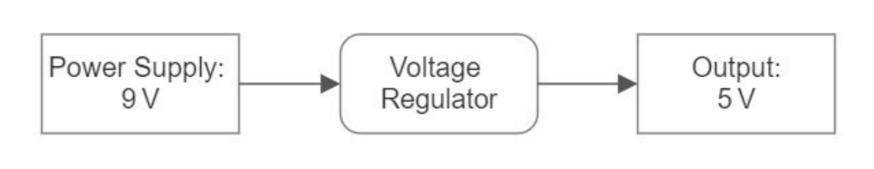
Metal Detection Project
Team 3

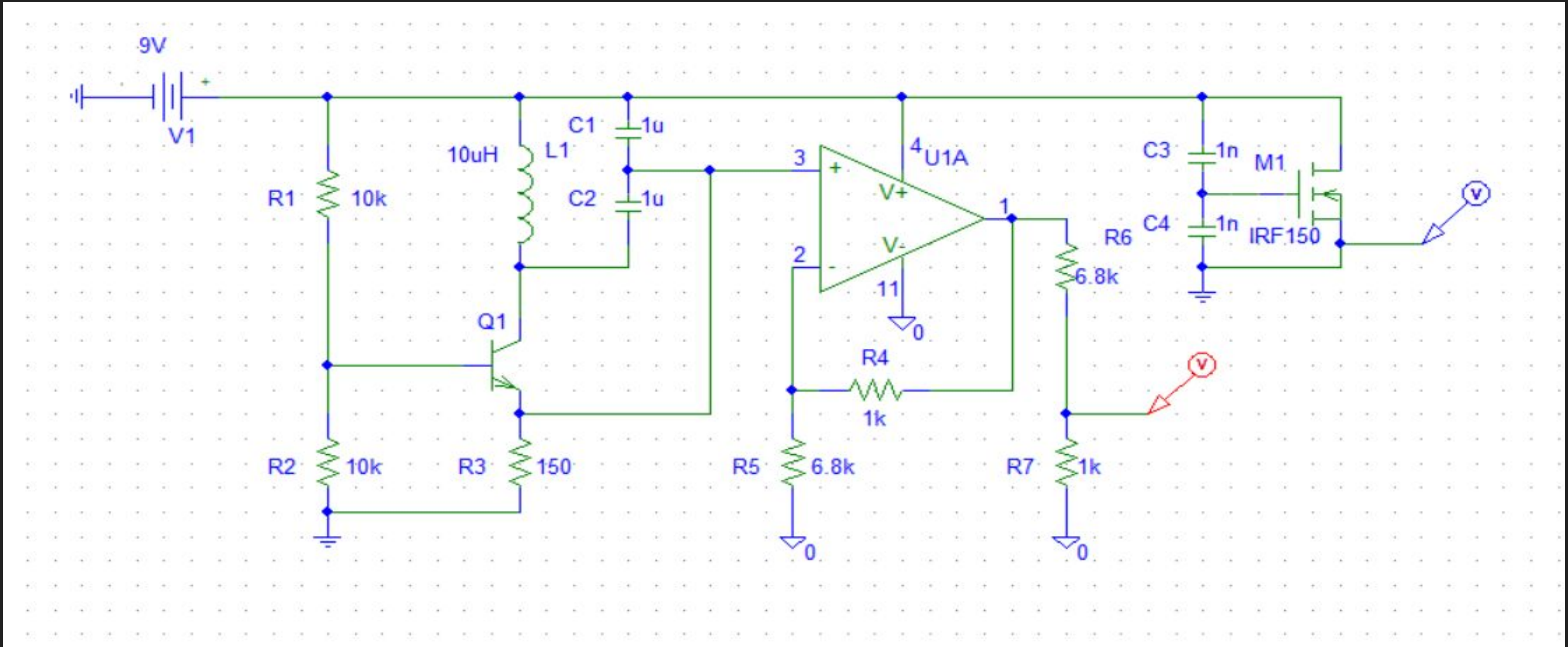
Group Members: Stephen Bauer, Jack Bonfiglio, Venkateswararao Gutta, Bartosz Obszynski

