System Integration Issues

- Communicating FSMs
- Clocking, theory and practice

Encoding numbers
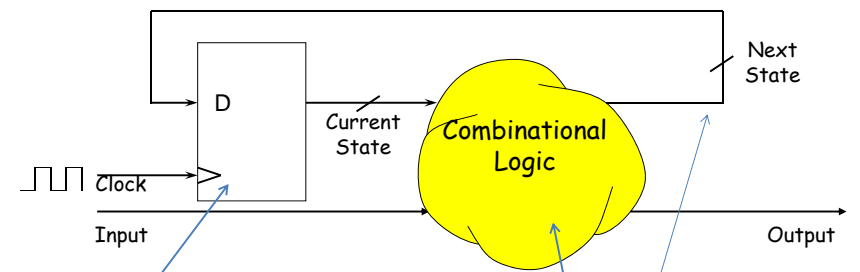

Memory Controller

State 0 State 1 State 0
State 2 State 4
State 3
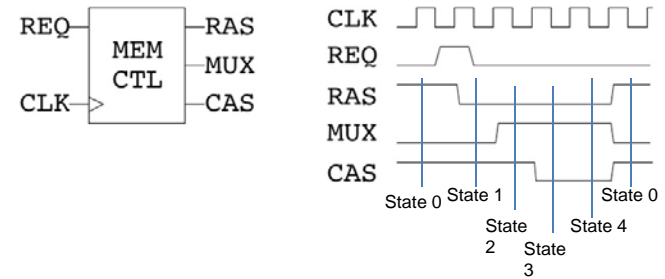
1GB RAM

FSM


```
always @(posedge clock)
   state <= next_state;
```

```
always @ *
  begin // logic to determine next_state
    case (state)
    state_1:  next_state = . . .
    state_2:  next_state = . . .
     .  .  .
        default: next_state = STATE_0;
    endcase
  end
```

Glitchy Solution

```
module (
  input req, clk,
  output reg ras, mux, cas
  );

reg [3:0] state, next_state:

parameter [3:0] STATE_0 = 0;  // 0000
parameter [3:0] STATE_1 = 1;  // 0001
parameter [3:0] STATE_2 = 2;  // 0010
parameter [3:0] STATE_3 = 3;  // 0011
parameter [3:0] STATE_4 = 4;  // 0100

always @(posedge clk)  state <= next_state;

always @ * begin
  case (state)
    STATE_0: next_state = req ? STATE_1 : STATE_0;
    STATE_1: next_state = STATE_2;
    STATE_2: next_state = STATE_3;
    STATE_3: next_state = STATE_4;
    STATE_4: next_state = STATE_0;
    default: next_state = state_0;
  endcase
end

assign ras = !((state==STATE_1)||(state==STATE_2)||(state==STATE_3)||(state==STATE_4));
assign mux = (state==STATE_2)||(state==STATE_3)||(state==STATE_4);
assign cas = !((state==STATE_3)||(state==STATE_4));

endmodule
```
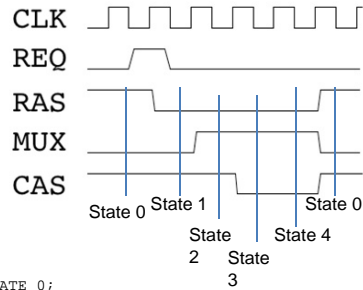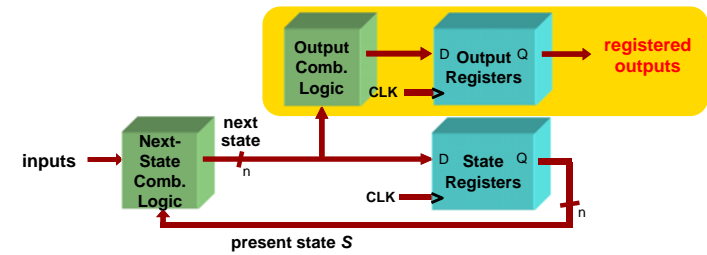
CLK

