Quick TTA to FPGA tutorial

Part 1: Generating the processor VHDL files

1. If Prode is not already open, start it

   Note: Your processor has data memory and instruction memory. Instruction memory contains the binary version of your C program. The number of entries of this memory should be more than number of instructions in your program. You can check number of instructions in your program in **proxim** by scrolling down to end of the instructions list. Data memory of course is where you store variables, arrays, etc.

2. If your processor has very large instruction and data memory spaces, reduce them now through “Edit” → ”Address spaces …”

3. Click on "Tools” → “Processor implementation"
   Note: This way you direct the tool to appropriate HDL implementation of each functional unit and RF.

4. Select implementations for each FU and RF. When available, select implementations that have an asterisk (*)
   NOTE: **for LSU select implementation from *stratixII.hdb*; and for stream unit select *fifo_stream.hdb***

   Hint :You can check the actual VHDL files of each unit in following folder: “/usr/local/share/tce/hdb”

A set of different FU implementations exist in the standard TCE libraries. Some additional implementations are in stratixII.hdb and stream.hdb. All are written in VHDL.

5. Switch to the last tab “IC/Decoder plugin”

6. Set "hdb file" to "/usr/local/share/tce/hdb/asic_130nm_1.5V.hdb"

7. Press "Save IDF" and save the file under the name that is proposed

8. Exit prode

The *bem* (binary encoding map) file defines the coding of instruction words.

9. In the work folder, type “createbem *processor*.adf" (substitute *processor* with the name of your processor)

10. Generate the processor VHDL files with the command
    generateprocessor -b *processor*.bem -e toplevel -l 'vhdl' -i *processor*.idf -o hdl_files *processor*.adf
    Note: generateprocessor belongs to TCE toolchain and elaborates your design in a few HDL files

11. The VHDL files are now in the “hdl_files” folder
    Note: These files contain generated HDL code for implementing your processor.

Part 2: Generating data and instruction memory content

Though you have the processor ready to be implemented in hardware, but what that processor should execute? In this step you convert your compile code to .mif format (Memory Initialization File), that can to be written to instruction memory.

1. If you have not yet compiled your program for the current adf file, do it with the following command:
   - for C code, type "tcecc -O3 -o *program*.tpef *source*.c -a *processor*.adf"
   - for assembly, type “tceasm -o *program*.tpef *processor*.adf *source*.tceasm”
2. Type "generatebits -e toplevel -b *processor*.bem -d -w 4 -p *program*.tpef -x hdl_files -f 'mif' -o 'mif' *processor*.adf"

Open generated mif files and examine their content.

Part 3: Working with Quartus II

1. Create a subfolder “quartus” under the work folder

2. Copy the files from ~/Desktop/ASSP/fpga_files/quartus to that directory
3. Also, depending on the filter you are working with, copy the .mif files from ~/Desktop/ASSP/fpga_files/*filter* to the same folder
   Note: Check the source code of your program to determine the filter type. This mif file contains your signal samples, this file is different from the mif file that contains your program. The blocks for streaming IO that are used here, are only meant to imitate real IO streaming instruments. For other projects you can use different methods such as a serial port.

Signal names that have a "_n" or "_x" are inverted. Because the processor provides an inverted signal, and the memory expects a non-inverted one, we have to place an inverter (not) in between.

4. Start Quartus II from the desktop icon

5. Select "File” → ”New project wizard" and press "next"

6. Set the project working directory as "*work*/quartus"

7. Set the project name as "tta" and click "next"

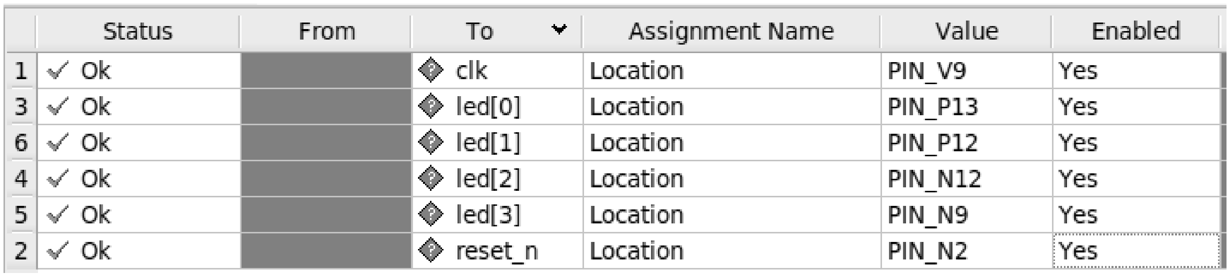
8. By going through "..." and "add" repeatedly, add all .vhdl and .v files to the list from folders *work*/hdl_files/vhdl and *work*/ hdl_files /gcu_ic and *work*/quartus

