Digital Design with Verilog

Verilog - Synthesis

Lecture 25: Logic Synthesis with Verilog HDL – Part8 (Misc)

Learning Objectives

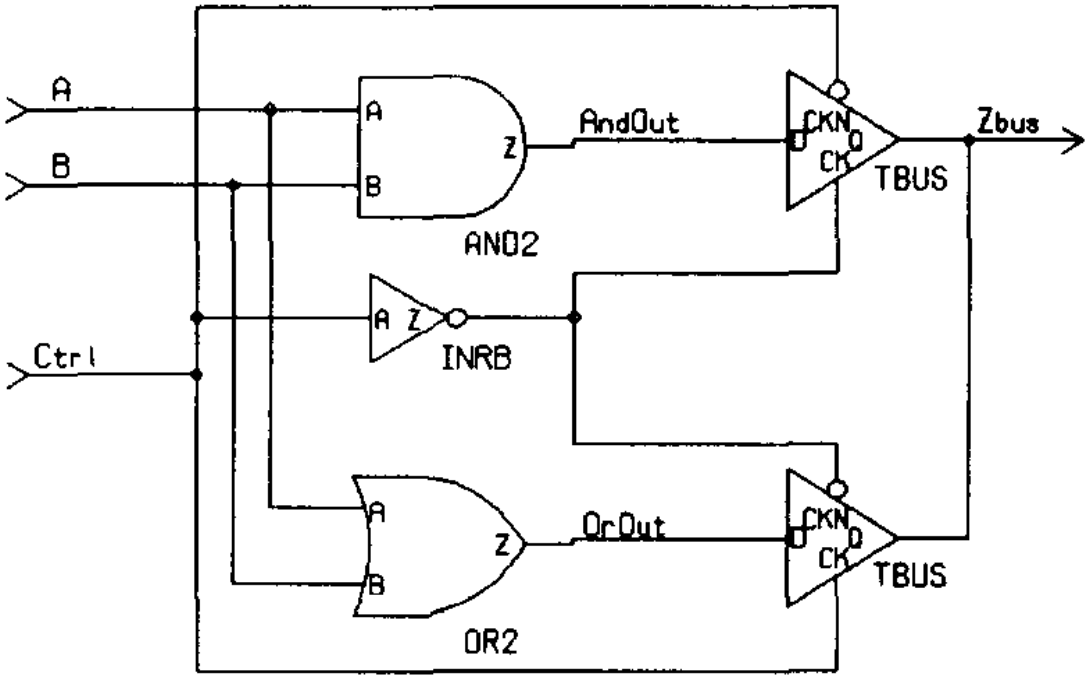
- Gate Level Modeling
- Parameterized Designs
- Memory Modeling
- Model Optimizations
  - Resource Allocation
  - Common Subexpressions
  - Moving Code
  - Common Factoring
  - Commutativity and Associativity

Gate Level Modeling

- Delays are ignored, else it is a direct translation.

```
module GateLevel(A,B, Ctrl, Zbus);
input A,B, Ctrl;
output Zbus;
and A1 (AndOut, A, B);
or O1(OrOut, A, B);
bufif0 B1 (Zbus, AndOut, Ctrl);
bufif0 B2 (Zbus, OrOut, !Ctrl);
endmodule
```

Gate Level Modeling



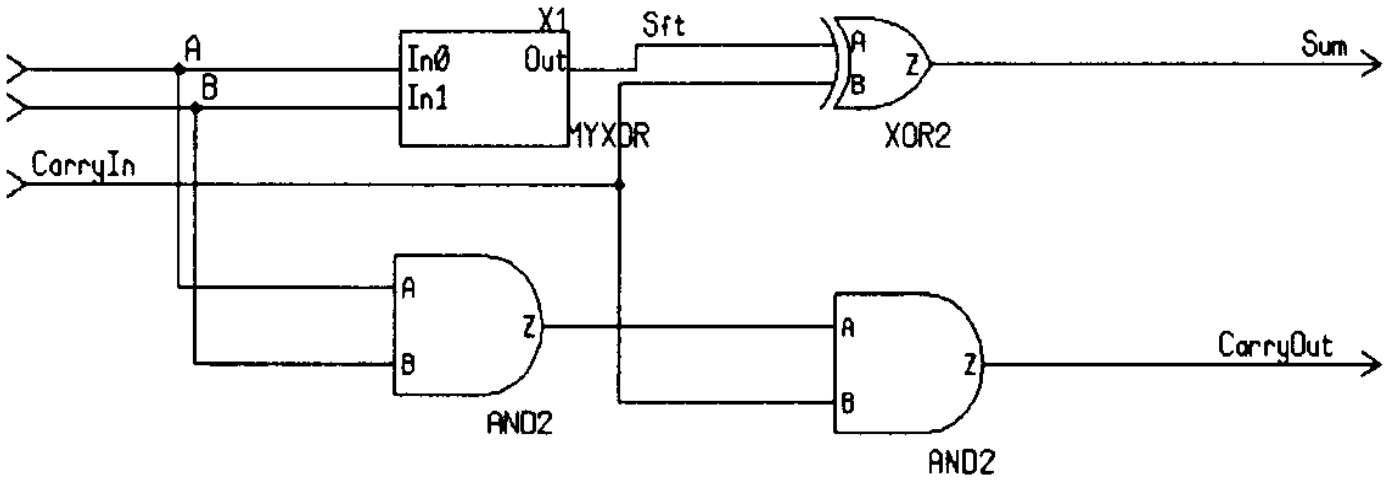
Gate instantiations.