Circuit Design Breakdown

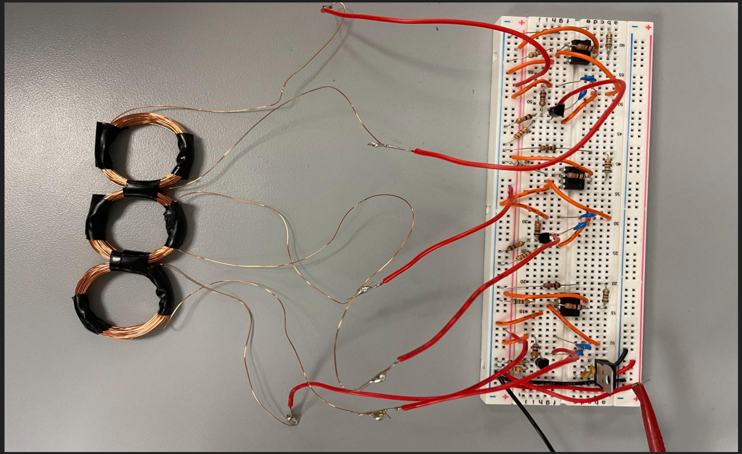


- 9 V feeds into the Colpitts Oscillator which then generates a sinusoidal wave signal
- Op Amp acts as a buffer for the signal
- Then signal voltage is lowered in order for it to be processed by the Basys 3 Board
- The result is the output signal which is used for further implementation
- A voltage regulator is used to power the Basys 3 Board

