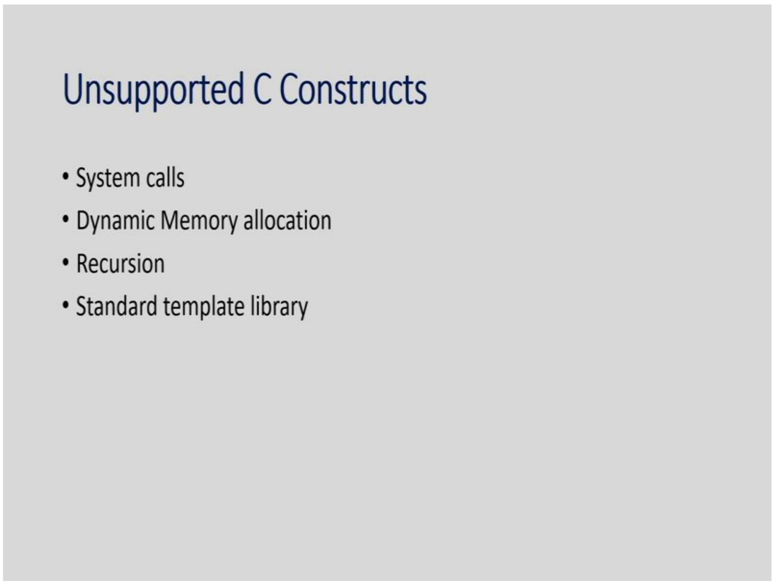
C-Based VLSI Design

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

Module - 08
Hardware Efficient C Coding
Lecture - 27
Hardware Efficient C Coding - Part 2

Welcome everyone, In today's class we are continue our discussion on Hardware Efficient C Coding. So, as we discuss this Hardware Efficient C Coding in we try to mean that we sometime if we write our code without understanding how it will be getting be implemented in hardware we may end up getting a very hardware which is not efficient at all right. So, in this class we try to understand what are the things we should be should consider during writing the C code and then converting into hardware ok.

(Refer Slide Time: 01:27)

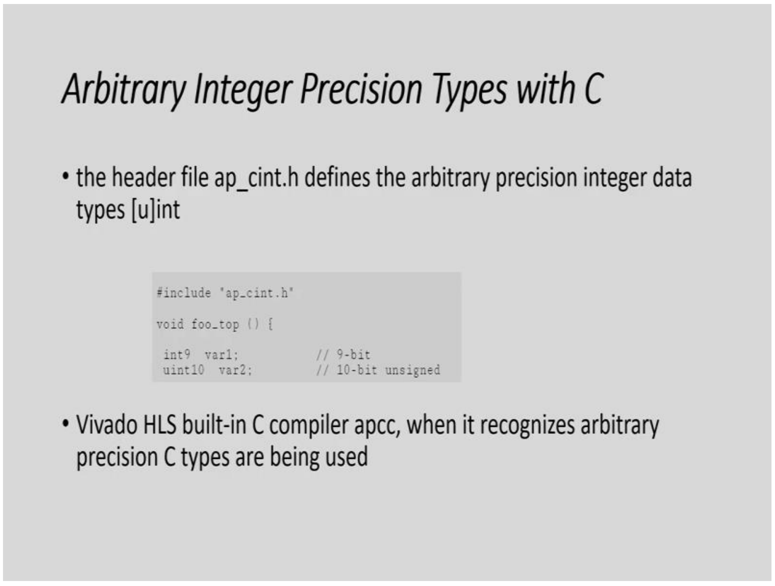


Unsupported C Constructs

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

So, in we have already discussed about this efficient coding for array and loops, specifically for array we have discussed many things specifically the array access is a bottleneck for hardware realization of a C code. And also in the previous class we have discussed various construct which is not getting supported in during hardware synthesis we have also discussed that.

(Refer Slide Time: 01:55)



Arbitrary Integer Precision Types with C

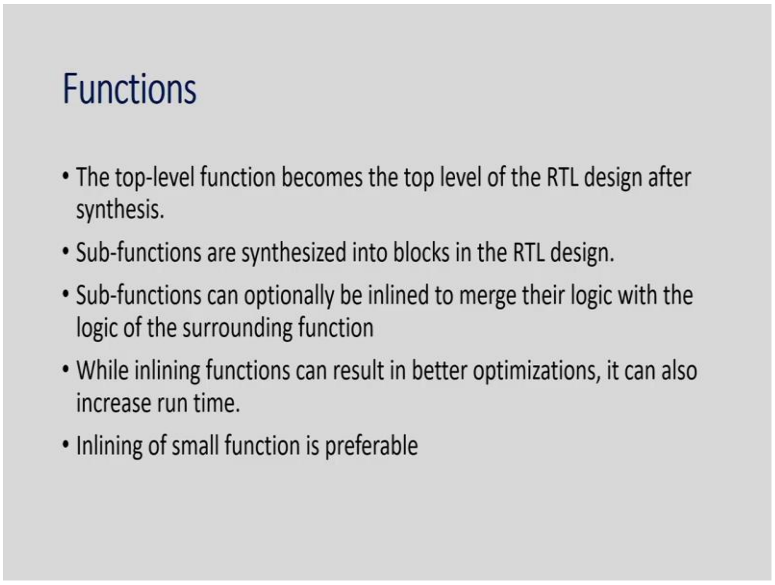
- the header file ap_cint.h defines the arbitrary precision integer data types [u]int

```
#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

And also another important aspect that we have discussed is this precision, that the data width in general our data width is 32 bit 64 in float and all. But in hardware you can actually have any arbitrary precisions and how to make utilize of that because otherwise our this area will be too much right. So, we have also discussed that part.

(Refer Slide Time: 02:15)

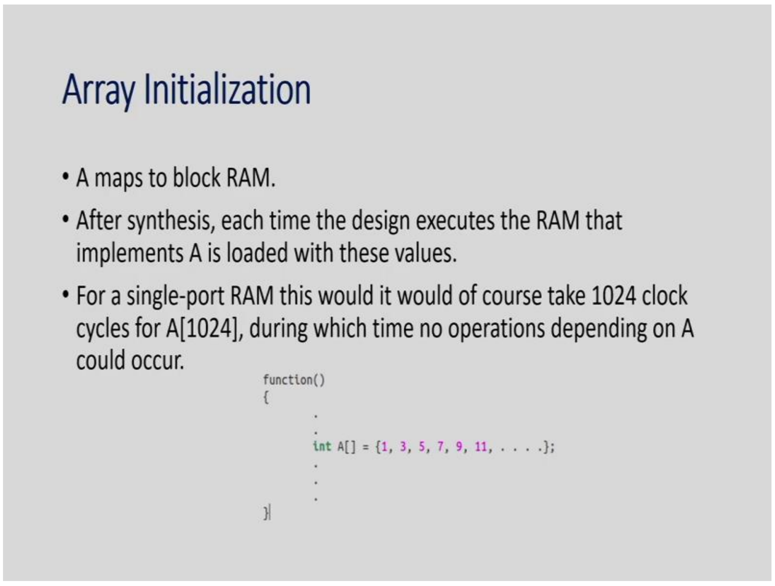


Functions

- 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

We have also discussed about the function how do how the function gets implement in hardware and specifically this hardware how this resource sharing and is done when the when hardware functions create a separate module. So, we have all discuss those things in detail. We also discussed about the array initialization because array initialization is something Create lot of latency if you do not take care of them properly.

(Refer Slide Time: 02:33)

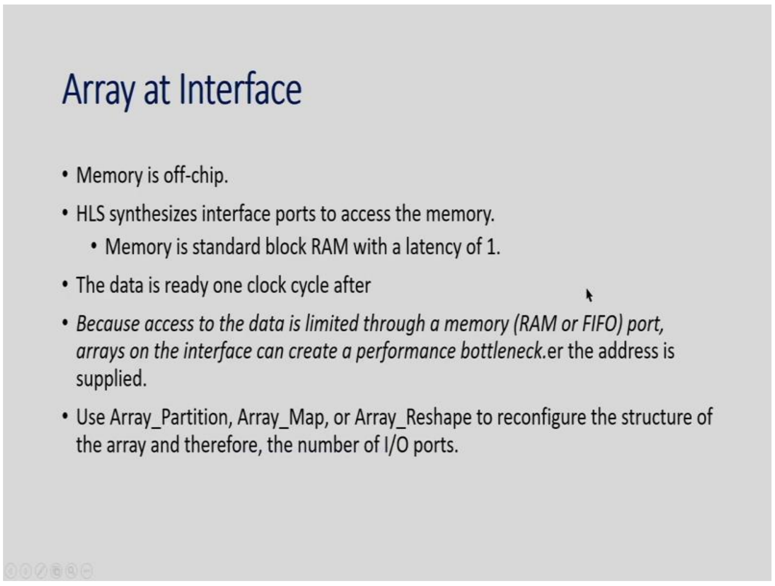


Array Initialization

- 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, . . . .};
        .
        .
        .
}
```

(Refer Slide Time: 02:36)

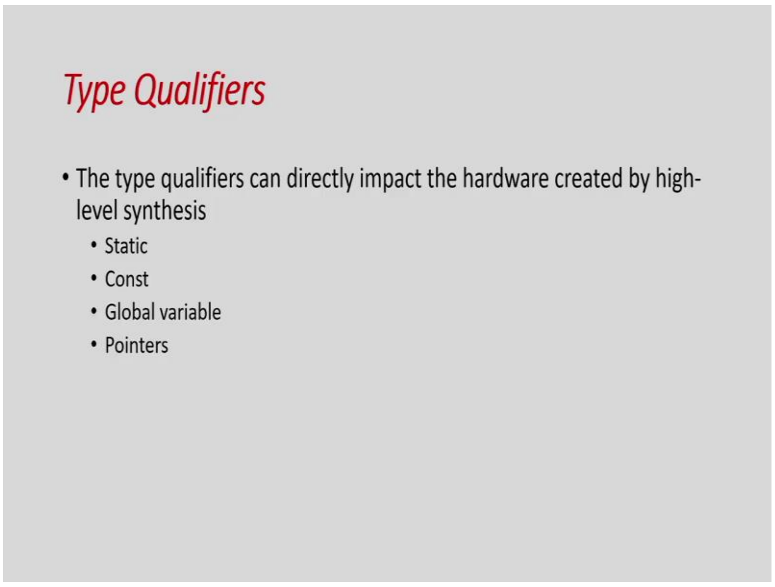


Array at Interface

- 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.

And also we have discussed about this we keep array as a function top level function we just put array as the argument right. So, how it will get reflect in the hardware and how that gets managed. So, those things we have already discussed in detail right.

(Refer Slide Time: 02:53)


Type Qualifiers

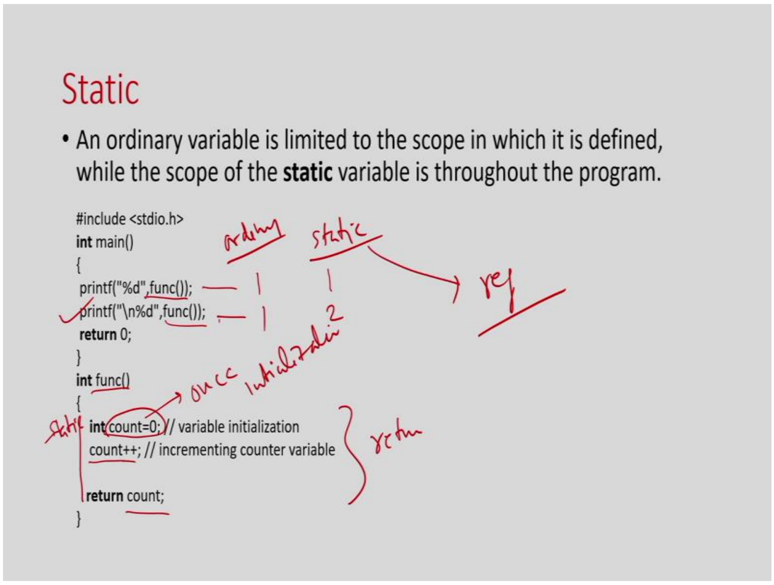
- The type qualifiers can directly impact the hardware created by high-level synthesis
  - Static
  - Const
  - Global variable
  - Pointers

So, in today's class I am going to discuss a primarily two three things; one is basically there are some common construct like static const global variable pointers how they gets reflected in hardware. Also I am going to discuss about another important topic which is this dependency right.

