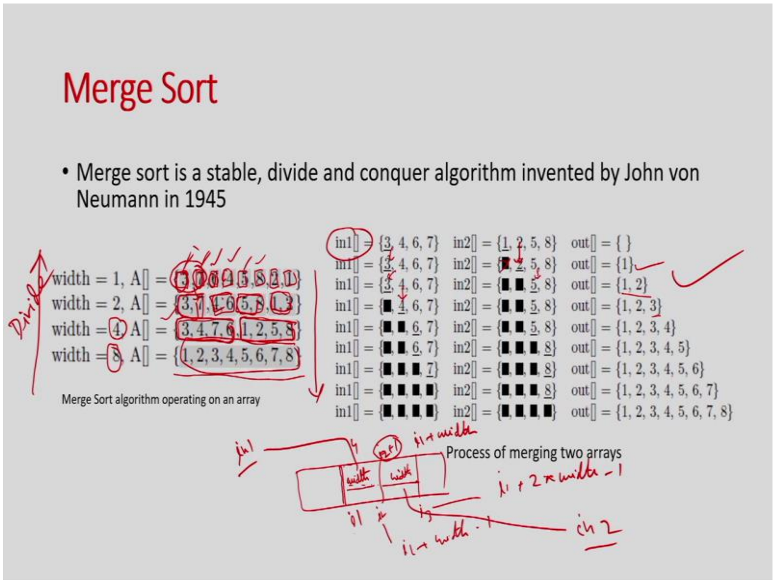
C-Based VLSI Design
Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 09
Impact of Compiler Optimizations in Hardware
Lecture - 31
Optimizations in HLS: Case Study 2

Welcome everyone. In today's class we are going to take a example C code and we will try to see how we can design a efficient hardware using high level synthesis from that input C code ok. So, what we are going to take is merge sort and is merge sort is something is very common everybody knows. So, I took a example which is something everybody understand.

(Refer Slide Time: 01:12)



Merge Sort

- Merge sort is a stable, divide and conquer algorithm invented by John von Neumann in 1945

width = 1, A[] = {3, 7, 6, 4, 5, 8, 2, 1}
width = 2, A[] = {3, 7, 4, 6, 5, 8, 1, 2}
width = 4, A[] = {3, 4, 7, 6, 1, 2, 5, 8}
width = 8, A[] = {1, 2, 3, 4, 5, 6, 7, 8}

Merge Sort algorithm operating on an array

in1[] = {3, 4, 6, 7} in2[] = {1, 2, 5, 8} out[] = { }
in1[] = {3, 4, 6, 7} in2[] = {■, 2, 5, 8} out[] = {1}
in1[] = {3, 4, 6, 7} in2[] = {■, ■, 5, 8} out[] = {1, 2}
in1[] = {■, 4, 6, 7} in2[] = {■, ■, 5, 8} out[] = {1, 2, 3}
in1[] = {■, ■, 6, 7} in2[] = {■, ■, 5, 8} out[] = {1, 2, 3, 4}
in1[] = {■, ■, 6, 7} in2[] = {■, ■, ■, 8} out[] = {1, 2, 3, 4, 5}
in1[] = {■, ■, ■, 7} in2[] = {■, ■, ■, 8} out[] = {1, 2, 3, 4, 5, 6}
in1[] = {■, ■, ■, ■} in2[] = {■, ■, ■, 8} out[] = {1, 2, 3, 4, 5, 6, 7}
in1[] = {■, ■, ■, ■} in2[] = {■, ■, ■, ■} out[] = {1, 2, 3, 4, 5, 6, 7, 8}

Process of merging two arrays

And then we will try to first I will just try to explain what is merge sort and then how that algorithm works and then I will talk about the high level synthesis of merge sort ok. So, if you already know that merge sort is a stable sorting algorithm, so, it is given a set of numbers it can be integer or floating point does not matter, set of numbers. And then you basically sort them in ascending order right and we usually apply merge sort which is developed by John von Neumann in 1945 which is kind of a divide and conquer algorithm ok.

So, the first thing is it is basically how the algorithm works, let me try to explain. So, basically we divide the big array into always half right. So, we have divide by two basically. So, we have the big array we divide it by two and then again I keep dividing the by two by two and so on right. So that means, I have a log n step until I reach a unit size array right.

So, first suppose if you think about a array size of 8, the width is 8. So, I just divide into two arrays right. So, this is my divide part right. So, there is a divide part going on and then I have two arrays and then I am going to take this array and again I am going to divide into two right.

So, and I will take this array I will again divide into two right, this is how the algorithm works then I will take this array I will divide into two. So, now, it is becoming unit size array right. So, then I am going to take this I will divide it into by 2 and then I will take this array I will divide into 2 and I will take this I will divide into 2. So, this process will go on until I individual array size become 1, right. So, this is my divide approach and then I am going to conquer right. So, I am going to merge it.

So, once you reach this individual array level I can assume that this is already sorted array of one elements right. So, this is a sorted array of one elements. So, I am going to merge these two sorted array into another one. So, the problem here is that I have given two sorted array and I have to merge them into a single sorted array right. So, what is happening in this direction?

So, I take this array and this array I will I have these are the two sorted array I am going to merge them into a sorted array like this. Then I am going to take this and this I am going to merge them I am going to get this right. So, similarly I am going to take these two sorted array. You can understand they are actually two consecutive sorted array, but the index is very much interesting.

So, if I start from say location say i1 I have a sorted arrays of width size say 4. So, this is the so, it depends on the which level you are doing the merging and then this is my i2 and the next array start from i2 plus 1 and it goes say some i3 again this width is basically width right. So, this is how merging happen.

So, this is also width. So, the basic idea of merging is that we have two sorted array and they are actually consecutive and we have to merge these two sorted array into another sorted array of size which is double of these two size right. So, basically the summation of these two size.

And if you look into the problem, so, I if I start a follow location i1 and if the width of this say sorted array is width then this index is i2 is nothing but i1 plus width minus 1. This is my width and this and the next location the next array start from i2 plus 1 right. So, which is nothing but i i1 plus width right and this is still i3. So, i3 is nothing but basically this i1 plus 2 into width minus 1 right. So, this is how this happen.

So, basically if you understand here is that one once I take these two array I just sort them into this I take these two array I sort this into this and so on. And then I am going to take this two array of size 2, I am going to create a sorted array of size 4 right similarly I am going to take these two sorted array of size 2, I am going to create a sorted array of size 4.

And then in the last step when the my width become 4, I will take two array of size 4 and I am going to create a sorted array of size 8 right. So, then I am going to get a sorted array. So, this is the whole process and this is a and this is how this merging happens. So, suppose I have given two sorted array. So, this is one sorted array this is one sorted array. You remember this is basically nothing. So, this is basically my in1, right. So, this is my in1 array the way it is written here and this is my in2 array.

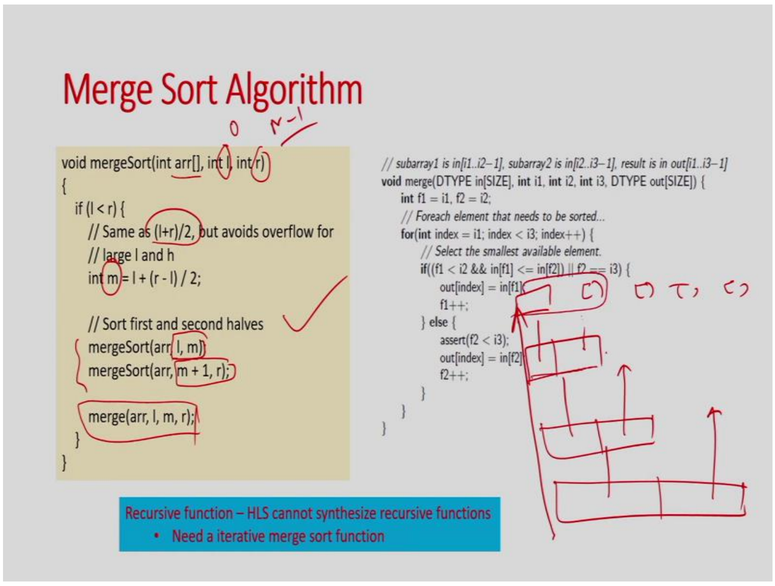
And how the merging happens? So, basically I will maintain a pointer at the current point because these are all sorted arrays. So, what I should do? I just compare these two element and who whichever is the smaller one I am going to copy that into my output array right. So, among 3 and 1, 1 is smaller. So, I just put it 1 into the sorted array and then I am going to increase this pointer from 1 to 2 right.

So, this is where I am going to do and this is the this since this array is still my pointer will be at the location 3. Now I am going to compare 3 and 2 and I found that. So, this means this part is already sorted. So, 2 is smaller. So, I am going to copy 2 into my sorted array next location and I increase the pointer. So, my current point will be here and it will be still hit here.

So, now, I am going to compare 3 and 5 and I found 3 smaller. So, I will copy 3 into the output array the next location and I will increase the pointer here right. So, the every iterations I will basically take the current locations of two arrays compare which number is smaller. I am going to copy that particular element into the output array and increase that pointer by 1, right.

So, this process will go on. And at some point of time what is going to happen? It might happen that one array is finished that all the elements of one array is copied then the rest of the elements of the other will be copied one by one right. So, that is all. So, this is overall merge sort everybody you aware, but I just for re-capsulations of the things I just explained right.

