VHDL AND PROGRAMMABLE LOGIC FOR CUSTOM PERIPHERALS

or

How to build the bits your microcontroller doesn’t have.
Potential Uses for Programmable Logic

PORT expansion
Counters / Timers
Quadrature Decoders
Locked antiphase / Sign-Magnitude PWM / H-Bridge control
Stepper Motor Sequencers: full/half/microstep
DC Motor PWM generation
Speed and Position Measurement
Control Algorithms: P, PD, PID etc.
Embedded processors: custom or IP cores (open source)
Programmable Logic Devices

The three base types of field programmable logic devices are:

Small - PLD, Programmable Logic Devices
Medium - CPLD, Complex Programmable Logic Devices
Large - FPGA, Field Programmable Logic Devices

As may be expected, the cost of a device increases with size, complexity, functionality. FPGA’s can be costly, with many different architectures, and are beyond the scope of this introductory presentation, although some of the lower cost devices are comparable in price with the top-end CPLD’s with a huge increase in capacity.
PLD devices

Lattice Semiconductor GAL

<table>
<thead>
<tr>
<th>VCC</th>
<th>Device</th>
<th>Pins</th>
<th>Fastest Speed</th>
<th>Is (mA)</th>
<th>IσB (µA)</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>GAL16V8</td>
<td>16</td>
<td>163</td>
<td>155</td>
<td>100</td>
<td>3.3V Zero-Power PLD</td>
</tr>
<tr>
<td></td>
<td>GAL16V8EZD</td>
<td>16</td>
<td>163</td>
<td>155</td>
<td>100</td>
<td>3.3V Zero-Power PLD</td>
</tr>
<tr>
<td>5</td>
<td>GAL14V8</td>
<td>10</td>
<td>250</td>
<td>75</td>
<td>--</td>
<td>Universal PLD</td>
</tr>
<tr>
<td></td>
<td>GAL14V8EZD</td>
<td>10</td>
<td>250</td>
<td>75</td>
<td>--</td>
<td>Universal PLD</td>
</tr>
<tr>
<td></td>
<td>GAL18V10</td>
<td>7.6</td>
<td>105</td>
<td>115</td>
<td>--</td>
<td>22V10 Subset</td>
</tr>
<tr>
<td></td>
<td>GAL20V10</td>
<td>7.6</td>
<td>105</td>
<td>115</td>
<td>--</td>
<td>22V10 Subset</td>
</tr>
<tr>
<td></td>
<td>GAL20A10</td>
<td>7.6</td>
<td>105</td>
<td>115</td>
<td>--</td>
<td>22V10 Subset</td>
</tr>
<tr>
<td></td>
<td>GAL20A10</td>
<td>7.6</td>
<td>105</td>
<td>115</td>
<td>--</td>
<td>22V10 Subset</td>
</tr>
<tr>
<td></td>
<td>GAL20B10</td>
<td>7.6</td>
<td>105</td>
<td>115</td>
<td>--</td>
<td>22V10 Subset</td>
</tr>
<tr>
<td></td>
<td>GAL20B10</td>
<td>7.6</td>
<td>105</td>
<td>115</td>
<td>--</td>
<td>22V10 Subset</td>
</tr>
<tr>
<td></td>
<td>GAL22V10</td>
<td>7.6</td>
<td>105</td>
<td>115</td>
<td>--</td>
<td>22V10 Subset</td>
</tr>
<tr>
<td></td>
<td>GAL22V10</td>
<td>7.6</td>
<td>105</td>
<td>115</td>
<td>--</td>
<td>22V10 Subset</td>
</tr>
<tr>
<td></td>
<td>GAL22V10</td>
<td>7.6</td>
<td>105</td>
<td>115</td>
<td>--</td>
<td>22V10 Subset</td>
</tr>
<tr>
<td></td>
<td>GAL22V10</td>
<td>7.6</td>
<td>105</td>
<td>115</td>
<td>--</td>
<td>22V10 Subset</td>
</tr>
</tbody>
</table>

