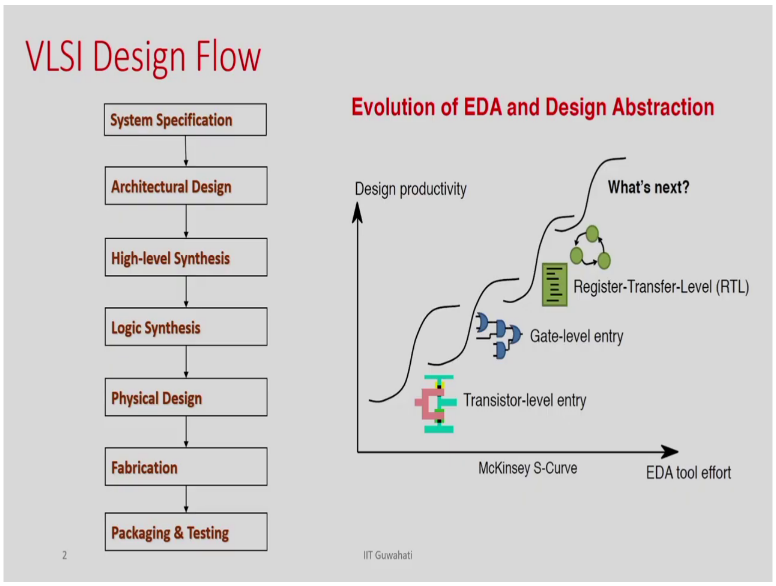
C-Based VLSI Design

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

Module - 01
Introduction to C-based VLSI Design
Lecture - 02
Introduction to C-based VLSI Design

Welcome to this course C-Based VLSI Design. In today's class, we are going to discuss the C-based VLSI design with an example.

(Refer Slide Time: 00:55)



So, in the previous class, we have discussed the overall C-based VLSI design flow, and we have already discussed that whatever the IC or integrated circuits we design it goes through different synthesis steps. Like high-level synthesis, logic synthesis, physical synthesis, then finally, fabrication and packaging, right. So, this is something is the overall flow.

And we have already seen that because of this large number of transistors present, so the designing circuit at the transistor level was not impossible. Then we go for a higher abstraction level which is the gate level. So, when you design your circuit using gate level and then use an automation tool or software that will convert that gate-level circuit into the transistor-level circuit, right.

And then, the whole design process goes further abstraction, in the further abstraction is that register transfer level design. So, in the register transfer level design, you design the circuit is more abstraction level in terms of the register transfers, and then you use the logic synthesis tool which is another kind of software that will convert those register transfer level designs automatically into the gate-level design, right. So, that was the standard of the EDA automation flow.

But with the complexity growing and more components coming, that and even designing certain things, and the register transfer level becoming difficult because of certain parameters or optimization goals like power, area, timing, and all, right. So, then the whole design process goes one level up, one more abstraction level up, which is called high-level synthesis, right.

And what this is something is very interesting because now so far whatever the design you are talking about this register transfer level or gate level or transistor level. We have a clock, you have reset, all the hardware components are there, and you have to break your head on the hardware side.

But this high-level synthesis breaks the brings the revolution because now you can design your circuit at the C or higher abstraction level like code like C, C++, Python, Simulink, MATLAB and so on, system C and so on. And then you use software which is called high-level synthesis which is something to convert that C code into this RTL.

So, this is something very interesting because you have the input which is no hardware at all, there is no clock, there is no time, it is untimed behaviour that is sequential, and this hardware is special, right, so everything is parallel. So, you have a sequential behaviour which is executing top-down manner and now you are converting those particular designs into a special design, hardware, or you bringing clock, you are bringing all other harder integrators register, RAM, ROM, everything, right.

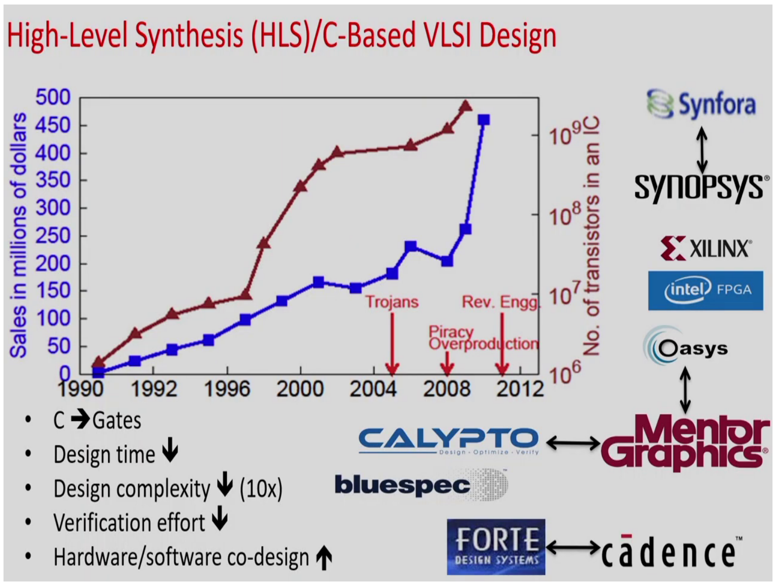
So, this is something is very interesting and that is something is the focus of this course which is that is why we are talking about this course is C-based VLSI design. That means, I am taking a C behaviour, C is a representative of a high-level behaviour, and then how we convert this particular C code into equivalent register-transfer design or hardware, ok.

So, and then from that RTL, you can again use apply logic synthesis for convert into RTL to gate level from using physical synthesis converts into this gate-level design to transistor-level and so on, and then we will get the this is the overall C-based VLSI design flow. Since this logic synthesis and physical synthesis, things are more mature, and we have well-established tools in this domain, so I am not focusing on those things in this particular course.

Rather, the focus of this course is on the first part, how we convert the C code into register transfer level design. Obviously, in the subsequent part of the course, at the later stage, we will follow we will briefly introduce what is logic synthesis, what is physical synthesis, but I am not going to go into detail about that particular topic in this course. Rather, my focus of this course will be particularly on that high-level synthesis, right.

That is why this course is named C-based VLSI design, and specifically, C to hardware conversion is the topic of interest in this course, right.

(Refer Slide Time: 05:01)

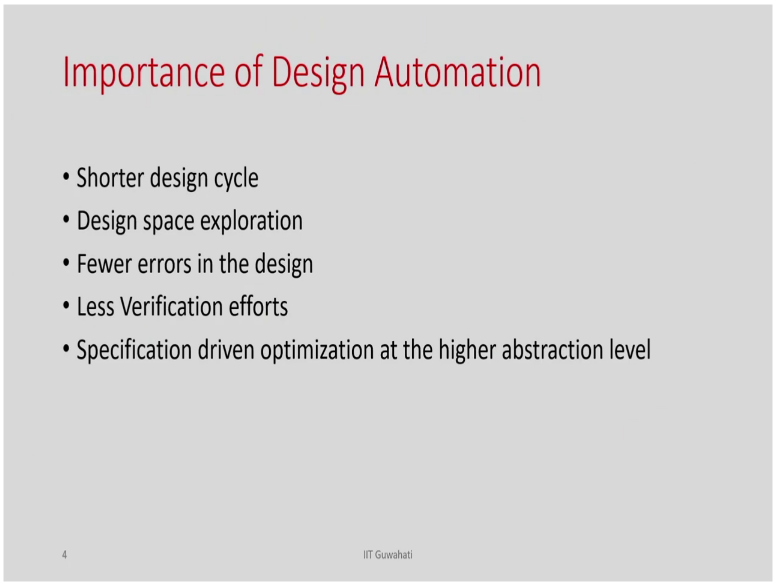


High-Level Synthesis (HLS)/C-Based VLSI Design

- C ➔ Gates
- Design time ↓
- Design complexity ↓ (10x)
- Verification effort ↓
- Hardware/software co-design ↑

So, and this is something this high-level synthesis is something that is getting a lot of interest in the design houses. If you can see all the EDA majors, and EDA vendors like Synopsys, Cadence, and Xilinx, everybody has a high-level synthesis tool, as a commercial tool which is you can buy or you can use for your development of hardware from C components, C or C++ behaviour, right.

(Refer Slide Time: 05:20)



Importance of Design Automation

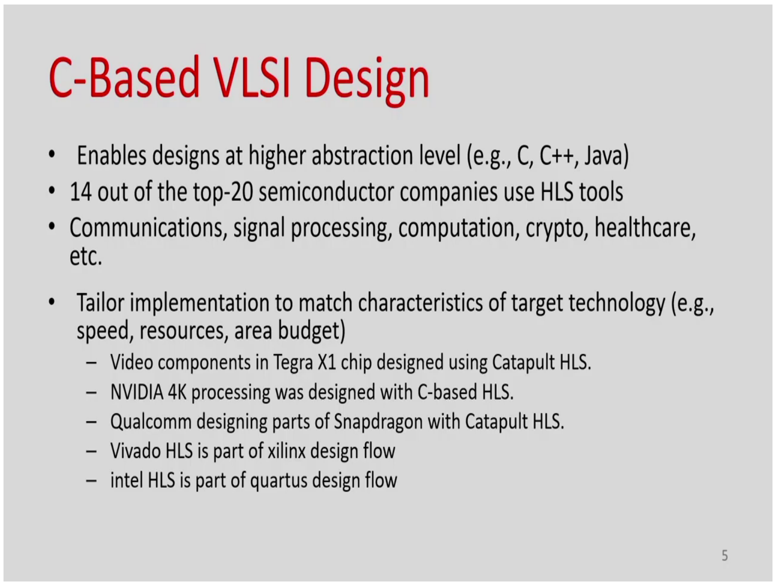
- Shorter design cycle
- Design space exploration
- Fewer errors in the design
- Less Verification efforts
- Specification driven optimization at the higher abstraction level

4 IIT Guwahati

And, obviously, we have understood the advantage of this higher level of abstraction that it reduces the overall design cycle and it enables you to design space exploration. You have the possibility of fewer errors because your design size is smaller at the C level, and you need less verification time.

Because you verify your specification at C and then you can use it, you can convert it into subsequent stages and also you can apply many optimizations at the higher level of abstraction. So, these are the unique benefits that high-level synthesis brings to the electronic design automation flow, right.

(Refer Slide Time: 05:58)



C-Based VLSI Design

- Enables designs at higher abstraction level (e.g., C, C++, Java)
- 14 out of the top-20 semiconductor companies use HLS tools
- Communications, signal processing, computation, crypto, healthcare, etc.
- Tailor implementation to match characteristics of target technology (e.g., speed, resources, area budget)
  - Video components in Tegra X1 chip designed using Catapult HLS.
  - NVIDIA 4K processing was designed with C-based HLS.
  - Qualcomm designing parts of Snapdragon with Catapult HLS.
  - Vivado HLS is part of xilinx design flow
  - intel HLS is part of quartus design flow

5

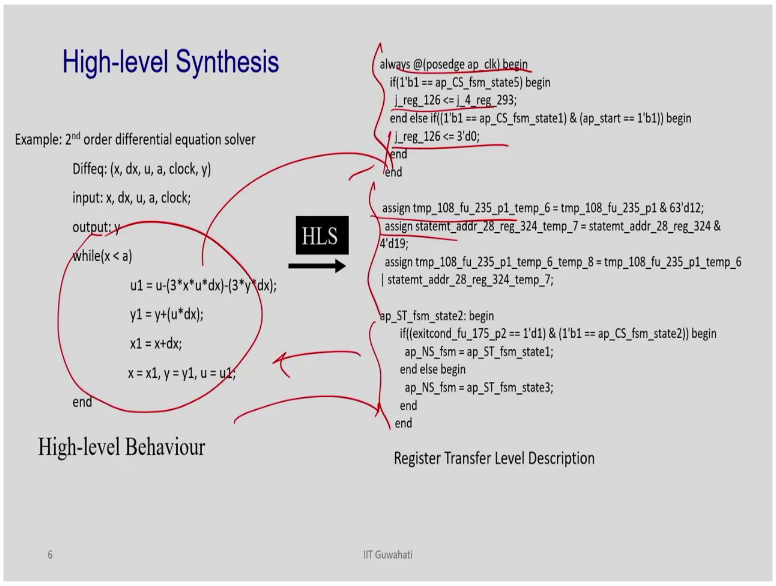
So, because of these advantages if you look into this current industry out of the top 20 semiconductor companies. They are 14 is using this VLSI, this high-level synthesis tool or they design certain components of their big resources using this high-level synthesis tool, ok. So, that is why this particular topic is so important and something that has immense potential.