So, the dependency can cause lot of problem in loop pipe planning. So, we are going to discuss with some examples how the dependency is causing problem. And at the end we I am going to discuss about how we can improve the latency by just taking care of some minor thing in the input C code ok.

So, let us start with this type qualify qualifiers right. So, we have used this static, const those things in our programmer at a lot right. So, what about the static? So, you know this static variable which is something if it is defined within a function its scope remain within the function right, but it lifetime remain across the complete program execution program right so; that means, if I visit the function multiple times whatever the value it was there previously it will start with that value right.

(Refer Slide Time: 03:43)



Static

- An ordinary variable is limited to the scope in which it is defined, while the scope of the **static** variable is throughout the program.

```
#include <stdio.h>
int main()
{
 printf("%d",func());
 printf("\n%d",func());
 return 0;
}
int func()
{
  int count=0; // variable initialization
  count++; // incrementing counter variable

 return count;
}
```

ordinary
static
1
1
1
2
ref
once initialization
static
return

So, there is a very nice example here say, suppose I have a function called func where I am just having a int count and then I am just doing count plus plus and returning the count ok. So, what I have done here in the main I just call func right and then I func call func again ok. So, what is going to happen? So, if you call first time your count is 0 then count become 1 and it will return count. So, it will print 1.

You call second time again count is 0 because it is you call second time count 0 again it will become count equal to 1 and return 1. So, it will print 1 right. So, now, if you just put static int if you make it static int then what will happen for the first call you come count its lies to 0 and then become count equal to 1 and return count. So, it will print 1 here.

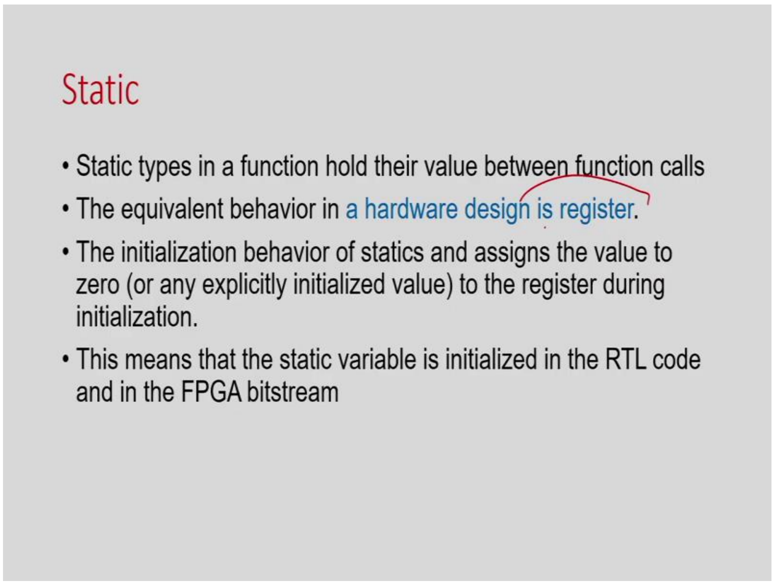
For the next call, it will it would not execute this because it is only initialize once right. So, next time it will remain remember the value of this count that was 1. So, it will do count plus plus equal to 2 and it will return 2 and it will print 2 right, this is for static and this is for ordinary right.

So, this is what is the static and sometime it is important to utilize static in our program because sometime we want to keep this old value remember right. So, now, you think about this normal variable, the ordinary variable versus static variable, how they gets mapped into hardware. So, for the ordinary variable here what I can do? I can just this count may not be stored in some register because I can just do what I just because this count equal to 0. So, once I realize this it will always say return 1 right.

So, this function I can actually do a constant propagation and I can actually return 1 right. So, what is going to happen in the ordinary case, I do not have to store this count in a register at alright, but for static this must be stored in register right. So, you have to store in register because if it is register it will initialize with 0 and then whenever you call it will do that count plus plus right and it will give you the extra value.

And because in hardware all modules are actually running in parallel and they will remain live for throughout the execution of the program. So, whenever you call second time it will actually return you the modified value. So, the basic point that I try to mention here is that for static variable we have to realize this using a hardware and it must be initialized right.

(Refer Slide Time: 06:30)

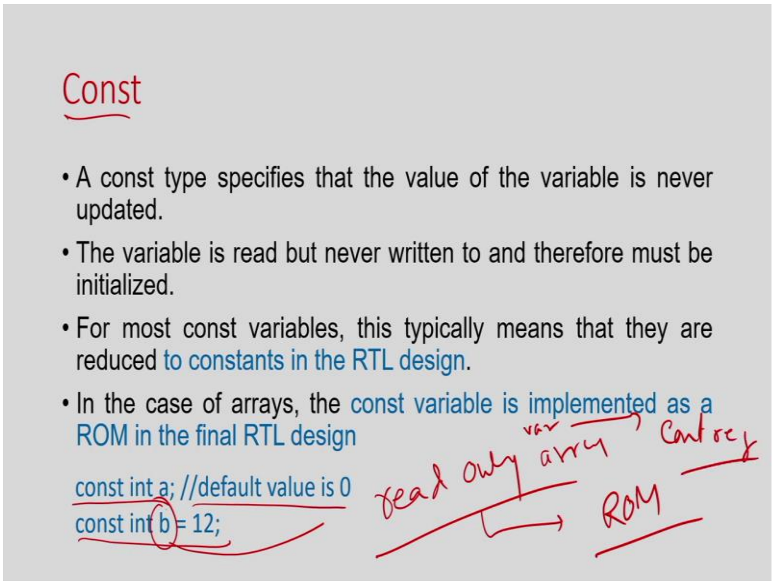


Static

- Static types in a function hold their value between function calls
- The equivalent behavior in a hardware design is register.
- The initialization behavior of statics and assigns the value to zero (or any explicitly initialized value) to the register during initialization.
- This means that the static variable is initialized in the RTL code and in the FPGA bitstream

So, the whatever the initial value it may be 0 or it may be some explicit value instead of 0 you can could have a count to 10 also right. So, you can initialize this 10. So, then next time it will 10, 11, 12, 13 and 14 right. So, basically it has to be explicitly initialized and it must be stored in a hardware in register ok. And for FPGA bitstream also it has to be initialize during the loading of the bitstream. So, this is the simple thing, but you should understand how this ordinary variable versus static variable get realized into hardware.

(Refer Slide Time: 07:10)



Const

- A const type specifies that the value of the variable is never updated.
- The variable is read but never written to and therefore must be initialized.
- For most const variables, this typically means that they are reduced to constants in the RTL design.
- In the case of arrays, the const variable is implemented as a ROM in the final RTL design

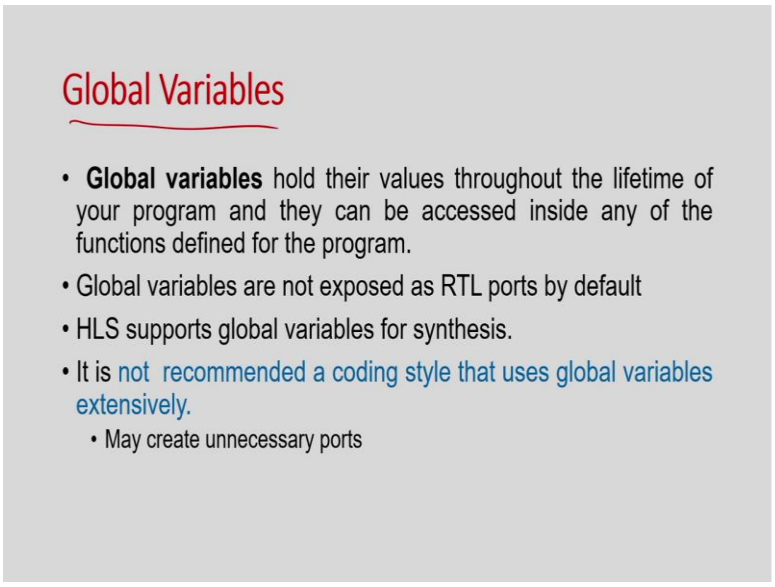
```
const int a; //default value is 0
const int b = 12;
```

So, now there is a const. So, we usually use const int right. So, or const int some value. So, by default if you do not specify anything the default value is 0, and if you specify then this value b value remain 12 for throughout the program right. So, it is basically read only. Const is basically read only.

So, if it is read only, so the basic idea is that if its say array read only array then it will be a ROM right. So, this is if say array it will be mapped to ROM. And if it is a variable read only variable, then it will be mapped to a constant register right. So, you can just think about there is a constant in the RTL which will realize this.

So, this is something very simple, but let I just complete this for sake of completeness ok. So, the conclusion is the const is basically read only aspects, so it will be implement as a constant register or a ROM in the hardware ok.

(Refer Slide Time: 08:06)

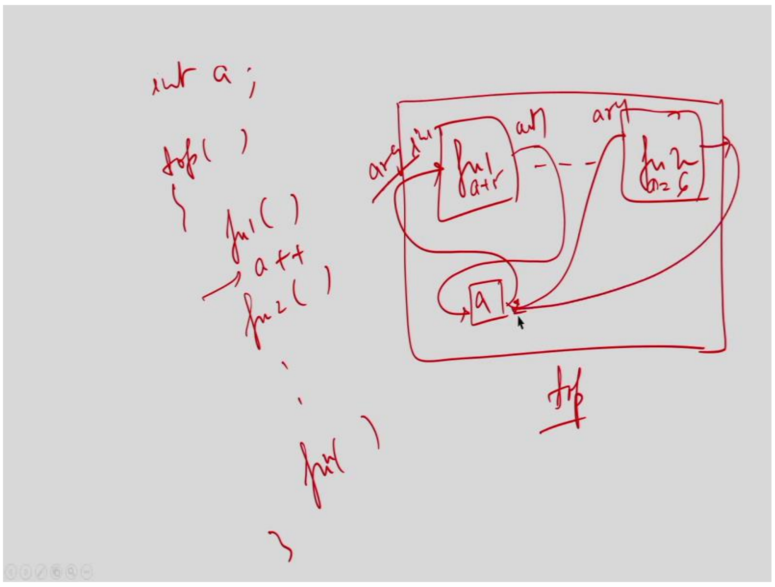


Global Variables

- **Global variables** hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.
- Global variables are not exposed as RTL ports by default
- HLS supports global variables for synthesis.
- It is not recommended a coding style that uses global variables extensively.
  - May create unnecessary ports

So, now next the global variables. So, we all know global variables. So, global variable is something it is basically live throughout the program executions and it can be accessed from any functions right. So, that is the purpose of the global variable.

(Refer Slide Time: 08:32)



So, in generally the how this global variable will get exposed. So, you have to understand that right. So, suppose you have a function where you have a global variable say int a right and now you have a top level function, you are calling say function 1, function 2, say function n right. So, this a is actually accessible everywhere. So, now, you think about this a may be accessed here as well as say a plus plus you can do it right.

