Day 11

Programmable Logic
Programmable Logic

• Broad topic, will only scratch the surface

• Lab this week will give you a first look at the possibilities

• You will need to learn more to make use of this in your project

• Will look at 3 types of programmable logic: EEPROMs, PALs, and FPGAs

• Will not look at µProcessors like 16-series PICs (used to do this, have left the information on the web site) or more capable processors like Arduinos
Why?

- Implementing combinatoric or sequential logic in discrete components is painful.
- Function fixed in hardware design.
- Programmable devices are more compact, and in principle can be reprogrammed “firmware update.”
- FPGAs (and to a lesser extent CPLDs) are the state of the art here.
Lab Notes

• The lab demonstrates 3 programmable logic devices EEPROM, PAL, FPGA

• The lab is too long, so do (EEPROM or PAL) + FPGA

• I am not convinced the PAL programming works yet, although I will figure this out tonight

• We have a limited number of programmers, so work in pairs and be a little patient…
Memory as Logic

- Memory chips can implement arbitrary combinatoric logic
- Address lines pick a single memory location
- Data output provides values stored at that address
- Each bit in the output is a function of input lines
- Chip Enable (usually neg. true) must be asserted to get output, sometimes also an Output Enable line
- Memory is usually not synchronous, although CE or OE can be used in that role

Read-only Memory
1k x 8 = 8kbits shown
ROMs through history

• **ROM** - *read-only memory*, was not programmable, memory values hardcoded (think Atari 2600 cartridges)

• **PROM** - *programmable*, burn fuses to store data, one-time

• **EPROM** - *erasable*, use UV light through a window to erase

• **EEPROM** - *electrically erasable*, data stored in FET-style gates, change values with large (~12V) programming voltage

• All of these are *non-volatile* - memory stays even when power is switched off

• Flash RAM has pretty much taken over, serial (rather than parallel) interface, difficult to use without microprocessor
Classic EEPROM

- AT28C16 EEPROM (2k x 8 = 16k bits)
- 3 control lines, 11 address lines, 8 data lines
- Writing involves special voltages (what the chip programmer does)
- ‘Burn’ pattern on programmer, read data in circuit

Features
- Fast Read Access Time - 150 ns
- Fast Byte Write - 200 µs or 1 ms
- Self-Timed Byte Write Cycle
- Direct Microprocessor Control - DATA POLLING
- Low Power - 30 mA Active Current, 100 µA CMOS Standby Current
- High Reliability - Endurance: 10^4 or 10^5 Cycles, Data Retention: 10 Years
- 5V ±10% Supply
- CMOS & TTL Compatible Inputs and Outputs
- JEDEC Approved Byte Wide Pinout
- Commercial and Industrial Temperature Ranges

Description
The AT28C16 is a low-power, high-performance Electrically Erasable and Programmable Read Only Memory with easy to use features. The AT28C16 is a 16K memory organized as 2,048 words by 8 bits. The device is manufactured with Atmel’s reliable nonvolatile CMOS technology.

PDIP, SOIC
Top View

Table 1: Truth table to light up 4 LEDs in order. Note the address and data values are written in hex.
EEPROM programming

Select chip
Download data
LAB circuit

- EEPROM used as decoder
- 4-bit address from clock, select 16 memory locations
- Drive transitions with debounced button
- Store pattern to show on LEDs in EEPROM memory
EEPROM Summary

• Advantages
  • All logic equally easy/hard
  • Can naturally implement 8 bits at a time
  • Very complex logic (11 or more inputs) feasible

• Disadvantages
  • Chips are huge
  • Must remove chip to program
  • Serial ROMs more common (one address + clock line)
PAL overview

- Programmable Array Logic
- Originally ‘write-once’ - burned fuses to make connections
- More modern devices programmed like EEPROMS
- Array connections define logic through product terms
Classic PAL

- ATMEL ATF 16v8 - 8 inputs + 8 prog. I/O
- Up to 16 inputs*
- Only 8 unique product terms
- Outputs can OR several product terms
- 3:8 decoder would just fit, better for complicated functions with lots of inputs, few outputs (only 8 product terms!)
- 20v10 about the largest made in DIP package
CUPL

- One common language to program PAL chips
- .pld file converted into ‘fuse map’ that gets downloaded to array to implement logic
- Derived from VLSI tools for ASIC design
CUPL Example

