

SDSoC LABS

by Stefano Tabanelli, Avnet Memec Silica, v1.3

Contents

Requirements	2
L0 Hello World	3
L1 One line code DMA, no PL acceleration.....	7
L1_1 One line code DMA, PL accelerated, non-contiguous memory.....	13
L1_2 One line code DMA, PL accelerated, contiguous memory	20
L1_3 One line code DMA, PL accelerated, zero copy memory	26
L1_4 One line code DMA, PL accelerated, zero copy memory, async tasks.....	28
L2 Two operands function, PL accelerated.....	33
L2_1 Two operands function, PL accelerated, pipelined	40
L2_2 Perf estimation on two operands function, PL accelerated, pipelined	42
L2_3 Perf estimation on two operands function, PL accelerated, pipelined, loop unrolled	44
L3 Matrix floating point multiplication example on Linux	48

Requirements

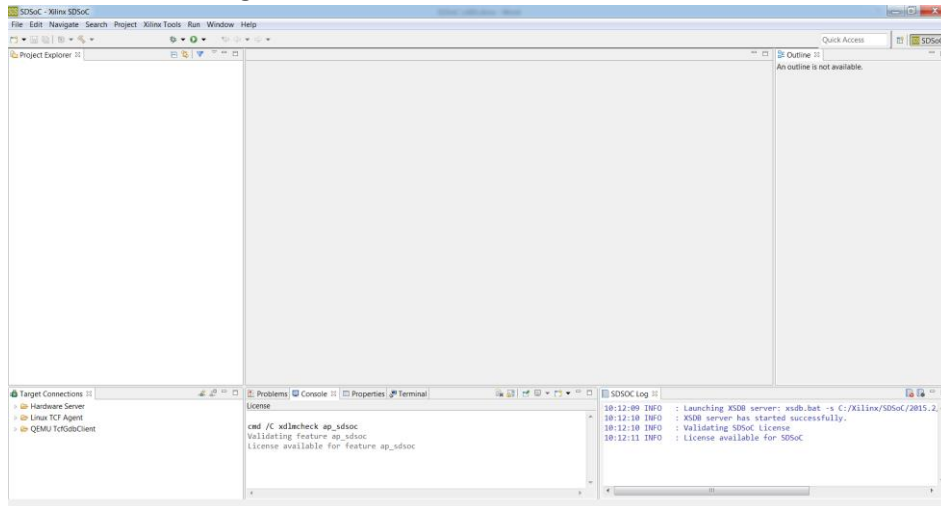
The lab has been tested on the following setup:

- SDSoc 2015.2 installed and licensed on Windows 7
(<http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html>)
- Microzed board (<http://zedboard.org/product/microzed>) equipped with XC7Z010 Zynq SoC
- JTAG cable (<http://www.xilinx.com/products/boards-and-kits/hw-usb-ii-g.html>)
- microUSB cable
- uSD card & card reader (only for the last lab)
- CAT5 Ethernet cable (only for the last lab)

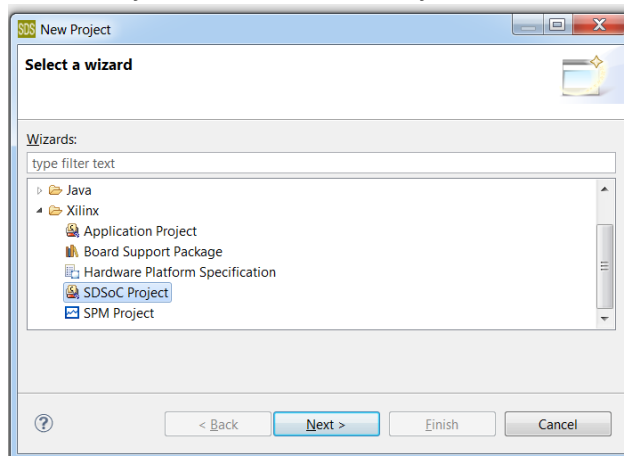
L0 Hello World

During this lab we'll verify that SDSoC environment and the eval board are set up correctly.

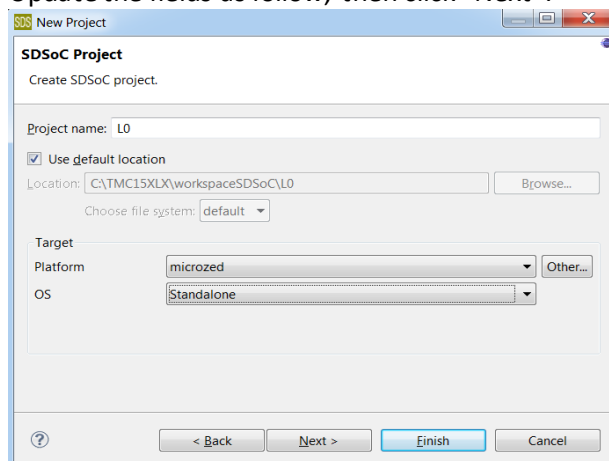
1. Unzip supplied file TMC15XLX.ZIP to C:\
2. Launch SDSoC
3. When asked, set the workspace path to C:\TMC15XLX\workspaceSDSoC
4. Close the "Welcome page"
5. You'll see something like:



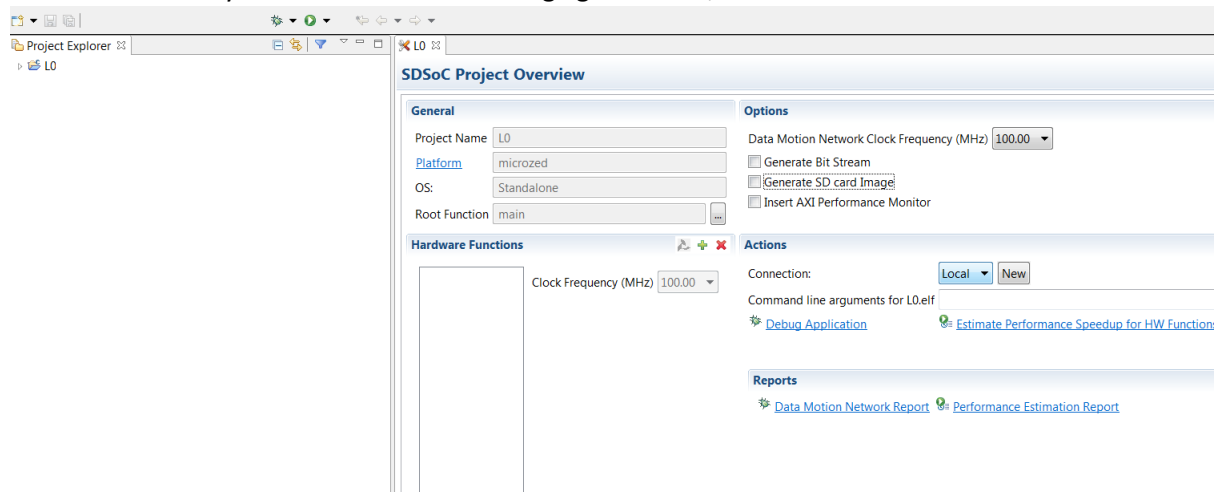
6. Right-click on an empty area in the "Project Explorer" window and select New→Project→Xilinx→SDSoC Project, then click on "Next" button



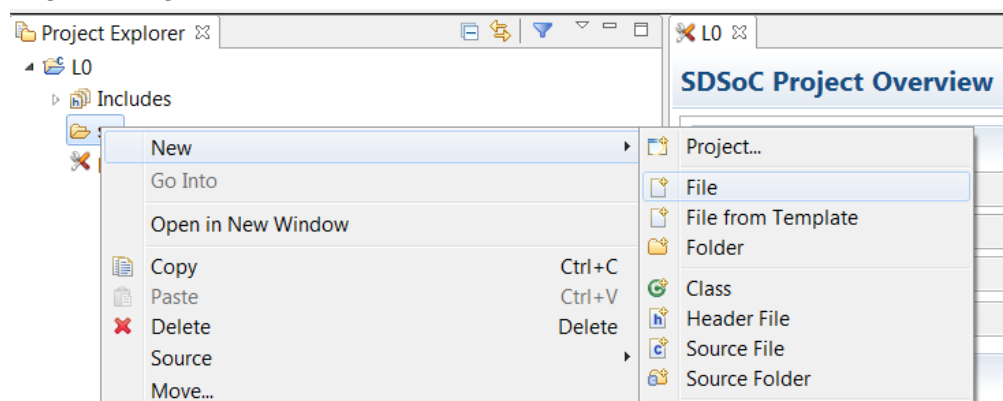
7. Update the fields as follow, then click "Next":



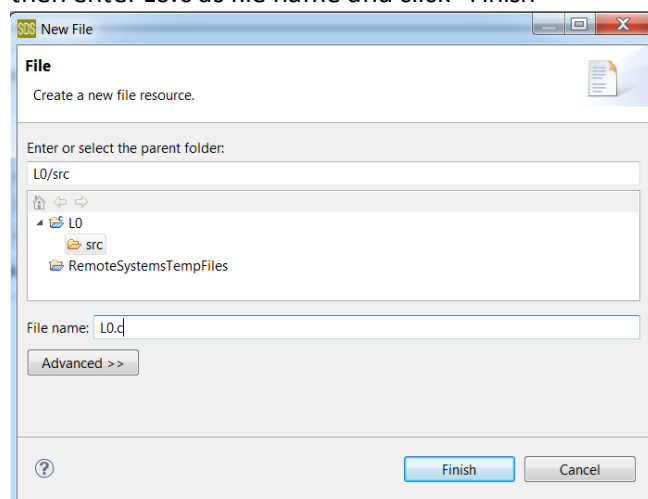
8. Select “Empty Application” then click “Finish”
9. We don’t want any Bitstream or SD card image generation, therefore unselect the two fields



10. Left-click on “L0” to unfold the content, then right-click on “L0” → “src” and select “New” → “File”



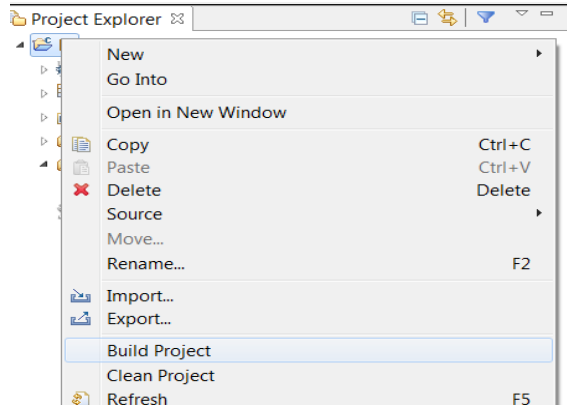
then enter L0.c as file name and click “Finish”



11. Enter (or copy paste) the following source code, then save using CTRL-S:

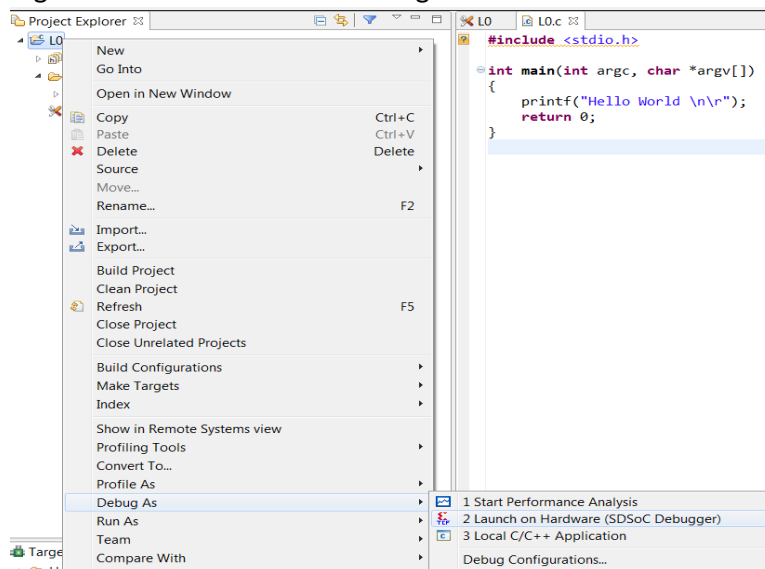
```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("Hello World \n\r");
    return 0;
}
```

12. To build the project right click on L0 and select “Build Project”



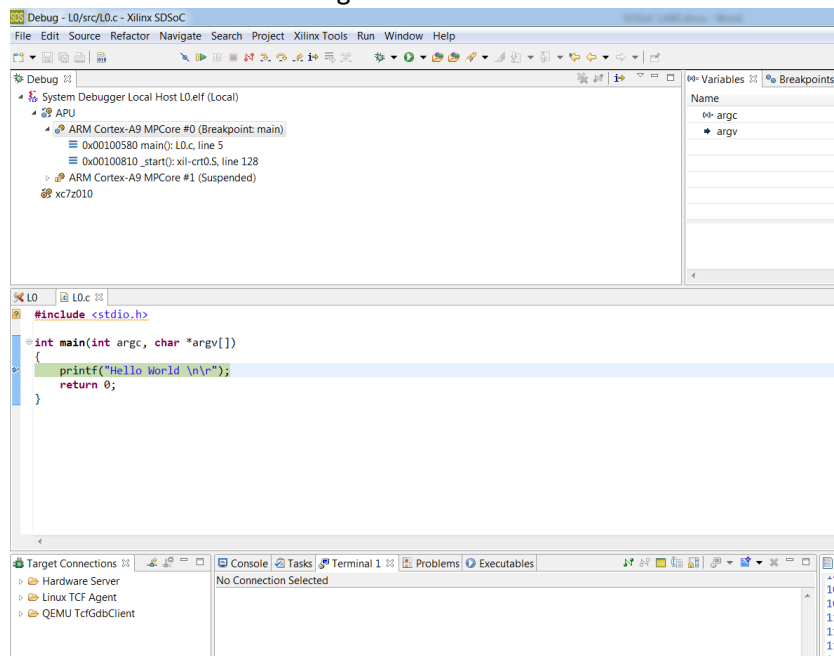
13. Now connect the usb jtag cable and the uzed board via micro usb cable, making sure no uSD is present in the slot


14. Right-click on L0 and select “Debug as” → “Launch on Hardware (SDSoC Debugger)”

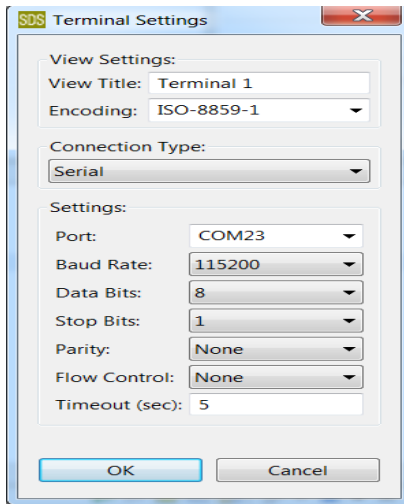


15. If a “Confirm perspective switch” popup appears, select “Remember my decision” and “Yes”

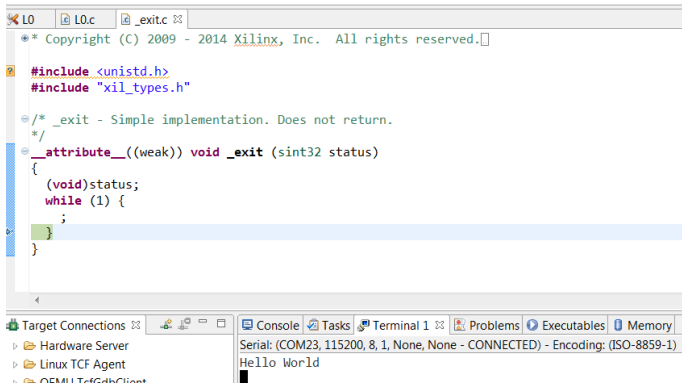
16. You should see the following



17. Select "Terminal 1" window and click on icon  to set serial port parameters, the device number will likely be the highest, in case of doubt use "Windows control panel→System→Device Manager" to double-check Click "OK" and wait for the serial connection to be established



18. Click on icon "Resume"  or press F8, you should see

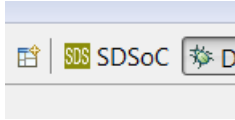


19. This confirms we have a working setup for the next labs

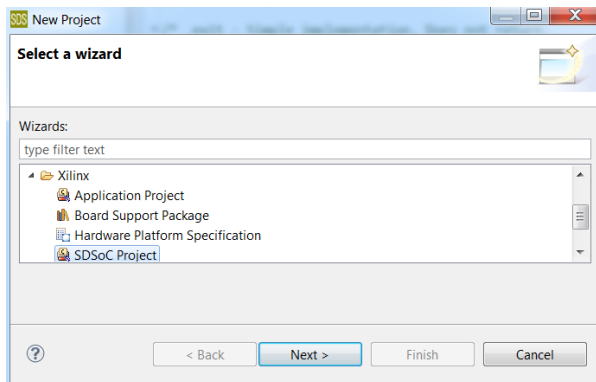
L1 One line code DMA, no PL acceleration

During this lab we'll implement a DMA function able to copy data between a source and a destination address. Initially we'll use no Programmable Logic but only simple ANSI C, to be sure the code is correct and working, therefore in this context SDSoc will act like XSDK tool.

