Announcements

- Problem set 3 due
- Lab 4 hints
  - Do a good clean design
    - Lab 5 and the final project will reuse most of this lab
    - You'll regret any mistakes here for the rest of the quarter
    - We've detailed the interfaces to try and make this easier
  - Understand how the lab works, not just how to build it
    - How sine waves are generated (handout)
    - How to make FSM Timing Diagrams to deal with memories and data paths
      (last section of lab handout)
    - How all the modules interact (a good block diagram really helps)
  - Split up module design and testing
    - Decide on how a module should work as a group
    - Have one person design it and the other person write the test bench
    - Finds errors in design and description
  - Watch your timing
    - Check your synthesis reports to make sure you are meeting timing
      (if your critical path is too long your design invalid.)
    - Run the timing analyzer and find the paths that are too slow
    - Find those paths on your block diagram and figure out where to insert pipeline registers to split up the long paths without losing synchronization.
      (Hint: before and after multipliers, but watch out for control paths!)
Microcode – an FSM realized with a memory array

- Original concept by Wilkes (1951)
  - Put state table in a memory (ROM or RAM)
  - ROM Address = \{current state, inputs\}
  - ROM Output = \{next state, outputs\}

Effectively replace the “combinational next state logic” block with the full truth table.

Why is this good? Why might this be bad?

Microcode – the picture

Looks a lot like the combinational logic block from regular FSMs.

Memory size: \(2^{s+i}\) addresses
s+o bits per address
Total: \(2^{s+i}(s+o)\) bits
Example: Simple Light-Traffic Controller

Re-writing State Table Of Example

Note: We’ve enumerated every possible combination of inputs and state. Is this efficient?
Microcode Of Light-Traffic Controller

module ucodeTLC(clk, rst, in, out) ;
parameter n = 1 ; // input width
parameter m = 6 ; // output width
parameter k = 2 ; // bits of state

input clk, rst ;
input [n-1:0] in ;
output [m-1:0] out ;

wire [k-1:0] next, state ;
wire [k+m-1:0] uinst ;

DFF #(k) state_reg(clk, next, state) ; // state register
DFF #(m) out_reg(clk, uinst[m-1:0], out) ; // output register
ROM #(n+k,m+k) uc((state, in), uinst) ; // microcode store
assign next = rst ? {k{1'b0}} : uinst[m+k-1:m] ; // reset state
endmodule
Waveforms Of Light-Traffic Controller Microcode

A New-and-Improved Light-Traffic Controller

Additional functions to original Light-Traffic Controller:

- Light stays green in east-west direction as long as car_ew is true.
- Light stays green in north-south direction for a minimum of 3 states (GNS1, GNS2, and GNS3).
- After a yellow light, lights should go red in both directions for 1 cycle before turning new light green.
Light-Traffic Control State Diagram

