

EE533 NETWORK PROCESSOR DESIGN & PROGRAMMING
LAB#3: (Mini-Intrusion Detection Engine Design)

Instructor: Prof. Young Cho, PhD

CREATED AND COMPILED BY:
SARTHAK JAIN

(MS EE, UNIVERSITY OF SOUTHERN CALIFORNIA)

GITHUB LINK FOR MY REPOSITORY:

You'll find all the codes and the executables on this repository.

<https://github.com/SARTHAK-JAIN-ASIC/EE533/tree/main/LAB3>

QUESTIONS:

1. Look at the created Verilog. Do they make sense? Which do you think easier: entering the schematics or writing Verilog? Why? In which cases might you do the other?

A1:

Generated code is gibberish to us as it is generated by the tool using very specific instance names, wire names, module names etc. While the simulation would be perfectly fine, making heads or tails of the generated code is a challenge.

It would personally be way easier to write rtl logic for a huge circuit like this as code logic can be reused easily. Behavioral model of HDL utilizes much less effort to design logic than it would take to create a schematic. The debugging process in case something goes wrong is also easier.

2. What does the testbench do?

A2:

For a mini intrusion system, we need an input data stream, a pattern matcher logic that detects certain intrusion patterns in the input data stream and an output FIFO wrapper to drop the intrusive pattern matched packets and write the rest of the data into a dual port FIFO (which essentially decouples the packets (intrusion free) from the 'to be processed circuit')

The test bench has 4 components for this:

1. fallthrough fifo

Essentially this is the fifo we use to send in the packets to our pattern detection logic (again, this is a decoupler that decouples our input data stream from the pattern matching logic)

2. detect7B matcher

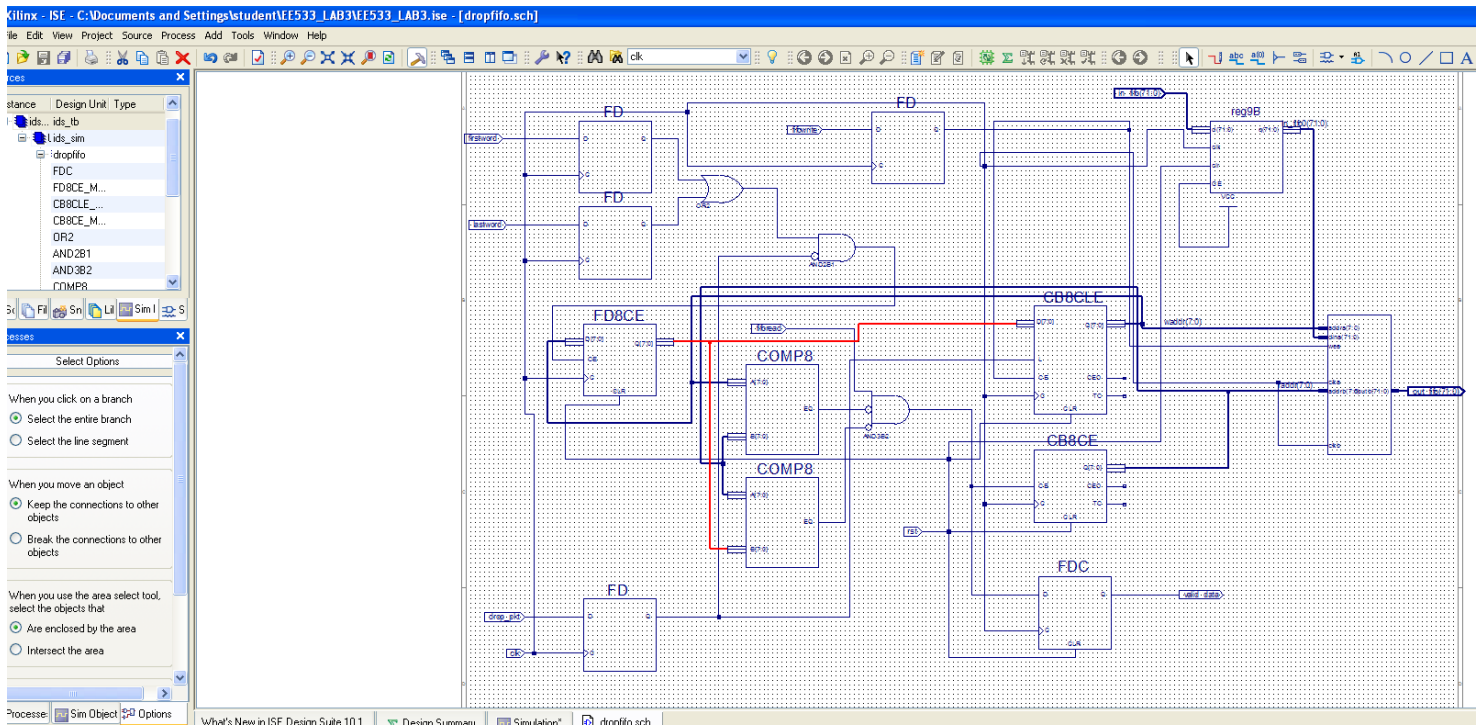
This is the logic that detects the "intrusive data bit pattern" with the input data stream from the fallthrough fifo.

