UNIVERSITY OF SOUTHAMPTON

Synthesis of Multi-FPGA Systems with Asynchronous Communications

Volume 2 of 2

by

Tack Boon Yee

A thesis submitted for the degree of
Doctor of Philosophy.

School of Electronics and Computer Science,
University of Southampton

April, 2007

Appendix A

Paper

This appendix contains the paper published in the proceedings of the International Federation for Information Processing International Conference on Very Large Scale Integration 2005 (IFIP VLSI-SOC 2005).

The following published papers were included in the bound thesis. These have not been digitised due to copyright restrictions, but the links are provided.

Y. Tack Boon, M. Zwolinski, A.D. Brown (2005) "Multi-FPGA Synthesis with Asynchronous Communication Subsystems." *IFIP International Conference on Very Large Scale Integration (VLSI-SOC 2005).*

Appendix B

Hardware demonstrator in detail

This appendix contains implementation details of the hardware demonstrator and the Digilent D2-SB FPGA-based development board and DIO4 peripheral board used to implement the JPEG decoder.

The first few sections of this appendix provide the information on the JPEG decoder and a full profile of test images and photographs of the test images decoded by the multi-FPGA JPEG decoder. The rest of this appendix provides the detailed information on the hardware demonstrator. The information provided includes: circuit description of the BT121 VideoDAC on the I/O VGA peripheral board, user manuals of the development board, and the setting up of the hardware demonstrator.

B.1 JFIF (JPEG File Interchange Format)

The JPEG File Interchange Format is a minimal file format, which enables JPEG bitstreams to be exchanged between a wide variety of platforms and applications. The JFIF is entirely compatible with the standard JPEG interchange format and it conforms to the JPEG standard (ISO/IEC 10918-1 | ITU-T Recommendation T.81); the only additional requirement is the presence of a JFIF application segment marked by an *APP0* marker. The rest of this section provides the specifications and syntax of a JPEG file defined in Annex B of the ISO/IEC 10918-1 | ITU-T Recommendation T.81 and the JFIF application segment. The set of marker assignments and their description supported by the lossy sequential DCT-based JPEG decoder is listed in Table B-1 below.

Symbol	Code Assignment	Description
SOI	0xD8	Start of image
APP_0	0xE0	JFIF application segment
APP_n	0xE1 - 0xEF	Other APP segments
DQT	0xDB	Quantisation table
SOF_0	0xC0	Start of frame
DHT	0xC4	Huffman table
SOS	0xDA	Start of scan
COM	0xFE	Comment, may be ignored (skipped)
EOI	0xD9	End of image

Table B-1 Marker identifiers in the JFIF file

JFIF marker identifiers are preceded by an all '1' byte (0xFF). A two-byte SOI header (0xFF, 0xD8) identifies the JFIF file format, the APP_0 marker immediately follows the SOI header and subsequently by the other segments and markers. The end of file is identified by the EOI (0xFF, 0xD9) marker. Normally, the only marker identifier that should be found once the image data is started is the EOI marker. When a 0xFF byte is found followed by a zero byte, the zero byte must be discarded.

The following describes the JPEG file format and descriptions of the key segments given in Table B-1:

Header: It occupies two bytes (SOI: start of image – 0xFF, 0xDB)

Segments: Following the SOI marker, there can be any number of segments or markers described in Table B-1 above.

Trailer: It occupies two bytes. (EOI: end of image – 0xFF, 0xD9).

SOF0 (Start of Frame 0) marker

Field	Size in byte(s)	Description
Marker identifier	2	0xFF, 0xC0 to identify SOF0 marker.
Length	2	This value equals to 8+ component*3 value.
Data precision	1	This is in bits/sample, usually 8.
Image height	2	This must be >0.
Image width	2	This must be >0.
Number of components	1	Usually 1= grayscaled, 3= colour YCbCr or YIQ, 4= colour CMYK.
Each component	3	Read each component data of 3 bytes. It contains: Component ID (1 byte) (1= Y, 2= Cb, 3= Cr, 4= I, 5= Q), sampling factors (1 byte) (bits 0-3 vertical, bits 4-7 horizontal), quantisation table number (1 byte).

- The JFIF uses either 1 component (Y, grayscaled) or 3 components (YCbCr, sometimes called YUV, colour).

APP0 (JFIF segment) marker

Field	Size in byte(s)	Description
Marker identifier	2	0xFF, 0xE0 to identify APP0 marker.
Length	2	This must be >=16
File identifier mark	5	This identifies JFIF. 'JFIF#0' (0x4A, 0x46, 0x49, 0x46, 0x00)
Major revision number	2	Should be 1, otherwise error.
Minor revision number	2	Should be 0 to 2, otherwise try to decode anyway
Units for x/y densities	1	0= no units, x/y-density specifies the aspect ratio instead: 1= x/y-density are dots/inch, 2= x/y-density are dots/cm.
X-density	2	It should be >0.
Y-density	2	It should be >0.
Thumbnail width	1	-
Thumbnail height	1	-
Bytes to be read	n	For thumbnails (RGB 24-bits), n= width*height*3 bytes should be read immediately followed by the thumbnail height.

- If there is no 'JFIF#0' in the file identifier, or the length is <16, then it is probably not a JFIF segment and should be ignored.

- Normally units= 0, x-density= 1, y-density= 1 means the image has an aspect ratio of 1:1 (evenly scaled).

- JFIF files including thumbnails are very rare, the thumbnail can usually be ignored. If there is no thumbnail, then width= 0 and height= 0.

DHT (Define Huffman Table) marker

Field	Size in byte(s)	Description
Marker identifier	2	0xFF, 0xC4 to identify DHT marker.
Length	2	This specifies the length of Huffman table.
Huffman Table (HT) information	1	Bits 0-3: number of HT (0 to 3, otherwise error), Bit 4: type of HT (= DC table, 1= AC table), Bits 5-7: not used, must be 0.
Number of symbols	16	Number of symbols with codes of length 1 to 16, the sum(n) of these bytes is the total number of codes, which must be <= 256.
Symbols	n	Table containing the symbols in order of increasing code length (n= total number of codes).

- A single DHT segment may contain multiple Huffman tables, each with its own information byte.

DRI (Define Restart Interval) marker

Field	Size in byte(s)	Description
Marker identifier	2	0xFF, 0xDD to identify DRI marker.
Length	2	This must be 4.
Restart interval	2	This is in unit of MCU blocks, means that every *n* MCU blocks, a RST*n* marker can be found. The first marker will be RST*0*, then RST*1*, etc, after RST*7*, repeating from RST*0*.

DQT (Define Quantisation Table) marker

Field	Size in byte(s)	Description
Marker identifier	2	0xFF, 0xDB to identify DQT marker.
Length	2	This specifies the length of the quantisation table.
Quantisation Table (QT) information	1	Bits 0-3: number of QT (0 to 3, otherwise error), Bits 4-7: precision of QT (0= 8-bit, otherwise 16-bit).
Bytes	n	This gives the QT values, n= 64*(precision+ 1).

- A single DQT segment may contain multiple quantisation tables, each with its own information byte.

- For precision= 1 (16 bits), the order is high-low for each of the 64 words.

SOS (Start of Scan) marker

Field	Size in byte(s)	Description
Marker identifier	2	0xFF, 0xDA to identify SOS marker.
Length	2	This must be equal to 6+2* (number of components in scan).
Number of components in scan	1	This must be >=1 and <= 4 (otherwise error), usually 1 or 3.
Each component	2	For each component, read 2 bytes. It contains 1 byte: Component ID (1= Y, 2= Cb, 3= Cr, 4= I, 5= Q), 1 byte: Huffman table to use (bits 0-3: AC table 0 to 3, bits 4-7: DC table 0 to 3).
Ignorable bytes	3	Skip the next 3 bytes.

- The image data (scans) is immediately following the SOS segment.

B.2 JFIF test images

A complete profile of the test images decoded by the multi-FPGA JPEG decoder is given below. The following diagrams include the original JFIF file and photographs of the decoded test image using the hardware demonstrator system. Figure B-1 to Figure B-3 illustrate three 64-pixel by 64-pixel test images (LENA.jpg, MANDRILL.jpg, and DRAGON.jpg) decoded using the multi-FPGA JPEG decoder.

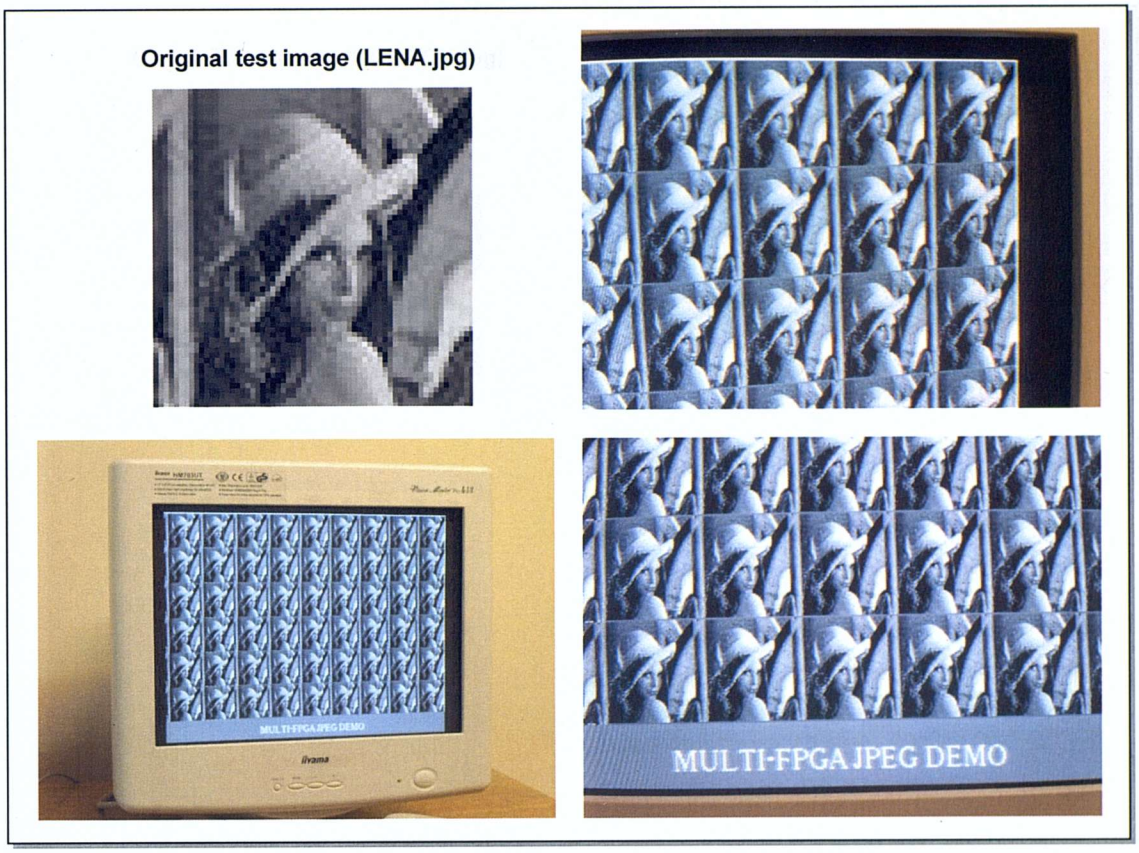


Figure B-1 JFIF test image (LENA.jpg)

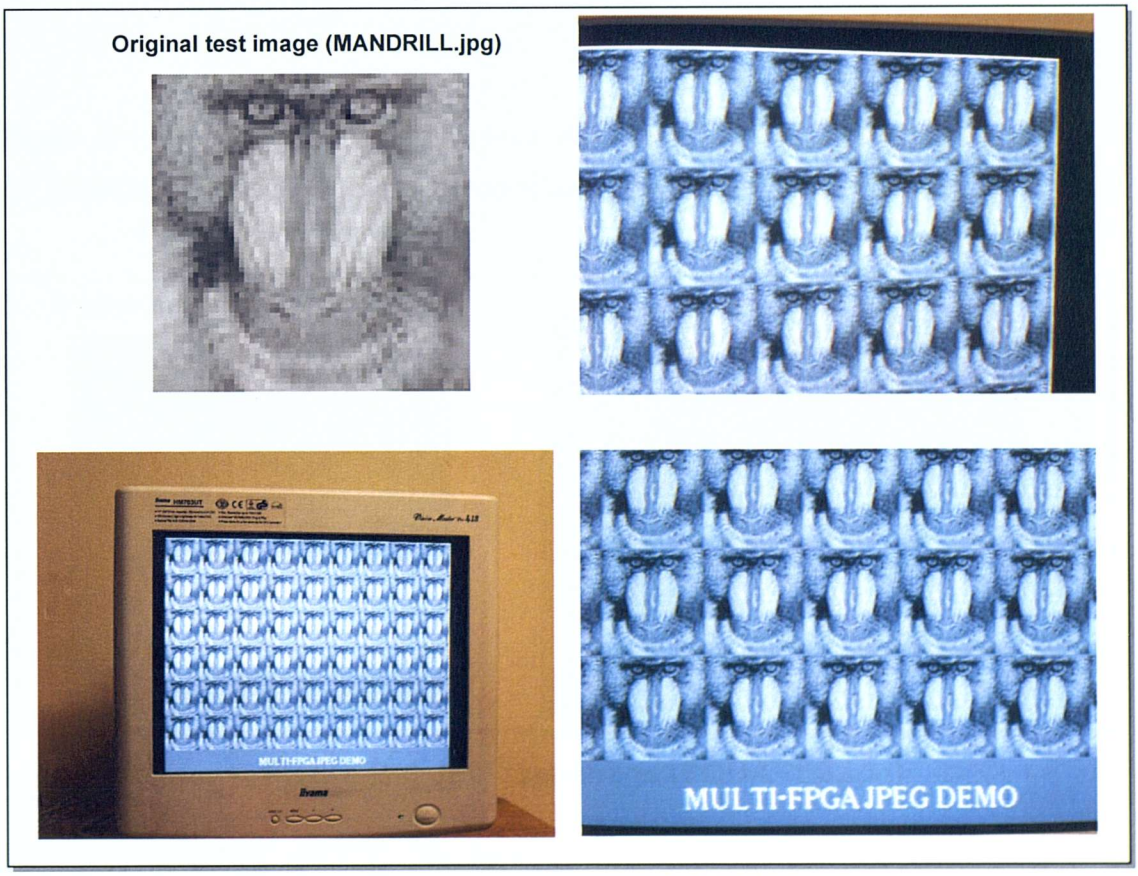


Figure B-2 JFIF test image (MANDRILL.jpg)

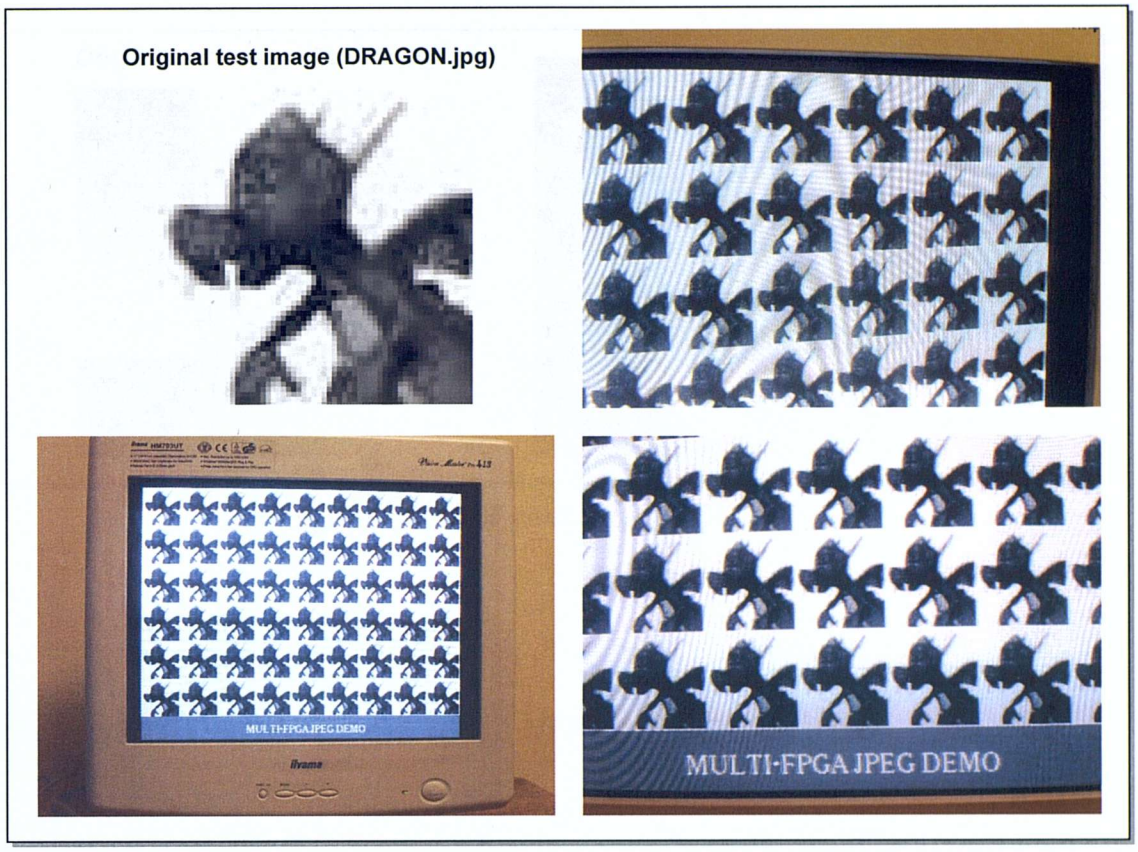


Figure B-3 JFIF test image (DRAGON.jpg)

Figure B-4 and Figure B-5 illustrate two 128-pixel by 128-pixel test images (SQUARES.jpg and SLOPE.jpg) decoded using the multi-FPGA JPEG decoder.

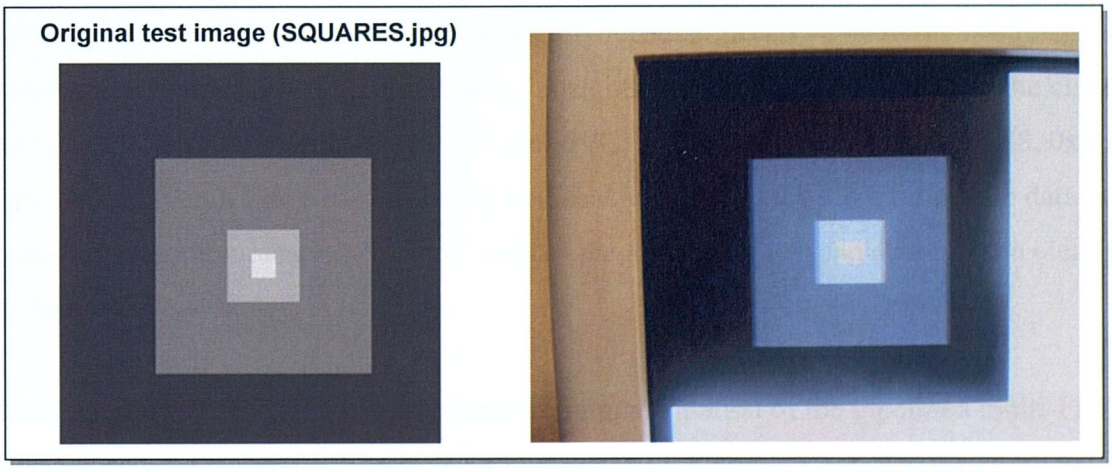


Figure B-4 JFIF test image (SQUARES.jpg)

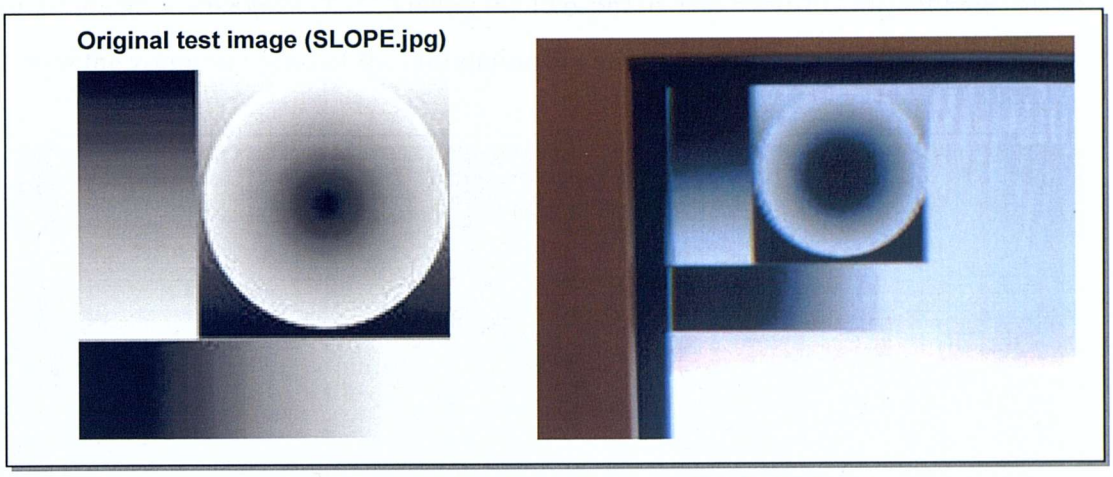


Figure B-5 JFIF test image (SLOPE.jpg)

B.3 Simulations of test image decoding

Post-MOODS multi-FPGA synthesis simulation results of the decoding of the LENA test image using the non-pipelined multi-FPGA JPEG decoder are given in Figure B-6 and a zoom in view in Figure B-7. The simulations show the signal transitions and data transfers of the various components (e.g. the UART RTL module, Frame buffer controller RTL module, etc), and the communication channels in the multi-FPGA JPEG decoder. The decoded pixel data are given in signal "/sim_top_level/decoded_data" (under the multi-FPGA JPEG decoder core divider) in Figure B-7, and cursors 1 and 2 mark the first decoded (pixel) value and the end of the eighth decoded (pixel) value in the test image respectively in the figure (e.g. the first to the eighth pixel values obtained from the close up view of the simulation in Figure B-7 are 0x7C, 0x94, 0x8A, 0x6F, 0x8C, 0x88, 0x8E, 0x65 respectively). The two-phase data handshaking scheme for the inter-device data in subprogram communication channel 2 (under the *SpC* 2 divider) can also be seen clearly in Figure B-7.

Simulation results for the 2-,3- and 6-device implementation of the pipelined multi-FPGA JPEG decoder are given in Figure B-8 to Figure B-13. Cursors 1 and 2 in the zoom in views of the simulations mark the first decoded (pixel) value and the end of the eighth decoded (pixel) value respectively. Inter-device data sent through the explicit

communication channels (*ExC*s) using the two-phase data handshaking scheme can be seen in the zoom in views of the simulations (e.g. *ExC* 4 in Figure B-9).

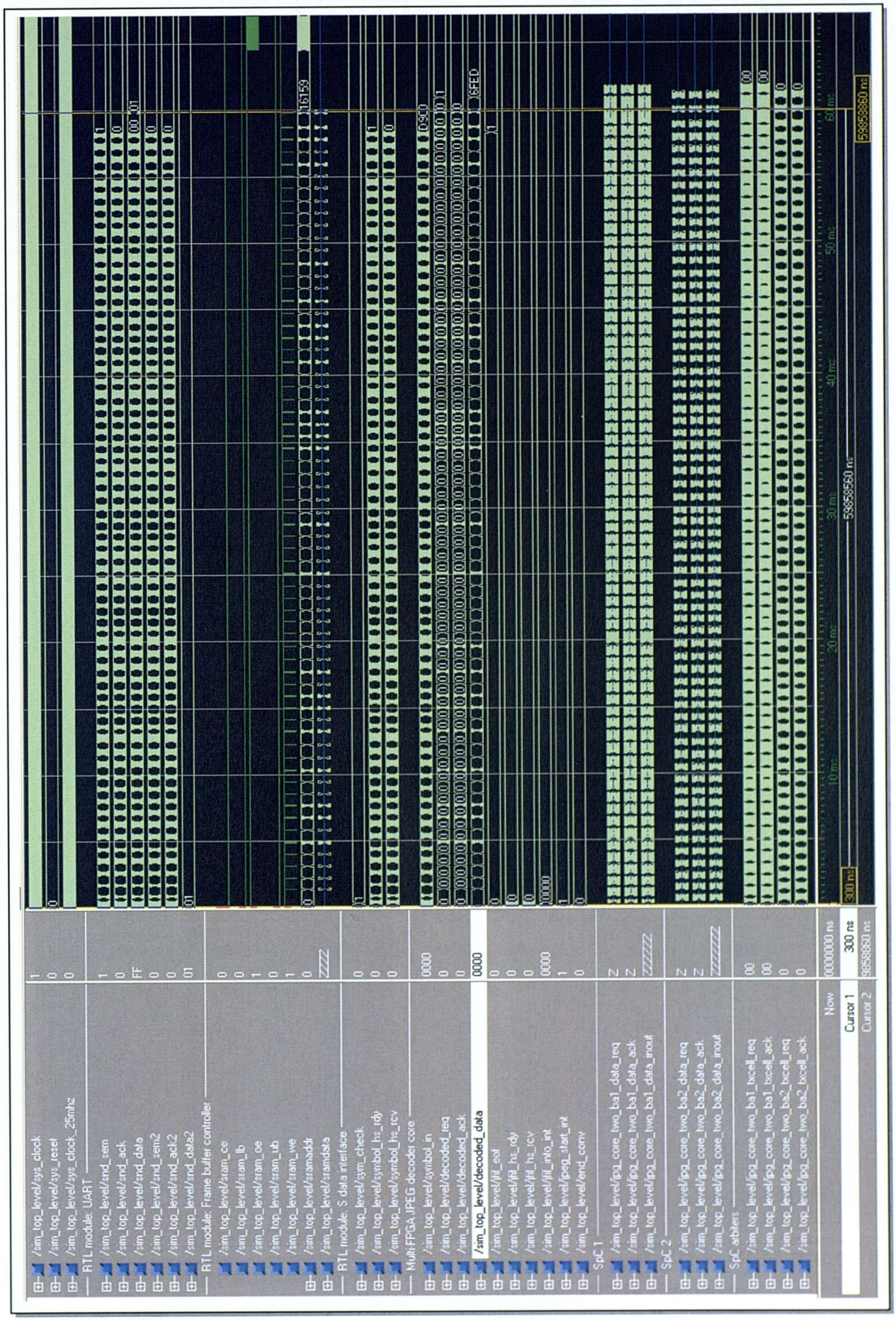


Figure B-6 Simulation of test image (LENA.JPG) decoding in a non-pipelined multi-FPGA JPEG decoder

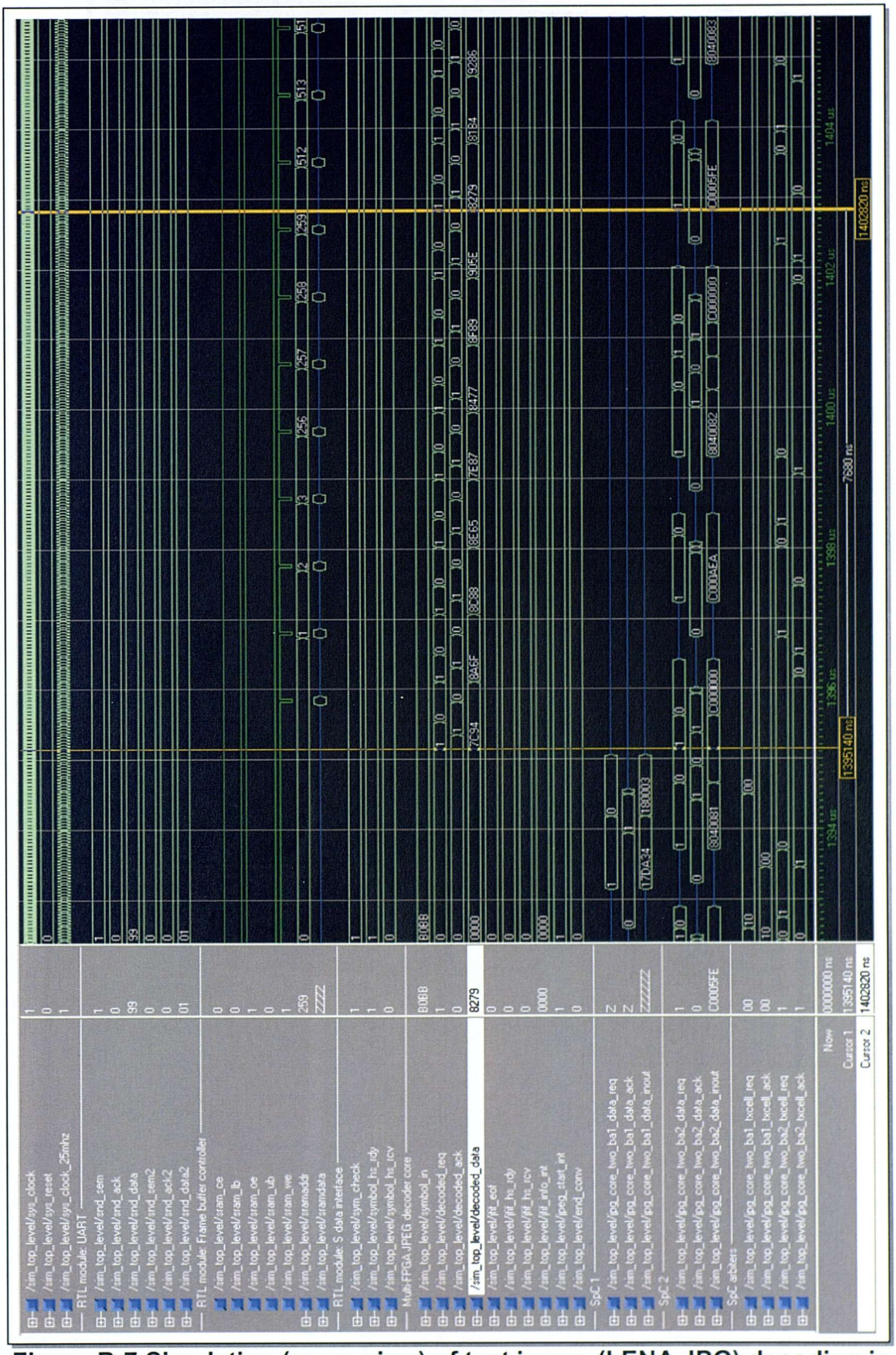


Figure B-7 Simulation (zoom view) of test image (LENA.JPG) decoding in a non-pipelined multi-FPGA JPEG decoder

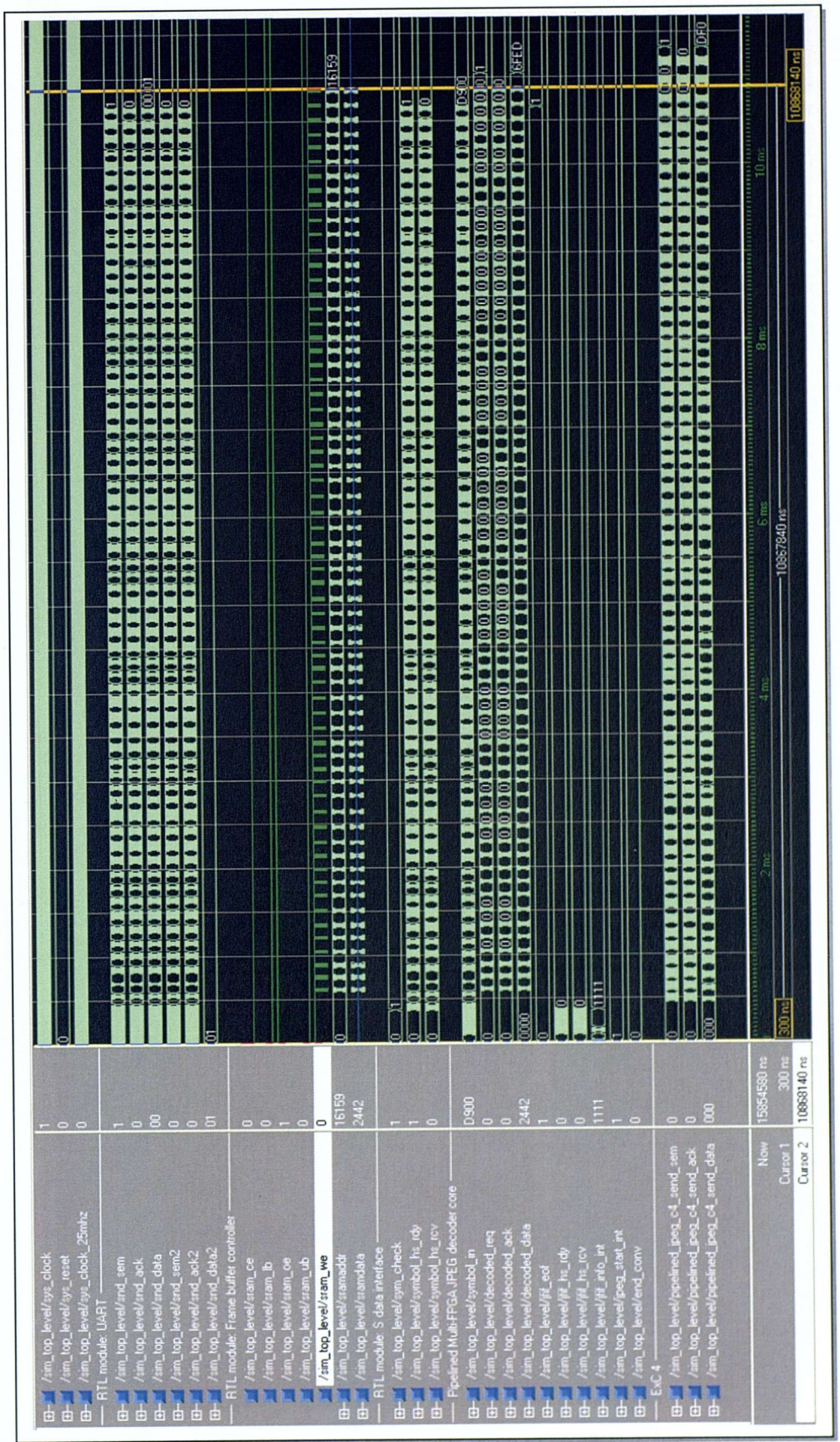


Figure B-8 Simulation of test image (LENA.JPG) decoding in a pipelined multi-FPGA JPEG decoder (2-device implementation)

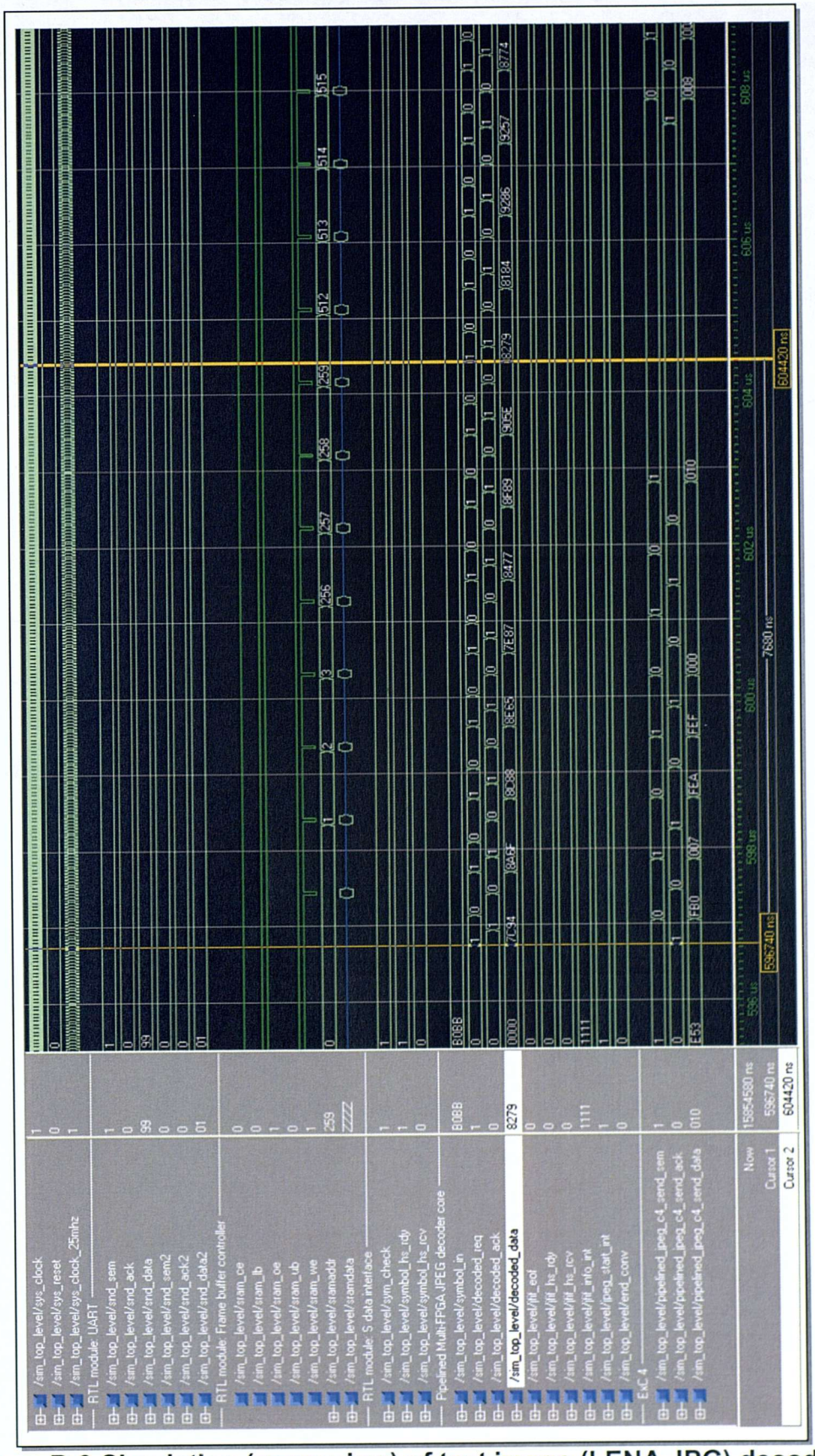


Figure B-9 Simulation (zoom view) of test image (LENA.JPG) decoding in a pipelined multi-FPGA JPEG decoder (2-device implementation)

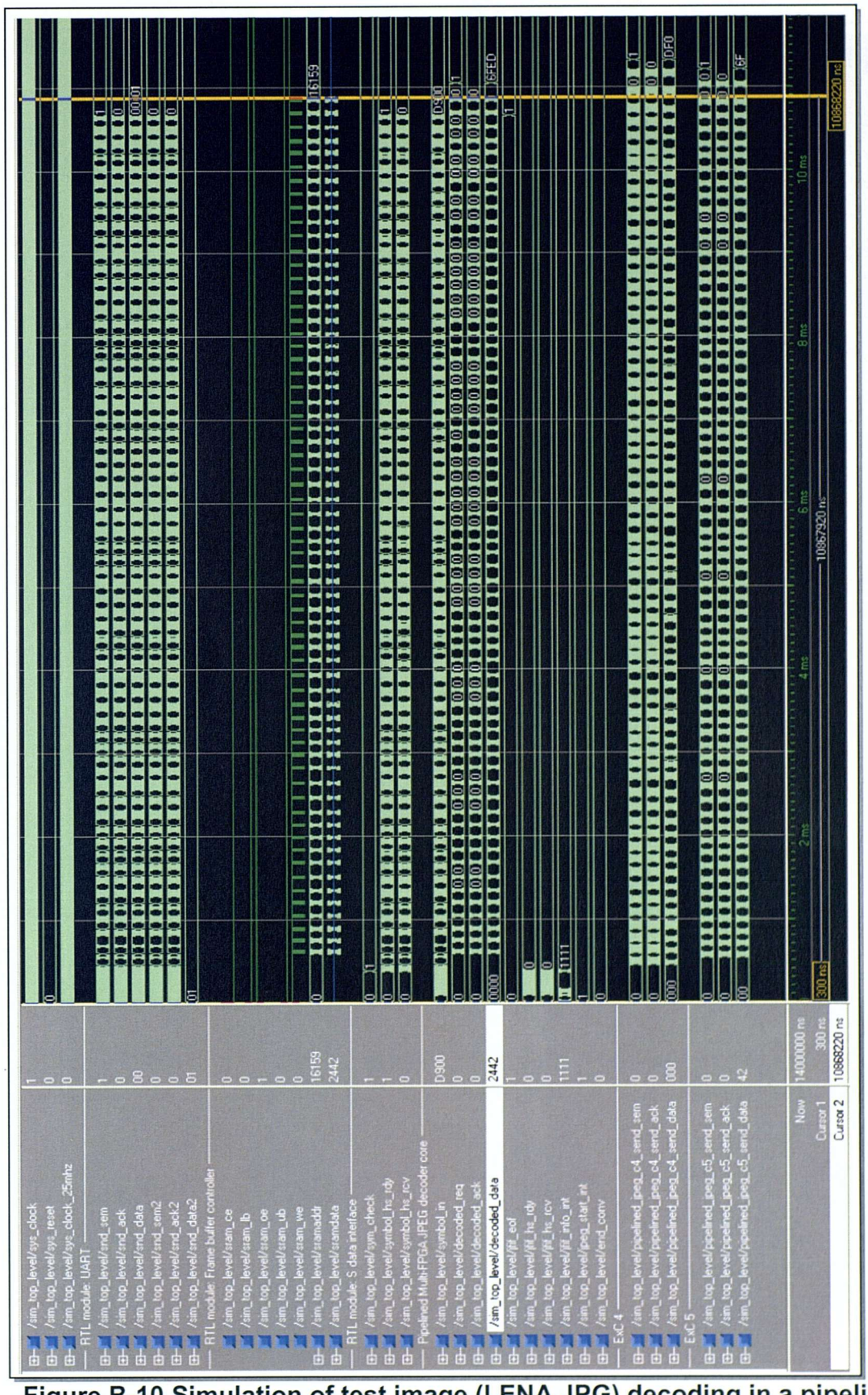


Figure B-10 Simulation of test image (LENA.JPG) decoding in a pipelined multi-FPGA JPEG decoder (3-device implementation)

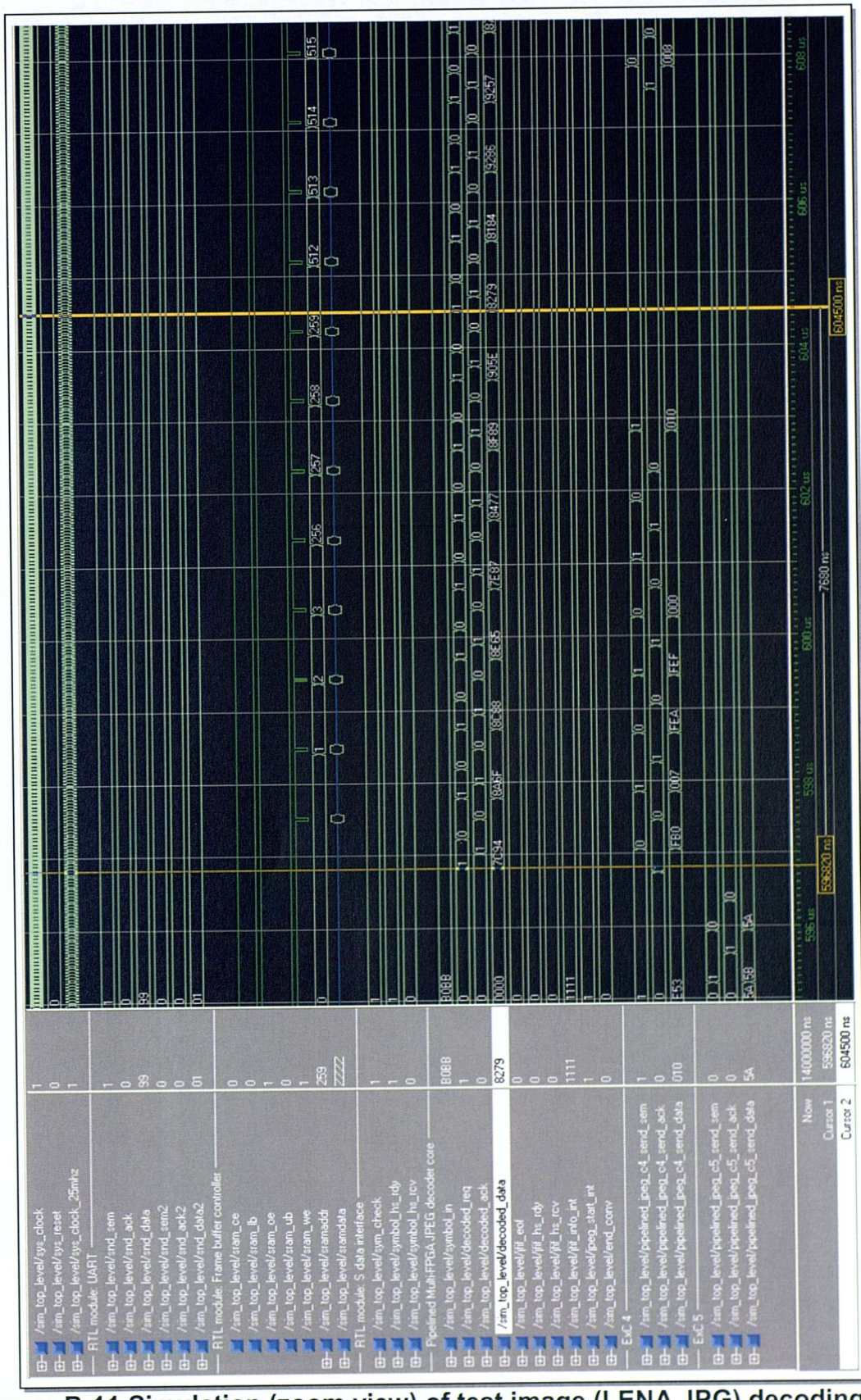


Figure B-11 Simulation (zoom view) of test image (LENA.JPG) decoding in a pipelined multi-FPGA JPEG decoder (3-device implementation)