Circuit Diagram



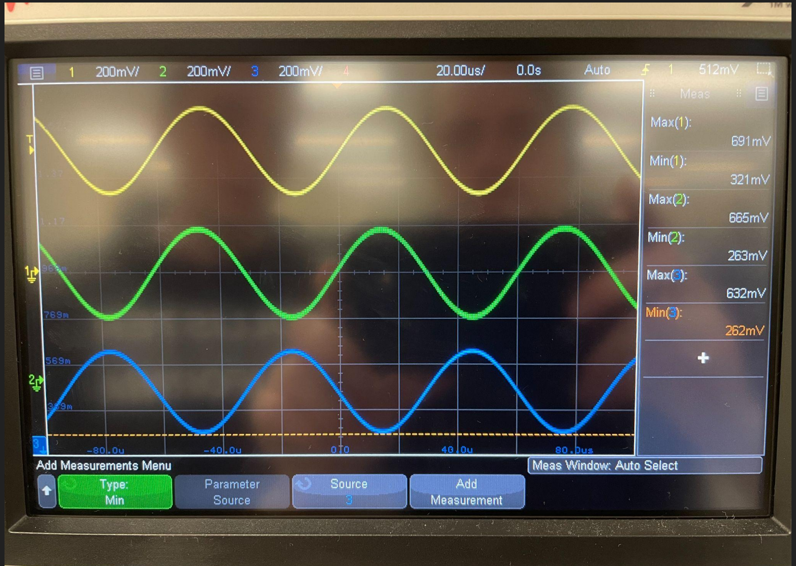
Circuit Explanation

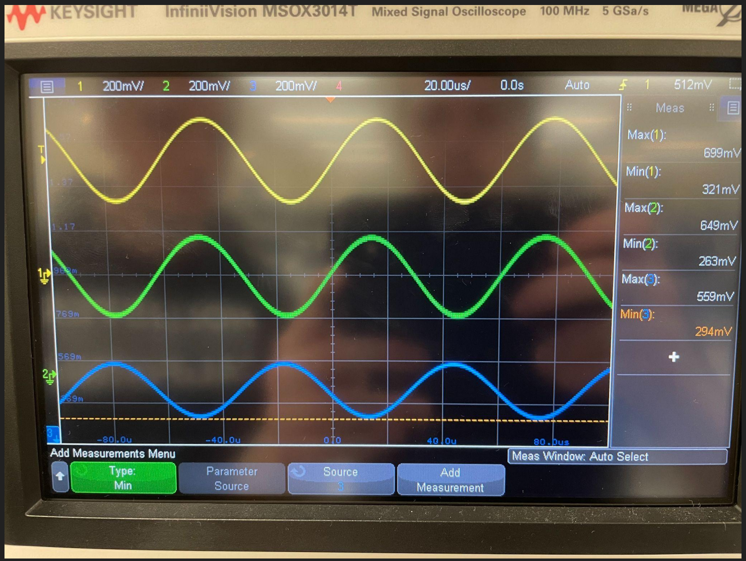


- Implemented a colpitts oscillator to turn the DC voltage into AC.
- Op amp is used to buffer the input voltage so that the impedance is not too high for the output voltage.
- Voltage divider is incorporated to drop down the voltage to below 1V in order to work on the Basys board
- The voltage regulator regulates the voltage to 5V to power the Basys board
- This process was repeated three times for each inductor coil
- The coils generate an electromagnetic field, this field is disrupted when a suitable metal is near. The voltage over the coil will drop thus metal is able to be detected.

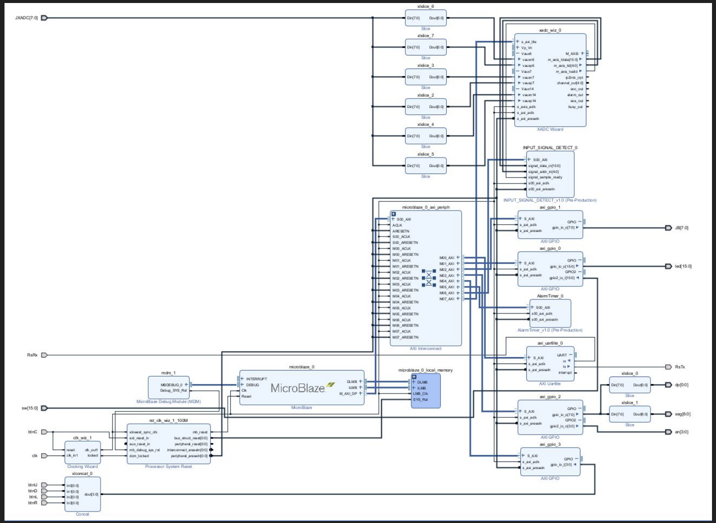
Circuit Results

- It was decided to use AC to measure the coil signals.
- Everytime the metal screw comes near the coil, the corresponding AC signal voltage drops by a measurable amount.
- Based on which coil drops we can detect the location of the metal screw in relation to the coils. In this way our project is a metal detector





Hardware Implementation



- Using the MicroBlaze Softcore Processor, General Purpose I/O from our Basys 3 FPGA, the onboard XADC wizard, and custom IP blocks we were able to implement a hardware schematic in Vivado to power our system.
- Inputs and Outputs were declared as they appear on the Basys 3 (eg. buttons being treated as inputs and leds being treated as outputs)

Writing From Our Custom IP

The max clear flag, who’s use will be show later, is controlled via our write process. If we write to register “00”, this clear flag goes high, and will go low if register “01” is written to.

```
process (S_AXI_ACLK)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      slv_reg0 <= (others => '0');
      slv_reg1 <= (others => '0');
      slv_reg2 <= (others => '0');
      slv_reg3 <= (others => '0');
    else
      loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
      if (slv_reg_wren = '1') then
        case loc_addr is
          when b"00" =>
               max_clear <= '1';
          when b"01" =>
               max_clear <= '0';
          when b"10" =>

          when b"11" =>

          when others =>
            slv_reg0 <= slv_reg0;
            slv_reg1 <= slv_reg1;
            slv_reg2 <= slv_reg2;
            slv_reg3 <= slv_reg3;
        end case;
      end if;
    end if;
  end if;
end process;
```