Because if we can have a very robust tool high-level synthesis tool which is can convert this high-level behaviour into some efficient hardware. Then it can have a very it has huge importance in the context of this electronic design automation, right.

So, that is why this particular topic has a huge potential and is of interest to this course. And this is something that is still maturing, right. You do not have any very robust tool or we cannot which can compute still it is not maybe not compatible with handwritten RTL. Because once expert designers write something in the hand, when RTL directly that may be much faster or much more efficient than the RTL generated by a high-level synthesis tool.

But the last 20 or 25 years of research on this high-level synthesis make a lot of advancements and because of that, this huge change is coming that the 14-semiconductor industry is still using. Now we are using I have the confidence to use these high-level synthesis tools, ok.

(Refer Slide Time: 07:27)



High-level Synthesis

Example: 2nd order differential equation solver

```
Diffeq: (x, dx, u, a, clock, y)
input: x, dx, u, a, clock;
output: y
while(x < a)
        u1 = u-(3*x*u*dx)-(3*y*dx);
        y1 = y+(u*dx);
        x1 = x+dx;
        x = x1, y = y1, u = u1;
end
```

High-level Behaviour

HLS

```
always @(posedge ap_clk) begin
  if(1'b1 == ap_CS_fsm_state5) begin
   j_reg_126 <= j_4_reg_293;
  end else if((1'b1 == ap_CS_fsm_state1) & (ap_start == 1'b1)) begin
   j_reg_126 <= 3'd0;
  end
end

 assign tmp_108_fu_235_p1_temp_6 = tmp_108_fu_235_p1 & 63'd12;
 assign statemt_addr_28_reg_324_temp_7 = statemt_addr_28_reg_324 &
4'd19;
 assign tmp_108_fu_235_p1_temp_6_temp_8 = tmp_108_fu_235_p1_temp_6
| statemt_addr_28_reg_324_temp_7;

ap_ST_fsm_state2: begin
    if((exitcond_fu_175_p2 == 1'd1) & (1'b1 == ap_CS_fsm_state2)) begin
     ap_NS_fsm = ap_ST_fsm_state1;
    end else begin
     ap_NS_fsm = ap_ST_fsm_state3;
    end
  end
```

Register Transfer Level Description

6 IIT Guwahati

So, what I am going to do here in this course, in today's class specifically is understand that high-level synthesis is taking a C code and convert into a register transfer level design, but how, alright. So, we should try to go into a little bit deeper on that particular topic is, ok we have a C code which basically if you have if-else, we have for loop, you have a branch, you have this basic block operation, and so on.

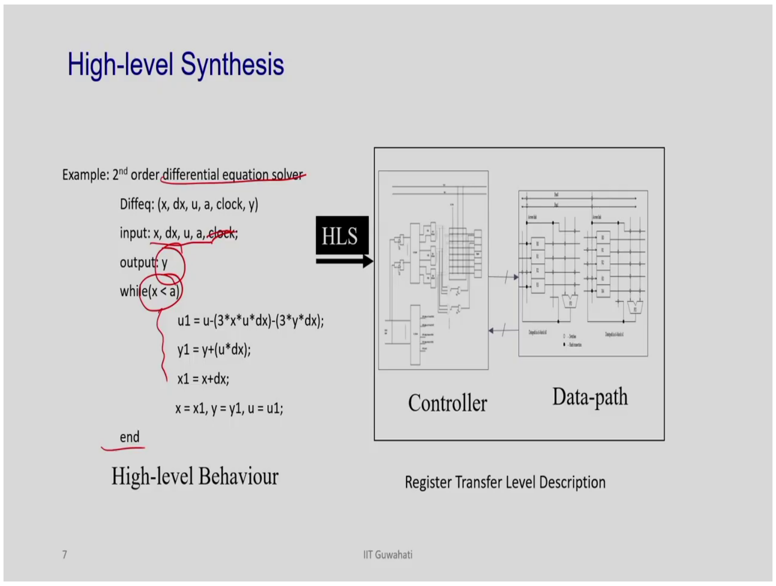
And then you have register transfer level design, where you have this clock, you have registered, you are assigning the registers, you have a controller FSM, you decide the what will be the next state, you have some data assignments between the wires. So, many things are there, right.

So, once you have that kind of RTL how we can generate such RTL from a C code because it does not have anything, it does not have any clock, it does not have any register, it does not have any control FSM, and nothing, right. So, in today's class, we will try to see in that particular context, right. So, intuitively if we try to realize the C code using hardware what will be the steps to be followed to get that, ok. So, that is the idea.

So, here we have taken a behaviour like say where we have a while loop and we are doing something and you say converting into an RTL. You can understand that here there are some registers under a certain posedge of the clock. You are assigning certain registers. Here there is a sort of wire assignments operations.

This is a part of the controller FSM. So, this is just a representative RTL. It is not the equivalent of this behaviour. But this is what the conversion is high-level synthesis does, and we should understand how it does, right.

(Refer Slide Time: 09:11)



High-level Synthesis

Example: 2nd order differential equation solver

```
Diffeq: (x, dx, u, a, clock, y)
input: x, dx, u, a, ~~clock~~;
output: y
while(x < a)
        u1 = u-(3*x*u*dx)-(3*y*dx);
        y1 = y+(u*dx);
        x1 = x+dx;
        x = x1, y = y1, u = u1;
end
```

High-level Behaviour

HLS
Controller
Data-path

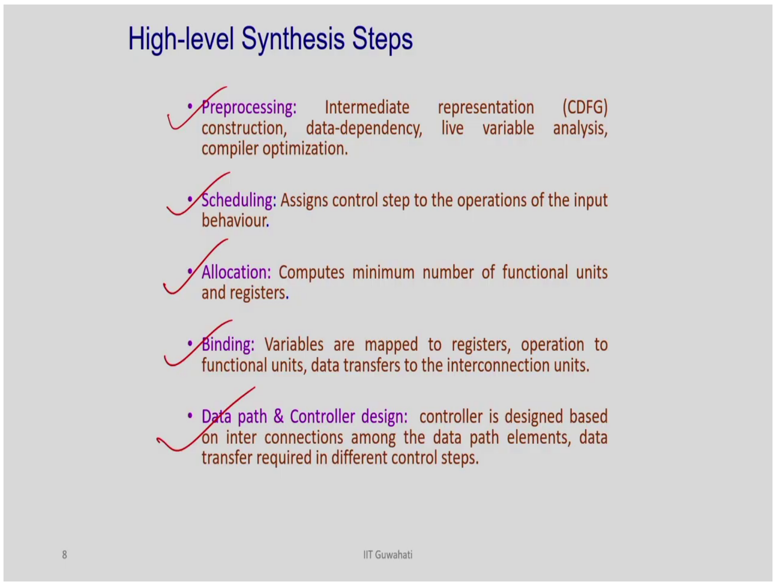
Register Transfer Level Description

7 IIT Guwahati

So, what I am going to do in today’s class is that we going to take this example which is called differential equation solver, which you take input as x, dx, u, and a, and then it basically in a loop we just calculate this u1, u1, x1 and then once your x is equal to or greater than a. It stops and then its output is y, ok. Let us understand this is something that is given to us, ok.

And I want to do the same thing in hardware, ok. So, that is something is the high-level synthesis is going to do, and let see how, if I am the high-level synthesis tool how I should do these things step by step, ok.

(Refer Slide Time: 09:53)



High-level Synthesis Steps

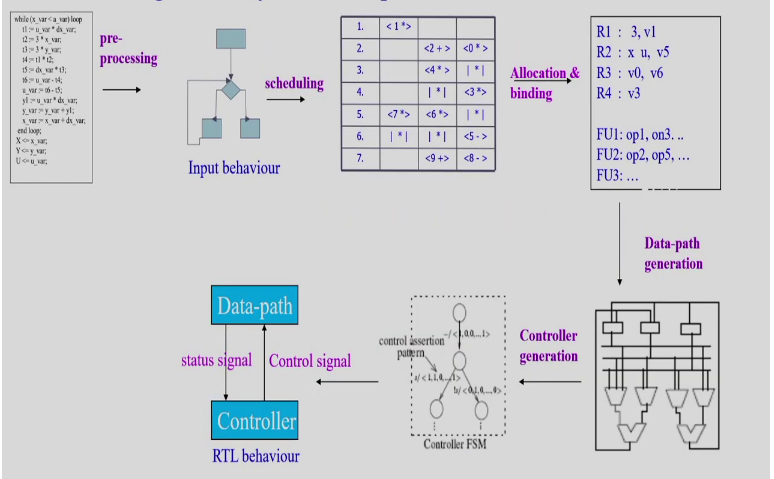
- Preprocessing: Intermediate representation (CDFG) construction, data-dependency, live variable analysis, compiler optimization.
- Scheduling: Assigns control step to the operations of the input behaviour.
- Allocation: Computes minimum number of functional units and registers.
- Binding: Variables are mapped to registers, operation to functional units, data transfers to the interconnection units.
- Data path & Controller design: controller is designed based on inter connections among the data path elements, data transfer required in different control steps.

8 IIT Guwahati

So, let us try to understand that. So, in the high-level synthesis, the steps are pre-processing, scheduling, allocation, binding, and data path and controller generation. Let us forget these things. Let us try to concentrate on the conversion steps, ok.

(Refer Slide Time: 10:04)

High-level Synthesis Steps



```
while (x_var < a_var) loop
    t1 := u_var * dx_var;
    t2 := 3 * x_var;
    t3 := 3 * y_var;
    t4 := t1 * t2;
    t5 := dx_var * t3;
    t6 := u_var - t4;
    u_var := t6 - t5;
    y1 := u_var * dx_var;
    y_var := y_var + y1;
    x_var := x_var + dx_var;
 end loop;
X <= x_var;
Y <= y_var;
U <= u_var;
```

1.	< 1 *>		
2.		<2 + >	<0 * >
3.		<4 * >	| * |
4.		| * |	<3 *>
5.	<7 *>	<6 *>	| * |
6.	| * |	| * |	<5 - >
7.		<9 +>	<8 - >

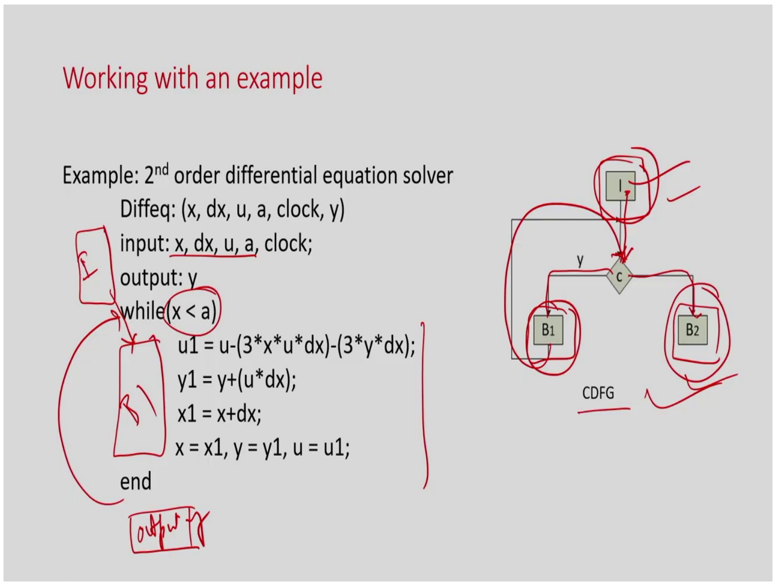
R1 : 3, v1
R2 : x u, v5
R3 : v0, v6
R4 : v3

FU1: op1, on3. ..
FU2: op2, op5, …
FU3: …

9 IIT Guwahati

So, we have this behaviour that is given to us, right. So, first of all, we should understand what is the control and data flow in this behaviour, right. That is something we should understand and that is given by the CDFG, control, and data flow graph.