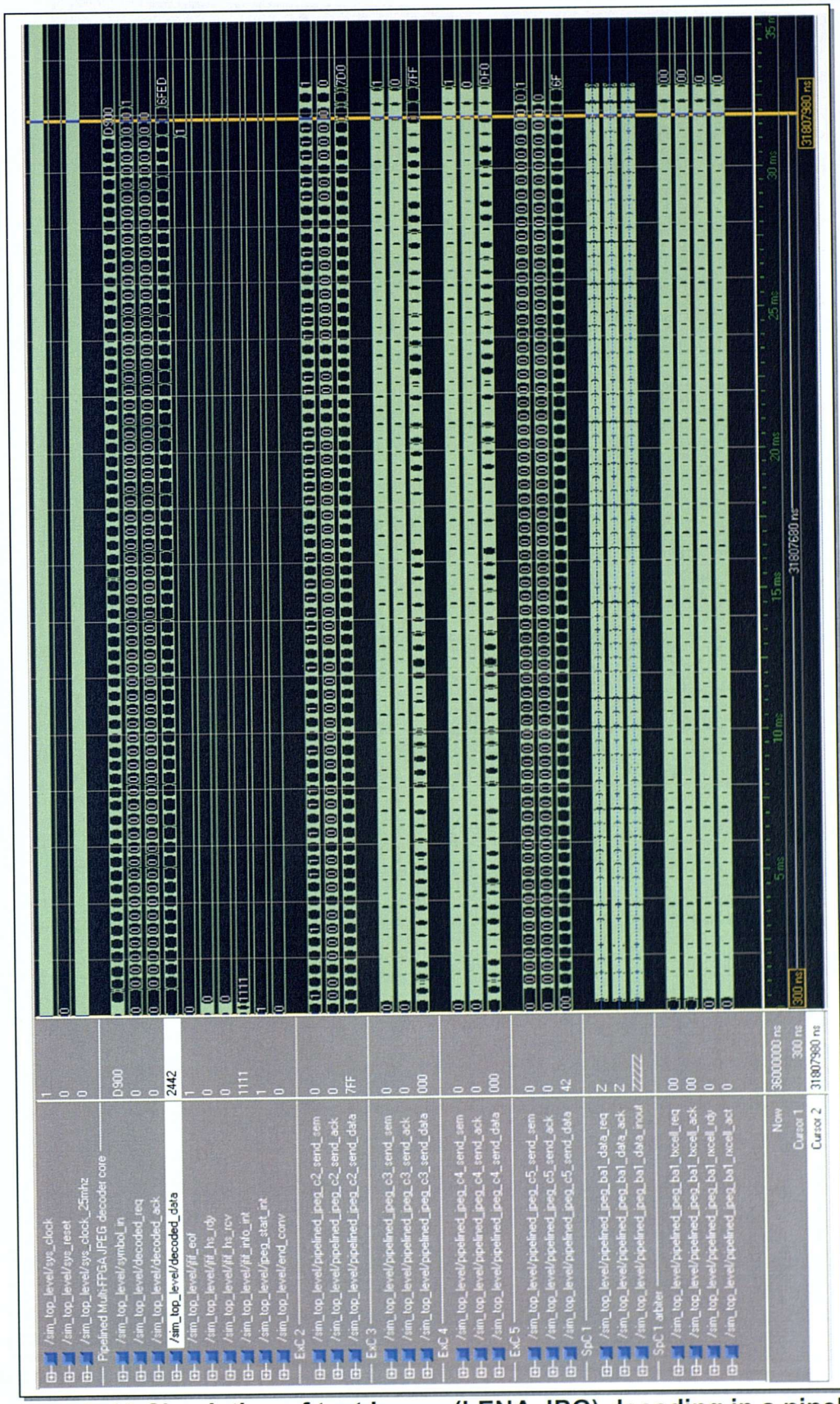


Figure B-12 Simulation of test image (LENA.JPG) decoding in a pipelined multi-FPGA JPEG decoder (6-device implementation)

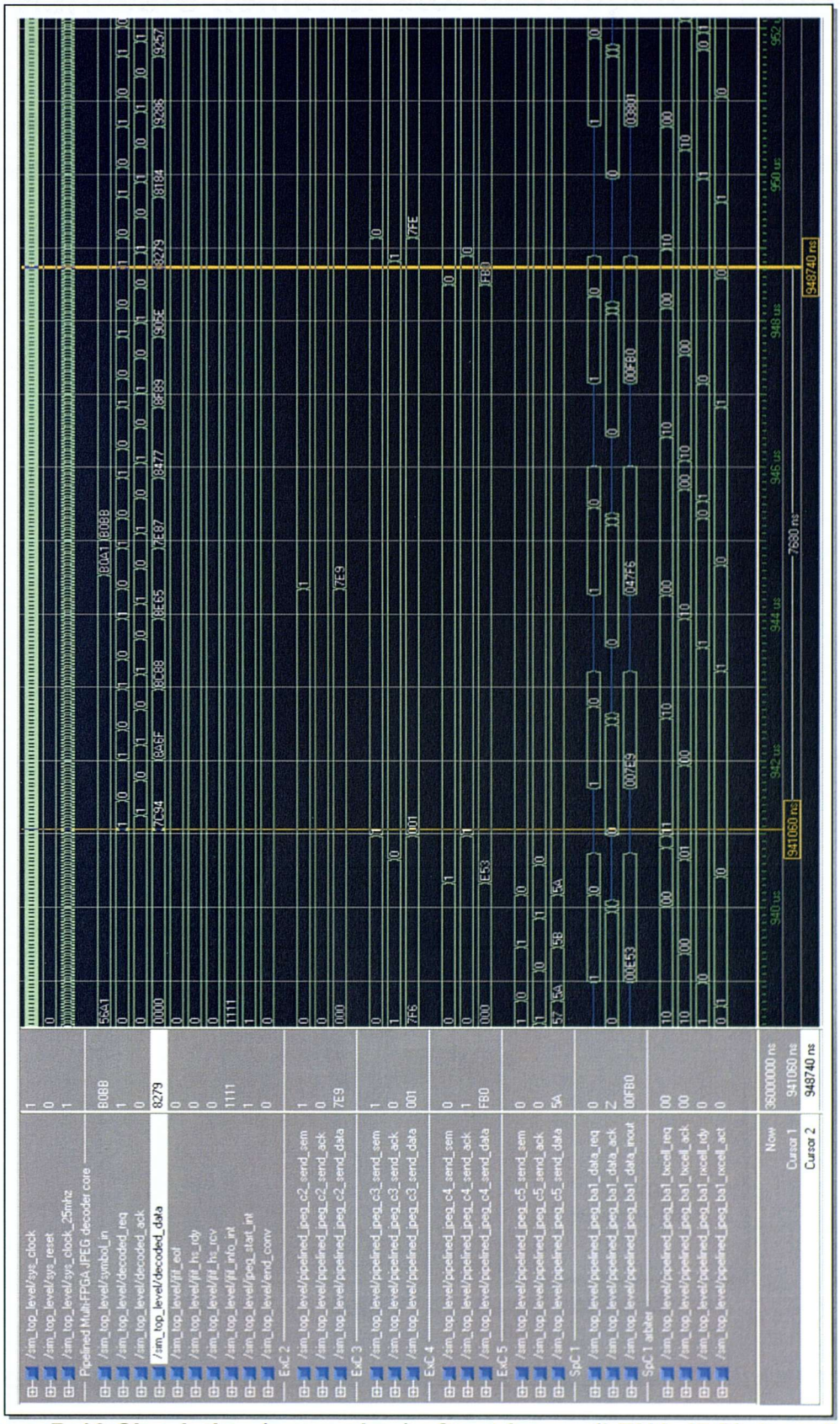


Figure B-13 Simulation (zoom view) of test image (LENA.JPG) decoding in a pipelined multi-FPGA JPEG decoder (6-device implementation)

B.4 Hardware demonstrator development board pin assignments

The multi-FPGA JPEG decoder hardware demonstrator is targeted onto three Digilent D2-SB development boards and one of the boards connected to the I/O VGA peripheral board as shown in the photograph of Figure B-14.

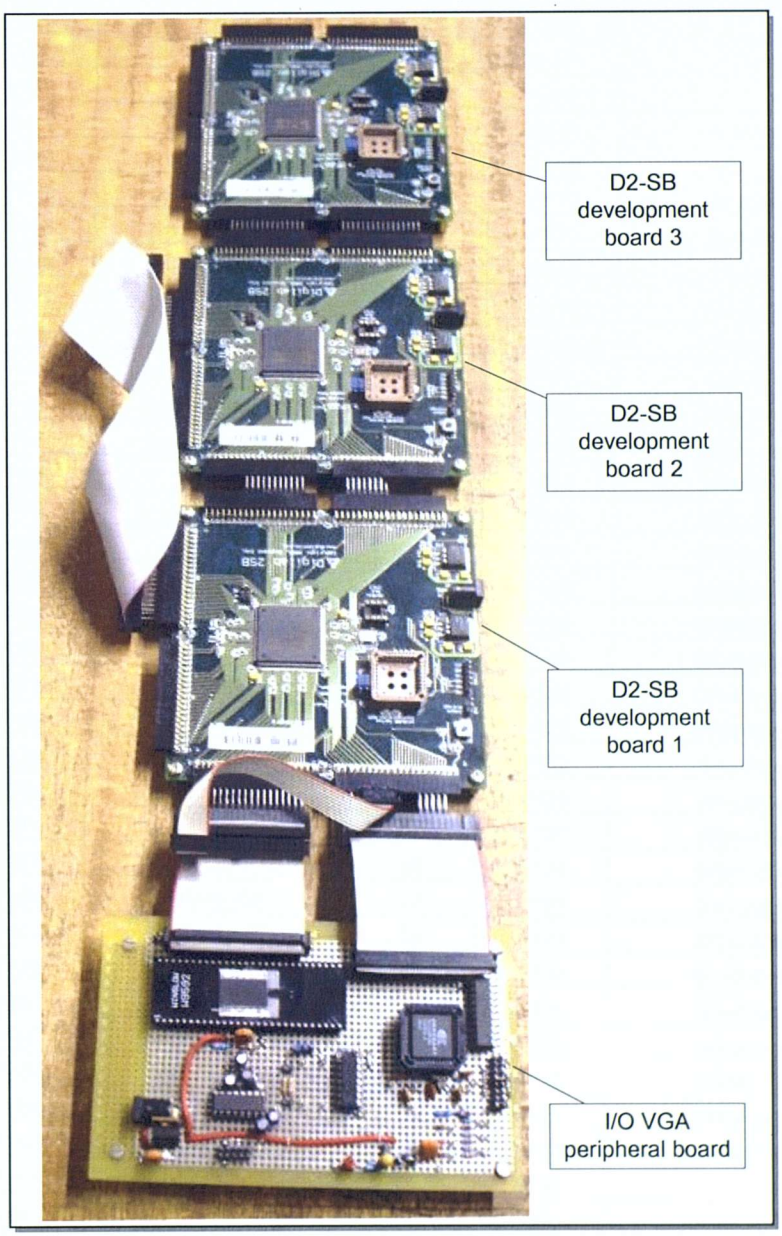


Figure B-14 Multi-FPGA board connections

The following tables give the pin assignments of the three Digilent D2-SB development boards; connectors that are not available (N/A) for user I/O assignments (e.g. VCC, GND) or not connected (n/c) are highlighted in grey. Table B-2 lists signals assigned to

connectors A1 and A2, Table B-3 lists signals assigned to connectors B1 and B2, and signals assigned to connectors C1, and C2 on development board 1 are given in Table B-4.

Connector A1			Connector A2		
Conn. pin	FPGA pin	signal	Conn. pin	FPGA pin	signal
1	N/A	GND	1	N/A	GND
2	N/A	VU	2	N/A	VU
3	N/A	VCC33	3	N/A	VCC33
4	P112		4	P162	
5	P111	vga_hsync_n	5	P161	SRAMAddr(1)
6	P110	vga_vsync_n	6	P160	SRAMAddr(0)
7	P109	pin_vga_gray(1)	7	P152	SRAMAddr(3)
8	P108	pin_vga_gray(0)	8	P151	SRAMAddr(2)
9	P102	pin_vga_gray(3)	9	P150	SRAMAddr(5)
10	P101	pin_vga_gray(2)	10	P149	SRAMAddr(4)
11	P100	pin_vga_gray(5)	11	P148	SRAMAddr(7)
12	P99	pin_vga_gray(4)	12	P147	SRAMAddr(6)
13	P98	pin_vga_gray(7)	13	P146	SRAMAddr(9)
14	P97	pin_vga_gray(6)	14	P145	SRAMAddr(8)
15	P96		15	P141	SRAMAddr(11)
16	P95		16	P140	SRAMAddr(10)
17	P94		17	P139	SRAMAddr(13)
18	P93		18	P138	SRAMAddr(12)
19	P89		19	P136	SRAMAddr(15)
20	P181		20	P135	SRAMAddr(14)
21	P87		21	P134	SRAMAddr(17)
22	P180		22	P133	SRAMAddr(16)
23	P179	SRAMData(12)	23	P132	SRAMData(1)
24	P178	SRAMData(13)	24	P129	SRAMData(0)
25	P176	SRAMData(14)	25	P127	SRAMData(3)
26	P175	SRAMData(15)	26	P126	SRAMData(2)
27	P174	SRAM_CE	27	P125	SRAMData(5)
28	P173	SRAM_WE	28	P123	SRAMData(4)
29	P169	SRAM_LB	29	P122	SRAMData(7))
30	P168	SRAM_UB	30	P121	SRAMData(6)
31	P167	SRAM_OE	31	P120	SRAMData(9)
32	P166		32	P116	SRAMData(8)
33	P165	RD	33	P115	SRAMData(11)
34	P164	TD	34	P114	SRAMData(10)
35	P163	pin_vgaclk_25Mhz	35	P113	
36	n/c		36	n/c	
37	n/c		37	n/c	
38	n/c		38	n/c	
39	n/c		39	P80	GCLK0
40	n/c		40	n/c	

Table B-2 Pin assignment of signals to connector A1 and A2 of development board 1

Connector B1			Connector B2		
Conn. pin	FPGA pin	signal	Conn. pin	FPGA pin	signal
1	N/A	GND	1	N/A	GND
2	N/A	VU	2	N/A	VU
3	N/A	VCC33	3	N/A	VCC33
4	P112		4	P71	
5	P111		5	P70	
6	P110		6	P69	
7	P109		7	P68	
8	P108		8	P64	
9	P102		9	P63	
10	P101		10	P62	Data_Symbol(0)
11	P100		11	P61	Data_Symbol(1)
12	P99		12	P60	Data_Symbol(2)
13	P98		13	P59	Data_Symbol(3)
14	P97		14	P58	Data_Symbol(4)
15	P96		15	P57	Data_Symbol(5)
16	P95		16	P56	Data_Symbol(6)
17	P94		17	P55	Data_Symbol(7)
18	P93		18	P49	JFIF_hs_rdy
19	P89		19	P48	JFIF_hs_rcv
20	P88		20	P47	Symbol_hs_rdy
21	P87		21	P46	Symbol_hs_rcv
22	P86		22	n/c	
23	P84		23	n/c	
24	P83		24	n/c	
25	P82		25	n/c	
26	P81		26	n/c	
27	P75		27	n/c	
28	P74		28	n/c	
29	P73		29	n/c	
30	n/c		30	n/c	
31	n/c		31	n/c	
32	n/c		32	n/c	
33	n/c		33	n/c	
34	n/c		34	n/c	
35	n/c		35	n/c	
36	n/c		36	n/c	
37	n/c		37	n/c	
38	n/c		38	n/c	
39	n/c		39	n/c	
40	n/c		40	n/c	

Table B-3 Pin assignment of signals to connector B1 and B2 of development board 1

Connector C1			Connector C2		
Conn. pin	FPGA pin	signal	Conn. pin	FPGA pin	signal
1	N/A	GND	1	N/A	GND
2	N/A	VU	2	N/A	VU
3	N/A	VCC33	3	N/A	VCC33
4	P112		4	P23	
5	P111		5	P22	decoded_data(5)
6	P110		6	P21	
7	P109		7	P20	decoded_data(6)
8	P108		8	P18	decoded_data(7)
9	P102		9	P17	decoded_data(8)
10	P101		10	P16	decoded_data(9)
11	P99		11	P15	decoded_data(10)
12	P99		12	P11	decoded_data(11)
13	P98		13	P10	decoded_data(12)
14	P97		14	P9	decoded_data(13)
15	P96	JFIF_eof	15	P8	decoded_data(14)
16	P95	Data_Symbol(8)	16	P7	decoded_data(15)
17	P94	Data_Symbol(9)	17	P6	end_conv
18	P93	Data_Symbol(10)	18	P5	s_sym_check
19	P89	Data_Symbol(11)	19	P4	
20	P45	Data_Symbol(12)	20	P3	JPEG_start
21	P87	Data_Symbol(13)	21	P206	
22	P44	Data_Symbol(14)	22	P205	
23	P43	Data_Symbol(15)	23	P204	
24	P42	JFIF_info(0)	24	P203	
25	P41	JFIF_info(1)	25	P202	
26	P40	JFIF_info(2)	26	P201	
27	P36	JFIF_info(3)	27	P200	
28	P35		28	P199	
29	P34	decoded_req	29	P198	
30	P33	decoded_ack	30	P194	
31	P31	decoded_data(0)	31	P193	
32	P30	decoded_data(1)	32	P192	
33	P29	decoded_data(2)	33	P191	
34	P27	decoded_data(3)	34	P189	
35	P24	decoded_data(4)	35	P188	
36	n/c		36	n/c	
37	n/c		37	n/c	
38	n/c		38	n/c	
39	n/c		39	P77	GCLK1
40	n/c		40	n/c	

Table B-4 Pin assignment of signals to connector C1 and C2 of development board 1

Table B-5 lists signals assigned to connectors A1 and A2, Table B-6 lists signals assigned to connectors B1 and B2, and signals assigned to connectors C1, and C2 on development board 2 are given in Table B-7.

Connector A1			Connector A2		
Conn. pin	FPGA pin	signal	Conn. pin	FPGA pin	signal
1	N/A	GND	1	N/A	GND
2	N/A	VU	2	N/A	VU
3	N/A	VCC33	3	N/A	VCC33
4	P112		4	P162	
5	P111		5	P161	decoded_data(4)
6	P110		6	P160	
7	P109		7	P152	decoded_data(2)
8	P108		8	P151	decoded_data(3)
9	P102		9	P150	decoded_data(0)
10	P101		10	P149	decoded_data(1)
11	P100		11	P148	decoded_req
12	P99		12	P147	decoded_ack
13	P98		13	P146	JFIF_info(3)
14	P97		14	P145	
15	P96		15	P141	JFIF_info(1)
16	P95		16	P140	JFIF_info(2)
17	P94		17	P139	Data_Symbol(15)
18	P93		18	P138	JFIF_info(0)
19	P89		19	P136	Data_Symbol(13)
20	P181		20	P135	Data_Symbol(14)
21	P87		21	P134	Data_Symbol(11)
22	P180	JPEG_start	22	P133	Data_Symbol(12)
23	P179	end_conv	23	P132	Data_Symbol(9)
24	P178	s_sym_check	24	P129	Data_Symbol(10)
25	P176	decoded_data(14)	25	P127	JFIF_eof
26	P175	decoded_data(15)	26	P126	Data_Symbol(8)
27	P174	decoded_data(12)	27	P125	
28	P173	decoded_data(13)	28	P123	
29	P169	decoded_data(10)	29	P122	
30	P168	decoded_data(11)	30	P121	
31	P167	decoded_data(8)	31	P120	
32	P166	decoded_data(9)	32	P116	
33	P165	decoded_data(6)	33	P115	
34	P164	decoded_data(7)	34	P114	
35	P163	decoded_data(5)	35	P113	
36	n/c		36	n/c	
37	n/c		37	n/c	
38	n/c		38	n/c	
39	n/c		39	P80	GCLK0
40	n/c		40	n/c	

Table B-5 Pin assignment of signals to connector A1 and A2 of development board 2

Connector B1			Connector B2		
Conn. pin	FPGA pin	signal	Conn. pin	FPGA pin	signal
1	N/A	GND	1	N/A	GND
2	N/A	VU	2	N/A	VU
3	N/A	VCC33	3	N/A	VCC33
4	P112		4	P71	
5	P111		5	P70	
6	P110		6	P69	
7	P109		7	P68	
8	P108		8	P64	
9	P102		9	P63	
10	P101		10	P62	Data_Symbol(0)
11	P100		11	P61	Data_Symbol(1)
12	P99		12	P60	Data_Symbol(2)
13	P98		13	P59	Data_Symbol(3)
14	P97		14	P58	Data_Symbol(4)
15	P96		15	P57	Data_Symbol(5)
16	P95		16	P56	Data_Symbol(6)
17	P94		17	P55	Data_Symbol(7)
18	P93		18	P49	JFIF_hs_rdy
19	P89		19	P48	JFIF_hs_rcv
20	P88		20	P47	Symbol_hs_rdy
21	P87		21	P46	Symbol_hs_rcv
22	P86		22	n/c	
23	P84		23	n/c	
24	P83		24	n/c	
25	P82		25	n/c	
26	P81		26	n/c	
27	P75		27	n/c	
28	P74		28	n/c	
29	P73		29	n/c	
30	n/c		30	n/c	
31	n/c		31	n/c	
32	n/c		32	n/c	
33	n/c		33	n/c	
34	n/c		34	n/c	
35	n/c		35	n/c	
36	n/c		36	n/c	
37	n/c		37	n/c	
38	n/c		38	n/c	
39	n/c		39	n/c	
40	n/c		40	n/c	

Table B-6 Pin assignment of signals to connector B1 and B2 of development board 2

Connector C1			Connector C2		
Conn. pin	FPGA pin	signal	Conn. pin	FPGA pin	signal
1	N/A	GND	1	N/A	GND
2	N/A	VU	2	N/A	VU
3	N/A	VCC33	3	N/A	VCC33
4	P112		4	P23	
5	P111		5	P22	jpg_core_two_ba2_Data_inout(2)
6	P110		6	P21	
7	P109	jpg_core_two_ba1_Data_req	7	P20	jpg_core_two_ba2_Data_inout(3)
8	P108	jpg_core_two_ba1_Data_ack	8	P18	jpg_core_two_ba2_Data_inout(4)
9	P102	jpg_core_two_ba1_Data_inout(0)	9	P17	jpg_core_two_ba2_Data_inout(5)
10	P101	jpg_core_two_ba1_Data_inout(1)	10	P16	jpg_core_two_ba2_Data_inout(6)
11	P99	jpg_core_two_ba1_Data_inout(2)	11	P15	jpg_core_two_ba2_Data_inout(7)
12	P99	jpg_core_two_ba1_Data_inout(3)	12	P11	jpg_core_two_ba2_Data_inout(8)
13	P98	jpg_core_two_ba1_Data_inout(4)	13	P10	jpg_core_two_ba2_Data_inout(9)
14	P97	jpg_core_two_ba1_Data_inout(5)	14	P9	jpg_core_two_ba2_Data_inout(10)
15	P96	jpg_core_two_ba1_Data_inout(6)	15	P8	jpg_core_two_ba2_Data_inout(11)
16	P95	jpg_core_two_ba1_Data_inout(7)	16	P7	jpg_core_two_ba2_Data_inout(12)
17	P94	jpg_core_two_ba1_Data_inout(8)	17	P6	jpg_core_two_ba2_Data_inout(13)
18	P93	jpg_core_two_ba1_Data_inout(9)	18	P5	jpg_core_two_ba2_Data_inout(14)
19	P89	jpg_core_two_ba1_Data_inout(10)	19	P4	jpg_core_two_ba2_Data_inout(15)
20	P45	jpg_core_two_ba1_Data_inout(11)	20	P3	jpg_core_two_ba2_Data_inout(16)
21	P87	jpg_core_two_ba1_Data_inout(12)	21	P206	jpg_core_two_ba2_Data_inout(17)
22	P44	jpg_core_two_ba1_Data_inout(13)	22	P205	jpg_core_two_ba2_Data_inout(18)
23	P43	jpg_core_two_ba1_Data_inout(14)	23	P204	jpg_core_two_ba2_Data_inout(19)
24	P42	jpg_core_two_ba1_Data_inout(15)	24	P203	jpg_core_two_ba2_Data_inout(20)
25	P41	jpg_core_two_ba1_Data_inout(16)	25	P202	jpg_core_two_ba2_Data_inout(21)
26	P40	jpg_core_two_ba1_Data_inout(17)	26	P201	jpg_core_two_ba2_Data_inout(22)
27	P36	jpg_core_two_ba1_Data_inout(18)	27	P200	jpg_core_two_ba2_Data_inout(23)
28	P35	jpg_core_two_ba1_Data_inout(19)	28	P199	jpg_core_two_ba2_Data_inout(24)
29	P34	jpg_core_two_ba1_Data_inout(20)	29	P198	jpg_core_two_ba2_Data_inout(25)
30	P33	jpg_core_two_ba1_txcell_req1(0)	30	P194	jpg_core_two_ba2_Data_inout(26)
31	P31	jpg_core_two_ba1_txcell_req1(1)	31	P193	jpg_core_two_ba2_Data_inout(27)
32	P30	jpg_core_two_ba1_txcell_ack1(0)	32	P192	jpg_core_two_ba2_txcell_req1
33	P29	jpg_core_two_ba1_txcell_ack1(1)	33	P191	jpg_core_two_ba2_txcell_ack1
34	P27	jpg_core_two_ba2_Data_inout(0)	34	P189	jpg_core_two_ba2_Data_req
35	P24	jpg_core_two_ba2_Data_inout(1)	35	P188	jpg_core_two_ba2_Data_ack
36	n/c		36	n/c	
37	n/c		37	n/c	
38	n/c		38	n/c	
39	n/c		39	P77	GCLK1
40	n/c		40	n/c	

Table B-7 Pin assignment of signals to connector C1 and C2 of development board 2

Table B-8 lists signals assigned to connectors A1 and A2, and Table B-9 lists signals assigned to connectors B1, B2, C1, and C2 on development board 3.

Connector A1			Connector A2		
Conn. pin	FPGA pin	signal	Conn. pin	FPGA pin	signal
1	N/A	GND	1	N/A	GND
2	N/A	VU	2	N/A	VU
3	N/A	VCC33	3	N/A	VCC33
4	P112		4	P162	
5	P111	vga_hsync_n	5	P161	SRAMAddr(1)
6	P110	vga_vsync_n	6	P160	SRAMAddr(0)
7	P109	pin_vga_gray(1)	7	P152	SRAMAddr(3)
8	P108	pin_vga_gray(0)	8	P151	SRAMAddr(2)
9	P102	pin_vga_gray(3)	9	P150	SRAMAddr(5)
10	P101	pin_vga_gray(2)	10	P149	SRAMAddr(4)
11	P100	pin_vga_gray(5)	11	P148	SRAMAddr(7)
12	P99	pin_vga_gray(4)	12	P147	SRAMAddr(6)
13	P98	pin_vga_gray(7)	13	P146	SRAMAddr(9)
14	P97	pin_vga_gray(6)	14	P145	SRAMAddr(8)
15	P96		15	P141	SRAMAddr(11)
16	P95		16	P140	SRAMAddr(10)
17	P94		17	P139	SRAMAddr(13)
18	P93		18	P138	SRAMAddr(12)
19	P89		19	P136	SRAMAddr(15)
20	P181		20	P135	SRAMAddr(14)
21	P87		21	P134	SRAMAddr(17)
22	P180		22	P133	SRAMAddr(16)
23	P179	SRAMData(12)	23	P132	SRAMData(1)
24	P178	SRAMData(13)	24	P129	SRAMData(0)
25	P176	SRAMData(14)	25	P127	SRAMData(3)
26	P175	SRAMData(15)	26	P126	SRAMData(2)
27	P174	SRAM_CE	27	P125	SRAMData(5)
28	P173	SRAM_WE	28	P123	SRAMData(4)
29	P169	SRAM_LB	29	P122	SRAMData(7))
30	P168	SRAM_UB	30	P121	SRAMData(6)
31	P167	SRAM_OE	31	P120	SRAMData(9)
32	P166		32	P116	SRAMData(8)
33	P165	RD	33	P115	SRAMData(11)
34	P164	TD	34	P114	SRAMData(10)
35	P163	pin_vgaclk_25Mhz	35	P113	
36	n/c		36	n/c	
37	n/c		37	n/c	
38	n/c		38	n/c	
39	n/c		39	P80	GCLK0
40	n/c		40	n/c	

Table B-8 Pin assignment of signals to connector A1 and A2 of development board 3

Connector B1			Connector B2			Connector C1			Connector C2		
Conn. pin	FPGA pin	signal	Conn. pin	FPGA pin	signal	Conn. pin	FPGA pin	signal	Conn. pin	FPGA pin	signal
1	N/A	GND	1	N/A	GND	1	N/A	GND	1	N/A	GND
2	N/A	VU	2	N/A	VU	2	N/A	VU	2	N/A	VU
3	N/A	VCC33	3	N/A	VCC33	3	N/A	VCC33	3	N/A	VCC33
4	P112		4	P71		4	P112		4	P23	
5	P111		5	P70		5	P111		5	P22	
6	P110		6	P69		6	P110		6	P21	
7	P109		7	P68		7	P109		7	P20	
8	P108		8	P64		8	P108		8	P18	
9	P102		9	P63		9	P102		9	P17	
10	P101		10	P62		10	P101		10	P16	
11	P100		11	P61		11	P99		11	P15	
12	P99		12	P60		12	P99		12	P11	
13	P98		13	P59		13	P98		13	P10	
14	P97		14	P58		14	P97		14	P9	
15	P96		15	P57		15	P96		15	P8	
16	P95		16	P56		16	P95		16	P7	
17	P94		17	P55		17	P94		17	P6	
18	P93		18	P49		18	P93		18	P5	
19	P89		19	P48		19	P89		19	P4	
20	P88		20	P47		20	P45		20	P3	
21	P87		21	P46		21	P87		21	P206	
22	P86		22	n/c		22	P44		22	P205	
23	P84		23	n/c		23	P43		23	P204	
24	P83		24	n/c		24	P42		24	P203	
25	P82		25	n/c		25	P41		25	P202	
26	P81		26	n/c		26	P40		26	P201	
27	P75		27	n/c		27	P36		27	P200	
28	P74		28	n/c		28	P35		28	P199	
29	P73		29	n/c		29	P34		29	P198	
30	n/c		30	n/c		30	P33		30	P194	
31	n/c		31	n/c		31	P31		31	P193	
32	n/c		32	n/c		32	P30		32	P192	
33	n/c		33	n/c		33	P29		33	P191	
34	n/c		34	n/c		34	P27		34	P189	
35	n/c		35	n/c		35	P24		35	P188	
36	n/c		36	n/c		36	n/c		36	n/c	
37	n/c		37	n/c		37	n/c		37	n/c	
38	n/c		38	n/c		38	n/c		38	n/c	
39	n/c		39	n/c		39	n/c		39	P77	GCLK1
40	n/c		40	n/c		40	n/c		40	n/c	

Table B-9 Pin assignment of signals to connector B1, B2, C1, and C2 of development board 3

B.5 Circuit description of the Bt121 triple 8-bit VideoDAC

The BT121 is a triple 8-bit videoDAC designed specifically for high-performance, high-resolution colour graphics. The BT121 generates RS-343A-compatible video signals into a doubly-terminated 75Ω load, and RS-170-compatible video signals into a singly-terminated 75Ω load, without requiring external buffering. Both the differential and integral linearity errors of the D/A converters are guaranteed to be a maximum of ± 1 LSB over the full temperature range. The functional block diagram of the BT121 is given in Figure B-15.

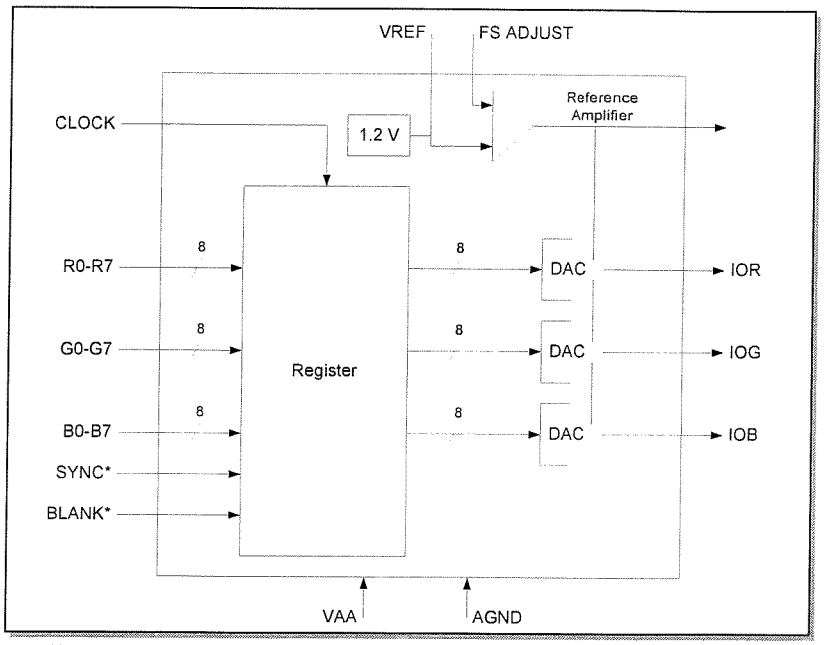


Figure B-15 Functional block diagram of the BT121 videoDAC

As illustrated in the functional block diagram, the BT121 contains three 8-bit D/A converters, input registers, and a reference amplifier. On the rising edge of CLOCK, 24 bits of colour information (R0-R7, G0-G7, and B0-B7) are latched into the device and presented to the three 8-bit D/A converters. Latched on the rising edge of CLOCK to maintain synchronisation with the colour data, the SYNC* and BLANK* inputs add appropriately weighted currents to the analogue outputs, producing the specific output levels required for video applications.

The D/A converters on the BT121 use a segmented architecture in which bit currents are routed to either the outputs or GND by a sophisticated decoding scheme. This architecture eliminates the need for precision component ratios and greatly reduces the switching

transients associated with turning current sources on and off. Monotonicity and low glitch are guaranteed by use of identical current sources and current steering their outputs. An on-chip operational amplifier stabilises the full-scale output current against temperature and power supply variations. The analogue outputs of the BT121 can directly drive a 37.5 Ω load, such as a doubly-terminated 75 Ω coaxial cable. The pin diagram of the BT121 videoDAC is illustrated in Figure B-16 and the pin descriptions are given in Table B-10.

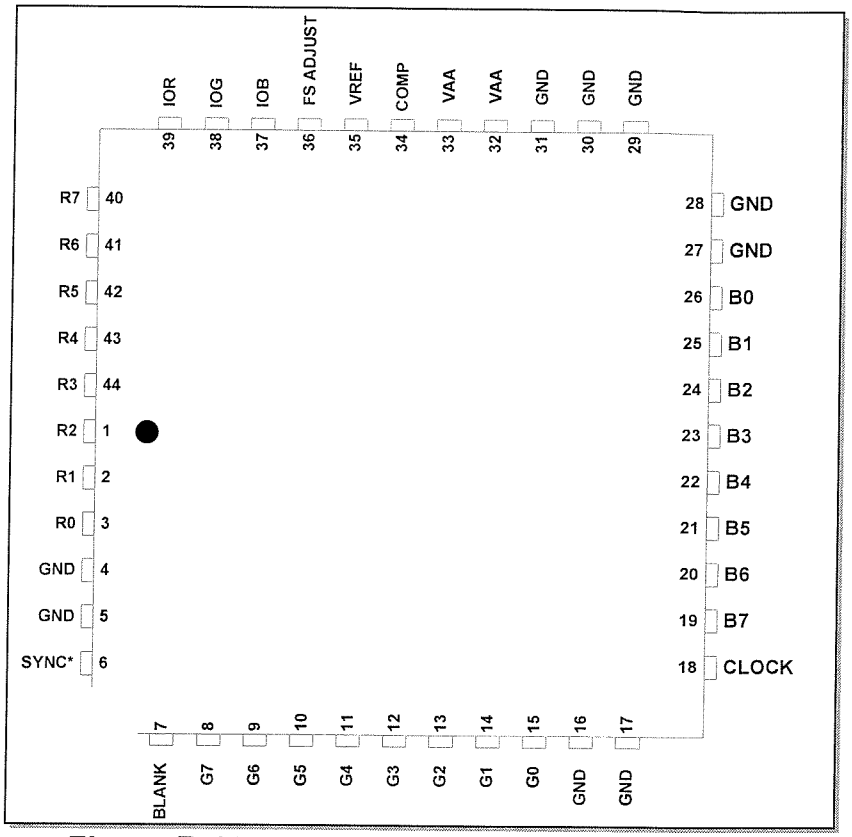


Figure B-16 Pin diagram of the BT121 videoDAC

Pin name	Description
BLANK*	Composite blank control input (TTL compatible). A logical zero drives the IOR, IOG, and IOB outputs to the blanking level. BLANK* is latched on the rising edge of CLOCK. When BLANK* is a logical zero, the R0-R7, G0-G7, and B0-B7 inputs are ignored.
SYNC*	Composite sync control input (TTL compatible). SYNC* does not override any other control or data input. SYNC* should be asserted only during the blanking interval. It is latched on the rising edge of CLOCK.
R0-R7, G0-G7, B0-B7	Red, green, and blue data inputs (TTL compatible). R0, G0, and B0 are the least-significant data bits. They are latched on the rising edge of CLOCK. Coding is binary.
CLOCK	Clock input (TTL compatible). The rising edge of CLOCK latches the R0-R7, G0-G7, B0-B7, SYNC*, and BLANK* inputs. It is typically the pixel clock rate of the video system. It is recommended that the

	CLOCK input be driven by a dedicated TTL buffer to avoid reflection-induced jitters.
IOR, IOG, IOB	Red, green, and blue current outputs. These high-impedance current sources can directly drive a doubly-terminated 75 Ω coaxial cable. All outputs, whether used or not, should have a common output load.
FS ADJUST	Full-scale adjust control. A resistor (RSET) connected between this pin and GND controls the magnitude of the full-scale video signal.
COMP	Compensation pin. This pin provides compensation for the internal reference amplifier. A 0.1 µF ceramic capacitor in series with a resistor should be connected between this pin and the nearest VAA pin (see Figure B-16) for optimum settling time. Connecting the capacitor to VAA rather than to GND provides the highest possible power supply noise rejection. The COMP resistor and capacitor must be as close to the device as possible to keep lead lengths to an absolute minimum.
VREF	Voltage reference input. The internal voltage reference is used and this pin is only connected to a 0.1 µF ceramic capacitor that decouples this input to GND.
GND	Analogue ground. All GND pins must be connected together on the same PCB plane to prevent latchup.
VAA	Analogue power. All VAA pins must be connected on the same PCB plane to prevent latchup.

Table B-10 Pin descriptions of the BT121

The typical connection diagram using the internal voltage reference is shown in Figure B-17 and the parts lists listed in Table B-11.

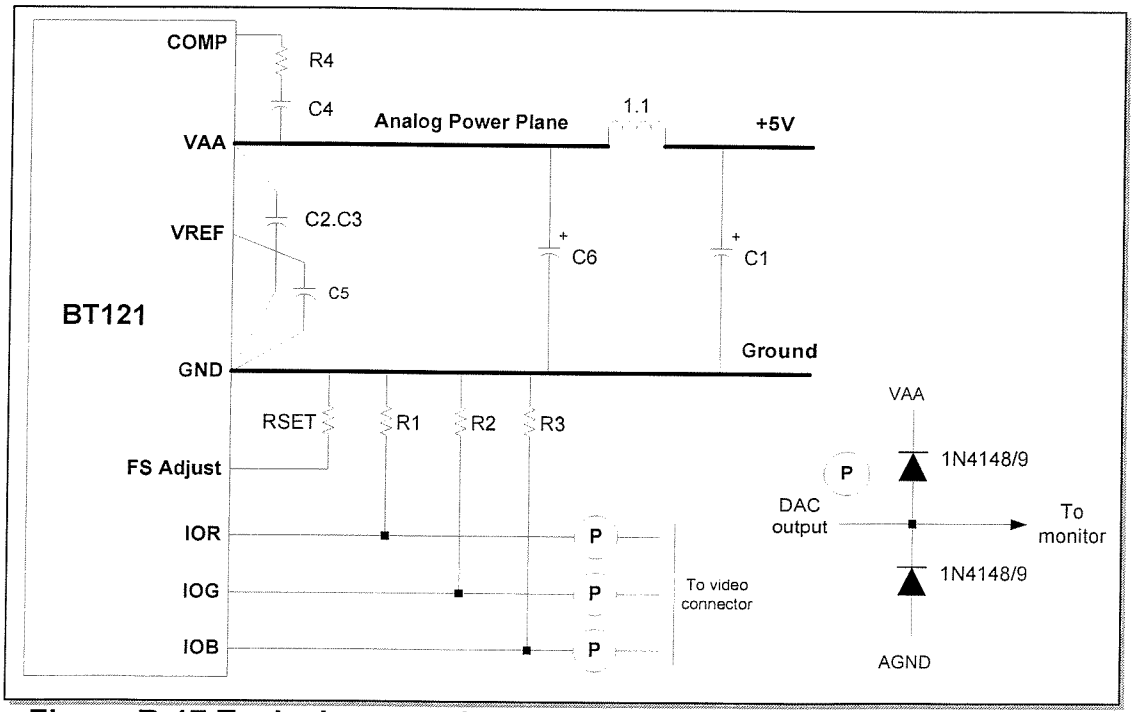


Figure B-17 Typical connection diagram with internal voltage reference

Location	Description	Vendor part number
C1	33 µF tantalum capacitor	Mallory CSR13F336KM
C2, C3, C4, C5	0.1 µF ceramic capacitor	Erie RPE112Z5U104M50V
C6	10 µF capacitor	Mallory CSR13G106KM
L1	Ferrite bead	Fair-Rite 2743001111
R1, R2, R3	75 Ω 1% metal film resistor	Dale CMF-55C
R4	15 Ω 1% metal film resistor	Dale CMF-55C
RSET	143 Ω 1% metal film resistor	Dale CMF-55C
Note: The vendor numbers above are listed only as a guide. Substitution of devices with similar characteristics will not affect the performance of the BT121.		

Table B-11 Typical connection parts list

B.6 Digilent D2-SB system board reference manual

Overview

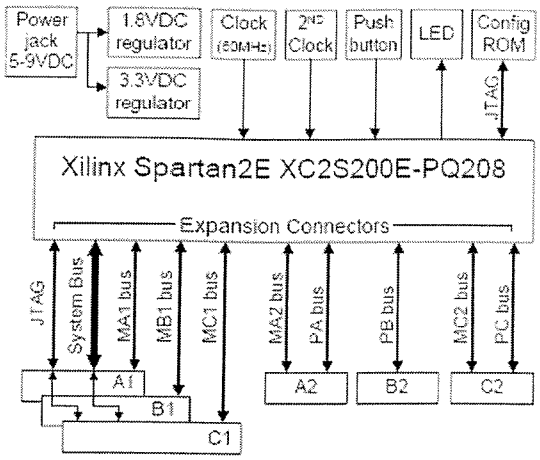
The Digilent D2-SB circuit board provides a complete circuit development platform centered on a Xilinx Spartan 2E FPGA. D2-SB features include:

- A Xilinx XC2S200E-200 FPGA with 200K gates and 350MHz operation;
- 143 user I/Os routed to six standard 40-pin expansion connectors;
- A socket for a JTAG-programmable 18V02 configuration Flash ROM;
- Dual on-board 1.5A power regulators (1.8V and 3.3V);
- An SMD 50MHz oscillator, and a socket for a second oscillator;
- A JTAG programming port;
- A status LED and pushbutton for basic I/O;

The D2-SB has been designed to work seamlessly with all versions of the Xilinx ISE CAD tools, including the free WebPack tools available from the Xilinx website. A growing collection of low-cost expansion boards can be used with the D2-SB to add analog and digital I/O capabilities, as well as various data ports like Ethernet and USB. The D2-SB board ships with a power supply and programming cable, so designs can be implemented immediately without the need for any additional hardware.

Functional Description

The Digilab D2-SB provides a minimal system that can be used to rapidly implement FPGA based circuits, or to gain exposure to Xilinx CAD tools and Spartan 2E devices. The D2-SB provides only the essential supporting devices for the Spartan 2E FPGA, including clock sources and power supplies. All available I/O signals are routed to standard expansion connectors that mate with 40-pin, 100 mil spaced DIP headers available from any catalog distributor.



D2-SB circuit board block diagram

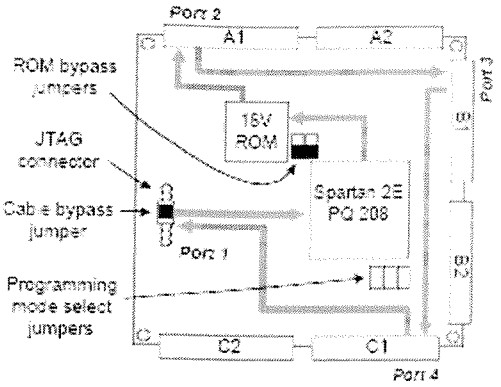
A pushbutton and LED are also included for basic I/O. The D2-SB board has been designed to serve primarily as a host for peripheral boards. Each of the six expansion connectors provides the unregulated supply voltage (VU), 3.3V, GND, and 32 FPGA I/O signals. Because there are more connector pins than FPGA pins, the A1, B1 and C1 connectors share an 18-pin "system bus", and not all pins on the B expansion connectors are used. JTAG signals are also routed to the A1, B1, and C1 expansion connectors. This allows peripheral boards to drive the scan chain, or to be configured along with the Spartan 2E FPGA. Application-specific peripheral boards can be created to mate with the D2-SB, or readymade peripheral boards that offer many standard functions can be obtained from Digilent (see www.digilentinc.com).