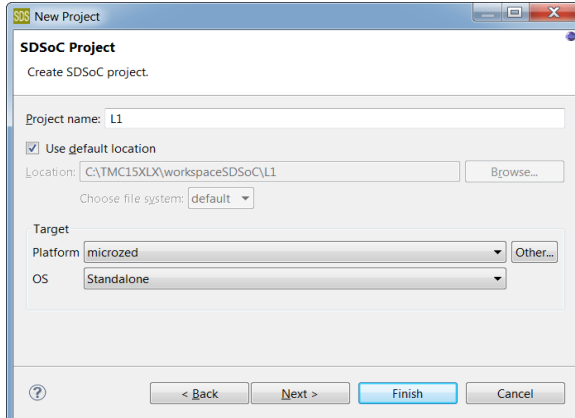
1. SDSoc should be still opened on Debug Perspective, switch back to C/C++ Perspective clicking the upper right button "SDSoC"



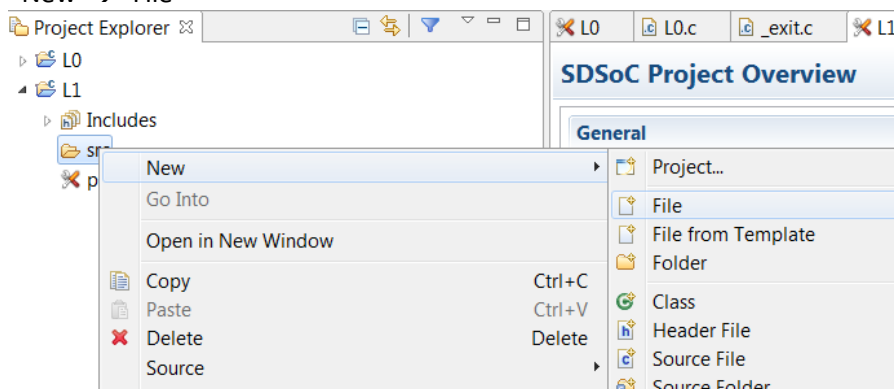
2. Right-click on an empty area in the "Project Explorer" window and select "New" → "Project" → "Xilinx" → "SDSoC Project", then click on "Next" button



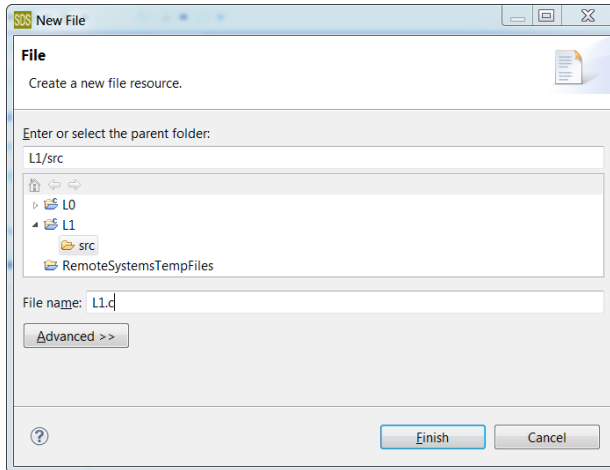
3. Enter L1 as project name, then click "Next"



4. Select "Empty Application" then click "Finish"
5. Left-click on L1 to unfold the content, then right-click on "L1" → "src" and select "New" → "File"



6. Enter L1.c as file name, then click “Finish”

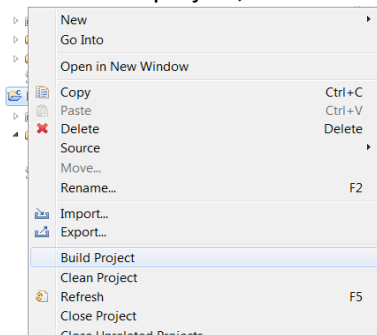


7. Type (or copy-paste) the following code, then save using CTRL-S

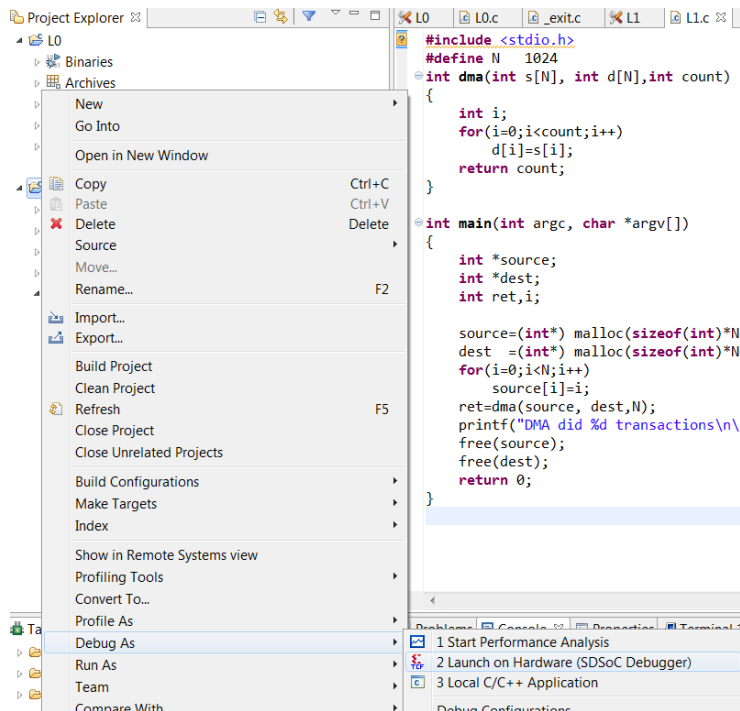
```
#include <stdio.h>
#include <stdlib.h>
#define N 1024
int dma(int s[N], int d[N],int count)
{
    int i;
    for(i=0;i<count;i++)
        d[i]=s[i];
    return count;
}
int main(int argc, char *argv[])
{
    int *source;
    int *dest;
    int ret,i;

    source=(int*) malloc(sizeof(int)*N);
    dest =(int*) malloc(sizeof(int)*N);
    for(i=0;i<N;i++)
        source[i]=i;
    ret=dma(source, dest,N);
    printf("DMA did %d transactions\n\r",ret);
    free(source);
    free(dest);
    return 0;
}
```

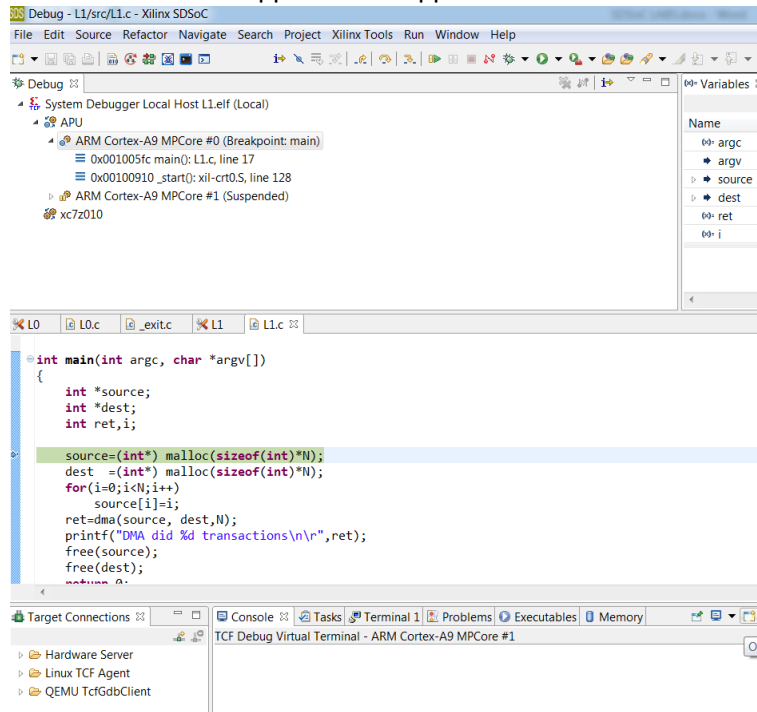
8. Analyse the code, it should be easy to understand that we’re allocating two integer vectors (1024 elements) from the dynamic memory (heap), initializing the source with a known value, then invoking function dma to copy the source vector into the dest
9. To build the project, left-click on L1 then right-click and select “Build”



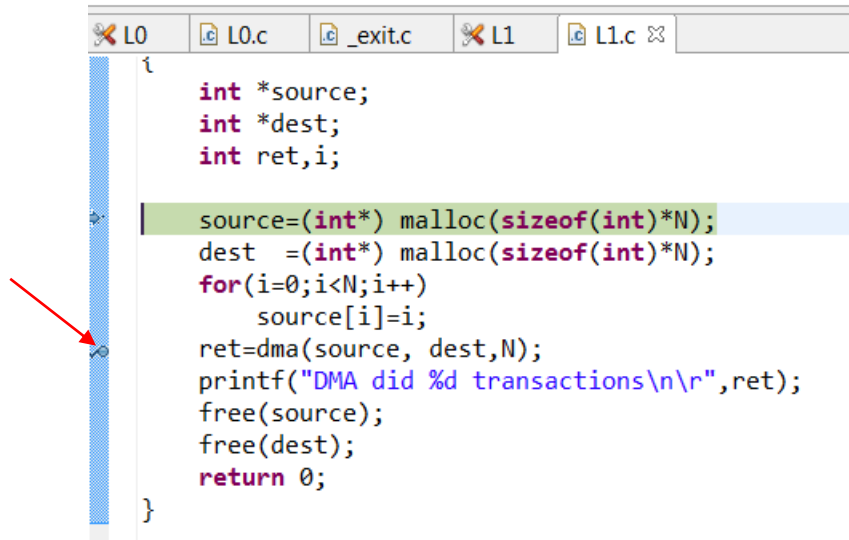
10. If the compilation is correct, right-click on “Project Explorer”→”L1” then select “Debug as”→”Launch on Hardware(SDSoC Debugger)”

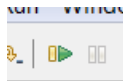


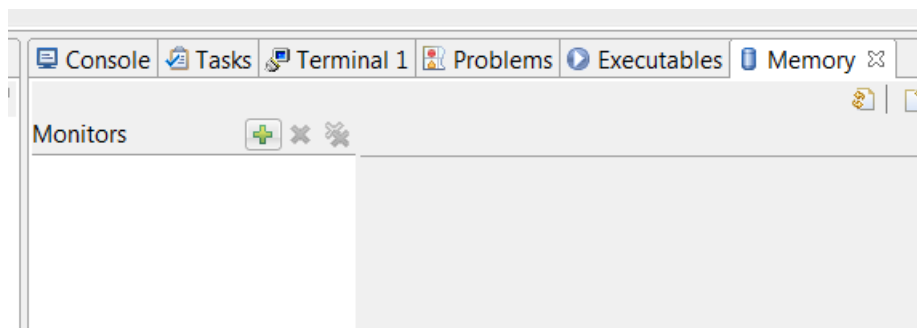
11. You should see the application stopped at the main:



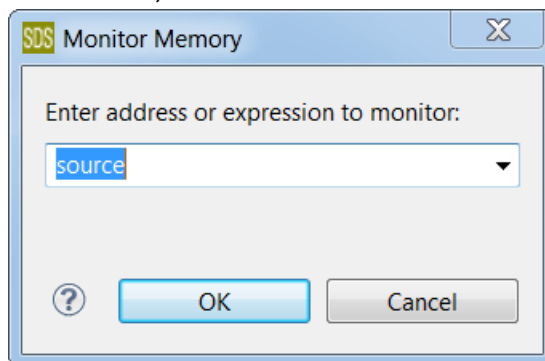
12. Set a break point by double left clicking on “dma” function call (on the blue column)



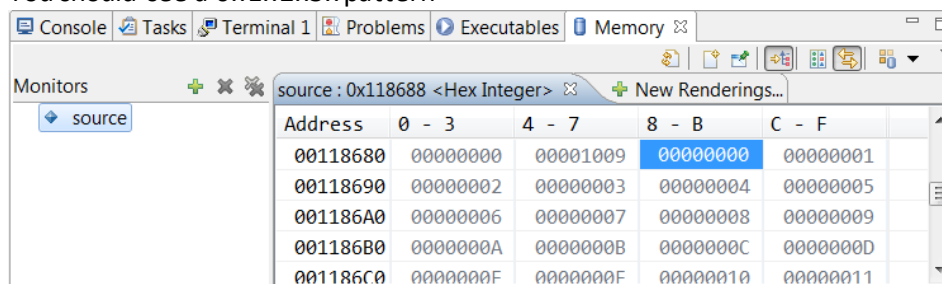
13. Now resume the execution clicking on  or pressing key F8
 14. To see source and dest content, select the “Memory” window on the screen bottom and click “+”



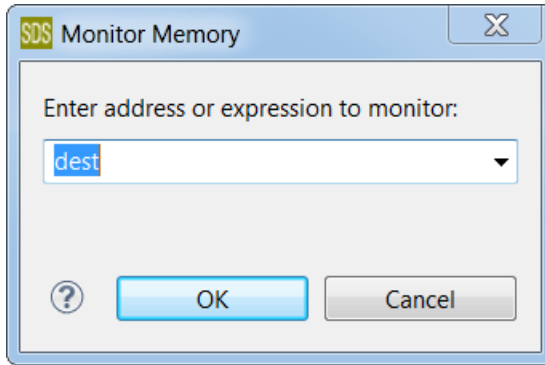
15. When asked, insert “source”



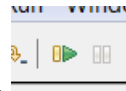
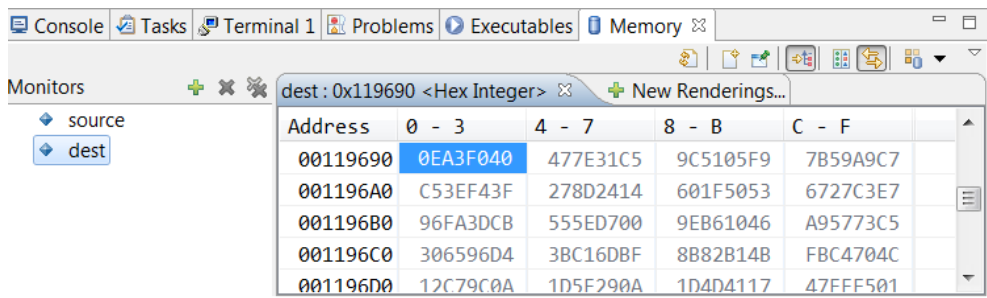
16. You should see a 0..1..2..3.. pattern



17. Click again on “+” and when asked insert “dest”

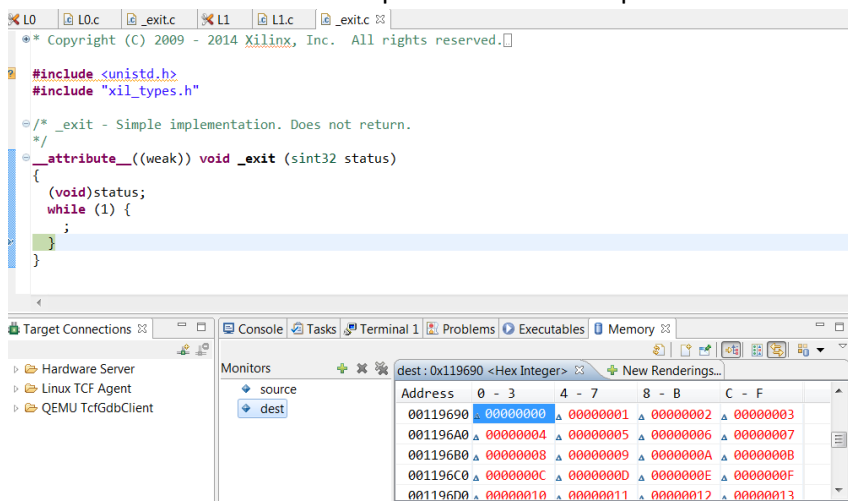


18. You should see a random content



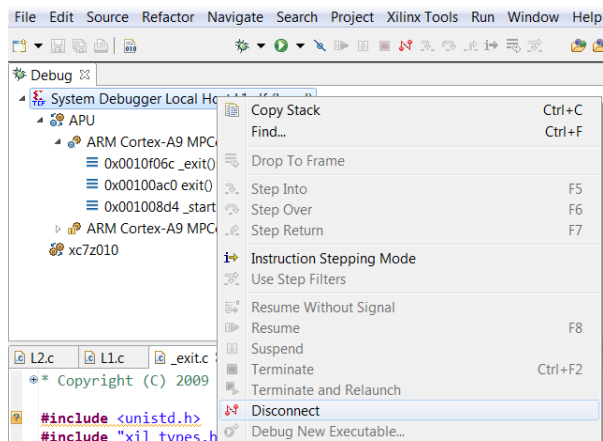
19. Now resume again the execution clicking on [play button icon] or pressing key F8

20. You should see that the source pattern has been copied to dest

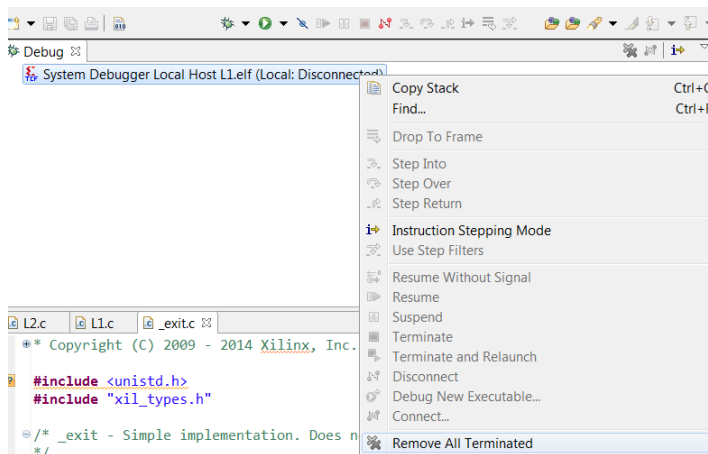


21. This confirm the code is working

22. To close the debug session, right click on the upper left “System Debugger Local Host” and select “Disconnect”



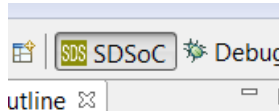
23. Right click on the upper left “System Debugger Local Host” entry then select “Remove All Terminated”



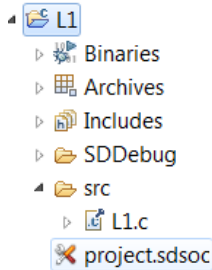
L1_1 One line code DMA, PL accelerated, non-contiguous memory

During this lab we'll use the previous L1 project and we'll move function dma to the Programmable Logic.