(Refer Slide Time: 10:06)



Working with an example

Example: 2nd order differential equation solver

```
    Diffeq: (x, dx, u, a, clock, y)
    input: x, dx, u, a, clock;
    output: y
    while(x < a)
        u1 = u-(3*x*u*dx)-(3*y*dx);
        y1 = y+(u*dx);
        x1 = x+dx;
        x = x1, y = y1, u = u1;
    end
```

I
B1
output y
I
c
y
B1
B2
CDFG

So, you can understand that there is a block here where we are reading the inputs, right. You have to read this data. And there is a basic block here that is going to repeat. So, this is my basic block I, and this is a basic block where the series of operations is going to happen and this is going to repeat in the loop, right. So, this is what.

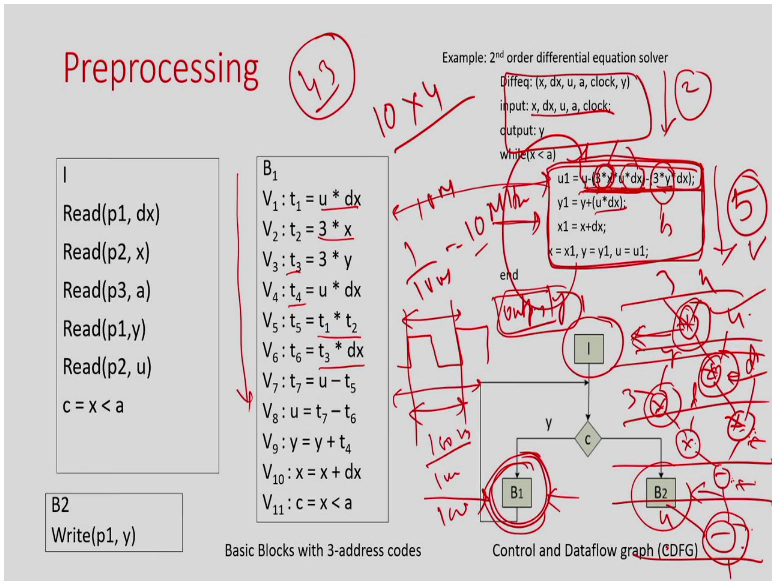
And then once this is done, we have a basic block here where we are going to output the y, right. So, this is how we identify the basic block. So, a basic block is a series of operations, sequential operations, there is your control branch, and there is no control flow, right.

So, this is my basic block, this is one of the basic blocks where I am going to do the loop body and this is a basic block where I am going to do the output things, right. So, there is 3 basic blocks. And then we will add the control flow because whenever you are it is doing this you will always come here, right. So, I am adding this control flow.

And then this is since this is a loop this is going to be repeated many times, right. So, whenever this condition is true, I am going to do these things on repeat and once this is done, I will go to this, right. So, this is the control flow. So, this is the basic block we will come into how the data flows would be represented inside this in the next slide. But this is how I will capture the control flow.

So, once we are given the behaviour, I should identify the first thing is that I should identify what are basic blocks in the clock behaviour and what is the control flow, right. So, this is what we will extract first, right. So, this is something we can automate. And there are obviously, this modern-day compiler can do it easily. So, this will be done first.

(Refer Slide Time: 12:05)



Preprocessing

Example: 2nd order differential equation solver

```
Diffeq: (x, dx, u, a, clock, y)
input: x, dx, u, a, clock;
output: y
while(x < a)
    u1 = u-(3*x*u*dx)-(3*y*dx);
    y1 = y+(u*dx);
    x1 = x+dx;
    x = x1, y = y1, u = u1;
end
```

I
Read(p1, dx)
Read(p2, x)
Read(p3, a)
Read(p1,y)
Read(p2, u)
c = x < a

B2
Write(p1, y)

B_1
$V_1 : t_1 = u * dx$
$V_2 : t_2 = 3 * x$
$V_3 : t_3 = 3 * y$
$V_4 : t_4 = u * dx$
$V_5 : t_5 = t_1 * t_2$
$V_6 : t_6 = t_3 * dx$
$V_7 : t_7 = u - t_5$
$V_8 : u = t_7 - t_6$
$V_9 : y = y + t_4$
$V_{10} : x = x + dx$
$V_{11} : c = x < a$

Basic Blocks with 3-address codes

Control and Dataflow graph (CDFG)

Now, inside this in this basic block. So, we have now we understand this is one basic block where the input will be taken, this is one basic block and then there is (Refer Time: 12:12) block here just to output y, right. So, this is what is the control flow. So, now, what we should understand is the data flow within each basic block, right.

Since here there is only input operation there are no dependencies as such. We will not talk about that much. But within this basic block, there is some dependency, ok. So, what we are going to do is we will take each basic block at a time. We will try to identify the data dependency or the data flow within that basic block, ok.

So, the first thing is to be understood how we will finally realize the whole hardware is instead of thinking about the whole design at a time, let us concentrate on one basic block at a time, right. So, this should be our approach, ok. Because once if we decide this will be the hardware, so this is my hardware for this, this will be the hardware for this, this will be hardware for this, I can combine them and I can get the complete hardware, right. And we will talk about that in more detail.

So, this is how. So, basically, we will consider one basic block at a time. So, now if you let us see if I just take this basic block at a time, ok. So, now, we have this basic block and we have this operation. The first thing we have to need is some hardware, right that is going to execute this behaviour. And you can see here this u is a very complex operation there are 1, 2, 3, 5 multiplication and 2 subtraction operations and so on, right.

So, the first thing I should understand is that this is a very complex operation and if I try to do it I have to put the multiplier in a chain, right because here I am going to do 3 x, then I will multiply with u, then I am going to multiply with dx, then I in the parallel I am going to do 3 into y, right, so 3 and y, then I will multiply that with dx, dx and then I will do the subtraction here and then finally, I will do another subtraction from u, right.

So, this is how I am going to do this. So, this is the data dependency, right. But the problem here is that if we try to, but in hardware, I have to execute this, in hardware, right. So, suppose you have made these connections like this. So, you add a multiplier 1, 2, 3, 3; multiplier and then these 2 multipliers in parallel, then there is a 5-multiplication followed by 2 subtraction operation going to happen to just to execute this, right.

And it has to be completed in a single clock, right in the same clock within this time frame for all these operations to be executed. And what is the impact of this? If you try to do this; obviously, it is possible your clock this length will be more, right you have need lot of time because you need 3 multipliers followed by 2 subtraction and so on. So, this length or the clock period will be more, and hence the clock speed will be less.

So, the kind of design you are going to get will be very slow, right because you are your, the clock period is very high and the clock frequency is one by clock period. So, say, for example, if this is a 100 nanosecond, it will be 10 mega Hertz, right. So, it is 1 by 100 nanoseconds which is 10 mega Hertz, right. So, if this is taking say 1 milliseconds that means, 1000 nanoseconds then it will be 1 mega Hertz, and if it is taking say 10 milliseconds then it will become 0.1 million mega Hertz, and so on.

You understand that. So obviously, if the clock period is more and then my frequency will be less and my clock speed will be less, right. So, the first thing we have to understand once we are going to execute the whole behaviour, I cannot execute everything in a single clock because if we try to do that then my whole circuit will be become very slow, ok. So, the first thing that comes to my mind is that once I have that behaviour, I have to decide how many clocks I am going to distribute the operations into multiple clocks.

And once I distribute then I can execute this operation in say 4 clocks. Say for example, given this behaviour I decided to do this multiplier at a time these two multipliers at a time, these two multipliers at a time, and then this and this. So, 1, 2, 3, 4, 5, say in 5 clocks I want to do this, ok. So, if I do this in 5 clocks; that means, my clock period will be 5 times lesser and my clock will be 5 times faster, right.

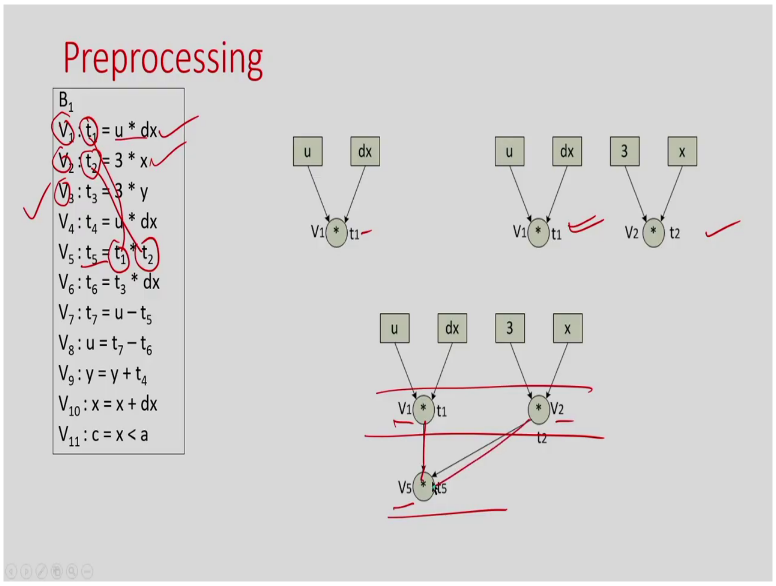
So, the first thing once you try to map this behaviour into hardware, the first thing is important to understand that realizing even first thing we have to understand that we have to take one basic block at a time, and once we take a one basic block at a time, I cannot execute the whole behaviour within that basic block in a single clock because that is not a practical design. So, what first thing we have to understand is that we have to break this behaviour into multiple clocks so that my clock will become faster, right. So, that is something is to do.

So, now the question is there are these operations are there, the, so many operations are there, so many multiplications are there which one; so, let us say I decided to do in 5 clocks, right. In the first clock, what are the operations I am going to do, in the second clock what is the operation I am going to do, in the third clock what are the operations I am going to do, and so on, right? If that was the question and you have to it cannot be random that I just put this operation here and here because in that case, your functionality might change, right.

So, to identify this operation to be executed in this clock and this operation in this clock, you need to understand the data flow. For example, in the diagram I have drawn here I cannot execute this behaviour in clock 1, right, until all the operations are done, I cannot do these operations, right. So, this depends on this operation. So, this operation depends on these two multipliers, this multiplier depends on this multiplier, and this multiplier depends on this multiplier.

So, until this is done, I cannot do this until this is done, I cannot do this, and so on. So, this is the data dependency. So, we have to identify the data dependency of the operations inside a basic block, right. So, that is something that is needed. So, we will do that.