3. dropfifo

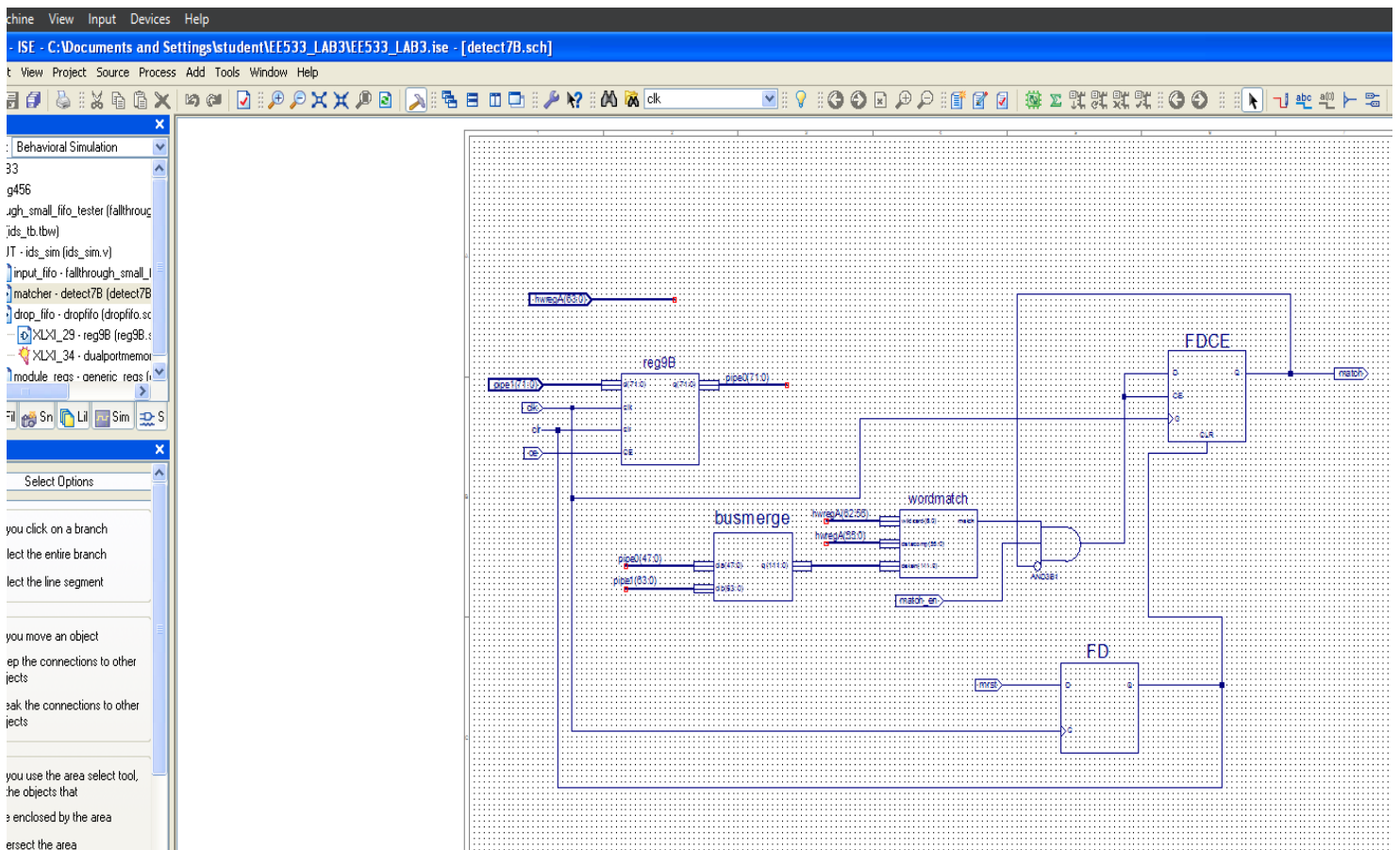
This is the fifo wrapper that has the logic to drop "pattern matched" intrusive packets from the data stream

