Report of experience with Vivado HLS and SDSoC in the summer internship at IIT Bombay

Sumit Kumar Yadav

22 May - 8 July, 2017

Past Experience : Worked with Xilinx ISE to design hardware by coding in Verilog

I arrived at IIT Bombay on 22nd May,2017 and was alloted a room in Tansa House. I first got familiarized with the campus and the High Performance Computing Lab in the EE Deptt. I got an idea of why verilog is not a good language to code large designs involving algorithms etc. fastly and thus the need for resorting to a high level language for hardware description. I got to know of Vivado HLS, a tool that can generate verilog/vhld files for a C/C++ description. Various pragmas/directives are used to direct HLS synthesize the C/C++ description to a sought for verilog/vhdl description. I started by getting familiar with the HLS environment and design flow from the HLS User Guide https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug902-vivado-high-level-synthesis.pdf . Then I started with creating small examples of addition/multiplicaton and experimenting with the various pragmas. After that I experimented with the behaviour of loops and the pragmas related to them. Then I tried understanding how function calls are made and what is the effect of various pragmas on behaviour of function hardware. I went on to see the interfaces that are generated for function arguments. I also tried replacing the default handshake interfaces by ap_none for simpler functions to remove handshaking if redundant.

1 HLS provides "any precision" libraries

When you need to use a random precision like 13-bit integers in your hardware, using standard 32-bit integers will waste resources. HLS provides you libraries to use any pecision integers(ap_int.h), half floats etc. which save a lot of hardware resources while synthesizing.

2 Use HLS math library instead of standard C/C++ math libraries

The HLS math functions are implemented as synthesizable bit-approximate functions from the hls_math.h library. Bit-approximate HLS math library functions do not provide the same accuracy as the standard C function. To achieve the desired result, the bit-approximate implementation may use a different underlying algorithm than the standard C math library version. So, while checking for errors during C-RTL Co-simulation, make sure that you use "hls_math.h" so that the inherent error doesn't creep in error calculations. Thiis can easily be seen with sin() or log() functions.

3 HLS doesn't check/optimize for cross function hardware redundancies

If you strictly want a function to be instantiated more than once without HLS deciding how many instances of the function to create, you can use the HLS PIPELINE pragma with premeditated II(Initiation Interval) to direct HLS. In some cases where you don't get the hardware you want even with pragmas, you can explicitly write many functions with different names but same definitions. For eg. pe0(),pe1(),pe2(),pe3() etc. all with the body of pe().

HLS doesnot look for cross function redundancies and thus creates seperate hardware instances. Now you can get the hardware design you want easily.

4 II indications are not strictly followed(if smaller II is acheivable)

If you try pipeling a function in HLS by specifying some II(say 16) but the function can be implemented with a smaller II(say 1), HLS will ignore your II directive and synthesize the function with II=1. If you deliberately want this to be 16(or something) you can direct HLS to do so by restricting resources via necessary pragmas. Anyways, the question is why do you want a particular II ? If you are planning so just for matching with the following function's II in the flow which will take the result of this function as arguments, don't try to do so unless you care of resource utilization. But, If the following function works at some high II, then you can always try to increase the preceeding function's II and reduce resource utilization instead of letting it do all calculations in parallel within a few cycles and then sitting idle waiting for the other function to need its results.

Even after this, if you want a function to be of a particular latency and II without really using resource restrictions to shape the latency and II, you can tell HLS to implement it with a particular latency by using the latency directive. Now, if you direct HLS to synthesize with a larger latency than required, the calculations will still happen in only a few cycles that are actually needed and some dummy states will be added before them to give you your required latency. The required pragma here is : #pragma HLS LATENCY min=16 max=16 (this will give you a strict latency of 16 even without any resource restrictions)