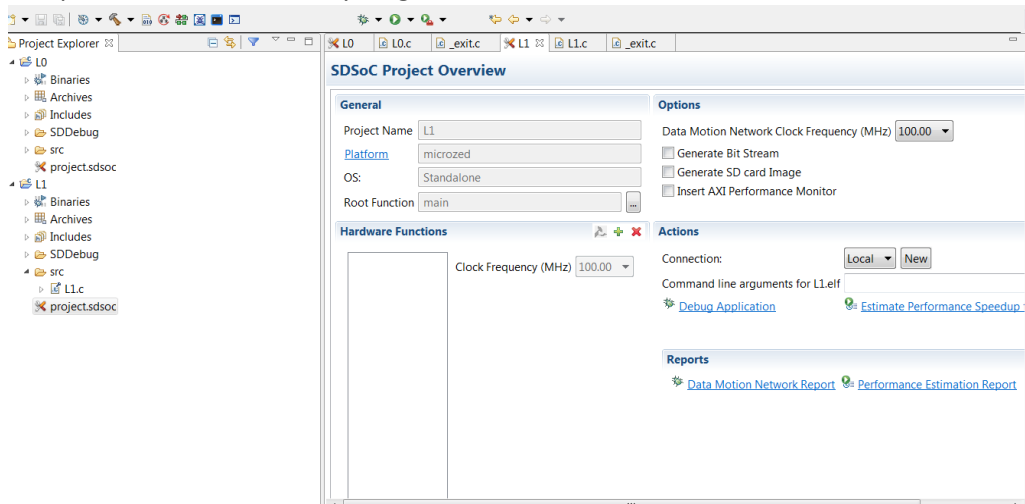
1. SDSoc should be still opened on Debug Perspective, switch back to C/C++ Perspective clicking the upper right button "SDSoC"



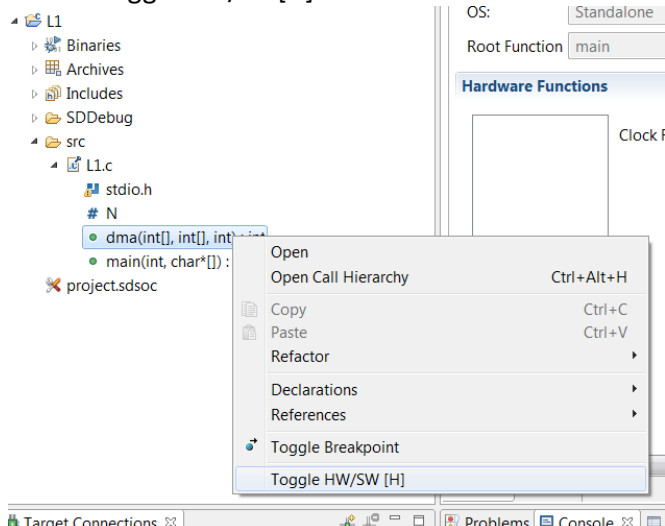
2. Open the L1 "SDSoC Project Overview" by double clicking on "project.sdsoc"



3. Unselect both "Generate Bit Stream" and "Generate SD card image", otherwise the compilation time will be very long



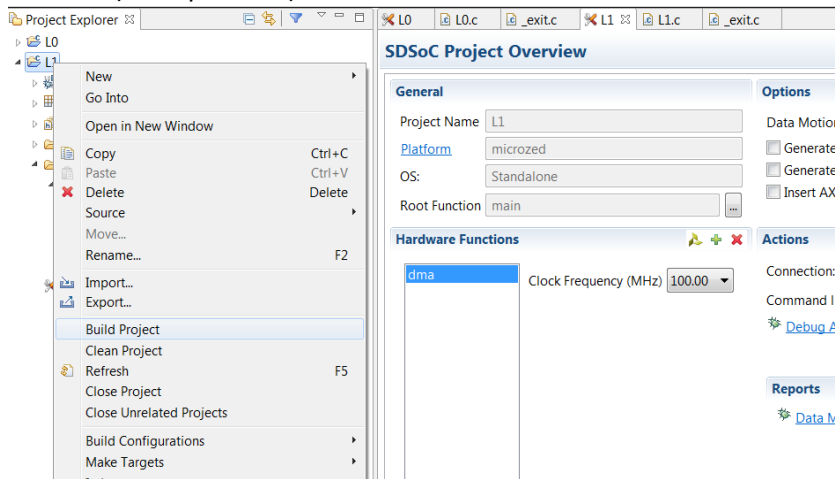
4. To move function dma to PL, left click on "L1" → "src" → "L1.c" then right click on "dma" and select "Toggle HW/SW [H]"



- Now the function dma is highlighted with an “H” meaning that it will be implemented in PL

`dma(int[], int[], int) : int [H]`

- To build the project right click on L1 and select “Build Project”, the process will likely take 5 minutes (PC dependent)



- On L1 “SDSoC Project Overview” window, click “Data motion Network Report”



- The report shows that source and destination vectors of dma function will be moved in and out by a Scatter Gather DMA, via the AXI ACP port of Zynq Processing System (CortexA9) Partition 0

Data Motion Network

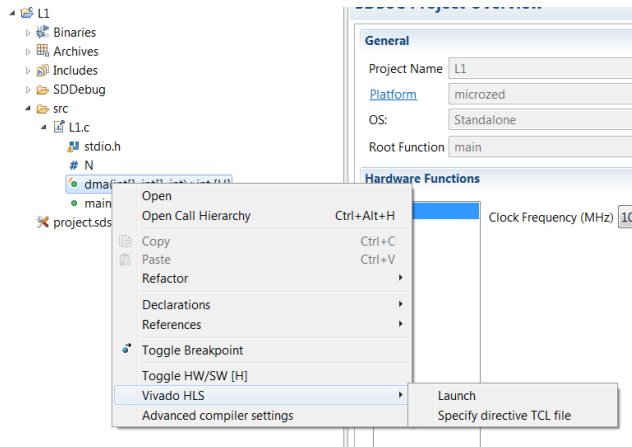
Accelerator	Argument	IP Port	Direction	Declared Size(bytes)	Pragmas	Connection
dma_0	s	s_PORTA	IN	1024*4		S_AXI_ACP:AXIDMA_SG
	d	d_PORTA	OUT	1024*4		S_AXI_ACP:AXIDMA_SG
	count	count	IN	4		M_AXI_GP0:AXILITE:0x80
	return	ap_return	OUT	4		M_AXI_GP0:AXILITE:0xC0

Accelerator Callsites

Accelerator	Callsite	IP Port	Transfer Size(bytes)	Paged or Contiguous	Cacheable or Non-cacheable
dma_0	L1.c:21:6	s_PORTA	1024 * 4	paged	cacheable
		d_PORTA	1024 * 4	paged	cacheable
		count	4	paged	cacheable
		ap_return	4	paged	cacheable

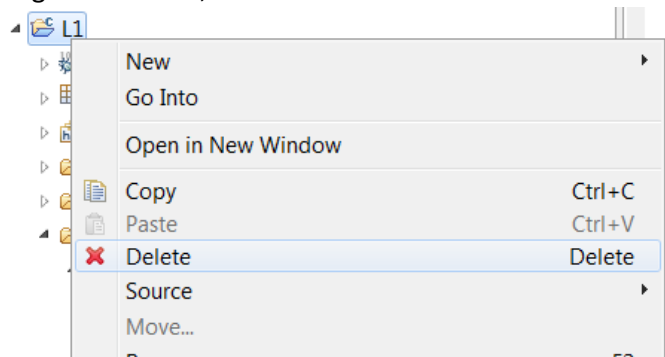
The cache coherency is guaranteed by the ACP port, while the scatter gather dma will work even in case the memory would be not physically contiguous.

9. Consider that SDSoc has automatically generated a Vivado HLS sub-project to convert the dma C code into HDL (picture below), for the moment don't launch Vivado HLS

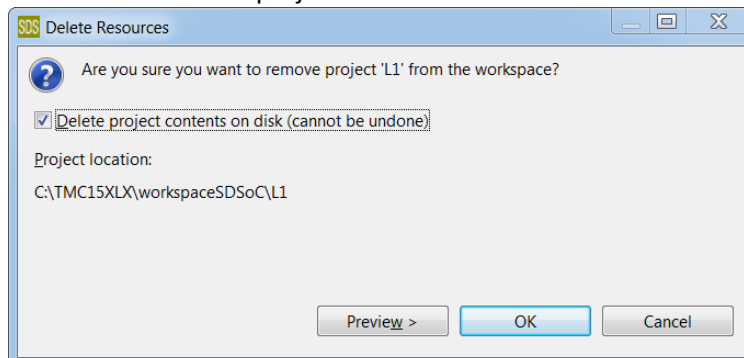


10. OPTIONAL: if you have enough time (approx. 20 min) you may enable “Generate Bit Stream” tick box on L1 “SDSoC Project Overview” window and do a “Build”, in this case skip to step 18

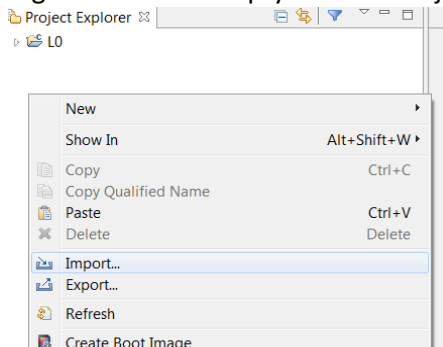
11. Right click on L1, then select “Delete”



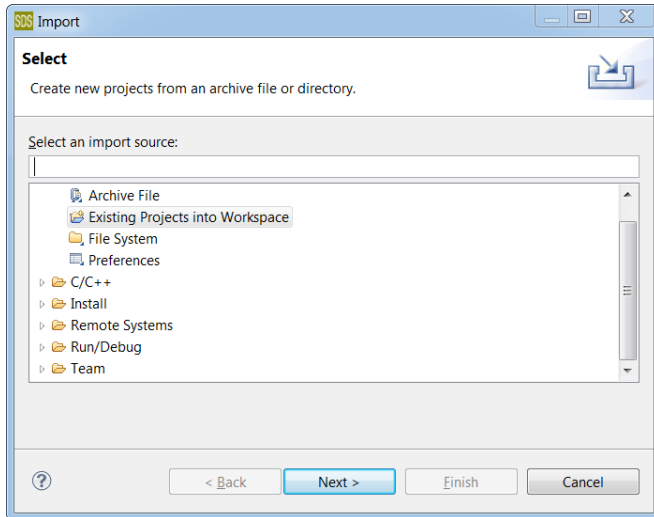
12. Tick the box “Delete project contents on disk” the hit “OK”



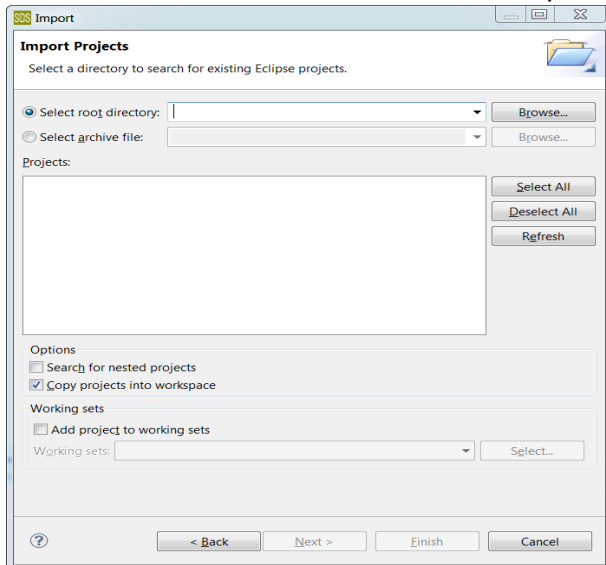
13. Right click on an empty area of “Project explorer” and select “Import”



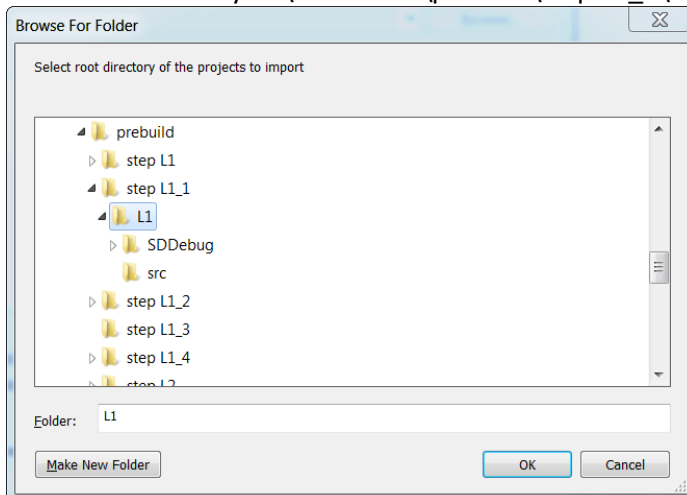
14. Select “General” → “Existing Projects into Workspace” then “Next”



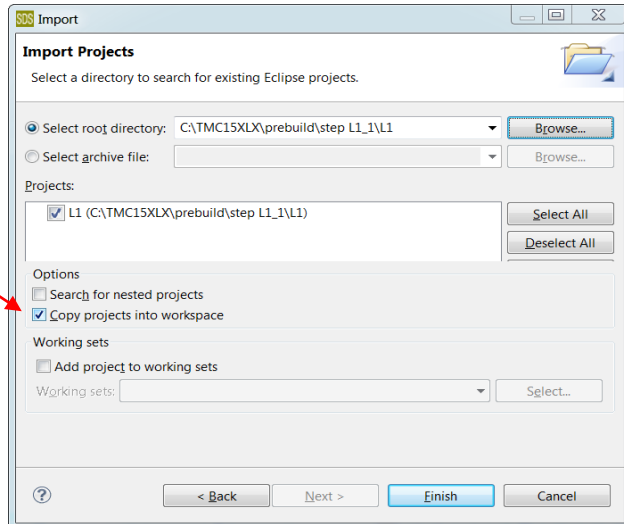
15. Click on “Browse” in the “Select root directory” row



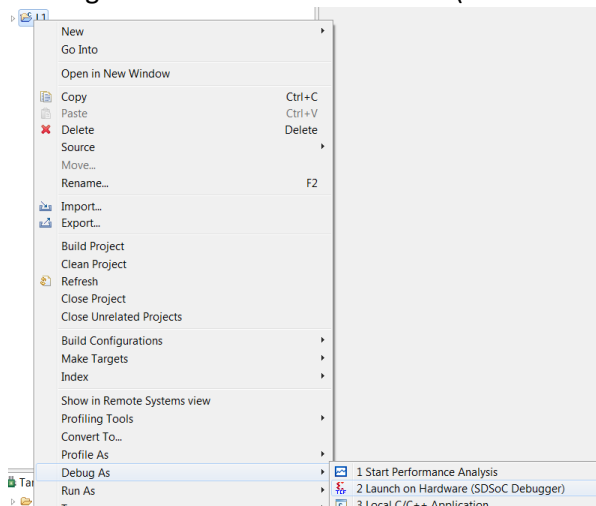
16. Browse to directory “c:\TMC15XLX\prebuild\step L1_1\L1”, hit “OK”



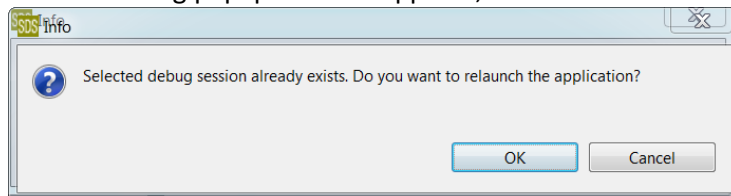
17. Make sure you select “Copy projects into workspace” then hit “Finish”



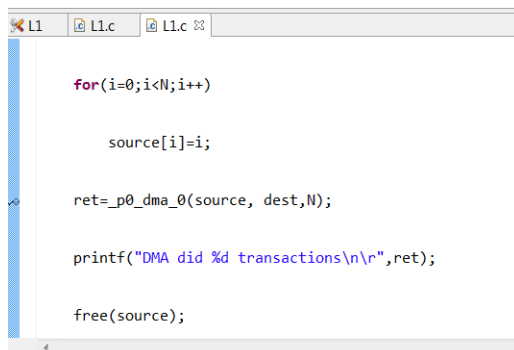
18. To download the project right click on L1 and select “Debug as..” → “Launch on Hardware (SDSoC Debugger)”



19. If the following popup window appears, select “OK”

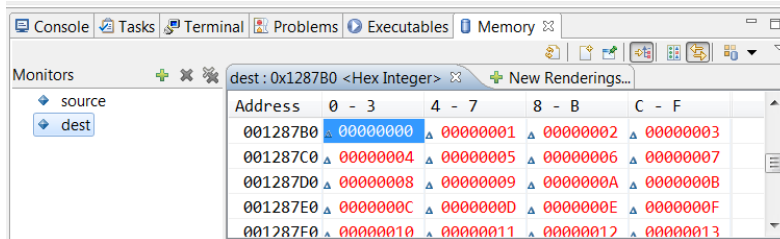


20. Double left click on the blue column beside function `_p0_dma_0` to put a breakpoint



Note: the code is different respect the one we entered, this is because SDSoC substitutes the original function with a version which takes care of data movers and synchronization points

21. Resume (F8)
22. On the lower window "Memory" remove "source" and "dest" (they point to non updated values) then add them again using the green "+"
23. Resume (F8) and verify that also this HW accelerated version of function dma is copying "source" vector to "dest"



It's very likely that you obtained the results highlighted above, but there is a pitfall to know to avoid possible misbehaviours :

http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug1027-intro-to-sdsoc.pdf page 29

Hardware Function Call Guidelines

- Stub functions generated in the SDSoC™ environment transfer the exact number of bytes according the compile-time determinable array bound of the corresponding argument in the hardware function declaration. If a hardware function admits a variable data size, you can use the following pragma to direct the SDSoC environment to generate code to transfer data whose size is defined by an arithmetic expression:

```
#pragma SDS data copy|zero_copy(arg[0]:<C_size_expr>]
```

where the `<C_size_expr>` must compile in the scope of the function declaration.

