FPGA Implementation of the Badger Pipelined 16-bit Processor

Amit Kumar and Karthikeyan Sankaralingam
Vertical Research Group
Department of Computer Sciences, University of Wisconsin-Madison
akumar4@wisc.edu, karu@cs.wisc.edu

Contents

Chapter 1 Introduction 4

Chapter 2 Overview 5

Chapter 3 Cache design guidelines 11

3.1 Top level interface 11

3.2 Design constraints 11

3.3 Hardware testbench 12

Chapter 4 Cache FPGA synthesis 13

4.1 Files and setup 13

4.2 Running the script 13

4.3 Cookbook 13

Chapter 5 Processor design guidelines 14

5.1 Top level module 14

5.2 In_micro 14

5.3 Out_micro 15

5.4 Micro_bench 15

Chapter 6 Processor FPGA synthesis 16

6.1 Files 16

6.2 Running the script 16

6.3 Cookbook 17

Chapter 7 Processor FPGA web-portal execution 18

Chapter 8 Coding problems 25

8.1 State machine coding 25

8.2 Interfacing to port 25

8.3 Multiple sources on single variable 26

8.4 Recursive module 26

8.5 Wide buses 26

8.6 Define and referenced in single clock cycle 26

Chapter 9 Design and Implementation 27

9.1 Introduction 27

9.2 Badger Chip 27

9.3 Implementation Overview 28

9.4 Synthesis 28

9.4.1 Overview 28

9.4.2 Base System Builder Project 30

9.4.3 User logic peripheral 30

9.4.4 Ethernet Mac 30

9.4.5 DDR2 30

9.4.6 MDM (Microblaze debug module) 30

9.4.7 RS232 31

9.4.8 Clock generator 31

9.4.9 Synthesis example 31
9.4.10 Synthesis scripts 31
9.5 FPGA Loading and Execution 32
9.5.1 Loading 33
9.5.2 Program Execution 33
9.5.3 Program verification 34
9.6 Verification 34
9.6.1 Individual module testing 34
9.6.2 Integrated testing 34
9.6.3 Synthesis debugging 34
9.6.4 Hardware real time debugging 34
9.6.5 Software real time debugging 34
9.7 Web interface design 34
Chapter 10 Tool installation and problems encountered in development 36
10.1 Tool Introduction 36
10.2 Tool Installation 36
EDK Installation: 36
ISE Installation: 36
BFM Installation: 36
Modelsim SE 6.3c Installation 36
10.3 Installation problems 36
10.4 Simulation setup problems 36
10.5 XPS BSB problems 37

Chapter 1 **Introduction**

This document describes how to map a 16-bit pipelined processor to an FPGA. This document is meant as a design manual for students detailing the steps they need to follow to run their design on the FPGA. Through a set of automated scripts and a web-portal the synthesis process is simplified and the details abstracted away. In fact, students need to follow a simple set of design guidelines described in A web interface allows the upload of this bit-file to the FPGA and execution of a set of user-specified programs on the chip.

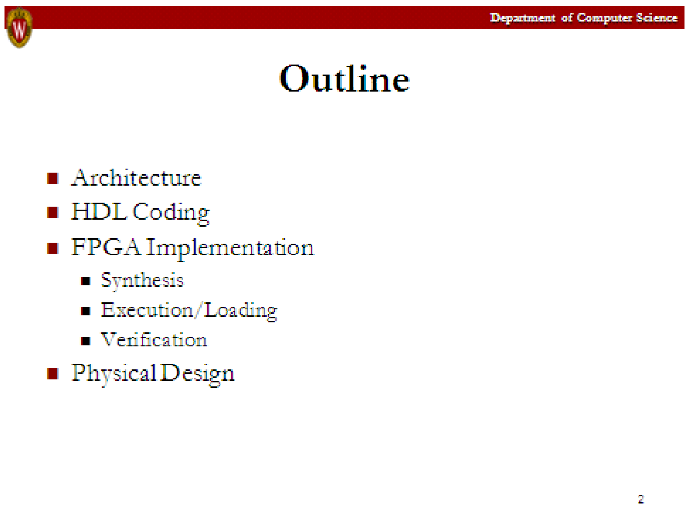
Before mapping the entire processor, the cache alone is mapped to the FPGA and verified by executing a random testbench which is also synthesized to the FPGA. This is much simpler to verify and introduces FPGA synthesis with a simpler design before doing the full processor.

This document is organized as follows. Chapter 2 provides an overview of the process of synthesis and verification on the FPGA. Chapter 3 describes design guidelines to follow while designing the cache. Chapter 4 describes synthesis steps and how to use the scripts to synthesize the cache and execute it on the FPGA. Chapter 5 describes design guidelines to follow for the full processor and verification hooks to add to the processor. Chapter 6 describes synthesis steps for the full processor and Chapter 7 describes the web-portal and how to map this bitfile on the FPGA and run programs on it. Chapter 8 describes describes common errors encountered during synthesis and specific design styles to avoid.

Chapter 9 in this document describes the details of the different steps involved in automating and simplifying the process of FPGA synthesis. Chapter 10 describes tool installation steps.

Chapter 2 **Overview**

Figure out a way to put all slides here, 4 on a page

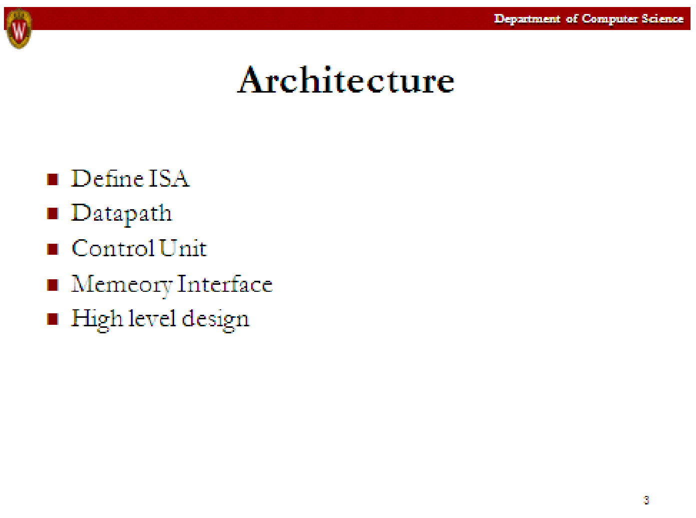


Department of Computer Science

Outline

- Architecture
- HDL Coding
- FPGA Implementation
  - Synthesis
  - Execution/Loading
  - Verification
- Physical Design

2

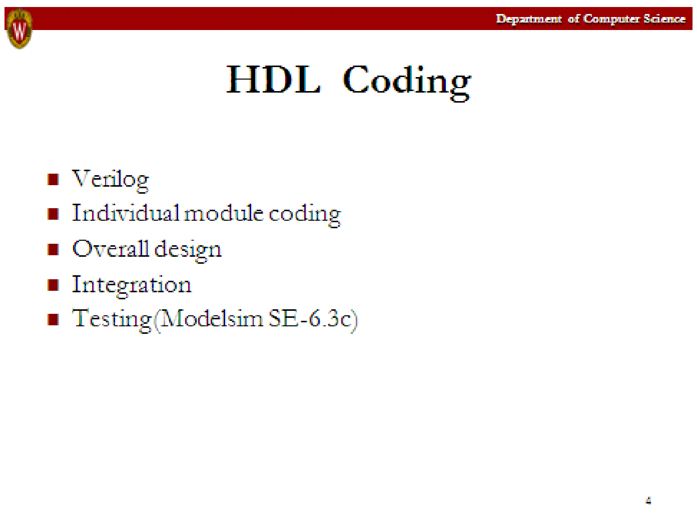


Department of Computer Science

Architecture

- Define ISA
- Datapath
- Control Unit
- Memeory Interface
- High level design

3

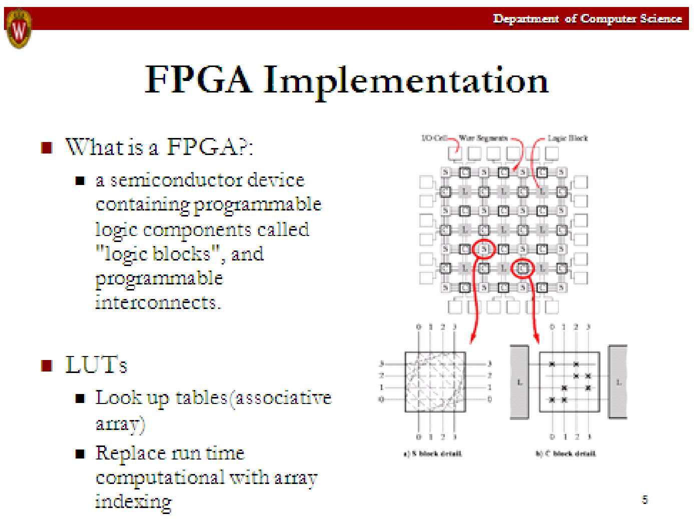


Department of Computer Science

HDL Coding

- Verilog
- Individual module coding
- Overall design
- Integration
- Testing(Modelsim SE-6.3c)

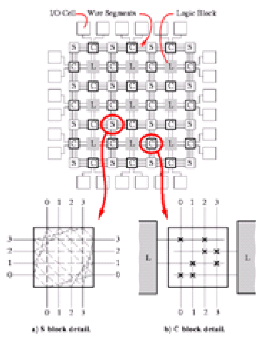
4



Department of Computer Science

FPGA Implementation

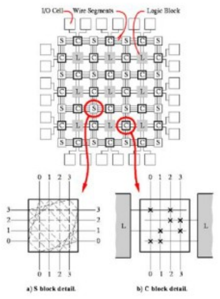
- What is a FPGA?:
  - a semiconductor device containing programmable logic components called "logic blocks", and programmable interconnects.
- LUTs
  - Look up tables(associative array)
  - Replace run time computational with array indexing



5

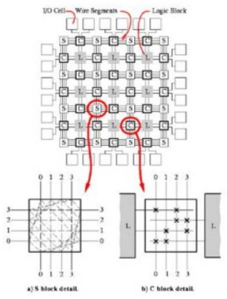
FPGA Implementation

- What is a FPGA?:
  - a semiconductor device containing programmable logic components called "logic blocks", and programmable interconnects.
- Bram
  - On chip block of memory.
  - Configured as fast RAM or ROM.



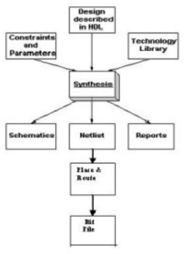
FPGA Implementation

- What is a FPGA?:
  - a semiconductor device containing programmable logic components called "logic blocks", and programmable interconnects.
- CLBs
  - Logic cells.
  - Build design by configuring the interconnection b/w them.

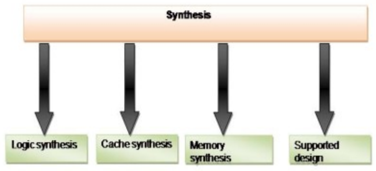


FPGA Implementation

- Synthesis:
  - HDL design
  - Adding constraints
  - Invoke XST
  - Place and Route
  - Get bitfile
  - Automated by Web interface

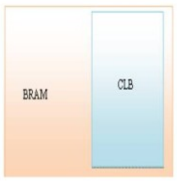


FPGA Implementation



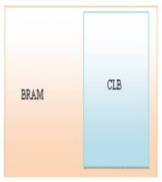
FPGA Implementation

- Memory synthesis
  - Two separate memory.
  - Single ported byte addressable.
  - Each memory have four bank
  - Each bank utilize onchip 36k bram blocks.
  - 0.51ns setup time , 0ns hold time
  - Each block single ported with unregistered output.
  - Default initialization is set to 0.



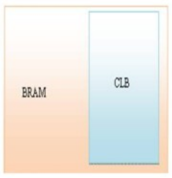
FPGA Implementation

- Cache synthesis
  - Two way set associative cache.
  - Have 256 cache lines.
  - Utilize on chip 16K size BRAM blocks.
  - 0.51ns setup time , 0ns hold time
  - Tag and data array are mapped to Block address.
  - Each block single ported with unregistered output.
  - Default initialization is set to 0.



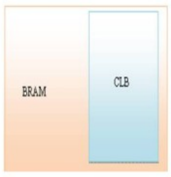
FPGA Implementation

- Logic synthesis
  - Arithmetic unit
  - Register file
  - Data path
  - Control Unit



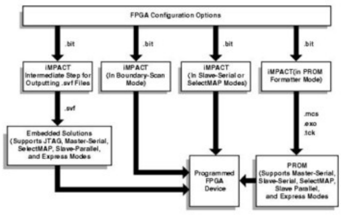
FPGA Implementation

- Supported design synthesis
  - Microblaze soft core processor.
  - DDR2 controller.
  - Ethernet mac controller.
  - Serial/parallel interface controller.
  - Clock generator.
  - ILA ohchip debugger controller.



