Debug Interface for Clone of 56000 DSP

Examensarbete utfört i Elektroniksystem
av

Andreas Nilsson

LITH-ISY-EX-ET--07/0319--SE
Linköping 2007
Debug Interface for Clone of 56000 DSP

Examensarbete utfört i Elektroniksystem
vid Linköpings tekniska högskola
av

Andreas Nilsson

LITH-ISY-EX-ET--07/0319--SE

Handledare: Thomas Johansson
Examinator: Kent Palmkvist
Linköping 2007-06-04
Sammanfattning

The scope for this thesis was to design a debug interface for a DSP (digital signal processor). The DSP is a research version of a Motorola 56000 that is designed for a project on asynchronous processor and for use in education. The DSP and debug interface are controlled via a standard PC with RS232 interface equipped with Linux operation system.

In the project 4 blocks has been designed:

The first block can set the DSP core in debug mode or run mode. The second block sends a debug instruction to the DSP core, these debug instructions were prerequisite to the project. The third block enable read and write connection to the memory buses between the DSP core and the three memory blocks. The forth block can override the control signals to the memories from the DSP core.

The project also uses an UART for interpreting and sending control signals and data between the different blocks and the computer.

A text terminal program for Linux has also been programmed for handling the PC side communication. The hardware has been constructed and tested together with a dummy DSP core and dummy memories, but it has not been tested together with the live DSP core.

The Linux program has been tested the same way and seems to do what it's supposed to, though it leaves a lot work to be easy to handle.

Antal sidor: 39

Nyckelord
Hardware testing, Debug interface, DSP, FPGA, ASIC
Abstract
The scope for this thesis was to design a debug interface for a DSP (digital signal processor). The DSP is a research version of a Motorola 56000 that is designed for a project on asynchronous processor and for use in education.

The DSP and debug interface are controlled via a standard PC with RS232 interface equipped with Linux operation system.

In the project 4 blocks has been designed:

The first block can set the DSP core in debug mode or run mode. The second block sends a debug instruction to the DSP core, these debug instructions were prerequisite to the project. The third block enable read and write connection to the memory buses between the DSP core and the three memory blocks. The forth block can override the control signals to the memories from the DSP core.

The project also uses an UART for interpreting and sending control signals and data between the different blocks and the computer.

A text terminal program for Linux has also been programmed for handling the PC side communication.

The hardware has been constructed and tested together with a dummy DSP core and dummy memories, but it has not been tested together with the live DSP core.

The Linux program has been tested the same way and seems to do what it's supposed to, though it leaves a lot work to be easy to handle.
## Contents

1. Introduction ................................................................................................... 1  
   1.1 Background ............................................................................................... 1  
   1.2 Purpose ...................................................................................................... 1  
2. Prerequisite .................................................................................................... 3  
   2.1 Debug instructions .................................................................................... 4  
3. Hardware ....................................................................................................... 5  
   3.1 Hardware used in project .......................................................................... 5  
   3.2 Software for hardware construction .......................................................... 5  
   3.3 Communication between computer and design ........................................ 5  
   3.4 Communication protocol ........................................................................... 5  
      3.4.1 Instruction list ................................................................................... 6  
      3.4.2 Data format of return data................................................................. 6  
4. Hardware construction .................................................................................. 7  
   4.1 UART ........................................................................................................ 8  
      4.1.1 Receiver ............................................................................................. 8  
      4.1.2 Transmitter ........................................................................................ 8  
   4.2 Processor control ....................................................................................... 9  
      4.2.1 In signals ........................................................................................... 9  
      4.2.2 Out signals ........................................................................................ 9  
      4.2.3 Description ...................................................................................... 10  
   4.3 Debug instruction memory ...................................................................... 11  
      4.3.1 In signals ......................................................................................... 11  
      4.3.2 Out signals ....................................................................................... 11  
      4.3.3 Description ...................................................................................... 11  
   4.4 Read and write buffers ............................................................................ 12  
      4.4.1 Read buffer ...................................................................................... 12  
      4.4.2 Write buffer ..................................................................................... 14  
   4.5 Memory control ....................................................................................... 15  
      4.5.1 Controller ........................................................................................ 16  
      4.5.2 Memory address register .................................................................. 17  
      4.5.3 Memory address MUX ..................................................................... 17  
      4.5.4 Control signal MUX ......................................................................... 18  
5. Software ....................................................................................................... 19  
   5.1 Communication thread ............................................................................ 19  
   5.2 Initiate step .............................................................................................. 19  
   5.3 Step program step .................................................................................... 20  
   5.4 Read memories step ................................................................................ 20  
6. Results ......................................................................................................... 21  
   6.1 Synthesized for FPGA ............................................................................. 21  
      6.1.1 Stratix II .......................................................................................... 21  
      6.1.2 Cyclone II ......................................................................................... 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2 Testing</td>
<td>21</td>
</tr>
<tr>
<td>6.3 Possible improvements</td>
<td>22</td>
</tr>
<tr>
<td>6.3.1 Hardware</td>
<td>22</td>
</tr>
<tr>
<td>6.3.2 Software</td>
<td>22</td>
</tr>
<tr>
<td>7 Conclusions</td>
<td>23</td>
</tr>
<tr>
<td>8 References</td>
<td>24</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Background
Electronics Systems (ES), under the Department of Electrical Engineering at the University of Linköping are making a clone of a Motorola 56000 digital signal processor (DSP). The processor is designed as VHDL code that can be synthesized for a FPGA (field programmable gate array). The main purpose of producing the processor clone is to use it for testing ideas in the research of the department and to use it in education.

