CEG 5010: Reconfigurable Computing Finite State Machines

"Tortise: You don't need to give an infinite number of monkeys an infinite amount of time to write Hamlet. A finite number of monkeys and a finite amount of time will do just fine. And like my father used to say, never use an infinite number of monkeys when a finite number will do."

- Mike Schiraldi, in a sci.math posting

Course Summary (Subject to change)

- **INTRODUCTION AND FPGAs**
- Introduction / Reconfigurable computing machines
- FPGA Architecture

- **COMPUTER ARITHMETIC**
- Fixed point arithmetic
- Elementary functions
- Digit serial processing
- Distributed arithmetic
- Floating point arithmetic

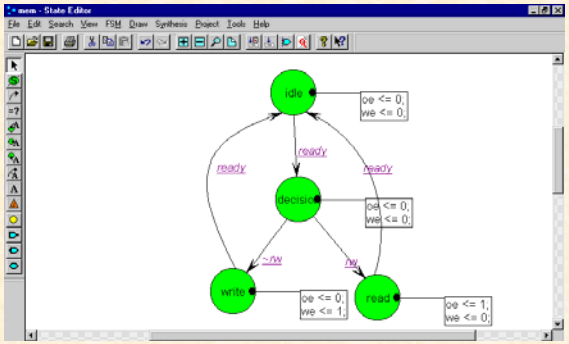
- **COMPUTER ARCHITECTURE**
- Data path design
- Translating C to VHDL
- Microcoding

- **CASE STUDIES**
- Digital signal processing
- Cryptography
- Rapid prototyping
- Coprocessors

Introduction

- Most practical digital designs built from datapath+control
  - control implemented as a finite state machine
- Describe how to implement finite state machines
  - how to construct a FSM
  - how to describe using VHDL
  - examples using memory controller and UART
  - how to optimize for time and area

Simple Example: A memory controller



First step: define a enumeration type

- type state_t is (idle, decision, read, write);
- signal curstate, nextstate : state_t;

Second step: make combinational process to implement transisitions

```
statecomb: process(curstate, rw, ready)
begin
   …
end process statecomb;
```

Combinatorial process

- This process gives the next state and outputs from the current state
  - Moore state machine
  - outputs functions only of the current state
- Note that the resulting circuit is all combinatorial
- Is easily derived from the state diagram

```
statecomb: process(curstate, rw, ready)
begin
   case curstate is
      when idle =>
         oe <= '0'; we <= '0';
         if (ready = '1') then
            nextstate <= decision;
         else  -- this statement not necessary
            nextstate <= idle;
         end if;
      when decision =>
         oe <= '0'; we <= '0';
         if (rw = '1') then
            nextstate <= read;
         else
            nextstate <= write;
         end if;
      ….
   end case;
end process statecomb;
```


Sequential process

- We also need a clocked process to make the actual state transition

```
stateclkd: process(clk)
begin
   if (clk'event and clk = '1')
   then
         curstate <= nextstate;
   end if;
end process stateclkd;
```

Adding synchronous reset (how NOT to do it)

```
if (reset = '1') then
    nextstate <= idle;
else
    case curstate is
        when idle =>
            oe <= '0'; we <= '0';
            if (ready = '1') then
                nextstate <= decision;
            else
                nextstate <= idle;
            end if;
    …
end if;
```

- Hint: what happens to we when reset?

Previous code

- What does this do?

```
process begin
  if (cond)
    a <= b;
  end if;
end process;
```

- How about this?

```
process begin
  if (reset)
    x <= '1';
  else
    a <= b;
  end if;
end process;
```


Previous code

- What does this do?

```
process begin
  if (cond)
    a <= b;
  end if;
end process;
```

- How about this?

```
process begin
  if (reset)
    x <= '1';
  else
    a <= b;
  end if;
end process;
```

- In VHDL, a retains value after the assignment. This is "implied memory"
  - a = cond.b + /cond.a
  - x=reset+/reset.x
  - a=/reset.b+reset.a

Implied memory

- What circuit does the following generate?

```
if (curstate = s0) then
    outa <= '1';
elsif (curstate = s1) then
    outb <= '1';
else
    outc <= '1';
end if;
```

