C-Based VLSI Design

Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 08
Hardware Efficient C Coding
Lecture - 26
Hardware Efficient C Coding

Welcome everyone. in this today's class, I am going to discuss on Hardware Efficient C Coding. So, the topic that we are discussing is we are synthesizing a C code into hardware, right so, into RTL design.

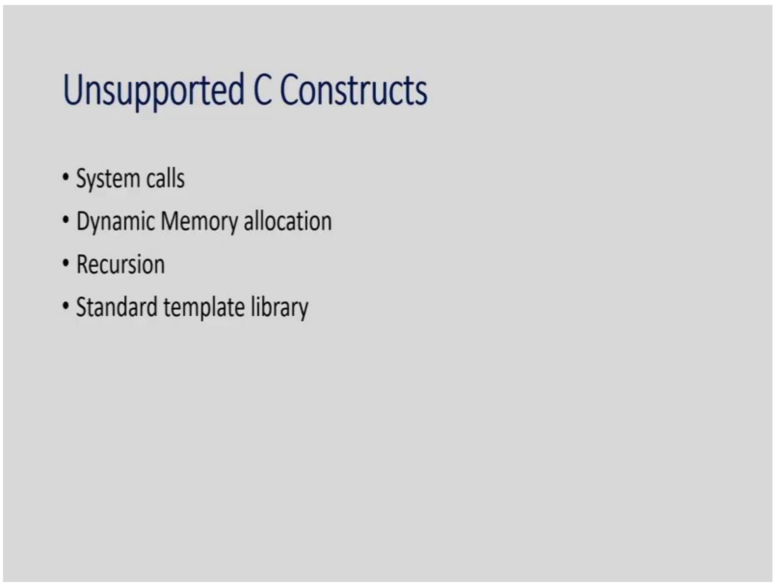
Now, the primary concern is that if you write the code and if you just synthesize into a processor by compilers, we never bother about certain things right. So, which because it is actually mapped into instructions and processors and those instruction get executed sequentially in the processors, right.

So, we usually do not much bother about those part, but once we convert them into this C code into hardware, it is a dedicated hardware, right. So, there is no processor nothing right. It is a dedicated hardware application specific hardware that is going to execute only that C code, nothing else right.

So, once we consider that kind of code, these transformations sometime this certain construct of c might create a lot of effect in the hardware, right. So, if I do not take care of those certain things, then the generated hardware would not be efficient or we are not able to I mean match or the achieve the target or performance or area whatever the targets, right.

So, in todays discussion I am going to discuss as about certain such things that we should keep into mind, for generating a f hardware, right. So, that is why the topic is hardware efficient C coding ok.

(Refer Slide Time: 02:29)

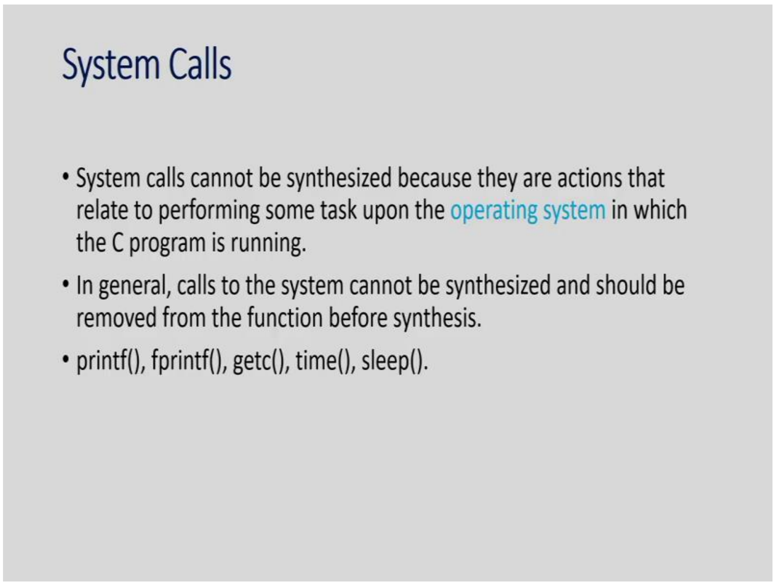


Unsupported C Constructs

- System calls
- Dynamic Memory allocation
- Recursion
- Standard template library

So, to start this discussion we will first see what are the things cannot be synthesize into hardware, right. So, that something we should know and we should also try to understand what is the way out as well right. So, if those kind of construct is already there, how we should able to have a bypass or I should have a path just to make sure that the corresponding code gets synthesized right.

(Refer Slide Time: 02:51)



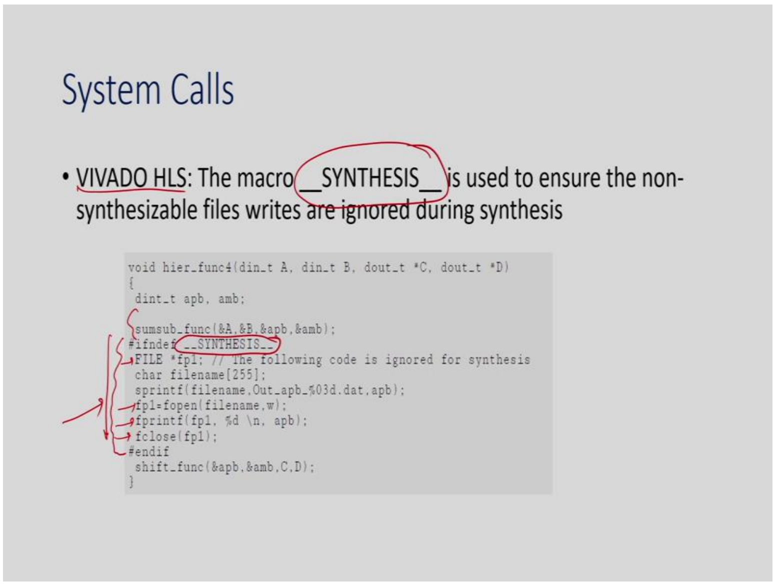
System Calls

- System calls cannot be synthesized because they are actions that relate to performing some task upon the operating system in which the C program is running.
- In general, calls to the system cannot be synthesized and should be removed from the function before synthesis.
- printf(), fprintf(), getc(), time(), sleep().

So, we will take one by one such things. The first thing does not get synthesized is the system calls, ok. So, system calls is like say print f, f print f, get c, time c, sleep, those kind of system calls right. So, commands and the primary reason for not getting supported if you try to understand that is. So, the first of all it needs the operating system, right and for your dedicated hardware there is no operating system which is missing. So, this kind of calls getting processed by the operating system in the general purpose processor, but hardware is dedicated processor, there is no operating system right.

So, that is why we do not able to take this print f, f print f, get c those kind of commands right. So, if we have the basic idea is that we have to remove them very simple right. So, if you if they have the your high level synthesize tool I say [FL] I am not able to synthesize this kind of constraints ok.

(Refer Slide Time: 03:43)



System Calls

- VIVADO HLS: The macro __SYNTHESIS__ is used to ensure the non-synthesizable files writes are ignored during synthesis

```
void hier_func4(din_t A, din_t B, dout_t *C, dout_t *D)
{
 dint_t apb, amb;

 sumsub_func(&A,&B,&apb,&amb);
#ifndef __SYNTHESIS__
 FILE *fp1; // The following code is ignored for synthesis
 char filename[255];
 sprintf(filename,Out_apb_%03d.dat,apb);
 fp1=fopen(filename,w);
 fprintf(fp1, %d \n, apb);
 fclose(fp1);
#endif
 shift_func(&apb,&amb,C,D);
}
```

So, I am in this particular slide, I am going to talk about a specific features of high level synthesis tool call VIVADO which basically says that, if you have that code, how to make sure that you can modify your code minimally, so that your code can be synthesized as well as that particular features is available in the c level right.

