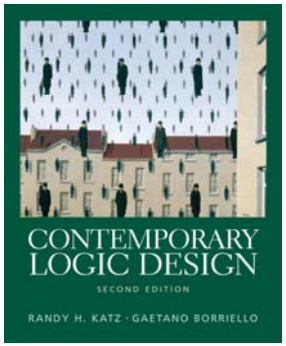
CSE 370 Spring 2006
Introduction to Digital Design

Lecture 23: Factoring FSMs
Design Examples



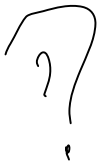
Last Lecture
- FSM minimization

Today
- Factoring FSMs
- Output encoding
- Communicating FSMs
- Design Example

Administrivia

- No class this Friday (check out the undergraduate research symposium)
- HW 8 due Monday, May 22

Quiz on Monday.



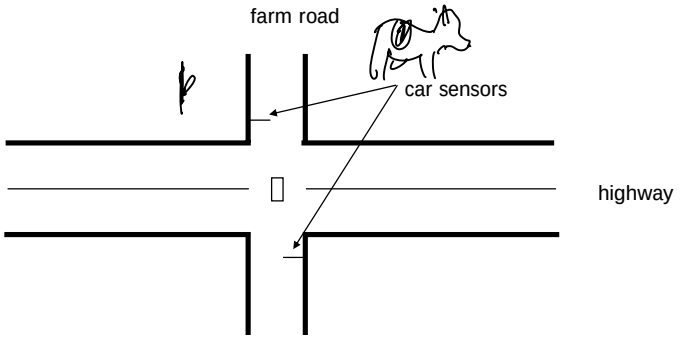
Example: traffic light controller

- A busy highway is intersected by a little used farmroad
- Detectors C sense the presence of cars waiting on the farmroad
  - with no car on farmroad, light remain green in highway direction
  - if vehicle on farmroad, highway lights go from Green to Yellow to Red, allowing the farmroad lights to become green
  - these stay green only as long as a farmroad car is detected but never longer than a set interval
  - when these are met, farm lights transition from Green to Yellow to Red, allowing highway to return to green
  - even if farmroad vehicles are waiting, highway gets at least a set interval as green

  - Assume: short time interval of yellow light is five cycles
  - Assume: max time for green on farm road and minimum green on highway is 20 cycles

Example: traffic light controller (cont')

- Highway/farm road intersection



Example: traffic light controller (cont')

- Tabulation of inputs and outputs

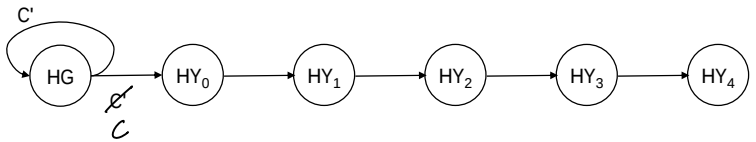
inputs	description	outputs	description
reset	place FSM in initial state	HG, HY, HR	assert green/yellow/red highway lights
C	detect vehicle on the farm road	FG, FY, FR	assert green/yellow/red highway lights

- Tabulation of unique states – some light configurations imply others

state	description
HG	highway green (farm road red)
HY	highway yellow (farm road red)
FG	farm road green (highway red)
FY	farm road yellow (highway red)

Example: traffic light controller (cont')

- Initial attempt:



- Continuing in this way would quickly get us to huge FSM
- Solution: Factor the FSM

Example: traffic light controller (cont')

- Assume you have an interval timer that generates:
  - a short time pulse (TS) and
  - a long time pulse (TL),
  - in response to a set (ST) signal.
  - TS is to be used for timing yellow lights and TL for green lights

More inputs to FSM + outputs

Example: traffic light controller (cont')

- Tabulation of inputs and outputs

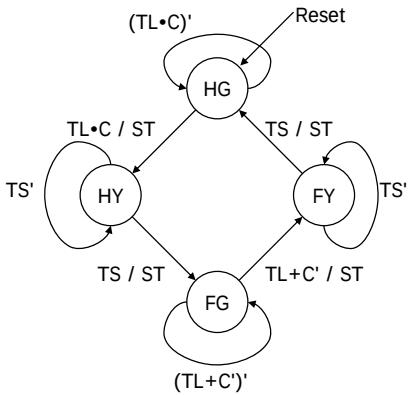
inputs	description	outputs	description
reset	place FSM in initial state	HG, HY, HR	assert green/yellow/red highway lights
C	detect vehicle on the farm road	FG, FY, FR	assert green/yellow/red highway lights
TS	short time interval expired	ST	start timing a short or long interval
TL	long time interval expired		

