

ASIC 设计实验报告



姓名： 黎加骏

学号： 2020210905

班级： 2020211209

学院： 电子工程学院

同组人： 余一帆

维特比译码器（2， 1， 9）的 VERILOG 实现

一、实验目的：

1. 学习如何设计大型逻辑电路。
2. 学习如何设计系统化、模块化的电路功能。
3. 学习如何描述电路的结构和行为。
4. 学习如何验证逻辑电路的功能和系统级别。
5. 学习如何编写测试程序。
6. 注意代码规范性要求。

二、实验内容：

0. 了解维特比译码原理
1. 建立维特比译码文件，将其综合，仿真，满足实际译码需要
2. 编写测试文件，观察并分析波形是否符合译码要求

三、设计思路

3.1 维特比译码原理以及初步实验构思

维特比译码器有以下几个特点：

1. 系统参数

首先，其系统参数(也就是题目设计中的 (2, 1, 9), (n,k,m)), 分别代表下列参数：在卷积码编码器中寄存状态为 2^{km} 各个。

其中 K 代表约束长度，N 代表存储路径的个数，而每个存储路径的宽度为 NL, L 则代表译码时存储器存储的码序列节点长度。

2. 工作原理

维特比实际是一种算法的名称，他根据路径汉明距离大小决定（作为权重）决定最大似然路径的选取。在这之中，它的具体工作方法是这样的。Viterbi 译码的整体过程是通过比较不同分支的度量值来选择最有可能的分支，然后更新状态的度量值，并根据比较结果生成状态转移表。最后，通过回溯算法使用状态转移表完成译码。

可以看出，在其中是需要编码，状态转移表，判决器，回溯等操作。

因此需要的区域有：编码器，存储器，译码器，数据选择器等区域来完成对应工作的。

3. VERILOG 的具体算法实现（分块过程）

具体来说，在 (2, 1, 9) 译码器中，译码器的算法如下

一 • 通过输入端口输入相应码元，输出的信息码元主要通过分支度量模块 (BMU) 计算汉明距离 (状态期望码元于接受信息码元之间的)，将累积得到的分支度量值传入 ACS 单元。

二 • 在 ACS 单元中进行加比选操作，使用前一条幸存路径进行度量计算于比较，产生判断向量 (eg.survivor) 在寄存器中储存，以此判断状态路径来源。

三 • 在 ACS 单元中进行加比选操作

经过一系列计算的得到最小路径，将其地址送入 TUB 单元，经过一系列反推得到幸存路径模块地址，从其中读出模块信息，再返回 TBU 计算译码，以回溯深度确定正确的译码，最后经过倒叙输出，得到结果。

3.2 分块实现维特比译码

一 • 设计步骤

经过以上分析，我们可以得到其大致的结构设计要求如下：

(1) 需求分析

需要分析程序所实现的输入条件和输出的要求，满足预期功能要求。并设置足够的输入参数与输出参数。

(2) 边界与功能定义

最主要的定义包括规则，状态，与时间。

(3) 关系定义。

这样一个定义，输出，输入，状态，规则，时刻之间的关系。实现正确无误的连接

(4) 结构

至少需要在输入之后存储规则进行状态判决，然后设置存储器，以保存输出变量。

(5) 接口

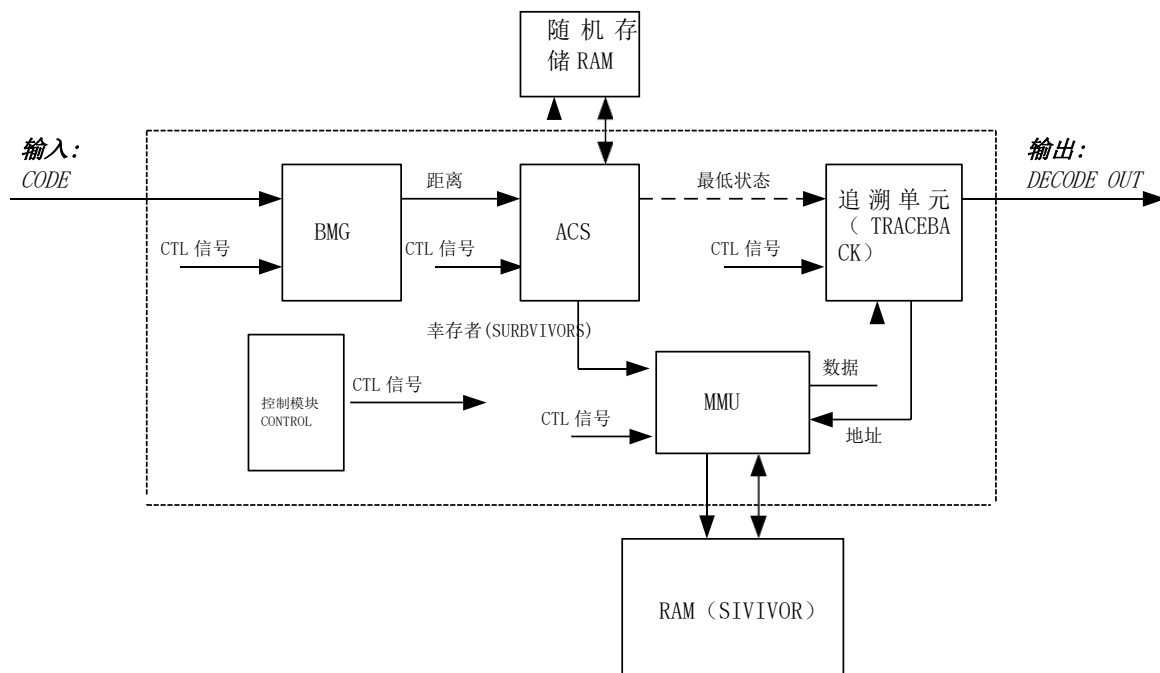
实现模块之间相互关系。紧密连接，传输变量的功能。

(6) 分模块设计

即是，最后的代码模块。包含最具体的逻辑设计。

二 • 主要结构

电路由五个主要组件来组成，维特比译码器分别是BMG。(分支度量生成器)单元、ACS(加法比较选择)单元、回溯单元、MMU(存储器管理单元)及其幸存存储器和控制单元。其电路连接图，如下图所示。下面将简要描述每个组件的功能。



图一。VDK (2, 1, 9) 框图

3.3 参数设定

设计规格:

码率=1/2, 约束长度(K) = 9

使用约束长度为 9 的一些结果如下:

分支度量的数量是 $2^K = 512$ 个分支。

状态度量的数量是 $2^{K-1} = 256$ 个状态。

追溯过程的深度至少= $5 * (K-1) = 40$ 。

VDK (2, 1, 9) 的以 63 个阶段的深度进行回溯。

设计考虑:

1.使用约束长度= 9 的卷积码主要用于 CDMA 应用。因此,输出数据速率至少应为 9.6 kbps。仅使用 4 个 ACS 处理器来执行相加-比较-选择操作来实现。

2.内存需求。

维特比解码器的最大部分是存储器。让我们以约束长度值为 9 为例。如果每个度量都是 8 位宽,则保存所有度量所需的内存大小为: $256 * 8 = 2.048$ 位。因此,我们需要 4.096 位的存储空间。最小追溯深度为 45, 幸者存储器大小为: $256(\text{状态}) * 40(\text{深度}) = 10.240$ 位。

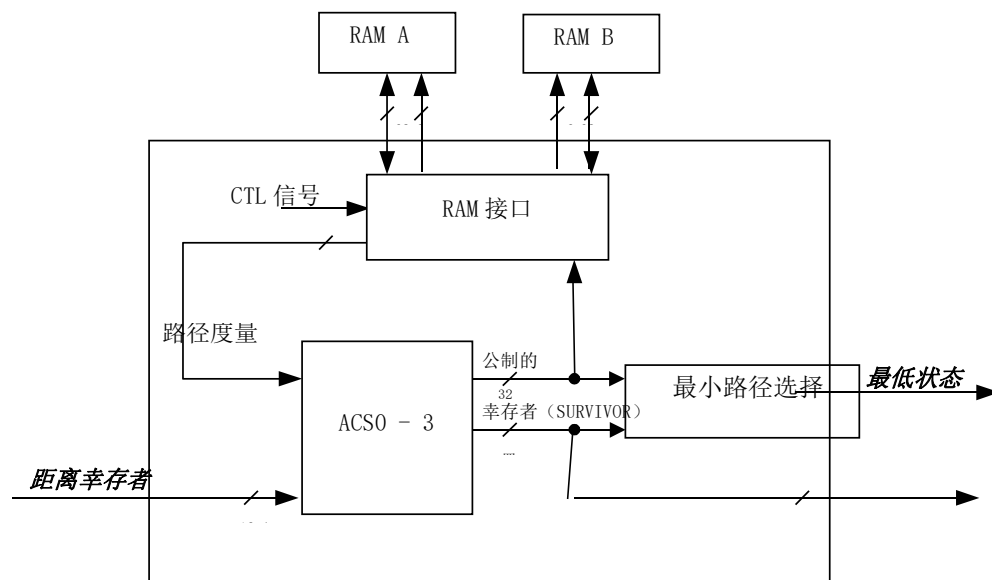
因此本实验分为 11 个代码块实现

1. asc.v (加比选比较器模块)

如图所示是ACS的单位框图。

这个加比选比较器由四个分块组成。ACS0 - ACS3.其中包括有RAM接口模块最小路径选择模块和比较模块。

图2 ACS 单元框图

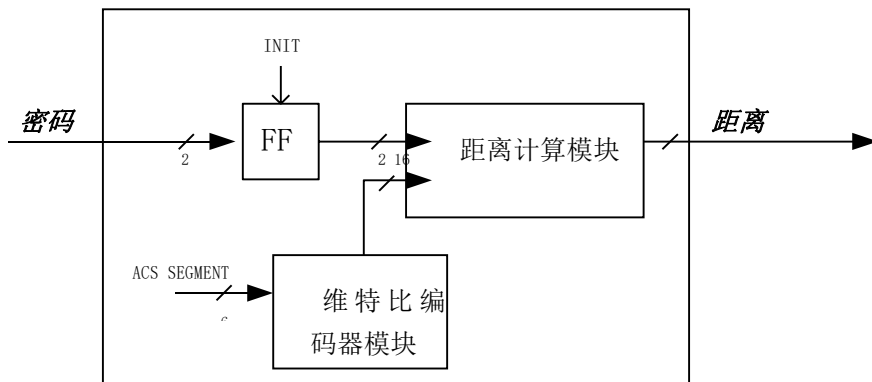


2. bmg.v(分支度量模块)

BMG 的功能是接收解码信号，用所有可能的分支度量计算机距离，然后给出距离信号的输出。

其中所有可能的分支度量值都是维特比编码器模块产生的。生成的值取决于 ACSegment 的值。

距离计算模块，使用的是来自 ViterbiEncoder 块的分支度量, 以此计算代码的硬距离。



如图所示是BMG的单位框图。图三 BMG单元框图

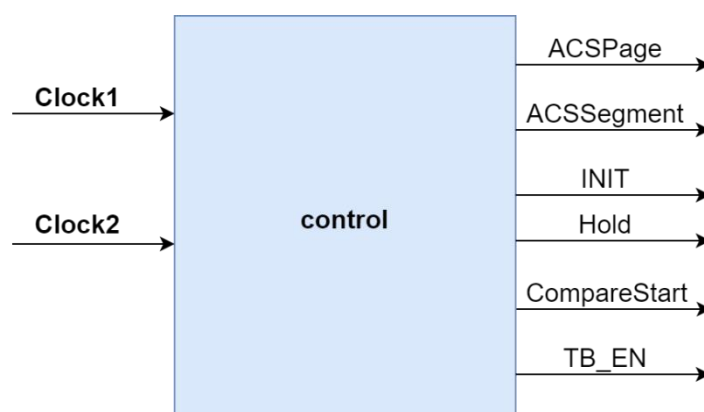
3. control.v（控制单元）

control 模块是一个控制模块，它与其他各个模块提供时钟、使能、迭代范围等控制信号，保证各模块有序进行解码。控制单元用于为其他组件提供控制信号。这些信号列于表一。

表一. 控制单元信号

信号名称	描述
Clock1和 Clock 2	时钟1和时钟2是频率等于的信号 时钟频率。 <ul style="list-style-type: none"> 两个信号相位相差90°。时钟2比时钟1超前90°。 控制单元使用时钟1更新ACS Segment (从而更新BMG单元和ACS单元的输出)。 Clock2用于指示何时将地址放入地址总线。
ACSPage [5:0]	指示当前活动页面内存。这里的“活动”是指 幸存者数据将被写入的存储器页面的位置。
ACS SEGMENT[5:0]	指示已处理的状态段。因为只有 4个可用的ACS，状态数为256，我们将需要64次迭代。
INIT	指示1个代码信号(解码器输入)的处理开始。
Hold	指示1个代码信号(解码器输入)的处理结束。
CompareStart	如网格图所示，对于第一个(K - 1)过程，在每个节点上，只 有一个输入进入。那么，幸存者一定是那些分支。 CompareStart将向ACS发送信号 在第一个(K-1)过程中不进行比较操作。
TB_EN	只有在幸存者存储器中至少有深度组的幸存者之后，追溯 过程才会开始。在版本1.0上，因为深度的值是63，那么回 溯 将在ACSPage值为63 (3F十六进制)时开始。

它的引脚图如图所示。



4. decoder.v (译码单元)

其中的主要部分已经在提示代码中给出。主要的用途是连接输入与输出以及地址总线来进行度量的计算

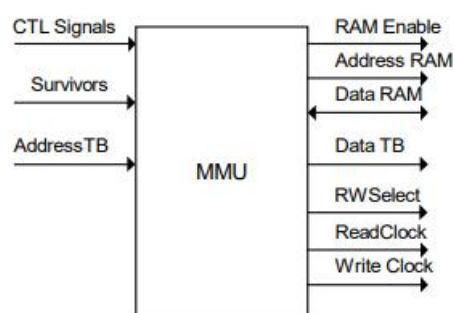
5. dff.v (触发器单元)

是卷积编码 `viterbi_encoder9` 模块; 的辅助此规则生成的 D 触发器模块 `dff`, 该模块实现卷积编码 中的移位操作。在验证时, 给卷积编码器输入相应的码字得出编码结果, 可以人为修改一些 内容模拟一下信道的干扰, 然后输入到译码器中检查这个译码结果和卷积编码器的输入是否 相同, 如果相同或者近似相同就可以验证这个算法的正确性。主要实现信号的复位, 数据接收等功能。

6. mmu.v (内存管理模块)

它的作用是保存度量值。它可以被称为度量储存器, 这需要两块 RAM 来形成, 这使得我们知道当前的度量值。并可以继续保存刚刚计算的下一个度量值。这需要 32 位的数据总线。

它的框图如图所示:



7. params.v (参数模块)

另外为了提高代码的可读性、优化的方便性, 这个模块里保存了所有实验中所需要的参数。包括输入输出引脚数目, 信号位宽等等。

8. ram.v (随机存储器)

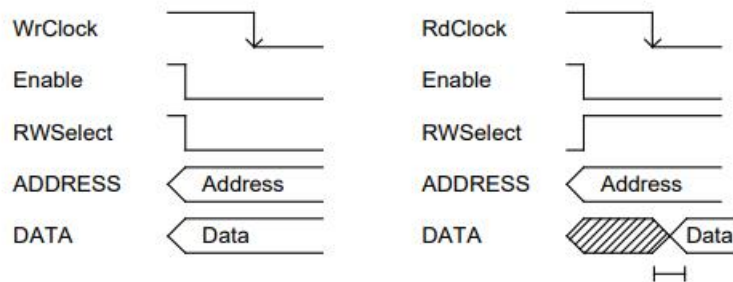
本题中使用的是一个典型的 RAM, 它有输入端口: 地址、使能、写时钟、

读时钟和 RWSelect，以及一个 inout 端口数据。

执行写操作时，我们必须将数据和地址放在数据总线和地址总线上，然后给出 WrClock 信号。为了确保信号的有效性，在我们把数据和地址放在线路上之后，在给出 WrClock 之前，必须有足够的时间。