(Refer Slide Time: 18:42)



Preprocessing

B_1

$V_1 : t_1 = u * dx$

$V_2 : t_2 = 3 * x$

$V_3 : t_3 = 3 * y$

$V_4 : t_4 = u * dx$

$V_5 : t_5 = t_1 * t_2$

$V_6 : t_6 = t_3 * dx$

$V_7 : t_7 = u - t_5$

$V_8 : u = t_7 - t_6$

$V_9 : y = y + t_4$

$V_{10} : x = x + dx$

$V_{11} : c = x < a$

So, for doing that first thing we are going to do is we will break this bigger expression into 3 address forms or smaller operations. What do we need? Because we, so this is nothing but 3 into x, right, this is this operation. This is nothing, but u into dx sorry, this is nothing but u into u, right. So, this is into u, 3 x into u, and so on. So, you need to identify the unique operations or the 3-address operation because I am going to execute the smaller operation in a clock, not the complete expression, right.

So, I have to break this expression into 3 address forms, ok. What is the 3-address form? Individual operation. So, for example, here see that 3 into; so, I just do this u into dx which I am going to put in t 1, then 3 into x I am going to put it into t 2, then this 3 into y I am going to put it into 3. Then what I am doing?

This u into dx I am going to put it into t 4. Then, I am doing this t 1 and t 2, right. So, this is my t 2 and this is my t 1, if I just multiply t 1 and t 2, I will get this expression, right. So, this is what I am doing here. So, I am just doing t 1 into t 2. Similarly, 3 and y are basically my t 3, so then t 3 into dx I will get that 3 into y into d x. So, this is how I will just break this bigger expression into smaller 3 address forms, ok.

And why do we need that? Because these are the unique operation or the unit operation that is to be scheduled. We have already understood that I cannot execute the complete expression in a single clock which will make my circuit very slow. So, I need to identify the unit operations and this is called 3 address operations.

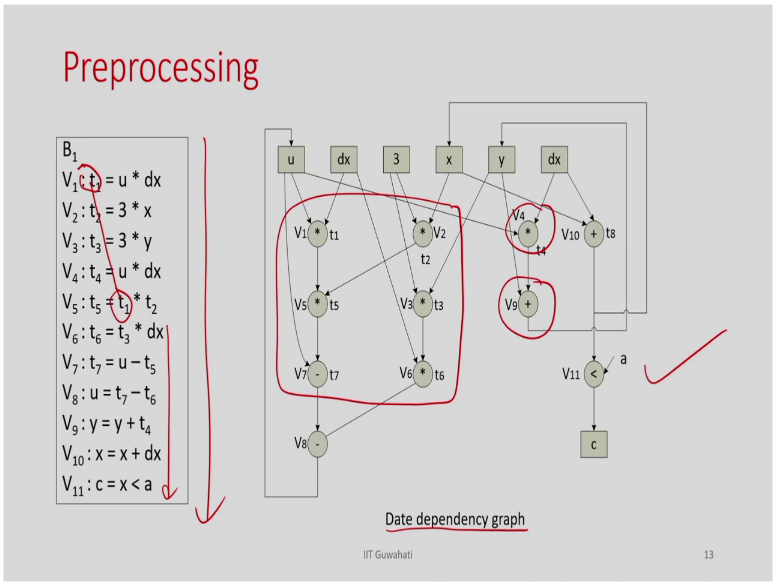
So, this is what you have to do first, that you break this bigger expression into smaller unit operations which are all 3 address operations. And then, as I mentioned because to identify which operation to be executed where you need to identify the data flow, right. So, data dependency, how we can do that?

So, this is my 3-address form for this basic block, right. So, this is I am talking about this basic block, right. And then for this basic block, so we have to identify the dependency. So, here I am doing u into dx and storing in d 1, then this is 3 into x is storing in d 2 and this is u into dx is for these operations. So, this is how if we just do that.

Then what is happening? This t5 is multiplying t1 into t2. So, now, these two operations this operation V5 depends on V1 and V2. So, this will give you the dependency that until these two operations are completed, I cannot do this. See, if I just you make a node for each operation and whenever there is a data dependency; that means if there is a variable defined here and it getting used here, so these two I will add an edge from this node to this node, right.

Similarly, I have this t2 is defined here it is getting used here. So I am going to add an edge between these two.

(Refer Slide Time: 21:44)



Preprocessing

B_1
$V_1: t_1 = u * dx$
$V_2: t_2 = 3 * x$
$V_3: t_3 = 3 * y$
$V_4: t_4 = u * dx$
$V_5: t_5 = t_1 * t_2$
$V_6: t_6 = t_3 * dx$
$V_7: t_7 = u - t_5$
$V_8: u = t_7 - t_6$
$V_9: y = y + t_4$
$V_{10}: x = x + dx$
$V_{11}: c = x < a$

Date dependency graph

IIT Guwahati

13

So, if we just keep doing then I will get this data dependency. So, this data dependency dictates what operation to be executed in which time step. It makes sure that this operation V

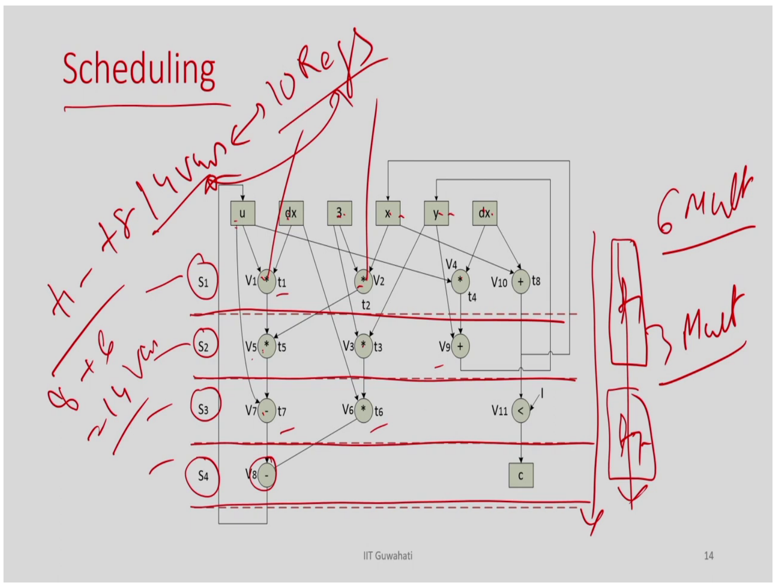
8 would not happen in clock 1, it can only happen when all these operations are completed, right. This V 9 cannot execute before V4. V4 is done then only I can do V9, ok. So, this is what is called a data dependency graph, ok.

So, the first thing if you understand now given a C code now, what I have to do; I have to do it? I have to identify the basic blocks. Within the basic block, you distribute this big expression into 3 address forms, and from these 3 address forms, you identify the data dependency for each basic block. So, this gives me the idea of how many basic blocks are there, what is the control flow, and within the basic block what are the operations available and what is the dependency among them, ok.

Dependency in terms of reading after writing, right. So, the read after write dependencies, ok. So, now the way the I should proceed is that instead of taking everything together, if say I will take this basic block at a time, then I am going to take this basic block at a time, and then I am going to take this basic block at a time. I will identify how many clocks I need to complete this behaviour, how many clocks I need to complete this behaviour, how many clocks I need to complete this basic block, and when you combine this using this control flow.

And that will give you the overall behaviour order this is how the things will execute. And so that is something I am going to do, ok.

(Refer Slide Time: 23:35)



So, what I am doing here is that I identify this basic block, and then I am going to take this basic block V 1 because that has a lot of operations others are smaller operations. So, and then as I decide that I am going to do the scheduling, I will decide how many time steps I need to execute that behaviour, ok.

So, here say I just decide that I am going to do these operations in timestamp S1, these are the operation I am going to do in timestamp S2, these are the operation I am going to do in timestamp S3 and these are the operation I am going to do in timestamp S4. So, this is something is satisfying the data dependencies. You can see here that when I am trying to schedule this both his predecessor operations are done.

I am trying to schedule here which predecessors are already available and when I am trying to do this its predecessors are already executed, right. So, that means, scheduling determines the operation for each timestamp, right. So, that is called that means it is assigning timestamp to each operation; it determines in which each clock what are the operation going to be executed, ok. So, this is what scheduling is.

So, we have understood that within the basic block there are big expressions. I cannot execute in one clock, and I will distribute them into 3 address form, I will identify the data dependency, and then I will assign the operation to the timestamp which satisfy the data dependencies, right. So, if what one; once it is done, I understand that this loop body, right, so that loop body I can execute in 4 clocks, right.

So, this loop body I can execute in 4 clocks because that is what I have decided. And then since this is a loop, I will just keep doing it every time. So, if the loop executes 10 times and each time I need 4 clocks, so I need 40 cycles, 40 clock cycles to execute the loop body, and let us say I need 2 clock cycles to execute this and I 1 need clock cycle to execute this, so total clock cycle needed to execute this behaviour is 2 + 40 + 1, 43, right.

So, although this is a C code is given just like this, it needs 43 clock cycles to complete the execution. I am just giving an idea that this is how this behaviour is going to execute in the hardware that in the first clock 2 clocks it will read this data, next 40 clocks it will just do these operations of the loop, it will just repeatedly these doing these 4 operations in the loop and once this loop is done, it will come out and I will print the output in one clock. So, this is how this particular behaviour is going to be executed in the hardware, ok.

So, this is understood, right. So, this is what I am going to do overall I understand this particular control and data flow is something is important and from there I can understand what are the order of the clock and what the total clock is needed after the scheduling, ok. So, that is understood that this is the operation to be executed in timestamp 1 and so on, right.

Now, the question is how these operations going to execute in hardware because there is no hardware with me. I just decide that for the loop while loop I have given, I need 43 clock cycles, but I never decide where this operation will be going to execute because there is no hardware to me, there is no data path to me, and we have to realize that, right. We have to realize a data path.

So, now the question is here how do we going to realize this. I am going to consider a multiplier for each such multiplication then there are 1, 2, 3, 4, 5, 6 multipliers, then I need 6 mult, right. So, then I need 6 multipliers which is something that is obvious, I mean obviously, it is a good solution it is the solution, but it is not a good solution because there may be a possibility that I can execute the same behaviour using 3 multipliers as well. So, instead of using 6 multipliers, I can use only 3 multipliers to do these things, right.

So, the problem that will come to mind is that given this schedule what is the minimum number of function units needed to execute this behaviour. So, that is the problem now, right. So, I have decided that in clock 1 I am going to operate do this 5-operation clock, 2 I am going to execute these 3 operations, and so on, but I never told you how many multipliers I need to execute this behaviour. So, that is what we need to identify next.

So, I know that there are 6 multiplication operations happening across 4 clock steps, now do we need 6 multipliers or we can use a lesser multiplier to do this the answer is it is possible to use a lesser multiplier than 6 multipliers to execute this. So, then the question you are going to ask is how many minimum numbers of multipliers is needed to execute this behaviour, right. So, that is what is called function unit allocation. You identify the minimum number of function unit is needed to execute this behaviour.

Next, the question is come, ok. So, there are 6 multiplication operations and there are you identify 3 multipliers say for example, but which multiplier is going to do which operation, right. So, there are 6 multiplications, and 3 multiplier function units. So, you have to map this, right, you have to map this multiplication operation to the multiplier and that is what is called binding, ok.

So, the next step is you identify or allocate a minimum number of function units and also bind the operations of the behaviour to the multiplier or the function units. So, that is what is called function unit and allocation and binding, right. So, that is the next problem. Similarly, you have many variables in the program, right in the hardware, there is no variable, right, in the hardware you have memory, registers, RAM, and ROM. So, if you have an array, you map that array to RAM, ok. If you have variables, you should map them into the register.