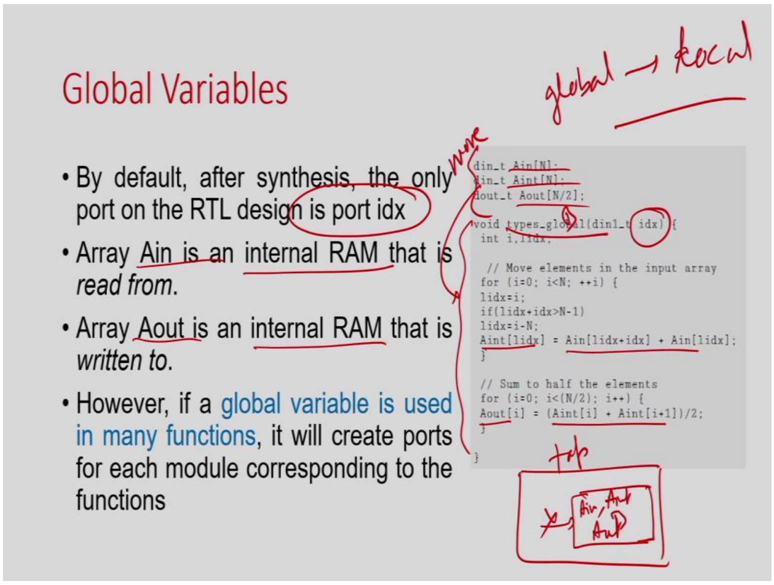
So, now where this a will store in the hardware. So, you remember that once you have a function it will create a module right. So, every function will be realized by a module. So, there is a module for a function 1 to say function n right. And now where this a will be stored? Within the functions on top. If it is get accessed in the top level module, this is your top level module. So, there will be say some register which is storing a right.

Now, you think about that whenever you have this a and if you want to do something here right, so some operation on a plus 5 here. So, you have to pass it here as a argument right. So, it will be an argument. If you want to access this here also you have to create an argument right and now, if this gets value updated there will be argument and it will actually update this also. So, this will get some input port or output port also.

It make here only the one port because if it is only updated there right you are not accessing you are just updating that value. So, suppose you are you are doing a equal to 6 so; that means, you should have a output port here which will update this. So, the bottom line here is that once you have a global variable we in the hardware because it is become a it is accessible to all functions. So, it will create lot of ports right.

So, by default in a at least in Vivado HLS they analyzed right. So, what are what are the functions it is going to use and if it is only local to a function; it is basically moved that global variable to a local to the variable to that function right. So, for example, if you take this example here you have this.

(Refer Slide Time: 10:30)



Global Variables

- By default, after synthesis, the only port on the RTL design is port idx
- Array Ain is an internal RAM that is *read from.*
- Array Aout is an internal RAM that is *written to.*
- However, if a global variable is used in many functions, it will create ports for each module corresponding to the functions

```
din_t Ain[N];
din_t Aint[N];
dout_t Aout[N/2];

void types_global(din1_t idx) {
 int i,lidx;

  // Move elements in the input array
 for (i=0; i<N; ++i) {
 lidx=i;
 if(lidx+idx>N-1)
 lidx=i-N;
 Aint[lidx] = Ain[lidx+idx] + Ain[lidx];
 }

 // Sum to half the elements
 for (i=0; i<(N/2); i++) {
 Aout[i] = (Aint[i] + Aint[i+1])/2;
 }

}
```

There are three global variable here A in, A int and A out and in the this function I mean this is the only function where this is getting used and you see here that I am actually computing A int using A in, then I am computing A out using A int right so, but this is within this function. So, what is going to do it?

So, this is your top level module and this is this module that glow this function. So, this is this module. So, this is the module B and this is your module B. Since it identify them in the whole program these variables this although they are global variable, but they are only accessed within this. So, they will move it here right. So, this global become kind of local right.

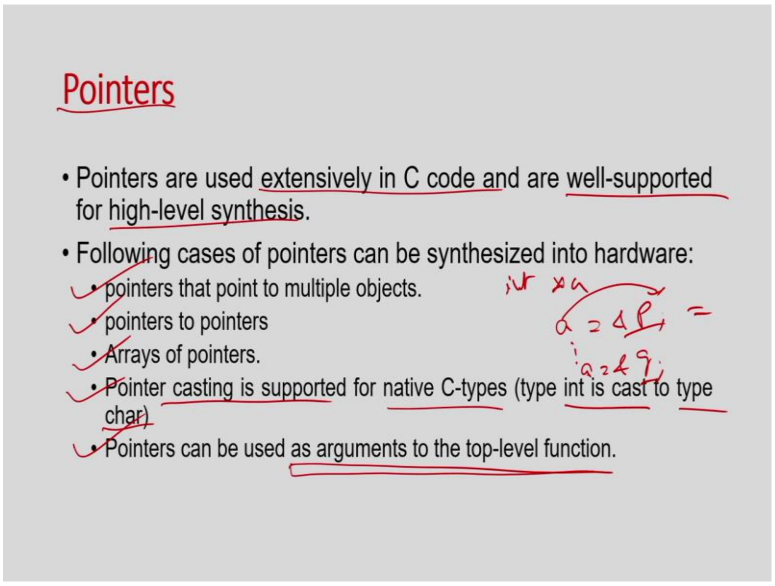
So, if possible it will do it. So, then it would not create any port right. So, no port will be created. So, for this function since it has only one port it will actually create only one port for idx right. So, for this example as I mentioned here it get only one port idx and it will move this A in as a internal ram, A out also internal RAM right. So, in this particular case it would not create anything because now this all this A in, A int and A out all become RAM in that right. So, the point that I try to establish here is that for global variable if I declare a global variable ideally it will create lot of extra ports right.

So, a lot of communication among these functions and lot of ports because it can access right. So, wherever it get access it will create port, but if it is local probably high level synthesis (Refer Time: 12:12) move and make it to a local although you have declared it as a global variable, but whenever there is a array you understand that it is not a single port you have to create three four ports, it is a read address port, data port, data in data out port, as well as the write enable and clock enable.

So, now you think about there are say five such arrays are getting you have declared as global, it will create lot of unnecessary ports to this to this modules corresponding to the functions right. So, in general so, it is not recommended to use global variable right. So, unless and until it is necessary right.

So, because it is creates lot of dependencies among the modules and it might actually hinder some optimizations applying of the tux level pipelining. So, in general this is we should avoid this use of global variables ok.

(Refer Slide Time: 13:01)



Pointers

- Pointers are used extensively in C code and are well-supported for high-level synthesis.
- Following cases of pointers can be synthesized into hardware:
  - pointers that point to multiple objects.
  - pointers to pointers
  - Arrays of pointers.
  - Pointer casting is supported for native C-types (type int is cast to type char)
  - Pointers can be used as arguments to the top-level function.

int *a

a = &p;

a = &q;

So, now I have move to discussion on the pointers. And we know pointers is something very important functionality in C through which I can access the address of a registers or an array or a variables right. So, it is extensively use in C and if and it is actually very well supported in high level synthesis right. So, it is actually getting supported and if you look into pointer what is that it is basically you are accessing a location right.

So, in hardware if you think about you have a array you are accessing this to a pointer which is nothing, but you have a RAM and you are accessing say through some address right. So, supporting a pointer in hardware is basically it is easy right it is just nothing, but having the address right. So, you have to convert those pointer into corresponding address in the hardware right. So, in general this high level synthesis is to support all type of pointers right.

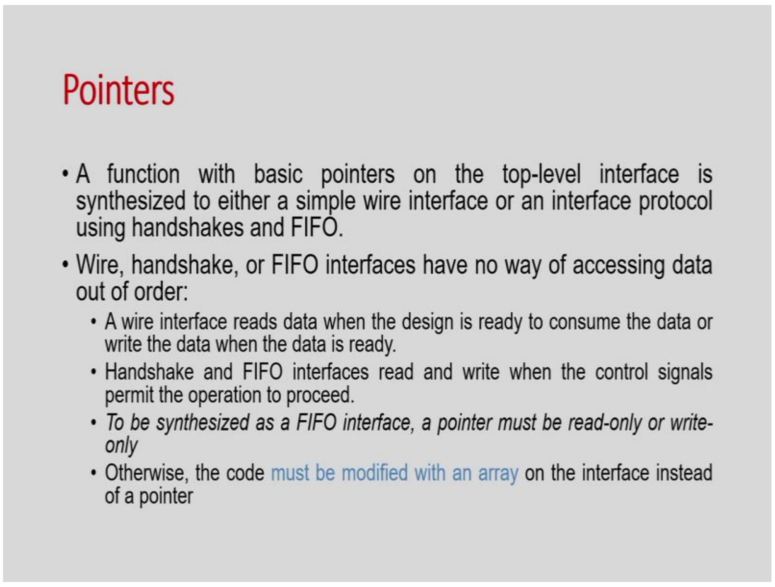
So, pointer that actually pointing to multiple objects right. So, suppose you have int star a and then say a storing the address of p after some time a storing address of q. So, this type of things are supporting because it is nothing, but this is a register probably and it will its nothing, but a is basically it become this register right. So, it is basically pointing to that register and here it pointing to another register q right.

So, pointer through that point multiple objects is getting supported pointer to pointer is basically indirect addressing right. So, you have this pointer you point to some location and that actually stored the actual address. So, it is just the indirect addressing and its very easy to support that in hardware right. So, that is also supported. Array of pointers also is supported in high level synthesis because it is nothing, but a set of addresses right. So, it is not difficult to support that.

Even pointer casting is also supported for native C types right. So, for example, you have a integer pointer you can you cast it to point to a character type right. So, even that things are also supported because it is just again in the hardware if you think about the casting is nothing, but you just mentioning that this variable can store the address of this register instead of this register right.

So, that way it is not a problem pointer casting is supported. Even pointer is supported as the argument of top level function ok. So, basically if you think about you have a top level function and you try to pass a address of a point at a pointer right. So, in integer pointer that is actually pointing to some array even that is supported, but here we have certain restrictions ok I am going to discuss this in bit detail.

(Refer Slide Time: 15:43)

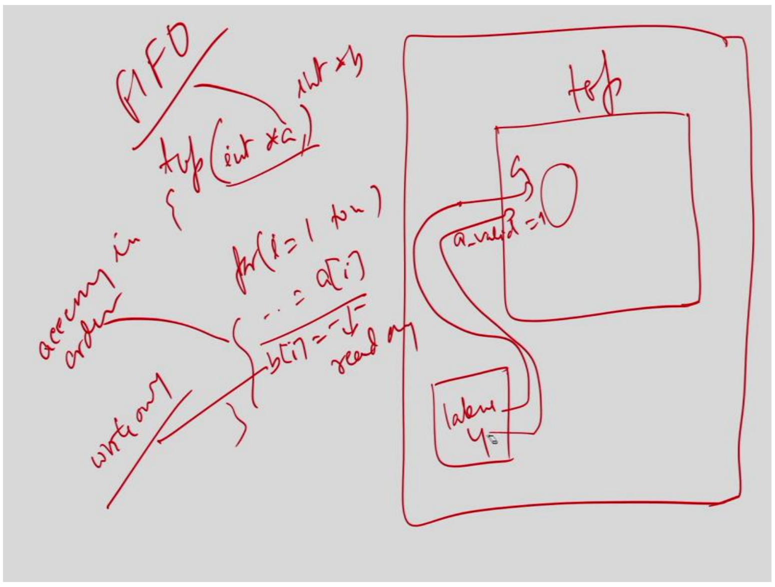


Pointers

- A function with basic pointers on the top-level interface is synthesized to either a simple wire interface or an interface protocol using handshakes and FIFO.
- Wire, handshake, or FIFO interfaces have no way of accessing data out of order:
  - A wire interface reads data when the design is ready to consume the data or write the data when the data is ready.
  - Handshake and FIFO interfaces read and write when the control signals permit the operation to proceed.
  - *To be synthesized as a FIFO interface, a pointer must be read-only or write-only*
  - Otherwise, the code must be modified with an array on the interface instead of a pointer

So, how this point this argument at the top level get implemented in hardware, right. So, you now think about say suppose you have a top level function where you have a star a right.

(Refer Slide Time: 15:46)



So, you have a int star a. So, now, in this particular program so, this is basically you can access this right. So, you can actually have some for loop you 1 to n you try to access this a i right you want to access this. So, what is going to happen once you have this int pointer at the top level function, what it creates? It is basically create i 3 types of connections right.

So, it basically either create a wire or it will actually create some handshake or a FIFO right. So, wire means basically if it is a wire at the interface, it means that you are going to read the data only when the data is ready to consume or a data when the data is ready. So, basically you can have a handshaking signal right. So, you basically have you get a data valid in the next clock you are going to get the whatever the value is coming it is actually get read.