JTAG Ports and Device Configuration

The Spartan 2E FPGA and the 18V00 ROM on the D2-SB, and any programmable devices on peripheral boards attached to the D2-SB can be programmed via the JTAG port. The JTAG scan chain is routed to the FPGA and ROM on the D2-SB and then around the board to four connection ports as

shown in the figure below. The primary configuration port (Port 1) uses a standard 6-pin JTAG header (J7) that can accommodate Digilent's JTAG3 cable (or cables from Xilinx or other vendors). The other three JTAG programming ports are available on the A1, B1, and C1 expansion connectors, and these ports are bi-directional. If no peripheral board is present, a buffer on the D2-SB removes the expansion connector from the JTAG chain. If a peripheral board with a JTAG device is attached, the scan chain is driven out the expansion connector so that any JTAG programmable parts can be configured. If a Digilent port module is connected to one of the three JTAG-enabled expansion connectors, then the port module can drive the JTAG chain to program all devices in the scan chain (port modules include Ethernet, USB, EPP parallel, and serial modules -- see www.digilentinc.com for more information).

The scan chain can be driven from the primary port by powering on the D2-SB, connecting it to a PC with a JTAG programming cable, and running the "auto-detect" feature of the configuration software. The configuration software allows devices in the scan chain to be selectively programmed with any available configuration file. If no programming ROM is loaded in the IC5 socket (or if ROM is present but is not to be included in the scan chain), jumper-shunts must be loaded at JP1 and JP2 in the "Bypass ROM" location to route the JTAG chain around the ROM socket. If an 18V02 (or larger) ROM is loaded in the IC5 socket, it can be included in the scan chain by loading the JP1 and JP2 jumper-shunts in the "Include ROM" positions. If a programming ROM is present in the IC5 socket, the FPGA will automatically access the ROM for configuration data if jumper shunts are loaded in all three positions of J8 (M2, M1, and M0). Port modules attached to ports A1, B1, or C1 can drive the scan chain if a jumper-shunt is installed on the primary JTAG header across the TDI and TDO pins. In their default state, Digilent port modules will appear as a JTAG cable to the configuration software. Port modules can disable their JTAG drivers; if more than one JTAG driver is enabled on the scan chain, programming may fail.



JTAG signal routing on D2-SB

Power Supplies

The D2-SB board uses two LM317 voltage regulators to produce a 1.8VDC supply for the Spartan 2E core, and 3.3VDC supply for the I/O ring. Both regulators have good bypass capacitance, allowing them to supply up to 1.5A of current with less than 50mV of noise (typical). Power can be supplied from a lowcost wall transformer supply. The external supply must use a 2.1mm center-positive connector, and it must produce between 6VDC and 12VDC of unregulated voltage. The D2-SB uses a four layer PCB, with the inner layers dedicated to VCC and GND planes. Most of the VCC plane is at 3.3V, with an island under the FPGA at 1.8V. The FPGA and the other ICs on the board all have 0.047uF bypass capacitors placed as close as possible to each VCC pin. Total board current is dependant on FPGA configuration, clock frequency, and external connections. In test circuits with roughly 50K gates routed, a 50MHz clock source, and a single expansion board attached (the DIO5 board), approximately 200mA +/- 30% of supply current is drawn from the 1.8V supply, and approximately 200mA +/- 50% is drawn from the 3.3V supply. These currents are strongly dependent on FPGA and peripheral board configurations. All FPGA I/O signals use

the VCCO voltage derived from the 3.3V supply. If other VCCO voltages are required, the regulator output can be modified by changing R12 according to:

VCCO = 1.25(1 + R12/R11).

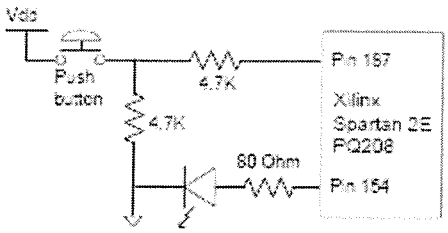
Refer to the LM317 data sheet and D2-SB schematic for further information.

Oscillators

The D2-SB provides a 50MHz SMD primary oscillator and a socket for a second oscillator. The primary oscillator is connected to the GCK2 input of the Spartan 2E (pin 182), and the secondary oscillator is connected to GCK3 (pin 185). Both clock inputs can drive the DLL on the Spartan 2E, allowing for internal frequencies up to four times higher than the external clock signals. Any 3.3V oscillator in a half-size DIP package can be loaded into the secondary oscillator socket.

Pushbutton and LED

A single pushbutton and LED are provided on the board allowing basic status and control functions to be implemented without a peripheral board. As examples, the LED can be illuminated from a signal in the FPGA to verify that configuration has been successful, and the pushbutton can be used to provide a basic reset function independent of other inputs. The circuits are shown below.



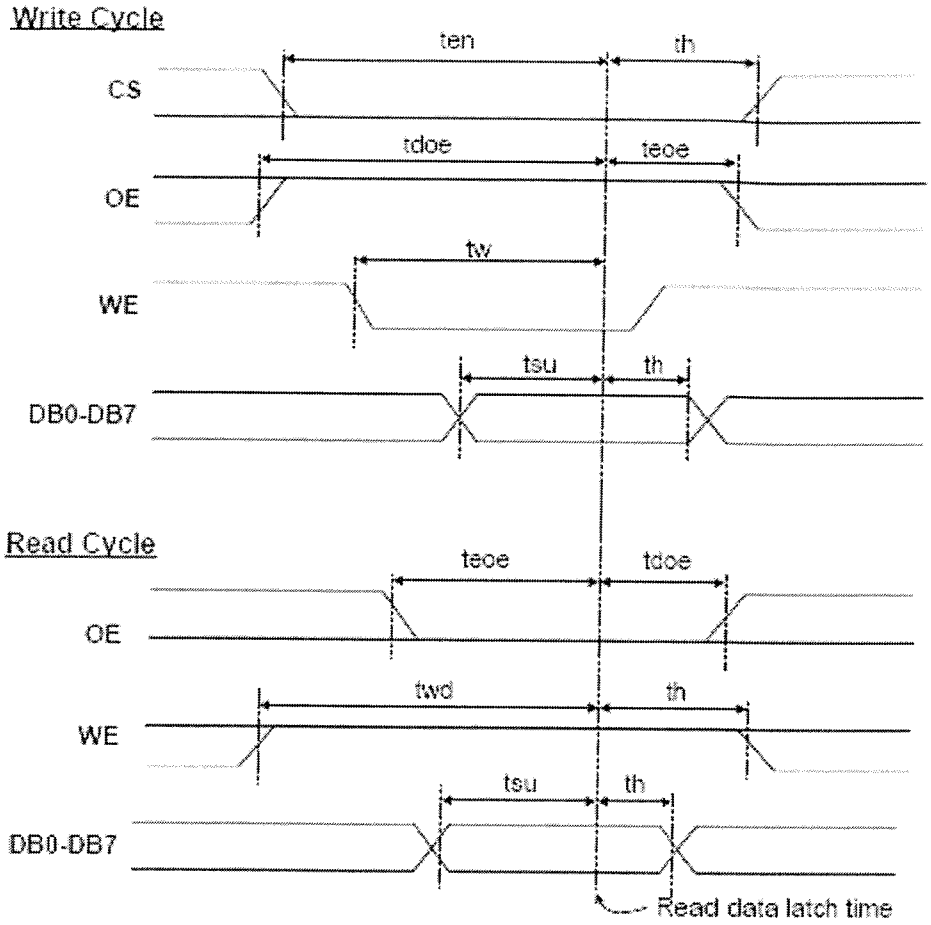
Expansion Connectors

The six expansion connectors labeled A1-A2, B1-B2, and C1-C2 use 2x20 right-angle headers with100 mil spacing. All six connectors have GND on pin 1, VU on pin 2, and 3.3V on pin 3. Pins 4-35 route to FPGA I/O signals, and pins 36-40 are reserved for JTAG and/or clock signals. The expansion headers provide 192 signal connections, but the Spartan 2E-PQ208 has only 143 available I/O signals. Thus, some FPGA signals are routed to more than one connector. In particular, the lower 18 pins (pins 4-21) of the A1, B1, and C1 connectors are all connected to the same 18 FPGA pins, and they are designated as the "system bus" (a unique chip select signal is routed to each connector). Other than these 18 shared signals, all remaining FPGA signals are routed to individual expansion connector positions. The lower 18 pins of the A2, B2, and C2 connectors are designated as "peripheral busses", and each of these busses (named PA, PB, and PC) use 18 unique signals. The 14 upper pins of each expansion connector (pins 22-35) have been designated as "module busses". The A1, A2, C1, and C2 connectors each have fully populated module busses (named MA1, MA2, MC1, and MC2). Insufficient FPGA pins were available to route full module buses to the B connectors; only the 8 data pins of MB1 are routed, and no pins are routed to the upper B2 expansion connector (i.e., MB2 is a "no connect").

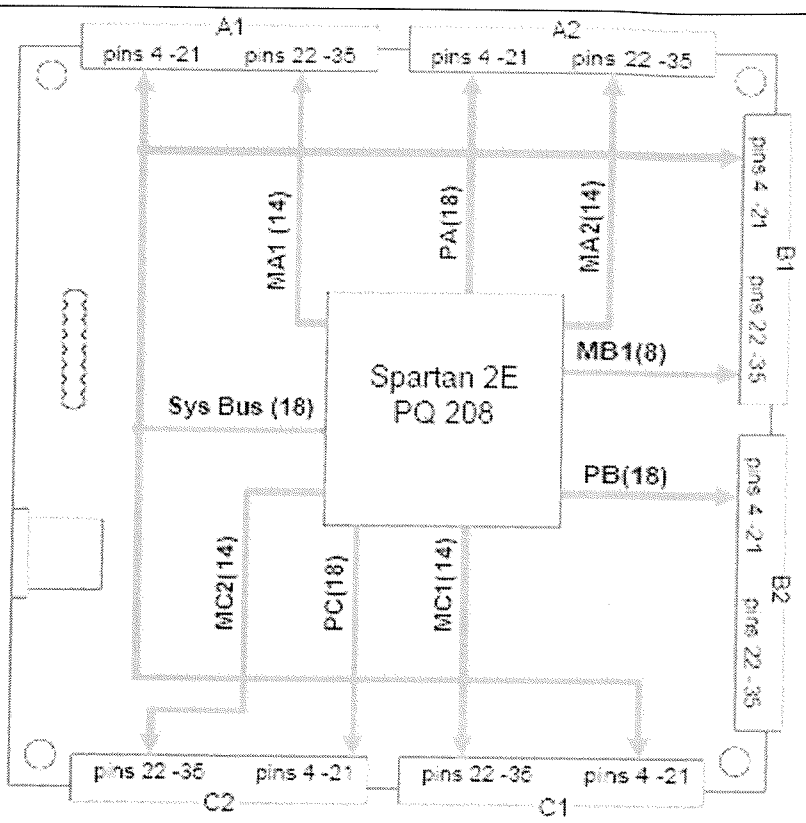
System Bus

The "system bus" is a protocol used by certain expansion boards that mimics a simple 8-bit microprocessor bus. It uses eight data lines, six address lines, a write-enable (WE) strobe that can be used by the peripheral to latch written data, an output-enable (OE) strobe that can be used by the peripheral to enable read data, a chip select, and a clock to enable synchronous transfers.

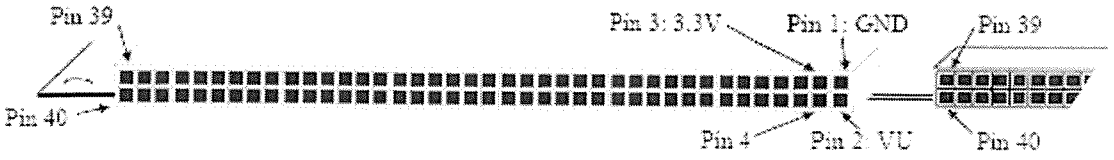
The diagrams below show signal timings assumed by Digilent to create peripheral devices. However, any bus and timing models can be used by modifying circuits in the FPGA and attached peripheral devices.



Symbol	Parameter	Time (typ)
ten	Time to enable after CS asserted	10 ns
th	Hold time	1 ns
tdoe	Time to disable after OE de-asserted	10 ns
teoe	Time to enable after OE asserted	15 ns
tw	Write strobe time	10 ns
tsu	Data setup time	5 ns
twd	Write disable time	0 ns



Expansion Connector Signal Routing



Expansion connector pin locations

Pin #	A1		A2		B1		B2		C1		C2	
	Signal	FPGA Pin	Signal	FPGA Pin	Signal	FPGA Pin	Signal	FPGA Pin	Signal	FPGA Pin	Signal	FPGA Pin
1	GND		GND		GND		GND		GND		GND	
2	VU		VU		VU		VU		VU		VU	
3	VCC33		VCC33		VCC33		VCC33		VCC33		VCC33	
4	ADR0	112	PAI01	162	ADR0	112	PBI01	71	ADR0	112	PCI01	23
5	DB0	111	PAI02	161	DB0	111	PBI02	70	DB0	111	PCI02	22
6	ADR1	110	PAI03	160	ADR1	110	PBI03	69	ADR1	110	PCI03	21
7	DB1	109	PAI04	152	DB1	109	PBI04	68	DB1	109	PCI04	20
8	ADR2	108	PAI05	151	ADR2	108	PBI05	64	ADR2	108	PCI05	18
9	DB2	102	PAI06	150	DB2	102	PBI06	63	DB2	102	PCI06	17
10	ADR3	101	PAI07	149	ADR3	101	PBI07	62	ADR3	101	PCI07	16
11	DB3	100	PAI08	148	DB3	100	PBI08	61	DB3	100	PCI08	15
12	ADR4	99	PAI09	147	ADR4	99	PBI09	60	ADR4	99	PCI09	11
13	DB4	98	PAI010	146	DB4	98	PBI010	59	DB4	98	PCI010	10
14	ADR5	97	PAI011	145	ADR5	97	PBI011	58	ADR5	97	PCI011	9
15	DB5	96	PAI012	141	DB5	96	PBI012	57	DB5	96	PCI012	8
16	WE	95	PAI013	140	WE	95	PBI013	56	WE	95	PCI013	7
17	DB6	94	PAI014	139	DB6	94	PBI014	55	DB6	94	PCI014	6
18	OE	93	PAI015	138	OE	93	PBI015	49	OE	93	PCI015	5
19	DB7	89	PAI016	136	DB7	89	PBI016	48	DB7	89	PCI016	4
20	CSA	181	PAI017	135	CSB	88	PBI017	47	CSC	45	PCI017	3
21	LSBCLK	87	PAI018	134	LSBCLK	87	PBI018	46	LSBCLK	87	PCI018	206
22	MA1DB0	180	MA2DB0	133	MB1DB0	86			MC1DB0	44	MC2DB0	205
23	MA1DB1	179	MA2DB1	132	MB1DB1	84			MC1DB1	43	MC2DB1	204
24	MA1DB2	178	MA2DB2	129	MB1DB2	83			MC1DB2	42	MC2DB2	203
25	MA1DB3	176	MA2DB3	127	MB1DB3	82			MC1DB3	41	MC2DB3	202
26	MA1DB4	175	MA2DB4	126	MB1DB4	81			MC1DB4	40	MC2DB4	201
27	MA1DB5	174	MA2DB5	125	MB1DB5	75			MC1DB5	36	MC2DB5	200
28	MA1DB6	173	MA2DB6	123	MB1DB6	74			MC1DB6	35	MC2DB6	199
29	MA1DB7	169	MA2DB7	122	MB1DB7	73			MC1DB7	34	MC2DB7	196
30	MA1ASTB	168	MA2ASTB	121					MC1ASTB	33	MC2ASTB	194
31	MA1DSTB	167	MA2DSTB	120					MC1DSTB	31	MC2DSTB	193
32	MA1WRT	166	MA2WRT	116					MC1WRT	30	MC2WRT	192
33	MA1WAIT	165	MA2WAIT	115					MC1WAIT	29	MC2WAIT	191
34	MA1RST	164	MA2RST	114					MC1RST	27	MC2RST	189
35	MA1INT	163	MA2INT	113					MC1INT	24	MC2INT	188
36	JTSELA				JTSELB				JTSELC			
37	TMS				TMS				TMS			
38	TCK				TCK				TCK			
39	TDO		GCLK0	80	TDO				TDO		GCLK1	77
40	TDI		GND		TDI				TDI		GND	

D2-SB Expansion Connector Pinout

Pin #	Function	Pin #	Function	Pin #	Function	Pin #	Function
1	GND	53	VCCO	105	VCCO	157	TDO
2	TMS	54	M2	106	PROG	158	GND
3	PC-IO17	55	PB-IO14	107	INIT	159	TDI
4	PC-IO16	56	PB-IO13	108	ADR2	160	PA-IO3
5	PC-IO15	57	PB-IO12	109	DB1	161	PA-IO2
6	PC-IO14	58	PB-IO11	110	ADR1	162	PA-IO1
7	PC-IO13	59	PB-IO10	111	DB0	163	MA1-INT
8	PC-IO12	60	PB-IO9	112	ADR0	164	MA1-RST
9	PC-IO11	61	PB-IO8	113	MA2-INT	165	MA1-WAIT
10	PC-IO10	62	PB-IO7	114	MA2-RST	166	MA1-WRT
11	PC-IO9	63	PB-IO6	115	MA2-WAIT	167	MA1-DSTB
12	GND	64	PB-IO5	116	MA2-WRT	168	MA1-ASTB
13	VCCO	65	GND	117	GND	169	MA1-DB7
14	VCCINTT	66	VCCO	118	VCCO	170	GND
15	PC-IO8	67	VCCINT	119	VCCINT	171	VCCO
16	PC-IO7	68	PB-IO4	120	MA2-DSTB	172	VCCINT
17	PC-IO6	69	PB-IO3	121	MA2-ASTB	173	MA1-DB6
18	PC-IO5	70	PB-IO2	122	MA2-DB7	174	MA1-DB5
19	GND	71	PB-IO1	123	MA2-DB6	175	MA1-DB4
20	PC-IO4	72	GND	124	GND	176	MA1-DB3
21	PC-IO3	73	MB1-DB7	125	MA2-DB5	177	GND
22	PC-IO2	74	MB1-DB6	126	MA2-DB4	178	MA1-DB2
23	PC-IO1	75	MB1-DB5	127	MA2-DB3	179	MA1-DB1
24	MC1-INT	76	VCCINT	128	VCCINT	180	MA1-DB0
25	GND	77	GCLK1	129	MA2-DB2	181	CSA
26	VCCO	78	VCCO	130	VCCO	182	GCLK2
27	MC1-RST	79	GND	131	GND	183	GND
28	VCCINT	80	GCLK0	132	MA2-DB1	184	VCCO
29	MC1-WAIT	81	MB1-DB4	133	MA2-DB2	185	GCLK3
30	MC1-WRT	82	MB1-DB3	134	PA-IO18	186	VCCINT
31	MC1-DSTB	83	MB1-DB2	135	PA-IO17	187	BTN
32	GND	84	MB1-DB1	136	PA-IO16	188	MC2-INT
33	MC1-ASTB	85	GND	137	GND	189	MC2-RST
34	MC1-DB7	86	MB1-DB0	138	PA-IO15	190	GND
35	MC1-DB6	87	LSBCLK	139	PA-IO14	191	MC2-WAIT
36	MC1-DB5	88	CSB	140	PA-IO13	192	MC2-WRT
37	VCCINT	89	DB7	141	PA-IO12	193	MC2-DSTB
38	VCCO	90	VCCINT	142	VCCINT	194	MC2-ASTB
39	GND	91	VCCO	143	VCCO	195	VCCINT
40	MC1-DB4	92	GND	144	GND	196	VCCO
41	MC1-DB3	93	OE	145	PA-IO11	197	GND
42	MC1-DB2	94	DB6	146	PA-IO10	198	MC2-DB7
43	MC1-DB1	95	WE	147	PA-IO9	199	MC2-DB6
44	MC1-DB0	96	DB5	148	PA-IO8	200	MC2-DB5
45	CSC	97	ADR5	149	PA-IO7	201	MC2-DB4
46	PB-IO18	98	DB4	150	PA-IO6	202	MC2-DB3
47	PB-IO17	99	ADR4	151	PA-IO5	203	MC2-DB2
48	PB-IO16	100	DB3	152	PA-IO4	204	MC2-DB1
49	PB-IO15	101	ADR3	153	DIN	205	MC2-DB0
50	M1	102	DB2	154	LED	206	PC-IO19
51	GND	103	GND	155	CCLK	207	TCK
52	M0	104	DONE	156	VCCO	208	VCCO

FPGA Pin Assignment

B.7 Digilent DIO4 peripheral board reference manual

Overview

The DIO4 circuit board provides a low-cost, ready-made source for many of the most common I/O devices found in digital systems. It can be attached to a Digilent system board to create a circuit design platform capable of hosting a wide array of circuits. DIO4 features include:

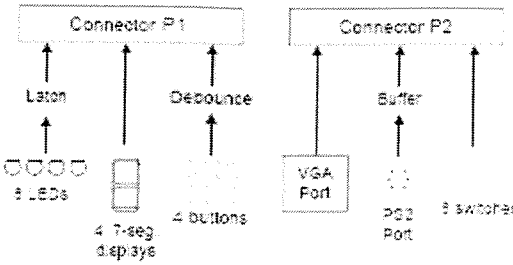
- A 4-digit seven segment LED display;
- 8 individual LEDs;
- 4 pushbuttons;
- 8 slide switches;
- 3-bit VGA port;
- PS/2 mouse or keyboard port;

Functional Description

The DIO4 can be attached to Digilent system boards to quickly and easily add several useful I/O devices. The DIO4 draws power from the system board, and signals from all I/O devices are routed to individual pins on the system board connectors. These features allow the DIO4 to be incorporated into system-board circuits with minimal effort.
All devices on the DIO4 use the 3.3V supply from the system board, except for the PS/2 port which needs a 5VDC supply (the DIO4 contains a 5VDC regulator). Signals coming from the PS/2 port are routed through level shifting buffers to protect system boards that do not have 5V tolerant inputs.

Power Supplies

The DIO4 draws power from three pins on the 40-pin connectors: pin 37 supplies 3.3V; pin 39 provides system GND, and pin 40 supplies unregulated voltage (VU). VU is connected to a 5VDC LDO regulator to produce a 5VDC supply for the PS/2 interface. The 3.3V supply is used to drive all other I/O devices on the board. The DIO4 consumes 5-10mA from the VU supply, and 10-50mA from the 3.3V supply (depending on how many LEDs are illuminated).

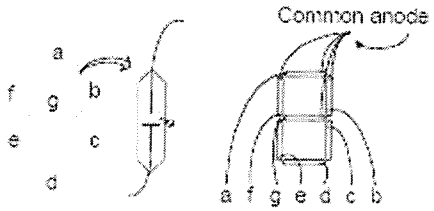


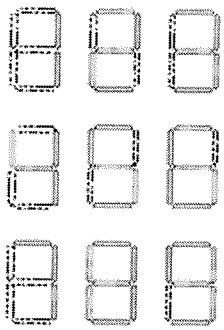
DIO4 circuit board block diagram

Seven-Segment LED display

The DIO4 board contains a modular 4-digit, common anode seven-segment LED display. In a common anode display, the seven anodes of the LEDs forming each digit are connected to four common circuit nodes (labeled AN1 through AN4 on the DIO4). Each anode, and therefore each digit, can be independently turned on and off by driving these signals to a '1' or a '0'. The cathodes of similar segments on all four displays are also connected together into seven common circuit nodes labeled CA through CG. Thus, each cathode for all four displays can be turned on and off independently. This connection scheme creates a multiplexed display, where driving the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession can create a 4-digit display. In order for each of the four digits to appear bright and continuously illuminated, all four digits should be driven once every 1 to 16ms (for a refresh frequency of 1 KHz to 60KHz). For example, in a 60Hz refresh scheme, each digit would be illuminated for ¼ of the refresh cycle, or 4ms. The controller must assure that the correct cathode pattern is present when the

corresponding anode signal is driven. To illustrate the process, if AN1 is driven high while CB and CC are driven low, then a "1" will be displayed in digit position 2. Then, if AN2 is driven high while CA, CB and CC are driven low, then a "7" will be displayed in digit position 2. If AN1 and CB, CC are driven for 4ms, and then AN2 and CA, CB, CC are driven for 4 ms in an endless succession, the display will show "17" in the first two digits. An example timing diagram is provided below. When configured with the code shown in the appendix, the CPLD on the DIO4 board implements a seven-segment controller provided a suitable clock (256Hz to 1 KHz) is provided on the SCLK pin. The controller accepts four 4-bit binary numbers in two successive registers, and decodes and displays them.

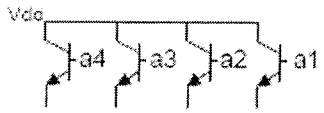




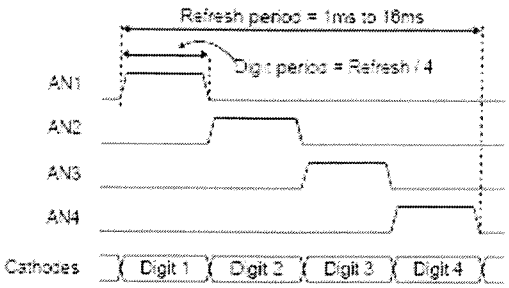
Digit Shown	Illuminated Segment a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Seven-segment display detail and cathode patterns to display the decimal digits

Anodes are connected via transistors for greater current

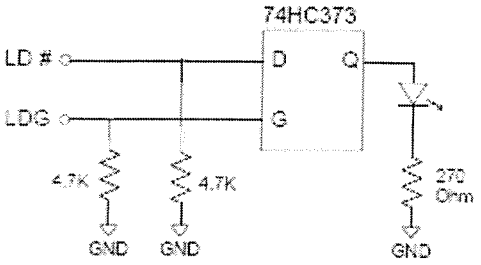


Cathodes are connected to Xilinx device via 100Ω resistors



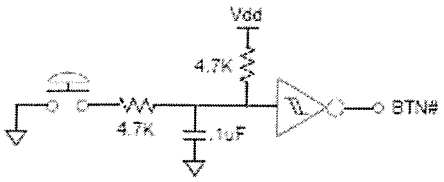
Discrete LEDs

Eight individual LEDs are provided for circuit outputs. The LED cathodes are tied to GND via 270-ohm resistors, and the LED anodes are driven from a 74HC373. The '373 allows LED data to be latched on the DIO4, so that the LD# signals from the system board do not need to be driven continuously (the LD# signals use connector pins that are used in the "system bus" on some Digilent boards). If the system bus is not needed, then the LDG signal can be tied high.



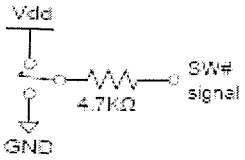
Button Inputs

The DIO4 contains 4 N.O. (normally open) pushbuttons. Button outputs are connected to Vdd via a 4.7K resistor. When the button is pressed, the output is connected directly to GND. This results in a logic signal that is low only while the button is actively pressed and high at all other times. The buttons are debounced with an RC filter and Schmitt trigger inverter as shown in the figure below. This circuit creates a logic high signal when the button is pressed. The debounce circuit provides ESD protection and creates a signal with clean edges, so the BTN# signals can be used as clock signals if desired.



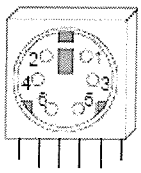
Switch Inputs

The eight slide switches on the DIO4 can be used to generate logic high or logic low inputs to the attached system board. The switches exhibit about 2ms of bounce, and no active debouncing circuit is employed. A 4.7K-ohm series resistor is used for nominal input protection.

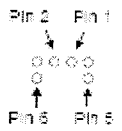


PS2 Port

The DIO4 board includes a 6-pin mini-DIN connector that can accommodate a PS2 mouse or PS2 keyboard connection. A 5VDC regulator and voltage-mapping buffers are provided on the board to interface lower voltage system boards with keyboards and/or mice.



PS2 Connector

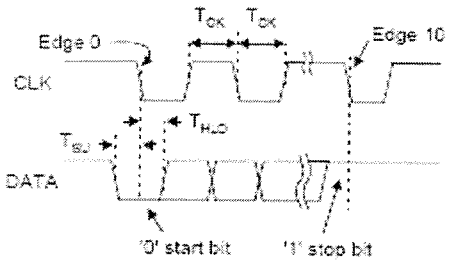


Bottom-up hole pattern

Pin Definitions

Pin	Function
1	Data
2	Reserved
3	GND
4	Vdd
5	Clock
6	Reserved

Both the mouse and keyboard use a two-wire serial bus (including clock and data) to communicate with a host device, and both drive the bus with identical signal timings. Both use 11-bit words that include a start, stop and odd parity bit, but the data packets are organized differently, and the keyboard interface allows bidirectional data transfers (so the host device can illuminate state LEDs on the keyboard). Bus timings are shown below. The clock and data signals are only driven when data transfers occur, and otherwise they are held in the "idle" state at logic '1'. The timings define signal requirements for mouse-to-host communications and bi-directional keyboard communications.



Symbol	Parameter	Min	Max
T_{CK}	Clock time	30 us	50 us
T_{SU}	Data-to-clock setup time	5 us	25 us
T_{HLD}	Clock-to-data hold time	5 us	25 us

Keyboard

The keyboard uses open collector drivers so that either the keyboard or an attached host device can drive the two-wire bus (if the host device will not send data to the keyboard, then the host can use simple input-only ports).

PS2-style keyboards use scan codes to communicate key press data (nearly all keyboards in use today are PS2 style). Each key has a single, unique scan code that is sent whenever the corresponding key is pressed. If the key is pressed and held, the scan code will be sent repeatedly once every 100ms or so. When a key is released, a "F0" key-up code is sent, followed by the scan code of the released key.

If a key can be "shifted" to produce a new character (like a capital letter), then a shift character is sent in addition to the original scan code, and the host device must determine which character to use. Some keys, called extended keys, send an "E0" ahead of the scan code (and they may send more than one scan code). When an extended key is released, an "E0 F0" key-up code is sent, followed by the scan code. Scan codes for most keys are shown in the figure below.

A host device can also send data to the keyboard. Below is a short list of some often used commands:

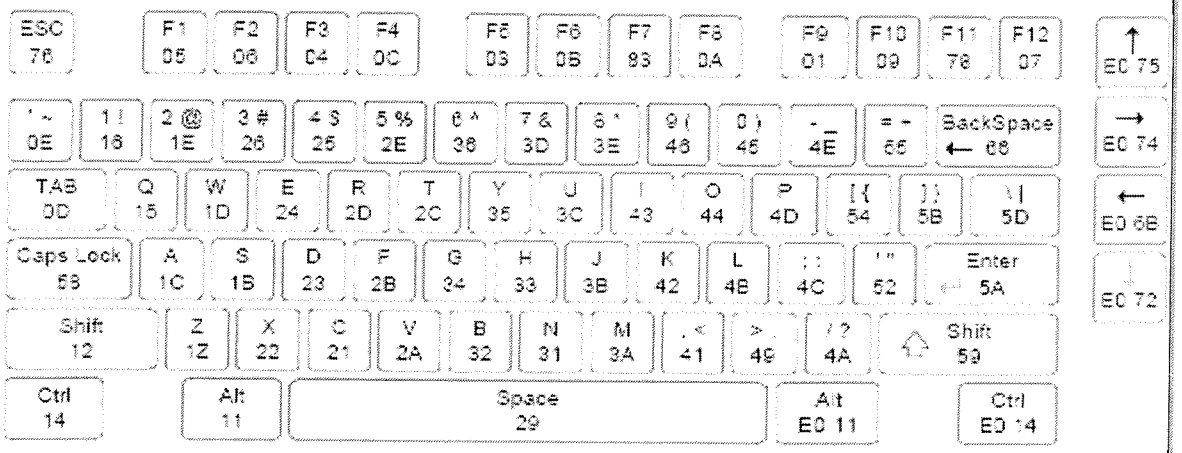
ED Set Num Lock, Caps Lock, and scroll Lock LEDs. After receiving an "ED", the keyboard returns an "FA"; then the host sends a byte to set LED status: Bit 0 sets Scroll Lock; bit 1 sets Num Lock; and Bit 2 sets Caps lock. Bits 3 to 7 are ignored.

EE Echo. Upon receiving an echo command, the keyboard replies with "EE".

F3 Set scan code repeat rate. The keyboard acknowledges receipt of an "F3" by returning an "FA", after which the host sends a second byte to set the repeat rate.

FE Resend. Upon receiving FE, the keyboard resends the last scan code sent.

FF Reset. Resets the keyboard.



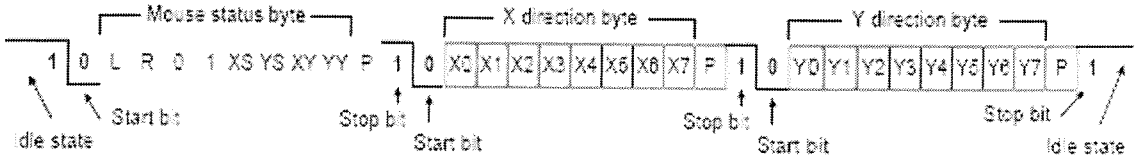
The keyboard should send data to the host only when both the data and clock lines are high (or idle). Since the host is the "bus master", the keyboard should check to see whether the host is sending data before driving the bus. To facilitate this, the clock line can be used as a "clear to send" signal. If the host pulls the clock line low, the keyboard must not send any data until the clock is released (host-to-keyboard data transmission will not be dealt with further here).
The keyboard sends data to the host in 11-bit words that contain a '0' start bit, followed by 8-bits of scan code (LSB first), followed by an odd parity bit and terminated with a '1' stop bit. The keyboard generates 11 clock transitions (at around 20 – 30 KHz) when the data is sent, and data is valid on the falling edge of the clock.

Mouse

The mouse outputs a clock and data signal when it is moved; otherwise, these signals remain at logic '1'. Each time the mouse is moved, three 11-bit words are sent from the mouse to the host device. Each of the 11-bit words contains a '0' start bit, followed by 8 bits of data (LSB first), followed by an odd parity bit, and terminated with a '1' stop bit.

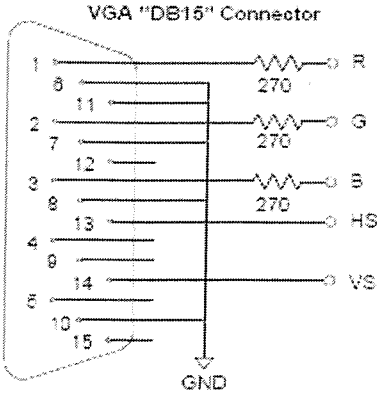
Thus, each data transmission contains 33 bits, where bits 0, 11, and 22 are '0' start bits, and bits 11, 21, and 33 are '1' stop bits.

The three 8-bit data fields contain movement data as shown below. Data is valid at the falling edge of the clock, and the clock period is 20 to 30 KHz.
The mouse assumes a relative coordinate system wherein moving the mouse to the right generates a positive number in the X field, and moving to the left generates a negative number. Likewise, moving the mouse up generates a positive number in the Y field, and moving down represents a negative number (the XS and YS bits in the status byte are the sign bits – a '1' indicates a negative number). The magnitude of the X and Y numbers represent the rate of mouse movement – the larger the number, the faster the mouse is moving (the XV and YV bits in the status byte are movement overflow indicators – a '1' means overflow has occurred). If the mouse moves continuously, the 33-bit transmissions are repeated every 50ms or so. The L and R fields in the status byte indicate Left and Right button presses (a '1' indicates the button is being pressed).



VGA Port

The five standard VGA signals Red (R), Green (G), Blue (B), Horizontal Sync (HS), and Vertical Sync (VS) are routed directly to the VGA connector. A 270-ohm series resistor is used on each color signal. This resistor forms a divider with the 75-ohm VGA cable termination, resulting in a signal that conforms to the VGA specification (i.e., 0V for fully off and .7V for fully on). VGA signal timings are specified, published, copyrighted and sold by the VESA organization (www.vesa.org).

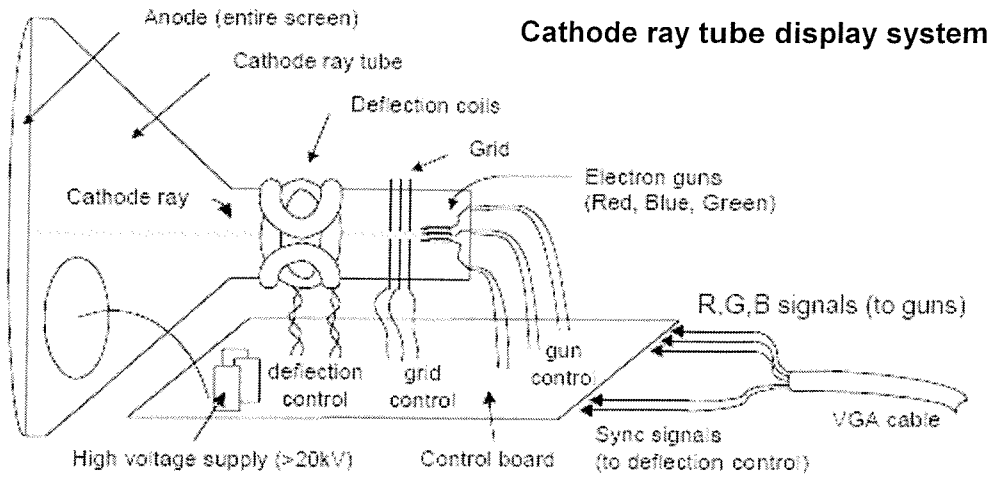


The following VGA system timing information is provided as an example of how a VGA monitor might be driven in 640 by 480 mode. For more precise information, or for information on higher VGA frequencies, refer to document available at the VESA website (or experiment!).

VGA system timing

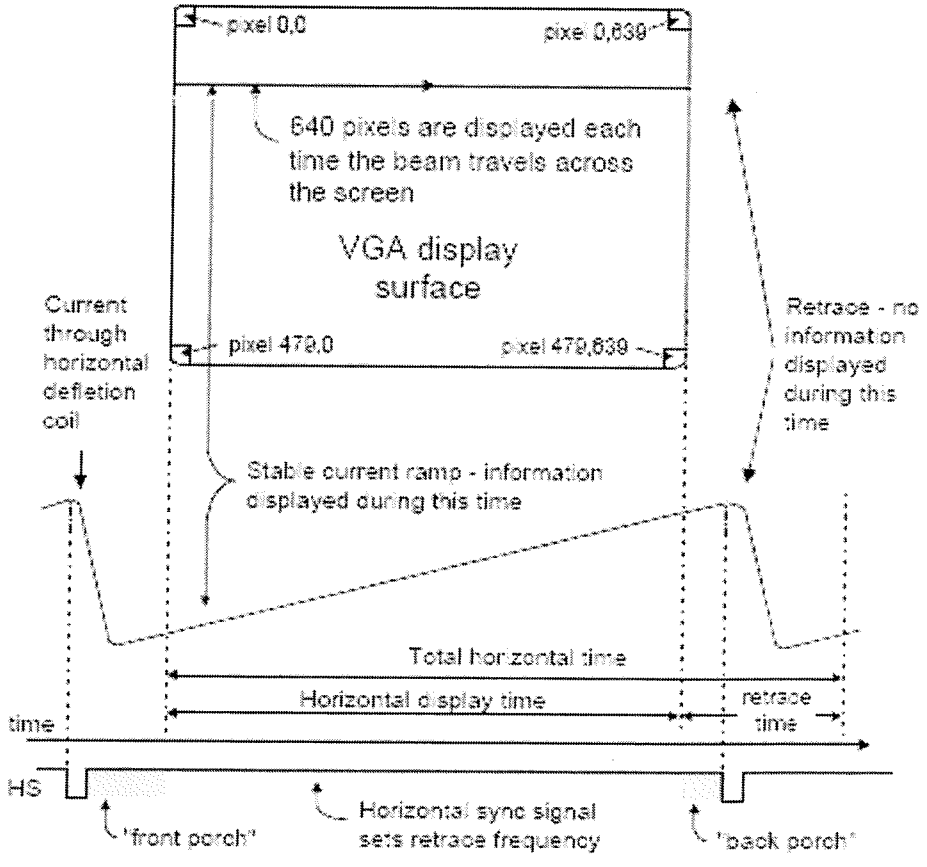
CRT-based VGA displays use amplitude modulated, moving electron beams (or cathode rays) to display information on a phosphor-coated screen. LCD displays use an array of switches that can impose a voltage across a small amount of liquid crystal, thereby changing light permitivity through the crystal on a pixel-by-pixel basis. Although the following description is limited to CRT displays, LCD displays have evolved to use the same signal timings as CRT displays (so the "signals" discussion below pertains to both CRTs and LCDs).
CRT displays use electron beams (one for red, one for blue and one for green) to energize the phosphor that coats the inner side of the display end of a cathode ray tube (see drawing below). Electron beams emanate from "electron guns", which are a finely pointed, heated cathodes placed in close proximity to a positively charged annular plate called a "grid".
The electrostatic force imposed by the grid pulls away rays of energized electrons as current flows into the cathodes.

These particle rays are initially accelerated towards the grid, but they soon fall under the influence of the much larger electrostatic force that results from the entire phosphor coated display surface of the CRT being charged to 20kV (or more). The rays are focused to a fine beam as they pass through the center of the grids, and then they accelerate to impact on the phosphor coated display surface.
The phosphor surface glows brightly at the impact point, and the phosphor continues to glow for several hundred microseconds after the beam is removed. The larger the current fed into the cathode, the brighter the phosphor will glow. Between the grid and the display surface, the beam passes through the neck of the CRT where two coils of wire produce orthogonal electromagnetic fields. Because cathode rays are composed of charged particles (electrons), they can be deflected by these magnetic fields. Current waveforms are passed through the coils to produce magnetic fields that interact with the cathode rays and cause them to transverse the display surface in a "raster" pattern, horizontally from left to right and vertically from top to bottom. As the cathode ray moves over the surface of the display, the current sent to the electron guns can be increased or decreased to change the brightness of the display at the cathode ray impact point.



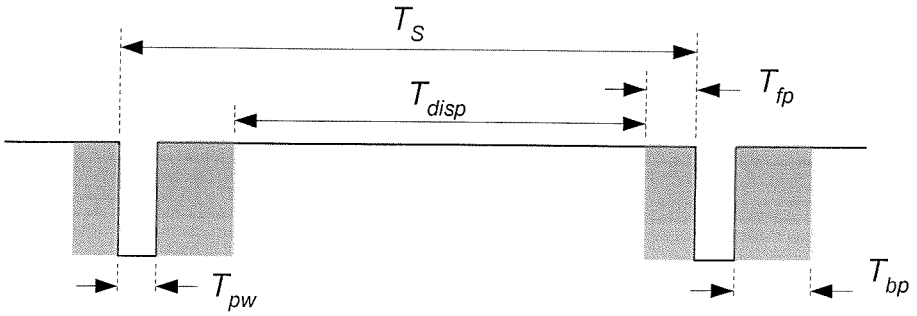
Cathode ray tube display system

Information is only displayed when the beam is moving in the "forward" direction (left to right and top to bottom), and not during the time the beam is reset back to the left or top edge of the display. Much of the potential display time is therefore lost in "blanking" periods when the beam is reset and stabilized to begin a new horizontal or vertical display pass.
The size of the beams, the frequency at which the beam can be traced across the display, and the frequency at which the electron beam can be modulated determine the display resolution. Modern VGA displays can accommodate different resolutions, and a VGA controller circuit dictates the resolution by producing timing signals to control the raster patterns. The controller must produce synchronizing pulses at 3.3V (or 5V) to set the frequency at which current flows through the deflection coils, and it must ensure that video data is applied to the electron guns at the correct time.
Raster video displays define a number of "rows" that corresponds to the number of horizontal passes the cathode makes over the display area, and a number of "columns" that corresponds to an area on each row that is assigned to one "picture element" or pixel. Typical displays use from 240 to 1200 rows, and from 320 to 1600 columns. The overall size of a display, and the number of rows and columns determines the size of each pixel.
Video data typically comes from a video refresh memory, with one or more bytes assigned to each pixel location (the DIO4 board uses 3-bits per pixel). The controller must index into video memory as the beams move across the display, and retrieve and apply video data to the display at precisely the time the electron beam is moving across a given pixel.
A VGA controller circuit must generate the HS and VS timings signals and coordinate the delivery of video data based on the pixel clock. The pixel clock defines the time available to display 1 pixel of information. The VS signal defines the "refresh" frequency of the display, or the frequency at which all information on the display is redrawn.



The minimum refresh frequency is a function of the display's phosphor and electron beam intensity, with practical refresh frequencies falling in the 50Hz to 120Hz range. The number of lines to be displayed at a given refresh frequency defines the horizontal "retrace" frequency. For a 640-pixel by 480-row display using a 25MHz pixel clock and 60 +/-1Hz refresh, the signal timings shown in the table below can be derived. Timings for sync pulse width and front and back porch intervals (porch intervals are the pre- and post-sync pulse times during which information cannot be displayed) are based on observations taken from VGA displays.
A VGA controller circuit decodes the output of a horizontal-sync counter driven by the pixel clock to generate HS signal timings. This counter can be used to locate any pixel location on a given row. Likewise, the output of a vertical-sync counter that increments with each HS pulse can be used to generate VS signal timings, and this counter can be used to locate any given row.
These two continually running counters can be used to form an address into video RAM. No time relationship between the onset of the HS pulse and the onset of the VS pulse is specified, so the designer can arrange the counters to easily form video RAM addresses, or to minimize decoding logic for sync pulse generation.