- How about?

```
outa <= '0';
outb <= '0';
outc <= '0';
if (curstate = s0) then
    outa <= '1';
elsif (curstate = s1) then
    outb <= '1';
else
    outc <= '1';
end if;
```

Combinatorial logic

- *To synthesize combinational logic using a process, all inputs to the design must appear in the sensitivity list.*
  - If not synthesis and simulation may differ
- *To synthesize combinational logic using a process, all objects must be assigned under all conditions.*
- Rule: for combinatorial logic, all input combinations involve the variable being assigned before it is read
- Hint: best way is to assign first as in previous slide column 2

http://www.doulos.com/knowhow/vhdl_designers_guide

To do synchronous reset

```
if (reset = '1') then
    oe <= '-'; we <= '-';
    nextstate <= idle;
else
    case curstate is
       when idle =>
            oe <= '0'; we <= '0';
            if (ready = '1') then
               nextstate <= decision;
            else
               nextstate <= idle;
            end if;
    …
end if;
```

- '-' means "don't care"
- it is easy to make a mistake here so be careful!
  - Check the number of inferred latches/registers is what you expect

A better way

```
statecomb: process(curstate, rw, ready)
begin
    case curstate is
       when idle =>
            oe <= '0'; we <= '0';
            if (ready = '1') then
               nextstate <= decision;
            else
               nextstate <= idle;
            end if;
       ….
    end case;
    if (reset = '1') then
       nextstate <= idle;
    end if;
end process statecomb;
```

- This takes advantage of the fact that the last assignment to nextstate is applied
- We do not need to remember to make all the outputs "don't cares"

Asynchronous reset

```
stateclkd: process(clk)
begin
   if (reset = '1') then
         curstate <= init;
   else if (clk'event and clk = '1')
   then
         curstate <= nextstate;
   end if;
end process stateclkd;
```

- May require less hardware since the FPGA can use the reset of the flip-flop rather than additional combinatorial logic
  - could also use GSR feature

Another way to describe the circuit (one process instead of two)

```
if CLK'event and CLK = '1' then
    if reset='1' then
        state <= idle;
    else
    case state is
        when decision =>
            if rw then
                state <= read;
            elsif ~rw then
                state <= write;
            end if;
        when idle =>
            if ready then
                state <= decision;
            end if;
    ...
end process;

-- signal assignment statements for
    combinatorial outputs
oe <= 0 when (state = decision) else
    1 when (state = read) else
    0 when (state = write) else
    0;

we <= 0 when (state = decision) else
    0 when (state = read) else
    1 when (state = write) else
    0;

end state_arch;
```


Optimization for speed/area

- Often there is no need
  - FPGA usually fast enough so almost any description is acceptable
    - in this case use the most straightforward implementation (easiest to write/read/debug)
  - otherwise giving timing constraints often does the job
- Optimization
  - FSM dependent
    - most efficient FSM depends on # states, complexity of logic etc
  - architecture dependent
    - different for FPGA/CPLD/ASIC
  - speed max freq?, smallest clock to output delay?
    - interrelated but we can optimize one in favor of another

Other FSM architectures

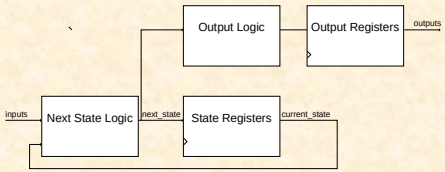
- We will look at 4 different implementations
  - outputs decoded from state bits combinatorially
  - output decoded in parallel output registers
  - outputs encoded within state bits
  - one hot encoding

Outputs decoded from state bits combinatorially

- This is the technique described earlier
- Output logic is a combinatorial function of the state
  - suffers from an additional delay from output of state bits to the output signals

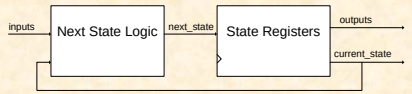
Output decoded in parallel output registers

- Decode the outputs before the state bits are registered i.e. at the output of the nextstate
- Outputs are generated earlier than the previous version this is because the output combinatorial logic delay is removed
- Area larger because we have additional registers for the output
- Speed we have two combinatorial delays to the output so maximum speed may be affected



