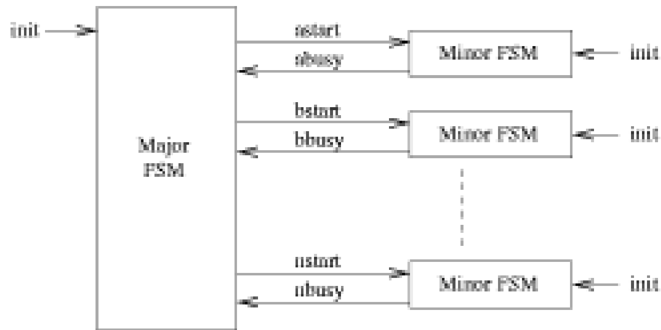
L20.1 FSM Hierarchy

Problem Statement:

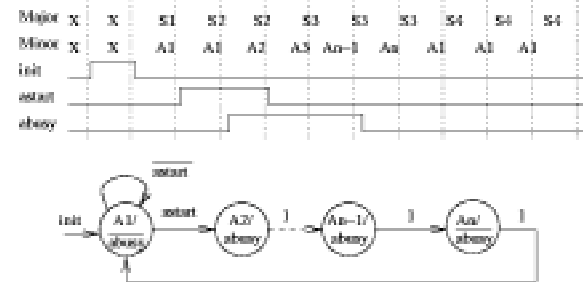
We want to coordinate (control) a sequence of operations performed by other FSMs. To do this we divide the problem into multiple FSMs which may take multiple (sometimes an unknown number) clock cycles. These minor FSMs are controlled by a major FSM. Minor FSMs may also be decomposed into multiple FSMs. All FSMs operate on the rising edge of the same clock.



Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 Introductory Digital Systems Laboratory
Monday
April 1, 2002

L20.2 Minor FSMs

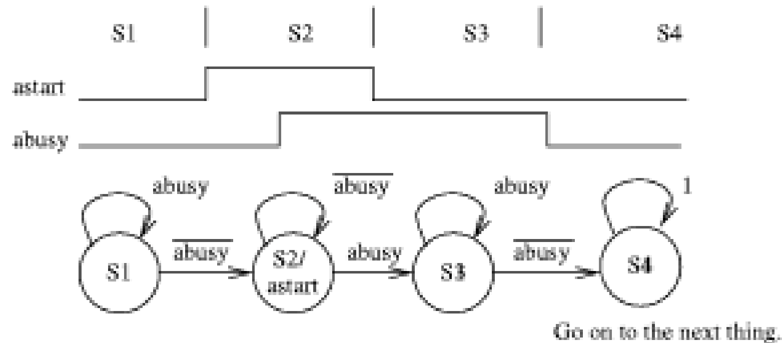
Minor FSMs are started by a control signal (start) generated by the major FSM which controls this particular minor FSM.
All minor FSMs are initialized by a signal (init) which is synchronized to the clock and asserted for a single clock period.
The makeup of minor FSMs is arbitrary, but with some constraints. The shortest path back to the starting state (reached by init) must be at least two clock cycles.



Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 Introductory Digital Systems Laboratory
Monday
April 1, 2002

L20.3 Major FSMs

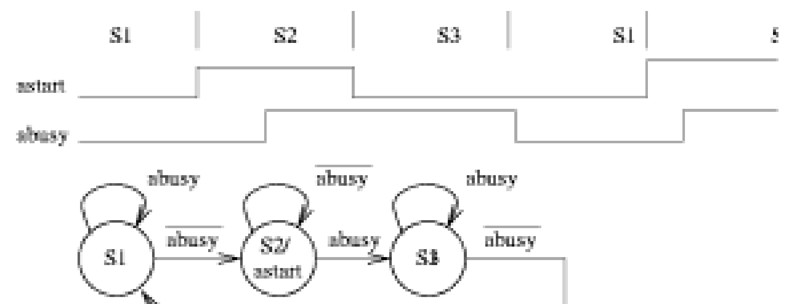
The major FSM is initialized by the same init signal supplied to the minor FSMs.
Major FSMs generate a control signal (start) for the minor FSMs. They test the busy signal generated by the minor FSM to determine both when the minor FSM is available and when it is done.



Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 Introductory Digital Systems Laboratory
Monday
April 1, 2002

L20.4 A Tight Loop

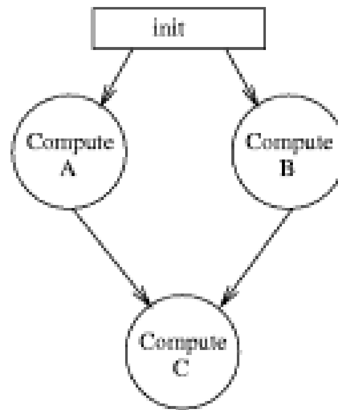
A tight loop is generated this way.
Actually, one would never want this with only one minor FSM.
One could have the start signal be a function of the state and the input, i.e., astart <= S2 and not abusy;
If one "knows" that the minor FSM will assert the busy signal one clock period after the start signal then the start signal is exactly one clock period long.



L20.5 Example Computation

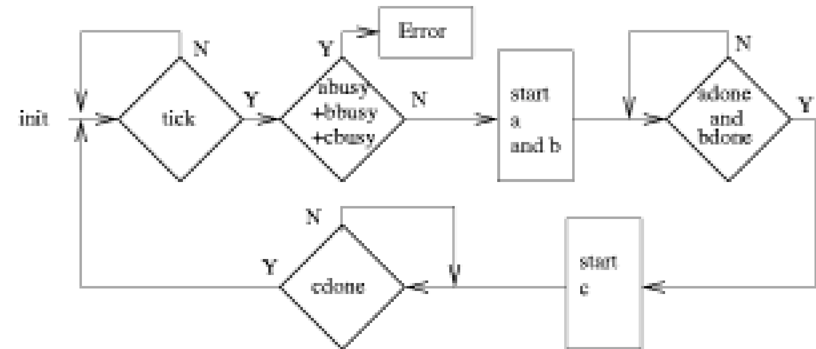
Suppose one wants to perform computation A in parallel with computation B and then, when both are finished, perform computation C.

Furthermore, we want to do this repetitvely whenever a control signal, tick, occurs. We assume that tick is generated by a separate FSM (namely, a counter of some length with appropriate combinational logic to produce the tick signal).



L20.6 Example Flow Diagram

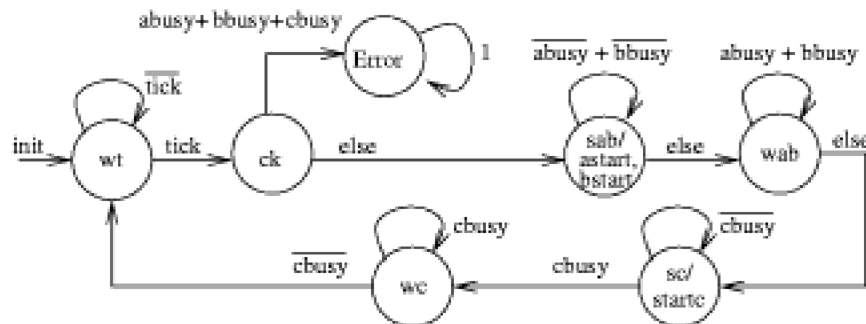
We wait for a tick to happen. Then we check to see that the previous computations have finished. If they have not, then we go to an error state and stay there until the system is reinitialized. Then we compute A and B in parallel, then C. We then wait for the next tick and start over.



L20.7 Example Major FSM

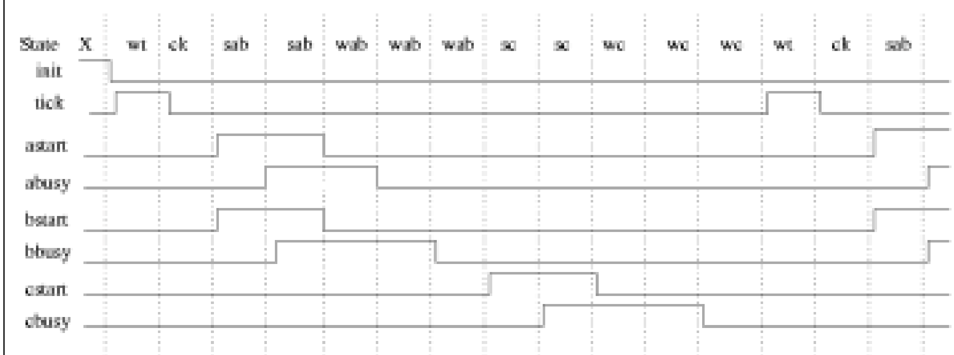
Minor FSMs for A, B, and C are as previously specified. Here is a major FSM for the previous flow diagram.