Module Instantiations

- A module instantiation is treated like a primitive component

```
module FullAdderMix(A, B, CarryIn, Sum, CarryOut);
 input A, B, CarryIn;
 output Sum, CarryOut;
 wire Sft;
 MyXor X1 (.In0(A), .In1(B), .Out(Sft));
 assign CarryOut= A & B & CarryIn;
 assign Sum = Sft^CarryIn;
endmodule
```

Module Instantiations (Cont'd)



A module instance mixed with behavior.

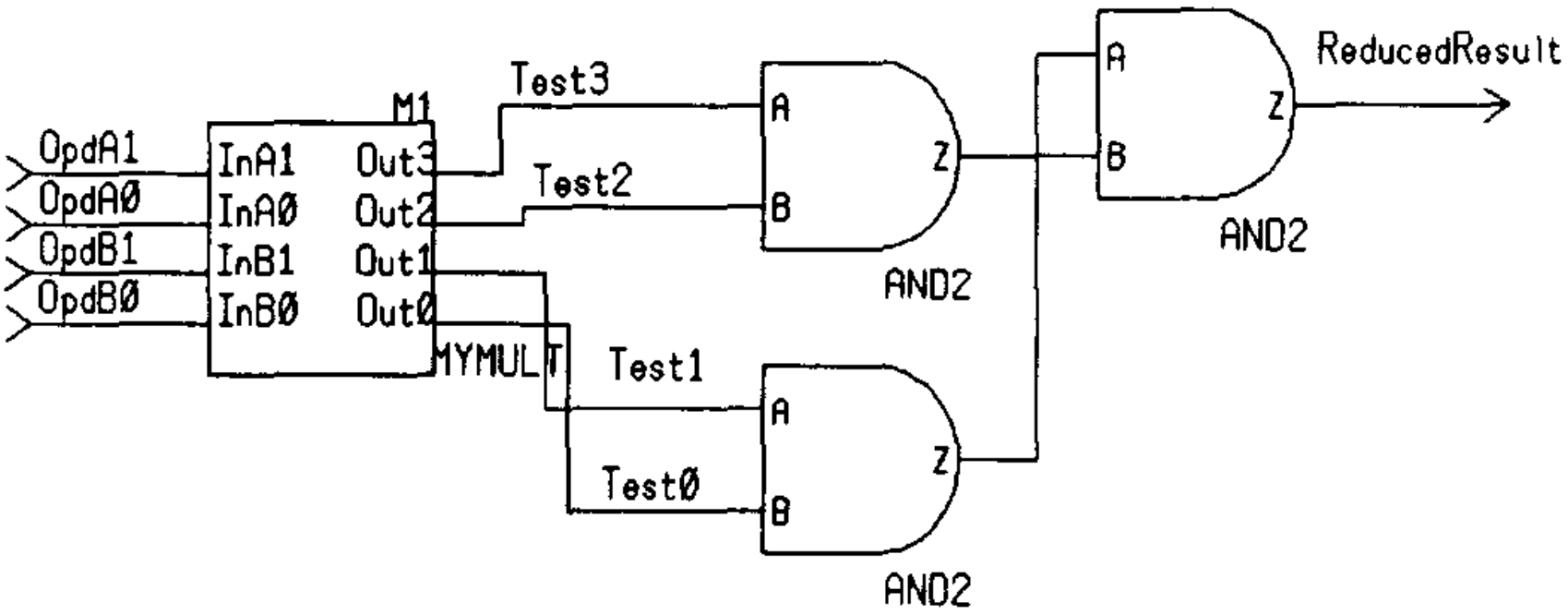
Using Predefined Blocks

- Designer not satisfied with the quality of circuits produced by a synthesis tool, then he uses his own block.
- Multipliers are very good examples.
- Custom-made Flip-flops may also be used for synthesis.