$1-5
In-system programmable: ispGAL

$2-4
20V8 structure

Logic Block Diagram (PDIP/CDIP/QSOP)

PAL is a registered trademark of Advanced Micro Devices, Inc.
20V8 Macrocell

Diagram showing the internal connections of the 20V8 Macrocell with various logic levels and signals like OE, VCC, and inputs and outputs.
CPLD devices

Bigger devices : Higher cost

128 macrocells ➔ $20

<table>
<thead>
<tr>
<th>Table 3. MAX 7000S Device Overview (5.0 V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Useable Gates</td>
</tr>
<tr>
<td>Macrocells</td>
</tr>
<tr>
<td>Maximum User I/O Pins</td>
</tr>
<tr>
<td>$t_{PL}(ns)$ (J)</td>
</tr>
<tr>
<td>$t_{PL}(ns)$ (Z)</td>
</tr>
<tr>
<td>$t_{PS}(ns)$ (J)</td>
</tr>
<tr>
<td>$t_{PS}(ns)$ (Z)</td>
</tr>
<tr>
<td>$t_{OH}(nA)$ (Z)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Package</th>
<th>I/O Pins</th>
</tr>
</thead>
<tbody>
<tr>
<td>44-Pin PLCC (Q)</td>
<td>38</td>
</tr>
<tr>
<td>44-Pin TQFP (J)</td>
<td>36</td>
</tr>
<tr>
<td>84-Pin PLCC</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ispMACH 44AS Family Selector Guide (VCC = 5.0V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>M4A5-32</td>
</tr>
<tr>
<td>M4A5-64</td>
</tr>
</tbody>
</table>

$5-10$

<table>
<thead>
<tr>
<th>ispMACH 4000V (VCC = 3.3V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>4032V</td>
</tr>
<tr>
<td>4034V</td>
</tr>
<tr>
<td>4064V</td>
</tr>
<tr>
<td>4066V</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

cost ➔ $3

AJW Feb 2008
Low-Cost FPGA

But .... only $20-30 for the EP1C6 with almost 6000 Logic Elements (LE), each one of which can be configured in different modes for efficient logic implementations.
Figure 2-5. Cyclone LE
Tools and tool cost

Design Software: generally free

Altera Quartus II Web Edition (student licence)
Lattice ispLever

Programming tools: generally JTAG, can be home made.
Also can use generic programmers, e.g. LV Micromaster

Target boards: Matrix Multimedia CPLD/FPGA for Altera

(... this bit could be costly!)
Byteblaster clones ....

An LS244 and a few resistors
On to some VHDL .....
VHDL - background

VHDL - what is it?

Came out of the US Very High Speed Integrated Circuit programme - VHSIC.

It is a Hardware Description Language - HDL

Very High Speed Integrated Circuit Hardware Description Language: VHSICHDL

..... contracted to VHDL.

What does it do?

It is a specification, simulation and synthesis tool for digital logic design.

What does it consist of?

It is, in effect, several languages in one ... or rather a language with several levels of abstraction within it..

Functionality can be specified at:

- High Level : Behavioural VHDL - not all constructs can be synthesised (e.g. Assert)
- Medium Level : Dataflow VHDL - fully synthesisable
- Low-Level : Structural VHDL - fully synthesisable
VHDL - elements

The ENTITY declaration specifies the inputs and outputs of the design.

The architecture defines the functionality of the design. It can consist of one or more levels of abstraction. There can be several architectures for a specific design - each at different levels of abstraction e.g. one for simulation, one for synthesis.

The VHDL language at the behavioural level is similar to many high-level programming languages, with sequential constructs such as IF..THEN..ELSE and CASE statements. However, VHDL describes hardware, and is therefore fundamentally a CONCURRENT programming language.
Let's take the example of a simple 3-8 line decoder.