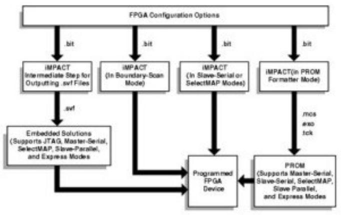
FPGA Implementation

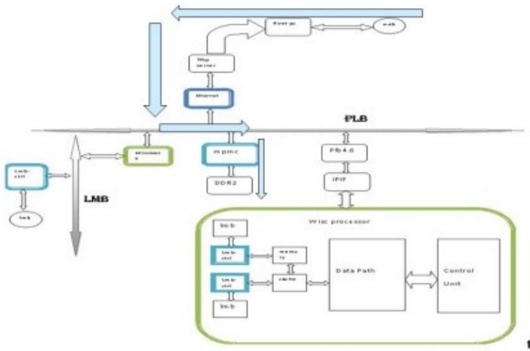
- Configure FPGA's LUTs:
  - Impact tool.
  - Parallel cable-5
  - Slave parallel mode
- Load porgrams:
  - XMD Tool
  - Create Image files.
  - Download to DDR2.

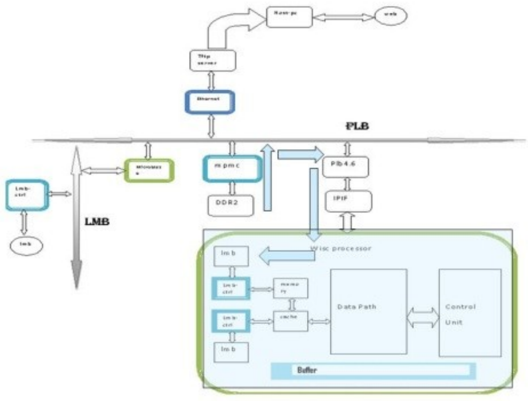

FPGA Configuration Options
.bit
.bit
.bit
.bit
iMPACT Intermediate Step for Outputting .svf Files
iMPACT (In Boundary-Scan Mode)
iMPACT (In Slave-Serial or SelectMAP Modes)
iMPACT(in PROM Formatter Mode)
.svf
.mcs
.exo
.tck
Embedded Solutions (Supports JTAG, Master-Serial, SelectMAP, Slave-Parallel, and Express Modes
Programmed FPGA Device
PROM (Supports Master-Serial, Slave-Serial, SelectMAP, Slave Parallel, and Express Modes

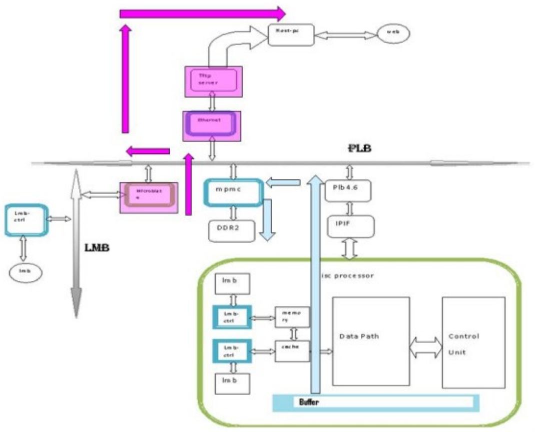
FPGA Implementation

- Running Programs:
  - Initialize instruction memory
  - Start processor.
  - Store result in DDR.
  - Transfer to Host via TFT


FPGA Configuration Options
.bit
.bit
.bit
.bit
iMPACT Intermediate Step for Outputting .svf Files
iMPACT (In Boundary-Scan Mode)
iMPACT (In Slave-Serial or SelectMAP Modes)
iMPACT(in PROM Formatter Mode)
.svf
.mcs
.exo
.tck
Embedded Solutions (Supports JTAG, Master-Serial, SelectMAP, Slave-Parallel, and Express Modes
Programmed FPGA Device
PROM (Supports Master-Serial, Slave-Serial, SelectMAP, Slave Parallel, and Express Modes

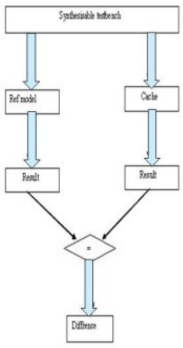






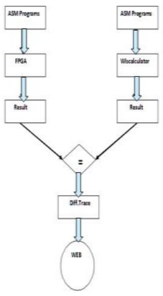
FPGA Implementation

- Cache verification:
  - Synthesizable testbench
  - Reference model.
  - Initialization
  - Comparison
  - Result



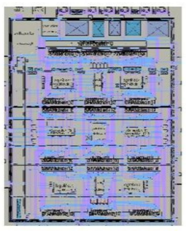
FPGA Implementation

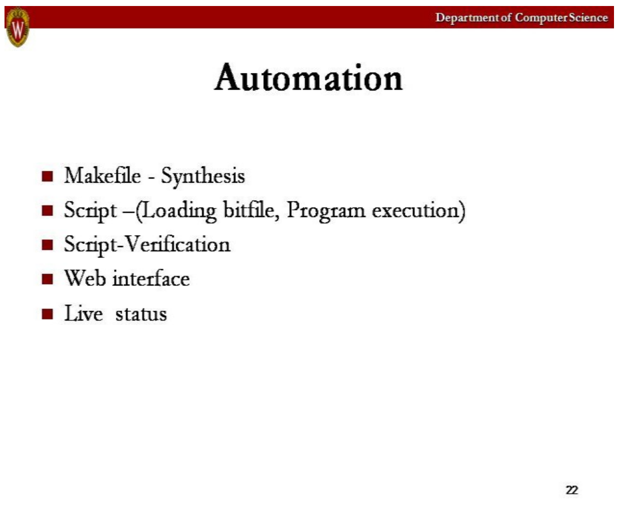
- Processor verification:
  - Run wiscalculator
  - Compare the two results.
  - Give diff.trace file
  - Controlled by Web Interface



FPGA Implementation

- Physical design:
  - Xilinx Ml505
  - Viretx-5
  - EdK,ISE,Coregen,Chipscope,BFM
  - DDR,Ethernet Mac,Microblaze





Automation

- Makefile - Synthesis
- Script –(Loading bitfile, Program execution)
- Script-Verification
- Web interface
- Live status

Chapter 3 Cache design guidelines

This chapter describes the guidelines that students must follow while designing the cache. Section 1 describes the top level interface, Section 2 describes design rules for the cache and Section 3 describes the steps to run the hardware testbench.

3.1 Top level interface

The cache top level module should be mem_system_hier.v and follow the interface shown below:

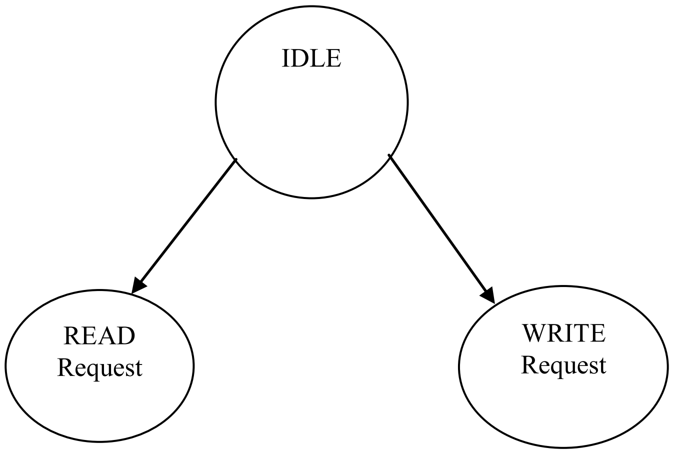
```
`define Width 15
module mem_system_hier(
   // Outputs
   DataOut, Done, Stall, CacheHit,
   // Inputs
   Addr, DataIn, Rd, Wr, createdump,
  clk,rst );
   input [`Width:0] Addr;
   input [`Width:0] DataIn;
   input        Rd;
   input        Wr;
   input        createdump;
   input        clk,rst;
   output [`Width:0] DataOut;
   output Done;
   output Stall;
   output CacheHit;
```

3.2 Design constraints

The cache design must adhere to the conventions described below for hit/miss detection and stalls.

Hit/miss detection: The cache must take two cycles to detect where the request is a hit or a miss. On a miss data can take longer than 2 cycles to arrive. The example below shows how we can implement this logic in cache.

- Assume you have three states: idle state, read state, and write state.
- The cache should remain in the idle state when not receiving a request. When it receive the read/write request then based on the request type we should go to corresponding read/write state and then make a decision of cache hit or not and return that data in next state.



Explanation: The reason why we need to create this model is that, in hardware cache hit search and data returning will take one cycle so we must take 2 cycles.

Stall signal: The cache stall signal should be low when there is no read or write signal from the testbench. At the reset or at the starting our stall signal will be low.

Explanation: The hardware testbench gives the read/write request to cache when the stall signal from the cache is low. If our stall signal is high then hardware test bench will never give a read or write request and the process can't be started.

3.3 Hardware testbench

To make the debugging easy we created a software simulation environment exactly similar to hardware synthesized environment. This environment contains all the synthesizable files which we need for the synthesis. Student must first run this hardware testbench to make sure their design meets the synthesis constraints before doing actual synthesis. We need to do the following steps to run this hardware testbench.

1. Copy the synthesizable testbench zip from /p/vertical/projects/fpga-wisc/downloads/wisc_cache.zip file to your home directory.
2. Extract the zip by giving the following command **(unzip wisc_cache.zip –d .)**
3. Copy your cache design file into this folder.
4. Now create a Modelsim project with all these files and run the **main_cache** (top level module for this project**)** module and see if our design is correct in the Modelsim display panel.
5. If all goes well, proceed to synthesis. Else first debug your design in Modelsim.

Chapter 4 **Cache FPGA synthesis**

This chapter describes synthesis steps for the cache design. In this cache synthesis exercise the cache is mapped to the FPGA along with the random testbench. This chapter provides details about the synthesis scripts and steps involved in running these scripts. Section 4.1 describes files which need to be provided for synthesis. Section 4.2 describes the steps to run the scripts. Prior to synthesizing the cache on the FPGA you must have adhered to the cache design guidelines and verified RTL-only Modelsim execution and synthesized-Modelsim execution describe in the cache design guidelines chapter.

4.1 Files and setup

To start the synthesis for the cache design create on folder on your local machine (for e.g. emperor01) and put your verilog files in that folder. While putting verilog files to this folder we need to take care that no unnecessary files are placed in this directory. Specifically all the files in the folder should be synthesizable. In addition, **you must NOT have the following files in this folder.**

```
        Dff.v
        Clkrst.v
        Cache.v
        Four_bank_mem.v
        Memory4c.v
        Mem_system_ref.v
        Mem_system_randbench.v
```

These files are automatically added to our design by the script.

Download the workspace folder from the `/p/vertical/projects/fpga-wiscsp08/tools/workspace/cache` to your local home directory. Place this directory at the same level as your verilog_code directory. For example:

```
home/
    cs552/
         cache/
              verilog_code
              workspace
```

4.2 Running the script

Type the following the command to run the script

```
prompt> cache_script  path1 path2.
```

Here `path1` is the path of our folder where we have our verilog files. Here `path2` is the path of the workspace folder where we have our workspace. Below is an example which shows how to run the command:

```
prompt> /p/vertical/projects/fpga-wisc08/tools/scripts/cache_script
~foobar/cache/verilog_code ~foobar/cache/workspace
```

The workspace folder contains several useful files required for synthesis. These files contain the configuration details of FPGA pin, I/O, Xilinx inbuilt controller (i.e. DDR2, Ethernet Mac), microblaze port settings .All these files are required to run our design in the FPGA.

4.3 Cookbook

1. Make one folder on your local machine (for e.g. emperor01) which contains the verilog files.
2. Remove all the garbage verilog files from this folder that you are not using in your design.
3. Remove the above mention 6 files (diff.v, cache.v etc) from this folder.
4. Modify the design according to the design constraints as mention in the cache design help document.
5. Download the workspace folder from the `/afs/cs.wisc.edu/p/vertical/projects/fpga-wiscsp08/tools/workspace/cache` to our local home directory.
6. Run the command `cache_script path1 parth2.`
7. Here path1 is the path of our verilog files folder and path2 is the path of our workspace folder.