4. Packet separation logic

This state machine basically separates the packet into header and the payload which then allows us to test each byte of the input packet bitwise with the intrusion pattern.



Dropfifo



Detect7B

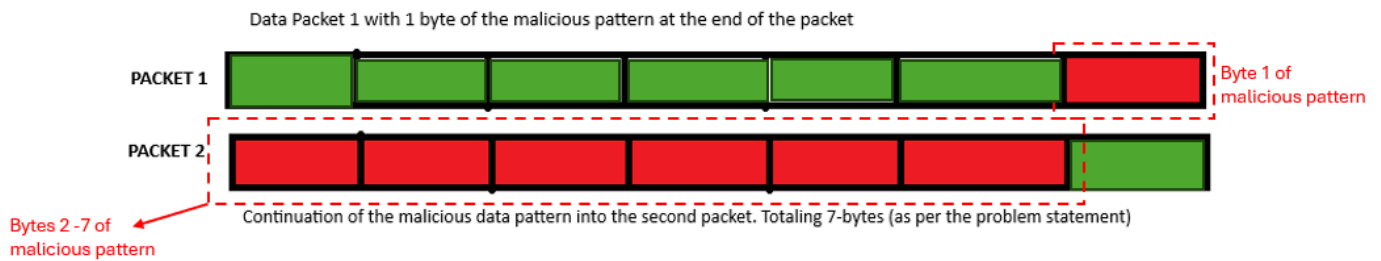
WAVEFORM AND GENERATED OUTPUT SCREENSHOT:



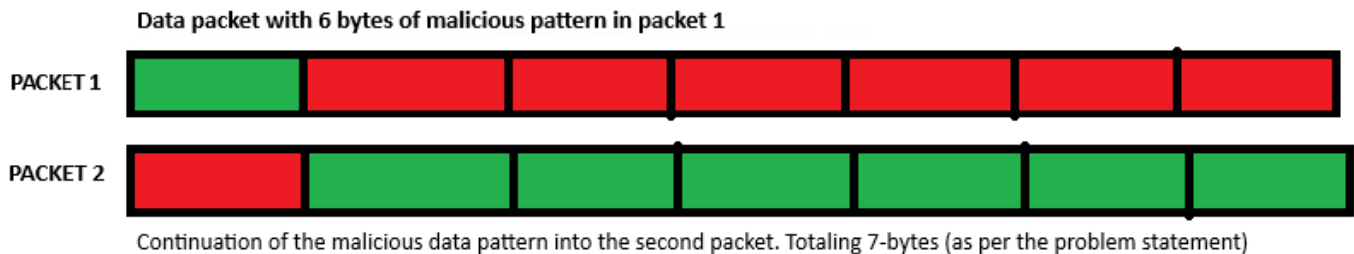
a)

i) The idea of this intrusion detection system is to detect a 7-byte malicious intrusive pattern in the incoming network data stream. The input stream is 8-byte data packet + 1 Byte control signal. Totaling 9-byte data. This data goes into the fallthrough fifo and then into the pattern matcher 7B module. Now, we need to consider the possibility that the malicious pattern is hidden in the packets a variety of ways.

