




Gowin HDL Coding User Guide

SUG949-1.1E, 06/10/2021

Copyright © 2021 Guangdong Gowin Semiconductor Corporation. All Rights Reserved.

GOWIN, , Gowin, GowinSynthesis, and GOWINSEMI are trademarks of Guangdong Gowin Semiconductor Corporation and are registered in China, the U.S. Patent and Trademark Office, and other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders. No part of this document may be reproduced or transmitted in any form or by any denotes, electronic, mechanical, photocopying, recording or otherwise, without the prior written consent of GOWINSEMI.

Disclaimer

GOWINSEMI assumes no liability and provides no warranty (either expressed or implied) and is not responsible for any damage incurred to your hardware, software, data, or property resulting from usage of the materials or intellectual property except as outlined in the GOWINSEMI Terms and Conditions of Sale. All information in this document should be treated as preliminary. GOWINSEMI may make changes to this document at any time without prior notice. Anyone relying on this documentation should contact GOWINSEMI for the current documentation and errata.

Revision History

Date	Version	Description
08/31/2020	1.0E	Initial version published.
06/10/2021	1.1E	Infer case of single-ended BUF added.

Contents

Contents	i
List of Figures	v
List of Tables	vi
1 About This Guide	1
1.1 Purpose	1
1.2 Related Documents	1
1.3 Terminology and Abbreviations	1
1.4 Support and Feedback	2
2 Guideline on HDL Coding.....	3
2.1 General Requirements of HDL Coding.....	3
2.1.1 Coding for Hierarchical Synthesis	3
2.1.2 Pipeline Design Requirements	4
2.1.3 Compare If-Then-Else with Case Statements.....	4
2.1.4 Avoid Unintentional Latches	4
2.1.5 Global Reset and Local Reset.....	4
2.1.6 Clock Enable	4
2.1.7 Multiplexer	5
2.1.8 Bidirectional Buffer	5
2.1.9 Cross Clock Domains	5
2.1.10 BSRAM and SSRAM Coding	5
2.1.11 DSP Coding	6
2.1.12 Resource Sharing.....	7
2.2 Finite State Machine Coding Requirements.....	7
2.2.1 General Description.....	7
2.2.2 State Coding Method.....	7
2.2.3 Initialization and Default State for Safe State Machine	7
2.2.4 Full Case and Parallel Case.....	8
2.3 Lower Power Coding	8
2.4 Coding to Avoid Simulation/Synthesis Mismatch	8

2.4.1 Sensitivity List	8
2.4.2 Blocking/Nonblocking Assignment	8
2.4.3 Signal Fanout	8
3 Guideline on Buffer Coding.....	9
3.1 IBUF	9
3.2 TLVDS_IBUF	9
3.3 ELVDS_IBUF	10
3.4 OBUF	10
3.5 TLVDS_OBUF	11
3.6 ELVDS_OBUF	11
3.7 TBUF	11
3.8 TLVDS_TBUF	12
3.9 ELVDS_TBUF	12
3.10 IOBUF	12
3.11 TLVDS_IOBUF	13
3.12 ELVDS_IOBUF	13
4 Guideline on CLU Coding.....	15
4.1 LUT	15
4.1.1 Generated by LUT	15
4.1.2 Generated by Selector	15
4.1.3 Generated by Logical Operation	15
4.2 ALU	16
4.2.1 ADD	16
4.2.2 SUB	16
4.2.3 ADDSUB	17
4.2.4 NE	17
4.3 FF	17
4.3.1 DFFSE	17
4.3.2 DFFRE	18
4.3.3 DFFPE	18
4.3.4 DFFCE	19
4.4 LATCH	19
4.4.1 DLCE	19
4.4.2 DLPE	20
5 Guideline on BSRAM Coding.....	21
5.1 DPB/DPX9B	21
5.1.1 Read Address Through Register	21

5.1.2 Read Address Not Through Register	22
5.1.3 Assignment When Memory Defined	24
5.1.4 Assignment by readmemb/readmemh	26
5.2 SP/SPX9.....	28
5.2.1 Read Address Through Register	28
5.2.2 Read Address Not Through Register	29
5.2.3 Assignment When Memory Defined	30
5.2.4 Assignment by readmemb/readmemh	30
5.2.5 Generated by Shift Register	32
5.2.6 Generated by Decoder	33
5.3 SDPB/SDPX9B.....	34
5.3.1 Memory Without Initial Value	34
5.3.2 Assignment When Memory Defined	35
5.3.3 Assignment by readmemb/readmemh	36
5.3.4 Generated by Shift Register	37
5.3.5 Generated by Different Widths	38
5.3.6 Generated by Decoder	39
5.4 pROM/pROMX9	40
5.4.1 Assignment by Cases Statement	41
5.4.2 Assignment When Memory Defined	42
5.4.3 Assignment by readmemb/readmemh	43
6 Guideline on SSRAM Coding	45
6.1 RAM16S	45
6.1.1 Generated by Decoder	45
6.1.2 Generated by Memory	46
6.1.3 Generated by Shift Register	47
6.2 RAM16SDP	48
6.2.1 Generated by Decoder	48
6.2.2 Generated by Memory	49
6.2.3 Generated by Shift Register	50
6.3 ROM16	50
6.3.1 Generated by Decoder	50
6.3.2 Generated by Memory	51
7 Guideline on DSP Coding.....	53
7.1 Pre-adder.....	53
7.1.1 Pre-adding	53
7.1.2 Pre-subtracting	55
7.1.3 Shifting.....	57

7.2 Multiplier	58
7.3 ALU54D	60
7.4 MULTALU	61
7.4.1 $A*B\pm C$	61
7.4.2 $\Sigma(A*B)$	63
7.4.3 $A*B+CASI$	64
7.5 MULTADDALU	66
7.5.1 $A0*B0\pm A1*B1\pm C$	66
7.5.2 $\Sigma(A0*B0\pm A1*B1)$	68
7.5.3 $A0*B0\pm A1*B1+CASI$	70

List of Figures

Figure 2-1 Gowin Software IP Core Generator-Memory	6
Figure 2-2 Gowin Software IP Core Generator-DSP	6

List of Tables

Table 1-1 Abbreviations and Terminology	1
Table 6-1 Primitive Related with SSRAM.....	45

1 About This Guide

1.1 Purpose

This manual describes the guideline on Gowin HDL coding and its primitive implementation, which aims to help you be familiar with Gowin HDL coding to improve design efficiency.

1.2 Related Documents

The latest user guides are available on GOWINSEMI Website. You can find the related documents at www.gowinsemi.com:

- [SUG100](#), Gowin Software User Guide
- [SUG283](#), Gowin Primitive User Guide

1.3 Terminology and Abbreviations

Table 1-1 shows the abbreviations and terminology used in this manual.

Table 1-1 Abbreviations and Terminology

Terminology and Abbreviations	Meaning
HDL	Hardware Description Language
FSM	Finite State Machine
DRC	Design Rule Check
CLU	Configurable Logic Unit
CLS	Configurable Logic Section
BSRAM	Block Static Random Access Memory
SSRAM	Shadow Static Random Access Memory
DSP	Digital Signal Processing

1.4 Support and Feedback

Gowin Semiconductor provides customers with comprehensive technical support. If you have any questions, comments, or suggestions, please feel free to contact us directly by the following ways.

Website: www.gowinsemi.com

E-mail: support@gowinsemi.com

2 Guideline on HDL Coding

2.1 General Requirements of HDL Coding

2.1.1 Coding for Hierarchical Synthesis

Complex system designs require a hierarchical approach as opposed to the use of single modules. A hierarchical coded design can be synthesized as a whole or have each module synthesized separately. When the design is synthesized as a whole, it can be synthesized as either a flat module or multiple hierarchical modules.

Each methodology has its associated advantages and disadvantages. Hierarchical coding is preferable for complex system designs because designs in smaller blocks are easier to keep track of and reduce the design period by allowing the reuse of design modules. Here are some tips for building hierarchical structures.

- The top level should be only used to call the submodule. It is better to implement control logic in submodules.
- Any I/O buffer instantiations should be at the top level of the hierarchy.
- All input, output, or bidirectional pins instantiations should be at the top level of the hierarchy.
- The tristate statement for bidirectional pins should be written at the top module.
- Ensure that the output signals of all modules use registers. The advantages are as follows:
 - The related logics placed in one module can realize resource sharing and key paths optimization.
 - Divide irrelevant logics into different modules, then you can use different optimization strategies, such as speed first or area first.

2.1.2 Pipeline Design Requirements

Pipeline design can improve design performance by restructuring a long data path into several levels of logic and breaking it up over multiple clock cycles. Pipeline structure is an effective way to improve data path speed; however, it should be noted that the data path latency is added.

2.1.3 Compare If-Then-Else with Case Statements

The if-then-else statement generates priority logic, whereas the case statement implements parallel logic. If-then-else statements can contain a set of different expressions, but a case statement can only contain one expression. If-then-else statements and case statements are equivalent if the conditions are mutually exclusive.

2.1.4 Avoid Unintentional Latches

You should avoid using latches in FPGA designs. The synthesis tool can implement latches through combinational logic loops, inevitably resulting in wasted resources and degraded performance, and the combinational logic loops bring great difficulty to static timing analysis.

The synthesis tool generates latches when conditional expressions are incomplete; for example, an if-then-else statement without an else clause or a case statement without a default clause. In order to avoid unnecessary latches, all conditions need to be traversed in the condition statement.

2.1.5 Global Reset and Local Reset

A global set/reset (GSR) network is built into Gowin devices, and there is a direct connection to the core logic. It can be used as an asynchronous/synchronous set/reset. The registers in CLU and I/O can be individually configured to use the GSR. The global reset resource provides a convenient mechanism by which design components can be reset without using any common routing resources. There are two primary ways to use GSR hardware resource in your design: use the GSR to reset all components on your FPGA or use the reset with maximum fanout as the global reset to save routing resources. Local reset usually has smaller fanout. It is recommended to use common routing as a reset signal.

2.1.6 Clock Enable

Gating clock should be avoided in FPGA designs because it can cause timing issues, such as unexpected clock skews. The CLS structure in Gowin devices contains dedicated clock enable signals. You can use clock enable as a better alternative to achieve the same functions without worrying about timing issues. You should take the following into consideration when using the clock enable functionality of Gowin devices.

- Clock enable is only supported by flip-flops, not latches.

