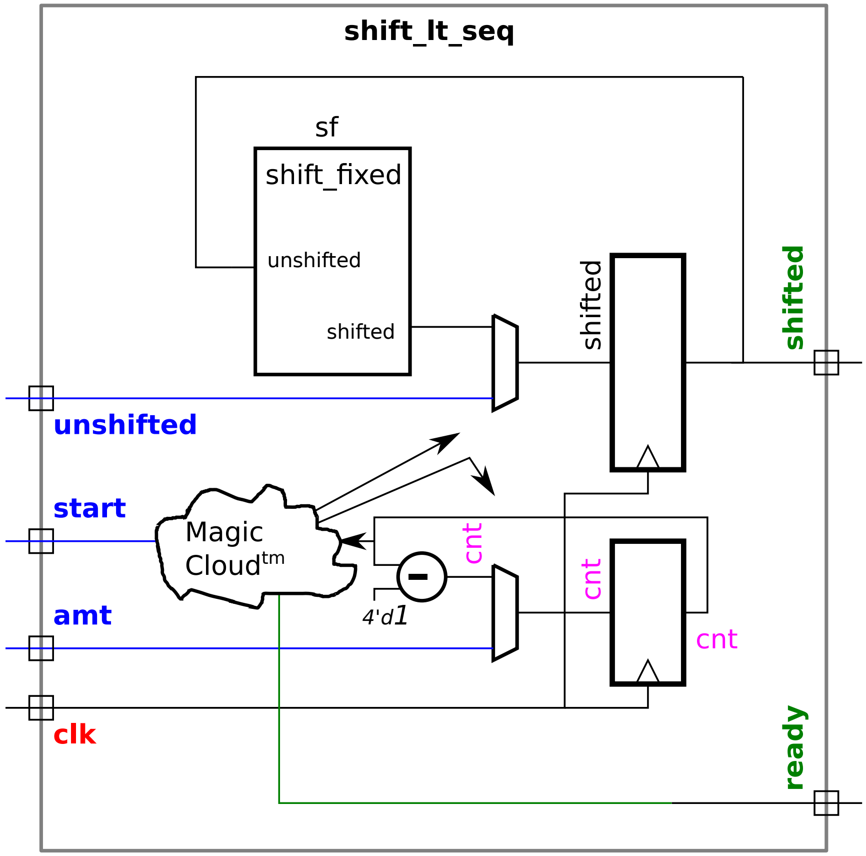
We'll start with one fixed shifter.

Idea sketch for sequential shifter.



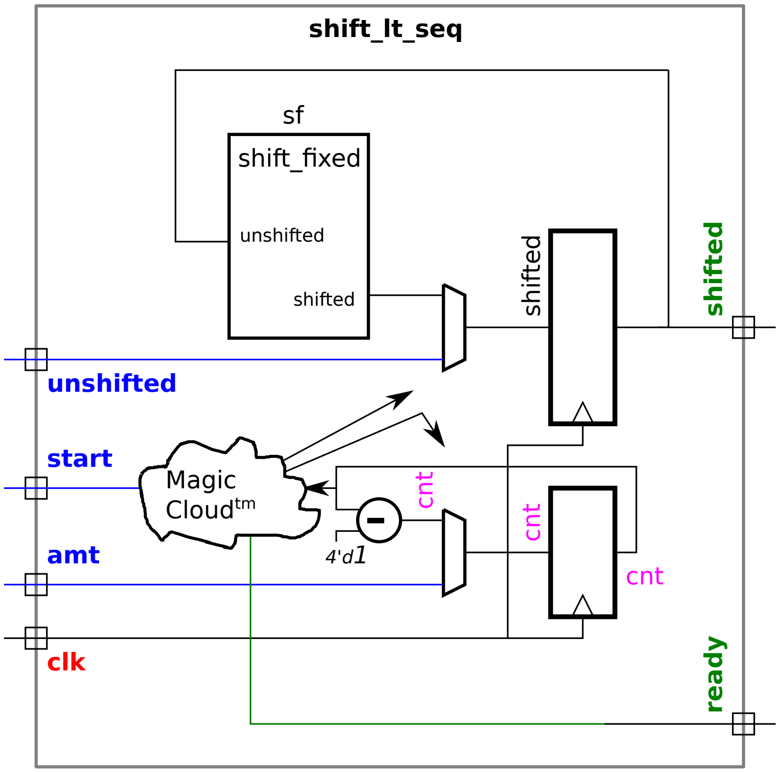
Pass value through shifter `amt` times.

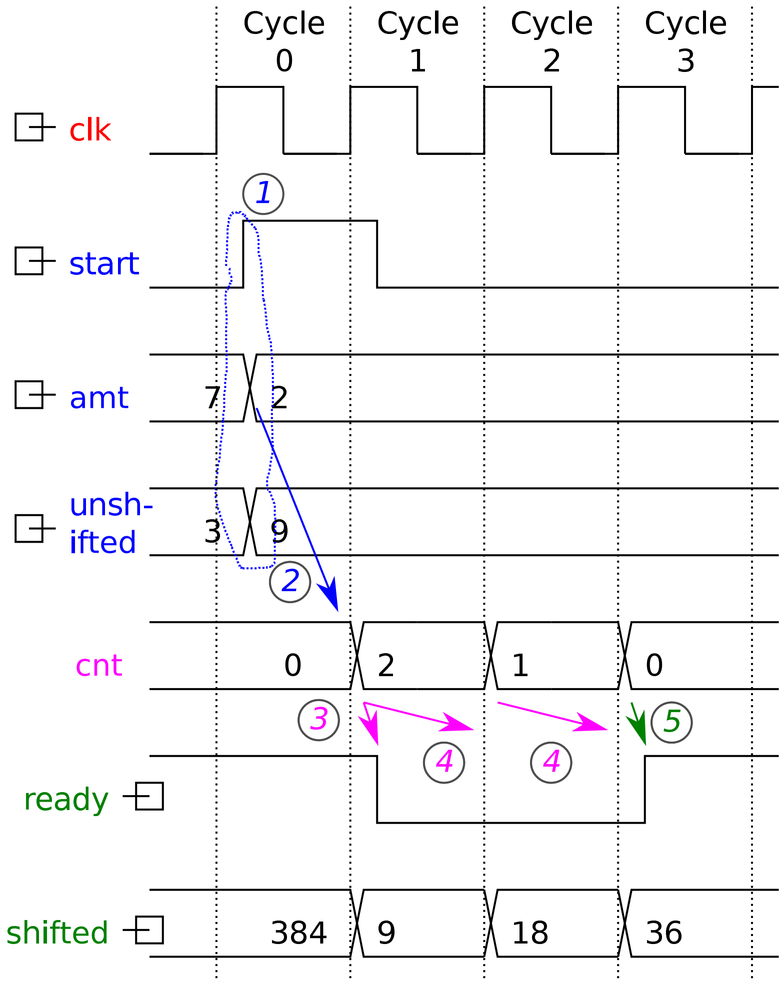
Idea sketch for sequential shifter.