Again the same question I am going to ask. So, obviously, in the hardware, you need registers and RAM for the array, right. So, let us forget now the array part only say think about you have the variables we have, ok. So, you have a set of variables, as said in this program you can understand that you have variables like t 1 to t 8 variables and 1, 2, 3, 4, 5, 6, right. So, 8 plus 6, 14 variables are there, right, in this variable. So, 1, 2, 3, 4, 5, 6 plus t 1 t 2 that intermediate variables, right.

So, now the question is do we need 14 registers or can I use a smaller number of registers to store this, ok. The answer is, yes, you do not need 14 registers because that is the maximum possible thing and which does not give you the best hardware. So, the question or the problem that you should need to solve now given this variable available and the schedule, identify the minimum number of registers needed to execute this behaviour or store this variable to this register that is called register allocation, ok.

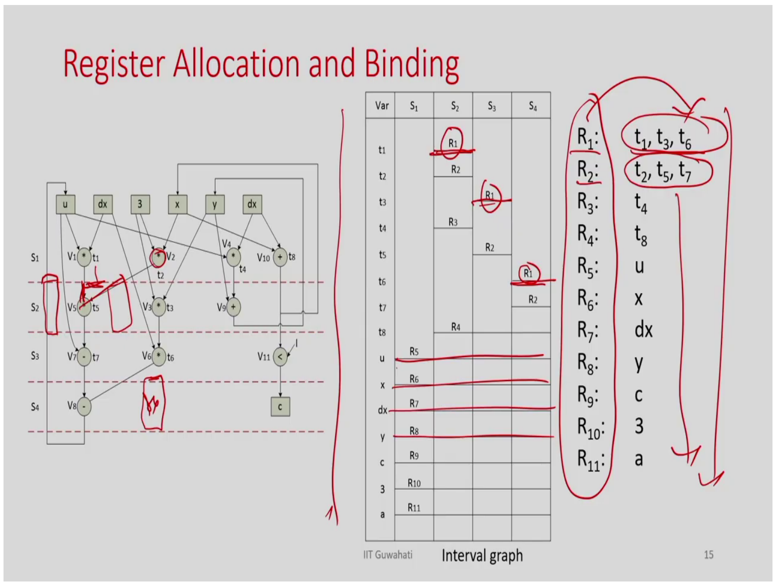
And once you have the same, once the allocation says, for example, say for these 14 registers you identify you need 10 registers, right. So, there are 14 variables, and say I am giving you one example that I need 10 registers, from by some algorithm by some analysis, I identify, I need 10 registers.

So, the question is that again the binding problem is that you have 14 variables which register will store which variable and which time. So, that is something that is the binding problem. So, basically, once the schedule is done based on the schedule you identify the next two problems as the register allocation binding problem and function unit allocation and binding problem, ok.

So, that you have to do now because that is how you actually realize the data path or the hardware. Because so far you just decide the clocks, you just decide so far you after scheduling you only know you need 43 clocks to execute this behaviour, nothing else. So, now, you have to realize, ok I need 43 clocks, but what is the resource I need. So, the resource you have to determine.

And the resource is two types of the resource data path and the function unit and the registers, right. Registers store the variable, and function unit to execute the operations, ok.

(Refer to Slide Time: 31:56)



Register Allocation and Binding

R_1:	t_1, t_3, t_6
R_2:	t_2, t_5, t_7
R_3:	t_4
R_4:	t_8
R_5:	u
R_6:	x
R_7:	dx
R_8:	y
R_9:	c
R_{10}:	3
R_{11}:	a

So, let us consider this register allocation binding problem and how we should do that, right. So, now, the question again I am going to ask you how you should do that, right. So, how we can share the register, right. So, for example, if you understand that if there are 14 variables and say 10 registers; that means, one register must store more than one variable, right, but at a time it can store only one value, right.

The question here is that there are 10 registers and 14 variables; definitely, some of the registers are storing a variable more than one variable. And obviously, at a time it can store only one variable, right. So, that means, it is actually for some timestamp it is going to store one variable and for another timestamp, it is going to store another variable.

So, for example, here there are 4 timestamps, right. So, there is 4 times 4 clock, maybe one register is needed to store, so which is storing say t 1 for this time and say t 7 here, right. So, then only it is possible, alright. So, that is the problem that given this schedule how do you identify the minimum registers and their binding, ok. So, that is something that lets us understand. We can do this this way.

So, obviously, understand that a register has to share more than one variable, but how do determine that. So, once you write this program, right. So, once you write this program in the C code, we just write t 1 equal to this and that, right. So, once you write this program, we just write this, we never bother about what happened to this t 1 after this, right. After this t 1, I do not care how this t is happening, right.

So, here you understand that this t 1 is defined here and it is getting used here, right so that is all. And the rest of the time I do not need this. So, once this t 1 is defined and then it is getting used after that I do not need t 1. In the hardware, I can remove that value because I do not have any use of that t 1 for the rest of the circuit or the rest of the time, so let us remove that value from the register and store something else.

So, that is how the sharing happens, ok. And how we should determine? I should this is determined by the lifetime of the variable, so that means, the time where it is getting defined and where it is getting last used, right. So, for example, you see here that t1 is defined exactly at this point at the end of timestamp one and it is just used here, right. After that, it has no use. So, the lifetime is basically just this, right. So, just at the end of S1 and in S2. So, that is what is represented by this. So, this is the lifetime of the variable.

So, if you just think about this t6 it is exactly defined at this point and then it is getting used here. So, the lifetime is this, right. So, this is my t6. So, this is how for each variable I can identify the lifetime. So, this is my t 1 the lifetime is only for S2, it is just starting from this. t2 is again it is basically defined here and it is getting used here, so just the lifetime is here. So, this is how you can identify the lifetime of the variables. So, that is the time where I need that value in the hardware, ok.

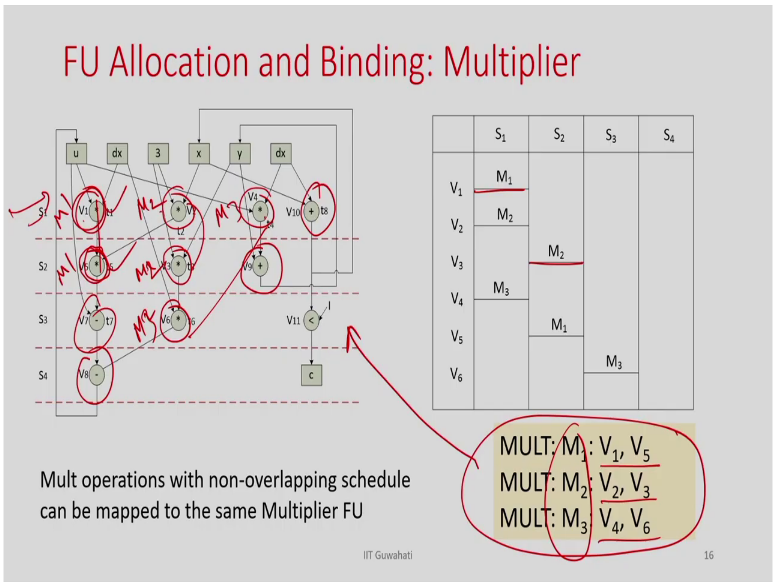
And then, I can share register if their life is normal; that means, if this variable is needed here, I need this variable is needed here, I and they do not need in a common time or there is no overlap in their lifetime. So, then I can store them in a single register, right.

So, for example, here say t 1 lifetime is this t 3 lifetime is this t 6 lifetime is this and they are non-overlapping. So, they are they need in the harder in a different time. So, I can store this t 1, t 3, and t 6 into register R 1, right that is what is the register sharing. And so, this is how the sharing happens. So, this is how I will identify the sharing of variables within a register and hence I can reduce the number of registers.

So, if you take this behaviour, I can show that I need only 11 registers for these variables, ok. And the sharing is registered 1 is going to store t 1, t 2, t 3, and t 6; register 2 is going to store t 2, t 5, and t 7, and there is no sharing for the other register because their life is kind of overlapping. For example, u, I need for all the time. So, I cannot share it with anybody because it needs to be in the register all the time and so on say. So, this is what is called register allocation and binding.

So, what I am doing here is after scheduling I identify what is the minimum number of registers needed, and for each register what are the variables that can be stored, ok. And the problem here is that the objective is to reduce this number of registers and hence improve the share, ok. So, this is the register allocation and binding.

(Refer Slide Time: 36:49)



FU Allocation and Binding: Multiplier

Mult operations with non-overlapping schedule can be mapped to the same Multiplier FU

MULT: M_1: V_1, V_5
MULT: M_2: V_2, V_3
MULT: M_3: V_4, V_6

IIT Guwahati

16

The same thing is to be done for function unit allocation and binding. So, what is the idea here? That is in function allocation and binding, so I am doing this multiplier here and I am doing this multiplier here. So, they cannot share the same multiplier, right this multiplication and multiplication happening in parallel in timestamp 1, and this 3-multiplication happening in timestamp 1. So, I need different FU.

But this multiplier happening in timestamp 2, this multiplier in timestamp 2, so I can share the same multiplier to do this and this, right because in the timestamp 1 this multiplier is going to do this operation V 1 and this multiplier is going to do this operation V 5 in timestamp 2. So, I can do using the same multiplier, I can do these two same two multiplication because they are actually not happening at the same time.

But I need a different multiplier here because this multiplier and this multiplier happening in parallel in hardware, alright this. So, I need 3 multipliers and I can do this, right. So, I can do this V 1 and V 5 using multiplier 1, V 2 and V 3 is in multiplier 2, and this is for multiplier 3. And this also I can do using any of them, right. So, let us, I just do in multiplier 2.

So, that means, again I can identify the kind of the timestamp where it is getting used and then if a multiplier is used in different, is happening in different timestamp I can share the same multiplier, right. So, this is how I can actually identify the minimum number of multipliers needed and the binding, ok. And let us say this is one of the possible bindings I have identified for this.

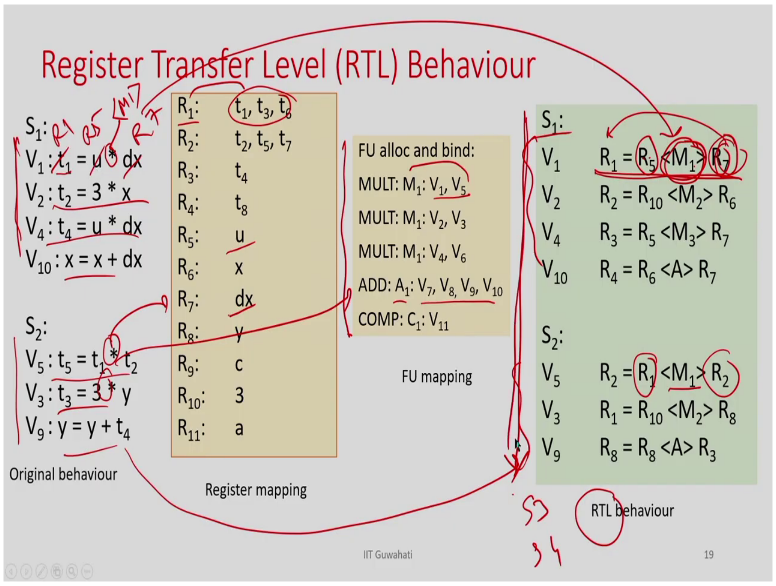
So, as I mentioned earlier that I need 3 multipliers for the 6 operations, and 3 multiplication operations, so this is happening here. And this multiplier is going to do this V 1 and V 5, multiplier 2 is going to do V 2 and V 3, and then multiplier 3 is going to do V 4 and V 6, ok. So, this is what is the mapping.