- Each CLS in the same CLU shares one clock enable signal;
- All flip-flops have a positive clock enable input;
- The clock-enable input has higher priority than the synchronous set/reset.

2.1.7 Multiplexer

The flexible configurations of LUTs can realize 2-to-1, 3-to-1, 4-to-1, or 5-to-1 multiplexers, etc. You can create larger multiplexers by programming multiple four-input LUTs.

2.1.8 Bidirectional Buffer

Using bidirectional buffer allows for fewer device pins, controlling output enable, and reducing power. You can disable automatic I/O insertion in your synthesis tool and then manually instantiate the I/O pads for specific pins as needed.

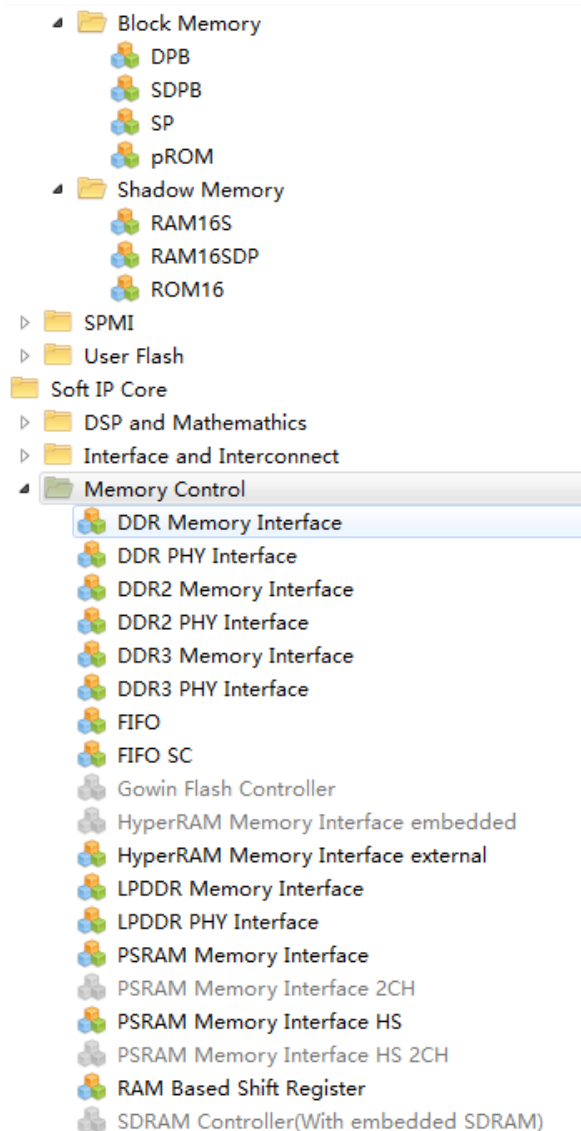
2.1.9 Cross Clock Domains

When the data path crosses from one clock domain to another, the transmission of signals across the clock domain needs to be handled with care to avoid the generation of metastability. For single bit signals, it is suggested to use three-stage register to eliminate metastability. For multiple bits signals, asynchronous FIFOs are recommended.

2.1.10 BSRAM and SSRAM Coding

Although the RTL description of random memory is customizable and very intuitive, different codings may generate different synthesis results.

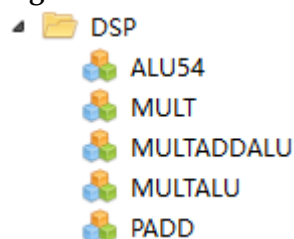
For Gowin devices, it's recommended to use the IP Core Generator in Gowin software to generate BSRAM and SSRAM. Gowin devices support multiple memory structures, including dual-port RAM, single port RAM, semi dual-port RAM, ROM, synchronous FIFO and asynchronous FIFO, as shown in Figure 2-1.

Figure 2-1 Gowin Software IP Core Generator-Memory

2.1.11 DSP Coding

Although the RTL description of DSP is customizable and very intuitive, different codings may generate different synthesis results.

For Gowin devices, it is recommended to use the IP Core Generator in Gowin software to generate DSP. Gowin devices support multiple DSP structures, including MULT, MULTALU, MULTADDALU, and PADD.

Figure 2-2 Gowin Software IP Core Generator-DSP

2.1.12 Resource Sharing

Resource sharing can save resources, but it can increase routing resources and cause local congestion. If timing issues are caused, turn the global resource sharing off, and use attributes on specific modules

```
/*synthesis syn_sharing = "on" */.
```

2.2 Finite State Machine Coding Requirements

A finite state machine advances from the current state to the next state at the clock edge. This following section discusses the methods and strategies for state machine coding.

2.2.1 General Description

There are two ways to implement finite state machine. One is to process state-jump and state output simultaneously in one process; the other is to process state-jump and state output in two independent processes respectively. It is recommended to use the second way. It is easier to be read and modified, and it will not cause extra delay for output directly without registering.

2.2.2 State Coding Method

There are several ways to code a state machine, including binary, one-hot coding, and gray coding. State machines with binary code and gray code have fewer of flip-flops but more combinatorial logics, whereas one-hot coding is exactly the opposite.

The greatest advantage of one-hot coding is that only one bit is required for state comparison. As a result, it decreases the coding logic. Although one-hot coding uses more bits, i.e. more flip-flops for same states, its coding circuit saves an equivalent area to offset the area consumed by flip-flops.

For small state machines, less than five states, binary code and gray code are recommended. For state machines with more than 5 states, it is recommended to use one-hot.

2.2.3 Initialization and Default State for Safe State Machine

A state machine must be initialized to an active state after power on. You can initialize it by power on reset or global reset. In the same manner, a state machine should have a default state to ensure that the state machine does not go into an illegal state. This could happen if all the possible combinations have been clearly defined in the design source code.

2.2.4 Full Case and Parallel Case

RTL has attributes to define the default state. Full case attribute can be used to ensure no illegal state. Parallel case attributes can be used to ensure that all the statements in a case statement are mutually exclusive and only one case can be true at a time.

2.3 Lower Power Coding

Optimize area to reduce logic and routing utilization, and then to reduce power. It is recommended to use the IP Core Generator in Gowin Software to call the basic cells in Gowin devices for the most power-efficient (least area and resources) implementation. Eliminate known glitches for power reasons, reduce I/O toggle rate, and enter into the sleep state to reduce system power.

2.4 Coding to Avoid Simulation/Synthesis Mismatch

Certain coding can lead to synthesis result that differs from simulation. This is caused by error information that is ignored by a simulator and can not be passed to the synthesis tool. It can usually be detected by running BKM check. Therefore, Gowin coding is recommended.

2.4.1 Sensitivity List

Sensitivity lists must contain all input and output signals to avoid mismatches between simulation and synthesis logic.

2.4.2 Blocking/Nonblocking Assignment

Use blocking assignments to generate combinational logic; Use nonblocking assignments to generate sequential logic.

2.4.3 Signal Fanout

Signal fanout control is designed to maintain reasonable post-synthesis fanouts. The synthesis tool reduces signal fanout by duplicating circuits. Use `syn_maxfan` for specific signals to acquire better timing closure. Gowin FPGA device architectures are designed to handle high fanout clock signal with dedicated clock and handle high fanout reset signal with dedicated global reset network. However, synthesis tool tends to replicate logic. This type of logic replication occupies more resources in devices and makes performance checking more difficult. Use `syn_maxfan` flexibly based on actual conditions to avoid the generation of much logic replication.

3 Guideline on Buffer Coding

Buffer can be divided to single-ended buffer, emulated LVDS (ELVDS) and true LVDS (TLVDS) according to their different functions. It needs to add attribute constraints to implement the primitive of emulated LVDS and true LVDS, and it is recommended to code in instantiation.

3.1 IBUF

Input Buffer (IBUF), the coding can be in following two ways:

The First Way:

```
module ibuf (o, i);  
  input i ;  
  output o ;  
  assign o = i ;  
endmodule
```

The Second Way:

```
module ibuf (o, i);  
  input i ;  
  output o ;  
  buf IB (o, i);  
endmodule
```

3.2 TLVDS_IBUF

True LVDS Input Buffer (TLVDS_IBUF). It needs to add attribute constraints to implement the primitive, and the coding is as follows:

```
module tlvsd_ibuf_test(in1_p, in1_n, out);  
  input in1_p/* synthesis syn_tlvds_io = 1*/;  
  input in1_n/* synthesis syn_tlvds_io = 1*/;  
  output reg out;
```

```

always@(in1_p or in1_n) begin
    if (in1_p != in1_n) begin
        out = in1_p;
    end
end
endmodule

```

3.3 ELVDS_IBUF

Emulated LVDS Input Buffer (ELVDS_IBUF). It needs to add attribute constraints to implement the primitive, and the coding is as follows:

```

module elvds_ibuf_test (in1_p, in1_n, out);
input in1_p/* synthesis syn_elvds_io = 1*/;
input in1_n/* synthesis syn_elvds_io = 1*/;
output reg out;
always@(in1_p or in1_n)begin
    if (in1_p != in1_n) begin
        out = in1_p;
    end
end
endmodule

```

3.4 OBUF

Output Buffer (OBUF), the coding can be in the following two ways:

The First Way:

```

module obuf (o, i);
input i ;
output o ;
assign o = i ;
endmodule

```

The Second Way:

```

module obuf (o, i);
input i ;
output o ;
buf OB (o, i);
endmodule

```

3.5 TLVDS_OBUF

True LVDS Output Buffer (TLVDS_OBUF). It needs to add attribute constraints to implement the primitive, and the coding is as follows:

```
module tlvsd_obuf_test(in,out1,out2);  
input in;  
output out1/* synthesis syn_tlvds_io = 1*/;  
output out2/* synthesis syn_tlvds_io = 1*/;  
assign out1 = in;  
assign out2 = ~out1;  
endmodule
```

3.6 ELVDS_OBUF

Emulated LVDS Output Buffer (ELVDS_OBUF). It needs to add attribute constraints to implement the primitive, and the coding is as follows:

```
module elvds_obuf_test(in,out1,out2);  
input in;  
output out1/* synthesis syn_elvds_io = 1*/;  
output out2/* synthesis syn_elvds_io = 1*/;  
assign out1= in;  
assign out2 = ~out1;  
endmodule
```

3.7 TBUF

Output Buffer with Tri-state Control (TBUF), active-low, and the coding can be in the following two ways:

The First Way:

```
module tbuf (in, oen, out);  
input in, oen;  
output out;  
assign out= ~oen ? in :1'bz;  
endmodule
```