Use register `cnt` to count number of times.

Timing.



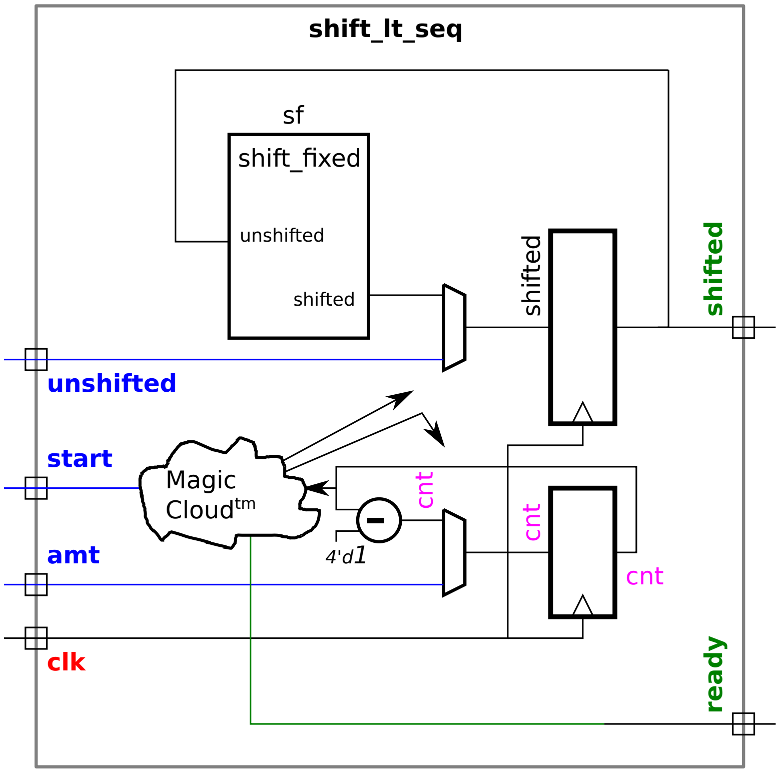


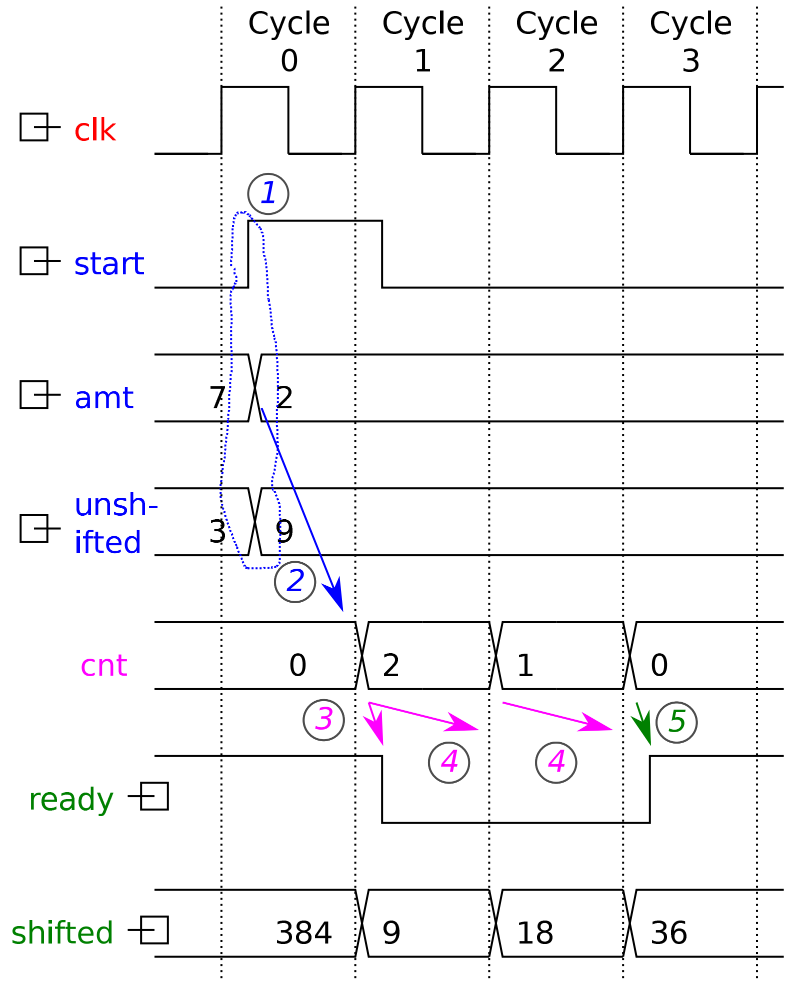
1: External device provides inputs.