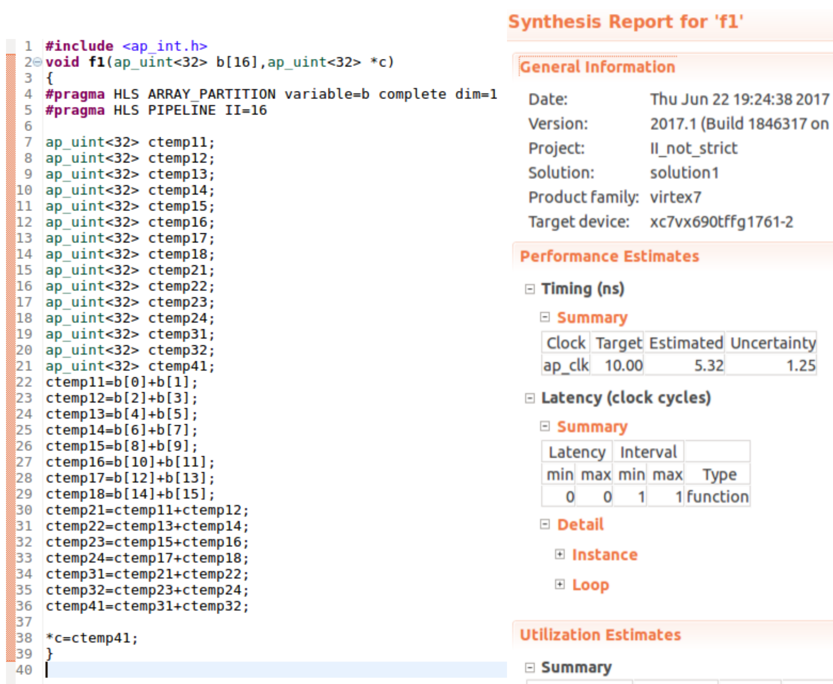

Figure 1: Example of II specification being neglected by HLS

5 DATAFLOW pragma

Dataflow pragma is commonly used to parallelize a flow of functions which are in a dataflow. Say two functions f1 and f2 are in a dataflow. The pragma will establish a fifo data connection between the two functions so that the f2 can start working on some of the data produced by f1 while f1 still functioning to produce rest of the data. This behaviour lets data "seep" into a design rather than being passed on in batches(the default behaviour of C/C++) from one function to another. Now this directive is different from pipeling as it lets a batch of data seep in a design to produce output from the last function in dataflow but doesn't allow to pump in new batches in the flow from top until the final output corrsponding to earlier input is produced by the last function in design flow. This pragma does parallelizes processing but not completely.

6 Results can't be streamed out of any argument of a function in a single call.

By the definition of a function in C++, functions can't keep changing their pass by reference arguments or return values in a single call while you are reading them in the caller function. This can prove to be a drawback while using C or C++ for describing hardware using HLS tools. There may be some situations where you want to create some hardware which continuously takes input and streams output. Although there is a trick for implementing this, you need to write the function calls in a loop and make sure that only one instance of the function is created in hardware. Then use pipeline directive in the top function. What it does essentially is create extra hardware(proportional to loop count) implementing MUXes at the input of the first function in dataflow and sending different inputs as required in the loop iterations by selecting from the inputs from the MUX. Although this may do your job for small loop counts but this trick is quite inefficient for large loop counts as extra hardware is created per extra loop count thus wasting resources and decreasing the clock frequency significantly.

```
void top(ap_uint<32> a[32],ap_uint<32> c[2])
{
//#pragma HLS DATAFLOW/PIPELINE
#pragma HLS INTERFACE ap_fifo port=a
#pragma HLS INTERFACE ap_fifo port=c
    ap_uint<32> b[16];
    #pragma HLS ARRAY_PARTITION variable=b complete dim=1
    for(int ctr=0;ctr<2;ctr++)
    {
    f1(a+ctr*16,b);
    f2(b,c+ctr);
    }
}
```

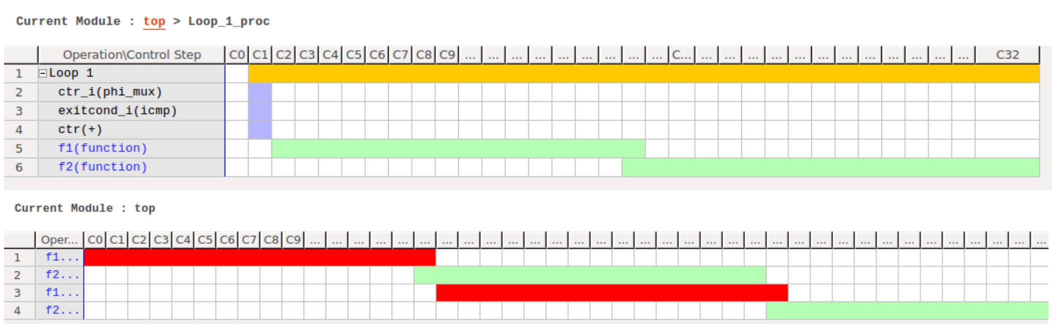

Figure 2: Using a loop to achieve streaming output from a function instance in hardware: 1. DATAFLOW pragma 2. PIPELINE pragma (the trick !)