(Refer Slide Time: 07:31)



Merge Sort Algorithm

```
void mergeSort(int arr[], int l, int r)
{
   if (l < r) {
      // Same as (l+r)/2, but avoids overflow for
      // large l and h
      int m = l + (r - l) / 2;

      // Sort first and second halves
      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);

      merge(arr, l, m, r);
   }
}
```

```
// subarray1 is in[i1..i2−1], subarray2 is in[i2..i3−1], result is in out[i1..i3−1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
   int f1 = i1, f2 = i2;
   // Foreach element that needs to be sorted...
   for(int index = i1; index < i3; index++) {
      // Select the smallest available element.
      if((f1 < i2 && in[f1] <= in[f2]) || f2 == i3) {
         out[index] = in[f1];
         f1++;
      } else {
         assert(f2 < i3);
         out[index] = in[f2];
         f2++;
      }
   }
}
```

Recursive function – HLS cannot synthesize recursive functions
- Need a iterative merge sort function

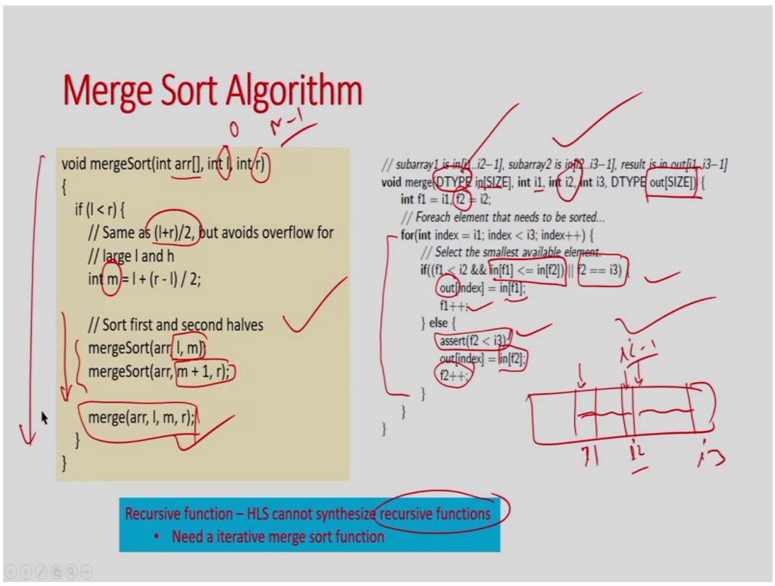
And this is there in any data structure books that merge sort algorithm. So, this is my that divide process that the way I do basically I always find out the mid which is basically l plus r by 2. And then what I do? I just do the merge sorting for the same array. So, from l to m right. So, initially I call this merge sort for from index l to r right. So, initially I will just call from 0 to n minus 1 right if size of the array is n.

And then I am going to take the middle and I will call this for left part of the array and this is for the right part of the array and this is a recursive call right, so, basically you can understand that if you just do this. So, it will recursively break this part only right. So, because if you take this, so, it will break this then from it will be called for this then it will break like this and it will call for this. It will break like this and then you will call for this.

Once this is done then it will be called recursion call come back to this then it will break this then recursion come back again, it will break this then recursion come backs and then start to working on this part right. This is how the recursion will happen. And then once i so, these two process will make sure that I will still I reach all the individual array at this level then this recursion will happen I mean stop and then this tail recursion part that merge part will going to happen right.

So, that now the merging will happen recursively because I have so, many calls. So, automatically so, many merge will happen. So, finally, the first iteration this level of merge will happen then the merge of 2 will happen then merge of 4 will happen and so on right. So, this is how the whole thing will work right. So, these are all we aware off.

(Refer Slide Time: 09:10)



Merge Sort Algorithm

```
void mergeSort(int arr[], int l, int r)
{
   if (l < r) {
      // Same as (l+r)/2, but avoids overflow for
      // large l and h
      int m = l + (r - l) / 2;

      // Sort first and second halves
      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);

      merge(arr, l, m, r);
   }
}
```

```
// subarray1 is in[i1..i2-1], subarray2 is in[i2..i3-1], result is in out[i1..i3-1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // Foreach element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
        // Select the smallest available element.
        if((f1 < i2 && in[f1] <= in[f2]) || f2 == i3) {
            out[index] = in[f1];
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = in[f2];
            f2++;
        }
    }
}
```

Recursive function – HLS cannot synthesize recursive functions
- Need a iterative merge sort function

And then this merge function is basically whatever I just explained in this diagram. So, this particular algorithm is written here right. So, here to make it a little bit generic I just put a data type is DTYPE which is basically can be integer float anything right. So, this is my size and this is the first index that the way I talked about. So, basically this is my i1 and as I mentioned so, basically I am going to take a array like this. So, this is my i2 and i3 is given by this i3 right.

So, this is also width this is also width. So, from these three pointers I can understand the start location of the first array and start location of the second array right that is what is happening here that my start location of second array is f2 because that is i2 this is the start location of the second array and this is basically i2 minus 1 is the n location of the first array right. So, the both ways.

So, the in early earlier I just talked about this is i2, but you can consider the start location of the second array also r 2, the similar thing right. So, we need basically three pointer not four pointers because this i2 and i2 minus 1 is basically we can manage from this i2 ok.

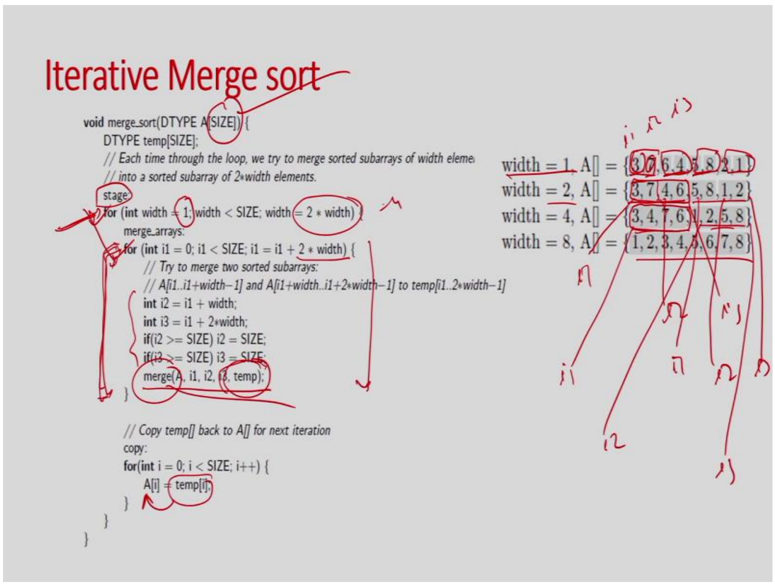
And then this is the i3 is the last location. So, what is happening here? You can understand this is my the loop that I talked about in each iteration. I ideally I am checking whether this element less than this then I am copying this the element of the first array in a output array and I increase the pointer.

Otherwise, I just copy the element of the second array and increase the second pointer. So, this part of the code make sure that if one array is completed it will copy the other part of the other array directly right. So, that is actually answered by these two lines ok. So, this is the overall algorithm and once this is done this out is the output array right and ideally I should copy this out to the original array because if I want to sort the array into the same place so, I need to copy this output array into input array right. So, that is all.

So, the first thing is that suppose I so, you get this code you have written this code right for merge sort and now try to do the h high level synthesis for this, what is the first problem you are going to face? The first problem I am going to face is the recursion right. If you remember that high level synthesis does not support this recursions right. So, if you try to synthesize this code using high level synthesis it will say that I cannot handle this recursion, so, I am not able to generate a hardware from this right.

So, this although this is the right version of the merge sort algorithm you will not get able to generate a hardware from this because of the recursion. So, what is the option? So, you have to make a iterative version of the merge sort and which is not very difficult. So, we can write that. So, I am not going to change this function this function will (Refer Time: 12:09). So, I have to write a iterative version of this algorithm, ok.

(Refer Slide Time: 12:15)



Iterative Merge sort

```
void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
    // Each time through the loop, we try to merge sorted subarrays of width eleme
    // into a sorted subarray of 2*width elements.
    stage:
    for (int width = 1; width < SIZE; width = 2 * width) {
        merge_arrays:
        for (int i1 = 0; i1 < SIZE; i1 = i1 + 2 * width) {
            // Try to merge two sorted subarrays:
            // A[i1..i1+width-1] and A[i1+width..i1+2*width-1] to temp[i1..2*width-1]
            int i2 = i1 + width;
            int i3 = i1 + 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            merge(A, i1, i2, i3, temp);
        }

        // Copy temp[] back to A[] for next iteration
        copy:
        for(int i = 0; i < SIZE; i++) {
            A[i] = temp[i];
        }
    }
}
```

width = 1, A[] = {3, 7, 6, 4, 5, 8, 2, 1}
width = 2, A[] = {3, 7, 4, 6, 5, 8, 1, 2}
width = 4, A[] = {3, 4, 7, 6, 1, 2, 5, 8}
width = 8, A[] = {1, 2, 3, 4, 5, 6, 7, 8}

And this is this. So, the idea here is to explain this, so, I do not have any recursion. You see here there is no recursion. So, I just recursion I just replace by two for loop right. And what is the idea? If you remember that finally, the that recursion what it does? It basically break the whole thing till the size width is 1, right.

So, what is happening here? So, I will just maintain outer loop which is called stage. So, it is basically the divide is this loop will take care of that. So, basically the divide is basically is a trivial text task. You can assume that basically what is happening. I am going to have the array. First I am actually merging the individual array into two and then individual array into two then I am taking two consecutive array size of 2 merging into a and create a sorted area of size 4.

And again I am going to take this, I am going to create this and from this two size 4 I am going to create 8, right. So, this outer loop is basically that stage. What is happening here is it just assume that my width is 1 initially right. So, you basically maintain that width is 1 then 2 then 4 then 8 is basically 2 to the power i because it is a double. So, by 2 is happening right.

So, basically this outer loop the stage loop will basically will give you the size width. So, when width is 1 I am going to take two array size of array size of 1 and I am going to merge them into 1 right. So, that that process is happening here right. So, what is happening here is that whenever my width is 1; So, how many merging is going to happen? I am going to take this two array. So, for these two array I have to decide what is my i1 this what is my i2 and what is my i3. So, that calculation is happening here and then I am going to call the merge right and after merging the result is going to store in array temp ok.

And then since so, when is width is 1, so, how many times if the array size is 8, how many times is going to happen? This loop will go for four times because it is 2 into width right. So, basically it merge these two then you will take these two then it will take these two then these two right. So, the inner loop is going to do that.