Using Predefined Blocks

```
module MultiplyAndReduce(OpdA,OpdB,ReducedResult);
 input [1:0] OpdA, OpdB;
 output ReducedResult;
 wire [3:0] Test;
 assign Test = OpdA* OpdB; // Multiply Operator
 assign ReducedResult= & Test;
endmodule

module PreDefMultiplyAndReduce(OpdA,OpdB,ReducedResult);
 input [1:0] OpdA,OpdB;
 output ReducedResult;
 wire [3:0] Test;
 MyMult M1 (.Input1(OpdA),.Input2(OpdB),Result(Test));
 assign ReducedResult= &Test;
endmodule
```

Using Predefined Blocks (Cont'd)

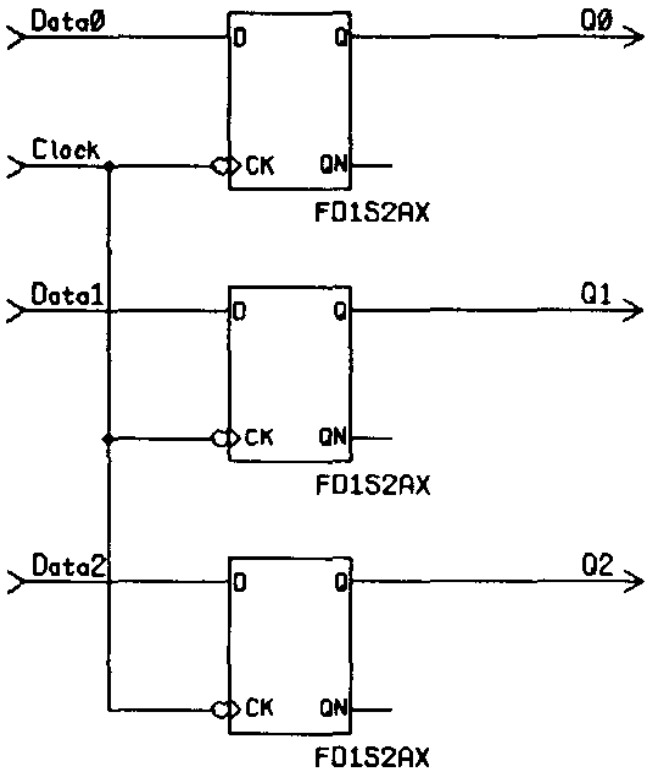


Instantiating a predefined multiplier.

Parameterized Designs

- Change of design structure without modifying internals.

```
module NbitRegister(Data,Clock,Q);
parameter N = 3;
input [N-1:0] Data;
input Clock;
output [N-1:0] Q;
reg[N-1:0] Q;
always @(negedgeClock)
Q <= Data;
endmodule
```

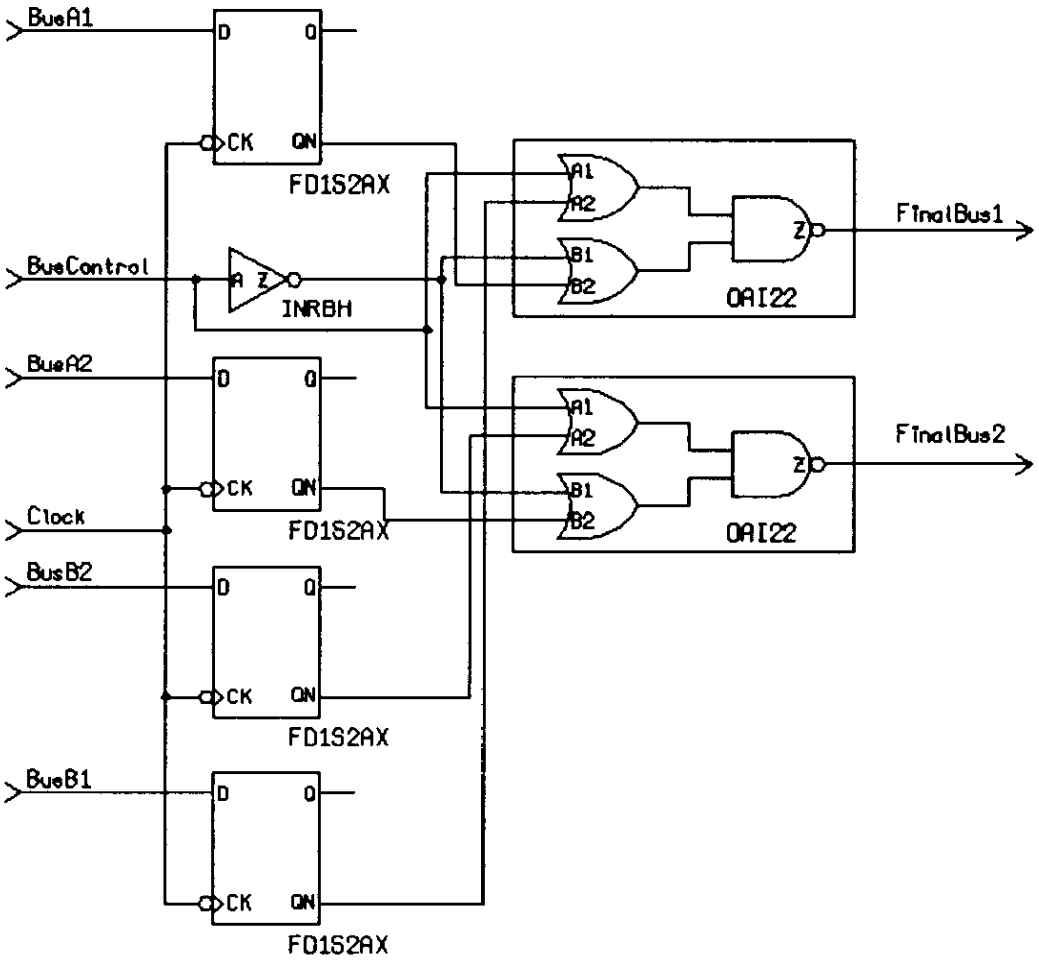


A parameterized register.