Symbol	Parameter	Vertical sync			Horizontal sync	
		Time	Clocks	Lines	Time	Clocks
T_S	Sync pulse time	16.7 ms	416800	521	32 us	800
T_{disp}	Display time	15.36 ms	384000	480	25.6 us	640
T_{pw}	VS pulse time	64 us	1600	2	3.84 us	96
T_{fp}	VS front porch	320 us	8000	10	640 ns	16
T_{bp}	VS back porch	928 us	23200	29	1.92 us	48



Expansion Connectors
Connector pinouts are shown below. Separately available tables show pass-through connections for the devices on the DIO4 board when it is attached to various system boards.

P1	Signal	Dir	P2	Signal	Dir
1	nc		1	nc	
2	nc		2	nc	
3	nc		3	nc	
4	nc		4	nc	
5	nc		5	nc	
6	nc		6	nc	
7	nc		7	nc	
8	nc		8	nc	
9	nc		9	nc	
10	nc		10	nc	
11	nc		11	nc	
12	nc		12	nc	
13	AN3	in	13	VS	in
14	AN4	in	14	HS	in
15	AN1	in	15	GRN	in
16	AN2	in	16	RED	in
17	BTN4	out	17	PS2D	bidi
18	BTN5	out	18	BLU	in
19	nc		19	BTN2	out
20	BTN3	out	20	PS2C	bidi
21	LED8	in	21	DP	in
22	LEDG	in	22	BTN1	out
23	LED7	in	23	CG	in
24	nc		24	SW8	out
25	LED6	in	25	CF	in
26	nc		26	SW7	out
27	LED5	in	27	CE	in
28	nc		28	SW6	out
29	LED4	in	29	CD	in
30	nc		30	SW5	out
31	LED3	in	31	CC	in
32	nc		32	SW4	out
33	LED2	in	33	CB	in
34	nc		34	SW3	out
35	LED1	in	35	CA	in
36	nc		36	SW2	out
37	VCC33		37	VCC33	
38	nc		38	SW1	
39	GND		39	GND	
40	VU		40	VU	

DIO4 Expansion Connector Pinout

Appendix C

File formats

This appendix explains the format of various data files used within the MOODS synthesis environment. The first is the ICODE (Intermediate CODE) generated from the VHDL compiler. Two other data files are used within the multi-FPGA partitioning process, the first is the partitioning information (*.par*) file which provides input information to the partitioning algorithm. The MOODS synthesis tool generates the second file; a module call list (*.mcl*) output file listing the call structure in the module call graph.

C.1 ICODE

The ICODE file is a textual representation of the user's design generated by the source compiler. This input file to the MOODS synthesis system is a language independent representation of the original source code, which allows the translation from other high-level languages such as (SystemC, Verilog). At present, the MOODS synthesis system only has a VHDL compiler, which converts a VHDL description into an equivalent ICODE representation.

The rest of this section provides a complete ICODE language grammar in Backus-Naur Format (BNF). Throughout this grammar, non-italicised entries refer either to other entries, or base entries. Italics are used to distinguish between different occurrences of a particular type of entry (e.g. *label*_name is a "name", *width*_number a "number"). The base entries used are:

- string – any combination of ASCII characters not including ICODE delimiters. Delimiters may be used if preceded by the escape character, e.g. \" rather than ".

- integer – a binary/decimal/octal/hexadecimal integer number
- real – floating-point number using the standard C++ formats for real numbers (including exponents).

ICODE description ::=
{ info }
program_declaration
{ submodule_declaration }
{ component_declaration }

act_list ::=
*label*_name { ',' *label*_name}

actf_list ::=
ACTF act_list

actt_list ::=
ACT act_list
| **ACTT** act_list

alias_declaration ::=
ALIAS *alias_var*_name [*alias*_range]**FROM** *parent_var*_name [*var_sub*_range]

component_declaration ::=
COMPONENT *component*_name io_list [info]

conditional_inst ::=
conditional_inst_name *cond*_var actt_list actf_list [info]

conditional_inst_name ::=
IF | **IFNOT**

constant ::=
'#' number [':' *width*_number]

declaration ::=
io_port_declaration
| variable_declaration

declaration_part ::=
{ declaration [info] }

decode_inst ::=
DECODE *decode*_var [info]
{ **CASE** constant actt_list [info] }
ENDCASE

file_info ::=
ln ':' *decimal*_integer
| **pos** ':' *decimal*_integer
| **file** ':' *decimal*_integer

filemap_info ::=
filemap ':' *decimal*_integer *filename*_string

general_inst[1] ::=
general_inst_name io_list [actt_list] [info]

general_inst_name ::=
NOOP | MOVE | UEXT | SEXT | CONCAT
| UNOT | UAND | UOR | UNAND | UNOR | UXOR | UXNOR
| SNOT | SAND | SOR | SNAND | SNOR | SXOR | SXNOR
| UEQ | UNEQ | ULT | ULTE | UGT | UGTE

[1] General instructions are defined in the ICODE instruction database, ICInstDB and may be enhanced as required.

| SEQ | SNEQ | SLT | SLTE | SGT | SGTE
| USLL | USRL | USLA | USRA | UROL | UROR
| SSLL | SSRL | SSLA | SSRA | SROL | SROR
| UMINUS | UADD | USUB | UMUL
| UDIV | UMOD | UREM | UINC | UDEC
| SMINUS | SADD | SSUB | SMUL
| SDIV | SMOD | SREM | SABS | SINC | SDEC

index ::=
*decimal*_integer

info ::=
'{' info_specification { ',' info_specification } '}'

info_specification ::=
probability_info
| iteration_info
| filemap_info
| file_info

instruction ::=
general_inst
| memory_inst
| conditional_inst
| switch_inst
| protect_inst
| decode_inst
| moduleap_inst

instruction_part ::=
{ ['.' *label*_name] instruction }

number ::=
'**%**'*binary*_integer | *decimal*_ integer | '**&**'*octal*_ integer | '**$**'*hex*_ integer

inport_declaration ::=

INPORT *io_port*_name *io_port*_range [**CLOCK** | **RESET**]

io_list ::=

term { ',' term }

io_port_declaration ::=

inport_declaration | outport_declaration

iteration_info ::=

its ':' *decimal*_integer

memory_data ::=

'['constant { ',' constant }']'

memory_inst ::=

memory_read_inst | memory_write_inst

memory_read_inst ::=

MEMREAD *memory*_var_name '[' *address*_term '],' *read*_var_name [info]

memory_write_inst ::=

MEMWRITE *write*_term ',' *memory*_var_name ',[' *address*_term ']' [info]

moduleap_inst ::=

MODULEAP *module*_name io_list [actt_list] [info]

name ::=

string

outport_declaration ::=

OUTPORT *io_port*_name *io_port*_range [**INIT** constant]

probability_info ::=

pt | **pf** '**:**' real

program_declaration ::=

PROGRAM *program*_name io_list [actt_list] [info]

declaration_part

instruction_part

ENDMODULE [*program*_name] [info]

protect_instruction ::=

PROTECT real [actt_list]

ram_declaration ::=

RAM *ram_var*_name *data*_range **ADDRESS** *address*_range

range ::=

'**[**' *msb*_index '**:**' *lsb*_ index '**]**'

register_declaration ::=

REGISTER *var*_name *var*_range [**INIT** constant]

rom_declaration ::=

ROM *rom_var*_name *data*_range **ADDRESS** *address*_range **DATA** memory_data

submodule_declaration ::=

MODULE *module*_name io_list [actt_list] [info]

declaration_part

instruction_part

['.' *label*_name] **ENDMODULE** [*module*_name] [info]

switch_inst ::=

SWITCHON *switch*_var [info]

{ **CASE** constant actt_list [info] }

DEFAULT actt_list [info]

ENDCASE

term ::=
constant | var

var ::=
*var*_name

variable_declaration ::=
register_declaration
| alias_declaration
| ram_declaration
| rom_declaration

Notes :

- Each entry is considered to occupied one line unless extended using '\'.
- Comments may be included using the standard C++ delimiter '\\'.
- Most instructions are defined in the ICODE database (ICInstDB), which also specifies the exact format of their parameters lists.
- CASES in DECODES must be in sequential ascending order with no gaps within the sequence. Any missing cases at the start or end of the sequence default to the first choice.
- Info entries may contain any form of application-dependent information such as source line numbers, variables etc. Syntactically, everything within the braces is ignored (although the key entries are identified in the BNF). In MOODS, info records specify instruction activation probabilities ("pt", "pf"), loop iterations ("its"), file mappings ("filemap") and back annotation information ("file", "ln", "pos").

C.2 Partitioning information (*.par*) file

The partitioning information (*.par*) file is an input file to the MOODS synthesis system. The file format of the partitioning information file is similar to the standard Microsoft initialisation (*.ini*) file.

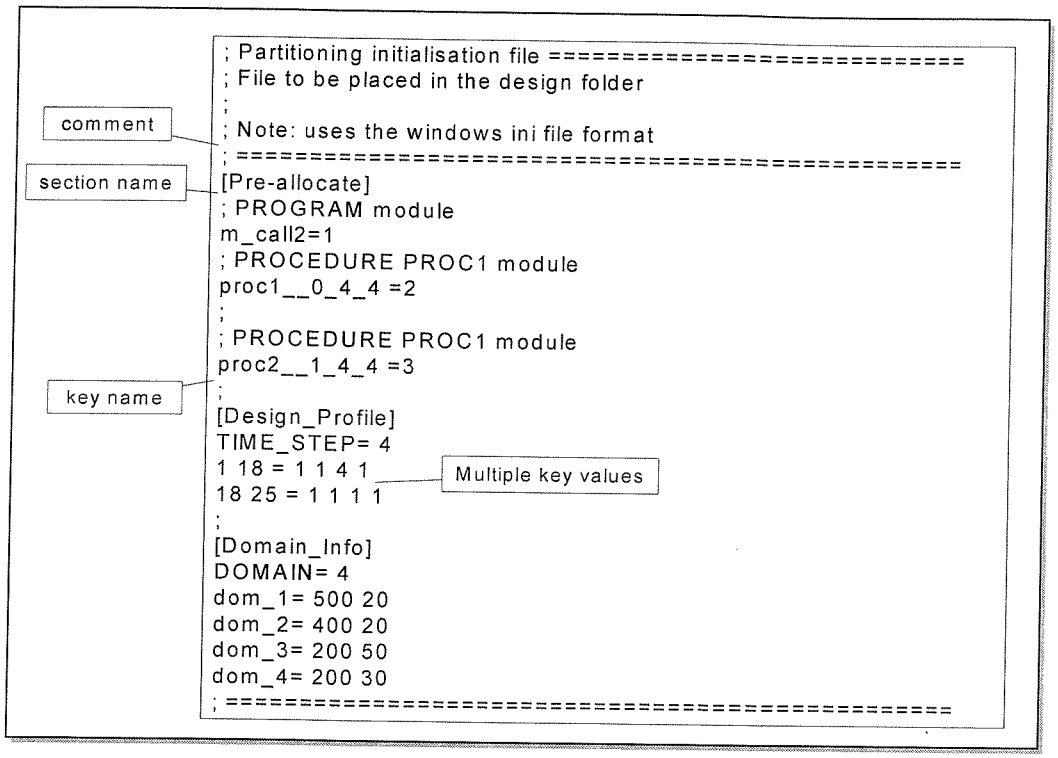


Figure C-1 Partitioning information (*.par*) file

Section names are enclosed in square brackets and the items under it are related to that section. The next lines are broken into two parts: the *key name* and the *key value*(s). Multiple values for a key are separated by a space and comments are introduced by a semicolon character. This input file provides various types of data to the K-way partitioner and these are grouped under different section headers listed below:

- **[Module_lock]** – Items under this section header are module name (key name) and the domain number (key value) that the module is locked to during K-way

partitioning. This allows manual assignment of design modules to a fixed domain. This feature is useful in assigning modules that needs special peripheral devices on a target device PCB board (e.g. a VGA connector, external memory modules).

- **[Pre-allocate]** – Items under this section are similar to the ones mentioned above, where the key names are module names in the design but the key value under this section header are the initial domains that the modules are assigned to. This forms the starting partition of the K-way partitioning algorithm.

- **[Design_Profile]** – The items under this section header give the design activity profiling information. The first key name under this section header is *TIME_STEP* and the key value gives the number of time steps in each profile data. The next lines are the profile data and these are made up of the source-destination module node numbers as the key names, and the key values are made up of activation count values with a space between each time step. The activation count value is the number of times the source module calls (or activates) the destination module (e.g. Figure C-1 illustrates a design profile with 4 time steps. Module 1 calls module 18 four times in time step 3 and only once in time steps 1, 2, and 4.).

- **[Domain_Info]** – The domain info section contains information on the target devices available for the multi-FPGA system. The first key name under this section header is *DOMAIN* and the key value gives the number of devices available. The next lines give the available area and I/O resources for each device. The first key value gives the area available, and the second key value gives the I/O resources available for device *n* denoted by the key name, dom_*n*.

C.3 Module call list (*.mcl*) file

The module call list (*.mcl*) file is an output file generated by the MOODS synthesis system and it lists all the subprogram module calls in a design. The module node numbers of the source and destination modules are used to identify modules with subprogram calls when

simulating a design to obtain the design activity information using ModelSim simulation package.

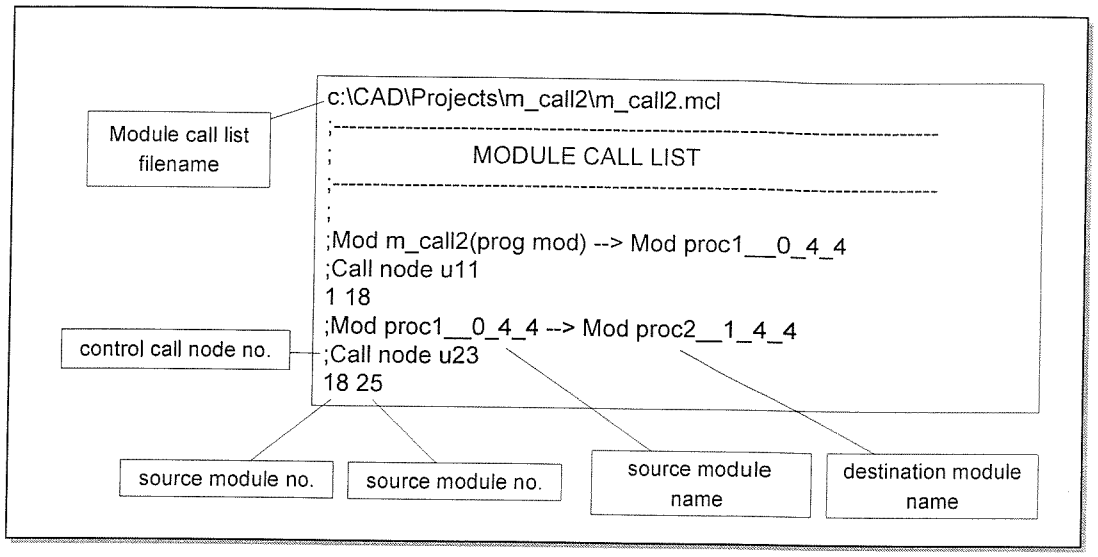


Figure C-2 Module call list (.mcl) file

Appendix D

VHDL code listings

This appendix gives a complete listing of all the example VHDL designs used in the experiments conducted in Chapter 6. The VHDL codes for the hardware demonstrator have been omitted from this appendix due to its size (the behavioural VHDL of the JPEG decoder is approximately 2000 lines of codes).

This appendix gives some background and idea of the complexity and implementation methods for the example VHDL designs. Post-MOODS synthesis simulation results of the multi-FPGA implementations are included for all the example designs.

D.1 Behavioural VHDL example designs

The five behavioural VHDL examples given in this section are used in experiments (without explicit communication channels) described in Section 6.2. All the VHDL packages which contain the definitions of constants, types, signals, functions, and procedures are also included.

D.1.1 Quadratic equation solver

The design solves quadratic equations using the formula of Equation D.1. The 32-bit, fixed-point quadratic equation solver example given in Figure D-3 uses the integer-maths library given in Figure D-1 and the quadratic procedure in the VHDL package given in Figure D-2.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad \text{(D.1)}$$

```
--************************************ --
--  Integer-maths library package  --
--************************************ --
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package c_types is
  -- c style integer and unsigned types
  subtype int is signed(31 downto 0);
  subtype uint is unsigned(31 downto 0);

  function to_int(arg: integer) return int;
end c_types;

package body c_types is
  function to_int
  -- moods inline
  (  arg: integer
  )  return int is
  begin
    return to_signed(arg,32);
  end to_int;
end c_types;

use work.c_types.all;
package imath is

  -- simple constants
  constant neg: boolean := false;
  constant pos: boolean := true;

  -- constants for acosi function
  constant acos_x0: int := X"00000000";
  constant acos_x1: int := X"00003333";
  constant acos_x2: int := X"00006666";
  constant acos_x3: int := X"00009999";
  constant acos_x4: int := X"0000CCCC";
  constant acos_x5: int := X"0000FFFF";
  constant acos_y0: int := X"00019220";
  constant acos_y1: int := X"00015E94";
  constant acos_y2: int := X"000128C7";
  constant acos_y3: int := X"0000ED63";
  constant acos_y4: int := X"0000CCCD";
  constant acos_y5: int := X"00000000";

  -- constants for cosi function
  constant s2pi:    int := X"0006487E";
  constant spi_2:   int := X"0001921F";
  constant spi:     int := X"0003243F";
  constant s3pi_2:  int := X"0004B65F";
  constant cos_x0:  int := X"00000000";
  constant cos_x1:  int := X"0000506D";
  constant cos_x2:  int := X"0000A0D9";
  constant cos_x3:  int := X"0000F146";
  constant cos_x4:  int := X"000141B3";
  constant cos_x5:  int := X"00019220";
  constant cos_y0:  int := X"0000FFFF";
  constant cos_y1:  int := X"0000F378";
  constant cos_y2:  int := X"0000CF1C";
  constant cos_y3:  int := X"00009679";
  constant cos_y4:  int := X"00004F1B";
  constant cos_y5:  int := X"00000000";
```

```
  -- integer cubed root function
  function cbrti(a: in int) return int;
  -- integer square rooot function
  function sqrti(a: in int) return int;
  -- integer arccos function (inputs and outputs scaled by 65536)
  function acosi(a: in int) return int;
  -- integer cosine function (inputs and outputs scaled by 65536)
  function cosi(a: in int) return int;
  -- signed integer division
  function sdivi(a: in int; b: in int) return int;
  -- unsigned integer divide
  function udivi(a: in uint; b: in uint) return uint;
  -- sign test
  function sign(x: in int) return boolean;
  -- to_bool conversion
  function to_bool(a: in std_logic) return boolean;
  -- moods map move u:1 u:1
  -- signed sqi
  function sqi(a: in int) return int;
  -- signed cbi
  function cbi(a: in int) return int;
  -- signed multi
  function multi(a,b: in int) return int;
  -- unsigned sqi
  function sqi(a: in uint) return uint;
  -- unsigned cbi
  function cbi(a: in uint) return uint;
  -- unsigned multi
  function multi(a,b: in uint) return uint;
end imath;

package body imath is

 -- integer cubed root function
 function cbrti
 ( a: in int
 ) return int is
   variable mask: int := X"00000400";
   variable best: int := X"00000000";
   variable sb: boolean;
   variable a_int: int;
 begin
   -- a simple test for basic solutions
   if(a=0 or a=-1 or a=1) then return a; end if;
   if(a<0) then
     sb := neg;
     a_int := -a;
   else
     sb := pos;
     a_int := a;
   end if;

   while (mask /= 0) loop
    if (cbi(best+mask) <= a_int) then
      best := best or mask;
    end if;
    mask := mask srl 1;
   end loop;

   if(not sb) then
     best := -best;
   end if;

   return best;
 end cbrti;
--============================================================--
```

```
    -- integer square root function
  function sqrti
  ( a: in int
  )  return int is
    variable mask: int := X"00008000";
    variable best: int := X"00000000";
    variable sb: boolean;
    variable a_int: int;
  begin
    if (a <= 0) then return best; end if;

    while(mask /= 0) loop
      if (((best+mask)*(best+mask)) <= a) then
        best := best or mask;
      end if;
      mask := mask srl 1;
    end loop;

    return best;
  end sqrti;
--=====================================================================--
  -- integer arccos function (inputs and outputs scaled by 65536)
  function acosi
  ( a: in int
  ) return int is
    variable sb: boolean;
    variable a_int: int;
    variable x0,x1,y0,y1,y0b,y1b: int;
    variable result: int;
  begin
    if(a<0) then
      a_int := -a;
      sb := neg;
    else
      a_int := a;
      sb := pos;
    end if;

    if (a_int<acos_x1) then
      x0 := acos_x0;
      x1 := acos_x1;
      y0 := acos_y0;
      y1 := acos_y1;
    elsif (a_int<acos_x2) then
      x0 := acos_x1;
      x1 := acos_x2;
      y0 := acos_y1;
      y1 := acos_y2;
    elsif (a_int<acos_x3) then
      x0 := acos_x2;
      x1 := acos_x3;
      y0 := acos_y2;
      y1 := acos_y3;
    elsif (a_int<acos_x4) then
      x0 := acos_x3;
      x1 := acos_x4;
      y0 := acos_y3;
      y1 := acos_y4;
    else
      x0 := acos_x4;
      x1 := acos_x5;
      y0 := acos_y4;
      y1 := acos_y5;
    end if;
```

```
    y0b := shift_left(y0,8);
    y1b := shift_left(y1,8);
    result := shift_right(y0b + multi(sdivi(y1b-y0b,x1-x0),(a_int-x0)),8);

    if(sb=neg) then
      result := X"0003242F" - result;
    end if;

    return result;
  end acosi;
--=====================================================================--
  -- integer cosine function (inputs and outputs scaled by 65536)
  function cosi
  ( a: in int
  ) return int is
    variable sb: boolean;
    variable a_int: int;
    variable temp: int;
    variable x0,x1,y0,y1,y0b,y1b: int;
    variable result: int;
  begin
    if (a<0) then
      a_int := -a;
    else
      a_int := a;
    end if;

    if(a_int > s2pi) then
      temp := signed(udivi(unsigned(a_int),unsigned(s2pi)));
      a_int := a_int - multi(temp,s2pi);
    end if;

    if(a_int<spi_2) then
      sb := pos;
    elsif(a_int<spi) then
      a_int := spi - a_int;
      sb := neg;
    elsif(a_int<s3pi_2) then
      a_int := a_int - spi;
      sb := neg;
    else
      a_int := s2pi - a_int;
      sb := pos;
    end if;

    if(a_int < cos_x1) then
      x0 := cos_x0;
      x1 := cos_x1;
      y0 := cos_y0;
      y1 := cos_y1;
    elsif(a_int < cos_x2) then
      x0 := cos_x1;
      x1 := cos_x2;
      y0 := cos_y1;
      y1 := cos_y2;
    elsif(a_int < cos_x3) then
      x0 := cos_x2;
      x1 := cos_x3;
      y0 := cos_y2;
      y1 := cos_y3;
    elsif(a_int < cos_x4) then
      x0 := cos_x3;
      x1 := cos_x4;
      y0 := cos_y3;
      y1 := cos_y4;
```

```
    else
      x0 := cos_x4;
      x1 := cos_x5;
      y0 := cos_y4;
      y1 := cos_y5;
    end if;

    y0b := shift_left(y0,8);
    y1b := shift_left(y1,8);
    result := shift_right(y0b + multi(sdivi(y1b-y0b,x1-x0),(a_Int-x0)),8);

    if(sb=neg) then
      result := -result;
    end if;

   return result;
  end cosi;
--=====================================================================--
  -- signed integer division
  function sdivi
  ( a: in int;
    b: in int
  ) return int is
    variable sa,sb: boolean;
    variable ua,ub: int;
    variable temp: int;
  begin
    sa := sign(a);
    sb := sign(b);

    if(sa=pos) then
      ua := a;
    else ua := -a;
    end if;

    if(sb=pos) then
      ub := b;
    else ub := -b;
    end if;

    temp := signed(udivi(unsigned(ua),unsigned(ub)));

    if(sa=sb) then
      return temp;
    else return -temp;
    end if;
  end sdivi;
--=====================================================================--
  -- unsigned integer divide
  function udivi
  ( a: in uint;
    b: in uint
  ) return uint is
    variable mask: uint := X"40000000";
    variable best: uint := X"00000000";
  begin
    while(mask/=0) loop
      if((best+mask)*b <= a) then
        best := best or mask;
      end if;
      mask := mask srl 1;
    end loop;
    return best;
  end udivi;
```

```
  -- sign test
  function sign
  -- moods inline
  ( x: in int
  ) return boolean is
  begin
    return not to_bool(x(31));
  end sign;
--=================================================================--
    -- to_bool conversion
  function to_bool
  -- moods map move u%1 u%1
  ( a: in std_logic
  ) return boolean is
  begin
    if(a='1') then return true;
    else return false;
    end if;
  end to_bool;

  function sqi
  ( a: in int
  ) return int is
    variable rl: signed(63 downto 0);
  begin
    rl := a*a;
    return rl(31 downto 0);
  end sqi;
--=================================================================--
  function cbi
  ( a: in int
  ) return int is
    variable rl: signed(95 downto 0);
  begin
    rl := a*a*a;
    return rl(31 downto 0);
  end cbi;
--=================================================================--
  function multi
  ( a,b: in int
  ) return int is
    variable rl: signed(63 downto 0);
  begin
    rl := a * b;
    return rl(31 downto 0);
  end multi;
--=================================================================--
  function sqi
  ( a: in uint
  ) return uint is
    variable rl: unsigned(63 downto 0);
  begin
    rl := a*a;
    return rl(31 downto 0);
  end sqi;
  function cbi
  ( a: in uint
  ) return uint is
    variable rl: unsigned(95 downto 0);
  begin
    rl := a*a*a;
    return rl(31 downto 0);
  end cbi;
```

```
  function multi
  ( a,b: in uint
  ) return uint is
    variable rl: unsigned(63 downto 0);
  begin
    rl := a * b;
    return rl(31 downto 0);
  end multi;
end imath;
```

Figure D-1 Integer-maths library package of quadratic and cubic equation solvers

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.c_types.all;
use work.imath.all;

package algeqn_package is
  procedure quadratici(a,b,c: in int; x1,x2: out int; no_real: out int);
  procedure cubici(a1,a2,a3: in int; x1,x2,x3: out int; no_real: out int);
end algeqn_package;

package body algeqn_package is

procedure quadratici
  ---- moods inline
(
   a,b,c: in int;
   x1,x2: out int;
   no_real: out int
) is
  variable d, rd, a2 : int;
begin
  d := sqi(b) - multi(multi(to_int(4),a),c);
  a2 := multi(a,to_int(2));

  if(d < 0) then
    no_real := to_int(0);
  else
    rd := sqrti(d);
    x1 := sdivi((-b + rd),a2);
    x2 := sdivi((-b - rd),a2);
    no_real := to_int(2);
  end if;
end quadratici;
```

```
 procedure cubici
   ---- moods inline
 (
    a1,a2,a3: in int;
    x1,x2,x3: out int;
    no_real: out int
 ) is
   variable q,r,q3,d,s,a1_3,srd,t_1, t_2,theta3,t1,t2: int;
 begin
    t_1 := multi(to_int(3),a2) - sqi(a1);
    q := sdivi(t_1,to_int(9));
    t_2 := multi(multi(to_int(9),a1),a2) - multi(to_int(27),a3) - multi(to_int(2),cbi(a1));
    r := sdivi(t_2,to_int(54));

    q3 := cbi(q);
    d := q3 + sqi(r);

   if(d=0) then
     s := cbrti(r);
     a1_3 := sdivi(a1,to_int(3));
     x1 := shift_left(s,1) - a1_3;
     t1 := -s - a1_3;
     x2 := t1;
     x3 := t1;
     no_real := to_int(3);
  elsif (d >0) then
     srd := sqrti(d);
     s := cbrti(r+srd);
     t1 := cbrti(r-srd);
     x1 := s+t1-sdivi(a1,to_int(3));
     no_real := to_int(1);
  else
     theta3 := sdivi(acosi(sdivi(shift_left(r,16),sqrti(-q3))),to_int(3));
     t1 := sdivi(a1,to_int(3));
     t2 := shift_left(sqrti(-q),1);
     x1 := shift_right(multi(t2,cosi(theta3)),16)-t1;
     x2 := shift_right(multi(t2,cosi(theta3+X"00021828")),16)-t1;
     x3 := shift_right(multi(t2,cosi(theta3+X"00043050")),16)-t1;
     no_real := to_int(3);
  end if;
end cubici;
end algeqn_package;
```

Figure D-2 VHDL package of quadratic and cubic equation solvers

```
--************************************ --
-- Quadratic equation solver         --
--************************************ --
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.c_types.all;
use work.algeqn_package.all;

entity eq_solver is
  port (
    a1,a2,a3: in int;
    x1,x2: out int;
    no_real: out int
  );
end eq_solver;
architecture behaviour of eq_solver is
  begin
    process is
      variable b1: int := X"00000000";
      variable b2: int := X"00000000";
      variable b3: int := X"00000000";
      variable y1: int := X"00000000";
      variable y2: int := X"00000000";
      variable vreal: int := X"00000000";
   begin
      b1 := a1;
      b2 := a2;
      b3 := a3;
      quadratici(b1,b2,b3,y1,y2,vreal);
      x1 <= y1;
      x2 <= y2;
      no_real <= vreal;
      wait for 40 ns;
    end process;
end behaviour;
```

Figure D-3 VHDL of quadratic equation solver example

Figure D-4 shows the post-MOODS synthesis simulation of the non-pipelined multi-FPGA quadratic equation solver. This two-device implementation has a single subprogram communication channel (*SpC 1*). Integer inputs a1, a2, and a3 of the quadratic equation solver are given values 1, -25 and 150 respectively. Outputs x1, x2 and number of real numbers (no_real) are updated after 9100 ns. With a system clock period of 40 ns, the non-pipelined multi-FPGA quadratic equation solver takes 224 clock cycles (i.e. clock cycles = (9100 ns -140 ns) / 40 ns) to complete the application and output the result.

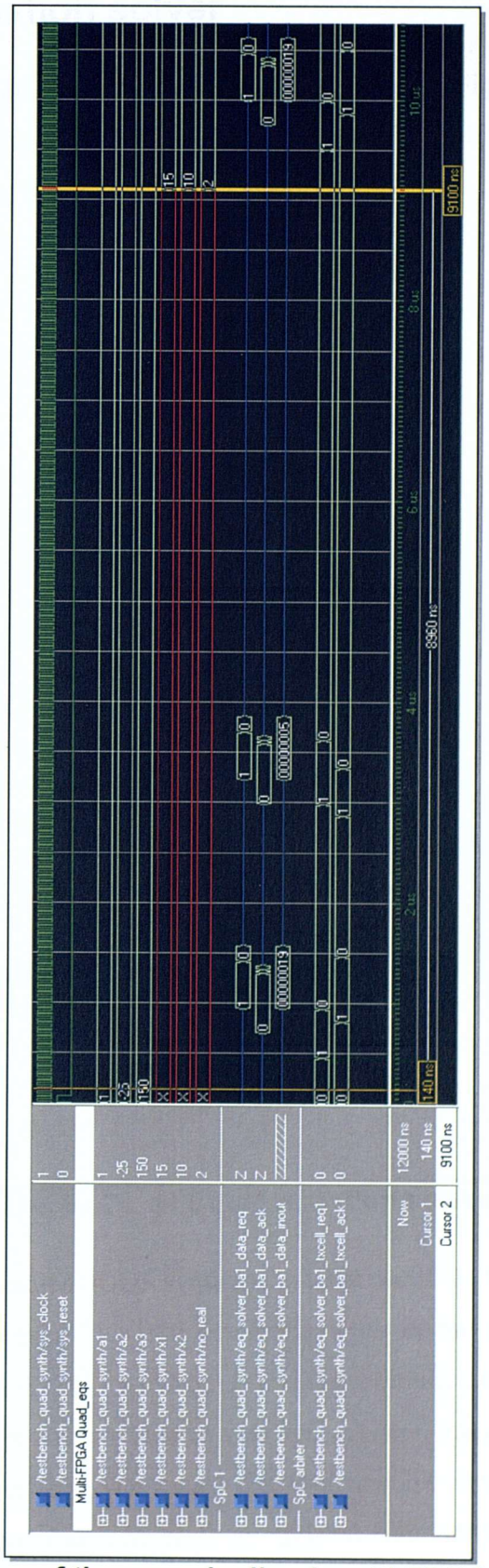


Figure D-4 Simulation of the non-pipelined multi-FPGA quadratic equation solver

D.1.2 Cubic equation solver

The 32-bit, fixed-point cubic equation solver example given in Figure D-5 is capable of finding real solutions to Equation D.2. It uses the integer-maths library given in Figure D-1 and the cubic procedure in the VHDL package given in Figure D-2.

$$x^3 + ax^2 + bx + c = 0. \tag{D.2}$$

```
--****************************** --
--  Cubic equation solver      --
--****************************** --
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.c_types.all;
use work.algeqn_package.all;
entity eq_solver is
  port (
    a1,a2,a3: in int;
    x1,x2,x3: out int;
    no_real: out int );
end eq_solver;
architecture behaviour of eq_solver is
 begin
  process is
    variable b1,b2,b3, y1,y2,y3: int;
    variable vreal: int;
  begin
    b1 := a1;
    b2 := a2;
    b3 := a3;
    cubici(b1,b2,b3,y1,y2,y3,vreal);
    x1 <= y1;
    x2 <= y2;
    x3 <= y3;
    no_real <= vreal;
    wait for 40 ns;
 end process;
end behaviour;
```

Figure D-5 VHDL of Cubic equation solver example

Figure D-6 shows the post-MOODS synthesis simulation of the non-pipelined multi-FPGA cubic equation solver. This 2-device implementation has two subprogram communication channels (*SpC 1* and *SpC 2*) and the arbitration of these two shared communication channels are provided by two SpC arbiters. Integer inputs a1, a2, and a3 of the cubic equation solver are given values -20, -100 and 2000 respectively. Outputs x1, x2, x3 and number of real numbers (no_real) are updated after 70900 ns. With a system clock period of 40 ns, the non-pipelined multi-FPGA cubic equation solver takes 1770 clock cycles (i.e. clock cycles = (70900 ns -100 ns) / 40 ns) to complete the application.

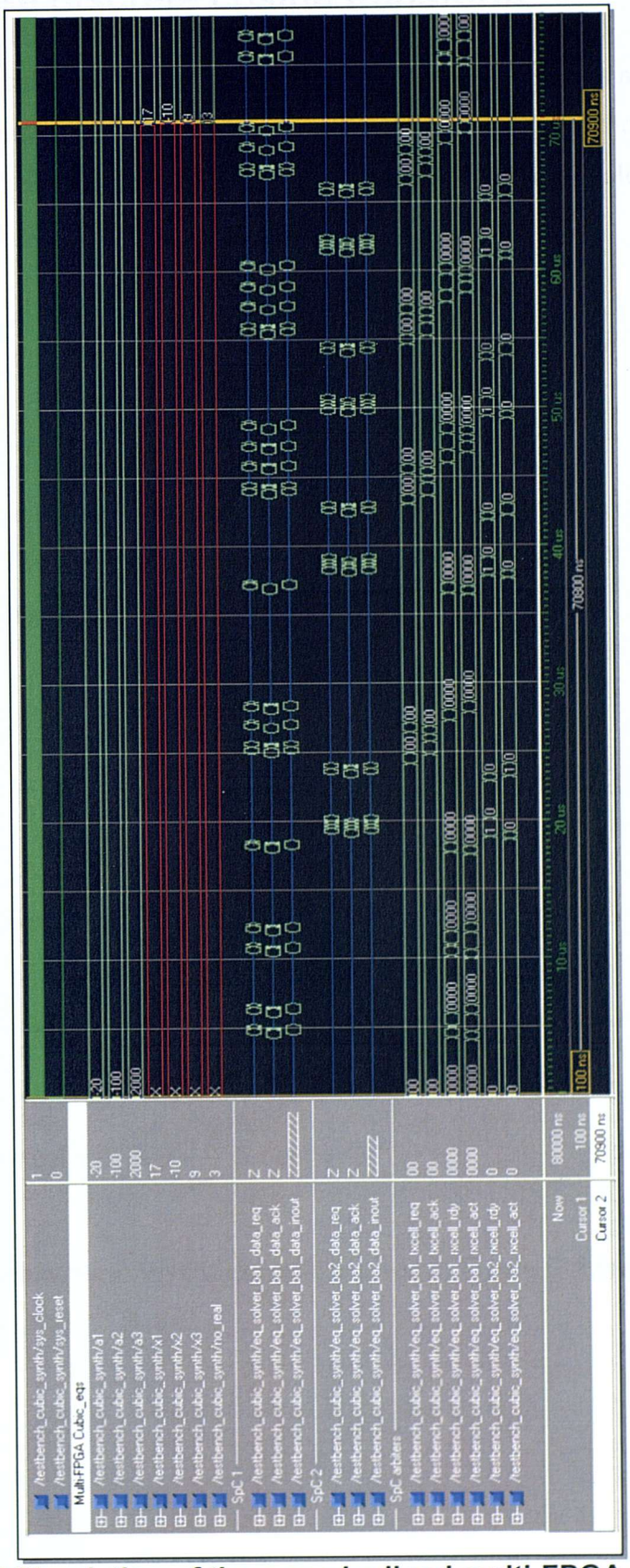


Figure D-6 Simulation of the non-pipelined multi-FPGA cubic equation solver

D.1.3 Inverse discrete cosine transform

The 2-D IDCT architecture is adapted from [142]. The architecture is made up of a one-dimensional 8-point IDCT followed by an internal double buffer memory, followed by another one-dimensional 8-point IDCT. The algorithm used for the calculation of the 2-D IDCT is based on Equation (D.3).

$$XC_{pq} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} XN_{mn} \bullet \frac{c(p)c(q)}{4} \bullet \cos\left(\frac{\pi(2m+1)p}{2M}\right) \bullet \cos\left(\frac{\pi(2n+1)q}{2N}\right) \quad \text{(D.3)}$$

Equation (D.3) can be separated into the row part and column part as shown in equations (D.4) and (D.5). The 2-D IDCT is computed by first applying 1-D IDCT on the rows and then on the columns.

$$C = \mathrm{K} \bullet \cos \frac{(2 \cdot col\ number\ +1) \bullet row\ \mathrm{number}\ \bullet \pi}{2 \bullet \mathrm{M}} \quad \text{(D.4)}$$

where $\mathrm{K} = \frac{\sqrt{1}}{\mathrm{N}}$ for row = 0, $\mathrm{K} = \frac{\sqrt{2}}{\mathrm{N}}$ for row $\neq$ 0.

$$C^{t} = \mathrm{K} \bullet \cos \frac{(2 \cdot row\ number\ +1) \bullet col\ \mathrm{number}\ \bullet \pi}{2 \bullet \mathrm{N}} \quad \text{(D.5)}$$

where $\mathrm{K} = \frac{\sqrt{1}}{\mathrm{M}}$ for col = 0, $\mathrm{K} = \frac{\sqrt{2}}{\mathrm{M}}$ for col $\neq$ 0.

The 2-D IDCT behavioural VHDL example is given in Figure D-8 and it uses the VHDL package in Figure D-7.