9. Click "next"

10. As "Family", select "Cyclone III" (or “Cyclone IV” depending on the available FPGA board to you)

11. From "Available devices", select the device EP3C25F324C6. (In case of Cyclone IV, the device name is **EP4CE115F29C7**)

12. Click “Finish”



	Status	From	To	Assignment Name	Value	Enabled
1	✓ Ok		clk	Location	PIN_V9	Yes
3	✓ Ok		led[0]	Location	PIN_P13	Yes
6	✓ Ok		led[1]	Location	PIN_P12	Yes
4	✓ Ok		led[2]	Location	PIN_N12	Yes
5	✓ Ok		led[3]	Location	PIN_N9	Yes
2	✓ Ok		reset_n	Location	PIN_N2	Yes

Figure 1. The pin assignment settings for Cyclone III FPGA

13. Select "File” → ”New...", "Block diagram/schematic file" and press "ok"

14. Open the file *work*/hdl_files/vhdl/toplevel.vhdl in Quartus and perform “File” → “Create/Update” → “Create Symbol Files for Current File” in Quartus
    Hint: If the toplevel does not appear or Quartus complains after compiling, it is very probable that you have not added all of the necessary files, go to files tab in Project Navigator window and try to add the all files.

15. Double-click on the drawing canvas in Block1.bdf

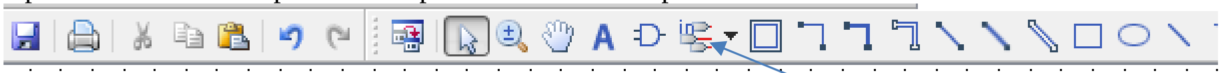
16. From "Project" select "toplevel" and press "ok"
Note: This block is actually your processor’s core, in the next steps you connect clock signal, reset signal, etc and provide memory blocks to be used as instruction and data memory.

17. You can now place the TTA processor on the canvas. Put in the top-middle area.

Adding the VHDL files to the project here enables the Quartus compiler to find the implementations of TTA processor parts.

18. Double-click on the empty drawing canvas

19. Open "/home/user/...->primitives->pin" and click on "input" and "ok"



20. Place the symbol close to the "clk" input of the processor

This button

21. When the just created pin is selected (surrounded in blue), press ctrl-c to copy it.

22. Press ctrl-v to paste and place the copy of the pin just below the previous one.

23. By clicking on the names of the pins, change their names to "clk" and "reset_n"

24. Connect the "clk" pin with "clk_0" of the processor and "reset_n" with "reset_n"

Note: As the name indicates "clk" signal provides clock to the processor and "reset", resets the processor. The reset can be connected to a button just like any other processor.

pc_init is a dummy component that provides the first instruction memory address, where the processor starts computing after a reset. In this implementation it provides a zero.

25. Just like in steps 18-19, create an output pin and place it close to "fu_output_ext_data" .

Hint: The location of processor pins might be slightly different from what is depicted in the figure. Try to match using pin name and width.

26. Rename this pin as "led[3..0]" (yes, two dots) and connect it like in Figure 2. The connections can be named by right-clicking the line and selecting *properties*

27. Select "File" → "Save project" and name the as "tta", like Quartus proposes. Save the file to *work*/quartus.

28. In the button bar on the top, press the button. Here we assign the drawn pins to their physical counterparts

29. Set the values of the chart so that it matches Figure 1.

Note: "clk" is connected to FPGA's clock source, "reset_n" is connected to CPU_Reset button on the board and "led"s are connected to blue LEDs on the board.