Parameterized Designs (Cont'd)

```
module ResolveBuses(BusA,BusB,BusControl,Clock,FinalBus);
parameter NBITS = 2;
input [NBITS:1] BusA,BusB;
input BusControl,Clock;
output [NBITS:1] FinalBus;
wire [NBITS:1] FinalBus;
RegisterFile#(NBITS) RfOne(.Data(BusA),.Clock(Clock),.Q(SavedA));
RegisterFile#(NBITS) RfTwo(.Data(BusB),.Clock(Clock),.Q(SavedB));
assign FinalBus= BusControl? SavedA: SavedB;
endmodule
```

Parameterized Designs (Cont'd)



Instantiating a parameterized register.

Modeling a Memory

- A memory is best modelled as a component since synthesis tools are not efficient at designing a memory.

```
module ROM (Clock, OutEnable, Address, Q, Qbar);
input Clock, OutEnable;
input [M - 1: 0] Address;
output [N - 1: 0] Q, Qbar;
// Memory description here (might not be
// synthesizable)
...
endmodule
```

```
module MyModule ( ... ) ;
wire Clk, Enable;
wire [M - 1: 0] Abus;
wire [N - 1: 0] Dbus;
ROM R1 (.Clock(Clk), .OutEnable(Enable),.Address(Abus), .Q(Dbus), .Qbar());
...
endmodule
```

Modeling a Memory (Cont'd)

- A register file can be modelled as two dimensional reg variable (a two-dimensional reg variable is referred to as a memory in Verilog HDL), which can be then synthesized.

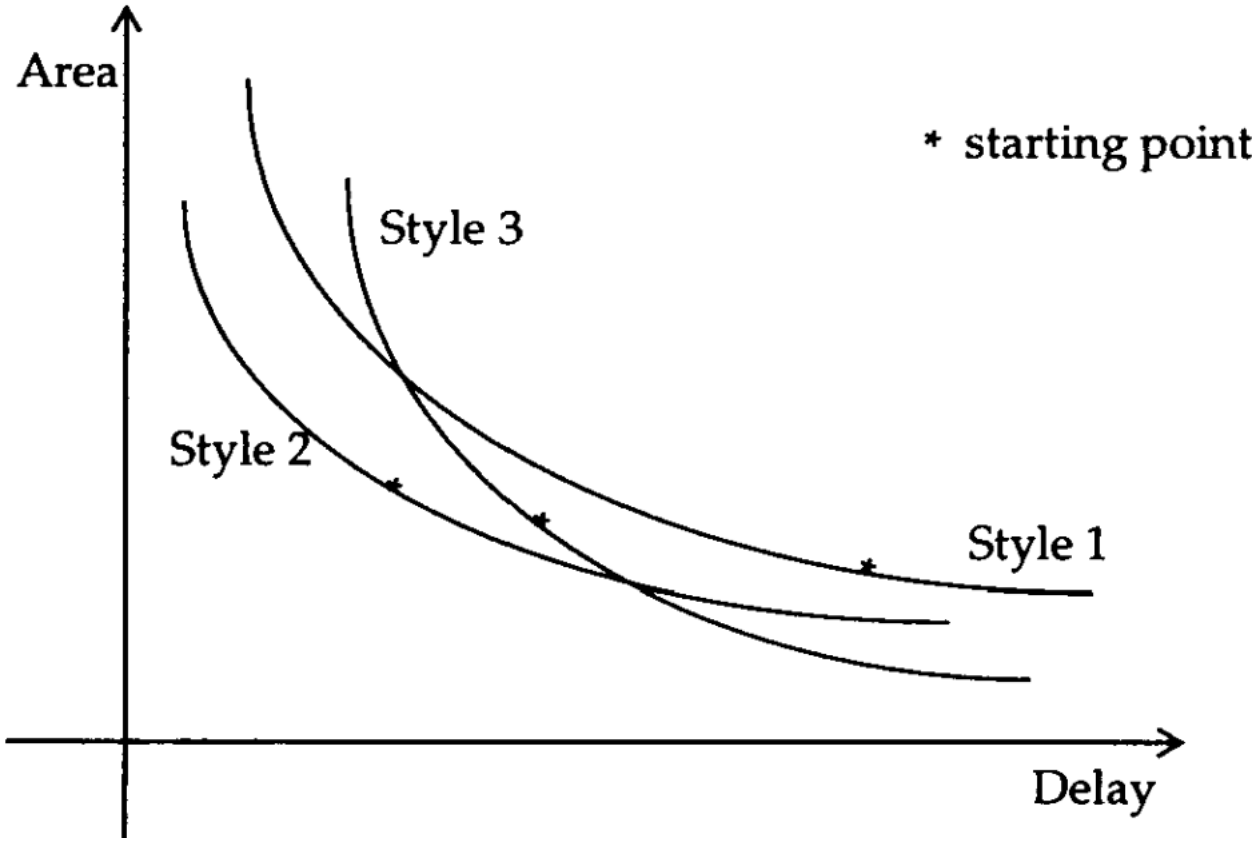
```
module RegFileWithMemory (Clk, ReadWrite, Index, Datain, DataOut);
parameter N = 2, M = 2;
input Clk, ReadWrite;
input [1:N] Index;     // Range need not be that large.
input [0:M - 1] DataIn;
output [0:M - 1] DataOut;
reg [0:M - 1] DataOut;

    reg [0:M - 1] RegFile [0:N-1];
    always @ (negedge Clk)
    if (ReadWrite)
        DataOut <= RegFile[Index];
    else
        RegFile[Index] <= DataIn;
endmodule
```