To consider all corner cases we'll take this example:



Apart from this we certainly can have other combinations as well (such as 3 bytes in packet 1 and 4 bytes in packet 2) but the above should cover the worst-case scenario. Or it could be the other way around as well.



For a mini-intrusion system, we need an input data stream, a pattern matcher logic that detects certain intrusion patterns in the input data stream and an output FIFO wrapper to drop the intrusive pattern matched packets and write the rest of the data into a dual port FIFO (which essentially decouples the packets (intrusion free) from the 'to be processed circuit')

The test bench has 4 components for this:

1. fallthrough fifo

Essentially this is the fifo we use to send in the packets to our pattern detection logic (again, this is a decoupler that decouples our input data stream from the pattern matching logic)

2. detect7B matcher

This is the logic that detects the "intrusive data bit pattern" with the input data stream from the fallthrough fifo.

3. dropfifo

This is the fifo wrapper that has the logic to drop “pattern matched” intrusive packets from the data stream

4. Packet separation logic

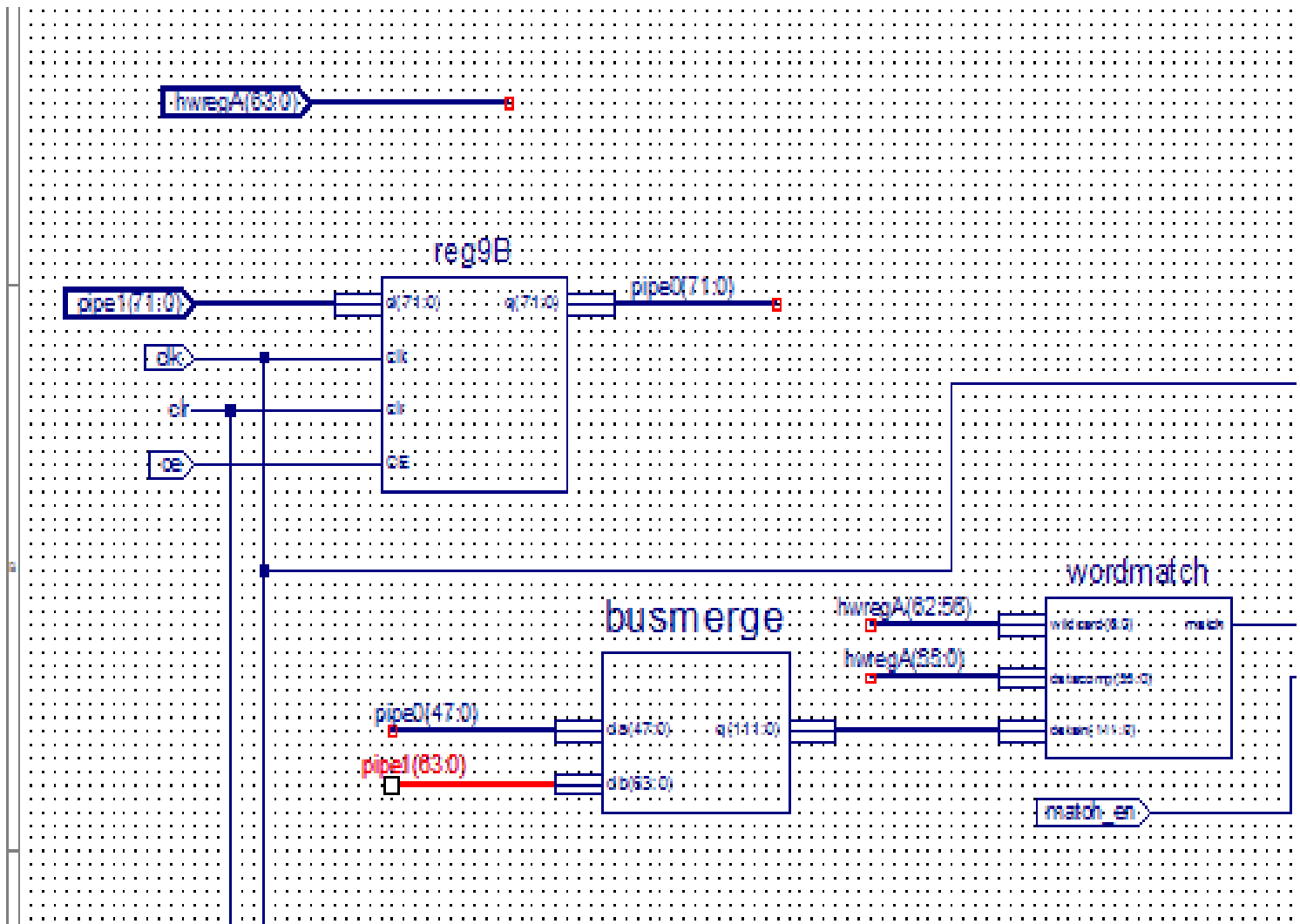
This state machine basically separates the packet into header and the payload which then allows us to test each byte of the input packet bitwise with the intrusion pattern.

