MPEG-2 Decoder User Guide

Koenraad De Vleeschauwer
kdv@kdvelectronics.eu

August 24, 2017

Copyright Notice

Copyright ©2007-2009, Koenraad De Vleeschauwer.

Redistribution and use in source (LyX format) and ‘compiled’ forms (PDF, PostScript, HTML, RTF, etc.), with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code (LyX format) must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in compiled form (transformed to other DTDs, converted to PDF, PostScript, HTML, RTF, and other formats) must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of the author may not be used to endorse or promote products derived from this documentation without specific prior written permission.

This documentation is provided by the author “as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the author be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this documentation, even if advised of the possibility of such damage.

MPEG-2 License Notice

Commercial implementations of MPEG-1 and MPEG-2 video, including shareware, are subject to royalty fees to patent holders. Many of these patents are general enough such that they are unavoidable regardless of implementation design.

MPEG-2 intermediate product. Use of this product in any manner that complies with the MPEG-2 standard is expressly prohibited without a license under applicable patents in the MPEG-2 patent portfolio, which license is available from MPEG LA, L.L.C., 250 Stelle Street, suite 300, Denver, Colorado 80206.

Contents

1 Processor Interface **5**
1.1 Decoder Block Diagram 5
1.2 Ports 7
1.2.1 Clocks 9
1.2.2 Reset 9
1.2.3 Stream Input 9
1.2.4 Register File Access 9
1.2.5 Memory Controller 9
1.2.6 Memory Request FIFO 10
1.2.7 Memory Response FIFO 10
1.2.8 Video Output 10
1.2.9 Test Point 11
1.2.10 Status 11
1.3 Processor Tasks 12
1.4 Registers 12
1.5 Read-only Registers 15
1.6 On-Screen Display 16
1.7 Frame Store 18
1.8 Video Modeline 18
1.9 Interrupts 21
1.10 Watchdog 22
1.11 Trick mode 24
1.12 Test point 26

2 Decoder Sources **27**
2.1 Source Directory Structure 27
2.2 MPEG2 Decoder 27
2.2.1 FIFO sizes 27
2.2.2 Dual-ported memory and FIFO models 30
2.2.3 Memory mapping 30
2.2.4 Modeline 31
2.2.5 Inverse Discrete Cosine Transform 32
2.2.6 Bilinear chroma upsampling 32
2.3 Simulation 33
2.3.1 Icarus Verilog Simulation 33
2.3.2 Conformance Tests 37

2.4 Tools 38
2.4.1 Logic Analyzer 38
2.4.2 Finite State Machine Graphs 38
2.4.3 IEEE-1180 IDCT Accuracy Test 41
2.4.4 Reference software decoder 41
2.4.5 MPEG2 Test Streams 41

1 Processor Interface

An MPEG2 decoder, implemented in Verilog, is presented. Chapter 1 describes the decoder for the software engineer who wishes to write a device driver.

1.1 Decoder Block Diagram

Figure 1.1 shows the MPEG2 decoder block diagram. An external source such as a DVB tuner or DVD drive provides an MPEG2 stream. The video elementary stream is extracted and sent to the decoder. The *video buffer* acts as a fifo between the incoming MPEG2 video stream and the variable length decoder. The video buffer evens out temporary differences between the bitrate of the incoming MPEG2 bitstream and the bitrate at which the decoder parses the bitstream.

The MPEG2 codec is a variable length codec; codewords which occur often occupy less bits than codewords which occur only rarely. *Getbits* provides a sliding window over the incoming stream. As the codewords have a variable length, the sliding window moves forward a variable amount of bits at a time.

Variable length decoding does the actual parsing of the bitstream. Variable length decoding stores stream parameters such as horizontal and vertical resolution, and produces run/length values and motion vectors. Run/length values and motion vectors are different ways of describing an image. The run/length values describe an image as compressed data contained within the bitstream. The motion vectors describe an image as a mosaic of already decoded images.

Run-length decoding, inverse quantizing and *inverse discrete cosine transform* decompress the run/length values.

Motion compensation retrieves already decoded images from memory and applies the motion vector translations.

The reconstructed image is the sum of the decompressed run/length values and translated pieces of already decoded images. The reconstructed image is stored in the *frame store* for later display and reference.