```
--************************************************************************ --
-- VHDL package for 2-D Inverse discrete cosine transform --
--************************************************************************ --
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
package idct_package is

  procedure idct1_mult_add (
    signal index : in unsigned(2 downto 0);
    signal in1a : in signed(11 downto 0);
    signal in2a : in signed(11 downto 0);
    signal in3a : in signed(11 downto 0);
    signal in4a : in signed(11 downto 0);
    signal in5a : in signed(11 downto 0);
    signal in6a : in signed(11 downto 0);
    signal in7a : in signed(11 downto 0);
    signal in8a : in signed(11 downto 0);
    result_a : out signed(21 downto 0) );

  procedure idct2_mult_add (
    signal index : in unsigned(2 downto 0);
    signal in1b : in signed(10 downto 0);
    signal in2b : in signed(10 downto 0);
    signal in3b : in signed(10 downto 0);
    signal in4b : in signed(10 downto 0);
    signal in5b : in signed(10 downto 0);
    signal in6b : in signed(10 downto 0);
    signal in7b : in signed(10 downto 0);
    signal in8b : in signed(10 downto 0);
    result_b : out signed(20 downto 0) );
end idct_package;

package body idct_package is

  procedure idct1_mult_add
  (
    signal index : in unsigned(2 downto 0);
    signal in1a : in signed(11 downto 0);
    signal in2a : in signed(11 downto 0);
    signal in3a : in signed(11 downto 0);
    signal in4a : in signed(11 downto 0);
    signal in5a : in signed(11 downto 0);
    signal in6a : in signed(11 downto 0);
    signal in7a : in signed(11 downto 0);
    signal in8a : in signed(11 downto 0);
    result_a : out signed(21 downto 0)
  ) is
  variable p1_tmp,p2_tmp,p3_tmp,p4_tmp,p5_tmp,p6_tmp,p7_tmp,p8_tmp :
signed(21 downto 0);
  begin
  p1_tmp := resize(signed(in1a * (91)), 22);
  case index is
  when "000" =>
    p2_tmp := resize(signed(in2a * (126)), 22);
    p3_tmp := resize(signed(in3a * (118)), 22);
    p4_tmp := resize(signed(in4a * (106)), 22);
    p5_tmp := resize(signed(in5a * (91)), 22);
    p6_tmp := resize(signed(in6a * (71)), 22);
    p7_tmp := resize(signed(in7a * (49)), 22);
    p8_tmp := resize(signed(in8a * (25)), 22);
```

```
 when "001" =>
    p2_tmp := resize(signed(in2a * (106)), 22);
    p3_tmp := resize(signed(in3a * (49)), 22);
    p4_tmp := resize(signed(in4a * (-25)), 22);
    p5_tmp := resize(signed(in5a * (-91)), 22);
    p6_tmp := resize(signed(in6a * (-126)), 22);
    p7_tmp := resize(signed(in7a * (-118)), 22);
    p8_tmp := resize(signed(in8a * (-71)), 22);
  when "010" =>
    p2_tmp := resize(signed(in2a * (71)), 22);
    p3_tmp := resize(signed(in3a * (-49)), 22);
    p4_tmp := resize(signed(in4a * (-126)), 22);
    p5_tmp := resize(signed(in5a * (-91)), 22);
    p6_tmp := resize(signed(in6a * (25)), 22);
    p7_tmp := resize(signed(in7a * (118)), 22);
    p8_tmp := resize(signed(in8a * (106)), 22);
  when "011" =>
    p2_tmp := resize(signed(in2a * (25)), 22);
    p3_tmp := resize(signed(in3a * (-118)), 22);
    p4_tmp := resize(signed(in4a * (-71)), 22);
    p5_tmp := resize(signed(in5a * (91)), 22);
    p6_tmp := resize(signed(in6a * (106)), 22);
    p7_tmp := resize(signed(in7a * (-49)), 22);
    p8_tmp := resize(signed(in8a * (-126)), 22);
  when "100" =>
    p2_tmp := resize(signed(in2a * (-25)), 22);
    p3_tmp := resize(signed(in3a * (-118)), 22);
    p4_tmp := resize(signed(in4a * (71)), 22);
    p5_tmp := resize(signed(in5a * (91)), 22);
    p6_tmp := resize(signed(in6a * (-106)), 22);
    p7_tmp := resize(signed(in7a * (-49)), 22);
    p8_tmp := resize(signed(in8a * (126)), 22);
    when "101" =>
    p2_tmp := resize(signed(in2a * (-71)), 22);
    p3_tmp := resize(signed(in3a * (-49)), 22);
    p4_tmp := resize(signed(in4a * (126)), 22);
    p5_tmp := resize(signed(in5a * (-91)), 22);
    p6_tmp := resize(signed(in6a * (-25)), 22);
    p7_tmp := resize(signed(in7a * (118)), 22);
    p8_tmp := resize(signed(in8a * (-106)), 22);
  when "110" =>
    p2_tmp := resize(signed(in2a * (-106)), 22);
    p3_tmp := resize(signed(in3a * (49)), 22);
    p4_tmp := resize(signed(in4a * (25)), 22);
    p5_tmp := resize(signed(in5a * (-91)), 22);
    p6_tmp := resize(signed(in6a * (126)), 22);
    p7_tmp := resize(signed(in7a * (-118)), 22);
    p8_tmp := resize(signed(in8a * (71)), 22);
  when "111" =>
    p2_tmp := resize(signed(in2a * (-126)), 22);
    p3_tmp := resize(signed(in3a * (118)), 22);
    p4_tmp := resize(signed(in4a * (-106)), 22);
    p5_tmp := resize(signed(in5a * (91)), 22);
    p6_tmp := resize(signed(in6a * (-71)), 22);
    p7_tmp := resize(signed(in7a * (49)), 22);
    p8_tmp := resize(signed(in8a * (-25)), 22);
  when others => NULL;
  end case;
end procedure idct1_mult_add;
--=================================================================--
```

```
procedure idct2_mult_add
  (
    signal index : in unsigned(2 downto 0);
    signal in1b : in signed(10 downto 0);
    signal in2b : in signed(10 downto 0);
    signal in3b : in signed(10 downto 0);
    signal in4b : in signed(10 downto 0);
    signal in5b : in signed(10 downto 0);
    signal in6b : in signed(10 downto 0);
    signal in7b : in signed(10 downto 0);
    signal in8b : in signed(10 downto 0);
    result_b : out signed(20 downto 0)
  ) is
  variable p1_tmp,p2_tmp,p3_tmp,p4_tmp,p5_tmp,p6_tmp,p7_tmp,p8_tmp : signed(20 downto 0);
  begin
  p1_tmp := resize(signed(in1b * (91)), 21);
  case index is
  when "000" =>
    p2_tmp := resize(signed(in2b * (126)), 21);
    p3_tmp := resize(signed(in3b * (118)), 21);
    p4_tmp := resize(signed(in4b * (106)), 21);
    p5_tmp := resize(signed(in5b * (91)), 21);
    p6_tmp := resize(signed(in6b * (71)), 21);
    p7_tmp := resize(signed(in7b * (49)), 21);
    p8_tmp := resize(signed(in8b * (25)), 21);
  when "001" =>
    p2_tmp := resize(signed(in2b * (106)), 21);
    p3_tmp := resize(signed(in3b * (49)), 21);
    p4_tmp := resize(signed(in4b * (-25)), 21);
    p5_tmp := resize(signed(in5b * (-91)), 21);
    p6_tmp := resize(signed(in6b * (-126)), 21);
    p7_tmp := resize(signed(in7b * (-118)), 21);
    p8_tmp := resize(signed(in8b * (-71)), 21);
  when "010" =>
    p2_tmp := resize(signed(in2b * (71)), 21);
    p3_tmp := resize(signed(in3b * (-49)), 21);
    p4_tmp := resize(signed(in4b * (-126)), 21);
    p5_tmp := resize(signed(in5b * (-91)), 21);
    p6_tmp := resize(signed(in6b * (25)), 21);
    p7_tmp := resize(signed(in7b * (118)), 21);
    p8_tmp := resize(signed(in8b * (106)), 21);
  when "011" =>
    p2_tmp := resize(signed(in2b * (25)), 21);
    p3_tmp := resize(signed(in3b * (-118)), 21);
    p4_tmp := resize(signed(in4b * (-71)), 21);
    p5_tmp := resize(signed(in5b * (91)), 21);
    p6_tmp := resize(signed(in6b * (106)), 21);
    p7_tmp := resize(signed(in7b * (-49)), 21);
    p8_tmp := resize(signed(in8b * (-126)), 21);
  when "100" =>
    p2_tmp := resize(signed(in2b * (-25)), 21);
    p3_tmp := resize(signed(in3b * (-118)), 21);
    p4_tmp := resize(signed(in4b * (71)), 21);
    p5_tmp := resize(signed(in5b * (91)), 21);
    p6_tmp := resize(signed(in6b * (-106)), 21);
    p7_tmp := resize(signed(in7b * (-49)), 21);
    p8_tmp := resize(signed(in8b * (126)), 21);
when "101" =>
    p2_tmp := resize(signed(in2b * (-71)), 21);
    p3_tmp := resize(signed(in3b * (-49)), 21);
    p4_tmp := resize(signed(in4b * (126)), 21);
    p5_tmp := resize(signed(in5b * (-91)), 21);
    p6_tmp := resize(signed(in6b * (-25)), 21);
    p7_tmp := resize(signed(in7b * (118)), 21);
    p8_tmp := resize(signed(in8b * (-106)), 21);
```

```
  when "110" =>
    p2_tmp := resize(signed(in2b * (-106)), 21);
    p3_tmp := resize(signed(in3b * (49)), 21);
    p4_tmp := resize(signed(in4b * (25)), 21);
    p5_tmp := resize(signed(in5b * (-91)), 21);
    p6_tmp := resize(signed(in6b * (126)), 21);
    p7_tmp := resize(signed(in7b * (-118)), 21);
    p8_tmp := resize(signed(in8b * (71)), 21);
  when "111" =>
    p2_tmp := resize(signed(in2b * (-126)), 21);
    p3_tmp := resize(signed(in3b * (118)), 21);
    p4_tmp := resize(signed(in4b * (-106)), 21);
    p5_tmp := resize(signed(in5b * (91)), 21);
    p6_tmp := resize(signed(in6b * (-71)), 21);
    p7_tmp := resize(signed(in7b * (49)), 21);
    p8_tmp := resize(signed(in8b * (-25)), 21);
  when others => NULL;
  end case;

  result_b := p1_tmp + p2_tmp + p3_tmp + p4_tmp + p5_tmp + p6_tmp + p7_tmp + p8_tmp;
  end procedure idct2_mult_add;

end idct  package;
```

Figure D-7 VHDL package for IDCT example

```
--************************************************** --
--  2-D Inverse discrete cosine transform         --
--************************************************** --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.idct_package.all;
entity idct is
port (
     in_hs_rdy:  in unsigned(0 downto 0);              -- Handshake ready
     in_hs_rcv:  buffer unsigned(0 downto 0) := "0";   -- Handshake receive
     dct_2d_in:  in signed(11 downto 0);
     idct_out:   out signed(7 downto 0) := (others=>'0'); --  8 bit output.
     out_hs_rdy: buffer unsigned(0 downto 0) := "0";      -- Handshake ready
     out_hs_rcv: in unsigned(0 downto 0);              -- Handshake receive
     sys_clock:  in unsigned(0 downto 0);
     --moods clock
     sys_reset:   in unsigned(0 downto 0)
     --moods reset
      );
end idct;
ARCHITECTURE behaviour of idct is
-- IDCT_2 signals
signal xa0_reg, xa1_reg, xa2_reg, xa3_reg,
      xa4_reg, xa5_reg, xa6_reg, xa7_reg:  signed(11 downto 0):= (others=>'0');
-- IDCT_2 signals
signal xb0_reg, xb1_reg, xb2_reg, xb3_reg,
      xb4_reg, xb5_reg, xb6_reg, xb7_reg:  signed(10 downto 0):= (others=>'0');
```

```
-- memory section
type RAM_mem_type is array (0 to 63) of signed(10 downto 0);
signal ID_ram1_mem:   RAM_mem_type;
--moods ram
signal ID_input_cnt:  unsigned(3 downto 0):= "0000";
signal ID_wr_cntr:    unsigned(6 downto 0):= "0000000";
signal ID_rd_cntr:    unsigned(3 downto 0):= "0000";
-- Handshake signals
signal ID_stage2_rdy:   unsigned(0 downto 0):= "0";
signal ID_stage2_rcv:   unsigned(0 downto 0):= "0";
signal ID_index_i : unsigned(3 downto 0):= "0000";
signal ID_index_j : unsigned(3 downto 0):= "0000";
begin

--*************************************************************************
ID1: process -- IDCT Process 1
 variable z_out_int : signed(21 downto 0) := (others=>'0');
begin
 reset_loop: loop

  ID_wr_cntr    <= "0000000";
  in_hs_rcv     <= "0";
  ID_input_cnt(3 downto 0)  <= "0000";
  ID_rd_cntr(3 downto 0)   <= "0000";
  ID_stage2_rdy <= "0";
  ID_index_i <= "0000";
 wait until sys_clock'event and sys_clock = "1";
 exit reset_loop when sys_reset = "1";
 main_loop: loop
  if(ID_wr_cntr(6) = '0') then

   while(ID_input_cnt(3) = '0') loop
     while(in_hs_rdy = in_hs_rcv) loop
       wait until sys_clock'event and sys_clock = "1";
     end loop;

     case ID_input_cnt(2 downto 0) is
     when "000" =>  xa0_reg <= dct_2d_in;
     when "001" =>  xa1_reg <= dct_2d_in;
     when "010" =>  xa2_reg <= dct_2d_in;
     when "011" =>  xa3_reg <= dct_2d_in;
     when "100" =>  xa4_reg <= dct_2d_in;
     when "101" =>  xa5_reg <= dct_2d_in;
     when "110" =>  xa6_reg <= dct_2d_in;
     when "111" =>  xa7_reg <= dct_2d_in;
     when others => NULL;
     end case;


     in_hs_rcv <= not in_hs_rcv;
     ID_input_cnt(3 downto 0) <= ID_input_cnt(3 downto 0) + "0001";
     wait until sys_clock'event and sys_clock = "1";
   end loop;

   while (ID_index_i /= "1000") loop
   idct1_mult_add(ID_index_i(2 downto 0),xa0_reg,xa1_reg,xa2_reg,
              xa3_reg,xa4_reg,xa5_reg,xa6_reg,xa7_reg,z_out_int);
    if(z_out_int(20) = '0' and z_out_int(7) = '1') then
      ID_ram1_mem(to_integer(ID_wr_cntr(5 downto 0)))<= z_out_int(18 downto 8)+to_signed(1,11);
      ID_wr_cntr <= ID_wr_cntr + "0000001";
   else
      ID_ram1_mem(to_integer(ID_wr_cntr(5 downto 0))) <= z_out_int(18 downto 8);
      ID_wr_cntr <= ID_wr_cntr + "0000001";
    end if;
```

```
      ID_index_i <= ID_index_i + "0001";
    end loop;
    ID_index_i <= "0000";
  else
    while(ID_rd_cntr(3) = '0') loop

    -- Semaphore Master
     while(ID_stage2_rdy /= ID_stage2_rcv) loop
       wait until sys_clock'event and sys_clock = "1";
     end loop;

     case ID_rd_cntr(2 downto 0) is
     when "000" =>  xb0_reg <= ID_ram1_mem(0);
                    xb1_reg <= ID_ram1_mem(8);
                    xb2_reg <= ID_ram1_mem(16);
                    xb3_reg <= ID_ram1_mem(24);
                    xb4_reg <= ID_ram1_mem(32);
                    xb5_reg <= ID_ram1_mem(40);
                    xb6_reg <= ID_ram1_mem(48);
                    xb7_reg <= ID_ram1_mem(56);
     when "001" =>  xb0_reg <= ID_ram1_mem(1);
                    xb1_reg <= ID_ram1_mem(9);
                    xb2_reg <= ID_ram1_mem(17);
                    xb3_reg <= ID_ram1_mem(25);
                    xb4_reg <= ID_ram1_mem(33);
                    xb5_reg <= ID_ram1_mem(41);
                    xb6_reg <= ID_ram1_mem(49);
                    xb7_reg <= ID_ram1_mem(57);
     when "010" =>  xb0_reg <= ID_ram1_mem(2);
                    xb1_reg <= ID_ram1_mem(10);
                    xb2_reg <= ID_ram1_mem(18);
                    xb3_reg <= ID_ram1_mem(26);
                    xb4_reg <= ID_ram1_mem(34);
                    xb5_reg <= ID_ram1_mem(42);
                    xb6_reg <= ID_ram1_mem(50);
                    xb7_reg <= ID_ram1_mem(58);
     when "011" =>  xb0_reg <= ID_ram1_mem(3);
                    xb1_reg <= ID_ram1_mem(11);
                    xb2_reg <= ID_ram1_mem(19);
                    xb3_reg <= ID_ram1_mem(27);
                    xb4_reg <= ID_ram1_mem(35);
                    xb5_reg <= ID_ram1_mem(43);
                    xb6_reg <= ID_ram1_mem(51);
                    xb7_reg <= ID_ram1_mem(59);
     when "100" =>  xb0_reg <= ID_ram1_mem(4);
                    xb1_reg <= ID_ram1_mem(12);
                    xb2_reg <= ID_ram1_mem(20);
                    xb3_reg <= ID_ram1_mem(28);
                    xb4_reg <= ID_ram1_mem(36);
                    xb5_reg <= ID_ram1_mem(44);
                    xb6_reg <= ID_ram1_mem(52);
                    xb7_reg <= ID_ram1_mem(60);
     when "101" =>  xb0_reg <= ID_ram1_mem(5);
                    xb1_reg <= ID_ram1_mem(13);
                    xb2_reg <= ID_ram1_mem(21);
                    xb3_reg <= ID_ram1_mem(29);
                    xb4_reg <= ID_ram1_mem(37);
                    xb5_reg <= ID_ram1_mem(45);
                    xb6_reg <= ID_ram1_mem(53);
                    xb7_reg <= ID_ram1_mem(61);
```

```
      when "110" =>  xb0_reg <= ID_ram1_mem(6);
                     xb1_reg <= ID_ram1_mem(14);
                     xb2_reg <= ID_ram1_mem(22);
                     xb3_reg <= ID_ram1_mem(30);
                     xb4_reg <= ID_ram1_mem(38);
                     xb5_reg <= ID_ram1_mem(46);
                     xb6_reg <= ID_ram1_mem(54);
                     xb7_reg <= ID_ram1_mem(56);
      when "111" =>  xb0_reg <= ID_ram1_mem(7);
                     xb1_reg <= ID_ram1_mem(15);
                     xb2_reg <= ID_ram1_mem(23);
                     xb3_reg <= ID_ram1_mem(31);
                     xb4_reg <= ID_ram1_mem(39);
                     xb5_reg <= ID_ram1_mem(47);
                     xb6_reg <= ID_ram1_mem(55);
                     xb7_reg <= ID_ram1_mem(63);
      when others => NULL;
      end case;

      ID_stage2_rdy <= not ID_stage2_rdy;
      ID_rd_cntr(3 downto 0) <= ID_rd_cntr(3 downto 0) + "0001";
      wait until sys_clock'event and sys_clock = "1";
    end loop;
    ID_input_cnt(3 downto 0)  <= "0000";
    ID_wr_cntr(6 downto 0)    <= (others=>'0');
    ID_rd_cntr(3 downto 0)    <= (others=>'0');
   end if;
  wait until sys_clock'event and sys_clock = "1";
  exit reset_loop when sys_reset = "1";
  end loop;
  end loop;
end process ID1;

--*************************************************************************
ID2: process -- IDCT Process 2
  variable idct2d_int: signed(20 downto 0):= (others=>'0');
begin
  reset_loop: loop
    ID_stage2_rcv <= "0";
    out_hs_rdy <= "0";
    idct2d_int    := (others=>'0');
    ID_index_j <= "0000";
    wait until sys_clock'event and sys_clock = "1";
    exit reset_loop when sys_reset = "1";
    main_loop: loop

     while(ID_stage2_rdy = ID_stage2_rcv) loop
       wait until sys_clock'event and sys_clock = "1";
     end loop;

     while (ID_index_j /= "1000") loop
     idct2_mult_add(ID_index_j(2 downto 0),xb0_reg,xb1_reg,xb2_reg,xb3_reg,
                    xb4_reg,xb5_reg,xb6_reg,xb7_reg,idct2d_int);
      while(out_hs_rdy /= out_hs_rcv) loop
        wait until sys_clock'event and sys_clock = "1";
       end loop;
       idct_out <= signed(idct2d_Int(15 downto 8));
       out_hs_rdy <= not out_hs_rdy;
       ID_index_j <= ID_index_j + "0001";
       wait until sys_clock'event and sys_clock = "1";
     end loop;
```

```
    ID_index_j <= "0000";
    ID_stage2_rcv <= not ID_stage2_rcv;
    wait until sys_clock'event and sys_clock = "1";
    exit reset_loop when sys_reset = "1";
    end loop;
  end loop;
end process ID2;
--*****************************************************************************
end behaviour;
```

Figure D-8 VHDL of IDCT example

The post-MOODS synthesis simulation of the non-pipelined multi-FPGA IDCT is given in Figure D-9. Zoom in views of the simulation showing inputs and outputs updates are given in Figure D-10. The multi-FPGA IDCT has a single subprogram communication channel (*SpC 1*) and a single channel arbiter. With a system clock period of 40 ns, the non-pipelined multi-FPGA IDCT takes 4175 clock cycles (i.e. (167480 ns - 480 ns) / 40 ns) to complete the application.

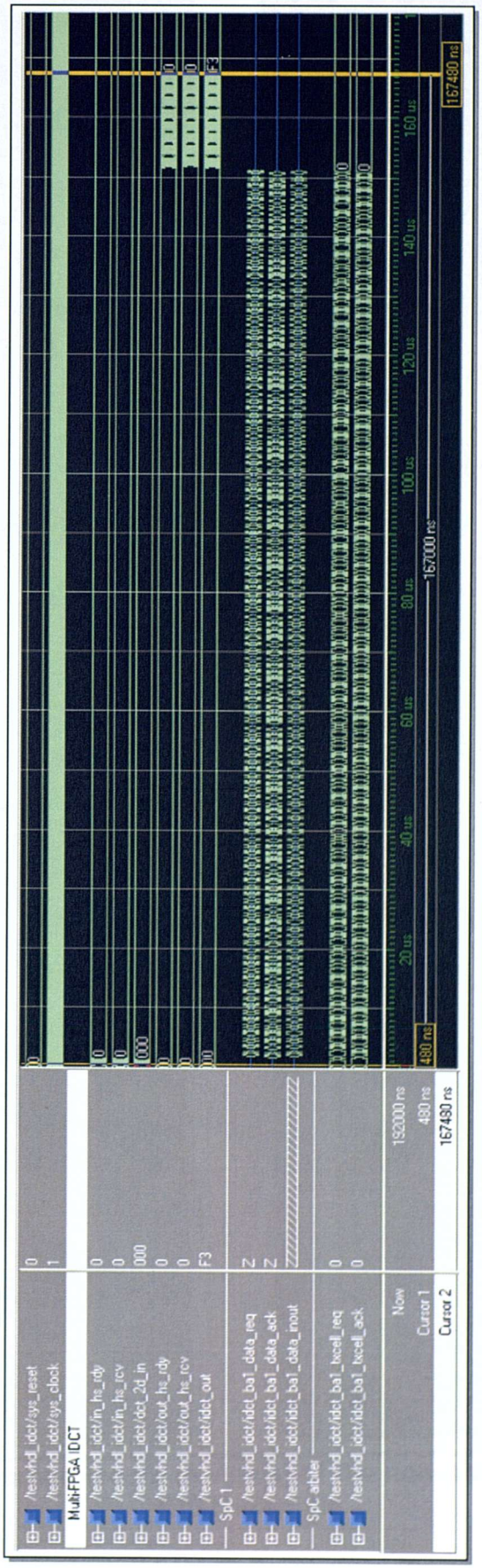


Figure D-9 Simulation of the non-pipelined multi-FPGA IDCT example

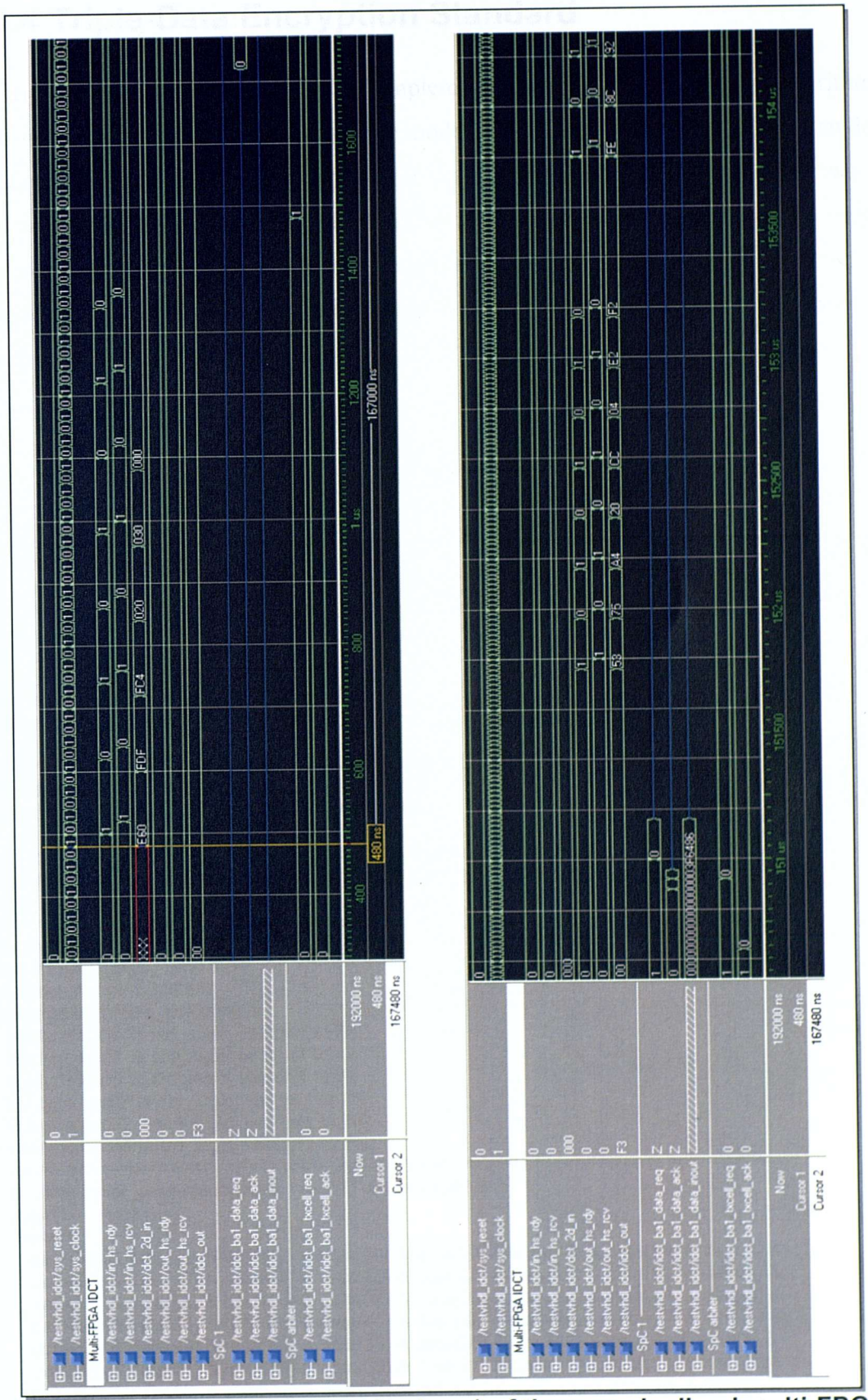


Figure D-10 Simulation (zoom in views) of the non-pipelined multi-FPGA IDCT example

D.1.4 Triple-Data Encryption Standard

The triple-data encryption standard core implements the triple data encryption algorithm (TDEA) in the electronic codebook (ECB) mode [144]. The idea of triple DES is that data is encrypted three times (i.e. encrypted, decrypted and then encrypted again) using two different keys. In this case, the two encryptions use the first key and the decryption uses the second key. The VHDL package of the triple-DES is given in Figure D-11 and the behavioural VHDL of the triple-data encryption standard (triple-DES) core is given in Figure D-12.

```
--************************************ --
--  VHDL package for Triple-DES  --
--************************************ --
library ieee;
use ieee.std_logic_1164.all;
package des_functions is
  subtype vec56 is std_logic_vector(1 to 56);
  subtype vec64 is std_logic_vector(1 to 64);

  -- The key_reduce function reduces a 64-bit key to a 56-bit key by stripping off parity bits
  function key_reduce1(key : in vec64) return vec56;
  function key_reduce2(key : in vec64) return vec56;

  -- The des_core function implements a DES encrypt/decrypt cycle
  function des_core(plaintext : vec64; key : vec56; encrypt : std_logic) return vec64;
end;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
package body des_functions is
  subtype vec3  is std_logic_vector(1 to 3);
  subtype vec4  is std_logic_vector(1 to 4);
  subtype vec6  is std_logic_vector(1 to 6);
  subtype vec28 is std_logic_vector(1 to 28);
  subtype vec32 is std_logic_vector(1 to 32);
  subtype vec48 is std_logic_vector(1 to 48);
--=================================================================--
  function initial_permutation(data : vec64) return vec64 is
  begin
    return
      data(58) & data(50) & data(42) & data(34) & data(26) & data(18) & data(10) & data(2) &
      data(60) & data(52) & data(44) & data(36) & data(28) & data(20) & data(12) & data(4) &
      data(62) & data(54) & data(46) & data(38) & data(30) & data(22) & data(14) & data(6) &
      data(64) & data(56) & data(48) & data(40) & data(32) & data(24) & data(16) & data(8) &
      data(57) & data(49) & data(41) & data(33) & data(25) & data(17) & data(9)  & data(1) &
      data(59) & data(51) & data(43) & data(35) & data(27) & data(19) & data(11) & data(3) &
      data(61) & data(53) & data(45) & data(37) & data(29) & data(21) & data(13) & data(5) &
      data(63) & data(55) & data(47) & data(39) & data(31) & data(23) & data(15) & data(7);
  end;
```

```
--=================================================================--
 function final_permutation(data : in vec64) return vec64 is
 begin
  return
    data(40) & data(8) & data(48) & data(16) & data(56) & data(24) & data(64) & data(32) &
    data(39) & data(7) & data(47) & data(15) & data(55) & data(23) & data(63) & data(31) &
    data(38) & data(6) & data(46) & data(14) & data(54) & data(22) & data(62) & data(30) &
    data(37) & data(5) & data(45) & data(13) & data(53) & data(21) & data(61) & data(29) &
    data(36) & data(4) & data(44) & data(12) & data(52) & data(20) & data(60) & data(28) &
    data(35) & data(3) & data(43) & data(11) & data(51) & data(19) & data(59) & data(27) &
    data(34) & data(2) & data(42) & data(10) & data(50) & data(18) & data(58) & data(26) &
    data(33) & data(1) & data(41) & data(9)  & data(49) & data(17) & data(57) & data(25);
 end;
--=================================================================--
 function expand(data : vec32) return vec48 is
 begin
  return
    data(32) & data(1)  & data(2)  & data(3)  & data(4)  & data(5) & data(4)  & data(5)  &
    data(6)  & data(7)  & data(8)  & data(9) & data(8)  & data(9)  & data(10) & data(11) &
    data(12) & data(13) & data(12) & data(13) & data(14) & data(15) & data(16) & data(17) &
    data(16) & data(17) & data(18) & data(19) & data(20) & data(21) & data(20) & data(21) &
    data(22) & data(23) & data(24) & data(25) & data(24) & data(25) & data(26) & data(27) &
    data(28) & data(29) & data(28) & data(29) & data(30) & data(31) & data(32) & data(1);
 end;
--=================================================================--
 function substitute(data : vec48) return vec32 is
  type S_block_type is array(0 to 63) of natural range 0 to 15;
  constant S_block0 : S_block_type :=
   --moods ROM
   ( 14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7, 0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
     4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0, 15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13 );
  constant S_block1 : S_block_type :=
   --moods ROM
   ( 15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10, 3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
     0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15, 13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9 );
  constant S_block2 : S_block_type :=
   --moods ROM
   ( 10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8, 13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
     13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7, 1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12 );
  constant S_block3 : S_block_type :=
   --moods ROM
   ( 7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15, 13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
     10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4, 3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14 );
  constant S_block4 : S_block_type :=
   --moods ROM
   ( 2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9, 14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
     4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14, 11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3 );
  constant S_block5 : S_block_type :=
   --moods ROM
   ( 12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11, 10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
     9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6, 4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13 );
  constant S_block6 : S_block_type :=
   --moods ROM
   ( 4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1, 13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
     1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2, 6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12 );
  constant S_block7 : S_block_type :=
   --moods ROM
   ( 13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7, 1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
     7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8, 2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11 );
 begin
 return
 std_logic_vector(to_unsigned(S_block0(to_integer(unsigned(data(1)&data(6)&data(2 to 5)))),4))&
 std_logic_vector(to_unsigned(S_block1(to_integer(unsigned(data(7)&data(12)&data(8 to 11)))),4))&
 std_logic_vector(to_unsigned(S_block2(to_integer(unsigned(data(13)&data(18)&data(14 to 17)))),4))&
 std_logic_vector(to_unsigned(S_block3(to_integer(unsigned(data(19)&data(24)&data(20 to 23)))),4))&
```

```
 std_logic_vector(to_unsigned(S_block4(to_integer(unsigned(data(25)&data(30)&data(26 to 29)))),4))&
 std_logic_vector(to_unsigned(S_block5(to_integer(unsigned(data(31)&data(36)&data(32 to 35)))),4))&
 std_logic_vector(to_unsigned(S_block6(to_integer(unsigned(data(37)&data(42)&data(38 to 41)))),4))&
 std_logic_vector(to_unsigned(S_block7(to_integer(unsigned(data(43)&data(48)&data(44 to 47)))),4));
 end;
--=====================================================================--
 function permute (data : in vec32) return vec32 is
 begin
  return
   data(16) & data(7)  & data(20) & data(21) & data(29) & data(12) & data(28) & data(17) &
   data(1)  & data(15) & data(23) & data(26) & data(5)  & data(18) & data(31) & data(10) &
   data(2)  & data(8)  & data(24) & data(14) & data(32) & data(27) & data(3)  & data(9) &
   data(19) & data(13) & data(30) & data(6)  & data(22) & data(11) & data(4)  & data(25);  end;
--=====================================================================--
 function f(data : vec32; subkey : vec48) return vec32 is – Cipher function,f
 begin
  return permute(substitute(expand(data) xor subkey));  end;
--=====================================================================--
 function key_reduce1(key : in vec64) return vec56 is
 begin
  return
   key(57) & key(49) & key(41) & key(33) & key(25) & key(17) & key(9)  & key(1) &
   key(58) & key(50) & key(42) & key(34) & key(26) & key(18) & key(10) & key(2) &
   key(59) & key(51) & key(43) & key(35) & key(27) & key(19) & key(11) & key(3) &
   key(60) & key(52) & key(44) & key(36) & key(63) & key(55) & key(47) & key(39) &
   key(31) & key(23) & key(15) & key(7)  & key(62) & key(54) & key(46) & key(38) &
   key(30) & key(22) & key(14) & key(6)  & key(61) & key(53) & key(45) & key(37) &
   key(29) & key(21) & key(13) & key(5)  & key(28) & key(20) & key(12) & key(4);
 end;
--=====================================================================--
 function key_reduce2(key : in vec64) return vec56 is
 begin
  return
   key(57) & key(49) & key(41) & key(33) & key(25) & key(17) & key(9)  & key(1) &
   key(58) & key(50) & key(42) & key(34) & key(26) & key(18) & key(10) & key(2) &
   key(59) & key(51) & key(43) & key(35) & key(27) & key(19) & key(11) & key(3) &
   key(60) & key(52) & key(44) & key(36) & key(63) & key(55) & key(47) & key(39) &
   key(31) & key(23) & key(15) & key(7)  & key(62) & key(54) & key(46) & key(38) &
   key(30) & key(22) & key(14) & key(6)  & key(61) & key(53) & key(45) & key(37) &
   key(29) & key(21) & key(13) & key(5)  & key(28) & key(20) & key(12) & key(4);
 end;
--=====================================================================--
 function key_rotate(key : vec56; round : natural range 0 to 15; encrypt : std_logic) return vec56 is
  type distance_type is array (natural range 0 to 31) of integer range 0 to 31;
  constant shift_distance : distance_type :=
   --moods ROM
   ( 0,  1,  2,  2,  2,  2,  2,  2,  1,  2,  2,  2,  2,  2,  2,  1,
     27, 27, 26, 26, 26, 26, 26, 26, 27, 26, 26, 26, 26, 26, 26, 27);
  variable distance : natural range 0 to 31;
 begin
  distance := shift_distance(to_integer(unsigned(encrypt & to_unsigned(round,4))));
  return vec28(unsigned(key(1 to 28)) ror distance) & vec28(unsigned(key(29 to 56)) ror distance);
 end;
--=====================================================================--
 function key_compress(key : in vec56) return vec48 is
 begin
  return
   key(14) & key(17) & key(11) & key(24) & key(1)  & key(5)  & key(3)  & key(28) &
   key(15) & key(6)  & key(21) & key(10) & key(23) & key(19) & key(12) & key(4) &
   key(26) & key(8)  & key(16) & key(7)  & key(27) & key(20) & key(13) & key(2) &
   key(41) & key(52) & key(31) & key(37) & key(47) & key(55) & key(30) & key(40) &
   key(51) & key(45) & key(33) & key(48) & key(44) & key(49) & key(39) & key(56) &
   key(34) & key(53) & key(46) & key(42) & key(50) & key(36) & key(29) & key(32);
 end;
--=====================================================================--
```

```
  function des_core(plaintext : vec64; key : vec56; encrypt : std_logic) return vec64 is
    --moods inline
    variable data : vec64;
    variable working_key : vec56 := key;
  begin
    data := initial_permutation(plaintext);
    for round in 0 to 15 loop
      working_key := key_rotate(working_key,round,encrypt);
      data := data(33 to 64) & (f(data(33 to 64),key_compress(working_key)) xor data(1 to 32));
    end loop;
    return final_permutation(data(33 to 64) & data(1 to 32));
  end;
end;
```

Figure D-11 VHDL package for triple-DES example

```
--************************** --
--  Triple-DES              --
--************************** --
library ieee;
use ieee.std_logic_1164.all;
use work.des_functions.all;
entity tdes_ede2 is
  port(
    plaintext:   in std_logic_vector(1 to 32);  -- now uses 32-bit input
    keys:        in std_logic_vector(1 to 32);  -- now uses 32-bit key (4 x 32-bits = 128-bit key)
    in_hs_rdy:   in std_logic_vector(0 downto 0);
    in_hs_rcv:   buffer std_logic_vector(0 downto 0) := "0";
    encrypt:     in std_logic;
    out_hs_rdy:  buffer std_logic_vector(0 downto 0) := "0";
    out_hs_rcv:  in std_logic_vector(0 downto 0);
    ciphertext:  out std_logic_vector(1 to 32); -- now uses 32-bit
    sys_reset: in std_logic;
    --moods reset
    sys_clock: in std_logic
    --moods clock
    );
end;
architecture behaviour of tdes_ede2 is
  process
    variable data,  key1, key2 : vec64;
    variable key : vec56;
    variable mode : std_logic;
  begin
    reset_loop: loop
      in_hs_rcv <= "0";
      out_hs_rdy <= "0";
      wait until sys_clock'event and sys_clock = '1';
    exit reset_loop when sys_reset = '1';
    main_loop: loop
      for in_cnt in 0 to 3 loop
        while(in_hs_rdy = in_hs_rcv) loop
          wait until sys_clock'event and sys_clock = '1';
        end loop;
        case in_cnt is
        when 0 =>
          data(33 to 64) := plaintext(1 to 32);
          key1(33 to 64) := keys(1 to 32);
```

```
      when 1 =>
        data(1 to 32) := plaintext(1 to 32);
        key1(1 to 32) := keys(1 to 32);
      when 2 =>
        key2(33 to 64) := keys(1 to 32);
      when 3 =>
        key2(1 to 32) := keys(1 to 32);
      when others => NULL;
      end case;
      in_hs_rcv <= not in_hs_rcv;
      wait until sys_clock'event and sys_clock = '1';
    end loop;

    for loop_cnt  in 0 to 2 loop
      case loop_cnt is
        when 1 =>
          key := key_reduce2(key2);
          mode := not encrypt;
        when others =>
          key := key_reduce1(key1);
          mode := encrypt;
        end case;
        data := des_core(data, key, mode);
    end loop;
    for out_cnt in 0 to 1 loop
      while(out_hs_rdy /= out_hs_rcv) loop
        wait until sys_clock'event and sys_clock = '1';
    end loop;

    case out_cnt is
       when 0 =>  ciphertext(1 to 32) <= data(1 to 32);
       when others =>  ciphertext(1 to 32) <= data(33 to 64);
    end case;
    out_hs_rdy <= not out_hs_rdy;
    wait until sys_clock'event and sys_clock = '1';
    end loop;
   wait until sys_clock'event and sys_clock = '1';
   exit reset_loop when sys_reset = '1';
   end loop;
  end loop;
  end process;
end;
```

Figure D-12 VHDL of triple-DES example

The post-MOODS synthesis simulation of the non-pipelined multi-FPGA triple-DES core is given in Figure D-13. Zoom in views of the simulation showing inputs and outputs updates are given in Figure D-14. With a system clock period of 40 ns, the non-pipelined multi-FPGA triple-DES core takes 3950 clock cycles (i.e. clock cycles = (158420 ns - 420 ns) / 40 ns) to encrypt 64-bit plaintext using a 128-bit key.

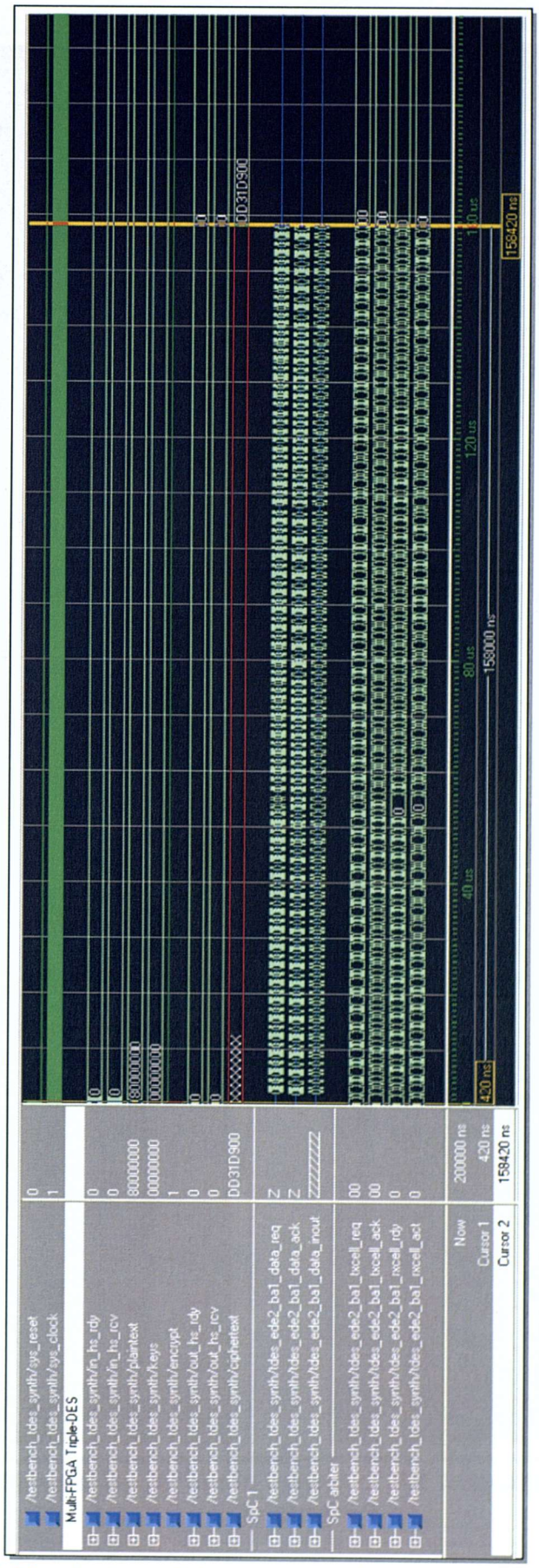


Figure D-13 Simulation of the non-pipelined multi-FPGA Triple-DES

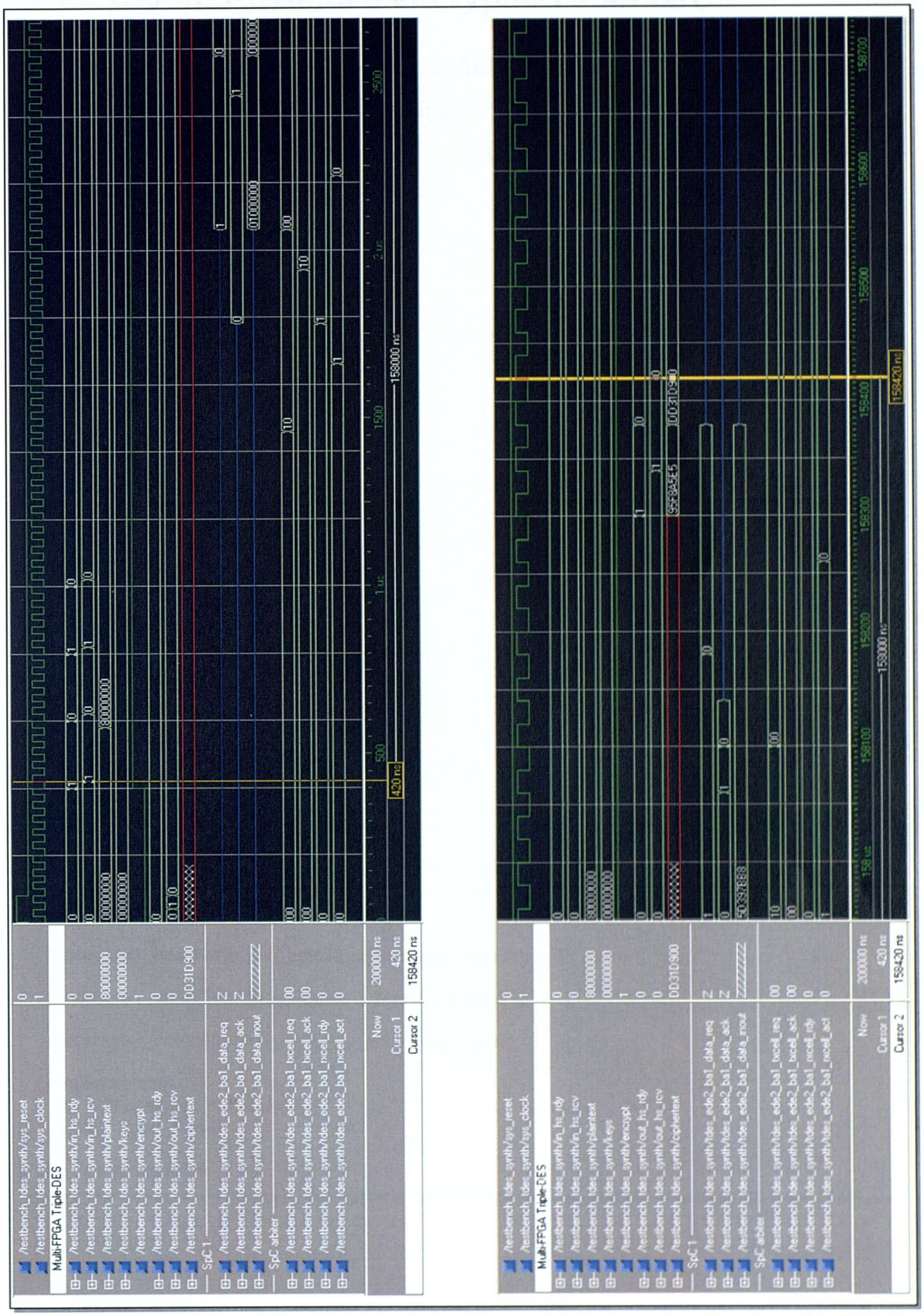


Figure D-14 Simulation (zoom in views) of the non-pipelined multi-FPGA Triple-DES

D.1.5 256-bit Advanced encryption standard

The 256-bit advanced encryption standard (AES) [146] implements the Rijndael algorithm that processes data blocks of 128 bits using a 256-bit cipher key. The behavioural VHDL of the 256-bit AES example is given in Figure D-16 and it uses the VHDL package in Figure D-15.