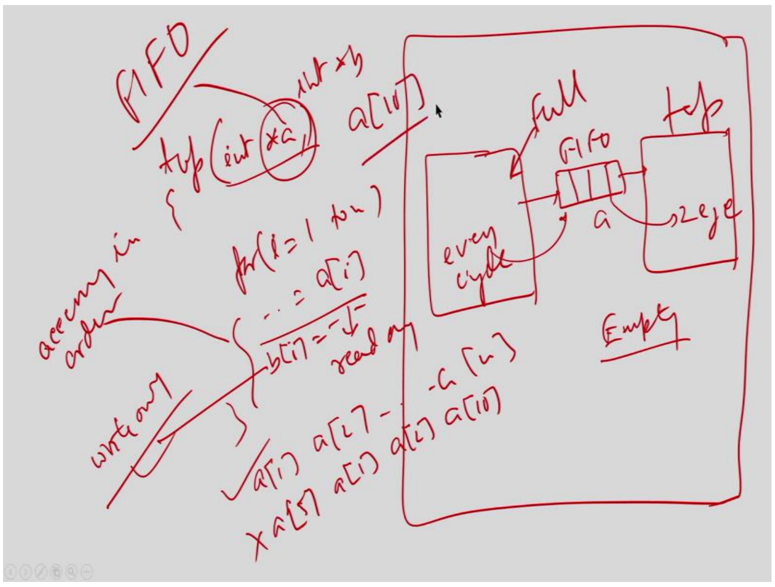
Or you before computing this you send the signal that this is the valid data and then you send the data and it is actually so, in the same clock. So, you send this data valid and the data. So, now, the receiver process understand that this particular data that is coming is now valid data right. So, you have to understand that whenever you have written a function, so this will be implemented your top and it will be integrated with so many other modules in a big bigger systems right.

So, suppose this is taking a from this function. So, this might may not produce this a in every cycle right. So, there may be some latency for this say latency is 4 so; that means, every fourth cycle is getting a is actually generating a valid data right. So, once it is generating a valid data it has to told that; so, this is a valid right. So that means, so, whenever this a valid is equal to 1 then only whatever data id is here that it is going to access here. So, this is a wire connection, but that is something handshaking is happening right.

And now if this so, if it is a array i i it is only read access right. So, this is only read only right or you can have another say access say int star b, where you are just doing this b i equal to something. So, you are actually writing something to b i right, so that means, this is the write only and you can understand that they are actually accessing in order.

So, both are accessing in order, accessing in order, order in the sense you can understand that I am not accessing randomly to this location; that means, I am accessing 1 2 3 a 1, a 2, a 3, a 4 and so on. Similarly, I am accessing b 1, b 2, b 3, b 4. So, this is in accessing in order. In that case if its say only read only or write only, I can realize this kind of connection as a FIFO right, first in first out kind of connection. So, instead of this instead of this handshake what I am going to do it now? I can actually do this right.

(Refer Slide Time: 18:58)



So, instead of doing this handshaking that valid and all between these two module what I am going to do? I am actually having a module here which is my top and say it is actually getting this a from here right. So, what I am going to do it is instead of making this connection reject I am going to put a FIFO here, this is FIFO first in first out. So, the in the whatever the order d is going to written and the same order I am going to read it right. So, now, this is my a. So, now, what is going to happen before? Because of this now this become asynchronous right.

This two model earlier the diagram that I have drawn its basically synchronous that whenever this data valid signal comes it has to consume the data right but here this particular module can actually generate in any different clock the data into this FIFO and this guy can read it in its own space right.

So, it may produce in every cycle it might read it in every two cycle right. So, every cycle. So, this is possible because you do not have to have a make a explicit handshake between this access. So, in such kind of cases whenever you have these [FL] this the high level synthesis tool analyze whether this access is in order because so, unless this read and write is in the same order you cannot do it right.

So, its can you can only realize FIFO if it is only read only or write only right. So, if you read and write both from the same FIFO you cannot realize, I mean from the same array you cannot realize that particular array as a FIFO because then because FIFO you understand first in first out. So, the order it is going to be data is written this model is going to read it in the same order.

So, in the sense that if you do not have this order same order things are there you cannot do it and also this guy only can write it read it from here and this can only read. So, this if this particular array is both write and read from the same module this would not be implemented as a FIFO ok. So, the point here is that once you have this access. So, based the tool high level synthesis tool analyze whether it is a read only or write only right.

And it is the access random. So, are you accessing like a 1, a 2 to a n or your accessing a 5, a 1 right say this is random access, a 1 then say a 2 then say a 10. So, this is the random order right. So, if it is this order it is only write only or read only then it will automatically be converted into a FIFO. And FIFO has a biggest advantage that now this two module actually can run asynchronously. And how they synchronize effectively by the blocking read property of this right.

So, if this is empty, if this FIFO is empty; that means, there is no data this process has to wait until some data comes right. So, that is how the synchronization happens. And similarly if this is full, if the FIFO is full this process has to wait right. So, until some data is consumed by the consumer process.

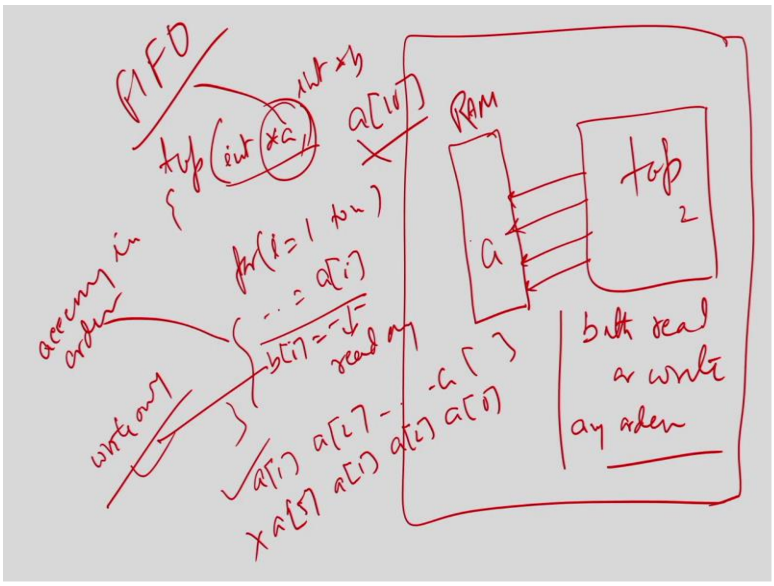
So, basically in such case whenever you have this access is read only or write only and it is accessing is in order it is not a random access, it is basically converted into a FIFO and this both producer and consumer can actually run asynchronously and they are actually getting synchronized by the blocking read property or on and for wait on full right.

So, this is how this can be done. Now the point that I try to make here is that once you have declared a pointer high level synthesis tool either try to implement by this wire, wire is basically I just make the handshake the data valid things or FIFO right. So, if it is not if it is a random access an all it would not able to implement that right. So, that pointer the in the point I try to make it here. So, to, so, if it is handshake and few read and write should be in the same order.

So, what I am trying to say is that if it is not it is not following that, it would not be able to implement that FIFO or this handshake or wire right. So, in that case what you have to do instead of this pointer you should use a explicit array right. So, if you do a explicit array what the high level synthesis tool assume. So, what I am trying to mean is here is that here instead of pointer you use a 10 right.

So, now the tool will understand that this is an array which is basically be an RAM, external RAM. So, the tool will assume that there is a RAM is present in the s o c which is not part of this top module, but it is a RAM right. So, what is going to access? Now instead of doing this; so, whenever you put is as a array right.

(Refer Slide Time: 23:57)



So, what is going to do is this, that say suppose you have this top module and you have a array right this is your top module it will assume if you write this as a array like this. So, it will assume there is a RAM here a and this is a RAM and now we can access randomly right. So, we can both read and write both read or write, both is possible any order right. So, both is possible because it just creates some address right.

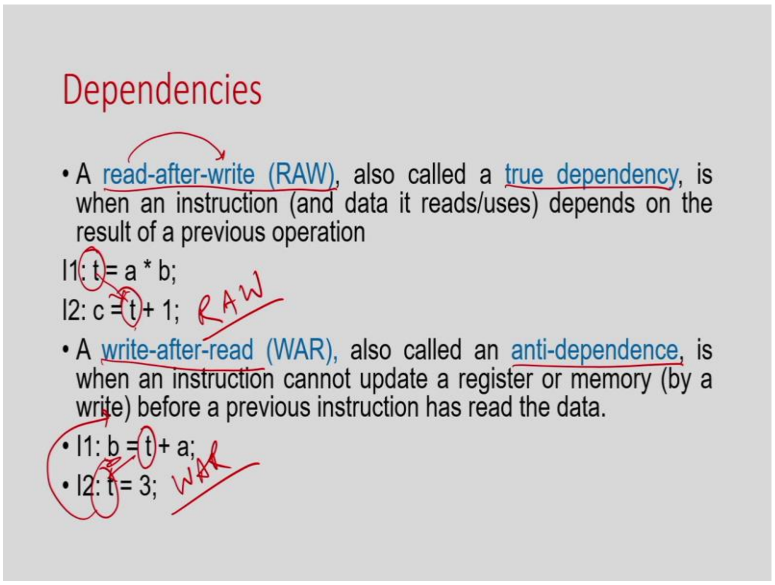
So, it will create address, it will create data, it will create the write enable, it will create the read enable and all those things and then it will just give some address if you want to write it will just make the write enable or if you want to read it will just make sure that it is not write enable write enable right.

So, the point is that if you use pointer at the argument it must be should be implemented by wire handshake or FIFO mode if it is not you should implement that instead of pointer you must use an array. So, that this can be realized as a RAM external RAM in the during high level synthesis ok. So, that is all about the pointer, I will now move onto the dependencies.

So, data dependency is something we all knows, there are and we have already discussed during scheduling that whenever you schedule an operations you have to make sure that the data dependency is not violated ok. So, that is already discussed and that is already taken care by high level synthesis the scheduling part ok. So, what I am going to talk about is there what kind of dependencies are there among these operations and how the dependency can actually impact on the loop pipe lining.

Because we have already discussed and convinced ourself that this loop pipe lining is one of the most important features of high level synthesis to realize efficient implementation of the loops in hardware ok. So, let us understand what are the dependencies are there. So, there are four type of dependencies.

(Refer Slide Time: 25:57)



Dependencies

- A read-after-write (RAW), also called a true dependency, is when an instruction (and data it reads/uses) depends on the result of a previous operation

I1: t = a * b;
I2: c = t + 1;

- A write-after-read (WAR), also called an anti-dependence, is when an instruction cannot update a register or memory (by a write) before a previous instruction has read the data.

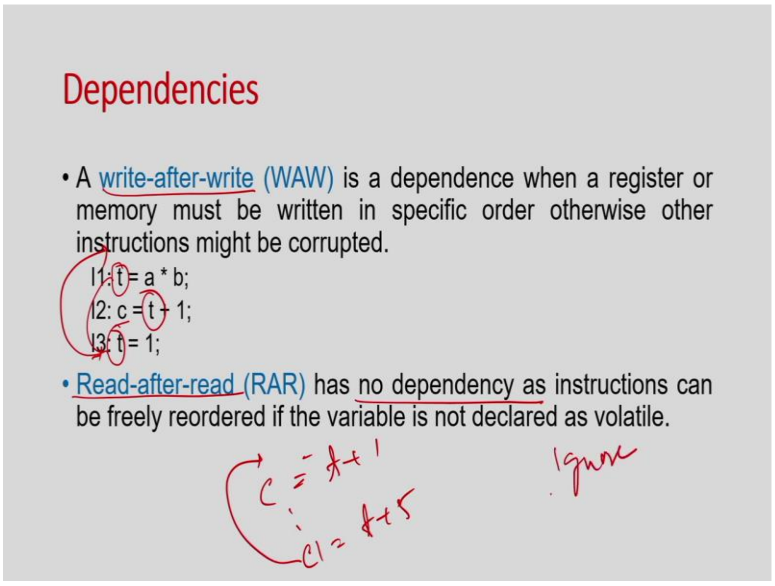
- I1: b = t + a;
- I2: t = 3;

What is called RAW is a read after write as and this is called true dependencies what is that. So, from the name you can understand that you are reading after writing something right. So, you are writing something in this value t and then you are reading here right so, that means, you are read after write right.