Inputs assumed to be available. . .

. . . early in clock cycle.

Timing.

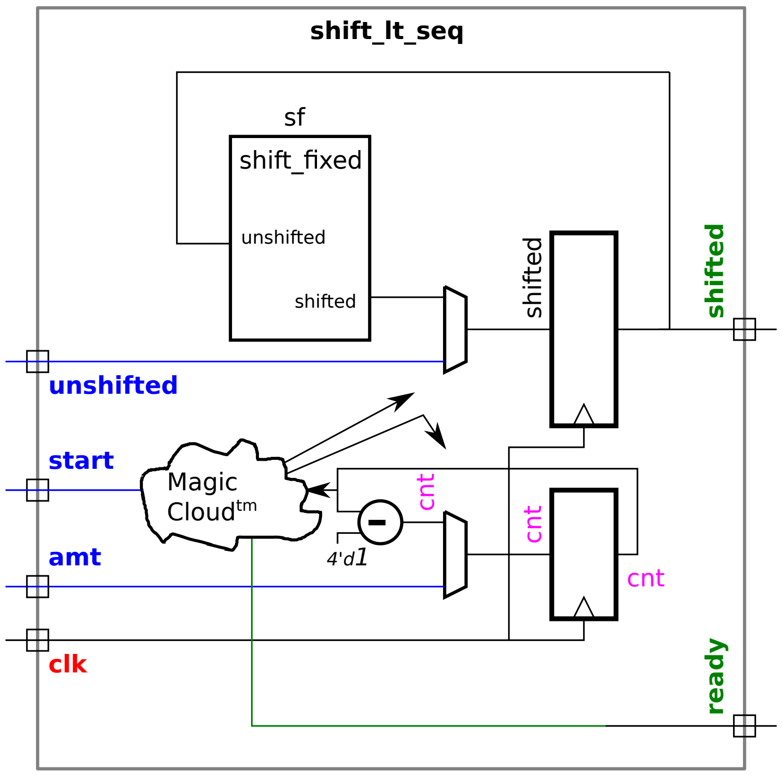


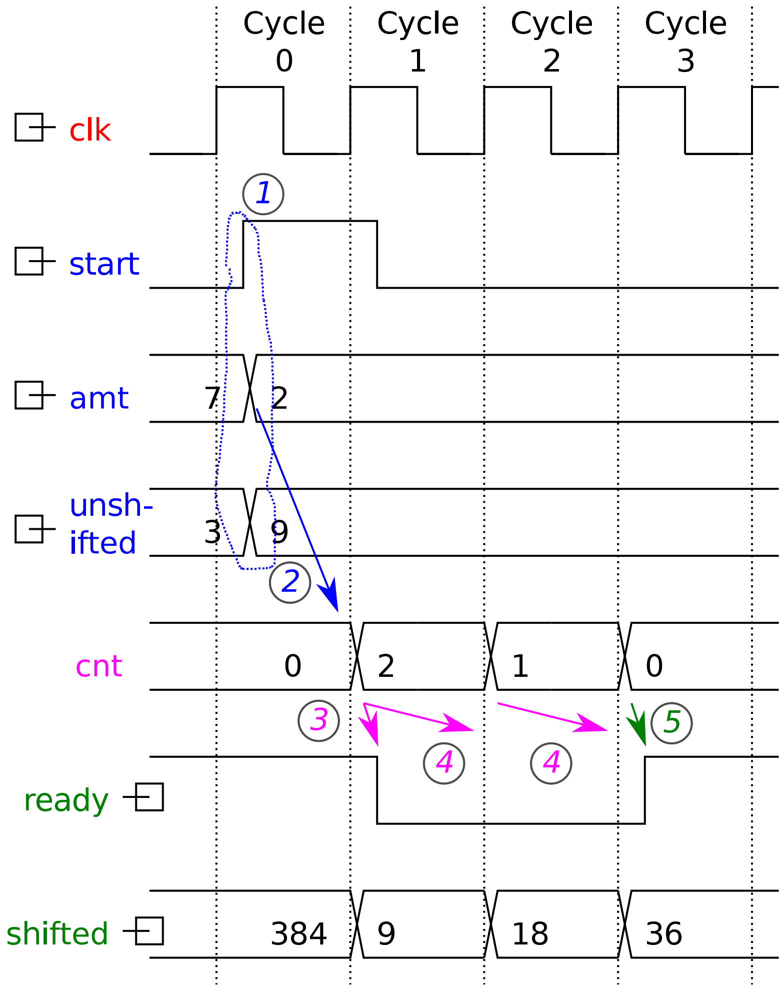


2: At positive edge:

`cnt` initialized to `amt`.

`shifted` initialized to `unshifted`.