Model Optimizations

Optimizations

- Verilog code for optimal performance
  - The logic generated is very sensitive to the way a model is described.
  - Moving a statement from one place to another or splitting up expressions may have a profound impact on the generated logic
    - it might increase or decrease the number of synthesized gates and change its timing characteristics.
- Different styles give different timing delays and area for their corresponding synthesized circuit

Optimizations (Cont'd)



Different writing styles produce different area-delay trade-off.

Optimizations (Cont'd)

- Normally as Area increases – timing delay decreases
- Optimal balance between them is what one desires
  - Next set of slides explore some optimizations that may lead to better quality designs and reduced synthesis run-times.

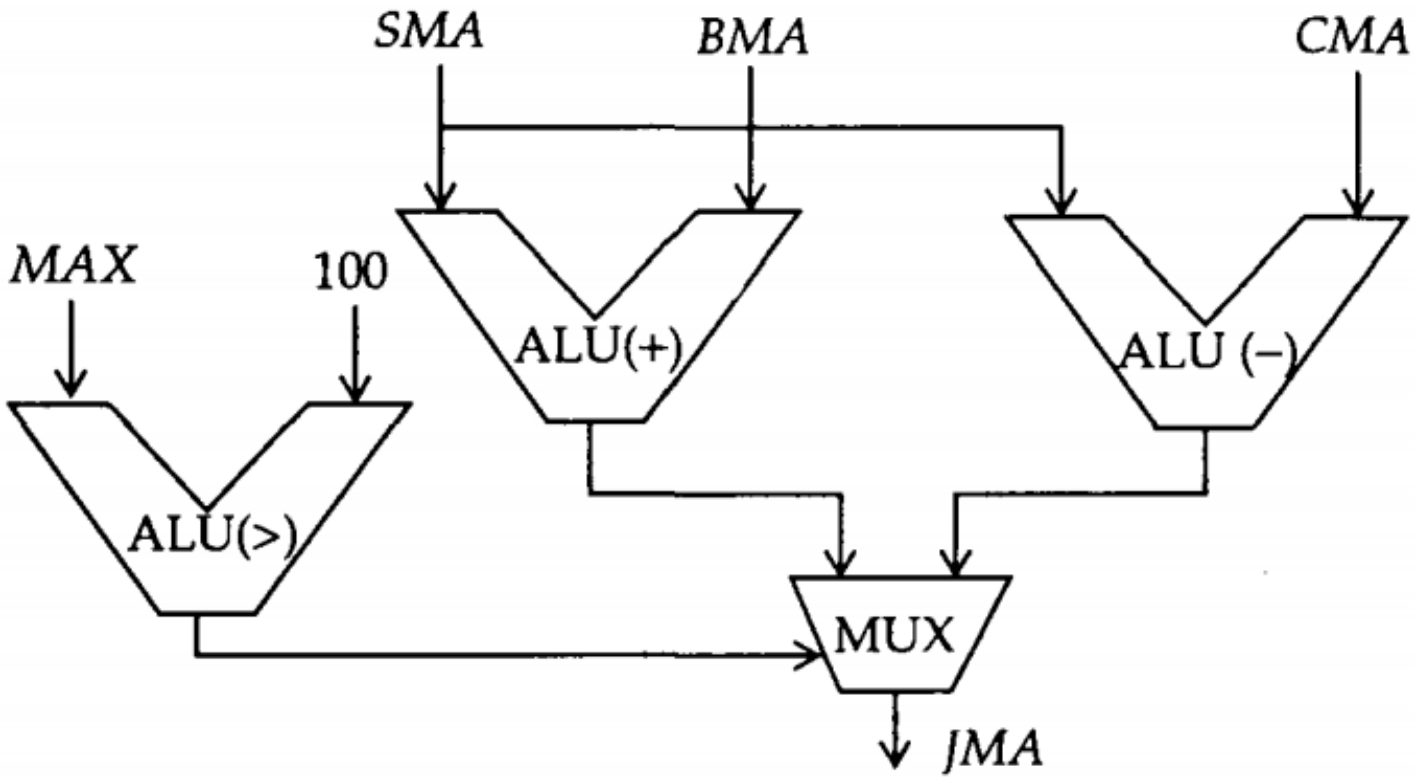
Resource Allocation

- Resource allocation refers to the process of sharing an arithmetic logic-unit (ALU) under mutually-exclusive conditions.

```
if (MAX > 100)
      JMA = SMA + BMA;
else
      JMA = SMA - CMA;
```