```
--************************************************ --
--  VHDL package for 256-AES packages    --
--************************************************ --

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package aes_procedures is
  subtype u_sign8 is unsigned(1 to 8);
  subtype u_sign16 is unsigned(1 to 16);
  subtype u_sign32 is unsigned(1 to 32);
  subtype u_sign64 is unsigned(1 to 64);
  subtype u_sign128 is unsigned(1 to 128);
  type rom_tab_1  is array(0 to 255)  of u_sign8;
  type rom_tab_2  is array(0 to 29)   of u_sign8;
  type rom_tab_5  is array(0 to 255)  of u_sign32;
  type rom_tab_7  is array(0 to 255)  of integer;
  type tab_4      is array(0 to 3)    of u_sign8;
  type tab_a8     is array(0 to 7)    of u_sign32;
  type tab_a6     is array(0 to 5)    of u_sign32;
  type tab_a4     is array(0 to 3)    of u_sign32;
  type tab_90     is array(0 to 89)   of u_sign32;
  type tab_44     is array(0 to 43)   of u_sign32;
  type tab_64     is array(0 to 63)   of u_sign32;

  function word (a :  in u_sign8) return u_sign32;

  procedure r_oneto24(a: in u_sign32; q_out: out u_sign32);

  procedure r_oneto16(a: in u_sign32; q_out: out u_sign32);

  procedure r_oneto8(a: in u_sign32; q_out: out u_sign32);

  procedure rco(
    a:      in unsigned(4 downto 0);
    a_out:  out u_sign32  );

end aes_procedures;
```

```
package body aes_procedures is

--=====================================================================--
function word(a :  in u_sign8) return u_sign32 is
  variable q : u_sign32;
begin
  q := a(1 to 7) & "00000000000000000000000";
  return q;
end word;
--=====================================================================--
procedure r_oneto24(
  a:      in u_sign32;
  q_out:  out u_sign32
) is
begin
  q_out := a(25 to 32) & a(1 to 24);
end r_oneto24;
--=====================================================================--
procedure r_oneto16(
  a:      in u_sign32;
  q_out:  out u_sign32
) is
begin
  q_out := a(17 to 32) & a(1 to 16);
end r_oneto16;
--=====================================================================--
procedure r_oneto8(
---- moods inline
  a:      in u_sign32;
  q_out:  out u_sign32
) is
begin
  q_out := a(9 to 32) & a(1 to 8);
end r_oneto8;
--=====================================================================--
procedure rco (
---- moods inline
  a:  in unsigned(4 downto 0);
  a_out: out u_sign32
) is
  constant rcotab: rom_tab_2 :=
  -- moods rom
    ( "00000001", "00000010", "00000100", "00001000", "00010000",
      "00100000", "01000000", "10000000", "00011011", "00110110",
      "01101100", "11011000", "10101011", "01001101", "10011010",
      "00101111", "01011110", "10111100", "01100011", "11000110",
      "10010111", "00110101", "01101010", "11010100", "10110011",
      "01111101", "11111010", "11101111", "11000101", "10010001" );
begin
    a_out := rcotab(to_integer(a)) & "00000000000000000000000";
  end rco;
end aes_procedures;
```

```
--************************************************ --
--  Encryption tables for 256-AES example       --
--************************************************ --
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.aes_procedures.all;

package encryption_tables is

  function fbsub(a: in u_sign8 ) return u_sign8;

  function fbsub_quad(a: in u_sign32) return u_sign32;

  function ftable( a :  in u_sign8 ) return u_sign32;

  function ftable_double( a, b : in u_sign8)return u_sign64;

  procedure ftable_quad(
    a:       in u_sign32;
    q_out:   out u_sign32
  );
end encryption_tables;

package body encryption_tables is

  function fbsub ( a :  in u_sign8) return u_sign8 is
  -- moods inline
    constant fbsubtab : rom_tab_1 :=
    -- moods rom
    ( "01100011","01111100","01110111","01111011","11110010","01101011","01101111","11000101",
"00110000","00000001","01100111","00101011","11111110","11010111","10101011","01110110",
"11001010","10000010","11001001","01111101", "11111010","01011001","01000111","11110000",
"10101101","11010100","10100010","10101111","10011100","10100100","01110010","11000000",
"10110111","11111101","10010011","00100110","00110110","00111111","11110111","11001100",
"00110100","10100101","11100101","11110001","01110001","11011000","00110001","00010101",
"00000100","11000111","00100011","11000011","00011000","10010110","00000101","10011010",
"00000111","00010010","10000000","11100010","11101011","00100111","10110010","01110101",
"00001001","10000011","00101100","00011010","00011011","01101110","01011010","10100000",
"01010010","00111011","11010110","10110011","00101001","11100011","00101111","10000100",
"01010011","11010001","00000000","11101101", "00100000","11111100","10110001","01011011",
"01101010","11001011","10111110","00111001","01001010","01001100","01011000","11001111",
"11010000","11101111","10101010","11111011","01000011","01001101","00110011","10000101",
"01000101","11111001","00000010","01111111","01010000","00111100","10011111","10101000",
"01010001","10100011","01000000","10001111","10010010","10011101","00111000","11110101",
"10111100","10110110","11011010","00100001","00010000","11111111","11110011","11010010",
"11001101","00001100","00010011","11101100","01011111","10010111","01000100","00010111",
"11000100","10100111","01111110","00111101","01100100","01011101","00011001","01110011",
"01100000","10000001","01001111","11011100","00100010","00101010", "10010000","10001000",
"01000110","11101110","10111000","00010100","11011110","01011110","00001011","11011011",
"11100000","00110010","00111010","00001010","01001001","00000110","00100100","01011100",
"11000010","11010011","10101100","01100010","10010001","10010101","11100100","01111001",
"11100111","11001000","00110111","01101101","10001101","11010101","01001110","10101001",
"01101100","01010110","11110100","11101010","01100101","01111010","10101110","00001000",
"10111010","01111000","00100101","00101110","00011100","10100110","10110100","11000110",
"11101000","11011101","01110100","00011111","01001011","10111101","10001011","10001010",
"01110000","00111110","10110101","01100110","01001000","00000011","11110110","00001110",
"01100001","00110101","01010111","10111001","10000110","11000001","00011101","10011110",
"11100001","11111000","10011000","00010001","01101001","11011001","10001110","10010100",
"10011011","00011110","10000111","11101001","11001110","01010101","00101000","11011111",
"10001100","10100001","10001001","00001101","10111111","11100110","01000010","01101000",
"01000001","10011001","00101101","00001111","10110000","01010100","10111011","00010110" );
```

```
    variable b : natural range 0 to 255;
    variable q : u_sign8;
    begin
      b := to_integer(a);
      q := fbsubtab(b);
      return q;
   end fbsub;
--=================================================================--
   function fbsub_quad ( a :  in u_sign32) return u_sign32 is
   ---- moods inline
    constant fbsubtab : rom_tab_1 :=
    -- moods rom
    ( "01100011","01111100","01110111","01111011","11110010","01101011","01101111","11000101",
 "00110000","00000001","01100111","00101011","11111110","11010111","10101011","01110110",
 "11001010","10000010","11001001","01111101","11111010","01011001","01000111","11110000",
 "10101101","11010100","10100010","10101111","10011100","10100100","01110010","11000000",
 "10110111","11111101","10010011","00100110","00110110","00111111","11110111","11001100",
 "00110100","10100101","11100101","11110001","01110001","11011000","00110001","00010101",
 "00000100","11000111","00100011","11000011","00011000","10010110","00000101","10011010",
 "00000111","00010010","10000000","11100010","11101011","00100111","10110010","01110101",
 "00001001","10000011","00101100","00011010","00011011","01101110","01011010","10100000",
 "01010010","00111011","11010110","10110011","00101001","11100011","00101111","10000100",
 "01010011","11010001","00000000","11101101","00100000","11111100","10110001","01011011",
 "01101010","11001011","10111110","00111001","01001010","01001100","01011000","11001111",
 "11010000","11101111","10101010","11111011","01000011","01001101","00110011","10000101",
 "01000101","11111001","00000010","01111111","01010000","00111100","10011111","10101000",
 "01010001","10100011","01000000","10001111","10010010","10011101","00111000","11110101",
 "10111100","10110110","11011010","00100001","00010000","11111111","11110011","11010010",
 "11001101","00001100","00010011","11101100","01011111","10010111","01000100","00010111",
 "11000100","10100111","01111110","00111101","01100100","01011101","00011001","01110011",
 "01100000","10000001","01001111","11011100","00100010","00101010","10010000","10001000",
 "01000110","11101110","10111000","00010100","11011110","01011110","00001011","11011011",
 "11100000","00110010","00111010","00001010","01001001","00000110","00100100","01011100",
 "11000010","11010011","10101100","01100010","10010001","10010101","11100100","01111001",
 "11100111","11001000","00110111","01101101","10001101","11010101","01001110","10101001",
 "01101100","01010110","11110100","11101010","01100101","01111010","10101110","00001000",
 "10111010","01111000","00100101","00101110","00011100","10100110","10110100","11000110",
 "11101000","11011101","01110100","00011111","01001011","10111101","10001011","10001010",
 "01110000","00111110","10110101","01100110","01001000","00000011","11110110","00001110",
 "01100001","00110101","01010111","10111001","10000110","11000001","00011101","10011110",
 "11100001","11111000","10011000","00010001","01101001","11011001","10001110","10010100",
 "10011011","00011110","10000111","11101001","11001110","01010101","00101000","11011111",
 "10001100","10100001","10001001","00001101","10111111","11100110","01000010","01101000",
 "01000001","10011001","00101101","00001111","10110000","01010100","10111011","00010110" );
    variable q: u_sign32;
    begin
      q(1 to 8)   := fbsubtab(to_integer(a(1 to 8)));
      q(9 to 16)  := fbsubtab(to_integer(a(9 to 16)));
      q(17 to 24) := fbsubtab(to_integer(a(17 to 24)));
      q(25 to 32) := fbsubtab(to_integer(a(25 to 32)));
      return q;
   end fbsub_quad;
--=================================================================--
  function ftable(a :  in u_sign8) return u_sign32 is
  ---- moods inline
  constant ftabletab : rom_tab_5 :=
  -- moods rom
  (
  -- Hex(C6,63,63,A5), Hex(F8,7C,7C,84), Hex(EE,77,77,99), Hex(F6,7B,7B,8D)
   "11000110011000110110001110100101", "11111000011111000011111010000100",
 "11101110011101110111011110011001", "11110110011110110111101110001101",
```

```
  -- Hex(FF,F2,F2,0D), Hex(D6,6B,6B,BD), Hex(DE,6F,6F,B1), Hex(91,C5,C5,54)
   "11111111111100101111001000001101", "11010110011010110110101110111101",
  "11011110011011110110111110110001", "10010001110001011100010101010100",
   -- Hex(60,30,30,50), Hex(02,01,01,03), Hex(CE,67,67,A9), Hex(56,2B,2B,7D)
   "01100000001100000011000001010000", "00000010000000010000000100000011",
  "11001110011001110110011110101001", "01010110001010110010101101111101",
   -- Hex(E7,FE,FE,19), Hex(B5,D7,D7,62), Hex(4D,AB,AB,E6), Hex(EC,76,76,9A)
   "11100111111111101111111000011001", "10110101110101111101011101100010",
  "01001101101010111010101111100110", "11101100011101100111011010011010",
   -- Hex(8F,CA,CA,45), Hex(1F,82,82,9D), Hex(89,C9,C9,40), Hex(FA,7D,7D,87)
   "10001111110010101100101001000101", "00011111100000101000001010011101",
  "10001001110010011100100101000000", "11111010011111010111110110000111",
   -- Hex(EF,FA,FA,15), Hex(B2,59,59,EB), Hex(8E,47,47,C9), Hex(FB,F0,F0,0B)
   "11101111111110101111101000010101", "10110010010110010101100111101011",
  "10001110010001110100011111001001", "11111011111100001111000000001011",
   -- Hex(41,AD,AD,EC), Hex(B3,D4,D4,67), Hex(5F,A2,A2,FD), Hex(45,AF,AF,EA)
   "01000001101011011010110111101100", "10110011110101001101010001100111",
  "01011111101000101010001011111101", "01000101101011111010111111101010",
   -- Hex(23,9C,9C,BF), Hex(53,A4,A4,F7), Hex(E4,72,72,96), Hex(9B,C0,C0,5B)
   "00100011100111001001110010111111", "01010011101001001010010011110111",
  "11100100011100100111001010010110", "10011011110000001100000001011011",
   -- Hex(75,B7,B7,C2), Hex(E1,FD,FD,1C), Hex(3D,93,93,AE), Hex(4C,26,26,6A)
   "01110101101101111011011111000010", "11100001111111011111110100011100",
  "00111101100100111001001110101110", "01001100001001100010011001101010",
   -- Hex(6C,36,36,5A), Hex(7E,3F,3F,41), Hex(F5,F7,F7,02), Hex(83,CC,CC,4F)
   "01101100001101100011011001011010", "01111110001111110011111101000001",
  "11110101111101111111011100000010", "10000011110011001100110001001111",
   -- Hex(68,34,34,5C), Hex(51,A5,A5,F4), Hex(D1,E5,E5,34), Hex(F9,F1,F1,08)
   "01101000001101000011010001011100", "01010001101001011010010111110100",
  "11010001111001011110010100110100", "11111001111100011111000100001000",
   -- Hex(E2,71,71,93), Hex(AB,D8,D8,73), Hex(62,31,31,53), Hex(2A,15,15,3F)
   "11100010011100010111000110010011", "10101011110110001101100001110011",
  "01100010001100010011000101010011", "00101010000101010001010100111111",
   -- Hex(08,04,04,0C), Hex(95,C7,C7,52), Hex(46,23,23,65), Hex(9D,C3,C3,5E)
   "00001000000001000000010000001100", "10010101110001111100011101010010",
  "01000110001000110010001101100101", "10011101110000111100001101011110",
   -- Hex(30,18,18,28), Hex(37,96,96,A1), Hex(0A,05,05,0F), Hex(2F,9A,9A,B5)
   "00110000000110000001100000101000", "00110111100101101001011010100001",
  "00001010000001010000010100001111", "00101111100110101001101010110101",
   -- Hex(0E,07,07,09), Hex(24,12,12,36), Hex(1B,80,80,9B), Hex(DF,E2,E2,3D)
   "00001110000001110000011100001001", "00100100000100100001001000110110",
  "00011011100000001000000010011011", "11011111111000101110001000111101",
   -- Hex(CD,EB,EB,26), Hex(4E,27,27,69), Hex(7F,B2,B2,CD), Hex(EA,75,75,9F)
   "11001101111010111110101100100110", "01001110001001110010011101101001",
  "01111111101100101011001011001101", "11101010011101010111010110011111",
   -- Hex(12,09,09,1B), Hex(1D,83,83,9E), Hex(58,2C,2C,74), Hex(34,1A,1A,2E)
   "00010010000010010000100100011011", "00011101100000111000001110011110",
  "01011000001011000010110001110100", "00110100000110100001101000101110",
   -- Hex(36,1B,1B,2D), Hex(DC,6E,6E,B2), Hex(B4,5A,5A,EE), Hex(5B,A0,A0,FB)
   "00110110000110110001101100101101", "11011100011011100110111010110010",
  "10110100010110100101101011101110", "01011011101000001010000011111011",
   -- Hex(A4,52,52,F6), Hex(76,3B,3B,4D), Hex(B7,D6,D6,61), Hex(7D,B3,B3,CE)
   "10100100010100100101001011110110", "01110110001110110011101101001101",
  "10110111110101101101011001100001", "01111101101100111011001111001110",
   -- Hex(52,29,29,7B), Hex(DD,E3,E3,3E), Hex(5E,2F,2F,71), Hex(13,84,84,97)
   "01010010001010010010100101111011", "11011101111000111110001100111110",
  "01011110001011110010111101110001", "00010011100001001000010010010111",
   -- Hex(A6,53,53,F5), Hex(B9,D1,D1,68), Hex(00,00,00,00), Hex(C1,ED,ED,2C)
   "10100110010100110101001111110101", "10111001110100011101000101101000",
  "00000000000000000000000000000000", "11000001111011011110110100101100",
   -- Hex(40,20,20,60), Hex(E3,FC,FC,1F), Hex(79,B1,B1,C8), Hex(B6,5B,5B,ED)
   "01000000010000000100000011000000", "11100011111111100111111100000011111",
  "01111001101100011011000111001000", "10110110010110110101101111101101",
```

```
  -- Hex(D4,6A,6A,BE), Hex(8D,CB,CB,46), Hex(67,BE,BE,D9), Hex(72,39,39,4B)
   "11010100011010100110101010111110", "10001101110010111100101101000110",
  "01100111101111101011111011011001", "01110010001110010011100101001011",
   -- Hex(94,4A,4A,DE), Hex(98,4C,4C,D4), Hex(B0,58,58,E8), Hex(85,CF,CF,4A)
   "10010100010010100100101011011110", "10011000010011000100110011010100",
  "10110000010110000101100011101000", "10000101110011111100111101001010",
   -- Hex(BB,D0,D0,6B), Hex(C5,EF,EF,2A), Hex(4F,AA,AA,E5), Hex(ED,FB,FB,16),
   "10111011110100001101000001101011", "11000101111011111110111100101010",
  "01001111101010101010101011100101", "11101101111110111111101100010110",
   -- Hex(86,43,43,C5), Hex(9A,4D,4D,D7), Hex(66,33,33,55), Hex(11,85,85,94)
   "10000110010000110100001111000101", "10011010010011010100110111010111",
  "01100110001100110011001101010101", "00010001100001011000010110010100",
   -- Hex(8A,45,45,CF), Hex(E9,F9,F9,10), Hex(04,02,02,06), Hex(FE,7F,7F,81)
   "10001010010001010100010111001111", "11101001111110011111100100010000",
  "00000100000000010000000100000110", "11111110011111110111111110000001",
   -- Hex(A0,50,50,F0), Hex(78,3C,3C,44), Hex(25,9F,9F,BA), Hex(4B,A8,A8,E3)
   "10100000010100000101000011110000", "01111000001111000011110001000100",
  "00100101100111111001111110111010", "01001011101010001010100011100011",
   -- Hex(A2,51,51,F3), Hex(5D,A3,A3,FE), Hex(80,40,40,C0), Hex(05,8F,8F,8A)
   "10100010010100010101000111110011", "01011101101000111010001111111110",
  "10000000010000000100000011000000", "00000101100011111000111111000101 0",
   -- Hex(3F,92,92,AD), Hex(21,9D,9D,BC), Hex(70,38,38,48), Hex(F1,F5,F5,04)
   "00111111100100101001001010101101", "00100001100111011001110110111100",
  "01110000001110000011100001001000", "11110001111101011111010100000100",
   -- Hex(63,BC,BC,DF), Hex(77,B6,B6,C1), Hex(AF,DA,DA,75), Hex(42,21,21,63)
   "01100011101111001011110011011111", "01110111101101101011011011000001",
  "10101111110110101101101001110101", "01000010001000010010000101100011",
   -- Hex(20,10,10,30), Hex(E5,FF,FF,1A), Hex(FD,F3,F3,0E), Hex(BF,D2,D2,6D)
   "00100000000100000001000000110000", "11100101111111111111111111100011010",
  "11111101111100111111001100001110", "10111111110100101101001001101101",
   -- Hex(81,CD,CD,4C), Hex(18,0C,0C,14), Hex(26,13,13,35), Hex(C3,EC,EC,2F)
   "10000001110011011100110101001100", "00011000000011000000110000010100",
  "00100110000100110001001100110101", "11000011111011001110110000101111",
   -- Hex(BE,5F,5F,E1), Hex(35,97,97,A2), Hex(88,44,44,CC), Hex(2E,17,17,39)
   "10111110010111110101111111100001", "00110101100101111001011110100010",
  "10001000010001000100010011001100", "00101110000101110001011100111001",
   -- Hex(93,C4,C4,57), Hex(55,A7,A7,F2), Hex(FC,7E,7E,82), Hex(7A,3D,3D,47)
   "10010011110001001100010001010111", "01010101101001111010011111110010",
  "11111100011111100111111010000010", "01111010001111010011110101000111",
   -- Hex(C8,64,64,AC), Hex(BA,5D,5D,E7), Hex(32,19,19,2B), Hex(E6,73,73,95)
   "11001000011001000110010010101100", "10111010010111010101110111100111",
  "00110010000110010001100100101011", "11100110011100110111001110010101",
   -- Hex(C0,60,60,A0), Hex(19,81,81,98), Hex(9E,4F,4F,D1), Hex(A3,DC,DC,7F)
   "11000000011000000110000010100000", "00011001100000011000000110011000",
  "10011110010011110100111111010001", "10100011110111001101110001111111",
   -- Hex(44,22,22,66), Hex(54,2A,2A,7E), Hex(3B,90,90,AB), Hex(0B,88,88,83)
   "01000100001000100010001001100110", "01010100001010100010101001111110",
  "00111011100100001001000010101011", "00001011100010001000100010000011",
   -- Hex(8C,46,46,CA), Hex(C7,EE,EE,29), Hex(6B,B8,B8,D3), Hex(28,14,14,3C)
   "10001100010001100100011011001010", "11000111111011101110111000101001",
  "01101011101110001011100011010011", "00101000000101000001010000111100",
   -- Hex(A7,DE,DE,79), Hex(BC,5E,5E,E2), Hex(16,0B,0B,1D), Hex(AD,DB,DB,76)
   "10100111110111101101111001111001", "10111100010111100101111011100010",
  "00010110000001011000010110001101", "10101011101101111011011011110110",
   -- Hex(DB,E0,E0,3B), Hex(64,32,32,56), Hex(74,3A,3A,4E), Hex(14,0A,0A,1E)
   "11011011111000001110000000111011", "01100100001100100011001001010110",
  "01110100001110100011101001001110", "00010100000010100000101000011110",
   -- Hex(92,49,49,DB), Hex(0C,06,06,0A), Hex(48,24,24,6C), Hex(B8,5C,5C,E4)
   "10010010010010010100100111011011", "00001100000001100000011000001010",
  "01001000001001000010010001101100", "10111000010111000101110011100100",
   -- Hex(9F,C2,C2,5D), Hex(BD,D3,D3,6E), Hex(43,AC,AC,EF), Hex(C4,62,62,A6)
   "10011111110000101100001001011101", "10111101110100111101001101101110",
  "01000011101011001010110011101111", "11000100011000100110001010100110",
```

```
  -- Hex(39,91,91,A8), Hex(31,95,95,A4), Hex(D3,E4,E4,37), Hex(F2,79,79,8B)
   "00111001100100011001000110101000", "00110001100101011001010110100100",
  "11010011111001001110010000110111", "11110010011110010111100110001011",
   -- Hex(D5,E7,E7,32), Hex(8B,C8,C8,43), Hex(6E,37,37,59), Hex(DA,6D,6D,B7)
   "11010101111001111111001100110010", "10001011110010001100100001000011",
  "01101110001101110011011101011001", "11011010011011010110110110110111",
   -- Hex(01,8D,8D,8C), Hex(B1,D5,D5,64), Hex(9C,4E,4E,D2), Hex(49,A9,A9,E0)
   "00000001100011011000110110001100", "10110001110101011101010101100100",
  "10011100010011100100111011010010", "01001001101010011010100111100000",
   -- Hex(D8,6C,6C,B4), Hex(AC,56,56,FA), Hex(F3,F4,F4,07), Hex(CF,EA,EA,25)
   "11011000011011000110110010110100", "10101100010101100101011011111010",
  "11110011111101001111010000000111", "11001111111010101111010100100101",
   -- Hex(CA,65,65,AF), Hex(F4,7A,7A,8E), Hex(47,AE,AE,E9), Hex(10,08,08,18)
   "11001010011001010110010110101111", "11110100011110100111101010001110",
  "01000111101011101010111011101001", "00010000000010000000100000011000",
   -- Hex(6F,BA,BA,D5), Hex(F0,78,78,88), Hex(4A,25,25,6F), Hex(5C,2E,2E,72)
   "01101111101110101011101011010101", "11110000011110000111100010001000",
  "01001010001001010010010101101111", "01011100001011100010111001110010",
   -- Hex(38,1C,1C,24), Hex(57,A6,A6,F1), Hex(73,B4,B4,C7), Hex(97,C6,C6,51)
   "00111000000111000001110000100100", "01010111010011010100111011110001",
  "01110011101101001011010011000111", "10010111110001101100011001010001",
   -- Hex(CB,E8,E8,23), Hex(A1,DD,DD,7C), Hex(E8,74,74,9C), Hex(3E,1F,1F,21)
   "11001011111010001110100000100011", "10100001110111011101110101111100",
  "11101000011101000111010010011100", "00111110000111110001111100100001",
   -- Hex(96,4B,4B,DD), Hex(61,BD,BD,DC), Hex(0D,8B,8B,86), Hex(0F,8A,8A,85)
   "10010110010010110100101111011101", "01100001101111011011110111011100",
  "00001101100010111000010111000000110", "00001111100010101000101010000101",
   -- Hex(E0,70,70,90), Hex(7C,3E,3E,42), Hex(71,B5,B5,C4), Hex(CC,66,66,AA)
   "11100000011100000111000010010000", "01111100001111100011111001000010",
  "01110001101101011011010111000100", "11001100011001100110011010101010",
   -- Hex(90,48,48,D8), Hex(06,03,03,05), Hex(F7,F6,F6,01), Hex(1C,0E,0E,12)
   "10010000010010000100100011011000", "00000110000000011000000011000000101",
  "11110111111101101111011000000001", "00011100000011100000111000010010",
   -- Hex(C2,61,61,A3), Hex(6A,35,35,5F), Hex(AE,57,57,F9), Hex(69,B9,B9,D0)
   "11000010011000010110000110100011", "01101010001101010011010101011111",
  "10101110010101110101011111111001", "01101001101110011011100111010000",
   -- Hex(17,86,86,91), Hex(99,C1,C1,58), Hex(3A,1D,1D,27), Hex(27,9E,9E,B9)
   "00010111100001101000011010010001", "10011001110000011100000101011000",
  "00111010000111010001110100100111", "00100111100111101001111010111001",
   -- Hex(D9,E1,E1,38), Hex(EB,F8,F8,13), Hex(2B,98,98,B3), Hex(22,11,11,33)
   "11011001111000011110000100111000", "11101011111110001111100000010011",
  "00101011100110001001100010110011", "00100010000100010001000100110011",
   -- Hex(D2,69,69,BB), Hex(A9,D9,D9,70), Hex(07,8E,8E,89), Hex(33,94,94,A7)
   "11010010011010010110100110111011", "10101001110110011101100101110000",
  "00000111100011101000111010001001", "00110011100101001001010010100111",
   -- Hex(2D,9B,9B,B6), Hex(3C,1E,1E,22), Hex(15,87,87,92), Hex(C9,E9,E9,20)
   "00101101100110111001101110110110", "00111100000111100001111000100010",
  "00010101100001111000011110010010", "11001001111010011110100100100000",
   -- Hex(87,CE,CE,49), Hex(AA,55,55,FF), Hex(50,28,28,78), Hex(A5,DF,DF,7A)
   "10000111110011101100111001001001", "10101010010101010101010111111111",
  "01010000001010000010100001111000", "10100101110111111110111111011111010",
   -- Hex(03,8C,8C,8F), Hex(59,A1,A1,F8), Hex(09,89,89,80), Hex(1A,0D,0D,17)
   "00000011100011001000110010001111", "01011001101000011010000111111000",
  "00001001100010011000100110000000", "00011010000011010000110100010111",
   -- Hex(65,BF,BF,DA), Hex(D7,E6,E6,31), Hex(84,42,42,C6), Hex(D0,68,68,B8)
   "01100101101111111101111111011010", "11010111111001101110011000110001",
  "10000100010000100100001011000110", "11010000011010000110100010111000",
   -- Hex(82,41,41,C3), Hex(29,99,99,B0), Hex(5A,2D,2D,77), Hex(1E,0F,0F,11)
   "10000010010000001010000011100011", "00101001100110011001100110110000",
  "01011010001011010010110101110111", "00011110000011110000111100010001",
   -- Hex(7B,B0,B0,CB), Hex(A8,54,54,FC), Hex(6D,BB,BB,D6), Hex(2C,16,16,3A)
   "01111011101100001011000011001011", "10101000010101000101010011111100",
  "01101101101110111011101111010110", "00101100000101100001011000111010" );
```

```
  variable b  : natural range 0 to 255;
  variable q : u_sign32;
  begin

     b := to_integer(a);
     q := ftabletab(b);
     return q;

  end ftable;
--=================================================================---
    function ftable_double( a, b :  in u_sign8 ) return u_sign64 is
  --  -- moods inline
    constant ftabletab : rom_tab_5 :=
  -- moods rom
  (
  -- Hex(C6,63,63,A5), Hex(F8,7C,7C,84), Hex(EE,77,77,99), Hex(F6,7B,7B,8D)
   "11000110011000110110001110100101", "11111000011111000111110010000100",
 "11101110011101110111011110011001", "11110110011110110111101110001101",
   -- Hex(FF,F2,F2,0D), Hex(D6,6B,6B,BD), Hex(DE,6F,6F,B1), Hex(91,C5,C5,54)
   "11111111111100101111001000001101", "11010110011010110110101110111101",
 "11011110011011110110111110110001", "10010001110001011100010101010100",
   -- Hex(60,30,30,50), Hex(02,01,01,03), Hex(CE,67,67,A9), Hex(56,2B,2B,7D)
   "01100000001100000011000001010000", "00000010000000010000000100000011",
 "11001110011001110110011110101001", "01010110001010110010101101111101",
   -- Hex(E7,FE,FE,19), Hex(B5,D7,D7,62), Hex(4D,AB,AB,E6), Hex(EC,76,76,9A)
   "11100111111111101111111000011001", "10110101110101111101011101100010",
 "01001101101010111010101111100110", "11101100011101100111011010011010",
   -- Hex(8F,CA,CA,45), Hex(1F,82,82,9D), Hex(89,C9,C9,40), Hex(FA,7D,7D,87)
   "10001111110010101100101001000101", "00011111100000101000001010011101",
 "10001001110010011100100101000000", "11111010011111010111110110000111",
   -- Hex(EF,FA,FA,15), Hex(B2,59,59,EB), Hex(8E,47,47,C9), Hex(FB,F0,F0,0B)
   "11101111111110101111101000010101", "10110010010110010101100111101011",
 "10001110010001110100011111001001", "11111011111100001111000000001011",
   -- Hex(41,AD,AD,EC), Hex(B3,D4,D4,67), Hex(5F,A2,A2,FD), Hex(45,AF,AF,EA)
   "01000001101011011010110111101100", "10110011110101001101010001100111",
 "01011111101000101010001011111101", "01000101101011111010111111101010",
   -- Hex(23,9C,9C,BF), Hex(53,A4,A4,F7), Hex(E4,72,72,96), Hex(9B,C0,C0,5B)
   "00100011100111001001110010111111", "01010011101001001010010011110111",
 "11100100011100100111001010010110", "10011011110000001100000001011011",
   -- Hex(75,B7,B7,C2), Hex(E1,FD,FD,1C), Hex(3D,93,93,AE), Hex(4C,26,26,6A)
   "01110101101101111011011111000010", "11100001111111011111110100011100",
 "00111101100100111001001110101110", "01001100001001100010011001101010",
   -- Hex(6C,36,36,5A), Hex(7E,3F,3F,41), Hex(F5,F7,F7,02), Hex(83,CC,CC,4F)
   "01101100001101100011011001011010", "01111110001111110011111101000001",
 "11110101111101111111011100000010", "10000011110011001100110010001001111",
   -- Hex(68,34,34,5C), Hex(51,A5,A5,F4), Hex(D1,E5,E5,34), Hex(F9,F1,F1,08)
   "01101000001101000011010001011100", "01010001101001011010010111110100",
 "11010001111001011110010100110100", "11111001111110001111100010001000",
   -- Hex(E2,71,71,93), Hex(AB,D8,D8,73), Hex(62,31,31,53), Hex(2A,15,15,3F)
   "11100010011100010111000110010011", "10101011110110001101100001110011",
 "01100010001100010011000101010011", "00101010000101010001010100111111",
   -- Hex(08,04,04,0C), Hex(95,C7,C7,52), Hex(46,23,23,65), Hex(9D,C3,C3,5E)
   "00001000000001000000010000001100", "10010101110001111100011101010010",
 "01000110001000110010001101100101", "10011101110000111100001101011110",
   -- Hex(30,18,18,28), Hex(37,96,96,A1), Hex(0A,05,05,0F), Hex(2F,9A,9A,B5)
   "00110000000110000001100000101000", "00110111100101101001011010100001",
 "00001010000001010000010100001111", "00101111100110101001101010110101",
   -- Hex(0E,07,07,09), Hex(24,12,12,36), Hex(1B,80,80,9B), Hex(DF,E2,E2,3D)
   "00001110000001110000011100001001", "00100100000100100001001000110110",
 "00011011100000001000000010011011", "11011111111000101110001000111101",
   -- Hex(CD,EB,EB,26), Hex(4E,27,27,69), Hex(7F,B2,B2,CD), Hex(EA,75,75,9F)
   "11001101111010111110101100100110", "01001110001001110010011101101001",
 "01111111101100101011001011001101", "11101010011101010111010110011111",
```

```
  -- Hex(12,09,09,1B), Hex(1D,83,83,9E), Hex(58,2C,2C,74), Hex(34,1A,1A,2E)
   "00010010000010010000100100011011", "00011101100000111000001110011110",
  "01011000001011000010110001110100", "00110100000110100001101000101110",
   -- Hex(36,1B,1B,2D), Hex(DC,6E,6E,B2), Hex(B4,5A,5A,EE), Hex(5B,A0,A0,FB)
   "00110110000110110001101100101101", "11011100011011100110111010110010",
  "10110100010110100101101011101110", "01011011101000001010000011111011",
   -- Hex(A4,52,52,F6), Hex(76,3B,3B,4D), Hex(B7,D6,D6,61), Hex(7D,B3,B3,CE)
   "10100100010100100101001011110110", "01110110001110110011101101001101",
  "10110111110101101101011001100001", "01111101101100111011001111001110",
   -- Hex(52,29,29,7B), Hex(DD,E3,E3,3E), Hex(5E,2F,2F,71), Hex(13,84,84,97)
   "01010010001010010010100101111011", "11011101111000111110001100111110",
  "01011110001011110010111101110001", "00010011100001001000010010010111",
   -- Hex(A6,53,53,F5), Hex(B9,D1,D1,68), Hex(00,00,00,00), Hex(C1,ED,ED,2C)
   "10100110010100110101001111110101", "10111001110100011101000101101000",
  "00000000000000000000000000000000", "11000001111011011111011010010 1100",
   -- Hex(40,20,20,60), Hex(E3,FC,FC,1F), Hex(79,B1,B1,C8), Hex(B6,5B,5B,ED)
   "01000000001000000010000001100000", "11100011111111100111111100000011111",
  "01111001101100011011000111001000", "10110110010110110101101111101101",
   -- Hex(D4,6A,6A,BE), Hex(8D,CB,CB,46), Hex(67,BE,BE,D9), Hex(72,39,39,4B)
   "11010100011010100110101010111110", "10001101110010111100101101000110",
  "01100111101111101011111011011001", "01110010001110010011100101001011",
   -- Hex(94,4A,4A,DE), Hex(98,4C,4C,D4), Hex(B0,58,58,E8), Hex(85,CF,CF,4A)
   "10010100010010100100101011011110", "10011000010011000100110011010100",
  "10110000010110000101100011101000", "10000101110011111100111101001010",
   -- Hex(BB,D0,D0,6B), Hex(C5,EF,EF,2A), Hex(4F,AA,AA,E5), Hex(ED,FB,FB,16),
   "10111011110100001101000001101011", "11000101111011111110111100101010",
  "01001111101010101010101011100101", "11101101111110111111101100010110",
   -- Hex(86,43,43,C5), Hex(9A,4D,4D,D7), Hex(66,33,33,55), Hex(11,85,85,94)
   "10000110010000110100001111000101", "10011010010011010100110111010111",
  "01100110001100110011001101010101", "00010001100001011000010110010100",
   -- Hex(8A,45,45,CF), Hex(E9,F9,F9,10), Hex(04,02,02,06), Hex(FE,7F,7F,81)
   "10001010010001010100010111001111", "11101001111110011111100100010000",
  "00000100000000010000000100000110", "11111110011111111011111111110000001",
   -- Hex(A0,50,50,F0), Hex(78,3C,3C,44), Hex(25,9F,9F,BA), Hex(4B,A8,A8,E3)
   "10100000010100000101000011110000", "01111000001111000011110001000100",
  "00100101100111111001111110111010", "01001011101010001010100011100011",
   -- Hex(A2,51,51,F3), Hex(5D,A3,A3,FE), Hex(80,40,40,C0), Hex(05,8F,8F,8A)
   "10100010010100010101000111110011", "01011101101000111010001111111110",
  "10000000010000000100000011000000", "00000101100011111000111110001010",
   -- Hex(3F,92,92,AD), Hex(21,9D,9D,BC), Hex(70,38,38,48), Hex(F1,F5,F5,04)
   "00111111100100101001001010101101", "00100001100111011001110110111100",
  "01110000001110000011100001001000", "11110001111101011111010100000100",
   -- Hex(63,BC,BC,DF), Hex(77,B6,B6,C1), Hex(AF,DA,DA,75), Hex(42,21,21,63)
   "01100011101111001011110011011111", "01110111101101101011011011000001",
  "10101111110110101101101001110101", "01000010001000010010000101100011",
   -- Hex(20,10,10,30), Hex(E5,FF,FF,1A), Hex(FD,F3,F3,0E), Hex(BF,D2,D2,6D)
   "00100000000100000001000000110000", "11100101111111111111111100011010",
  "11111101111100111111001100001110", "10111111110100101101001001101101",
   -- Hex(81,CD,CD,4C), Hex(18,0C,0C,14), Hex(26,13,13,35), Hex(C3,EC,EC,2F)
   "10000001110011011100110101001100", "00011000000011000000110000010100",
  "00100110000100110001001100110101", "11000011111011001110110000101111",
   -- Hex(BE,5F,5F,E1), Hex(35,97,97,A2), Hex(88,44,44,CC), Hex(2E,17,17,39)
   "10111110010111110101111111100001", "00110101100101111001011110100010",
  "10001000010001000100010011001100", "00101110000101110001011100111001",
   -- Hex(93,C4,C4,57), Hex(55,A7,A7,F2), Hex(FC,7E,7E,82), Hex(7A,3D,3D,47)
   "10010011110001001100010001010111", "01010101101001111010011111110010",
  "11111100011111100111111010000010", "01111010001111010011110101000111",
   -- Hex(C8,64,64,AC), Hex(BA,5D,5D,E7), Hex(32,19,19,2B), Hex(E6,73,73,95)
   "11001000011001000110010010101100", "10111010010111010101110111100111",
  "00110010000110010001100100101011", "11100110011100110111001110010101",
   -- Hex(C0,60,60,A0), Hex(19,81,81,98), Hex(9E,4F,4F,D1), Hex(A3,DC,DC,7F)
   "11000000011000000110000010100000", "00011001100000011000000110011000",
  "10011110010011110100111111010001", "10100011110111001101110001111111",
```

```
  -- Hex(44,22,22,66), Hex(54,2A,2A,7E), Hex(3B,90,90,AB), Hex(0B,88,88,83)
   "01000100001000100010001001100110", "01010100001010100010101001111110",
  "00111011100100001001000010101011", "00001011100010001000100010000011",
   -- Hex(8C,46,46,CA), Hex(C7,EE,EE,29), Hex(6B,B8,B8,D3), Hex(28,14,14,3C)
   "10001100010001100100011011001010", "11000111111011101110111000101001",
  "01101011101110001011100011010011", "00101000000101000001010000111100",
   -- Hex(A7,DE,DE,79), Hex(BC,5E,5E,E2), Hex(16,0B,0B,1D), Hex(AD,DB,DB,76)
   "10100111110111101101111001111001", "10111100010111100101111011100010",
  "00010110000010110000101100011101", "10101101110110111101101101110110",
   -- Hex(DB,E0,E0,3B), Hex(64,32,32,56), Hex(74,3A,3A,4E), Hex(14,0A,0A,1E)
   "11011011111000001110000000111011", "01100100001100100011001001010110",
  "01110100001110100011101001001110", "00010100000010100000101000011110",
   -- Hex(92,49,49,DB), Hex(0C,06,06,0A), Hex(48,24,24,6C), Hex(B8,5C,5C,E4)
   "10010010010010010100100111011011", "00001100000000110000001100000101 0",
  "01001000001001000010010001101100", "10111000010111000101110011100100",
   -- Hex(9F,C2,C2,5D), Hex(BD,D3,D3,6E), Hex(43,AC,AC,EF), Hex(C4,62,62,A6)
   "10011111110000101100001001011101", "10111101110100111101001101101110",
  "01000011101011001010110011101111", "11000100011000100110001010100110",
   -- Hex(39,91,91,A8), Hex(31,95,95,A4), Hex(D3,E4,E4,37), Hex(F2,79,79,8B)
   "00111001100100011001000110101000", "00110001100101011001010110100100",
  "11010011111001001110010000110111", "11110010011110010111100110001011",
   -- Hex(D5,E7,E7,32), Hex(8B,C8,C8,43), Hex(6E,37,37,59), Hex(DA,6D,6D,B7)
   "11010101111001111110011100110010", "10001011110010001100100001000011",
  "01101110001101110011011101011001", "11011010011011010110110110110111",
   -- Hex(01,8D,8D,8C), Hex(B1,D5,D5,64), Hex(9C,4E,4E,D2), Hex(49,A9,A9,E0)
   "00000001100011011000110110001100", "10110001110101011101010101100100",
  "10011100010011100100111011010010", "01001001101010011010100111100000",
   -- Hex(D8,6C,6C,B4), Hex(AC,56,56,FA), Hex(F3,F4,F4,07), Hex(CF,EA,EA,25)
   "11011000011011000110110010110100", "10101100010101100101011011111010",
  "11110011111101001111010000000111", "11001111111010101110101000100101",
   -- Hex(CA,65,65,AF), Hex(F4,7A,7A,8E), Hex(47,AE,AE,E9), Hex(10,08,08,18)
   "11001010011001010110010110101111", "11110100011110100111101010001110",
  "01000111101011101010111011101001", "00010000000010000000100000011000",
   -- Hex(6F,BA,BA,D5), Hex(F0,78,78,88), Hex(4A,25,25,6F), Hex(5C,2E,2E,72)
   "01101111101110101011101011010101", "11110000011110000111100010001000",
  "01001010001001010010010101101111", "01011100001011100010111001110010",
   -- Hex(38,1C,1C,24), Hex(57,A6,A6,F1), Hex(73,B4,B4,C7), Hex(97,C6,C6,51)
   "00111000000111000001110000100100", "01010111101001101010011011110001",
  "01110011101101001011010011000111", "10010111110001101100011001010001",
   -- Hex(CB,E8,E8,23), Hex(A1,DD,DD,7C), Hex(E8,74,74,9C), Hex(3E,1F,1F,21)
   "11001011111010001110100000100011", "10100001110111011101110101111100",
  "11101000011101000111010010011100", "00111110000111110001111100100001",
   -- Hex(96,4B,4B,DD), Hex(61,BD,BD,DC), Hex(0D,8B,8B,86), Hex(0F,8A,8A,85)
   "10010110010010110100101111011101", "01100001101111011011110111011100",
  "00001101100010111000101110000110", "00001111100010101000101010000101",
   -- Hex(E0,70,70,90), Hex(7C,3E,3E,42), Hex(71,B5,B5,C4), Hex(CC,66,66,AA)
   "11100000011100000111000010010000", "01111100001111100011111001000010",
  "01110001101101011011010111000100", "11001100011001100110011010101010",
   -- Hex(90,48,48,D8), Hex(06,03,03,05), Hex(F7,F6,F6,01), Hex(1C,0E,0E,12)
   "10010000010010000100100011011000", "00000110000000011000000110000101",
  "11110111111101101111011000000001", "00011100000011100000111000010010",
   -- Hex(C2,61,61,A3), Hex(6A,35,35,5F), Hex(AE,57,57,F9), Hex(69,B9,B9,D0)
   "11000010011000010110000110100011", "01101010001101010011010101011111",
  "10101110010101110101011111111001", "01101001101110011011100111010000",
   -- Hex(17,86,86,91), Hex(99,C1,C1,58), Hex(3A,1D,1D,27), Hex(27,9E,9E,B9)
   "00010111100001101000011010010001", "10011001110000011100000101011000",
  "00111010000111010001110100100111", "00100111100111101001111010111001",
   -- Hex(D9,E1,E1,38), Hex(EB,F8,F8,13), Hex(2B,98,98,B3), Hex(22,11,11,33)
   "11011001111000011110000100111000", "11101011111110001111100000010011",
  "00101011100110001001100010110011", "00100010000100010001000100110011",
   -- Hex(D2,69,69,BB), Hex(A9,D9,D9,70), Hex(07,8E,8E,89), Hex(33,94,94,A7)
   "11010010011010010110100110111011", "10101001110110011101100101110000",
  "00000111100011101000111010001001", "00110011100101001001010010100111",
```

```
  -- Hex(2D,9B,9B,B6), Hex(3C,1E,1E,22), Hex(15,87,87,92), Hex(C9,E9,E9,20)
  "00101101100110111001101110110110", "00111100000111100001111000100010",
 "00010101100001111000011110010010", "11001001111010011110100100100000",
  -- Hex(87,CE,CE,49), Hex(AA,55,55,FF), Hex(50,28,28,78), Hex(A5,DF,DF,7A)
  "10000111110011101100111001001001", "10101010010101010101010111111111",
 "01010000001010000010100001111000", "10100101110111111101111101111010",
  -- Hex(03,8C,8C,8F), Hex(59,A1,A1,F8), Hex(09,89,89,80), Hex(1A,0D,0D,17)
  "00000011100011001000110010001111", "01011001101000011010000111111000",
 "00001001100010011000100110000000", "00011010000011010000110100010111",
  -- Hex(65,BF,BF,DA), Hex(D7,E6,E6,31), Hex(84,42,42,C6), Hex(D0,68,68,B8)
  "01100101101111111101111111111011010", "11010111111001101110011000110001",
 "10000100010000100100001011000110", "11010000011010000110100010111000",
  -- Hex(82,41,41,C3), Hex(29,99,99,B0), Hex(5A,2D,2D,77), Hex(1E,0F,0F,11)
  "10000010010000001010000011000011", "00101001100110011001100110110110000",
 "01011010001011010010110101110111", "00011110000011110000111100010001",
  -- Hex(7B,B0,B0,CB), Hex(A8,54,54,FC), Hex(6D,BB,BB,D6), Hex(2C,16,16,3A)
  "01111011101100001011000011001011", "10101000010101000101010011111100",
 "01101101101110111011101111010110", "00101100000101100001011000111010"
  );
     variable c  : natural range 0 to 255;
     variable r : u_sign32;
     variable q : u_sign64;
   begin
     c := to_integer(a);
     r := ftabletab(c);
     c := to_integer(b);
     q := ftabletab(c) & r;
     return q;

   end ftable_double;
--=====================================================================--
  procedure ftable_quad (
  ---- moods inline
    a:       in u_sign32;
    q_out:   out u_sign32
  ) is
  constant ftabletab : rom_tab_5 :=
  -- moods rom
  (
  -- Hex(C6,63,63,A5), Hex(F8,7C,7C,84), Hex(EE,77,77,99), Hex(F6,7B,7B,8D)
  "11000110011000110110001110100101", "11111000011111000111110010000100",
 "11101110011101110111011110011001", "11110110011110110111101110001101",
  -- Hex(FF,F2,F2,0D), Hex(D6,6B,6B,BD), Hex(DE,6F,6F,B1), Hex(91,C5,C5,54)
  "11111111111100101111001000001101", "11010110011010110110101110111101",
 "11011110011011110110111110110001", "10010001110001011100010101010100",
  -- Hex(60,30,30,50), Hex(02,01,01,03), Hex(CE,67,67,A9), Hex(56,2B,2B,7D)
  "01100000001100000011000001010000", "00000010000000010000000100000011",
 "11001110011001110110011110101001", "01010110001010110010101101111101",
  -- Hex(E7,FE,FE,19), Hex(B5,D7,D7,62), Hex(4D,AB,AB,E6), Hex(EC,76,76,9A)
  "11100111111111110111111000011001", "10110101110101111101011101100010",
 "01001101101010111010101111100110", "11101100011101100111011010011010",
  -- Hex(8F,CA,CA,45), Hex(1F,82,82,9D), Hex(89,C9,C9,40), Hex(FA,7D,7D,87)
  "10001111110010101100101001000101", "00011111100000101000001010011101",
 "10001001110010011100100101000000", "11111010011111101011111010000111",
  -- Hex(EF,FA,FA,15), Hex(B2,59,59,EB), Hex(8E,47,47,C9), Hex(FB,F0,F0,0B)
  "11101111111110101111101000010101", "10110010010110010101100111101011",
 "10001110010001110100011111001001", "11111011111100001111000000001011",
  -- Hex(41,AD,AD,EC), Hex(B3,D4,D4,67), Hex(5F,A2,A2,FD), Hex(45,AF,AF,EA)
  "01000001101011011010110111101100", "10110011110101001101010001100111",
 "01011111101000101010001011111101", "01000101101011111101011111101010",
  -- Hex(23,9C,9C,BF), Hex(53,A4,A4,F7), Hex(E4,72,72,96), Hex(9B,C0,C0,5B)
  "00100011100111001001110010111111", "01010011101001001010010011110111",
 "11100100011100100111001010010110", "10011011110000001100000001011011",
```