要执行读操作，输入一个地址，给出一个 RdClock 信号，在延迟后，数据就可以在数据总线上获得了。

它的信号时序图如图所示。

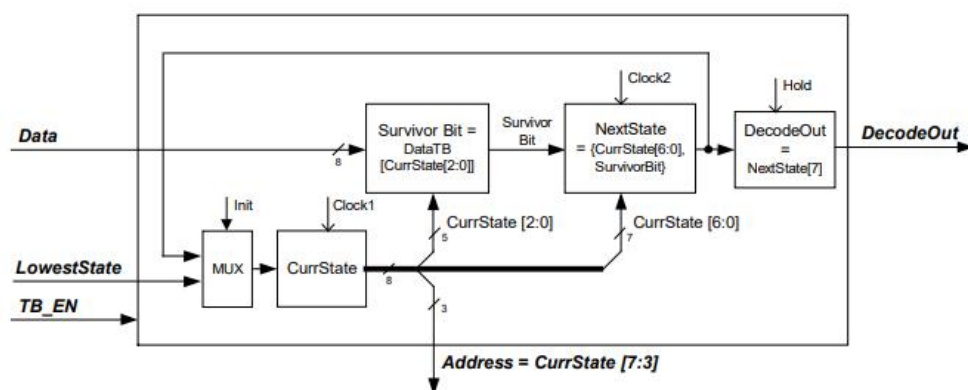


9. tbu.v (路径回溯单元)

在VDK (2, 1, 9) 上实现的回溯算法可以描述如下：

1. 在时间 t ，选择一个将开始回溯过程的节点。
在VDK9R1/2中，起始节点是具有最小度量值的状态。该值来自ACS装置。
2. 从残存物存储器中，获得那些节点的残存物值。
3. 通过将节点值左移1位来计算前一状态($t - 1$ 电平)，然后LSB部分将填充从存储器收集的幸存值。
4. 返回步骤2，了解 $t-1$ 级的状态。继续，直到 t 层的节点 - 深度。
5. 解码值是 t 层深度节点的MSB。

其框图如图所示：

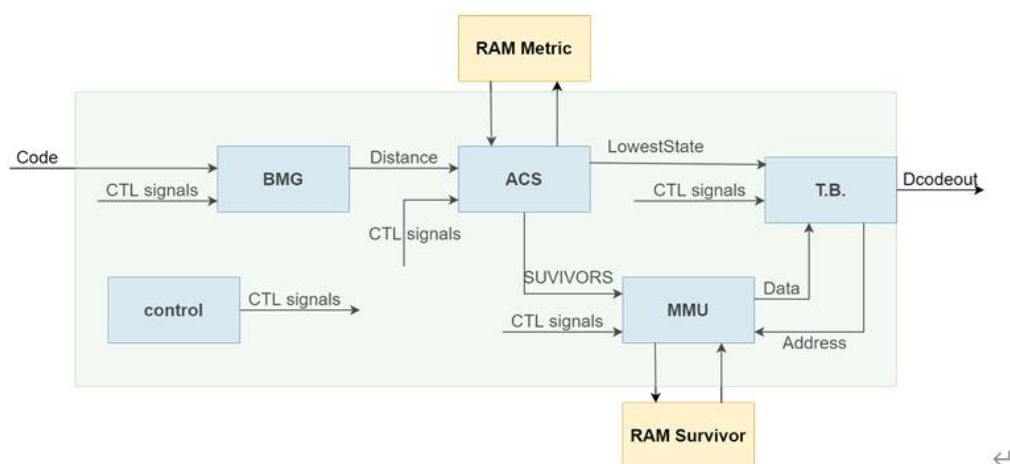


10.testbench.v (测试单元)

测试文件。包括输入以及输出解码结果。也是我们之后主要分析波形的目标。

11.Viterbi_encode9.v (综合完成单元)

综合输出的总模块，由所有子模块综合整合后得出。



四、实验记录

具体代码（重要部分）如下

1. asc.v（加比选模块）

ACS 模块由三个相同的 ASC 模块组成。（ACS0~ACS3 组成）

其中，每一个 ACS 模块均由两个全加器与两个异或门组成。相关的代码如下：

```

226 // 在RST置零, CLOCK2下降沿更新
227 always @(negedge Clock2 or negedge Reset)
228 begin
229     if (~Reset)
230     begin
231         Reg_Metric <= 0;
232         Reg_State <= 0;
233     end
234     else if (Active)
235     begin
236         if (Init)
237         begin
238             Reg_Metric <= Lowest_Metric4;
239             Reg_State <= Lowest_State4;
240         end
241         else
242         begin
243             Reg_Metric <= MetricCompareResult;
244             Reg_State <= StateCompareResult;
245         end
246     end
247 end

```

```

248
249 // 当HOLD有效与CLOCK1下降沿同时出现时,使得寄存器输出
250 always @(negedge Clock1 or negedge Reset)
251 begin
252     if (~Reset)
253     begin
254         LowestMetric <=0;
255         LowestState <= 0;
256     end
257     else if (Active)
258     begin
259         if (Hold)
260         begin LowestMetric <= Reg_Metric;
261             LowestState <= Reg_State;
262         end
263     end
264 end
265
266 endmodule
267 //比较得出低位
268

```

然后调用模块中之前已经写好的比较器和全加器模块在 ASC 主模块中。实施规则的比较功能。

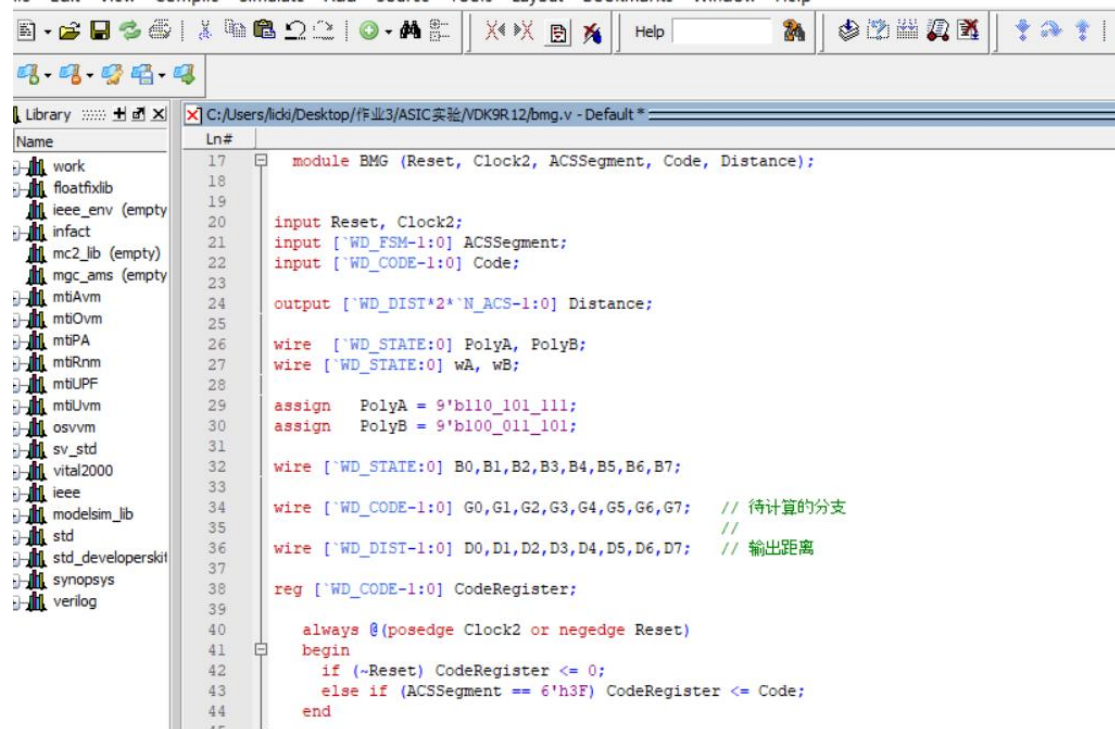
```

276
277 input Active;
278 input [`WD_FSM-1:0] ACSSegment;
279 input [`WD_METR-1:0] Metric3, Metric2, Metric1, Metric0;
280
281 output [`WD_STATE-1:0] Lowest_State4;
282 output [`WD_METR-1:0] Lowest_Metric4;
283
284 wire Surv1, Surv2, Surv3, Bit_One;
285 wire [`WD_METR-1:0] MetricX, MetricY;
286
287 // 比较 met1clri和 metric0
288 COMPARATOR comp1 (Active, Metric1, Metric0, Surv1);
289 // 比较 met1lri2和 metric3
290 COMPARATOR comp2 (Active, Metric3, Metric2, Surv2);
291
292 // MetricX --> Smaller metric between Metric1 and Metric0
293 // MetricY --> Smaller metric between Metric3 and Metric2
294 assign MetricX = (Surv1) ? Metric1:Metric0;
295 assign MetricY = (Surv2) ? Metric3:Metric2;
296
297 // Compare MetricY and MetricX.
298 COMPARATOR comp3 (Active, MetricY, MetricX, Surv3);
299
300 // 定义最小值状态
301 assign Bit_One = (Surv3) ? Surv2:Surv1;
302 assign Lowest_State4 = {ACSSegment, Surv3, Bit_One};
303
304 // 分配最小指值
305 assign Lowest_Metric4 = (Surv3) ? MetricY:MetricX;
306
307 endmodule

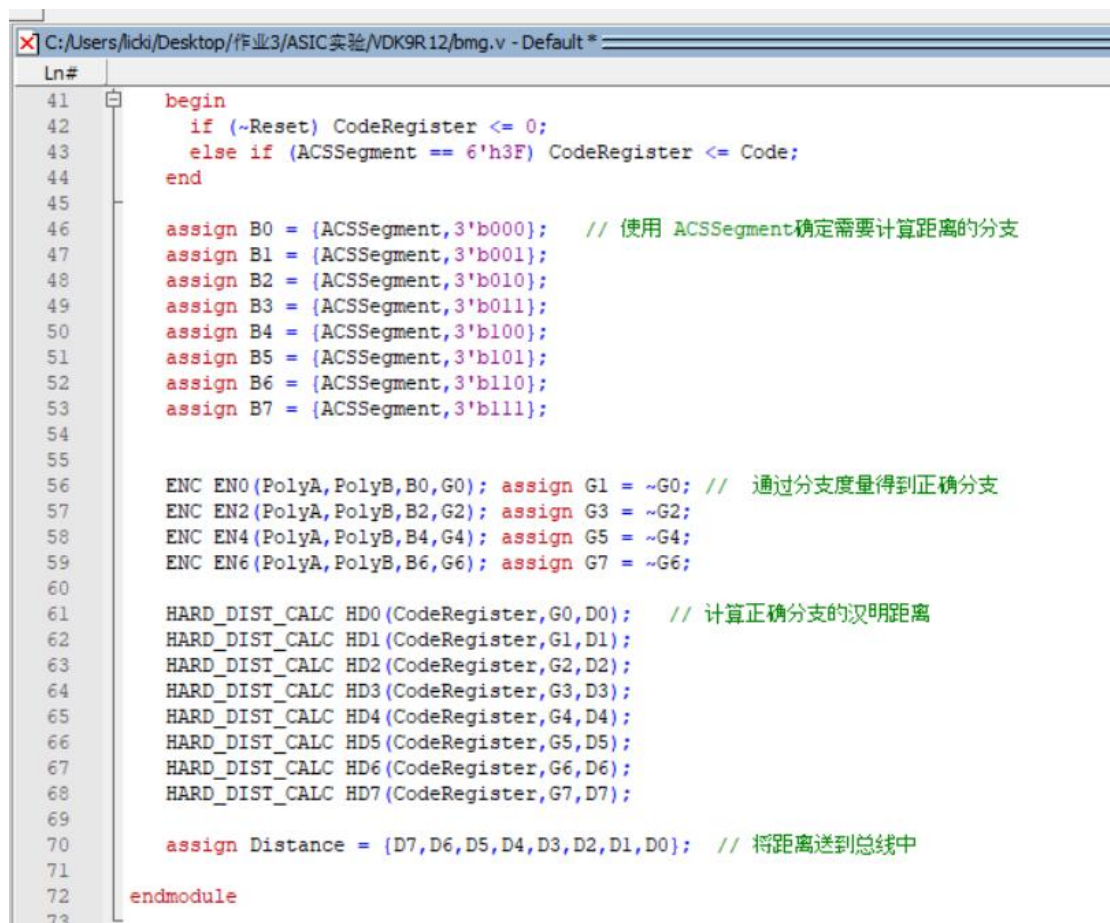
```

2. bmg.v (分支度量模块)

这个模块会计算根据 ACSSEGMENT。信号产生分支度量的取值来选择是否选择正确分支来计算并传输其汉明距离。这是通过调用维特比编码器模块和距离计算模块来形成的。最终计算得出的距离被传送到总线上。



```
Ln# 17 module BMG (Reset, Clock2, ACSegment, Code, Distance);
18
19
20 input Reset, Clock2;
21 input [`WD_FSM-1:0] ACSegment;
22 input [`WD_CODE-1:0] Code;
23
24 output [`WD_DIST*2*`N_ACS-1:0] Distance;
25
26 wire [`WD_STATE:0] PolyA, PolyB;
27 wire [`WD_STATE:0] wA, wB;
28
29 assign PolyA = 9'b110_101_111;
30 assign PolyB = 9'b100_011_101;
31
32 wire [`WD_STATE:0] B0,B1,B2,B3,B4,B5,B6,B7;
33
34 wire [`WD_CODE-1:0] G0,G1,G2,G3,G4,G5,G6,G7; // 待计算的分支
35 //
36 wire [`WD_DIST-1:0] D0,D1,D2,D3,D4,D5,D6,D7; // 输出距离
37
38 reg [`WD_CODE-1:0] CodeRegister;
39
40 always @(posedge Clock2 or negedge Reset)
41 begin
42 if (~Reset) CodeRegister <= 0;
43 else if (ACSSegment == 6'h3F) CodeRegister <= Code;
44 end
45
```



```
Ln# 41 begin
42 if (~Reset) CodeRegister <= 0;
43 else if (ACSSegment == 6'h3F) CodeRegister <= Code;
44 end
45
46 assign B0 = {ACSSegment, 3'b000}; // 使用 ACSegment 确定需要计算距离的分支
47 assign B1 = {ACSSegment, 3'b001};
48 assign B2 = {ACSSegment, 3'b010};
49 assign B3 = {ACSSegment, 3'b011};
50 assign B4 = {ACSSegment, 3'b100};
51 assign B5 = {ACSSegment, 3'b101};
52 assign B6 = {ACSSegment, 3'b110};
53 assign B7 = {ACSSegment, 3'b111};
54
55
56 ENC EN0(PolyA, PolyB, B0, G0); assign G1 = ~G0; // 通过分支度量得到正确分支
57 ENC EN2(PolyA, PolyB, B2, G2); assign G3 = ~G2;
58 ENC EN4(PolyA, PolyB, B4, G4); assign G5 = ~G4;
59 ENC EN6(PolyA, PolyB, B6, G6); assign G7 = ~G6;
60
61 HARD_DIST_CALC HD0(CodeRegister, G0, D0); // 计算正确分支的汉明距离
62 HARD_DIST_CALC HD1(CodeRegister, G1, D1);
63 HARD_DIST_CALC HD2(CodeRegister, G2, D2);
64 HARD_DIST_CALC HD3(CodeRegister, G3, D3);
65 HARD_DIST_CALC HD4(CodeRegister, G4, D4);
66 HARD_DIST_CALC HD5(CodeRegister, G5, D5);
67 HARD_DIST_CALC HD6(CodeRegister, G6, D6);
68 HARD_DIST_CALC HD7(CodeRegister, G7, D7);
69
70 assign Distance = {D7, D6, D5, D4, D3, D2, D1, D0}; // 将距离送到总线中
71
72 endmodule
73
```