And the same thing I have to do for adder, right. So, this is what I am just doing for the multiplier, I can do it for the adder. Adder means additional subtraction is the same in the hardware. So, I can use the same ALU for that, ok. So, this is what is the next step. So, first what I did do? I identify the total number of clocks is needed that is first because you cannot do everything in a single clock, then it is not feasible.

So, I have to do it in multiple clocks, and given the behaviour, I have decided to do it in say 43 clocks. And then, to execute that behaviour in 43 clocks I have to determine what is the minimum number of registers and what are the minimum number of function units needed and that is what is called the allocation and binding step, right.

So, there I am going to identify the minimum number of multipliers, the minimum number of adders, minimum number of registers with that. So, once this is done. So, I have both available, I know how many clocks are needed, each clock what are the operations to be executed, which operation to be executed using which function unit, and which variable is stored in which register. So, this information we have now, ok.

(Refer Slide Time: 39:50)



Register Transfer Level (RTL) Behaviour

Original behaviour

S_1:
$V_1 : t_1 = u * dx$
$V_2 : t_2 = 3 * x$
$V_4 : t_4 = u * dx$
$V_{10} : x = x + dx$

S_2:
$V_5 : t_5 = t_1 * t_2$
$V_3 : t_3 = 3 * y$
$V_9 : y = y + t_4$

Register mapping

R_1:	t_1, t_3, t_8
R_2:	t_2, t_5, t_7
R_3:	t_4
R_4:	t_8
R_5:	u
R_6:	x
R_7:	dx
R_8:	y
R_9:	c
R_{10}:	3
R_{11}:	a

FU mapping

FU alloc and bind:
MULT: M_1: V_1, V_5
MULT: M_1: V_2, V_3
MULT: M_1: V_4, V_6
ADD: A_1: V_7, V_8, V_9, V_{10}
COMP: C_1: V_{11}

RTL behaviour

S_1:
V_1 $R_1 = R_5 <M_1> R_7$
V_2 $R_2 = R_{10} <M_2> R_6$
V_4 $R_3 = R_5 <M_3> R_7$
V_{10} $R_4 = R_6 <A> R_7$

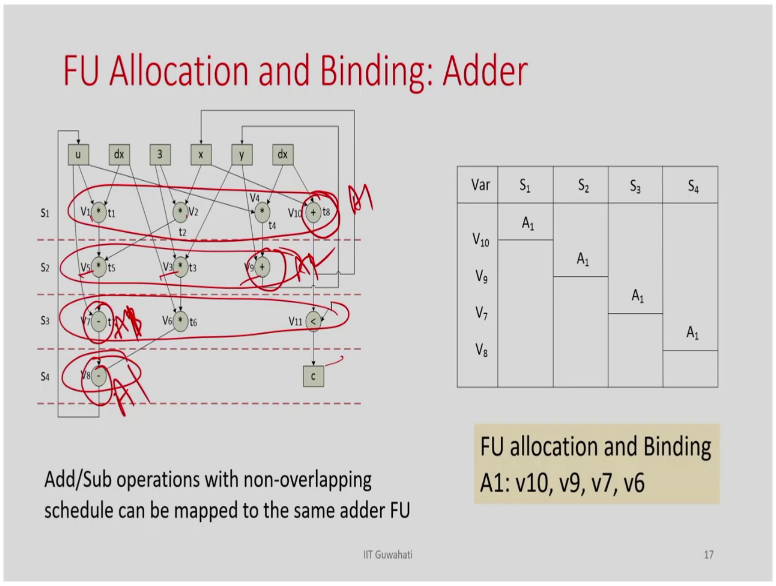
S_2:
V_5 $R_2 = R_1 <M_1> R_2$
V_3 $R_1 = R_{10} <M_2> R_8$
V_9 $R_8 = R_8 <A> R_3$

S3
S4

IIT Guwahati 19

So, if you have this information what I can do is this. So, this is my I know this is the schedule in timestamp one this 4 operation is happening, in timestamp 2 these 3 operations happening, and so on. I am not writing for the other 2.

(Refer Slide Time: 40:04)



FU Allocation and Binding: Adder

Var	S_1	S_2	S_3	S_4
V_{10}	A_1			
V_9		A_1		
V_7			A_1	
V_8				A_1

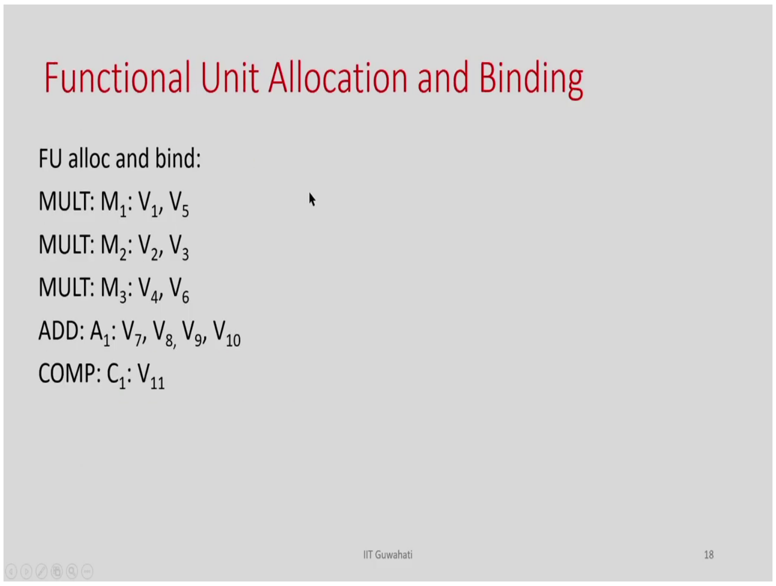
Add/Sub operations with non-overlapping schedule can be mapped to the same adder FU

FU allocation and Binding
A1: v10, v9, v7, v6

IIT Guwahati 17

So, these 4 operations are happening in timestamp 1, these 3 operations are happening in timestamp 2, these 3 operations are happening in 3 timestamps 3, and these two, this operation is happening in timestamp 4, right. So, I just write this V 1, V 2, V 4, and V 10 into here, right.

(Refer Slide Time: 40:19)



Functional Unit Allocation and Binding

FU alloc and bind:

MULT: M_1: V_1, V_5

MULT: M_2: V_2, V_3

MULT: M_3: V_4, V_6

ADD: A_1: V_7, V_8, V_9, V_{10}

COMP: C_1: V_{11}

IIT Guwahati 18

So, this is what is happening here. And this V 5, V 3, and V 9 are happening, in time V 5, V 3, and V 9 are happening in timestamp 3. So, I just wrote it for 2 clocks. Similarly, in S 3 and S 4 the operations are there. And I know these variables are mapped to these registers and so on. So, this is my register mapping information.

And this is my FU mapping information, I need 3 multipliers and these are the operation to be executed in multiplier 1 and so on. And I need one adder to execute all these 4 operations because you see here there is only one addition operation here, one addition operation here, one addition operation here, and operation. These are all in the different timestamps. So, I can just use adder one to do all these 3, all 4, right. The same adder to execute all the additional operations, ok.

Say this is the mapping information is available, right, so then what I can do is very interesting. So, I can rewrite this behaviour in terms of register and FU. How? Because I know this t 1 is nothing, but R 1. So, what I am going to do I will just write R 1 here. This u is nothing but R 5, dx is nothing but R 7, and this multiplier is nothing but multiplier 1, right. So, that means, if I just rewrite this, I will get this expression, right. So, this is the expression I am going to do.

So, the same thing I can do for this, for this and this, so I will get this. Similarly, I can do all operations. I just replace the variable using the corresponding register, and replace the operation by the corresponding FU from this mapping information, then I can rewrite the scheduling behaviour in terms of this. And what is this? This is nothing but RTL, register transfer level behaviour.

Why? Because here you can see that I have these two registers, and this clock in clock 1 will be multiplied and this value is stored in R 1. So, this is a transfer of value among registers, right through some function unit, ok. So, this is what the transformation from the behavioural code into register transfer level code, right.

This is what is the C 2 RTL transformation, ok. And this is what is happening for every clock, right. So, I can do the same thing for S 3 and S 4 and so on, right. So, this is how I replace the operations of the C behaviour in terms of registers and the functional unit of the data path. So, that means, I have now clearly understood what is the operation to be executed in clock 1, clock 2, clock 3 and clock 4, and so on. And this 4 we are going to repeat, right. This is done, right.

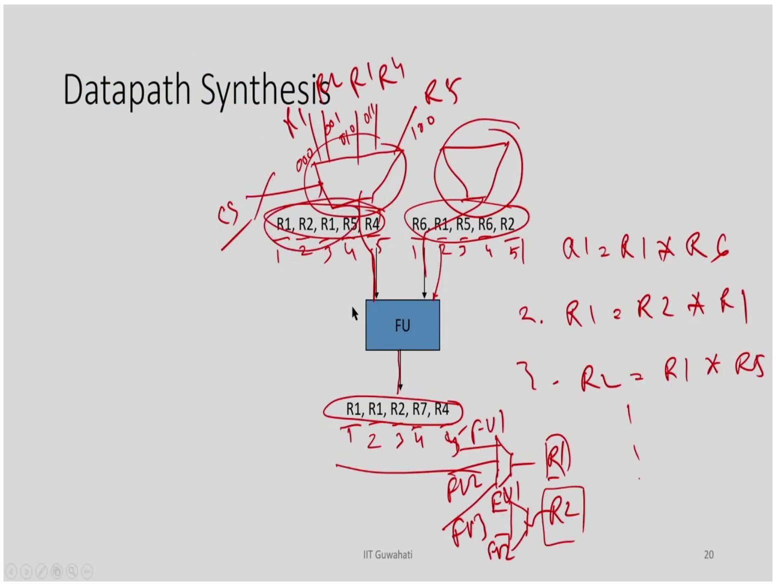
So, till this point, so I already have I can rewrite my C code in terms of register transfer level mode. So, all this my t 1, t 2 these things are there is they do not exist now. I have only R 1, R 2, everything. I do not have this star plus and all, instead of I have the FUs now, M 1, M 2, M 3, and so on, ok.

So, once you have this what is missing here? The missing is that is, ok I understand that I have to do this, but I have to make the connection from this R 5 to this multiplier 1, R 7 to this multiplier 1, and then the output of this multiplier to this R 1 then only this operation is possible, right. Unless you make the connections from the register to the FUs, FUs to the register again you cannot realize these things in the hardware, right and that is what is called data path synthesis.

So, given this operation to be executed they are not connected. I have, I only identify I need 11 registers and 3 multipliers and one adder, but the connection from this register to this multiplier, the connection from this register to this adder, and so on, is not determined yet, right.

So, I have to determine that and that is what is called data path synthesis. I have to make the connections from my components, register to FUs, and FUs to register again and that is what is called data path synthesis. And that step is something very interesting.

(Refer Slide Time: 44:19)



See here, what I have to do in timestamp 1 for each multiplier I have to realize what the input is coming, right. So, I realize that says for certain cases say in timestamp 1 I need R1 and here say R6, and the result will be going to store in one. So, this is my 1 timestamp 1 operation. In timestamp 2, say I need this R2 here, R1, and I need the result will be stored here.

So, it is basically this FU is doing R1 equal to R1 star R6, in timestamp 1. In timestamp 2, it is doing R 1 equal to R2 star R1. In timestamp 3, it is doing this, right, so that means, R2 is equal to R1 star R5 and so on, right. So, I can understand what is the register to be connected to the left input of this FU, right and what is the register to be connected to the right input of this FU.

Similarly, the output of the register to go to which are the resistors. And remember they are actually time-division multiplexing, right. So, they are not at the same time. So, they are in a different time, right. So, in time 1, I need R1 and R6, and the result should go to R1 and so on, right.

