1. DSP project info

2. Analog-to-digital converters

3. Digital-to-analog converters

4. 2-dimensional register memory
Reminder: Project proposal is due Friday, October 26, 2007 by 5pm.

Modification of technical specs of lock-in amplifier:
Phase control: 0 to 360° with a resolution of at least 3.6°.
You must be able to shift the phase by exactly 90° from any phase position.

Basic guideline for project proposal (also see Project Guidelines document):
You must convince your reader that your design concept will work well enough that it deserves to be funded. This means that your design has reached the point that you can draw up a budget. Your proposal must include a budget which is as specific as possible and an expected timeline.
As physicists, we know that:

- We live in an **analog** world of continuously varying signals.
- Almost all physical quantities (observables) are continuous in nature.

As electronics designers, we know that:

- Digital electronics is very **powerful**, **cheap**, and relatively **easy** to design (at least compared to analog circuits).
- Digital electronics only works with **digital signals**.

**THEREFORE …**

If we’re ever going to make anything useful, we need to a way to convert (or approximate) an ANALOG signal to (by) a sequence of digital-binary numbers, and vice versa.
Algorithm: At each clock cycle, round your analog voltage to the nearest digital value.

→ The size of a digital step is defined by $V_{\text{ref}}/2^n$ for an n-bit converter.
**Shannon-Nyquist Sampling Theorem**

**THEOREM:**

A continuous-time finite bandwidth signal can be exactly reconstructed from its samples if the sampling frequency is greater than 2 times the signal bandwidth $B$, where $B$ is the largest (non-zero) frequency component of the signal.

$F=2B$ is referred to as the Nyquist frequency (the lowest possible sampling frequency).

**Practical Considerations:**

- Any finite duration signal has $B \to +\infty$, so exact mathematical application of the theorem is impossible.

- The theorem indicates the frequency scale that one should use in order to usefully sample a signal → *always use a sampling rate which is greater than twice the highest frequency component of “reasonable amplitude”.*
The fastest (and most expensive) n-bit ADCs use $2^{n-1}$ comparators to determine which of the $2^n$ numbers the analog input is closest to.

8-bit ADC: 255 comparators.

12-bit ADC: 4096 comparators

In general, higher the bit resolution results in a slower ADC (and more expensive).

Most digital oscilloscopes use an 8-bit ADC.
In order to keep the number of comparators small, it holds the input voltage, and converts it in steps:

→ Converts the upper four bits with a 4-bit ADC.
→ Converts the digitized value back into an analog value with a Digital-to-Analog Converter (DAC).
→ Subtracts this from the input to generate the smaller, difference voltage.
→ Finally, it uses a 2nd 4-bit ADC to convert the lower 4-bits.

The entire process takes less than 800 ns when operating off the internal timing of the ADC0820 (RD mode).

→ You can run it somewhat faster with some clever timing (WR-RD) mode but we will opt for simplicity.
ADC0820: Half-Flash ADC (II)

[figure from the National Semiconductor ADC0820 datasheet]
### ADC0820: Half-Flash ADC (III)