REQ

RAS

MUX

CAS

State 0   State 1   State 0

State 2   State 4

State 3

Registered FSM Outputs are Glitch-Free

- Move output generation into the sequential always block
- Calculate outputs based on **next** state
- Delays outputs by one clock cycle. Problematic in some application.
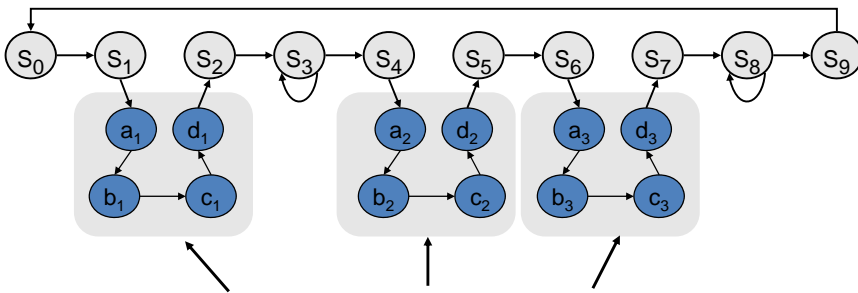
```
reg DC,DN,DD;

// Sequential always block for state assignment
always @ (posedge clk or negedge reset) begin
  if (!reset)   state <= IDLE;
  else if (clk) state <= next;

  DC <= (next == GOT_30c || next == GOT_35c ||
         next == GOT_40c || next == GOT_45c ||
         next == GOT_50c);
  DN <= (next == RETURN_5c);
  DD <= (next == RETURN_20c || next == RETURN_15c ||
         next == RETURN_10c);
end
```


Toward FSM Modularity

- Consider the following abstract FSM:

- Suppose that each set of states $a_x...d_x$ is a "sub-FSM" that produces exactly the same outputs.
- Can we simplify the FSM by removing equivalent states?
  No! The outputs may be the same, but the next-state transitions are not.
- This situation closely resembles a procedure call or function call in software...how can we apply this concept to FSMs?

Acknowledgements: Rex Min

The Major/Minor FSM Abstraction
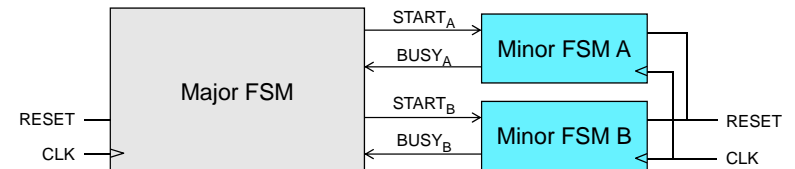
- Subtasks are encapsulated in minor FSMs with common reset and clock
- Simple communication abstraction:
  - START: tells the minor FSM to begin operation (the call)
  - BUSY: tells the major FSM whether the minor is done (the return)
- The major/minor abstraction is great for...
  - Modular designs (*always* a good thing)
  - Tasks that occur often but in different contexts
  - Tasks that require a variable/unknown period of time
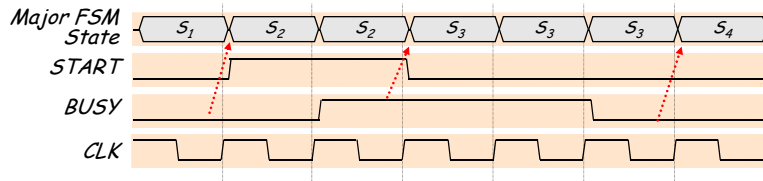 - Event-driven systems

Inside the Major FSM
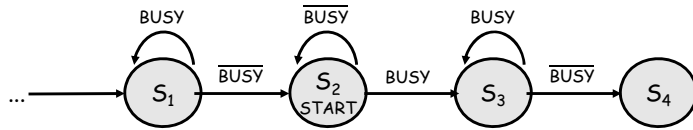
1. Wait until the minor FSM is ready
2. Trigger the minor FSM (and make sure it's started)
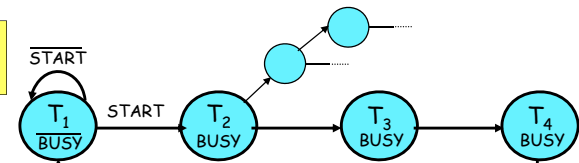3. Wait until the minor FSM is done

Variations:
- Usually don't need both Step 1 and Step 3
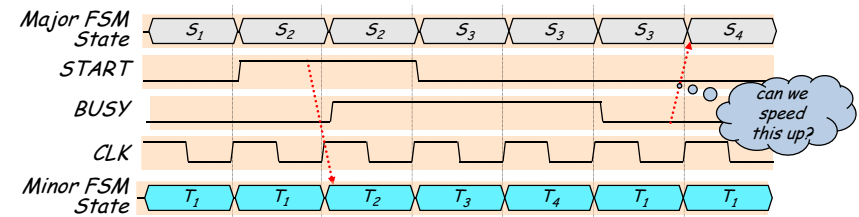- One cycle "done" signal instead of multi-cycle "busy"

Inside the Minor FSM
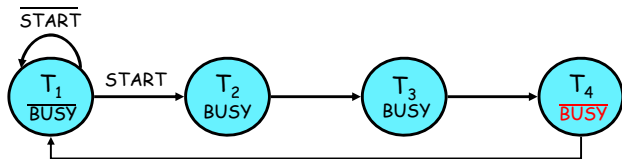

1. Wait for a trigger from the major FSM
3. Signal to the major FSM that work is done
2. Do some useful work

can we speed this up?