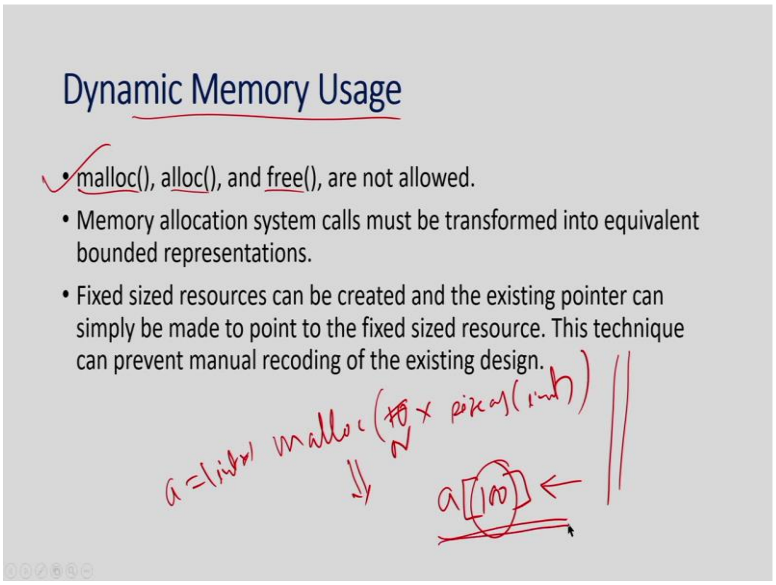
Because once you write a C code you want you can actually do some C based simulation, right. For the for example, you want to do certain say you take an input, you have to get what is the corresponding output. So, that is the simulation at the C base and that time you probably want to read, print certain things in the command prompt, you want to you try to take some data from the file. So, you need a print f print f kind of things, right.

So, the way I should modify my code that such a way the that, if I try to synthesize into hardware, it will ignore those commands and if you try to simulate, it should take that command right. So, that should be the basic idea and that is what exactly that VIVAFO HLS supports. It basically you give you the macro which is called underscore synthesis underscore which basically if you just specify the kind of code here, so that won't be synthesized right. So, that will not be synthesized into hardware.

So, say suppose here what I am doing here in this function you can see I just open a file and then, I just open the file and then I just print certain things in the file and then I close this. It is a very common thing that after doing processing certain data say this function is doing something and then at the output, I want to put into some file right, so which is where is important for in the context of c more simulation, because after simulation I will do open that file name, see whether the correct outputs are there or not right..

So, what if you just put this kind of macro what is going to happen? So, when you do the c simulation, this it will take this part of the code other way it will just simply ignore it right. So, that is what is the this common and it is actually the minimum thing. I do not have to remove it with this macro. I can actually make them disable for hardware synthesis right.

(Refer Slide Time: 05:43)



Dynamic Memory Usage

- malloc(), alloc(), and free(), are not allowed.
- Memory allocation system calls must be transformed into equivalent bounded representations.
- Fixed sized resources can be created and the existing pointer can simply be made to point to the fixed sized resource. This technique can prevent manual recoding of the existing design.

a =(int*) malloc(~~10~~ N x sizeof(int))
a[100]

So, next thing does not support is most important was the dynamic memory usage which is basically this malloc, alloc and free and they are very important in the context of C coding, right. So, because if you if your input is not known to you, in that scenario you use malloc to dynamically allocate memory right. So, and now, if you try to remove them, obviously your features is little bit your code become little bit restricted, but the unfortunately this kind of dynamic memory allocation is not getting supported high level. I mean high level synthesis and there we should understand the reason as well, right.

The primary reason is that suppose finally your code will be mapped to either FBGA or ASIC, right which has a fixed set of memories. It has a say block RAM or it has some ROM and registers and all right. So, now, how can you actually make sure that for a different run or different input, I need say 1000 locations in the block run. Next time I need 5000 location of memory in the block run because the block run time is fixed, right.

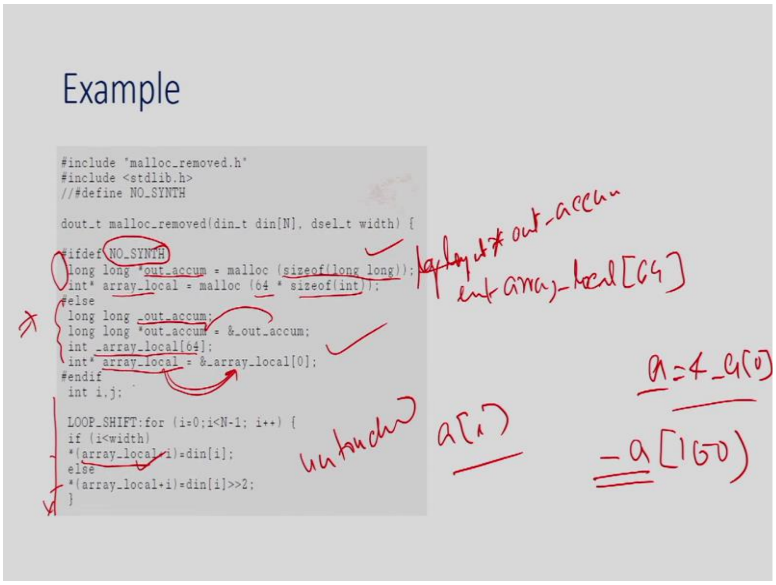
So, you cannot just support that kind of dynamic need of memory in a fixed architecture, right. That is why it is not get supported and the obvious solution is that you have to rewrite your code by replacing that malloc by a fixed size suppose you have a malloc, right.

So, you are doing say a equal to int star malloc and then say 10 into size of int, right. So, what is basically you are doing is or say it is the n, right. So, you basically have you try to use some n integer, so into that arrays right. So, I have to replace this by simply some a 100, right. So, the way I have to do it, I have to assume that in that particular domain what could be the maximum value of n and you have to allocate this much of memory.

Obviously, it is something is not so good because maybe your input may not go into 100 all the time. It may be always around 10-15. So, that means mo many of the idea the RAM part or the array part will be unused, but because of this dynamic nature cannot be support in hardware, this is the only way out things and this is particularly doable.

Once you are an application engineer, you know the domain and you know the what is the input possibilities and based on input possibilities, you can actually identify what is the maximum possible input size and then based on that value, you can statically allocate memory, right instead of dynamic memory allocate, ok.

(Refer Slide Time: 08:14)



Example

```
#include "malloc_removed.h"
#include <stdlib.h>
//#define NO_SYNTH

dout_t malloc_removed(din_t din[N], dsel_t width) {

#ifdef NO_SYNTH
 long long *out_accum = malloc (sizeof(long long));
 int* array_local = malloc (64 * sizeof(int));
#else
 long long _out_accum;
 long long *out_accum = &_out_accum;
 int _array_local[64];
 int* array_local = &_array_local[0];
#endif
 int i,j;

 LOOP_SHIFT:for (i=0;i<N-1; i++) {
 if (i<width)
 *(array_local+i)=din[i];
 else
 *(array_local+i)=din[i]>>2;
 }
```

So, this is the solution I am going to again show the very interesting features of this VIVADO again, VIVADO HLS tool again. So, where again the my desirable things is that for C simulation. I want to keep my dynamic memory allocation features alive and for hardware synthesis, I want to use a static allocation right.

So, if I want to do that say that things, then what I can do, I can use the macro which is NO SYNTH right. So, if I just do this NO SYNTH, then if I just put this, this is the dynamic memory allocation part. So, what is going to happen it is basically it do not be synthesized into hardware. So, I am going to and then, if it is in the case of stat synthesis that else part will be executed right.

So, the if part is the dynamic memory allocation part and this if else part is the static memory allocation right, so you can see here that I am a actually use a out accumulator to a long int right. So, long int pointer right long int and then I just use a array local which is a 64 bit integer array, right. So, this is basically nothing, but a star out and this is basically some array which is array local 64, right which is of size int int and this is long right.

So, what I am doing here is that I am just keeping these pictures alive for simulation, so that I can actually dynamically allocate memory and for this hardware synthesis what I am doing here, I am just basically creating an long long int variable which is underscore out and then I am just taking that pointer to this star out accumulator right.