- Tabulation of unique states – some light configurations imply others

state	description
HG	highway green (farm road red)
HY	highway yellow (farm road red)
FG	farm road green (highway red)
FY	farm road yellow (highway red)

Example: traffic light controller (cont')

- State diagram



Example: traffic light controller (cont')

- Generate state table with symbolic states
- Consider state assignments

output encoding – similar problem to state assignment
(Green = 00, Yellow = 01, Red = 10)

Inputs C	TL	TS	Present State	Next State	Outputs ST	H	F
0	–	–	HG	HG	0	Green	Red
–	0	–	HG	HG	0	Green	Red
1	1	–	HG	HY	1	Green	Red
–	–	0	HY	HY	0	Yellow	Red
–	–	1	HY	FG	1	Yellow	Red
1	0	–	FG	FG	0	Red	Green
0	–	–	FG	FY	1	Red	Green
–	1	–	FG	FY	1	Red	Green
–	–	0	FY	FY	0	Red	Yellow
–	–	1	FY	HG	1	Red	Yellow

SA1:	HG = 00	HY = 01	FG = 11	FY = 10	
SA2:	HG = 00	HY = 10	FG = 01	FY = 11	
SA3:	HG = 0001	HY = 0010	FG = 0100	FY = 1000	(one-hot)

Logic for different state assignments

- SA1

NS1 = C•TL'•PS1•PS0 + TS•PS1'•PS0 + TS•PS1•PS0' + C'•PS1•PS0 + TL•PS1•PS0
NS0 = C•TL•PS1'•PS0' + C•TL'•PS1•PS0 + PS1'•PS0

ST = C•TL•PS1'•PS0' + TS•PS1'•PS0 + TS•PS1•PS0' + C'•PS1•PS0 + TL•PS1•PS0
H1 = PS1 H0 = PS1'•PS0
F1 = PS1' F0 = PS1•PS0'

- SA2

NS1 = C•TL•PS1' + TS'•PS1 + C'•PS1'•PS0
NS0 = TS•PS1•PS0' + PS1'•PS0 + TS'•PS1•PS0

ST = C•TL•PS1' + C'•PS1'•PS0 + TS•PS1
H1 = PS0 H0 = PS1•PS0'
F1 = PS0' F0 = PS1•PS0

- SA3

NS3 = C'•PS2 + TL•PS2 + TS'•PS3 NS2 = TS•PS1 + C•TL'•PS2
NS1 = C•TL•PS0 + TS'•PS1 NS0 = C'•PS0 + TL'•PS0 + TS•PS3

ST = C•TL•PS0 + TS•PS1 + C'•PS2 + TL•PS2 + TS•PS3
H1 = PS3 + PS2 H0 = PS1
F1 = PS1 + PS0 F0 = PS3

Output-based encoding

- Reuse outputs as state bits - use outputs to help distinguish states
  - why create new functions for state bits when output can serve as well
  - fits in nicely with synchronous Mealy implementations

Inputs C	TL	TS	Present State	Next State	Outputs ST	H	F
0	–	–	HG	HG	0	00	10
–	0	–	HG	HG	0	00	10
1	1	–	HG	HY	1	00	10
–	–	0	HY	HY	0	01	10
–	–	1	HY	FG	1	01	10
1	0	–	FG	FG	0	10	00
0	–	–	FG	FY	1	10	00
–	1	–	FG	FY	1	10	00
–	–	0	FY	FY	0	10	01
–	–	1	FY	HG	1	10	01

HG = ST' H1' H0' F1 F0' + ST H1 H0' F1' F0
HY = ST H1' H0' F1 F0' + ST' H1' H0 F1 F0'
FG = ST H1' H0 F1 F0' + ST' H1 H0' F1' F0'
HY = ST H1 H0' F1' F0' + ST' H1 H0' F1' F0

Output patterns are unique to states, we do not need ANY state bits – implement 5 functions (one for each output) instead of 7 (outputs plus 2 state bits)

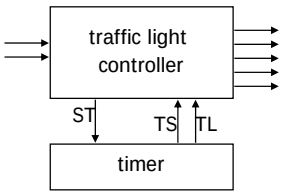
Current state assignment approaches

- For tight encodings using close to the minimum number of state bits
  - best of 10 random seems to be adequate (averages as well as heuristics)
  - heuristic approaches are not even close to optimality
  - used in custom chip design
- One-hot encoding
  - easy for small state machines
  - generates small equations with easy to estimate complexity
  - common in FPGAs and other programmable logic
- Output-based encoding
  - ad hoc - no tools
  - most common approach taken by human designers
  - yields very small circuits for most FSMs

Sequential logic optimization summary

- State minimization
  - straightforward in fully-specified machines
  - computationally intractable, in general (with don't cares)
- State assignment
  - many heuristics
  - best-of-10-random just as good or better for most machines
  - output encoding can be attractive (especially for PAL implementations)