3. control.v (控制模块)

```
C:/Users/lick/Desktop/作业3/ASIC实验/NDK9R12/control.v - Default *
Ln#
1  `include "params.v"
2
3  module CONTROL (Reset, CLOCK, Clock1, Clock2, ACSPage, ACSegment, Active,
4                  CompareStart, Hold, Init, TB_EN);
5
6
7  input Reset, CLOCK, Active;
8
9  output [`WD_FSM-1:0] ACSegment;
10 output [`WD_DEPTH-1:0] ACSPage;
11 output Clock1, Clock2;
12 output Hold, Init, CompareStart;
13 output TB_EN;
14
15 reg [`WD_FSM-1:0] ACSegment;
16 reg [`WD_DEPTH-1:0] ACSPage;
17
18 reg Init, Hold;
19
20 wire EVENT_1, EVENT_0;
21
22 reg TB_EN;
23
24 reg CompareStart;
25 reg [3:0] CompareCount;
26
27 reg count, Clock1, Clock2;
28
29 // 在这里定义时钟信号CLOCK1与clock2
30 always @(posedge CLOCK or negedge Reset)
31     if (~Reset) count <= 0; else count <= ~count;
32
33
34     always @(posedge CLOCK or negedge Reset)
35     begin
36         if (~Reset)
37         begin
38             Clock1 <= 0;
39             Clock2 <= 0;
40         end
41         else
42         begin
43             if (count) Clock1 <= ~Clock1;
44             if (~count) Clock2 <= ~Clock2;
45         end
46     end
47     // ---
48
49     assign EVENT_1 = (ACSSegment == 6'h3E);
50     assign EVENT_0 = (ACSSegment == 6'h3F);
51
52     always @(posedge Clock1 or negedge Reset)
53     begin
54         if (~Reset)
55         begin
56             {ACSPage, ACSegment} <= 'hFFFFFF;
57             Init <= 0;
58             Hold <= 0;
59             TB_EN <= 0;
60         end
61         else if (Active)
62         begin
63             // 增加ACSSegment 与 Page的值

```



```

62      // 增加ACSSegment 与 Page的值
63      {ACSPage,ACSSegment} <= {ACSPage,ACSSegment} + 1;
64
65      // 以Init 与 Hold signal 判断
66      if (EVENT_1) begin Init <= 0; Hold <= 1; end
67      else if (EVENT_0) begin Init <= 1; Hold <= 0; end
68      else begin {Init,Hold} <= 0; end
69
70      // 在63个survivor产生中, 使TB_EN使能置1
71      if ((ACSSegment == 'h3F') && (ACSPage == 'h3E')) TB_EN <= 1;
72  end
73 end
74
75 // 在最开始的K-1个, 不进行比较, 幸存分支默认为"0"分支
76 always @(posedge Clock2 or negedge Reset)
77 begin
78     if (~Reset)
79     begin
80         CompareCount <= 0;
81         CompareStart <= 0;
82     end
83     else begin
84         if (~CompareStart && EVENT_1) CompareCount <= CompareCount + 1;
85         if (CompareCount == `CONSTRAINT-1 && EVENT_0) CompareStart <= 1;
86     end
87 end
88
89 endmodule
90

```

4. tbu.v (回溯单元模块)

TRACEUNIT 模块包含输入和输出端口, 其中输入端口包括 Reset、Clock1、Clock2、Enable、InitState、Init、Hold 和 Survivor, 输出端口包括 OutState 和 AddressTB。TRACEUNIT 模块实现了 TBU 的状态转移和输出计算功能。

回溯单元 tbu 的主要作用就是在 init 初始化信号有效之后进行发送地址操作。发送地址的目标就是存储器模块, 并且可以从幸存存储器中取出相应的值。通过判断 survivor0/1。寻找出迭代路径, 通过多次迭代回溯所有的状态 (需要判断相应的状态转移模块)。当后的信号发出之后, 回溯模块会输出解码的结果。整个代码的作用是实现 Viterbi 解码器中的 TBU 单元, 用于解码卷积编码的数字信号。

```

1  `include "params.v"
2
3  |
4
5  module TBU (Reset, Clock1, Clock2, TB_EN, Init, Hold, InitState,
6             DecodedData, DataTB, AddressTB);
7
8  input Reset, Clock1, Clock2, Init, Hold;
9  input [`WD_STATE-1:0] InitState;
10 input TB_EN;
11
12 input [`WD_RAM_DATA-1:0] DataTB;
13 output [`WD_RAM_ADDRESS-`WD_FSM-1:0] AddressTB;
14
15 output DecodedData;
16
17 wire [`WD_STATE-1:0] OutStateTB;
18
19 TRACEUNIT tb (Reset, Clock1, Clock2, TB_EN, InitState, Init, Hold,
20              DataTB, AddressTB, OutStateTB);
21
22 assign DecodedData = OutStateTB [`WD_STATE-1];
23
24 endmodule
25
26 /*-----*/
27 module TRACEUNIT (Reset, Clock1, Clock2, Enable, InitState, Init, Hold,
28                  Survivor, AddressTB, OutState);
29 /*-----*/
30
31 input Reset, Clock1, Clock2, Enable;
32 input [`WD_STATE-1:0] InitState;
33 input Init, Hold;
34 input [`WD_RAM_DATA-1:0] Survivor;
35
36 output [`WD_STATE-1:0] OutState;
37
38 output [`WD_RAM_ADDRESS-`WD_FSM-1:0] AddressTB;
39
40 reg [`WD_STATE-1:0] CurrentState;
41 reg [`WD_STATE-1:0] NextState;
42 reg [`WD_STATE-1:0] OutState;
43
44 wire SurvivorBit;
45
46 always @(negedge Clock1 or negedge Reset)
47 begin
48     if (~Reset) begin
49         CurrentState <= 0; OutState <= 0;
50     end
51     else if (Enable)
52     begin
53         if (Init) CurrentState <= InitState;
54         else CurrentState <= NextState;
55
56         if (Hold) OutState <= NextState;
57     end
58 end
59
60 assign AddressTB = CurrentState [`WD_STATE-1:`WD_STATE-5];
61

```

```

61
62     always @(negedge Clock2 or negedge Reset)
63     begin
64         if (~Reset) NextState <= 0;
65         else
66             if (Enable) NextState <= {CurrentState [`WD_STATE-2:0], SurvivorBit};
67         end
68
69     assign SurvivorBit =
70         (Clock1 && Clock2 && ~Init) ? Survivor [CurrentState [2:0]]:'bz;
71
72 endmodule

```

5. testbench.v (测试模块)

测试模块分为 VD 和 ERR_VD，分别代表正确和错误的情况。

设定不同的输入代码，验证输出是否正确。

VD 模块代码如下

:

```

C:\Users\liki\Desktop\作业3\ASIC实验\VDK9R12\testbench.v - Default *
Ln#
1  `include "params.v"
2
3  `define D_PER
4
5  /*****
6
7  module VD();
8
9
10     reg CLOCK;
11     initial CLOCK = 0;
12     always #(`HALF/2) CLOCK = ~CLOCK; //设置CLOCK, RESET DRESET等参数
13
14     reg Reset;
15     reg DRESET;
16
17     initial begin
18         DRESET = 1;
19         Reset = 1;
20         #200 Reset = 0; DRESET=0;
21         #300 Reset = 1;
22         DRESET = 1;
23     end
24
25     reg X;
26     wire [`WD_CODE-1:0] Code; //输入X, 为之后判断输出做准备
27     initial X = 0;
28     initial begin
29         #475 X = 1;
30         #`DPERIOD X = 1;
31         #`DPERIOD X = 1;
32         #`DPERIOD X = 1;
33         #`DPERIOD X = 1;
34         #`DPERIOD X = 0;

```



```

103     end
104
105     reg D_CLOCK;
106     initial D_CLOCK = 0;
107
108     always #(`DPERIOD/2) D_CLOCK <= ~D_CLOCK; //设置DCOLCK信号
109
110
111
112     viterbi_encode9 enc(X,Code,D_CLOCK,DRESET);
113
114     reg Active;
115     always @(Code or Reset) //
116     if (~Reset) Active <= 0; //
117     else if (Code!=0) Active <= 1; // RESET不起作用，CODE为1时，状态为ACTIVE
118
119     wire DecodeOut;
120
121     VITERBIDECODER vd (Reset, CLOCK, Active, Code, DecodeOut); //链接VITERBIDECODER 模块
122
123 endmodule
124
125
126

```

```

127 module VD_err();
128
129     reg CLOCK;
130     initial CLOCK = 0;
131     always #(`HALF/2) CLOCK = ~CLOCK; //设置CLOCK信号
132
133     reg Reset;
134
135     initial begin
136         Reset = 1;
137         #200 Reset = 0; //设置RESET信号
138         #300 Reset = 1;
139     end
140
141     reg [`WD_CODE-1:0] CorrectCode;
142
143     initial CorrectCode = 2'b00;
144
145     initial begin
146         #475 CorrectCode = 2'b11; //设定正确代码2'b11
147         #`DPERIOD CorrectCode = 2'b10; //2'b10 10
148         #`DPERIOD CorrectCode = 2'b11; //2'b11 11
149         #`DPERIOD CorrectCode = 2'b00; //2'b00 * 10
150         #`DPERIOD CorrectCode = 2'b10; //2'b10 10
151         //2
152         #`DPERIOD CorrectCode = 2'b11; //2'b11 11
153         #`DPERIOD CorrectCode = 2'b01; //2'b01 * 00
154         #`DPERIOD CorrectCode = 2'b01; //2'b01 01
155         #`DPERIOD CorrectCode = 2'b10; //2'b10 10
156         #`DPERIOD CorrectCode = 2'b01; //2'b01 * 11

```

```

    reg [`WD_CODE-1:0] Code;
    initial Code = 2'b00;
    initial begin
        #475 Code = 2'b11; //设置读入CODE2'b11 11
        #`DPERIOD Code = 2'b10; //2'b10 10
        #`DPERIOD Code = 2'b11; //2'b11 11
        #`DPERIOD Code = 2'b10; //2'b10 * 10
        #`DPERIOD Code = 2'b10; //2'b10 10
        //2
        #`DPERIOD Code = 2'b11; //2'b11 11
        #`DPERIOD Code = 2'b00; //2'b01 * 00
        #`DPERIOD Code = 2'b01; //2'b01 01
        #`DPERIOD Code = 2'b10; //2'b10 10
        #`DPERIOD Code = 2'b11; //2'b01 * 11
        //3
        #`DPERIOD Code = 2'b10; //2'b10 10
        #`DPERIOD Code = 2'b10; //2'b10 10
        #`DPERIOD Code = 2'b01; //2'b00 * 01

```

```

88     #`DPERIOD Code = 2'b00;          //2'b00          00
89
90 //12    #`DPERIOD Code = 2'b10;          //2'b10          10
91         #`DPERIOD Code = 2'b11;          //2'b11          11
92         #`DPERIOD Code = 2'b00;          //2'b00          00
93     end
94
95     reg Active;
96
97     always @(Code or Reset)            // 设定的RESET ACTIVE DCLOCK
98     if (~Reset) Active <= 0;           // 同样链接到VITERBIDECODER说
99     else if (Code!=0) Active <= 1;     // 获得输出DECODEOUT
100
101
102     reg D_CLOCK;
103     initial D_CLOCK = 0;
104
105     always #(`DPERIOD/2) D_CLOCK <= ~D_CLOCK;
106
107     wire DecodeOut;
108
109     VITERBIDECODER vd (Reset, CLOCK, Active, Code, DecodeOut);
110 endmodule

```

6.mmu.v（内存管理模块）

```

1  `include "params.v"
2
3  module MMU (CLOCK, Clock1, Clock2, Reset, Active, Hold, Init, ACSPage,
4             ACSegment_minusLSB, Survivors,
5             DataTB, AddressTB,
6             RWSelect, ReadClock, WriteClock,
7             RAMEnable, AddressRAM, DataRAM);
8
9
10 // ;链接到Control模块
11 input CLOCK, Clock1, Clock2, Reset, Active, Hold, Init;
12 input [`WD_DEPTH-1:0] ACSPage;
13 input [`WD_FSM-2:0] ACSegment_minusLSB;
14
15 // i连接到ACS Unit模块
16 input [`N_ACS-1:0] Survivors;
17
18 // 双向连接 TB Unit
19 output [`WD_RAM_DATA-1:0] DataTB;
20 input [`WD_RAM_ADDRESS-`WD_FSM-1:0] AddressTB;
21
22 // 双向连接RAM
23 output RWSelect, ReadClock, WriteClock, RAMEnable;
24 output [`WD_RAM_ADDRESS-1:0] AddressRAM;
25 inout [`WD_RAM_DATA-1:0] DataRAM;
26
27 wire [`WD_RAM_DATA-1:0] WrittenSurvivors;
28
29 reg dummy, SurvRDY;
30 reg [`WD_RAM_ADDRESS-1:0] AddressRAM;
31 reg [`WD_DEPTH-1:0] TBPage;
32
33 wire [`WD_DEPTH-1:0] TBPage_;
34 wire [`WD_DEPTH-1:0] ACSPage;
35 wire [`WD_TB_ADDRESS-1:0] AddressTB;

```

```

36
37 // 设置读写时钟
38
39 // Write Clock 2个周期发生一次
40 always @(posedge Clock2 or negedge Reset)
41     if (~Reset) dummy <= 0; else if (Active) dummy <= ~dummy;
42
43 assign WriteClock = (Active && ~dummy) ? Clock1:0;
44 assign ReadClock = (Active && ~Hold) ? ~Clock1:0;
45 // --
46
47 // Survivor Buffer () 缓冲器
48 // Data Bus 宽为 8比 ACS output () 4bit多
49 //
50
51 always @(posedge Clock1 or negedge Reset)
52     if (~Reset) SurvRDY <= 1; else if (Active) SurvRDY <= ~SurvRDY;
53
54 ACSSURVIVORBUFFER buff (Reset, Clock1, Active, SurvRDY, Survivors,
55                          WrittenSurvivors);
56 // --
57
58 //
59 // 每当CLOCK2下降沿时, TBPAGE-1, INIT为ACTIVES时TBPAGE=ACSPage - 1
60 //
61 always @(negedge Clock2 or negedge Reset)
62 begin
63     if (~Reset) begin
64         TBPage <= 0;
65     end
66     else if (Init) TBPage <= ACSPage-1;
67     else TBPage <= TBPage_;