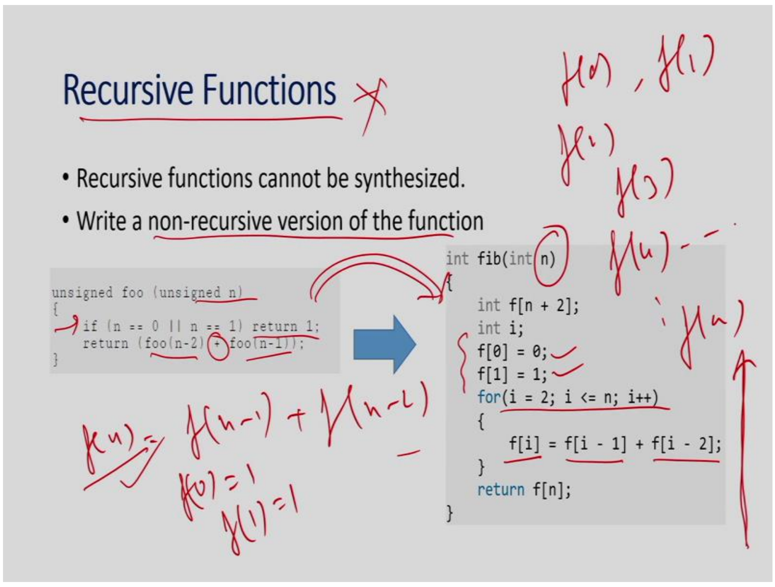
So, this underscore out accumulator you can under see and it is underscore array local which is of 64 and then, array local is pointing to the first low address of that. So, if you just do this, you do not have to task rest of the code right. So, it will be untouched. So, because this so now, see here I am actually using as a array right. So, as a pointer array local so basically this is array local plus i right.

So, now still since my this is underscore array local 64 and this is pointing to the first location of this, so this will actually work for both right. So, the for do it will work for this as well as this. Similarly for accumulator also it will work like that. So, the idea here is I just I do not want to because if you have to change everywhere, it is too much of work right. If you replace all this star array to some a i which is lot of works right.

So, I do not want to touch this is the most of the code is there right. So, I do not want to touch that part of the code. Only thing what I do, I just at the start of the declaration I just use the NO SYNTH. I say that for synthesis, no synthesis. You just use the dynamic memory allocation and for synthesis purpose uses static allocation and I actually you use some a dummy array right. So, whatever the arrays if the variable name is a I am using underscore a, this is my static array right. So, static array and then what I am doing is basically, I am just putting this as a address of underscore a 0 right. That is what I am doing.

So, now I can use that pointer a for the rest of the code. So, as a result what is happening here is, I do not have to touch the rest of the code. Only for the declaration part I modify this way and I can use both the features dynamic memory allocation for c synthesis and static allocation for hardware synthesis, ok. So, that is what is the solution.

(Refer Slide Time: 11:58)



Recursive Functions

- Recursive functions cannot be synthesized.
- Write a non-recursive version of the function

```
unsigned foo (unsigned n)
{
  if (n == 0 || n == 1) return 1;
  return (foo(n-2) + foo(n-1));
}
```

```
int fib(int n)
{
    int f[n + 2];
    int i;
    f[0] = 0;
    f[1] = 1;
    for(i = 2; i <= n; i++)
    {
        f[i] = f[i - 1] + f[i - 2];
    }
    return f[n];
}
```

$f(0), f(1)$

$f(2)$

$f(3)$

$f(4)$ - -

$f(n)$

$f(n) = f(n-1) + f(n-2)$

$f(0) = 1$

$f(1) = 1$

The next one is the recursive function. Again it can not get supported. The reason is again understood that recursive function is whenever I call from one function to again, we need we maintain some stack and then once you come back, we have to retrieve the state of the previous function and it will it will go multiple times right and the another important point is the recursive function. It can call many times. I do not know how many time it will call right but in the hardware you have you have to define the resource requirement for that function right.

So, if say it calls say 10 times probably you need 2 copy of the function. If you call it say for 20 times while you need 10 copy of the function right. So, I am just giving some examples right. So, which basically saying that based on the number of recursive calls the hardware requirement may vary right which cannot be supported once you decide the fixed size right.

So, as a result this recursive function is not supported, ok and again what is the solution that you have to write a non-recursive version of the function right. So, I have taken that Fibonacci number examples. So, you know that it is very well known that if you give a number n, so Fibonacci number is nothing, but f of n minus 1 plus f of n minus 2 where f 0 is equal to 1, f 1 equal to 1.

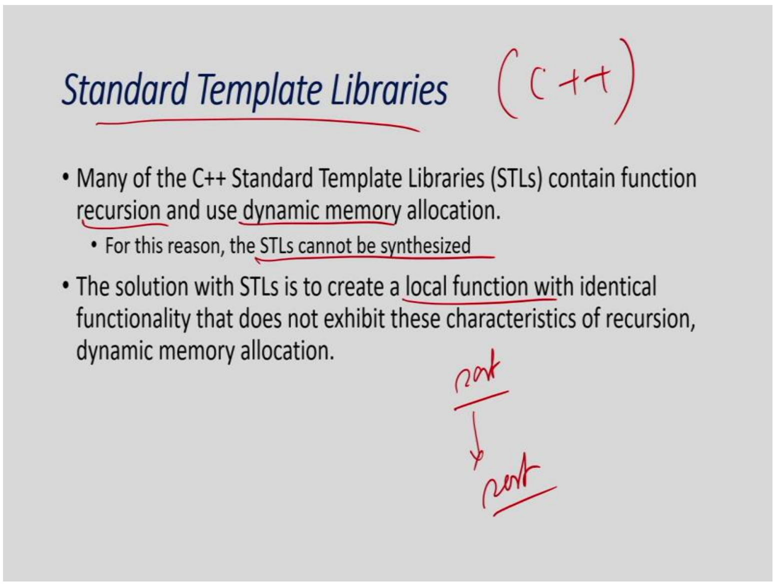
So, this is the best condition that if n equal to 0 or 1, you return 1 or else you call recursively two times and then, you add it right. This is what the recursive called two line function and it will actually do if you give any n, it will give you the Fibonacci number nth Fibonacci number, but since this is a recursive and then the number of recursion depends on this value n, it will not support it right. So, you have to convert into a non-recursive version which is something is written using a for loop n. So, for a given Fibonacci number n, I just define my f 0 and f 1 is basically 0 and 1 and just I declare that f i equal to nothing, but f i minus 1 plus i minus 2.

So, it is basically here what I am doing is, actually I am doing bottom up right. I calculate I know my f 0, I know my f 1. I am going to calculate my f 2, then I will calculate my f 3, then I calculate my f 4 and so on. This is how I calculate till f n, right. So, then I will calculate f n here. What is happening, I am just recursively call f n minus 1, then f n minus 2 to f n minus 3, f n minus 4 and so on.

Finally this top down recursion reach them f 0 and then, it is in bottom up manner you will fill up the value, right. So, so this recursion can manage it, but here I have to do it in the that unroll version of the recursion where I am actually calculating bottom up. I will calculate f 2 f 3 and so on right.

So, this is a non-recursive function. So, if you have this Fibonacci code, you have to replace by this kind of for loop iterative version of this code and then, it will be synthesized right. So, the bottom line is that recursive function n is not supported and it must be replaced by a and non-recursive function of the code, ok.

(Refer Slide Time: 14:56)



Standard Template Libraries (C++)

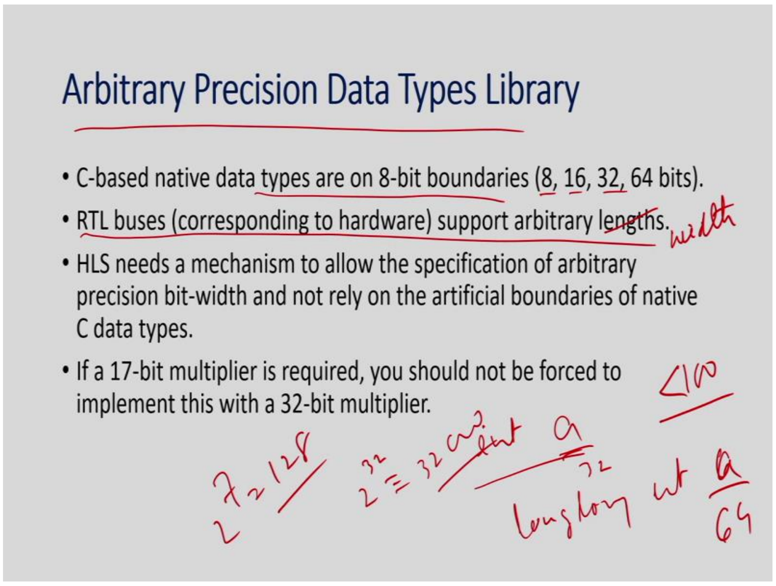
- Many of the C++ Standard Template Libraries (STLs) contain function recursion and use dynamic memory allocation.
  - For this reason, the STLs cannot be synthesized
- The solution with STLs is to create a local function with identical functionality that does not exhibit these characteristics of recursion, dynamic memory allocation.

the same thing applies for standard template library which is basically in for case of C plus plus. We have STL because C plus plus and C has most of the features common except certain additional features of C plus plus.