Sample waveforms are in the next slide.



L20.8 Major FSM Waveforms for Example

abusy, bbusy, and cbusy, are shown as three clock periods each in order to save room. They need not all be the same and could vary in length as long as they are two or more.



L20.9 Different Clocks

Suppose one or more minor FSMs use different clocks.
What then? Suppose the clocks are at different frequencies?

The answer is deceptively simple.
It is easy to implement and extremely difficult to debug.

Simply synchronize the start and busy signals BEFORE they are used to control transitions in the receiving state machine.
Make sure you synchronize ALL signals that change asynchronously with your clock.
If you are not sure, synchronize the signal.

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 Introductory Digital Systems Laboratory
Monday
April 1, 2002

L20.10–L3.17 Using DeMorgan's Theorem Mon. February 11, 2002

```
-- z = /((/a + c) * (a + /b) * (a + /c))
library ieee;
use ieee.std_logic_1164.all;
entity comb is
  port (a, b, c : in std_logic;
        z : out std_logic);
end comb;

architecture sf of comb is
  signal t1, t2, t3 : std_logic;
begin
  z <= not (t1 and t2 and t3);
  t1 <= (not a) or c;
  t2 <= a or (not b);
  t3 <= a or (not c);
end sf;

Compiling:  sf.vhd
DESIGN EQUATIONS
   /z = /a * /b * /c + a * c
```

```
-- z = /((/a + c) * (a + /b) * (a + /c))
-- by DeMorgan's Theorem
-- z = (a * /c) + (/a * b) + (/a * c)
-- z = (a * /c) + (/a * (b + c))
library ieee;
use ieee.std_logic_1164.all;
entity comb is
  port (a, b, c : in std_logic;
        z : out std_logic);
end comb;

architecture dm of comb is
  signal t1, t2, t3 : std_logic;
begin
  z <= (a and (not c)) or ((not a) and (b or
c));
end dm;

Compiling:  dm.vhd
DESIGN EQUATIONS
   /z = /a * /b * /c + a * c
```

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 Introductory Digital Systems Laboratory
Monday
April 1, 2002

L20.11–L5.3 Negative or Positive True? Fri. February 15, 2002

```
Library ieee;
use ieee.std_logic_1164.all;
entity neg is
  port (a1, b1, a2, b2, a3, b3 : in std_logic;
        x, n_y, n_z : out std_logic);
end neg;
architecture equations of neg is
  signal y : std_logic;
begin
  -- The only clue we have are the signal names.
  -- The next two are positive true.
  x <= a1 AND b1;
  y <= a2 AND b2;
  -- The next two are negative true.
  n_y <= not y;
  n_z <= not (a3 OR b3);
end equations;
```

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 Introductory Digital Systems Laboratory
Monday
April 1, 2002

L20.12–L6.1 VHDL Statements Tues. (Mon.) February 19, 2002

Concurrent and sequential

- Signal assignment

Concurrent

- Instantiation
- when–else
- with–select–when
- process (as a wrapper for sequential statements)

Sequential – ONLY within a process

- if–then–elsif–else
- case–when

L20.13–L6.2 Signal Assignment Tues. (Mon.) February 19, 2002

outc <= ina AND (inb OR inc); - - One needs parentheses to define order.

Basic Operators:

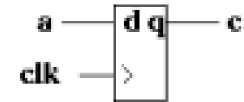
Unary Arithmetic
–
Arithmetic
+, – * also, but it is not synthesizable!
Concatenation – defined for strings and signal values
&

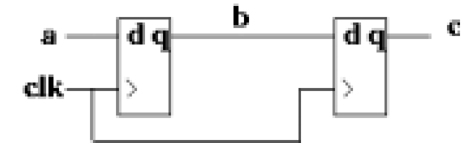
Need a use clause for the next two – use ieee.std_logic_1164.all;

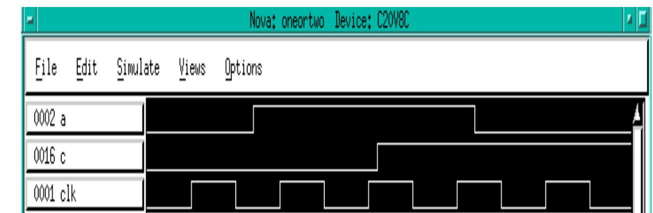
Logical
AND, NAND, OR, NOR, XOR, XNOR, NOT
Relational
=, /=, <, <=, >, >= Note that ,<= and => have other meanings also.

