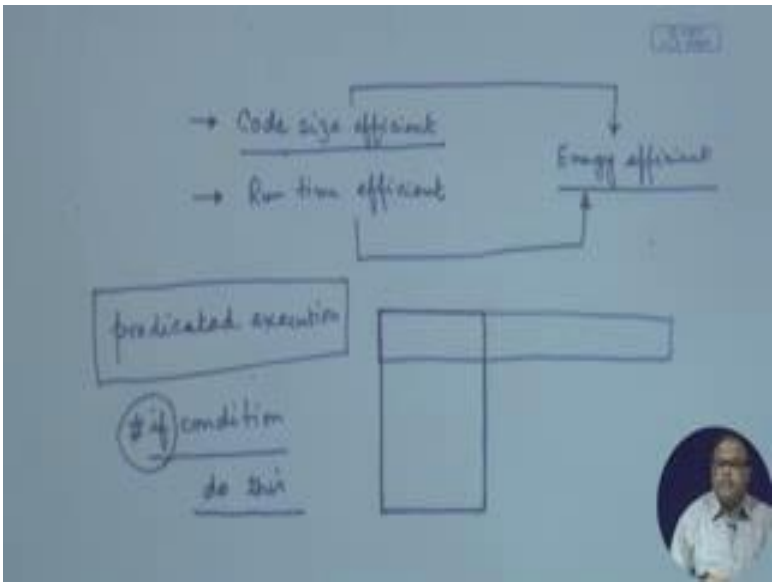
Embedded Systems Design
Prof. Anupam Basu
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 24
Code Efficiency

We were in the discussion of how we can make an embedded system power efficient more power efficient.

(Refer Slide Time: 00:32)



But besides power there are other efficiencies which we are interested in: one is it should be code size efficient and it should be also runtime efficient alright, besides being energy efficient. Now if I write down energy efficiency on the other side it should also be energy efficient. Now we have seen some measures of how to make it energy efficient by dynamic voltage scaling and dynamic power management and other measures right, some parallelisms. Now this parallelism will also make it runtime efficient and the run time efficiency will also make it energy efficient because energy is an integral of power over time.

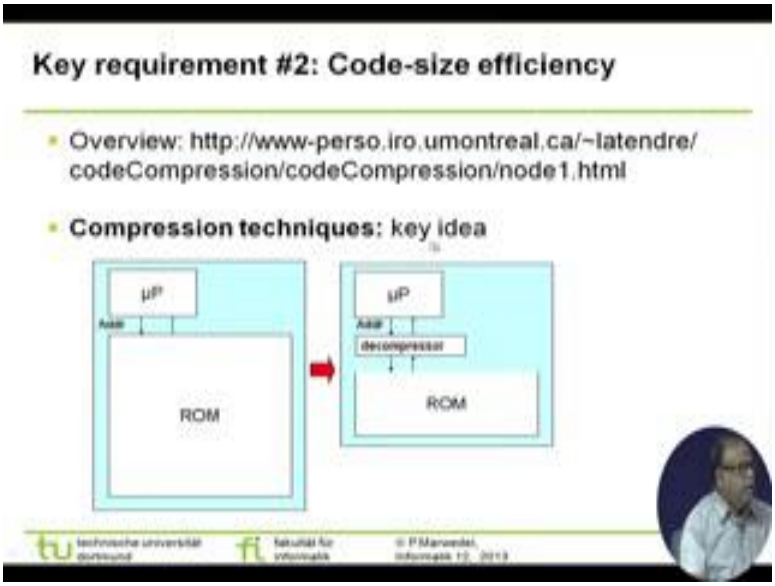
Similarly, there are independent reasons besides power; there are independent reasons of making an embedded software code size efficient. The reason is that if the code size is less then we need less memory to store it, so the area consequently will come down. But along with that as the memory increases the power also increases and more memory

accesses will be required. And if we have a small piece of code lengthwise then also it will run in less time. So therefore, code size efficiency also has got relationship with energy efficiency.

Now, but when we look at this code size efficiency; that means if I have a huge say- wide word length and I want to make some writings on that then there will be obviously more number of switching's, because for each 1 to 0 there will be a switch. So, if I can make it narrow that is also advantageous. And over the time if I make the size less also this size is less the number of words in a program that will be accessed, then also we will need less number of memory accesses. Each memory access consumes power, because I have to float the address on the address bus get the data on the data bus all those things will come into play.