Light-Traffic Controller State Microcode

<table>
<thead>
<tr>
<th>Address</th>
<th>State</th>
<th>car_ew</th>
<th>Next State</th>
<th>Output</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>GNS1(000)</td>
<td>0</td>
<td>GNS2(001)</td>
<td>00001</td>
<td>001100001</td>
</tr>
<tr>
<td>0001</td>
<td>GNS1(000)</td>
<td>1</td>
<td>GNS2(001)</td>
<td>00001</td>
<td>001100001</td>
</tr>
<tr>
<td>0010</td>
<td>GNS2(001)</td>
<td>0</td>
<td>GNS3(010)</td>
<td>00001</td>
<td>010100001</td>
</tr>
<tr>
<td>0011</td>
<td>GNS2(001)</td>
<td>1</td>
<td>GNS3(010)</td>
<td>00001</td>
<td>010100001</td>
</tr>
<tr>
<td>0100</td>
<td>GNS3(010)</td>
<td>0</td>
<td>GNS3(010)</td>
<td>00001</td>
<td>010100001</td>
</tr>
<tr>
<td>0101</td>
<td>GNS3(010)</td>
<td>1</td>
<td>YNS(011)</td>
<td>00001</td>
<td>011100001</td>
</tr>
<tr>
<td>0110</td>
<td>YNS(011)</td>
<td>0</td>
<td>RNS(100)</td>
<td>00001</td>
<td>100010001</td>
</tr>
<tr>
<td>0111</td>
<td>YNS(011)</td>
<td>1</td>
<td>RNS(100)</td>
<td>00001</td>
<td>100010001</td>
</tr>
<tr>
<td>1000</td>
<td>RNS(100)</td>
<td>0</td>
<td>GEW(101)</td>
<td>00100</td>
<td>101001001</td>
</tr>
<tr>
<td>1001</td>
<td>RNS(100)</td>
<td>1</td>
<td>GEW(101)</td>
<td>00100</td>
<td>101001001</td>
</tr>
<tr>
<td>1010</td>
<td>GEW(101)</td>
<td>0</td>
<td>YEW(110)</td>
<td>00100</td>
<td>110100000</td>
</tr>
<tr>
<td>1011</td>
<td>GEW(101)</td>
<td>1</td>
<td>GEW(101)</td>
<td>00100</td>
<td>110100000</td>
</tr>
<tr>
<td>1100</td>
<td>YEW(110)</td>
<td>0</td>
<td>REW(111)</td>
<td>00100</td>
<td>111000100</td>
</tr>
<tr>
<td>1101</td>
<td>YEW(110)</td>
<td>1</td>
<td>REW(111)</td>
<td>00100</td>
<td>111000100</td>
</tr>
<tr>
<td>1110</td>
<td>REW(111)</td>
<td>0</td>
<td>GNS(000)</td>
<td>00100</td>
<td>0000001001</td>
</tr>
<tr>
<td>1111</td>
<td>REW(111)</td>
<td>1</td>
<td>GNS(000)</td>
<td>00100</td>
<td>0000001001</td>
</tr>
</tbody>
</table>
Waveforms Of Light-Traffic Controller Microcode

Instruction Sequencing

• With lot of inputs, size of memory increases rapidly (exponentially).
• Can reduce memory size by observing:
  – Most of the time we move to the next state.
  – We usually only need to branch to one other state based on (or a few) inputs.
• Add a microprogram counter (μPC) register to simplify state sequencing. (Think of the PC as the “current state” of the FSM.)
  • Next state is now μPC+1 unless we are branching
Instruction Sequencing – the picture

Branching Logic

- Branch logic selects between \( \mu PC + 1 \) and branch_target, depending on input and branch_instruction.
- Instructions are of the form branch if \( f(inputs) \)
  - For example branch if car_ew or branch if not car_ew

NOTE: The example has changed here.
Inputs are now car East/West and Left Turn for North/South.
Outputs are now: NS{gyr}LT{gyr}EW{gyr}
Logic: stay green East/West until a Left Turn input comes in.
Branch Microinstructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>000</td>
<td>No branch – always go to next instruction</td>
</tr>
<tr>
<td>br</td>
<td>100</td>
<td>Always branch</td>
</tr>
<tr>
<td>brlt</td>
<td>001</td>
<td>Branch when left-turn car is detected</td>
</tr>
<tr>
<td>brnlt</td>
<td>101</td>
<td>Branch if no left-turn car is detected</td>
</tr>
<tr>
<td>brew</td>
<td>010</td>
<td>Branch when east-west car is detected</td>
</tr>
<tr>
<td>brnew</td>
<td>110</td>
<td>Branch if no east-west car is detected</td>
</tr>
</tbody>
</table>

Microcode Of Traffic-Light Controller With Branches

<table>
<thead>
<tr>
<th>State</th>
<th>Addr</th>
<th>Inst</th>
<th>Target</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsg1</td>
<td>0000</td>
<td>brlt</td>
<td>lt1</td>
<td>10000101001</td>
</tr>
<tr>
<td>nsg2</td>
<td>0001</td>
<td>brnew</td>
<td>nsg1</td>
<td>100001001</td>
</tr>
<tr>
<td>ew1</td>
<td>0010</td>
<td>nop</td>
<td></td>
<td>010001001</td>
</tr>
<tr>
<td>ew2</td>
<td>0011</td>
<td>brew</td>
<td>ew2</td>
<td>0010011000</td>
</tr>
<tr>
<td>ew3</td>
<td>0100</td>
<td>br</td>
<td>nsg1</td>
<td>0011001010</td>
</tr>
<tr>
<td>lt1</td>
<td>0101</td>
<td>nop</td>
<td></td>
<td>010001001</td>
</tr>
<tr>
<td>lt2</td>
<td>0110</td>
<td>brlt</td>
<td>lt2</td>
<td>0011000001</td>
</tr>
<tr>
<td>lt3</td>
<td>0111</td>
<td>br</td>
<td>nsg1</td>
<td>0010010100</td>
</tr>
</tbody>
</table>
**Microinstruction Format**