Chapter 5 Processor design guidelines

This chapter describes the high level interface between the student design and the glue logic (hardware set up environment used for FPGA synthesis). Specifically this document explains the testbench signals and other signals which will be used to monitor and control the design. Here we describe the flow and actual use of three signals *in_micro, out_micro* and *micro_bench* in detail.

5.1 Top level module

The top level verilog module should be proc_hier.v and must follow the interface shown below:

```
`define In_Width 40
`define Out_Width 20
`define Bench_width 110

module proc_hier(clk,rst,err,in_micro,out_micro,micro_bench);
input clk,rst,err;
input [`In_width :0] in_micro;
output [`Out_Width :0] out_micro;
output [`Bench_Width:0] micro_bench;
```

In_micro signals are used for initialization of the instruction memory. Out micro signals are used to obtain confirmation of memory initialization and micro_bench signals are used for monitoring the processor design. All these three signals must be carried from top level module to the module where they are instantiated. You must **not** refer to these signals as wires as done in Modelsim simulation. For example, `micro_bench [101] = proc.idecode.ifetch.reg.regwrite` is NOT allowed. The reason for this is XST (Xilinx synthesis tool) does not recognize the path of a signal unless it is explicitly defined with module hierarchy as shown in the Figure 1 or Figure 2. Further details on these signals and their use is described below.

5.2 In_micro

These are the signals which are used to initialize the instruction memory. They are used to load the actual program that is executed into the processor's memory. These signals should be carried out from top level module to the instruction memory.

```
mem_system      instr_mem(.DataOut(data_out),   .Done(ready),   .Stall(stall),
.CacheHit(hit),.err(err), .Addr(currentpc_t), .DataIn(data_in_t), .Rd(rd_t),
 .Wr(wr_t),                .createdump(1'b0),                   .clk(clk),
.rst(mem_rst),in_micro(in_micro),out_micro(out_micro));
```

In_micro signals are only necessary for the hardware design as in hardware we can't use the $readmemh to read the memory data from a file. Here we need to explicitly initialize the memory using firmware and Microblaze acts as the firmware for the Badger chip.

Below is an example which shows how this hierarchy is carried through.

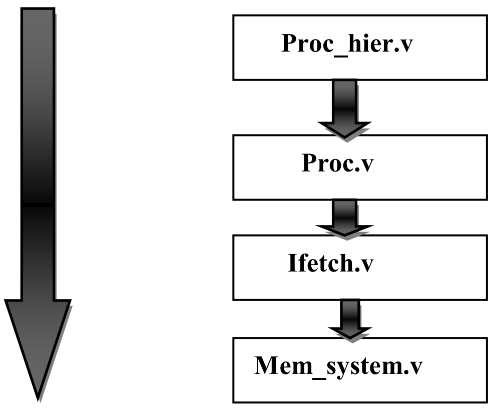


Figure 1: Processor hierarchy for in_micro signals.

5.3 Out_micro

These are the signals which are used to monitor the initialization of the instruction memory. These signals should be carried out from instruction memory module to the top level module. For all practical purposes Out_micro's syntax is identical to In_micro.

```
mem_system  instr_mem(.DataOut(data_out), .Done(ready), .Stall(stall),
.CacheHit(hit),.err(err), .Addr(currentpc_t), .DataIn(data_in_t), .Rd(rd_t),
 .Wr(wr_t), .createdump(1'b0), .clk(clk),
.rst(mem_rst),in_micro(in_micro),out_micro(out_micro));
```

Below is an example which shows how this hierarchy is carried through.

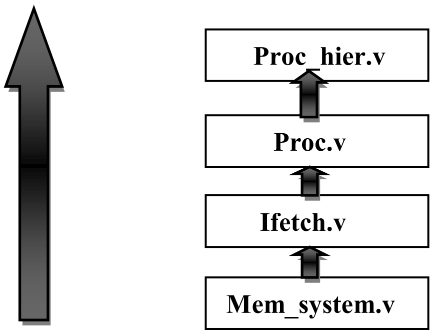


Figure 2: Processor hierarchy for in_micro signals

5.4 Micro_bench

These are the testbench signals which are used to monitor the result of the processor's execution. Table-1 describes these signals. Column 1 in the table describes the actual name of the testbench signal. Column 2 provides a description of the signal, and Column 3 gives the example of the most likely signal in the design that students can hook to the corresponding column 1 signal.

Table 1: Microbench signals

Signal name	Description	Example
micro_bench[0:15]	16bit data coming out of the data memory	Memout
micro_bench[31:16]	16bit data going into the data memory	Memdatain
micro_bench[47:32]	16bit data memory address.	Memaddress
micro_bench[48]	Data memory write signal	Memwrite
micro_bench[49]	Data memory read signal	Memread
micro_bench[81:66]	PC address	Currentpc
micro_bench[65:50]	Instruction vlaue	Instr
micro_bench[97:82]	Register 16 bit data signal.which is to be written into the register.	Regwritedata
micro_bench[100:98]	Register select signal	Regwriteregsel
micro_bench[101]	Register write signal	Regwrite
micro_bench[102]	Ready singal from instruction memory	Dataready
micro_bench[102]	Halt signal	Halt

Chapter 6 **Processor FPGA synthesis**

This chapter describes synthesis steps for the processor design. The output of this step is a bit file which you can upload and run on the FPGA using the web interface. Specifically it provides the details about the synthesis scripts and steps involved in running these scripts. Section1 describes files which need to be provided for synthesis. Section 2 describes the steps to run the scripts.

6.1 Files

To start the synthesis for the processor design we will make one folder on our local machine (for e.g. emperor01) and put our verilog files in that folder. While putting verilog files into this folder we need to take care that no unnecessary files are placed in this directory. Specifically all the files in the folder should be synthesizable. In addition, **we must NOT have the following files in this folder.**

1. **dff.v**
2. **clkrst.v**
3. **cache.v**
4. **four_bank_mem.v**
5. **memory4c.v**
6. **proc_hier_pbench.v**

These files are automatically added to our design by the script.

Download the workspace folder from the **/p/vertical/projects/fpga-wiscsp08/tools/workspace/proc** to our local home directory. Place this directory at the same level as your verilog_code directory. For example:

```
home/
    cs552/
         proc/
              verilog_code
              workspace
```

This workspace folder contains files used for synthesis. These files contain the configuration details of FPGA pin, I/O, Xilinx inbuilt controller (i.e. DDR2, Ethernet Mac), microblaze port settings. All these files are required to run our design in the FPGA. There are several workspace files corresponding to different clock frequency. Start with the lowest frequency before trying higher frequencies.

6.2 Running the script

Type the following the command to run the script

```
prompt> proc_script  path1 path2
```

Here path1 is the path of our folder where we have our verilog files. Here path2 is the path of the workspace folder where we have our workspace folder. Here is an example which shows how to run the command.

```
/p/vertical/projects/fpga-wisc08/tools/scripts/proc_script
~foobar/cs552/proc/verilog_code ~foobar/cs552/proc/workspace/
```

6.3 Cookbook

1. Make one folder on our local machine which contains the verilog files.
2. Remove all the garbage verilog files from this folder that we are not using in our design.
3. Remove the above mention 5 files (diff.v, cache.v etc) from this folder.
4. Modify the design according to the design constraints as mention in the processor design help document.
5. For 125 MHz design download the workspace folder from the /afs/cs.wisc.edu/p/vertical/projects/fpga-wiscsp08/tools/workspace/proc125 to our local home directory.
6. For 75 MHz design download the workspace folder from the /afs/cs.wisc.edu/p/vertical/projects/fpga-wiscsp08/tools/workspace/proc75 to our local home directory.
7. Run the command `proc_script path1 parth2`.
8. Here path1 is the path of our verilog files folder and path2 is the path of our workspace folder.

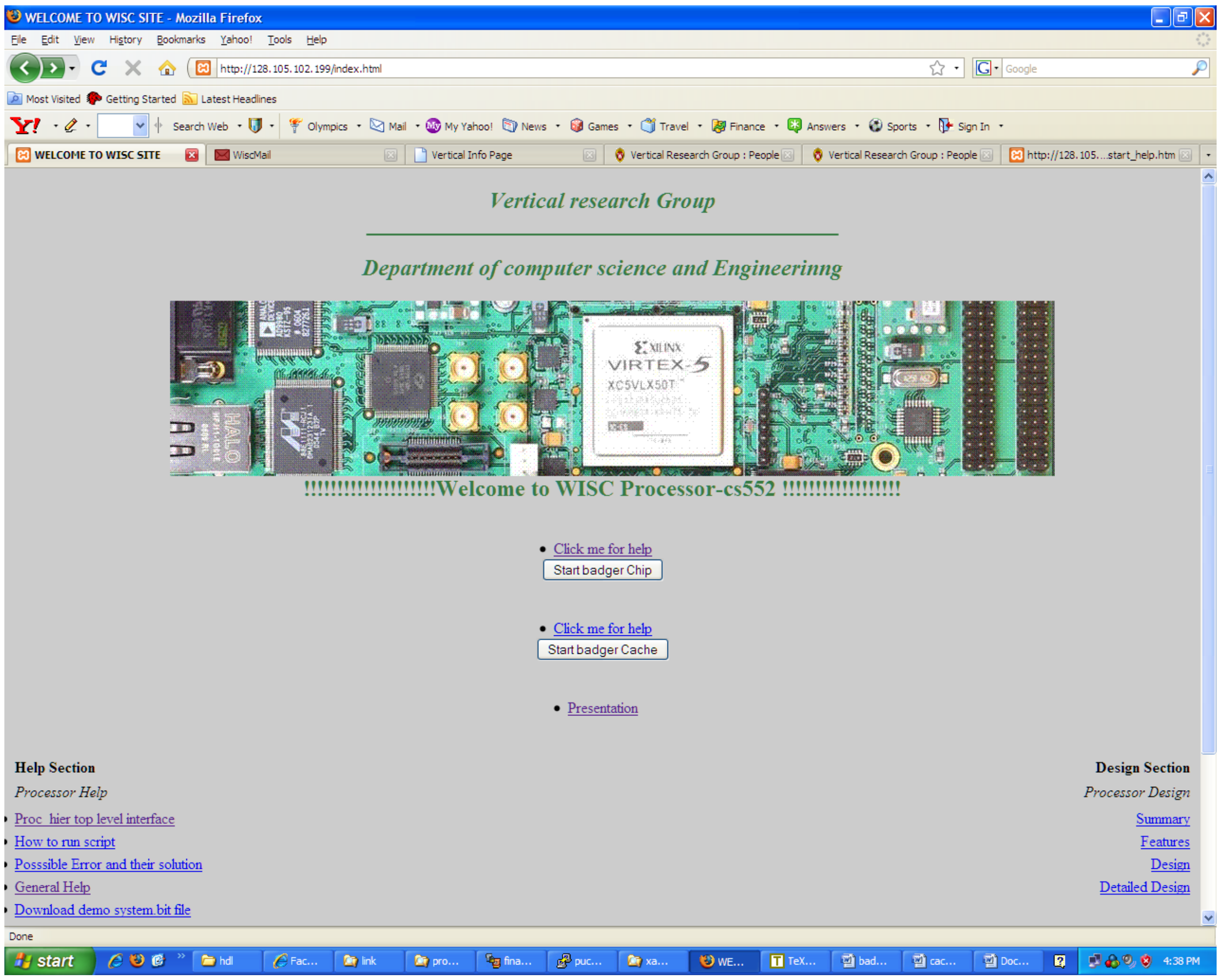
Chapter 7 **Processor FPGA web-portal execution**

This chapter provides general instructions which help the students to run their design on the FPGA via web interface. We discuss the web interface pages with screens shots and step by step instruction. The main steps are:

1. Acquire lock on the FPGA

All the screen shots are displayed in sequence.

Step1: http://www.helena.cs.wisc.edu/index.html
This is the homepage for the badger chip. To start the badger processor click on the start badger chip button. To start the badger cache design we need to click on badger cache button. For help click on the corresponding help link.


WELCOME TO WISC SITE - Mozilla Firefox
File Edit View History Bookmarks Yahoo! Tools Help
http://128.105.102.199/index.html
Most Visited Getting Started Latest Headlines
Search Web Olympics Mail My Yahoo! News Games Travel Finance Answers Sports Sign In
WELCOME TO WISC SITE | WiscMail | Vertical Info Page | Vertical Research Group : People | Vertical Research Group : People | http://128.105...start_help.htm
Start badger Chip
Start badger Cache
Done
start hdl Fac... link pro... fina... puc... xa... WE... TeX... bad... cac... Doc... 4:38 PM

Vertical research Group

Department of computer science and Engineerinng

!!!!!!!!!!!!!!!!!!!!Welcome to WISC Processor-cs552 !!!!!!!!!!!!!!!!