The frame store receives requests to store and retrieve pixels from three different sources:

- *motion compensation*, which writes reconstructed image frames to memory
- *chroma resampling*, which reads reconstructed image frames from memory for displaying

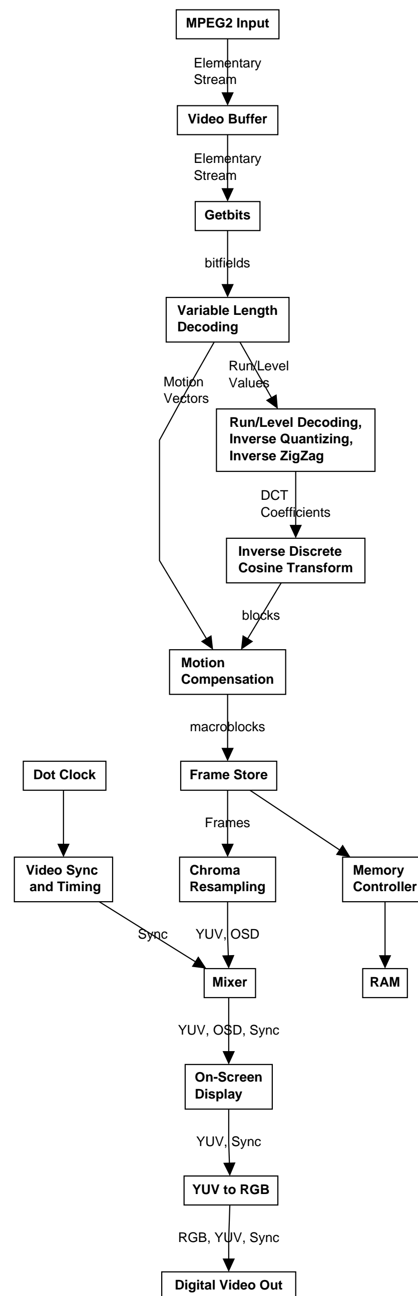

Figure 1.1: Decoder Block Diagram

- writes to the *on-screen display*, under software control.

Some of these blocks have multiple accesses to the frame store. Within the MPEG2 decoder a total of six memory read or write requests may occur simultaneously. The frame store prioritizes these requests and serializes them into a single stream of memory read/write requests, which is sent to the memory controller.

The *memory controller* is external to the MPEG2 decoder. The memory controller handles the low-level details of interfacing with the memory chips. If memory is static RAM, interfacing requires little more than a buffer; dynamic memory requires a more complex controller.

The MPEG2 decoder accepts 4:2:0 format video, in which color and brightness information have a different resolution: color information (chrominance) is sent at half the horizontal and half the vertical resolution of brightness information (luminance). This makes sense because the human eye uses different mechanisms to perceive color and brightness; and the different mechanisms used have different sensitivities.

Sending color information at half the horizontal and half the vertical resolution of brightness information implies the reconstructed image in the frame store has only one color pixel for every four brightness pixels. Assigning the same color information to the four pixels of brightness information would result in a chunky image. *Chroma resampling* does horizontal and vertical interpolation of the color information, resulting in a smooth color image.

A *dot clock* marks the frequency at which pixels are sent to the display. The dot clock is external to the MPEG2 decoder and can be either free running or synchronized to another clock.

The *video synchronization generator* counts pixels, lines and image frames at the dot clock frequency. At any given moment, the video synchronization generator knows the horizontal and vertical coordinate of the pixel to be displayed.

The pixels generated in chroma resampling and the coordinates generated by the video synchronization generator are joined in the *mixer*. The result is a stream of pixels, at the current horizontal/vertical coordinate, at the dot clock frequency.

At this point the *on-screen display* is added. The on-screen display has the same resolution as the video and uses a 256-color palette. Software can choose to put the on-screen display on top, completely hiding the video; or to blend on-screen display and video, as if they were two translucent glass plates.

The MPEG2 decoder works with chrominance (color) and luminance (brightness) information throughout. The final step is converting chrominance and luminance to red, green and blue in *yuv2rgb*. The red, green and blue information is the output of the decoder.

1.2 Ports

Table 1.2 lists MPEG2 decoder input/output ports.