7 Datamovers in SDSoC

To transfer data between PL(programmable logic) and DDR3(or any external memory), various datamovers can be used depending on the type and size of arguments of the hardware function. If the argument is small enough, it is firstly copied to a register or BRAM in PL via AXI and then further used. But if the argument is quite large(eg a large sized array), the user needs to specify pragmas like zero_copy or SEQUENTIAL access to form a streaming interface implemented by AXI_MASTER/AXI_SIMPLE/AXI_SCATTER-GATHER depending on the type of allocation(paged or contiguous). Using malloc() to allocate memory for an array may lead to a large number of page changes and other overheads while memory allocated by sds_alloc() is mostly on page boundaries and thus overheads are reduced. Thus, a sequential acces on an array allocated with sds_alloc() can be implemented with AXI_SIMPLE whereas the same array, if allocated using malloc(), uses AXI_SCATTER-GATHER. The following table shows the datamover implementation behaviour of SDSoC while implementing a function which simply copies a float array to another array using a hardware implemented function.

Size	Allocation	Pragma	Estimated H/W cycles	Generated Interface
2000	malloc()	-	44876	AXI_ACP:AXIDMA_SG
		zero_copy	105191	AXI_ACP:AXIMM:0xC
		SEQUENTIAL	88010	AXI_ACP:AXIDMA_SG
	sds_alloc()	-	59261	AXI_ACP:AXIDMA_SIMPLE
		zero_copy	58185	AXI_ACP:AXIMM:0xC
		SEQUENTIAL	19284	AXI_ACP:AXIDMA_SIMPLE
20000	malloc()	zero_copy	1046951	AXI_ACP:AXIMM:0xC
		SEQUENTIAL	364785	AXI_ACP:AXIDMA_SG
	sds_alloc()	zero_copy	537753	AXI_ACP:AXIMM:0xC
		SEQUENTIAL	139252	AXI_ACP:AXIDMA_SIMPLE

Table 1: Datamover dependence of a float array copying H/Wfunction on size, allocation and access pattern.

It is very clear that the bandwidth that can provided by the AXI from off chip memory to PL proves to be the most critical bottleneck in any implementation on SoC's like zedboard etc.

8 Measuring performance with programs involving r/w from RAM

Reading and writing from RAM(DDR3) present on the zedboard or any other SoC board is not completely deterministic at higher levels. This variation in performance comes into picture due to many factors including allocation strategy (random, paged or contiguous), and the random distribution of data in the

RAM leading to uneven access times. Best is to run the program for a large number of times and take an average of the execution times.

9 RAM read and write latencies for various cases.

I many hardware designs, external memory accesses prove to be the bottleneck in performance and thus, it is very critical to analyze the data access of external memory from a Programmable Logic(PL) IP. I wrote a simple memory copying function : memcpy() which simply copies contents of an array to another array. The only arguments are the arrays and a for loop does the copying. First of all, the arguments to memcpy() should be arrays and not just pointers so that the depth is known to the function at compile time. The variuos ways to carry out this copying are discussed below :

9.1 Using sds_zerocopy() pragma

This pragma specifies that the array should be shared by both software and hardware without any copying. This directs SDSoC to build a AXI_ACP:AXIMM (AXI Master) Interface which can be visible in the block diagram.