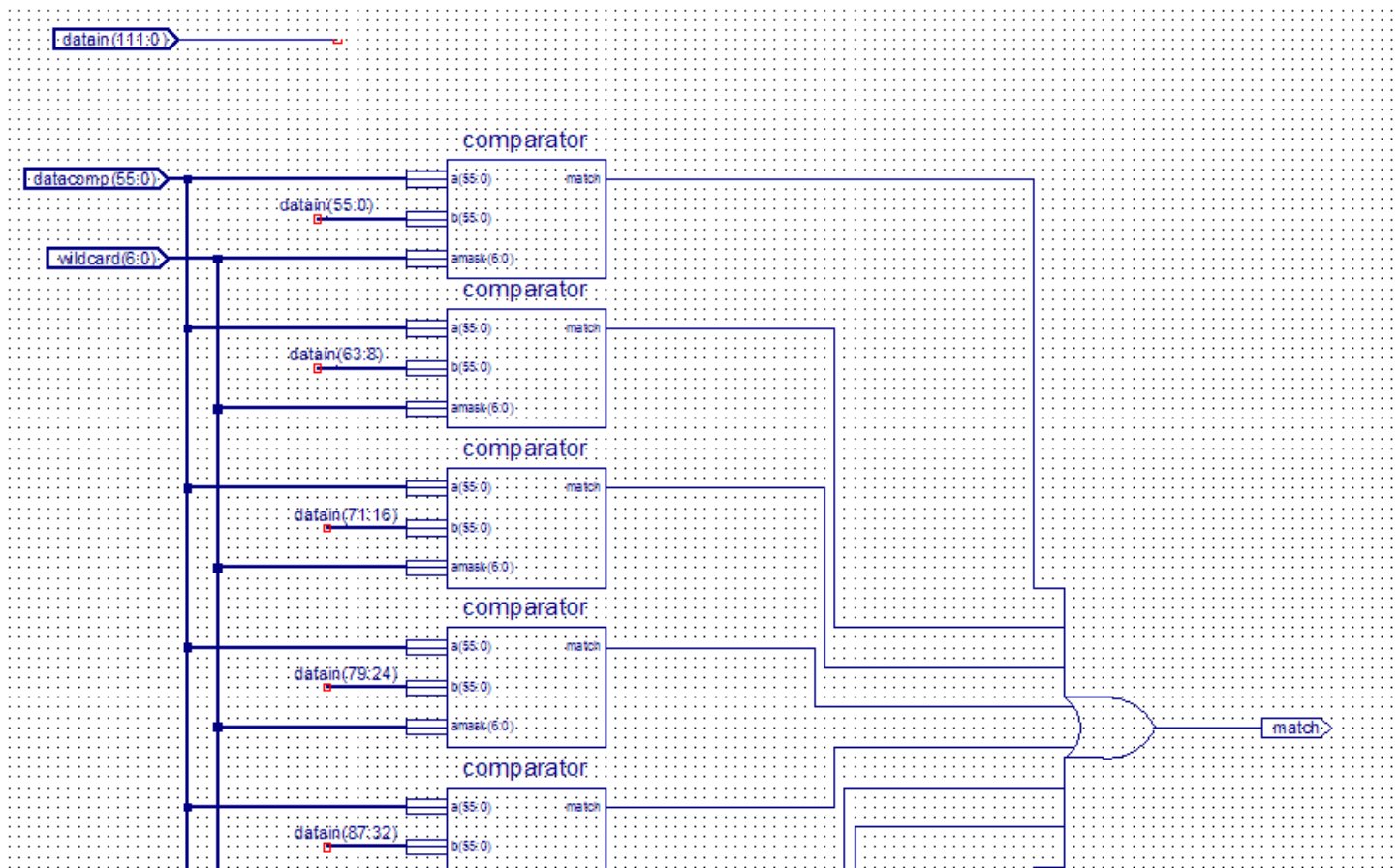
b)