So, I mean this C plus plus this standard template libraries. Again it is not supported because of the primary reason is that it actually most of the STL actually use recursion and dynamic memory allocation because it is a standard template library, because it is a template you never know what is the size of the array or say what is the value of the n and so it is always based on the dynamic value, right.

So, because of that this cannot be synthesized, right and again solution, what is the solution because if you have C code or C plus plus code where you have used basically that is STLs what is a solution you have to write a local function of the same function suppose you have use a sort STL, right. So, there is a STL function sort. So, you and you have used it in some code now, you actually write a sort function locally, right. So, non-recursive function of the sort function locally and then, use it right. So, this is what you have to do it right. So, you cannot use the library functions ok, the template function ok.

(Refer Slide Time: 16:13)



Arbitrary Precision Data Types Library

- C-based native data types are on 8-bit boundaries (8, 16, 32, 64 bits).
- RTL buses (corresponding to hardware) support arbitrary ~~lengths~~. width
- HLS needs a mechanism to allow the specification of arbitrary precision bit-width and not rely on the artificial boundaries of native C data types.
- If a 17-bit multiplier is required, you should not be forced to implement this with a 32-bit multiplier.

<16

$2^7 = 128$

$2^{32} = 32$ int a 32

long long int a 64

So, these are the four things is not supported the four things. I just mentioned system calls and then, dynamic memory allocation recursion function and standard template libraries. Other than that most of the syntax of C code is supported certain things like function pointer and those things is not supported, but I mean abstractly those are very not so commonly used, right. So, most of the features of like strap union pointers multi multi-dimensional arrays for loop while loop switch case everything is mostly get supported during C to RTL synthesis.

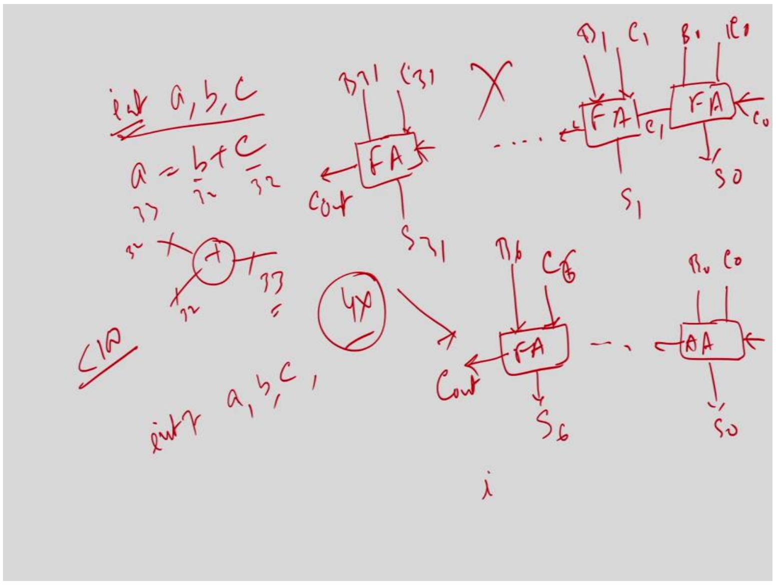
So, I now move to my discussion on the efficient coding part and the part here for which synthesized its code C is synthesizable, right. The one of the most important is the arbitrary precision data type. Let me try to explain what is that and what is the problem and what is the kind of solution we have, right.

So, in C, we always have a fixed data types right. So, data types up of say 8-bit, 62-bit, 32-bits or 70-bit I mean 64-bits and soon right. So, for example if you declare a int a which is basically 32-bits right, I am assuming I mean it might vary to processor, but just for explosion I am assuming it. So, if you declare say long long int, then it is basically a 64-bit right.

So, and you have no option to declare a 7-bit integer or say 13-bit integer in a C, right. You either you have to go for int or long int long long int unsigned long int and so on, right. So, your the precision of the data width is fixed is either 32-bit or 64-bit or 80-bit and so on right, but in hardware this RTL buses of any arbitrary length or every every every arbitrary width right. You can have a data path of 33-bit 7-bit 32-bits 16-bit 15-bits and most of the time for your application, you declare something int, but you know this value is always less than 100 right.

So, it is you know always the values less than 100 and in that case you do not need 32-bits because 32-bit can represent some 2 to the power 32 which is something some 32,000 something right so which is very high number. I need only 100. So, for representing 100, I need 7-bits right what is which is basically 128, right. So, why I am going to use 32-bits where I can actually represent my circuit using 7-bits, ok.

(Refer Slide Time: 18:58)



And the impact on the hardware synthesis or the high level synthesis is huge. I will just take an example and explain. Suppose you have a you just declare say int a equal to b plus c where a, b, c all are int right. So, basically you have this a b c and you are doing a equal to b plus c, suppose b and c is something value is known.

So, now for this case if you think about they are all 32-bits. So, this is 32-bit, this is 32-bit and then this output will be 33-bits right. So, basically you need a adder which is of input is 32-bit, another input is also 30-bit and output will be 33-bits right. So, now if you assume it say basically the basic implementation of the adder in the hardware, so you need kind of full adder right. So, you need a full adder which take a single into the addition of single bits, right.

So, it will take a b 0 c 0, the zeroth bit and the carry in 0 and it will give the s 0 first bit and then, it will give the c 1, then you do this c 1 b 1, it will give you s 1 and c 2. This way it will go to the unit 32, such full adders right because this is a full adder circuit. So, you need c 31 b 31 and it will give you the s 31 and this is your final carry out, right. So, that is the 33, 33rd bit right.

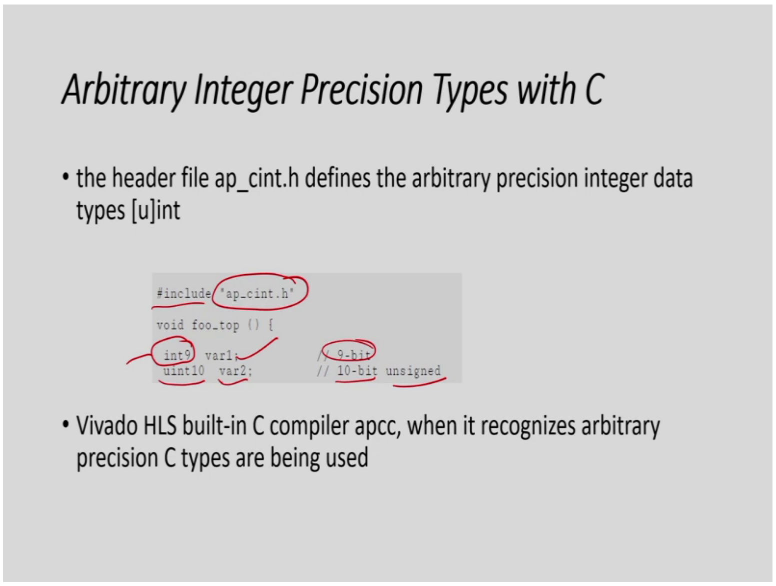
So, now if you assume that this full adder is around 4-5 gates right, so you need 32 full adder to represent this right and now assume that you know that your data of a, b, c is always less than 100, right. So, then you need basically a 7-bit adder right. So, if you just design a full adder of 7-bits, I am just quickly write. So, this way I need only 7 such full adders right which will actually do this c 6 and then b 6. This is my s 0, this is my s 6 and this is my c out.

So, you can understand you because you declare int, you need 32 such full adders and, but for your requirement because your input is only 7-bits, you need only 7 full adder right. So, the you can understand that it is almost four to five times extra, right. So, four time 4 x extra area.

So, just because you have declared int, your area of that full adder or the adder will be four times of your required one, right and you cannot do anything because you have declared int and since you have declared int that hardware will assume that your input can be till 32,000 value and to do a 32 values and hence, it will create a full adder of 32 bits because it cannot reduce it. It has no idea that it can actually can be it purpose can be solved using only 7 bits, ok.