- Click me for help

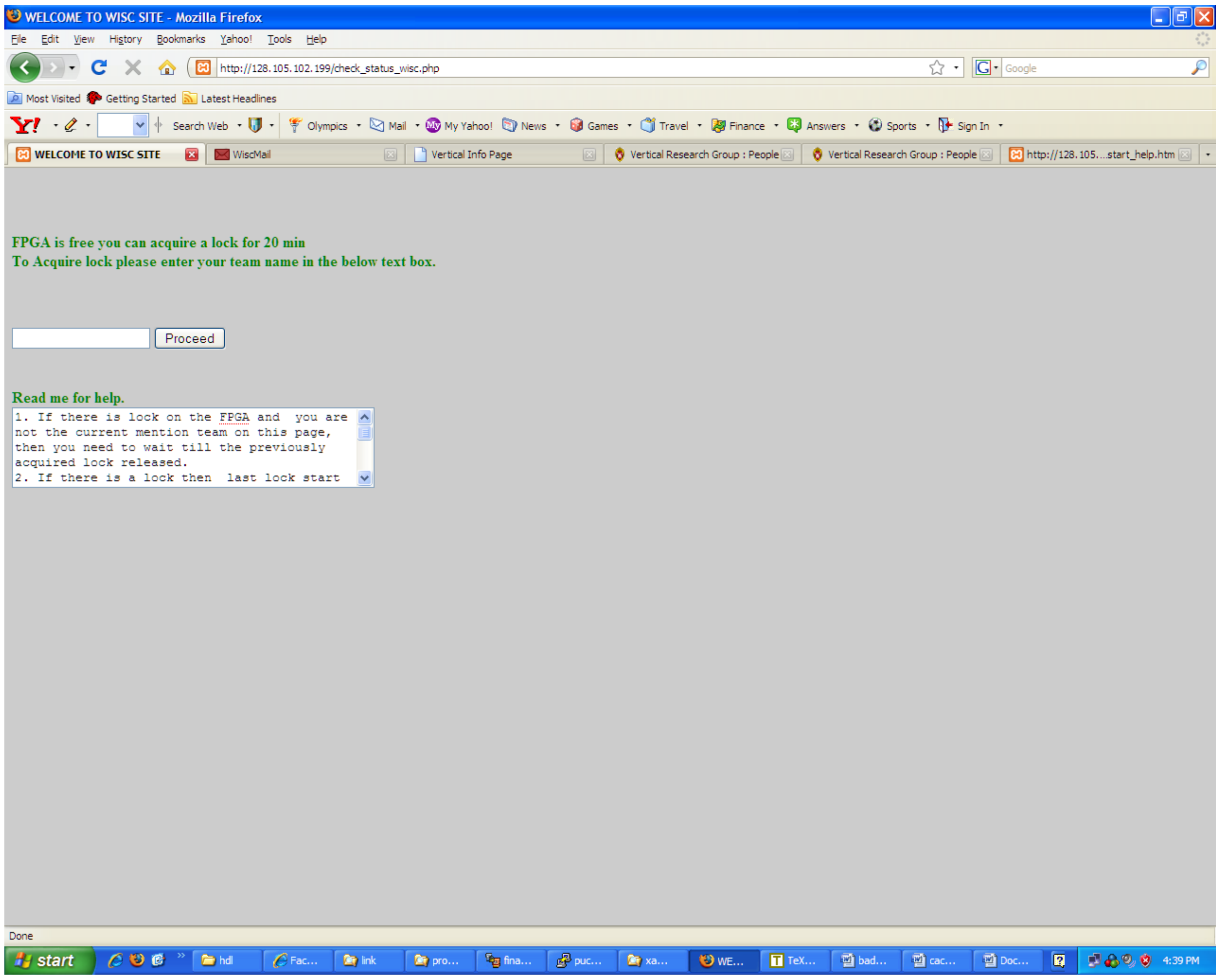
- Click me for help

- Presentation

Help Section
Processor Help
- Proc_hier top level interface
- How to run script
- Posssible Error and their solution
- General Help
- Download demo system.bit file

Design Section
Processor Design
- Summary
- Features
- Design
- Detailed Design

Step 2: This is the badger chip FPGA status page. This page will tell us whether FPGA is locked by some team or not. If FPGA is not locked and is currently free then the page looks like as it is shown in the figure 2. To acquire a lock on FPGA we need to enter our registered cs-552 team name and need to click on the proceed button.

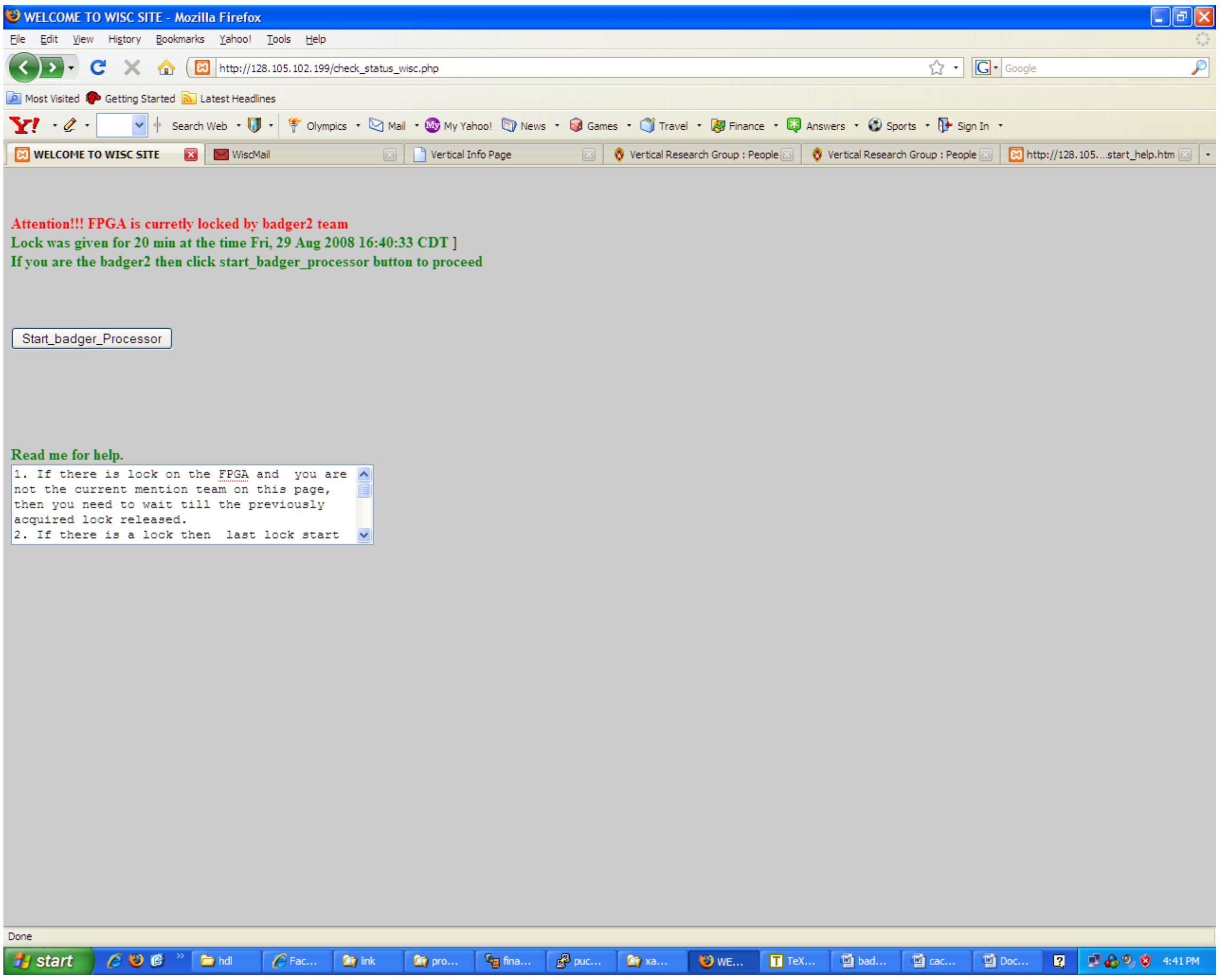

WELCOME TO WISC SITE - Mozilla Firefox
http://128.105.102.199/check_status_wisc.php
Proceed
Done

FPGA is free you can acquire a lock for 20 min
To Acquire lock please enter your team name in the below text box.

Read me for help.

```
1. If there is lock on the FPGA and  you are
not the current mention team on this page,
then you need to wait till the previously
acquired lock released.
2. If there is a lock then  last lock start
```

Step 3: This is the badger chip FPGA status page. This page will tell us whether FPGA is locked by some team or not. If FPGA is locked and is currently used by some team then the page looks like as it is shown in figure 3. It shows the team name that is currently accessing the FPGA, if we are the same team then we can proceed by clicking the start button. If we are not the same team whose name is mention on the page then we need to wait till the last lock will release. Last lock time is mentioned on the page to calculate the lock release time just add the 20 min to the start time.

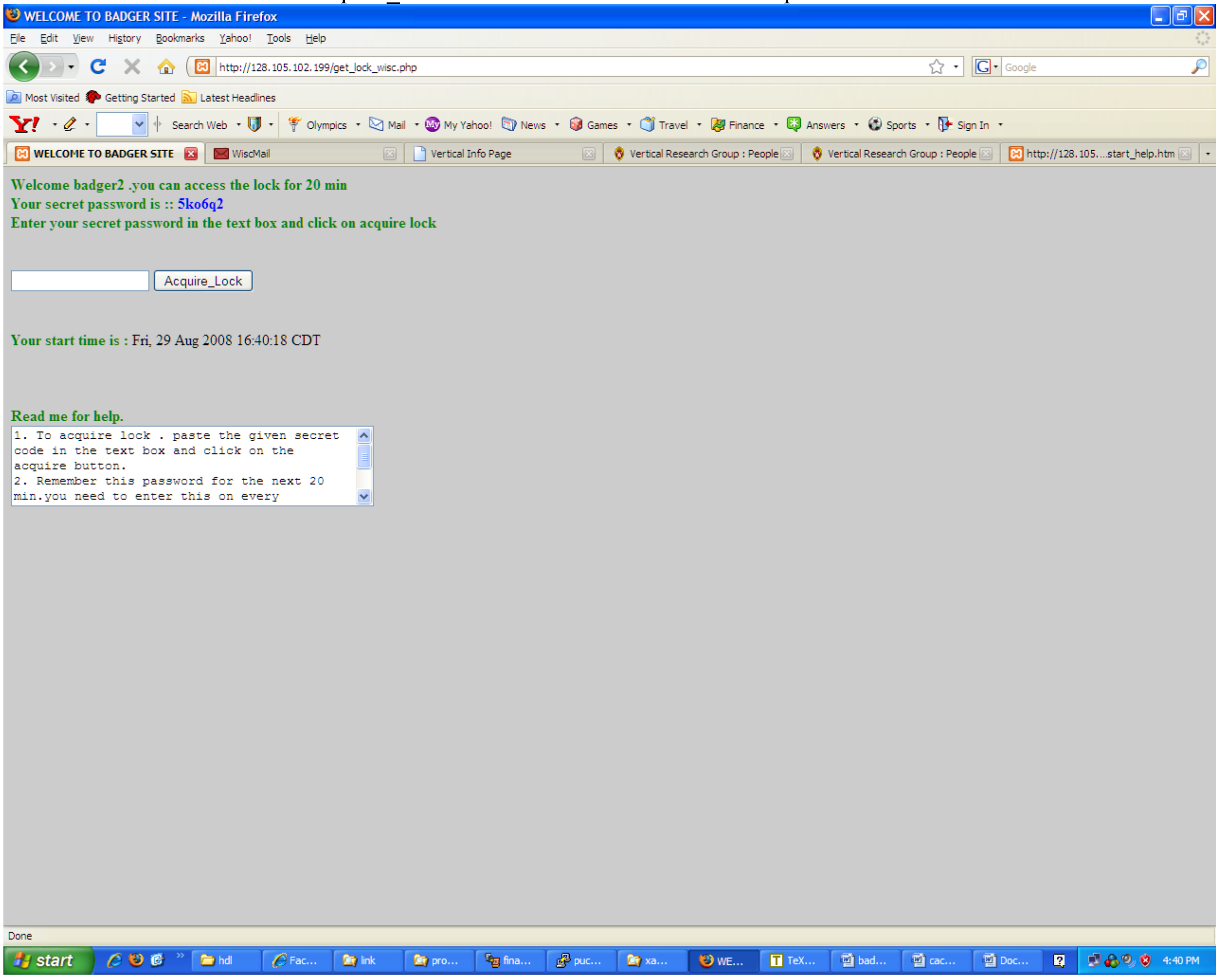

WELCOME TO WISC SITE - Mozilla Firefox
http://128.105.102.199/check_status_wisc.php
Start_badger_Processor
Done
4:41 PM

Attention!!! FPGA is curretly locked by badger2 team
Lock was given for 20 min at the time Fri, 29 Aug 2008 16:40:33 CDT]
If you are the badger2 then click start_badger_processor button to proceed

Read me for help.

```
1. If there is lock on the FPGA and  you are
not the current mention team on this page,
then you need to wait till the previously
acquired lock released.
2. If there is a lock then  last lock start
```

Step 4: This is the lock page .To acquire the lock on the FPGA we need to enter the given password (blue color) in to the text box and need to click the acquire_lock button. We must remember this password for the next 20 minute.



