Reconfigurable Computing

Partial reconfiguration design

Chapter 8

Prof. Dr.-Ing. Jürgen Teich
Lehrstuhl für Hardware-Software-Co-Design
Reconfiguration advantages
- Fast computation compared to GPP
- Flexible computation compared to ASIC

Partial device re-use allows
- Space saving
- Power saving
Partial Reconfiguration Design - Introduction

- A partially reconfigurable design consists of:
  - A set of full reconfigurable designs
  - A set of partial designs which can be separately downloaded
- The full designs as well as the partial modules are available as full (partial) bitstream used to configure the device
- The partially reconfigurable modules are used to move the system from one configuration to the next one
Partial reconfiguration design - Introduction

- The purpose of this section is to learn how to design a partially reconfigurable system using the current CAD tools and devices.
- The Xilinx FPGAs (Virtex-II and Spartan-II) are some of the few devices on the market allowing partial reconfiguration.
- This section will focus on two Xilinx-based methodologies for designing a partially reconfigurable system:
  - The Xilinx partial design flow
  - The Xilinx small bit modification using JBits
Partial reconfiguration design - Approach

- Traditional design flow
  - Full circuit
  - Constraints define:
    - Placement Constraints
    - Relative Location Constraints
    - Timing Constraints
- Only one full circuit is generated
Partial reconfiguration design - Approach

- Partial reconfiguration flow:
  - Placement constraints must be provided
  - Modules are compiled separately
  - The result is a set of full and partial implementations (EDIF and bitstream).
  - The partial reconfigurable bitstreams are used to move the device from one configuration to another.
Partial reconfiguration design - Approach

- Delaying placement constraints increases degree of freedom

Area constraints provided after a first evaluation

Run-time Relocation

Placement constraints

Placement Constraints (block positions and area)

Technology Mapping
Place and route

Netlist 1 2 3
Full

Netlist 1 2 3
Partial

Full Bitstream 1 2 3

Partial Bitstream 1 2 3

Basic constraints

VHDL
SystemC
HandelC

Basic constraints

Basic constraints

VHDL
SystemC
HandelC

Basic constraints

Basic constraints

Basic constraints

Reconfigurable Computing
Partial reconfiguration on Xilinx Virtex FPGAs

- Create a bitstream database for full and partial modules to be used at run-time for device reconfiguration
The partial design flow

- Modular implementation of a large project
- The team manager defines the structure of the overall project (top-level)
- Each designer or team of designers implement and test each module separately
- The implemented modules are integrated in the final design
- A top-level consists of
  - A set of independent modules
  - Interfaces between the modules
  - Interfaces with the pins
- Each module is assigned a given position and area on the device by means of area constraints
The partial design flow

- For partial reconfiguration, the goal is to generate
  - A set of full designs
  - A set of partial designs
  - The partial designs are used to move from one full design to another
- The input is structured as follows:
  - Top_level
    - Module_1
    - Module_2
    - ...
    - Module_N
- The input language can be any HDL
The partial design flow – Example

- Modular design
  - Static module is fixed for all times
  - Only partial module can be reconfigured

- Insertion of communication macros at fixed positions
The partial design flow – Example VHDL

entity TOP is
port (  x: in std_logic;
        y: out std_logic);
end TOP;

architecture ARCH of Top is

component MACRO
port (   in1, in2 : in std_logic;
         out1, out2 : out std_logic);
end component;

component STATIC_MODULE
port (   x_in, d_in: in std_logic;
         d_out: out std_logic);
end component;

component PARTIAL_MODULE
port (   d_in: in std_logic;
         y_out, d_out: out std_logic);
end component;

signal t1, t2, t3, t4 : std_logic;
The partial design flow – Example VHDL

begin

static: STATIC_MODULE
port map(x_in=>x, d_in=>t4, d_out=>t1);

macro_right2left: MACRO
port map (in1=>t1, in2=>open, out1=>t2, out=>open);

macro_left2right: MACRO
port map (in1=>t3, in2=>open, out1=>t4, out=>open);

partial: PARTIAL_MODULE
port map(y_out=>y, d_in=>t2, d_out=>t3);

end ARCH;

- Macro component is provided by Xilinx and must be inserted in the TOP level for the communication with partial modules
- Static and partial modules do not require any additional changes
The partial design flow – Four Steps

1) Build the top level context
   - Slice Macros at fixed positions are used to communicate between static design logic and reconfigurable logic.
   - Note that there is NO logic except IOs and clocks in the top level design. All logic is contained in one or more 'modules' (E.G. AREA_GROUP).
   - Required files: top.ngc and top.ucf (for constraints)
   - Output file: top.ngo.