```

Ln#	
65	end
66	else if (Init) TBPage <= ACSPage-1;
67	else TBPage <= TBPage_;
68	end
69	
70	assign TBPage_ = TBPage - 1;
71	
72	assign RAMEnable = 0;
73	assign RWSelect = (Clock2) ? 1:0;
74	assign DataRAM = (~Clock2) ? WrittenSurvivors:'bz';
75	assign DataTB = (Clock2) ? DataRAM:'bz';
76	//对于RAM, 每当CLOCK2改变时, 地址线需要被设定
77	//以便读写时钟 (iDCLOCK1) 来临时做好准备!
78	// every time Clock2 changes, the Address and Enable for each RAM has to
79	
80	always @(posedge CLOCK or negedge Reset)
81	begin
82	if (~Reset) AddressRAM <= 0;
83	else
84	if (Active) begin//写周期开始
85	if (Clock2 == 0)
86	begin
87	AddressRAM <= {ACSPage, ACSSegment_minusLSB};
88	end//读周期进行
89	else
90	begin
91	AddressRAM <= {TBPage [WD_DEPTH-1:0], AddressTB};
92	end
93	end
94	end
95	//--
96	
97	endmodule
98	

```

96
97   endmodule
98
99   /*-----*/
100  module ACSSURVIVORBUFFER (Reset, Clock1, Active, SurvRDY, Survivors,
101                           WrittenSurvivors);
102  //
103  // 为了适应8位宽RAM数据总线的使用, SURVIVOR
104  // 每个时钟上只有 4 个 必须首先缓冲。
105  /*-----*/
106
107  input Reset, Clock1, Active, SurvRDY;
108
109  input [`N_ACS-1:0] Survivors;
110
111  output [`WD_RAM_DATA-1:0] WrittenSurvivors;
112
113  wire  [`WD_RAM_DATA-1:0] WrittenSurvivors;
114  reg  [`N_ACS-1:0] WrittenSurvivors_;
115
116  always @(posedge Clock1 or negedge Reset)
117  begin
118      if (~Reset) WrittenSurvivors_ = 0;
119      else if (Active)
120          WrittenSurvivors_ = Survivors;
121      end
122
123  assign WrittenSurvivors = (SurvRDY ? {Survivors, WrittenSurvivors_}:8'bz;
124
125  endmodule

```

在其中测试的只有 **testbench.v** 文件（以及其他一些模块的 **testbench** 文件），他是维特比实验的测试文件，也是之后进行波形分析的主要对象。

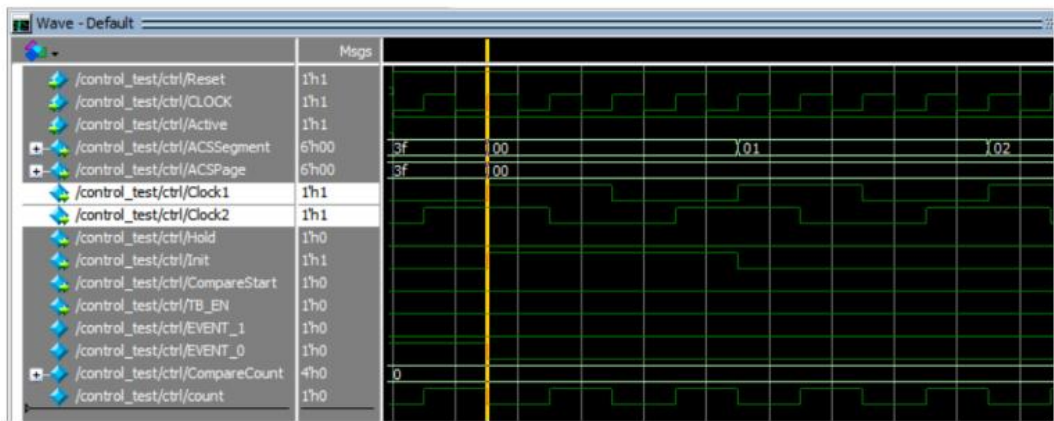
五、结果分析

仿真分析

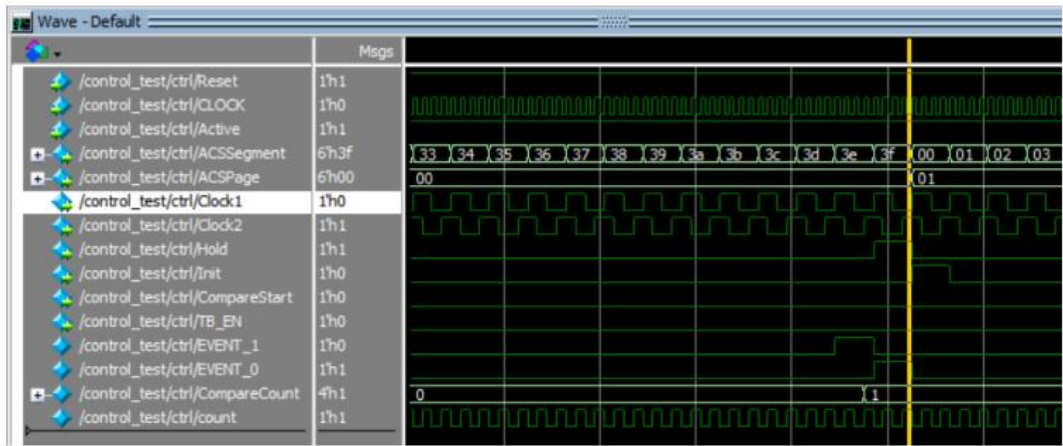
1.control.v（控制模块）

该模块包含以下几个功能：

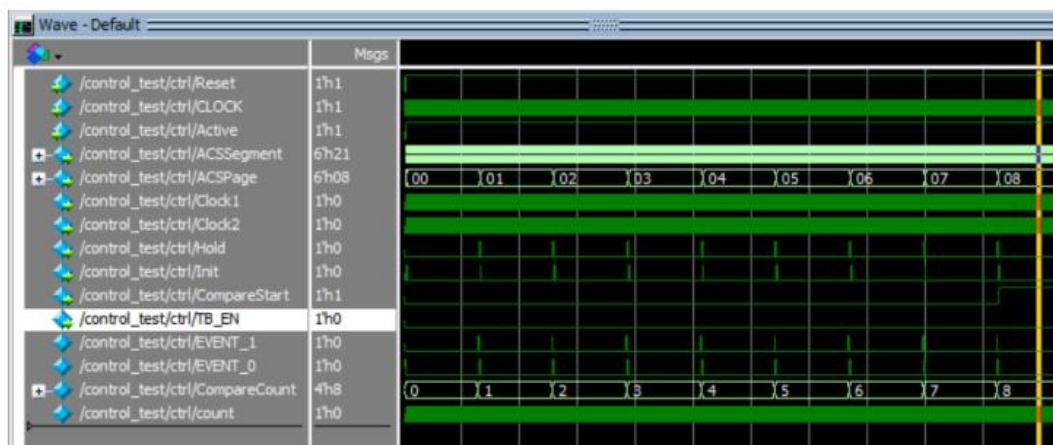
（1）生成 clock1 和 clock2 时钟，两者相位差为 90° 两者周期一样，都是总时钟的分频，T=400NS。时钟信号如下图。



（2）生成时刻跳转、迭代次数、码字开始提示信号及码字结束提示信号。一个码字的处理时间就是计数 64 个 ACSsegment。一个码字 在 4 个并行 ACS 单元下迭代 64 次，才能完成 256 个状态的路径度量值比较。



（3）同时，在 Clock2 的上升沿。comparecount 计数到 8 以后并且 ACSsegment = 6'h3F（init）。 CompareStart <= 1 开始比较



2.BMG.v（分支度量器模块）

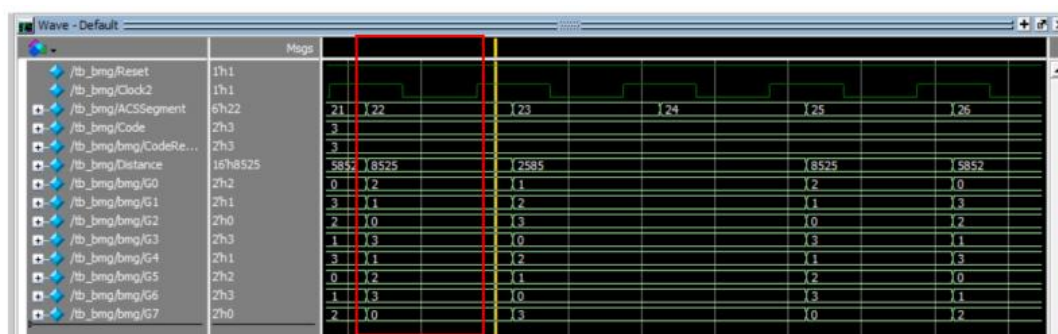
本模块只与 ACS 模块有关联，它为 ACS 模块提供八个分支的分支度量结果。供加比选进行计算。

需要实现的功能是：

（1）分支度量计算验证。

此模块功能为，给定一个 2 位的 code，输出其与 ACSegment 对应的 8 条支路相比 较的汉明距离结果。

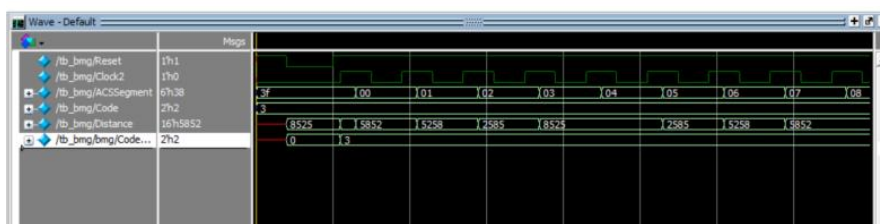
具体运算结果如下，输入的 code 为 11，输入的 ACSegement=6'b100010，则 8 个分支为 100010_000 到 100010_111。根据生成多项式求得这 8 个分支的输出为 00 11 10 01 11 00 01 10 与 code 进行汉明距离的计算结果为 10 00 01 01 00 10 01 01。



（2）复位功能

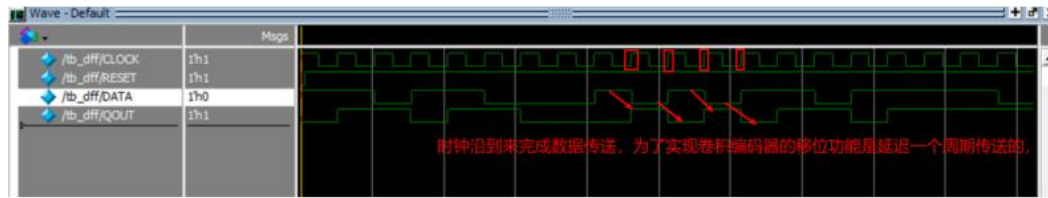
Reset 信号为 0 时输出 Distance

Reset 信号为 1 时，ACSSegment 为 3FH 时，且 code 传给 coderegister 此时计算汉明距离

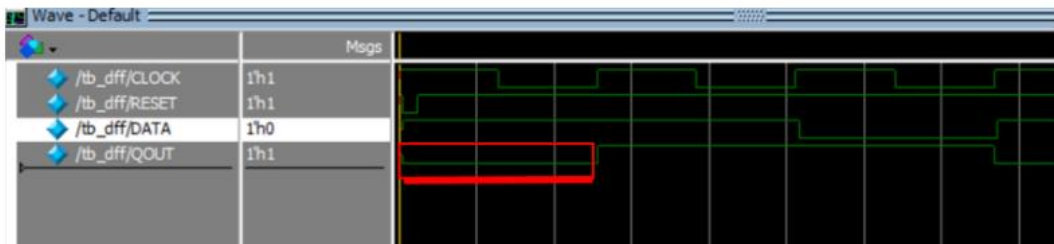


3.dff.v（触发器模块）

需要实现的是置位，复位功能，并且未来能够实现卷积编码 原理中的移位功能 提供了前提。要求在时钟上升沿到来之前实现数据传输



reset 等于零的时候要完成复位。复位信号如下。



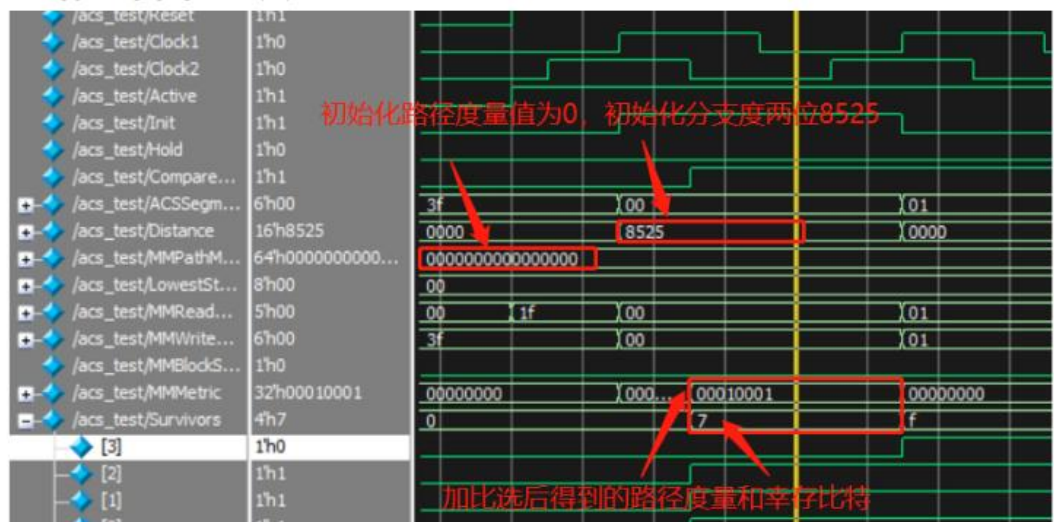
4. ASC.v（加比选模块）

绝对的译码核心单位。具体功能为：ACS 单元从 ram 中取出前一时刻幸存路径度量值，然后与送入 ACS 单元的分支度量值累计相加进行比较，相加度量值较小的 作为该状态新的路径度量值存入 RAM 里的 MetricMemory 以备下一时刻加比选使用。并产生幸存比特信息 survivor 给 MMU 模块。

要求实现的功能如下：

- （1）加比选功能：实现八个分支值的 DISTANCE 距离度量。得到的输出在 MMMetric=00_01_00_01， Survivors=0111（7）种。具体数值如下图。

Distance 显示 8525。路径度量值初始化为 0，进行四个状态的加比选。经过加比选，得到四状态的路径度量值和幸存路径信息。



5. RAM.v (随机存储模块)

该模块与 ACS 模块，MMU 模块直接相连，根据地址读写幸存比特一次一个字节；根据地址读写路径度量值，读时一次 8 个字节，写时一次 4 个字节。然后输入 ACS 用于计算。

它的功能如下

(1) 传送 SUIVIVOR 功能：

每当 RWSelect 读 写选择下降沿的时候，将 data 自加 1，验证在相应的时刻是否能够将 data 中的数据传送到 DataRAM 中去，同时 AddressRAM 也自加 1，即定义了数据和地址。



(2) 路径度量值传送功能

Metricmemory 度量存储器模块，负责向 ACS 模块提供上一时刻的路径度量值，同时，ACS 加比选完成后，再将新的幸存路径存储进来。



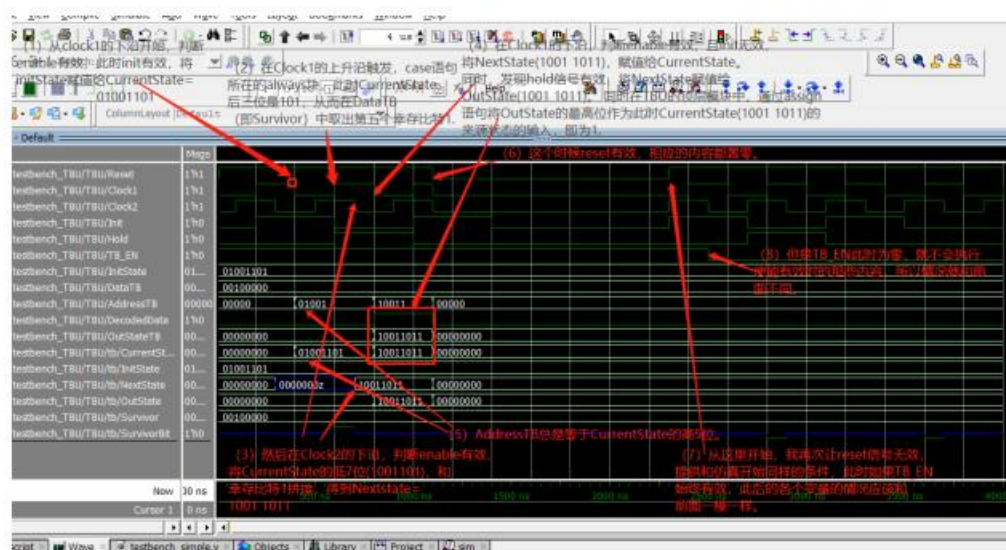
当 ACSPage 为 01 的时候，从 B 区读出相应的路径度量值，并赋值给 MMPPathMetric=0010101001010111;。