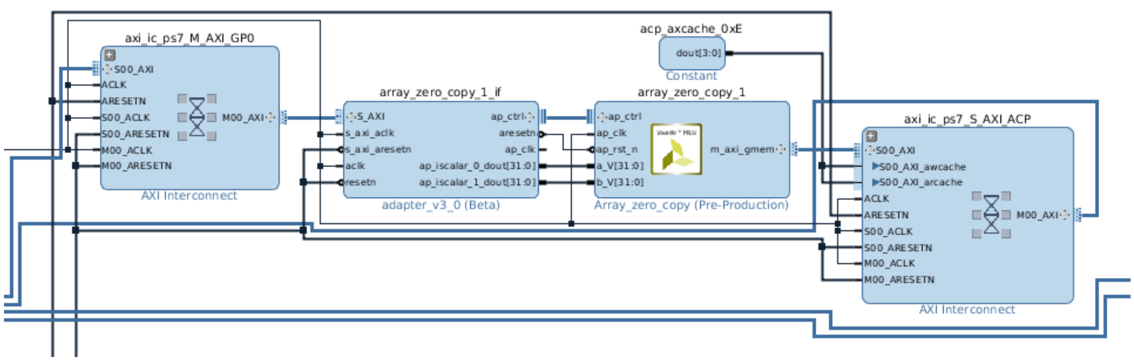

Figure 3: Block diagram for AXI Master used for array copying

For copying 1000 ap_uint<32> elements, 2.60 H/W cycles per copy was achieved which improved to 2.01 for 100000 elements on an average. As the performance is saturated at around 2 cycles per copy, this is not a good way for simply copying array elements.

9.2 Using HLS ap_fifo or SDS SEQUENTIAL access pragma

This pragma directs HLS that both the arrays will be FIFO's and as no other pragma is given to SDSoC, they are implemented as AXI_SIMPLE interfaces as visible in the block diagram.

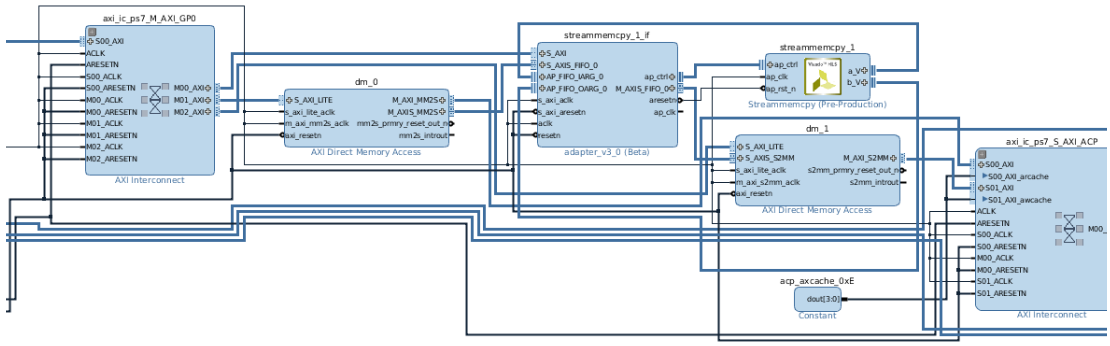

Figure 4: Block diagram for AXI SIMPLE used for array copying

For copying 1000 ap_uint<32> elements, 1.9 H/W cycles per copy was achieved which improved to 1.17 for 100000 elements on an average. The best performance in 100000 iterations was 1.09 clock cycles per copy. Building with HLS SEQUENTIAL access pragma also generates similar hardware and shows similar performance.

10 Managing GPIO from software

While using SoC boards like zedboard etc., there may be some cases where you want to read from or write to the general purpose input output pins on the board such as LED's, switches, buttons etc. to take some flow deciding inputs or using LED's to see progress in flow. Usually these pins are directly managed by kernel modules but there is an easy way to manage these pins also from user space.

Standard Linux kernels have inside them, a special interface to allow access to GPIO pins. After executing kernel menuconfig you can easily verify whether this interface is active in your kernel or not and in case, enable it. The kernel tree path is the following:

Device Drivers —> GPIO Support —> /sys/class/gpio/... (sysfs interface)

If not, enable this feature and recompile the kernel before continuing to read. The interface to allow working with GPIO is at the following filesystem path:

/sys/class/gpio/

If you want to work with a particular GPIO you must first reserve it, set the input/output direction and start managing it. Once you reserve the GPIO and finish to use it, you need to free it for allowing other modules or processes to use it. This rule is valid for both cases : use GPIO from kernel level or user level.

10.1 Manage GPIO from command line or script

From the user level side this "operation" for reserving the GPIO is called "export". For making this export operation, you simply need to echo the GPIO number you are interested to a special path as follows (change XX with the GPIO number you need):

echo XX > /sys/class/gpio/export

If operation is successful (the possible case of operation failed is explained below) a new "folder" will show up in the GPIO interface path as shown below:

/sys/class/gpio/gpioXX/

This new "folder" will allow you to work with the GPIO you just reserved. In particular if you want to set the in/out direction you simply need to execute the following echo commands:

echo "out" > /sys/class/gpio/gpioXX/direction

or

echo "in" > /sys/class/gpio/gpioXX/direction

In case you set “out” direction, you can directly manage the value of GPIO by executing additional echo commands like:

echo 1 > /sys/class/gpio/gpioXX/value

or

echo 0 > /sys/class/gpio/gpioXX/value

The possible states allowed are high (1) and low (0). In case you set “in” direction, you can read the current pin value by using the following command:

cat /sys/class/gpio/gpioXX/value

Once finished to use your GPIO, you can free it by making the following echo command :

echo XX > /sys/class/gpio/unexport

If GPIO folder did not showed up after export operation, it is very likely that the GPIO is already reserved by some module. For verifying the current reserved GPIO map, you must first verify if your kernel has the following feature enabled:

Kernel configuration —> Kernel hacking —> Debug FS

As usual, if not enabled, enable it and recompile the kernel. The next step is to launch the following commands for mount debugfs:

mount -t debugfs none /sys/kernel/debug

and dump the current GPIO configuration by using:

cat /sys/kernel/debug/gpio

The output will show you the current list of reserved GPIOs.

10.2 Manage GPIO from an application(or simply C/C++) code

Following short lines of C code show how the reproduce the same steps as above (remember to change XX with the GPIO number you want to use).

10.2.1 Reserve (export) the GPIO:

```
int fd;
    char buf[MAX_BUF];
    int gpio = XX;
    fd = open("/sys/class/gpio/export", O_WRONLY);
    sprintf(buf, "%d", gpio);
    write(fd, buf, strlen(buf));
    close(fd);
```

10.2.2 Set the direction in the GPIO folder just created:

```
sprintf(buf, "/sys/class/gpio/gpio%d/direction", gpio);
    fd = open(buf, O_WRONLY);
    // Set out direction write(fd, "out", 3);
    // Set in direction write(fd, "in", 2);
    close(fd);
```

10.2.3 In case of “out” direction, set the value of GPIO:

```
sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);
    fd = open(buf, O_WRONLY);
    // Set GPIO high status write(fd, "1", 1);
    // Set GPIO low status write(fd, "0", 1);
    close(fd);
```

10.2.4 In case of “in” direction, get the current value of GPIO:

```
char value;
    sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);
    fd = open(buf, O_RDONLY);
    read(fd, &value, 1);
    if(value == ’0’) { // Current GPIO status low } else { // Current GPIO
status high }
    close(fd);
```

10.2.5 Once finished, free (unexport) the GPIO:

```
fd = open("/sys/class/gpio/unexport", O_WRONLY);
    sprintf(buf, "%d", gpio);
    write(fd, buf, strlen(buf));
    close(fd);
```

10.2.6 NOTE : You have to keep this in mind if you plan to set or, more important, get the value of a GPIO through this way in continous mode. If you open the "value" file for getting the current GPIO status (1 or 0) remember that, after the fist read operation, the file pointer will move to the next position in the file. Since this interface was made to be read from cat command, the returned string will be terminated by the new line character (\n). This mean after the first "valid" read all the next read operation will return always the last character in the file, in this case only the new line '\n'. For obtaining a correct status value for each read operation you simply have to set the file pointer at the beginning of the file before read by using the command below:

lseek(fp, 0, SEEK_SET);

You will not have this problem if you open and close GPIO value file every time you need to read it, but such continuous read may introduce short delays. Since these short lines of codes are only an example if you want to use them in your code remember to add control for error in opening the GPIO file.

11 Changing/Tweaking verilog files generated in sdsoc build flow

We may sometimes be unable to get the desired hardware that we want by writing C/C++ code and using pragmas to direct HLS or SDSoC. So as a last resort, you may feel the need to tweak the generated verilog files or replace them with your (compatible) verilog files. But in SDSoC build flow, sds++ calls vivado_hls to synthesize the C/C++ source files and generate coressponding verilog files and export this as an IP. Then the flow returns to sds++ to use this IP in the corresponding vivado project formation and generating the .elf file(s).

If we observe closely, sds++ generates a .tcl script with commands to include C/C++ source files, synthesize the design and export it as an IP and passes this .tcl file to vivado_hls as an argument. **What we want to do is to somehow halt the vivado_hls action after synthesis and before exporting, replace/tweak the generated verilog files, and then let vivado_hls to continue exporting the IP and returning the flow to sds++.**

A "trick" to do this is to let sds++ call a script written by us named vivado_hls instead of the original vivado_hls. Now when this script is called (or the original vivado_hls that was meant to be called), the .tcl to tell vivado_hls(or the original vivado_hls that was meant to be called) what to do has already been generated. So in our vivado_hls bash script, we replace this auto-generated .tcl by our own .tcl(my.tcl). This replaced .tcl(my.tcl) is almost similar to the auto-generated .tcl except with a bash script(my_verilog_converter.sh) call to pause the flow for sometime between csynth and export commands.

