Embedded Systems

"System On Programmable Chip"
Programmable interface design
Parallel port Design
Odometry - PWM

René Beuchat
Laboratoire d'Architecture des Processeurs
rene.beuchat@epfl.ch
Design of a Parallel Port

Example of a development methodology of a programmable parallel port interface

- Processor Interface
- Processor view: registers model
- Interface design
- Realization in VHDL
- Test Bench and simulation
Programmable **Parallel Input/Output** Interfaces are very common on a SOPC. They allow access to simple bits of information between the processor and the external world.

The control is done by the processor through registers in the programmable interface.

Each register is seen at a specific address.
Typical SOPC

Avalon master interface

Altera FPGA

Processor

SDRAM Memory

Flash Memory

SDRAM controller

Tristate Bridge

Avalon slave interface

PIO: Parallel Input Output

Avalon Switch Fabric

UAR

TIMER

Ethernet Interface

Ethernet MAC + PHY

Buttons + LEDs

LCD Screen

TxD

RxD

Reset

Clock

Avalon master interface

Avalon slave interface

PIO: Parallel Input Output
Goal

The objective here is to design one interface for an Avalon bus as a slave module.
The main characteristics of the module are:

- Bidirectional Port,
- Programmable Direction for each bit
- Special features for modifying the port bits

- Realization in VHDL for FPGA (that can be simulate and synthesizable)
Avalon Parallel Port, main features

• Realization of the parallel Port on Avalon Bus with:
  ➢ 8 bits port
  ➢ Programmable direction for each bit
  ➢ Write direct to PortPar
  ➢ Function Setbits
  ➢ Function Clrbits
• We have a softcore processor on an Avalon bus in a FPGA.

• In the FPGA a specific parallel port is to be developed, it has to be added on the Avalon with SOPC Builder in the Quartus II environment.
Avalon Slave bus Cycles

- Read and Write Access
- Synchronous
- Separate data bus for read and write access
Slave Avalon Bus Specifications

The Avalon bus provide signals to the module:

- **nReset**: Initialization
- **Clk**: Clock
- **Address(n..0)**: Address, the address is a register number address.
- **ChipSelect**: Selection of this module
- **Read**: Read access
- **ReadData[7..0]**: Data to provide by the module in read access
- **Write**: Write access
- **WriteData[7..0]**: Data send to the module in write access
Parallel Port Accesses (1)

- 8 bits bidirectional Port,
  - Each pin can be specified as input or output
  - The direction is specified in \texttt{RegDir}, (0 : input, 1 : output)
  - The direction can be read back

- The state of the port at the pin level can be read in : \texttt{RegPin}
Parallel Port Accesses (2)

- The state value is memorized in a register: \( \text{RegPort} \rightarrow \text{Port Register} \)
- To update this register, 3 accesses are available:

  1. **RegPort**: Direct memorized value: '0' or '1'
  2. **RegSet**: The bits specified at '1' level during the write cycle at this address, are saved as '1' in the register, the others bits are not changed
  3. **RegClr**: The bits specified at '1' level during the write cycle at this address, are saved as '0' in the register, the others bits are not changed

- This register can be read back
Parallel Port External interface

- **ParPort** is the signal name for the pins.
- If the direction is output:
  - The value memorized in **RegPort** is Outputted.
- If the direction is input:
  - The output value is 'Z': High impedance (tristated)

- In both case the value at the pin interface can be read with an access at **RegPin**
Parallel Port Module on Avalon

Avalon

- Clk
- nReset
- Address[2..0]
- ChipSelect
- Write
- WriteData[7..0]
- Read
- ReadData[7..0]

ParPort[7..0]

FPGA Pins

ParallelPort

- 7 0
- RegDir
- RegPort
- RegSet
- RegClr
- RegPin
I/O Addresses in the module, access map