The `zero_copy` pragma directs the SDSoC environment to map the argument into shared memory.

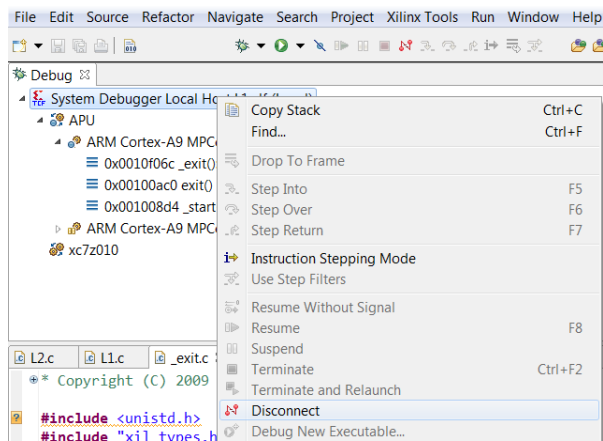
Be aware that mismatches between intended and actual data transfer sizes can cause the system to hang at runtime, requiring laborious hardware debugging.

- Align arrays transferred by DMAs on cache-line boundaries (for L1 and L2 caches). Use the `sds_alloc` API provided with the SDSoC environment or `posix_memalign()` instead of `malloc()` to allocate these arrays.
- Align arrays to page boundaries to minimize the number of pages transferred with the scatter-gather DMA, for example, for arrays allocated with `malloc`.
- You must use `sds_alloc` to allocate an array for the following two cases:
 - You are using zero-copy pragma for the array.
 - You are using pragmas to explicitly direct the system compiler to use Simple-DMA or

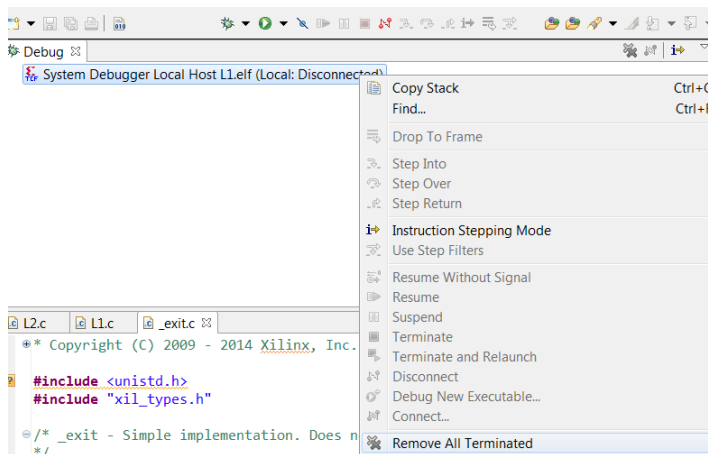
Basically when using the PL DMA inferred by SDSoC the memory must be aligned to L1 and L2 cache-line boundaries; moreover, it's suggested to use memory aligned to page boundaries (eg MMU), to minimize the number of pages transferred.

We'll fix this subtle pitfall during next L1_2 lab.

24. To close the debug session, right click on the upper left “System Debugger Local Host” and select “Disconnect”



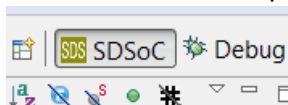
25. Right click on the upper left “System Debugger Local Host” entry then select “Remove All Terminated”



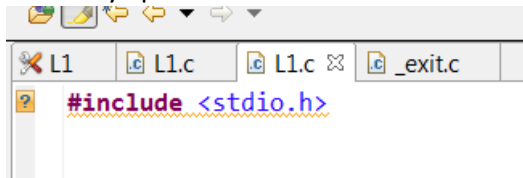
L1_2 One line code DMA, PL accelerated, contiguous memory

During this lab we'll use a SDSoC specific dynamic allocation function to get a physically contiguous chunk of memory. This will imply the use of a simple DMA in PL vs the previous scatter gather DMA. This will also solve the pitfall seen during L1_1.

1. Switch back to SDSoC project perspective clicking the upper right icon



2. Close any opened window on the center of the IDE



3. Double click on "L1" → "src" → "L1.c", on the opened source window do the following changes
 - a. Just after #include <stdio.h> insert line **#include "sds_lib.h"**
 - b. Substitute every malloc occurrence with **sds_alloc**
 - c. Substitute every free occurrence with **sds_free**

You should see

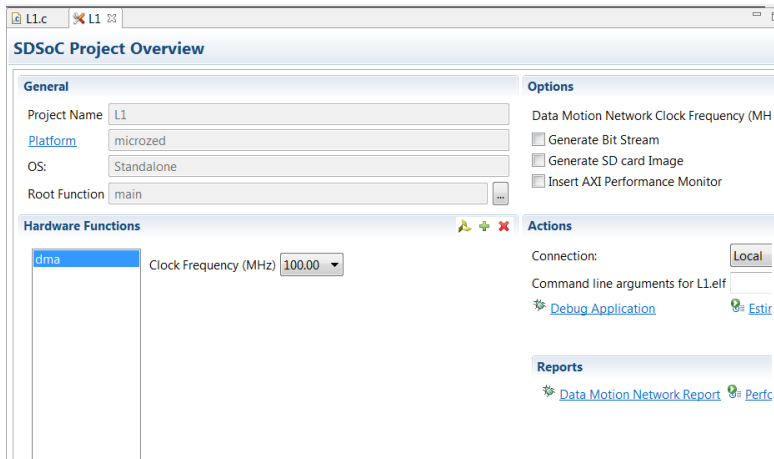
```
#include <stdio.h>
#include <stdlib.h>
#include "sds_lib.h"
#define N 1024
int dma(int s[N], int d[N], int count)
{
    int i;
    for(i=0; i<count; i++)
        d[i]=s[i];
    return count;
}

int main(int argc, char *argv[])
{
    int *source;
    int *dest;
    int ret, i;

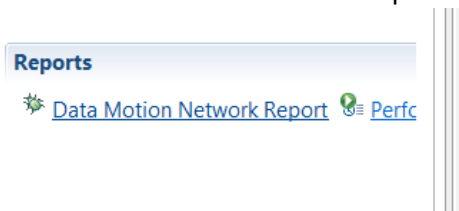
    source=(int*) sds_alloc(sizeof(int)*N);
    dest = (int*) sds_alloc(sizeof(int)*N);
    for(i=0; i<N; i++)
        source[i]=i;
    ret=dma(source, dest, N);
    printf("DMA did %d transactions\n\r", ret);
    sds_free(source);
    sds_free(dest);
    return 0;
}
```

4. Save (CTRL-S)

- Double click on “L1” → “project.sdsoc” and make sure the flags “Generate Bit Stream” and “Generate SD card Image” are not selected



- Build the project (right click on L1 then “Build Project”)
- Now click on “Data Motion Network Report” of the same “SDSoC Project Overview” window



- From the report it’s clear that a simple DMA is now used to transfer data between PS and PL instead of a Scatter Gather DMA:

Partition 0

Data Motion Network

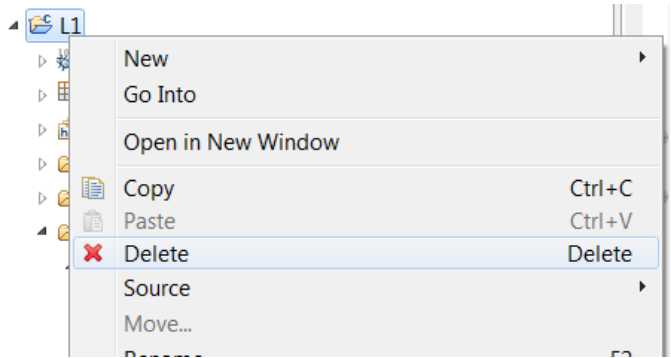
Accelerator	Argument	IP Port	Direction	Declared Size (bytes)	Pragmas	Connection
dma_0	s	s_PORTA	IN	1024*4		S_AXI_ACP:AXIDMA_SIMPLE
	d	d_PORTA	OUT	1024*4		S_AXI_ACP:AXIDMA_SIMPLE
	count	count	IN	4		M_AXI_GP0:AXILITE:0x80
	return	ap_return	OUT	4		M_AXI_GP0:AXILITE:0xC0

Accelerator Callsites

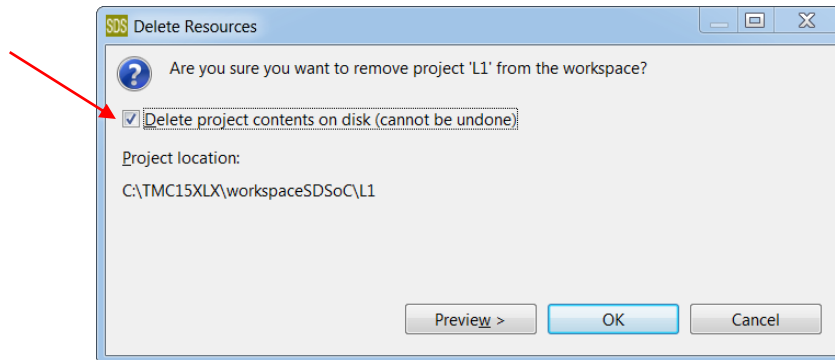
Accelerator	Callsite	IP Port	Transfer Size (bytes)	Paged or Contiguous	Cacheable or Non-cacheable
dma_0	L1.c:23:6	s_PORTA	1024 * 4	contiguous	cacheable
		d_PORTA	1024 * 4	contiguous	cacheable
		count	4	paged	cacheable
		ap_return	4	paged	cacheable

- OPTIONAL: if you have enough time (approx. 20 min) you may enable “Generate Bit Stream” tick box on L1 “SDSoC Project Overview” window and do a “Build”, in this case skip to step 18
- For the sake of the time we’ll import an already compiled project

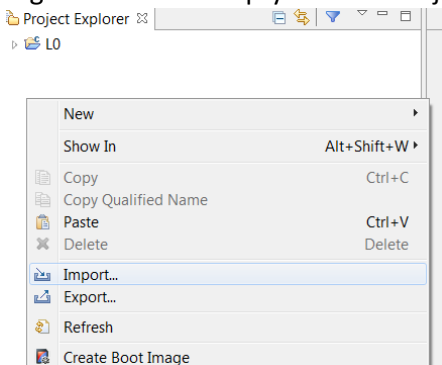
11. Right click on L1, then select “Delete”



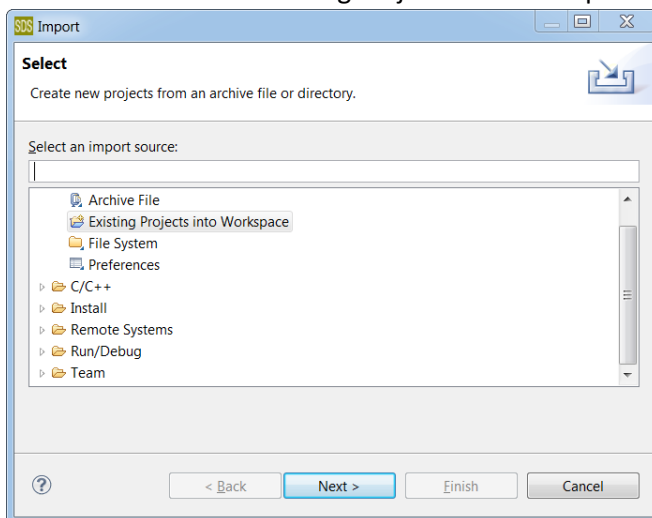
12. Tick the box “Delete project contents on disk” then hit “OK”



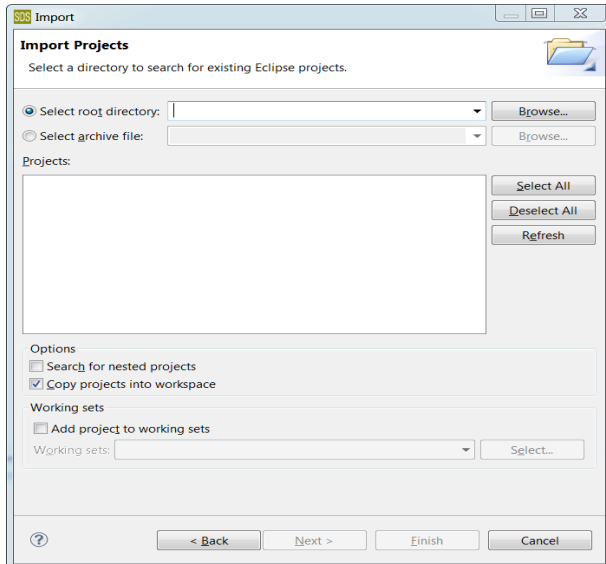
13. Right click on an empty area of “Project explorer” and select “Import”



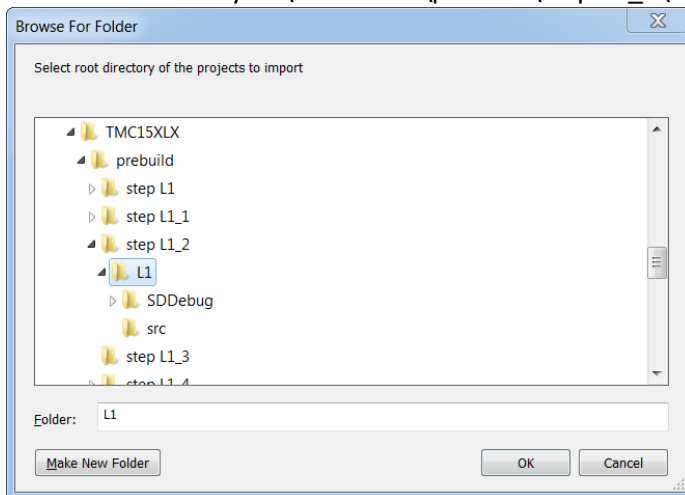
14. Select “Generals” → “Existing Projects into Workspace” then “Next”



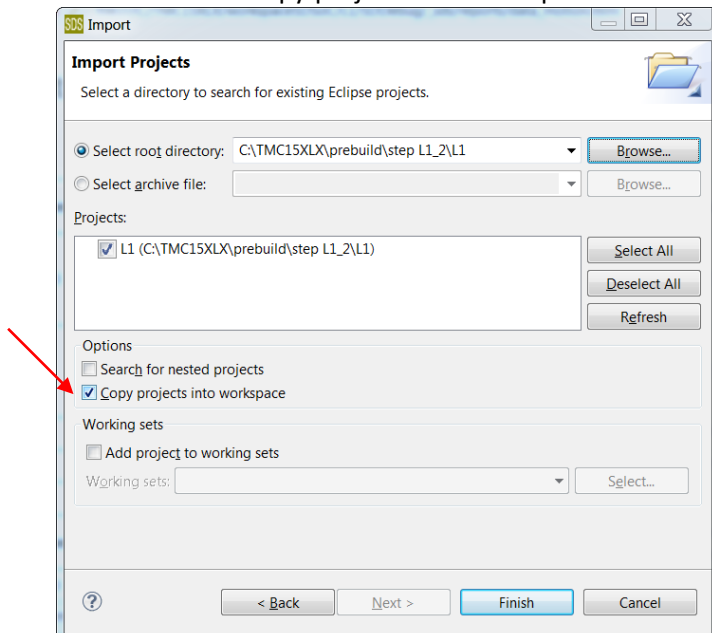
15. Click on “Browse” from “Select root directory” row