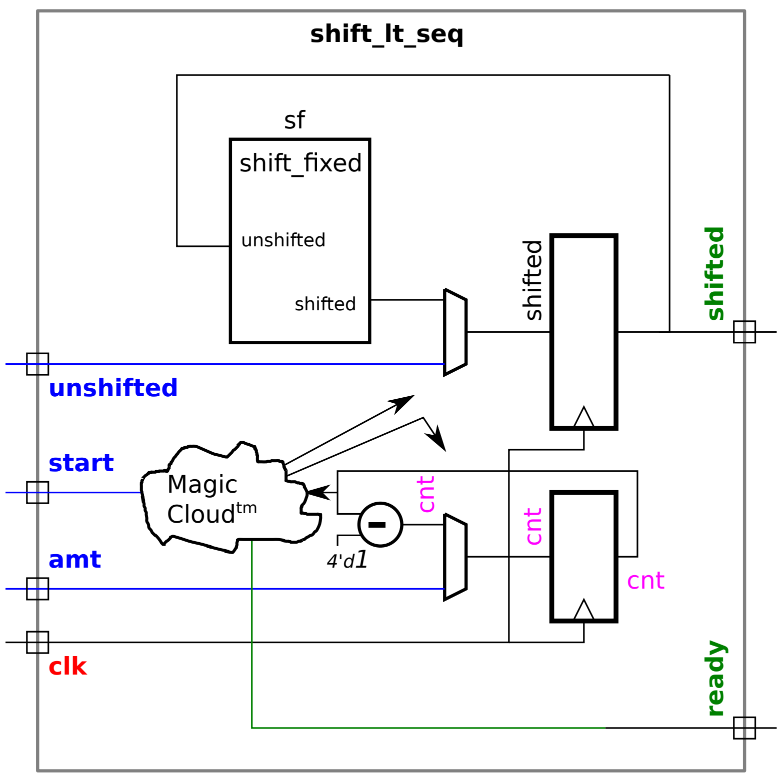


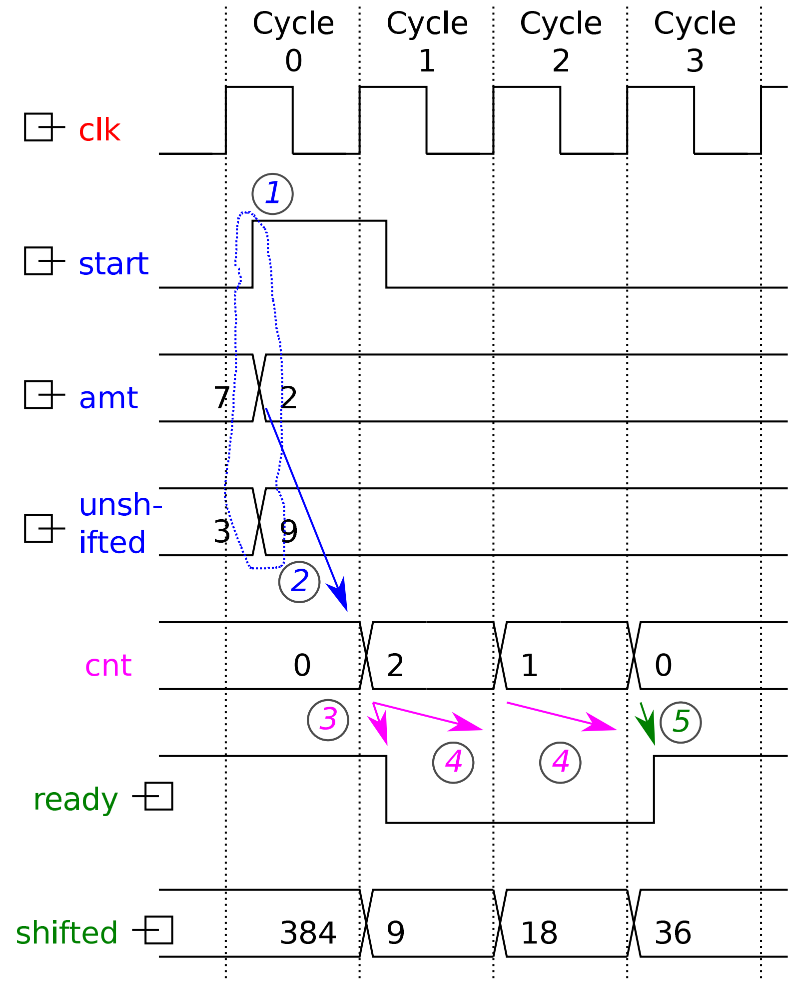


3: Early in Cycle 1:

`ready` goes to zero.

Timing.