30. Click "File" → "Save Project" and close the Assignment Editor

31. Double-click on the canvas

32. Open "project", click on "pc_init" and "ok"

33. Place and connect the component as you can see in Figure 3. **Set the width parameter to 7.**

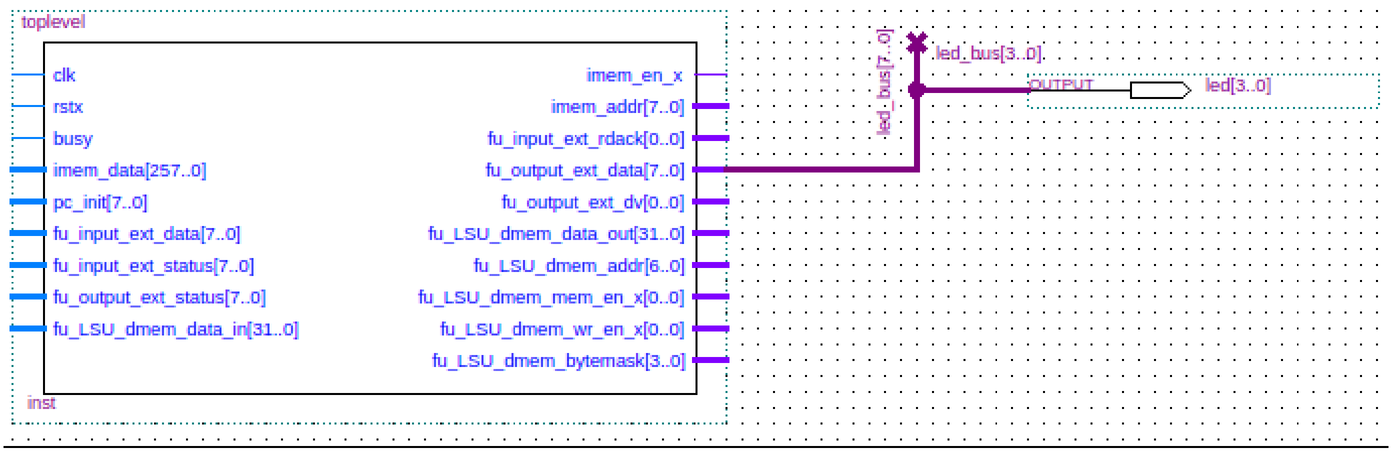


Figure 2.

Part 4: Working with Quartus II: instantiating memories

In this section we will add memory (data and instruction) to our processor.

1. Select from the menu bar “Tools”→”Megawizard plugin manager”

2. Select “Create new …”

3. From the left, select “Memory compiler” and “RAM: 1-port”

4. Select "VHDL" and set the name to "*work*/quartus/dataram"

5. Click "next"

6. "How wide should the q output bus be?" → "32".

7. Set "How many 32-bit words of memory?" to "64". Click "next". Note: You have a memory with 32-bit words and 64 entries. This has been set up in **prode** software.

8. Check the "Create a byte enable for port A"

9. **In the box "which ports should be registered", make sure “q” is *not* checked**

10. Press "next" two times so that you arrive to the "Mem init"-tab

11. Select "Yes, use this file for the memory content data"

12. Click Browse and select the file *work*/*program*_data.mif to the field Note: You have generated this file

13. Click on "finish", place a checkmark on .bsf file and press “finish” again.

14. Answer ”Yes”

15. Place the memory component below the processor, as in Figure 3.

16. Right-click on datamem and select "flip horizontal"

17. Repeat steps 1 to 16, but set the component name to "instrom", "ROM: 1-port", *q* width to *imem_data* width, size to 256 words. The memory initialization file is "../*program*.mif". Remember to perform step 9!

When you connect components with each other in Quartus, make sure you have the same type of connection as in the Figures of this paper (thin or thick). The width of the type can be changed by right-clocking the line and selecting *node line* or *bus line*

Note: The program mif file was generated in previous section of this tutorial.
Since here, the instructions are always are read, we use Read Only Memory as the instruction memory.

Note: If you have different address settings in **prode** and different number of buses, sockets, etc., your instruction width would be different.