1.2 Purpose
The purpose of this thesis is to design a debug interface for this DSP core where it is possible to control the DSP core via a standard PC.

Features that are needed are:

- Writing to memories.
- Reading from memories.
- Running the processor in normal mode.
- Running the processor one instruction at a time.
- Reading processor registers when processor is in debug mode (not running).

In my project it was also included a task to make a program to control the hardware from the PC-computer.
2 Prerequisite

The goal of this project is to be able to load and run a test program in the DSP core and to read the values of the registers in the DSP between every instruction step. After the execution of the program the content of the DSP memories are to be read and all the gathered data compared with expected result. All this is to be controlled via a standard PC with Linux operation system. The control is supposed to be done manually via a terminal program or automatically via a program on the PC.

![Figure 1: A sketch over the usable data buses, memory control bus and DSP debug control signals.](image)

The DSP has the following I/Os that is interesting to this project and where it is possible to connect extra hardware.

- Debug signal (processor control), an in signal to switch between run and debug mode.
- ds0, an out signal that goes high when the last clock cycle of an instruction is executed
- Debug instruction bus, a 6-bit in instruction bus that sends instructions to the DSP core when it is in debug mode. If this bus is set to a value of a valid debug instruction when the DSP is in debug mode a register value assigned by this instruction will be copied to the X data bus and other buses will be tri-stated, P and Y data bus will always be tri-stated in debug mode. If a non valid debug instruction is set the X data bus will be tri-stated.
- 3*24-bit bidirectional data buses between the DSP core and the memories, the 3 memories are Program memory, X data memory and Y data memory.
- 3 control signals from the DSP core for each memory these signals are read/write, output enable and chip select.
2.1 Debug instructions

The debug instructions for the DSP core have the following format [1].

<table>
<thead>
<tr>
<th>d5 d4 d3 d2 d1 d0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0</td>
<td>Tristated output</td>
</tr>
<tr>
<td>0 0 0 0 X 1</td>
<td>Reserved</td>
</tr>
<tr>
<td>0 0 0 1 D D</td>
<td>Data ALU register, Table A-18 Motorola FM</td>
</tr>
<tr>
<td>0 0 1 D D D</td>
<td>Data ALU register, Table A-18 Motorola FM</td>
</tr>
<tr>
<td>0 1 0 T T T</td>
<td>Address ALU register, Table A-18 Motorola FM</td>
</tr>
<tr>
<td>0 1 1 N N N</td>
<td>Address ALU register, Table A-18 Motorola FM</td>
</tr>
<tr>
<td>1 0 0 F F F</td>
<td>Address ALU register, Table A-18 Motorola FM</td>
</tr>
<tr>
<td>1 0 1 P P P</td>
<td>Pipeline table, own definition</td>
</tr>
<tr>
<td>1 1 0 X X X</td>
<td>Reserved</td>
</tr>
<tr>
<td>1 1 1 G G G</td>
<td>PCU register, Table A-18 Motorola FM</td>
</tr>
</tbody>
</table>

Triple bit register encodings

<table>
<thead>
<tr>
<th>Code</th>
<th>DDD</th>
<th>DD</th>
<th>FFF</th>
<th>NNN</th>
<th>TTT</th>
<th>GGG</th>
<th>PPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A0</td>
<td>*</td>
<td>M0</td>
<td>N0</td>
<td>R0</td>
<td>*</td>
<td>PipePC</td>
</tr>
<tr>
<td>001</td>
<td>B0</td>
<td>*</td>
<td>M1</td>
<td>N1</td>
<td>R1</td>
<td>SR</td>
<td>PipeWord</td>
</tr>
<tr>
<td>010</td>
<td>A2</td>
<td>*</td>
<td>M2</td>
<td>N2</td>
<td>R2</td>
<td>OMR</td>
<td>SSHdebug</td>
</tr>
<tr>
<td>011</td>
<td>B2</td>
<td>*</td>
<td>M3</td>
<td>N3</td>
<td>R3</td>
<td>SP</td>
<td>LCdebug</td>
</tr>
<tr>
<td>100</td>
<td>A1</td>
<td>X0</td>
<td>M4</td>
<td>N4</td>
<td>R4</td>
<td>SSH</td>
<td>LAdebug</td>
</tr>
<tr>
<td>101</td>
<td>B1</td>
<td>X1</td>
<td>M5</td>
<td>N5</td>
<td>R5</td>
<td>SSL</td>
<td>*</td>
</tr>
<tr>
<td>110</td>
<td>*</td>
<td>Y0</td>
<td>M6</td>
<td>N6</td>
<td>R6</td>
<td>LA</td>
<td>*</td>
</tr>
<tr>
<td>111</td>
<td>*</td>
<td>Y1</td>
<td>M7</td>
<td>N7</td>
<td>R7</td>
<td>LC</td>
<td>*</td>
</tr>
</tbody>
</table>
3 Hardware

3.1 Hardware used in project
During this project I have used a development board from Altera with a Stratix II FPGA chip, in the end of the project changed to a development board with a Cyclone II FPGA chip. The Department of Electrical Engineering are also planning to make an ASIC including a synthesised version of my project.