Outputs encoded within state bits

- An example is a counter - the output is also the state
- Find a state encoding which directly generates the output
  - design more difficult
    - we need to find such an encoding
    - code more complicated and difficult to read
  - can resolve clashes by adding outputs to differentiate similar bits
    - e.g. for the memory controller can add ST0 to differentiate the IDLE and DECISION states
- Efficiency depends on the particular circuit
  - may be more/less efficient in area/speed, best way is to try it for a critical design



One hot encoding

- Use n flip-flops to represent an n-state FSM
- Significantly reduces the logic for outputs and nextstate
  - why?
  - also reduces logic to generate outputs
  - at the expense of more registers
- FPGAs
  - rich in registers
  - for max speed and small to medium FSMs often the best choice

Mealy machines

- All FSMs we have discussed are Moore machines since outputs are functions only of the current state
  - for Mealy FSMs, outputs functions of current state and inputs

Fault tolerance

- What happens if for whatever reason the FSM gets into an undefined state
  - fault tolerant designs can recover

```
case curstate is
   …
   when others => nextstate <= idle;
end case;
```

  - alternatively could enter an error state that somehow reports
  - explicit “don’t care”

```
when others => nextstate <= "------";
```


Fault tolerance for one-hot designs

- There are many more possible illegal states for 1-hot designs
  - could detect with

```
badstate <= (s1 and (s2 or s3 or s4)) or
            (s2 and (s1 or s3 or s4))
            (s3 and (s1 or s3 or s4))
            (s4 and (s1 or s2 or s3));
```

  - this is quite expensive, affects performance and area
  - could be pipelined

BlockRAM Applications, State Machine

Store next address in the ROM

Conditional jump info is being entered as additional address inputs

One BlockRAM can be split in two:

One 18K dual-port = two 9K BlockRAMs

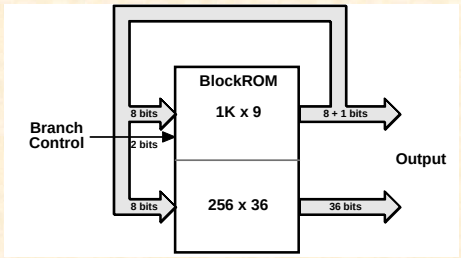
with independent everything:

(R/W, clock, address, data content, aspect ratio)

200 MHz, independent of complexity

Slide courtesy of Peter Alfke

Fast State Machine in one BlockRAM



256 states, 4-way branch (or 128 states, 8-way)

36 optional parallel outputs from the second port

200 MHz operation, independent of code

Slide courtesy of Peter Alfke

Conclusions

- Studied many different ways to implement FSMs
- Mostly we use
  - outputs decoded combinatorially from state registers
  - one-hot (particularly good for high speed small-medium FSMs on FPGAs)
  - RAM/ROM

Review Questions

- Implement a latch, D-flip flop and SR flip flop in VHDL (check with any text book)
- Implement a simple 2-bit counter with an “up” and “down” switch in VHDL (i.e. pressing up increases the count and down decreases it)
- Add a synchronous reset to it
- Study the FPGA resource usage and make sure what you implemented is what was expected
- Make sure you understand how to implement a ROM-based FSM

References

- http://www.actel.com/documents/hdlcode_ug.pdf is an excellent tutorial on HDL, particularly the “Technology Independent Coding Style” chapter.

Project Ideas

Functions well suited to FPGA acceleration

- searching
- sorting
- signal processing
- audio/video/image manipulation
- basestations
- encryption
- error correction
- coding/decoding
- packet processing
- random-number generation for Monte Carlo simulations

- Specific ideas
  - Parallel sort/search
  - Linear regression
  - Artificial neural network
  - Genetic algorithm
  - RC4/AES/RSA encryption
  - Regular expression compiler
  - Mersenne twister RNG
  - Elementary functions
  - Module generators for different applications
  - Simple high level synthesis tools
  - Graphics adaptor (vga/svga)
  - Floating point unit
  - FSL link
  - Compression (MP3, video, Huffman, Lempel-Ziv)
  - DLL, PLL, DAC, wireless link etc
  - SD, CF, IR, hard disk etc
  - Embedded applications using the microblaze