6. TBU.v (随机存储模块)

本模块主要是通过所合成的地址在 ram 中取得 8 位的幸存比特，然后再模块内部通过 case 语句取得确定的 1 位幸存比特，从而推出前一状态，然后在相应的 时机将译码结果输出。

要求功能如下：

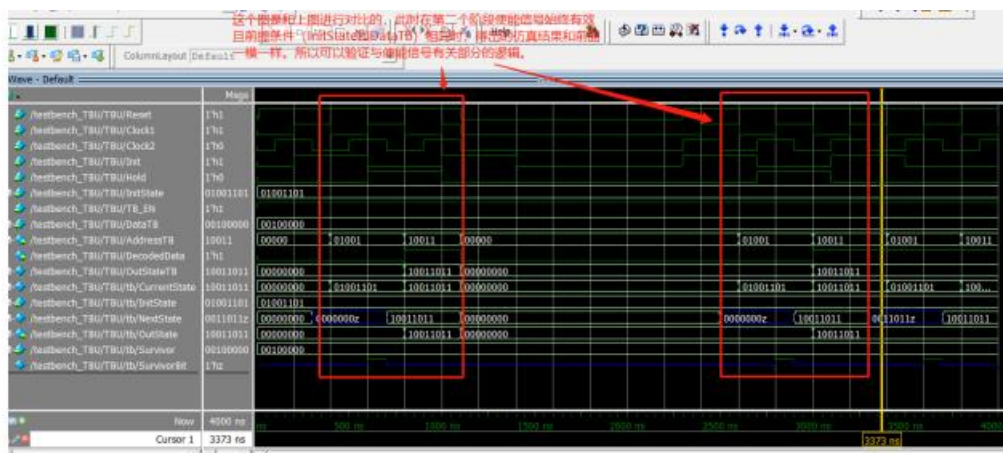
(1) 状态跳转及译码输出



从时序图可以发现，在时钟1的下

降沿将 `initState` 复制给了 `currentState`，在时钟1的上升沿触发了 `case` 语句，取出了此状态的幸存比特1，在时钟2的下降沿将幸存比特和 `currentState` 的后七位进行拼接得到 `nextState=1001_1011`；然后在时钟1的下降沿将 `nextstate` 赋值给 `currentState`，完成状态的跳转，同时在这个边沿 `hold` 信号有效，会将 `nextState` 赋值给 `outstate`，最终取出 `outstate` 的最高位1作为本次译码的输出

(2) 使能信号验证



和上图进行对比，可以发现：在右边框出的区域内，`TB_EN` 信号持续为高电平，此时初始条件 `initstate` 和 `dataTB` 都不变，所以右边框内和左边框内的执行结果完全相同，而在上图中右边框内的 `TB_EN` 信号无效，所以执行结果不同。通过对比可以得出此模块的使能信号在正常得发挥作用。

7. VD.v (整合模块)

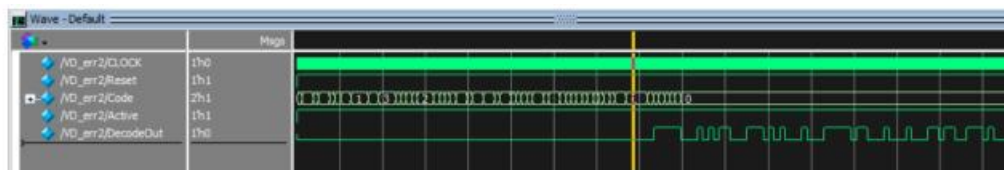
在这里面实现了整体的仿真，实现了其他模块的调用，从而将系统进行了整合，因此整体模块的验证也即对 VD 模块的验证。

仿真结果分析：

(1) 它的初始信号设置如下：

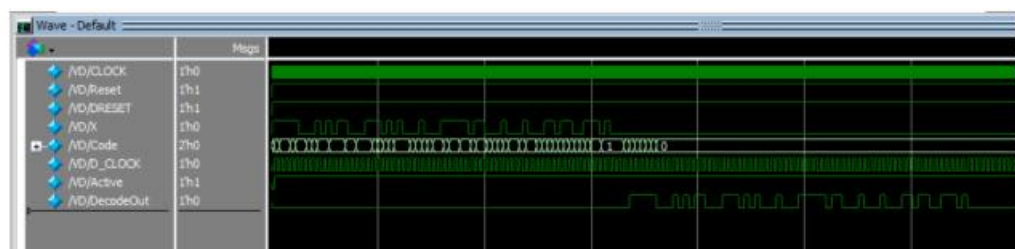
在 testbench 中建立原始时钟信号 CLOCK，周期是 100ns，频率 10MHz。再建立复位信号，低有效；Active 信号为 1，开始工作；创建编码器时钟 D_CLOCK，为卷积编码器提供驱动型号。最后设置输入码字 X 或者卷积码 code。

(2) 无编码器仿真



直接输入卷积码 CODE 给解码器，解码器输出 DECODEOUT。如图正确解码，下图的输入卷积码是和上图 X 输入，在编码器生成的卷积码一致。由图可见，波形一致，仿真正确。

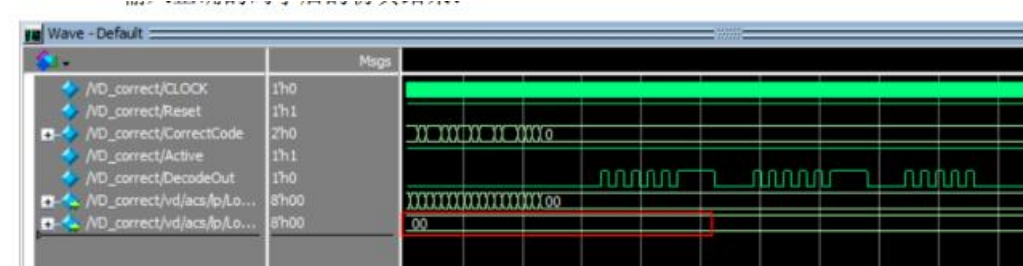
(3) 有编码器仿真



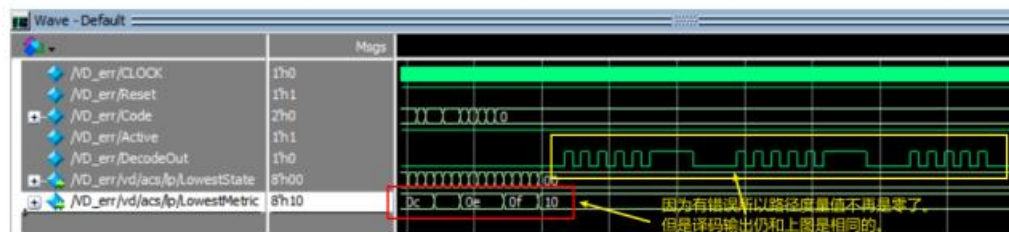
输入码字 X 给卷积编码器，编码器输出卷积码 CODE 给解码器，解码器输出 DECODEOUT。如图正确解码。

(4) 纠错能力

输入正确的码字后的仿真结果：



因为维特比会选择一个路径度量最小的状态回溯，也即差错最小。可以看到错误代码中，lowestmetric 不再为 0，说明有差错，差错状态的路径度量值为 10h. 可见最小度量的值不再是零，但是这个译码输出的结果仍然和上图一致。证明了算法具备一定的纠错能力。



六、实验总结

首先，在进行本实验前。我就使用 Verilog 语言实现维特比译码器，查阅了相关资料（CSDN,B 站），了解 Verilog 语法和状态机的基本知识。在实验过程中，需要测试各个模块，确保它们能够正常工作，并且能够与其他模块进行通信。

在实现维特比译码器时，需要使用内存和总线来存储和传输数据。这些组件需要仔细设计和调试，以确保它们能够正常工作，并且能够满足需求。

在实验过程中，可能会遇到一些问题和挑战。这些问题可能涉及到 Verilog 语法、状态机设计、内存和总线的使用等方面。

在实验中我遇到的具体问题如下。

1. 在编写测试模块中不知道如何设置 clk 与 D_clk，但同时也不知道在何时设置 REST 比较好。还有一个问题，就是输出 Decode out 解码的输出结果时经常显示高阻态与不定态。这对于我们编写测试文件的要求很高，因此我查阅资料得到了结果。
2. 在回溯单元模块时编写状态机的状态。需要先确定其他变量的值。对于逻辑要求比较高。并且写出的状态，其应达到最简化。这是要反复推敲。
3. 在分支度量模块中，我一度不知道怎么判断汉明距离的最小值。计算汉明距离以及使用 survivor 来进行回溯。通过阅读。实验手册的译码器原理，我得到了正确答案。
4. 我个人对端口的连接与使用并不是很了解，因此在将全加器接入。比如加选模块，就是 ASC 模块中是遇到了问题。另外，在测试模块的。Decoder output，的设置方面，我也遇到了问题。我通过复习课件以及查阅一些资料。了解了端口的正确使用使用方法。

总之，使用 Verilog 语言实现维特比译码器是一项有趣和有挑战性的实验。通过认真学习和实践，我掌握了 Verilog 语法和状态机设计的基本知识，同时也可以提高我的编程和调试能力。

最后，感谢韩老师以及助教老师的辛勤付出。

七、实验代码（附录）

1PARAM.v（整合模块）

```
*****/
//function: 存储器单元 RAM:包括留存路径存储器和路径度量存储器
/*****/
`include "params.v"

/*****调用 RAM 模型构成留存路径存储器*****/
module RAM (RAMEnable,AddressRAM,DataRAM,RWSelect,ReadClock,WriteClock);
input RAMEnable, RWSelect, ReadClock, WriteClock;
input [`WD_RAM_ADDRESS-1:0] AddressRAM;
inout [`WD_RAM_DATA-1:0] DataRAM;

RAMMODULE #(2048,8,11) ram (RAMEnable,DataRAM,AddressRAM,RWSelect,ReadClock,
WriteClock);

endmodule

/*****对 RAM 进行建模*****/
module RAMMODULE (_Enable,Data,Address,RWSelect,RClock,WClock);
parameter SIZE=2048; /*定义存储单元数*/
parameter DATABITS=8; /*定义各存储单元的空间*/
parameter ADDRESSBITS=11; /*定义地址位宽*/

input RClock,WClock,_Enable; /*读时钟、写时钟、工作使能信号*/
input [ADDRESSBITS-1:0] Address; /*地址总线*/
input RWSelect; /*读写模式选择信号,RWSelect=0 时向 RAM 写入数
据,RWSelect=1 时从 RAM 读出数据*/
inout [DATABITS-1:0] Data; /*双向数据总线*/

reg [DATABITS-1:0] Data_Regs [SIZE-1:0];/*RAM 空间*/
reg [DATABITS-1:0] DataBuff; /*数据总线缓冲单元*/

/*对 RAM 进行写操作*/
always@(negedge WClock)
begin
    if(!_Enable)/*RAM 使能*/
    begin
        if(!RWSelect)/*允许写入*/
            Data_Regs [Address]<=Data;
        else/*进行读操作时保证 RAM 内数据不变*/

```

```

        Data_Regs [Address]<=Data_Regs [Address];
    end
else/*RAM 未使能*/
    Data_Regs [Address]<=Data_Regs [Address];
end

/*对 RAM 进行读操作*/
always@(negedge RClock)
begin
    if(!_Enable)/*仅在进行读操作时,DataBuff 才会影响总线输出,进行写操作时不关心 DataBuff
    的取值;故进行写操作时 DataBuff 取值保持不变,减少电平翻转次数以降低功耗*/
    begin
        if(RWSelect)/*允许读取*/
            DataBuff<=Data_Regs [Address];
        else
            DataBuff<=DataBuff;
        end
    else
        DataBuff<=DataBuff;
    end
end

/*控制数据总线*/
assign Data=(RWSelect)?DataBuff:'bz; /*RWSelect=1 时,RAM 处于被读取状态,对总线输出数
据;RWSelect=0 时,RAM 处于被写入状态,不对总线进行控制*/

endmodule

```

2.VD.v (整合模块)

```
`include "params.v"
`include "MMU.v"
`define D_PER
`timescale 1ns/1ns
module VD();
wire Clock1,Clock2;
assign Clock1=vd.Clock1;
assign Clock2=vd.Clock2;

wire Init,Hold;
assign Init=vd.Init;
assign Hold=vd.Hold;

wire [`WD_DIST*2*`N_ACS-1:0] Distance;
assign Distance=vd.Distance;

wire [`N_ACS-1:0] Survivors;
assign Survivors=vd.Survivors;

wire SurvRDY;
assign SurvRDY=vd.mmu.SurvRDY;

wire [`WD_RAM_DATA-1:0] WrittenSurvivors;
assign WrittenSurvivors=vd.mmu.WrittenSurvivors;

wire WriteClock;
assign WriteClock=vd.mmu.WriteClock;

wire ReadClock;
assign ReadClock=vd.mmu.ReadClock;

wire [`WD_FSM-1:0] ACSegment;
assign ACSegment=vd.ACSegment;

wire [`WD_DEPTH-1:0] ACSPage;
assign ACSPage=vd.ACSPage;

wire [`WD_RAM_ADDRESS-1:0] AddressRAM;
assign AddressRAM=vd.AddressRAM;

wire [`WD_DEPTH-1:0] TBPage;
assign TBPage=vd.mmu.TBPage;
```

```
wire [`WD_STATE-1:0] CurrentState,NextState;
assign CurrentState=vd.tbu.CurrentState;
assign NextState=vd.tbu.NextState;
```

```
wire [`WD_RAM_DATA-1:0] DataTB;
assign DataTB=vd.DataTB;
```

```
wire SurvivorBit;
assign SurvivorBit=vd.tbu.SurvivorBit;
```

```
wire [`WD_RAM_DATA-1:0] DataRAM;
assign DataRAM=vd.DataRAM;
```

```
reg CLOCK;
initial CLOCK = 0;
always #(`HALF/2) CLOCK = ~CLOCK;
reg Reset;
reg DRESET;
initial begin
DRESET = 1;
Reset = 1;
#200 Reset = 0;DRESET=0;
#300 Reset = 1;
DRESET = 1;
end
reg X;
wire [`WD_CODE-1:0] Code;
initial X = 0;
initial begin
#475 X = 1;
#`DPERIOD X = 0;
#`DPERIOD X = 0;
#`DPERIOD X = 0;
#`DPERIOD X = 0;
#`DPERIOD X = 0;
#`DPERIOD X = 0;
#`DPERIOD X = 0;
#`DPERIOD X = 0;
#`DPERIOD X = 1;
#`DPERIOD X = 1;
#`DPERIOD X = 0;
#`DPERIOD X = 1;
#`DPERIOD X = 1;
```

```

#`DPERIOD X = 0;
#`DPERIOD X = 1;
#`DPERIOD X = 0;
#`DPERIOD X = 0;
#`DPERIOD X = 0;
#`DPERIOD X = 1;
end
reg D_CLOCK;
initial D_CLOCK = 0;

always #(`DPERIOD/2) D_CLOCK <= ~D_CLOCK;
viterbi_encode9 enc(D_CLOCK,DRESET,X,Code);
reg Active;
always @(Code or Reset)
if (~Reset) Active <= 0;
else if (Code!=0) Active <= 1;
wire DecodeOut;
VITERBIDECODER vd (Reset, CLOCK, Active, Code, DecodeOut);
initial begin
$monitor("X=%b DecodeOut=%b",X,DecodeOut);
end
endmodule