18. Double-click on the canvas

19. Create a "not" component from "/home/user...->primitives->logic"

20. Place "not" to the right of dataram and flip horizontally

21. Save your project.

Part 5: Working with Quartus II: instantiating stream devices

1. Double-click on the canvas and instantiate components ***stream_source_1*** and ***checksum*** from *Project*.

2. Make all connections as in Figure 3.

3. Edit the parameter **cntr_limit of checksum** so that it matches the number of samples in input data.

4. Save your project.

5. Double-click on stream_source_1 to open the component.

6. Edit the **cntr_limit paramer of addrgen** and set it to the number of samples in input data.

7. Save the diagram and close it.

Part 6: Configuring SignalTap

Note: Signal Tap is a tool in Quartus software to investigate selected locations in the design in real time. We will use this tool to verify our processor's functionality.

1. Perform *analysis & synthesis* to the project by pressing button on the toolbar.

2. If you receive errors, ask for help or try to figure out the reason.

3. If the analysis & synthesis ends without errors, open "Tools" → "SignalTap II Logic Analyzer"

4. Set all Signaltap options as in Figure 4.

5. Select "File" → "Save" in SignalTap

Part 7: Working with Quartus II: Compiling the project

1. Hit the button to start the full synthesis and placement

Part 8: Working with Quartus II: setting the clock constraints

1. Start TimeQuest by clicking "Tools" → "TimeQuest Timing Analyzer"

2. In the TimeQuest command-line prompt write

   - create_timing_netlist -model slow
   - create_clock -period 20 -name **clk** [get_ports **clk**]
   - write_sdc -expand "**tta**.sdc"

   If needed, replace "clk" with the name of your clock input pin and "tta" with name of your design
   Period 20 means that the clock has a 20 ns period = 50 MHz.

3. Close TimeQuest

4. Perform full compilation again

Part 9: Working with Quartus II: programming the FPGA

1. You need to connect the FPGA board now.

2. Turn on the board power by pressing the blue switch on the circuit board

3. Make sure that the FPGA is connected to VMWare Linux by looking at the blue pane on top of your screen (it might be hidden), which says "ASSP …". If "Virtual Machine" → "Removable devices" →"Altera usb blaster" has a checkmark, everything is ok. Otherwise, select "connect (disconnect from host)" behind "Altera usb blaster".

4. On the Quartus button bar, press the button that is 4th from the right .

5. If USB blaster is not mentioned in the emerging window, press "hardware setup" and enable USB blaster.

6. **Press down the CPU_RESET button on the FPGA board and keep it pressed until part 10.4 in these instructions**

7. Add tta.sof from *work*/output_files by clikcing on "AddFile…"

8. Click on "Start"

Part 10: Checking functionality

1. Open SignalTap, if it is not running. On the right-top edge, select the Hardware as USB-Blaster

2. Press the button.

3. The Status of Signaltap changes to "waiting for trigger".

4. **You can release the CPU_RESET button on the FPGA now**

5. By right-clicking on the signal name, set the data display format to decimal from "bus display format"

6. Check the correctness of the processor operation by checking the last value of "checksum:inst|sum" and comparing it to the sum that you get from running the *typeout* software from the command prompt.

