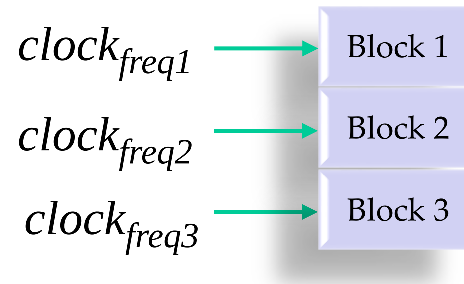
VARIABLE FREQUENCY CLOCKING HARDWARE

Variable-Frequency Clocking Hardware

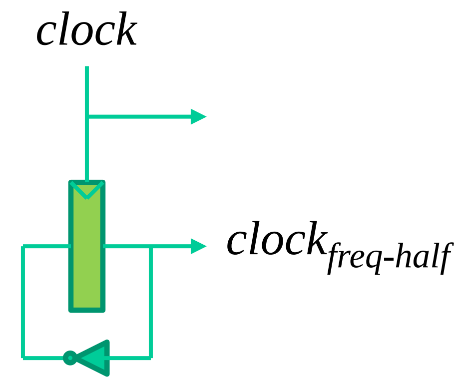
- Many complex digital systems have components clocked at different frequencies



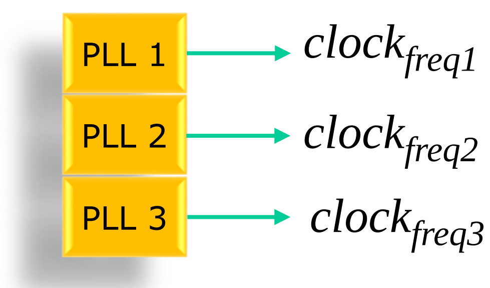
- Reason 1: to reduce power dissipation
  - The dominant "active" component of power is proportional to the clock frequency
  - If a module's clock frequency can be reduced while maintaining acceptable performance, a reduced frequency will reduce the *active* power dissipation
- Reason 2: Because a sub-module requires a specific clock frequency that is different than the main system's frequency.
  - For example, the DDR4-3200 synchronous DRAM memory interface has an I/O bus that operates at 1.60 GHz and so the module certainly requires a 1.60 GHz (actually probably 0.8 GHz) clock

Multi-rate Clocking Hardware

1) Build slower divided clocks with FFs
   - Some FFs are clocked by the real *clock* signal, others are clocked by a delayed slower $clock_{freq\text{-}half}$ signal coming from a frequency divider. Significant clock skew → potential for dead chip ☹
   - Could risk your job security (moderate exaggeration)

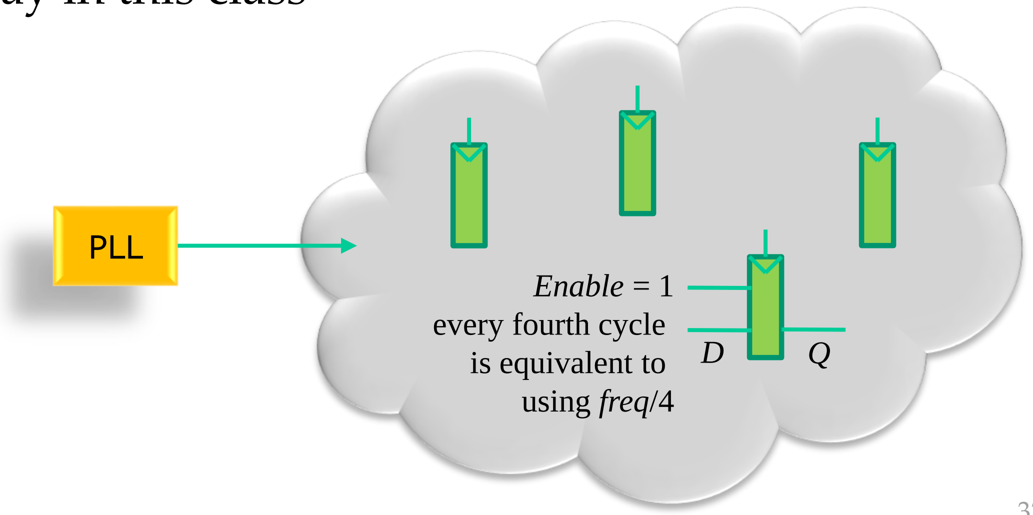


2) Use multi-frequency clocks
   + May save significant power in large active circuits
   - Requires a complete and independent clock tree for each frequency and possibly an independent phased-locked loop (PLL) for each
   - Each PLL uses significant power