#### Functional Table

<table>
<thead>
<tr>
<th>Pin</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VIN</td>
<td>Analog input; range GND ≤ VIN ≤ VCC</td>
</tr>
<tr>
<td>2-5</td>
<td>DB0-DB3</td>
<td>TRI-STATE data outputs; bit 0 (LSB) to bit 3</td>
</tr>
</tbody>
</table>
| 6   | WR / RDY  | **WR-RD Mode - WR**: With CS low, the conversion is started on the falling edge of WR.  
**RD Mode - RDY**: RDY will go low after the falling edge of CS; RDY will go TRI-STATE when the result of the conversion is strobed into the output latch. |
| 7   | MODE      | Select mode: LOW = RD Mode HIGH = WR-RD Mode                              |
| 8   | RD        | **WR-RD Mode**  
With CS low, the TRI-STATE data outputs (DB0-DB7) will be activated when RD goes low.  
**RD Mode**  
With CS low, the conversion will start with RD going LOW; also RD will enable the TRI-STATE data outputs at the completion of the conversion. RDY going TRI-STATE and INT going low indicates the completion of the conversion. |
| 9   | INT       | INT going LOW indicates that the conversion is completed and the data result is in the output latch. INT is reset by rising edge on RD or CS. |
| 11  | V_{REF(-)} | Bottom of resistor ladder; range: GND ≤ V_{REF(-)} ≤ V_{REF(+)}}           |
| 12  | V_{REF(+)} | Top of resistor ladder; range: V_{REF(-)} ≤ V_{REF(+)} ≤ V_{CC}           |
| 13  | CS        | CS must be low for the RD or WR to be recognized.                        |
| 14-17 | DB4-7    | TRI-STATE data output—bits 4-7                                         |
| 18  | OFL       | Overflow—If the analog input is higher than the V_{REF(+)}, OFL will be LOW at the end of conversion. Can be used to cascade. |
In RD mode, the RD# line going LOW initiates the conversion.

When the conversion is complete, the INT# line goes LOW & the data is latched into output buffers.

The output buffers will be put in a Z state when RDY# goes LOW until the INT# line goes LOW.
In RD mode, the RD# line going LOW initiates the conversion.

When the conversion is complete, the INT# line goes LOW & the data is latched into output buffers.

The output buffers will be put in a Z state when WR# goes LOW until the INT# line goes LOW.

[figure from the National Semiconductor ADC0820 datasheet]
A Digital-to-Analog Converter (DAC) is used to convert a digital signal into an analog voltage.

**A DAC is useful for:**

- Generating a voltage from a computer, microprocessor, or FPGA that will then control part of an experiment.
- Producing a digitally synthesized waveform (triangle, sine, or more complex).
- Converting digital music to sound, etc …

  (the CD standard is 16-bits at 44.1 KHz)

DACs are generally much faster than ADCs for a same bit resolution.
R-2R resistor ladder (I)

- It’s easy to make the DAC output voltages.

- Look from the right-hand side
  - 2 parallel resistors
  - Each with a value of $2R$
It’s easy to make the DAC output voltages.

Look from the right-hand side
- 2 parallel resistors
- Each with a value of $2R$
Continuing farther to the left, we find that the effective resistance to ground is $R$ at every dot on the top line.
The ladder acts like a series of voltage dividers that reduces the voltage by an additional factor of 2 at each \( R-2R \) junction.

\( V \) decreases by half at each connection point along the top rail.

Thus each output voltage is related to the input voltage by a power of two.
We can generate an analog voltage by adding together the voltages represented by the various stages in the ladder.

If we sum the ladder outputs based on a simple a binary representation in switches then we have a DAC.

[figure from the Art of Electronics (2nd edition, 1999) by P. Horowitz and W. Hill, p. 616]
We can generate an analog voltage by adding together the voltages represented by the various stages in the ladder.

If we sum the ladder outputs based on a simple binary representation in switches then we have a DAC.
We can generate an analog voltage by adding together the voltages represented by the various stages in the ladder.

If we sum the ladder outputs based on a simple a binary representation in switches then we have a DAC.

[figure from the Art of Electronics (2nd edition, 1999) by P. Horowitz and W. Hill, p. 616]
A real DAC: the TLC7524

[figure from Texas Instruments TLC7524 datasheet]
The data on the digital inputs is sent to the analog output when WR goes LOW.

When WR is HIGH, the digital inputs and analog output remain latched to their present values.