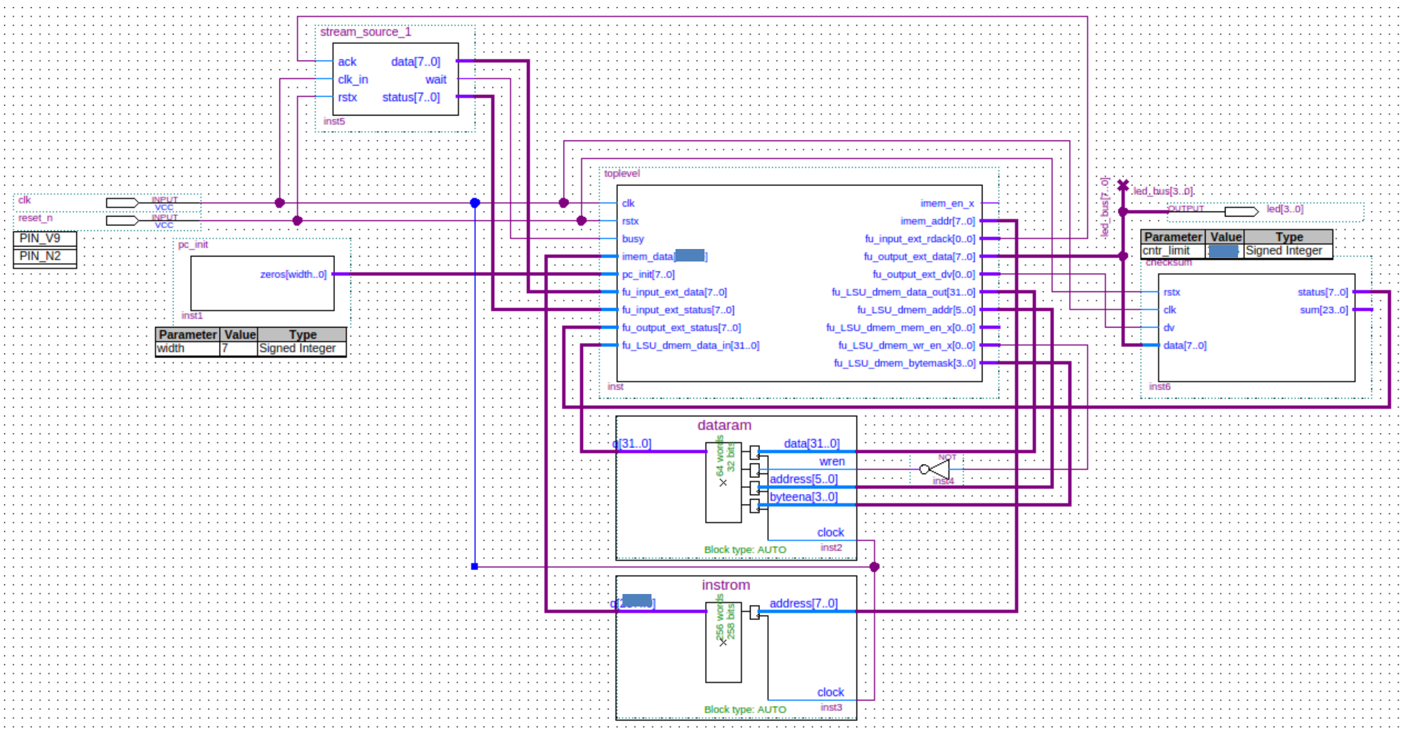
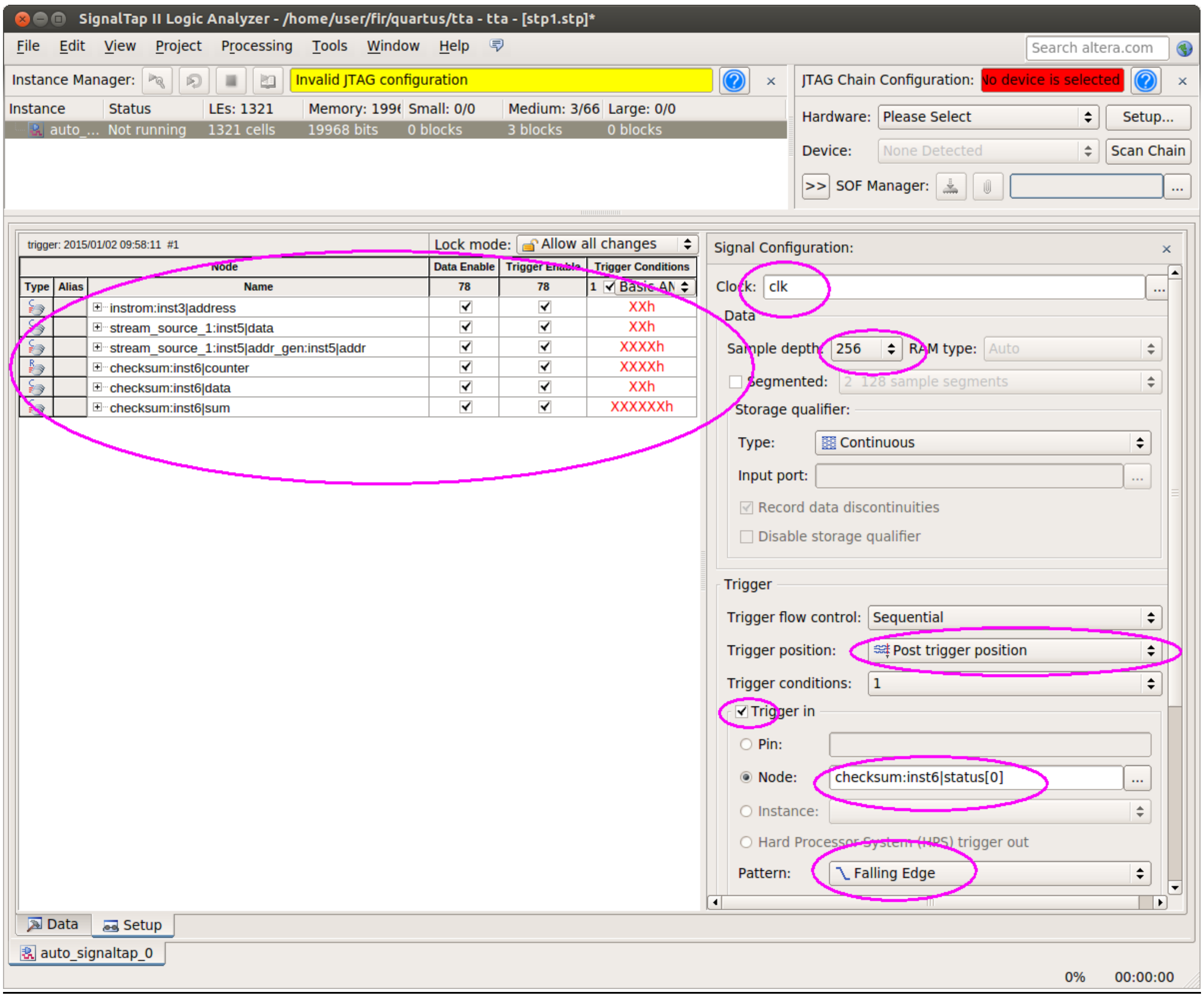