<table>
<thead>
<tr>
<th>A2</th>
<th>A1</th>
<th>A0</th>
<th>Q0</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

ENTITY ARCHITECTURE

This is a combinational logic design ......
-- 3 to 8 line decoder  AJW 11/07
-- using conditional signal assignment
entity simple_io2 is
    port (            
        -- Input ports
        SW  : in BIT_VECTOR(2 downto 0);  
        -- Output ports
        LED : out BIT_VECTOR(7 downto 0));
end simple_io2;
architecture dflow of simple_io2 is
begin
    -- Conditional Signal Assignment
    -- each assignment has it's own conditional expression
    LED <= "00000000" when (SW = "000") else
        "00000010" when (SW = "001") else
        "00000100" when (SW = "010") else
        "00001000" when (SW = "011") else
        "00100000" when (SW = "100") else
        "01000000" when (SW = "101") else
        "10000000" when (SW = "110") else
        "10000000";          
end dflow;
Aim: To use a PLD to switch between sign-magnitude and locked anti-phase PWM
Connecting to the L293 Dual H-Bridge Driver

Sign-Magnitude: One way to do this is to set the direction of the bridge with IN1,IN2 (IN3,IN4) and to generate PWM by pulsing the ENABLE line.

IN1 = 1, IN2 = 0 gives forward.
IN1 = 0, IN2 = 1 gives reverse.
IN1 = IN2 gives braking with ENABLE active.
Connecting to the L293 Dual H-Bridge Driver

Locked Anti-Phase: Here the two inputs are driven in anti-phase.

PWM > 50% duty

PWM < 50% duty

At 50% duty, the average current through the motor is zero - therefore the motor is stopped.
One channel L293 driver with SM/LA option

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY L293driver_1ch IS
  PORT
  (   BRAKE, MODE, PWM, DIR_EN : IN STD_LOGIC;
      IN1, IN2, EN1 : OUT STD_LOGIC
  );
END L293driver_1ch;
ARCHITECTURE dfow of L293driver_ich IS
BEGIN
  Ch1:
  PROCESS (MODE, PWM, BRAKE, DIR_EN)
  BEGIN
    IF (MODE = '0') THEN -- MODE = LOCKED ANTI-PHASE
      EN1 <= DIR_EN; -- pass enable through
      IF (BRAKE = '1') THEN
        IN1 <= '0'; -- BRAKE
        IN2 <= '0'; --
      ELSE
        IN1 <= PWM; --
        IN2 <= NOT PWM; -- anti-phase
      END IF;
    ELSE -- MODE = SIGN/MAG
      IF (BRAKE = '1') THEN
        IN1 <= '0'; -- BRAKE
        IN2 <= '0'; --
        EN1 <= '1';
      ELSE
        IN1 <= PWM; -- pulse enable
        IN1 <= DIR_EN; -- set direction on bridge
        IN2 <= NOT DIR_EN;
      END IF;
    END IF;
  END PROCESS Ch1;
END dfow;
One channel L293 driver with SM/LA option

Anti-phase operation

Sign-Magnitude Operation
Concurrent signal assignments give combinational logic.

Processes are a high level, behavioural constructs.

A Process is a concurrent statement - but it enables us to use both combinational and sequential constructs such as:

\[ \text{if ... then ... else} \]
\[ \text{case .... endcase} \]

In addition ... we can infer registered (clocked) logic by the use of

\[ \text{wait until (expression) e.g. wait until (clk = '1');} \]

This last statement is edge sensitive .... and means wait until a +ve edge occurs.
entity dtype is
  (  
    port
     D,CLK : in std_logic;
     Q     : out std_logic
  );

architecture behv of dtype is
begin
  dff: process
    begin
      wait until CLK = '1';
      Q <= D;
    end process;
end behv;

When CLK goes to '1' then
D is applied to Q.

Sequential logic MUST be in a
PROCESS.
Sequential Logic - The simple quadrature decoder

What is quadrature encoding/decoding?