3.2 Software for hardware construction
During the design process I have been using HDL Designer from Mentor Graphics for VHDL coding and Quartus from Altera for synthesising the code.

3.3 Communication between computer and design
For communication between the hardware design and a standard PC-computer the RS232 communication protocol without echo of sent data was used, this due to the simplicity of this protocol.

The communication speed shall be set to a fixed speed during synthesis of the design. The speed is depending on the clock frequency used for the implementation and the number of bits per second (bps) can not be higher than 1/10 of that clock frequency.

In the case of this thesis the clock frequency was set to 48 MHz and the communication speed was set to 57600 bps. Other examples of functioning communication speed are 4800, 9600, 19200, 38400, 115200 and 230400.

3.4 Communication protocol
The communication protocol used is 8-bit ASCII. Different ASCII signs are used for sending instructions and data.

0-9 and A-F are used for sending data in hex format.

Letters except for A-F are used for sending instructions (case sensitive).

“;” and “:” are used for flow control.
3.4.1 Instruction list
Here below follows a list of the ASCII instructions used for sending data to the
debug interface from the computer.

x = read X-bus, returns data.
y = read Y-bus, returns data.
p = read P-bus, returns data.
X = write to X-bus, followed by data in hex format (typ 24 bit)
Y = write to Y-bus, followed by data in hex format (typ 24 bit)
P = write to P-bus, followed by data in hex format (typ 24 bit)
I = set instruction, followed by instruction data in hex format (typ 6 bit)
M = set memory address, followed by address data in hex format
R = run complete program
r = run a single instruction and return to debug mode
S = stop program and enter debug mode
0-9 and A-F = hex data.
“return” = end of data

3.4.2 Data format of return data
Here below follows a list of the ASCII instructions used for sending data from
the debug interface to the computer. Data is only returned on instructions from
the computer, see chapter 3.4.1 for these instructions.

0-9 and A-F = returned data.
; after input data x, y or p = end of data
; after input data r = end of instruction
: after input data r = end of program.
: after input data R = end of program
4 Hardware construction

The debug interface contains of the following blocks:

- UART – A communication interface that translates and sends data between the computer and the rest of the debug interface.
- Processor controller – A controller for setting debug mode or run mode on the processor.
- Debug instruction memory – A memory for saving the debug instruction received from the computer and forward it to the processor core.
- Read and write buffer – These buffers handle the communication between data buses and the UART
- Memory control – A memory control override to use when core is in debug mode.

![Diagram](image)

*Figure 2: A sketch over the different blocks and how they are connected to the data buses, control buses and DSP.*
4.1 UART
I have used a ready UART and decoder unit built by Peter Kröger for the Department of Electrical Engineering. The UART has been modified to trigger on the ASCII signals used in the instruction list. By default, the decoder part detects ASCII signals 0-9 and A-F as hexadecimal data and converts it to 4 bit binary data.

4.1.1 Receiver
The receiver reads incoming RS232 ASCII. If an ASCII sign in the instruction list is received, the UART sends a trigger signal to the affected construction block.

If ASCII for 0-9 or A-F is received, it is converted to 4 bit binary packets to all other construction blocks followed by a trigger signal telling that data is present.

If ASCII for “;” or <return> a signal telling each block that there are no more data to receive is sent.

4.1.2 Transmitter
The transmitter sends ASCII data when told to do so from the other construction blocks. If one block sends a signal to send data when the transmitter is already sending data, no action will be taken of this signal.
4.2 Processor control

Figure 3: The state diagram for the processor control state machine.

4.2.1 In signals
Here below follows a list of in signals to this block and were they come from

- Run full from UART.
- Run single instruction from UART.
- Stop signal from UART.
- End of instruction from DSP core.
- Ds0 from DSP core.
- Illegal instruction from DSP core.

4.2.2 Out signals
Here below follows a list of out signals to this block and were they go

- Debug mode to DSP core.
- End of instruction to UART.
- End of program to UART.
4.2.3 Description

The processor control is a state machine that generates the control signal to the DSP core to set it into debug or run mode. It also generates return signals for end of instruction and end of program to the UART.

In the “idle” state the DSP is set into debug mode and the state machine waits for a signal to run a single instruction to go to the “run single state” or to run until end of program to go to the “run full state”.

In the “run full state” the state machine sends a run signal to the DSP and waits for a stop signal from the UART or an illegal signal from the DSP. Either of these signals makes the state machine go on to the stop end of program state.

In the “stop end of program” state the state machine sets the DSP in debug mode, sends an end of program signal to the UART and goes on to the “Idle” state.

In the “run single” state the state machine sends a run signal to the DSP and waits for signals for end of instruction or signals for end of program. In case of an end of program signal the state machine goes to the “stop end of program” state. In case of an “end of instruction” signal the state machine goes to the “stop end of instruction” state.