So, one of the technique of code size efficiency is being shown here.

(Refer Slide Time: 03:37)

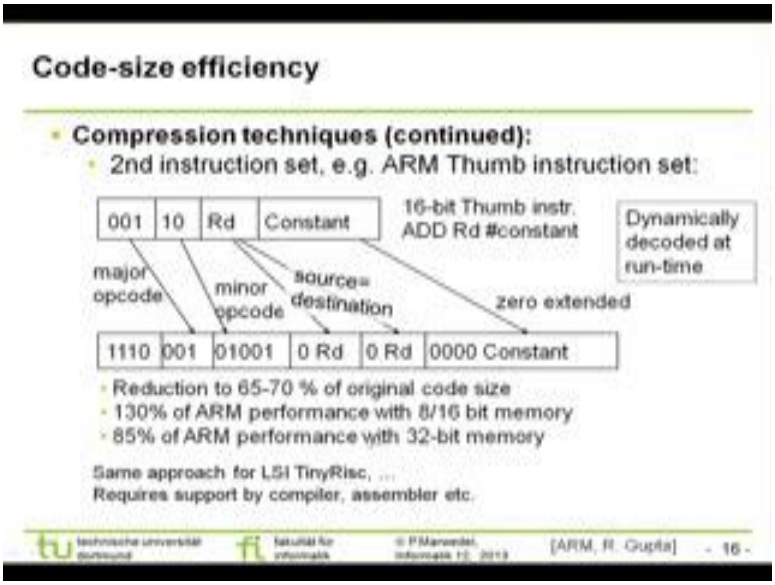


It is a compression technique; how we can we do the code compression. As is being shown here we have got the processor which is sending the address and getting back the data from the memory. Now, our objective is that we will have a compressor and decompressor which will send the address here and the address will be compressed, the code will be compressed here. And when we get it we will get it decompressed. The best way of one of the good examples of showing it is; if I can keep the codes compressed, if

I can compress the code obviously my ROM size becomes less. As soon as my ROM size becomes less my address width becomes less.

So, I can, but now if the microprocessor does not allow such compressions then once I fetch the compressed code then I have to decompress it and put it in the microprocessor itself.

(Refer Slide Time: 04:49)



A very nice example of this is in the ARM instruction set which I am repeatedly encouraging you to read, because it has got this very nice feature that is the thumb mode of operation; thumb mode of operation. In the thumb mode of operation we actually work with 16-bit instruction, whereas normally in non thumb or the expanded mode ARM can work with the 32-bit instruction. Now let us see that what is there in this 32-bit instruction. It is an example; and in this 32-bit operation here is the opcode part. This is the minor opcode, let us not consider that for the time being. We have got an opcode part.

Now there is a provision of predicated execution let me write it down; predicated execution. What is a predicate? A predicate is a sort of expressing a condition; for example, it rains is a predicate, its hot is a predicate. So some situation, now we can specify something like that if the condition; if a particular condition holds do this. Similarly we will see that we will come back to this hash if later on, but if some condition do this. So, here there is some condition some predicate that if this predicate

holds then do this. So, that is a conditional operation; conditional opcode that feature is there in thumb.

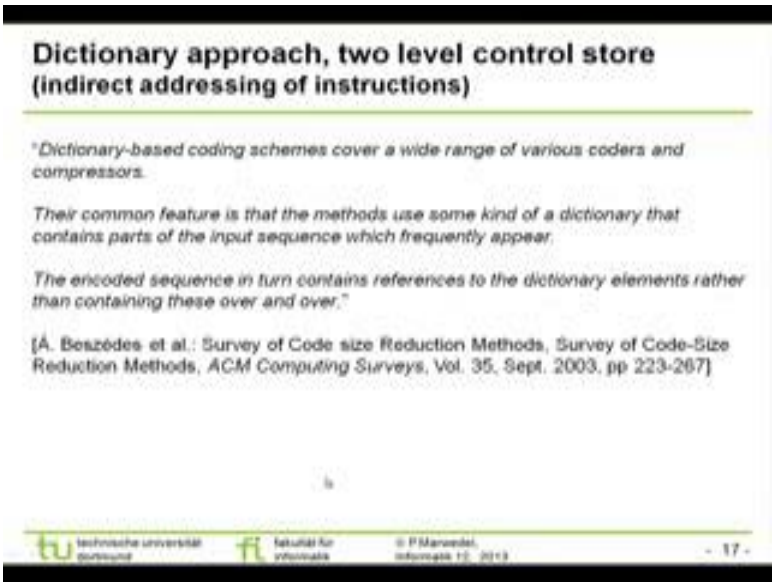
Then you can have some minor opcode which is long, then you have got the registers from where source and destination and the large width of constants. Now most of the time we do not need this; so it has been seen in that mostly we can manage with just the major opcode and a smaller minor opcode may be consider this major minor do not go into the separation say a minor not code. Therefore, the opcode field becomes much less and here I am coming with a reduced instruction set, and the source and destinations are fine but the constants are much reduced; the constants we need not deal with such big constants. Thereby we can have 16-bit thumb instruction. For example, add to some register some constant.