This is a true dependency and this is what we have actually talked about in scheduling time also right, this is what is the data dependencies ok so; that means, this t has to be execute completion of the state t has to be there then only this operations l I 2 can execute right.

So, now we have a write after read ok. This is called anti dependencies. So, you are reading something you are reading t here and then you are writing here right, this is write after read. So, this is also if dependency that has to be supported because otherwise if we just move this operation here this operation b will take a wrong value right. So, the function it will change. So, we have to obey this right after read dependencies as well. So, there are two more type of dependencies.

(Refer Slide Time: 27:09)



Dependencies

- A write-after-write (WAW) is a dependence when a register or memory must be written in specific order otherwise other instructions might be corrupted.

  I1: t = a * b;
  I2: c = t + 1;
  I3: t = 1;

- Read-after-read (RAR) has no dependency as instructions can be freely reordered if the variable is not declared as volatile.

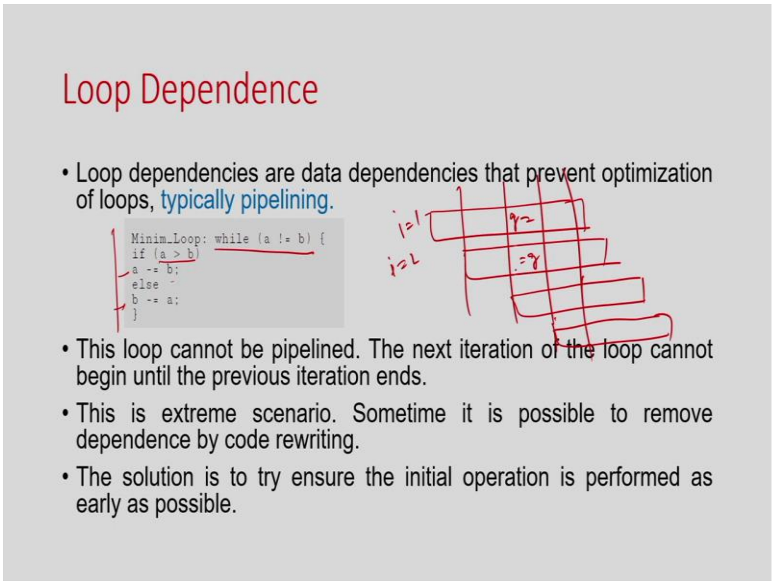
c = t+1
c1 = t+5
Ignore

The next one is write after write, so that means, you are writing to t and then you are writing again right. So, you cannot again change the order. This is also the dependency that has to be supported or it has to be honored during the hardware implementation otherwise if you just move this operation here the order will change and this C will get a wrong value right.

So, that means, we have to obey this write after write dependencies as well. And there is one more is read after read so, that means, you are reading something right. So, you are reading say t say t plus 1 and then again you are reading C 1 equal to t plus 5. So, here it does not matter whether I read this first or this first. So, in this case I do not have to because since the value of the t is remain same. So, in which order I update C 1 C 1 C 1 does not matter right. So, this is not something in no dependency.

So, we can actually ignore this one, but this three is something important and has to be obeyed right during hardware implementation. So, now, what I am going to discuss is we understand this kind of dependencies and this how this dependencies is actually creating impact on this loop pipelining ok.

(Refer Slide Time: 28:33)



Loop Dependence

- Loop dependencies are data dependencies that prevent optimization of loops, typically pipelining.

```
Minim_Loop: while (a != b) {
if (a > b)
a -= b;
else
b -= a;
}
```

- This loop cannot be pipelined. The next iteration of the loop cannot begin until the previous iteration ends.
- This is extreme scenario. Sometime it is possible to remove dependence by code rewriting.
- The solution is to try ensure the initial operation is performed as early as possible.

So, let us take an simplest example that. Suppose I have a loop here. And you have if a greater than b then I am updating a equal to a minus b or I am updating b equal to b minus a right say suppose this is what is happening and this is within a loop while right. So, now I want to if I want to do the pipelining.

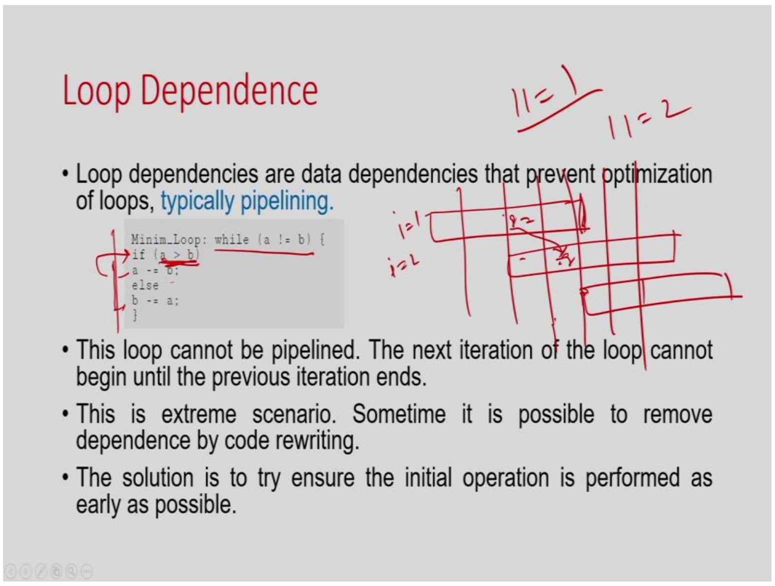
So, you have to remember that in the loop pipelining what we do we have one iteration of the loop and then I want to I do not want to wait till the completion of the that particular iteration and I want to start the next iteration early before completion right. So, that means, if this is the time taken by the iteration 1 right, I do not want to start my i equal to from here right this is the normal non pipeline version execution right.

So, if i equal to 2, this is the non pipeline version of the loop implementation, but in the pipeline mode, what I want to do? I want to start these things early right. I want to start say this is 1 the clocks right this is the clock 1 and clock. So, I want to start the next iteration immediately after first clock so; that means, what is going to happen is that the two iterations or multiple iteration rather is going to run in parallel right.

So, multiple iterations running at the same time, but on different set of data. So, which is good for earlier I mean ideal case. Now, you think about the i plus 1 iteration depends on ith iteration right. So, that whatever the data i need in i plus 1 iteration which depends on the data of ith iteration.

So, which is called its very known is the data hazard right in the pipeline in the normal processor level pipelining as well. So; that means, for example, say I am calculating q here and that is getting used here right. So, q is getting used here so, that means, I cannot do these things in parallel. So, I have to shift this loop at least by 1 because this is going to be it is defined in say this clock.

(Refer Slide Time: 30:41)



Loop Dependence

- Loop dependencies are data dependencies that prevent optimization of loops, typically pipelining.

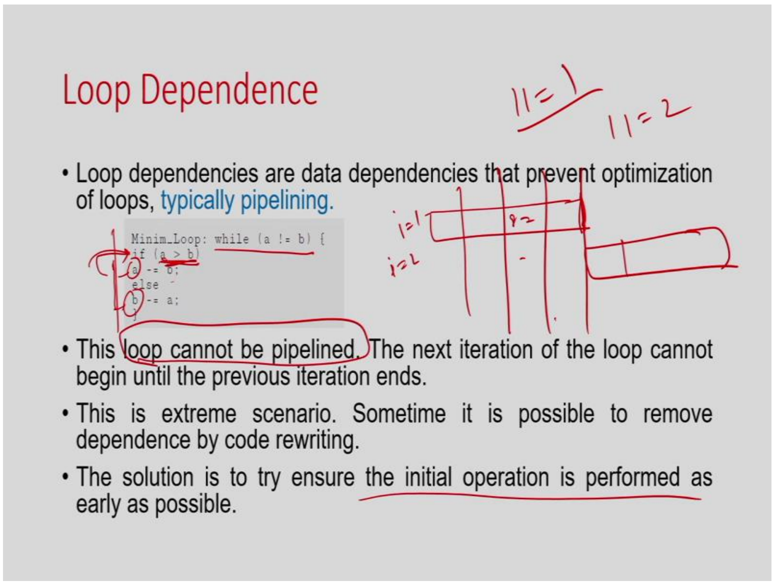
```
Minim_Loop: while (a != b) {
if (a > b)
a -= b;
else
b -= a;
}
```

- This loop cannot be pipelined. The next iteration of the loop cannot begin until the previous iteration ends.
- This is extreme scenario. Sometime it is possible to remove dependence by code rewriting.
- The solution is to try ensure the initial operation is performed as early as possible.

So, I have to move these things one bit earlier and so, that I can actually use q here right. So, you understand that. So, this is basically the dependency across iterations right. So, now, this actually so, earlier the diagram that I have drawn there initiation interval was 1 now the initiation interval become 2; that means, I have I cannot start my next iteration immediately in the next clock rather I have to start this things every second clock right.

So, I can start the third iteration here and so on. So, if there is such dependencies right say across loop iterations. And this particular example is the perfect example here because say suppose I want to pipeline this loop now and in whatever I am calculating in the ith iteration a and b, it is getting use the first time in the checking right. So, that means, I cannot do anything until I check this condition right and this value a and b will be getting at least end of this first iteration.

(Refer Slide Time: 31:53)



Loop Dependence

- Loop dependencies are data dependencies that prevent optimization of loops, typically pipelining.

```
Minim_Loop: while (a != b) {
if (a > b)
a -= b;
else
b -= a;
}
```

- This loop cannot be pipelined. The next iteration of the loop cannot begin until the previous iteration ends.
- This is extreme scenario. Sometime it is possible to remove dependence by code rewriting.
- The solution is to try ensure the initial operation is performed as early as possible.

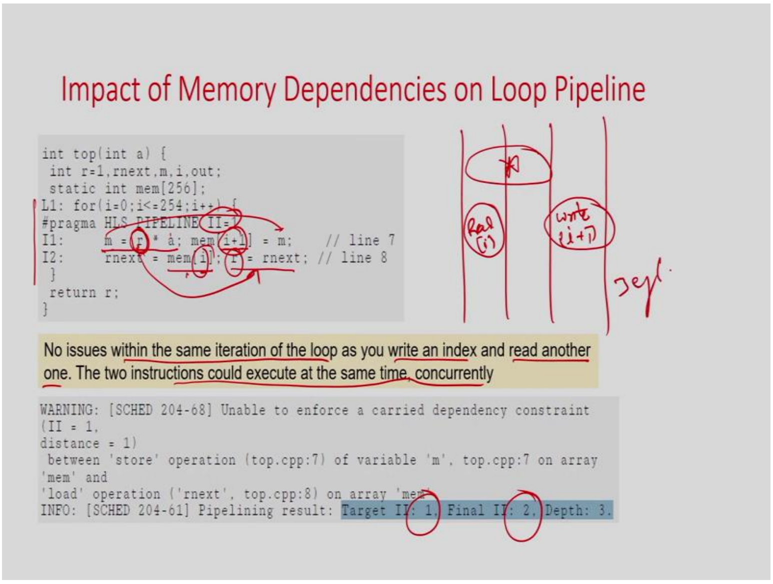
So, it means that I cannot start the next iteration in overlap with this. I can only start it at the end of the first iteration right. So, that means, I can only start in this particular example, that I can only start the next iterations once this a and b value is available then only I can check this. So, this is my i equal to 2. So, this means it is actually this loop cannot be pipelined at all right because here because of this loop carried dependencies I cannot pipeline at all the loop right.

So, because it the second iteration has to wait the completion of the first iteration. So, that is what I try to mean that this once we have the program we may not understand this, but I just apply loop pipeline right and if you apply pipeline to this loop say a pipeline equal to i i equal to 1, just simply compiler will high level synthesis. So, that it cannot be pipeline I just have a non pipeline implementation ok.

So, obviously, in this case we cannot do anything because it is very this is how the program is (Refer Time: 32:44), but sometime probably some tweaking is possible in the code. So, that this kind of dependency can be avoided or shortened right. So, the idea is that you try to ensure that initial operation is performed as early as possible. So, that the dependency is kind of removed ok.