WELCOME TO BADGER SITE - Mozilla Firefox

http://128.105.102.199/get_lock_wisc.php

Welcome badger2 .you can access the lock for 20 min
Your secret password is :: 5ko6q2
Enter your secret password in the text box and click on acquire lock

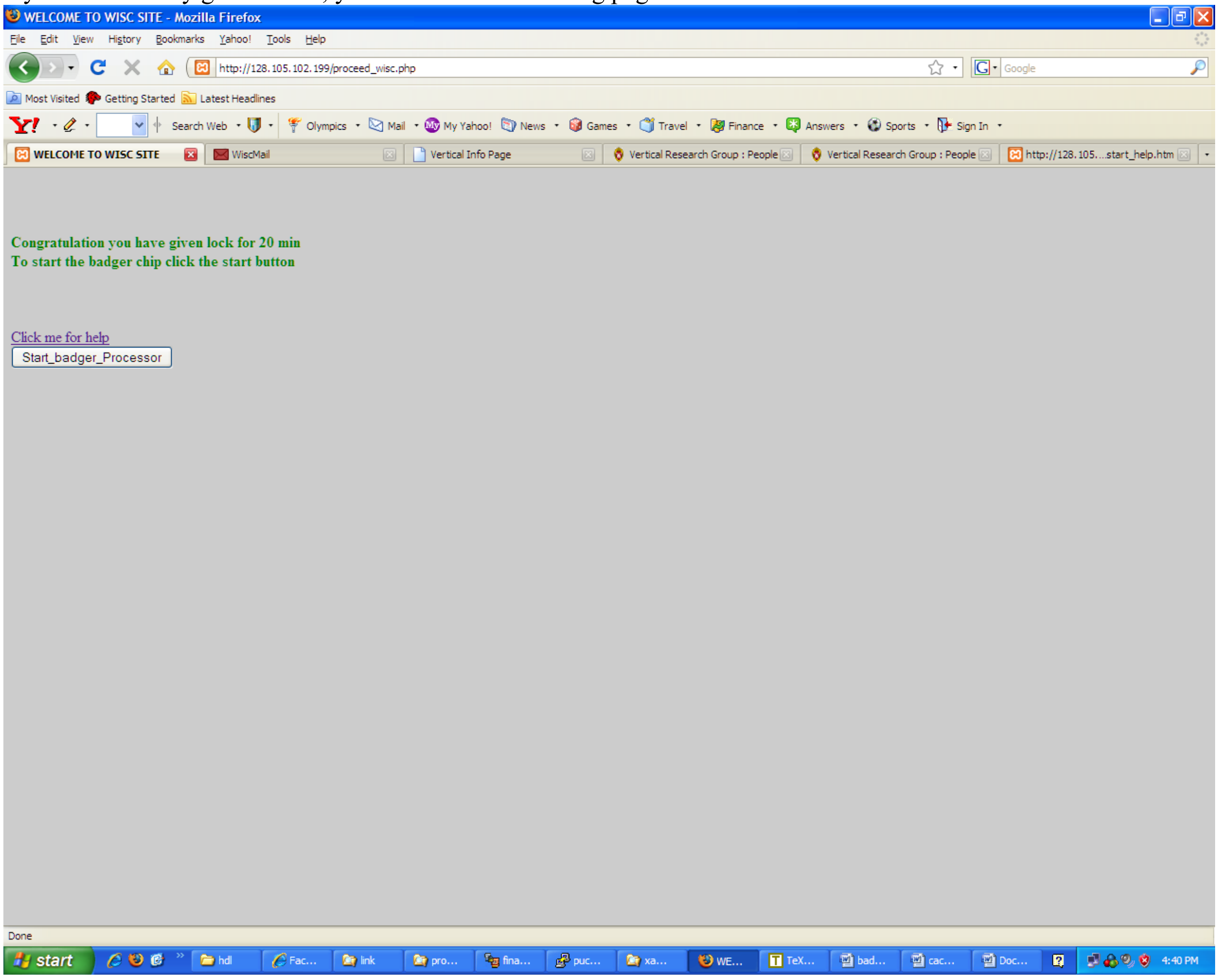
Acquire_Lock

Your start time is : Fri, 29 Aug 2008 16:40:18 CDT

Read me for help.

```
1. To acquire lock . paste the given secret
code in the text box and click on the
acquire button.
2. Remember this password for the next 20
min.you need to enter this on every
```

If you successfully get the lock, you will see the following page.

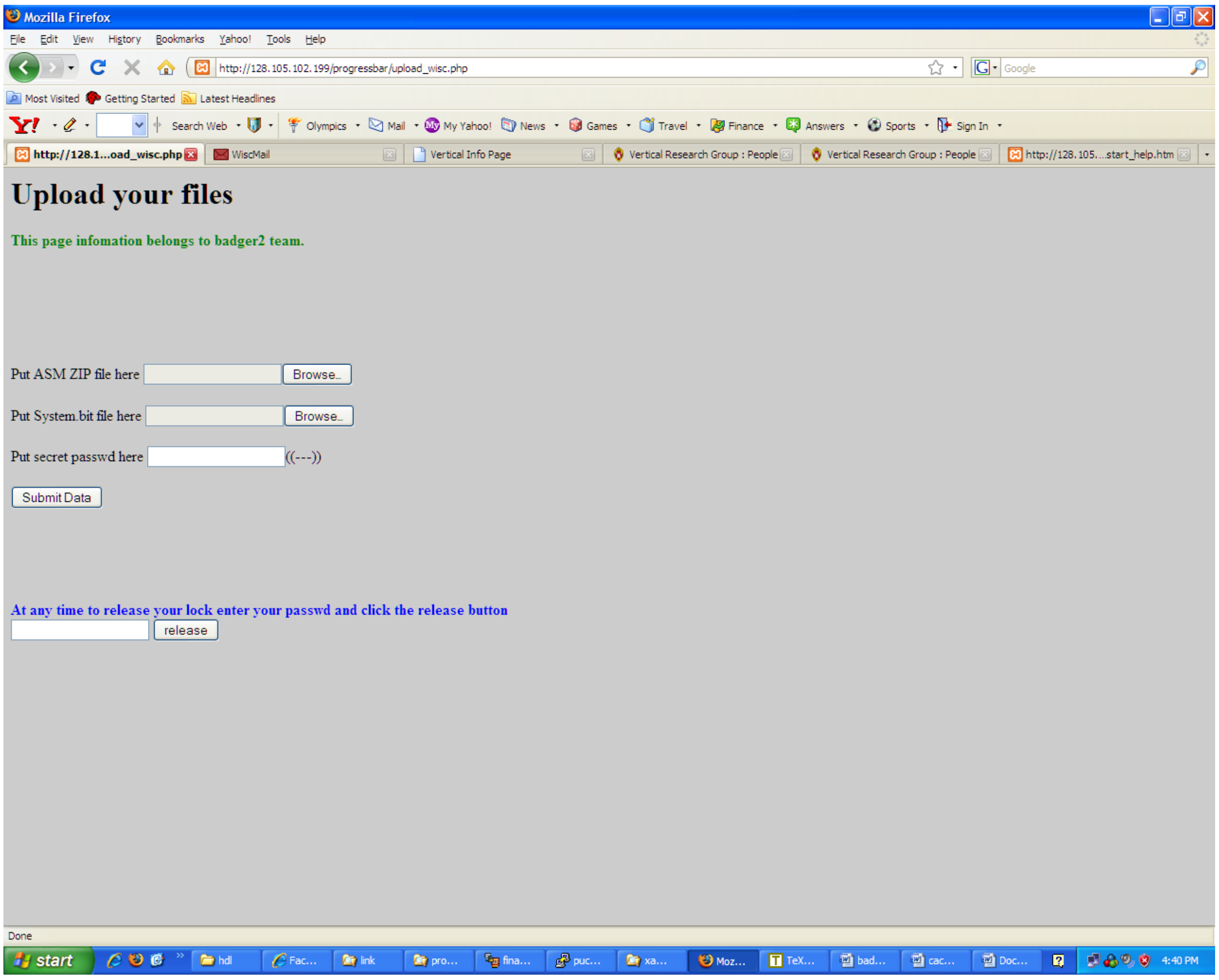

WELCOME TO WISC SITE - Mozilla Firefox
http://128.105.102.199/proceed_wisc.php
WELCOME TO WISC SITE

Congratulation you have given lock for 20 min
To start the badger chip click the start button

Click me for help

Start_badger_Processor

Step 6: This is the upload page. Here we need to put our system.bit file which we get from cache_script or proc_script download.zip file which we get from the asm_script, and in the last textbox we need to enter our secret password. At any time if we want to release the lock we need to enter our secret password into the last text box and click on the release button.



Upload your files

This page infomation belongs to badger2 team.

Put ASM ZIP file here [Browse...]

Put System.bit file here [Browse...]

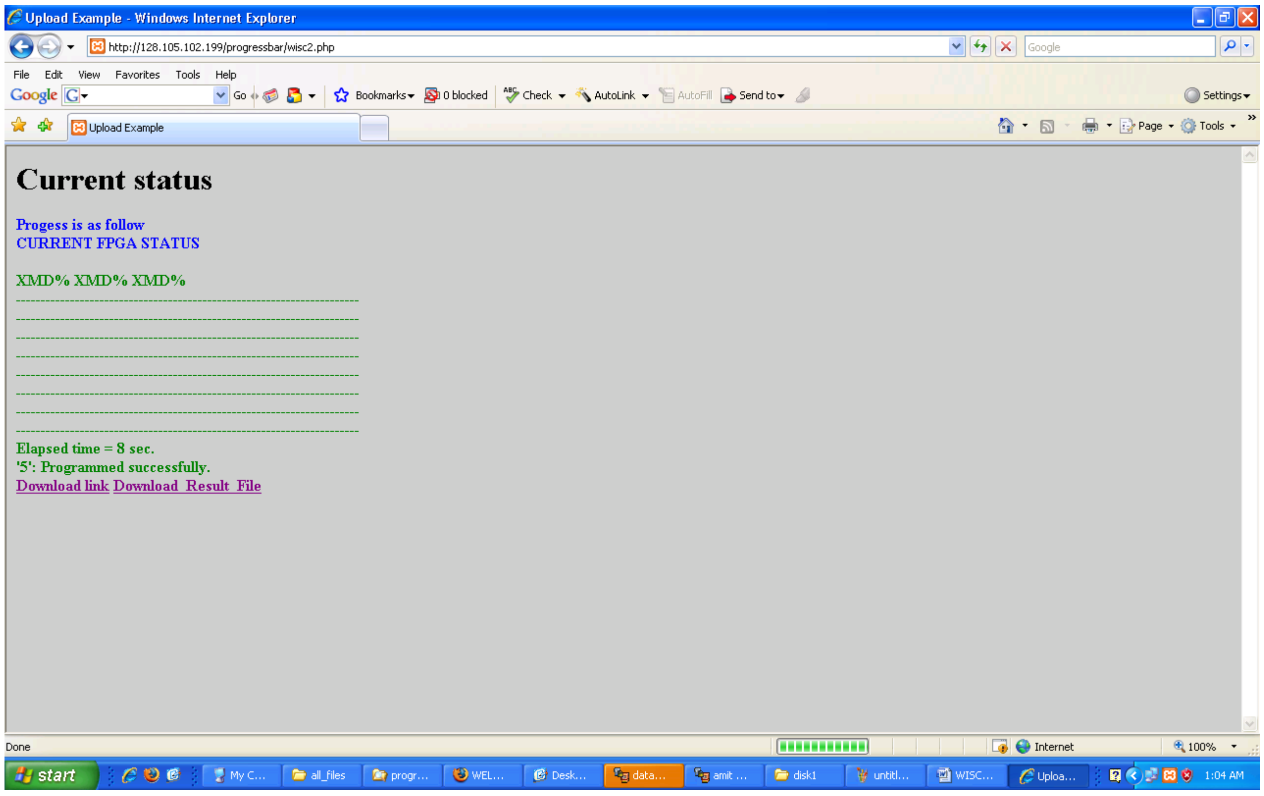
Put secret passwd here ((---))

[Submit Data]

At any time to release your lock enter your passwd and click the release button

[release]

Step 7: This is the FPGA status page. Text in the red will shows the ERROR, text in the blue shows the highlights, text in the green shows the current status. After completion of the projects its will ask us to download the result. To download the result we need to click on the download button.