Now, if we are lucky then for a particular task we may be able to have all these instructions compressed to 16-bit instructions. In that case we would get 50 percent reduction of space. In actual practice it has been found that we can have around 65 to 70 percent of original code size. Now another point is ARM performance with 8 to 16-bit, memory if I consider smaller memory is around 130 percent it goes up, whereas with 32-bit larger memory its less. Therefore, if I can now the compiler; now the compiler can produce the codes in this thumb mode. In ARM as you study ARM you will see that there are ways of setting the mode to the thumb mode or the normal mode. If you have it in the thumb mode then it will take 16-bit instruction, you can also switch from the thumb mode to the extended mode. So, thereby using this I can reduce the size of the memory and the code size will be made more efficient.

Student: (Refer Time: 10:01).

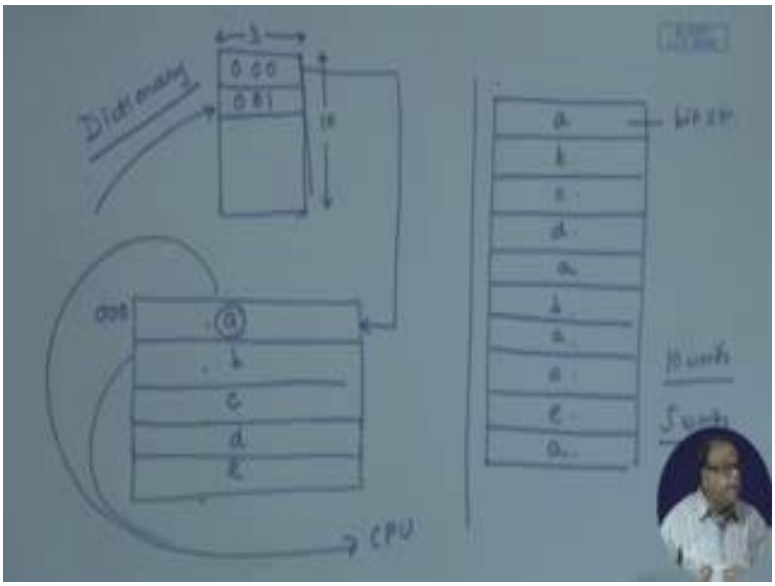
That is I mean when you write the program you will have to have those conditions given or the compiler can also do that we will show that later. That just like a compiler directives that we do in hash define or all those in see, there are hash if statements that you can add. But later on it has been found that it is not very much encouraged, because too much usage of such predicative statements does not lead to good readability. So, a number of besides ARM the same approach has been taken up by other processors like LSI candidates and all those. And it requires; in order to do that it requires the support of the compiler and the assembler.

(Refer Slide Time: 10:57)



Another very interesting approach is the dictionary approach.

(Refer Slide Time: 11:12)