The “stop end of instruction” state sets the DSP in debug mode, sends an end of instruction signal to the UART and the state machine goes on to the “Idle” state.
4.3 Debug instruction memory

4.3.1 In signals
Here below follows a list of in signals to this block and were they come from
- New instruction data trigger signal from UART.
- End of data signal from UART.
- Data available from UART.
- 4-bit data bus from UART.

4.3.2 Out signals
Here below follows a list of out signals to this block and were they go
- 6-bit debug instruction bus to DSP core.

4.3.3 Description
The debug instruction memory consists of a controller and one shift register. The controller triggers on a signal from the UART to start shifting in new data into the shift register and an end of data to stop shifting in data to the shift register.

The shift register triggers on an enable signal from the controller and a new data signal from the UART to shift in data from the UART.
4.4 Read and write buffers
The read and write buffers consists of one read buffer and one write buffer that has been linked together to one building block.

Figure 5: Block diagram for the read and write buffers.

4.4.1 Read buffer

Figure 6: State diagram for the read buffer state machine.
4.4.1.1 In signals
Here below follows a list of in signals to this block and were they come from

- 24-bit data bus from bus between DSP core and memory.
- Read bus trigger signal from UART.
- Send next data trigger signal from UART.

4.4.1.2 Out signals
Here below follows a list of out signals to this block and were they go

- 4-bit data bus to UART.
- Available data trigger signal to UART.
- No more data to send trigger to UART.

4.4.1.3 Description
The read buffer actually is a state machine that in the “idle” state waits for a read signal from the UART. When the trigger signal is received the state machine moves to the “send” state.

The first time in each send cycle that the “send” state is reached it will copy the 4 most significant data bits from the data bus and send it to the UART together with a send data trigger signal. The state machine goes on to the “send 2” state. The next time the “send” state is reached the next 4 bits from the bus will be copied and so on down to the 4 lowest significant bits.

In the “send 2” state the state machine waits for a handshake signal that tells that the UART is ready to receive more data and goes on to the “send check” state.

In the “send check” state the state machine determines if the all data has been sent or not, if all data has been sent the machine goes to the “wait idle” state otherwise back to “send” state to send one more 4-bit package to the UART.

In the “wait idle” state it’s determined if the UART is ready to sent data, if it is the machine goes to “send end of data” state.

In the “send end of data” state the machine sends an end of data signal to the UART and the machine goes in to the “idle” state.
4.4.2 Write buffer

Figure 7: Block diagram for the write buffer

4.4.2.1 In signals
Here below follows a list of in signals to this block and were they come from

- Trigger signal from UART
- End of data signal from UART
- New data signal from UART
- 4-bit input data bus from UART.
- Output enable from memory control

4.4.2.2 Out signals
Here below follows a list of out signals to this block and were they go

- Output data to data bus, tri-stated

4.4.2.3 Description
The write buffer part of the read/write block triggers on signals from the receiver block and from the memory control block. It receives data as 4 bit blocks from the receiver block and shifts this data into a shift register. When a write signal is received from the memory control bus the data from the shift register is set on the data bus linked to the block.
4.5 Memory control

Figure 8: Block diagram for the memory control block.

The memory control block handles if the control signals to the memories come from the DSP core or if the memories are controlled by internal control signals from the debug interface.

The memory control block consists of the following parts.

- Controller.
- Memory address register.
- One memory address MUX for each memory (P, X and Y).
- One control signal MUX for each memory (P, X and Y).
4.5.1 Controller

Figure 9: State diagram for the controller block in the memory control block.

4.5.1.1 In signals
Here below follows a list of in signals to this block and were they come from
- Read in memory trigger signal from UART.
- Write in memory trigger signal from UART.
- Data available signal from UART.
- 4-bit data bus from UART.

4.5.1.2 Out signals
Here below follows a list of out signals to this block and were they go.
- 3 select signals to memory address MUXes and control signal MUXes.
- 3 out put enable signals, one to each read/write block.
- Read/write signal for memory, goes to each control signal MUX.
- Output enable signal for memory, goes to each control signal MUX.
- Chip select signal for memory, goes to each control signal MUX.

4.5.1.3 Description
The controller generates control signals to the memories and handle if the control signals that’s internally generated in the controller or the signals from the DSP core are to be used.

The controller works as a state machine. When the state machine is in the idle state the memory control block is transparent to the DSP core.
4.5.2 Memory address register

4.5.2.1 In signals
Here below follows a list of in signals to this block and were they come from.

- New memory address data trigger signal from UART.
- End of memory address data trigger signal from UART.
- 4-bit data bus from UART.
- Data available signal from UART.

4.5.2.2 Out signals
Here below follows a list of out signals to this block and were they go.

- Memory address bus to memory address MUXes

4.5.2.3 Description
The memory address register works in the same way as the debug instruction memory (see 4.3) and is used for setting an override memory address to the memories.