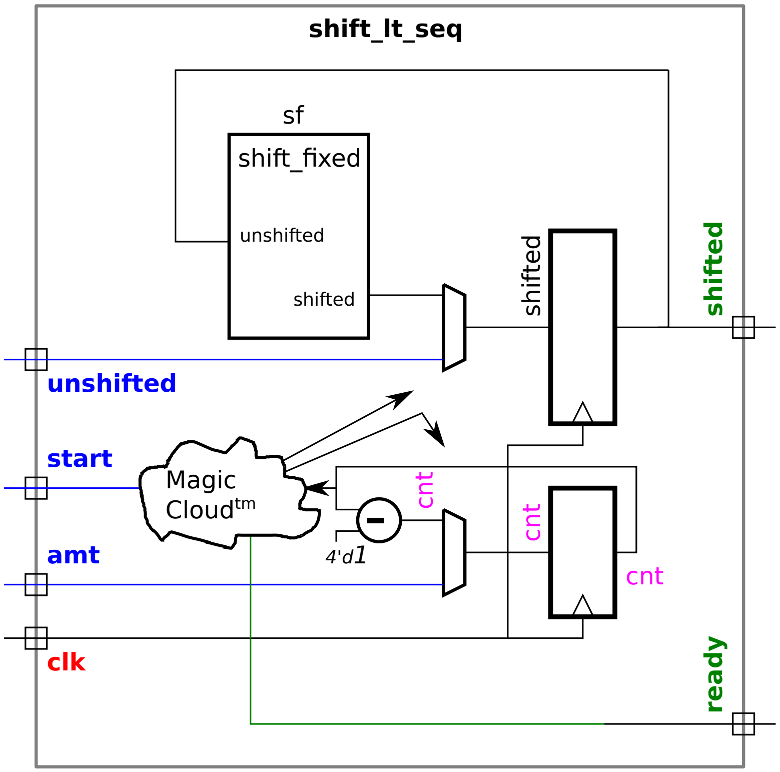


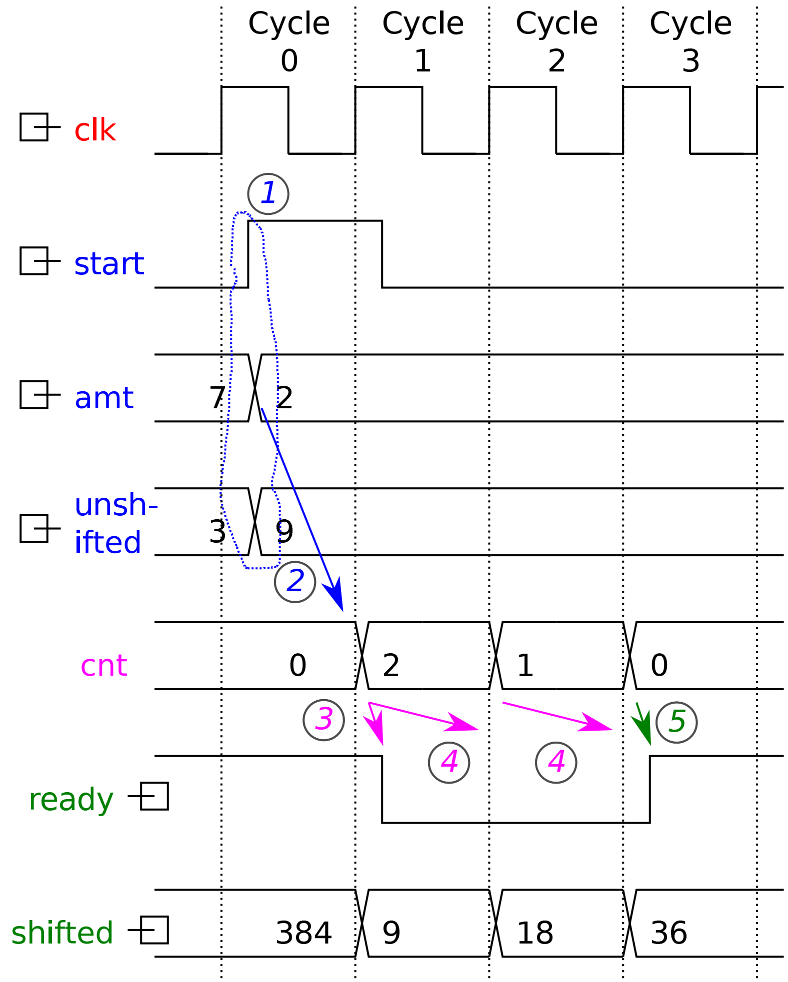


4: During cycles 1 and 2:

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.

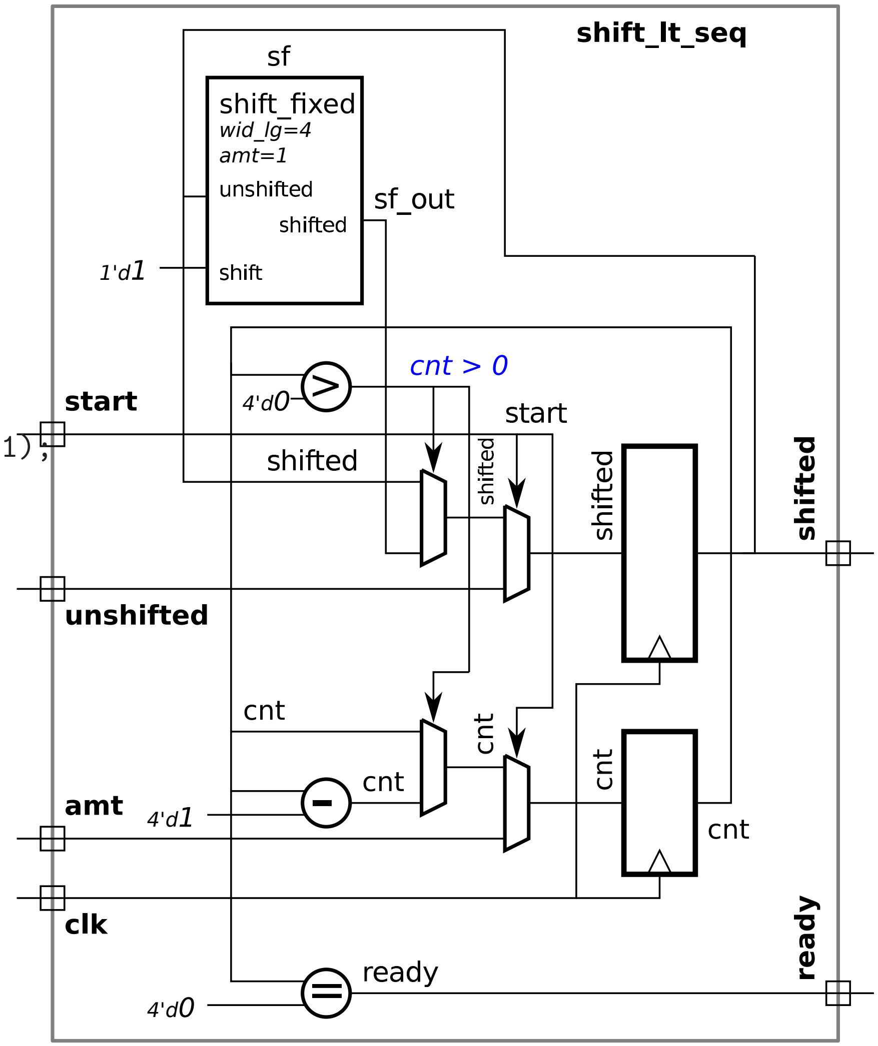
Ready signal available early in clock cycle.