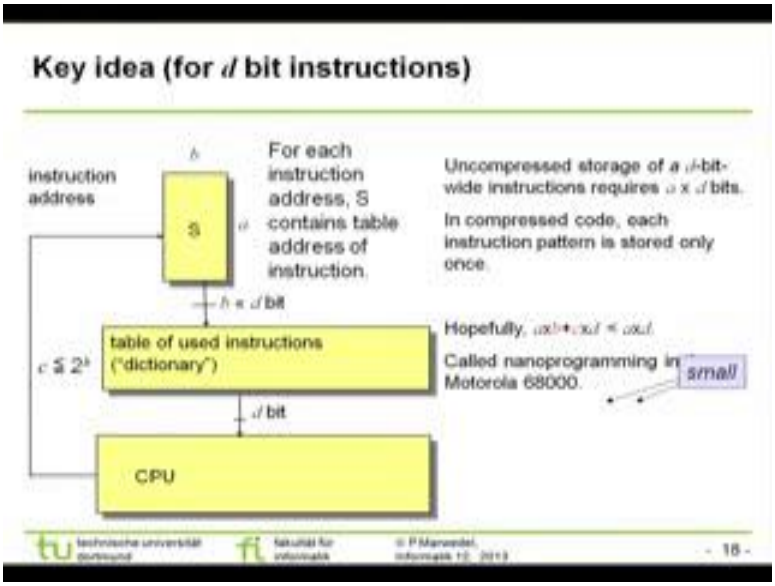
See there are; I mean if I consider a particular piece of code here and let me represent the bit strings in this code by some alphabets. So, a is a bit string say. So, my program is like a b each of them are bit strings alright c d but, I will not have distinct instructions also they may be that this same instruction has again at the machine level has appeared here, b has appeared here and may be again a has appeared here, may be c has appeared here in that way there can be repetitions, And may be another instruction e has appeared here

and a has appeared here. Therefore, actually I have got here I can see that my code size is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10- 10 words.

But actually I have got only a, b, c, d, and e 5 words- 5 distinct words distributed all over. So, if I have just the addresses of the distinct. So, I have got 10 and see I have store the only these 5 in my memory: a which is quite wide, b, c, d and e these are the instructions that are there generated by the compiler. But my sequence will be nothing but the addresses to these 5. So, how many bits do I need here? Only 3 bits; so with the 3 bit width in this example 3 bit width and ten such scenarios I can have access any one of those. So, essentially my program is a combination of these two, where I have made a reduction in this and here also I have made a reduction here.

So, this is an encoding sort of this is the dictionary. So, if I just do some calculation here; suppose I can also show it in the form of this slide, say here.

(Refer Slide Time: 14:59)



So, what is happening here for d bit instruction; I have got d bits coming to the CPU ultimately because of this any of these instructions will be executed, and they are quite wide they are d bits. Now as I come to these d bits, but here what I have is my dictionary where I have got b bits; just like here I had 3 bits and the depth was 10. So, here also I have got a addresses a instructions each of 3 bits. So why is it b bits? B bit is actually producing the address to this. So, this I have already used instructions the instructions that have been used.

So actually this is d bit, this is my what was this. So sorry, what happens is we get the b bits here which is much less than d bits that and that serves as an address and that instruction from here is chosen and fed to the CPU. Is it clear? I have got some instruction, that instruction is say these instructions again let us come here. Suppose I want to have the first instruction a; that means I am generating the address say 0 0 1 or 0 0 0.

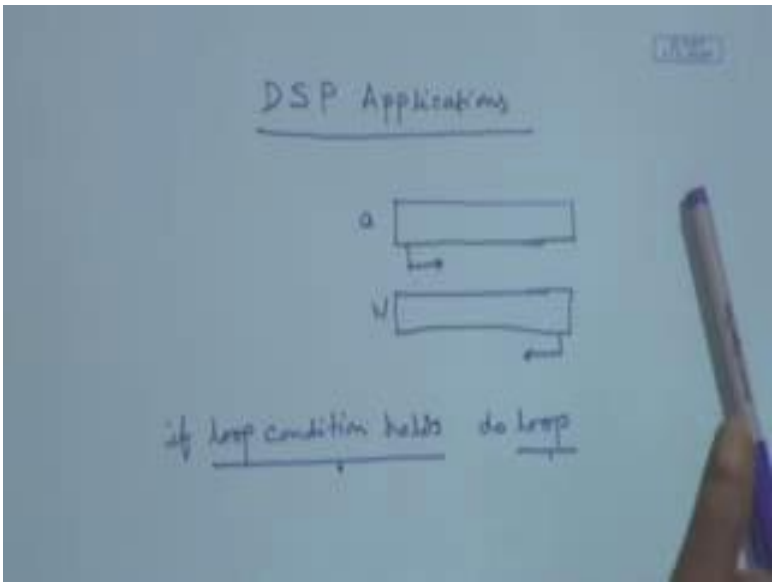
So, 0 0 0 here is pointing to this address and this address has got d bits. So, this one is going to the CPU. Next instruction may be again say 0 0 1, so this one will go and that this instruction will be fed to the CPU. Therefore, what is happening is if you look at this uncompressed storage of d bit wide instruction would have required a number of instructions were there, like here this is the total number of instructions. So, if I had stored all of them together the number of bits I require is a into d; a number into d width.