Traffic light controller as two communicating FSMs



Traffic light controller FSM

- Specification of inputs, outputs, and state elements

```
module FSM(HR, HY, HG, FR, FY, FG, ST, TS, TL, C, reset, Clk);
  output     HR;
  output     HY;
  output     HG;
  output     FR;
  output     FY;
  output     FG;
  output     ST;
  input      TS;
  input      TL;
  input      C;
  input      reset;
  input      Clk;

  reg [6:1]  state;
  reg        ST;
```

```
parameter highwaygreen   = 6'b001100;
parameter highwayyellow  = 6'b010100;
parameter farmroadgreen  = 6'b100001;
parameter farmroadyellow = 6'b100010;


assign HR = state[6];
assign HY = state[5];
assign HG = state[4];
assign FR = state[3];
assign FY = state[2];
assign FG = state[1];
```

specify state bits and codes for each state as well as connections to outputs

Traffic light controller FSM (cont'd)

```
initial begin state = highwaygreen; ST = 0; end

always @(posedge Clk)
  begin
    if (reset)
      begin state = highwaygreen; ST = 1; end
    else
      begin
        ST = 0;
        case (state)
          highwaygreen:
            if (TL & C) begin state = highwayyellow; ST = 1; end
          highwayyellow:
            if (TS) begin state = farmroadgreen; ST = 1; end
          farmroadgreen:
            if (TL | !C) begin state = farmroadyellow; ST = 1; end
          farmroadyellow:
            if (TS) begin state = highwaygreen; ST = 1; end
        endcase
      end
  end
endmodule
```

case statement triggerred by clock edge

Timer for traffic light controller

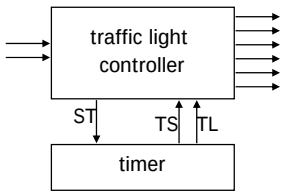
- Another FSM

```
module Timer(TS, TL, ST, Clk);
  output TS;
  output TL;
  input     ST;
  input     Clk;
  integer   value;

  assign TS = (value >=  4); //  5 cycles after reset
  assign TL = (value >= 14); // 15 cycles after reset

  always @(posedge ST) value = 0; // async reset

  always @(posedge Clk) value = value + 1;

endmodule
```

Complete traffic light controller

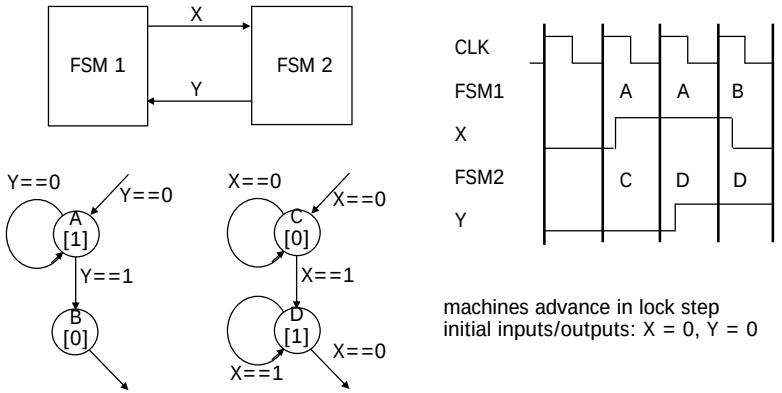
- Tying it all together (FSM + timer)
  - structural Verilog (same as a schematic drawing)

```
module main(HR, HY, HG, FR, FY, FG, reset, C, Clk);
 output HR, HY, HG, FR, FY, FG;
 input  reset, C, Clk;

  Timer part1(TS, TL, ST, Clk);
  FSM   part2(HR, HY, HG, FR, FY, FG, ST, TS, TL, C, reset, Clk);
endmodule
```



Communicating finite state machines

- One machine's output is another machine's input



machines advance in lock step
initial inputs/outputs: X = 0, Y = 0

Sequential logic examples

- Basic design approach: a 4-step design process
- Implementation examples and case studies
  - finite-string pattern recognizer
  - complex counter
  - door combination lock

General FSM design procedure

- (1) Determine inputs and outputs
- (2) Determine possible states of machine
  - state minimization
- (3) Encode states and outputs into a binary code
  - state assignment or state encoding
  - output encoding
  - possibly input encoding (if under our control)
- (4) Realize logic to implement functions for states and outputs
  - combinational logic implementation and optimization
  - choices in steps 2 and 3 can have large effect on resulting logic

Finite string pattern recognizer (step 1)