So, this is what I just try to give the impact of this loop carried dependencies on the loop pipelining ok.

(Refer Slide Time: 33:12)



Impact of Memory Dependencies on Loop Pipeline

```
int top(int a) {
 int r=1,rnext,m,i,out;
 static int mem[256];
L1: for(i=0;i<=254;i++) {
#pragma HLS PIPELINE II=1
I1:     m = r * a; mem[i+1] = m;    // line 7
I2:     rnext = mem[i]; r = rnext; // line 8
 }
 return r;
}
```

No issues within the same iteration of the loop as you write an index and read another one. The two instructions could execute at the same time, concurrently

```
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint
(II = 1,
distance = 1)
 between 'store' operation (top.cpp:7) of variable 'm', top.cpp:7 on array
'mem' and
'load' operation ('rnext', top.cpp:8) on array 'mem'
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

So, I will take two more examples just to highlight this is the same fact. So, here I have taken an example which is something if you see here I want to have a loop here right, and in the loop what I am doing I am just doing I am reading m mean I am calculating m then I am writing that value of m to a memory i plus 1 and I am also reading memory i to some value r ok.

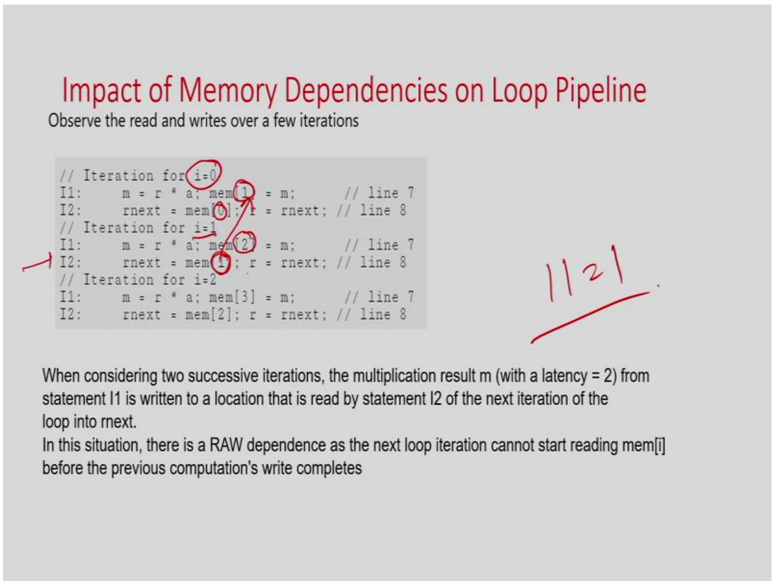
So, you may think about this scheduling. So, I can and say let us say this multiplier is a two cycle operation. So, you can actually do this multiplication in two cycle, at the same time because you want to write in i plus one location reading from i. So, you can actually do the read operation here right, read i I am writing simply. And I can only do this write operation write i plus 1 in once this multiplication is completed right.

Because that particular value is getting written in i plus 1 location. But this i and i plus 1 is something independent. So, this read can happen in parallel with this operations. This is this is the schedule I am going to get right. So, it is basically three state and I can pipeline it. If I now try to pipeline with my pipeline i i equal to 1, if i try to apply my high level synthesis tool you say that this i i cannot be possible 1 initiation interval it i a am going to get 2 ok.

So, why? Because you if you just try to understand this here. So, within the loop there is no problem right. So, there is no issue in the same iteration of the loop because the read index, write index and read index is completely different right. So, I can actually execute this at the same time concurrently whatever exactly I want to mean here. So, you can put this read here as well both is possible, but I just a try to schedule as early as possible.

So, there is no problem as such within one iteration. Now think about the loop carried dependencies ok. So, let us try to understand that.

(Refer Slide Time: 35:13)



Impact of Memory Dependencies on Loop Pipeline

Observe the read and writes over a few iterations

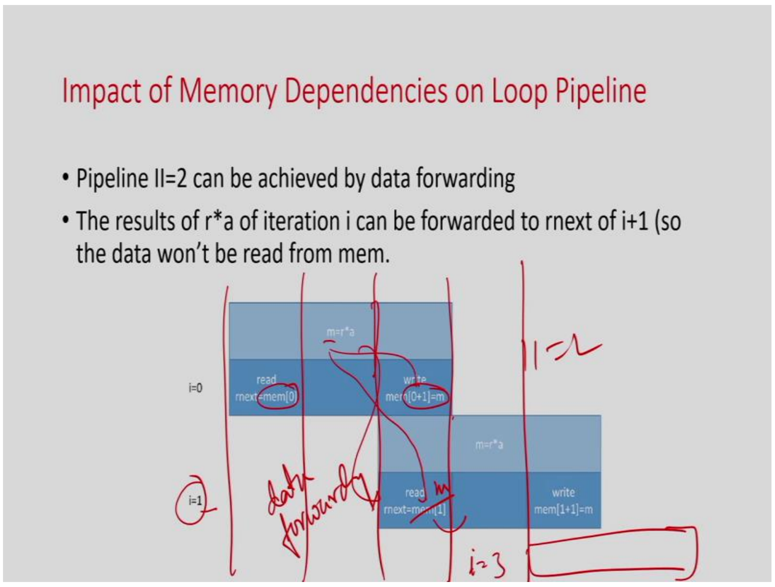
```
// Iteration for i=0
I1:     m = r * a; mem[1] = m;      // line 7
I2:     rnext = mem[0]; r = rnext; // line 8
// Iteration for i=1
I1:     m = r * a; mem[2] = m;      // line 7
I2:     rnext = mem[1]; r = rnext; // line 8
// Iteration for i=2
I1:     m = r * a; mem[3] = m;      // line 7
I2:     rnext = mem[2]; r = rnext; // line 8
```

When considering two successive iterations, the multiplication result m (with a latency = 2) from statement I1 is written to a location that is read by statement I2 of the next iteration of the loop into rnext.
In this situation, there is a RAW dependence as the next loop iteration cannot start reading mem[i] before the previous computation's write completes

So, if you just think about my iteration i equal to 0, I am actually writing to location memory 1 and reading from memory 0 which is fine there are two different locations. But in i equal to 1, I am writing to memory location 2 and reading from 1, now what is this 1? Which is getting written in the previous iteration right.

So, that means, whatever the data I am going to write in iteration 0, I am going to read it in iteration 1. So, what does it mean? I have to wait for till that writing is done right so, that means, unless this data is written I cannot start the next iteration because that r is also so, that is this is going to happen right. So, that is where the loop carried dependency is causing problem and this pipelining is getting hindered right.

(Refer Slide Time: 36:01)



Impact of Memory Dependencies on Loop Pipeline

- Pipeline II=2 can be achieved by data forwarding
- The results of r*a of iteration i can be forwarded to rnext of i+1 (so the data won't be read from mem.

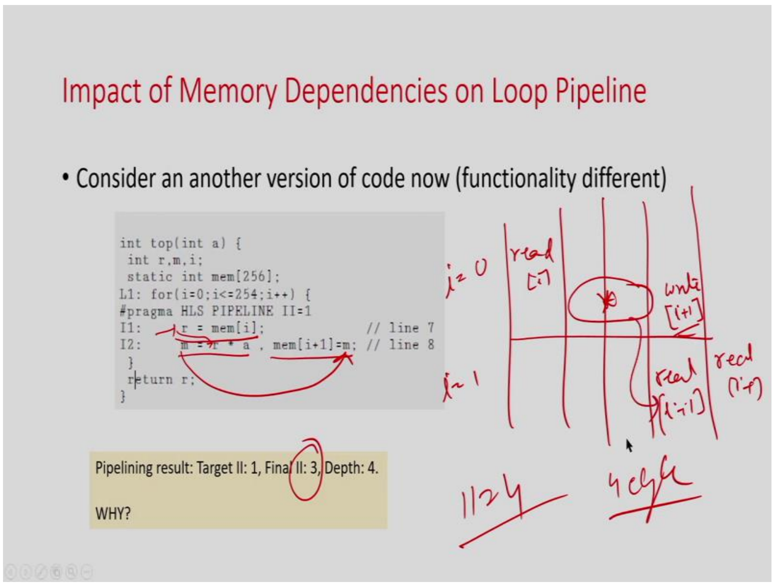
So, if you Understand this, so, basically this is your the multiplication operation is happening and you are doing this read equal to memory 0 and you are writing to memory 1, in this is for i 0 and I can only start this i equal to 1 here right. So, and you can ask me how it is possible because I am writing to memory 1 and reading it here, this actually possible by data forwarding right this is called data forwarding.

So, this is very well known technique for loop by improving the reducing the data hazard. So, ideally this has to delayed by one more and there is no pipelining is possible right. So, you have to do this from here only right. So, this read should come here and this also move by 1, but because this data forwarding is possible. So, I can write this m to this memory location 1 and I can directly assign this 2 r next right.

So, I can directly write m to this location. Instead of this I just write m here, this is what the data forwarding. And with this I can actually you can understand that. So, this is my clock 1, this is clock 2 and this is my i equal to 2 because I am starting the i equal to 2 iteration after two cycle right. So, this every two cycle I can start right. So, I can start the next iteration from here, this is for i equal to 3.

So, what I am trying to mean is that that because of this loop carried dependencies across two loops it actually hindered the pipelining and I cannot achieve my initialization level 1 for this case, but with data forwarding I can actually achieve I initialization interval level to 2.

(Refer Slide Time: 37:46)



Impact of Memory Dependencies on Loop Pipeline

- Consider an another version of code now (functionality different)

```
int top(int a) {
 int r,m,i;
 static int mem[256];
L1: for(i=0;i<=254;i++) {
#pragma HLS PIPELINE II=1
I1:      r = mem[i];                    // line 7
I2:      m = r * a , mem[i+1]=m;  // line 8
 }
 return r;
}
```

Pipelining result: Target II: 1, Final II: 3, Depth: 4.

WHY?

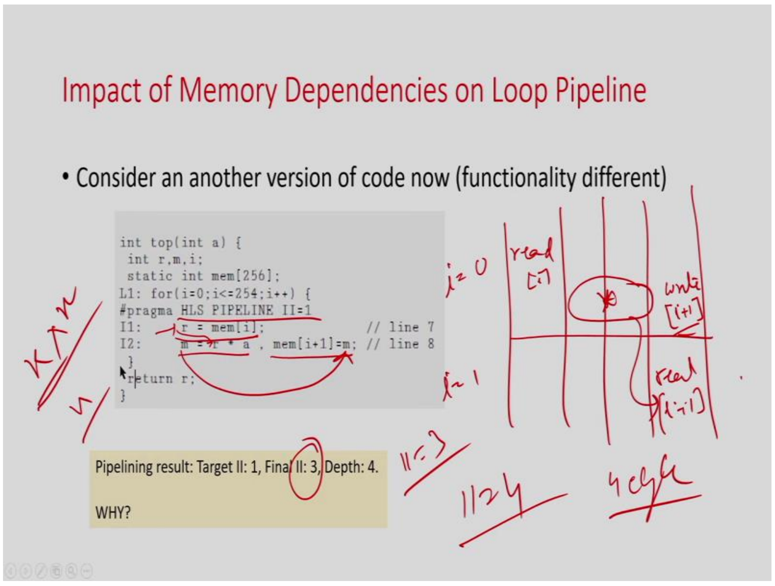
Now, let me take the some tweaking version of this example which is a complete different functionality. So, what I am doing here is I am going to read the memory first right. So, read I then I am going to calculate this and then I am going to do this. So, if you remember here, in this example whatever this r is getting used that is actually read in the previous cycle right.

So, that means, I can actually there is no dependency among this within the loop, this r and this are there is no dependencies right, but here since I am reading it fast now I am reading this fast and using it. So, there is a dependency comes here. So, how it will look like? So, now, I have to do this read i ith location then only I can start the multiplication. I can start the multiplication only after that. And once the multiplication done then this is going to return here; so, then only I can do this write to i plus 1 location right.

So, what does it mean? It will take 4 cycle. So, earlier example whatever I have shown, I have shown that I can actually do it in 3 cycle right because now this dependencies come I cannot. So, this one iteration of the loop will take 4 cycle and now if you think about this that this whatever this i plus 1 written here that is going to written in the this is for i equal to 0 right, and then for i equal to 1 whatever writing is happening that is going to read here. So, that I am going to start from here right i plus 1.

So, that means, in general this loop is cannot be pipelined because there is a complete this my i i equal to we call 4 which means basically it is a iterative execution right there is no pipelining at all right. But only thing is that by just kind of data forwarding I can actually read this from here as well right because I can just forward this data to this place as well.

(Refer Slide Time: 39:52)



Impact of Memory Dependencies on Loop Pipeline

- Consider an another version of code now (functionality different)