In compressed code each instruction pattern is stored only once. Therefore, actually I am needing a times b so much space and c times d because these are being stored only once, so c times d. Therefore, we hope that a times b plus c times b c times d a times b plus c times d would be much less than a times d. Thereby we can get the compression. That is the very nice way and this is used in Motorola 68000 and is called Nanoprogramming.

Student: we can also use (Refer Time: 18:44).

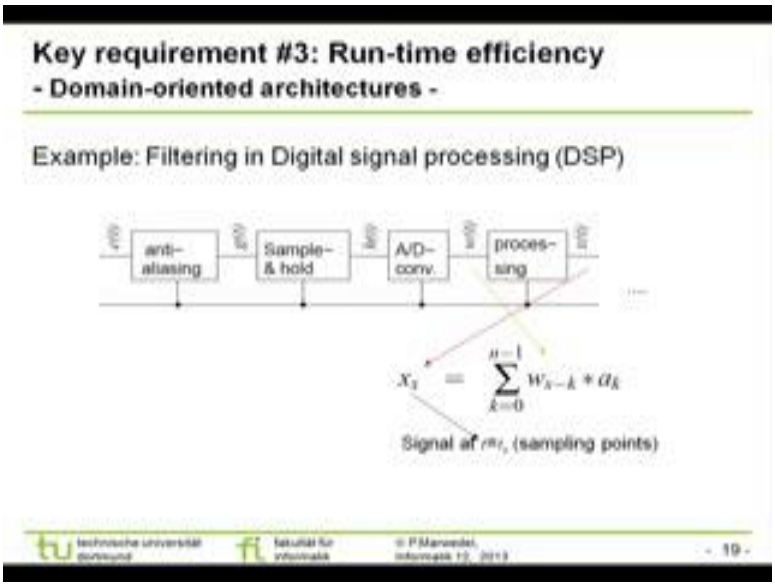
I will come to that later. So, with this now next we move to some other forms of; so these are the two approaches that we talked about for code size efficiency, now we will be talking about runtime efficiency. Now in runtime efficiency let us take for example the digital signal processing application; the DSP application.

(Refer Slide Time: 19:24)



Now, you know this figure is very familiar to us.

(Refer Slide Time: 19:44)

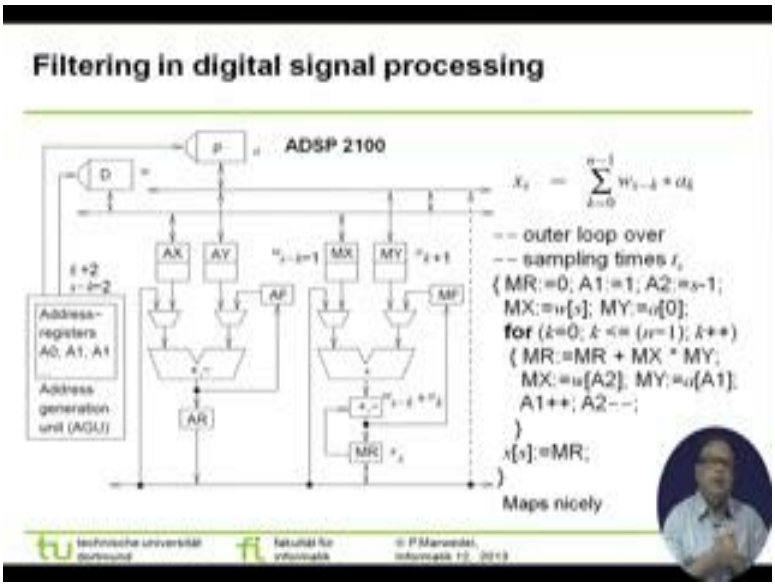


This figure is familiar to us we have encountered that number of times. So, we get a signal e t, we do the antialiasing get do sample and hold and that signal is converted A to D and it is processed. Now we are concerned about this processing today. We have already discussed about all these phases earlier. So, regarding processing we are getting some input from the A to D converter. And typically we want to generate a signal at a

particular time say sampling point is t s at some at t s point I have got some value. At this t s I have got some value and I want to find the value actual value I mean x s for that.