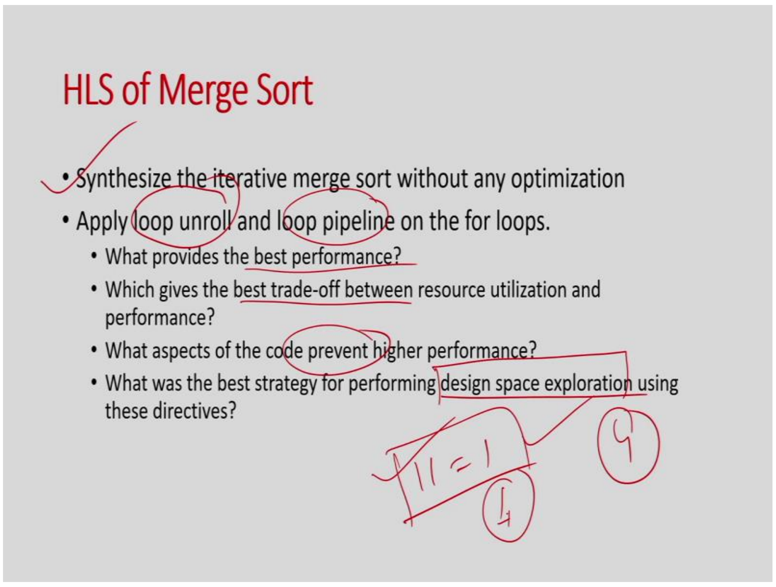
Then once this is done, so, I have a sorted array like this. So, at every two individual element is sorted then my width become 2 into width. So, now I am going to take the width 2 and then again this loop will be called. Now, this loop will again decide that this i1, i2, i3, this 3 index for these two array first right.

So, it will take this array. So, this will be my i1, this will be my i2 and this will be my i3 and it will call this after this sorting is done and this sorted array it is generated it will now start it will create this is my i1, this is my i2 and this is my i3 and it will sort right.

So, , since this is 8 size, so, this loop will go for 2 times then this width b will become 4 right. So, once this width is 4 this loop will go for one iteration. So, it will just decide this is my i1, this is my i2 and this is my i3 and it will just do the sorting and you will call the merge to do that.

So, this is my iterative version of the algorithm. And I think it is very much clear. And then once everything is done as I mentioned the temp is my sorted array I should copy to the original array right. So, I am just copying the data to that because that is the array I want to sort. So, this is there. So, this function with my this merge function. So, if I take this merge function and this merge sort array and then if I ask high level synthesis to synthesize it will be able to do a synthesis right, it will able to give you a RTL right.

(Refer Slide Time: 16:02)



HLS of Merge Sort

- Synthesize the iterative merge sort without any optimization
- Apply loop unroll and loop pipeline on the for loops.
  - What provides the best performance?
  - Which gives the best trade-off between resource utilization and performance?
  - What aspects of the code prevent higher performance?
  - What was the best strategy for performing design space exploration using these directives?

So, what I am going to ask you, you just do that right. So, you take these two version of the code and you synthesize it using any high level synthesis tool specifically Vivado HLS what we will going to take in this particular course. So, we will get a basic implementation right and we will see what is the latency of the design and what is the area, what is the performance, what is the clock you can achieve. So, you just try to measure that ok.

So, here you can actually fix the size because once you run something you can fix to say, so, any size. So, I the example I have shown for 8, but you can actually create to any size. So, one, but one you synthesize you just you have to specify the size because dynamic array is not supported right.

So, you cannot have a hardware which can execute size 4 size 8 and 1 right. So, it has to say this is array sorted array, this hardware can handle a array of say 16 or say 32 or 20 whatever it is. So, it is only going to handle that size ok. So, you will generate the RTL. Next as a developer what you want to achieve? So, the base level may not meet your target right.

So, I want to create a efficient hardware. So, that will be my objective and now I am going to try to develop a efficient hardware for this merge sort. So, basic I have created a version of the algorithm which is synthesizable first of all. The initial version was not synthesizable. So, I do some modification I make it synthesizable and I get a hardware basic hardware and which is obviously, not the efficient one. So, the next things what you we have to think of? We have to apply the tool features optimization features.

And since we can see here the it is there is a loop. So, effectively you can see here there is a for loop here there is a for loop here and this merge is also there is a for loop right. So, there is a three nested for loops are there. So, and there is a array right. So, this temp a are array.

So, the; obviously, the optimizations of the tool that will be relevant for this is unroll loop unroll, loop pipeline and array partitioning right. So, what I am going to suggest is you should actually apply this loop unroll, loop pipeline and also add a partitioning in different combinations ok. So, and see what is the how the performance is varying right. So, this is what is called design space exploration right.

So, basically you can apply this unroll you can apply pipeline. So, if you ideally you know that if I unroll my area will blow up because there is no loop it will create this many copies, but it will run faster. It the my time to execute the things will reduce right. So, earlier if the loop execute for 3 times sorry 10 times and if I unroll probably it might take 1 or 2 clock to complete it, right.

Ideally can 1 but because of some dependencies might take two 3 clocks. So, the design will be faster, but area avoided will be more and the pipeline is something is kind of a trade off right. So, as the area overhead is not that significant because you have to just add some pipeline stage and probably you cannot share some of the resource because of the stages, the stages happening independently.

But because with the pipeline I can actually execute the multiple iterations of the loop in parallel in pipeline manner my performance usually is order of n of the number of iterations right. So, that I we already know that. So, what I will strongly recommend you just you take this two version of the code.

You try to apply this unroll pipeline and array partitioning and see how the performance is varying and what is the particular scenario where you get the best performance from the time, what is the best area optimized design and what is the trade off right. So, what is the combinations where you actually get the area avoided is not that much, but performance is also significantly improved.

So, this is what you should actually do once you do a use high level synthesis to develop a efficient hardware from a given design ok and that is what is called design space exploration. So, this design space exploration you can actually automate. You can actually write some script which will add those pragmas like unroll, pipeline, those things in your code and just run it and then see the give you the results then you manually compare which is good which is bad and all right. So, you can actually develop a tool for that or, but most of the time people actually do it manually, right.

And if you look into the code again, since I just talked about three levels of loops; it is unroll you can actually unroll the innermost loop, the middle loop or the outer loop also right. So, and every scenario it will actually give you a different different hardware or you can actually think of pipelining the inner loop and say you can apply pipeline in every stage. But usually pipeline works better when you actually apply in the innermost loop right.

And also you can actually apply the pipeline at the top most loop, middle level loop and so on. You can actually unroll the inner most and then you pipeline the middle one. So, there are so many possibilities you can understand here and you can actually partition the arrays. You just think of those things which can actually give you the performance benefit right.

So, doing this all things is basically there are many combinations and I strongly recommend you to do this exercise it will actually help you to understand how this optimization features of the tool actually help impact in the hardware right. So, that something is doable. But, then is it something that I am able to achieve the best performance or is there any coding the because of the some part of the code it is actually create prevent me to create a good order.

So, what I am trying to say that I apply pipeline, but I cannot I am not getting pipeline initiation interval is 1 right. So, you know the in the context of pipelining the if you have say 4 stage pipeline, so, my I I is the how frequent I can give the inputs right. So, if I my initiation interval is 1; that means, every clock I can fed into the next iterations right or next input.

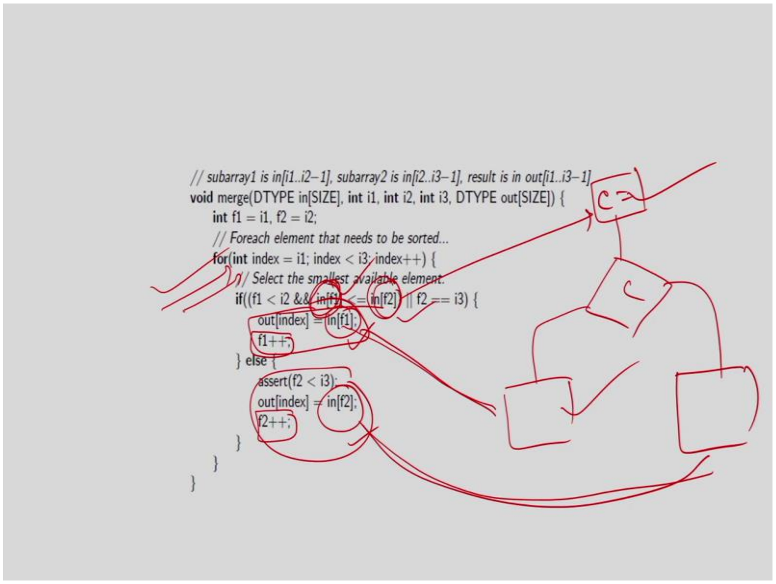
So, basically that is the maximum benefit, but if it is a say there is a 4 stage pipeline and then your I I is about 4. So, basically the pipeline is doing nothing right and this happen this. So, you will not able to get this. So, this initiation interval of a pipeline circuit is one because of primary two things. One is the data depend loop current dependencies right. So that means, you are doing calculating something in iteration I and that is going to be used in iteration say I plus 1.

And say that the last operation calculate that that particular things and the first iteration first operations of this next iteration use that. So, I cannot just start this until this data is ready. So, this kind of that stop this it adds stall in the pipeline right and you so, your pipeline does not give you the maximum benefit right.

So, and also the second thing is the memory, memory access and you remember I have already discussed that once you map this array this they will be mapped to block RAM in the hardware. And the one of the challenge in the developing hardware is that this block RAM has a limited port right. So, at most 2, 1 or 2 ports are there.

And so, whenever there is a more than two access to array in the loop so, it will actually create multiple it take multiple iteration just to read those element and that might create some problem in this getting the initiation interval of the loop is 1 or basically get the maximum performance benefit from the pipelining.

(Refer Slide Time: 23:36)



```
// subarray1 is in[i1..i2−1], subarray2 is in[i2..i3−1], result is in out[i1..i3−1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // Foreach element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
        // Select the smallest available element.
        if((f1 < i2 && in[f1] <= in[f2]) || f2 == i3) {
            out[index] = in[f1];
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = in[f2];
            f2++;
        }
    }
}
```

So, now once you apply this you will notice that you will not get I I equal to 1, if even if you apply this loop unroll and pipeline, specifically the pipeline to this loops innermost loops specifically the innermost loop and it says that I am not able to generate the II is equal to 1 because of this reason. So, it will give you some reasons right.

So, then you have to look into the code what causes the problem right, what creates the obstacle for me to get the best hardware right. And let us now look into the code because I talked about that I am going to apply this pipeline in the innermost loop because that gives me the benefit of benefit maximum benefit and there are two problems here ok.

So, let me try to explain that. The first problem is you have four memory access right. So, I am accessing this in over here in here in here and in here four memory access, but they are actually from two locations right this both this and this is actually the same memory locations alright and this and this also from same memory locations. So, although there are four memory access we are read actually two element.