Upload Example - Windows Internet Explorer
http://128.105.102.199/progressbar/wisc2.php
File Edit View Favorites Tools Help
Upload Example
Done
Internet
100%
1:04 AM

Current status

Progess is as follow
CURRENT FPGA STATUS

XMD% XMD% XMD%

Elapsed time = 8 sec.
'5': Programmed successfully.
Download link Download_Result_File

Chapter 8 Coding problems

8.1 State machine coding

Rule 1: All “next state” logic should be binary conjugative while coding state machine logic.

Good state encoding	Bad state encoding

`State1- 5’d0`
`State2- 5’d1`
`State3- 5’d2`
	`State1- 5’d0`
	`State2- 5’d22`
	`State3- 5’d11`

Rule 2: All “next state” logic should have a default value failing to which it will cause an error in the timing driven part in the synthesis. This is because if we don’t specify the default state logic, some combinational circuit is created in the hardware to maintain the logic when there is undefined state and the logic will cause timing error constraints in synthesis.

Failing to obey the above rules will result in the following error:

```
FATAL ERROR:- see the <module name>.xst.wrapper file for details.
```

When you open the wrapper file go to the last line. There will be something like: ../…./ FSM optimization unreached state. Timing driven phase fails. For details open the <module name / par files>.

How to debug: Open the project in the ISE and run the timing constrains rule it will show the worst case delay and the path in the routed FPGA cells. Go to the Verilog code and see the path.

8.2 Interfacing to port

Rule3: Whenever working on the input port of a module, you should not work directly on that port, take a wire from that port and then operate on that wire.

Good

```
module adder (input a,b output reg c);
   wire temp_a = a;
   wire temp_b = b;
   wire   temp_c = temp_a+temp_b;
   always@(*)
   begin
      c< = temp_c;
   end
```

Bad

```
module adder (input a,b output reg c);
  always@(*)
  begin
      c< = a+b;
  end
```

Rule 4: For the output ports we should not operate on them directly. Always operate on the temp wire or reg and then assign temp wire or reg value to the output port.

Good coding

```
module adder (input a,b output reg c);
  wire temp_a = a;
  wire temp_b = b;
  wire   temp_c = temp_a+temp_b;
  always@(*)
  begin
     c< = temp_c;
  end
```

Bad coding

```
module adder (input a,b output reg c);
  always@(*)
  begin
    c< = a+b;
  end
```

Failing to the above rule will case failure in the routing of the FPGA cells and timing driven problem.
Error: 713 timing driven phase fails.
Error: Par phase fails due to the one of the following reason

8.3 Multiple sources on single variable

Rule 5: You should not operate on a single variable on more than one behaviour

Good

```
always@(posedge clk)
       begin
       if(time==1'b0)
              temp_a<=temp_a+temp_b;
       else
       case(state)
       1'b1:
              temp_a<= 1'b1;
       1'b2:
              temp_a<= 1'b2;
       default:
              temp_a<= 1'b3;
       endcase
end
```

Bad

```
always@(posedge clk)
       if(time==1'b0)
              temp_a<=temp_a+temp_b;
       always@(state)
       begin
              case(state)
              1'b1:
                     temp_a<= 1'b1;
              1'b2:
                     temp_a<= 1'b2;
              default:
                     temp_a<= 1'b3;
       endcase
end
```

ERROR: - failure to obey the above rules will give the error stating multiple variables on the following variable.

8.4 Recursive module

Rule 6: We should not have a module call the same module in its body, This will cause recursive call in the hardware.

8.5 Wide buses

Rule 7: We should not use wire with width like 180 bits, those may cause routing problem and slow down the design speed too much. Failure to obey the above rule may give an error in routing phase of the synthesis.
Suggestion: Instead of using bunch of 180 wires use bunch of 32 wires at a time.

8.6 Define and referenced in single clock cycle

Rule 8: We should not define and reference a variable in a single cycle This will cause to instantiate flip flop. Consider the code below:

```
always@(posedge clk)
begin
       c<=a+b;
       d<=c?1'b1:1'b0;
end
```

Here the value of "c" updates in the same cycle and is referenced in the same cycle. So it will always fail in the hardware if we want the updated output in the same cycle. As the adder takes some time to calculate the value of "c" by the time set up time of the flip-flop D can't be meet, so it fails and gives the value from the previous cycle.

Chapter 9 Design and Implementation

9.1 Introduction

This chapter describes the details of how to map a 16-bit pipelined RISC processor to an FPGA. This processor is specifically designed for course work in CS552 – a senior undergraduate course in computer architecture that covers basic processor architecture. The purpose of this exercise of mapping the chip to an FPGA is to give the students an appreciation of the physical hardware design. To abstract away several details of this process, we automated many steps of FPGA synthesis, mapping, and verification and created a simple web interface. In this document, we describe the entire implementation of the different tool chain components required to accomplish this. We first provide a high level overview of the entire process and then discuss in detail, synthesis, various synthesis steps involved, FPGA execution and verification strategies. Finally we describe the simple web interface.

The remainder of this chapter is organized as follows. Section 9.2 describes the overview of the badger chip and the basic features of our FPGA and development board. Section 9.3 describes the implementation overview of the complete design. Section 9.4 describes synthesis, Section 9.5 describes the FPGA loading and execution details, and Section 9.6 describes verification. Section 9.7 describes the simple web interface.

9.2 Badger Chip

Architecture: The Badger processor is a 16-bit processor based on Von Neumen architecture. This is a 5-stage pipelined processor with a single level cache. Separate instruction and data caches are 2-way set associative. It uses a simple 16-bit RISC ISA with 40 integer only instructions, eight 16 bit registers, and simple branch instructions. Detailed specification is found on the course web page (http://pages.cs.wisc.edu/~karu/courses/cs552/spring2008/wiki/). The figure below shows the basic pipeline that is implemented.

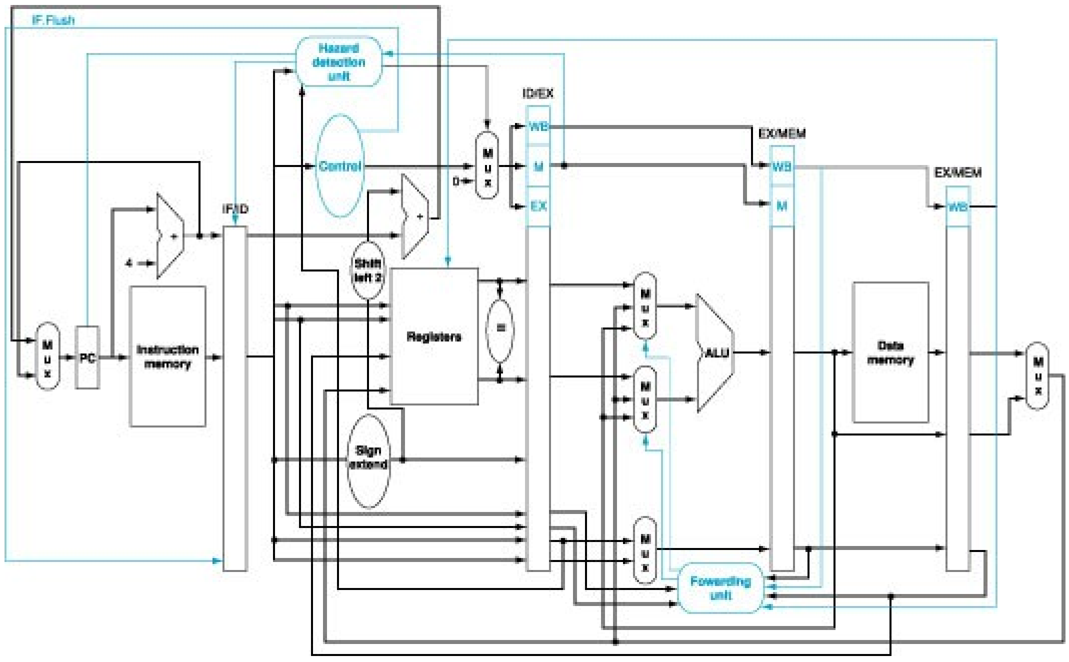


Figure 9-1: Processor pipeline

FPGA: This processor is mapped to a Xilinx Virtex-5 (xcvlx110T) FPGA mounted on an ML505 board. ML505 is a Xilinx advanced development board with onboard Virtex-5 FPGA, DDR2, SRAM memory and Ethernet port. The onboard Virtex-5 FPGA is based on 65-nm copper CMOS process technology that operates at 1 volt. It is based on dual 6 LUT technologies and has powerful clocked management tile (CMT). It also has advanced DSP slices and high performance SelectIO technology. The datasheets provide more information on the FPGA. We chose this board and FPGA as it supported Xilinx's largest FPGAs and thus the board could be used to map larger designs that are beyond the scope of this exercise. This FPGA has an on-chip 32-bit soft core processor called Microblaze. Microbalze is programmable via C language program and can be used to control all the peripherals attached on the board like DDR, Ethernet port etc.

9.3 Implementation Overview

This section provides an overview of the complete design implementation. We first describe the HDL coding of the design, then outline implementation on the FPGA, and finally describe how the design is loaded to the FPGA and executed through the web interface.

HDL coding: The processor design is implemented using Verilog and functionality is verified through ModelSim or Synopsys vcs. The design is incrementally completed by first implementing and verifying the separate component, then building a single cycle implementation, and finally pipelining the implementation. This final design is verified with a processor level testbench that loads random programs on to the processor and extracts a trace of program execution by monitoring specific signals in the design. This trace is then verified against a reference trace obtained from the architecture simulator called wiscalculator (this name will change to badgerchip_archsim).

FPGA implementation: Badger chip design is synthesized using the XST (Xilinx synthesis tool). Constraints are added to the HDL code to meet the synthesis requirement. Xilinx ISE tool is used for place and route and obtain a *bitfile*. This bitfile is used to configure the FPGA logic cells to build the hardware design. After configuring the FPGA, programs are loaded into the FPGA to test the Badger processor. A synthesizable testbench, similar to the processor testbench used for HDL verification, is developed and mapped to the FPGA along with the processor design. Verification is done by comparing the trace from FPGA execution to the wiscalculator trace . Several problems encountered during the design and verification processor are enumerated in section 6. We have used the Microblaze for assisting in the verification and testing of the Badger chip design. We have used the Microblaze for initialization of the processor instruction memory and monitoring the running state of the processor. Microblaze is also used to transfer the result data from FPGA to the host PC machine by starting TFTP server.

Web Interface: A web interface is created to provide students remote access to the FPGA and abstract away many of the details of FPGA loading and executing. This web interface can configure the remote FPGA, load the assembled program to execute, execute them and report back the output trace to the user. The web interface was developed using php scripts and using Apache web server running on Windows. The details of this implementation are described in Section 6.

9.4 Synthesis

9.4.1 Overview

Synthesis is done using the Xilinx tool chain. Figure 2 shows the high level overview of the synthesis flow starting from the HDL code to the bitfile generation. We have developed some predetermined constraints and parameters that are designed once for the synthesis. The constraints include FPGA pin configurations, design timing constraints, DDR interface signals, TFTP server drivers settings, microbalze signals settings, and other I/O interface standards. Students are not expected to optimize or modify these constraints. Technology libraries contain predefined Xilinx IP modules like DDR controller, Ethernet controller, and debug module. Those are provided by Xilinx and are part of the synthesis flow. They are added to the design to assist in testing and monitoring other aspects of the design flow.