<table>
<thead>
<tr>
<th>Adresses in the module</th>
<th>Write Registers</th>
<th>DataWrite [7..0]</th>
<th>Read Registers</th>
<th>DataRead [7..0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RegDir</td>
<td>→ iRegDir</td>
<td>RegDir</td>
<td>iRegDir →</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>Don't care</td>
<td>RegPin</td>
<td>ParPort →</td>
</tr>
<tr>
<td>2</td>
<td>RegPort</td>
<td>→ iRegPort</td>
<td>RegPort</td>
<td>iRegPort →</td>
</tr>
<tr>
<td>3</td>
<td>RegSet</td>
<td>→ iRegPort</td>
<td>-</td>
<td>0x00</td>
</tr>
<tr>
<td>4</td>
<td>RegClr</td>
<td>→ iRegPort</td>
<td>-</td>
<td>0x00</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>Don't care</td>
<td>-</td>
<td>0x00</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>Don't care</td>
<td>-</td>
<td>0x00</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>Don't care</td>
<td>-</td>
<td>0x00</td>
</tr>
</tbody>
</table>
 Registers selection

To do:

VHDL entity & architecture
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
ENTITY ParallelPort IS
  PORT(
      -- Avalon interfaces signals
      Clk : IN std_logic;
      nReset : IN std_logic;
      Address : IN std_logic_vector (2 DOWNTO 0);
      ChipSelect : IN std_logic;
      Read : IN std_logic;
      Write : IN std_logic;
      ReadData : OUT std_logic_vector (7 DOWNTO 0);
      WriteData : IN std_logic_vector (7 DOWNTO 0);
      -- Parallel Port external interface
      ParPort : INOUT std_logic_vector (7 DOWNTO 0)
  );
End ParallelPort;
Architecture: Internal signals

-- signals for register access

signal iRegDir : std_logic_vector (7 DOWNTO 0);
signal iRegPort : std_logic_vector (7 DOWNTO 0);
signal iRegPin : std_logic_vector (7 DOWNTO 0);
ARCHITECTURE comp OF ParallelPort IS

... SIGNAL ...
BEGIN

-- Parallel Port output value
pPort:
process(iRegDir, iRegPort)
begin
for i in 0 to 7 loop
  if iRegDir(i) = '1' then
    ParPort(i) <= iRegPort(i);
  else
    ParPort(i) <= 'Z';
  end if;
end loop;
end process pPort;

-- Parallel Port Input value
  iRegPin <= ParPort;
  -- others processes..

END comp;
Avalon write slave, 0 wait
-- Process Write to registers
pRegWr:

process(Clk, nReset)
begin
    if nReset = '0' then
        iRegDir <= (others => '0');    -- Input by default
    ....
    elsif rising_edge(Clk) then
        if ChipSelect = '1' and Write = '1' then -- Write cycle
            case Address(2 downto 0) is
                when "000" => iRegDir <= WriteData;
                when "010" => iRegPort <= WriteData;
                when "011" => iRegPort <= ?????;
                when "100" => iRegPort <= ?????;
                when others => null;
            end case;
        end if;
    end if;
end process pRegWr;
ParallelPort Architecture, registers access

-- Process Write to registers
pRegWr:
  process(Clk, nReset)
  begin
    if nReset = '0' then
      iRegDir <= (others => '0');   -- Input by default
    ..... 
    elsif rising_edge(Clk) then
      if ChipSelect = '1' and Write = '1' then  -- Write cycle
        case Address(2 downto 0) is
          when "000" => iRegDir <= WriteData;
          when "010" => iRegPort <= WriteData;
          when "011" => iRegPort <= iRegPort OR WriteData;
          when "100" => iRegPort <= iRegPort AND NOT WriteData;
          when others => null;
        end case;
      end if;
    end if;
  end process pRegWr;
Avalon slave
read, 0 wait, asynchronous peripheral

This Example Demonstrates
Read transfer from an asynchronous peripheral
Zero wait states
Zero setup

Relevant PTF Parameters
Read_Wait_States = “0”
Setup_Time = “0”