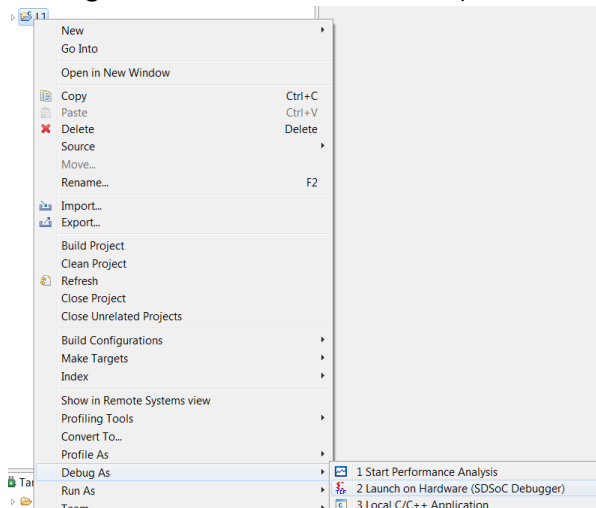
16. Browse to directory “c:\TMC15XLX\prebuild\step L1_2\L1”, hit “OK”



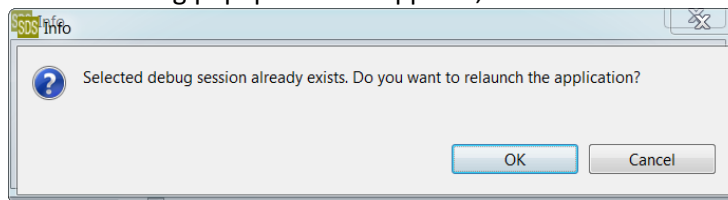
17. Make sure the “Copy projects into workspace” is ticked then hit “Finish”



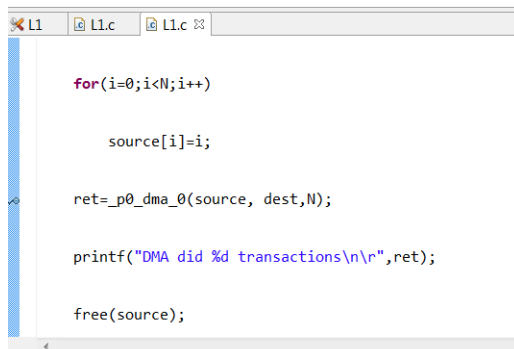
18. To download the project right click on L1 and select
“Debug As..”→“Launch on Hardware (SDSoC Debugger)”



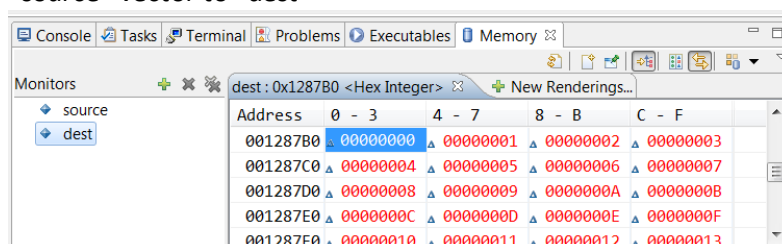
19. If the following popup window appears, select “OK”



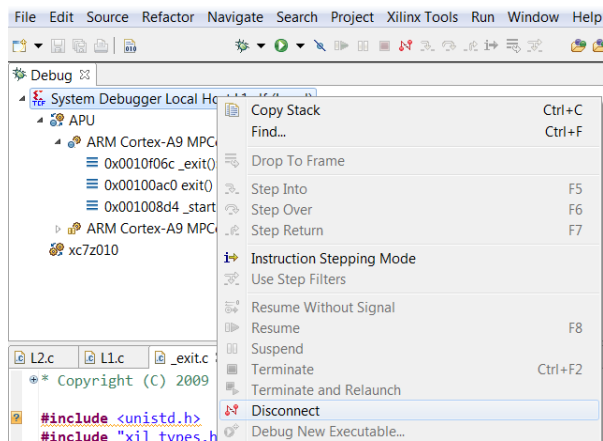
20. Double left click on the blue column beside function `_p0_dma_0` to put a breakpoint



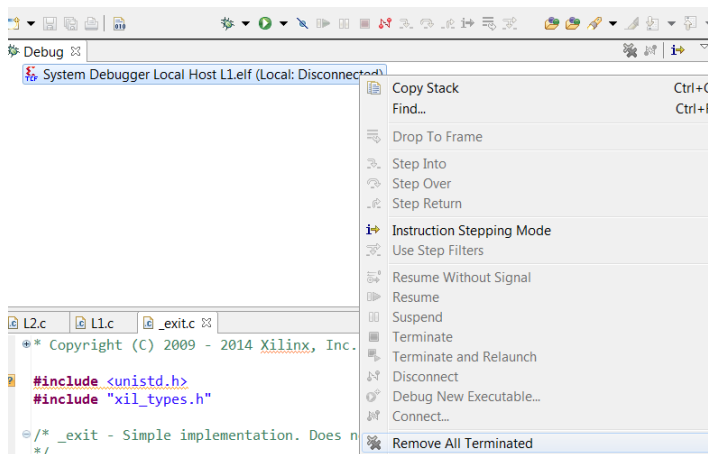
21. Resume (F8)
22. On the lower window “Memory” remove “source” and “dest” (they point to non-updated values) then add them again using the green “+”
23. Resume (F8) and verify that also this HW accelerated version of function dma is copying “source” vector to “dest”



24. To close the debug session, right click on the upper left “System Debugger Local Host” and select “Disconnect”



25. Right click on the upper left “System Debugger Local Host” entry then select “Remove All Terminated”



L1_3 One line code DMA, PL accelerated, zero copy memory

In the previous labs the data movements between PL and PS have relied on a PL DMA, but it's not the only possible implementation. During this lab we'll implement our dma c function as an AXI Master peripheral, therefore our function will generate autonomous access to the dynamic memory to fetch and write back the data.

1. Return to the L1.c source window and do the following changes
 - a. Just after the declaration of function dma, insert the line

#pragma SDS data zero_copy(s[0:N],d[0:N])

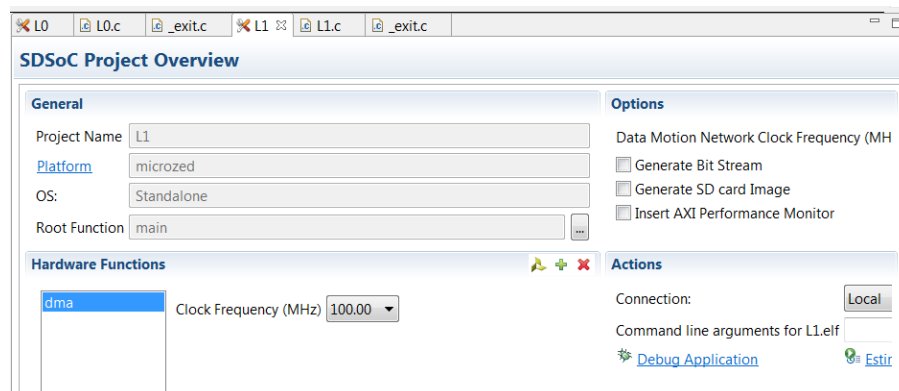
You should see

```
#include <stdio.h>
#include <stdlib.h>
#include "sds_lib.h"
#define N 1024
int dma(int s[N], int d[N],int count)
#pragma SDS data zero_copy(s[0:N],d[0:N])
{
    int i;
    for(i=0;i<count;i++)
        d[i]=s[i];
    return count;
}

int main(int argc, char *argv[])
{
    int *source;
    int *dest;
    int ret,i;

    source=(int*) sds_alloc(sizeof(int)*N);
    dest =(int*) sds_alloc(sizeof(int)*N);
    for(i=0;i<N;i++)
        source[i]=i;
    ret=dma(source, dest,N);
    printf("DMA did %d transactions\n\r",ret);
    sds_free(source);
    sds_free(dest);
    return 0;
}
```

2. Save (CTRL-S) and make sure the flags "Generate Bit Stream" and "Generate SD card Image" are not selected



3. Build the project (right click on L1 then “Build Project”)
4. Now click on “Data Motion Network Report” of the same “SDSoC Project Overview” window



5. The report is telling that source and destination data are now exchanged with the memory directly via AXI Master transactions generated by the PL function dma.

Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size (bytes)	Pragmas	Connection
dma_0	s	s	IN	1024*4	• length: (1024)	S_AXI_ACP:AXIMM:0x80
	d	d	OUT	1024*4	• length: (1024)	S_AXI_ACP:AXIMM:0x84
	count	count	IN	4		M_AXI_GP0:AXILITE:0x88
	return	ap_return	OUT	4		M_AXI_GP0:AXILITE:0xC0

Accelerator Callsites

Accelerator	Callsite	IP Port	Transfer Size(bytes)	Paged or Contiguous	Cacheable or Non-cacheable
dma_0	L1.c:25:6	s	(1024) * 4	contiguous	cacheable
		d	(1024) * 4	contiguous	cacheable
		count	4	paged	cacheable
		ap_return	4	paged	cacheable

L1_4 One line code DMA, PL accelerated, zero copy memory, async tasks

The default behaviour of SDSoC when a function is moved to PL is to wait for its completion, so the “main” function will be halted and PS CortexA9 will remain unused.

During this lab we'll see how to spawn our dma function to the PL and allow the PS to continue working on other tasks.

1. Return to the L1.c source window and do the following changes
 - a. Just before the function invocation `ret=dma(source, dest,N)` insert the line
#pragma SDS async(1)
 - b. Just after the function invocation `ret=dma(source, dest,N)` insert the line
printf("I'm free :)\n\r");
#pragma SDS wait(1)

You should see

```
#include <stdio.h>
#include <stdlib.h>
#include "sds_lib.h"

#define N    1024

int dma(int s[N], int d[N],int count)
#pragma SDS data zero_copy(s[0:N],d[0:N])
{
    int i;
    for(i=0;i<count;i++)
        d[i]=s[i];
    return count;
}

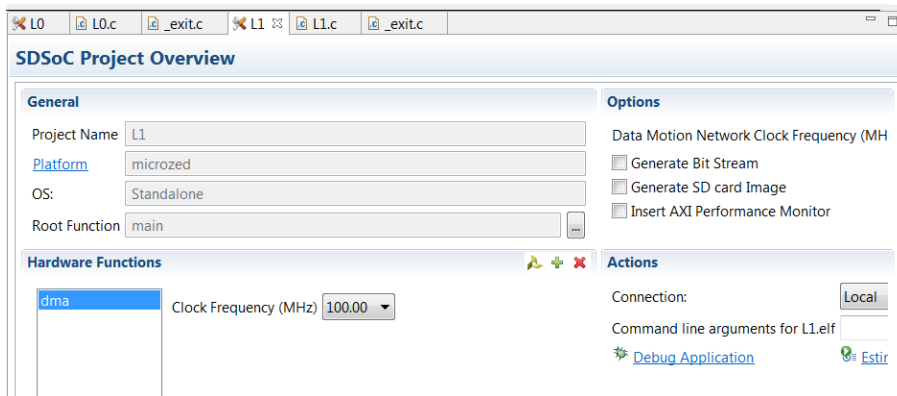
int main(int argc, char *argv[])
{
    int *source;
    int *dest;
    int ret,i;

    source=(int*) sds_alloc(sizeof(int)*N);
    dest  =(int*) sds_alloc(sizeof(int)*N);
    for(i=0;i<N;i++)
        source[i]=i;

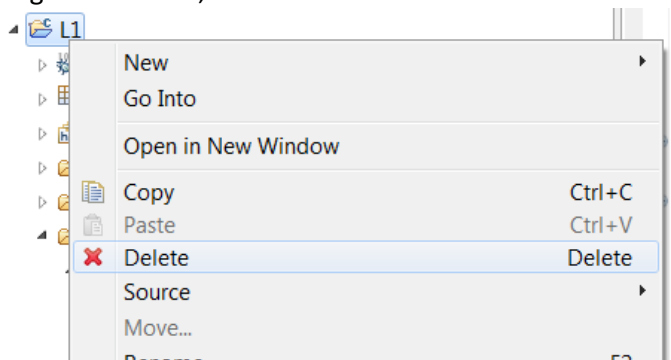
    #pragma SDS async(1)
    ret=dma(source, dest,N);
    //CPU available while dma is working, you may add some workload here
    printf("I'm free :)\n\r");
    #pragma SDS wait(1)

    printf("DMA did %d transactions\n\r",ret);
    sds_free(source);
    sds_free(dest);
    return 0;
}
```

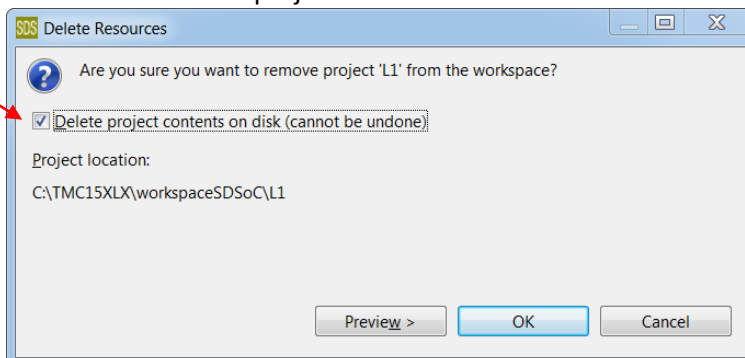
2. Save (CTRL-S) and make sure the flags “Generate Bit Stream” and “Generate SD card Image” are not selected



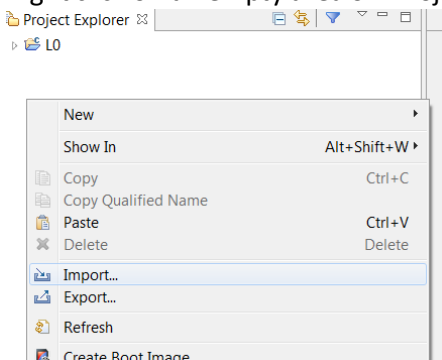
3. Build the project (right click on L1 then “Build Project”)
4. OPTIONAL: if you have enough time (approx. 20 min) you may enable “Generate Bit Stream” tick box on L1 “SDSoC Project Overview” window and do a “Build”, in this case skip to step 13
5. For the sake of the time import an already compiled project
6. Right click on L1, then select “Delete”



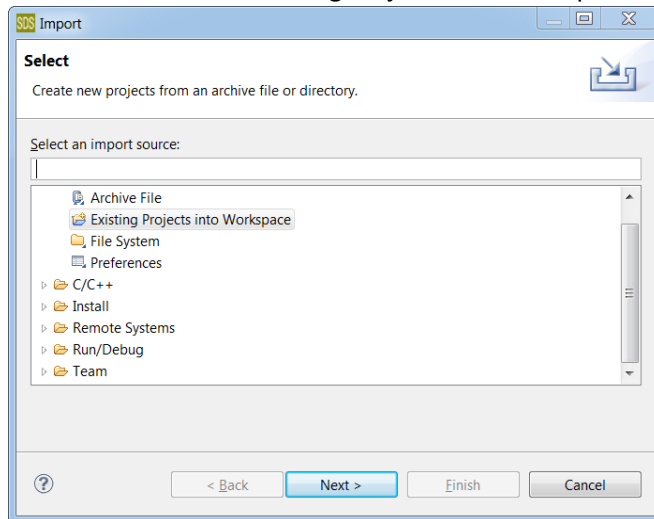
7. Tick the box “Delete project contents on disk” the hit “OK”



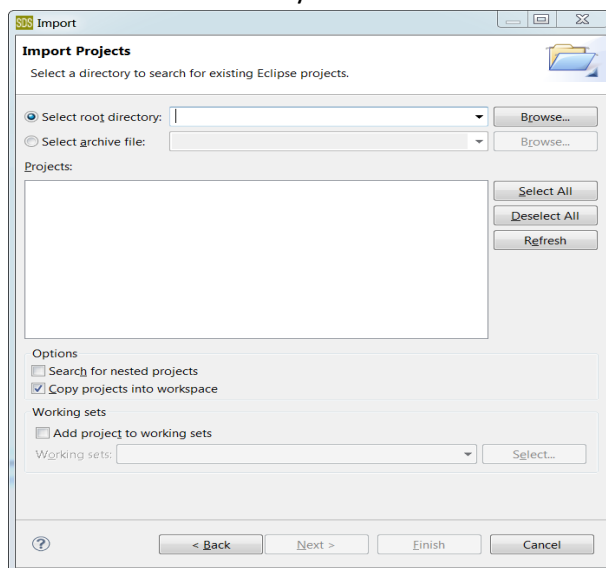
8. Right click on an empty area of “Project explorer” and select “Import”



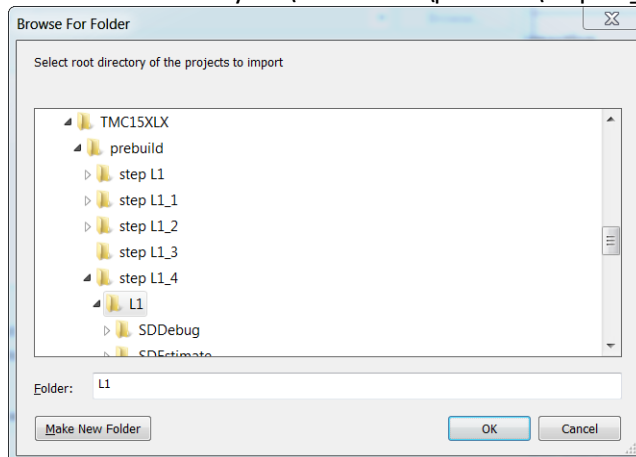
9. Select “General” → “Existing Projects into Workspace” then “Next”