The Second Way:

```
module tbuf (out, in, oen);  
input in, oen;  
output out;
```

```
bufif0 TB (out, in, oen);
endmodule
```

3.8 TLVDS_TBUF

True LVDS Tristate Buffer (TLVDS_TBUF), active-low, and it needs to add attribute constraints to implement the primitive, and the coding is as follows:

```
module tlvsd_tbuf_test(in, oen, out1,out2);
input in;
input oen;
output out1/* synthesis syn_tlvds_io = 1*/;
output out2/* synthesis syn_tlvds_io = 1*/;
assign out1 = ~oen ? in : 1'bz;
assign out2 = ~oen ? ~in : 1'bz;
endmodule
```

3.9 ELVDS_TBUF

Emulated LVDS Tristate Buffer (ELVDS_TBUF), active-low, and it needs to add attribute constraints to implement the primitive, and the coding is as follows:

```
module elvds_tbuf_test(in, oen, out1,out2);
input in, oen;
output out1/* synthesis syn_elvds_io = 1*/;
output out2/* synthesis syn_elvds_io = 1*/;
assign out1 = ~oen ? in : 1'bz;
assign out2 = ~oen ? ~in : 1'bz;
endmodule
```

3.10 IOBUF

Bi-Directional Buffer (IOBUF), when OEN is high, it is as input buffer; when OEN is low, it is as output buffer. The coding can be in the following two ways:

The First Way:

```
module iobuf (in, oen, io, out);
input in,oen;
output out;
inout io;
```

```

assign io= ~oen ? in :1'bz;
assign out = io;
endmodule

```

The Second Way:

```

module iobuf (out, io, i, oen);
input i,oen;
output out;
inout io;
buf OB (out, io);
bufif0 IB (io,i,oen);
endmodule

```

3.11 TLVDS_IOBUF

True LVDS Bi-Directional Buffer (TLVDS_IOBUF), when OEN is high, it is as true LVDS input buffer; when OEN is low, it is as true LVDS output buffer. It needs to add attribute constraints to implement the primitive, and the coding is as follows:

```

module tlvs_iobuf(o, io, iob, i, oen);
output reg o;
inout io /* synthesis syn_tlvds_io = 1 */;
inout iob /* synthesis syn_tlvds_io = 1 */;
input i, oen;
bufif0 ib(io, i, oen);
notif0 yb(iob, i, oen);
always @(io or iob)begin
    if (io != iob)begin
        o <= io;
    end
end
endmodule

```

3.12 ELVDS_IOBUF

Emulated LVDS Bi-Directional Buffer (ELVDS_IOBUF), when OEN is high, it is as emulated LVDS input buffer; when OEN is low, it is as emulated LVDS output buffer. It needs to add attribute constraints to implement the primitive, and the coding is as follows:

```

module elvds_iobuf(o, io, iob, i, oen);

```

```
output o;
inout io /* synthesis syn_elvds_io = 1 */;
inout iob /* synthesis syn_elvds_io = 1 */;
input i, oen;
reg o;
bufif0 ib(io, i, oen);
notif0 yb(iob, i, oen);
always @(io or iob)begin
    if (io != iob)begin
        o <= io;
    end
end
endmodule
```

4 Guideline on CLU Coding

Configurable logic unit (CLU) is the basic unit of FPGA product. CLU module can realize the functions of MUX/LUT/ALU/FF/LATCH modules.

4.1 LUT

LUT1, LUT2, LUT3 and LUT4 are commonly used, and the difference is that the LUT has different bit widths. Its implementation can be as follows:

4.1.1 Generated by LUT

```
module rtl_LUT4 (f, i0, i1,i2,i3);  
  parameter INIT = 16'h2345;  
  input i0, i1, i2,i3;  
  output f;  
  assign f=INIT[{i3,i2,i1,i0}];  
endmodule
```

4.1.2 Generated by Selector

```
module rtl_LUT3 (f,a,b,sel);  
  input a,b,sel;  
  output f;  
  assign f=sel?a:b;  
endmodule
```

4.1.3 Generated by Logical Operation

```
module top(a,b,c,d,out);  
  input [3:0]a,b,c,d;  
  output [3:0]out;
```



```
assign out=a&b&c|d;  
endmodule
```

4.2 ALU

Here ALU is a 2-input arithmetic logic unit. The synthesis tool can generate ADD/SUB/ADDSUB/NE.

4.2.1 ADD

Take 4-bit full adder and 4-bit semi-adder as examples to introduce the ADD function of ALU.

4-bit Full Adder

4-bit full adder coding is as follows:

```
module add(a,b,cin,sum,cout);  
input [3:0] a,b;  
input cin;  
output [3:0] sum;  
output cout;  
assign {cout,sum}=a+b+cin;  
endmodule
```

4-bit Half Adder

4-bit half added coding is as follows:

```
module add(a,b,sum,cout);  
input [3:0] a,b;  
output [3:0] sum;  
output cout;  
assign {cout,sum}=a+b;  
endmodule
```

4.2.2 SUB

```
module sub(a,b,sub);  
input [3:0] a,b;  
output [3:0] sub;  
assign sub=a-b;  
endmodule
```

4.2.3 ADDSUB

```
module addsub(a,b,c,sum);
input [3:0] a,b;
input c;
output [3:0] sum;
assign sum=c?(a-b):(a+b);
endmodule
```

4.2.4 NE

```
module ne(a, b, cin, cout);
input [11:0] a, b;
input cin;
output cout;
assign cout = (a != b) ? 1'b1 : 1'b0;
endmodule
```

4.3 FF

Flip-flop is a basic cell in the timing circuit. Timing logic in FPGA can be implemented through an FF structure. The commonly used FF includes DFF, DFFE, DFFS, DFFSE, etc. The differences between them are reset and triggering modes. This section takes DFFSE, DFFRE, DFFPE, and DFFCE as examples to describe the implementation of Flip-flop. For the implementation of other Flip-flop primitives, see these Flip-flops. When coding the definition of the reg signal, it is recommended to add the initial value, the initial value of different types of registers can be referred to [UG288](#), Gowin Configurable Function Unit (CFU) User Guide.

4.3.1 DFFSE

```
module dffse_init1 (clk, d, ce, set, q );
input clk, d, ce, set;
output reg q=1'b1;
always @(posedge clk)begin
    if (set)begin
        q <= 1'b1;
    end
    else begin
        if (ce)begin
```

```
        q <= d;
    end
end
end
endmodule
```

4.3.2 DFFRE

```
module dffre_init1 (clk, d, ce, rst, q );
input clk, d, ce, rst;
output reg q= 1'b0;
always @(posedge clk)begin
    if (rst)begin
        q <= 1'b0;
    end
    else begin
        if (ce)begin
            q <= d;
        end
    end
end
end
endmodule
```

4.3.3 DFFPE

```
module dffpe_test (clk, d, ce, preset, q );
input clk, d, ce, preset;
output reg q= 1'b1;
always @(posedge clk or posedge preset )begin
    if (preset)begin
        q <= 1'b1;
    end
    else begin
        if (ce)begin
            q <= d;
        end
    end
end
end
```

```

    end
endmodule

```

4.3.4 DFFCE

```

module dffce_test (clk, d, ce, clear, q );
input clk, d, ce, clear;
output reg q= 1'b0;
always @(posedge clk or posedge clear )begin
    if (clear)begin
        q <= 1'b0;
    end
    else begin
        if (ce)begin
            q <= d;
        end
    end
end
endmodule

```

4.4 LATCH

LATCH is a memory cell circuit and its status can be changed under the specified input level. This section takes DLCE and DLPE as examples to describe the implementation of latch. For the implementation of other latch primitives, see these latches. When coding the definition of the reg signal, it is recommended to add the initial value, the initial value of different types of latches can be referred to [UG288](#), Gowin Configurable Function Unit (CFU) User Guide.

4.4.1 DLCE

```

module rtl_DLCE (Q, D, G, CE, CLEAR);
input D, G, CLEAR, CE;
output reg Q=1'b0;
always @(D or G or CLEAR or CE ) begin
    if (CLEAR)begin
        Q <= 1'b0;
    end
end

```

```
        else begin
            if (G && CE)begin
                Q <= D;
            end
        end
    end
end
endmodule
```

4.4.2 DLPE

```
module rtl_DLPE (Q, D, G, CE, PRESET);
input D, G, PRESET, CE;
output reg Q= 1'b1;
always @(D or G or PRESET or CE ) begin
    if(PRESET)begin
        Q <= 1'b1;
    end
    else begin
        if (G && CE)begin
            Q <= D;
        end
    end
end
end
endmodule
```

5 Guideline on BSRAM Coding

Block Memory is a block static random access memory. According to the configuration mode, Block Memory includes single port mode (SP/SPX9), dual port mode (DP/DPX9), semi-dual mode (SDP/SDPX9), and read-only mode (ROM/ROMX9).

5.1 DPB/DPX9B

DPB/DPX9B works in dual port mode with its memory space of 16K bit/18K bit. Port A and port B support read/write operations respectively. DPB/DPX9B supports 2 read modes (bypass mode and pipeline mode) and 3 write modes (normal mode, write-through mode and read-before-write mode). This section will take the read address through or not through register and initial value read as examples to introduce the implementation.