<table>
<thead>
<tr>
<th>br inst</th>
<th>br target</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

For our example:

- \( b = 3 \)
- \( s = 4 \)
- \( o = 9 \)

---

**Implementing Microcode Of Light-Traffic Controller With Branches Using Verilog**

```verilog
module ucodeIS(clk, rst, in, out) ;
parameter n = 2 ; // input width
parameter m = 9 ; // output width
parameter k = 4 ; // bits of state
parameter j = 3 ; // bits of instruction
input clk, rst;
input [n-1:0] in ;
output [m-1:0] out ;
wire [k-1:0] nupc, upc ; // microprogram counter
wire [j+k+m-1:0] uinst ;   // microinstruction word
// split off fields of microinstruction
wire [m-1:0] nxt_out ; // = uinst[n-1:0] ;
wire [k-1:0] br_upc ; // = uinst[k-1:m] ;
wire [j-1:0] brinst ; // = uinst[m+j+k-1:m+k] ;
assign {brinst, br_upc, nxt_out} = uinst ;
DFF #(k) upc_reg(clk, nupc, upc) ; // microprogram counter
DFF #(m) out_reg(clk, nxt_out, out) ; // output register
ROM #(k,m+k) uc(upc, uinst) ; // microcode store
// branch instruction decode
// sequencer
assign nupc = rst ? {k{1'b0}} : branch ? br_upc : upc + 1'b1 ;
endmodule
```
Microcode Of Light-Traffic Controller With Left Turn

<table>
<thead>
<tr>
<th>Address</th>
<th>State</th>
<th>Br Inst</th>
<th>Target</th>
<th>NS LT EW</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NS1</td>
<td>BLT (001)</td>
<td>LT1(0101)</td>
<td>1000010001</td>
<td>00101011100001001</td>
</tr>
<tr>
<td>0001</td>
<td>NS2</td>
<td>BNEW (110)</td>
<td>NS1(0000)</td>
<td>1000010001</td>
<td>11000000100001001</td>
</tr>
<tr>
<td>0010</td>
<td>EW1</td>
<td>NOP (000)</td>
<td></td>
<td>0100010001</td>
<td>00000000001001001</td>
</tr>
<tr>
<td>0011</td>
<td>EW2</td>
<td>BEW (010)</td>
<td>EW2(0011)</td>
<td>0010011001</td>
<td>01000110001001001</td>
</tr>
<tr>
<td>0100</td>
<td>EW3</td>
<td>BR (100)</td>
<td>NS1(0000)</td>
<td>0010010101</td>
<td>10000000001001001</td>
</tr>
<tr>
<td>0101</td>
<td>LT1</td>
<td>NOP (000)</td>
<td></td>
<td>0100010001</td>
<td>00000000001001001</td>
</tr>
<tr>
<td>0110</td>
<td>LT2</td>
<td>BLT (001)</td>
<td>LT2(0110)</td>
<td>0011000001</td>
<td>001011000001001001</td>
</tr>
<tr>
<td>0111</td>
<td>LT3</td>
<td>BR (100)</td>
<td>NS1(0000)</td>
<td>0010101001</td>
<td>10000000001001001</td>
</tr>
</tbody>
</table>

Waveforms Of Light-Traffic Controller With Left Turn Microcode

```
00101011100001001
1100000100001001
000000010001001
0100111001101100
1000000100101001
000000010001001
0010111001100001
1000000001010001
```
Alternate Microcode For Light-Traffic Controller With Left Turn