```


3.BMG.v (分支度量器模块)

```

/*****
`include "params.v"

/*****BMG 主模块*****/
module BMG (Reset,Clock2,ACSSegment,Code,Distance);
input Clock2;                                /*时钟*/
input Reset;                                /*低电平异步复位*/
input [`WD_FSM-1:0] ACSSegment;              /*记录 ACS 当前复用次数(0-63,位宽为 6):
寄存器有 8 位,故有 2^8=256 个状态;而 ACS 模块一次只能对 4 个状态进行路径度量,因此要对
ACS 复用 64 次,每次用到 4 个状态的 8 条输入路径的分支度量*/
input [`WD_CODE-1:0] Code;                  /*卷积编码输入,位宽为 2*/
output [`WD_DIST*2`N_ACS-1:0] Distance;      /*将 8 条分支度量进行拼接后输出,位宽共
8*2=16*/

reg [`WD_CODE-1:0] CodeRegister;             /*存储卷积编码输入(位宽为 2),完成对 ACS
的 64 次复用后更新数据*/

wire [`WD_STATE:0] PolyA,PolyB;              /*卷积编码的生成多项式*/
wire [`WD_STATE:0] B0,B1,B2,B3,B4,B5,B6,B7; /*分支标号*/
wire [`WD_CODE-1:0] G0,G1,G2,G3,G4,G5,G6,G7; /*每条分支对应的卷积编码理论输出,位宽
为 2*/
wire [`WD_DIST-1:0] D0,D1,D2,D3,D4,D5,D6,D7; /*每条分支的理论输出与当前实际卷积编码
输入的汉明距离*/

/*指定生成多项式*/
assign PolyA=9'b110_101_111;
assign PolyB=9'b100_011_101;

/*采样卷积编码输入*/
always@(posedge Clock2 or negedge Reset)
begin
    if (!Reset)
        CodeRegister<=2'b00;
    else if (ACSSegment==6'd63)
        CodeRegister<=Code;
    else
        CodeRegister<=CodeRegister;
end

/*对分支进行标号*/
assign B0 = {ACSSegment,3'b000};
assign B1 = {ACSSegment,3'b001};

```

```

assign B2 = {ACSSegment,3'b010};
assign B3 = {ACSSegment,3'b011};
assign B4 = {ACSSegment,3'b100};
assign B5 = {ACSSegment,3'b101};
assign B6 = {ACSSegment,3'b110};
assign B7 = {ACSSegment,3'b111};

/*计算每条分支的卷积编码理论输出*/

/*原理*/
/*1.分支标号由两部分组成,即"最高位的 0/1 输入+后八位的状态值"*/
/*2.对于偶数状态及其后一个奇数状态而言,二者在状态值上的区别在于,奇数状态最低位为
1,偶数状态最低位为 0,而其余七位则保持相同*/
/*3.当采样外部输入时,寄存器进行右移位,原有的最低位被移除;因此,当外部输入相同时,偶
数状态及其后一个奇数状态指向的下一状态相同*/
/*4.由生成多项式得,状态值的最低位始终会影响卷积输出,而偶数状态及其后一个奇数状态
的最低位必然相反,因此异或得到的结果也相反;
    所以只要计算某一输入下偶数状态的分支输出,即可通过取反得到同一输入时奇数状态
的分支输出,这两条分支指向的下一状态相同*/
/*5.ACSSegment 在 0_00000-0_11111 变化时,遍历了各状态在 0 输入下的输出分
支;ACSSegment 在 1_00000-1_11111 变化时,则遍历了各状态在 1 输入下的输出分支*/

ENC EN0(PolyA,PolyB,B0,G0); assign G1 = ~G0;
ENC EN2(PolyA,PolyB,B2,G2); assign G3 = ~G2;
ENC EN4(PolyA,PolyB,B4,G4); assign G5 = ~G4;
ENC EN6(PolyA,PolyB,B6,G6); assign G7 = ~G6;

/*进行分支度量,计算每条分支的汉明距离*/
HARD_DIST_CALC HD0(CodeRegister,G0,D0);
HARD_DIST_CALC HD1(CodeRegister,G1,D1);
HARD_DIST_CALC HD2(CodeRegister,G2,D2);
HARD_DIST_CALC HD3(CodeRegister,G3,D3);
HARD_DIST_CALC HD4(CodeRegister,G4,D4);
HARD_DIST_CALC HD5(CodeRegister,G5,D5);
HARD_DIST_CALC HD6(CodeRegister,G6,D6);
HARD_DIST_CALC HD7(CodeRegister,G7,D7);

/*将 8 个汉明距离拼接后作为距离总线输出*/
assign Distance = {D7,D6,D5,D4,D3,D2,D1,D0};

endmodule

/*****BMG 子模块,计算分支的卷积编码理论输出*****/
module ENC (PolyA,PolyB,BranchID,EncOut);

```

```

input [`WD_STATE:0] PolyA,PolyB;    /*生成多项式,位宽为 9*/
input [`WD_STATE:0] BranchID;      /*分支编号,位宽为 9*/
output reg [`WD_CODE-1:0] EncOut;   /*分支的卷积编码输出,位宽为 2*/

/*确定生成位*/
wire [`WD_STATE:0] wA,wB;
assign wA=PolyA&BranchID;
assign wB=PolyB&BranchID;

always@(wA or wB)
begin
    EncOut[1]=^wA;
    EncOut[0]=^wB;
end

endmodule

/**********BMG 子模块,计算分支理论输出与实际卷积编码的汉明距离*****/
module HARD_DIST_CALC (InputSymbol,BranchOutput,OutputDistance);
input [`WD_CODE-1:0] InputSymbol, BranchOutput; /*实际卷积编码、分支卷积编码,位宽为 2*/
output reg [`WD_DIST-1:0] OutputDistance;      /*汉明距离输出,位宽为 2*/

wire MS,LS;

assign MS=(InputSymbol[1]^BranchOutput[1]);    /*判断实际卷积编码与分支卷积编码的高
位是否相同*/
assign LS=(InputSymbol[0]^BranchOutput[0]);    /*判断实际卷积编码与分支卷积编码的低
位是否相同*/

/*真值表化简得到 InputSymbol、BranchOutput 和 OutputDistance 的逻辑关系*/
always@(MS or LS)
begin
    OutputDistance[1]=MS&LS;
    OutputDistance[0]=MS^LS;
end

endmodule

```

4.mmu.v（内存管理模块）

```
`include "params.v"

module MMU
(CLOCK,Clock1,Clock2,Reset,Active,Hold,Init,ACSPage,ACS_Segment_minus_LSB,Survivors,Data_TB,Address_TB,RW_Select,Read_Clock,Write_Clock,RAM_Enable,Address_RAM,Data_RAM);
// connection from Control 控制连接
input CLOCK; /*系统时钟*/
input Clock1,Clock2; /*分频时钟*/
input Reset; /*低电平异步复位*/
input Active; /*系统使能信号*/
input Hold,Init; /*单次 Code 解码的起始、终止信号*/
input [`WD_DEPTH-1:0] ACSPage; /*输入 Code 的累计长度,位宽为 6*/
input [`WD_FSM-2:0] ACS_Segment_minus_LSB; /*留存路径存储器的每个 ACSPage 子存储空间
的内部地址信号*/
input [`N_ACS-1:0] Survivors; /*从 ACS 单元输入的 4 条幸存路径,位宽为 4*/
input [`WD_RAM_ADDRESS-`WD_FSM-1:0] Address_TB; /**/
output [`WD_RAM_DATA-1:0] Data_TB;

// connection from/to RAM RAM 连接
output RW_Select, Read_Clock, Write_Clock,RAM_Enable;
output [`WD_RAM_ADDRESS-1:0] Address_RAM;
inout [`WD_RAM_DATA-1:0] Data_RAM;
wire [`WD_RAM_DATA-1:0] Written_Survivors;
reg dummy, Surv_RDY;
reg [`WD_RAM_ADDRESS-1:0] Address_RAM;
reg [`WD_DEPTH-1:0] TB_Page;
wire [`WD_DEPTH-1:0] TB_Page_;
wire [`WD_DEPTH-1:0] ACSPage;
wire [`WD_TB_ADDRESS-1:0] Address_TB;

/*产生留存路径存储器的读写控制信号*/
always@(posedge Clock2 or negedge Reset)
begin
    if(!Reset)
        dummy<=0;
    else if(Active)
        dummy<=!dummy;
    else
        dummy<=dummy;
end

assign Write_Clock = (Active && ~dummy)?Clock1:0;
```

```

assign ReadClock = (Active && ~Hold) ? !Clock1:0;

/*对 Clock1 进行二分频产生 SurvRDY 信号,用于控制 Survivors 缓冲输出*/
always@(posedge Clock1 or negedge Reset)
begin
    if(!Reset)
        SurvRDY<=1;
    else if (Active)
        SurvRDY<=!SurvRDY;
    else
        SurvRDY<=SurvRDY;
end

/*对幸存路径缓冲输出*/
ACSSURVIVORBUFFER buff (Reset,Clock1,Active,SurvRDY,Survivors,WrittenSurvivors);

/*TBPage 代表当前进行回溯的状态对应的是第几次 Code 输入*/
always@(negedge Clock2 or negedge Reset)
begin
    if(!Reset)
        TBPage<=0;
    else if(Init)
        TBPage<=ACSPage-1;
    else
        TBPage<=TBPage_;
end
assign TBPage_ =TBPage-1;

// For RAMs
assign RAMEnable = 0;
assign RWSelect = (Clock2) ? 1:0;
assign DataRAM = (!Clock2) ? WrittenSurvivors:'bz;
assign DataTB = (Clock2) ? DataRAM:'bz;

```

/*设定留存路径存储器的读/写地址*/

/*地址划分原理*/

/*1.留存路径存储器用于存储出现的所有状态的幸存路径(包括从前的各状态),因此,共需存储 63 次*256 条路径;

/*2.故对留存路径存储器的存储空间进行两层划分,首先将 ACSPage 作为 11 位地址总线的高 6 位,开辟 64 个子存储空间用于不同次 Code 输入下的计算,

每个子存储空间下又存在 64 个基本存储单元,用于存储当次 Code 下 256 个状态各自的

幸存路径;

/*3.数据总线为 8 位,故一次通信输入 8 个状态的幸存路径号,256 个状态需进行 32 次输入,即在每个 ACSPage 下要进行 32 次总线通信,且每隔两个时钟周期通信一次;

而 ACSegment 在每个 ACSPage 下都进行一轮 0-63 的变化,且每隔一个时钟周期变化一次,故 ACSegment 的高 5 位可实现在每个 ACSPage 下于 0-31 内每隔两个时钟周期变化一次,

故将 ACSegment 的高 5 位作为子存储空间内的地址,所以 11 位写地址为 {ACSPage,ACSSegment[`WD_FSM-1:1]}*/

```
always @(posedge CLOCK or negedge Reset)
```

```
begin
```

```
    if(!Reset)
```

```
        AddressRAM<=0;
```

```
    else if(Active)
```

```
        begin
```

```
            if(!Clock2)/*设定留存路径存储器的写地址:数据总线为 8 位,故一次通信输入 8 个状态的幸存路径号,256 个状态需进行 32 次输入;即在每个 ACSPage 下要进行 32 次总线通信,且,故取*/
```

```
                AddressRAM <= {ACSPage, ACSSegment_minusLSB};
```

```
            else/*设定留存路径存储器的读地址*/
```

```
                AddressRAM <= {TBPPage [`WD_DEPTH-1:0],AddressTB};
```

```
        end
```

```
    end
```

```
endmodule
```

```
/******幸存路径缓冲模块******/
```

```
/*由于 ACS 每次只输出 4 个状态的幸存路径,而留存路径存储器的数据总线为 8 位,故需将两次 ACS 的 Survivors 输出合并后进行一次总线通信*/
```

```
module ACSSURVIVORBUFFER (Reset,Clock1,Active,SurvRDY,Survivors,WrittenSurvivors);
```

```
input Clock1;                                /*时钟*/
```

```
input Reset;                                /*低电平异步复位*/
```

```
input Active;                                /*系统使能信号*/
```

```
input SurvRDY;                                /*Survivors 缓冲完成的标志信号(对 Clock1 进行二分频)*/
```

```
input [`N_ACS-1:0] Survivors;                /*ACS 每经一个 Clock1 周期输出一个 Survivors*/
```

```
output [`WD_RAM_DATA-1:0] WrittenSurvivors; /*输出合并后的两次 Survivors*/
```

```
reg [`N_ACS-1:0] WrittenSurvivors_;          /*寄存前一次 Survivors*/
```

```
/*对 Survivors 进行采样*/
```

```
always@(posedge Clock1 or negedge Reset)
```

```
begin
```

```
    if(!Reset)
```

```

        WrittenSurvivors_<=0;
    else if(Active)
        WrittenSurvivors_<=Survivors;
    else
        WrittenSurvivors_<=WrittenSurvivors_;
    end

    /*WrittenSurvivors_为 Clock1 的上升沿前的数据,由 Clock1 采样得到;Survivors 为 Clock1 的上
    升沿后的数据,直接通过连线输入*/
    assign WrittenSurvivors=(SurvRDY)?{Survivors,WrittenSurvivors_}:8'bz;

endmodule

```

5.tbuv (回溯单元模块)

```

`include "params.v"

module TBU (Reset,Clock1,Clock2,TB_EN,Init,Hold,InitState,DecodedData,DataTB,AddressTB);
input Reset,Clock1,Clock2,Init,Hold;
input [`WD_STATE-1:0] InitState;
input TB_EN;
input [`WD_RAM_DATA-1:0] DataTB;
output [`WD_RAM_ADDRESS-`WD_FSM-1:0] AddressTB;
output DecodedData;
wire [`WD_STATE-1:0] OutStateTB;

wire [`WD_STATE-1:0] CurrentState,NextState;
wire SurvivorBit;

    TRACEUNIT tb (Reset, Clock1, Clock2, TB_EN, InitState, Init, Hold,
        DataTB, AddressTB, OutStateTB,CurrentState,NextState,SurvivorBit);

assign DecodedData = OutStateTB [`WD_STATE-1];
endmodule

/*回溯模块*/
module                                     TRACEUNIT
(Reset,Clock1,Clock2,Enable,InitState,Init,Hold,Survivor,AddressTB,OutState,CurrentState,NextState,SurvivorBit);
input Clock1,Clock2;                                /*分频时钟*/
input Reset;                                         /*低电平异步复位*/
input Enable;                                       /*回溯使能信号 TB_EN*/
input Init,Hold;                                   /*单次 Code 解码的起始、终止信号*/
*/
input [`WD_STATE-1:0] InitState;                    /*输入 ACS 计算得到的幸存路径长度最短的状态*/
input [`WD_RAM_DATA-1:0] Survivor;
output reg [`WD_STATE-1:0] OutState;
output [`WD_RAM_ADDRESS-`WD_FSM-1:0] AddressTB;
output reg [`WD_STATE-1:0] CurrentState;
output reg [`WD_STATE-1:0] NextState;
output reg SurvivorBit;

always @(negedge Clock1 or negedge Reset)
begin
if (~Reset) begin
CurrentState <=0; OutState <=0;
end
else if (Enable)