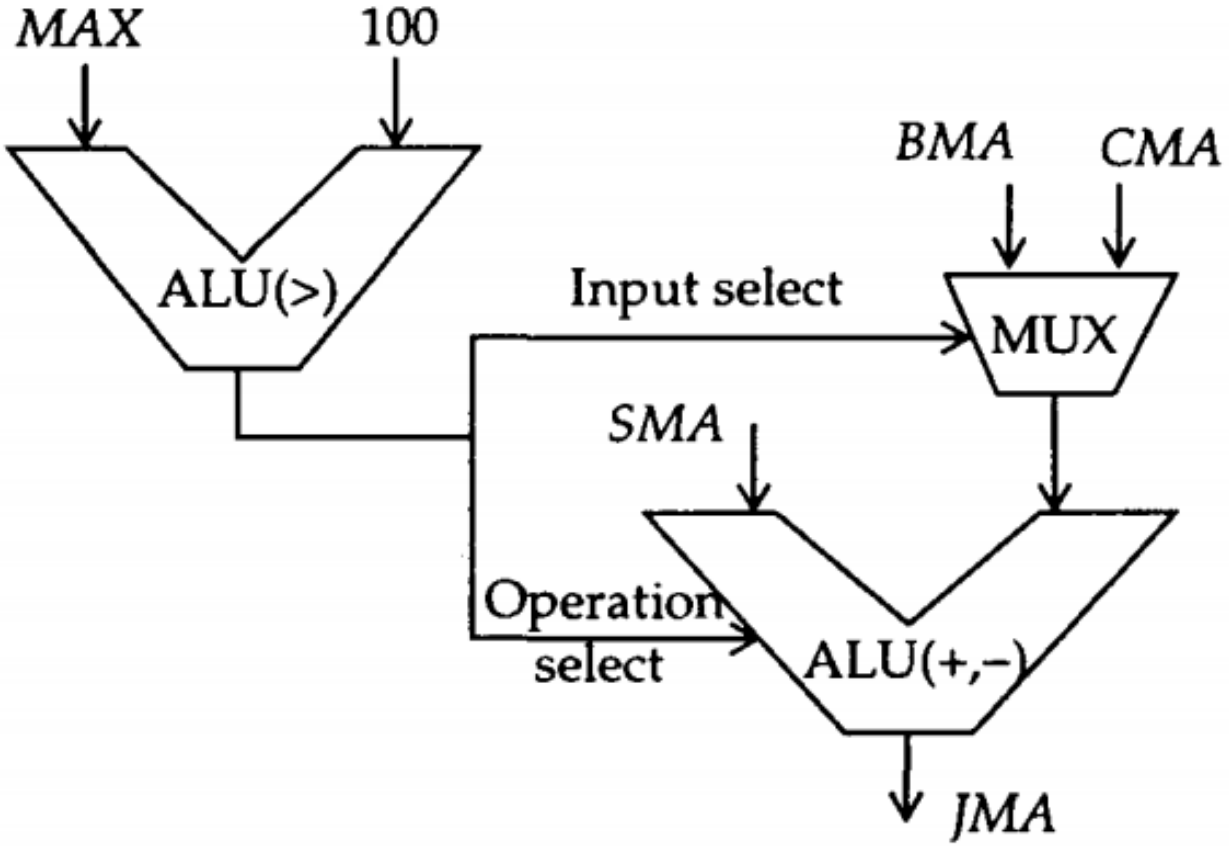
- If no resource allocation is specified then an adder and a subtractor may be synthesized.

Resource Allocation (Cont'd)



Without resource allocation

Resource Allocation (Cont'd)



With resource allocation

Resource Allocation (Cont'd)

- Tradeoff: Logic is reduced but delay is increased.