Optimizing the Minor FSM

Good idea: de-assert BUSY one cycle early

Bad idea #1:

T_4 may not immediately return to T_1
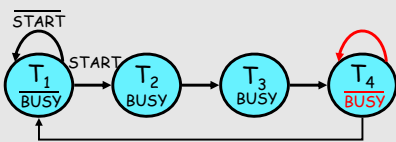
Bad idea #2:

BUSY never asserts!

A Four-FSM Example

Operating Scenario:
- Major FSM is triggered by TICK
- Minors A and B are started simultaneously
- Minor C is started once both A and B complete
- TICKs arriving before the completion of C are ignored

Assume that $BUSY_A$ and $BUSY_B$ both rise before either minor FSM completes. Otherwise, we loop forever!
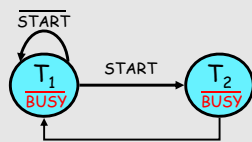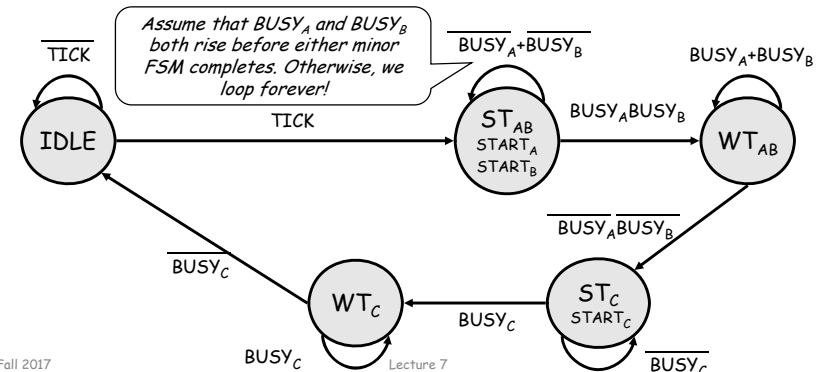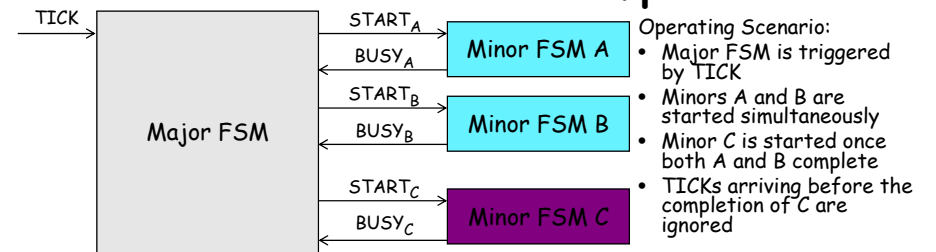
Four-FSM Sample Waveform
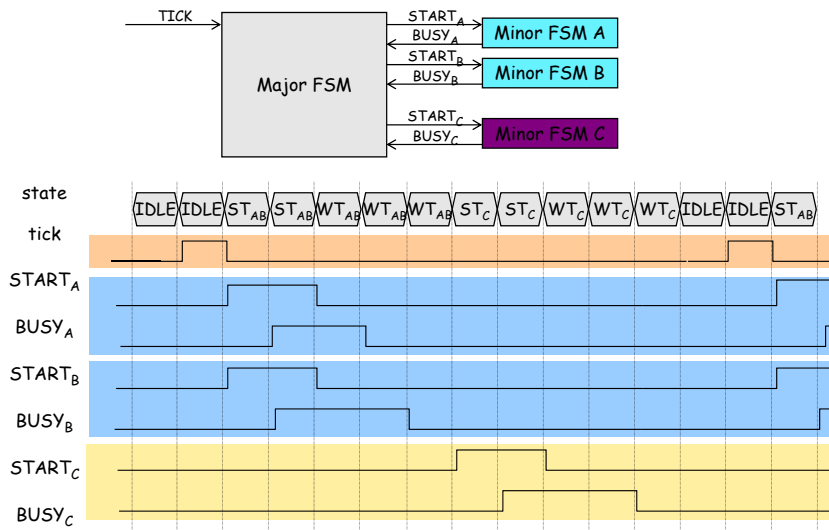
Clocking and Synchronous Communication