5.1.1 Read Address Through Register

When read address through register, it is only supported no control signal. Take DPB in write-through, bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```
module normal(data_outa, data_ina, addra, clka, cea, wrea, data_outb,
data_inb, addrb, clkb, ceb, wreb);
    output [7:0] data_outa, data_outb;
    input [7:0] data_ina, data_inb;
    input [10:0] addra, addrb;
    input clka, wrea, cea;
    input clkb, wreb, ceb;
    reg [7:0] mem [2047:0];
    reg [10:0] addra_reg, addrb_reg;

    always@(posedge clka) begin
```

```

        addra_reg<=addra;
    end
    always @(posedge clka)begin
        if (cea & wrea) begin
            mem[addra] <= data_ina;
        end
    end
    assign data_outa = mem[addra_reg];

    always@(posedge clk)begin
        addrb_reg<=addrb;
    end

    always @(posedge clk)begin
        if (ceb & wreb) begin
            mem[addrb] <= data_inb;
        end
    end
    assign data_outb = mem[addrb_reg];
endmodule

```

5.1.2 Read Address Not Through Register

When the read address is not through register, the output must pass through register. It is bypass mode after passing through one-stage register, and it is pipeline mode after passing through two-stage register. Take DPB in normal, pipeline and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module normal(data_outa, data_ina, addra, clka, cea, ocea, wrea, rsta,
data_outb, data_inb, addrb, clk, ceb, oceb, wreb, rstb);
    output reg [15:0]data_outa,data_outb;
    input reg [15:0]data_ina,data_inb;
    input [9:0]addra,addrb;
    input clka,wrea,cea,ocea,rsta;
    input clk,wreb,ceb,oceb,rstb;
    reg [15:0] mem [1023:0];

```

```
reg [15:0] data_outa_reg=16'h0000;
reg [15:0] data_outb_reg=16'h0000;

always@(posedge clka)begin
    if(rsta)begin
        data_outa <= 0;
    end
    else begin
        if (oceas)begin
            data_outa <= data_outa_reg;
        end
    end
end
always@(posedge clka)begin
    if(rsta)begin
        data_outa_reg <= 0;
    end
    else begin
        if(cea & !wrea)begin
            data_outa_reg <= mem[addra];
        end
    end
end
always @(posedge clka)begin
    if (cea & wrea) begin
        mem[addra] <= data_ina;
    end
end

always@(posedge clk_b)begin
    if(rstb)begin
        data_outb <= 0;
    end
    else begin
```



```

        if (oceb)begin
            data_outb <= data_outb_reg;
        end
    end
end
always@(posedge clkb )begin
    if(rstb)begin
        data_outb_reg <= 0;
    end
    else begin
        if(ceb & !wreb)begin
            data_outb_reg <= mem[addrb];
        end
    end
end
end

always @(posedge clkb)begin
    if (ceb & wreb) begin
        mem[addrb] <= data_inb;
    end
end
endmodule

```

5.1.3 Assignment When Memory Defined

Only GowinSynthesis[®] supports the initial value assigned when memory defined, and Verilog language needs to select system Verilog. Take DPB in read-before-write, bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module normal(data_outa, data_ina, addra, clka, cea,
wrea, rsta, data_outb, data_inb, addrb, clkb, ceb, wreb, rstb);
    output [3:0]data_outa,data_outb;
    input [3:0]data_ina,data_inb;
    input [2:0]addra,addrb;
    input clka,wrea,cea,rsta;
    input clkb,wreb,ceb,rstb;
    reg [3:0] mem [7:0]={4'h1,4'h2,4'h3,4'h4,4'h5,4'h6,4'h7,4'h8};

```

```
reg [3:0] data_outa=4'h0;
reg [3:0] data_outb=4'h0;

always@(posedge clka)begin
    if(rsta)begin
        data_outa <= 0;
    end
    else begin
        if(cea)begin
            data_outa <= mem[addra];
        end
    end
end

always @(posedge clka)begin
    if (cea & wrea) begin
        mem[addra] <= data_ina;
    end
end

always@(posedge clkb)begin
    if(rstb)begin
        data_outb <= 0;
    end
    else begin
        if(ceb)begin
            data_outb <= mem[addrb];
        end
    end
end

always @(posedge clkb)begin
    if (ceb & wreb) begin
```

```

        mem[addrb] <= data_inb;
    end
end
endmodule

```

5.1.4 Assignment by readmemb/readmemh

When you use readmemb/readmemh for assignment, you need to pay attention to the path writing. Use '/' as the path delimiter. Take DPB in read-before-write, bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module normal(data_outa, data_ina, addra, clka, cea,
wrea, rsta, data_outb, data_inb, addrb, clkb, ceb, wreb, rstb);
    output [3:0] data_outa, data_outb;
    input [3:0] data_ina, data_inb;
    input [2:0] addra, addrb;
    input clka, wrea, cea, rsta;
    input clkb, wreb, ceb, rstb;
    reg [3:0] mem [7:0];
    reg [3:0] data_outa = 4'h0;
    reg [3:0] data_outb = 4'h0;

    initial begin
        $readmemb ("E:/dpb.mi", mem);
    end

    always@(posedge clka) begin
        if(rsta) begin
            data_outa <= 0;
        end
        else begin
            if(cea) begin
                data_outa <= mem[addra];
            end
        end
    end
end

```

```
always @(posedge clka)begin
    if (cea & wrea) begin
        mem[addra] <= data_ina;
    end
end

always@(posedge clk)begin
    if(rstb)begin
        data_outb <= 0;
    end
    else begin
        if(ceb)begin
            data_outb <= mem[addrb];
        end
    end
end

always @(posedge clk)begin
    if (ceb & wreb) begin
        mem[addrb] <= data_inb;
    end
end
endmodule
```

The writing form of dpb.mi is as follows:

```
0001
0010
0011
0100
0101
0110
0111
1000
```

Note !

If you use path delimiter '\' supported by the windows system, you need to add escape characters, such as, E:\\dpcb.mi.

5.2 SP/SPX9

SP/SPX9 works in single port mode with a memory space of 16K bit/18K bit. The read and write operations of the single port are controlled by a clock. SP/SPX9 supports two read modes (bypass mode and pipeline mode) and three write modes (normal mode, write-through mode and read-before-write mode). When SP/SPX9 is generated, memory needs to meet at least one of the following conditions:

1. Data bit width * address depth > 1024
2. Use syn_ramstyle = "block_ram".

This section will take read address through or not through register and initial value read as examples to introduce the implementation.

5.2.1 Read Address Through Register

When read address through register, it is only supported with no control signal, and it will generate SP in write-through mode. It will take the output not through register as an example to introduce the implementation. The coding is as follows:

```
module normal(data_out, data_in, addr, clk, ce, wre);
    output [9:0]data_out;
    input [9:0]data_in;
    input [9:0]addr;
    input clk,wre,ce;
    reg [9:0] mem [1023:0];
    reg [9:0]addr_reg=10'h000;

    always@(posedge clk)begin
        addr_reg<=addr;
    end
    always @(posedge clk)begin
        if (ce & wre) begin
            mem[addr] <= data_in;
        end
    end
    assign data_out = mem[addr_reg];
endmodule
```

5.2.2 Read Address Not Through Register

When the read address is not through register, the output must pass through register. It is bypass mode after passing through one-stage register, and it is pipeline mode after passing through two-stage register. Take SPX9 in write-through, bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module wt(data_out, data_in, addr, clk, ce, wre, rst);
output reg [17:0]data_out=18'h000000;
input [17:0]data_in;
input [9:0]addr;
input clk,wre,ce,rst;
reg [17:0] mem [1023:0];
always@(posedge clk )begin
    if(rst)begin
        data_out <= 0;
    end
    else begin
        if(ce & wre)begin
            data_out <= data_in;
        end
        else begin
            if (ce & !wre)begin
                data_out <= mem[addr];
            end
        end
    end
end
always @(posedge clk)begin
    if (ce & wre)begin
        mem[addr] <= data_in;
    end
end
endmodule

```

5.2.3 Assignment When Memory Defined

Verilog language needs to select system Verilog. Take SP in read-before-write, bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```
module rbw(data_out, data_in, addr, clk, ce, wre, rst) /*synthesis
syn_ramstyle="block_ram"*/;
    output [15:0]data_out;
    input [15:0]data_in;
    input [2:0]addr;
    input clk,wre,ce,rst;
    reg [15:0] mem [7:0]={16'h0123,16'h4567,16'h89ab,16'hcdef,
16'h0147,16'h0258,16'h789a,16'h5678};
    reg [15:0] data_out=16'h0000;
    always@(posedge clk )begin
        if(rst)begin
            data_out <= 0;
        end
        else begin
            if(ce)begin
                data_out <= mem[addr];
            end
        end
    end
    always @(posedge clk)begin
        if (ce & wre)begin
            mem[addr] <= data_in;
        end
    end
endmodule
```

5.2.4 Assignment by readmemb/readmemh

When you use readmemb/readmemh for assignment, you need to pay attention to the path writing. Use '/' as the path delimiter. Take SP in normal, pipeline and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```
module normal(data_out, data_in, addr, clk, ce, oce, wre,
```

```
rst)/*synthesis syn_ramstyle="block_ram"*/;
    output reg [7:0]data_out=8'h00;
    input [7:0]data_in;
    input [2:0]addr;
    input clk,wre,ce,oce,rst;
    reg [7:0] mem [7:0];
    reg [7:0] data_out_reg=8'h00;
    initial begin
        $readmemh ("E:/sp.mi", mem);
    end

    always@(posedge clk )begin
        if(rst)begin
            data_out <= 0;
        end
        else begin
            if(oce)begin
                data_out <= data_out_reg;
            end
        end
    end
end

always@(posedge clk)begin
    if(rst)begin
        data_out_reg <= 0;
    end
    else begin
        if(ce & !wre)begin
            data_out_reg <= mem[addr];
        end
    end
end

always @(posedge clk)begin
    if (ce & wre) begin
        mem[addr] <= data_in;
    end
end
```



```

        end
    end
endmodule

```

The writing form of sp.mi is as follow:

```

12
34
56
78
9a
bc
de
ff

```

5.2.5 Generated by Shift Register

It needs to meet one of the following conditions when SP/SPX9 generated by shift register:

1. memory depth ≥ 3 , memory depth * data bit width > 256 , and memory depth $= 2^n + 1$;
2. Add attribute constraints syn_srlstyle= "block_ram", and memory depth $= 2^n + 1$.

Take SPX9 in read-before-write, bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module  p_seqshift(clk, we, din, dout);
parameter width=18;
parameter depth=17;
input clk, we;
input [width-1:0] din;
output [width-1:0] dout;

reg [width-1:0] regBank[depth-1:0];

always @(posedge clk) begin
    if (we) begin
        regBank[depth-1:1] <= regBank[depth-2:0];
        regBank[0] <= din;
    end
end