[figure from Texas Instruments TLC7524 datasheet]
ADC $\rightarrow$ DAC
Analog-to-Analog

Diagram:
- ADC
- DAC
- 8 Bits
- Out
2-dimensional registers with Quartus II

```verilog
// This module produces a sequence of 8-bit binary numbers
// with a period of 1 second (based on a 50.0 MHz clock).
module TwoD_Register(clock, output_signal);

input clock;  // input clock at 50 MHz
output reg [7:0] output_signal;  // output 8-bit binary number sequence

reg [25:0] frequency_counter;  // frequency counter used to convert 50 MHz clock to 1 Hz clock
reg [5:0] i;  // 5-bit (integers: 0-63) counter/register for addressing memory

// Simultaneously Declare and Load the data file "data" into 2-d register memory on FPGA
// This "(* ... *)" code is compiler specific to Quartus II (it is not universal Verilog).
// The 2-d register memory consists of 30 words which are 8-bits long each.
(* ram_init_file = "File_with_Data.mif" *) reg [7:0] Register_2D [0:29];

// initialize 1-d registers
initial
begin
  i = 5'b00000;
frequency_counter = 26'b00000000000000000000000000000000;
end
```
2-dimensional registers with Quartus II

```verilog
module TwoD_Register(clock, output_signal);
    input clock;                      // input clock at 50 MHz
    output reg [7:0] output_signal;  // output 8-bit binary number sequence

    reg [25:0] frequency_counter;    // frequency counter used to convert 50 MHz clock to 1 Hz clock
    reg [5:0] i;                     // 5-bit (integers: 0-31) counter/registers for addressing memory

    // Simultaneously Declare and Load the data file "data" into 2-d register memory on FPGA
    // This "(* ... *)" code is compiler specific to Quartus II (it is not universal Verilog).
    // The 2-d register memory consists of 30 words which are 8-bits long each.
    (* ram_init_file = "File_with_Data.mif" *) reg [7:0] Register_2D [0:29];

    // initialize 1-d registers
    initial
        begin
            i = 5'b00000;
            frequency_counter = 26'b00000000000000000000000000000000;
        end
```

Declare 2-d register & Initialize with data in file.
Using a 2-d Register in Verilog

```verilog
// This 50 MHz clock counter counts up to 50e6 in 1 seconds and then resets
// and increases in the "i" counter by 1 ("i" clock is effectively 1 Hz).
always@(posedge clock)
begin

  // increase 50 MHz frequency_counter by 1.
  frequency_counter <= frequency_counter + 1;

  // if "frequency_counter" is equal to 50 million,
  // then reset frequency_counter to ZERO
  // and send the 8-bit word corresponding to Register_2D[i] to the output
  // and increase the "i" counter by ONE.
  if (frequency_counter == 26'b1011110101111000010000000)
    begin
      frequency_counter <= 26'b00000000000000000000000000;
      output_signal <= Register_2D[i];
      i <= i + 5'b00001;
      if (i == 5'b10000) i <= 5'b00000;  // reset "i" counter to ZERO if it becomes 32.
    end

end
```

endmodule
Using a 2-d Register in Verilog

```verilog
// This 50 MHz clock counter counts up to 50e6 in 1 seconds and then resets
// and increases in the "i" counter by 1 ("i" clock is effectively 1 Hz).
always@(posedge clock)
begin

    // increase 50 MHz frequency_counter by 1.
    frequency_counter <= frequency_counter + 1;

    // if "frequency_counter" is equal to 50 million,
    // then reset frequency_counter to ZERO
    // and send the 8-bit word corresponding to Register_2D[i] to the output
    // and increase the "i" counter by ONE.
    if (frequency_counter == 26'b101111101011110000100000000)
        begin
            frequency_counter <= 26'b00000000000000000000000000;
            output_signal <= Register_2D[i];
            i <= i + 3'b000;
            if (i == 5'b10000) i <= 5'b00000; // reset "i" counter to ZERO if it becomes 32.
        end
endmodule
```

Reads the 8-bit word at address “i” of “Register_2D” & sends it to the 8-bit 1-d register “output_signal”.
Generating the **Memory Initialization File** (1)