```
int top(int a) {
 int r,m,i;
 static int mem[256];
L1: for(i=0;i<=254;i++) {
#pragma HLS PIPELINE II=1
I1:      r = mem[i];                    // line 7
I2:      m = r * a , mem[i+1]=m;  // line 8
 }
 return r;
}
```

Pipelining result: Target II: 1, Final II: 3, Depth: 4.

WHY?

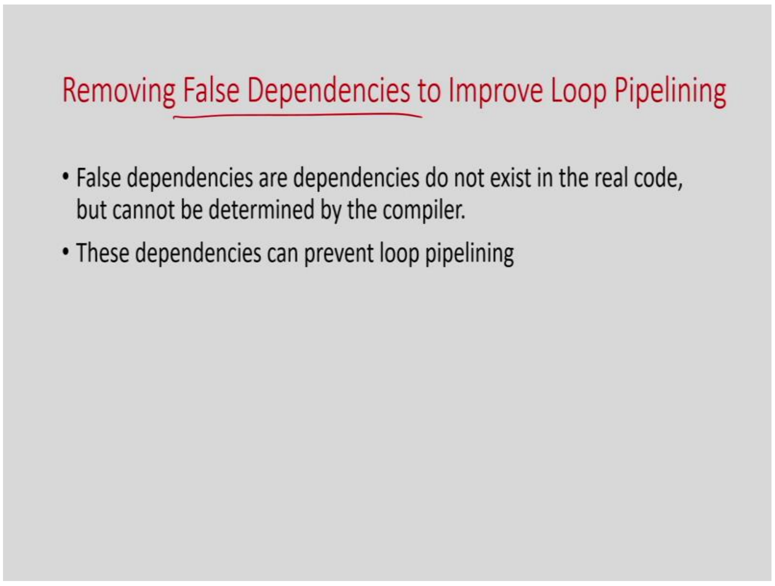
So, with this I can actually achieve my i i equal to 3 for this example, but you can understand that because of this loop carried dependencies this the performance of the loop pipelining is severely getting impacted right. So, and we have to be we should at least aware of this kind of fact that if I have code is like that applying pipe lining does not make sense in certain cases right for example, here this i i can only by achieved 3 3 not 1 or 2 right and previous case the maximum i r the initiation interval can be achieved for loop pipelining 2.

So, once we write our code and try to apply this pipelining kind of pragma we have to be careful or we should at least aware that whether this pipelining will help you or not. In generally if there is no loop carried dependency pipeline give you the best performance I can execute the loop in order of n time.

But because of this dependencies its basically this pipeline can get impacted in some cases there is no pipelining is possible and it will if the one iteration take k k cycle it will take k into n times, but in pipeline even if one iteration take k cycle I can still execute the loop in n n times in n clocks right. That is the biggest advantage that we usually get through pipelining, but this dependencies can actually impacted there ok.

So, the motivation was just to convince that sometime this loop this dependencies can actually impact the loop pipelining ok.

(Refer Slide Time: 41:25)

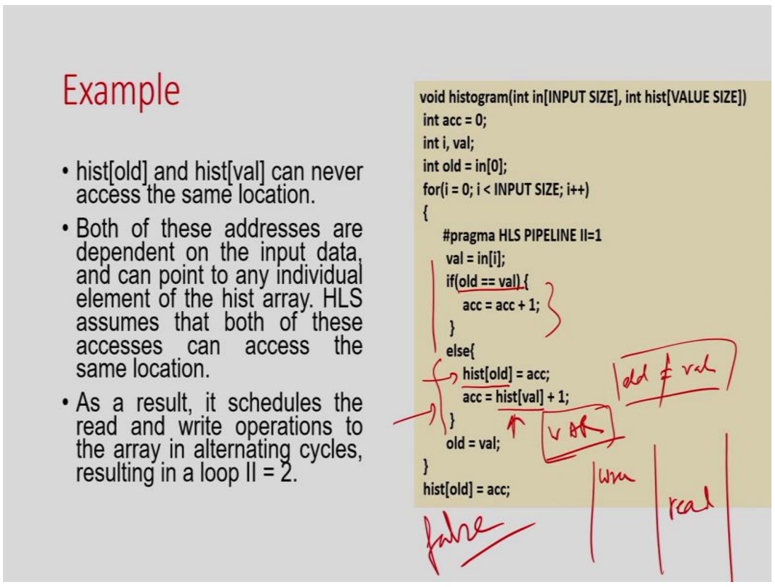


Removing False Dependencies to Improve Loop Pipelining

- False dependencies are dependencies do not exist in the real code, but cannot be determined by the compiler.
- These dependencies can prevent loop pipelining

So, there are certain time there is no actual dependencies but tool cannot understand that. And as a result it cannot do the pipe lining or it actually impact the pipelining performance. So, in such cases if we see that I mean manually we understand there is no dependencies, but tool do not understand, tool not be able to automatically identify that we have to specify that this is not a correct dependency you can ignore this ok and which actually improve the pipelining. So, I will take a small example quickly.

(Refer Slide Time: 41:52)



Example

- hist[old] and hist[val] can never access the same location.
- Both of these addresses are dependent on the input data, and can point to any individual element of the hist array. HLS assumes that both of these accesses can access the same location.
- As a result, it schedules the read and write operations to the array in alternating cycles, resulting in a loop II = 2.

```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE])
int acc = 0;
int i, val;
int old = in[0];
for(i = 0; i < INPUT SIZE; i++)
{
    #pragma HLS PIPELINE II=1
    val = in[i];
    if(old == val){
        acc = acc + 1;
    }
    else{
        hist[old] = acc;
        acc = hist[val] + 1;
    }
    old = val;
}
hist[old] = acc;
```

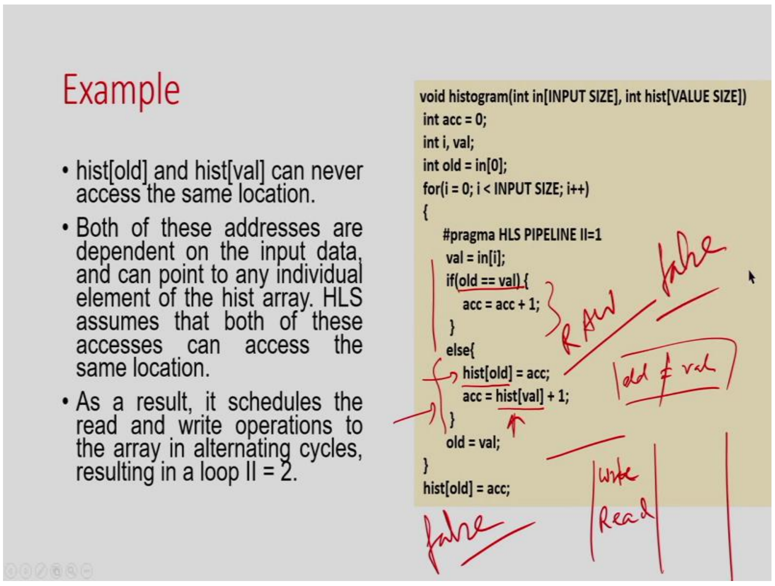
So, suppose in this loop, if you see I am just reading this histogram locations, I am actually reading and writing from the histogram array ok. So, now, in general if you does not look the other part of the code, you can assume that this old and new value can be same also right. Old can be in general right, it can be same and then you are actually reading and writing from the same location and.

So, if since this code was given and there is tool cannot identify that this I mean can actually worst case your old and new val the old and val can be the same value and so, it is basically order this execution it will actually write first and then read, this is basically write after read right. So, I am actually write wire kind of dependencies. So, it actually try to obey this dependencies and is going to schedule this write first and then next clock it is read right.

So, as a result I cannot paralyze this operations which is correct in that sense that is it has to be done like that if otherwise if you just do it in parallel this will get the wrong value, but if you look into the other part of the code. So, what I am doing here if this two value is same old and val equal to same I am not going to do this I am going to do something else.

So, it means that if so, in this case if I come to this else part, this old is always not equal to val so, that means, although there is a w a r kind of dependencies here which is a false dependencies because this is not going to happen in practice because I if old and val is same, I am not going to come to this part of the code. So, as a result actually it is possible to perform this read and write in the same clock right.

(Refer Slide Time: 43:40)

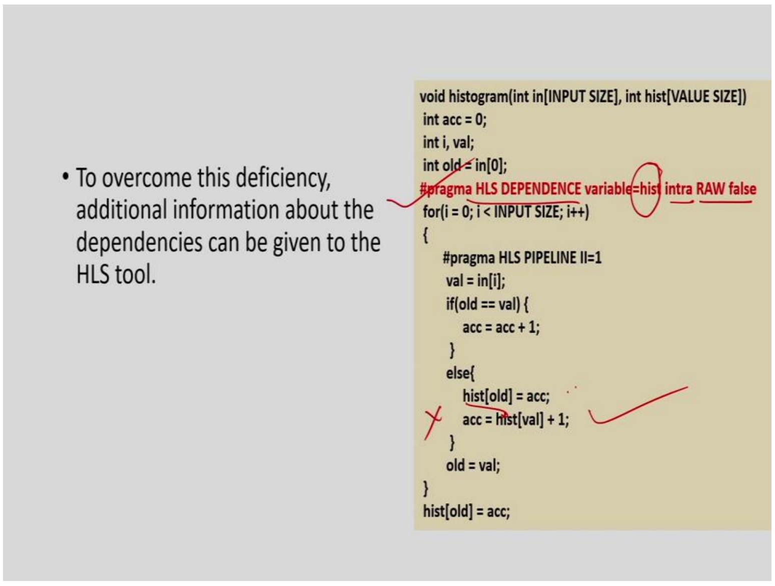


Example

- hist[old] and hist[val] can never access the same location.
- Both of these addresses are dependent on the input data, and can point to any individual element of the hist array. HLS assumes that both of these accesses can access the same location.
- As a result, it schedules the read and write operations to the array in alternating cycles, resulting in a loop II = 2.