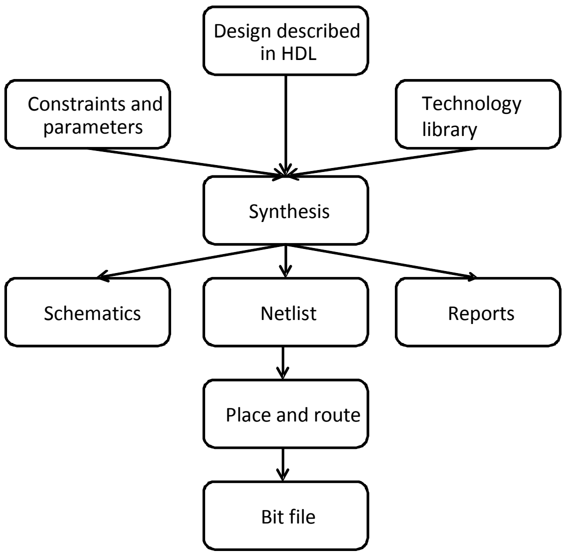


Figure 2: Synthesis overview

For the Badger processor synthesis involves three types of tasks: 1) synthesis of random logic in the datapath and control path of the processor, 2) cache synthesis of the instruction and data caches which included mapping of the tag and data arrays to BRAM blocks in the FPGA, and 3) memory synthesis which again included mapping of the 64K memory banks to the BRAM blocks in the FPGA. We first describe the mapping of these tasks and then describe the details of how the EDK tool was used to automate the entire synthesis flow.

Random-logic Synthesis: The processor's logic is mapped to the LUTs on the FPGA. Some components in the datapath need to communicate off-chip to components like the DDR and Ethernet. For example we monitor the badger processor running status and store the result in on chip buffer. Now after every programs run of the badger processor we need to empty the buffer by putting its content on the DDR. This communication with the outside world is through the PLB (Processor local bus). Plb4.6 master bus configuration is used to tunnel the user logic to the bus and to external environment like DDR and Ethernet. Xilinx IPIF module is used as a tunnel for the whole user logic at one end and the IBM PLB bus at the other end. This IPIF module is configured with DMA and user logic memory space so that microblaze can communicate to the badger chip at any time via the tunnel. Most of the badger logic cell are designed with our own HDL code rather than the Xilinx readymade code like for Alu, register file, multiplier etc.

Cache Synthesis: The processor's cache is 2-way set associate and includes separate instruction and data cache. Each cache is synthesized identically by using the BRAM blocks available on the FPGA. Both the tag and data arrays are mapped to BRAM, and each block is single ported with unregistered output to reduce the access time. All data is initialized to 0. The specific block address of the BRAM the caches are mapped to is pre-determined, and is used during initialization. The D2M(data to memory) of the Xilinx system performs this initialization and is part of loading the design to the FPGA. The cache is connected to the memory and uses burst transfer of 4 words at a time.ILA(intergrated logic analyser) and ICON(integrated core generator) debug module are added to the cache design to check the cache state at any given time. The cache is visible to the Microblaze as well.

Memory Synthesis: The processor main memory of 64 Kbytes is also mapped to BRAM blocks in the FPGA. The BRAM blocks are relatively small, typically around 16KB. To generate large blocks of memory, we need to connect various small blocks appropriately. This work of interconnecting small memory block that results into the large memory block is done by the tool COREGEN. RAMB 36 (block of memory on the FPGA fabric) block are used for both the data memory as well as instruction memory.

The mapping of memory to the FPGA is identical to how the cache is mapped. Since the memory is provided as a pre-designed module in the class project, for FPGA synthesis a completely synthesized version of the memory is provided as an NGC file. All the blocks are initialized to a default value of 0. Block address is being tracked so that main memory can be initialized by partial bit stream as well using D2M of Xilinx system. Main memory is directly accessible via the cache. All the Bram blocks are single ported with unregistered output to reduce the latency of the memory. Direct NGC files of the memory module are included in the design to reduce the final synthesis time.

XILINX provides a number of tools for synthesis. There are two primary tools: ISE (Integrated software environment) and EDK (Embedded development kit). ISE is generally used for the simple design and preferred by beginners. EDK is used for the complex designs which use several peripherals. For our design we used EDK because we need to interface with the DDR and Ethernet controller. We first describe the different steps in the synthesis process and different IP modules used. Section 9.4.9 describes our final synthesis scripts that automate the entire process.

9.4.2 Base System Builder Project

Xilinx provides a base system builder (BSB) project that can be used for most designs that use the peripherals on the FPGA board. We changed the original FPGA and the board and hence the default definitions in the BSB do not work. We instead configured each IP block manually and handled the pin configuration, timing constraints and location constraints for each of the Xilinx controllers. We created a simple BSB project that includes the following IP modules:

- DDR module.
- Ethernet Mac for Ethernet interface.
- RS232 for serial data transfer.
- MDM for debugging.
- ILA for internal signal monitoring.
- Wisc_proc (the user logic design for Badger processor).
- LMB (local memory bus) block for acting as Bram.
- LMB controller for interfacing the Bram to PLB.
- Clock Generator for getting various clock frequencies.

A number of changes are made in the configuration of the DDR module design to work with the modified FPGA on the ML505 board. The include the timing constraints and location constraints. We took the DDR module's timing constraints from the MIG (memory interface generator) generator for the xcvlx110t FPGA. Using a Perl script we modify them according to the MPMC(multiport multichannel memory controller) constraints. We modified the DDR module's location constraints to the x1y1-x2y2-x6y6 to match the xcvlx110T FPGA. We implement similar changes for the Ethernet MAC controller. We inserted ILA modules in various place to make the debugging easy.

9.4.3 User logic peripheral

User logic is the top level interface module below which all the logic for the processor resides. We created a User logic peripheral using the "create and import" peripheral. We have used plb 4.6 master-slave configuration and IPIF(intellectual property module) module with the bi-directional configuration buffer to act as a DMA(direct memory access) as well. We had made IPIF a DMA that help us in transferring the data from badger processor to offchip DDR. The user logic also hase some memory mapped I/O that can be controlled by the software driver and accessible to every other master peripheral attached to the PLB bus. It also has some software accessible registers which be used to determine the state of the processor and help in debugging. For details refer to the user logic IP of the module.

9.4.4 Ethernet Mac

We used Ethernet to transfer the result data from FPGA to the host PC. Xilinx Ethernet controller is used for the Ethernet and predefined software drivers are used to implement the UDP protocol. The Ethernet driver is modified according to the current FPGA (xcvlx110t) to include the location constraint and the timing constraints. TX and Rx buffer length is increased to 12kb to ensure the long packets transmission. Ethernet Mac is directly interfaced to the PLB so that it can communicate to the DDR that is also interfaced to the PLB bus. TFTP (trivial file transfer protocol) server is used for the data transfer from DDR2 to the pc host that used no time out mechanism and can operate as fast as 100Mbs.This is the raw mode operation of the TFTP that the Xilinx used. We use Ethernet to transfer trace data generated inside the processor back to the host PC.

9.4.5 DDR2

We use the onboard DDR to store assembly programs and the resulting trace from execution. Xilinx MPMC (multi port multi channel) 4 is used to interface to the on-board 256Mb Micron DDR2 memory. DDR2 runs at 125 MHz but is configurable to run at up to 200MHz. The MPMC interfaces directly to the PLB bus and thus the DDR is visible to the Ethernet MAC, Badger Chip, and Microblaze. We build a filesystem on the DDR and each program that needs to run on the FPGA is saved as a separate file.

9.4.6 MDM (Microblaze debug module)

Microblaze debug module is used for debugging the design. The MDM is also attached to the PLB bus and can thus communicate with the processor which is also attached to the PLB bus. The MDM reset signal is attached to the system reset so that system can be reset by the MDM module

9.4.7 RS232

RS232 connection is used to take the FPGA status to the host machine. The serial port is attached to the PLB bus, communicates at 11500 Baud rate, with no parity, and using hardware flow control.

9.4.8 Clock generator

The on-chip Xilinx clock generator is used to meet the different clock requirements of our design. The clock generator has four clock output ports. The configuration of these four output port are as follows.

- Clk0- 65.5 MHz going to DDR2
- Clk1- 125 MHz going to PLB
- Clk2- 200MHz going to DDR2
- Clk3- 100 MHz going to Microblaze.

For the details configuration of the IP modules refer to the following files.

- MHS (master hardware specification) file.
- MSS (master software specification) file.
- System UCF (pin configuration file) file.
- System assembly view (FPGA port configuration) file.

Refer to the http://www.xilinx.com/support/documentation/sw_manuals/edk92i_ctt.pdf for the explanation of the above files.

9.4.9 Synthesis example

Below is an example showing the resource utilized by one design implementation on the FPGA. This shows the utilization in terms of number of memory block used, no of logic cell slides used, no of DSP slides used etc.

```
Device Utilization:
   Number of BSCANs                          1 out of 4       25%
   Number of BUFGs                          10 out of 32      31%
   Number of BUFIOs                          8 out of 80      10%
   Number of DSP48Es                         3 out of 64       4%
   Number of IDELAYCTRLs                     5 out of 22      22%
   Number of LOCed IDELAYCTRLs               3 out of 5       60%

   Number of ILOGICs                        84 out of 800     10%
   Number of External IOBs                 163 out of 640     25%
      Number of LOCed IOBs                 163 out of 163    100%

   Number of IODELAYs                       91 out of 800     11%
   Number of OLOGICs                       137 out of 800     17%
   Number of PLL_ADVs                        1 out of 6       16%
   Number of RAMB18X2s                      15 out of 148     10%
   Number of RAMB18X2SDPs                    2 out of 148      1%
   Number of RAMB36_EXPs                    85 out of 148     57%
   Number of TEMACs                          1 out of 2       50%
   Number of Slice Registers             11073 out of 69120   16%
   Number used as Flip Flops             11069
      Number used as Latches                 1
      Number used as LatchThrus              3
   Number of Slice LUTS                  11754 out of 69120   17%
   Number of Slice LUT-Flip Flop pairs   16681 out of 69120   24%
```

9.4.10 Synthesis scripts

We experimented with the EDK GUI while developing the different project configurations and mappings. After our final solution was developed we automated the entire process with a Makefile which takes the HDL for the processor and calls the Xilinx synthesis tools with the different configuration and constraint files and provides a final bitfile. This Makefile is embedded in the workspace directory tree. Details on how to use the final scripts and their location in the file system are provided in Chapter 4 and Chapter 6.

9.5 FPGA Loading and Execution

This section provides the configuration details of the FPGA. Figure 30 shows the entire design flow. Figure 11 shows the physical mapping of these different steps to the board, FPGA components, and host PC. Blocks shown in green are BRAM memory blocks. Blocks shown in blue are user logic synthesized to the FPGA, and blocks shown in grey are pre-define IP that is synthesized to the FPGA. The white box shows the different programs that are executed on the Microblaze processor. All components on the FPGA primarily communicate with each other through the Processor Local Bus (PLB).

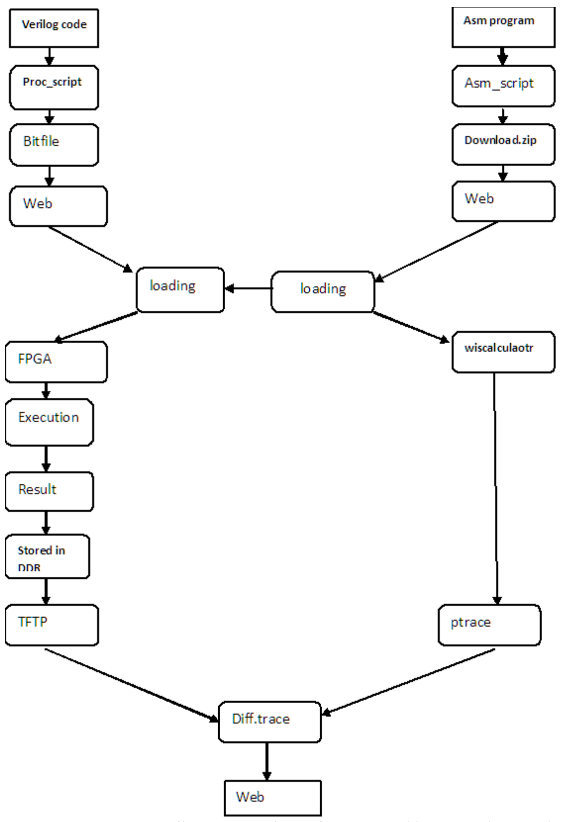


Figure 3: Program flow starting from verilog code to the final result.