Sequential Shifter Verilog

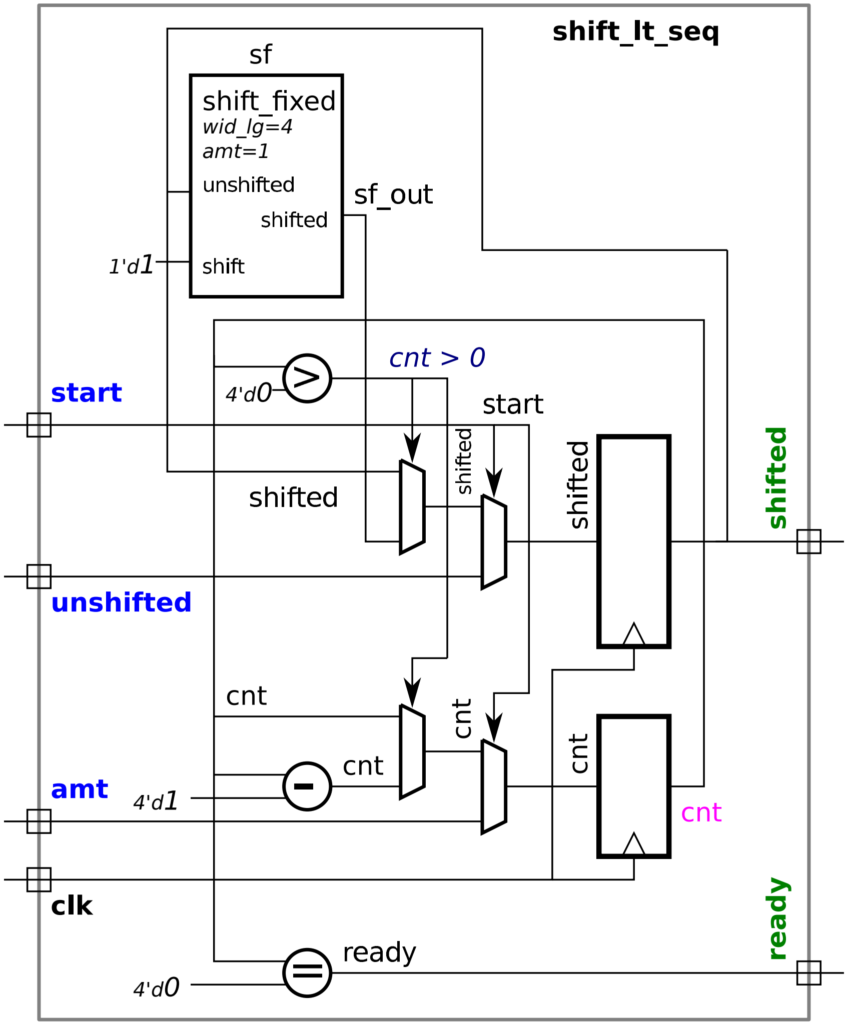
```
module shift_lt_seq #( int wid_lg = 4, int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,       output wire ready,
     input [wid-1:0] unshifted,            input [wid_lg-1:0] amt,
     input start, input clk );

   wire [wid-1:0]       sf_out;
   shift_fixed #(wid_lg,1) sf( sf_out, shifted, 1'b1 );  // Fixed Shifter

   logic [wid_lg-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
```