We have two channels from a rotary encoder, CHA and CHB

The direction of rotation is indicated by a +/-90° phase difference between the two channels.

We can generate a X2 pulse stream, and a direction signal, with some quite simple circuitry - a quadrature decoder.
The simple quadrature decoder

Examination of the CHA, CHB waveforms shows:

CHB lagging CHA: CHB = ‘0’ at CHA rising edge (\(\_\_\_\_\_\_\_\_\)\)

CHB leading CHA: CHB = ‘1’ at CHA \(\_\_\_\_\_\_\_\_\)

So CHB can be used to indicate direction, if we can hold the value on CHB at CHA \(\_\_\_\_\_\_\_\_\)

ALSO, we can double the pulse rate by XOR’ing CHA with CHB (this is a phase difference detector)
The simple quadrature decoder

```vhdl
library ieee;
use ieee.std_logic_1164.all;

ENTITY qdec IS
  PORT(
    CHA, CHB: IN STD_LOGIC;
    pulse   : OUT STD_LOGIC;
    direction : OUT STD_LOGIC);
END qdec;

ARCHITECTURE dflow OF qdec IS
BEGIN

  pulse <= CHA XOR CHB;

  fsm: PROCESS
  BEGIN
    WAIT UNTIL (CHA = '1');
    direction <= CHB;
  END PROCESS fsm;
END dflow;
```
library ieee;
use ieee.std_logic_1164.all;

ENTITY qdec IS
PORT(
    CHA, CHB: IN STD_LOGIC;
pulse : OUT STD_LOGIC;
direction : OUT STD_LOGIC);
END qdec;

ARCHITECTURE dflow OF qdec IS
BEGIN

    pulse <= CHA XOR CHB;

    fsm: PROCESS
    BEGIN
        WAIT UNTIL (CHA = '1');
direction <= CHB;
    END PROCESS fsm;
    END dflow;

<table>
<thead>
<tr>
<th>Name</th>
<th>80.0 ns</th>
<th>160.0 ns</th>
<th>240.0 ns</th>
<th>320.0 ns</th>
<th>400.0 ns</th>
<th>480.0 ns</th>
<th>560.0 ns</th>
<th>640.0 ns</th>
<th>720.0 ns</th>
<th>800.0 ns</th>
<th>880.0 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>direction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pulse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Modifications to the simple Quadrature decoder

With the addition of more sequential logic, we can multiply the x2 output from the XOR function by 2 to give 4 times the pulse rate.

To accomplish this we need to build an edge-detect circuit that detects both positive-going and negative-going edges – a double-edge detector.

This is often done with a monostable, but this method requires an external RC.

As we are dealing with digital design in VHDL, we'll do it with a simple state machine.
A digital double-edge detector

The state machine waits for a positive edge from CHA, pulses the output, then waits for a negative edge from CHA, and pulses the output again. The cycle repeats indefinitely.

One modification to this FSM would be to determine the start-up level of CHA, and to enter the cycle at +ve or -ve edge detect as appropriate.
A digital double-edge detector

```verilog
always @ (posedge clk or negedge rst)
begin
  if (!rst)
  begin
    state <= 0;
  end
  else
  begin
    case (state)
      0:
      begin
        if (pulse == '1')
          state <= 1;
        else
          state <= 0;
      end
      1:
      begin
        state <= 2;
      end
      2:
      begin
        if (pulse == '1')
          state <= 2;
        else
          state <= 3;
      end
      3:
      begin
        if (pulse == '1')
          state <= 0;
        else
          state <= 0;
      end
      others:
      begin
        state <= 0;
      end
    endcase
  end
end
```

END PROCESS;

END module;