- The other operators that are shared by a synthesis tool are
  - Relational
  - Addition - Subtraction
  - Multiplication - Division

Resource Allocation (Cont'd)

ALU Sharing Possibilities:

- Same operator, Same operands --- definitely share A+B, A+B
- Same operator, one different operand --- Tradeoff (One multiplexer) (A+B, A+C)
- Same operator, different operands --- Tradeoff (Two Multiplexers) (A+B, C+D)

Resource Allocation (Cont'd)

ALU Sharing Possibilities:

- Different operators, Same operands – useful to share (A+B, A-B)
- Different operators, one different operand – Tradeoff (One multiplexer) (A+B, A-C)
- Different operators, different operands – Tradeoff (Two Multiplexers) (A+B, C-D)

Manual Resource Allocation

- Resource allocation may also be performed manually by rewriting the model.

```
if (!ShReg)
    DataOut = AddrLoad + ChipSelectN;
else if (ReadWrite)
    DataOut = ReadN + WriteN;
else
    DataOut = AddrLoad + ReadN;
```

- This yields three adders

Manual Resource Allocation (Cont'd)

```
// After manual resource allocation:
if (! ShReg)
begin
     Temp1 = AddrLoad;
     Temp2 = ChipSelectN;
end
else if (ReadWrite)
begin
     Temp1 = ReadN;
     Temp2 = WriteN;
end
else
begin
     Temp1 = AddrLoad;
     Temp2 = ReadN;
end
DataOut = Temp1 + Temp2;
```

- This guarantees only one adder and the multiplexers at the input ports of the adder are implied by the if statement.

Common Subexpressions

- Two expressions such that both will be executed always – they are not in different branches of a conditional construct etc.

```
Run= R1 + R2;
...
Car= R3 - (R1 + R2);
// Assume that the second assignment is executed every
// time the first statement is executed. Note that this
// assumption may not be true if either of the statements
// is inside an if statement or a case statement.
```

- Two adders will be used.

Common Subexpressions (Cont'd)

```
Car = R3- Run;
```

- One less adder --- Synthesis tool should identify such common subexpressions.

Moving Code

```
Car= ...
...
for (Count= 1; Count<= 5; Count= Count+ 1)
begin
...
Tip = Car - 6;
// Assumption: Car is not assigned a new value within
// the loop.
...
end
```

- This will give rise to six adders on loop unrolling

Moving Code (Cont'd)

```
Car= ...
...
Temp= Car- 6; // A temporary variable is introduced.
for (Count= 1; Count<= 5; Count= Count+ 1)
begin
...
Tip= Temp;
// Assumption: Car is not assigned a new value within
// the loop.
...
end
```

- This will give only one adder

- Such movement of the code should be performed by the designer to produce more efficient code.

Common Factoring

- Common factoring is the extraction of common subexpressions in mutually-exclusive branches of an if statement or a case statement.

```
if (Test)
     Ax = A & (B + C);
else
     By = (B + C) | T;
```

```
Temp = B + C;    // A temporary variable is introduced.
if (Test)
     Ax= A & Temp;
else
     By= Temp | T;
```

Commutativity and Associativity

```
Run= R1 + R2;
...
Car= R3 - (R2 + R1);
```

- Recognize that R1+R2 = R2+R1 and commutativity helps in identifying common subexpression.

```
Lam= A + B + C;
...
Bam= C + A - B;
```

- Three adders and one subtractor

Commutativity and Associativity (Cont'd)

```
Temp = C + A; // A temporary variable is introduced.
Lam = Temp + B;
Bam = Temp - B;
```

- Two adders and one subtractor

Other Optimizations

- Dead-code Elimination
  - Deletes the code that never gets executed.

```
if (2 > 4)
Oly = Sdy & Rdy;
```

- Constant Folding
  - Implies the computation of constant expressions during compile time as opposed to implementing logic and then allowing logic optimizer to eliminate the logic.

```
parameter FAC = 4;
...
Yak= 2 * FAC;
```

  - The expression is evaluated at compile time and assigned to Yak
  - Does not need a multiplier.

Flip-Flop/Latch Optimizations

- Flip-flop rules may vary from synthesis tool to synthesis tool.
- Care must be taken, else one lands up with lot of flip-flops
- See the next slide for example.

Flip-Flop Optimization

```
reg PresentState;
reg [0:3] Zout;
wire ClockA;
...
always @ (posedge ClockA)
case (PresentState)
0 :
begin
     PresentState <= 1;
     Zout <= 4'b0100;
end
1 :
begin
     PresentState <= 0;
     Zout <= 4'b0001;
end
endcase
```

Flip-Flop Optimization

- After synthesis there is one flip-flop for PresentState and four flip-flops for Zout.

- To reduce this

```
always @ (posedge ClockA)  // flip flop inference
case (PresentState)
0 : PresentState <= 1;
1 : PresentState <= 0;
endcase

always @ (PresentState)    // Combinational Logic
case (PresentState)
0 : Zout = 4'b0100;
1 : Zout = 4'b0001;
endcase
```

Latch Optimization

- From the synthesis tool find out all inferred latches and find out if it is necessary to infer a latch at all in each case.

- Remember: Latches are inferred when a ‘reg’ variable is not initialized before OR not assigned in one of the branches of ‘if’ or ‘case’ construct.