4.5.3 Memory address MUX

4.5.3.1 In signals
Here below follows a list of in signals to this block and were they come from

- Memory address bus from DSP processor
- Memory address bus from memory address memory
- Select signal from controller in memory control

4.5.3.2 Out signals
Here below follows a list of out signals to this block and were they go

- Memory address bus to external memory

4.5.3.3 Description
The memory address MUX selects memory address from the memory address register (4.5.2) or from the DSPs memory address bus. The selection is made from the select signal from the controller.
4.5.4 Control signal MUX

4.5.4.1 In signals
Here below follows a list of in signals to this block and were they come from

- Chip select signal from DSP core.
- Chip select signal from memory control controller.
- Output enable signal from DSP core.
- Output enable signal from memory control controller.
- Memory read/write signal from DSP core.
- Memory read/write signal from memory control controller.
- Select signal from memory control controller.

4.5.4.2 Out signals
Here below follows a list of out signals to this block and were they go

- Chip select signal to memories.
- Output enable signal to memories.
- Memory read/write signal to memories.

4.5.4.3 Description
The control signal MUX selects if the control signal to the memories come from the controller or from the DSP. The selection is made from the select signal from the controller
5 Software
The software sends control instructions via RS232 from computer to the debug interface and receives the returned data and formats it.

The program runs in two different threads, the first thread handles the actual communication over RS232, the other thread generates data to transmit and formats the received data. The two threads communicate via a shared memory area.

The program has three different modes.

- **Init** – It loads data into the P, X and Y memory in the processor. P for program memory. X and Y for data memory,
- **Step program** – It steps the program in the core and reads register values from the core.
- **Read memories** – It makes a memory dump of P, X and Y memories.

5.1 Communication thread
The communication thread receives the input from the RS232 interface and puts the input data in a cyclic shared buffer. The data can be polled from the different steps in the 3 program steps. The communication thread has been very influenced by a rs232server program by David Björkevik.

5.2 Initiate step
The initiate step opens the file called “file1.txt” that is supposed to be placed in the same catalogue as the software program. The ASCII in this file is then sent on to the RS232 port. Returned data will be ignored in this step. ASCII sign “/” is used for commenting in the text file, which means that ASCII between two “/” signs will be ignored.

This step is supposed to be used for loading the different memories. The ASCII sign “;” in the text file will end this step and the software will go on to the Step program step.
5.3 Step program step
The Step program step starts with sending the ASCII sign for running a single instruction on the DSP core and waits for the return of the ASCII sign for either end of instruction or the ASCII sign for end of program.

It then sends instructions to the debug interface to get the data from the different registers in the DSP core. When all data is collected it will be formatted and presented in the text terminal. If the ASCII sign for end of program was received in after executing the instruction the step is terminated and the program will continue on to the read memories step. Else if the end of instruction was received this step will be run again until end of program signal is received.

5.4 Read memories step
The read memories step is supposed to be used for reading the data in the DSP memories.

It sends the ASCII from the file called “file3.txt” placed in the same catalogue as the software to the RS232 interface and waits for return data where expected. Returned data will be presented on in the text terminal window. ASCII placed between two “”” signs in the text file will also be presented in the text terminal. And ASCII placed between two “/” will be ignored as comments in the text file, it is not possible to make comments in text that is supposed to bee presented in the text terminal.

When a “;” ASCII sign is detected in the text file this step will end and the program will terminate and close the RS232 port.
6 Results

6.1 Synthesized for FPGA

6.1.1 Stratix II
The design has been synthesized for the Stratix II FPGA from Altera with no errors.

The design uses 438 LUTs (Logical Units) and 355 registers of the FPGAs capacity.

The design also uses 200 I/O ports, out of those 200 I/Os 196 will be internal when the design is connected to the DSP processor and the memory, 2 will be shared with the DSP processor (clk and rst), this means that the design actually need 2 I/O ports from the FPGA chip.

Maximum clocking speed for the design for the Stratix II chip is 183.45 MHz

For complete area report see Appendix 1
For complete time analyse report see Appendix 2

6.1.2 Cyclone II
The design has been synthesized for the Cyclone II FPGA from Altera with no errors.

The design uses 356 LUTs (Logical Units) and 336 registers of the FPGAs capacity.

The design also uses 200 I/O ports, out of those 200 I/Os 196 will be internal when the design is connected to the DSP processor and the memory, 2 will be shared with the DSP processor (clk and rst), this means that the design actually need 2 I/O ports from the FPGA chip.

Maximum clocking speed for the design for the Stratix II chip is 91.94 MHz.

For complete area report see Appendix 3
For complete time analyse report see Appendix 4

6.2 Testing
The hardware and software has been tested with a dummy core and dummy memories since the finished Core and the real memories were not available to me during the design period.

A standard PC with Linux has been connected via a RS232 to a FPGA development board with the test design downloaded the FPGA. The PC program has been run and the result has been compared with expected result.
6.3 Possible improvements

6.3.1 Hardware
As of now there is one known bug in the Core dummy that is used for debugging, this means that the dummy Core cannot send a correct end of program signal to the debug interface while stepping the DSP program.

The memory control has been coded for a generic s-ram and may need to be replaced to work with the memory that will be used in the end.

Also there has been no validation done with the actual DSP core and the actual memories which mean more bugs might be found.

6.3.2 Software
The PC program (Linux) has a lot of improvements to do.

As it is today it is fixed to one RS232 device from the compilation, in an improved version this could be set in a configuration file.