And how do we compute that? We compute this at this point by one very common operation in the filtering is we take all the earlier values of this sample of this variable and multiply that with some coefficient. So, it is a multiply and accumulate. What do I multiply, I multiply a current coefficient with the W values these values which are coming from the A to D converter for k equal to 0 to k n minus 1. So, I am going back all the previous samples I am taking and I am making a multiply and add with respect to that.

(Refer Slide Time: 21:54)



Now as I do this, we do this using a digital signal processor. So, here we see a typical digital signal processor which is ADSP 2100. Now let us look at this piece of code very carefully what we are trying to do before that let me quickly come to this architecture of this DSP chip we have got registers MX MY are your registers we have got MR as another register MF as another register. On the other side we have got AX and AY and there are two arrays A and W in these two memories.

Because, here we are having the W values all stored in an array and the coefficients are stored in another array. So, these two arrays are already stored in the memory. We have got an address generation unit this is very important, we will come to this a little later where the address is being generated from this unit. There is some address registers. Now

let us look at what we are doing. Again every time look at this, first of all keep in mind that we want to achieve runtime efficiency. And what do we want to achieve we want to achieve this, so let us start with this point.

In the register MX I am putting from the W array which is here the current element Sth element alright, we are starting with that. And in y we put from the array a the first element we start with that. Sth element means what here the last one that we have got right. Now we do in a loop. Now this is the initialization let us start with this we did this and then we do MR gets MR plus MX times MY. So, what is there in MY A 0 and what is there in MX W S. So, A 0 W S minus 0 k is starting from 0, so I am taking W S fine. Then and I am adding it to MR. Then this MX is getting the next the earlier value. Earlier value will get from some address register A 2 W S minus 1. Now look at these address registers A 1 and A 2 are separate here, they are here in this address bank address register bank. And then I increment A 1 and decrement A 2 clear.

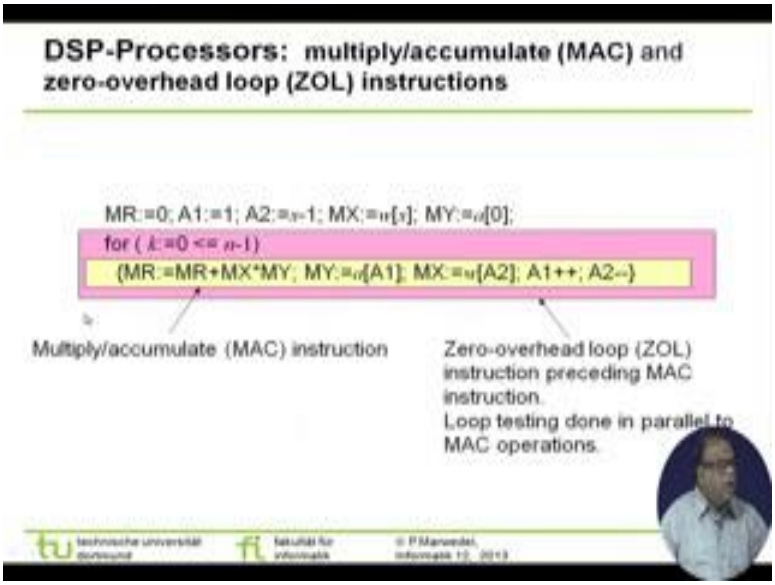
So, there are two arrays all of you are clear about this. I have got two arrays one is from W and one is for A; and W I am going this way and A I am going this way. And I initialize this sum there is an adder and what is this is an adder this is an adder also this is a multiplier sorry, this is a multiplier and this is an adder. So, all these are hardware so that the multiplication and this addition this multiplication is being taking place here and this addition is taking place in the same cycle. We are given a MAC instruction multiply accumulates. So, we are getting this and the current result is MR.

Now consequently what is happening? And we are decrementing this now look at the initialization. So, all these things are being done as if in a pipeline and we are initializing this pipeline by making this initial sum to be 0; A 1 to be 1 because I am starting with A 0 A 1 to be 0 and then coming back here. So, then 1 is there and this is S minus 1 I will started already with S. So, this thing goes on goes on and it ultimately I get the sum here.

So, the point to stress is this, I want to have since look at this architecture. The entire thing, forget about this A 1 plus plus and A 2 minus minus for the time being, let us not consider that. Besides that all the other things in this loop can be done in one cycle, because there are parallel paths coming here and this entire operation is being done in one cycle. So, from here to here I can get the entire thing done in one cycle.