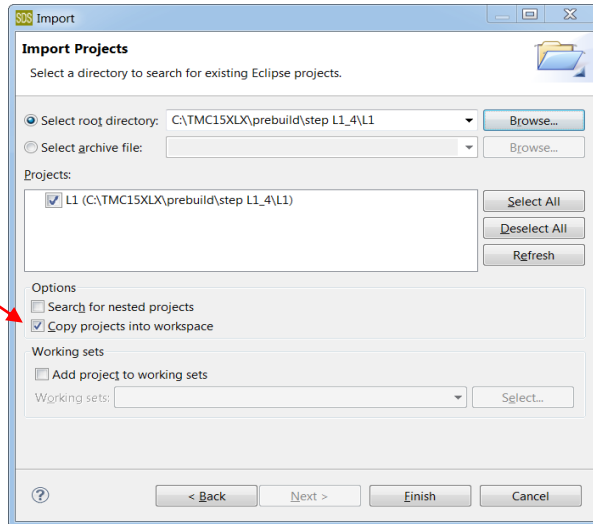
10. On “Select root directory” row click on “Browse”



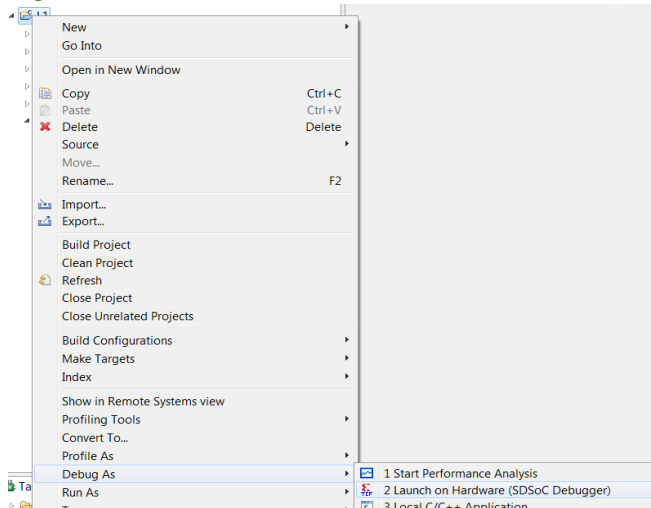
11. Browse to directory “c:\TMC15XLX\prebuild\step L1_4\L1”, hit “OK”



12. Make sure “Copy projects into workspace” tick is enabled, then hit “Finish”

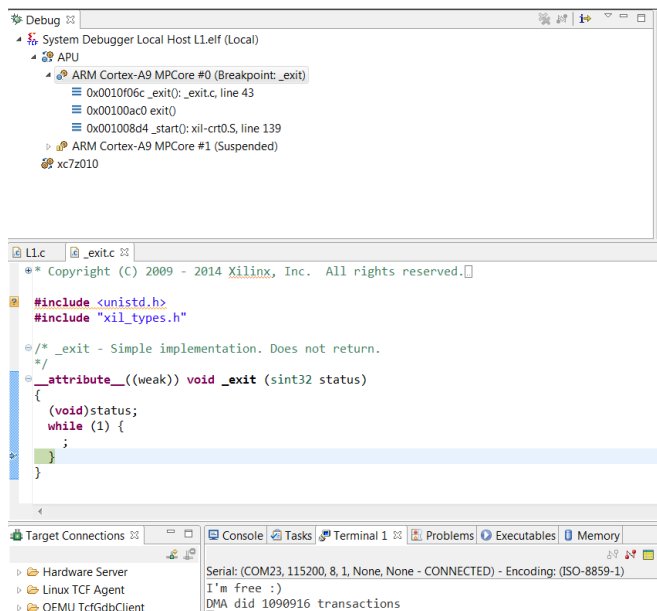


13. Right-click on L1 than select “Launch on Hardware(SDSoC Debugger)”

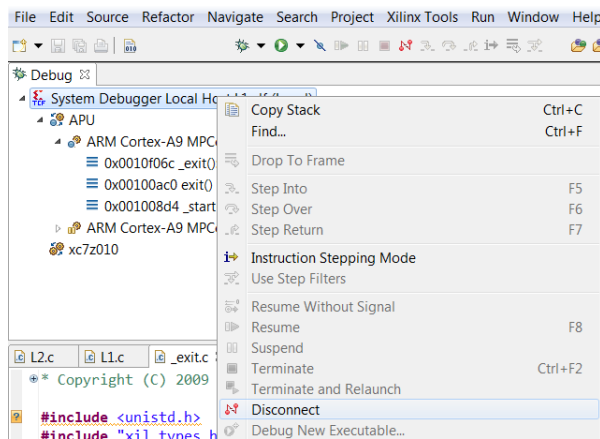


14. Confirm that “Terminal 1” window, on the bottom of the screen, is in the “CONNECTED” status

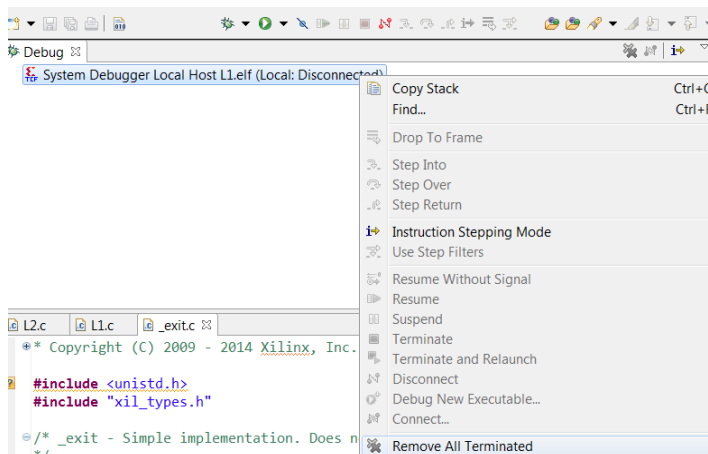
15. Resume execution (F8)



16. Can you imagine why the DMA transactions number printed here is not correct? It's because we invoked `ret=dma(source, dest,N)` and didn't wait for the function to return, therefore the `ret` value remained unassigned. This side effect confirms that the function `dma` has been executed in an asynchronous way respect the normal C sequential execution flow.
17. To close the debug session, right click on the upper left "System Debugger Local Host" and select "Disconnect"



18. Right click on the upper left "System Debugger Local Host" entry then select "Remove All Terminated"

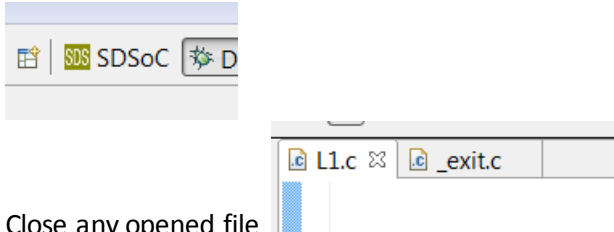


L2 Two operands function, PL accelerated

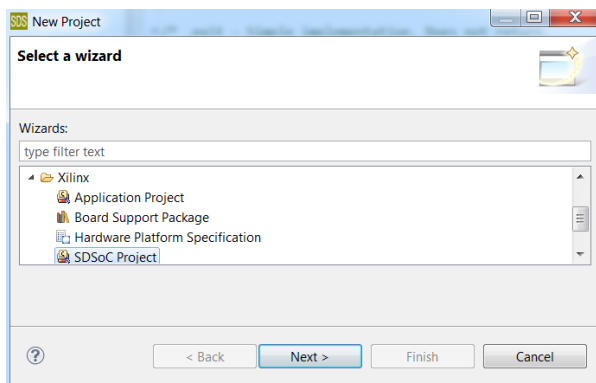
During this lab we'll evolve the previous C code so that instead of a simple copy of data we'll do a computation on incoming data (two integer vectors) and we'll write the result to memory.

We'll also move the computation to PL and we'll have a look to the latency and the resources used.

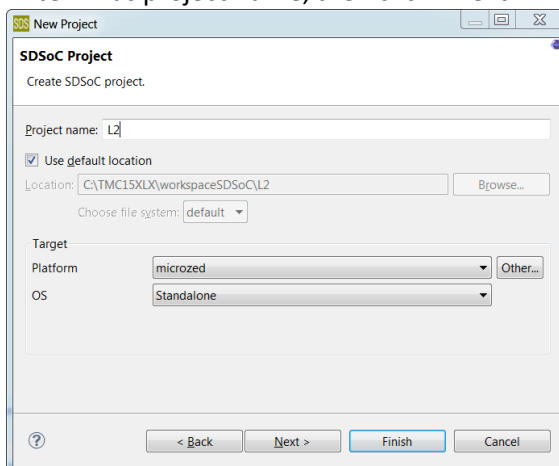
1. SDSoC should be still opened on Debug Perspective, switch back to C/C++ Perspective clicking the upper right button "SDSoC"



2. Close any opened file
3. Right-click on an empty area in the "Project Explorer" window and select "New" → "Project" → "SDSoC Project", then click on "Next" button

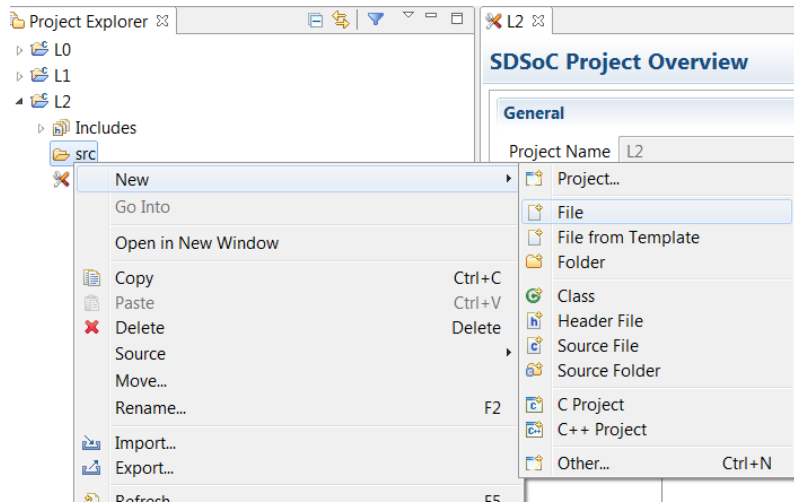


4. Enter L2 as project name, then click "Next"

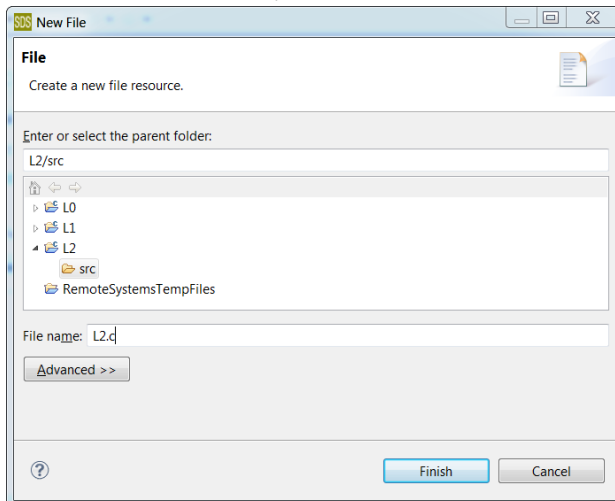


5. Select "Empty Application" then click "Finish"

6. Left-click on L2 to unfold the content, then create an empty c file by a right-click on “L2”→”src” and select “New”→”File”



7. Enter L2.c as file name, then hit “Finish”



8. Copy paste in the L2.c source window the following

```
#include <stdio.h>
#include <stdlib.h>
#include "sds_lib.h"
#define N 1024
int my_computation(int a,int b)
{
    return a*b;
}

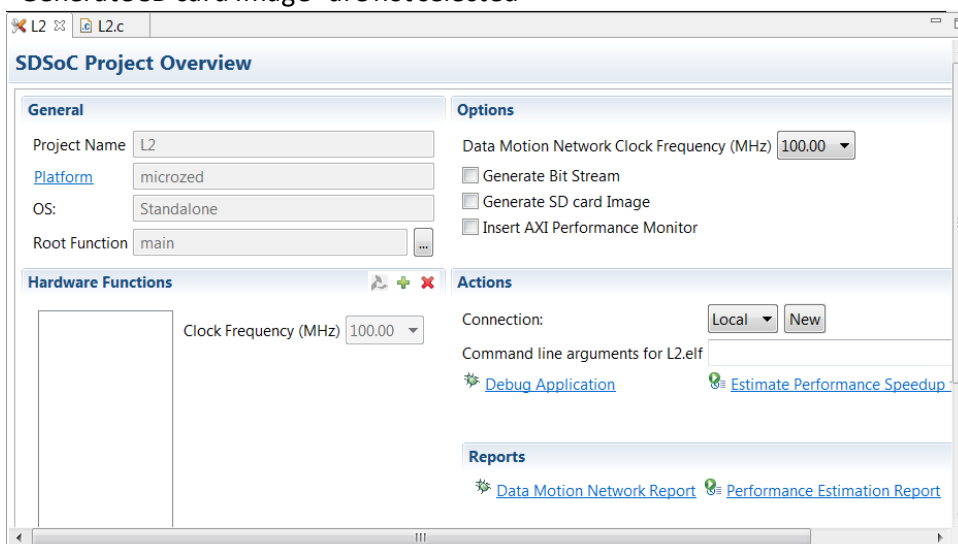
int op(int s1[N], int s2[N], int d[N],int count)
{
    int i;
    for(i=0;i<count;i++)
        d[i]=my_computation(s1[i],s2[i]);
    return count;
}

int main(int argc, char *argv[])
{
    int *source1,*source2;
    int *dest;
    int ret,i;

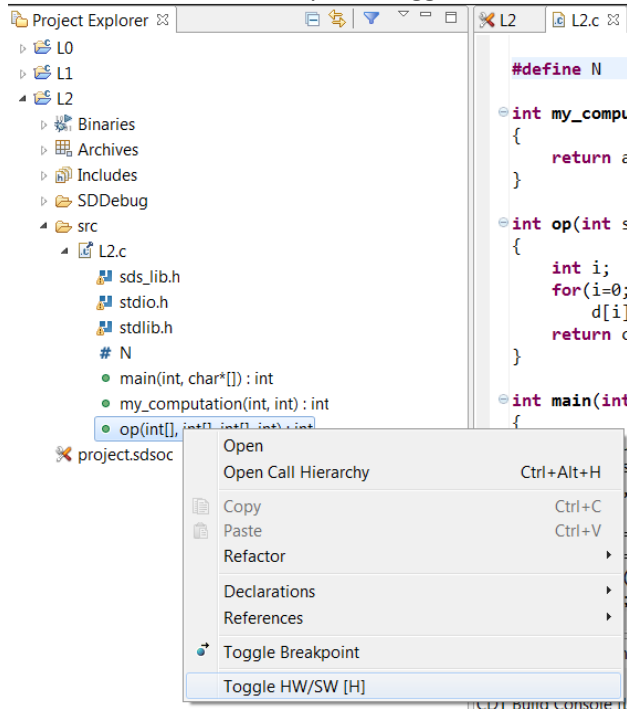
    source1=(int*) sds_alloc(sizeof(int)*N);
    source2=(int*) sds_alloc(sizeof(int)*N);
    dest =(int*) sds_alloc(sizeof(int)*N);
    for(i=0;i<N;i++){
        source1[i]=i;
        source2[i]=N-i;
    }
    ret=op(source1, source2, dest,N);
    printf("op did %d operations \n\r",ret);
    sds_free(source1);
    sds_free(source2);
    sds_free(dest);
    return 0;
}
```

19. Save (CTRL-S)

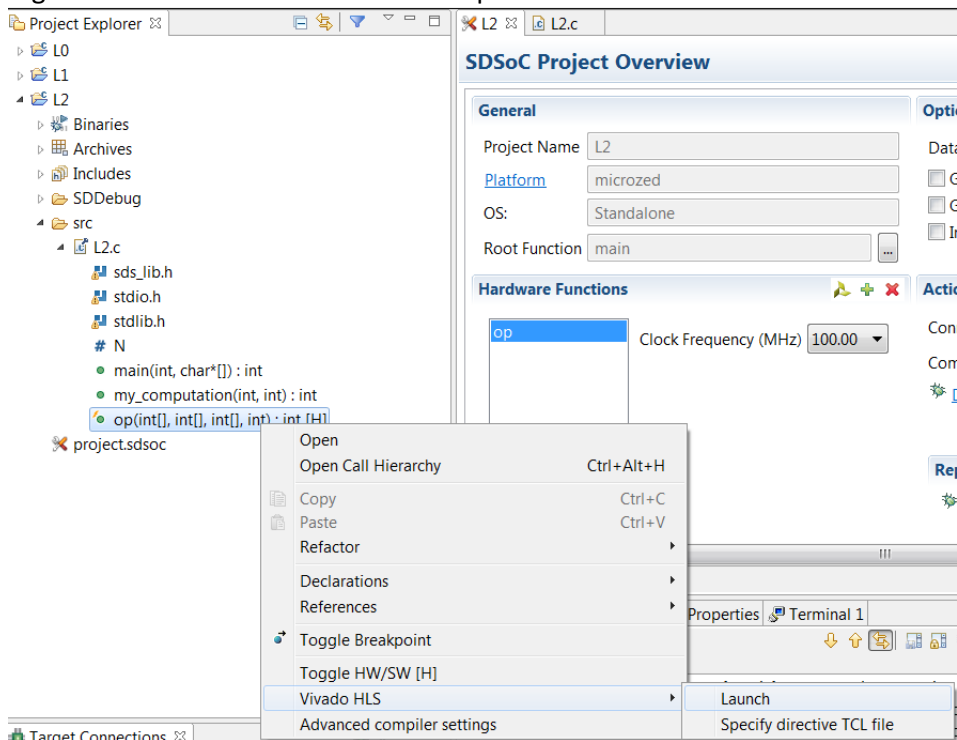
20. On “SDSoC Project Overview” window make sure the flags “Generate Bit Stream” and “Generate SD card Image” are not selected