Another problem with the program is that all output data is sent to standard output, which means that it is presented as text output in the text terminal. A better way would be to save the data directly to output files.

A GUI interface could be nice to.
7 Conclusions

The hardware debug interface has been successfully developed and works nice together with a dummy core and a dummy memory, though it has not been tested together with the live core and the live memories.

The software manages to gather the wanted data from the DSP via RS232 and the debug interface. The data is correctly formatted and presented on the screen but the software still is far from ready, this due to lack of time.


8 References


Reference literature for VHDL:
Appendix 1

Area report for Stratix II FPGA

Device Utilization for EP2S15F484C

<table>
<thead>
<tr>
<th>Resource</th>
<th>Used</th>
<th>Avail</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOs</td>
<td>200</td>
<td>343</td>
<td>58.31%</td>
</tr>
<tr>
<td>LUTs</td>
<td>438</td>
<td>12480</td>
<td>3.51%</td>
</tr>
<tr>
<td>Registers</td>
<td>355</td>
<td>14410</td>
<td>2.46%</td>
</tr>
<tr>
<td>Memory Bits</td>
<td>0</td>
<td>419328</td>
<td>0.00%</td>
</tr>
<tr>
<td>DSP block 9-bit elems</td>
<td>0</td>
<td>96</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Library: hv_debint_m0_1    Cell: debuger    View: struct

Number of ports :                     200
Number of nets :                       591
Number of instances :                 345
Number of references to this view :   0
Total accumulated area :
Number of IOs :                       200
Number of LCs :                       0
Number of LUTs :                      438
Number of Registers :                355
Number of accumulated instances :    1036
# Appendix 2

## Timing report for Stratix II

### Timing Analyzer Summary

| Type          | Worst-case tsu   | Slack | N/A  | Required Time | None  | Actual Time | 4.254 ns | From | seriell_data_in | To   | U_0_I0|r_tecken_3_ | --   | To Clock | clk | Failed Paths | 0   |
|---------------|-----------------|------|------|---------------|-------|-------------|---------|------|-----------------|------|------------------|------|----------|----|-------------|-----|
| Type          | Worst-case tco  | Slack| N/A  | Required Time | None  | Actual Time | 9.291 ns | From | U_7|s_x             | To   | adress_x_out[8] | --   | To Clock | clk | Failed Paths | 0   |
| Type          | Worst-case tpd  | Slack| N/A  | Required Time | None  | Actual Time | 10.274 ns | From | rst             | To   | x_bus[12]       | --   | To Clock | --  | Failed Paths | 0   |
| Type          | Worst-case th   | Slack| N/A  | Required Time | None  | Actual Time | -2.062 ns | From | y_bus[17]       | To   | U_4_U_0|r_data_temp_17_ | --   | To Clock | clk | Failed Paths | 0   |

**Type**: Clock Setup: 'clk'

<table>
<thead>
<tr>
<th>Slack</th>
<th>4.549 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Time</td>
<td>100.00 MHz ( period = 10.000 ns )</td>
</tr>
<tr>
<td>Actual Time</td>
<td>193.45 MHz ( period = 5.451 ns )</td>
</tr>
</tbody>
</table>

From: U_0_I0|r_tecken_1_  
To: U_5_U_1|U_1_sr_0_  
Failed Paths: 0

**Type**: Clock Hold: 'clk'

<table>
<thead>
<tr>
<th>Slack</th>
<th>0.018 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Time</td>
<td>100.00 MHz ( period = 10.000 ns )</td>
</tr>
<tr>
<td>Actual Time</td>
<td>N/A</td>
</tr>
</tbody>
</table>

From: U_0_sr_out_4_  
To: U_0_sr_out_3_  
Failed Paths: 0
Type : Total number of failed paths
Slack :
Required Time :
Actual Time :
From :
To :
From Clock :
To Clock :
Failed Paths : 0

---------------------------------------------------------------
# Appendix 3

## Area report for Cyclone II

*******************************

**Device Utilization for EP2C35F672C**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Used</th>
<th>Avail</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOs</td>
<td>200</td>
<td>475</td>
<td>42.11%</td>
</tr>
<tr>
<td>LUTs</td>
<td>565</td>
<td>33216</td>
<td>1.70%</td>
</tr>
<tr>
<td>Registers</td>
<td>336</td>
<td>33216</td>
<td>1.01%</td>
</tr>
<tr>
<td>Memory Bits</td>
<td>0</td>
<td>483840</td>
<td>0.00%</td>
</tr>
<tr>
<td>DSP block 9-bit elems</td>
<td>0</td>
<td>70</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