So, the point here is that given this register transfer level behaviour, I can just go by each step and each multiplier and I identify the register input. For example, here for M 1, I need R5 and R7 in timestamp 1; for M 1 I need R1 and R2 for the next timestamp, and so on. This is how I can identify the possible register inputs to the FUs possible register to the FUs and output of the FU to the possible registers, right.

Once you have this, what do you have to do? You have to make a connection, right. You just need a Mux here, right and you have to put this R1, R2; R1, R2, R1, R4, and R5, right. So, you need a Mux for that and this will come here. And you need a controller which will basically control this, right. So, if this controller is in 3 bits, so if you give 000, then this R1 should come here.

If you give 001 then this R2 should come here, if you give 010 then this R1 should come here, and so on. And then if you give this 011 then this R4 should come here, right. And then if you give 100, then this R5 should come here and so on. So, that is what is called data path synthesis.

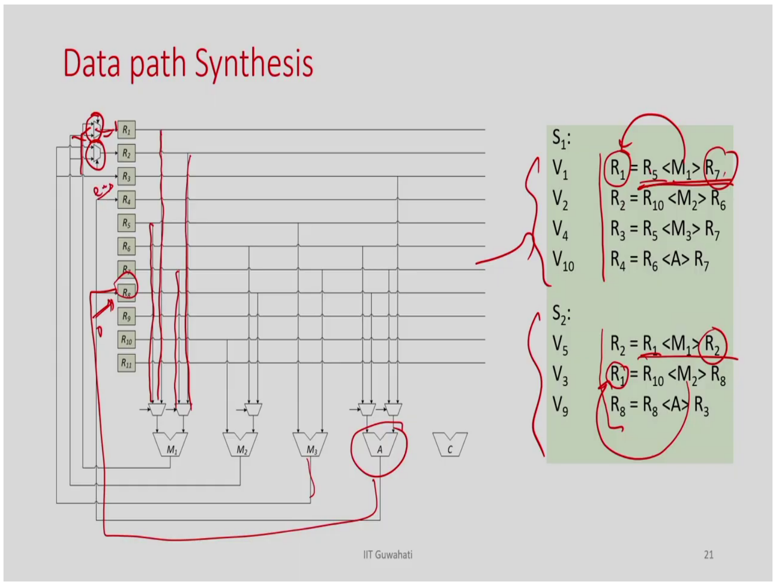
You identify the registers that need to be connected to the left input of the FU for different clocks and accordingly you put a mux there and you assign the controls so that the corresponding value can go to that FU in that correspond if you give the correct control signal, right. If I give the correct control signal that corresponding value should go here.

Similarly, I have to put a Mux here which will go this. And similarly, this value can go to R1, so for each register R1, R2, and so on you need a Mux at the wire. Because this data can come from a FU 1, also FU 2, and come from FU 3 and so on, right. Similarly, for R2 also the data can come from say FU 1 and say FU 2, right. So, you need to put multiplexers at the input of each function unit, you need to put a multiplexer at the input of each register.

And the size of the multiplexers and the exact connection to be made can be identified from this RTL behaviour, right. So, best because I have to realize this behaviour in the hardware. So, from this behaviour I can identify the possible register inputs and it is also in the time-division multiplex, right.

So, I have to understand each which time which register is needed, and based on that I just put the multiplexer at the inputs, and then, and also because one register input can come from many various hour function units, I need a multiplexer at the input of the register as well. So, this is how I will make the connections from the register to function unit, function unit to registers, right.

(Refer Slide Time: 48:42)



Data path Synthesis

S_1:

V_1 $R_1 = R_5 <M_1> R_7$

V_2 $R_2 = R_{10} <M_2> R_6$

V_4 $R_3 = R_5 <M_3> R_7$

V_{10} $R_4 = R_6 <A> R_7$

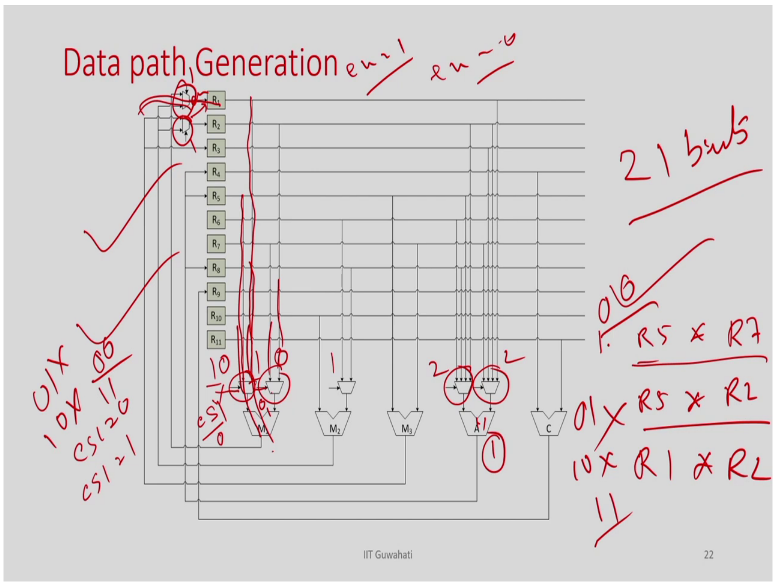
S_2:

V_5 $R_2 = R_1 <M_1> R_2$

V_3 $R_7 = R_{10} <M_2> R_8$

V_9 $R_8 = R_8 <A> R_3$

(Refer Slide Time: 48:45)



Data path Generation

So, if I just take this behaviour, the behaviour I have is the data path I am going to get. So, no need to go into detail about the correct correctness of this, but you can understand this is the multiplier 1 and this data should come from R 5 or R 1. Similarly, this multiplier input should come from R 7 or R 2 which you can understand from this behaviour. See for multiplier 1, in the first clock the left input should come from R 5, right.

So, I just put R 5 here and the next clock should come from R 1. So, I just connected R 5 and R 1 to the left input. Similarly, for the right input, I need R 7 or R 2. So, I have just connected

R 7 and R 2 and this adder is doing 4 operations in a different clock, so I need to put these 4 connections. So, that is why these adder inputs are 4. But this multiplier is doing only 2 multiplication and that is why I have a 2 in 2 as multiplexers.

So, this is something optimization I am not going to discuss, but this is how I should make the connections. And see here you also you can see that register 1 if you understand so, arrange this register 1 input is coming from multiplier 1 in the clock 1 and register 1 multiplier is coming from multiplier 2 in clock 2. So, I have to make FU 1 and FU 2 at the input to this multiplexer, right. That is what is happening here. And similarly for registers 2 and 1.

So, if I just go step by step to this behaviour this schedule behaviour, I can make this connection, right. So, first I will identify the register's possible register inputs, and based on that I will make the connections and finally, my data path is ready. So, this data path is ready to execute the behaviour clockwise, then the first clock it is going to do this R 1 into R 5, R 1 into R 7, in the next clock it is sorry R 5 into R 7, next clock it is going to do R 1 into R 2 and so on.

So, the data path connections are also done, right. So, I have a data path where the connections from the registers to the multi-function unit and function unit to the registers are done. What else is needed? Right. So, this data path connection is over what is needed further needed the further needed is the controller, right because I know that it is a 0 single bit signal say S, say CS 1.

So, if I put CS 1 equal to 0 then this input will come and it will go to this, right. And if I give CS 1 equal to 1 then this input will come to this, right, to this left input of the multiplier. But I have to determine in each clock which operation to be executed because in the first clock, in clock 1 I am going to do R 5 into R 7, right. So, I have to give CS 1 equal to 0 and this one also 0, right.

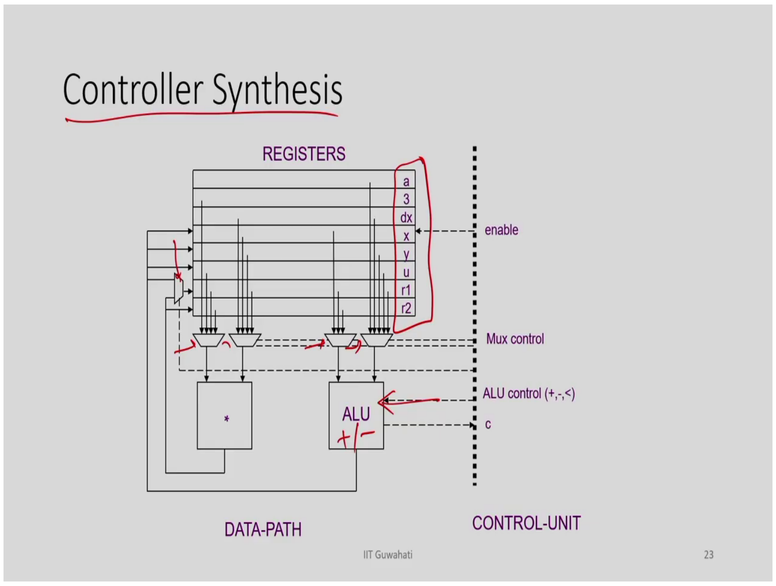
If I give this then only this multiplier will do R 5 into R 7 if I give 0 here and 1 here, then operation R 5 into R 2 is going to execute. But that is wrong I do not need that, right. Because I need to execute R 5 into R 7 in clock 1, in the first clock I have to give CS 1 equal to 0 here 0, I should not give 01, right. So, I should give 00 for the first clock and the next clock I need to give this and this, so I should give 11.

So, 01 and 10 are not valid input for this multiplexer, these two multiplexers. If you give 0 and 0 and 1 here, 10 here it will do some of the multiplication, but that is not something you need to do because I need to do this R 5, R 7, clock 1, R 1 into R 2 in the clock 2. I do not need to do R 5 into R 2 or R 7 into R 1, right. So, we have to avoid that.

So, that means, although the connections are ready, I need a controller. So, that will tell this data path you do this R 5 into R 7 in clock 1 and R 1 into R 2 in clock 2 by giving the control signal 00 here and 11 here. It never gives you 01 or 10. It does not give that control signal.

So, that means, I need somebody to monitor or control the operation to be executed in the data path because this data path can do many operations, but all of them may not be relevant or the correct for my purpose and so I need a controller, so that will determine the operation to be executed in that particular clock. And that is nothing, but the operation is mentioned this what operation to be executed is given to us. I have to just make sure that the controller is executing that operations only nothing else, right.

(Refer Slide Time: 53:22)



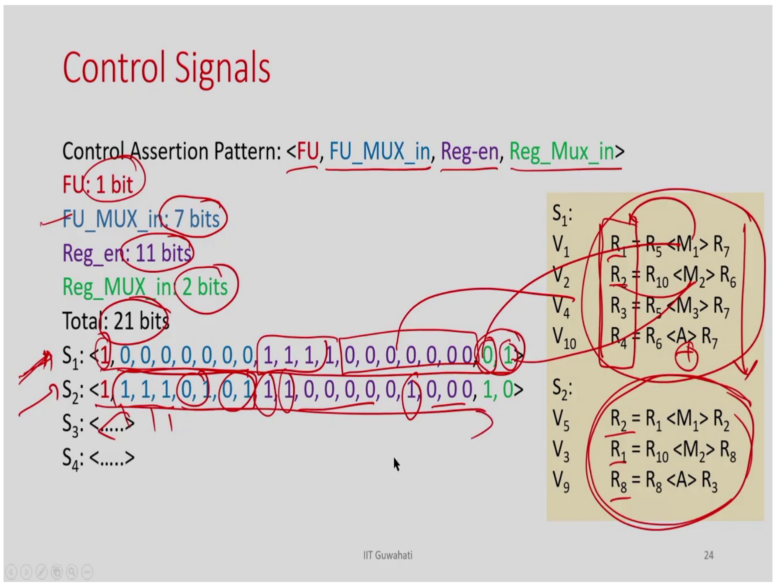
Controller Synthesis

So, that means, the next step is the controller synthesis. So, you have to identify the control signal for the Mux, you need to identify the control signal for this input Mux to the registers, and you need some control signal for the FU. If it is doing say plus and minus, whether it is doing plus or minus, and also you need to enable signal for the registers because that these registers they are connected here, so it may be that in every clock I am not writing this data register, right.