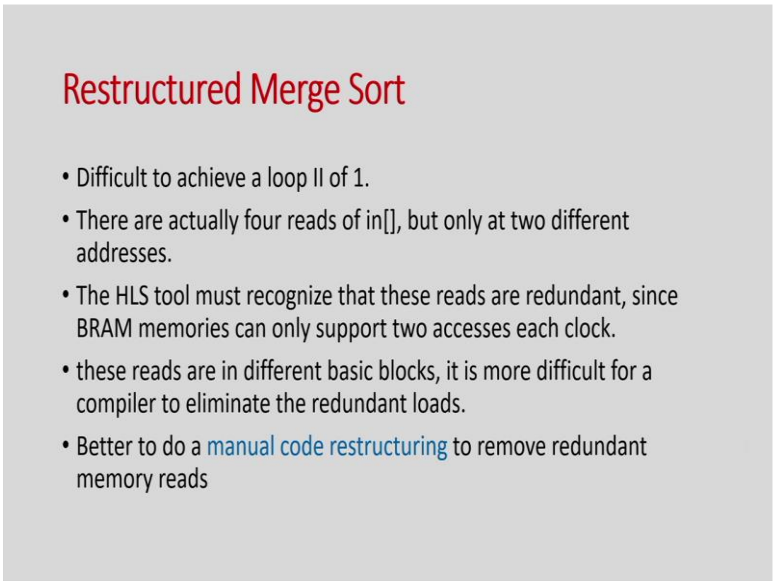
Now, the question is does the high level synthesis compiler understand this, can in actually understand this is the these are not 4 reads these are 2 reads? Unfortunately most of the compiler will not because there are this operation happening in different basic block right. So, let me try to explain.

So, this is the condition right. So, this condition is this say. So, there some reading is happening. So, this read will happen here right. This read will this in one and it will happen here and then this condition will be calculated here and based on the condition you will go here. This is the this if block this is this and else you are going to do this right. So, this part is going to happen here.

So, you see here this read happen in different basic blocks. This read is happening here, this read is happening here and these two reads is going to happen here right. So, because this reads happen in different basic block the compiler may not able to understand this are actually single. So, these two reads of in using f1 is same right. As a result it may actually read 4 times from the memory and that might create a bottleneck for this right. So, that is something is the first problem.

The second problem is there is a loop current dependency here is also because you do f plus plus or f2 plus plus and that is going to use in the next iteration right. So, in current iteration I am going to increase this index f1 or f2 and that we will going to use here, but that you cannot change here because that something is needed here. There is no other alternative. So, I cannot change that, but this I can obviously, change. What I can do? I can just read this in f1 and in f2 before start of the loop using some temporary variable and I just use in everywhere right. So, that will solve my problem that it, ok.

(Refer Slide Time: 26:30)

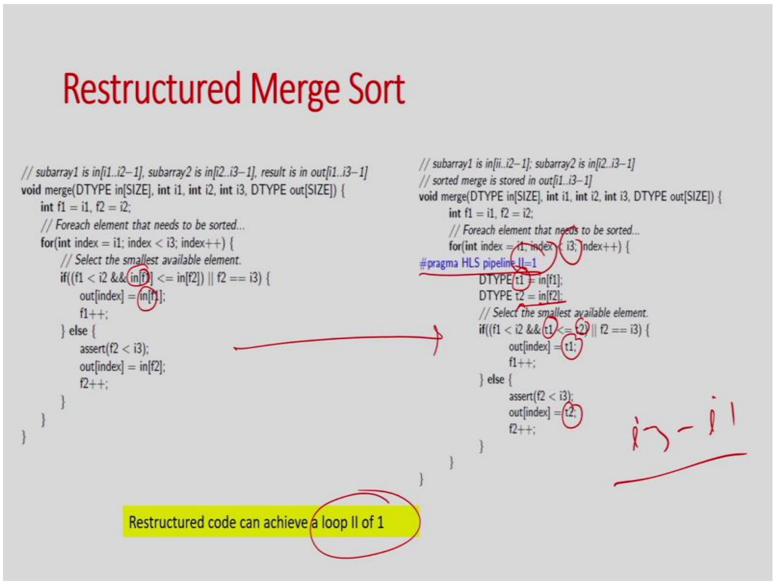


Restructured Merge Sort

- Difficult to achieve a loop II of 1.
- There are actually four reads of in[], but only at two different addresses.
- The HLS tool must recognize that these reads are redundant, since BRAM memories can only support two accesses each clock.
- these reads are in different basic blocks, it is more difficult for a compiler to eliminate the redundant loads.
- Better to do a manual code restructuring to remove redundant memory reads

So, as I told you this is that it is very difficult to achieve by initiation interval 1 for this big example because of this memory read. And this as I mentioned there are this in happening four times but there are two different address. So, tool may not able to understand that this reads are redundant. And as a result because of this they are actually happening in different basic block and so, this it is very difficult for the tool to identify they are the same read. So, I we should do some manual code restructuring here to achieve that ok.

(Refer Slide Time: 27:07)



Restructured Merge Sort

```
// subarray1 is in[i1..i2−1], subarray2 is in[i2..i3−1], result is in out[i1..i3−1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // Foreach element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
        // Select the smallest available element.
        if((f1 < i2 && in[f1] <= in[f2]) || f2 == i3) {
            out[index] = in[f1];
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = in[f2];
            f2++;
        }
    }
}
```

```
// subarray1 is in[ii..i2−1]; subarray2 is in[i2..i3−1]
// sorted merge is stored in out[i1..i3−1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // Foreach element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
#pragma HLS pipeline II=1
        DTYPE t1 = in[f1];
        DTYPE t2 = in[f2];
        // Select the smallest available element.
        if((f1 < i2 && t1 <= t2) || f2 == i3) {
            out[index] = t1;
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = t2;
            f2++;
        }
    }
}
```

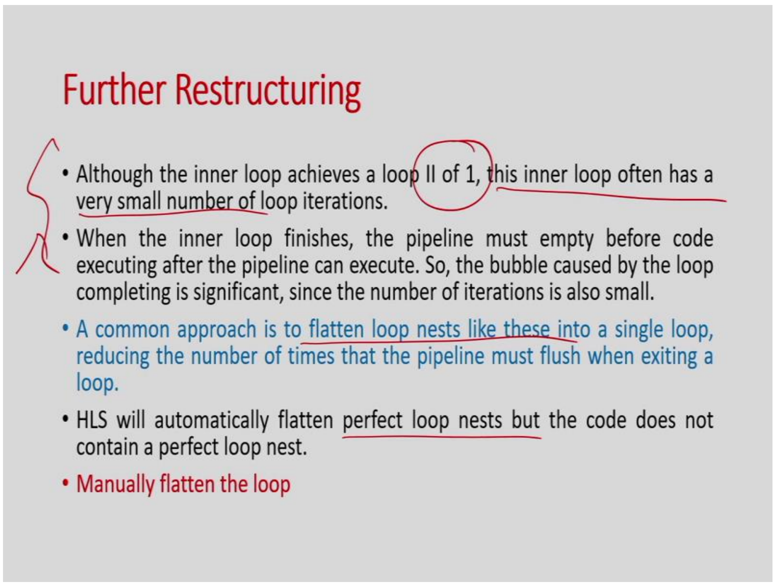
$i_3 - i_1$

Restructured code can achieve a loop II of 1

So, the suggestion is I already mentioned that I am going to I am not going to read this way. I just read this in f1 and I will store in a temporary variable. I will read in f2 and store in a temporary variable t 2. And everywhere instead of using a this in 1 and in 2, I am going to use t1 t2. I am going to use t1 or t2. So, that is all my problem because I do not have four reads.

So, this simple modifications help you to achieve pipeline 1 that is a simple modification, but this is something you have to understand once you write this code and you apply pipeline and you are expecting that this should happening my II equal to 1, but this is not happening. And then finally, you have to realize this is because of this and just do this modification and you should able to the initiation interval of I if I apply pipeline in this particular loop ok. So, that something is good.

(Refer Slide Time: 28:04)



Further Restructuring

- Although the inner loop achieves a loop II of 1, this inner loop often has a very small number of loop iterations.
- When the inner loop finishes, the pipeline must empty before code executing after the pipeline can execute. So, the bubble caused by the loop completing is significant, since the number of iterations is also small.
- A common approach is to flatten loop nests like these into a single loop, reducing the number of times that the pipeline must flush when exiting a loop.
- HLS will automatically flatten perfect loop nests but the code does not contain a perfect loop nest.
- Manually flatten the loop

Now, the question is does this restructuring help ok, does that give you good benefit? The answer is not much because of the two problems. The first problem is that most of the time this loop actually call for smaller arrays. You just remember that when you merge to size of 1, so, this loop will actually execute only 1 time right or 2 times because there are 2 elements right.

When you are actually merging size 2, it is basically it had 4 times ok and most of the time it is basically has smaller arrays because it call 8 times when the if I assume my array size is 8 this particular function will call 4 times and every time it will just read out for 2 times ok. So, the number of iterations is basically small and we are calling many times for smaller arrays.

Only for bigger arrays when you are merging to 4 arrays it will be called 1s right. So, because of the bigger array we have the less number of arrays right. So, the primary problem is that although I achieve my initiation interval 1 for this loop, but this most of the time this loop will actually transfer very less number 1 time to type 4 time and so on.

So, this does not give you much input benefit on pipelining right. So, even if I apply pipeline I have achieve my II is equal to 1, but this may not be the best design it does not give you much performance benefit right. And there is another problem is that because once you apply pipeline you get a you have pipeline stages and you have to flush the pipeline before you start the next iteration.

And here because I call this function often for smaller arrays and my first call does not give me any benefits and then you need some clock to flush the data because otherwise this pipeline will calculate something wrong right. So, you have to flush the data and then you have to call the next function. So, this actually adds some extra over rate of making it pipeline although my pipeline is not giving benefits.

So, this point two point suggest that this applying pipelining to the innermost loop is not something is useful. Also there is another problem here, this loop is actually data dependent right because this how many times this loop will execute is basically i3 minus i1 times right and so, this is an input to this particular function.