- Logic to convert binary to Gray code
- File starts with ‘header’, must be in .pld file
- C-style comments

```c
/*
 * University of Oregon PHYS432
 * This program converts a 4-bit binary word to Gray's code.
 */
Name        ledenc;
Partno      PHYS432;
Revision    01;
Date        20/03/2006;
Designer    E. Torrence;
Company     University of Oregon;
Location    None;
Assembly    None;
Device      g16v8;
```
CUPL Example

• Define pins, input and output are defined by function, names are irrelevant, [2..5] is shorthand for [2, 3, 4, 5]

• Define logic as out = f(in)

• Standard boolean operations (here XOR, eats up 7 of 8 available logic blocks)

```c
/** Inputs **/
pin [2..5] = [in0..3]; /* 4-bit input lines */

/** Outputs **/
pin [19..16] = [out0..3]; /* 4-bit output lines */

/* Define Gray's code conversion as out(i) = in(i+1) XOR in(i) */
out3 = in3;
out2 = in3 $ in2;
out1 = in2 $ in1;
out0 = in1 $ in0;
```
CUPL Commands

- Not really C-like command syntax

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>!A</td>
<td>NOT</td>
</tr>
<tr>
<td>&amp;</td>
<td>A &amp; B</td>
<td>AND</td>
</tr>
<tr>
<td>#</td>
<td>A # B</td>
<td>OR</td>
</tr>
<tr>
<td>$</td>
<td>A $ B</td>
<td>XOR</td>
</tr>
</tbody>
</table>

- Possible to make compound combinations
  (2:4 neg. true decoder, only 4 logic cells)

```plaintext
out0 = !(in1 & !in0);
out1 = !(in1 & in0);
out2 = !(in1 & !in0);
out3 = !(in1 & in0);
```
Registered PALs

- Add D-type flip-flops to all outputs
- Special syntax for making counters internally
- Clock must be input on Pin 1, output-enable for expansion

```c
/* Define the state machine values */
field counter = [q1..0]; /* Define "count" as the two counter bits */
$define s0 'b'00 /* Define the values of the counter bits in */
$define s1 'b'01 /* each of the four counter states */
$define s2 'b'10
$define s3 'b'11

sequence counter { /* Describe the action of the state machine */
    present s0 next s1;
    present s1 next s2;
    present s2 next s3;
    present s3 next s4;
}
```
Registered PALs

• More involved example including inputs and outputs

• Software figures out logic, just specify states, transitions, and outputs

```
sequence counter {
  present s0
    if up & enabled next s1;
    if down & enabled next s2;
    default next s0 OUT stuck;
  present s1
    if up & enabled next s2;
    if down & enabled next s0;
    default next s1 OUT stuck;
  present s2
    if up & enabled next s0;
    if down & enabled next s1;
    default next s2 OUT stuck;
  present s3
    next s0;
}
```
PAL Summary

- Advantages
  - Simple syntax for implementing logic
  - Registered PALs allow (simple) state machine on one chip

- Disadvantages
  - Limited resource for implementing logic in DIP package, quickly run out of product terms
  - Chips are obsolete, getting hard to find, tools questionable for future, CPLDs are more common now
  - Must remove chip to program
  - Syntax a bit wonky at times
FPGAs

- Field-Programmable Gate Arrays

- Evolution of PALs - CPLDs - FPGAs

- Dramatic increase in available logic and other functionality, Xilinx chips can have ~5M logic blocks

- Not a processor, still implementing boolean logic (although can build processors internally in FPGAs)

- Usually complicated (and expensive) software to program
FPGA Logic Cell

Similar to one PAL...
Simple FPGA Architecture

Lattice iCE40 LP8k has 7680 LUTs, or 960 PLBs
Interconnect ‘fabric’ routes in/out signals between PLBs, I/O banks
Typical FPGA chip

Ball Grid Array - BGA, impossible for non-pros to deal with
Usually lots and lots of pins
Must buy these on boards to use effectively
High-end FPGAs can cost $2k/chip
TinyFPGA BX board

- Lattice iCE40 LP8k (3.3V CMOS)
- 28-pin DIP package
- Program over USB using free command-line tools (Win, OS X, Linux)
- 16 MHz clock
- 24 I/O pins easy to get to
- Relatively cheap ($40)
Programming FPGAs

- Hardware Description Language (HDL) - Verilog and VHDL

- Also higher-level synthesis tools, python translators, graphical programming, …

- We will look at Verilog, supported by TinyFPGA, fairly easy to read

- Similar (but more capable) than CUPL, describes logic, HDL is not a procedural program
Multiplexer in Verilog

// 4-1 multiplexer
module mux(
    input [1:0] select, // 2 bits for select
    input [3:0] data,  // 4 inputs
    output q
);