So, if, if I give a, enable 1, if the enable is equal to 1 then only whatever the data is coming here will update the register. If the enable is equal to 0 it will ignore the data. So, for example, say if you see here this R 1 is getting executed here and here, so R 1, R 2, R 3, and R 4, right. Other registers are not getting executed. So, I should put this R 8 equal to 0 in time step one because this R 8 is connected from this FU, but I do not want to update this register.

So, I just need to enable it here to stop this updating of the register in clock 1 because in clock 1 I need to only update R 1, R 2, R 3, and R 4. In the second clock, I need to update R 1, R 2, and R 8. I do not need to update R 3 and something else. So, I need to put an enable equally to 0 in clock 2 and so on, right. So, I need this many control signals, right. So, that is what is called controller synthesis.

(Refer Slide Time: 54:45)



Control Signals

Control Assertion Pattern: <FU, FU_MUX_in, Reg-en, Reg_Mux_in>

FU: 1 bit

FU_MUX_in: 7 bits

Reg_en: 11 bits

Reg_MUX_in: 2 bits

Total: 21 bits

S_1: <1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0 0, 0, 1>

S_2: <1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0 0, 1, 0>

S_3: <.....>

S_4: <.....>

S_1:

V_1 $R_1 = R_5 <M_1> R_7$

V_2 $R_2 = R_{10} <M_2> R_6$

V_4 $R_3 = R_5 <M_3> R_7$

V_{10} $R_4 = R_6 <A> R_7$

S_2:

V_5 $R_2 = R_1 <M_1> R_2$

V_3 $R_1 = R_{10} <M_2> R_8$

V_9 $R_8 = R_8 <A> R_3$

IIT Guwahati

24

So, you identify how many control signals are needed, what are the control signal is needed for FU, whether it is doing addition or plus-minus, you need the multiplexer at the FU input, you need the enable for the registers, you need the multiplex at the register input. So, these are the possible control signals, right.

So, for the example that I have here, I need only this is doing a plus or minus, so I need 1 bit here, I need 1 bit here, 1 bit here, 1 bit here, 2 bits here, 2 bits. And so, that means, 1, 2, 3, and 4, 7 bits, right. So, I need that 7-bit inputs control signal for multipliers are the function in 1. I need one input for the FU because another multiplier is doing multiplication only. So, there is no control signal needed every time it will do multiplication only, right.

And I have 11 registers, so I need 11 bits to enable the signal. And there are only 2 multiplexers, so I need 1 bit here, so I need 2 bits per the multiplex at the register input, ok. So, that is this is the total control signal. So, I need 21 control bits for my behaviour. So, this is the behaviour that I am talking about that example is for loop, right. So, for that behaviour, this is the final data path, and to control this data path I need 21 bits of control signals.

And what is the next thing? So, I identify that I need 21-bit control signals, but for each clock what is the value of that control signal that you have to determine. For example, here I already told you that this might be 0 this must be 0 in clock 1, and this must be 1 and 1 in clock 2, right. So, that means, each time you have to determine the value of the control signals, and this value you have to assign such a way that this value will execute these 4 operationally, right.

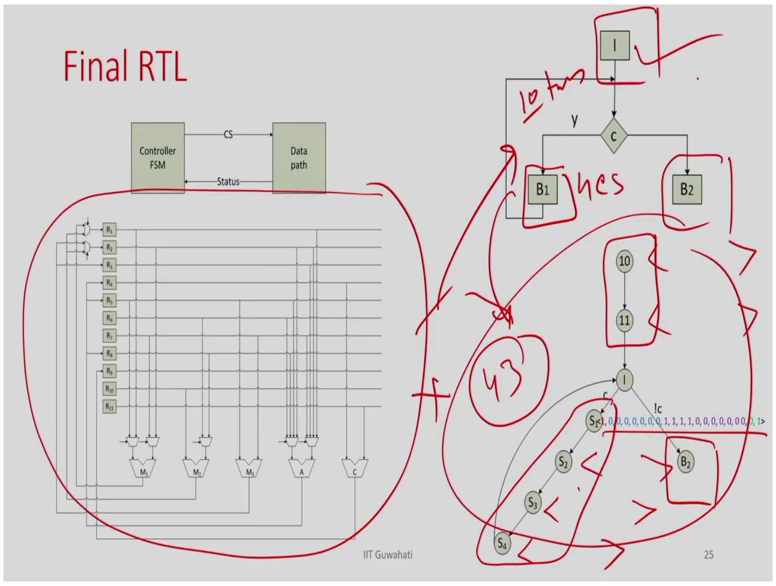
So, if I explain you see here, I am doing 1, say one means a plus since I am doing a plus here, I am putting 1 here, and since every multiplexer is taking the left input, right. So, that is why I am putting all 0, right, so all the multiplexer inputs are 0 here. You see here all are 0.

And since I am only updating register R 1 to R 3, R 4, these 4 bits are 1, and the rest of the register enable are 0. So, that means, in this clock, only R 1, R 2, R 3, and R 4 will be updated and not anything else. And this is for the Mux input of the FU and you can see here this R 1 is taking from M 1 and R 2 is taking from M 2. So, that is why I am giving 01. So, 0 means it will take from M 1, and 1 means it will equal M 2.

So, if I just give this control signal in this data path it will make sure that in the first clock only exactly these 4 operations to be executed. So, that means, the controller controls the data path and makes sure by giving this control signal value that the data path is executing these 4 operations. Similarly, if you give this control signal it will do these 3 operations in the second clock.

You can see here I am on updating R 1, R 2, and R 8. So, that is why giving R 1, R 2, and R 8 and the rest of the enable is 0. So, that means, I am not going to update the other registers, only these 3 registers I am going to update. And you can see here that I am giving 1 for the Mux, some cases 01, this is 01 and 01 and this is 11 because I am going to select the right one. So, that means, if I give this control signal the data path will execute these 3 operations. So, similarly, I identify the control signal for S 3 and S 4.

(Refer Slide Time: 58:19)



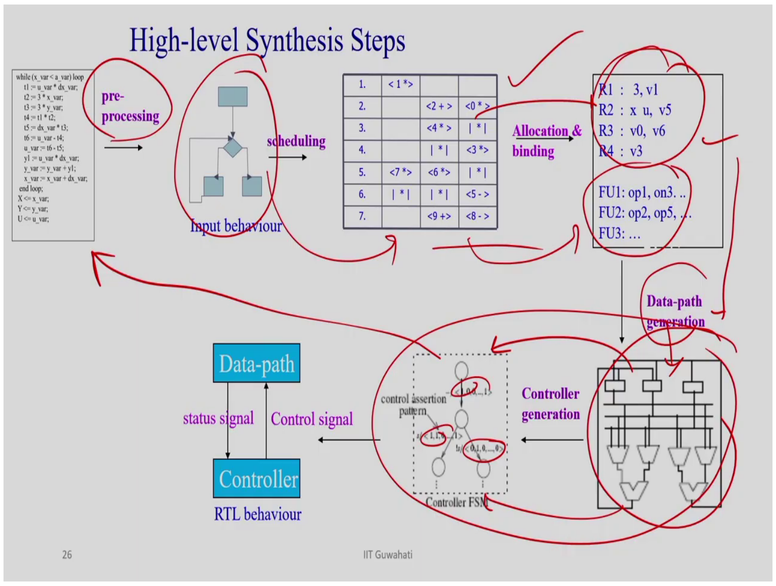
So, now, if you come back to this FSM. So, I have already determined say I need 4 control steps for this block, so I need 4 control steps. And in the control step, this is the control signal and similarly, I have a control signal here, I have a control signal here, I have a control signal here. And let us say for this block I need 2 cycles. So, I need a 2 cycle here and I need one claw in one state for this operation, ok. So, this is the final controller FSM, right. So, this is my final control FSM.

Here also there should be some control signals so that it actually does the operation to be done here and if you just combine this plus this, this, this is my data path and this is my controller, so this will do this operation, right. So, you can understand that this is my final RTL register transfer level behaviour. So, I have given this behaviour and I can determine this exact data path able to determine the exact the controller, and if these two in combination will execute this behaviour, right.

So, now, you can understand that this is 4 and I already told you that it will take 43 clocks, so the C code is untimed, it just gives the input, and you will get the output after some time. So, I do not know how many clocks is needed because untimed, but once I execute this behaviour into hardware these data path will make sure it will take a 43-clock cycle. I am assuming this loop will execute for 10 times, right. So, then it will give you the output. If the loop executes for 20 times it will be 83 and so on, ok.

So, you understand the overall idea. So, this concludes the class where we say that given a behaviour how this high-level synthesis goes step by step to realize a register transfer level behaviour from a C code, ok.

(Refer Slide Time: 60:08)

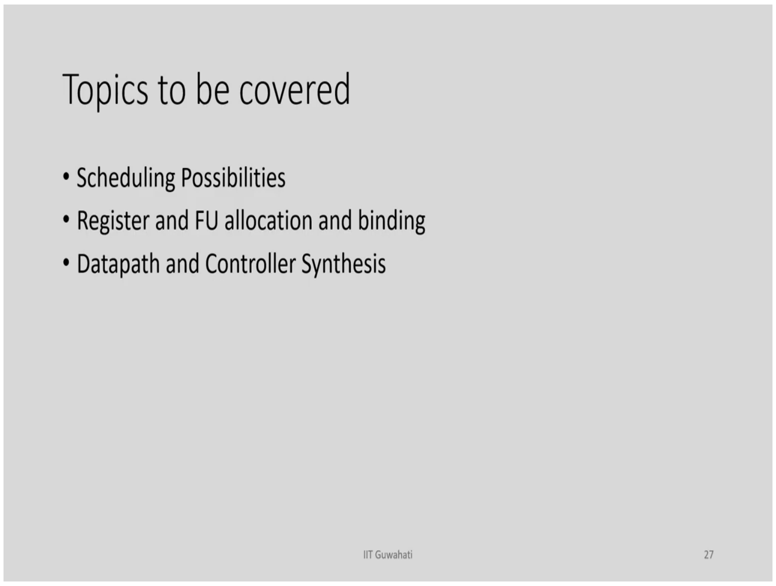


So, what are the; and we have understood that the steps are initially pre-processing which will identify the data path that cause CDFG, and the data dependency graphs and all, then schedule do this ascending time step to the operation, then you do the allocation and binding where you identify the registers and FUs. And then the data path gives you the exact data path connections, then the controller gives you the exact control signal is a timestamp and then finally, this is nothing but this, right. So, this is the same thing that is happening, ok.

So, this concludes the overall process. But what are the interesting things here, what are the things we will cover here in this course is this schedule is how I am going to do it automatically, right. How can I do this algorithmically? What can I do to this algorithmically?

How given this how can I do these things algorithmically? How given this data path and this information, how can I synthesize the controller? Right. So, these are the things I am going to discuss in detail in this course, that this is something how it will be automated, what are the various algorithms available there, and what are the algorithm to be applied there and what are they for a given target objective what algorithm should suits there and so on, right.

(Refer to Slide Time: 61:20)



Topics to be covered

- Scheduling Possibilities
- Register and FU allocation and binding
- Datapath and Controller Synthesis

IIT Guwahati 27

So, that should be what I am going to cover in this class in detail. And I hope you will enjoy this course, all these topics.

Thank you.