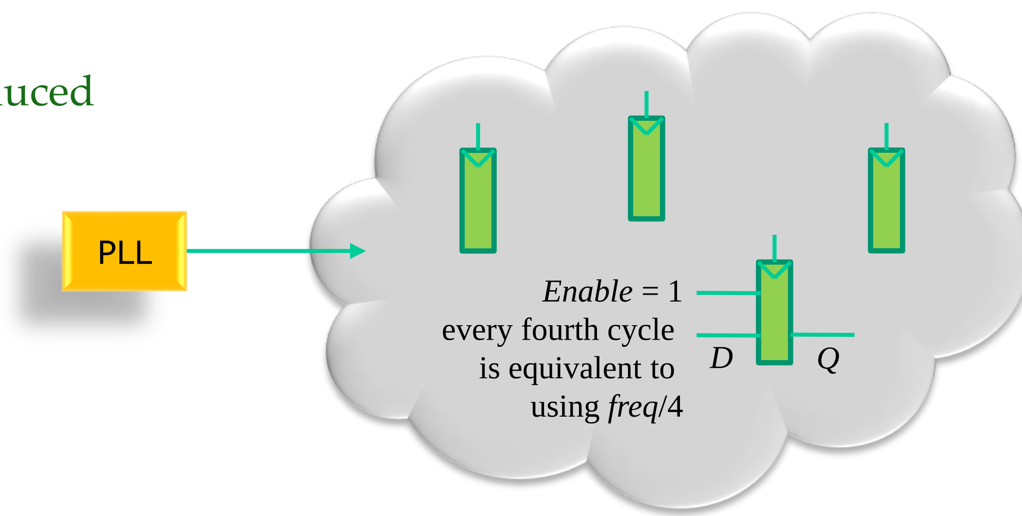
Multi-rate Clocking Hardware

3) Pseudo-multi-rate: Clock all logic with the highest-rate clock
   - Utilize simple counters that load registers or route signals on only certain clock edges (for example, every fourth clock edge for *freq*/4).
   - \+ Definitely the simplest and most robust
   - – Counters must be reset simultaneously and the *reset* signal must meet timing requirements at the highest frequency
   - – Design in only this way in this class



Multi-rate Clocking Hardware

3) Pseudo-multi-rate: Clock all logic with highest-rate clock
   - Possible issue if there are a very large number of FFs requiring the same enable signal
     1. delay of the fanout tree reduces available cycle time
     2. the enable signal could be modestly pipelined
   - Effectiveness
     + Logical operation—same as if frequency was reduced
     + Power reduction of logic—same as if frequency was reduced
     – Power reduction of clock signal—none at all