So, this is something is a huge area overhead just because of the fixed the data width right the solution. Obviously, c does not have that features right. So, c does not have the syntax of declaring a 7-bit integer right.

(Refer Slide Time: 22:18)



Arbitrary Integer Precision Types with C

- the header file ap_cint.h defines the arbitrary precision integer data types [u]int

```
#include "ap_cint.h"

void foo_top () {

 int9  var1;               // 9-bit
 uint10  var2;             // 10-bit unsigned
```

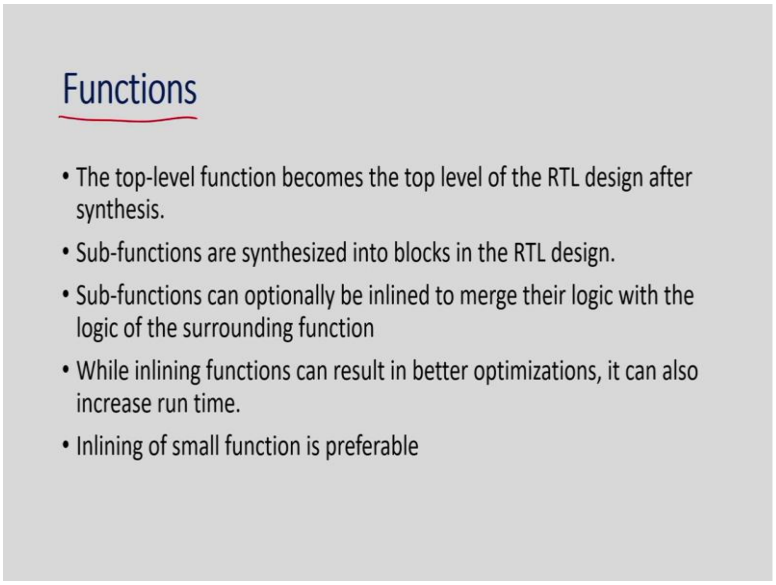
- Vivado HLS built-in C compiler apcc, when it recognizes arbitrary precision C types are being used

So, for that this the high level synthesis tool comes with a library which is called arbitrary precision library, right. So, for every tool like VIVADO HLS or main every tool has such libraries and those libraries are actually publicly available as well right in that libraries, that arbitrary precision things are defined. If you just add that, then you cannot just if you just right int 9 this variable is nothing, but a 9-bit integer right and if you just you write u int 10, then this variable is nothing, but 10-bit unsigned integer. By default it is signed and this is unsigned.

So, if we just add this library to the library where your the libraries are available for your GCC and if you just write this code, this code will be synthesis means we can be simulated in GCC as well and if you just write this the hardware during if you just write this int 7 for this example that I am taking that instead of int a b c if I just write int 7 a b c.

So, it will create this structure, not this structure and as a result by basically just doing nothing you will get 4 x area less area for this adder, right. So, this is what is the arbitrary precision data types and by defaults c does not support and most of the tools I have I have publicly that commercial tool available, they have that arbitrary precision thing. So, you should actually include that and you should write our code such a way that we avoid this fixed type data path and it has a impact, huge impact on the overall area of the design ok. So, this is another important thing you should keep into mind.

(Refer Slide Time: 23:55)



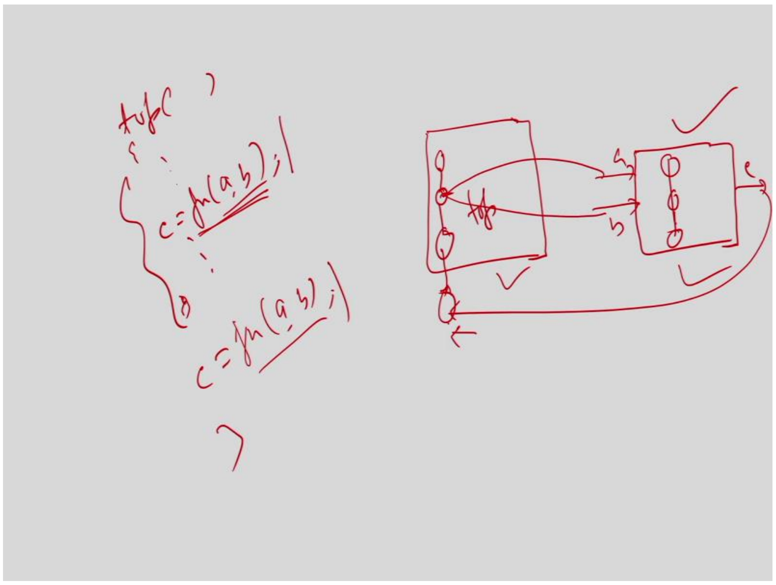
Functions

- The top-level function becomes the top level of the RTL design after synthesis.
- Sub-functions are synthesized into blocks in the RTL design.
- Sub-functions can optionally be inlined to merge their logic with the logic of the surrounding function
- While inlining functions can result in better optimizations, it can also increase run time.
- Inlining of small function is preferable

The next things we want to discuss is the function right. So, function is very common in our C code because we use function for two purposes. The first purpose is that to make our code modular, more readable instead of a one flat big code. We just put these the component that you calculating a particular data, we will just put a function and calculate that.

So, that will make our code modular and the second advantage is that if the same particular code is repeating many places, I can just copy the whole thing into a single function and then I call multiple times. It will reduce the code size. So, function has this two advantage making more code modular, readable and also read make the reduce the code size, ok. So, now you think about how once we convert this C code into hardware, how the function gets implemented right.

(Refer Slide Time: 24:50)



So, so you remember that we always have a top function. It is not the main function you remember, because main function is using that test bench and from there I am going to call them my primary function, the top function right. So, here some operations are there, then you are suppose you are calling a function with say a and b and it is retaining c, right. So, and then you are doing this.

So, once you do this, so in hardware this top will be created a module right. So, it is a hardware module which will decide the kind of resource required for this particular function, how many adders needed, how many multiplier needed, what is the mux that exact interconnections, everything is get decided here and that is the hardware. It is top right and this from top you have a controller FSM also, right. So, it will basically have a controller and say in this particular state this particular function is getting scheduled. So, for this particular function, it will usually it will create a another module right.

So, it will create another module and again in that module the resource require for the hardware is determined at the exact schedule is schedule allocation binding everything happen independent of the top function and the resource get defined here. So, in this particular set what I do, it will have some port. So, it will create a two port because it has two input a and b and it will create a output c, right. So, what is going to happen now in this particular state the value of a and b will be assigned right.

So, as a result that this value will come here and this particular module will be going to execute. So, it is going to run every time, but at this time only the data is valid and probably if say it is a say three cycle say it has say three latency three. So, basically at this time I am going to read the data, right. So, in this controller FSM here, I will do the write enable equal to 1, so that this valid that I am going to read right.

So, this is how this module get implemented. So, having a function has both disadvantage and advantage in the context of hardware synthesis. So, the let me try to explain the advantage first. The advantage is that the function is something is going to create a different module right.

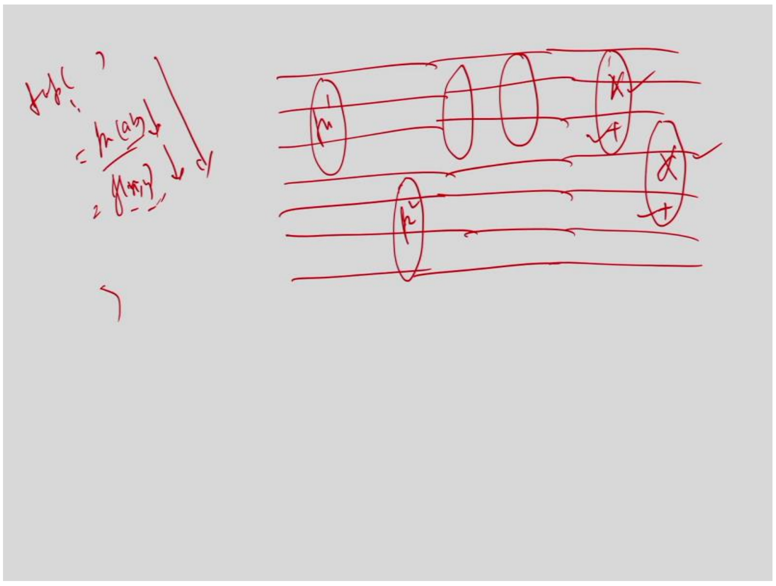
So, now if this function get call multiple times right, then a it since a single module. So, I can actually for the multiple calls I am going to use the same module right. So, whereas if it is a straight line of code, so there will be a the copy of the of the function here and the copy of the function here. So, in that case what is going to happen the code size of the top is large and there is no guarantee that it will create only one instance for the resource of for both right, because it will be interleaved with other operations, right.