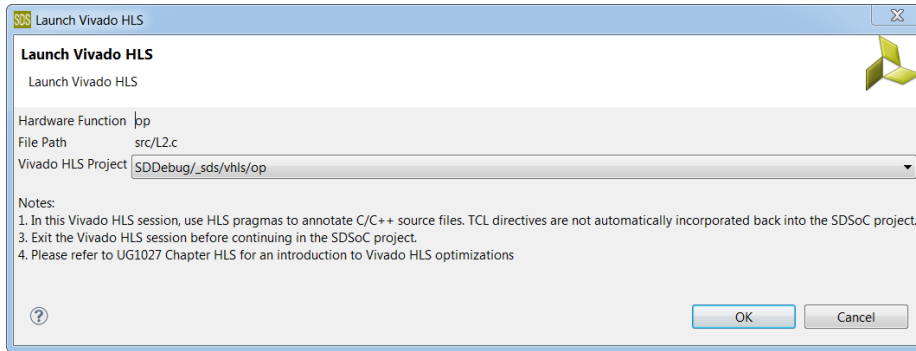
21. Build the project (right click on L2 then “Build Project”) to be sure the syntax is correct
22. Left-click on L2 to unfold the content, then right click on “L2”→”src”→”L2.c”→”op”→”Toggle HW/SW[H]” to move function “op” to PL



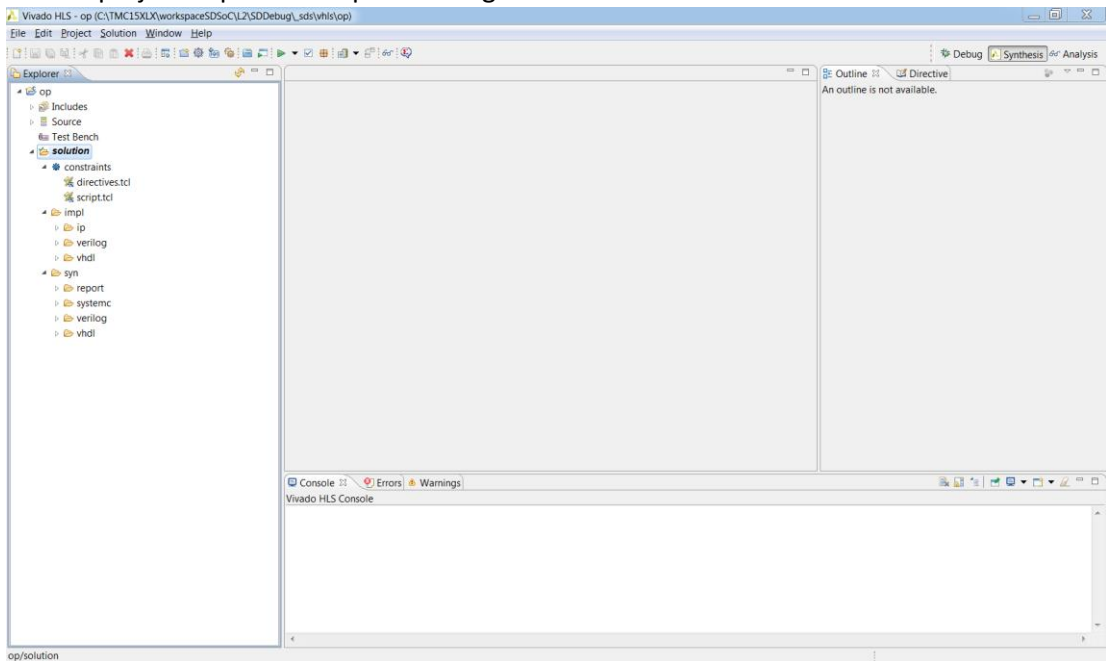
23. Build again the project (right click on L2 then “Build Project”)
24. To better analyse how the function “op” has been translated to PL we’ll use Vivado HLS on the sub project automatically generated by SDSoc
25. Right-click on “L2”→”src”→”L2.c”→”op”→”Vivado HLS”→”Launch”



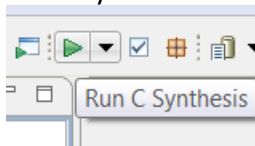
26. When a popup window appears select “OK”



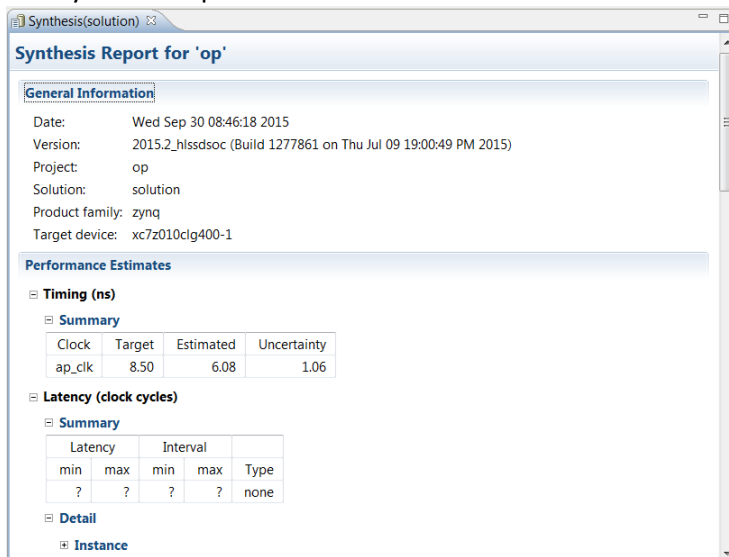
27. The subproject “op” will be opened using Vivado HLS



28. Run C Synthesis clicking on the green triangle



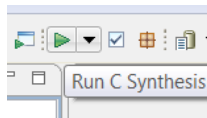
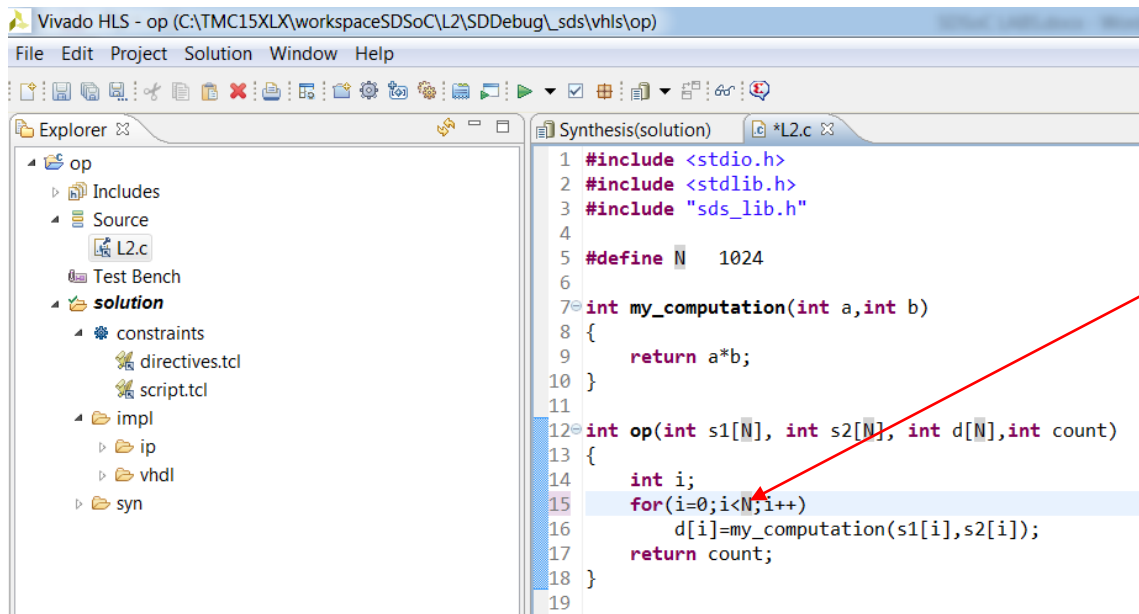
29. The Synthesis report should be like:



30. Note that the op function has an unknown latency, because the “for” loop is bounded to “count” which is unknown at compilation time

```
int op(int s1[N], int s2[N], int d[N],int count)
{
    int i;
    for(i=0;i<count;i++)
        d[i]=my_computation(s1[i],s2[i]);
    return count;
}
```

31. Left click on “op”→”Source” to unfold the content, then double left click on “op”→”Source”→”L2.c”
32. To simplify our analysis change “count” to const “N” (Note: the alternative is to add a specific “#pragma HLS LOOP_TRIPCOUNT” without source modification)



33. Save (CTRL-S) and Run C Synthesis
34. The Synthesis report will show a considerable latency

Synthesis Report for 'op'

General Information

Date: Wed Sep 30 08:55:30 2015
Version: 2015.2_hlssdsoc (Build 1277861 on Thu Jul 09 19:00:49 PM 2015)
Project: op
Solution: solution
Product family: zynq
Target device: xc7z010clg400-1

Performance Estimates

Timing (ns)

Summary

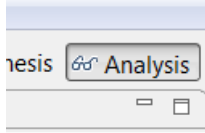
Clock	Target	Estimated	Uncertainty
ap_clk	8.50	6.08	1.06

Latency (clock cycles)

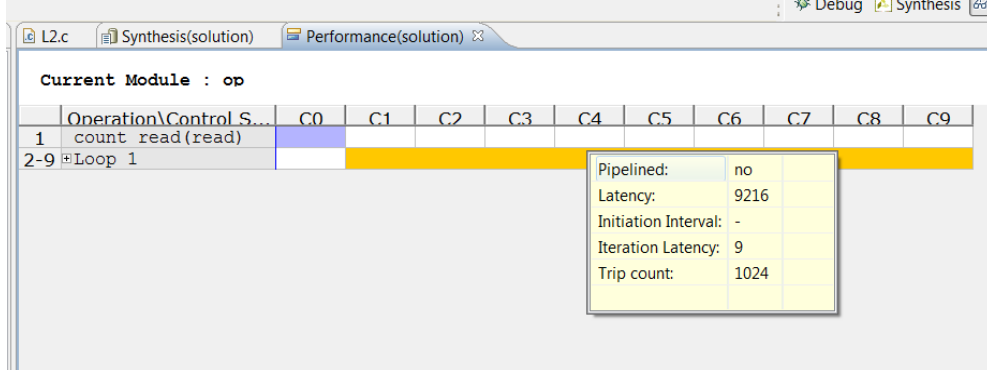
Summary

Latency		Interval		Type
min	max	min	max	
9217	9217	9218	9218	none

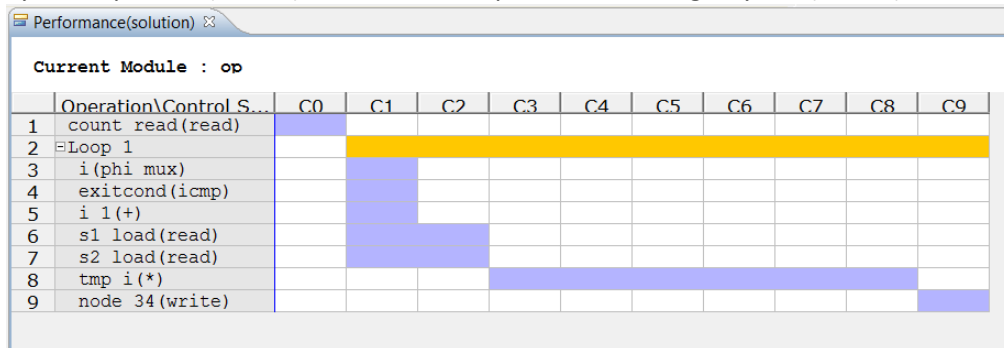
35. To investigate switch to Analysis perspective clicking on the upper right icon



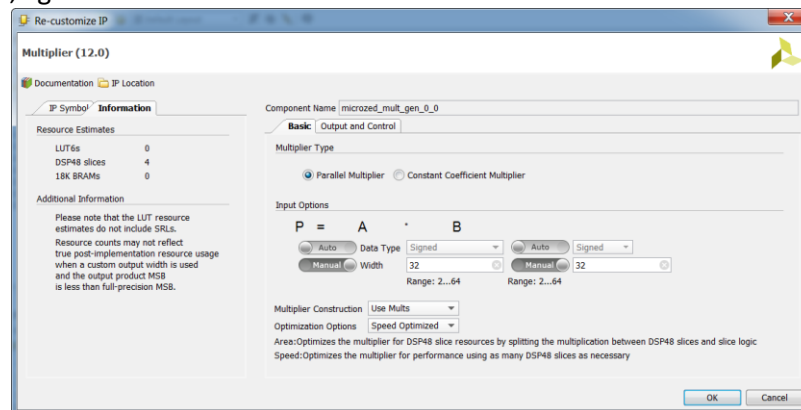
36. The graph (below) shows that the op function is taking 10 cycles to be executed; moreover, hovering the mouse on the yellow bar will show that the “for” loop is not pipelined, eg it will take 9 cycles every time it’s executed and it will not accept any input data before the completion. The loop will iterate 1024 times, so the total latency is $1024 \times 9 = 9216$ cycles.



37. Expand the “Loop 1” by clicking on “+”, it’s showed that operands s1 and s2 are read in two cycles in parallel (C1..C2), while the multiplication is taking 6 cycles (C3..C8)



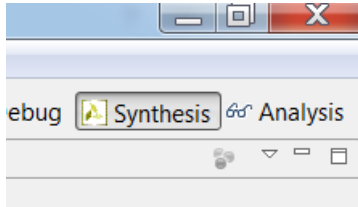
38. On the left windows it’s reported the use of 4 DSP48 blocks to implement the 32 bit int multiplication. This is well aligned to the resources required by a Multiplier IP instantiated using Vivado IPI, eg



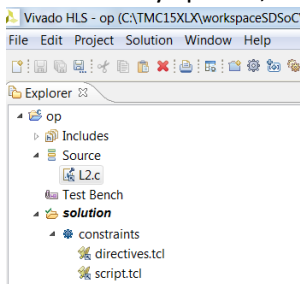
L2_1 Two operands function, PL accelerated, pipelined

In the previous lab we've implemented a multiplication of two integer vectors and we've seen that the latency of every multiplication is 9 cycles, therefore the 1024 vector element multiplication will take 9216 cycles. During this lab we'll see how to reduce the latency of every multiplication to 1 cycle.