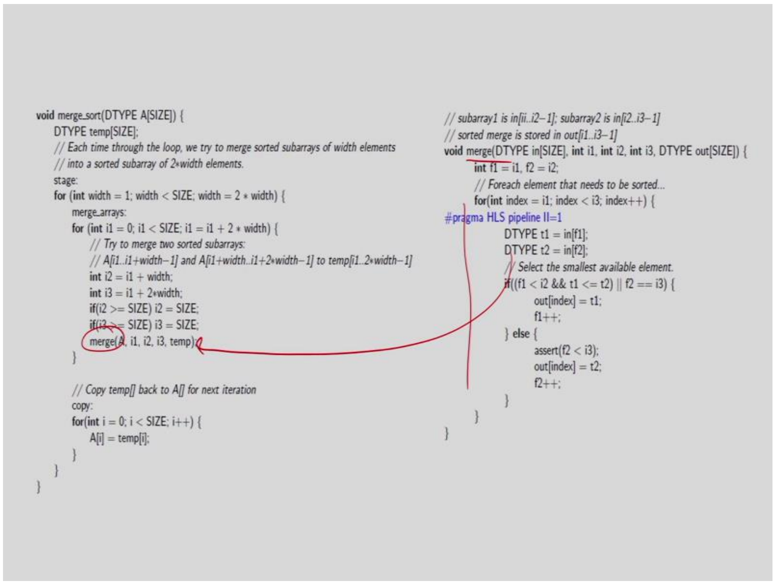
So, it is basically and if you know that high level synthesis once the number of iteration is not known statically known it cannot give you any performance estimation whether it will it you do not say that this particular loop will take 3 cycle or 4 cycle or 5 cycle. So, you will not literally can see the benefits right. So, it will not show you that. It just give you question mark on the minimum and maximum number of latency needed for this because this is basically data dependent loop ok.

So, basically we actually I know what are three problems. First thing is that pipeline does not help here, it create unnecessary overhead flushing the pipeline and third thing is the performance benefit is not visible here because the data dependent loop ok. So, in this case the common approach that we should be taken is the flatten the loop right, so, into a single loop right.

So, here what I am trying to say is that i1 because this loop is not so, important what about just putting this loop into this original merge sort itself right. So, if I just go back here. So, instead of calling this merge sort I can actually take the body of the merge and I can place it here. So, that is what is basically flatten right.

So, actually this flatten is done by high level synthesis tool for perfect loops ok, but this is not a perfect loop because there are lot of conditions and all. So, for this HLS tool may not able to flatten the loop. So, you should do manually flatten right. So, that is what I did next.

(Refer Slide Time: 32:05)

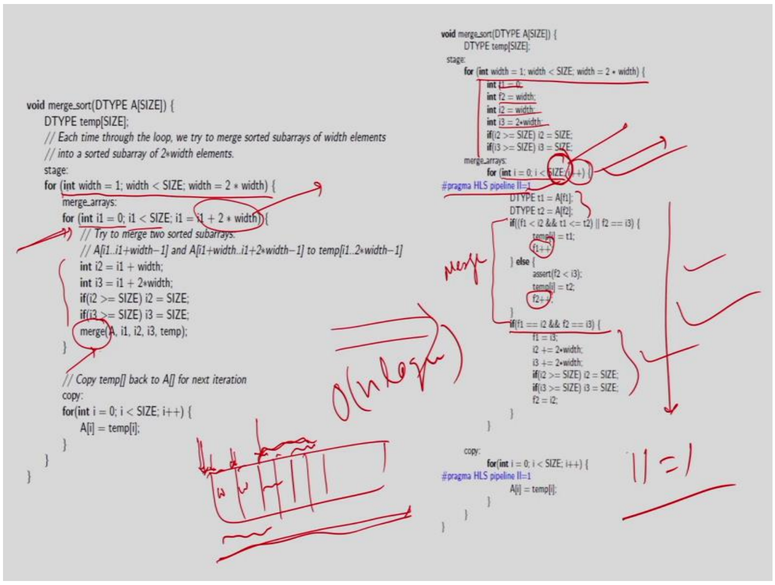


```
void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
    // Each time through the loop, we try to merge sorted subarrays of width elements
    // into a sorted subarray of 2*width elements.
    stage:
    for (int width = 1; width < SIZE; width = 2 * width) {
        merge_arrays:
        for (int i1 = 0; i1 < SIZE; i1 = i1 + 2 * width) {
            // Try to merge two sorted subarrays:
            // A[i1..i1+width-1] and A[i1+width..i1+2*width-1] to temp[i1..2*width-1]
            int i2 = i1 + width;
            int i3 = i1 + 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            merge(A, i1, i2, i3, temp);
        }

        // Copy temp[] back to A[] for next iteration
        copy:
        for(int i = 0; i < SIZE; i++) {
            A[i] = temp[i];
        }
    }
}
```

```
// subarray1 is in[ii..i2-1]; subarray2 is in[i2..i3-1]
// sorted merge is stored in out[i1..i3-1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // Foreach element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
#pragma HLS pipeline II=1
        DTYPE t1 = in[f1];
        DTYPE t2 = in[f2];
        // Select the smallest available element.
        if((f1 < i2 && t1 <= t2) || f2 == i3) {
            out[index] = t1;
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = t2;
            f2++;
        }
    }
}
```

So, what I did it here is basically this is my merge sort, this is my merge function and you remember that this is the merge call happening. I just try to bring this into here right this loop into here.

(Refer Slide Time: 32:22)



```
void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
    // Each time through the loop, we try to merge sorted subarrays of width elements
    // into a sorted subarray of 2*width elements.
    stage:
    for (int width = 1; width < SIZE; width = 2 * width) {
        merge_arrays:
        for (int i1 = 0; i1 < SIZE; i1 = i1 + 2 * width) {
            // Try to merge two sorted subarrays:
            // A[i1..i1+width-1] and A[i1+width..i1+2*width-1] to temp[i1..2*width-1]
            int i2 = i1 + width;
            int i3 = i1 + 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            merge(A, i1, i2, i3, temp);
        }

        // Copy temp[] back to A[] for next iteration
        copy:
        for(int i = 0; i < SIZE; i++) {
            A[i] = temp[i];
        }
    }
}
```

```
void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
  stage:
    for (int width = 1; width < SIZE; width = 2 * width) {
        int f1 = 0;
        int f2 = width;
        int i2 = width;
        int i3 = 2*width;
        if(i2 >= SIZE) i2 = SIZE;
        if(i3 >= SIZE) i3 = SIZE;
    merge_arrays:
        for (int i = 0; i < SIZE; i++) {
#pragma HLS pipeline II=1
            DTYPE t1 = A[f1];
            DTYPE t2 = A[f2];
            if((f1 < i2 && t1 <= t2) || f2 == i3) {
                temp[i] = t1;
                f1++;
            } else {
                assert(f2 < i3);
                temp[i] = t2;
                f2++;
            }
            if(f1 == i2 && f2 == i3) {
                f1 = i3;
                i2 += 2*width;
                i3 += 2*width;
                if(i2 >= SIZE) i2 = SIZE;
                if(i3 >= SIZE) i3 = SIZE;
                f2 = i2;
            }
        }
    copy:
        for(int i = 0; i < SIZE; i++) {
#pragma HLS pipeline II=1
            A[i] = temp[i];
        }
    }
}
```

And this result in to my this code ok. So, if you remember here, so, I am just replacing the body of the merging two sorted array here and this transform into this code ok, but this is very interesting. It is not only that. So, I did something more here. So, you just try to notice it. So, this is my that stage step right; so, the array size merging. So, that remain the same, but the next thing is very interesting. Here you see here the earlier the loop was going on from 0 to SIZE and it was increasing by ops width.

Now, what I did? I just make it 0 to SIZE and I am increasing by i plus plus. Earlier it was it was increasing thus 2 times of the width because I just take two array, a smaller array I merge them and then I go to the next block right. So, basically you first you consider these two array of size width then you consider these two array right. So, I my I is here next I want to start this. So, this is my width this is width. So, 2 into width, I should modify right. So, what I did now?

I just merge because my that loop comes here. So, this loop basically I just take this loop and I put this loop into this loop ok in this loop and then what is happening here is very interesting. So, you have to make sure that we do the same thing and this actually is use benefit ok. So, what is happening here? You just see here is.

So, basically I have to calculate this i1, i2, i3, this three index and also merge these two array right. So, what I did? Initially I just put my initial the initialization value of. So, my f1 should be 0 and f2 is the width, width h is 1 and i2 is the width and i3 is 2 into width. So, these are the initial value then I come to this loop ok.

So, here what is happening here? Here the merging is happening, the same code whatever is there in my here right, the same thing. So, this is the copy pasted here, the merging happening here. This is basically the merge right. And because the whole thing is happening in the same array, so, the idea is that you just go this way.

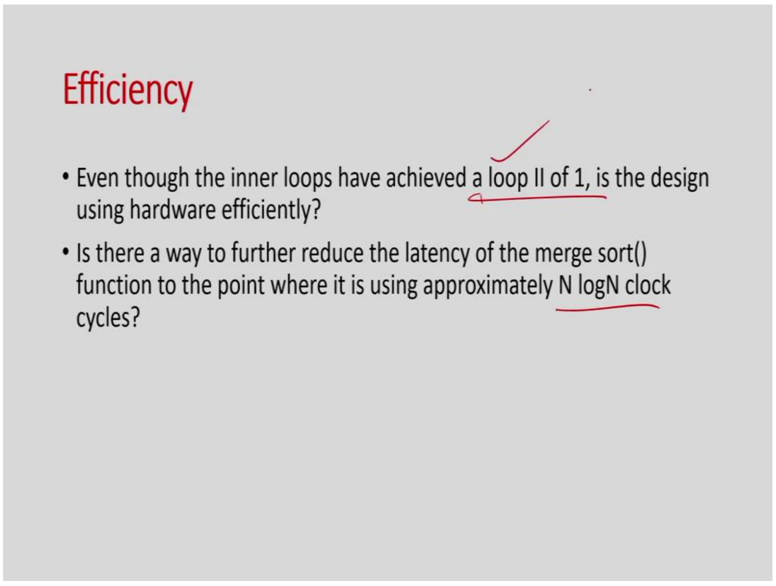
So, you just take this every iteration because finally, you have to go through the whole array of size and you just try to merge these two array and you after having one iteration you just check that you increase f1 or f2 and you just check whether you actually reach the next one or not right. So, if you completed that part or not. If you have completed you basically go to the next block.

So, the next block going things is happening here what was calculating earlier. So, basically that merging and this next block shifting things are happening in the same loop. So, basically you can understand that with this you can actually first merge these two then you will take these two array and you merge these two then you will take these two array you can merge these two because all things happening in together right.