So, that way if there are multiple calls, the resource requirement will be less and one disadvantage is that this function is basically independent right. So, as I mentioned that once you have written a function, the scheduling allocation binding will be done independent on the main function.

So, as a result the resource sharing possibility is reduced right. So, if probably if you have a single copy of the function, there is not multiple calls and if you put the whole code here, then the complete operation could have a better schedule right. So, parallelism can improve and it can have a better schedule. Total latency can be reduced right or maybe total resource requirement is also reduced because you do not have to create two copy of the adder and so on right but if there are multiple calls, obviously creating a module is something should give benefits because ideally it will create a separate module and the I do not have to create multiple copies of the same function, ok.

So, that way it might give you the benefits. So, there are one more important point here because once you have two copies during the schedule, there is no guarantee that their schedule will be not overlapping right.

(Refer Slide Time: 28:58)



So, let me just try to example say suppose you have I mean the same example I am taking. You have a function call here in that top and then the next line, you have another function. The same function you call with say this is a b and this is you are calling with x y right. So, two calls of the same function.

So, now there is no guarantee that the scheduler will schedule this non overlapping manner, right. So, this is my first call, this is function call 1 and this is my function call 2. There is no guarantee that is going to happen. If it is happen I can use a single module for the hardware, but since the there is if there is no dependency between this function because it is taking x and y and it is taking and b. So, there is no dependency between these two function.

So, a scheduler might schedule them parallely right. So, if it is getting scheduled exactly in parallel, so you need two copies of the hardware because if something is running parallel, you cannot share the resource right. So, then the resource requirement will be double even if there is a function, but because of the schedule the resource requirement get doubled right or the scheduler might schedule it partially overlap manner also right because of some dependencies right. So, it is getting scheduled here and it is getting scheduled here second call.

So, in that time there may be some opportunity of resource. Obviously, you cannot have a single module because there are some overlap, but you can actually have say some portion is overlapping right. So, for example suppose this is a multiplier it is a two cycle and this is adder. So, this is my multiplier and this is adder. So, you can see here this multiplier is actually non overlapping and the adder is also non overlapping

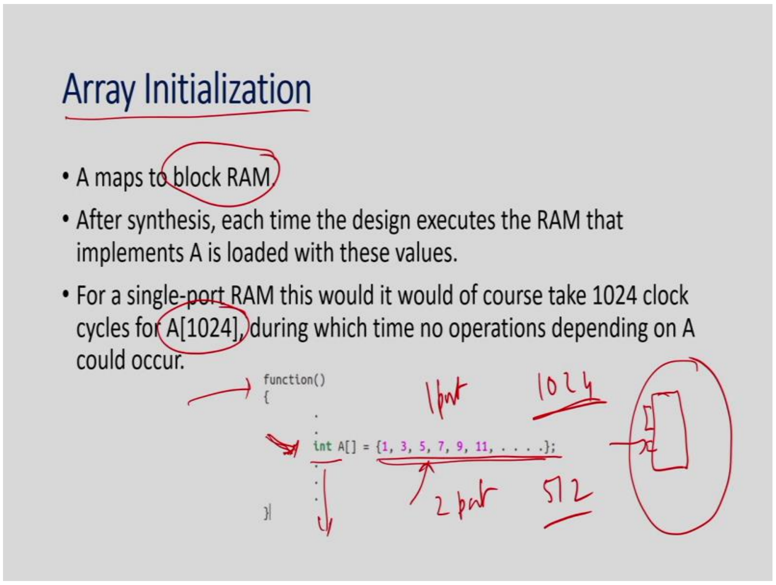
So, if is that is the case probably I can have a single module and then I can actually use it a pipeline manner or I can actually use the same adder to schedule this two addition operation or I can use the same multiplier to execute both the multiplication because they are although these functions body is overlapping, but the operations they are the kind of operation which is mutual same type of operations they are not overlapping right.

So, the bottom line of all this discussion is that once you have multiple functions, it is not guaranteed that you will have a single copy of the module in the hardware because it depends on the scheduler and you cannot and the scheduling is something where you cannot you usually do not have control whether they will actually schedule in a sequence manner or in parallel manner or partially overlap manner and based on this schedule, it will decide the scheduler decide your resource requirement will change, ok.

So, that way the modular function whatever you do in the in the c, where you write this way this will be executed first, then it will be executed right. This is a sequential execution right, but in hardware once you map, schedule it, their execution may not be sequential right. So, they may be parallel, partially overlapped or completely sequential. So, based on their execution things which is determined by the scheduler the exact requirement of the hardware will be determined ok.

So, obviously it is you from this discussion you understand that it is advisable to inline the small function because then it will it will not create multiple copies and the optimizations within the flat code is more right. So, but if the particular function call many times, then we should actually keep the function as it is and then we should synthesize in hardware and based on the schedule the resource requirement will be determined ok. So, this is about function.

(Refer Slide Time: 32:26)



Array Initialization

- A maps to block RAM
- After synthesis, each time the design executes the RAM that implements A is loaded with these values.
- For a single-port RAM this would it would of course take 1024 clock cycles for A[1024], during which time no operations depending on A could occur.

```
function()
{
    .
    int A[] = {1, 3, 5, 7, 9, 11, . . . .};
    .
}
```

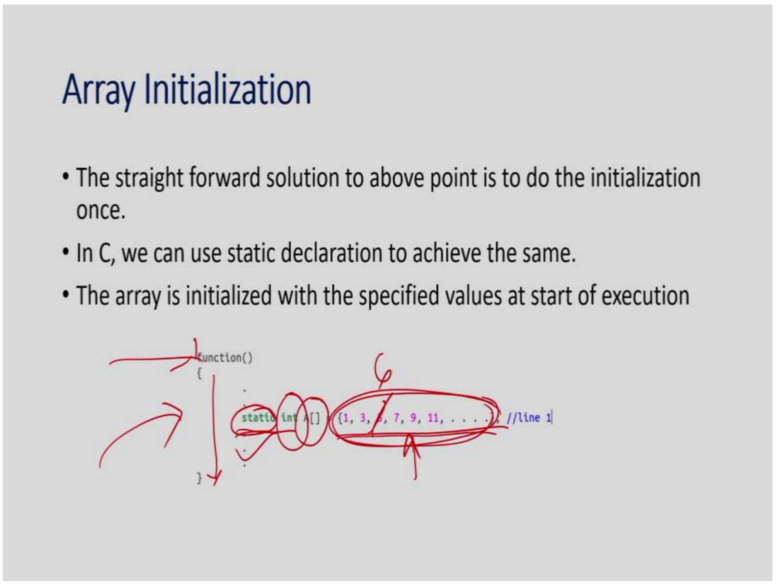
So, there is another important thing which we usually ignore during C coding that array initialization, ok and I am going to explain once you initialize an array what could be an impact in the hardware, ok.

So, we have already discussed that array maps to block right because it is a continuous location some data. So, it gets mapped to block RAM, ok. So, now you think about you have just in a function you just declare int a and it has some data right. So, and say suppose this a size is 1024, ok.

So, once you call this function and say assume it is create a separate module, right so it is create a separate module and it has a block RAM and at the start of the function you have to store this data into this block RAM. And suppose this particular block RAM has a single port right, so how many cycle it will need to just store this data into this? Block RAM is 1024 cycle right because it is a single port. If it is a dual port, you need 512 cycles. So, you can understand this particular array initialization has lot of impact in the hardware. Just the latency will increase by 1024 because of this array initialization, ok.

So, and you cannot avoid it. You have to wait the rest of the operation here has to wait for this 1024 cycles, because this data to be loaded into the memories, ok. So, many a time this particular declaration is something is could have been avoided in the sense that if that particular function call only once right and then it is just the one time and I am not going to use it ever, right.