Ideal world:

M1 and M2 clock edges aligned in time

Clock Skew

1. Wire delay
2. Different clocks!

Oops! Skew has caused a hold time problem!
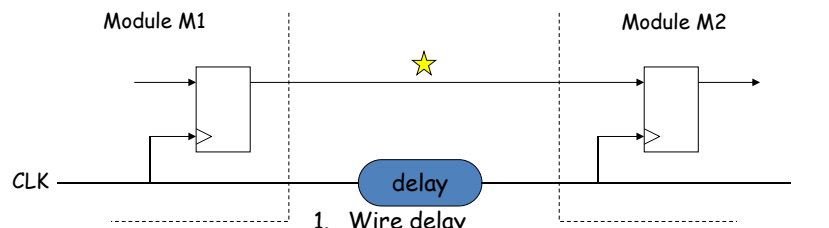
Real world has clock skew:
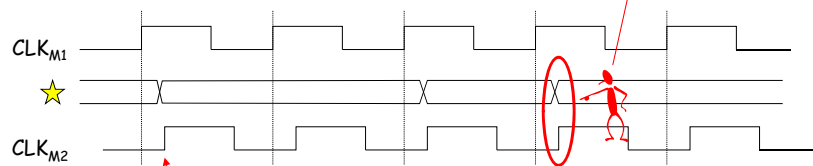
M2 clock delayed with respect to M1 clock

Low-skew Clocking in FPGAs

Global Clock Distribution Network

Figures from Xilinx App Notes

Goal: use as few clock domains as possible

Suppose we wanted clocks at f/2, f/4, f/8, etc.:

```
reg clk2,clk4,clk8,clk16;
always @(posedge clk) clk2 <= ~clk2;
always @(posedge clk2) clk4 <= ~clk4;
always @(posedge clk4) clk8 <= ~clk16;
always @(posedge clk8) clk16 <= ~clk16;
```

No! don't do it this way

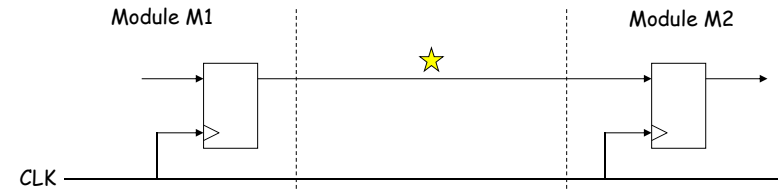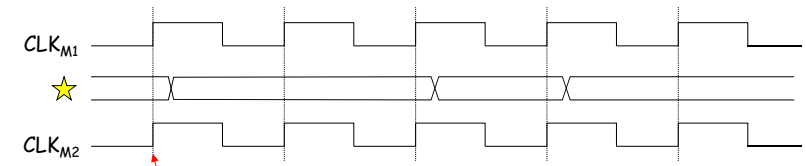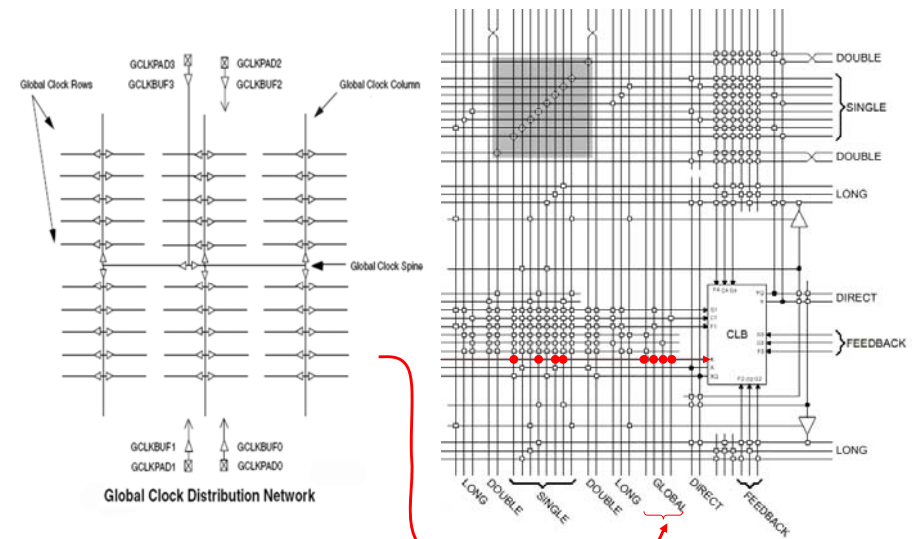No vsync!
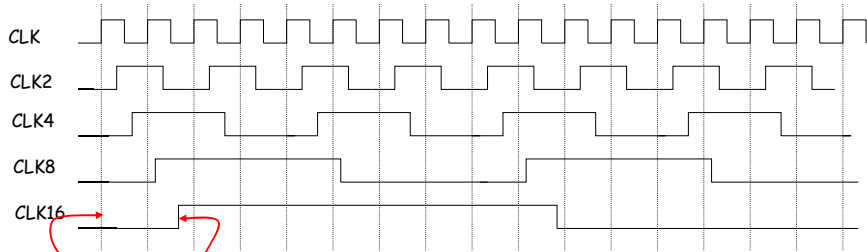
Very hard to have synchronous communication between clk and clk16 domains