Port	Bits	Description	I/O	Clock
clk	1	Decoder clock	I	-
dot_clk	1	Video clock	I	-
mem_clk	1	Memory Controller clock	I	-
rst	1	Reset	I	-
stream_data	8	Program stream data	I	clk
stream_valid	1	stream_data valid	I	clk
busy	1	Decoder busy flag	O	clk
reg_addr	4	Register address	I	clk
reg_dta_in	32	Register write data	I	clk
reg_wr_en	1	Register write enable	I	clk
reg_dta_out	32	Register read data	O	clk
reg_rd_en	1	Register read enable	I	clk
error	1	Decoding error flag	O	clk
interrupt	1	Interrupt	O	clk
watchdog_rst	1	Watchdog-generated Reset	O	clk
r	8	Red	O	dot_clk
g	8	Green	O	dot_clk
b	8	Blue	O	dot_clk
y	8	Y Luminance	O	dot_clk
u	8	Cr Chrominance	O	dot_clk
v	8	Cb Chrominance	O	dot_clk
pixel_en	1	Pixel enable	O	dot_clk
h_sync	1	Horizontal synchronization	O	dot_clk
v_sync	1	Vertical synchronization	O	dot_clk
c_sync	1	Composite synchronization	O	dot_clk
mem_req_rd_cmd	2	Memory request command	O	mem_clk
mem_req_rd_addr	22	Memory request address	O	mem_clk
mem_req_rd_dta	64	Memory request data	O	mem_clk
mem_req_rd_en	1	Memory request read enable	I	mem_clk
mem_req_rd_valid	1	Memory request valid	O	mem_clk
mem_res_wr_dta	64	Memory response data	I	mem_clk
mem_res_wr_en	1	Memory response enable	I	mem_clk
mem_res_wr_almost_full	1	Memory response almost full	O	mem_clk
testpoint_dip_en	1	Testpoint dip switches enable	I	-
testpoint_dip	4	Testpoint dip switches	I	-
testpoint	34	Logical analyzer test point	O	-

Table 1.2: Ports

1.2.1 Clocks

Up to three different clocks may be supplied to the MPEG2 decoder.

`clk` Main decoder clock, input.

`dot_clk` Video clock, input. Variable frequency, varying with current video modeline.

`mem_clk` Memory Controller Clock, input.

The decoder produces pixels at a maximum rate of one per `clk` cycle.

1.2.2 Reset

`rst` Asynchronous reset, input, active low, internally synchronized.

1.2.3 Stream Input

`stream_data` 8-bit elementary stream data, input, synchronous with `clk`, byte aligned. The elementary stream is an MPEG2 4:2:0 video elementary stream.

`stream_valid` elementary stream data valid, input, synchronous with `clk`. Assert when stream_data valid.

`busy` busy, active high, output, synchronous with `clk`. When high, indicates maintaining `stream_valid` high will overflow decoder input buffers.

1.2.4 Register File Access

`reg_addr` 5-bit register address, input, synchronous with `clk`.

`reg_dta_in` 32-bit register data in, input, synchronous with `clk`.

`reg_wr_en` register write enable, input, active high, synchronous with `clk`. Assert to write `reg_dta_in` to `reg_addr`.

`reg_dta_out` 32-bit register data out, output, synchronous with `clk`.

`reg_rd_en` Active high register read enable, input, synchronous with `clk`. Assert to obtain the contents of register `reg_addr` at `reg_dta_out`.

1.2.5 Memory Controller

The interface between MPEG2 decoder and memory controller consists of two fifos. The memory request FIFO sends memory read, write or refresh requests from decoder to memory controller. The memory response FIFO sends data read from memory controller to MPEG2 decoder. The data from the memory read requests appears in the memory response FIFO in the same order as the memory reads were issued in the memory request FIFO.

mem_req_rd_cmd	Mnemonic	Description
0	CMD_NOOP	No operation
1	CMD_REFRESH	Refresh memory
2	CMD_READ	Read 64-bit word
3	CMD_WRITE	Write 64-bit word

Table 1.3: Memory controller commands

1.2.6 Memory Request FIFO

`mem_req_rd_cmd` memory request command, output, synchronous with `mem_clk`. Valid values are defined in table 1.3.

`mem_req_rd_addr` 22-bit memory request address, output, synchronous with `mem_clk`.

`mem_req_rd_dta` 64-bit memory request data, output, synchronous with `mem_clk`.

`mem_req_rd_en` memory request read enable, input, active high, synchronous with `mem_clk`.

`mem_req_rd_valid` memory request read valid, output, active high, synchronous with `mem_clk`. Indicates when `mem_req_rd_cmd`, `mem_req_rd_addr` and `mem_req_rd_dta` have meaningful values.