```
  -- Hex(75,B7,B7,C2), Hex(E1,FD,FD,1C), Hex(3D,93,93,AE), Hex(4C,26,26,6A)
   "01110101101101111011011111000010", "11100001111111011111110100011100",
  "00111101100100111001001110101110", "01001100001001100010011001101010",
   -- Hex(6C,36,36,5A), Hex(7E,3F,3F,41), Hex(F5,F7,F7,02), Hex(83,CC,CC,4F)
   "01101100001101100011011001011010", "01111110001111110011111101000001",
  "11110101111101111111011100000010", "10000011110011001100110001001111",
   -- Hex(68,34,34,5C), Hex(51,A5,A5,F4), Hex(D1,E5,E5,34), Hex(F9,F1,F1,08)
   "01101000001101000011010001011100", "01010001101001011010010111110100",
  "11010001111001011110010100110100", "11111001111100011111000100001000",
   -- Hex(E2,71,71,93), Hex(AB,D8,D8,73), Hex(62,31,31,53), Hex(2A,15,15,3F)
   "11100010011100010111000110010011", "10101011110110001101100001110011",
  "01100010001100010011000101010011", "00101010000101010001010100111111",
   -- Hex(08,04,04,0C), Hex(95,C7,C7,52), Hex(46,23,23,65), Hex(9D,C3,C3,5E)
   "00001000000001000000010000001100", "10010101110001111100011101010010",
  "01000110001000110010001101100101", "10011101110000111100001101011110",
   -- Hex(30,18,18,28), Hex(37,96,96,A1), Hex(0A,05,05,0F), Hex(2F,9A,9A,B5)
   "00110000000110000001100000101000", "00110111100101101001011010100001",
  "00001010000001010000010100001111", "00101111100110101001101010110101",
   -- Hex(0E,07,07,09), Hex(24,12,12,36), Hex(1B,80,80,9B), Hex(DF,E2,E2,3D)
   "00001110000001110000011100001001", "00100100000100100001001000110110",
  "00011011100000001000000010011011", "11011111111000101110001000111101",
   -- Hex(CD,EB,EB,26), Hex(4E,27,27,69), Hex(7F,B2,B2,CD), Hex(EA,75,75,9F)
   "11001101111010111110101100100110", "01001110001001110010011101101001",
  "01111111101100101011001011001101", "11101010011101010111010110011111",
   -- Hex(12,09,09,1B), Hex(1D,83,83,9E), Hex(58,2C,2C,74), Hex(34,1A,1A,2E)
   "00010010000010010000100100011011", "00011101100000111000001110011110",
  "01011000001011000010110001110100", "00110100000110100001101000101110",
   -- Hex(36,1B,1B,2D), Hex(DC,6E,6E,B2), Hex(B4,5A,5A,EE), Hex(5B,A0,A0,FB)
   "00110110000110110001101100101101", "11011100011011100110111010110010",
  "10110100010110100101101011101110", "01011011101000001010000011111011",
   -- Hex(A4,52,52,F6), Hex(76,3B,3B,4D), Hex(B7,D6,D6,61), Hex(7D,B3,B3,CE)
   "10100100010100100101001011110110", "01110110001110110011101101001101",
  "10110111110101101101011001100001", "01111101101100111011001111001110",
   -- Hex(52,29,29,7B), Hex(DD,E3,E3,3E), Hex(5E,2F,2F,71), Hex(13,84,84,97)
   "01010010001010010010100101111011", "11011101111000111110001100111110",
  "01011110001011110010111101110001", "00010011100001001000010010010111",
   -- Hex(A6,53,53,F5), Hex(B9,D1,D1,68), Hex(00,00,00,00), Hex(C1,ED,ED,2C)
   "10100110010100110101001111110101", "10111001110100011101000101101000",
  "00000000000000000000000000000000", "11000001111011011110110100101100",
   -- Hex(40,20,20,60), Hex(E3,FC,FC,1F), Hex(79,B1,B1,C8), Hex(B6,5B,5B,ED)
   "01000000001000000010000001100000", "11100011111111001111110000011111",
  "01111001101100011011000111001000", "10110110010110110101101111101101",
   -- Hex(D4,6A,6A,BE), Hex(8D,CB,CB,46), Hex(67,BE,BE,D9), Hex(72,39,39,4B)
   "11010100011010100110101010111110", "10001101110010111100101101000110",
  "01100111101111101011111011011001", "01110010001110010011100101001011",
   -- Hex(94,4A,4A,DE), Hex(98,4C,4C,D4), Hex(B0,58,58,E8), Hex(85,CF,CF,4A)
   "10010100010010100100101011011110", "10011000010011000100110011010100",
  "10110000010110000101100011101000", "10000101110011111100111101001010",
   -- Hex(BB,D0,D0,6B), Hex(C5,EF,EF,2A), Hex(4F,AA,AA,E5), Hex(ED,FB,FB,16),
   "10111011110100001101000001101011", "11000101111011111110111100101010",
  "01001111101010101010101011100101", "11101101111110111111101100010110",
   -- Hex(86,43,43,C5), Hex(9A,4D,4D,D7), Hex(66,33,33,55), Hex(11,85,85,94)
   "10000110010000110100001111000101", "10011010010011010100110111010111",
  "01100110001100110011001101010101", "00010001100001011000010110010100",
   -- Hex(8A,45,45,CF), Hex(E9,F9,F9,10), Hex(04,02,02,06), Hex(FE,7F,7F,81)
   "10001010010001010100010111001111", "11101001111110011111100100010000",
  "00000100000000010000000100000110", "11111110011111110111111110000001",
   -- Hex(A0,50,50,F0), Hex(78,3C,3C,44), Hex(25,9F,9F,BA), Hex(4B,A8,A8,E3)
   "10100000010100000101000011110000", "01111000001111000011110001000100",
  "00100101100111111001111110111010", "01001011101010001010100011100011",
   -- Hex(A2,51,51,F3), Hex(5D,A3,A3,FE), Hex(80,40,40,C0), Hex(05,8F,8F,8A)
   "10100010010100010101000111110011", "01011101101000111010001111111110",
  "10000000010000000100000011000000", "00000101100011111000111110001010",
```

```
  -- Hex(3F,92,92,AD), Hex(21,9D,9D,BC), Hex(70,38,38,48), Hex(F1,F5,F5,04)
   "00111111100100101001001010101101", "00100001100111011001110110111100",
  "01110000001110000011100001001000", "11110001111101011111010100000100",
   -- Hex(63,BC,BC,DF), Hex(77,B6,B6,C1), Hex(AF,DA,DA,75), Hex(42,21,21,63)
   "01100011101111001011110011011111", "01110111101101101011011011000001",
  "10101111110110101101101001110101", "01000010001000010010000101100011",
   -- Hex(20,10,10,30), Hex(E5,FF,FF,1A), Hex(FD,F3,F3,0E), Hex(BF,D2,D2,6D)
   "00100000000100000001000000110000", "11100101111111111111111100011010",
  "11111101111100111111001100001110", "10111111110100101101001001101101",
   -- Hex(81,CD,CD,4C), Hex(18,0C,0C,14), Hex(26,13,13,35), Hex(C3,EC,EC,2F)
   "10000001110011011001101010011100", "00011000000011000000110000010100",
  "00100110000100110001001100110101", "11000011111011001110110000101111",
   -- Hex(BE,5F,5F,E1), Hex(35,97,97,A2), Hex(88,44,44,CC), Hex(2E,17,17,39)
   "10111110010111110101111111100001", "00110101100101111001011110100010",
  "10001000010001000100010011001100", "00101110000101110001011100111001",
   -- Hex(93,C4,C4,57), Hex(55,A7,A7,F2), Hex(FC,7E,7E,82), Hex(7A,3D,3D,47)
   "10010011110001001100010001010111", "01010101101001111010011111110010",
  "11111100011111100111111010000010", "01111010001111010011110101000111",
   -- Hex(C8,64,64,AC), Hex(BA,5D,5D,E7), Hex(32,19,19,2B), Hex(E6,73,73,95)
   "11001000011001000110010010101100", "10111010010111010101110111100111",
  "00110010000110010001100100101011", "11100110011100110111001110010101",
   -- Hex(C0,60,60,A0), Hex(19,81,81,98), Hex(9E,4F,4F,D1), Hex(A3,DC,DC,7F)
   "11000000011000000110000010100000", "00011001100000011000000110011000",
  "10011110010011110100111111010001", "10100011110111001101110001111111",
   -- Hex(44,22,22,66), Hex(54,2A,2A,7E), Hex(3B,90,90,AB), Hex(0B,88,88,83)
   "01000100001000100010001001100110", "01010100001010100010101001111110",
  "00111011100100001001000010101011", "00001011100010001000100010000011",
   -- Hex(8C,46,46,CA), Hex(C7,EE,EE,29), Hex(6B,B8,B8,D3), Hex(28,14,14,3C)
   "10001100010001100100011011001010", "11000111111011101110111000101001",
  "01101011101110001011100011010011", "00101000000101000001010000111100",
   -- Hex(A7,DE,DE,79), Hex(BC,5E,5E,E2), Hex(16,0B,0B,1D), Hex(AD,DB,DB,76)
   "10100111110111101101111001111001", "10111100010111100101111011100010",
  "00010110000010110000101100011101", "10101101110110111101101101110110",
   -- Hex(DB,E0,E0,3B), Hex(64,32,32,56), Hex(74,3A,3A,4E), Hex(14,0A,0A,1E)
   "11011011111000001110000000111011", "01100100001100100011001001010110",
  "01110100001110100011101001001110", "00010100000010100000101000011110",
   -- Hex(92,49,49,DB), Hex(0C,06,06,0A), Hex(48,24,24,6C), Hex(B8,5C,5C,E4)
   "10010010010010010100100111011011", "00001100000001100000011000001010",
  "01001000001001000010010001101100", "10111000010111000101110011100100",
   -- Hex(9F,C2,C2,5D), Hex(BD,D3,D3,6E), Hex(43,AC,AC,EF), Hex(C4,62,62,A6)
   "10011111110000101100001001011101", "10111101110100111101001101101110",
  "01000011101011001010110011101111", "11000100011000100110001010100110",
   -- Hex(39,91,91,A8), Hex(31,95,95,A4), Hex(D3,E4,E4,37), Hex(F2,79,79,8B)
   "00111001100100011001000110101000", "00110001100101011001010110100100",
  "11010011111001001110010000110111", "11110010011110010111100110001011",
   -- Hex(D5,E7,E7,32), Hex(8B,C8,C8,43), Hex(6E,37,37,59), Hex(DA,6D,6D,B7)
   "11010101111001111111001110011 0010", "10001011110010001100100001000011",
  "01101110001101110011011101011001", "11011010011011010110110110110111",
   -- Hex(01,8D,8D,8C), Hex(B1,D5,D5,64), Hex(9C,4E,4E,D2), Hex(49,A9,A9,E0)
   "00000001100011011000110110001100", "10110001110101011101010101100100",
  "10011100010011100100111011010010", "01001001101010011010100111100000",
   -- Hex(D8,6C,6C,B4), Hex(AC,56,56,FA), Hex(F3,F4,F4,07), Hex(CF,EA,EA,25)
   "11011000011011000110110010110100", "10101100010101100101011011111010",
  "11110011111110100111101000000111", "11001111111010101110101000100101",
   -- Hex(CA,65,65,AF), Hex(F4,7A,7A,8E), Hex(47,AE,AE,E9), Hex(10,08,08,18)
   "11001010011001010110010110101111", "11110100011110100111101010001110",
  "01000111101011101010111011101001", "00010000000010000000100000011000",
   -- Hex(6F,BA,BA,D5), Hex(F0,78,78,88), Hex(4A,25,25,6F), Hex(5C,2E,2E,72)
   "01101111101110101011101011010101", "11110000011110000111100010001000",
  "01001010001001010010010101101111", "01011100001011100010111001110010",
   -- Hex(38,1C,1C,24), Hex(57,A6,A6,F1), Hex(73,B4,B4,C7), Hex(97,C6,C6,51)
   "00111000000111000001110000100100", "01010111101001101010011011110001",
  "01110011101101001011010011000111", "10010111110001101100011001010001",
```

```
  -- Hex(CB,E8,E8,23), Hex(A1,DD,DD,7C), Hex(E8,74,74,9C), Hex(3E,1F,1F,21)
   "11001011111010001110100000100011", "10100001110111011101110101111100",
  "11101000011101000111010010011100", "00111110000111110001111100100001",
   -- Hex(96,4B,4B,DD), Hex(61,BD,BD,DC), Hex(0D,8B,8B,86), Hex(0F,8A,8A,85)
   "10010110010010110100101111011101", "01100001101111011011110111011100",
  "00001101100010111000101110000110", "00001111100010101000101010000101",
   -- Hex(E0,70,70,90), Hex(7C,3E,3E,42), Hex(71,B5,B5,C4), Hex(CC,66,66,AA)
   "11100000011100000111000010010000", "01111100001111100011111001000010",
  "01110001101101011011010111000100", "11001100011001100110011010101010",
   -- Hex(90,48,48,D8), Hex(06,03,03,05), Hex(F7,F6,F6,01), Hex(1C,0E,0E,12)
   "10010000010010000100100011011000", "00000110000000110000001100000101",
  "11110111111110110111011000000001", "00011100000011100000111000010010",
   -- Hex(C2,61,61,A3), Hex(6A,35,35,5F), Hex(AE,57,57,F9), Hex(69,B9,B9,D0)
   "11000010011000010110000110100011", "01101010001101010011010101011111",
  "10101110010101110101011111111001", "01101001101110011011100111010000",
   -- Hex(17,86,86,91), Hex(99,C1,C1,58), Hex(3A,1D,1D,27), Hex(27,9E,9E,B9)
   "00010111100001101000011010010001", "10011001110000011100000101011000",
  "00111010000111010001110100100111", "00100111100111101001111010111001",
   -- Hex(D9,E1,E1,38), Hex(EB,F8,F8,13), Hex(2B,98,98,B3), Hex(22,11,11,33)
   "11011001111000011110000100111000", "11101011111110001111100000010011",
  "00101011100110001001100010110011", "00100010000100010001000100110011",
   -- Hex(D2,69,69,BB), Hex(A9,D9,D9,70), Hex(07,8E,8E,89), Hex(33,94,94,A7)
   "11010010011010010110100110111011", "10101001110110011101100101110000",
  "00000111100011101000111010001001", "00110011100101001001010010100111",
   -- Hex(2D,9B,9B,B6), Hex(3C,1E,1E,22), Hex(15,87,87,92), Hex(C9,E9,E9,20)
   "00101101100110111001101110110110", "00111100000111100001111000100010",
  "00010101100001111000011110010010", "11001001111010011110100100100000",
   -- Hex(87,CE,CE,49), Hex(AA,55,55,FF), Hex(50,28,28,78), Hex(A5,DF,DF,7A)
   "10000111110011101100111001001001", "10101010010101010101010111111111",
  "01010000001010000010100001111000", "10100101110111111101111101111010",
   -- Hex(03,8C,8C,8F), Hex(59,A1,A1,F8), Hex(09,89,89,80), Hex(1A,0D,0D,17)
   "00000011100011001000110010001111", "01011001101000011010000111111000",
  "00001001100010011000100110000000", "00011010000011010000110100010111",
   -- Hex(65,BF,BF,DA), Hex(D7,E6,E6,31), Hex(84,42,42,C6), Hex(D0,68,68,B8)
   "01100101101111111011111111011010", "11010111111001101110011000110001",
  "10000100010000100100001011000110", "11010000011010000110100010111000",
   -- Hex(82,41,41,C3), Hex(29,99,99,B0), Hex(5A,2D,2D,77), Hex(1E,0F,0F,11)
   "10000010010000010100000111000011", "00101001100110011001100110110000",
  "01011010001011010010110101110111", "00011110000011110000111100010001",
   -- Hex(7B,B0,B0,CB), Hex(A8,54,54,FC), Hex(6D,BB,BB,D6), Hex(2C,16,16,3A)
   "01111011101100001011000011001011", "10101000010101000101010011111100",
  "01101101101110111011101111010110", "00101100000101100001011000111010"
  );
  variable r, s, t, u : u_sign32;
  begin

     r := ftabletab(to_integer(a(1 to 8)));
     s := ftabletab(to_integer(a(9 to 16)));
     t := ftabletab(to_integer(a(17 to 24)));
     u := ftabletab(to_integer(a(25 to 32)));

     q_out(1 to 32):=  (r(1 to 8) xor s(25 to 32) xor t(17 to 24) xor u(9 to 16)) &
                       (r(9 to 16) xor s(1 to 8) xor t(25 to 32) xor u(17 to 24)) &
                       (r(17 to 24) xor s(9 to 16) xor t(1 to 8) xor u(25 to 32)) &
                       (r(25 to 32) xor s(17 to 24) xor t(9 to 16) xor u(1 to 8));
  end ftable_quad;
end encryption_tables;
```

Figure D-15 VHDL package for 256-bit AES example

```
--***************** --
--  256-Bit AES  --
--***************** --
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.aes_procedures.all;
use work.encryption_tables.all;
entity aes256 is
port(
   key, d_block: in u_sign32;
   in_hs_rdy:   in unsigned(0 downto 0);
   in_hs_rcv:   buffer unsigned(0 downto 0) := "0";
   ciphertext:  out u_sign32;
   out_hs_rdy:  buffer unsigned(0 downto 0):= "0";
   out_hs_rcv:  in unsigned(0 downto 0)
 );
end aes256;

architecture behaviour of aes256 is
begin
  process
     variable bb, cc, dd, ee, ff, temp_vec1, temp_vec2: u_sign32;
     --variable transition_state,  temp_transition_state : tab_a4;
     variable transition_state,  temp_transition_state : u_sign128;
     variable i: unsigned(6 downto 0) := "0000000";
     variable j: unsigned(4 downto 0) := "00000";

     variable fkey : tab_64;
     -- moods ram
     variable temp1, temp2, temp3, keyloop: unsigned(6 downto 0) :="0000000";
     variable index1, index2, index3 : unsigned(1 downto 0) :="00";
begin

   for keyloop1 in 0 to 7 loop
     while(in_hs_rdy = in_hs_rcv) loop
       wait for 10 ns;
     end loop;
     fkey(keyloop1) := key;
       case keyloop1 is
       when 0 =>
         temp_transition_state(1 to 32)   := d_block;
       when 1 =>
         temp_transition_state(33 to 64)  := d_block;
       when 2 =>
         temp_transition_state(65 to 96)  := d_block;
       when 3 =>
         temp_transition_state(97 to 128) := d_block;
       when others => NULL;
       end case;

     in_hs_rcv <= not in_hs_rcv;
     wait for 10 ns;
   end loop;
   -- For AES-256 (Nk=8, Nr=14)
   -- For 256-bit : Nb * (Nr +1 )  = 4 * 15 = 60 ("0111100")
   -- For 192-bit :              = 4 * 13 = 52 ("0110100")
   -- For 128-bit :              = 4 * 11 = 44 ("0101100")

   i := "0001000";  -- start off with the value of Nk, in this case = 8
   j := "00000";     -- round counter
```

```
  while(i < "0111100") loop        -- i < Nb * (Nr +1 )
   temp1 := i - "0000001";
   wait for 10 ns;
    ---------- i mod Nk = 0 ----------
    bb:= fkey(to_integer(temp1(5 downto 0)));
    r_oneto8(bb, dd);                                    -- RotWord(w[i-1])
    temp_vec1(1 to 32):=   fbsub_quad(dd(1 to 32));  -- SubWord(RotWord(w[i-1]))
    rco(j, cc);                                          -- Rcon
    temp1 := i - "0001000";
    fkey(to_integer(i(5 downto 0))) := fkey(to_integer(temp1(5 downto 0))) xor temp_vec1(1 to 32) xor
      cc;   -- w[i-Nk] xor SubWord xor Rcon
    ------------------------------------------

    keyloop := "0000001";
    while keyloop /= "0000100" loop
    ----moods unroll
     if (keyloop + i < "0111100") then
       temp1 := keyloop + i - "0001000";
       temp2 := keyloop + i - "0000001";
       temp3 := keyloop + i;
       fkey(to_integer(temp3(5 downto 0))) := fkey(to_integer(temp1(5 downto 0))) xor
         fkey(to_integer(temp2(5 downto 0)));   -- w[i]  = w[i-Nk] xor temp
     end if;
     keyloop := keyloop + "0000001";
    end loop;

    ---------- i mod Nk = 4 ----------
    if(i + "0000100" < "0111100") then
     temp1 := i + "0000011";
     cc := fkey(to_integer(temp1(5 downto 0)));
     temp_vec1(1 to 32):=   fbsub_quad(cc(1 to 32));   -- SubWord(RotWord(w[i-1]))

     temp2 := i - "0000100";
     temp3 := i + "0000100";
     fkey(to_integer(temp3(5 downto 0))):= fkey(to_integer(temp2(5 downto 0))) xor
       temp_vec1(1 to 32);
    end if;
    ------------------------------------------
    keyloop := "0000101";
    while keyloop /= "0001000" loop
    ----moods unroll
     if(keyloop + i < "0111100") then
       temp1 := keyloop + i - "0001000";
       temp2 := keyloop + i - "0000001";
       temp3 := keyloop + i;
       fkey(to_integer(temp3(5 downto 0))) := fkey(to_integer(temp1(5 downto 0))) xor
         fkey(to_integer(temp2(5 downto 0))); -- w[i]  = w[i-Nk] xor temp
     end if;
     keyloop := keyloop + "0000001";
    end loop;

    i := i + "0001000"; -- increment by Nk
    j := j + "00001";
  end loop;

--======================== First Round ========================--
  transition_state(1 to 32) :=   fkey(0) xor temp_transition_state(1 to 32);       -- AddRoundKey
  transition_state(33 to 64) :=  fkey(1) xor temp_transition_state(33 to 64);
  transition_state(65 to 96) :=  fkey(2) xor temp_transition_state(65 to 96);
  transition_state(97 to 128) :=  fkey(3) xor temp_transition_state(97 to 128);

 --************************************************************--
i := "0000100"; -- start off with the 4th key, 3 used in the first round
```

```
 for EncLoop2 in 1 to 13 loop -- For AES-256 (Nk=8, Nr=14)

    for EncLoop3 in 0 to 3 loop
     bb := fkey(to_integer(i(5 downto 0)));

     case EncLoop3 is
     when 0 =>
      temp_vec1(1 to 32) := transition_state(1 to 8) & transition_state(41 to 48) &
        transition_state(81 to 88) & transition_state(121 to 128);
      ftable_quad(temp_vec1, cc );                    -- Retrieve values from Forward Tables
      temp_transition_state(1 to 32) := bb(1 to 32)  xor cc(1 to 32);
     when 1 =>
      temp_vec1(1 to 32) := transition_state(33 to 40) & transition_state(73 to 80) &
        transition_state(113 to 120) & transition_state(25 to 32);
      ftable_quad(temp_vec1, cc );                    -- Retrieve values from Forward Tables
      temp_transition_state(33 to 64) := bb(1 to 32)  xor cc(1 to 32);
     when 2 =>
      temp_vec1(1 to 32) := transition_state(65 to 72) & transition_state(105 to 112) &
        transition_state(17 to 24) & transition_state(57 to 64);
      ftable_quad(temp_vec1, cc );                    -- Retrieve values from Forward Tables
      temp_transition_state(65 to 96) := bb(1 to 32)  xor cc(1 to 32);
     when 3 =>
      temp_vec1(1 to 32) := transition_state(97 to 104) & transition_state(9 to 16) &
        transition_state(49 to 56) & transition_state(89 to 96);
      ftable_quad(temp_vec1, cc );                    -- Retrieve values from Forward Tables
      temp_transition_state(97 to 128) := bb(1 to 32)  xor cc(1 to 32);
     when others => NULL;
     end case;
     i := i + "0000001";
    end loop;

    transition_state(1 to 128) := temp_transition_state(1 to 128);
 end loop;

 --======================== Last Round ============================--
 for EncLoop5 in 0 to 3 loop
    bb := fkey(to_integer(i(5 downto 0)));

    case EncLoop5 is
    when 0 =>
     temp_vec1(1 to 32) := transition_state(1 to 8) & transition_state(41 to 48) &
        transition_state(81 to 88) & transition_state(121 to 128);
     dd(1 to 32):= fbsub_quad( temp_vec1 );                    -- w[i-1] = SubWord(w[i-1])
     temp_transition_state(1 to 32) := bb(1 to 32) xor dd(1 to 32);  -- AddRoundKey
    when 1 =>
     temp_vec1(1 to 32) := transition_state(33 to 40) & transition_state(73 to 80) &
        transition_state(113 to 120) & transition_state(25 to 32);
     dd(1 to 32):= fbsub_quad( temp_vec1 );                    -- w[i-1] = SubWord(w[i-1])
     temp_transition_state(33 to 64) := bb(1 to 32) xor dd(1 to 32); -- AddRoundKey
    when 2 =>
     temp_vec1(1 to 32) := transition_state(65 to 72) & transition_state(105 to 112) &
       transition_state(17 to 24) & transition_state(57 to 64);
     dd(1 to 32):= fbsub_quad( temp_vec1 );                    -- w[i-1] = SubWord(w[i-1])
     temp_transition_state(65 to 96) := bb(1 to 32) xor dd(1 to 32); -- AddRoundKey
    when 3 =>
     temp_vec1(1 to 32) := transition_state(97 to 104) & transition_state(9 to 16) &
       transition_state(49 to 56) & transition_state(89 to 96);
     dd(1 to 32):= fbsub_quad( temp_vec1 );                    -- w[i-1] = SubWord(w[i-1])
     temp_transition_state(97 to 128) := bb(1 to 32) xor dd(1 to 32);  -- AddRoundKey
    when others => NULL;
    end case;
    i := i + "0000001";
 end loop;
```

```
  for EncLoop6 in 0 to 3 loop
   while(out_hs_rdy /= out_hs_rcv) loop
    wait for 10 ns;
   end loop;
   case EncLoop6 is
   when 0 =>
    ciphertext  <= temp_transition_state(1 to 32);
   when 1 =>
    ciphertext  <= temp_transition_state(33 to 64);
   when 2 =>
    ciphertext  <= temp_transition_state(65 to 96);
   when 3 =>
    ciphertext  <= temp_transition_state(97 to 128);
   when others => NULL;
   end case;

   out_hs_rdy <= not out_hs_rdy;
   wait for 10 ns;
  end loop;
  end process;
end behaviour;
```

Figure D-16 VHDL of 256-Bit AES example

The post-MOODS synthesis simulation of the non-pipelined multi-FPGA 256-bit AES example is given in Figure D-17. Zoom in views of the simulation showing inputs and outputs updates are given in Figure D-18. The simulation input values (shown in hexadecimals) are taken from the appendix (C.3 AES-256) of the AES specification [146]:

Input plaintext (d_block) values: 00112233, 44556677, 8899AABB, CCDDEEFF.

Key: 00010203, 04050607, 08090A0B, 0C0D0E0F, 10111213, 14151617, 18191A1B, 1C1D1E1F.

The output ciphertext is: 8EA2B7CA, 516745BF, EAFC4990, 4B496089.

With a system clock period of 200 ns, the non-pipelined multi-FPGA 256-bit AES takes 5257 clock cycles (i.e. clock cycles = (1055500 ns - 4100 ns) / 200 ns) to process the 128-bit data block using a 256-bit cipher key.

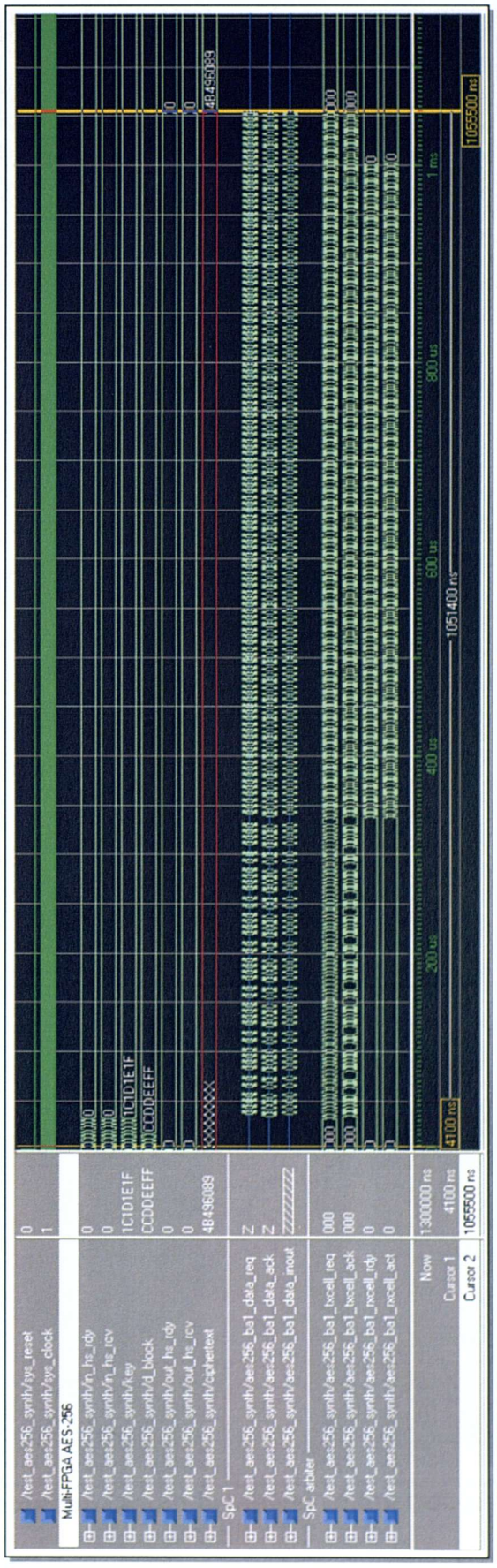


Figure D-17 Simulation of the non-pipelined multi-FPGA 256-bit AES core

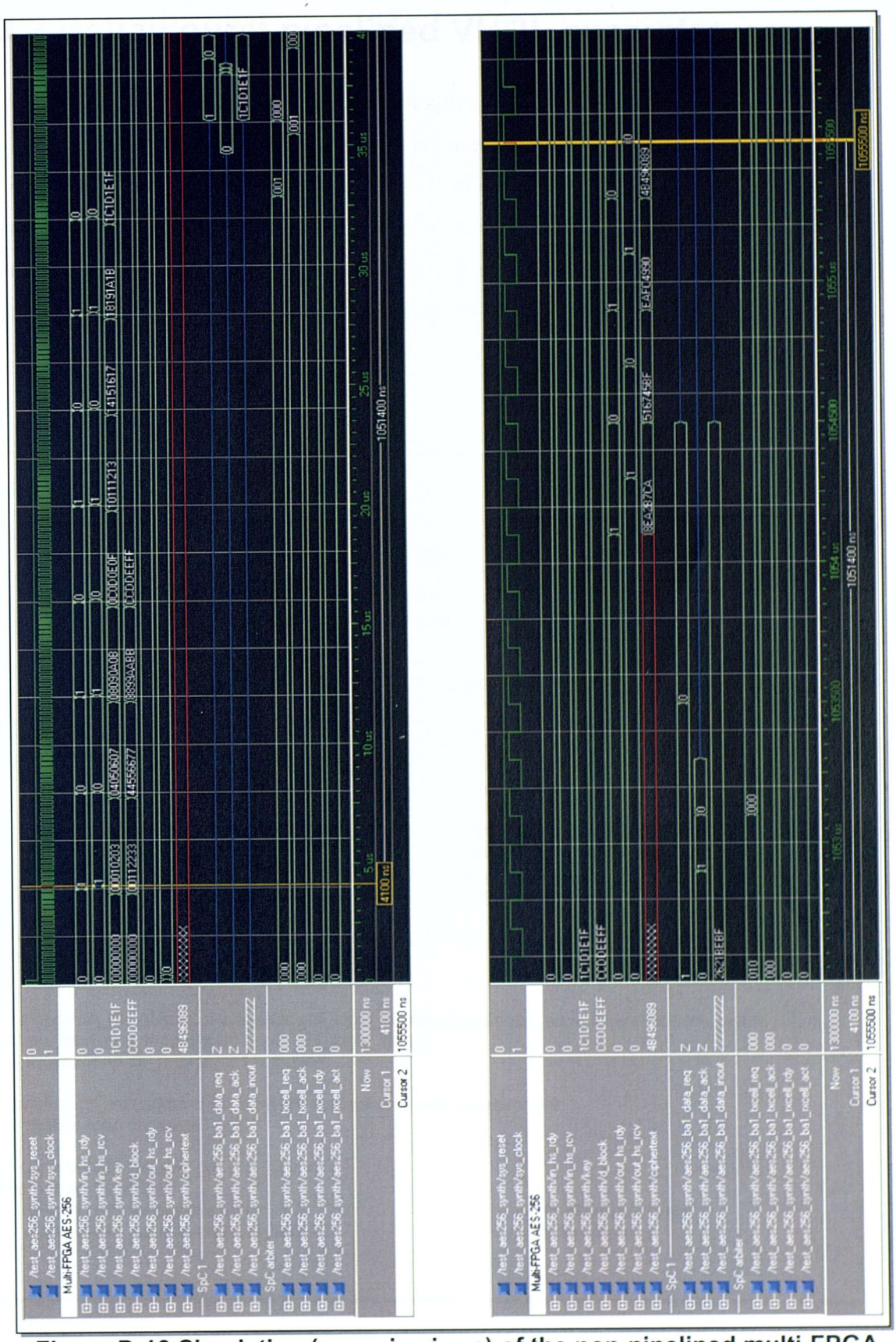


Figure D-18 Simulation (zoom in views) of the non-pipelined multi-FPGA 256-bit AES core

D.2 Behavioural pipelined VHDL examples

The three behavioural pipelined VHDL examples given in this section are used in experiments (with explicit communication channels) described in Section 6.3. All the VHDL packages which contain the definitions of constants, types, signals, functions, and procedures are similar to the non-pipelined implementation and they are found in the previous section. The explicit communication channel VHDL package used by all the pipelined VHDL examples in this section is given in Figure D-19.

```
library ieee;
use ieee.std_logic_1164.all;
package channel_package is
  subtype semaphore is std_logic_vector(0 downto 0);
  subtype int8 is integer range 0 to 255;
  subtype channel_sem is std_logic_vector(0 downto 0);
  subtype channel_ack is std_logic_vector(0 downto 0);

  -- initialise channel semaphore
  procedure init(signal sem: out channel_sem);          -- channel semaphore
  -- send data
  procedure send(signal sem: out channel_sem;           -- channel send semaphore
                 signal ack: in channel_ack;            -- channel send acknowledge
                 signal chan_data: out std_logic_vector; -- channel send data
                 d: in std_logic_vector);               -- data to send
  -- recv data
  procedure recv(signal sem: out channel_sem;           -- channel receive semaphore
                 signal ack: in channel_ack;            -- channel receive acknowledge
                 signal chan_data: in std_logic_vector; -- channel receive data
                 d: out std_logic_vector);              -- data received

function ch_send(d: std_logic_vector; signal chan_sem_in: semaphore; signal chan_sem_out:
semaphore) return std_logic_vector;
-- moods map ch_send u:* u:1 u:1 u:%1

function ch_recv(signal chan_data: std_logic_vector; signal chan_sem_in: semaphore; signal
chan_sem_out: semaphore) return std_logic_vector;
-- moods map ch_recv u:* u:1 u:1 u:%1

function ch_init(signal chan_sem_in: semaphore) return semaphore;
-- moods map ch_init u:1 u:1

-- channel component
component channel
   generic (width: positive := 1);                      -- width of channel data
     port (send_sem: in channel_sem;                    -- send semaphore
```

```
 procedure send(signal sem: out channel_sem; signal ack: in channel_ack; signal chan_data: out
 std_logic_vector; d: in std_logic_vector) is
   -- moods inline
   begin
     chan_data <= ch_send(d,ack,sem);
   end procedure send;
   --=================================================================--
   procedure recv(signal sem: out channel_sem; signal ack: in channel_ack; signal chan_data: in
 std_logic_vector; d: out std_logic_vector) is
   -- moods inline
   begin
     d := ch_recv(chan_data, ack, sem);
   end;
 --=================================================================--
 procedure init(signal sem: out channel_sem) is
   -- moods inline
   variable init_sig : channel_sem := "0";
   begin
     --sem <= ch_init("0");
     sem <= init_sig;
   end;
 --=================================================================--
 function ch_send(d: std_logic_vector; signal chan_sem_in: semaphore; signal chan_sem_out:
 semaphore) return std_logic_vector is
   -- moods map ch_send u:* u:1 u:1 u:%1
   begin
     return d;
   end;
 --=================================================================--
 function ch_recv(signal chan_data: std_logic_vector; signal chan_sem_in: semaphore; signal
 chan_sem_out: semaphore) return std_logic_vector is
   -- moods map ch_recv u:* u:1 u:1 u:%1
   begin
     return chan_data;
   end;
 --=================================================================--
 function ch_init(signal chan_sem_in: semaphore) return semaphore is
   -- moods map ch_init u:1 u:1
   begin
    return "0";
   end;
 end package body channel_package;
```

Figure D-19 VHDL package of the explicit communication channel

D.2.1 Pipelined quadratic equation solver

The pipelined quadratic equation solver is a two-stage pipelined version of the quadratic equation solver given in Section 6.2.1. The behavioural VHDL of the pipelined quadratic equation solver example is given in Figure D-20.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.c_types.all;
use work.algeqn_package.all;
use work.imath.all;
use work.channel_package.all;

 entity pipe_quad is
 port (
  a1,a2,a3: in int;
  x1,x2: out int;
  no_real: out int
 );
 end pipe_quad;
 architecture behaviour of pipe_quad is
 signal c1_send_sem, c1_recv_sem: channel_sem := "0";
 signal c1_send_ack, c1_recv_ack: channel_ack := "0";
 signal c1_send_data, c1_recv_data: std_logic_vector(95 downto 0) := (others=>'0');
 begin
 -- Explicit communication channel
 c1: entity work.SIMPLE_CHANNEL generic map (96)
 port map(c1_send_sem, c1_recv_sem, c1_send_data, c1_send_ack, c1_recv_ack, c1_recv_data);
 --=================================================================--
 Prs_1: process -- Process module p_MOD_1
  variable temp1 : std_logic_vector(95 downto 0);
  variable b1: int := X"00000000";
  variable b2: int := X"00000000";
  variable b3: int := X"00000000";
  variable d1,d2: int;
 begin
  init(c1_send_sem);
  forever: loop
   b1 := a1;
   b2 := a2;
   b3 := a3;

   d1 := sqi(b2) - multi(multi(to_int(4),b1),b3);
   d2 := multi(b1,to_int(2));
   temp1 := std_logic_vector(b2 & d2 & d1);
   send(c1_send_sem, c1_send_ack, c1_send_data, temp1);
   wait for 40 ns;
  end loop;
 end process Prs_1;
--=================================================================--
 Prs_2: process -- Process module p_MOD_2
  variable temp2 : std_logic_vector(95 downto 0);
  variable e1: int := X"00000000";
  variable e2: int := X"00000000";
  variable rd: int;
  variable f1: int;
 begin
  init(c1_recv_sem);
```

```
    forever: loop
      recv(c1_recv_sem, c1_recv_ack, c1_recv_data, temp2);
      e1 := int(temp2(31 downto 0));
      e2 := int(temp2(63 downto 32));
      f1 := int(temp2(95 downto 64));

      if(e1 < 0) then
        no_real <= to_int(0);
      else
        rd := sqrti(e1);
        x1 <= sdivi((-f1 + rd),e2);
        x2 <= sdivi((-f1 - rd),e2);
        no_real <= to_int(2);
      end if;
      wait for 40 ns;
    end loop;
  end process Prs_2;
end behaviour;
```

Figure D-20 VHDL of pipelined quadratic equation solver

Figure D-21 shows the post-MOODS synthesis simulation of the two-stage pipelined multi-FPGA quadratic equation solver. This two-device multi-FPGA implementation has a single explicit communication channel (*ExC 1*) connecting the pipeline stages. Integer inputs a1, a2, and a3 of the quadratic equation solver are given values 1, -25 and 150 respectively. Outputs x1, x2 and number of real numbers (no_real) are updated after 7660 ns. With a system clock period of 40 ns, the pipelined multi-FPGA quadratic equation solver takes 189 clock cycles (i.e. clock cycles = (7660 ns -100 ns) / 40 ns) to complete the application and output the result.