Solution: 1 clock, many enables

Use one (high speed) clock, but create enable signals to select a subset of the edges to use for a particular piece of sequential logic

```
reg [3:0] count;
always @(posedge clk) count <= count + 1;    // counts 0..15
wire enb2 = (count[0] == 1'b1);
wire enb4 = (count[1:0] == 2'b11);
wire enb8 = (count[2:0] == 3'b111);
wire enb16 = (count[3:0] == 4'b1111);
```

```
always @(posedge clk)
  if (enb2) begin
    // get here every 2nd cycle
  end
```
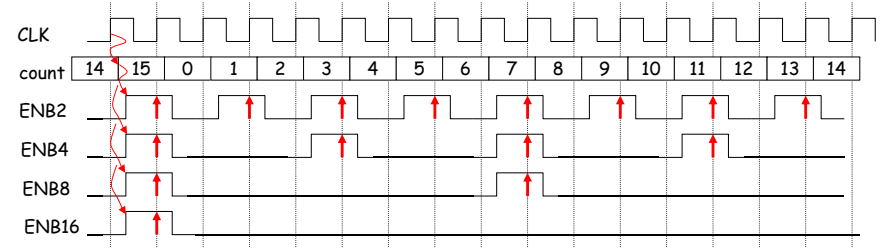

↑ = clock edge selected by enable signal

Using External Clocks

Sometimes you need to communicate synchronously with circuitry outside of the FPGA (memories, I/O, …)

Problem: different delays along internal paths for DATA and CLK change timing relationship

Solutions:

1) Bound internal delay from pin to internal reg; add that delay to setup time (t_{SU}) specification

2) Make internal clock edge aligned with external clock edge (but what about delay of pad and clock driver)
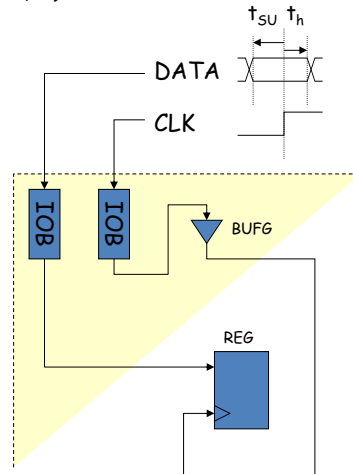
1) Bound Internal Data Delay

Solution: use registers built into the IOB pin interface:

Low-delay inputs

Low-delay tristate outputs

Virtex-II IOB Block
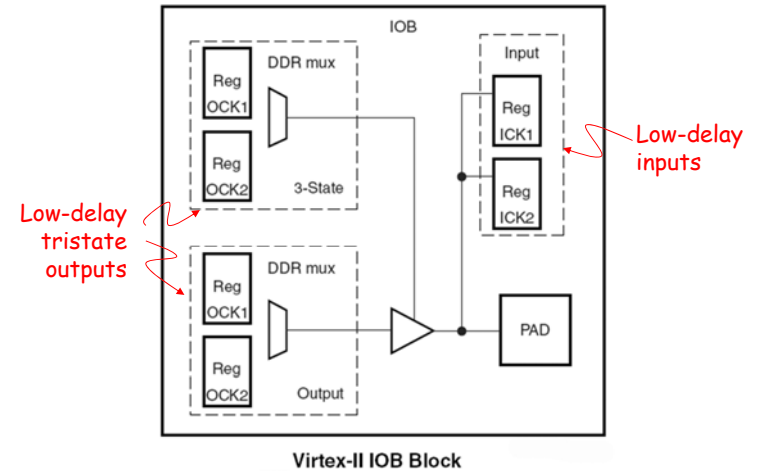
2) Align external and internal clocks
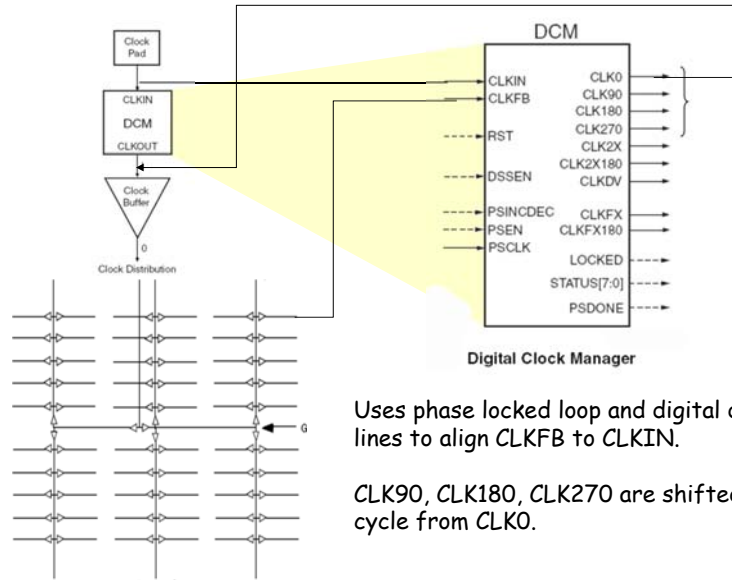
Uses phase locked loop and digital delay lines to align CLKFB to CLKIN.

CLK90, CLK180, CLK270 are shifted by $\frac{1}{4}$ cycle from CLK0.

Example: Labkit ZBT interface

The upper DCM is used to generate the de-skewed clock for the external ZBT memories. The feedback loop for this DCM includes a 2.0 inch long trace on the labkit PCB and matches in distance all of the PCB traces from the FPGA to the ZBT memories. The propagation delay from the output of the upper DCM back to its CLKFB input should be almost exactly the same as the propagation delay from the DCM output to the ZBT memories.
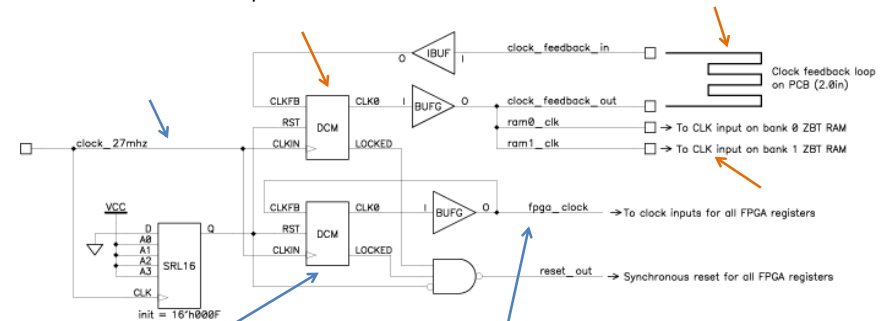
The lower DCM is used to ensure that the fpga_clock signal, which clocks all of the FPGA flip-flops, is in phase with the reference clock (clock_27mhz).

Generating Other Clock Frequencies (again)

The labkit has a 27MHz crystal (37ns period). Use DCM to generate other frequencies e.g., 65MHz to generate 1024x768 VGA video.
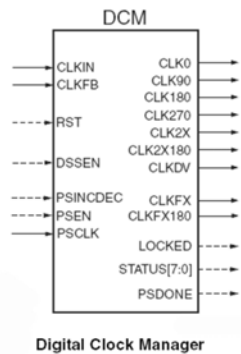
The DCM (ISE only) can also synthesize certain multiples of the CLKIN frequency (eg, multiples of 27MHz):

$$f_{CLKFX} = \left(\frac{M}{D}\right) f_{CLKIN}$$

Where M = 2--32 and D = 2--32 with a output frequency of range of 24MHz to 210MHz.

Vivado uses a Clock Wizard to simplify clock generation.

Verilog to generate 65MHz clock

```
// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));
```

$$f_{CLKFX} = \left(\frac{24}{10}\right)(27MHz) = 64.8MHz$$

RESETing to a known state

Just after configuration, all the registers/memories are in a known state (eg, default value for regs is 0). But you may need to include a RESET signal to set the initial state to what you want. *Note the Verilog initial block only works in simulation and has no effect when synthesizing hardware.*

Solution: have your logic take a RESET signal which can be asserted on start up and by an external push button:

```verilog
// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_27mhz), .Q(power_on_reset),
                .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
debounce db1(.reset(power_on_reset),.clock(clock_27mhz),
             .noisy(~button_enter),.clean(user_reset));
assign reset = user_reset | power_on_reset;
```

Debugging: making the state visible

To figure out what your circuit is doing it can be very useful to include logic that makes various pieces of state visible to the outside world. Some suggestions:

• turn the leds on and off to signal events, entry into particular pieces of code, etc.