L20.14–L6.12 One or Two Flip–Flops Tues. (Mon.) February 19, 2002

```
- - Does this have one or two flip-flops?
library ieee;
use ieee.std_logic_1164.all;
entity reg is
  port (a, clk : in std_logic;
        c : out std_logic);
end reg;
architecture top of reg is
  signal b : std_logic;
begin
reg2: PROCESS (clk)
  BEGIN
    if rising_edge(clk) then
      b <= a;
      c <= b;
    end if;
  END PROCESS reg2;
end top;
```







L20.15–L6.13 Implicit Latch Tues. (Mon.) February 19, 2002

Memory is either implicit or explicit.

```
- - this is an example of an implicit latch
library ieee;
use ieee.std_logic_1164.all;
entity reg is
  port (d, g : in std_logic;
        q : out std_logic);
end reg;
architecture top of reg is
begin
  process (d, g)
begin
  if g = '1' then q <= d;
  - - notice that there is no ELSE
  end if;
end process;
end top;
- - this produces q = /g * q + d * g
```


L20.16–L6.15 Implicit Clocked T FF Tues.(Mon.) February 19, 2002

```
- - an example of a clocked T type register
- - rising_edge() signifies a clocked register
library ieee;
use ieee.std_logic_1164.all;
entity clked_reg is
  port (t, clk : in std_logic;
        q : out std_logic);
end clked_reg;
architecture top of clked_reg is
signal s1: std_logic;
begin
  q <= s1;
  process (clk, s1)
begin
  if rising_edge(clk) then
    if t = '1'
      then s1 <= not s1;
    end if;
  end if;
end process;
end top;
- - this produces  q.D = t * /q.Q + /t * q.Q
- - and  q.C = clk
```

L20.17–7.10 2 Conc.,1 Process i1by5sv.vhd Wed. February 20, 2002

```
library ieee;
use ieee.std_logic_1164.all;
entity by5 is port (
  x, clk  : in std_logic;
  y       : out std_logic);
end by5;
architecture state_machine of by5 is
  signal p_s, n_s : std_logic_vector(2 downto 0);
  signal p_sx : std_logic_vector(3 downto 0);
  constant state0 : std_logic_vector(2 downto 0) := "000";
  constant state1 : std_logic_vector(2 downto 0) := "001";
  constant state2 : std_logic_vector(2 downto 0) := "010";
  constant state3 : std_logic_vector(2 downto 0) := "110";
  constant state4 : std_logic_vector(2 downto 0) := "100";
begin
  state_clocked:process(clk) -- register
  begin
    if rising_edge(clk) then p_s <= n_s;
    end if;
  end process state_clocked;
-  -  combinational output spec.
  y <= '1' when ((p_s = state4) or
        (p_s = state3) or
        ((p_s = state2) and (x = '1')))
        else '0';
-  -  combinational next state
specification
  p_sx <= p_s & x;
  with p_sx select
  n_s <= state0 when "0000",
         state1 when "0001",
         state2 when "0010",
         state3 when "0011",
         state4 when "0100",
         state0 when "0101",
         state2 when "0110",
         state1 when "0111",
         state3 when "1000",
         state4 when "1001",
         state0 when others;
end architecture state_machine;
```


L20.18–L9.1 Hierarchical Design Mon. February 25, 2002

Start with a one–block block diagram.

Expand to major blocks.

Repeat expansion until blocks are simple.

Implement these simple blocks and test.

Code them in VHDL and simulate.

Use structural instantiation in VHDL to wire the blocks together.

Test the design.