You can write the file yourself or …

```plaintext
FILE
WIDTH=8;
DEPTH=30;
ADDRESS_RADIX=UNS;
DATA_RADIX=BIN;
CONTENT
BEGIN
  0 : 00000001;
  1 : 00000010;
  2 : 00000100;
  3 : 00001000;
  4 : 00010000;
  5 : 01000000;
  6 : 10000000;
  7 : 10000000;
  8 : 11000000;
  9 : 10100000;
 10 : 10010000;
 11 : 10001000;
 12 : 10000100;
 13 : 10000010;
 14 : 10000001;
 15 : 10000001;
 16 : 10000001;
 17 : 10001001;
 18 : 10010001;
 19 : 10100001;
 20 : 11000001;
 21 : 11100001;
 22 : 11110001;
 23 : 11010001;
 24 : 11000001;
 25 : 11000001;
 26 : 11000001;
 27 : 11000001;
 28 : 11010001;
 29 : 11100001;
END;
```
You can use Quartus II to generate the file using the Memory Editor:

Select: New File
→ Other Files
→ Memory Initialization File
Memory Initialization File Editor

<table>
<thead>
<tr>
<th>Addr</th>
<th>+0</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
<th>+4</th>
<th>+5</th>
<th>+6</th>
<th>+7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000001</td>
<td>00000010</td>
<td>00001000</td>
<td>00010000</td>
<td>00100000</td>
<td>01000000</td>
<td>10000000</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>11000000</td>
<td>10100000</td>
<td>10010000</td>
<td>10001000</td>
<td>10000100</td>
<td>10000010</td>
<td>10000001</td>
<td>10000011</td>
</tr>
<tr>
<td>16</td>
<td>10001011</td>
<td>10001001</td>
<td>10010001</td>
<td>10100001</td>
<td>11000001</td>
<td>11100001</td>
<td>11010001</td>
<td>11001001</td>
</tr>
<tr>
<td>24</td>
<td>11000101</td>
<td>11000011</td>
<td>11000111</td>
<td>11001011</td>
<td>[11100011]</td>
<td>[11100011]</td>
<td>[11100011]</td>
<td>[11100011]</td>
</tr>
</tbody>
</table>

**Right-click** to enter type memory information: integer, hexadecimal, binary, etc…

- **Word memory address**
- **8-bit word** (enter by hand)
Logic Element or RAM memory?

If your memory space is not very large, the compiler will automatically choose to implement your Verilog circuit with D-type flip-flops of the Logic Elements. You can force the compiler to use the dedicated FPGA memory:

Assignments > Settings >
Logic Element or RAM memory?

If your memory space is not very large, the compiler will automatically choose to implement your Verilog circuit with D-type flip-flops of the Logic Elements. You can force the compiler to use the dedicated FPGA memory:

Assignments > Settings >

\{ turn ON \}
Check compiler report for memory usage:

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow Status</td>
<td>Successful - Mon Oct 22 03:03:49 2007</td>
</tr>
<tr>
<td>Quartus II Version</td>
<td>7.1 Build 156 04/30/2007 SJ Web Edition</td>
</tr>
<tr>
<td>Revision Name</td>
<td>TwoD_Register</td>
</tr>
<tr>
<td>Top-level Entity Name</td>
<td>TwoD_Register</td>
</tr>
<tr>
<td>Family</td>
<td>Cyclone II</td>
</tr>
<tr>
<td>Device</td>
<td>EP2C35F672C6</td>
</tr>
<tr>
<td>Timing Models</td>
<td>Final</td>
</tr>
<tr>
<td>Met timing requirements</td>
<td>N/A</td>
</tr>
<tr>
<td>Total logic elements</td>
<td>55</td>
</tr>
<tr>
<td>Total combinational functions</td>
<td>55</td>
</tr>
<tr>
<td>Dedicated logic registers</td>
<td>31</td>
</tr>
<tr>
<td>Total registers</td>
<td>31</td>
</tr>
<tr>
<td>Total pins</td>
<td>9</td>
</tr>
<tr>
<td>Total virtual pins</td>
<td>0</td>
</tr>
<tr>
<td>Total memory bits</td>
<td>256</td>
</tr>
<tr>
<td>Embedded Multiplier 9-bit elements</td>
<td>0</td>
</tr>
<tr>
<td>Total PLLs</td>
<td>0</td>
</tr>
</tbody>
</table>