L10: Major/Minor FSMs, Lab 3, and RAM/ROM Instantiation

Acknowledgements: Rex Min
Consider the following abstract FSM:

![Diagram of FSM](image)

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?
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
Inside the Minor FSM

1. Wait for a trigger from the major FSM

2. Do some useful work

3. Signal to the major FSM that work is done

---

Major FSM State

START

BUSY

CLK

Minor FSM State

---

can we speed this up?
Good idea: de-assert BUSY one cycle early

Good idea: de-assert BUSY one cycle early

Bad idea #1: T₄ may not immediately return to T₁

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

- Major FSM
- Minor FSM A
- Minor FSM B
- Minor FSM C

State transition diagram:
- States: IDLE, STAB, WTAB, STC, WT, IDLE
- Transitions:
  - STARTA → BUSYA
  - STARTB → BUSYB
  - STARTC → BUSYC

Waveform diagram:
- 4 columns for each state transition
- Time ticks (TICK)
- States: IDLE, STAB, WTAB, STC, WT, IDLE, STAB
- Must choose a shared bus for the A/D and DA
- A/D output in twos complement and ROM in sign/magnitude
- Use on-chip SRAM and ROM
- Sample rate is fixed at 20kHz
- Must use a memory based approach to FIR, not tapped delay line (i.e., registers)
- Must use Major/Minor FSM structure
**Control Flow for Lab 3**

- Wait for a fixed time interval.
- Output what was computed on the last sample.
- Store the previous A/D sample into memory.
- Start the next A/D conversion.
- Do the FIR filter algorithm which is a convolution.
- Go back for more...

---

**Diagram**

- Initialize
- Wait
- Sample
- Output Computed Sample to DAC
- Store A/D Sample
- Initiate A/D
- Convolve
- Sample
- Reset
Use LPM to Create ROM/RAM

- Click on File → MegaWizard Plug-In Manager
  - This starts up a series of windows so that you can specify parameters of the LPM module. You can choose
    - ROM
    - RAM
      - dp - Dual Ported
      - dq - Separate Inputs and Outputs
      - io - TriState Inputs and Outputs (like the 6264)
  - You choose the number of address bits and the word size.
  - You should specify a file to set the values of the ROM.
  - You can choose registered or unregistered inputs, outputs, and addresses.
ROM Contents

- Prepare a .dat file.
  - You can type this in, write a computer program, get it from another application (speech or graphics, etc.)
  - This has numbers separated by white space.
    - The default base is HEX but you can use binary or decimal if you include the following statement (before the numbers).
      # BASE = BINARY;
  - Insert, # SET_ADDRESS = 0; (specifies that data should start at address 0)

- Run dat2ntl on Athena to format your .dat file into Intel HEX
  - for details, after ‘setup 6.111’ type ‘man dat2ntl’
  - dat2ntl <filename>.dat <filename>.ntl

roman8x8.dat:
# SET_ADDRESS = 0;
7
6
5
4
3
2
1
0

dat2ntl tool on athena

See http://web.mit.edu/6.111/www/s2004/software.html for .mif format (memory initialization format)
module rom8x8 (address, q);
    input [2:0] address;
    output [7:0] q;

    wire [7:0] sub_wire0;
    wire [7:0] q = sub_wire0[7:0];

    lpm_rom lpm_rom_component (.address (address),
                              .q (sub_wire0));

    defparam lpm_rom_component.lpm_width = 8,
              lpm_rom_component.lpm_widthad = 3,
              lpm_rom_component.lpm_address_control = "UNREGISTERED",
              lpm_rom_component.lpm_outdata = "UNREGISTERED",
              lpm_rom_component.lpm_file = "rom8x8.ntl";
endmodule

Example: 8 deep by 8 bits wide

Path to location of Rom data

ROM delay
module ram4x2 (address, we, data, q);

input [1:0] address;
input we;
input [1:0] data;
output [1:0] q;

wire [1:0] sub_wire0;
wire [1:0] q = sub_wire0[1:0];

lpm_ram_dq_component (address (address),
data (data),
we (we),
q (sub_wire0));

defparam lpm_ram_dq_component.lpm_width = 2,
lpm_ram_dq_component.lpm_widthad = 2,
lpm_ram_dq_component.lpm_indata = "UNREGISTERED",
lpm_ram_dq_component.lpm_address_control = "UNREGISTERED",
lpm_ram_dq_component.lpm_outdata = "UNREGISTERED",
lpm_ram_dq_component.lpm_hint = "USE_EAB=ON";
Asynchronous RAM Simulation

module ram4x2 (address, we, data, q);
    input [1:0] address;
    input we;
    input [1:0] data;
    output [1:0] q;
endmodule

Latch interface: Data must be setup and held around the falling edge of the clock. Address must be setup before rising edge and held after falling edge.
Register interface:
Address, data and we should be setup and held on the rising edge of clock
If we=1 on the rising edge, a write operation takes place
If we=0 on the rising edge, a read operation takes place