i) What is the purpose of AMASK[6:0]?

AMASK[6:0] is used for deciding which byte of the pattern will be compared with the input data stream. So, 7'b1111111 means that all 7 bytes of the pattern will be matched against incoming packet bytes. If any bit is unset (i.e. set to 0), that byte won't be compared with the incoming data packet and will essentially be considered a don't care. This way our matching becomes smarter and not just blindly compares inputs to the same pattern irrespective of the situation

ii) What exactly does busmerge.v do?





Since the comparison that we need to make is 7-bytes (continuous), we need 2 chunks of 7-byte input data. This is because what if one chunk has a few bytes of matching pattern and the next incoming data chunk has the rest of the pattern. Therefore, what the BUSMERGE.v does is, concatenate 6-bytes of 'previous data' to the 8-bytes of 'newer data' (i.e. data from previous clock appended with data from latest clock), and send it for pattern matching and comparison.

iii) **What do the comp8 modules do in this schematic?**

The comparator's job is to compare bytes of incoming data packet and spit out whether there is a match or an equality between the reference data byte (the pattern stream byte) and the incoming data packet.

iv) **What is the purpose of dual9Bmem in dropfifo.sch?**

The job of the dual port 9B memory is to not let the pattern matched malicious data in and only let the non-malicious data in. This is 8B of data + 1B of control signal bits (ie. Total 9B). This fifo (which only lets in good data) decouples the data from the consumer.


```
comparator.vf detect7B.vf dropfifo.vf wordma
comparator.vf
22
23 module COMP8_MXILINX_comparator(A,
24                                     B,
25                                     EQ);
26
27     input [7:0] A;
28     input [7:0] B;
29     output EQ;
30
31     wire AB0;
32     wire AB1;
33     wire AB2;
34     wire AB3;
35     wire AB4;
36     wire AB5;
37     wire AB6;
38     wire AB7;
39     wire AB03;
40     wire AB47;
41
42     AND4 I_36_32 (.I0(AB7),
43                  .I1(AB6),
44                  .I2(AB5),
45                  .I3(AB4),
46                  .O(AB47));
47
48     XNOR2 I_36_33 (.I0(B[6]),
49                  .I1(A[6]),
50                  .O(AB6));
51
52     XNOR2 I_36_34 (.I0(B[7]),
53                  .I1(A[7]),
54                  .O(AB7));
55
56     XNOR2 I_36_35 (.I0(B[5]),
57                  .I1(A[5]),
58                  .O(AB5));
59
60     XNOR2 I_36_36 (.I0(B[4]),
61                  .I1(A[4]),
62                  .O(AB4));
63
64     AND4 I_36_41 (.I0(AB3),
65                  .I1(AB2),
66                  .I2(AB1),
67                  .I3(AB0),
68                  .O(AB03));
69
70     XNOR2 I_36_42 (.I0(B[2]),
71                  .I1(A[2]),
72                  .O(AB2));
73
74     XNOR2 I_36_43 (.I0(B[3]),
75                  .I1(A[3]),
76                  .O(AB3));
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937

```