So, you can understand the benefit that I am going to get. Earlier it was a three nested loop, I have now two nested loop because I merge this last two loops into one loop right. So, the operations of these two things into one ok and the biggest thing that I can see here is my this loop is now fixed size right. And I can apply this pipeline at this level not the innermost level. Earlier I was applying the pipeline at this level third nested loop, but I am actually applying in the second nest loop now, right.

So, and also since this loop is a fixed size is fixed for a program, so, I can actually get the all performance estimate of this ok. And also you can see here that memory read problem I have resolved and everything is happening here. So, this again will achieve my pi equal to this loop pipelining initiation integral equal to 1 and this is the kind of best restructure code ok.

(Refer Slide Time: 36:25)



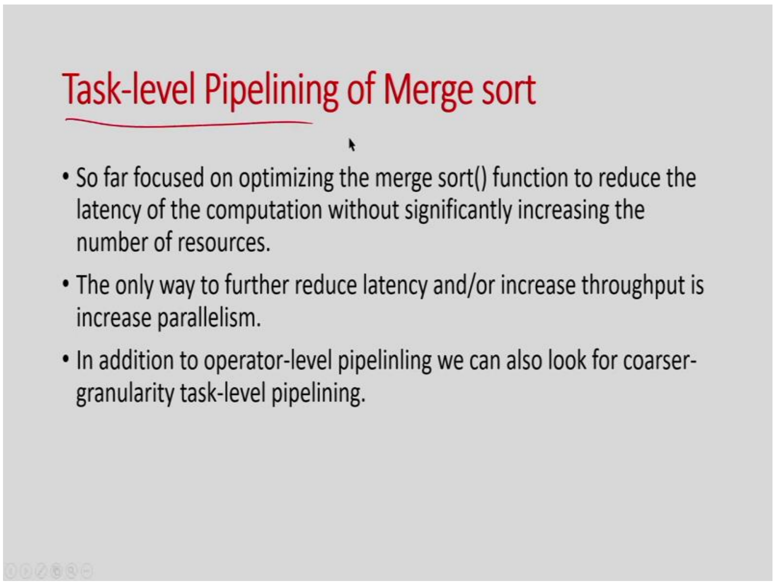
Efficiency

- Even though the inner loops have achieved a loop II of 1, is the design using hardware efficiently?
- Is there a way to further reduce the latency of the merge sort() function to the point where it is using approximately N logN clock cycles?

So, now we have pi initiation interval to 1. So, you can understand that from the initial code how I reach to this. And this is the same merge sort, but this is something if I synthesize into hardware you can actually compare this in the results of this with the best level what I asked you earlier to do. You compare the results ok. So, this is what you reach one level up, after that you cannot go further because everything is more or less control ok.

So, now, the question is it running in because you know that complexity of merge sort is n log n right, n is the size of the array. So, does it taking n log n clocks or is taking more clocks right, is it possible to improve the performance further right? So, can I reach this N logN clock? So, if you run this you will see that it does it will take more than N logN clocks, ok. So, can I improve the performance further? But as I already mentioned that this code is mostly the most efficient version of the code right. So, what next can be done?

(Refer Slide Time: 37:32)



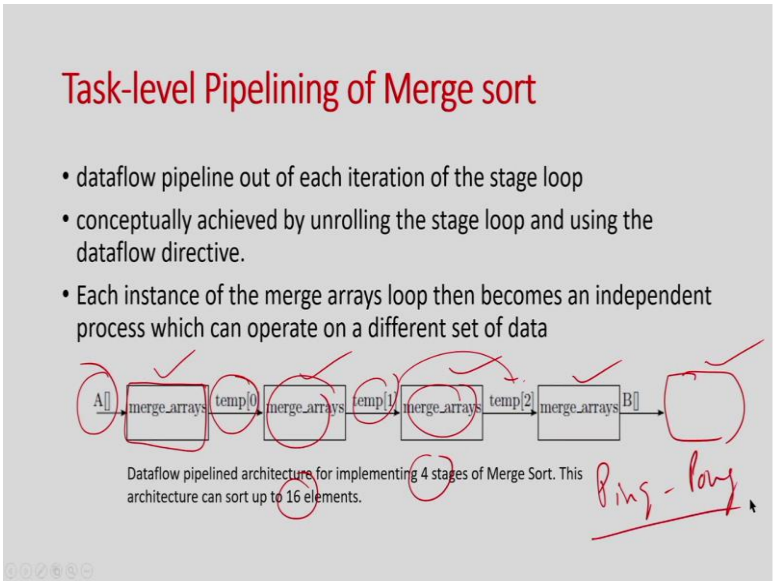
Task-level Pipelining of Merge sort

- So far focused on optimizing the merge sort() function to reduce the latency of the computation without significantly increasing the number of resources.
- The only way to further reduce latency and/or increase throughput is increase parallelism.
- In addition to operator-level pipelinling we can also look for coarser-granularity task-level pipelining.

The next is the task level pipelining. So, we have already a sequential version of the code and we make the code most efficient one right, where there is no further efficiency improvement is possible. So, the next thing is something is the parallelise the execution right. So, that is what is the task level pioneering and one of the most powerful that the most powerful optimization is high level synthesis.

So, idea here is can I parallelize the execution of the loop right. So, you can understand here this. So, but basically this there are operation the stage 1 operation where we actually merge by 1. So, this is one thing then you do merge by 2 then you merge by 4 and so on. So, can I do this operation merge by 1, merge by 2, merge by 4 things parallel right? So, that something if you parallelize that part then merge by 1 operations, merge 2 operations or merge by 4 operation parallel then it might give further benefits ok.

(Refer Slide Time: 38:28)



Task-level Pipelining of Merge sort

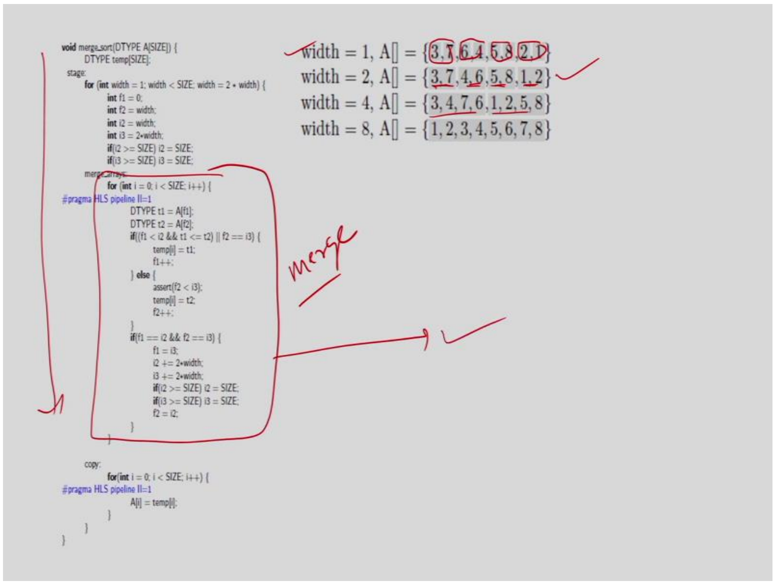
- dataflow pipeline out of each iteration of the stage loop
- conceptually achieved by unrolling the stage loop and using the dataflow directive.
- Each instance of the merge arrays loop then becomes an independent process which can operate on a different set of data

A[]
merge_arrays
temp[0]
merge_arrays
temp[1]
merge_arrays
temp[2]
merge_arrays
B[]
Ping - Pong

Dataflow pipelined architecture for implementing 4 stages of Merge Sort. This architecture can sort up to 16 elements.

And so, let us try to understand that what I can do is basically you have to create a data flow things right. So, as I mentioned I give the array A and then here this merge by one is happening.

(Refer Slide Time: 38:40)



```
void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
 stage:
    for (int width = 1; width < SIZE; width = 2 * width) {
        int f1 = 0;
        int f2 = width;
        int i2 = width;
        int i3 = 2*width;
        if(i2 >= SIZE) i2 = SIZE;
        if(i3 >= SIZE) i3 = SIZE;
    merge_arrays:
        for (int i = 0; i < SIZE; i++) {
#pragma HLS pipeline II=1
            DTYPE t1 = A[f1];
            DTYPE t2 = A[f2];
            if((f1 < i2 && t1 <= t2) || f2 == i3) {
                temp[i] = t1;
                f1++;
            } else {
                assert(f2 < i3);
                temp[i] = t2;
                f2++;
            }
            if(f1 == i2 && f2 == i3) {
                f1 = i3;
                i2 += 2*width;
                i3 += 2*width;
                if(i2 >= SIZE) i2 = SIZE;
                if(i3 >= SIZE) i3 = SIZE;
                f2 = i2;
            }
        }
    copy:
        for(int i = 0; i < SIZE; i++) {
#pragma HLS pipeline II=1
            A[i] = temp[i];
        }
    }
}
```

width = 1, A[] = {3, 7, 6, 4, 5, 8, 2, 1}
width = 2, A[] = {3, 7, 4, 6, 5, 8, 1, 2}
width = 4, A[] = {3, 4, 7, 6, 1, 2, 5, 8}
width = 8, A[] = {1, 2, 3, 4, 5, 6, 7, 8}

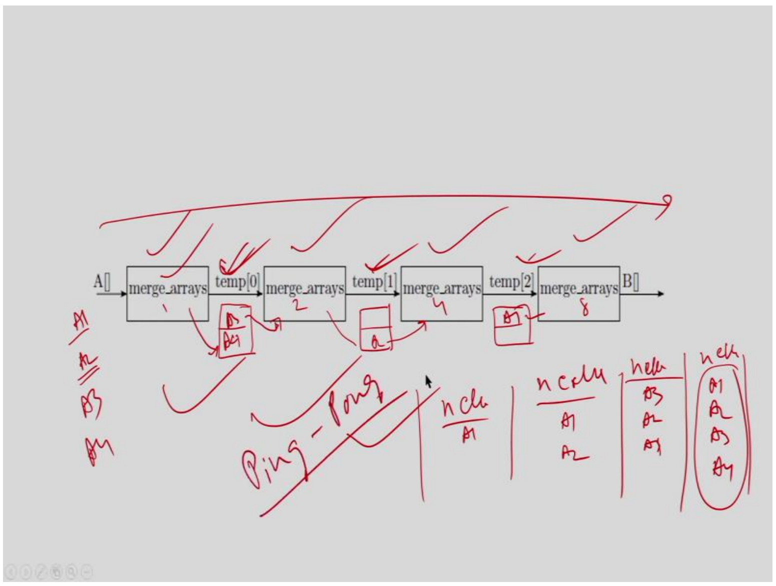
So, basically what is happening here is so, whenever I give this the first block will try to merge these two array into 1, these two array into 1 these two array. So, all these things will going to happen, but this block will do this operations and it will give you this array right where this each two elements are sorted ok.