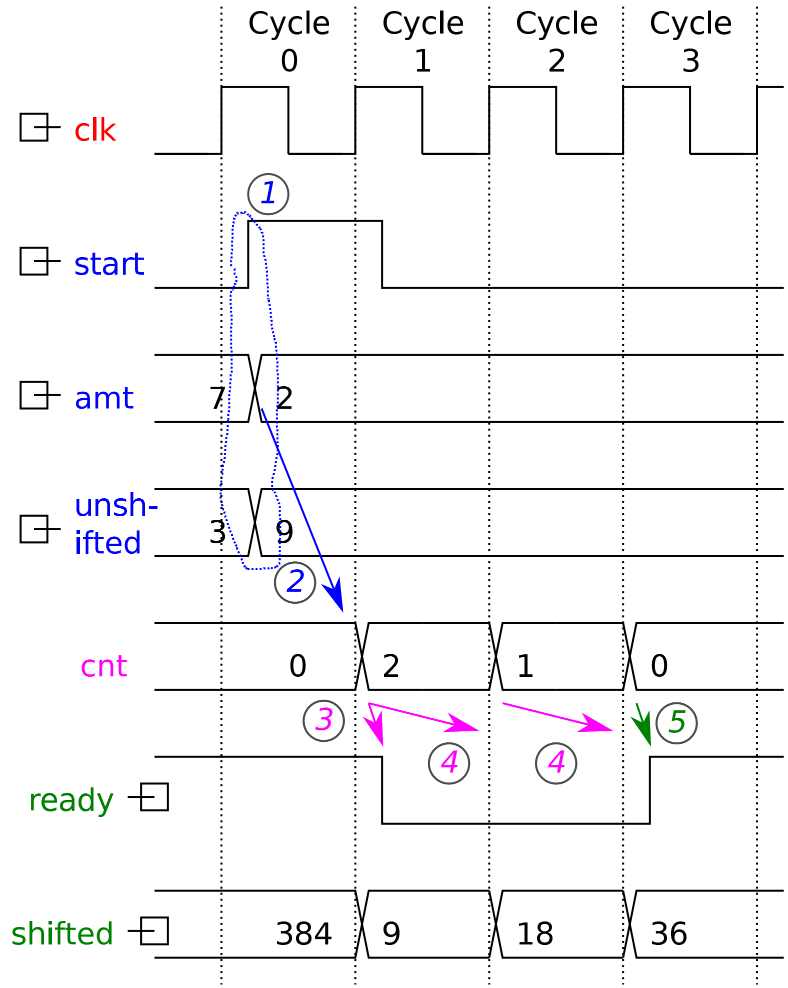
Inferred Hardware, No Optimization

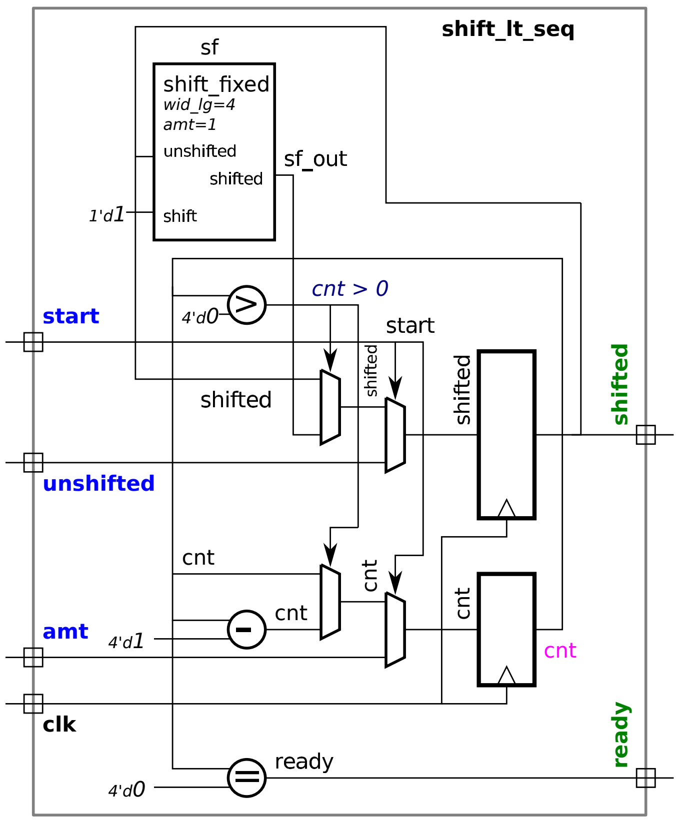
```
module shift_lt_seq
  #( int wid_lg = 4, int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,
     output wire ready,
     input [wid-1:0] unshifted,
     input [wid_lg-1:0] amt,
     input start, input clk );

   wire [wid-1:0]       sf_out;
   shift_fixed #(wid_lg,1) sf(sf_out,shifted,1'b1);

   logic [wid_lg-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
```

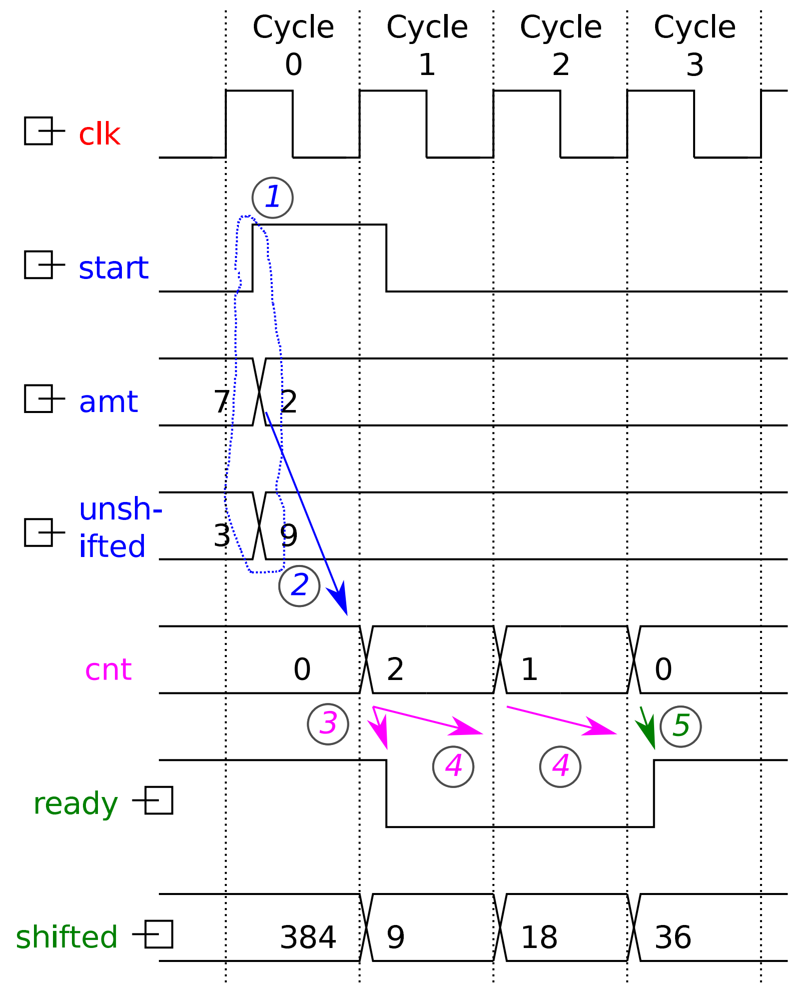


Inferred Hardware, No Optimization









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.