All these things are taking place parallely. This MX getting W A 2 is coming through this path while this multiplication is going on. But this address generation, now I am saying I will show how later that while this computation is being done for the current values of A's A 1 and A 2 parallely the A 1 and A 2 are being modified parallely in the same cycle, because I have got a separate address generation unit.

(Refer Slide Time: 28:09)



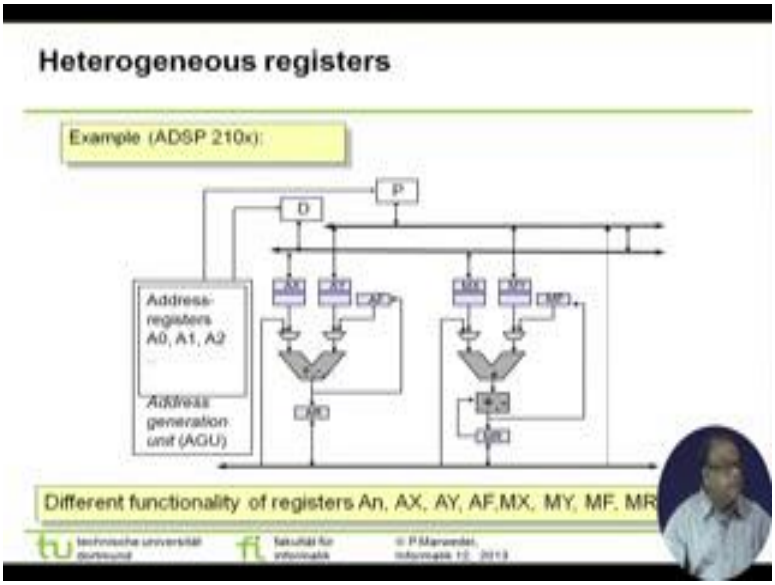
Therefore, this was actually if I write in this yellow box this was my entire body of the loop, this entire thing this entire thing initialization [FL]. Leaving aside the initialization if I look at this part is all in this. This part I will come in a moment. Now all these things can be done at the same time because I have got parallel paths. I can do this multiplication I can do this. So, this entire thing can be done in one cycle.

Now, the question arises; so what? I have got to carry out this loop and for that I have to check for this condition. Now for that again special hardware arrangements or special instructions it can be done in two different ways. This part is handled, this looping is handled by a thing called zero-overhead loop. What happens is for the; what do we do usually in our C programming, what did you do? We carry out the body of the code and then go and check the loop. Now in zero-overhead loop what we do while the body of the code is being executed parallely the loop condition is being checked that whether the loop condition will be executed.

Now here again I can refer to the predicated scenario, I can make this whole body to appear as something like; suppose I do in this way. Nnow if this one is carried out parallely along with the loop the earlier iteration of the loop and I am checking this parallely then I am not spending any extra time for this comparison. Now this can be done in hardware or this can be done using instruction prefix or by instruction prefix. So, this is an instruction and I am prefixing that. So, these are some very important novelties that came into the embedded processing in order to add to the runtime efficiency.

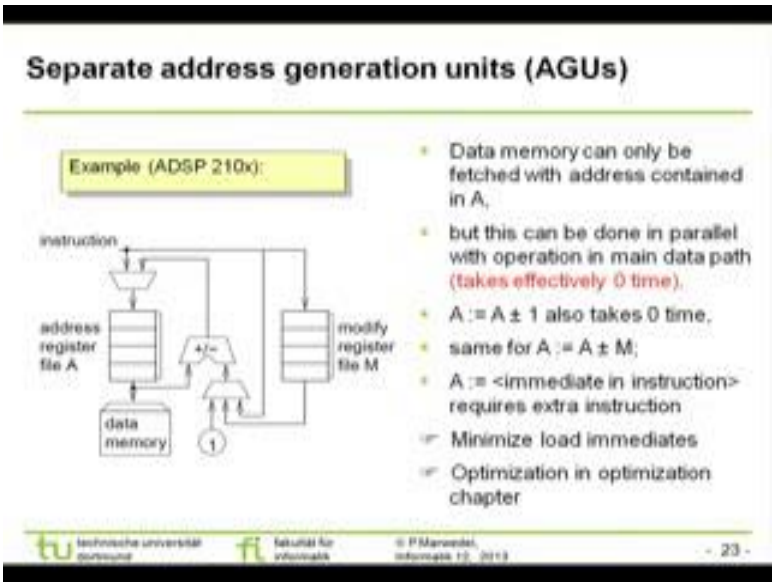
Now, the point that I have not yet addressed is this how is it that the address is generated parallely. For that if I go back here I am showing this address generation unit; the address generation unit is being shown here.