<table>
<thead>
<tr>
<th>Address</th>
<th>State</th>
<th>Br Inst</th>
<th>Target</th>
<th>NS LT EW</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>NS1</td>
<td>BNA (111)</td>
<td>NS1(0000)</td>
<td>0000010001</td>
<td>110000010000010001</td>
</tr>
<tr>
<td>0001</td>
<td>NS2</td>
<td>BLT (001)</td>
<td>LT1(0100)</td>
<td>0100010001</td>
<td>001010001000010001</td>
</tr>
<tr>
<td>0010</td>
<td>EW1</td>
<td>BEW (010)</td>
<td>EW1(0010)</td>
<td>0010011000</td>
<td>0100010001001100</td>
</tr>
<tr>
<td>0011</td>
<td>EW2</td>
<td>BR (100)</td>
<td>NS1(0000)</td>
<td>0010010100</td>
<td>10000000001001010</td>
</tr>
<tr>
<td>0100</td>
<td>LT1</td>
<td>BLT (001)</td>
<td>LT1(0100)</td>
<td>0011000001</td>
<td>00101000011000010</td>
</tr>
<tr>
<td>0101</td>
<td>LT2</td>
<td>BR (100)</td>
<td>NS1(0000)</td>
<td>0010100010</td>
<td>10000000001010001</td>
</tr>
</tbody>
</table>

Previous microcode has:

- 2 states with GNS light (NS1 & NS2)
- 2 states with YNS light (EW1 & LT1)

By adding a new branch instruction – BNA:

- 1 state with GNS light (NS1)
- 1 state with YNS light (NS2)

Waveforms Of Alternate Microcode

```
1110000100001001
0010100010001001
0100010001001100
1000000001001010
0010100001100001
1000000001010001
```
Multiple Instruction Types

- For some FSMs the micro-instruction word can start getting a bit long.
- To shorten it, we observe:
  - Not every state needs a branch
  - Not every output changes on every state
- Define instruction that does just a branch or a load of one register:
  - \text{brx} 1\text{yyyyvvvv} - branch to value \text{vvvv} on condition \text{yyy}
  - \text{ldx} 0\text{yyyyvvvv} - load register \text{yyy} with value \text{vvvv}

- Note the tradeoff: \textbf{more instructions} (can only branch or change outputs on each instruction) but \textbf{each instruction is much smaller}.

Instruction Format Of Microcode With 2 Instruction Types

- Branch Instructions:
  - Operates branch mux as before
  - No write to output registers
- Load Instructions:
  - Sets branch mux to +1
  - Update selected output register:
    - Can include other datapath components
    - e.g., timer in place of an output register
**Block Diagram Of Microcode With Output Instructions**

**Multiple output registers:**
E.g., NS/EW/LT

**Microcode For Traffic-Light Controller With brx & ldx Instructions**

Registers to load:
- ldns  – load north/south light with value
- ldlt  – load left-turn light with value
- ldew  – load east/west light with value
- ltim1 – load timer 1 with value – starts timer
Microcode For Traffic-Light Controller With brx & ldx Instructions (Cont)

<table>
<thead>
<tr>
<th>State</th>
<th>Addr</th>
<th>Inst</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rst1</td>
<td>00000</td>
<td>ldlt</td>
<td>RED</td>
</tr>
<tr>
<td>rst2</td>
<td>00001</td>
<td>ldew</td>
<td>RED</td>
</tr>
<tr>
<td>ns1</td>
<td>00010</td>
<td>ldn</td>
<td>GREEN</td>
</tr>
<tr>
<td>ns2</td>
<td>00011</td>
<td>lim</td>
<td>T_GREEN</td>
</tr>
<tr>
<td>ns3</td>
<td>00100</td>
<td>bntz</td>
<td>ns4</td>
</tr>
<tr>
<td>ns4</td>
<td>00101</td>
<td>bntle</td>
<td>ns5</td>
</tr>
<tr>
<td>ns5</td>
<td>00111</td>
<td>lim</td>
<td>T_YELLOW</td>
</tr>
<tr>
<td>ns6</td>
<td>01000</td>
<td>bnt</td>
<td>ns7</td>
</tr>
<tr>
<td>ns7</td>
<td>01010</td>
<td>bit</td>
<td>lt1</td>
</tr>
<tr>
<td>ns8</td>
<td>01100</td>
<td>bntz</td>
<td>lt2</td>
</tr>
<tr>
<td>ns9</td>
<td>01110</td>
<td>ldew</td>
<td>GREEN</td>
</tr>
<tr>
<td>ew4</td>
<td>01111</td>
<td>lim</td>
<td>T_GREEN</td>
</tr>
<tr>
<td>ew5</td>
<td>01111</td>
<td>bntz</td>
<td>ew6</td>
</tr>
<tr>
<td>ew6</td>
<td>10000</td>
<td>ldew</td>
<td>YELLOW</td>
</tr>
<tr>
<td>ew7</td>
<td>10011</td>
<td>lim</td>
<td>T_YELLOW</td>
</tr>
<tr>
<td>ew8</td>
<td>10110</td>
<td>bntz</td>
<td>ew8</td>
</tr>
<tr>
<td>ew9</td>
<td>10111</td>
<td>ldew</td>
<td>RED</td>
</tr>
<tr>
<td>lt1</td>
<td>10100</td>
<td>br</td>
<td>ns1</td>
</tr>
<tr>
<td>lt2</td>
<td>10101</td>
<td>lim</td>
<td>T_RED</td>
</tr>
<tr>
<td>lt3</td>
<td>10110</td>
<td>bntz</td>
<td>lt3</td>
</tr>
<tr>
<td>lt4</td>
<td>11000</td>
<td>lim</td>
<td>T_GREEN</td>
</tr>
<tr>
<td>lt5</td>
<td>11011</td>
<td>bntz</td>
<td>lt5</td>
</tr>
<tr>
<td>lt6</td>
<td>11110</td>
<td>ldid</td>
<td>GREEN</td>
</tr>
<tr>
<td>lt7</td>
<td>11111</td>
<td>lim</td>
<td>T_YELLOW</td>
</tr>
<tr>
<td>lt8</td>
<td>11100</td>
<td>bntz</td>
<td>lt8</td>
</tr>
<tr>
<td>lt9</td>
<td>11111</td>
<td>ldid</td>
<td>RED</td>
</tr>
<tr>
<td>lt10</td>
<td>11110</td>
<td>br</td>
<td>ns1</td>
</tr>
</tbody>
</table>