- Initialize or assign a value in all possible cases to reduce the number of latches

Latch Optimization

- Two ways of avoiding latch:

```
always @ (Probe or Count)
if (Probe)
     Luck = Count;
else          // Else clause added
     Luck= 0;
```

```
always @ (Probe or Count)
begin
Luck = 0;          //Value of variable is explicitly
                   //initialized.
if (Probe)
     Luck= Count;
end
```

Design Size

- Smaller designs synthesize faster
- Synthesis run-times, mainly logic optimization is exponential in design size.
- Always have ‘always’ statements less in size.
- No correlation between number of Verilog Lines and number of gates in the synthesized netlist.
- Maintain hierarchy so that smaller subcircuits are synthesized and integrated
- Use Macros – for memory/Multipliers etc. The synthesis tool leaves it as a black box which you may later integrate.

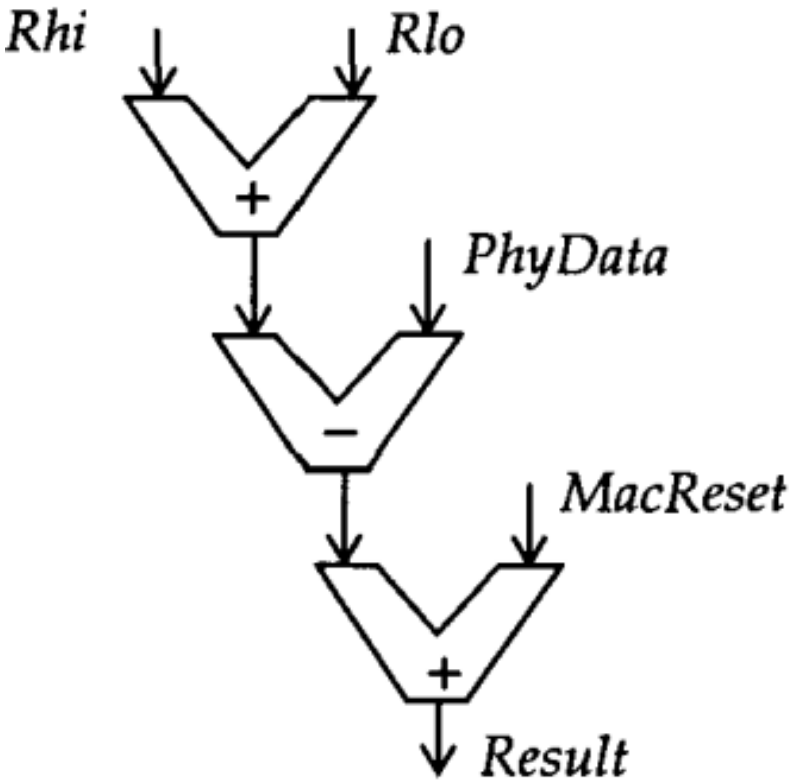
Using Parentheses

- When writing Verilog HDL code, the designer must be aware of the logic structure being generated.

- One such important point is the use of parentheses.

```
Result = Rhi + Rlo - PhyData + MacReset;
```

Using Parentheses (Cont'd)

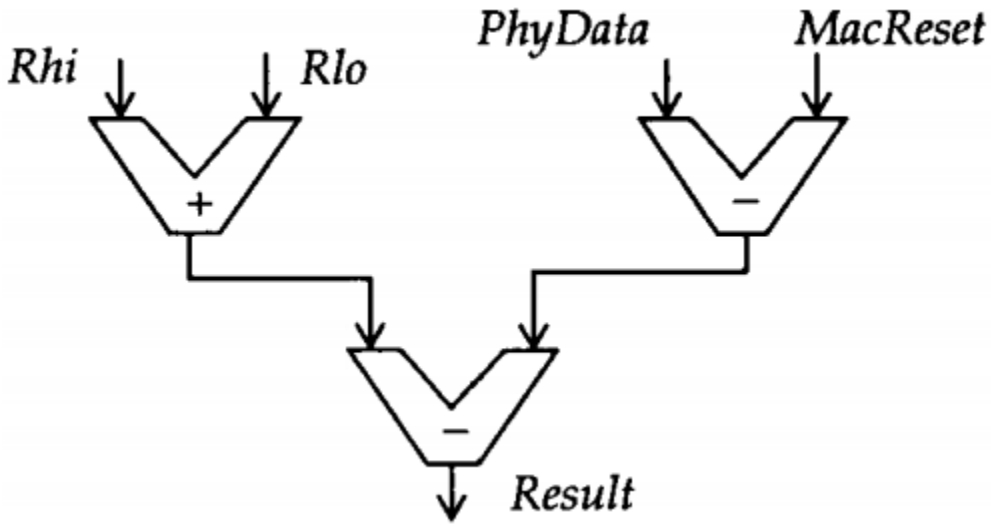


Without using parentheses

Three ALU delays

Using Parentheses (Cont'd)

```
Result = (Rhi + Rlo) - (PhyData- MacReset);
```



After using parentheses

Two ALU delays

References

- Chapter 2, 3 & 4, Verilog HDL Synthesis by J. Bhaskar

Thank you