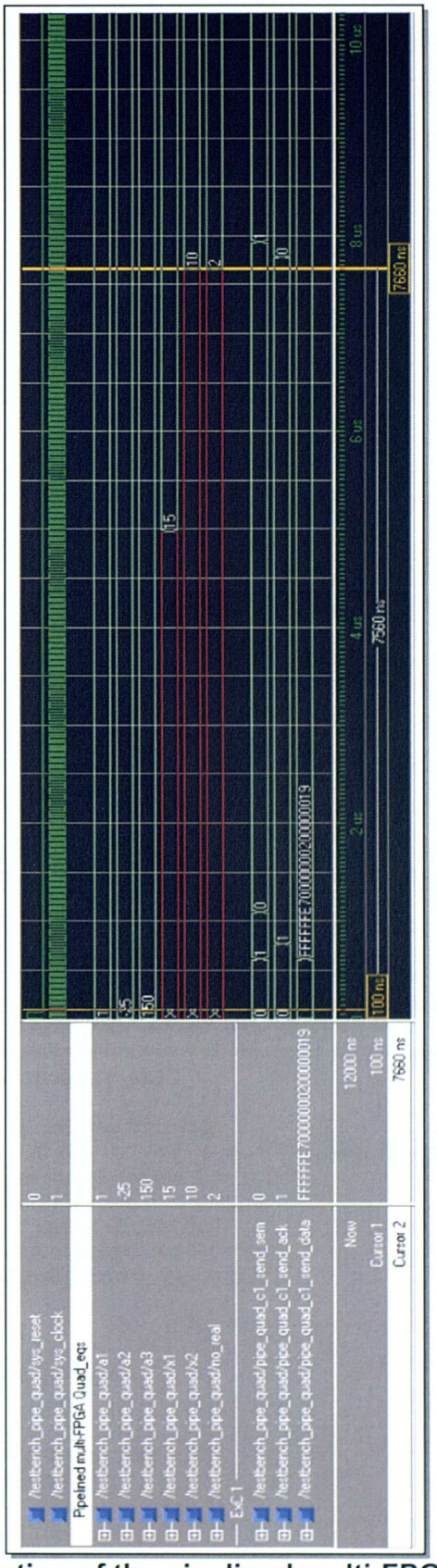


Figure D-21 Simulation of the pipelined multi-FPGA quadratic equation solver

D.2.2 Pipelined inverse discrete cosine transform

This second pipelined VHDL example is the two-stage pipelined version of the inverse discrete cosine transform (IDCT) core given in Section 6.2.3. The behavioural VHDL of the pipelined inverse discrete cosine transform example is given in Figure D-22.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.channel_package.all;
use work.idct_package.all;
entity pipe_Idct is
port (
     in_hs_rdy:  in unsigned(0 downto 0);            -- Handshake ready
     in_hs_rcv:  buffer unsigned(0 downto 0) := "0";  -- Handshake receive
     dct_2d_in:  in signed(11 downto 0);
     idct_out:   out signed(7 downto 0) := (others=>'0'); --  8 bit output.
     out_hs_rdy: buffer unsigned(0 downto 0) := "0";     -- Handshake ready
     out_hs_rcv: in unsigned(0 downto 0);            -- Handshake receive
     sys_clock:  in unsigned(0 downto 0);
     --moods clock
     sys_reset:  in unsigned(0 downto 0)
     --moods reset
      );
end pipe_Idct;

ARCHITECTURE behaviour of pipe_Idct is
signal c1_send_sem, c1_recv_sem: channel_sem := "0";
  signal c1_send_ack, c1_recv_ack: channel_ack := "0";
  signal c1_send_data, c1_recv_data: std_logic_vector(10 downto 0) := (others=>'0');
-- memory section
type RAM_mem_type is array (0 to 63) of signed(10 downto 0);
begin
--*************************************************************************
c1: entity work.SIMPLE_CHANNEL generic map (11) port map(c1_send_sem, c1_recv_sem,
c1_send_data, c1_send_ack, c1_recv_ack, c1_recv_data);
--=========================================================================--
Prs_1: process --  Process module p_MOD_1
-- IDCT_2 signals
  variable xa0_reg, xa1_reg, xa2_reg, xa3_reg,
         xa4_reg, xa5_reg, xa6_reg, xa7_reg:  signed(11 downto 0):= (others=>'0');
  variable ID_input_cnt:  unsigned(3 downto 0):= "0000";
  variable z_out_int : signed(21 downto 0) := (others=>'0');
  variable temp1 : std_logic_vector(10 downto 0);
  variable cnt_64: unsigned(6 downto 0) := "0000000";
  variable ID_index_i: unsigned(3 downto 0):= "0000";
begin
  reset_loop: loop
    in_hs_rcv <= "0";
    ID_input_cnt(3 downto 0)  := "0000";
    cnt_64 := "0000000";
    ID_index_i := "0000";
  wait until sys_clock'event and sys_clock = "1";
  exit reset_loop when sys_reset = "1";
  init(c1_send_sem);
  main_loop: loop
    while(cnt_64(6) = '0') loop
```

```
      while(ID_input_cnt(3) = '0') loop
       while(in_hs_rdy = in_hs_rcv) loop
        wait until sys_clock'event and sys_clock = "1";
       end loop;
       case ID_input_cnt(2 downto 0) is
       when "000" =>  xa0_reg := dct_2d_in;
       when "001" =>  xa1_reg := dct_2d_in;
       when "010" =>  xa2_reg := dct_2d_in;
       when "011" =>  xa3_reg := dct_2d_in;
       when "100" =>  xa4_reg := dct_2d_in;
       when "101" =>  xa5_reg := dct_2d_in;
       when "110" =>  xa6_reg := dct_2d_in;
       when "111" =>  xa7_reg := dct_2d_in;
       when others => NULL;
       end case;

       in_hs_rcv <= not in_hs_rcv;
       ID_input_cnt(3 downto 0) := ID_input_cnt(3 downto 0) + "0001";
       wait until sys_clock'event and sys_clock = "1";
      end loop;

      while (ID_index_i /= "1000") loop
       idct1_mult_add(ID_index_i(2 downto 0),xa0_reg,xa1_reg,xa2_reg,
                  xa3_reg,xa4_reg,xa5_reg,xa6_reg,xa7_reg,z_out_int);

       ID_index_i := ID_index_i + "0001";

       if(z_out_int(20) = '0' and z_out_int(7) = '1') then
          temp1 := std_logic_vector(z_out_int(18 downto 8) + to_signed(1,11));
       else
          temp1 := std_logic_vector(z_out_int(18 downto 8));
       end if;

       send(c1_send_sem, c1_send_ack, c1_send_data, temp1);
       --wait until sys_clock'event and sys_clock = "1";
      end loop;
      ID_index_i := "0000";
      cnt_64 := cnt_64 + "0000001";
     end loop;
     ID_input_cnt(3 downto 0)  := "0000";
     cnt_64 := "0000000";
   wait until sys_clock'event and sys_clock = "1";
   exit reset_loop when sys_reset = "1";
   end loop;
   end loop;
end process Prs_1;
--=================================================================--
Prs_2: process -- Process module p_MOD_2
  -- IDCT_2 signals
  variable xb0_reg, xb1_reg, xb2_reg, xb3_reg,
         xb4_reg, xb5_reg, xb6_reg, xb7_reg:  signed(10 downto 0):= (others=>'0');
  variable temp2 : std_logic_vector(10 downto 0);
  variable rcv_z_out : signed(10 downto 0) := (others=>'0');
  variable ID_wr_cntr: unsigned(6 downto 0):= (others=>'0');
  variable ID_rd_cntr: unsigned(3 downto 0):= (others=>'0');
  variable ID_index_j: unsigned(3 downto 0):= "0000";
  variable idct2d_int: signed(20 downto 0):= (others=>'0');
  variable ID_ram1_mem:   RAM_mem_type;
  --moods ram
begin
  reset_loop: loop
    ID_wr_cntr := "0000000";
    ID_rd_cntr := "0000";
```

```
out_hs_rdy <= "0";
idct2d_Int := (others=>'0');
ID_index_j := "0000";
wait until sys_clock'event and sys_clock = "1";
exit reset_loop when sys_reset = "1";
init(c1_recv_sem);
main_loop: loop

if(ID_wr_cntr(6) = '0') then
   recv(c1_recv_sem, c1_recv_ack, c1_recv_data, temp2);
   rcv_z_out := signed(temp2);
   ID_ram1_mem(to_integer(ID_wr_cntr(5 downto 0))) := rcv_z_out;
   ID_wr_cntr := ID_wr_cntr + "0000001";
else
 while(ID_rd_cntr(3) = '0') loop

   case ID_rd_cntr(2 downto 0) is
   when "000" => xb0_reg := ID_ram1_mem(0);
            xb1_reg := ID_ram1_mem(8);
            xb2_reg := ID_ram1_mem(16);
            xb3_reg := ID_ram1_mem(24);
            xb4_reg := ID_ram1_mem(32);
            xb5_reg := ID_ram1_mem(40);
            xb6_reg := ID_ram1_mem(48);
            xb7_reg := ID_ram1_mem(56);
  when "001" => xb0_reg := ID_ram1_mem(1);
            xb1_reg := ID_ram1_mem(9);
            xb2_reg := ID_ram1_mem(17);
            xb3_reg := ID_ram1_mem(25);
            xb4_reg := ID_ram1_mem(33);
            xb5_reg := ID_ram1_mem(41);
            xb6_reg := ID_ram1_mem(49);
            xb7_reg := ID_ram1_mem(57);
  when "010" => xb0_reg := ID_ram1_mem(2);
            xb1_reg := ID_ram1_mem(10);
            xb2_reg := ID_ram1_mem(18);
            xb3_reg := ID_ram1_mem(26);
            xb4_reg := ID_ram1_mem(34);
            xb5_reg := ID_ram1_mem(42);
            xb6_reg := ID_ram1_mem(50);
            xb7_reg := ID_ram1_mem(58);
 when "011" => xb0_reg := ID_ram1_mem(3);
            xb1_reg := ID_ram1_mem(11);
            xb2_reg := ID_ram1_mem(19);
            xb3_reg := ID_ram1_mem(27);
            xb4_reg := ID_ram1_mem(35);
            xb5_reg := ID_ram1_mem(43);
            xb6_reg := ID_ram1_mem(51);
            xb7_reg := ID_ram1_mem(59);
  when "100" => xb0_reg := ID_ram1_mem(4);
            xb1_reg := ID_ram1_mem(12);
            xb2_reg := ID_ram1_mem(20);
            xb3_reg := ID_ram1_mem(28);
            xb4_reg := ID_ram1_mem(36);
            xb5_reg := ID_ram1_mem(44);
            xb6_reg := ID_ram1_mem(52);
            xb7_reg := ID_ram1_mem(60);
  when "101" => xb0_reg := ID_ram1_mem(5);
            xb1_reg := ID_ram1_mem(13);
            xb2_reg := ID_ram1_mem(21);
            xb3_reg := ID_ram1_mem(29);
            xb4_reg := ID_ram1_mem(37);
            xb5_reg := ID_ram1_mem(45);
            xb6_reg := ID_ram1_mem(53);
            xb7_reg := ID_ram1_mem(61);
```

```
      when "110" =>  xb0_reg := ID_ram1_mem(6);
                     xb1_reg := ID_ram1_mem(14);
                     xb2_reg := ID_ram1_mem(22);
                     xb3_reg := ID_ram1_mem(30);
                     xb4_reg := ID_ram1_mem(38);
                     xb5_reg := ID_ram1_mem(46);
                     xb6_reg := ID_ram1_mem(54);
                     xb7_reg := ID_ram1_mem(56);
      when "111" =>  xb0_reg := ID_ram1_mem(7);
                     xb1_reg := ID_ram1_mem(15);
                     xb2_reg := ID_ram1_mem(23);
                     xb3_reg := ID_ram1_mem(31);
                     xb4_reg := ID_ram1_mem(39);
                     xb5_reg := ID_ram1_mem(47);
                     xb6_reg := ID_ram1_mem(55);
                     xb7_reg := ID_ram1_mem(63);
      when others => NULL;
      end case;

      ID_rd_cntr(3 downto 0) := ID_rd_cntr(3 downto 0) + "0001";
      while (ID_index_j /= "1000") loop
        idct2_mult_add(ID_index_j(2 downto 0),xb0_reg,xb1_reg,xb2_reg,xb3_reg,
                       xb4_reg,xb5_reg,xb6_reg,xb7_reg,idct2d_int);

        while(out_hs_rdy /= out_hs_rcv) loop
          wait until sys_clock'event and sys_clock = "1";
        end loop;
        idct_out <= signed(idct2d_int(15 downto 8));
        out_hs_rdy <= not out_hs_rdy;
        ID_index_j := ID_index_j + "0001";
      end loop;
      wait until sys_clock'event and sys_clock = "1";
    end loop;
    ID_index_j := "0000";
    ID_wr_cntr(6 downto 0)   := (others=>'0');
    ID_rd_cntr(3 downto 0)   := (others=>'0');
  end if;

  wait until sys_clock'event and sys_clock = "1";
  exit reset_loop when sys_reset = "1";
  end loop;
  end loop;
end process Prs_2;
--------------------------------------------------------------------------------
end behaviour;
```

Figure D-22 VHDL of pipelined inverse discrete cosine transform example

The post-MOODS synthesis simulation of the 2-stage pipelined multi-FPGA IDCT is given in Figure D-23. Zoom in views of the simulation showing inputs and outputs updates are given in Figure D-24. The pipelined multi-FPGA IDCT has a single explicit communication channel (*ExC 1*) connecting the pipeline stages. With a system clock period of 40 ns, the pipelined multi-FPGA IDCT takes 1167 clock cycles (i.e. clock cycles = (47160 ns - 480 ns) / 40 ns) to complete the application.

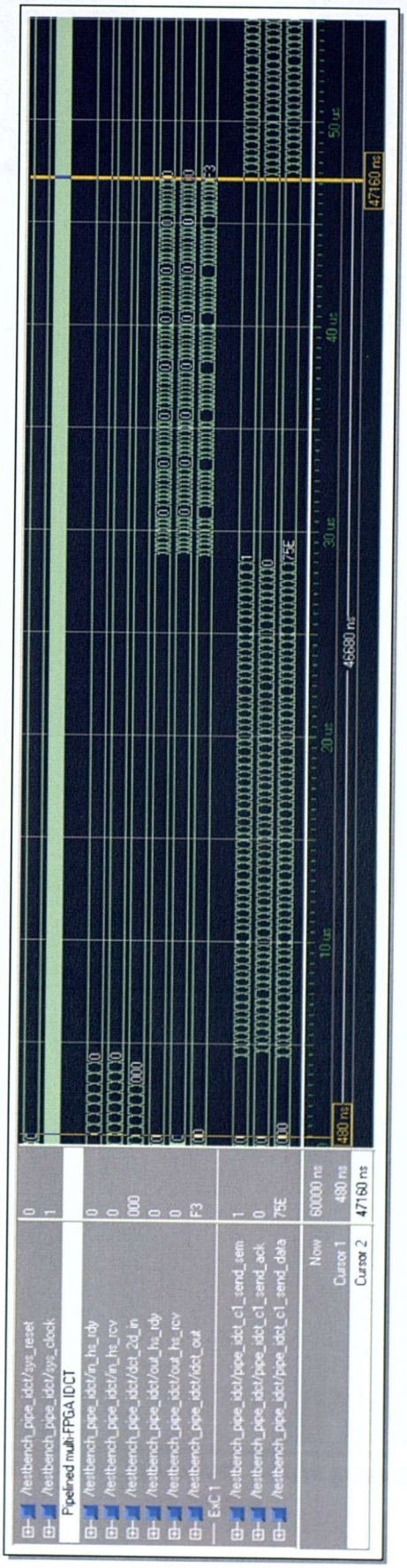


Figure D-23 Simulation of the pipelined multi-FPGA IDCT example

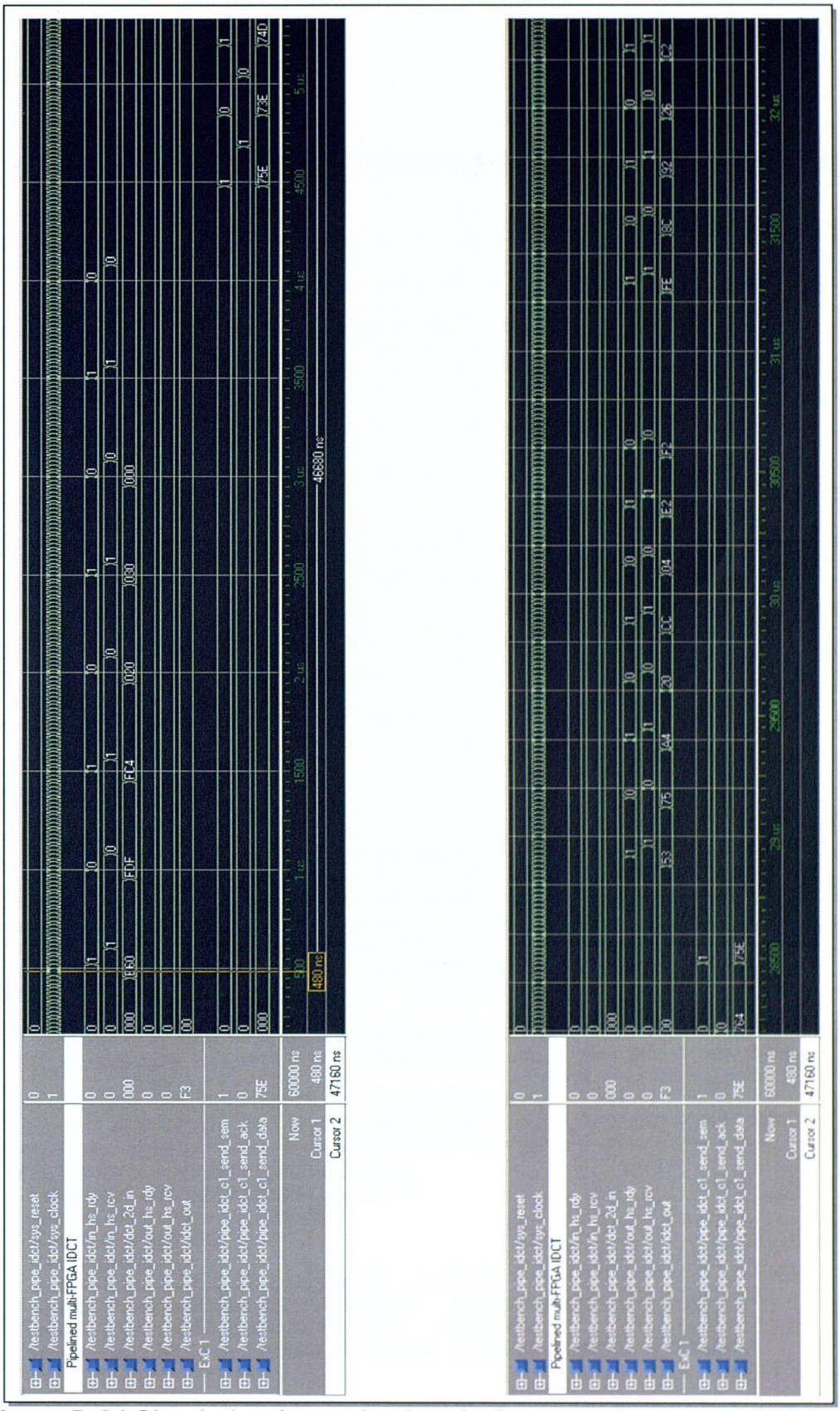


Figure D-24 Simulation (zoom in views) of the pipelined multi-FPGA IDCT example

D.2.3 Pipelined 256-bit advanced encryption standard

The last pipelined VHDL example is the two-stage pipelined version of the 256-bit advanced encryption standard (AES) core given in Section 6.2.5. The behavioural VHDL of the pipelined 256-bit AES core is given in Figure D-25.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.channel_package.all;
use work.aes_procedures.all;
use work.encryption_tables.all;
entity pipe_aes256 is
port(
   key, d_block: in u_sign32;
   in_hs_rdy:   in unsigned(0 downto 0);
   in_hs_rcv:   buffer unsigned(0 downto 0) := "0";
   ciphertext:  out u_sign32;
   out_hs_rdy:  buffer unsigned(0 downto 0):= "0";
   out_hs_rcv:  in unsigned(0 downto 0)
 );
end pipe_aes256;

architecture behaviour of pipe_aes256 is
  signal c1_send_sem, c1_recv_sem: channel_sem := "0";
  signal c1_send_ack, c1_recv_ack: channel_ack := "0";
  signal c1_send_data, c1_recv_data: std_logic_vector(31 downto 0) := (others=>'0');
  signal c2_send_sem, c2_recv_sem: channel_sem := "0";
  signal c2_send_ack, c2_recv_ack: channel_ack := "0";
  signal c2_send_data, c2_recv_data: std_logic_vector(31 downto 0) := (others=>'0');
begin

  c1: entity work.SIMPLE_CHANNEL generic map (32) port map
   (c1_send_sem, c1_recv_sem, c1_send_data, c1_send_ack, c1_recv_ack, c1_recv_data);

  c2: entity work.SIMPLE_CHANNEL generic map (32) port map
   (c2_send_sem, c2_recv_sem, c2_send_data, c2_send_ack, c2_recv_ack, c2_recv_data);
--=================================================================--
 Prs_1: process --  Process module p_MOD_2
   variable bb1, cc1, dd1, temp_vec1: u_sign32;
   variable temp_t_state : u_sign128;
   variable fkey : tab_64;
   -- moods ram
   variable temp1, temp3: std_logic_vector(31 downto 0);
   variable i: unsigned(6 downto 0) := "0000000"; -- loop counters
   variable j: unsigned(4 downto 0) := "00000"; -- loop counter
   variable temp_a1, temp_a2, temp_a3, keyloop: unsigned(6 downto 0) :="0000000";

 begin
 init(c1_send_sem);
 init(c2_send_sem);
 forever: loop
```

```
 for loopcnt1 in 0 to 7 loop
  while(in_hs_rdy = in_hs_rcv) loop
   wait for 10 ns;
  end loop;
   fkey(loopcnt1) := key;
   case loopcnt1 is
   when 0 => temp_t_state(1 to 32)   := d_block;
   when 1 => temp_t_state(33 to 64)  := d_block;
   when 2 => temp_t_state(65 to 96)  := d_block;
   when 3 => temp_t_state(97 to 128) := d_block;
   when others => NULL;
   end case;

  in_hs_rcv <= not in_hs_rcv;
  wait for 10 ns;
 end loop;
 -- For AES-256 (Nk=8, Nr=14)
 -- For 256-bit : Nb * (Nr +1 )  = 4 * 15 = 60 ("0111100")
 -- For 192-bit :              = 4 * 13 = 52 ("0110100")
 -- For 128-bit :              = 4 * 11 = 44 ("0101100")

 i := "0001000";  -- start off with the value of Nk, in this case = 8
 j := "00000";    -- round counter

 while(i < "0111100") loop        -- i < Nb * (Nr +1 )
  temp_a1 := i - "0000001";
  wait for 10 ns;
   ---------------- i mod Nk = 0 -----------------
   bb1:= fkey(to_integer(temp_a1(5 downto 0)));
   r_oneto8(bb1, dd1);                                          -- RotWord(w[i-1])
   temp_vec1(1 to 32):=   fbsub_quad1(dd1(1 to 32));     -- SubWord(RotWord(w[i-1]))
   rco(j, cc1);                                              -- Rcon
   temp_a1 := i - "0001000";
   fkey(to_integer(i(5 downto 0))) := fkey(to_integer(temp_a1(5 downto 0))) xor
                                temp_vec1(1 to 32) xor cc1;   -- w[i-Nk] xor SubWord xor Rcon
   ------------------------------------------------

   keyloop := "0000001";
   while keyloop /= "0000100" loop
   ----moods unroll
    if (keyloop + i < "0111100") then
      temp_a1 := keyloop + i - "0001000";
      temp_a2 := keyloop + i - "0000001";
      temp_a3 := keyloop + i;
     fkey(to_integer(temp_a3(5 downto 0))) := fkey(to_integer(temp_a1(5 downto 0))) xor
                          fkey(to_integer(temp_a2(5 downto 0)));   -- w[i]  = w[i-Nk] xor temp
    end if;
    keyloop := keyloop + "0000001";
   end loop;

   ---------------- i mod Nk = 4 -----------------
   if(i + "0000100" < "0111100") then
    temp_a1 := i + "0000011";
    cc1 := fkey(to_integer(temp_a1(5 downto 0)));
    temp_vec1(1 to 32):=   fbsub_quad1(cc1(1 to 32));     -- SubWord(RotWord(w[i-1]))

    temp_a2 := i - "0000100";
    temp_a3 := i + "0000100";
    fkey(to_integer(temp_a3(5 downto 0))):= fkey(to_integer(temp_a2(5 downto 0))) xor
                                          temp_vec1(1 to 32); -- fkey
   end if;
   ------------------------------------------------
```

```
        keyloop := "0000101";
        while keyloop /= "0001000" loop
        ----moods unroll
          if(keyloop + i < "0111100") then
            temp_a1 := keyloop + i - "0001000";
            temp_a2 := keyloop + i - "0000001";
            temp_a3 := keyloop + i;
            fkey(to_integer(temp_a3(5 downto 0))) := fkey(to_integer(temp_a1(5 downto 0))) xor
                          fkey(to_integer(temp_a2(5 downto 0))); -- w[i]  = w[i-Nk] xor temp
          end if;
          keyloop := keyloop + "0000001";
        end loop;

        i := i + "0001000"; -- increment by Nk
        j := j + "00001";
      end loop;

      for loopcnt2 in 0 to 3 loop
        case loopcnt2 is
          when 0 => temp1 := std_logic_vector(fkey(0) xor temp_t_state(1 to 32));
          when 1 => temp1 := std_logic_vector(fkey(1) xor temp_t_state(33 to 64));
          when 2 => temp1 := std_logic_vector(fkey(2) xor temp_t_state(65 to 96));
          when 3 => temp1 := std_logic_vector(fkey(3) xor temp_t_state(97 to 128));
        end case;
        send(c1_send_sem, c1_send_ack, c1_send_data, temp1);
      end loop;

      i := "0000100"; -- start off with the 4th key, Keys 0 to 3 used in the first round
      for EncLoop1 in 1 to 14 loop -- For AES-256 (Nk=8, Nr=14)
        for EncLoop2 in 0 to 3 loop
          bb1 := fkey(to_integer(i(5 downto 0)));      -- fkey
          temp3 := std_logic_vector(bb1);
          send(c2_send_sem, c2_send_ack, c2_send_data, temp3);
          i := i + "0000001";
          wait for 10 ns;
        end loop;
      end loop;

    end loop forever;
    end Process Prs_1;
  --======================== ENCRYPTION ==============================--
    Prs_2: process --  Process module p_MOD_3
      variable bb2, cc2, dd2, ee2, temp_vec2: u_sign32;
      variable transition_state,  temp_transition_state : u_sign128;
      variable temp2, temp4: std_logic_vector(31 downto 0);
    begin
      init(c1_recv_sem);
      init(c2_recv_sem);
      forever: loop
        --======================== First Round ==========================--
        for loopcnt3 in 0 to 3 loop
          recv(c1_recv_sem, c1_recv_ack, c1_recv_data, temp2);
          case loopcnt3 is
          when 0 => transition_state(1 to 32)   := unsigned(temp2);
          when 1 => transition_state(33 to 64)  := unsigned(temp2);
          when 2 => transition_state(65 to 96)  := unsigned(temp2);
          when 3 => transition_state(97 to 128) := unsigned(temp2);
          when others => NULL;
          end case;
        end loop;
        --*************************************************************--
```

```
for EncLoop3 in 1 to 14 loop -- For AES-256 (Nk=8, Nr=14)

    for EncLoop4 in 0 to 3 loop
      recv(c2_recv_sem, c2_recv_ack, c2_recv_data, temp4);
      bb2 := unsigned(temp4);      -- fkey

       case EncLoop4 is
       when 0 =>
        temp_vec2(1 to 32) := transition_state(1 to 8) & transition_state(41 to 48) &
                              transition_state(81 to 88) & transition_state(121 to 128);
        if(EncLoop3 = 14) then
          dd2(1 to 32):= fbsub_quad2( temp_vec2 );              -- w[i-1] = SubWord(w[i-1])
          ee2 := dd2;
        else
          ftable_quad(temp_vec2, cc2 );                        -- Retrieve values from Forward Tables
          ee2 := cc2;
        end if;
        temp_transition_state(1 to 32) := bb2(1 to 32)  xor ee2(1 to 32);
       when 1 =>
        temp_vec2(1 to 32) := transition_state(33 to 40) & transition_state(73 to 80) &
                              transition_state(113 to 120) & transition_state(25 to 32);
        if(EncLoop3 = 14) then
          dd2(1 to 32):= fbsub_quad2( temp_vec2 );              -- w[i-1] = SubWord(w[i-1])
          ee2 := dd2;
        else
          ftable_quad(temp_vec2, cc2 );                        -- Retrieve values from Forward Tables
          ee2 := cc2;
        end if;
        temp_transition_state(33 to 64) := bb2(1 to 32)  xor ee2(1 to 32);
       when 2 =>
        temp_vec2(1 to 32) := transition_state(65 to 72) & transition_state(105 to 112) &
                              transition_state(17 to 24) & transition_state(57 to 64);
        if(EncLoop3 = 14) then
          dd2(1 to 32):= fbsub_quad2( temp_vec2 );              -- w[i-1] = SubWord(w[i-1])
          ee2 := dd2;
        else
          ftable_quad(temp_vec2, cc2 );                        -- Retrieve values from Forward Tables
          ee2 := cc2;
        end if;
        temp_transition_state(65 to 96) := bb2(1 to 32)  xor ee2(1 to 32);
       when 3 =>
        temp_vec2(1 to 32) := transition_state(97 to 104) & transition_state(9 to 16) &
                              transition_state(49 to 56) & transition_state(89 to 96);
        if(EncLoop3 = 14) then
          dd2(1 to 32):= fbsub_quad2( temp_vec2 );              -- w[i-1] = SubWord(w[i-1])
          ee2 := dd2;
        else
          ftable_quad(temp_vec2, cc2 );                        -- Retrieve values from Forward Tables
          ee2 := cc2;
        end if;
        temp_transition_state(97 to 128) := bb2(1 to 32)  xor ee2(1 to 32);
       when others => NULL;
       end case;
    end loop; -- for EncLoop4 in 0 to 3 loop
    transition_state(1 to 128) := temp_transition_state(1 to 128);

end loop; -- for EncLoop3 in 1 to 14 loop
```

```
    for loopcnt4 in 0 to 3 loop
      while(out_hs_rdy /= out_hs_rcv) loop
        wait for 10 ns;
      end loop;
      case loopcnt4 is
        when 0 => ciphertext <= transition_state(1 to 32);
        when 1 => ciphertext <= transition_state(33 to 64);
        when 2 => ciphertext <= transition_state(65 to 96);
        when 3 => ciphertext <= transition_state(97 to 128);
        when others => NULL;
      end case;
      out_hs_rdy <= not out_hs_rdy;
      wait for 10 ns;
    end loop; -- for loopcnt4 in 0 to 3 loop
    end loop forever;
  end process Prs_2;
end behaviour;
```

Figure D-25 VHDL of pipelined 256-bit advanced encryption standard example

The post-MOODS synthesis simulation of the pipelined multi-FPGA 256-bit AES example is given in Figure D-26. Zoom in views of the simulation showing inputs and outputs updates are given in Figure D-27. This 3-device multi-FPGA implementation has two explicit communication channels (*ExC 1* and *ExC 2*) connecting the pipeline stages. With a system clock period of 200 ns, the pipelined multi-FPGA 256-bit AES takes 1137 clock cycles (i.e. clock cycles = (231100 ns - 3700 ns) / 200 ns) to process the 128-bit data block using a 256-bit cipher key.

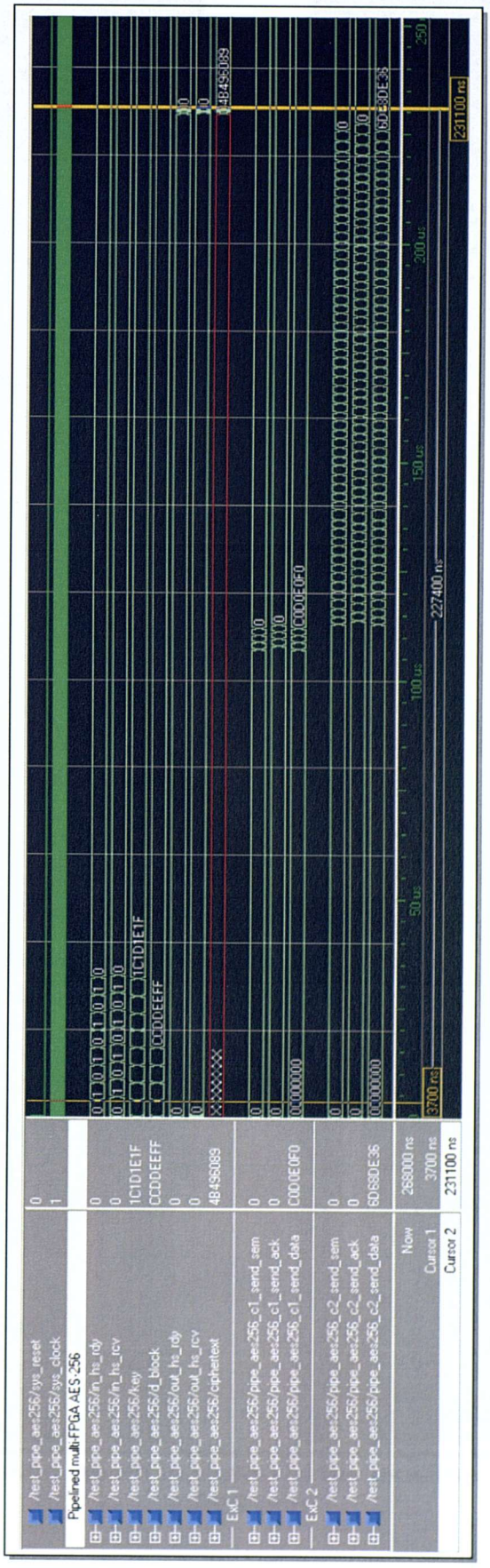


Figure D-26 Simulation of the pipelined multi-FPGA 256-bit AES core

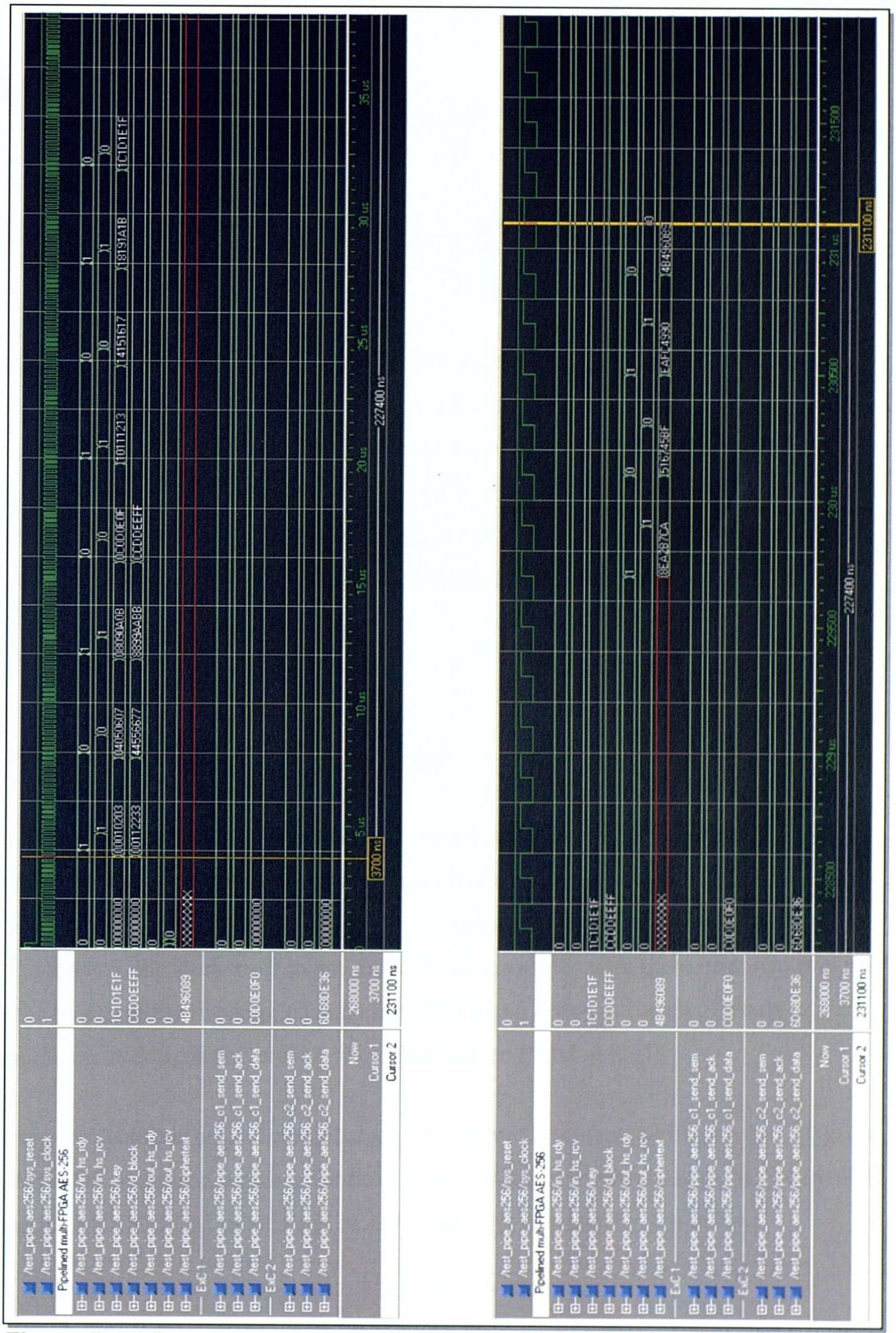


Figure D-27 Simulation (zoom in views) of the pipelined multi-FPGA 256-bit AES core

Appendix E

MOODS multi-FPGA synthesis guide

This appendix presents the partitioning options added to the MOODS synthesis system for multi-FPGA synthesis. The appendix covers the complete set of commands for multi-FPGA synthesis using the MOODS Command Line Interface (CLI) and the command line switches for the original MOODS synthesis core are briefly repeated when needed for the sake of completeness. Background information and a more detailed guide to the original MOODS synthesis system can be found in references [32, 39, 42, 161].

E.1 The MOODS optimiser

The MOODS Synthesis Suite organises the user designs into a project-based workspace environment with the inclusion and compilation of all the behavioural VHDL source files within the main project. Other projects can be imported, as *subprojects*, into an existing main project in the workspace to use the libraries associated with these imported projects. All the project files are compiled and assembled into a *library* structure. Details on the compilation of designs and project workspace can be found in [161].

Having the synthesis project compiled and set up into the corresponding project libraries, the MOODS optimiser, which is the heart of the MOODS Synthesis suite, can be invoked using the MOODS CLI in the form of a DOS-prompt command given below:

```
(MOODS root directory)\Bin\Moods design
-m "(project directory)\example.lmf"
    -w example
```

```
-pre-opt
-mult2shift
-prn_al
-prn_nl
-vhdl_out
-design_profile
-exchannels
{other arguments}
```

The above command assumes that a top-level design called *design* has been compiled and the main *project* name of the design is called *example*. File *example*.lmf contains information on the directory location of library files used in the project and this is passed to the optimiser through the -m argument preceding the location of the (.lmf) file. Argument -w specifies the directory where the output files generated from the synthesis are to be written to. The -pre-opt argument allows pre-scheduling optimisation to be performed on the design. At presents, the pre-scheduling optimisation only improves on designs with array and vector dynamic indexing. Argument -mult2shift forces constant divides, or multiplies by a positive power of two to be implemented as shift-left or -right operations respectively to get a significant hardware reduction. Argument -prn_al is included to append a dump of control arcs to the *design*.cg output file. Argument -prn_nl is included to append a dump of data path nets to the *design*.dpg output file. Argument -vhdl_out specifies that multiple VHDL netlist output files are generated, one for each target device. The first new argument, -design_profile is incorporated into the MOODS optimiser to enable multi-FPGA synthesis. It instructs MOODS to retrieve partitioning and design activity profile information (Section 4.5) in the *design*.par file in the project directory. A module call list *design*.mcl file is generated by MOODS during the prologue stage when the initial data structures are built. Details of the module call list file can be found in Appendix C.3. The second new argument, -exchannels enables the use of explicit communication channels (Section 4.2.2.1).

Other arguments exist [161], but exceed the scope of this appendix.

The basic steps in optimisation are:

1. Set up a "cost function" specifying the required target specification (e.g. target area and/or delay).
2. Run an optimisation algorithm.
3. Repeat the above if desired to achieve different results.
4. Set up and run the K-way partitioning process.
5. Repeat step 4 if desired to get different partitioning results.
6. Repeat steps 1 to 5 if desired to achieve different synthesis and partitioning results.
7. Run the communication subsystem optimisation algorithm.
8. Finish the design to produce final structural netlists suitable for targeting multiple FPGA devices.

E.1.1 Setting up a cost function

During the prologue stage in MOODS, the associated technology libraries are loaded and the input design is read in, followed by the initialisation of data structures. A number of messages about loading libraries and files, and preliminary tasks are displayed in the console window. When it finishes, a command prompt will appear, e.g.:

```
MOODS "C:\CAD\JPEG_demo\jpg_core_two\jpg_core_two" -->
```

The command "CF" is entered to get to the cost function definition menu of MOOD. At any point in the synthesis session, typing "?" at a command prompt gives a list of all available commands, as illustrated below in Figure E-1.

```
MOODS "c:\CAD\jpeg_demo\jpg_core_two\jpg_core_two" --> CF


Enter cost function command []: ?

SETTING THE COST FUNCTION
=========================

Type a two character string:
   first character:  A - to add a criterion
     (action)        D - to delete a criterion
                     C - change target
                     S - show the cost function
                     F - to finish
   second character: D - Total CP delay
     (criterion)     B - Delay between insts
                     A - Area
                     P - Power
                     N - Nets (no. of DP nets)
                     C - Clock period

Enter cost function command [?]:
```

Figure E-1 Cost function menu

The cost function allows the user to specify what the final optimised implementation should be like (e.g. how large or fast it is). Figure E-2 illustrates the typical steps to enter an area delay cost function, and specify a clock period for optimising the design. The following specifies area optimisation as the highest (first) priority with a target area of 0, and delay optimisation as the second priority with a target total delay of 0. Both target objectives are set to zero so that the final optimised implementation is as small and as fast as possible. Of course, non-zero target values can be given instead. A clock period of 20 ns is specified using the "AC" command and entering a value of 20 when asked to enter the new clock value at the subsequent prompt. With all of the cost function parameters set up, command "F" finishes the cost function definition and returns to the main MOODS prompt.

```
Enter cost function command [?]: AA

  Enter priority level (1 is highest) [1]: 1
     Initial total area is:  34505.6 Slices
     Enter target area (Slices) [ 34505.6]: 0


Enter cost function command [aa]: AD

  Enter priority level (1 is highest) [1]: 2
     Initial total CP delay is:   3016.2 ns
     Enter target total delay (ns) [  3016.2]: 0


Enter cost function command [ad]: AC

     Clock period has priority 1 and units in ns.
     Enter new clock value (ns) [    10.1]: 20


Enter cost function command [ac]: F
```

Figure E-2 Steps in setting a cost function in MOODS

E.1.2 Optimisation

After finishing the cost function set-up, the user can proceed to set up the optimisation algorithm and perform optimisation on the design. There are currently two main optimisation algorithms (described in Section 2.3.6) provided by the MOODS synthesis core. The quasi-exhaustive heuristics is the simplest and MOODS proceeds to optimise the design when the command "AOH" is entered at the MOODS prompt. Simulated annealing is slower and more complex, and is more difficult to operate, however it can produce better results, and also allow the design to be moved in many different directions round the design space. Figure E-3 illustrates the steps in setting up the optimisation parameters (annealing schedule), using the "AI" command. Once this data is entered, command "AO" starts the annealing process, optimising the design.

```
MOODS "c:\CAD\jpeg_demo\jpg_core_two\jpg_core_two" --> AI

Initializing Optimisation Data

   Enter start temperature [     0.0]: 50

   Enter terminating temperature [     0.0]: 0

   Enter factor to decrease temp (<1) OR -n for No. of steps [   100.0]: -100

   Enter maximum iteration per temperature range [0]: 500

MOODS "c:\CAD\jpeg_demo\jpg_core_two\jpg_core_two" -->
```

Figure E-3 Steps in setting up the annealing schedule in MOODS

E.2 K-way partitioning

When all the optimisation is completed, typing command "FI" at the MOODS prompt brings up the K-way partitioning prompt and typing "?" at the command prompt gives a list of all available commands, as illustrated below in Figure E-4.

```
K-way partitioning --> ?

                    K-way Partitioning Menu
                    =======================

               DS  - Display K-way partitioning setup
               EX  - Examine data structures
               EM  - Examine modules for partitions
               ET  - Examine target device details
               CT  - Change number of target devices
               CU  - Change max device utilisation (100 percent) value, D_max
               CL  - Change min device utilisation (20 percent) value, D_min
               CA  - Change offset target device areas
               CW  - Change data width
               TS  - Change to Strict balanced distribution over targeted devices
               MD  - Disable Multiple Subprogram Comm. Channels

               K-way partitioning (Optimised)
               KON     - with no added options
               KOL     - with locked modules

               K-way partitioning (with 2 partitions)
               KFN     - with no added options
               KFP     - with pre-allocated modules
               KFL     - with locked modules
               KFB     - with initial and locked modules
               RM      - Re-run MOODS optimisation

               FI  - Finish optimisation

K-way partitioning -->
```

Figure E-4 K-way partitioning menu

COMMAND	DESCRIPTION
DS	Displays the K-way partitioning set-up.
EX	This command is the same as the top-level MOODS command and it is used to examine the data structures for the design.
EM	The "EM" command leads to a set of further commands given in Figure E-5.
ET	This command leads to two further commands that allows the user to display and edit target device details (such as device area and I/O).
CT	This command is used to change the number of target devices used to implement the multi-FPGA system.
CU	This command is used to change the maximum percentage of device utilisation. The default value of 100 means the total logic (100%) capacity may be utilised if required.
CL	This command is used to specify the lowest percentage of the device area utilisation. This value is used to determine the balanced criterion in the K-way partitioning algorithm when a relaxed distribution of modules over the target devices is selected.
CA	This command is used to specify the device area offset percentage.
CW	CW is used to assign a fixed data bus width in the subprogram communication channel(s) for inter-device transfers.
TS/TR	Command TS changes the balanced criterion in the K-way partitioning algorithm to enforce a Strict balanced distribution of modules over targeted devices. Command TR allows a relaxed distribution of modules over targeted devices.
MD/ME	Command MD disables the generation of multiple subprogram communication channels, thereby connecting all communication cells to a single primary communication channel. Command ME enables the generation of multiple subprogram communication channels.
KON	This command invokes the K-way partitioning algorithm to partition the design with no pre-allocated and locked modules, and generate an optimised multi-FPGA system with the least number of target devices required.
KOL	This command invokes the K-way partitioning algorithm to partition the design with module(s) locked to specified target device(s), and generate an optimised multi-FPGA system with the least number of target devices required.
KFN	This command invokes the K-way partitioning algorithm to partition the design with no pre-allocated and locked modules, and generate an optimised multi-FPGA system using a fixed number of target devices.
KFP	This command invokes the K-way partitioning algorithm to partition the design with pre-allocated modules, and generate an optimised multi-FPGA system using a fixed number of target devices.
KFL	This command invokes the K-way partitioning algorithm to partition the design with module(s) locked to specified target device(s), and generate an optimised multi-FPGA system using a fixed number of target devices.
KFB	This command invokes the K-way partitioning algorithm to partition the design with pre-allocated and locked modules, and generate an optimised multi-FPGA system using a fixed number of target devices.
RM	This command is used to re-run the MOODS optimisation process.
FI	This command finishes and ends the K-way partitioning phase.

Table E-1 Complete set of commands in the K-way partitioning menu

A description of the complete set of commands in the K-way partitioning menu is given in Table E-1. Figure E-5 illustrates the further set of commands associated with the "EM" command in the main K-way partitioning menu. Process modules (locked/unlocked to the top-level architectural module) in the design are displayed using command "B". Commands "A" and "U" are used to lock and unlock process modules in the design. Commands "P" and "L" displays the pre-allocated and locked modules (if any) specified in the partitioning information (.par) file respectively. Command "E" allows the user to manually lock modules in the design to target devices, and a locked module can be unlocked using the "R" command.

```
K-way partitioning --> EM


Examine --> ?


                  Modules for k partitions
                  ========================

               B  - Display process modules
               A  - Lock process modules
               U  - Unlock process modules
               P  - Display pre-allocated modules
               L  - Display locked modules
               E  - Edit locked modules
               R  - Remove locked modules
               F  - Exit.

Examine -->
```

Figure E-5 Examine modules for partitioning menu

After setting up the partitioning parameters and running the K-way partitioning algorithm, the final partition of the design, together with the I/O utilisation and estimated area utilisation of target devices are displayed in the console window. The partitioning parameters can be altered and the K-way partitioning algorithm can be repeated to get different partitioning results, else command "FI" is entered at the K-way partitioning prompt to begin the communication subsystem optimisation. Alternatively, the MOODS optimisation process can be re-run using the "RM" command.

When the communication subsystem optimisation finishes, the system writes out VHDL packages for the subprogram communication channel arbiter(s) (Section 5.4.3), final netlists for all the target devices, and report files, leaving the system in the "EXAMINE" mode. The "FI" command is typed once more to end the session.

Adapting the same naming convention described in Section E.1, assuming the top-level design has been created from a behavioural VHDL file, `design.vhd`. After the multi-FPGA synthesis session in MOODS, VHDL netlist output files `design_synth_dom1.vhd`, `design_synth_dom2.vhd`, ..., `design_synth_dom`*k*`.vhd` for a multi-FPGA design targeting *k* devices are created in the project directory.