1.2.7 Memory Response FIFO

`mem_res_wr_dta` 64-bit memory response write data, input, synchronous with `mem_clk`.

`mem_res_wr_en` memory response write enable, input, active high, synchronous with `mem_clk`. Assert to write `mem_res_wr_dta` to the memory response FIFO.

`mem_res_wr_almost_full` memory response write almost full, output, active high, synchronous with mem_clk. When high, indicates maintaining `mem_res_wr_en` high will overflow the memory response FIFO. The current clock cycle can be completed without overflowing the memory response FIFO.

1.2.8 Video Output

`r` red component, output, synchronous with `dot_clk`.

`g` green component, output, synchronous with `dot_clk`.

`b` blue component, output, synchronous with `dot_clk`.

`y` Y luminance, output, synchronous with `dot_clk`.

`u` Cr chrominance, output, synchronous with `dot_clk`.

`v` Cb chrominance, output, synchronous with `dot_clk`.

`pixel_en` pixel enable, output, active high, synchronous with `dot_clk`. When `pixel_en` is high, `r`, `g`, `b`, `y`, `u` and `v` are valid; when `pixel_en` is low video is blanked.

`h_sync` horizontal synchronization, output, active high, synchronous with `dot_clk`.

`v_sync` vertical synchronization, output, active high, synchronous with `dot_clk`.

`c_sync` composite synchronization, output, active low, synchronous with `dot_clk`.

1.2.9 Test Point

The decoder provides a test point for connecting a logic analyzer. The signals available at the test point can be selected either by software control, or using dip switches. The signals available at the test point are not defined as part of this specification, may vary even for implementations with the same status register version number and are subject to change without notice. See Verilog source `probe.v` for details.

`testpoint_dip_en` 1-bit input. If `testpoint_dip_en` is high, the registers visible at `testpoint` are selected using `testpoint_dip`. If `testpoint_dip_en` is low, the registers visible at `testpoint` output are selected using the `testpoint_sel` field of register 15.

`testpoint_dip` 4-bit input. `testpoint_dip` selects test point output if `testpoint_dip_en` is high.

`testpoint` 34-bit output. `testpoint` is a test point to connect a 34-channel logic analyzer probe to the MPEG2 decoder. Up to 16 different sets of signals are available, hardware selectable using the `testpoint_dip` dip switches or software selectable by writing to register 15. Any clocks present are on bits 32 and/or 33; bits 0 to 31 are data only. Bits 0 to 31 can also be accessed by software, by reading register 15.

1.2.10 Status

`error` error, output, active high, synchronous with `clk`. Indicates variable length decoding encountered an error in the bitstream.

`interrupt` interrupt, output, active high, synchronous with `clk`. Reading the status register allows software to determine the cause of the interrupt, and will clear the interrupt.

`watchdog_rst` watchdog-generated reset signal, output, active low, synchronous with `clk`. Normally high; low during one clock cycle if the watchdog timer expires.

1.3 Processor Tasks

To decode an MPEG-2 bitstream, the processor should execute the following tasks, in order:

1. Initialize the horizontal, horizontal sync, vertical, vertical sync and video mode registers with reasonable defaults. Clear `osd_enable`, `picture_hdr_intr_en` and `frame_end_intr_en`. Set the `video_ch_intr_en` flag.

2. Start feeding the MPEG-2 bitstream to the `stream_data` port of the decoder.

3. The decoder will issue an interrupt when video resolution or frame rate changes. Whenever the decoder issues an interrupt, clear the interrupt by reading the status register. Read the size, display size and frame rate registers. Calculate a new modeline, change dot clock frequency if necessary, and write the new video timing parameters to the horizontal, horizontal sync, vertical, vertical sync and video mode registers.

4. At bitstream end, pad the stream with 8 times hex 000001b7, the sequence end code (ISO/IEC 13818-2, par. 6.2.1, Start Codes).

If the On-Screen Display (OSD) is used, the processor should execute the following tasks as well:

1. Initialize the On-Screen Display color look-up table.

2. Wait until `horizontal_size` and `vertical_size` have meaningful values.

3. Write to the On-Screen Display.

4. Set `osd_enable` to one.

5. If a video change interrupt occurs, and `horizontal_size` or `vertical_size` has changed, rewrite the On-Screen Display.