END WRONG;
```
FSM model, RTL view and Simulation for QDEC2
library ieee;
use ieee.std_logic_1164.all;

ENTITY qdec IS
    PORT
        |
        CHA, CHB : IN STD_LOGIC;
        clk : IN STD_LOGIC;
        pulse : BUFFER STD_LOGIC;
        pulseX2 : OUT STD_LOGIC;
        direction : OUT STD_LOGIC;
    );
END qdec;

ARCHITECTURE dflow OF qdec IS
    signal SR: std_logic_vector (1 downto 0);
BEGIN
    pulse <= CHA XOR CHB;
    dir:
    PROCESS
    BEGIN
        WAIT UNTIL (CHA = '1');
        direction <= CHB;
    END PROCESS dir;

    dbl_edges:
    PROCESS
        variable state: integer range 0 to 5;
    BEGIN
        WAIT UNTIL (clk = '1');
        case state is
            when 0 =>
                pulseX2 <= '0';
            when 1 =>
                pulseX2 <= '1';
            when 2 =>
                pulseX2 <= '0';
            when 3 =>
                pulseX2 <= '1';
            when others =>
                state := 0;
        end case;
    END PROCESS;
END dflow:
Further Modifications

The problem with the simple detector is that the direction indication is synchronised to the rising edge of CHA. By multiplying the pulses by 2 and 4, we have the situation where a pulse can be counted in the wrong direction.

Direction change is detected here ... ... but not indicated until here.

This pulse will be counted in the wrong direction.

Direction change is detected here ... ... but not indicated until here.

This pulse will be counted in the wrong direction.
Further Modifications

We can improve the response time for direction indication by looking at BOTH CHA and CHB events – i.e. changes in level.

We can do this by looking at the sequence of changes as indicated above.

(0,2,3,1), (0,2,3,1), etc results in a low on the direction bit, whereas (0,1,3,2), (0,1,3,2), etc results in a high on the direction bit
FSM model and Simulation for QDEC3

Key: CHA, CHB / DIR
example: 01/1
CHA, CHB input
DIR output

Direction change indicated by 10 to 00 transition
First pulse in new direction

Last pulse before direction change
BEGIN
    WAIT UNTIL (clk = '1');
    CASE state IS
    WHEN 0 =>
        if AB = "01" then
            direction <= '1';
            state := 1;
        elsif AB = "10" then
            direction <= '0';
            state := 2;
        else
            state := 0;
        end if;
    WHEN 1 =>
        if AB = "11" then
            direction <= '1';
            state := 3;
        elsif AB = "00" then
            direction <= '0';
            state := 2;
        else
            state := 1;
        end if;
    WHEN 2 =>
        if AB = "00" then
            direction <= '1';
            state := 0;
        elsif AB = "11" then
            direction <= '0';
            state := 3;
        else
            state := 2;
        end if;
    WHEN 3 =>
        if AB = "01" then
            direction <= '0';
            state := 1;
        elsif AB = "10" then
            direction <= '1';
            state := 2;
        else
            state := 3;
        end if;
    END CASE;
END PROCESS fsm;
RTL viewer output for QDEC3

Top level Entity Name: qdec
Family: MAX1000S
Device: EPM7128SLC84-10
Timing Models: Final
Mask timing requirements: N/A
Total macros: 8 / 128 (6 %)
Total pins: 10 / 68 (15 %)
Summary - Why use Programmable Logic?

Most, if not all, of the applications mentioned in the discussion can be implemented in software on a microcontroller - so why use programmable logic devices - PLD, CPLD, FPGA?

- Shortage of I/O pins on processor of choice (POC)
- Limited functionality on POC - e.g. no SPI, PWM, Quadrature decoders
- Functional partitioning - removing some functionality to other hardware (programmable logic or additional microcontrollers) often simplifies design, debugging and test.
- Flexibility with focus - peripherals are designed for a specific task
- The design is not limited by compromise - e.g. Int-On-Change vs. ICD Debug (All ICD PICs), RS232 vs SPI (18F4550).
- High speed: not an issue in most applications for micromouse - but this does make vision systems a possible sensor solution.

- And ..... why not?
That’s all Folks....