ReadData available at next rising edge of clk (E)
Avalon, read 1 with 1 wait

- 1 wait for read access
-- Read Process to registers

pRegRd:
    process(Clk)
    begin
        if rising_edge(Clk) then
            ReadData <= (others => '0');    -- default value
            if ChipSelect = '1' and Read = '1' then -- Read cycle
                case Address(2 downto 0) is
                    when "000" => ReadData <= iRegDir;
                    when "001" => ReadData <= iRegPin;
                    when "010" => ReadData <= iRegPort;
                    when others => null;
                end case;
            end if;
        end if;
    end process pRegRd;
Test and implementation

- This module can be now be tested by simulation with a test bench or timing stimulation → **to do as exercise**
  - Read, Write access have to be generated to control the registers and verify the result
- Then a **new component** can be created with **SOPC Builder**
- It can then be integrated in a NIOSII system
- The system can be generated from SOPC
- Compiled by QuartusII after added to a schematic design
- Downloaded on a real system
- Program the NIOSII processor to access the registers (NIOS IDE) and test the full system
Creation in QuartusII / SOPC Builder

1. You need to create a **Project** for each programmable interface you developed in QuartusII

2. **Don't use space** in directory/files names

3. When you create a VHDL entity/architecture, the name of the file is the name of the **entity**

4. Entity and architecture are in the same file

5. You create a **new Project** for the full design
Conclusion

• Now you are able to design a specific programmable interface for a bus for Programmable FPGA, here it was the Avalon

• The methodology for this kind of interface is similar for Amba, Wishbone or others internal bus

• The bus generation depend on the used tools, here SOPC Builder do the job!
Conclusion

• You know:
  ➢ The basic use of Avalon synchronous bus
  ➢ The memory map model of a programmable interface
  ➢ To translate this model in VHDL
  ➢ To control pins on a FPGA ('0', '1', 'Z')
  ➢ To implement the interface module in a complete system
Others Programmables Interfaces design exemples

- **PWM**: Pulse Width Modulation
- **Odometer**: Distance / Speed measurement
Pulse Width Modulation

- Generation of a continuous pulse train
- Can be used as a D/A converter
• To control a motor, a PWM module with 2 outputs can be used:
  - PWMa and PWMb
  - They are connected to a DC motor through a H-bridge
  - Depending which output send the PWM signal, the motor turn in one direction or the other.
    - PWMa ← PWM, PWMb ← Idle → Direction A
    - PWMa ← Idle, PWMb ← PWM → Direction B
Programmable Interface … PWM

PWM Output control

PWM
Direction
Polarity

PWM selection

H-Bridge
Si…..

M
Programmable Interface … PWM

- Polarity: Level of the Active part of the PWM
  The Idle level is the inverted value of the Polarity
Programmable Interface … PWM

- We want to be able to program:
  - The period of the PWM signal
  - The active Time (Duty) of the PWM
  - The polarity of the active time ('0') or ('1')
  - Be able to enable/disable the output → a command register
  - Direction is the sign of the Duty
    - If the Duty is positive:
      - PWMa is the PWM output, PWBb is in the Idle state
    - If the Duty is negative:
      - PWMb is the PWM output, PWBa is in the Idle state
  - Period value is on 15 bits
  - Duty is on 16 bits signed in cpl'2 → real duty is the abs(RegDuty)
  - The new Duty is to be used only at the end of a Period if the
    PWM output is enabled
Programmable Interface … PWM

- Propose a register map of this interface
- Create the entity in VHDL
- Implement the Architecture
- Simulate the access
- Create a component with SOPC Builder
- Integrate it in a NIOSII system
Odometer

• A 2 signals system to measure speed and displacement
• Depending on the phase of A and B the direction of the rotation can be determinate
Programmable Interface … Odometer

• An odometer module based on this captor has 2 input signals used to increment/decrement a readable counter for distance measurement

• How could we measure the speed?
  ➢ As: \( \text{speed} = \frac{\text{distance}}{\text{time}} \)
  ➢ \text{Distance} is proportional to the number of pulses (edge)
    1. Counting the time between 2 captor pulses (i.e. 2 rising edge of A)
    2. Counting the number pulses during a fixed time
Programmable Interface … Odometer