2) Build the static modules
   - These are the modules (E.G. logic and routing) that will NOT be dynamically reconfigured.
   - Required files: .ngc for each static 'module'.
   - Output files: top_routed.ncd (without reconfigurable logic)
3) Build the dynamic modules

- Build each flavor of each dynamically reconfigurable module.
- Required files: .ngc for each dynamic 'module'.
- Output files: Routed .ncd file for each flavor of a dynamically reconfigurable module WITHOUT logic and routing from static modules.
The partial design flow – Four Steps

4) Assemble full design with each flavor of each dynamically reconfigurable module. Generate the required bitstreams.
   - Required files: .ncd for static modules and for each flavor of each reconfigurable
   - Output files:
     - a bitstream for full design with each flavor of each reconfigurable module
     - a partial bitstream for each flavor of each reconfigurable module
     - report file
The partial design flow – directory structure

- **Synth** Contains the VHDL code and synthesised Netlists
  - top
  - static
  - Mod1
  - ...
  - ModN

- **Top/Initial** Built top level context
- **Static** Built static modules
- **ReconfigModules** Built reconfigurable modules
- **Merges** Contains assembled bitstreams
The partial design flow – Top level context

- Instantiate static and reconfigurable modules as “black-boxes”
- Connect the modules at the top-level using slice macros between reconfigurable and fixed modules
- Estimate a rectangular bounding region for each module and constrain it to this area
- Constrain top-level I/O ports and slice macros to a fixed locations
- The following command must be run in each initial directory under the corresponding Top-level
  - `cd Top/Initial/`
  - `ngdbuild -modular initial top.ngc`
The partial design flow – Static modules

- Slice macros must be located on boundaries correctly.
- Par will automatically exclude logic from being placed in reconfiguration areas.
- Par will exclude all glitchfull logic (rams / shift registers) from being placed in reconfig zones.
- Par will generate a file called static.used. This is a list of routing resources utilized in the reconfiguration areas by the static design.
- Map, place and route the static modules:
  - cd Static
  - ngdbuild -modular initial ../Top/Initial/top.ngo
  - map top.ngd
  - par -w top.ncd top_routed.ncd
The modular design flow – Dynamic modules

- Copy the static.used file from the static area to "arcs.exclude" in the module directory.
  - This will disallow part of the active module design from using routing utilized by the static design in the reconfig area.
- Par generates a file called "dynamic.used" which is a list of routing resources it utilizes in the Reconfig Area.
- Map, place and route the dynamic modules:
  - `cp Static/static.used ReconfigModules/ModN/arcs.exclude`
  - `cd ReconfigModules/ModN`
  - `ngdbuild -modular module -active rmodule ../Top/Initial/top.ngo`
  - `map top.ngd`
  - `par -w top.ncd top_routed.ncd`
The modular design flow – Create all bitstreams

- Create all bitstreams:
  - `cp Static/top_routed.ncd Merges/static.ncd`
  - `cp ReconfigModules/ModN/top_routed.ncd Merges/modN.ncd`
  - `cp ReconfigModules/ModM/top_routed.ncd Merges/modM.ncd`
  - `PR_verifydesign.bat Merges/static.ncd Merges/modN.ncd Merges/modM.ncd`
The partial design flow – Area constraints

The leftmost boundary of each module and of each Bus Macro must be a multiple of 4

INST "Instance0" AREA_GROUP = "AG_Instance0";
AREA_GROUP "AG_Instance0" RANGE = SLICE_X0Y0:SLICE_X3Y29;
AREA_GROUP "AG_Instance0" MODE=RECONFIG;

Instance name of the module in the top-level

Bounding Box of the module on the chip

State that the module can be reconfigured
The partial design flow – Use of Slice Macros

- The routing of two designs creates unpredictable paths.
- Signals connecting two reconfigurable modules in two different designs can be routed in different ways.
- This can produce malfunction of the design after reconfiguration.
- This can be avoided by providing fixed communication channels (slice macros) among reconfigurable modules.
- Bus macros are tri-state lines running over 4 CLBs in FPGA. Must be placed only at the top level!

![Diagram showing bus macros and their placement](attachment:diagram.png)
The partial design flow – Bus Macro constraints

- Slice Macros use the LUTs for the connection
- Each slice macro provides 8 unidirectional signals
- It is possible to disable the connection temporarily

```plaintext
INST "macro_1" LOC = "SLICE_X34Y40";
INST "macro_2" LOC = "SLICE_X34Y24";
INST "macro_3" LOC = "SLICE_X34Y8";
```
The partial design flow – Example

- Two modules
- One VGA controller
- One colour generator
- The two modules can be partially reconfigured
Small bits manipulation – JBits

- Java API for the Xilinx configuration Bitstream
- Provides function for an off-line modification of the Xilinx Virtex bitstreams with
  - Modification of CLBs, IOBs, Block RAM or PIP (Programmable interconnect points)
  - Access to LUT, MUXes and Flip Flops within a CLB
  - Run-Time manipulation by readback/modify/writeback possible
Small bits manipulation – JBits