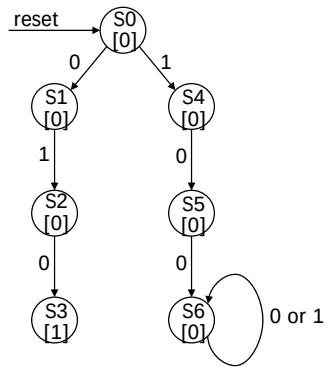
- Finite string pattern recognizer
  - one input (X) and one output (Z)
  - output is asserted whenever the input sequence …010… has been observed, as long as the sequence …100… has never been seen
- Step 1: understanding the problem statement
  - sample input/output behavior:

X: 0 0 1 0 1 0 1 0 0 1 0 …
Z: 0 0 0 1 0 1 0 1 0 0 0 …

X: 1 1 0 1 1 0 1 0 0 1 0 …
Z: 0 0 0 0 0 0 0 1 0 0 0 …

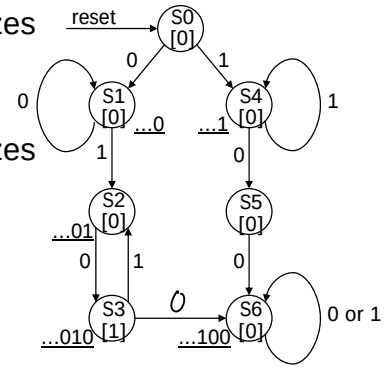
Finite string pattern recognizer (step 2)

- Step 2: draw state diagram
  - for the strings that must be recognized, i.e., 010 and 100
  - a Moore implementation



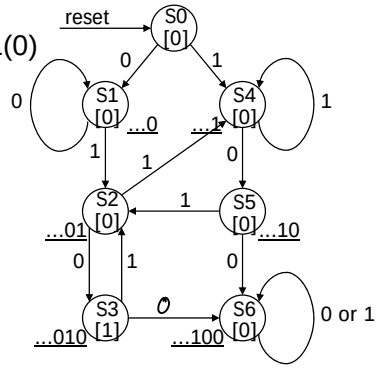
Finite string pattern recognizer (step 2, cont'd)

- Exit conditions from state S3: have recognized …010
  - if next input is 0 then have …0100 = ...100 (state S6)
  - if next input is 1 then have …0101 = …01 (state S2)
- Exit conditions from S1: recognizes strings of form …0 (no 1 seen)
  - loop back to S1 if input is 0
- Exit conditions from S4: recognizes strings of form …1 (no 0 seen)
  - loop back to S4 if input is 1



Finite string pattern recognizer (step 2, cont'd)

- S2 and S5 still have incomplete transitions
  - S2 = …01; If next input is 1, then string could be prefix of (01)1(00) S4 handles just this case
  - S5 = …10; If next input is 1, then string could be prefix of (10)1(0) S2 handles just this case
- Reuse states as much as possible
  - look for same meaning
  - state minimization leads to smaller number of bits to represent states
- Once all states have a complete set of transitions we have a final state diagram



Finite string pattern recognizer (step 3)

- Verilog description including state assignment (or state encoding)

```
module string (clk, X, rst, Q0, Q1, Q2, Z);
input clk, X, rst;
output Q0, Q1, Q2, Z;

parameter S0 = [0,0,0]; //reset state
parameter S1 = [0,0,1]; //strings ending in   ...0
parameter S2 = [0,1,0]; //strings ending in  ...01
parameter S3 = [0,1,1]; //strings ending in ...010
parameter S4 = [1,0,0]; //strings ending in   ...1
parameter S5 = [1,0,1]; //strings ending in  ...10
parameter S6 = [1,1,0]; //strings ending in ...100

reg state[0:2];

assign Q0 = state[0];
assign Q1 = state[1];
assign Q2 = state[2];
assign Z = (state == S3);
```

```
always @(posedge clk) begin
  if (rst) state = S0;
  else
    case (state)
      S0: if (X) state = S4 else state = S1;
      S1: if (X) state = S2 else state = S1;
      S2: if (X) state = S4 else state = S3;
      S3: if (X) state = S2 else state = S6;
      S4: if (X) state = S4 else state = S5;
      S5: if (X) state = S2 else state = S6;
      S6: state = S6;
      default: begin
        $display ("invalid state reached");
        state = 3'bxxx;
      end
    endcase
end

endmodule
```

Finite string pattern recognizer

- Review of process
  - understanding problem
    - write down sample inputs and outputs to understand specification
  - derive a state diagram
    - write down sequences of states and transitions for sequences to be recognized
  - minimize number of states
    - add missing transitions; reuse states as much as possible
  - state assignment or encoding
    - encode states with unique patterns
  - simulate realization
    - verify I/O behavior of your state diagram to ensure it matches specification