Writing to the OSD is described in detail on page 16. Interrupt handling is treated on page 21.

1.4 Registers

The processor interface to the decoder consists of two times 16 32-bit registers. These registers can be divided in 16 read-mode registers (Table 1.4) and 16 write-mode registers (Table 1.5). The read-mode registers allow reading decoder status, while the write-mode registers allow setting video timing parameters and writing to the On-Screen Display (OSD).

	register	bits	content	read/write
0	version	15-0	version	r
1	status	15-8	matrix_coefficients	r
		7	watchdog_status	r
		6	osd_wr_en	r
		5	osd_wr_ack	r
		4	osd_wr_full	r
		3	picture_hdr	r
		2	frame_end	r
		1	video_ch	r
		0	error	r
2	size	29-16	horizontal_size	r
		13-0	vertical_size	r
3	display size	29-16	display_horizontal_size	r
		13-0	display_vertical_size	r
4	frame rate	15-12	aspect_ratio_information	r
		11	progressive_sequence	r
		10-6	frame_rate_extension_d	r
		5-4	frame_rate_extension_n	r
		3-0	frame_rate_code	r
f	testpoint	31-0	testpoint	r

Table 1.4: Read-mode Registers

	register	bits	content	read/write
0	stream	15-8	watchdog_interval	w
		3	osd_enable	w
		2	picture_hdr_intr_en	w
		1	frame_end_intr_en	w
		0	video_ch_intr_en	w
1	horizontal	27-16	horizontal_resolution	w
		11-0	horizontal_length	w
2	horizontal sync	27-16	horizontal_sync_start	w
		11-0	horizontal_sync_end	w
3	vertical	27-16	vertical_resolution	w
		11-0	vertical_length	w
4	vertical sync	27-16	vertical_sync_start	w
		11-0	vertical_sync_end	w
5	video mode	27-16	horizontal_halfline	w
		2	clip_display_size	w
		1	pixel_repetition	w
		0	interlaced	w
6	osd clt yuvm	31-24	y	w
		23-16	u	w
		15-8	v	w
		7-0	osd_clt_mode	w
7	osd clt addr	7-0	osd_clt_addr	w
8	osd dta high	31-0	osd_dta_high	w
9	osd dta low	31-0	osd_dta_low	w
a	osd_addr	31-29	osd_frame	w
		28-27	osd_comp	w
		26-16	osd_addr_x	w
		10-0	osd_addr_y	w
b	trick mode	10	deinterlace	w
		9-5	repeat_frame	w
		4	persistence	w
		3-1	source_select	w
		0	flush_vbuf	w
f	testpoint	3-0	testpoint_sel	w

Table 1.5: Write-mode Registers

1.5 Read-only Registers

`version` contains a non-zero FPGA bitstream (hardware) version number. Software should at least print a warning ``Warning: hardware version (%i.%i) more recent than software driver'' if the hardware version is higher than expected.

`picture_hdr` is set whenever an picture header is encountered in the bitstream. `picture_hdr` is cleared whenever the status register is read. In a well-behaved MPEG-2 stream, `horizontal_size`, `vertical_size`, `display_horizontal_size`, `display_vertical_size`, `aspect_ratio_information` and `frame_rate` will have meaningful values when a picture header is encountered.

`frame_end` is set when video vertical synchronization begins. `frame_end` is cleared whenever the status register is read.

`video_ch` is set whenever video resolution or frame rate changes. `video_ch` is cleared whenever the status register is read.

`error` is set when variable length decoding cannot parse the bitstream. `error` is cleared whenever the status register is read.

`watchdog_status` is high if the watchdog timer expired. `watchdog_status` is cleared whenever the status register is read.

`horizontal_size` is defined in ISO/IEC 13818-2, par. 6.2.2.1, par. 6.3.3.

`vertical_size` is defined in ISO/IEC 13818-2, par. 6.2.2.1, par. 6.3.3.

`display_horizontal_size` is defined in ISO/IEC 13818-2, par. 6.2.2.4, par. 6.3.6.

`display_vertical_size` is defined in ISO/IEC 13818-2, par. 6.2.2.4, par. 6.3.6.

`aspect_ratio_information` is defined in ISO/IEC 13818-2, par. 6.3.3.

`matrix_coefficients` is defined in ISO/IEC 13818-2, par. 6.3.6.

`frame_rate_extension_n` is defined in ISO/IEC 13818-2, par. 6.3.3, par. 6.3.5.