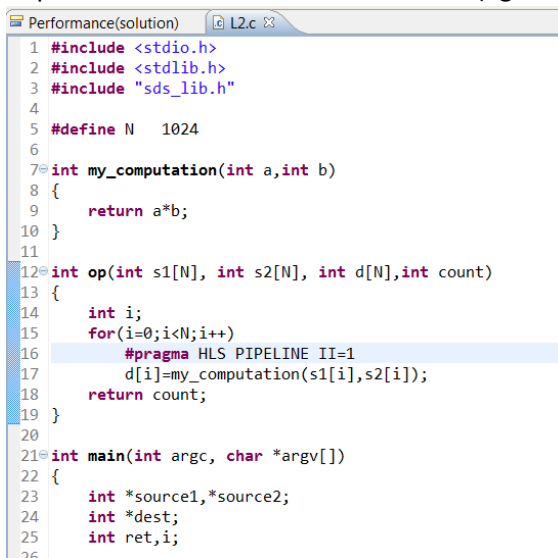
1. Vivado HLS should still be opened on sub project "op", switch to "Synthesis" view via the upper right button



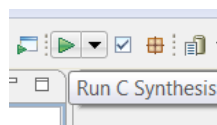
2. If not already opened, double click on "op" → "Source" → "L2.c" to show the source code



3. Just after the "for" add line **#pragma HLS PIPELINE II=1**, this will push the tool to find an implementation with Initiation Interval (eg latency) = 1



Note that any source modification will be reflected also to the SDSoC Project we started from



4. Save (CTRL-S) and run C Synthesis

5. The Synthesis report will show an improved latency

Synthesis Report for 'op'

General Information

Date: Wed Sep 30 15:55:28 2015
Version: 2015.2_hlssdsoc (Build 1277861 on Thu Jul 09 19:00:49 PM 2015)
Project: op
Solution: solution
Product family: zynq
Target device: xc7z010clg400-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	8.50	6.08	1.06

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1033	1033	1034	1034	none

6. Switch to “Analysis” perspective via the upper right icon and hover on the “Loop 1”, you will see that the “for” loop is now pipelined (eg my_computation sub function can accept a new input on every cycle)

Current Module : op

Operation\Control S...	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
1 count read(read)										
2 Loop 1										
3 i(phi mux)										
4 exitcond icmp)										
5 i 1(+)										
6 s1 load(read)										
7 s2 load(read)										
8 tmp i(*)										
9 node 36(write)										

Tooltip:

Pipelined:	yes
Latency:	1031
Initiation Interval:	1
Iteration Latency:	9
Trip count:	1024

7. To cross check against the C source code right click on line 8 (purple bar) and select “Goto source”, then select my_computation and click “OK”

Inline Stack Information Dialog

No.	Source File	Function	Line#
0	./././src/...	op	[17]
1	./././src/...	my_comput...	[9]

Properties C Source

File: C:\TMC15XLX\workspaceSDSoC\L2\src\L2.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "sds_lib.h"
4
5 #define N 1024
6
7 int my_computation(int a,int b)
8 {
9     return a*b;
10 }
11
12 int op(int s1[N], int s2[N], int d[N],int count)
13 {
14     int i;
15     for(i=0;i<N;i++)

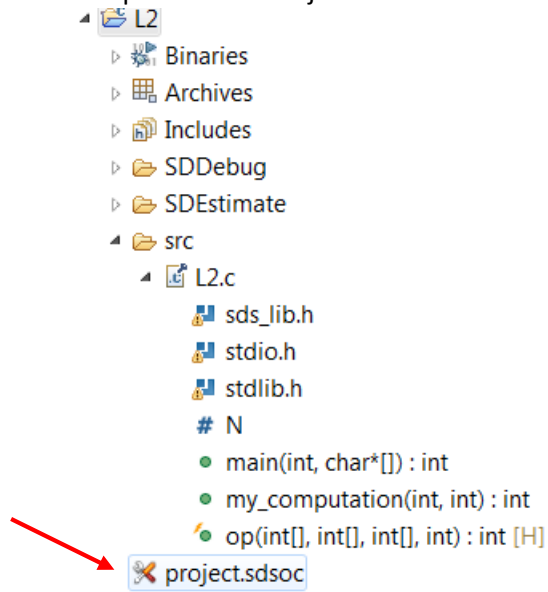
```

L2_2 Perf estimation on two operands function, PL accelerated, pipelined

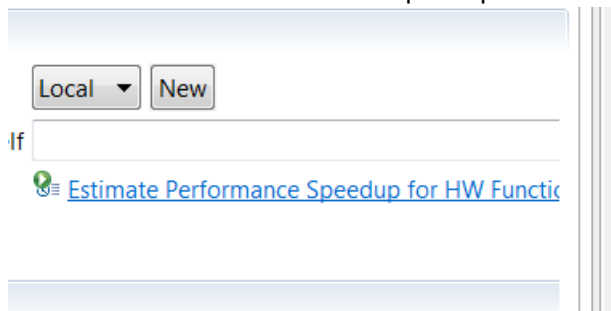
In the previous lab we've optimized the latency of function op by using **#pragma HLS PIPELINE II=1**.

During this lab we'll see the performances of the whole application with and without HW acceleration (eg pure CortexA9 execution).

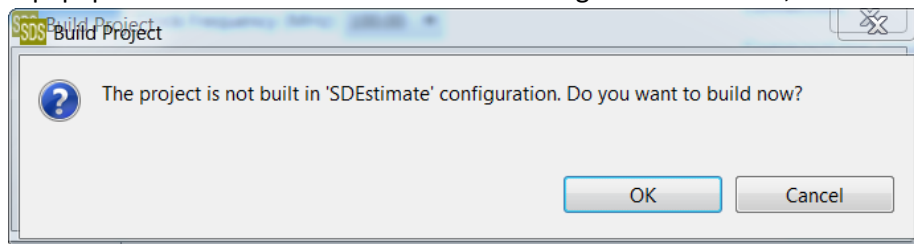
1. Close Vivado HLS and go back to SDSoC
2. Reopen "SDSoC Project Overview" window by double clicking "project.sdsoc"



3. Click on "Estimate Performance Speedup for HW Function"



4. A popup window will ask if we want to build target "SDEstimate", click "OK"



- You should see the following

Performance and resource estimation report for the 'L2' project

[Click Here](#) to get software-only application performance and speedup

Note: Performance estimation assumes worst-case latency of hardware accelerators, it also assumes worst-case data transfer size for arrays (if transfer size cannot be determined at compile time). If the accelerator latency and data transfer size at run-time is smaller than such assumptions, the performance estimation will be more pessimistic than the actual performance.

Details

Performance estimates for 'op' function

HW accelerated (Estimated cycles)	27273
-----------------------------------	-------

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	4	80	5
BRAM	0	60	0
LUT	45	17600	0,26
FF	131	35200	0,37

- Make sure used board is connected, then click on

[Click Here](#) to get software-only application performance and speedup

- Hit "OK" on the popup window

Run application to get its performance

Runs the application on the target to capture baseline software performance dat

Please select correct target connection, ensure the board is connected and powered on.

Connection: Local New

? OK Cancel

- The SW vs HW performances are shown, in particular the op function estimated speedup is 0.23

Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	28782
HW accelerated (Estimated cycles)	49855
Estimated speedup	0,58

Details

Performance estimates for 'op' function

SW-only (Measured cycles)	6200
HW accelerated (Estimated cycles)	27273
Estimated speedup	0,23

Note that this means the HW function is approx. 5 times slower than the pure SW version.

L2_3 Perf estimation on two operands function, PL accelerated, pipelined, loop unrolled

During this lab we'll try to enhance the op function performances invoking multiple instances of the same function (my_computation) in parallel on the PL.

1. We want to enhance the performances of the following code

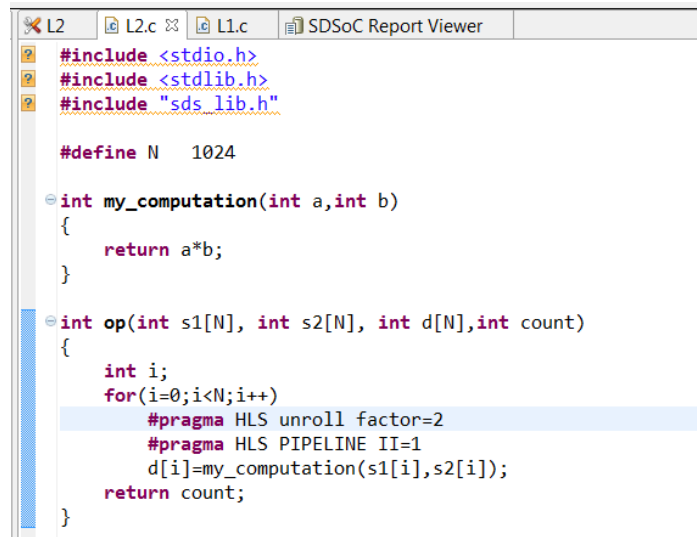
```
for(i=0;i<N;i++)
    #pragma HLS PIPELINE II=1
    d[i]=my_computation(s1[i],s2[i]);
```

one way can be to call two instances at a time of my_computation for each loop iteration
eg:

```
for(i=0;i<N;i++)
    #pragma HLS PIPELINE II=1
    d[i]=my_computation(s1[i],s2[i]);
    d[i+1]=my_computation(s1[i+1],s2[i+1]);
```

this will use twice the PL resources, because the tool will instantiate in HW two my_computation functions, but this should halve the total cycles required.
Instead of modifying the code we'll use a specific #pragma that will transparently do the same "duplication" job for us

2. Add **#pragma HLS unroll factor=2** just before #pragma HLS PIPELINE II=1



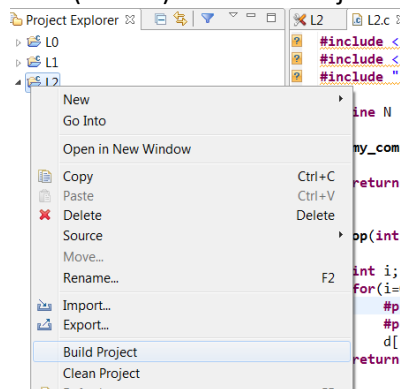
```
#include <stdio.h>
#include <stdlib.h>
#include "sds_lib.h"

#define N 1024

int my_computation(int a,int b)
{
    return a*b;
}

int op(int s1[N], int s2[N], int d[N],int count)
{
    int i;
    for(i=0;i<N;i++)
        #pragma HLS unroll factor=2
        #pragma HLS PIPELINE II=1
        d[i]=my_computation(s1[i],s2[i]);
    return count;
}
```

3. Save (CTRL-S) and Build Project



- On the “SDSoC Report viewer”, which opens up automatically, you should see

Summary

Performance estimates for 'main' function		
SW-only (Measured cycles)		28782
HW accelerated (Estimated cycles)		49855
Estimated speedup		0,58

Details

Performance estimates for 'op' function		
SW-only (Measured cycles)		6200
HW accelerated (Estimated cycles)		27273
Estimated speedup		0,23

Resource utilization estimates for hardware accelerators			
Resource	Used	Total	% Utilization
DSP	8	80	10
BRAM	0	60	0
LUT	204	17600	1,16
FF	254	35200	0,72

Note that the DSP block usage is in effect doubled (was 4 now 8) so the HW has been duplicated as requested, but the performances have not improved (0.23).

- To understand the reason of this behaviour we need to open the Vivado HLS subproject, right click on “L2” → “src” → “op” → “Vivado HLS” → “Launch”

Note: Performance estimation assumes worst-case latency of hardware accelerator at compile time). If the accelerator latency and data transfer size at run-time is smt actual performance.

Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	28782
HW accelerated (Estimated cycles)	49855
Estimated speedup	0,58

Details

Performance estimates for 'op' function

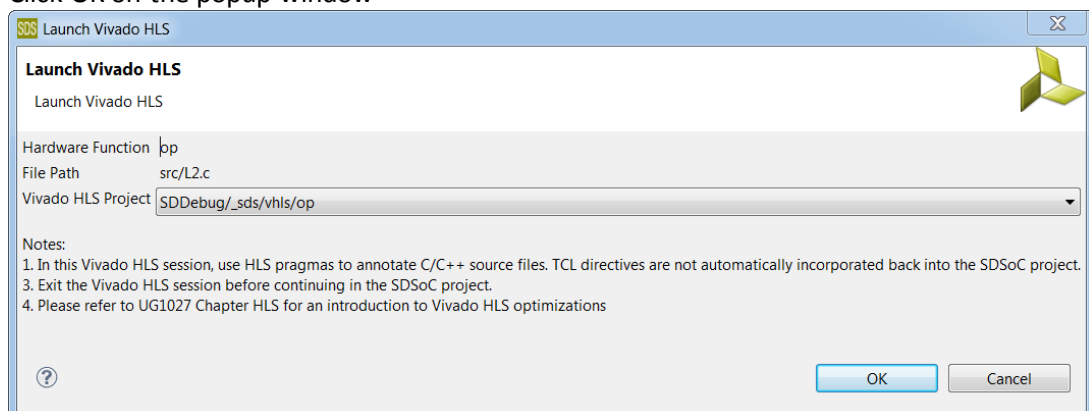
SW-only (Measured cycles)	6200
HW accelerated (Estimated cycles)	27273
Estimated speedup	0,23


Resource utilization estimates for hardware accelerators

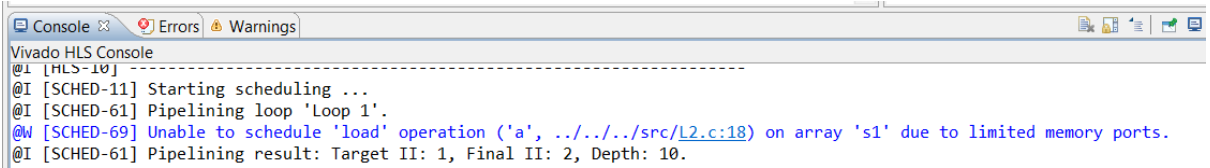
Total	% Utilization
80	10
60	0
17600	1,16

Target Connections: project.sdsoc

- Click OK on the popup window



- From HLS do the C Synthesis by clicking , then go to the lower window “Console” and scroll up the content to see the compilations messages, there should be the following blue message:



```
Vivado HLS Console
@I [HLS-10] -----
@I [SCHED-11] Starting scheduling ...
@I [SCHED-61] Pipelining loop 'Loop 1'.
@W [SCHED-69] Unable to schedule 'load' operation ('a', ../../src/L2.c:18) on array 's1' due to limited memory ports.
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 2, Depth: 10.
```

- Click on the highlighted hyperlink ['L2.c:18'](#) to see that our code has not been optimized because input vectors have limited memory ports. The memory access on s1 and s2 is single port (it's an incoming data flow generated via a DMA inferred by SDSoC), while we're asking to spawn two parallel my_computation functions that need to access to s1 and s2 at the same time eg a dual port access is needed

```
int op(int s1[N], int s2[N], int d[N],int count)
{
    int i;
    for(i=0;i<N;i++)
        #pragma HLS unroll factor=2
        #pragma HLS PIPELINE II=1
        d[i]=my_computation(s1[i],s2[i]);
    return count;
}
```

- To try to solve the issue we'll copy the data to backup vectors and tell the tool that we need a dual port access to them (therefore the vectors will be implemented using BRAMs)
- Close the Vivado HLS and go back to SDSoC
- Overwrite the L2.c code with the following

```
#include <stdio.h>
#include <stdlib.h>
#include "sds lib.h"

#define N    1024

int my_computation(int a,int b)
{
    return a*b;
}

int op2(int s1[N], int s2[N], int d[N],int count)
{
    #pragma HLS array_partition variable=s1 block factor=2 dim=1
    #pragma HLS array_partition variable=s2 block factor=2 dim=1
    #pragma HLS array_partition variable=d block factor=2 dim=1
    int i;

    for(i=0;i<N;i++){
        #pragma HLS unroll factor=2
        #pragma HLS PIPELINE II=1
        d[i]=my_computation(s1[i],s2[i]);}

    return count;
}
```

>>>>continue on next page<<<<<

```

int op(int s1[N], int s2[N], int d[N],int count)
{
    int i,ret;
    int s1_local[N], s2_local[N], d_local[N];

    for(i=0;i<N;i++){
        #pragma HLS PIPELINE II=1
        s1_local[i]=s1[i];}
    for(i=0;i<N;i++){
        #pragma HLS PIPELINE II=1
        s2_local[i]=s2[i];}
    ret=op2(s1_local,s2_local,d_local,count);
    for(i=0;i<N;i++){
        #pragma HLS PIPELINE II=1
        d[i]=d_local[i];}
    return ret;
}

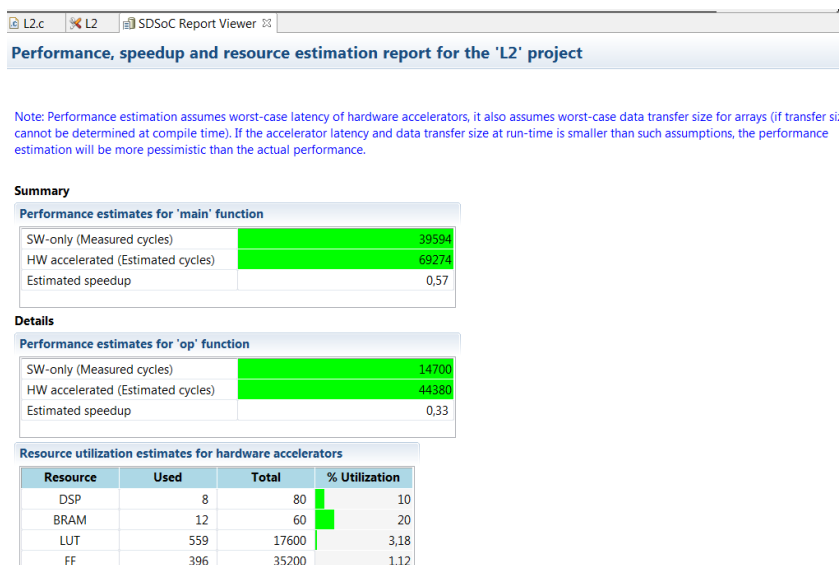
int main(int argc, char *argv[])
{
    int *source1,*source2;
    int *dest;
    int ret,i;

    source1=(int*) sds_alloc(sizeof(int)*N);
    source2=(int*) sds_alloc(sizeof(int)*N);
    dest  =(int*) sds_alloc(sizeof(int)*N);
    for(i=0;i<N;i++){
        source1[i]=i;
        source2[i]=N-i;
    }
    ret=op(source1, source2, dest,N);
    printf("op did %d operations \n\r",ret);
    sds_free(source1);
    sds_free(source2);
    sds_free(dest);
    return 0;
}

```

12. Save (CTRL-S) and Build Project

13. The SDSoC Report Viewer window will pop up:



14. We have a relative gain on Estimated speedup (0.33 vs the previous 0.23) of function op, but note that in absolute terms more cycles are required, for both SW-only and HW accelerated. This is due to the overhead of vectors copy to local buffers s1_local, s2_local

L3 Matrix floating point multiplication example on Linux

During the previous labs we've seen different aspects of SDSoC and function offloading to PL.

To obtain an IP performing better than a CortexA9 it's required the right combination of memory data movers, pragma directives and a function complexity that can balance the overhead of moving data back and forth from the PS domain to the PL one.

The functions we've moved to PL were able to offload the CortexA9, if used in conjunction with async/wait pragmas, but we were not able to obtain better absolute performances respect the SW only versions running on the CortexA9 because of the extreme simplicity of the function we've implemented.

During the next lab we'll build a floating point matrix multiplier and we'll measure significant performance advantages respect the SW only version running on the CortexA9.

NOTE: now proceed to UG1028 "SDSoC Getting Started" document and start from page 18. Please note that every time it's mentioned to select board ZC702, select microzed board instead. Moreover, when it's suggested to import an already compiled project, don't do it since there are no prebuilt projects for microzed, therefore do compile the project