Figure 3.


SignalTap II Logic Analyzer - /home/user/fir/quartus/tta - tta - [stp1.stp]*

Instance Manager: Invalid JTAG configuration

JTAG Chain Configuration: No device is selected

Instance	Status	LEs: 1321	Memory: 199(	Small: 0/0	Medium: 3/66	Large: 0/0
auto_...	Not running	1321 cells	19968 bits	0 blocks	3 blocks	0 blocks

Hardware: Please Select

Device: None Detected

SOF Manager:

trigger: 2015/01/02 09:58:11 #1

Lock mode: Allow all changes

Type	Alias	Name	Data Enable	Trigger Enable	Trigger Conditions
			78	78	1 ✓ Basic AN
		instrom:inst3|address	✓	✓	XXh
		stream_source_1:inst5|data	✓	✓	XXh
		stream_source_1:inst5|addr_gen:inst5|addr	✓	✓	XXXXh
		checksum:inst6|counter	✓	✓	XXXXh
		checksum:inst6|data	✓	✓	XXh
		checksum:inst6|sum	✓	✓	XXXXXXh

Signal Configuration:

Clock: clk

Data

Sample depth: 256 RAM type: Auto

Segmented: 2 128 sample segments

Storage qualifier:

Type: Continuous

Input port:

☑ Record data discontinuities

☐ Disable storage qualifier

Trigger

Trigger flow control: Sequential

Trigger position: Post trigger position

Trigger conditions: 1

☑ Trigger in

○ Pin:

● Node: checksum:inst6|status[0]

○ Instance:

○ Hard Processor System (HPS) trigger out

Pattern: Falling Edge

Data Setup

auto_signaltap_0

0% 00:00:00

Figure 4.

Changing the software

If you want to change the C/asm code and run the new code on the FPGA, you have to perform the following steps again after you have saved the C/asm code:

- Part 2	(Compiling C code, generating memory contents)
- Part 7, 9	(Compiling VHDL and programming the FPGA)
- Part 10	(Redoing logic analysis)

Changing the hardware

- When you modify the processor, you must perform again Part 1. If you add more transport buses to the processor, the instruction word width changes. You need to update the processor symbol in Quartus as in Part 3.14. You see that the imem_data width changes. Edit the instruction memory component so that the bus width matches.
- When you change to a different signal processing application (IIR/Farrow/adaptive/FIR), you must change the cntr_limit parameters as in Part 5. The right value of the parameter size is the length of the tta_stream_X.in –file, which was 128 for IIR. At the same time, you must copy and overwrite the data .mif files tta_stream_in_X.mif. Take the correct files from Desktop/fpga_files/”filter”/