*******************************

Library: hv_debint_m0_1_cyclone   Cell: debugger   View: struct

*******************************

<table>
<thead>
<tr>
<th>Cell</th>
<th>Library</th>
<th>References</th>
<th>Total Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>cycloneii</td>
<td>2 x</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>cycloneii</td>
<td>2 x</td>
<td></td>
</tr>
<tr>
<td>RS232_IF1</td>
<td>hv_debint_m0_1_cyclone</td>
<td>1 x 57</td>
<td></td>
</tr>
<tr>
<td>57 Registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 LCs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>157 LUTs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS232_send_if</td>
<td>hv_debint_m0_1_cyclone</td>
<td>1 x 36</td>
<td></td>
</tr>
<tr>
<td>36 Registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123 LUTs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 LCs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycloneii_io_bidir_outputreg_en</td>
<td>cycloneii</td>
<td>72 x 1</td>
<td>1</td>
</tr>
<tr>
<td>72 IOs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycloneii_io_input</td>
<td>cycloneii</td>
<td>62 x 1</td>
<td>1</td>
</tr>
<tr>
<td>62 IOs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycloneii_io_input_inputreg</td>
<td>cycloneii</td>
<td>1 x 1</td>
<td>1</td>
</tr>
<tr>
<td>1 IOs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycloneii_io_output</td>
<td>cycloneii</td>
<td>58 x 1</td>
<td>1</td>
</tr>
<tr>
<td>58 IOs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycloneii_io_output_outputreg</td>
<td>cycloneii</td>
<td>7 x 1</td>
<td>1</td>
</tr>
<tr>
<td>7 IOs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycloneii_lcell_ff</td>
<td>cycloneii</td>
<td>92 x 1</td>
<td>1</td>
</tr>
<tr>
<td>92 Registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycloneii_lcell_normal</td>
<td>cycloneii</td>
<td>22 x 1</td>
<td>1</td>
</tr>
<tr>
<td>22 LUTs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>hv_debint_m0_1_cyclone</td>
<td>1 x 34</td>
<td></td>
</tr>
<tr>
<td>34 Registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>122 LUTs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 LCs</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
read_bus                           hv_debint_m0_1_cyclone     1 x     39
39 Registers
47 LUTs
0 LCs
read_bus_unfolded0                 hv_debint_m0_1_cyclone     1 x     39
39 Registers
47 LUTs
0 LCs
read_bus_unfolded1                 hv_debint_m0_1_cyclone     1 x     39
39 Registers
47 LUTs
0 LCs

Number of ports :                     200
Number of nets :                      563
Number of instances :                 324
Number of references to this view :     0

Total accumulated area :
Number of IOs :                       200
Number of LCs :                         0
Number of LUTs :                      565
Number of Registers :                 336
Number of accumulated instances :    1129
Appendix 4

Timing report for Cyclone II

Timing Analyzer Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Worst-case tsu</th>
<th>Slack</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Time</td>
<td>None</td>
<td>Actual Time</td>
<td>4.798 ns</td>
</tr>
<tr>
<td>From</td>
<td>illegal_instr</td>
<td>To</td>
<td>U_2_r_state_2_</td>
</tr>
<tr>
<td>From Clock</td>
<td>--</td>
<td>To Clock</td>
<td>clk</td>
</tr>
<tr>
<td>Failed Paths</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Worst-case tco</th>
<th>Slack</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Time</td>
<td>None</td>
<td>Actual Time</td>
<td>12.151 ns</td>
</tr>
<tr>
<td>From</td>
<td>U_7</td>
<td>oe_r_w_regy</td>
<td>To</td>
</tr>
<tr>
<td>From Clock</td>
<td>clk</td>
<td>To Clock</td>
<td>--</td>
</tr>
<tr>
<td>Failed Paths</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Worst-case tpd</th>
<th>Slack</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Time</td>
<td>None</td>
<td>Actual Time</td>
<td>13.091 ns</td>
</tr>
<tr>
<td>From</td>
<td>adress_y_in[14]</td>
<td>To</td>
<td>adress_y_out[14]</td>
</tr>
<tr>
<td>From Clock</td>
<td>--</td>
<td>To Clock</td>
<td>--</td>
</tr>
<tr>
<td>Failed Paths</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Worst-case th</th>
<th>Slack</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Time</td>
<td>None</td>
<td>Actual Time</td>
<td>-0.220 ns</td>
</tr>
<tr>
<td>From</td>
<td>seriell_data_in</td>
<td>To</td>
<td>U_0_I0</td>
</tr>
<tr>
<td>From Clock</td>
<td>--</td>
<td>To Clock</td>
<td>clk</td>
</tr>
<tr>
<td>Failed Paths</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Clock Setup: 'clk'</th>
<th>Slack</th>
<th>9.123 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Time</td>
<td>50.00 MHz ( period = 20.000 ns )</td>
<td>Actual Time</td>
<td>91.94 MHz ( period = 10.877 ns )</td>
</tr>
<tr>
<td>From</td>
<td>U_0_I0</td>
<td>r_tecken_5_</td>
<td>To</td>
</tr>
<tr>
<td>From Clock</td>
<td>clk</td>
<td>To Clock</td>
<td>clk</td>
</tr>
<tr>
<td>Failed Paths</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Clock Hold: 'clk'</th>
<th>Slack</th>
<th>0.391 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Time</td>
<td>50.00 MHz ( period = 20.000 ns )</td>
<td>Actual Time</td>
<td>N/A</td>
</tr>
<tr>
<td>From</td>
<td>U_7</td>
<td>mem_adress_0_</td>
<td>To</td>
</tr>
<tr>
<td>From Clock</td>
<td>clk</td>
<td>To Clock</td>
<td>clk</td>
</tr>
<tr>
<td>Failed Paths</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Total number of failed paths</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Required Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>From</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>From Clock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>To Clock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Failed Paths</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix 5

Manual

How to connect hardware

Connect RS232 port on FPGA-board to RS232 port on the computer. If the computer does not have a comport use a USB to RS232 dongle. Most Linux distributions have native support for USB to RS232 dongles based on the FTDI chip.