```



```

begin
if (Init) CurrentState <= InitState;
else CurrentState <= NextState;
if (Hold) OutState <= NextState;
end
end

assign AddressTB = CurrentState [`WD_STATE-1:`WD_STATE-5];

always @(negedge Clock2 or negedge Reset)
begin
if (~Reset) NextState <= 0;
else
if (Enable) NextState <= {CurrentState [`WD_STATE-2:0],SurvivorBit};
end

always@(CurrentState or Clock1 or Clock2 or Init or Survivor)
begin
SurvivorBit<=(Clock1 && Clock2 && ~Init) ? Survivor [CurrentState [2:0]]:1'bz;
end

endmodule

```

6.ACS.v（加比选模块）

```

`include "params.v"
/*-----*/
// Module : ACSUNIT
// File: acs.v
// Description : Description of ACS Unit in Viterbi Decoder
// Simulator:
/*-----*/
// Revision Number: 1
// Description: Initial Design,该模块主要实现比选存的功能
/*-----*/

module ACSUNIT (
    input                Reset,           //低电平异步复位
    input                Clock1,          //操作时钟 Clock1
    input                Clock2,          //辅助时钟 Clock2
    input                Active,          //使能信号
    input                Init,            //解码开始 位于 ACSSegment 0
    input                Hold,            //解码完成 位于 ACSSegment 63
    input                CompareStart,    //比较使能, L-1 前不比较
    input  [`WD_FSM-1:0] ACSSegment,      //ACS 模块复用指示 256 状态 每
    次算出来四个 计数 64 6bit
    input  [`WD_DIST*2*`N_ACS-1:0] Distance, //BMG 输出的距离 8*2bit
    output [`N_ACS-1:0] Survivors,         //幸存路径标号 4*1bit
    output [`WD_STATE-1:0] LowestState,    //最短幸存路径状态 8bit
    output [`WD_FSM-2:0] MMReadAddress,    //路径度量存储器读地址 5bit
    output [`WD_FSM-1:0] MMWriteAddress,   //写地址 6bit
    output                MMBlockSelect,   //分块选择 路径度量存储器分
    两半 每次写一半读另一半
    output [`WD_METR*`N_ACS-1:0] MMMetric, //每次得到四个状态幸存路径长
    度 4*8bit
    input  [`WD_METR*2*`N_ACS-1:0] MMPathMetric //读出来的累计汉明距离
    8*8bit
);

//传输的时候都是 4 或八个状态拼接起来传的,用的时候拆分开,为了方便用位拼接的方式
//分成这么多个变量,用的时候不用再算下标范围
wire [`WD_DIST-1:0] D7,D6,D5,D4,D3,D2,D1,D0;
wire [`WD_METR*`N_ACS-1:0] Metric;
wire [`WD_METR-1:0] M0, M1, M2, M3;
wire [`WD_METR*2*`N_ACS-1:0] PathMetric;
wire [`WD_METR-1:0] PM7,PM6,PM5,PM4,PM3,PM2,PM1,PM0;
wire [`WD_METR-1:0] LowestMetric;
wire S3,S2,S1,S0;

```

```

//分割和拼接
assign {PM7,PM6,PM5,PM4,PM3,PM2,PM1,PM0}=PathMetric;
assign {D7,D6,D5,D4,D3,D2,D1,D0}=Distance;
assign Metric={M3,M2,M1,M0};
assign Survivors={S3,S2,S1,S0};

//四个 ACS 单元
ACS acs3(CompareStart,D7,D6,PM7,PM6,S3,M3);
ACS acs2(CompareStart,D5,D4,PM5,PM4,S2,M2);
ACS acs1(CompareStart,D3,D2,PM3,PM2,S1,M1);
ACS acs0(CompareStart,D1,D0,PM1,PM0,S0,M0);

LOWESTPICK lowestpick(
    .Reset(Reset),
    .Active(Active),
    .Hold(Hold),
    .Init(Init),
    .Clock1(Clock1),
    .Clock2(Clock2),
    .ACSSegment(ACSSegment),
    .M3(M3),
    .M2(M2),
    .M1(M1),
    .M0(M0),
    .LowestState(LowestState)
);
RAMINTERFACE raminterface(
    .Reset(Reset),
    .Clock2(Clock2),
    .Hold(Hold),
    .ACSSegment(ACSSegment),
    .Metric(Metric),
    .PathMetric(PathMetric),
    .MMReadAddress(MMReadAddress),
    .MMWriteAddress(MMWriteAddress),
    .MMBlockSelect(MMBlockSelect),
    .MMMetric(MMMetric),
    .MMPPathMetric(MMPPathMetric)
);
endmodule

/*-----*/
//RAM 接口模块

```

//一个 ACSegment 对应四个状态，ACSSegment 是状态号的高六位，低两位遍历得到四个状态号

//

/*-----*/

```
module RAMINTERFACE (
    //接口说明同主模块
    //连接到 ACS 单元
    input                                Clock2,
    input                                Reset,
    input                                Hold,
    input [`WD_FSM-1:0]                 ACSegment,
    input [`WD_METR*N_ACS-1:0]          Metric,
    output [`WD_METR*2*N_ACS-1:0]       PathMetric,
    //连接到度量内存
    output reg [`WD_FSM-2:0]             MMReadAddress,
    output [`WD_FSM-1:0]                 MMWriteAddress,
    output reg MMBlockSelect,
    output [`WD_METR*N_ACS-1:0]          MMMetric,
    input [`WD_METR*2*N_ACS-1:0]         MMPathMetric
);
```

```
always@(*)
begin
    if(!Reset)
        MMReadAddress=0;
    else
        MMReadAddress=ACSSegment[`WD_FSM-2:0];
end
```

//注意这里 Clock2 上升沿刷新路径度量存储器 block 选择，路径度量存储器的读出是组合逻辑

//所以 clock2 到 clock1 的相位差给了访问 ram 的延迟时间

//同时从 bmg 送来的 distance 数据是 clock2 上升沿送来的，在 clock2

always @(posedge Clock2 or negedge Reset)

```
begin
    if(!Reset)
        MMBlockSelect<=0;
    else if(Hold)
        MMBlockSelect<=!MMBlockSelect;
    else
        MMBlockSelect<=MMBlockSelect;
end
```

assign MMWriteAddress=ACSSegment;

```
assign PathMetric=MMPathMetric;
assign MMMetric=Metric;
```

```
endmodule
```

```
/*-----*/
```

```
/*-----*/
```

```
module ACS (
    input CompareEnable,
    input [`WD_DIST-1:0] D1,
    input [`WD_DIST-1:0] D0,
    input [`WD_METR-1:0] PM1,
    input [`WD_METR-1:0] PM0,
    output Survivor,
    output [`WD_METR-1:0] Metric
);
wire [`WD_METR-1:0] NewMetric0, NewMetric1;
wire [`WD_METR-1:0] Temp_Metric;

assign NewMetric0=PM0+{6'b0,D0};
assign NewMetric1=PM1+{6'b0,D1};
COMPARATOR comp_acs(CompareEnable,NewMetric1,NewMetric0,Survivor);
assign Metric=Survivor?NewMetric1:NewMetric0;
endmodule
```

```
/*-----*/
```

```
/*-----*/
```

```
module COMPARATOR (
    input CompareEnable,
    input [`WD_METR-1:0] M1,
    input [`WD_METR-1:0] M0,
    output reg Survivor
);
always@(*)
begin
    if(CompareEnable)
        Survivor=(M1<M0)?1:0;
    else
        Survivor=0;
end
endmodule
```

```

/*-----*/
// This module is used to find which of 256 states has the smallest metric.
//该模块用于查找 256 个状态中具有最小度量的状态。
// The value will be very useful for :该值对以下内容非常有用:
// - determine the first point of traceback 确定追溯的第一点
// - debugging your ACS Unit (Should no error on received data occurred,
// you'll find the state with the smallest metric is exactly based on
// the encoder input (X), and the lowest metric value should be 0)
// 调试 ACS 单元(如果接收数据没有错误,你将发现具有最小度量的状态完全基于编
//码器输入(X),并且最低度量值应为 0)
/*-----*/
module LOWESTPICK (
    input Reset,
    input Active,
    input Hold,
    input Init,
    input Clock1,
    input Clock2,
    input [`WD_FSM-1:0] ACSegment,
    input [`WD_METR-1:0]M3,
    input [`WD_METR-1:0]M2,
    input [`WD_METR-1:0]M1,
    input [`WD_METR-1:0]M0,
    output reg [`WD_STATE-1:0]LowestState
);

reg [`WD_METR-1:0] LowestMetric;
reg [`WD_METR-1:0] Reg_Metric;
reg [`WD_STATE-1:0] Reg_State;
wire [`WD_METR-1:0] MetricCompareResult;
wire [`WD_STATE-1:0] StateCompareResult;
wire [`WD_METR-1:0] Lowest_Metric4;
wire [`WD_STATE-1:0] Lowest_State4;
wire compare;

LOWEST_OF_FOUR lof(
    .Active(Active),
    .ACSSegment(ACSSegment),
    .M3(M3),
    .M2(M2),
    .M1(M1),
    .M0(M0),
    .Lowest_State4(Lowest_State4),

```

```

        .Lowest_Metric4(Lowest_Metric4)
    );
    COMPARATOR comp_iter(Active,Lowest_Metric4,Reg_Metric,compare);
    assign MetricCompareResult=compare?Lowest_Metric4:Reg_Metric;
    assign StateCompareResult=compare?Lowest_State4:Reg_State;
    //clock2 的下降沿处锁存
    always@(negedge Reset or negedge Clock2)
    begin
        if(!Reset)
        begin
            Reg_Metric<=`WD_METR'b0;
            Reg_State<=`WD_STATE'b0;
        end
        else if(Active)
        begin
            if(Init)
            begin
                Reg_Metric<=Lowest_Metric4;
                Reg_State<=Lowest_State4;
            end
            else
            begin
                Reg_Metric<=MetricCompareResult;
                Reg_State<=StateCompareResult;
            end
        end
    end

    wire Hold_Output_Flag;
    assign Hold_Output_Flag=Active && Hold;

    always@(negedge Reset or negedge Clock1)
    begin
        if(!Reset)
        begin
            LowestMetric<=`WD_METR'b0;
            LowestState<=`WD_STATE'b0;
        end
        else if(Hold_Output_Flag)
        begin
            LowestMetric<=Reg_Metric;
            LowestState<=Reg_State;
        end
    end
end

```

```
endmodule
```

```
/*-----*/
// This module is used to find ONE STATE among FOUR survivor and metric
// calculated in every cycle which has the smallest metric.
// 该模块用于在四个留存中找到一个状态,其在每个周期中计算出具有最小度量。
/*-----*/
module LOWEST_OF_FOUR (
    input Active,
    input [`WD_FSM-1:0]ACSSegment,
    input [`WD_METR-1:0]M3,
    input [`WD_METR-1:0]M2,
    input [`WD_METR-1:0]M1,
    input [`WD_METR-1:0]M0,
    output [`WD_STATE-1:0]Lowest_State4,
    output [`WD_METR-1:0]Lowest_Metric4
);
wire Surv1, Surv2, Surv3, Bit_One;
wire [`WD_METR-1:0] MetricX, MetricY;

//两两比较比了两轮，得到 Surv3
COMPARATOR comp1(Active,M1,M0,Surv1);
COMPARATOR comp2(Active,M3,M2,Surv2);
assign MetricX = Surv1 ? M1 : M0;
assign MetricY = Surv2 ? M3 : M2;
COMPARATOR comp3(Active,MetricY,MetricX,Surv3);

assign Bit_One=Surv3?Surv2:Surv1;//找出 Surv3 对应的标号
//拼接得到状态编号，关于 Surv3，Bit_One 的拼接，
//例如 Metric3 被选中时两次比较都选择了 1 而非 0 分支，这样可以通过拼接简单实现是
//和比较顺序密切相关的
assign Lowest_State4={ACSSegment,Surv3,Bit_One};
assign Lowest_Metric4=Surv3?MetricY:MetricX;//找出对应的 PM 总汉明距离
endmodule
```

7.CONTROL.v（控制模块）


```

`include "params.v"
module CONTROL(
    input Clock_sys,
    input Reset,
    input Active,
    output reg [`WD_FSM-1:0] ACSegment,
    output reg [`WD_DEPTH-1:0] ACSPage,
    output reg Clock1,
    output reg Clock2,
    output reg Hold,
    output reg Init,
    output reg TB_EN,
    output reg CompareStart
);

    //时钟生成
    reg count;
    //四分频 Clock 得到 Clock1 和 Clock2，Clock2 相位超前 90 度
    always@(posedge Clock_sys or negedge Reset)
    begin
        if(!Reset)
            count<=1'b0;
        else
            count<=!count;
    end
    always@(posedge Clock_sys or negedge Reset)
    begin
        if(!Reset)
        begin
            Clock1<=1'b0;
            Clock2<=1'b0;
        end
        else
        begin
            if(count)
                Clock1<=!Clock1;
            if(!count)
                Clock2<=!Clock2;
        end
    end

    //acs 复用计数和标志位控制
    wire hold_flag,init_flag,traceback_flag;

```

```

assign hold_flag=(ACSSegment=='d62')?1:0;
assign init_flag=(ACSSegment=='d63')?1:0;
assign traceback_flag=(ACSSegment=='d63' && ACSPage=='d62')?1:0;
always@(posedge Clock1 or negedge Reset)
begin
    if(!Reset)
    begin
        ACSPage<=`WD_DEPTH'b111111;
        ACSSegment<=`WD_FSM'b111111;
        Init<=0;
        Hold<=0;
        TB_EN<=0;
    end
    else if (Active)
    begin
        {ACSPage,ACSSegment}<={ACSPage,ACSSegment}+1;
        if(hold_flag)Hold<=1;
        else Hold<=0;
        if(init_flag)Init<=1;
        else Init<=0;
        if(traceback_flag)TB_EN<=1;
    end
end

```

//comparestart 信号逻辑

//在移位寄存器最后一 bit 被填满之前，每个状态都只来自于末尾为 0 的状态

//路径唯一，不用比较

//移位寄存器位数为关联长度-1

```
reg [3:0] waiter;
```

//每次

```
always@(posedge Clock2 or negedge Reset)
```

```

begin
    if(!Reset)
    begin
        waiter<=`CONSTRAINT-1;
        CompareStart<=0;
    end
    else
    begin
        if(hold_flag)
            waiter<=waiter-1;
        if(waiter=='b0 && init_flag)
            CompareStart<=1;
    end
end