```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE])
 int acc = 0;
 int i, val;
 int old = in[0];
 for(i = 0; i < INPUT SIZE; i++)
 {
    #pragma HLS PIPELINE II=1
    val = in[i];
    if(old == val) {
       acc = acc + 1;
    }
    else{
       hist[old] = acc;
       acc = hist[val] + 1;
    }
    old = val;
 }
 hist[old] = acc;
```

(Refer Slide Time: 43:54)



- To overcome this deficiency, additional information about the dependencies can be given to the HLS tool.

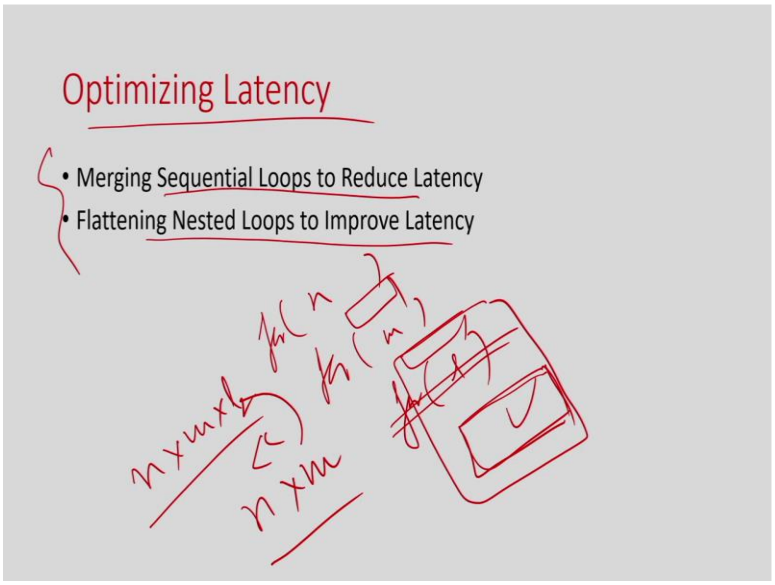
```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE])
 int acc = 0;
 int i, val;
 int old = in[0];
#pragma HLS DEPENDENCE variable=hist intra RAW false
 for(i = 0; i < INPUT SIZE; i++)
 {
    #pragma HLS PIPELINE II=1
    val = in[i];
    if(old == val) {
       acc = acc + 1;
    }
    else{
       hist[old] = acc;
       acc = hist[val] + 1;
    }
    old = val;
 }
 hist[old] = acc;
```

So, but for tool it is very difficult to analyze this kind of complex things. So, just to make sure that high level synthesis tool understand that what you can do to you have to just specify that this pragma that this particular for this variable histogram within that intra within this there is a raw dependency read after write, which is basically a false dependencies right. So, I think I have written it is wrongly it is basically read after write right.

So, this dependency is false dependency and you can ignore this. So, if you just specify this particular instruction then the high level synthesis should understand ok this dependency is wrong. So, I can ignore this and I can schedule this in the same clock and that will give you the performance benefit you can actually improve the pipeline benefit with this kind of thing.

So, the point here we try to make sure that if you apply loop pipelining and if you are not achieving you apply loop pipelining and you are not achieving what the kind of performance you are expecting because of this kind of false dependencies then you have to just specify that this dependencies are not correct right. So, then the tool will actually do the efficient implementation ok.

(Refer Slide Time: 45:07)



Optimizing Latency

- Merging Sequential Loops to Reduce Latency
- Flattening Nested Loops to Improve Latency

for(n
for(m
n×m×k
n×m

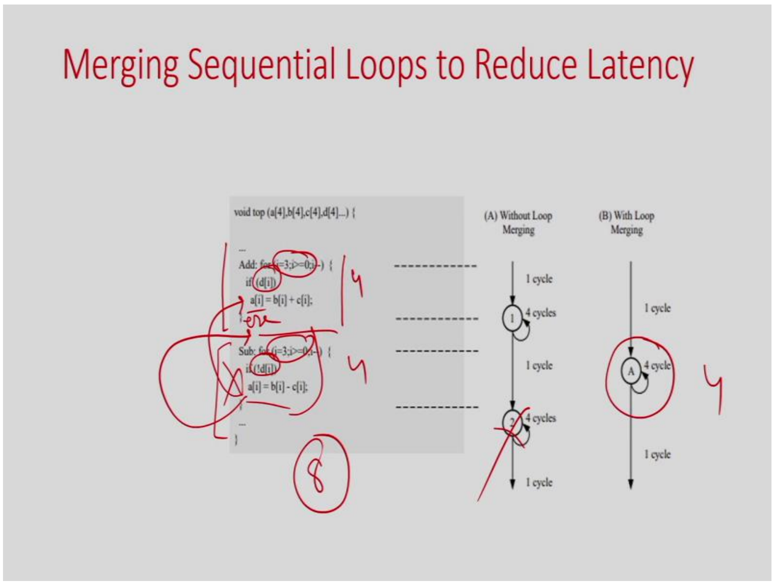
So, now I will move to the last part of this discussion that how we can optimize the Latency in some cases. So, there are two cases where we can actually improve the latency. So, the first thing is that, if there are multiple loops we can multi merge into one loop so, that the latency can be reduced and the second one is the flattening of the nested loop right. So, if there is a for loop under the for loop another for loop and for loop.

So, it is you understand that it is basically if you have something here, it will be executed this say if it is n m and l it will execute into n into m into l times right. And now there may be some part of the code here some part of the code here also right. So, if you just remove this flatten or that means, you unroll this loop and you put it here. So, these operations now can be parallelized right.

So, now the whole part of the code will execute n into m times and this code size is big right. So, it the all l iterations code will be here, but since all code all l iterations code are actually put in one iteration there may be some parallelism is possible some of the operations can run in parallel. As a result your number of iteration will be less than this right. So, this will be much less than of this value ok. So, this will be lesser.

So, as a result you can actually improve performance. So, if there is a long nested loop it is good idea to unroll the innermost loop right. So, as a result the parallelization will improve, as a results the total number of clock needed to execute that particular block will reduce ok.

(Refer Slide Time: 46:52)



Merging Sequential Loops to Reduce Latency

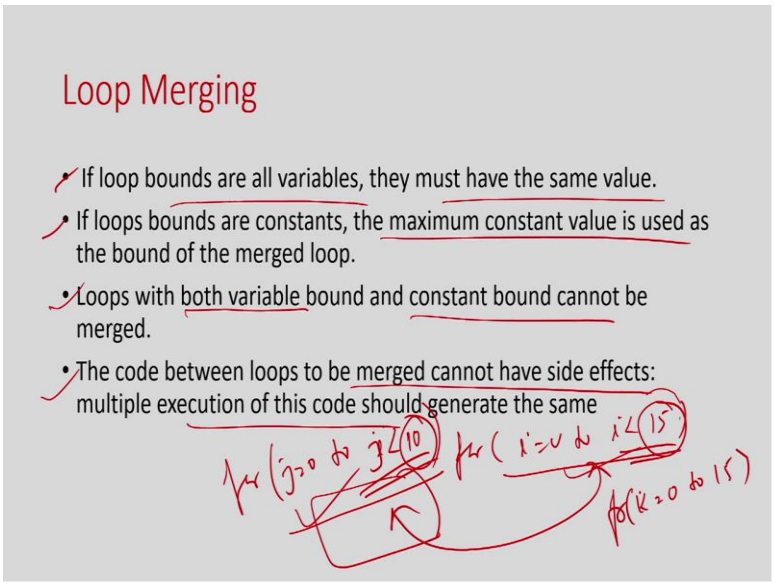
```
void top (a[4],b[4],c[4],d[4]...) {

 ...
 Add: for (i=3;i>=0;i--) {
   if (d[i])
    a[i] = b[i] + c[i];
 }
 Sub: for (i=3;i>=0;i--) {
   if (!d[i])
    a[i] = b[i] - c[i];
 }
 ...
}
```

So, this is one idea and I will just take a example for merging sequential loop say suppose you have a loop one. You are calculating a and you are calculating another loop where you are calculating a again, but here the condition was d i here not d i right. So; that means, this two part of the code is actually mutual exclusive right.

We can just write else here and we can do this, here right. But if you do not do this how many time cycles is taking. So, this will take 4 clock because the loop writing for 4 cycle, this will take 4 cycle. So, it will total take 8 cycles ok. But now here I understand that either this will execute or this will execute. So, I can just remove this loop and I put this as a else part and I have only single loop right. So, now, it will take 4 cycle.

So, you can actually remove one of the loop and because this if else sign is going to make exclusive of (Refer Time: 47:39) the number of operations also remain the same and you can actually just merging such sequential loops into one loop will reduce the latency largely.

(Refer Slide Time: 47:54)



Loop Merging

- If loop bounds are all variables, they must have the same value.
- If loops bounds are constants, the maximum constant value is used as the bound of the merged loop.
- Loops with both variable bound and constant bound cannot be merged.
- The code between loops to be merged cannot have side effects: multiple execution of this code should generate the same

for (j=0 to j<10
for (i=0 to i<15
for(k=0 to 15)

So, in general we can merge two loops if their bounds are variable and they must have the same value right. So, if there is a for loop which is i equal to 0 to i less than n and there is another for loop here which is j equal to 0 to j less than n. So, this condition basically the total number of time this loop will execute total number of this is must be the same right.

Although this is a variable bound if they are same then I can merge it. If there are two loops which has a constant bound say suppose this is running for 10 and this is running for 15, still I can merge I can only I will take the max of this. So, I will now merge in the merge loop I will run for k equal to 0 to 15, but I will put a conditional statement if i less than 10, I am going to do this part of this loop and if it is not then I am going to do this part also right.

So, I can take the maximum count as the loop bound, but I can actually take some conditional operation. So, that the operation here would not execute beyond 10 ok. So, but if one loop is of variable bound, that means, its i 1 to n and this is 1 to 10, I cannot merge them because i do not know the value of n right. So, that is why I cannot merge such loops into one and also once you merge this two code we have to make sure that there is no side effect right.

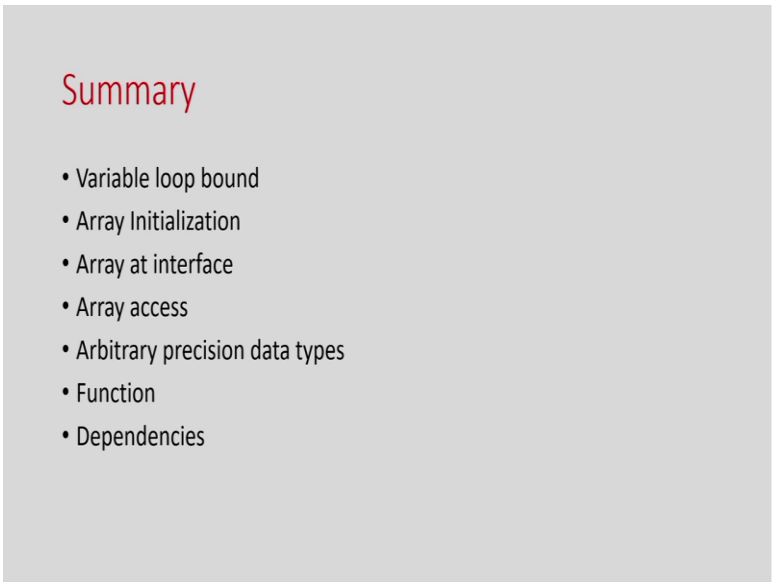
So that means, merging together would not change the functionality of the behaviour. So, we have to make sure that in such cases I can merge and merging of the loop is very important operations and it has to be done manually. We have to understand that this is something a manual operations and it actually improve the latency by a large because loops is something you have to understand that particular body is execute many times and if you merge the performance can most of the time is huge ok.

So, with this I will just summarize this part of the course where we try to understand when try to convince you that writing C code for hardware you have to taken care of various things into picture. Specifically that loops how the efficient the implementation loop is going to happen, the loop bounds loop carried dependencies those things we have to be very careful for efficient implementation of the loop.

Array is also another important portion of the code and its actually have the most its create a lot of bottleneck for efficient implementation because this array maps in to memory and memory has a limited number of ports right. So, for this array initializations, array interface, array access we have to basic idea is that accessed of the array should be minimized and that has to be reused right that also we have already discussed.

So, and also another important it is the Arbitrary precision data type we have to make sure that my data path is actually have the unless there is no redundant width which is something unnecessary creating the area.

(Refer Slide Time: 50:59)



Summary

- Variable loop bound
- Array Initialization
- Array at interface
- Array access
- Arbitrary precision data types
- Function
- Dependencies

And also we should have to think about the efficient implement of the function. We should use the function judiciously because it is created separate module and there is no kind of optimization possible across functions. So, if they are creating separate module. So, if the functions is really use only one or two time used or it is a smaller function it is better to inlined it rather than a function. For only function which is used many times then only it is better to use a module for that.

And also the dependencies that I have already discussed today that the dependency is actually impact lot of in the loop pipelining and we have to be careful about that if sometime this dependencies is false we have to ignore them right. So, these are the tricks you have to always play when you try to get a efficient hardware from C and the more you play and the more you try you will learn lot of integral I mean some complex part of it.

I just take very simple examples, but in general there may be some very complex scenarios comes and, but the bottom line is that you have to take care of these kind of things during the c to r till conversions ok. With this I conclude today’s class.

Thank you.