L20.19–L9.19 Package Declaration Mon. February 25, 2002

```
library ieee;
-  -  Take generic and port declarations from the entity.
use ieee.std_logic_1164.all;
package gridpkg is
  component synchronizer
  port (rdy, clk : in  std_logic;
        srdy     : out std_logic); end component;
  component fsm
  port (srdy, int, err, clk : in  std_logic;
        dav, count_enb, n_clr, n_ld  : out std_logic); end component;
  component ctr
  generic (size: integer := 4);
  port (count_enb, n_clr, clk : in  std_logic;
        err : out std_logic;
        count_data : out std_logic_vector(size - 1 downto 0)); end component;
  component reg
  generic (size: integer := 4);
  port (n_ld, clk : in  std_logic;
        count_data : in std_logic_vector(size - 1 downto 0);
        data : out std_logic_vector(size - 1 downto 0)); end component;
end gridpkg;
```


L20.20–L9.22 gridtop.vhd Architecture Mon. February 25, 2002

```
architecture top of grid is
-  -  Note the use of generic map to specify the width of the ctr and reg.
  signal srdy, err : std_logic;
  signal count_enb, n_clr, n_ld : std_logic;
  signal count_data : std_logic_vector(gridsize - 1 downto 0);
begin
  sync_ckt: synchronizer
    port map (clk => clk, rdy => rdy, srdy => srdy);
  fsm_ckt: fsm
    port map (srdy => srdy, int=> int, err => err,
         clk => clk, dav => dav, count_enb => count_enb,
         n_clr => n_clr, n_ld => n_ld);
  ctr_ckt: ctr
    generic map(size => gridsize)
    port map (count_enb => count_enb, n_clr => n_clr, clk => clk,
         err => err, count_data => count_data);
  grid_data <= count_data;
  reg_ckt: reg
    generic map(size => gridsize)
    port map (n_ld => n_ld, clk => clk, count_data => count_data,
         data => data);
end top;
```

L20.21–L10.22 Don’t Care, Ampersand Wed. February 27, 2002

Don’t cares are represented by a – (hyphen).

’–’ single quotes for a character

"–––" double quotes for a string (vector)

(others => ’–’) for something independent of length

& (ampersand) to concatenate strings or signals

"01" & "111" is the same as "01111"

’0’ & "1111" is the same as "01111"

Remember, VHDL is strongly typed.

a + b is valid only if a and b are of the same length.

The result is of the SAME length as a (or b).

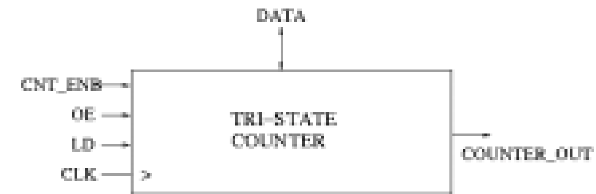
If you want it to be one bit longer, then use

c <= (’0’ & a) + (’0’ & b)
–– Of course, c must be defined to be
–– one bit longer than a.

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 Introductory Digital Systems Laboratory
Monday
April 1, 2002

L20.22–L11.9 Tri–State to Multiplex I/O pins Mon. March 1, 2002

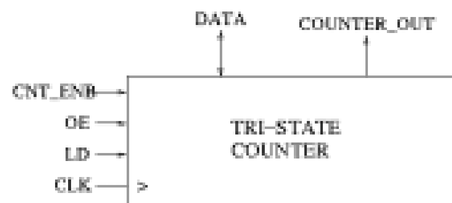
```
-- Use tri-state logic to multiplex IO pins.
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;  -- needed for integer + signal
entity ldcnt is
  generic (width : integer := 3);
  port(clk, ld, oe, cnt_enb : in std_logic;
       -- Use counter_out_out only for simulation.
       counter_out : out std_logic_vector(width -1 downto 0);
       data : inout std_logic_vector(width - 1 downto 0));
end ldcnt;
```



Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 Introductory Digital Systems Laboratory
Monday
April 1, 2002

L20.23–L11.10 Tri–State Architecture Mon. March 1, 2002

```
-- purpose: count with an output enable
architecture archldcnt of ldcnt is
signal counter : std_logic_vector(width - 1 downto 0);
begin
  counter_out <= counter;
  data <= counter when oe = '1' else (others => 'Z';
  -- N.B. Z must be UPPERCASE!
  cnt: process(clk)
  begin
    if rising_edge(clk) then
      if ld = '1' and oe = '0' then
        counter <= data;
      elsif cnt_enb = '1' then
        counter <= counter + 1;
      end if;
    end if;  -- rising_edge(clk)
  end process cnt;
end architecture archldcnt;
```



Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 Introductory Digital Systems Laboratory
Monday
April 1, 2002

L20.24–L11.22 Best Mon. March 1, 2002