(Refer Slide Time: 31:40)



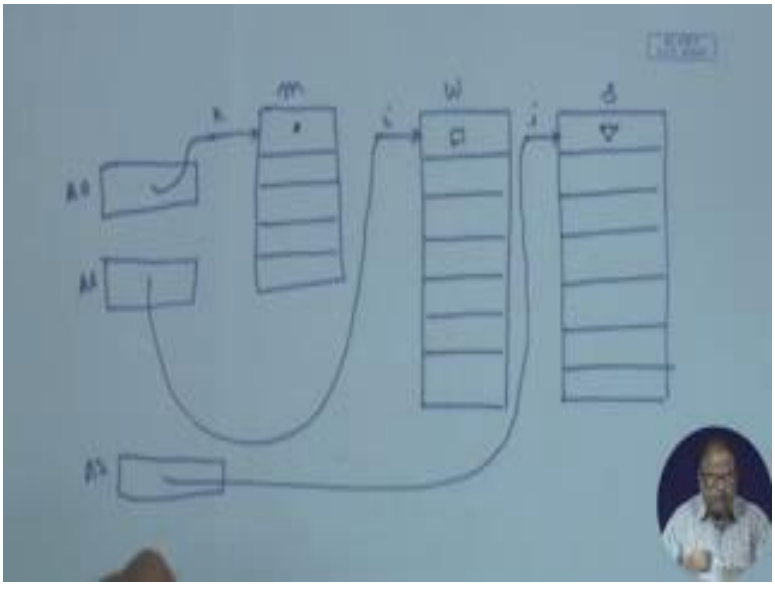
So this address generation unit is in this way.

(Refer Slide Time: 31:47)



Let me show this first in this form let us look at this. This is the address generation unit where I have got the address registers; it may be 4 5, what does it mean? That 4 or 5 different arrays I can these address generation registers are actually serving as the indices the index of the array.

(Refer Slide Time: 32:20)



There is an array; I will store all arrays, array is a very important component of any DSP processing and we will see later that we do not want linked lists to be used there.

So, I have got an array, so it may be W 1, W 2 where there are. And say this is an array W this an array s all arrays I am showing of the same size, but it may be another array of smaller size also. Suppose this is s this is m; there are three different arrays. And I am doing some operation with the indices here what we call say I, j, k. For each of these indices I am having an address register. May e A 0 is pointing to this A 1 is pointing to this, A 2 is pointing to this. Now when I carry out the operation suppose I am taking this element and multiplying with this element and may be storing in this element.

Now after that I have to increment all of them; A 0 plus plus A 1 plus plus A 2 plus plus that is taking time. Now address generation unit what it will do along with this operation it will either increment or decrement these indices so that that is automatically done. How is that done? This operation now it will be clear I have got these A 0 A 1's here they are pointing to the data memory; data memory are these wherever these arrays are. And I have got an adder with a multiplexer.

So, I have already got a 1 fed I will have some control for plus or minus auto increment or auto decrement. And I can along with the instruction I can increment this, while the instruction has been fetched the instruction is either A 1 plus plus or maybe A 1 plus m. If it be A 1 plus m then I will have some modified registers, you mean do you understand no; I mean I can increment it by one auto increment or I can shift it with a I mean with a offset, so what would be that offset value. That can come from a register memory or memory register that I add with a A 0 or whatever some value x so that is coming from the register.

So, either I can depend on the instruction I can either select this or I select 1. Whatever is coming I add either of them and that is coming to this point and it is automatically added. And so it takes effectively 0 time, if this can be done in parallel with the main data path; main data path was where I was doing that loop thing. A plus minus 1 also takes 0 time, A plus minus m also takes 0 time. And it minimizes the load immediate; load immediates I need not that is automatically coming. So, that is a nice way of optimizing this.

So summarizing, the DSP processors are specially tuned for such runtime efficiency not that others are not tuned but DS, but I am just showing how DSP processors are tuned that is using this address generation registers and also with heterogeneous registers, all these registers are not doing the same function. Here is an operational unit which is

doing multiply accumulate in the same cycle. So, that altogether allows us to have a very faster operation.

Now we will break for a while.