Implementing Traffic-Light Controller Microcode With brx & ldx Instructions In Verilog

```verilog
// Module ucodeMI
module ucodeMI(clk, rst, in, out) ;
parameter n = 2 ; // input width
parameter m = 9 ; // output width
parameter o = 3 ; // output sub-width
parameter k = 5 ; // bits of state
parameter j = 4 ; // bits of instruction
input clk, rst ;
input [n-1:0] in ;
output [m-1:0] out ;

wire [k-1:0] nupc, upc ; // microprogram counter
wire [j+k-1:0] uinst ; // microinstruction word
wire done ; // timer done signal

// Split off fields of microinstruction
wire opcode ; // opcode bit
wire [j-2:0] inst ; // condition for branch, dest for store
wire [k-1:0] value ; // target for branch, value for store
assign (opcode, inst, value) = uinst ;
```

To be continued on next page...
Implementing Traffic-Light Controller Microcode With brx & ldx Instructions In Verilog (Cont)

DFF #(k) upc_reg(clk, nupc, upc) ; // microprogram counter
ROM #(k,k+j) uc(upc, uint) ; // microcode store

// output registers and timer
DFFE #(o) or0(clk, e[0], value[0:1], out[0:1]) ; // NS
DFFE #(o) or1(clk, e[1], value[0:1], out[2*0:1]) ; // EW
DFFE #(o) or2(clk, e[2], value[0:1], out[3*0:2*0]) ; // LT
Timer #(k) tim(clk, rst, e[3], value, done) ; // timer

// enable for output registers and timer
wire [3:0] e = opcode ? 4'b0 : 1<inst ;

// branch instruction decode
wire branch = opcode ? (inst[2] ^ ((inst[1:0] == 0) & in[0]) | // BLT
   ((inst[1:0] == 1) & in[1]) | // BEW
   ((inst[1:0] == 2) & (in[0]|in[1])) | // BLE
   ((inst[1:0] == 3) & done)) // BTD
                  : 1'b0 ; // for a store opcode

// microprogram counter
assign nupc = rst ? {k{1'b0}} : branch ? value : upc + 1'b1 ;
endmodule

Summary

• Microcode is just FSM implemented with a ROM or RAM
  – One address for each state x input combination
  – Address contains next state and output
• Adding a sequencer reduces size of ROM/RAM
  – One entry per state rather than 2^inputs
  – μPC, incrementer, branch address, and branch control
• Adding instruction types reduces width of ROM/RAM
  – Branch or output in each instruction – rather than both
  – Type field specifies which one
  – Need more instructions, but each is smaller
• One step away from a full processor
  – Just add more instructions (defer until EE108B)