// One big logic function
assign q = (~select[0] & ~select[1] & data[0])
    | ( select[0] & ~select[1] & data[1])
    | (~select[0] & select[1] & data[2])
    | ( select[0] & select[1] & data[3]);
endmodule

Modules and execution blocks key to understanding Verilog
Direct mapping to concept of a ‘device’

Syntax is a little nicer than CUPL, but combinatoric logic similar
// 4-1 multiplexer
module mux(
    input [1:0] select, // 2 bits for select
    input [3:0] data,  // 4 inputs
    output q
);

// Array Logic
Assign q = d[select];
endmodule

Looks like C or python indexing, creates same logic as before, but meaning is more clear
Multiplexer III

// 4-1 multiplexer
module mux(
    input [1:0] select, // 2 bits for select
    input [3:0] data,  // 4 inputs
    output q
);

// Array Logic
always @ (d or select)
    q = d[select];

endmodule

Defines a block, this block executes whenever d or select change
q = ... is a blocking assignment (will happen before other assignments in the block)
// 4-1 multiplexer
module mux(
    input clk,
    input [1:0] select, // 2 bits for select
    input [3:0] data,  // 4 inputs
    output q
);

// Array Logic
always @(posedge clk)
    q <= d[select];

endmodule

Fully synchronous MUX, block executes on rising edge of clock
q <= ... is a non-blocking assignment, multiple of these
will happen in parallel, assignment made synchronously

always @(posedge clk) begin
    a <= b;
    b <= a;
end
// 4-1 multiplexer
module mux(
    input [1:0] select,  // 2 bits for select
    input [3:0] data,    // 4 inputs
    output q
);

// Conditional Logic
always @(select or data)
begin
    if( select == 0 )
        q = d[0];

    if( select == 1 )
        q = d[1];

    if( select == 2 )
        q = d[2];

    if( select == 3 )
        q = d[3];
end
endmodule

begin-end to indicate range of always block
C-like conditional logic

Note: this is not a program, will be synthesized into boolean logic that runs concurrently (in parallel)

Possibly even easier to understand
Counter Example

// 8-bit counter with async. reset and overflow
module count8(
    input clock,
    input reset,
    output [7:0] count,
    output overflow
);

// Be clear these are registers
reg [7:0] count;
reg overflow;

always @ (posedge clock or posedge reset) begin
    if (reset) // check if reset is HI
        count <= 8'h00; // Set to zero

    else begin
        count <= count + 1;
        // Overflow is set when counter is resetting to zero
        overflow <= (count == 8'hFF);
    end
end
endmodule

Can do simple math, comparisons and assignments with constants, must remember this gets translated to logic, this is not a program
module top (  
    input CLK, // 16MHz clock  
    output LED // User/boot LED next to power LED  
);  
    // Slow counter which takes 16MHz clock and  
    // divides it down to a 1s output clock.  
    secondcounter cntsec(CLK, LED);  
endmodule

Looks like any other module  
This represents the interface to the ‘outside world’  
Can input/output system clock, pins, and on-board LED  
Names defined in pins.pcf file

# Pin assignments  
set_io --warn-no-port PIN_1 A2  
set_io --warn-no-port -pullup yes PIN_2 A1  

# LED  
set_io --warn-no-port LED B3  

# 16MHz clock  
set_io --warn-no-port CLK B2 # input  

Assigns keywords (meaningful for BX) to actual FPGA pins (A1, A2, ...
module secondcounter (  
    input clock,  
    output second  
);  

reg second;  

// Want to produce output that toggles every 0.5 s  
// That means counting to 8M at 16 MHz, which needs 24 bits  
parameter countstart = 24'h7A1200;  

reg [23:0] count24;  

// Initialize to 8M  
initial begin  
    count24 <= countstart;  
end  

// Decrement on every clock, reset when it gets to zero  
always @ (posedge clock) begin  
    if (count24 == 24'h000000) begin  
        count24 <= countstart; // Reset to 8M  
        second <= !second; // Toggle output  
    end  
    else  
        count24 = count24 - 1;  
end  

endmodule
FPGA Summary

• Advantages
  • Lots of resource for implementing complicated logic
  • Verilog allows sophisticated programming constructs
  • Can have many blocks running in parallel - potentially very fast
  • Usually can re-program in place (firmware update)

• Disadvantages
  • Usually difficult to deal with manufacturing and programming
  • Each FPGA family has its own toolchain, features, development tools
  • Large FPGAs can be very expensive
  • Debugging can be a challenge