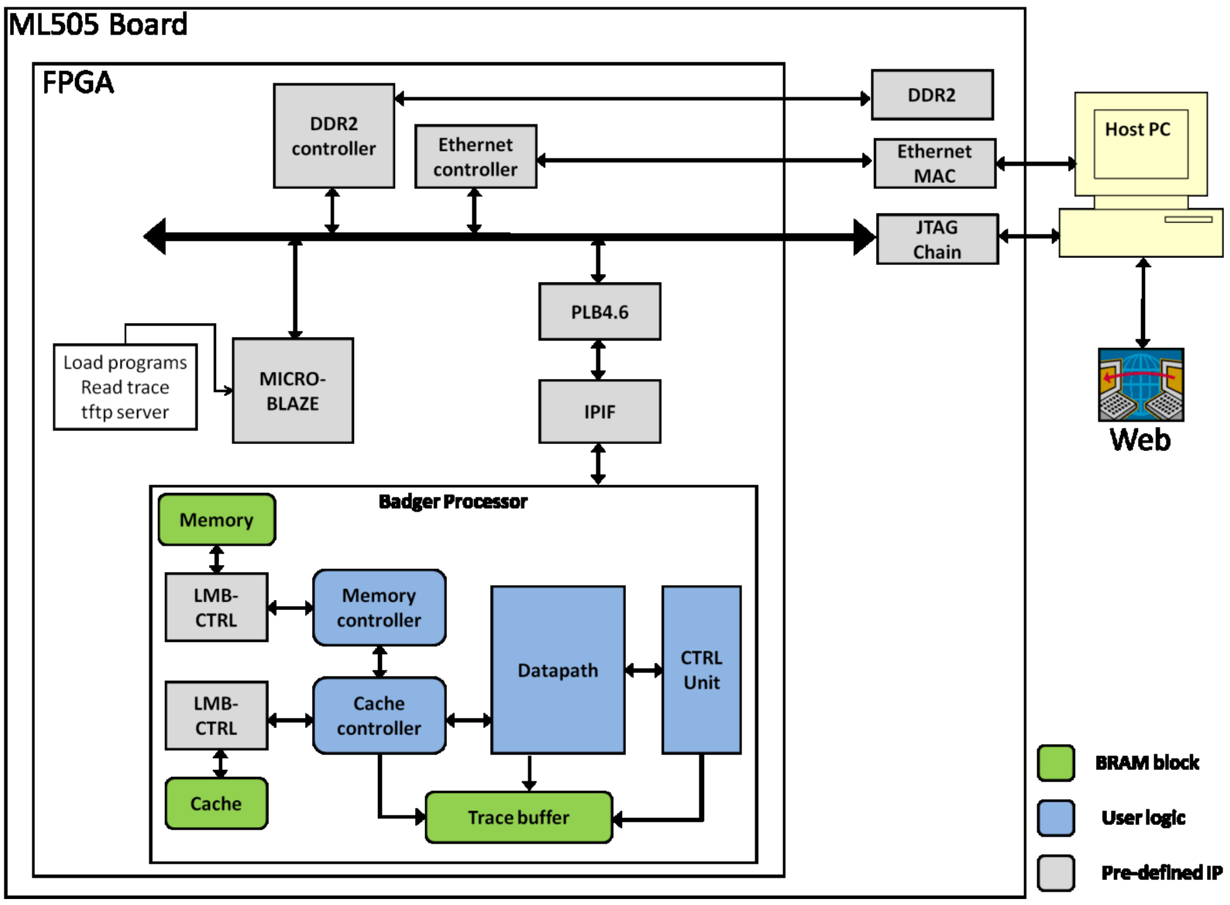


Figure 4: Overview of full system

9.5.1 Loading

The bitfile contains all the information that is used to interconnect the FPGA logic cells to build the design. The bitfile is loaded via the parallel port. The Parallel port is configured for Xilinx parallel cable driver for 5 MHz. This driver has CRC error code detection.

All the board CPLD (complex programmable logic device) device chains are initialized by the Impact tool chain and form the JTAG chain block shown in Figure 11. This JTAG chain can communicate with the processor local bus and interface with the rest of the FPGA components. We configured the Xilinx Impact tool for the parallel serial mode configuration. The bitfile is downloaded to the FPGA via this chain.

Image files are created for the hand-written assembly programs using the assembler. These image file are converted to the Xilinx MFS (Xilinx memory file system) format with the help of mfsgen tool. This tool returns one block file, which is loaded to the on board DDR2 memory via the XMD (Xilinx microblaze debug module) tool chain. Again the JTAG chain through the parallel port is used to transfer the files from the Host PC to the DDR2 on the board.

9.5.2 Program Execution

Execution of the program is triggered by the XMD tool. The status of the FPGA during execution is reported to the local host via the serial port. The entire program flow is as follows:

1. Microblaze loads the program form DDR2 memory to Badger processor's instruction memory. This is achieved by bypassing the processor regular datapath and loading the instruction memory with programs read from DDR through the PLB.
2. Badger processor is triggered to start.
3. Results are stored on the FPGA in the "Trace buffer."
4. The microblaze read's the buffer and stores the result in to the DDR memory.
5. This loop repeats for all programs.
6. When all programs are done, the TFTP server starts executing on the microblaze, reads the result stored into memory and write it to the host PC.

9.5.3 Program verification
All assembly programs to be verified are converted into one zip file by the “asm_script”. This Zip file contains the image file of all the assembly programs and some other files to configure the FPGA memory. This zip file is given to the web interface. The web interface executes scripts on the host PC to invoke the toolchain and downloads the image files of individual programs to FPGA. The processor’s execution is triggered to start by the XMD tool. Program trace results are stored to the DDR and transferred from DDR to the host PC via TFTP server. These results are compared with the results from wiscalculator. A diff file between the traces is created and fed back to the web interface.

9.6 Verification
This section provides the details of the various verification strategies used in our design. In general four methods are used for debugging the design. BFM (Bus functional model) simulation model is used for the post synthesis simulation testing. Modelsim 6.3 SE is used for the simulation environment.

9.6.1 Individual module testing
All individual modules are tested with the automated test bench in the simulation environment. The following individual modules are tested Alu, Register file, Data path, Control unit, Cache design, Memory design, DDR design, Ethernet mac design, and User logic tunnel design

9.6.2 Integrated testing
After the individual module were tested, we integrated them together and tested for the correct functionalities. First for the complete badger processor, then badger processor + microblaze, and finally the whole design is tested using the BFM functional model.

9.6.3 Synthesis debugging
After the design passed the simulation testing phase, we synthesized the design and used the automated synthesisable test bench to test the functionality of the design running in the hardware. We used the following two methods:

- Buffer stored method: All the expected result is being stored in some buffer and then read by the software to verify the functionality.

- Step debugging: Suspected logic of the design is needed to make memory mapped and then recorded by the software during the processor running stage.

9.6.4 Hardware real time debugging
ILA (integrated logic analyzer) module is provided in the design at many logic modules. This helps in the step debugging. The entire signals of cache and memory are hooked up to the ILA module and their regarding trigger signal. We trigger the Trigger signals via JTAG (joint test action group) and look into the hardware design through stepping and debug code in real time environment. For example we triggered the cache data input and output signal at each read/write request of the cache and monitor the cache state at every read/write signal. Cache, Bram block and DDR, Ethernet controller are tested using the chipscope by hooking their all the top-level interface signal to the ILA module.

9.6.5 Software real time debugging
Software testing is done using microblaze and the XMD and GDB(gnu debugger). All the top level design signals are mapped to memory and software driver can access them via MDM. C programs are written to give a virtual flow to the program and some break points are inserted into that so that we can step the software and hardware at any given moment and check the whole state of the design either on XMD, or HyperTerminal or on the chipscope. This testing is used for the memory file system and handling I/O output of the processor.

9.7 Web interface design
The purpose of the Web interface is to remotely configure the FPGA and to run the student programs on it. Apache 2.2 server is installed on the host machine that is connected to the board. Web script is written in php5.2. We created an interface for file upload and download that can take the student bitfiles and assemly programs and run some automation script in the background to configure the FPGA and run programs on the processor, get the results back, and verify the correctness of the result. The web interface also shows the current FPGA status and processor status. Table 2 describes the list of the php script used and their location.

Script name	**Description of the task**
Upload_cache	Use for uploading files.
Target_cache	Use for calling background function
Start_cache	Use for initializing the tool.
Cache	Use for status check.
Cache2	Use for monitoring current status.
Upload_wisc	Use for uploading files.
Target_wisc	Use for calling background function
Start_wisc	Use for initializing the tool.
Wisc	Use for status check.
Wisc2	Use for monitoring current status.
Reset_fpga	Use for reseting the FPGA
Check_status	Use for checking the current lock status
Release_lock	Use for releasing the lock
Get_lock	Use for acquiring a lock on FPGA

Table 2: Web interface scripts

Chapter 10 Tool installation and problems encountered in development

This section describes the list of the tools we used in the design development, the installation procedure for the tools, and problems encountered in developing the toolchain.

10.1 Tool Introduction

XILINX provides a number of tools that makes the synthesis task easy. The two main tools are ISE (integrated software environment) and EDK (Embedded development kit). ISE is generally used for the simple design and beginners prefer it. EDK is used for the complex design in which we one need to talk to the most of peripheral.

For our project we primarily used the EDK tool for building all the peripheral blocks. The ISE environment was used for Prototype Bit stream creation. We used the **Xilinx CORE GENERATOR** for building all memory blocks and their controllers, **Modelsim SE 6.3** for functional verilog simulation, and the **BFM (Bus Functional Model)** for creating the bus functional model for the simulation environment.

In our experience, we realized that like all other tools, the Xilinx tools are not perfect. So when you find a bug, the tool is probably equally guilty for the bug. Check the Xilinx official website and check if this is a tool bug, if yes you will find the patch for that and instructions to use that patch.

10.2 Tool Installation

EDK Installation: We can install EDK web pack form the Xilinx website which is fully functional for the 60 days or you can get the license and order DVD. Xilinx EDK default installation work fine, we don't need to change the directory during the installation else later we need to configure a number of path in the EDK that makes the task harder. By default it resides in C:/xilinx/10.1/EDK. Confirm this after the installation

ISE Installation: We can install ISE web pack form the Xilinx website which is fully functional for the 60 days or we can get the licence and order DVD. Xilinx ISE default installation work fine, we don't need to change the directory during the installation else later you need to configure a number of paths in the EDK that makes the task harder.
By default it resides in C:/xilinx/10.1/ISE. Confirm this after the installation.

BFM Installation: We can download the same from the Xilinx website. For windows double click the exe file and it will automatic install the files to the required directory. Don't specify any path during the installation.

Modelsim SE 6.3c Installation: We can download this from the official website of Modelsim and install it as a default installation. Licence set up is as follows. First, click on the Start button menu the window. Second, click program and go to modelsim and then to license wizard. Finally, run the license wizard and then enter the server address as `1717@frank.cs.wisc.edu.`

Modelsim doesn't have GCC in it by default so we downloaded it and unzipped the downloaded files to the root directory of the Modelsim installation at `C:/ModelSim SE 6.3c.`

10.3 Installation problems

1. Modelsim 6.3 SE does not have in-built gcc. So we had to download it separately and install it in modelsim directory.
2. For Modelsim simulation we need a number of BFM precompiled library. Xilinx tools and the tcl command don't compile the library so we need to download the precompiled library from Xilinx website.
3. For some reason XPS does not allow the project directory to be outside the local C drive. So all the projects created should be in the same drive in which XPS is installed.

10.4 Simulation setup problems

1. For the BFM simulations we need to include the needed library in the command line while simulating the modelsim. Modelsim don't import them automatically from the Xilinx directory.
2. GDB.v is the global clock module so we need to externally compile that module and need to include in the simulation that is not in precompiled library
3. When using the mix language simulation care must be taken in the create and import peripheral wizard to make sure that in the PAO file our user VHDL code must be at the last else it will leave us some library problem in the BFM simulation Model.

10.5 XPS BSB problems

1. By default all the projects created by the BSB give the error as the board definition has been changed due to the newer FPGA mount on the board.
2. DDR timing and location constraints created by the BSB are wrong so need to generate them by the MIG and then manually or convert them to MPMC constraints by perl script and change the entire IO pin in the UCF file accordingly.
3. Ethernet Mac constrain generating by the BSB are wrong so you need to remove the location constraints of the MAC in the system MHS (master hardware specification) file and need to comment out the already present constraints of the MAC.
4. Project Setting need to be changed after the BSB creation to make the synthesis possible, changed the setting that are according to the xcvlx110T FPGA.
5. By some reason the user logic memory space driver won't work correctly for the user logic so you need to create your own hardware module in the user logic to interact with the master and slave configuration of the user tunnel.
6. By default the TX and Rx buffer of the mac are 8kb, which are highly insufficient, so we need to increase them to the 32 KB.
7. The software driver written for the TFTP server don't make a memory free after each packets transfer on the receive file request so that cause the successive packets transmission problem so after each rcv packets call you need to call the pfree() function to make the buffer empty . This problem exists only in the raw mode set up.
8. Microblaze c compiler is highly insufficient so better to start the project with no optimisation else it will optimise al the empty delay function and much more things.
9. On the successive DDR call the file system don't make the buffer empty so we need to make them empty before any call to the same place.