```

end

endmodule

8.metricmemory.v

```

`include "params.v"
module METRICMEMORY(
    input Reset,
    input Clock1,
    input Active,
    input MMBlockSelect,
    input [`WD_METR*N_ACS-1:0] MMMetric,
    input [`WD_FSM-1:0] MMWriteAddress,
    input [`WD_FSM-2:0] MMReadAddress,
    output reg [`WD_METR*2*N_ACS-1:0] MMPathMetric
);
    reg [`WD_METR*N_ACS-1:0] M_REG_A [`N_ITER-1:0];    /*A 存储空间,共 64 个存储单元,每个存储单元有 32 位*/
    reg [`WD_METR*N_ACS-1:0] M_REG_B [`N_ITER-1:0];    /*B 存储空间,共 64 个存储单元,每个存储单元有 32 位*/

    always@(negedge Clock1 or negedge Reset)
    begin
        if(~Reset)/*初始化为全零*/
        begin
            M_REG_A [63] <= 0;M_REG_A [62] <= 0;M_REG_A [61] <= 0;M_REG_A [60] <= 0;
            M_REG_A [59] <= 0;M_REG_A [58] <= 0;M_REG_A [57] <= 0;M_REG_A [56] <= 0;
            M_REG_A [55] <= 0;M_REG_A [54] <= 0;M_REG_A [53] <= 0;M_REG_A [52] <= 0;
            M_REG_A [51] <= 0;M_REG_A [50] <= 0;M_REG_A [49] <= 0;M_REG_A [48] <= 0;
            M_REG_A [47] <= 0;M_REG_A [46] <= 0;M_REG_A [45] <= 0;M_REG_A [44] <= 0;
            M_REG_A [43] <= 0;M_REG_A [42] <= 0;M_REG_A [41] <= 0;M_REG_A [40] <= 0;
            M_REG_A [39] <= 0;M_REG_A [38] <= 'd1;M_REG_A [37] <= 0;M_REG_A [36] <= 0;
            M_REG_A [35] <= 0;M_REG_A [34] <= 0;M_REG_A [33] <= 0;M_REG_A [32] <= 0;
            M_REG_A [31] <= 0;M_REG_A [30] <= 0;M_REG_A [29] <= 0;M_REG_A [28] <= 0;
            M_REG_A [27] <= 0;M_REG_A [26] <= 0;M_REG_A [25] <= 0;M_REG_A [24] <= 0;
            M_REG_A [23] <= 0;M_REG_A [22] <= 0;M_REG_A [21] <= 0;M_REG_A [20] <= 0;
            M_REG_A [19] <= 0;M_REG_A [18] <= 0;M_REG_A [17] <= 0;M_REG_A [16] <= 0;
            M_REG_A [15] <= 0;M_REG_A [14] <= 0;M_REG_A [13] <= 0;M_REG_A [12] <= 0;
            M_REG_A [11] <= 0;M_REG_A [10] <= 0;M_REG_A [9] <= 0;M_REG_A [8] <= 0;
            M_REG_A [7] <= 0;M_REG_A [6] <= 0;M_REG_A [5] <= 0;M_REG_A [4] <= 0;
            M_REG_A [3] <= 0;M_REG_A [2] <= 0;M_REG_A [1] <= 0;M_REG_A [0] <= 0;

            M_REG_B [63] <= 0;M_REG_B [62] <= 0;M_REG_B [61] <= 0;M_REG_B [60] <= 0;
            M_REG_B [59] <= 0;M_REG_B [58] <= 0;M_REG_B [57] <= 0;M_REG_B [56] <= 0;
            M_REG_B [55] <= 0;M_REG_B [54] <= 0;M_REG_B [53] <= 0;M_REG_B [52] <= 0;
            M_REG_B [51] <= 0;M_REG_B [50] <= 0;M_REG_B [49] <= 0;M_REG_B [48] <= 0;
            M_REG_B [47] <= 0;M_REG_B [46] <= 0;M_REG_B [45] <= 0;M_REG_B [44] <= 0;
            M_REG_B [43] <= 0;M_REG_B [42] <= 0;M_REG_B [41] <= 0;M_REG_B [40] <= 0;

```

```

M_REG_B [39] <= 0;M_REG_B [38] <= 'd1;M_REG_B [37] <= 0;M_REG_B [36] <= 0;
M_REG_B [35] <= 0;M_REG_B [34] <= 0;M_REG_B [33] <= 0;M_REG_B [32] <= 0;
M_REG_B [31] <= 0;M_REG_B [30] <= 0;M_REG_B [29] <= 0;M_REG_B [28] <= 0;
M_REG_B [27] <= 0;M_REG_B [26] <= 0;M_REG_B [25] <= 0;M_REG_B [24] <= 0;
M_REG_B [23] <= 0;M_REG_B [22] <= 0;M_REG_B [21] <= 0;M_REG_B [20] <= 0;
M_REG_B [19] <= 0;M_REG_B [18] <= 0;M_REG_B [17] <= 0;M_REG_B [16] <= 0;
M_REG_B [15] <= 0;M_REG_B [14] <= 0;M_REG_B [13] <= 0;M_REG_B [12] <= 0;
M_REG_B [11] <= 0;M_REG_B [10] <= 0;M_REG_B [9] <= 0;M_REG_B [8] <= 0;
M_REG_B [7] <= 0;M_REG_B [6] <= 0;M_REG_B [5] <= 0;M_REG_B [4] <= 0;
M_REG_B [3] <= 0;M_REG_B [2] <= 0;M_REG_B [1] <= 0;M_REG_B [0] <= 0;
end
else
begin
if(Active)/*系统处于使能状态*/
begin
case(MMBlockSelect)
0 : begin M_REG_A [MMWriteAddress]<=MMMetric ; M_REG_B
[MMWriteAddress]<=M_REG_B [MMWriteAddress]; end /*对 A 空间写入*/
1 : begin M_REG_B [MMWriteAddress]<=MMMetric ; M_REG_A
[MMWriteAddress]<=M_REG_A [MMWriteAddress]; end /*对 B 空间写入*/
endcase
end
else
begin
M_REG_A [MMWriteAddress]<=M_REG_A [MMWriteAddress];
M_REG_B [MMWriteAddress]<=M_REG_B [MMWriteAddress];
end
end
end

/*对存储空间读取*/
always @(MMReadAddress or Reset)
begin
if(!Reset)
MMPathMetric=0;
else
begin
case(MMBlockSelect)
0 : case(MMReadAddress)/*对 B 空间进行读取:一个存储单元内存放 4 个状态的路径度
量值(32bit),而每次需取出 8 个状态的路径度量值,故每次取出两个存储单元*/
0 : MMPathMetric = {M_REG_B [1],M_REG_B[0]};
1 : MMPathMetric = {M_REG_B [3],M_REG_B[2]};
2 : MMPathMetric = {M_REG_B [5],M_REG_B[4]};
3 : MMPathMetric = {M_REG_B [7],M_REG_B[6]};

```

```

4 : MMPathMetric = {M_REG_B [9],M_REG_B[8]};
5 : MMPathMetric = {M_REG_B [11],M_REG_B[10]};
6 : MMPathMetric = {M_REG_B [13],M_REG_B[12]};
7 : MMPathMetric = {M_REG_B [15],M_REG_B[14]};

8 : MMPathMetric = {M_REG_B [17],M_REG_B[16]};
9 : MMPathMetric = {M_REG_B [19],M_REG_B[18]};
10 : MMPathMetric = {M_REG_B [21],M_REG_B[20]};
11 : MMPathMetric = {M_REG_B [23],M_REG_B[22]};
12 : MMPathMetric = {M_REG_B [25],M_REG_B[24]};
13 : MMPathMetric = {M_REG_B [27],M_REG_B[26]};
14 : MMPathMetric = {M_REG_B [29],M_REG_B[28]};
15 : MMPathMetric = {M_REG_B [31],M_REG_B[30]};

16 : MMPathMetric = {M_REG_B [33],M_REG_B[32]};
17 : MMPathMetric = {M_REG_B [35],M_REG_B[34]};
18 : MMPathMetric = {M_REG_B [37],M_REG_B[36]};
19 : MMPathMetric = {M_REG_B [39],M_REG_B[38]};
20 : MMPathMetric = {M_REG_B [41],M_REG_B[40]};
21 : MMPathMetric = {M_REG_B [43],M_REG_B[42]};
22 : MMPathMetric = {M_REG_B [45],M_REG_B[44]};
23 : MMPathMetric = {M_REG_B [47],M_REG_B[46]};

24 : MMPathMetric = {M_REG_B [49],M_REG_B[48]};
25 : MMPathMetric = {M_REG_B [51],M_REG_B[50]};
26 : MMPathMetric = {M_REG_B [53],M_REG_B[52]};
27 : MMPathMetric = {M_REG_B [55],M_REG_B[54]};
28 : MMPathMetric = {M_REG_B [57],M_REG_B[56]};
29 : MMPathMetric = {M_REG_B [59],M_REG_B[58]};
30 : MMPathMetric = {M_REG_B [61],M_REG_B[60]};
31 : MMPathMetric = {M_REG_B [63],M_REG_B[62]};
endcase

```

1 : case(MMReadAddress)/*对 A 空间进行读取*/

```

0 : MMPathMetric = {M_REG_A [1],M_REG_A[0]};
1 : MMPathMetric = {M_REG_A [3],M_REG_A[2]};
2 : MMPathMetric = {M_REG_A [5],M_REG_A[4]};
3 : MMPathMetric = {M_REG_A [7],M_REG_A[6]};
4 : MMPathMetric = {M_REG_A [9],M_REG_A[8]};
5 : MMPathMetric = {M_REG_A [11],M_REG_A[10]};
6 : MMPathMetric = {M_REG_A [13],M_REG_A[12]};
7 : MMPathMetric = {M_REG_A [15],M_REG_A[14]};

8 : MMPathMetric = {M_REG_A [17],M_REG_A[16]};
9 : MMPathMetric = {M_REG_A [19],M_REG_A[18]};

```

```

10 : MMPathMetric = {M_REG_A [21],M_REG_A[20]};
11 : MMPathMetric = {M_REG_A [23],M_REG_A[22]};
12 : MMPathMetric = {M_REG_A [25],M_REG_A[24]};
13 : MMPathMetric = {M_REG_A [27],M_REG_A[26]};
14 : MMPathMetric = {M_REG_A [29],M_REG_A[28]};
15 : MMPathMetric = {M_REG_A [31],M_REG_A[30]};

16 : MMPathMetric = {M_REG_A [33],M_REG_A[32]};
17 : MMPathMetric = {M_REG_A [35],M_REG_A[34]};
18 : MMPathMetric = {M_REG_A [37],M_REG_A[36]};
19 : MMPathMetric = {M_REG_A [39],M_REG_A[38]};
20 : MMPathMetric = {M_REG_A [41],M_REG_A[40]};
21 : MMPathMetric = {M_REG_A [43],M_REG_A[42]};
22 : MMPathMetric = {M_REG_A [45],M_REG_A[44]};
23 : MMPathMetric = {M_REG_A [47],M_REG_A[46]};

24 : MMPathMetric = {M_REG_A [49],M_REG_A[48]};
25 : MMPathMetric = {M_REG_A [51],M_REG_A[50]};
26 : MMPathMetric = {M_REG_A [53],M_REG_A[52]};
27 : MMPathMetric = {M_REG_A [55],M_REG_A[54]};
28 : MMPathMetric = {M_REG_A [57],M_REG_A[56]};
29 : MMPathMetric = {M_REG_A [59],M_REG_A[58]};
30 : MMPathMetric = {M_REG_A [61],M_REG_A[60]};
31 : MMPathMetric = {M_REG_A [63],M_REG_A[62]};
    endcase
  end
end
endmodule

```

9.. viterbi_encode9 v卷积编码模块

```

module viterbi_encode9 (Clock,Reset,X,Y);
input Clock;          /*系统时钟*/
input Reset;          /*低电平异步复位*/
input X;              /*输入*/
output reg [1:0] Y;   /*卷积编码输出*/

reg [7:0] X_reg;
wire [8:0] data_H,data_L;
wire [1:0] Y_reg;

/*设定生成多项式*/
wire [8:0] Poly_H,Poly_L;
assign Poly_H=9'b110_101_111;
assign Poly_L=9'b100_011_101;

/*进行移位寄存用于生成输出*/
always@(posedge Clock or negedge Reset)
begin
    if(!Reset)
        X_reg<=8'b0000_0000;
    else
        begin
            X_reg[7]<=X;
            X_reg[6]<=X_reg[7];
            X_reg[5]<=X_reg[6];
            X_reg[4]<=X_reg[5];
            X_reg[3]<=X_reg[4];
            X_reg[2]<=X_reg[3];
            X_reg[1]<=X_reg[2];
            X_reg[0]<=X_reg[1];
        end
    end

/*确定生成位(与生成多项式进行与运算后,仅保留生成位,其他位归零,不影响模二运算的结果)*/
assign data_H={X,X_reg}&Poly_H;
assign data_L={X,X_reg}&Poly_L;

/*生成输出(纯组合逻辑输出容易产生毛刺,加一级寄存器作为缓冲)*/
assign Y_reg[1]^=data_H;
assign Y_reg[0]^=data_L;

always@(posedge Clock or negedge Reset)

```



```
begin
  if(!Reset)
    Y<=2'b00;
  else
    Y<=Y_reg;
end

endmodule
```

10.. VITERBIDECODER v卷积编码模块

```

`include "params.v"

module VITERBIDECODER (Reset, CLOCK, Active, Code, DecodeOut);

input Reset, CLOCK, Active;
input [`WD_CODE-1:0] Code;
output DecodeOut;
wire [`WD_DIST*2*`N_ACS-1:0] Distance; // BMG Output
wire [`WD_FSM-1:0] ACSSegment; //
wire [`WD_DEPTH-1:0] ACSPage; // Control Output
wire CompareStart, Hold, Init; //
wire [`N_ACS-1:0] Survivors; // ACS Output
wire [`WD_STATE-1:0] LowestState;
wire TB_EN;
wire RAMEnable;
wire ReadClock, WriteClock, RWSelect;
wire [`WD_RAM_ADDRESS-1:0] AddressRAM;
// generated by TBU and ACSU
wire [`WD_RAM_DATA-1:0] DataRAM; // RAM Databus , RAM 数据总线
wire [`WD_RAM_DATA-1:0] DataTB;
wire [`WD_RAM_ADDRESS-`WD_FSM-1:0] AddressTB;
wire Clock1, Clock2;
// for metric memory connection 度量内存连接
wire [`WD_METR*2*`N_ACS-1:0] MMPathMetric;
wire [`WD_METR*`N_ACS-1:0] MMMetric;
wire [`WD_FSM-2:0] MMReadAddress;
wire [`WD_FSM-1:0] MMWriteAddress;
wire MMBlockSelect;
// instantiation of Viterbi Decoder Modules 维特比译码器模块的实例化

CONTROL control(
    .Clock_sys(CLOCK),
    .Clock1(Clock1),
    .Clock2(Clock2),
    .Reset(Reset),
    .Active(Active),
    .ACSSegment(ACSSegment),
    .ACSPage(ACSPage),
    .CompareStart(CompareStart),
    .TB_EN(TB_EN),
    .Hold(Hold),
    .Init(Init)
);

```

```
BMG bmg (Reset, Clock2, ACSegment, Code, Distance);
```

```
ACSUNIT acsunit(
```

```
    .Reset(Reset),  
    .Clock1(Clock1),  
    .Clock2(Clock2),  
    .Active(Active),  
    .Init(Init),  
    .Hold(Hold),  
    .CompareStart(CompareStart),  
    .ACSSegment(ACSSegment),  
    .Distance(Distance),  
    .Survivors(Survivors),  
    .LowestState(LowestState),  
    .MMReadAddress(MMReadAddress),  
    .MMWriteAddress(MMWriteAddress),  
    .MMBlockSelect(MMBlockSelect),  
    .MMMetric(MMMetric),  
    .MMPathMetric(MMPathMetric)
```

```
);
```

```
MMU mmu (CLOCK, Clock1, Clock2, Reset, Active, Hold, Init, ACSPage,  
ACSSegment [ `WD_FSM-1:1], Survivors,  
DataTB, AddressTB,  
RWSelect, ReadClock, WriteClock,  
RAMEnable, AddressRAM, DataRAM);  
TBU tbu (Reset, Clock1, Clock2, TB_EN, Init, Hold, LowestState,  
DecodeOut, DataTB, AddressTB);
```

```
METRICMEMORY mm(
```

```
    .Reset(Reset),  
    .Clock1(Clock1),  
    .Active(Active),  
    .MMBlockSelect(MMBlockSelect),  
    .MMMetric(MMMetric),  
    .MMWriteAddress(MMWriteAddress),  
    .MMReadAddress(MMReadAddress),  
    .MMPathMetric(MMPathMetric)
```

```
);
```

```
RAM ram (RAMEnable, AddressRAM, DataRAM, RWSelect, ReadClock, WriteClock);
```

```
// --
```

```
endmodule
```