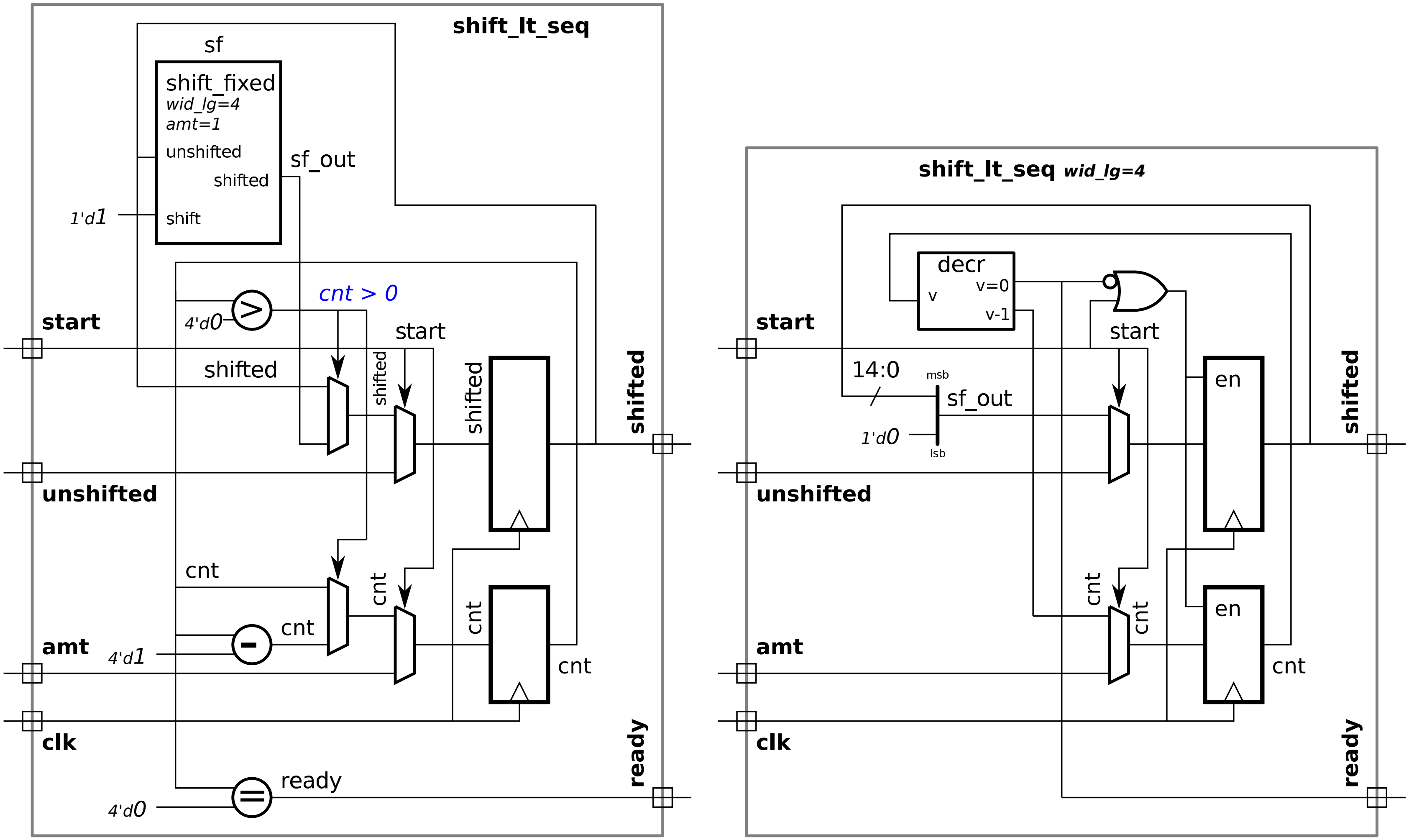
Include optimizations we hope synthesis program will make.

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 $c == 0 \ldots$
$\ldots$ 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
unshifted
cnt
amt
4'd1
clk
ready
shift_lt_seq wid_lg=4
decr
v
v=0
v-1
14:0
msb
lsb
1'd0
en

Sequential Shifter with Multiple Shifters

For example: Shift x by 9 bits.

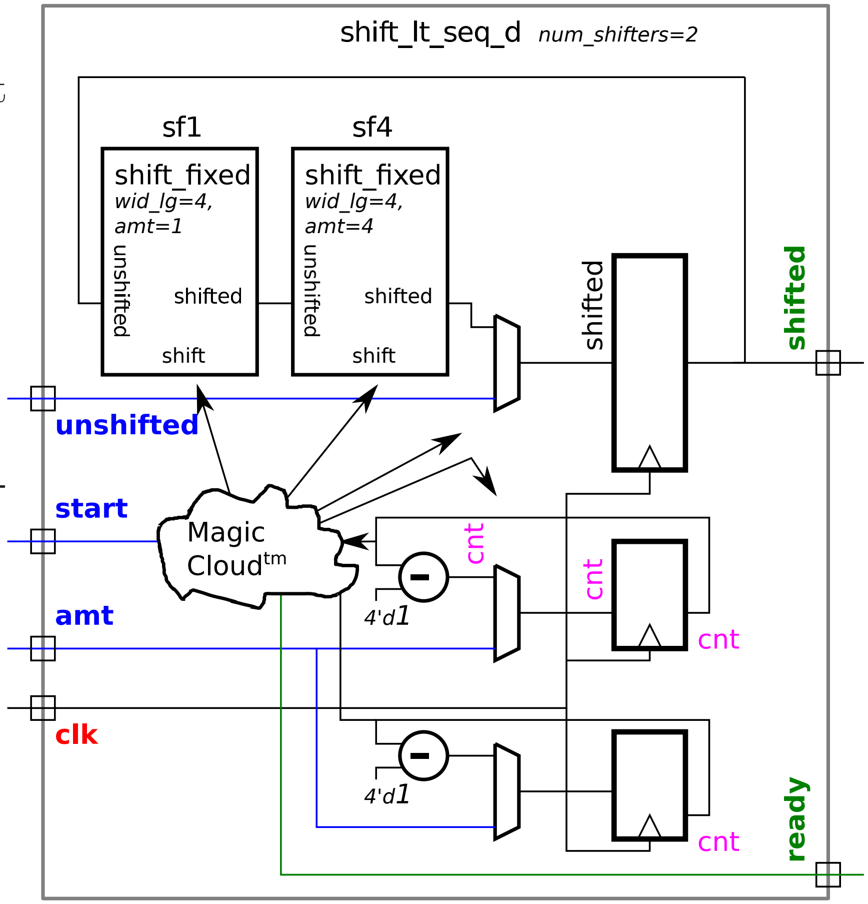
Use a sequential shifter with 4-bit and 1-bit shifters.

Shift by 4-bits twice and by 1-bit once.

Features

The `cnt` register divided into multiple segments.

Fixed shifter may or may not shift.



Performance Analysis and Design Optimization

Goal: Choose the best shifter for some larger design.