```

```

        end
    end

    assign dout = regBank[depth-1];
endmodule

```

5.2.6 Generated by Decoder

Take SP in read-before-write, bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module top (data_out, data_in, addr, clk,wre,rst)/*synthesis
syn_ramstyle="block_ram"*/;

    parameter init0 = 16'h1234;
    parameter init1 = 16'h5678;
    parameter init2 = 16'h9abc;
    parameter init3 = 16'h0147;

    output reg[3:0]data_out;
    input [3:0]data_in;
    input [3:0]addr;
    input clk,wre,rst;

    reg [15:0] mem0=init0;
    reg [15:0] mem1=init1;
    reg [15:0] mem2=init2;
    reg [15:0] mem3=init3;

    always @(posedge clk)begin
        if (wre) begin
            mem0[addr] <= data_in[0];
            mem1[addr] <= data_in[1];
            mem2[addr] <= data_in[2];
            mem3[addr] <= data_in[3];
        end
    end
end

always @(posedge clk)begin

```

```

    if(rst)begin
        data_out<=16'h00;
    end
    else begin
        data_out[0] <= mem0[addr];
        data_out[1] <= mem1[addr];
        data_out[2] <= mem2[addr];
        data_out[3] <= mem3[addr];
    end
end
endmodule

```

5.3 SDPB/SDPX9B

SDPB/SDPX9B works in semi-dual port mode with its memory space of 16K bit/18K bit. SDPB/SDPX9B supports two read modes (bypass mode and pipeline mode) and one write mode (normal mode). When SDPB/SDPX9B is generated, memory needs to meet at least one of the following conditions:

1. Data bit width * address depth >1024
2. Use syn_ramstyle = "block_ram"

5.3.1 Memory Without Initial Value

Take SDPB in bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module normal(dout, din, ada, adb, clka, cea, clkb, ceb, resetb);
    output reg[15:0]dout=16'h0000;
    input [15:0]din;
    input [9:0]ada, adb;
    input clka, cea,clkb, ceb, resetb;
    reg [15:0] mem [1023:0];

    always @(posedge clka)begin
        if (cea )begin
            mem[ada] <= din;
        end
    end
end

```

```

always@(posedge clkb)begin
    if(resetb)begin
        dout <= 0;
    end
    else if(ceb)begin
        dout <= mem[adb];
    end
end

endmodule

```

5.3.2 Assignment When Memory Defined

Take SDPB in pipeline and asynchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module normal(data_out, data_in, addra, addrb, clka, cea, clkb,
ceb,oce, rstb)/*synthesis syn_ramstyle="block_ram"*/;
    output reg[15:0]data_out=16'h0000;
    input [15:0]data_in;
    input [2:0]addra, addrb;
    input clka, cea, clkb, ceb, rstb,oce;
    reg [15:0] mem [7:0]={16'h0123,16'h4567,16'h89ab,16'hcdef,
16'h0147, 16'h0258,16'h789a,16'h5678};
    reg [15:0] data_out_reg=16'h0000;

    always @(posedge clka)begin
        if (cea )begin
            mem[addra] <= data_in;
        end
    end

    always@(posedge clkb or posedge rstb)begin
        if(rstb)begin
            data_out_reg <= 0;
        end
    end

```

```

        else if(ceb)begin
            data_out_reg<= mem[addrb];
        end
    end
always@(posedge clkb or posedge rstb)begin
    if(rstb)begin
        data_out <= 0;
    end
    else if(oce)begin
        data_out<= data_out_reg;
    end
end
endmodule

```

5.3.3 Assignment by readmemb/readmemh

When you use readmemb/readmemh for assignment, you need to pay attention to the path writing. Use '/' as the path delimiter. Take SDPB in bypass and asynchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module normal(dout, din, ada, adb, clka, cea, clkb, ceb,
resetb)/*synthesis syn_ramstyle="block_ram"*/;
    output reg[7:0]dout=8'h00;
    input [7:0]din;
    input [2:0]ada, adb;
    input clka, cea,clkb, ceb, resetb;
    reg [7:0] mem [7:0];
    initial begin
        $readmemh ("E:/sdpb.mi", mem);
    end

    always @(posedge clka)begin
        if (cea )begin
            mem[ada] <= din;
        end
    end
end

```

```

always@(posedge clkb or posedge resetb)begin
    if(resetb)begin
        dout <= 0;
    end
    else if(ceb)begin
        dout <= mem[adb];
    end
end

endmodule

```

The writing form of sdpb.mi is as follows:

```

12
34
56
78
9a
bc
de
ff

```

5.3.4 Generated by Shift Register

It needs to meet one of the following conditions when generating B-SRAM based on shift register:

1. memory depth ≥ 3 , memory depth * data bit width > 256 , and memory depth $\neq 2^n + 1$;
2. Add attribute constraints syn_srlstyle= "block_ram", and memory depth $\neq 2^n + 1$.

Take SDPX9B in bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module p_seqshift(clk, we, din, dout);
    parameter width=18;
    parameter depth=16;
    input clk, we;
    input [width-1:0] din;

```

```

output [width-1:0] dout;

reg [width-1:0] regBank[depth-1:0];

always @(posedge clk) begin
    if (we) begin
        regBank[depth-1:1] <= regBank[depth-2:0];
        regBank[0] <= din;
    end
end

assign dout = regBank[depth-1];
endmodule

```

5.3.5 Generated by Different Widths

When SDPB is generated in different widths, you should add constraint `syn_ramstyle = "no_rw_check"`. Take SDPB in bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module normal(dout, din, ada, clka, cea, adb, clkb, ceb,
rstb,oce)/*synthesis syn_ramstyle = "no_rw_check"*/;
    parameter adawidth = 8;
    parameter diwidth = 6;
    parameter adbwidth = 7;
    parameter dowidth = 12;

    output [dowidth-1:0]dout;
    input [diwidth-1:0]din;
    input [adawidth-1:0]ada;
    input [adbwidth-1:0]adb;
    input clka,cea,clkb,ceb,rstb,oce;
    reg [diwidth-1:0]mem [2**adawidth-1:0];
    reg [dowidth-1:0]dout_reg;
    localparam b = 2**adawidth/2**adbwidth ;
    integer j ;

```

```

always @(posedge clka)begin
    if (cea)begin
        mem[ada] <= din;
    end
end

always@(posedge clkb )begin
    if(rstb)begin
        dout_reg <= 0;
    end
    else begin
        if(ceb)begin
            for(j = 0;j < b;j = j+1)
                dout_reg[((j+1)*diwidth-1)-: diwidth]<=
mem[adb*b+j];
        end
    end
end
assign dout = dout_reg;
endmodule

```

5.3.6 Generated by Decoder

Take SDPB in bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module top (data_out, data_in, wad, rad,rst, clk,wre)/*synthesis
syn_ramstyle="block_ram"*/;;
    parameter init0 = 16'h1234;
    parameter init1 = 16'h5678;
    parameter init2 = 16'h9abc;
    parameter init3 = 16'h0147;

    output reg[3:0] data_out;
    input [3:0]data_in;
    input [3:0]wad,rad;

```



```

input clk,wre,rst;

reg [15:0] mem0=init0;
reg [15:0] mem1=init1;
reg [15:0] mem2=init2;
reg [15:0] mem3=init3;

always @(posedge clk)begin
    if (wre) begin
        mem0[wad] <= data_in[0];
        mem1[wad] <= data_in[1];
        mem2[wad] <= data_in[2];
        mem3[wad] <= data_in[3];
    end
end
always @(posedge clk)begin
    if(rst)begin
        data_out<=16'h00;
    end
    else begin
        data_out[0] <= mem0[rad];
        data_out[1] <= mem1[rad];
        data_out[2] <= mem2[rad];
        data_out[3] <= mem3[rad];
    end
end
endmodule

```

5.4 pROM/pROMX9

ROM/ROMX9 works in read only mode with the memory space of 16K bit/18K bit and supports two read modes (bypass mode and pipeline mode). PROM/pROMX9 can be assigned by case statement, readmemb/readmemh, memory definition, etc. When pROM is generated, memory needs to meet at least one of the following conditions:

1. Data bit width * address depth >1024
2. Use syn_ramstyle = "block_ram"

5.4.1 Assignment by Cases Statement

Take pROM in bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module normal (clk,rst,ce,addr,dout)/*synthesis
syn_romstyle="block_rom"*/ ;
    input clk;
    input rst,ce;
    input [4:0] addr;
    output reg [31:0] dout=32'h00000000;

    always @(posedge clk )begin
        if (rst) begin
            dout <= 0;
        end
        else begin
            if(ce)begin
                case(addr)
                    5'h00: dout <= 32'h52853fd5;
                    5'h01: dout <= 32'h38581bd2;
                    5'h02: dout <= 32'h040d53e4;
                    5'h03: dout <= 32'h22ce7d00;
                    5'h04: dout <= 32'h73d90e02;
                    5'h05: dout <= 32'hc0b4bf1c;
                    5'h06: dout <= 32'hec45e626;
                    5'h07: dout <= 32'hd9d000d9;
                    5'h08: dout <= 32'haacf8574;
                    5'h09: dout <= 32'hb655bf16;
                    5'h0a: dout <= 32'h8c565693;
                    5'h0b: dout <= 32'hb19808d0;
                    5'h0c: dout <= 32'he073036e;
                    5'h0d: dout <= 32'h41b923f6;
                    5'h0e: dout <= 32'hdce89022;
                    5'h0f: dout <= 32'hba17fce1;
                    5'h10: dout <= 32'hd4dec5de;

```

```

5'h11: dout <= 32'ha18ad699;
5'h12: dout <= 32'h4a734008;
5'h13: dout <= 32'h5c32ac0e;
5'h14: dout <= 32'h8f26bdd4;
5'h15: dout <= 32'hb8d4aab6;
5'h16: dout <= 32'hf55e3c77;
5'h17: dout <= 32'h41a5d418;
5'h18: dout <= 32'hba172648;
5'h19: dout <= 32'h5c651d69;
5'h1a: dout <= 32'h445469c3;
5'h1b: dout <= 32'h2e49668b;
5'h1c: dout <= 32'hdc1aa05b;
5'h1d: dout <= 32'hcebfe4cd;
5'h1e: dout <= 32'h1e1f0f1e;
5'h1f: dout <= 32'h86fd31ef;
default: dout <= 32'h8e9008a6;
endcase
end
end
end
endmodule

```

5.4.2 Assignment When Memory Defined

Verilog language needs to select system Verilog. Take pROM in bypass and synchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module prom_inference ( clk, addr,rst, data_out)/* synthesis
syn_romstyle = "block_rom" */;
input clk;
input rst;
input [3:0] addr;
output reg[3:0] data_out;
reg [3:0] mem [15:0]={4'h1,4'h2,4'h3,4'h4,4'h5,4'h6,4'h7,4'h8,4'h9,4'ha,
4'hb, 4'hc,4'hd,4'he,4'hf,4'hd};

always @(posedge clk)begin

```

```

        if (rst) begin
            data_out <= 0;
        end
        else begin
            data_out <= mem[addr];;
        end
    end
end
endmodule

```

5.4.3 Assignment by readmemb/readmemh

When you use readmemb/readmemh for assignment, you need to pay attention to the path writing. Use '/' as the path delimiter. Take pROM in pipeline and asynchronous reset mode as an example to introduce the implementation. The coding is as follows:

```

module rom_inference ( clk, addr, rst, oce, data_out);
input clk;
input rst, oce;
input [4:0] addr;
output reg [31:0] data_out;
reg [31:0] mem [31:0] /* synthesis syn_romstyle = "block_rom" */;
reg [31:0] data_out_reg;
initial begin
    $readmemh ("E:/prom.ini", mem);
end
always @(posedge clk or posedge rst)begin
    if(rst)begin
        data_out_reg <=0;
    end
    else begin
        data_out_reg <= mem[addr];
    end
end

always @(posedge clk or posedge rst)begin
    if(rst)begin

```

```
        data_out <= 0;
    end
    else begin
        data_out <= data_out_reg;
    end
end
endmodule
```

The writing form of prom.ini is as follows:

```
11001100
11001100
11001100
11001100
17001100
11001100
16001100
1f001100
11111100
11111100
11001110
11000111
11000111
11001110
11011100
11001110
```

6 Guideline on SSRAM Coding

The Shadow Memory can be configured as single port mode, semi-dual port mode and read-only mode, as shown in Table 6-1.

Table 6-1 Primitive Related with SSRAM

Primitive	Description
RAM16S1	Single port SSRAM with address depth 16 and data width 1
RAM16S2	Single port SSRAM with address depth 16 and data width 2
RAM16S4	Single port SSRAM with address depth 16 and data width 4
RAM16SDP1	Semi-dual port SSRAM with address depth 16 and data width 1
RAM16SDP2	Semi-dual port SSRAM with address depth 16 and data width 2
RAM16SDP4	Semi-dual port SSRAM with address depth 16 and data width 4
ROM16	ROM with address depth 16 and data width 1

6.1 RAM16S

RAM16S includes RAM16S1, RAM16S2, and RAM16S4. The difference is the width of the output bit. The RAM16S can be written in Decoder, Memory, shift register, etc. When RAM16S is generated, memory needs to meet at least one of the following conditions:

1. Read address and output not through register, and address depth * data bit width ≥ 8 ;
2. Output through register, and $8 \leq \text{address depth} * \text{data bit width} \leq 1024$;
3. Use attribute constraint `syn_ramstyle= "distributed_ram"`.

6.1.1 Generated by Decoder

Take RAM16S4 as an example to introduce the implementation. The coding is as follows:

```
module top (data_out, data_in, addr, clk, wre);
    parameter init0 = 16'h1234;
```

```

parameter init1 = 16'h5678;
parameter init2 = 16'h9abc;
parameter init3 = 16'h0147;

output [3:0]data_out;
input [3:0]data_in;
input [3:0]addr;
input clk,wre;
reg [15:0] mem0=init0;
reg [15:0] mem1=init1;
reg [15:0] mem2=init2;
reg [15:0] mem3=init3;

always @(posedge clk)begin
    if (wre) begin
        mem0[addr] <= data_in[0];
        mem1[addr] <= data_in[1];
        mem2[addr] <= data_in[2];
        mem3[addr] <= data_in[3];
    end
end

assign data_out[0] = mem0[addr];
assign data_out[1] = mem1[addr];
assign data_out[2] = mem2[addr];
assign data_out[3] = mem3[addr];
endmodule

```

6.1.2 Generated by Memory

The memory can be divided into memory without initial value, memory defined with initial value and readmemh/readmemb. For the last two kinds of memory, see 5.1.4 for details.

Take RAM16S4 as an example to introduce the implementation of memory without initial value. The coding is as follows:

```

module normal(data_out, data_in, addr, clk, wre);

```

```

output [3:0]data_out;
input [3:0]data_in;
input [3:0]addr;
input clk,wre;
reg [3:0] mem [15:0];
always @(posedge clk)begin
    if ( wre) begin
        mem[addr] <= data_in;
    end
end
assign data_out = mem[addr];
endmodule

```

6.1.3 Generated by Shift Register

It needs to meet at least one of the following conditions.

1. memory depth ≥ 4 , memory depth * data bit width ≤ 256 , and memory depth = 2^n ;
2. Add attribute constraints syn_srlstyle= " distributed_ram", and memory depth = 2^n .

Take RAM16S4 generated by GowinSynthesis as an example to introduce the implementation. The coding is as follows:

```

module  p_seqshift(clk, we, din, dout);
parameter width=18;
parameter depth=4;
input clk, we;
input [width-1:0] din;
output [width-1:0] dout;

reg [width-1:0] regBank[depth-1:0];

always @(posedge clk) begin
    if (we) begin
        regBank[depth-1:1] <= regBank[depth-2:0];
        regBank[0] <= din;
    end
end

```



```

end

assign dout = regBank[depth-1];
endmodule

```

6.2 RAM16SDP

RAM16SDP includes RAM16SDP1, RAM16SDP2 and RAM16SDP4. The difference is the width of the output bit. The RAM16SDP can be written in Decoder, Memory, shift register, etc. When RAM16SDP is generated, memory needs to meet at least one of the following conditions:

1. Read address and output not through register, and address depth * data bit width ≥ 8 ;
2. Read address or output through register, and $8 \leq \text{address depth} * \text{data bit width} \leq 1024$;
3. Use attribute constraint `syn_ramstyle= "distributed_ram"`.

6.2.1 Generated by Decoder

Take RAM16SDP4 as an example to introduce the implementation. The coding is as follows:

```

module top (data_out, data_in, wad, rad, clk, wre);
    parameter init0 = 16'h1234;
    parameter init1 = 16'h5678;
    parameter init2 = 16'h9abc;
    parameter init3 = 16'h0147;

    output [3:0] data_out;
    input [3:0] data_in;
    input [3:0] wad, rad;
    input clk, wre;
    reg [15:0] mem0=init0;
    reg [15:0] mem1=init1;
    reg [15:0] mem2=init2;
    reg [15:0] mem3=init3;

    always @(posedge clk) begin
        if (wre) begin
            mem0[wad] <= data_in[0];

```

```

        mem1[wad] <= data_in[1];
        mem2[wad] <= data_in[2];
        mem3[wad] <= data_in[3];
    end
end

    assign data_out[0] = mem0[rad];
    assign data_out[1] = mem1[rad];
    assign data_out[2] = mem2[rad];
    assign data_out[3] = mem3[rad];
endmodule

```

6.2.2 Generated by Memory

The memory can be divided into memory without initial value, memory defined with initial value and readmemh/readmemb. For the last two kinds of memory, see 5.1.4 for details.

Take RAM16SDP4 as an example to introduce the implementation of memory. The coding is as follows:

```

module normal(data_out, data_in, addra, clk, wre, addrb);
    output [3:0] data_out;
    input [3:0] data_in;
    input [3:0] addra ,addrb;
    input clk,wre;

    reg [3:0] mem [15:0];

    always @(posedge clk)begin
        if (wre)begin
            mem[addra] <= data_in;
        end
    end

    assign data_out = mem[addrb];

endmodule

```

6.2.3 Generated by Shift Register

It needs to meet at least one of the following conditions if the shift register is synthesized as RAM16SDP.

1. memory depth ≥ 4 , memory depth * data bit width ≤ 256 , and memory depth $\neq 2^n$;
2. Add attribute constraints `syn_srstyle= "distributed_ram"`, and memory depth $= 2^n$.

Take RAM16SDP4 generated by GowinSynthesis as an example to introduce the implementation. The coding is as follows:

```
module p_seqshift(clk, we, din, dout);
    parameter width=18;
    parameter depth=7;
    input clk, we;
    input [width-1:0] din;
    output [width-1:0] dout;

    reg [width-1:0] regBank[depth-1:0];

    always @(posedge clk) begin
        if (we) begin
            regBank[depth-1:1] <= regBank[depth-2:0];
            regBank[0] <= din;
        end
    end

    assign dout = regBank[depth-1];
endmodule
```

6.3 ROM16

ROM16 is a read-only memory with address depth 16 and data width 1. The memory is initialized using INIT. ROM16 can be written in the form of Decoder, Memory, etc. When ROM16 is synthesized, it needs to add attribute constraint `syn_romstyle = "distributed_rom"`.

6.3.1 Generated by Decoder

```
module test (addr,dataout)/*synthesis
syn_romstyle="distributed_rom"*/ ;
```

```

input [3:0] addr;
output reg dataout=1'h0;
always @(*)begin
    case(addr)
        4'h0: dataout <= 1'h0;
        4'h1: dataout <= 1'h0;
        4'h2: dataout <= 1'h1;
        4'h3: dataout <= 1'h0;
        4'h4: dataout <= 1'h1;
        4'h5: dataout <= 1'h1;
        4'h6: dataout <= 1'h0;
        4'h7: dataout <= 1'h0;
        4'h8: dataout <= 1'h0;
        4'h9: dataout <= 1'h1;
        4'ha: dataout <= 1'h0;
        4'hb: dataout <= 1'h0;
        4'hc: dataout <= 1'h1;
        4'hd: dataout <= 1'h0;
        4'he: dataout <= 1'h0;
        4'hf: dataout <= 1'h0;
        default: dataout <= 1'h0;
    endcase
end
endmodule

```

6.3.2 Generated by Memory

The memory can be divided into memory without initial value, memory defined with initial value and readmemh/readmemb. For the last two kinds of memory, see 5.1.4 for details. The coding is as follows:

```

module top (addr,dataout)/*synthesis
syn_romstyle="distributed_rom"*/;
    input [3:0] addr;
    output reg dataout=1'b0;

    parameter init0 = 16'h117a;

```

```
reg [15:0] mem0=init0;  
always @(*)begin  
    dataout <= mem0[addr];  
end  
endmodule
```

7 Guideline on DSP Coding

The digital signal processing (DSP) includes Pre-Adder, MULT, and ALU54D.

7.1 Pre-adder

The pre-adder performs the functions of pre-adding, pre-subtracting, and shifting. According to the bit width, a pre-adder includes 9-bit width PADD9 and 18-bit width PADD18. Pre-adder needs to be coupled with Multiplier in order to be inferred.

7.1.1 Pre-adding

Take AREG, BREG and PADD9-MULT9X9 in synchronous reset mode as an example to introduce the implementation of pre-adding function. The coding is as follows:

```
module top(a0, b0, b1, dout, rst, clk, ce);
  input [7:0] a0;
  input [7:0] b0;
  input [7:0] b1;
  input rst, clk, ce;
  output [17:0] dout;
  reg [8:0] p_add_reg=9'h000;
  reg [7:0] a0_reg=8'h00;
  reg [7:0] b0_reg=8'h00;
  reg [7:0] b1_reg=8'h00;
  reg [17:0] pipe_reg=18'h00000;
  reg [17:0] s_reg=18'h00000;

  always @(posedge clk)begin
```

```
    if(rst)begin
        a0_reg <= 0;
        b0_reg <= 0;
        b1_reg <= 0;
    end else begin
        if(ce)begin
            a0_reg <= a0;
            b0_reg <= b0;
            b1_reg <= b1;
        end
    end
end
always @(posedge clk)begin
    if(rst)begin
        p_add_reg <= 0;
    end else begin
        if(ce)begin
            p_add_reg <= b0_reg+b1_reg;
        end
    end
end
end
```

```
always @(posedge clk)
begin
    if(rst)begin
        pipe_reg <= 0;
    end else begin
        if(ce) begin
            pipe_reg <= a0_reg*p_add_reg;
        end
    end
end
end
always @(posedge clk)
```

```

begin
    if(rst)begin
        s_reg <= 0;
    end else begin
        if(ce) begin
            s_reg <= pipe_reg;
        end
    end
end

assign dout = s_reg;

endmodule

```

7.1.2 Pre-subtracting

Take AREG, BREG and PADD9-MULT9X9 in synchronous reset mode as an example to introduce the implementation of pre-subtracting function. The coding is as follows:

```

module top(a0, b0, b1, dout, rst, clk, ce);
    input [7:0] a0;
    input [7:0] b0;
    input [7:0] b1;
    input rst, clk, ce;
    output [17:0] dout;
    reg [8:0] p_add_reg=9'h000;
    reg [7:0] a0_reg=8'h00;
    reg [7:0] b0_reg=8'h00;
    reg [7:0] b1_reg=8'h00;
    reg [17:0] pipe_reg=18'h00000;
    reg [17:0] s_reg=18'h00000;

    always @(posedge clk)begin
        if(rst)begin
            a0_reg <= 0;
            b0_reg <= 0;

```



```
        b1_reg <= 0;
    end else begin
        if(ce)begin
            a0_reg <= a0;
            b0_reg <= b0;
            b1_reg <= b1;
        end
    end
end
always @(posedge clk)begin
    if(rst)begin
        p_add_reg <= 0;
    end else begin
        if(ce)begin
            p_add_reg <= b0_reg-b1_reg;
        end
    end
end
end
```

```
always @(posedge clk)
begin
    if(rst)begin
        pipe_reg <= 0;
    end else begin
        if(ce) begin
            pipe_reg <= a0_reg*p_add_reg;
        end
    end
end
always @(posedge clk)
begin
    if(rst)begin
        s_reg <= 0;
```

```

        end else begin
            if(ce) begin
                s_reg <= pipe_reg;
            end
        end
    end
end

assign dout = s_reg;

endmodule

```

7.1.3 Shifting

Take AREG, BREG and PADD18-MULT18X18 in asynchronous reset mode as an example to introduce the implementation of shifting function. The coding is as follows:

```

module top(a0, a1, b0, b1, p0, p1, clk, ce, reset);
    parameter a_width=18;
    parameter b_width=18;
    parameter p_width=36;
    input [a_width-1:0] a0, a1;
    input [b_width-1:0] b0, b1;
    input clk, ce, reset;
    output [p_width-1:0] p0, p1;
    wire [b_width-1:0] b0_padd, b1_padd;
    reg [b_width-1:0] b0_reg=18'h00000;
    reg [b_width-1:0] b1_reg=18'h00000;
    reg [b_width-1:0] bX1=18'h00000;
    reg [b_width-1:0] bY1=18'h00000;

    always @(posedge clk or posedge reset)
    begin
        if(reset)begin
            b0_reg <= 0;
            b1_reg <= 0;
            bX1 <= 0;

```

```

        bY1 <= 0;

    end else begin
        if(ce)begin
            b0_reg <= b0;
            b1_reg <= b1;
            bX1 <= b0_reg;
            bY1 <= b1_reg;
        end
    end

    assign b0_padd = bX1 + b1_reg;
    assign b1_padd= b0_reg + bY1;

    assign p0 = a0 * b0_padd;
    assign p1 = a1 * b1_padd;

endmodule

```

7.2 Multiplier

Multiplier is a DSP multiplier unit. Its input signals are MDIA and MDIB, and output signal is MOUT, and it can realize multiplication: $DOUT = A * B$.

Based on bit width, the multiplier can be configured as 9x9, 18x18 and 36x36, which corresponds to MULT9X9, MULT18X18, and MULT36X36 primitives. Take AREG, BREG OUT_REG, PIPE_REG and MULT18X18 in asynchronous reset mode as examples to introduce the implementation. The coding is as follows:

```

module top(a,b,c,clock,reset,ce);
    input signed [17:0] a;
    input signed [17:0] b;
    input clock;
    input reset;
    input ce;
    output signed [35:0] c;

    reg signed [17:0] ina=18'h00000;

```

```
reg signed [17:0] inb=18'h000000;
reg signed [35:0] pp_reg=36'h0000000000;
reg signed [35:0] out_reg=36'h0000000000;
wire signed [35:0] mult_out;

always @(posedge clock or posedge reset)begin
    if(reset)begin
        ina<=0;
        inb<=0;
    end else begin
        if(ce)begin
            ina<=a;
            inb<=b;
        end
    end
end
assign mult_out=ina*inb;
always @(posedge clock or posedge reset)begin
    if(reset)begin
        pp_reg<=0;
    end else begin
        if(ce)begin
            pp_reg<=mult_out;
        end
    end
end

always @(posedge clock or posedge reset)begin
    if(reset)begin
        out_reg<=0;
    end else begin
        if(ce)begin
            out_reg<=pp_reg;
        end
    end
end
```

```

        end
    end
    assign c=out_reg;
endmodule

```

7.3 ALU54D

ALU54D is 54-bit arithmetic logic unit. If ALU5D is synthesized, the data bit width should be in [48, 54]. Otherwise, the attribute constraint `syn_dspstyle= "dsp"` should be added. Take AREG, BREG, OUT_REG and ALU54D in asynchronous reset mode as examples to introduce the implementation. The coding is as follows:

```

module top(a, b, s, accload, clk, ce, reset);
    parameter width=54;
    input signed [width-1:0] a, b;
    input accload, clk, ce, reset;
    output signed [width-1:0] s;
    wire signed [width-1:0] s_sel;
    reg [width-1:0] a_reg=54'h0000000000000000;
    reg [width-1:0] b_reg=54'h0000000000000000;
    reg [width-1:0] s_reg=54'h0000000000000000;
    reg acc_reg=1'b0;

    always @(posedge clk or posedge reset)
    begin
        if(reset)begin
            a_reg <= 0;
            b_reg <= 0;
        end else begin
            if(ce)begin
                a_reg <= a;
                b_reg <= b;
            end
        end
    end
end
always @(posedge clk)
begin

```

```

        if(ce)begin
            acc_reg <= accload;
        end
    end

    assign s_sel = (acc_reg == 1) ? s : 0;
    always @(posedge clk or posedge reset)
    begin
        if(reset)begin
            s_reg <= 0;
        end else begin
            if(ce)begin
                s_reg <= s_sel + a_reg + b_reg;
            end
        end
    end
    assign s = s_reg;
endmodule

```

7.4 MULTALU

Multiplier with ALU (MULTALU) is a multiplier with ALU function, including 36X18 bits and 18X18 bits, and MULTALU18X18 only supports instantiation. MULTALU36X18 provides three operations: $DOUT=A*B\pm C$, $DOUT=\sum(A*B)$, $DOUT=A*B+CASI$.

7.4.1 $A*B\pm C$

Take AREG, BREG, CREG, PIPE_REG, OUT_REG and MULTALU36X18 in asynchronous reset mode as examples to introduce the implementation of $DOUT=A*B+C$. The coding is as follows:

```

module top(a0, b0, c,s, reset, ce, clock);
    parameter a_width=36;
    parameter b_width=18;
    parameter s_width=54;
    input signed [a_width-1:0] a0;
    input signed [b_width-1:0] b0;
    input signed [s_width-2:0] c;

```

```
input reset, ce, clock;
output signed [s_width-1:0] s;
wire signed [s_width-1:0] p0;
reg signed [a_width-1:0] a0_reg=36'h000000000;
reg signed [b_width-1:0] b0_reg=18'h00000;
reg signed [s_width-1:0] p0_reg=54'h000000000000000;
reg signed [s_width-1:0] o0_reg=54'h000000000000000;
reg signed [s_width-2:0] c_reg=54'h000000000000000;
always @(posedge clock or posedge reset)begin
    if(reset)begin
        a0_reg <= 0;
        b0_reg <= 0;
        c_reg <= 0;
    end else begin
        if(ce)begin
            a0_reg <= a0;
            b0_reg <= b0;
            c_reg <= c;
        end
    end
end

assign p0 = a0_reg * b0_reg;
always @(posedge clock or posedge reset)begin
    if(reset)begin
        p0_reg <= 0;
        o0_reg <= 0;
    end else begin
        if(ce)begin
            p0_reg <= p0;
            o0_reg <= p0_reg+c_reg;
        end
    end
end
```

```

assign s = o0_reg;
endmodule

```

7.4.2 $\sum(A*B)$

Take PIPE_REG, OUT_REG and MULTALU36X18 in asynchronous reset mode as examples to introduce the implementation of $DOUT = \sum(A*B)$. The coding is as follows:

```

module top(a,b,c,clock,reset,ce);
parameter a_width = 36;
parameter b_width = 18;
parameter c_width = 54;
input signed [a_width-1:0] a;
input signed [b_width-1:0] b;
input clock;
input reset,ce;
output signed [c_width-1:0] c;

reg signed [c_width-1:0] pp_reg=54'h0000000000000000;
reg signed [c_width-1:0] out_reg=54'h0000000000000000;
wire signed [c_width-1:0] mult_out,c_sel;
reg acc_reg0=1'b0;
reg acc_reg1=1'b0;

assign mult_out=a*b;
always @(posedge clock or posedge reset) begin
    if(reset)begin
        pp_reg<=0;
    end else begin
        if(ce)begin
            pp_reg<=mult_out;
        end
    end
end
end

```