Reading From Our Custom IP

Much like the writing from our custom IP, we use reading to grab single bit flags and registers from hardware to be used in software.

```
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
    -- Address decoding for reading registers
    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
    case loc_addr is
      when b"00" =>
            reg_data_out(0) <= peak_find_port6;
            reg_data_out(1) <= peak_find_port7;
            reg_data_out(2) <= peak_find_port14;
            reg_data_out(31 downto 3) <= (others => '0');
      when b"01" =>
            reg_data_out(15 downto 0) <= max_value_port6;
            reg_data_out(31 downto 16) <= (others => '0');
      when b"10" =>
            reg_data_out(15 downto 0) <= max_value_port7;
            reg_data_out(31 downto 16) <= (others => '0');
      when b"11" =>
            reg_data_out(15 downto 0) <= max_value_port14;
            reg_data_out(31 downto 16) <= (others => '0');
      when others =>
         reg_data_out  <= (others => '0');
    end case;
end process;
```

Peak Amplitude Detection Algorithm

- This process uses the XADC output data, address, and data ready flag to allow us to find the point of highest amplitude for each of our channels
- One 16 bit register was assigned for each channels maximum

```
process( S_AXI_ACLK ) is
begin
  if (rising_edge (S_AXI_ACLK)) then
    if ( S_AXI_ARESETN = '0' ) then

    else

      if signal_sample_ready = '1' then
          if signal_addr_in = "10110" then
              curr_value_port6 <= signal_data_in;
              prev_value_port6 <= curr_value_port6;

              if max_clear = '1' then
                  max_value_port6 <= (others => '0');
              elsif (unsigned(max_value_port6) < unsigned(prev_value_port6)) then
                  max_value_port6 <= prev_value_port6;
                  peak_find_port6 <= '0';
              else
                  peak_find_port6 <= '1';
              end if;
          elsif signal_addr_in = "10111" then
              curr_value_port7 <= signal_data_in;
              prev_value_port7 <= curr_value_port7;

              if max_clear = '1' then
                  max_value_port7 <= (others => '0');
              elsif (unsigned(max_value_port7) < unsigned(prev_value_port7)) then
                  max_value_port7 <= prev_value_port7;
                  peak_find_port7 <= '0';
              else
                  peak_find_port7 <= '1';
              end if;
          elsif signal_addr_in = "11110" then
              curr_value_port14 <= signal_data_in;
              prev_value_port14 <= curr_value_port14;

              if max_clear = '1' then
                  max_value_port14 <= (others => '0');
              elsif (unsigned(max_value_port14) < unsigned(prev_value_port14)) then
                  max_value_port14 <= prev_value_port14;
                  peak_find_port14 <= '0';
              else
                  peak_find_port14 <= '1';
              end if;


        end if;
      end if;
    end if;
```

Software Implementation

As shown above we declare I/O and custom IP registers in hardware. To use those in Software, we must define the memory Addresses of each register with its respective offset.

```c
#define LEDS (* (volatile unsigned *)0x40000000)
#define JXADC1_RESULT_REG (*(volatile unsigned *)0x44a20258)
#define JXADC2_RESULT_REG (*(volatile unsigned *)0x44a2025C)
#define ALARM_CNT (* (volatile unsigned *)0x44a00000)
#define ALARM1_ALARM0 (* (volatile unsigned *)0x44a00004)
#define ALARM0_VALUE (* (volatile unsigned *)0x44a00008)
#define ALARM1_VALUE (* (volatile unsigned *)0x44a0000C)
#define BTN (* (volatile unsigned *)0x40030000)
#define PEAK_FOUND (* (volatile unsigned *)0x44a10000)
#define MAX_VALUE_RIGHT (* (volatile unsigned *)0x44a10004)
#define MAX_VALUE_MIDDLE (* (volatile unsigned *)0x44a10008)
#define MAX_VALUE_LEFT (* (volatile unsigned *)0x44a1000C)
#define SEG (* (volatile unsigned *)0x40020000)
#define AN (* (volatile unsigned *)0x40020008)
```