The my_verilog_converter.sh simply creates a dummy file(halt.txt) and then waits until that dummy file is deleted by the user. While this scripts waits, you can tweak/replaced the synthesized verilog files and then delete the dummy file(halt.txt) mannually and let the script finish. After this script finishes, flow is returned to my.tcl which proceeds to export the design as an IP.

```
[sumityadav@localhost ~]$ ls -R /home/sumityadav/Vivado_HLS
/home/sumityadav/Vivado_HLS:
bin  common  data  examples  include  lib  lnx64  scripts  src  tps

/home/sumityadav/Vivado_HLS/bin:
apcc    rdiArgs.sh   unwrapped   vivadohlsArgs.sh    xlicclientmgr  xlictsinit
loader  setupEnv.sh  vivado_hls  vivado_hls_gui.vbs  xlicsrvrmgr
[sumityadav@localhost ~]$ export SDX_VIVADO_HLS=/home/sumityadav/Vivado_HLS
[sumityadav@localhost ~]$
```

Figure 5: Exporting path to use custom vivado_hls script. The exported Vivado_HLS directory here is merely a copy of the original Vivado_HLS directory with the vivado_hls replaced by our custom script.

```
#!/bin/bash

 cp /home/sumityadav/my.tcl /home/sumityadav/smalltests/streammemcpy/Release/_sd
s/vhls/streammemcpy_run.tcl

echo "Executing original vivado_hls : /opt/Xilinx/SDx/2016.3/Vivado_HLS/bin/viva
do_hls"
/opt/Xilinx/SDx/2016.3/Vivado_HLS/bin/vivado_hls $*

echo "Processing / modifying generated verilog files."
exit 0
```

Figure 6: Our script named vivado_hls to mimic as original vivado_hls

```
open_project streammemcpy
set_top streammemcpy
add_files /home/sumityadav/smalltests/streammemcpy/src/streammemcpy.cpp -cflags
"-I/home/sumityadav/smalltests/streammemcpy/src -Wall -O3 -fmessage-length=0 -D
__SDSCC__ -m32 -I /opt/Xilinx/SDx/2016.3/aarch32-linux/include -I/home/sumityada
v/smalltests/streammemcpy/src -D __SDSVHLS__ -I /home/sumityadav/smalltests/stre
ammemcpy/Release -w"
open_solution "solution" -reset
set_part { xc7z020clg484-1 }
# synthesis directives
create_clock -period 10.000000
config_rtl -reset_level low
source /home/sumityadav/smalltests/streammemcpy/Release/_sds/vhls/streammemcpy.t
cl
# end synthesis directives
csynth_design
export_design -acc
exit
```

Figure 7: Original .tcl auto-generated by sds++ for vivado_hls

```
open_project streammemcpy
set_top streammemcpy
add_files /home/sumityadav/smalltests/streammemcpy/src/streammemcpy.cpp -cflags
"-I/home/sumityadav/smalltests/streammemcpy/src -Wall -O3 -fmessage-length=0 -D
__SDSCC__ -m32 -I /opt/Xilinx/SDx/2016.3/aarch32-linux/include -I/home/sumityada
v/smalltests/streammemcpy/src -D __SDSVHLS__ -I /home/sumityadav/smalltests/stre
ammemcpy/Release -w"
open_solution "solution" -reset
set_part { xc7z020clg484-1 }
# synthesis directives
create_clock -period 10.000000
config_rtl -reset_level low
source /home/sumityadav/smalltests/streammemcpy/Release/_sds/vhls/streammemcpy.t
cl
# end synthesis directives
csynth_design
exec /home/sumityadav/my_verilog_converter.sh
export_design -acc
exit
```

Figure 8: my.tcl which replaces auto-generated .tcl

```
#!/bin/bash

echo "Processing / modifying generated verilog files."
touch /home/sumityadav/halt.txt
while [ -e /home/sumityadav/halt.txt ]
do
sleep 5
done

exit 0
```

Figure 9: my_verilog_converter.sh which is called by my.tcl