```

always @(posedge clock or posedge reset)
begin
    if(reset) begin
        out_reg <= 0;
    end else if(ce) begin
        out_reg <= c + pp_reg;
    end
end

assign c=out_reg;
endmodule

```

7.4.3 A*B+CASI

Take PIPE_REG, OUT_REG and MULTALU36X18 in asynchronous reset mode as examples to introduce the implementation of DOUT=A*B+CASI. The coding is as follows:

```

module top(a0, a1, a2, b0, b1, b2, s, reset, ce, clock);
parameter a_width=36;
parameter b_width=18;
parameter s_width=54;
input signed [a_width-1:0] a0, a1, a2;
input signed [b_width-1:0] b0, b1, b2;
input reset, ce, clock;
output signed [s_width-1:0] s;
wire signed [s_width-1:0] p0, p1, p2, s0, s1;
reg signed [s_width-1:0] p0_reg=54'h0000000000000000;
reg signed [s_width-1:0] p1_reg=54'h0000000000000000;
reg signed [s_width-1:0] p2_reg=54'h0000000000000000;
reg signed [s_width-1:0] s0_reg=54'h0000000000000000;
reg signed [s_width-1:0] s1_reg=54'h0000000000000000;
reg signed [s_width-1:0] o0_reg=54'h0000000000000000;

assign p0 = a0 * b0;
assign p1 = a1 * b1;
assign p2 = a2 * b2;

```

```
always @(posedge clock or posedge reset)
begin
    if(reset)begin
        p0_reg <= 0;
        p1_reg <= 0;
        p2_reg <= 0;
        o0_reg <= 0;
    end else begin
        if(ce)begin
            p0_reg <= p0;
            p1_reg <= p1;
            p2_reg <= p2;
            o0_reg <= p0_reg;
        end
    end
end
```

```
assign s0 = o0_reg + p1_reg;
always @(posedge clock or posedge reset)
begin
    if(reset)begin
        s0_reg <= 0;
    end else begin
        if(ce)begin
            s0_reg <= s0;
        end
    end
end
```

```
assign s1 = s0_reg + p2_reg;
always @(posedge clock or posedge reset)
begin
    if(reset)begin
        s1_reg <= 0;
    end
end
```

```

        end else begin
            if(ce)begin
                s1_reg <= s1;
            end
        end
    end
    assign s=s1_reg;
endmodule

```

7.5 MULTADDALU

MULTADDALU is the sum of two multipliers with ALU function to realize accumulating or reload operations after sum of multiplication. The corresponding primitive is MULTADDALU18X18. MULTADDALU provides three operations: $DOUT=A0*B0 \pm A1*B1 \pm C$, $DOUT=\sum(A0*B0 \pm A1*B1)$, $DOUT=A0*B0 \pm A1*B1 + CASI$.

7.5.1 $A0*B0 \pm A1*B1 \pm C$

Take A0REG, A1REG, B0REG, B1REG, PIPE0_REG, PIPE1_REG, OUT_REG and MULTADDALU18X18 in asynchronous reset mode as examples to introduce the implementation of $DOUT=A0*B0 \pm A1*B1 \pm C$. The coding is as follows:

```

module top(a0, a1, b0, b1, c,s, reset, clock, ce);
    parameter a0_width=18;
    parameter a1_width=18;
    parameter b0_width=18;
    parameter b1_width=18;
    parameter s_width=54;
    input signed [a0_width-1:0] a0;
    input signed [a1_width-1:0] a1;
    input signed [b0_width-1:0] b0;
    input signed [b1_width-1:0] b1;
    input [53:0] c;
    input reset, clock, ce;
    output signed [s_width-1:0] s;
    wire signed [s_width-1:0] p0, p1, p;
    reg signed [a0_width-1:0] a0_reg=18'h00000;
    reg signed [a1_width-1:0] a1_reg=18'h00000;

```

```
reg signed [b0_width-1:0] b0_reg=18'h00000;  
reg signed [b1_width-1:0] b1_reg=18'h00000;  
reg signed [s_width-1:0] p0_reg=54'h000000000000000;  
reg signed [s_width-1:0] p1_reg=54'h000000000000000;  
reg signed [s_width-1:0] s_reg=54'h000000000000000;
```

```
always @(posedge clock)begin
```

```
    if(reset)begin
```

```
        a0_reg <= 0;
```

```
        a1_reg <= 0;
```

```
        b0_reg <= 0;
```

```
        b1_reg <= 0;
```

```
    end
```

```
    else begin
```

```
        if(ce)begin
```

```
            a0_reg <= a0;
```

```
            a1_reg <= a1;
```

```
            b0_reg <= b0;
```

```
            b1_reg <= b1;
```

```
        end
```

```
    end
```

```
end
```

```
assign p0 = a0_reg*b0_reg;
```

```
assign p1 = a1_reg*b1_reg;
```

```
always @(posedge clock)begin
```

```
    if(reset)begin
```

```
        p0_reg <= 0;
```

```
        p1_reg <= 0;
```

```
    end
```

```
    else begin
```

```
        if(ce)begin
```

```
            p0_reg <= p0;
```

```

        p1_reg <= p1;
    end
end
end

assign p = p0_reg + p1_reg+c;

always @(posedge clock)begin
    if(reset)begin
        s_reg <= 0;
    end
    else begin
        if(ce) begin
            s_reg <= p;
        end
    end
end

assign s = s_reg;

endmodule

```

7.5.2 $\sum(A0*B0 \pm A1*B1)$

Take PIPE0_REG, PIPE1_REG, OUT_REG and MULTADDALU18X18 in asynchronous reset mode as examples to introduce the implementation of $DOUT = \sum(A0*B0 \pm A1*B1)$. The coding is as follows:

```

module acc(a0, a1, b0, b1, s, accload, ce, reset, clk);
    parameter a_width=18;
    parameter b_width=18;
    parameter s_width=54;
    input unsigned [a_width-1:0] a0, a1;
    input unsigned [b_width-1:0] b0, b1;
    input accload, ce, reset, clk;
    output unsigned [s_width-1:0] s;
    wire unsigned [s_width-1:0] s_sel;

```

```
wire unsigned [s_width-1:0] p0, p1;
reg unsigned [s_width-1:0] p0_reg=54'h0000000000000000;
reg unsigned [s_width-1:0] p1_reg=54'h0000000000000000;
reg unsigned [s_width-1:0] s=54'h0000000000000000;
reg acc_reg0=1'b0;
reg acc_reg1=1'b0;

assign p0 = a0*b0;
assign p1 = a1*b1;
always @(posedge clk or posedge reset)begin
    if(reset) begin
        p0_reg <= 0;
        p1_reg <= 0;
    end else if(ce) begin
        p0_reg <= p0;
        p1_reg <= p1;
    end
end
always @(posedge clk)begin
    if(ce) begin
        acc_reg0 <= accload;
        acc_reg1 <= acc_reg0;
    end
end
assign s_sel = (acc_reg1 == 1) ? s : 0;
always @(posedge clk or posedge reset)begin
    if(reset) begin
        s <= 0;
    end else if(ce) begin
        s <= s_sel + p0_reg - p1_reg;
    end
end
endmodule
```

7.5.3 $A0*B0 \pm A1*B1 + CASI$

Take PIPE0_REG, PIPE1_REG, OUT_REG and MULTADDALU18X18 in asynchronous reset mode as examples to introduce the implementation of $DOUT = A0*B0 \pm A1*B1 + CASI$. The coding is as follows:

```

module top(a0, a1, a2, b0, b1, b2, a3, b3, s, clock, ce, reset);
  parameter a_width=18;
  parameter b_width=18;
  parameter s_width=54;
  input signed [a_width-1:0] a0, a1, a2, b0, b1, b2, a3, b3;
  input clock, ce, reset;
  output signed [s_width-1:0] s;
  wire signed [s_width-1:0] p0, p1, p2, p3, s0, s1;
  reg signed [s_width-1:0] p0_reg=54'h0000000000000000;
  reg signed [s_width-1:0] p1_reg=54'h0000000000000000;
  reg signed [s_width-1:0] p2_reg=54'h0000000000000000;
  reg signed [s_width-1:0] p3_reg=54'h0000000000000000;
  reg signed [s_width-1:0] s0_reg=54'h0000000000000000;
  reg signed [s_width-1:0] s1_reg=54'h0000000000000000;
  reg signed [a_width-1:0] a0_reg=18'h000000;
  reg signed [a_width-1:0] a1_reg=18'h000000;
  reg signed [a_width-1:0] a2_reg=18'h000000;
  reg signed [a_width-1:0] a3_reg=18'h000000;
  reg signed [a_width-1:0] b0_reg=18'h000000;
  reg signed [a_width-1:0] b1_reg=18'h000000;
  reg signed [a_width-1:0] b2_reg=18'h000000;
  reg signed [a_width-1:0] b3_reg=18'h000000;
  always @(posedge clock or posedge reset)begin
    if(reset)begin
      a0_reg <= 0;
      a1_reg <= 0;
      a2_reg <= 0;
      a3_reg <= 0;
      b0_reg <= 0;
      b1_reg <= 0;

```

```
        b2_reg <= 0;
        b3_reg <= 0;
    end else begin
        if(ce)begin
            a0_reg <= a0;
            a1_reg <= a1;
            a2_reg <= a2;
            a3_reg <= a3;
            b0_reg <= b0;
            b1_reg <= b1;
            b2_reg <= b2;
            b3_reg <= b3;
        end
    end
end

assign p0 = a0_reg*b0_reg;
assign p1 = a1_reg*b1_reg;
assign p2 = a2_reg*b2_reg;
assign p3 = a3_reg*b3_reg;

always @(posedge clock or posedge reset)begin
    if(reset)begin
        p0_reg <= 0;
        p1_reg <= 0;
        p2_reg <= 0;
        p3_reg <= 0;
    end else begin
        if(ce)begin
            p0_reg <= p0;
            p1_reg <= p1;
            p2_reg <= p2;
            p3_reg <= p3;
        end
    end
end
```



```
        end
    end

    assign s0 = p0_reg + p1_reg;
    always @(posedge clock or posedge reset)begin
        if(reset)begin
            s0_reg <= 0;
        end else begin
            if(ce)begin
                s0_reg <= s0;
            end
        end
    end
end

assign s1 = s0_reg + p2_reg - p3_reg;
always @(posedge clock or posedge reset)begin
    if(reset)begin
        s1_reg <= 0;
    end else begin
        if(ce)begin
            s1_reg <= s1;
        end
    end
end

assign s=s1_reg;
endmodule
```