LED Detection & Strength Meter Algorithm

```c
void led_strength(_Bool* leftDetect, _Bool* middleDetect, _Bool* rightDetect)
{

    int mVConst = 0x0177;
    uint16_t mask = 1;
    uint16_t difference1;
    uint16_t difference2;
    uint16_t difference3;



    LEDS = 0x0;



    for(int i = 1; i < 9; i ++){
        difference1 = 0xA500 - (i *mVConst);
        difference2 = 0xA500 - (i *mVConst);
        difference3 = 0xB400 - (i *mVConst);

        if(difference1 >= MAX_VALUE_LEFT){
            LEDS |= mask;
        }

        if(difference2 >= MAX_VALUE_MIDDLE){
            LEDS |= mask;
        }

        if(difference3 >= MAX_VALUE_RIGHT){
            LEDS |= mask;
        }
        mask = (mask << 1);
    }



    if((MAX_VALUE_RIGHT < 0x9E00) & (MAX_VALUE_MIDDLE > 0x9D00) & (MAX_VALUE_LEFT > 0x9B00)){
        LEDS |= (1<<13);
        *leftDetect = 0;
        *middleDetect = 0;
        *rightDetect = 1;
    } else if ((MAX_VALUE_RIGHT > 0xA800) & (MAX_VALUE_MIDDLE < 0x9700) & (MAX_VALUE_LEFT > 0x9D00)){
        LEDS |= (1<<14);
        *leftDetect = 0;
        *middleDetect = 1;
        *rightDetect = 0;
    } else if (((MAX_VALUE_RIGHT > 0xAE00) & (MAX_VALUE_MIDDLE > 0xA100) & (MAX_VALUE_LEFT < 0x9300))){
        LEDS |= (1<<15);
        *leftDetect = 1;
        *middleDetect = 0;
        *rightDetect = 0;
    } else if (((MAX_VALUE_RIGHT > 0xAE00) & (MAX_VALUE_MIDDLE > 0xA4A0) & (MAX_VALUE_LEFT > 0xA000))) {
        *leftDetect = 0;
        *middleDetect = 0;
        *rightDetect = 0;
    }
```

To detect whether the metal right, left, or center on our coils. We used the values of our max registers from our peak detection IP and compared them to constant hex values to see if a coil had a object in range. A for loop was iterated 8 times to allow us to find the strength of the current detected coil.

Object Incrementation Finite State Machine

Once we have detected if a object is detected by a certain coil in the field we then use a boolean flag to increment that coils count. We used a very similar structure to a button debouncer to eliminate any noise in the signal and find a steady state for the signal.

```c
_Bool right_object_detect(_Bool rightDetect)
{

    static enum {NOT_DETECTED, DETECTED_PASS, DETECTED, DETECTED_RELEASE} state = NOT_DETECTED;
    static uint8_t cnt;
    _Bool retval = 0;
    _Bool B = rightDetect;

    switch(state){
    case NOT_DETECTED:
        if(B){
            cnt = 0;
            state = DETECTED_PASS;
        }
        break;

    case DETECTED_PASS:
        if(B & (cnt < 100)) {
            cnt++;
        }else if (B & (cnt >= 100)){
            retval = 1;
            state = DETECTED;
        } else if(!B){
            state = NOT_DETECTED;
        }
        break;

    case DETECTED:
        if(!B){
            cnt = 0;
            state = DETECTED_RELEASE;
        }
        break;

    case DETECTED_RELEASE:
        if(B){
            state = DETECTED;
        }else if (!B & (cnt >= 100)){
            state = NOT_DETECTED;
        }else if (!B & (cnt < 100)){
            cnt++;
        }
    }


    return retval;

}
```

Final Results

Although we were able to detect and increment our FPGA accordingly. It proved that through the use of specific values we encountered many bugs as noise and voltage shifts was inevitably prevalent in our system. A better approach would be to have a more generic software program in order to easily deal with amplitude issues.

Thank You