- @dt a Speed counter is save in a speed register
- Speed counter is Cleared and start counting again
- Speed is proportional to the number of counted Odo pulses between dt
- dt pulses can be generated by a programmable interface

OdoA

OdoB

dt

Distance Cnt

Speed Cnt

Speed
Programmable Interface … Odometer

• Propose a register model of the programmable interface

• Imagine you have to use it as a software programmer, are you happy with your model?
  ➢ Yes → OK go on and implement it.
  ➢ No → Correct it until it's a nice proposition

• Implement it in VHDL and test it with simulation
Odometer.. Sampling of input signals

• The OdoA and OdoB signals are **asynchronous** related to a Clock signal used in a FPGA or a microcontroller
• For a FPGA design, they have to be **synchronized** before use by a synchronous module inside the FPGA.
• At least 2 D Flip-Flop are needed for input synchronization
Odometer.. Sampling of input signals

- If an external asynchronous signal is used inside a synchronous system it needs to be synchronized before use

- Why?
  1. Metastability problem
  2. At the same clk sampling time (i.e. rising_edge(Clk)), all the logic elements using the signal and clocking it need to see it at the same level!!
Metastability problem:
To be correctly sampled by a FF a signal needs to respect 2 very important timings:

- \( T_{su} \): \textit{Set up time}
- \( T_{hold} \): \textit{Hold time}
Odometer.. Sampling of input signals

If the rule is NOT respected:

- The output can be '0' or '1' \(\rightarrow\) good
- The output can be in an intermediate level for an undefined time \(\rightarrow\) metastable level
Odometer.. Sampling of input signals

- The level of the metastable signal is between the '0' and the '1'.
- The time the metastable signal stay is probabilistic and theoretically could be infinitive. Practically it disappears at the next signal sampling.
- Usually a DFF sampling a metastable level would not propagate it. As for this intermediate level, a decision is take for a '0' or a '1'.
- It could propagate to a next DFF if the level change just at the sampling point to the metastable threshold, the probability is very low but not 0!
- Thus depending on the hardness of the design to do, more DFF are needed.

Manufacturer provides information about the parameters for metastability.
Odometer.. Sampling of input signals

If the rule is NOT respected:

- **Very bad for 1 DFF → worst if the same signal D is going to more than 1 DFF:**
- **each could see a different input level**

Qa, Qb: 2 DFF output
Odometer.. Sampling of input signals

→ At the same clk sampling time (i.e. rising_edge), all the logics using the signal and clocking it need to see it at the same level !!
→ A synchronizing system is necessary
The first DFF can have a metastable signal as output Qm.

The second one will probably filter it.

For very high reliability system more DFF could be necessary, delay added!!
The Qsn signal can be used by all the logic that need it: the level will be the same for all the logic.
Inside the FPGA all the DFF using the same D signal need to use the same Clock.

Special global lines are available inside a FPGA for that purpose.

They are limited in number

If we expect to use a normal signal as a clock for a FF, it's a very bad idea.

We need to use the Clock Enable feature of a DFF in a FPGA.
The clock is used only if the CE line is activated ('1')
What about the Odo Input?
We need to detect a rising and falling edge of OdoA and OdoB
Synchronization followed by logic for edge detection
Odometer.. Edge detection of input signals

REa: Rising edge detection on a
FEa: Falling edge detection on a
They can be used sampled by the Clk
Odometer.. Edge detection of input signals
Odometer.. Edge detection A and B
Odometer.. Edge detection of input signals
Odometer.. Edge detection to Counter

Clk

OdoA
OdoB

OdoAS1
OdoAS2
OdoAS3
REa
FEa

OdoBS1
OdoBS2
OdoBS3
REb
FEb

Inc
Dec
Cnt

+ + + + + - -
Odometer.. Edge detection .. Inc/Dec generation

Similar for Dec
Odometer.. Sampling of input signals

→ Now you can found how to use that to increment/decrement a counter for distance measurement

→ And translate in VHDL!!

→ Good work !