So, that something if I can do it will be great right and then I just store this resolute to some temp 0 array. Then I call this again the same function, but it will now do the merge by 2 ok. So, that will basically create take these two and it will create this then it will take these two and it will create these two right. So, it is merge by 2.

And then I store this result into the next temperature and then I am going to do the merge by 4 I store into here then I will do the merge by 8. So, here it depends on the size because size of the array is the number of stage will differ different right. So, if I take this is for 4 stage and 16 element. If you take 32 elements then you can actually have one more stage and so on. So, the number of stage will depend on the address size ok.

So, this particular implementation will make me do a very interesting thing is that now, because this is basically a data flow. We can see here it is a data flow right and what I can do now I can run this modules this modules merge modules in parallel and I can use a ping pong buffer style right. So, I can use this ping pong. What is that? So, what I can do now, I will explain here ok.

(Refer Slide Time: 40:21)



So, what I can do is now I can give say array A1 ok and I just do all this merge by 1 operations and I will create the temp 0 right. So, I will keep a two copy of the temp here right. So, two arrays, I will just store this into this ok. So, the first say n iterations of n clock. It will take n 1 and it will sort by the all individual elements and create that merge by 1 array into temp 0.

So, this is for first n clocks ok n clocks then in next n clocks. What I am going to do? I am going to take array A2 right and this so, now, this merge array merge by 1, this is merge by 1, this is merge by 2, this is merge by 4, this is merge by 8 and so on right. So, then now it will take process A1, A2 and it will do the merge by 1 and it will create resultant things it try to store in this the other buffer right so, or even buffer.

At the same time the second block will take this data that first corresponding to A1 and it try to do the merge by 2, right and it will store this things into say into this array right. So, this means here A1 was processing now A1 and A2 both are processing right. So, this is a next n clock, what is going to happen.

So that means, after doing this, this data is already done and after next n clock after 2 n clocks that sorted by 1, A will be here. Now, next consider the next n clocks. I am going to give next array A3 and it will now start the first block try to process A3 and it will do the merge by 1 and I am going to store it here right. At the same time this array will now take the data from the even array right.

So, it will take the data from the even array. So, it will now start storing the data A3 into here. It will now start the start data from here and it will try to and it will process this A2 now because A1 is already processed. A2 is it will now going to take A2 and then it will actually process this merge by 2 and it will store here.

At the same time this A1, this third block will take A1 and it will do the merge by 4 ok and it will store here right A1. So, here A3, A2 and A1, three array is processing in parallel. So, three set of data is processed parallel next clock you assume. So, now, what is going to happen?

This block will take this A1 and do the merge by 8 and it will give you the actual output right. So, it will do processing A1. So, this particular array now it will take data from A2 because this is already done. So, this is already processed, this is already processed. So, now, this will take this A2 and do merge by 4. So, A2 is also getting processed.

So, this will now take data from A3 because this is already processed, so and it will do the merge by 2. And I will at the same time I will put the next array A4 and it will now store this A4 into this one right here this A3 is there. So, now, at this point you can understand that all these four module actually running in parallel and they are running on different set of data right.

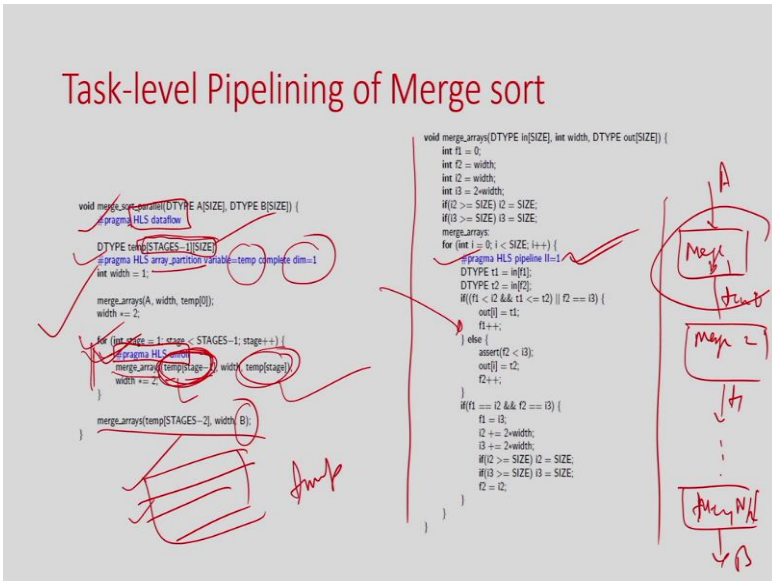
So, and basically you can understand that. So, this is the maximum anyway and this will continue. Next n clock it will now it will take A5, this will take A4, this will take A3 and this will take A2 right. So, this way things will continue. So, what is happening here? I am not able to improve the performance for one set of data right. So, that is already I have achieved, but now I am doing a task level parallelism right.

So, different array will be processed by different blocks in parallel right and I can actually get a maximum benefits ok. And this architecture will give me the best output best performance right because now because it is a task level pipelining is happened and I can actually get the best benefit. Now, the question is that what is the code modification I need to do to achieve this ok?

The first thing what we have to do is you remember this is the code I have modified right initially, this is the best efficient code I have identified. So, what we have to do? We have to put this, this is my the task right. So, this is the merge that merge from operation right. So, this is the merge operation what I just talked about this merge array.

So, this should be written as a function now again right otherwise tool will not understand this is this is individual things I am going to happen. So, I am going to first thing is that I am going to create a function out of this again. So, I am just going back to my old one, but I am going to use keep all the features here. So, I will I will talk about the difference ok. So, I will just create a function for that.

(Refer Slide Time: 45:38)



Task-level Pipelining of Merge sort

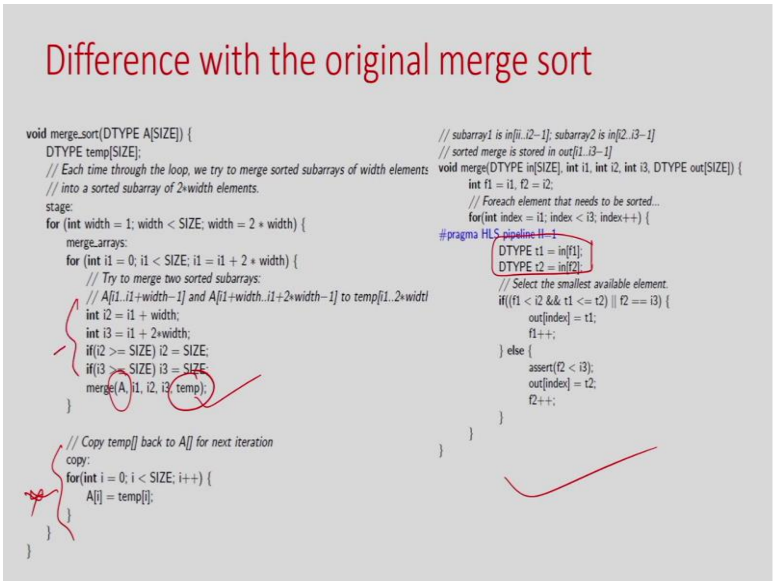
```
void merge_sort_parallel(DTYPE A[SIZE], DTYPE B[SIZE]) {
    #pragma HLS dataflow

    DTYPE temp[STAGES-1][SIZE];
    #pragma HLS array_partition variable=temp complete dim=1
    int width = 1;

    merge_arrays(A, width, temp[0]);
    width *= 2;

    for (int stage = 1; stage < STAGES-1; stage++) {
        #pragma HLS unroll
        merge_arrays(temp[stage-1], width, temp[stage]);
        width *= 2;
    }

    merge_arrays(temp[STAGES-2], width, B);
}
```

```
void merge_arrays(DTYPE in[SIZE], int width, DTYPE out[SIZE]) {
    int f1 = 0;
    int f2 = width;
    int i2 = width;
    int i3 = 2*width;
    if(i2 >= SIZE) i2 = SIZE;
    if(i3 >= SIZE) i3 = SIZE;
    merge_arrays:
    for (int i = 0; i < SIZE; i++) {
        #pragma HLS pipeline II=1
        DTYPE t1 = in[f1];
        DTYPE t2 = in[f2];
        if((f1 < i2 && t1 <= t2) || f2 == i3) {
            out[i] = t1;
            f1++;
        } else {
            assert(f2 < i3);
            out[i] = t2;
            f2++;
        }
        if(f1 == i2 && f2 == i3) {
            f1 = i3;
            i2 += 2*width;
            i3 += 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            f2 = i2;
        }
    }
}
```

And next thing what I have to do is the next thing so, I just create a merge sort here. So, my top level function is now become basically just calling this right for the stage array because I just bring out this part of the code into a function ok. Next thing I am doing here is you see is very important that I am actually calling earlier I am now keeping this into a different array. So, it was not temp, it is temp stage i.

So, that tool understand that this is a different array right. So, I am using actually two dimensional array now. So, that it understand it is not temp. It is not temp, temp, temp; it is temp 0, temp 1, temp 2 right. So, that is what is the difference. So, that tool can understand these are the two different array and they are not all writing into the same array. So that this kind of data flow things can be created.

(Refer Slide Time: 46:27)



Difference with the original merge sort

```
void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
    // Each time through the loop, we try to merge sorted subarrays of width elements
    // into a sorted subarray of 2*width elements.
    stage:
    for (int width = 1; width < SIZE; width = 2 * width) {
        merge_arrays:
        for (int i1 = 0; i1 < SIZE; i1 = i1 + 2 * width) {
            // Try to merge two sorted subarrays:
            // A[i1..i1+width-1] and A[i1+width..i1+2*width-1] to temp[i1..2*widtl
            int i2 = i1 + width;
            int i3 = i1 + 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            merge(A, i1, i2, i3, temp);
        }

        // Copy temp[] back to A[] for next iteration
        copy:
        for(int i = 0; i < SIZE; i++) {
            A[i] = temp[i];
        }
    }
}
```

```
// subarray1 is in[ii..i2-1]; subarray2 is in[i2..i3-1]
// sorted merge is stored in out[i1..i3-1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
        int f1 = i1, f2 = i2;
        // Foreach element that needs to be sorted...
        for(int index = i1; index < i3; index++) {
#pragma HLS pipeline II=1
                DTYPE t1 = in[f1];
                DTYPE t2 = in[f2];
                // Select the smallest available element.
                if((f1 < i2 && t1 <= t2) || f2 == i3) {
                        out[index] = t1;
                        f1++;
                } else {
                        assert(f2 < i3);
                        out[index] = t2;
                        f2++;
                }
        }
}
```

So, I just if you just remember with the original code, so, earlier code it was just a single temp right, I just compare here. Now, here it is temp 0, temp stage 1 right. So, this actually makes sure that these are different different array and it can actually create that ping pong buffer right and this is for the last level. So, just to make sure that this is not I am using temp stage, I am going to use temp basically B because that is my final output.