• use the 16-character fluorescent display to show more complex state information

• drive useful data onto the ANALYZER pins and use the adapters to hook them up to the logic analyzer. Include your master clock signal and the configure the logic analyzer to sample the data on the non-active edge of the clock (to avoid setup and hold problems introduced by I/O pad delays). The logic analyzer can capture thousands of cycles of data and display the results in useful ways (including interpreting multi-bit data as samples of an analog waveform).

Encoding numbers

It is straightforward to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an n-bit number encoded in this fashion is given by the following formula:

$$v = \sum_{i=0}^{n-1} 2^i b_i$$

weights: $2^{11}\,2^{10}\,2^9\,2^8\,2^7\,2^6\,2^5\,2^4\,2^3\,2^2\,2^1\,2^0$

| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

$= 2000_{10}$

7   d   0

Oftentimes we will find it convenient to cluster groups of bits together for a more compact notation. Two popular groupings are clusters of 3 bits and 4 bits.

Seems natural to me!

03720

Octal - base 8

```
000 - 0
001 - 1
010 - 2
011 - 3
100 - 4
101 - 5
110 - 6
111 - 7
```

0x7d0

Hexadecimal - base 16

```
0000 - 0    1000 - 8
0001 - 1    1001 - 9
0010 - 2    1010 - a
0011 - 3    1011 - b
0100 - 4    1100 - c
0101 - 5    1101 - d
0110 - 6    1110 - e
0111 - 7    1111 - f
```

Binary Representation of Numbers

How to represent negative numbers?

• Three common schemes:
  – sign-magnitude, ones complement, twos complement

• Sign-magnitude: MSB = 0 for positive, 1 for negative
  – Range: $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$
  – Two representations for zero: 0000... & 1000...
  – Simple multiplication but complicated addition/subtraction

• Ones complement: if N is positive then its negative is $\overline{N}$
  – Example: 0111 = 7, 1000 = -7
  – Range: $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$
  – Two representations for zero: 0000... & 1111...
  – Subtraction is addition followed by end-around carry (subtraction is different from addition unit)

Representing negative integers

To keep our arithmetic circuits simple, we'd like to find a representation for negative numbers so that we can use a single operation (binary addition) when we wish to find the sum of two integers, independent of whether they are positive are negative.

We certainly want A + (-A) = 0. Consider the following 8-bit binary addition where we only keep 8 bits of the result:

```
      11111111
  +   00000001
  ------------
      00000000
```

which implies that the 8-bit representation of -1 is 11111111. More generally

$$-A = 0 - A$$
$$= (-1 + 1) - A$$
$$= (-1 - A) + 1$$
$$= \sim A + 1$$

Negation:
Complement
and add 1

~ means bit-wise complement

$$-\frac{A_7 \quad A_6 \quad A_5 \quad A_4 \quad A_3 \quad A_2 \quad A_1 \quad A_0}{\overline{A_7} \quad \overline{A_6} \quad \overline{A_5} \quad \overline{A_4} \quad \overline{A_3} \quad \overline{A_2} \quad \overline{A_1} \quad \overline{A_0}}$$

(with 1 1 1 1 1 1 1 1 above)

Signed integers: 2's complement

N bits

| -2^{N-1} | 2^{N-2} | $\cdots$ | $\cdots$ | $\cdots$ | 2^3 | 2^2 | 2^1 | 2^0 |

"sign bit"   Range: -2^{N-1} to $2^{N-1} - 1$ "decimal" point
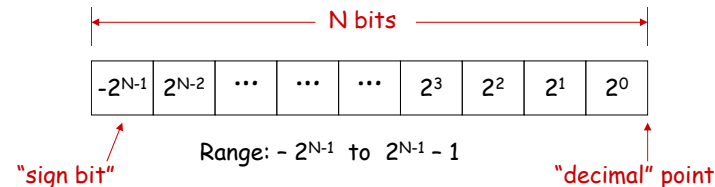
8-bit 2's complement example:

$$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$$

If we use a two's complement representation for signed integers, the same binary addition mod 2^n procedure will work for adding positive and negative numbers (don't need separate subtraction rules). The same procedure will also handle unsigned numbers!