(Refer Slide Time: 34:17)



Array Initialization

- The straight forward solution to above point is to do the initialization once.
- In C, we can use static declaration to achieve the same.
- The array is initialized with the specified values at start of execution

```
function()
{
        .
        .
        static int A[] = {1, 3, 5, 7, 9, 11, . . . .}; //line 1
        .
        .
}
```

So, one what is the solution here is you just add a static right. So, if you just add, if you make a static array, then what is going to happen? Let first understand for the FPGA mapping. So, for if you just do this. So, it that note this is static means what it is this particular array will remain valid for the throughout the execution of the program, but the access will limited to this particular function, right. If this array cannot be accessed from any or other than this function, but it will remain valid for throughout the execution of the program, so if you call this function second time you it will not get assigned. So, whatever the updated value is stored, it will remain there ok.

So, now suppose you first let us assume the simpler thing that this array is this function is getting call only once, ok and in that case what happen that you in the for FPGA after the synthesis of everything that code become a bit stream, right. So, that bit steam is basically because is mapped to the FPGA board and the FPGA is a kind of unit this 1 0 signal actually determine the connections inside the switch and also content of the block

RAM and all right. So, once you declare a static that content of this array or the block RAM will be during the initialization of the data, the block RAM will contain this data.

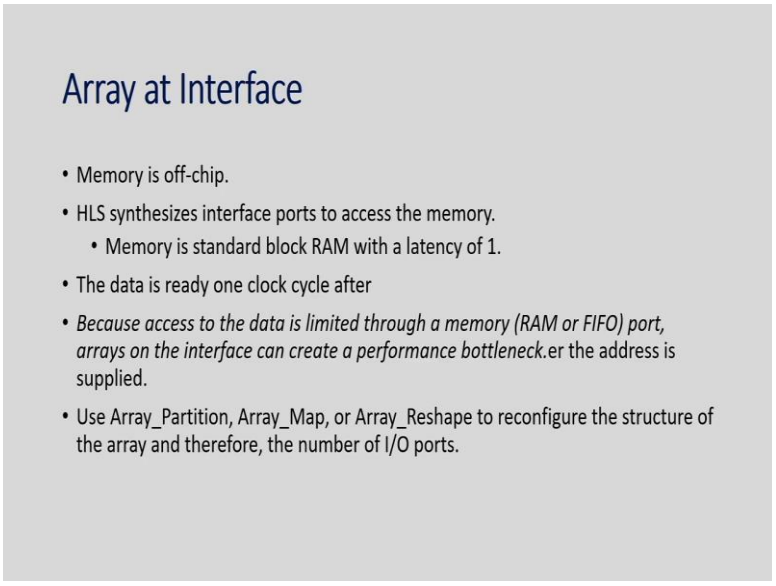
So, it won't take any extra cycle for that ok. So, that will give a huge benefit right. So, if you have a single call, obviously you should make it static because it will save you 1024 cycles; right the size of the array. Now, the question might come that if this particular array is something, I am going to call this function many times right and every time I want to declare this.

So, obviously if you need this data every time, there is no solution it you have to combustion that right. You have to keep it as normal int, because you this should be your initial data right. If it is modified to 6, I do not want that 6 because in the second call, I again I need 5.

So, if that is the case you have no solution you have to use int, but if it is the case that in the next iteration I need the updated value updated value, not the initial value. You must use the static right. So, that will give you that benefit alright.

So, to summarize this array initialization is that array initialization has a lot of impact in the latency in the hardware because the data has to be stored in the block RAM. Every time to avoid it, we should use static if it does not change the functionality of the program, ok.

(Refer Slide Time: 36:58)

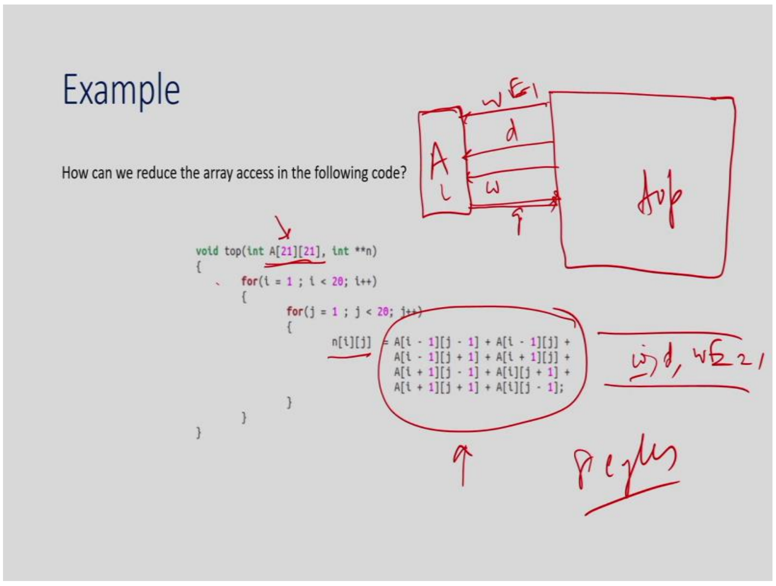


Array at Interface

- Memory is off-chip.
- HLS synthesizes interface ports to access the memory.
  - Memory is standard block RAM with a latency of 1.
- The data is ready one clock cycle after
- *Because access to the data is limited through a memory (RAM or FIFO) port, arrays on the interface can create a performance bottleneck.*er the address is supplied.
- Use Array_Partition, Array_Map, or Array_Reshape to reconfigure the structure of the array and therefore, the number of I/O ports.

So, now I am going to move to another concept which is called Array at interface ok.

(Refer Slide Time: 37:03)



Example

How can we reduce the array access in the following code?

```
void top(int A[21][21], int **n)
{
        for(i = 1 ; i < 20; i++)
        {
                for(j = 1 ; j < 20; j++)
                {
                        n[i][j] = A[i - 1][j - 1] + A[i - 1][j] +
                                  A[i - 1][j + 1] + A[i + 1][j] +
                                  A[i + 1][j - 1] + A[i][j + 1] +
                                  A[i + 1][j + 1] + A[i][j - 1];

                }
        }
}
```

So, many a time we write function the top level function as array at the as a input to the program, right and as I mentioned the top will create a module right.

So, top will create a module. Now, the question is this is my top hardware, hardware which is corresponding to the top. Now, the question is where this A will be there is it inside the top or outside the top because this is an input to the top, it must be outside of the top. So, this hardware HLS tool assume that A that block RAM is somewhere outside of the this particular module, right and how it will can access? It will give you this signal right write data write enable right.

So, these signals it will generate. So, it will create some output signals and it will also have some q which is the inputs, right. So, if you say write enable equal to 1, so it will get the data.

So, for this array it will create those interface signals and it will assume. So, in particular say time step in a particular clock it want to read something it will assign the value of this w d and w enable accordingly, so that this signal the corresponding data it will assume that it will be return into the this particular array. Similarly, if you want to access something, it will recreate w equal to 1. It will give the write address and then it in the next clock, it will read the data q, ok.

So, that is what is going to happen for such array at interface because it is an external entity. It will create a block RAM, it will the tool will assume it is it is somewhere else and it will create the interface signals and based on the read and write, it will create the it will give those value accordingly in that particular cross step and the corresponding data will read or write from the block, right.

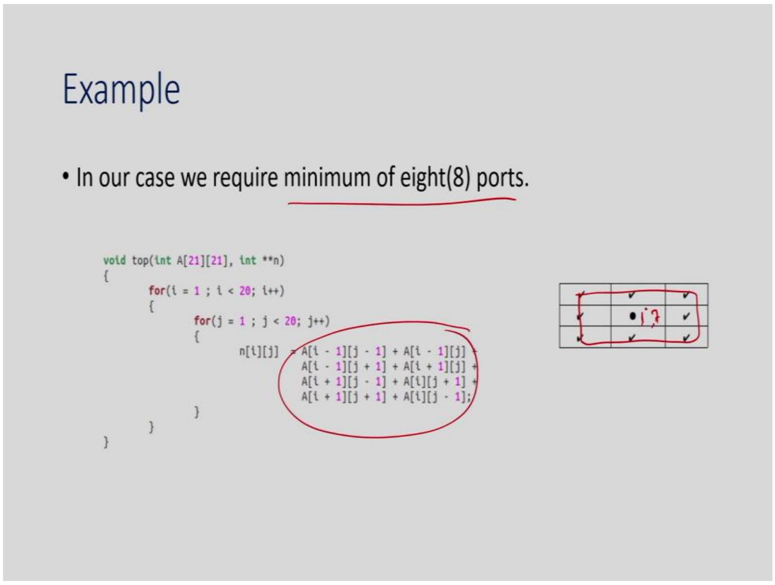
So, that is how the array at interface get synthesized, ok. That is first you should understand and now we will come to the problem. It is again the array access and array access the problem we already discussed that in the primary problem of this array is the block RAM, the array gets mapped into block RAM and block RAM has a limited number of ports, right.