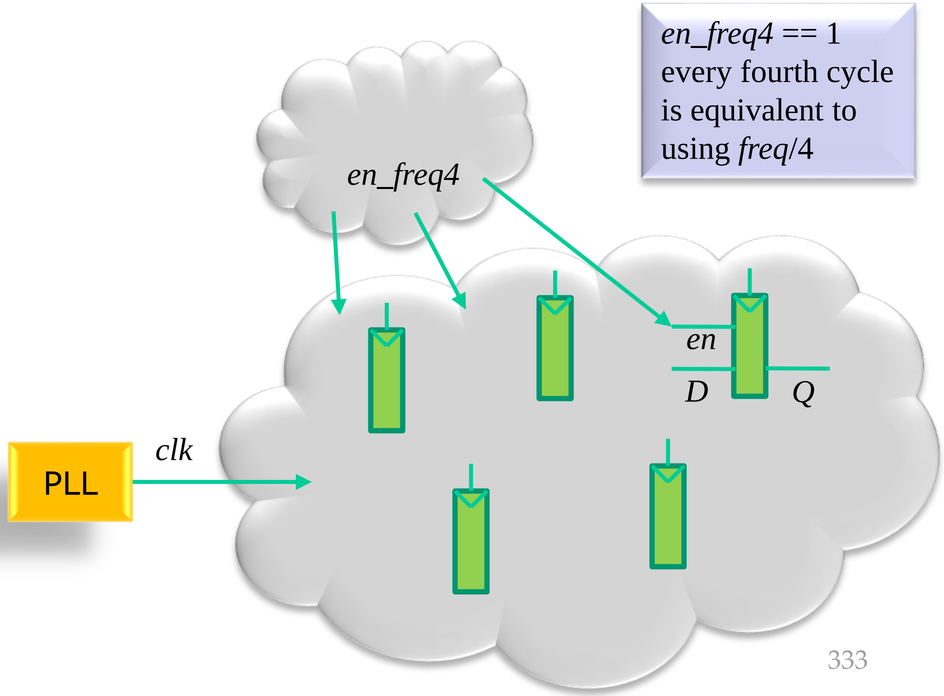
Multi-rate Clocking Hardware

- Example 1a to imitate a clock frequency of *freq*/4

```
reg [1:0] count, count_c;        // two bits counts 00, 01, 10, 11, 00, ...
reg       Q;                     // assume D comes from elsewhere
always @(*) begin
   count_c = count + 2'b01;      // let the counter wrap 2'b11 → 2'b00
end

// en_freq4 will be high every 4th cycle
reg en_freq4;
always @(*) begin
   if (count == 2'b00) begin
      en_freq4 = 1'b1;
   end
   else begin
      en_freq4 = 1'b0;
   end
end

// breaking a guideline with "if" here
always @(posedge clk) begin
   count <= #1 count_c;
   if (en_freq4 == 1'b1) begin
      Q     <= #1 D;
      state <= #1 state_c;
   end
end
```



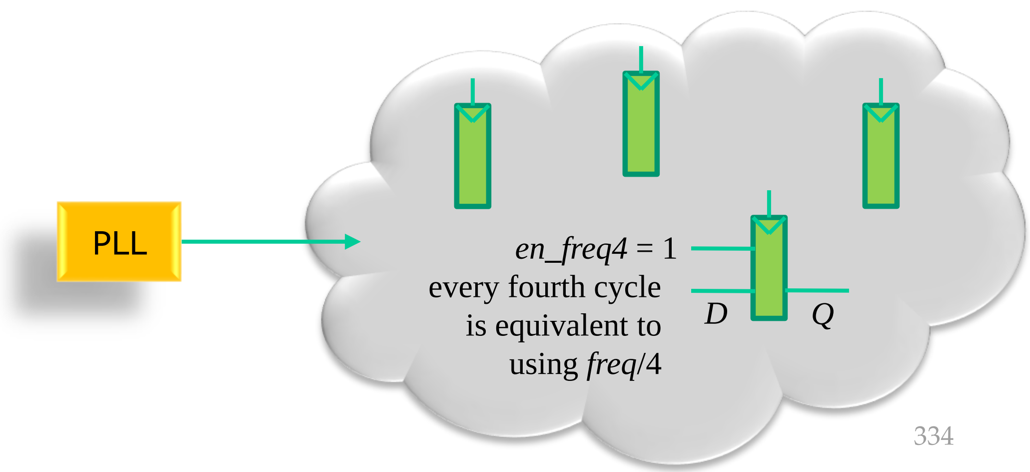
Multi-rate Clocking Hardware

- Example 1b to imitate a clock frequency of *freq*/4

```
reg [1:0] count, count_c;              // two bits counts 00, 01, 10, 11, 00, ...
reg       Q;                           // assume D comes from elsewhere
always @(*) begin
   count_c = count + 2'b01;            // let the counter wrap 2'b11 → 2'b00
end

wire en_freq4;                         // use a wire in this example
assign en_freq4 = (count == 2'b00);    // code is compact but slightly less clear

// breaking a guideline with "if" here
always @(posedge clk) begin
   count <= #1 count_c;
   if (en_freq4 == 1'b1) begin
      Q <= #1 D;
   end
end
```



Multi-rate Clocking Hardware

- Example 2 to imitate a clock toggling at 1 Hz, a with 500 MHz clock

```
reg [28:0] count, count_c;          // 29 bits counts up to 536 million
reg        en_increment;            // I use a reg in this example
reg        Q;                       // assume D comes from elsewhere
always @(*) begin
   // defaults
   count_c = count + 29'h0000_0001;   // "count" is a flip-flop register
   en_increment = 1'b0;               // a combinational logic signal

   if (count == 29'd499_999_999) begin
      count_c = 29'h0000_0000;        // wrap counter back to zero
      en_increment = 1'b1;            // pulse FF enable signal high
   end
end

always @(posedge clk) begin
   count <= #1 count_c;
   if (en_increment == 1'b1) begin
      Q <= #1 D;
   end
end
```