By moving the implicit location of "decimal" point, we can represent fractions too:

$$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625$$
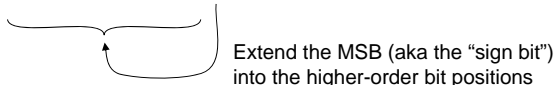

Sign extension

Consider the 8-bit 2's complement representation of:

```
42 = 00101010      -5 = ~00000101 + 1
                      =   11111010 + 1
                      =   11111011
```

What is their 16-bit 2's complement representation?

```
42  =  0000000000101010
```

```
-5  =  1111111111111011
```

Extend the MSB (aka the "sign bit") into the higher-order bit positions

---

Using Signed Arithmetic in Verilog

```
reg signed [63:0] data;
wire signed [7:0] vector;
input signed [31:0] a;
function signed [128:0] alu;

16'hC501 //an unsigned 16-bit hex value
16'shC501 //a signed 16-bit hex value
```

Use care with signed arithmetic!
```
wire signed [7:0] total;
wire [3:0] counter; // max value 15, counting widgets off the mfg line
wire signed [5:0] available;

assign total = available + counter; // does this give the correct answer?
```
NO! counter = 4'b1111 is treated as -1. Need to "append" a leading zero

```
assign total = available + {1'b0, counter};  // or use $unsigned()
assign total = available + $unsigned(counter);
```

Using Signed Arithmetic in Verilog

"<<<" and ">>>" tokens result in arithmetic (signed) left and right shifts: multiple by 2 and divide by 2.

Right shifts will maintain the sign by filling in with sign bit values during shift
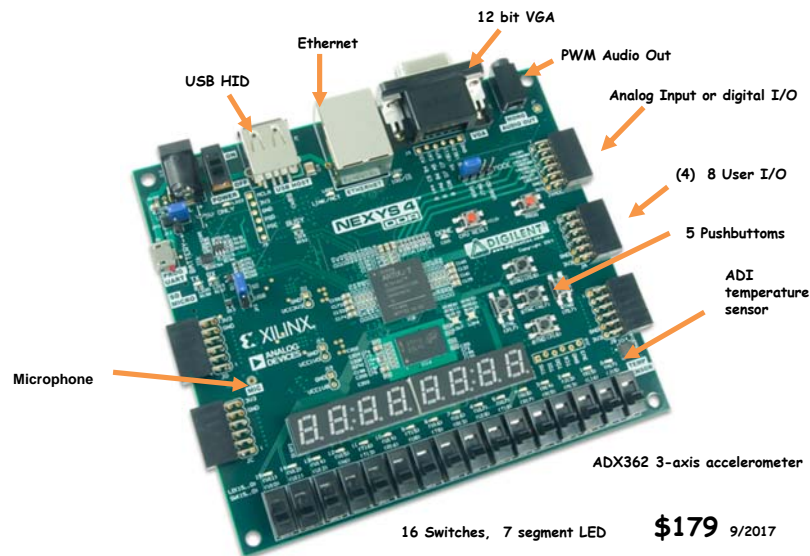
wire signed [3:0] value = 4'b1000; // -8

value >> 2   // results in 0010 or 2
value >>> 2   // results in 1110 or -2

Verilog Grading

- Logistics
  - Verilog submission with 2 days after lab checkoff. Lab must be checkoff first.
  - Resubmission for regrade permitted for Lab 2 and Lab 3 only (email grader for regrading)

- Grading
  - Proper use of blocking and non-blocking assignments
  - Readable Code with comments and consistent indenting
  - Use of default in case statement
  - Use of parameter statements for symbolic name and constants (state==5 vs state==DATA_READY)
  - Parameterized modules when appropriate
  - Readable logical flow, properly formatted (see "Verilog Editors")
  - No long nested if statements.
  - 20% off for each occurrence.

Nexys 4 - DDR

12 bit VGA
Ethernet
PWM Audio Out
USB HID
Analog Input or digital I/O
(4) 8 User I/O
5 Pushbuttoms
ADI temperature sensor
Microphone
ADX362 3-axis accelerometer
16 Switches,   7 segment LED   **$179** 9/2017

Low Cost FPGA Boards

- Basys3
  - Artix-7 FPGA
  - 12 bit VGA
  - Switches/LEDs
  - $89   (9/2017)
  - Vivado Webpack

- PYNQ-Z1
  - 650MHz dual core Cortex A9
  - Artix-7
  - 512MB DDR3
  - $65   (9/2017)
  - Vivado Webpack