- JBits flow

Source: Configuration Bitstream

Bitstream manipulation using JBITS

Generating new (partial) Bitstream

Reading source bitfile:

```java
jbits.read(infileName)
```

Some modifications:

```java
AND_F[] = Expr.F_LUT("F1 & F2 & F3 & F4")
LUTContents = Util.InvertIntArray(AND_F)
jbits.setCLBBits( clbRow, clbCol, BX0.BX0, BX0.BX0, BX0.BY0 )
```

Writing the modified (partial) bitstream (only changes will be saved):

```java
jbits.writePartial(outFileName)
```

(jbits.write(outfileName, JBits.FULL) full bitstream will be written)
Small bits manipulation – JBits

Accessing the LUTs:

### Reading the LUT contents:

```java
lut = jbits.getCLBBits(row, col, LUT.CONTENT[0][LUT.G])
```

(Array of integer values)

### Writing the LUT contents:

```java
int[] LutArray = new int[] {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1}

jbits.setCLBBits(clbRow, clbCol, LUT.CONTENT[0][LUT.G], LutArray)
```

<table>
<thead>
<tr>
<th>G4</th>
<th>G3</th>
<th>G2</th>
<th>G1</th>
<th>LUT Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</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>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
## Small bits manipulation – JBits

### Column extraction:

```java
System.out.println("Touching frames ... ");
for (int x=startColumn; x<endColumn; x++)
{
    column = x;
    System.out.println("Column :"+Integer.toString(x));
    /* touch the frames inside the column */
    for (int y=0; y<22; y++) {
        minorFrame = y;
        /* add minorFrame to the tmp resource */
        //res[0][0] = 0;
        res[0][1] = minorFrame;
        System.out.println("Touching frame :"+Integer.toString(y));
        /* record value of top most bit in minorFrame */
        value = jbits.getCLBBits(0,column,res);
        /* write bit back to touch frame */
        jbits.setCLBBits(0,column,res,value);
    }
}
jbits.enableCrc(true);
/* Generate Partial */
System.out.println("Writing partial bitstream to "+outFileName);
jbits.writePartial(outFileName);
```
Things to know

- A frame is the smallest unit of configuration. Spans the height of the FPGA
- Configuration is glitchless. If you reconfigure a frame with the same data, no glitches appear.
- Reconfiguration “reinitializes” SRL16s and LutRAM, not BlockRam
Case study

- Two Independent Systems on same FPGA:
  - Embedded Linux (PPC+Logic)
  - Audio filter application (Logic)
- Under OS control able to reconfigure the audio filters
- Modified partial flow to create the filters
Reconfigurable DSP Demo

 XC2VP50

 AC97 Core & Filter

 MP3 Player

 Linux System

 Speaker s

 FPGA VFS

 Telnet Login

 Reconfiguration

 FS on
Initial Linux System Floorplan
Physical Structure

- **DSP-Core:** AC97 Controller, Audio-Filter
- **Linux System:** Ethernet, LCD, RAM-Controller, ICAP, SystemACE, UART16550, JTAG, GPIO, PLB2OPB
- **Prohibit Area:** only OPB Routing

Reconfigurable Computing
Initial Budgeting

- Design split into two parts, the Linux system and the reconfigurable DSP region.

- The Reconfigurable region is above and between the two PowerPCs. Does not span whole height of device.

- Linux system - lower half of FPGA.

- Linux system partitioned into a left and right half to avoid SRL16s under reconfigurable region

- Only OPB routing between left and right half.
  - *No bus macros were used.*
Active Module Phase

- Create static and reconfigurable (partial) modules
- Each module was created separately
- Constrained to initial budget constraints

Static Module  Highpass filter  Lowpass filter
Final Assembly Phase

- Merge each reconfigurable module with the static system.
- From each merged design, generate a partial bitstream for the DSP filter.
- The partial bitstream contains the DSP module plus the part of the static system located in the same columns.
- As long as the static system is implemented exactly the same way in each partial bitstream, everything is okay.
- Use difference based partial flow to generate partial bitstreams
Merging Designs

- The XDL Tool is used in this modified partial flow
- `xdl -ncd2xdl {static.ncd, lowpass.ncd} -> {static.xdl, lowpass.xdl}`
- `cat static.xdl lowpass.xdl > merged_lowpass.xdl`
- *Resolve conflicts in merged_lowpass.xdl*
  - All nets and instances need to be unique or merged
    - `GLOBAL_LOGIC*`, `PWR_VCC*`, `PWR_GND*`
    - external io, clk net, and other shared nets needs to be merged
- `xdl -xdl2ncd merged_lowpass.xdl -> merged_lowpass.ncd`
The Last Step: Generating Partial Bitstreams

- Difference-Based Partial Flow
  - documented in XAPP 290
- Bitgen -r option used to create partial bitstream of the difference between the static design and the merged design.

Example:
- `bitgen -g ActiveReconfig:Yes -r static.bit merged_lowpass.ncd lowpass.bit`