So, this is the second thing I have did. So, I just separate out this and the third thing that I did is basically earlier you see here this A was my array and it is basically copied back right. So, that I am not doing back to copied back rather I am actually using two different stages that I so, this input array stage is minus 1 and this stage is 0. So, the data flow is getting created.

So, that this is the three difference from my original code. This is my original code I just kept it here. So, here you can understand that I just and this is my modified code. So, as I mentioned that stage things I separate out, input output I separate out, so, that temp i happen, this copies I am not doing because I just keep it to different array. And the third thing I did is basically this modification is also there and all this calculation is also now put into this function right. So, this calculation is no more there here right. So, these are all come to here.

So, I with all these modification what are the things I can do? I can still pipeline this loop because I can still pipeline my individual block which will improve the performance.

This is the fine grain pipelining and the other thing is the coarse grain pipelining where I actually do a task level pipelining ok. So, and then I basically what is happening here? I am actually unrolling this loop right.

So, this is what I am doing here that I actually do a do unroll of the top level loop then only it will create this stages. So, if I just do this unroll then what is going to happen? It will create this stages right. So, this is the first unrolled part will create the stage 0 merge by 1 then it will create. So, basically this is my merge by 1, merge by 1 and it will create temp 0 right and here I am getting A.

Then if I do the unroll, so, it will they go do this merge by 2 and it will create temp 1. It will create this finally, it will create it will give B and this is merge by n by 2 right. So, this unroll I am not doing manually right. So, ideally what is happening? To create this structure I have to unroll the outer loop right. So, this is what to do, but I am not going to unroll manually. I just apply the pragma unroll.

It will actually unroll and it will create this structure which is nothing but this it will create this structure and all this array I just separated out. So, I just apply unroll here and also you can see here the temp is a two dimensional array right. So, temp is now STAGE into SIZE. So, I just do a partition. So, although this is a single array, so, it is a two dimensional array I just say [FL] you partition by dimension one. So that means, each row will become one different different array.

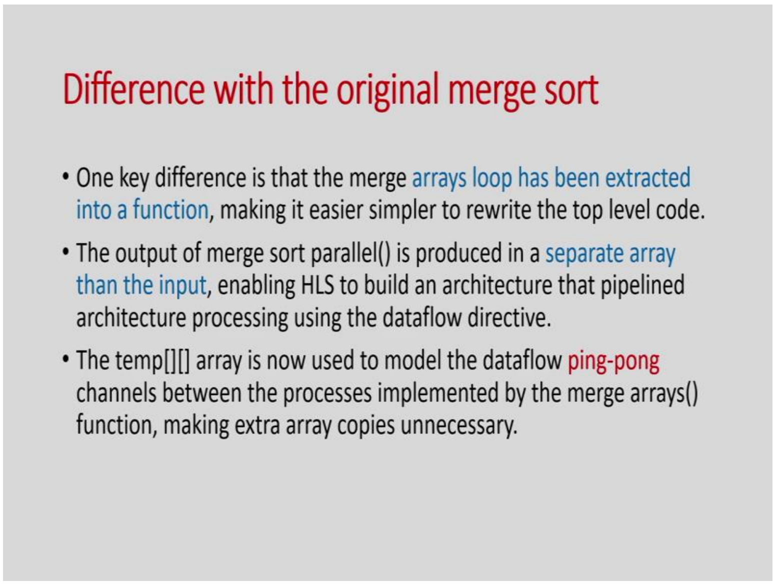
So, that is why it is able to do this. This is temp 0, this is temp 1, this is temp 2 and so on right. So, with all this, so, what I did? I just bring this code here, I apply pipeline here, so, that I can get the maximum performance benefit of this loop. I have to unroll this loop because otherwise this stages will not be created because it is a rolled implement is the same loop same function. But I have to create four copies of that loop. So, I will apply unroll here.

And I also make sure that this particular temp I just already separated them out, it is not the same array anymore, but I have to partition that array. Unless I partition the array it will not create because it is a two dimensional array, but here you can see it is a one dimensional array. So, two; so, I just have to say [FL] I am going to partition in the dimension one. So, that it get different different arrays.

And finally, I have to say that you apply HLS data flow right. So, if I just say [FL] if you apply data flow then only this kind of parallel execution that I just talked about will be executed and it will basically create a ping pong buffer the way I just explained here. So, you can understand this is what is the best possible design of merge sort right.

So, if I just write this version of the code and give the high level synthesis tool you apply data flow pragma, you partition this temp array into dimension wise, you apply unroll here, you apply pipeline here and this will give you the best efficient most efficient design form also ok.

(Refer Slide Time: 51:18)

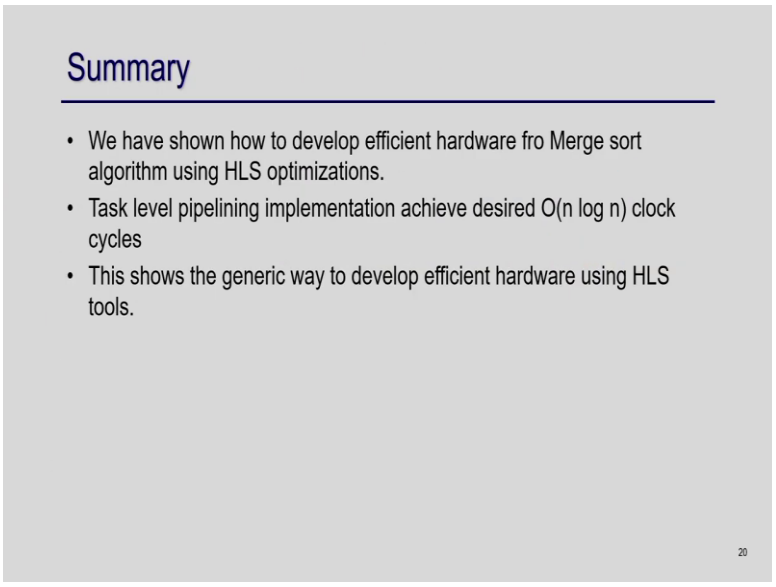


Difference with the original merge sort

- One key difference is that the merge arrays loop has been extracted into a function, making it easier simpler to rewrite the top level code.
- The output of merge sort parallel() is produced in a separate array than the input, enabling HLS to build an architecture that pipelined architecture processing using the dataflow directive.
- The temp[][] array is now used to model the dataflow ping-pong channels between the processes implemented by the merge arrays() function, making extra array copies unnecessary.

So, this is what and I will strongly recommend you just run this particular code and the best level code right and see what is the difference right. And if you can also try to apply the same pragmas whatever I just talked about here in the top level code. You try to apply this unroll here, you try to apply unroll here, you try to apply pipeline here, you try to apply HLS data flow here and you will see that you will not able to do see to say I am not I will not able to apply HLS data flow because of this dependencies and there is no data flow things are there right. But if you apply here you can understand that this will give you efficient order ok.

(Refer Slide Time: 51:53)


Summary

- We have shown how to develop efficient hardware fro Merge sort algorithm using HLS optimizations.
- Task level pipelining implementation achieve desired O(n log n) clock cycles
- This shows the generic way to develop efficient hardware using HLS tools.

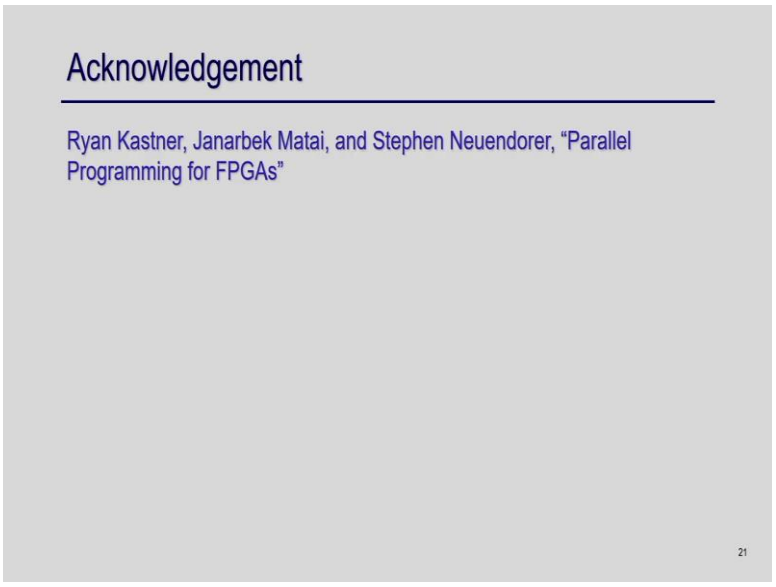
20

So, with this I summarize today's class that we take a example of merge sort and I just say step by step how a particular designer will go ahead, how either you try to apply the optimization of the tool or some manual modification of the code you will do to achieve the best efficient most efficient design ok.

Although I have taken the merge sort example here you can the same approach should be applicable for any kind of design right. So, you have to modify the design manually certain cases because if certain things actually create bottleneck for applying of some optimizations.

And you have to know very clearly which optimization should be applied where and what is the impact we are going to get right. So, the same process whatever I discussed today's class going to be applied if you try to develop a efficient hardware for a (Refer Time: 52:44) a design ok.

(Refer Slide Time: 52:47)



Acknowledgement

Ryan Kastner, Janarbek Matai, and Stephen Neuendorer, “Parallel Programming for FPGAs”

21

So, I just take all the examples from this book and I will strongly recommend you to follow this because this book has I think six or seven such examples where it is gradually shows how to get a most efficient design from a given specification C ok.

Thank you.