Manual control

Start a terminal program on the computer, for example miniterm under Linux. Under settings for the chosen program select the used RS232-port (e.g. ttyS0 in Linux of com 1 in windows) and set speed to the speed selected during synthesising of the design (see 3.3), no parity, 8-bit ASCII blocks and local echo.

Send instructions. See instruction list (3.4.1).

Example of instructions

Here follows some examples of how to communicate with the interface, DSP core and memories.

How to run step a single instruction

- Send the ASCII sign “r” the interface will return either “;” for end of instruction or “:” for end of program.

How to run the full DSP program

- Send the ASCII sign “R” the interface will return “:” when the DSP program reaches and illegal instruction which is used for end of program. The program can also be manually stopped by sending ASCII sign “S”.


How to load data into memory

- Start with making sure that the DSP core is in debug mode if you're not shure send the ASCII sign “S” if the DSP was running a “;” will be returned otherwise no return will be made.
- Now set the debug instruction to '000000' by sending “I00”
- Set memory address by sending “MZZZZZZ” where Z are hexadecimal digits.
- Set bus data for X,Y and P buses by sending “XZZZZ” “YZZZZ” and “PZ”, wher Z are hexadecimal digits.
- To write the data from the bus into the memories send the ASCII data “WZ” where Z is a hexadecimal digit. Lowest significant bit from the word triggers the X memory to store bus value, second bit triggers the Y memory and the third bit triggers the the P memory. The forth bit is don't care.

How to read memory

- Start with making shure that the DSP is in debug mode and that the debug instruction is set to '000000', similar to in writing to memory.
- Set Memory read address by sending “MZZZZZZ” where Z represents memory address hexadecimally.
- Sen “GZ” where Z is a hexadecimal value similar used as for the “W” instruction. This will copy the memory values to the data bus.
- Now send x,y or p to get the data returned from respected buses.
- The data will be returned as “ZZZZZ,” where Z represents hexadecimal data.
- Sen “g” to stop reading the data from the memories to the data buses and to increment the memory adress.
How to read A0 register

- Start with making sure that the DSP is in debug mode similar to previous.
- Set Debug instruction to '001000' (for other registers use the debug instruction for that register instead) by sending “I08d” (hexadecimal version of the debug instruction)
- Send “x” to read the register value from the data bus, the value will be returned as “ZZZZZZ;” where Z are hexadecimal data.

Software control

Start with locating the software file “ctrlprog” and generate the two text files called “file1.txt” and “file3.txt”. In “file1.txt” is supposed to have the ASCII data that is needed for loading the initial data for the DSP memories and the file “file3.txt” is supposed to have the instructions for reading the data from the DSP memories, see Appendix 6 for an example of how these files can look.

Make sure that you have permission to connect to the RS232 interface, if not contact the sys admin to get permission.

Run the program by typing ctrlprog in the text terminal and in the catalogue where the 3 files are present. If you like to get the output data to a text file instead of as on screen data use the command “ctrlprog > [file name]. See Appendix 7 for example on how output can look.
Appendix 6

Example of data in file1.txt

M0000  /set memory address/
X111111  /set X bus value/
Y222222  /set Y bus value/
P333333  /set P bus value/
W7  /write X, Y and P bus values to Memory address previously /
M0001  /set next memory address/
X444444  /set X bus value/
Y555555  /set Y bus value/
P666666  /set P bus value/
W7  /write X, Y and P bus values to Memory address previously /
;
   /end of file/

Example of data in file3.txt

M 00000000000  /set memory address for first reading/
G 7  /set X, Y and P buses to memory values/
p  /read P bus value/
"
"
   /print new line to text terminal/
x  /read X bus value/
"
"
   /print new line to text terminal/
y  /read Y bus value/
"
"
   /print new line to text terminal/
g  /tristate X, Y and P bus from mem and inc mem address/
;
   /end of file/
Appendix 7

Example of output data from software

```
step
  x= $400101000000  y= $400103400102
  a= $0A40010C400109  b= $0B40010D400109
    x1= $400101  x0= $000000  r7= $0117  n7= $011F  m7= $0127
    x1= $400103  x0= $400102  r6= $0116  n6= $011E  m6= $0126
  a2= $0A  a1= $40010C  a0= $400109  r5= $0115  n5= $011D  m5= $0125
  b2= $0B  b1= $40010D  b0= $400109  r4= $0114  n4= $011C  m4= $0124
    r3= $0113  n3= $011B  m3= $0123
  pc= $0128  sr= $0138  omr= $3A
  la= $013E  lc= $013F
  ssh= $013C  ssl= $013D  sp= $3B
  P:0128 400129
```

000000
400001
800002
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Andreas Nilsson