So, as a result if you try to access that array many times the in a particular clock, it is not possible because by default it will have a one or two ports. So, if there are many reads, you have to schedule them in multiple clocks right. So, I just take an example similarly to this problem say suppose in this array I have a 2 d array and then I am reading two complete i j. I am just reading these 8 elements from the array right. So, obviously an it is a if this RAM is mapped to a block RAM, it will need 8 cycle just to read the things because it is a single block RAM and then only this addition can happen.

So, it get lot of time to execute it right. So, obviously whenever this array is there in the interface, the first of all the way the things that we should understand that the same solution that we have discussed for the adder to block RAM mapping that you try to reduce the array access right. So, that that you must do right.

So, you should make sure that array access should be minimized and it should be stored locally and then you reuse, so that the interface communication is less and the read time is less and you can actually execute the program faster manner, right.

(Refer Slide Time: 40:25)



Example

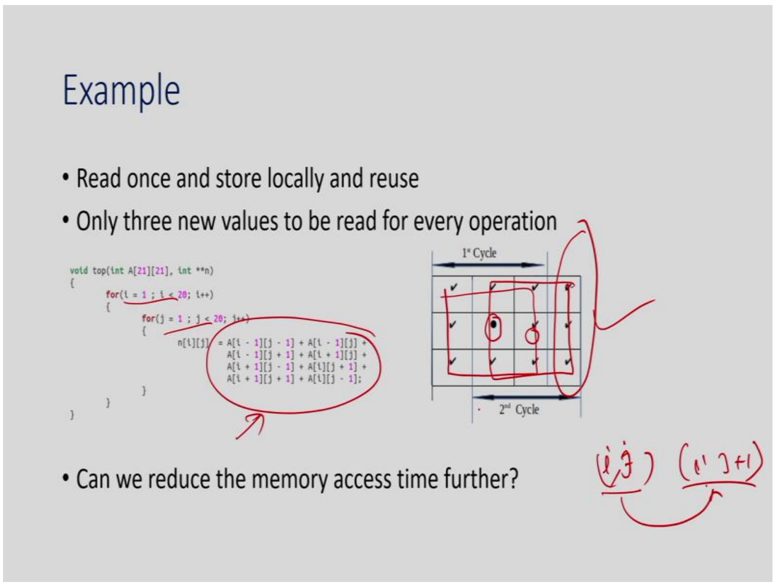
- In our case we require minimum of eight(8) ports.

```
void top(int A[21][21], int **n)
{
        for(i = 1 ; i < 20; i++)
        {
                for(j = 1 ; j < 20; j++)
                {
                        n[i][j]  = A[i - 1][j - 1] + A[i - 1][j] +
                                   A[i - 1][j + 1] + A[i + 1][j] +
                                   A[i + 1][j - 1] + A[i][j + 1] +
                                   A[i + 1][j + 1] + A[i][j - 1];
                }
        }
}
```

So, just to take this example since I have shown that I am reading 8 elements. So, what is that element? So, if it is this is i j these locations are nothing, but this 8 location right. These 8 locations I am writing. So, this and then I am doing some addition right.

So, now if you want to do this every for every data I have to read 8 data's right. So, either you create 8 ports and you can assume that there are 8 copy of the data because you say read array you can actually think about there are 8 copy of the block RAM. It might assume or it will need 8 cycle just to read it, right. So, which is too much either there are lot of too much of interface signals or too much of time right. So, both is bad right. So, how can I reduce this access for this particular program, let me try to explain ok.

(Refer Slide Time: 41:08)



Example

- Read once and store locally and reuse
- Only three new values to be read for every operation

```
void top(int A[21][21], int **n)
{
        for(i = 1 ; i < 20; i++)
        {
                for(j = 1 ; j < 20; j++)
                {
                        n[i][j] = A[i - 1][j - 1] + A[i - 1][j] +
                                  A[i - 1][j + 1] + A[i + 1][j] +
                                  A[i + 1][j - 1] + A[i][j + 1] +
                                  A[i + 1][j + 1] + A[i][j - 1];
                }
        }
}
```

- Can we reduce the memory access time further?

So, here let us try to understand that what is about this i j and the next value right because it is i and so, the first it will be i j and then, it is be i j and then it is i j plus 1 because j is varying right.

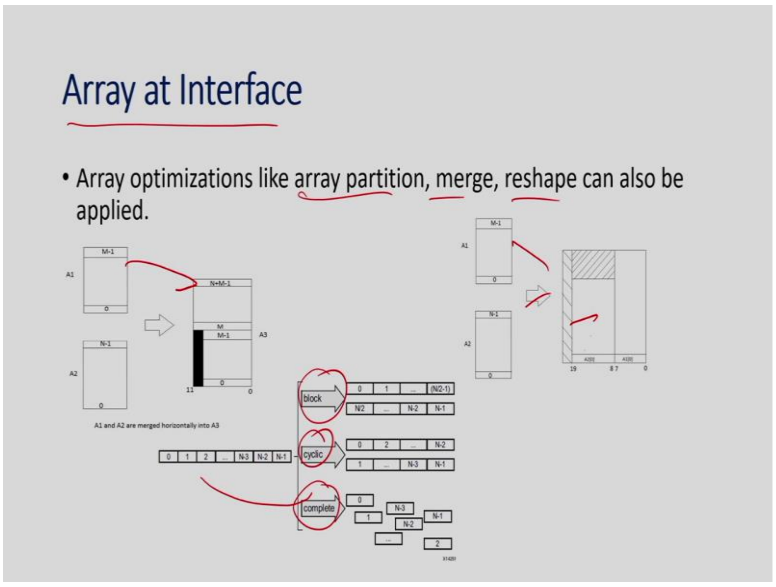
So, suppose this is my a j and for that this is the data I am going to read, right. So, these are the data I am going to read and for this is my i j plus 1. For that I am going to read this data. I can see here between these two cycle only the new data is these three, right.

So, this is for i from when I go from i j to i j plus 1, I am only reading the data i. This three data not the eight data. So, the idea should be that I should not write this code like this way. I should store this data's this 6 data into some temporary variable and then every time I am only going to read this 3 new data and then I can actually calculate this.

So, this I just remain keep that as a homework for you that you try to rewrite the code, such that in every clock I am not going to read 8 data. Rather I am going to read only 3 data and I am going to keep the data which is ready the previous cycle into some temporary register and I am going to read it again. So, this will reduce the number of read from the array and as a result either the number of cycle to read the things will be reduced or if you assume that there are multiple copy of the pro.

So, the number of interface signals will also get reduced ok by this kind of modification.

(Refer Slide Time: 42:45)



Array at Interface

- Array optimizations like array partition, merge, reshape can also be applied.

And also once you have an array interface, hence it is since it is in the outside based on the access pattern, we have already discussed various array optimization technique like array partition array merge array reshape and so on. So, I can apply the same strategy here also because it is an array. Only thing is that array is not inside the module; it is in the outside.

So, it impact the number of interface signals or the latency, but the impact is same right. So, what I can do based on the access the example that I have told you here I just do this. So, I can either merge multiple array into one or I can merge horizontally, I can merge vertically. These are all discussed in previous classes or I can split the array into multiple in a block manner or cyclic manner or complete manner based on the requirement and then, and usually actually reduce the number of array accesses ok.

So, in today's class we have seen what are the construct that is not supported in C to RTL synthesis, then we have discussed the impact of arbitrary precisions and then function calls array initialization and array an interface and how to make them efficient for C to RTL synthesis, ok. So, with this I conclude today’s discussion.

Thank you.