`frame_rate_code` is defined in ISO/IEC 13818-2, par. 6.3.3, Table 6-4.

`progressive_sequence` is defined in ISO/IEC 13818-2, par. 6.3.5.

`frame_rate_extension_d` is defined in ISO/IEC 13818-2, par. 6.3.3, par. 6.3.5.

osd_clt_mode	Comment
xxx00000	alpha = 0/16
xxx00001	alpha = 1/16
xxx00010	alpha = 2/16
xxx00011	alpha = 3/16
xxx00100	alpha = 4/16
xxx00101	alpha = 5/16
xxx00110	alpha = 6/16
xxx00111	alpha = 7/16
xxx01000	alpha = 8/16
xxx01001	alpha = 9/16
xxx01010	alpha = 10/16
xxx01011	alpha = 11/16
xxx01100	alpha = 12/16
xxx01101	alpha = 13/16
xxx01110	alpha = 14/16
xxx01111	alpha = 15/16
xxx11111	alpha = 16/16
xx0xxxxx	attenuate video pixel by alpha
xx1xxxxx	alpha blend osd and video pixel
00xxxxxx	display video pixel
01xxxxxx	display attenuated/alpha blended pixel
10xxxxxx	display osd pixel
11xxxxxx	display blinking osd pixel

Table 1.6: On-Screen Display Modes

1.6 On-Screen Display

The OSD has the same resolution and aspect ratio as the MPEG-2 video being displayed. If no MPEG-2 video is being displayed, the OSD is undefined. Note feeding the decoder a simple MPEG-2 sequence header with `horizontal_size` and `vertical_size` already satisfies the requirements for using the OSD.

The OSD is only shown if there is video output. If one wishes to display an OSD when no MPEG2 video is being reproduced, video output can be forced by setting `source_select` to 4, 5, 6 or 7.

The OSD may use up to 256 different colors. The OSD color lookup table (CLT) stores `y`, `u`, `v` and `osd_clt_mode` data for each color. The `y`, `u` and `v` values are interpreted as defined by `matrix_coefficients`. The `osd_clt_mode` value determines the color displayed according to Table 1.6. The different modes combine osd and video in various ways:

- video. This is the normal mode of operation.

- attenuated video. 16 discrete levels of attenuation can be used to fade video in or out.
- on-screen display.
- blend of on-screen display and video. 16 discrete levels of translucency.
- blinking on-screen display. Alternates between osd pixel and attenuated/alpha blended video pixel with a frequency of about one second.

`osd_enable` determines whether the On-Screen Display is shown or not. If `osd_enable` is low, the On-Screen Display is not shown. If `osd_enable` is high, the On-Screen Display is shown. The osd color lookup table has to be initialized and the osd has to be written before `osd_enable` is raised. `osd_enable` is 0 on power-up or reset.

`osd_wr_en` is set whenever an osd write is has been accepted, whether the osd write was successful or not. `osd_wr_en` is cleared whenever the status register is read.

`osd_wr_ack` is set whenever an osd write has been successful. `osd_wr_ack` is cleared whenever the status register is read.

`osd_wr_full` is set when the osd write fifo is full. When the osd write fifo is full, osd writes are not accepted.

When writing to the osd color lookup table:

1. Write `osd_clt_yuvm.`
2. Write `osd_clt_addr.`

Writes to the osd color lookup table take effect immediately.

When writing to the osd:

1. Only write to the osd when `horizontal_size` and `vertical_size` have meaningful values. This is the case when a picture header has been encountered.
2. Verify `osd_wr_full` is low. Writing when `osd_wr_full` is high has no effect.
3. Write the leftmost four pixels to `osd_dta_high`.
4. Write the rightmost four pixels to `osd_dta_low.`
5. Write x and y position of the leftmost pixel to `osd_addr`. Note x has to be a multiple of 8. `osd_frame` always has value 4 for OSD writes. `osd_comp` always has value 0 for OSD writes.
6. Read the status register until `osd_wr_en` is asserted. When `osd_wr_en` is high, the value of `osd_wr_ack` indicates whether the write was successful.

Writes to the osd pass through a 32-position fifo. This introduces some latency. Repeating the last osd write 32 times flushes fifo contents, ensuring osd memory has been updated.

osd_frame	Frame
0	0
1	1
2	2
3	3
4	OSD

Table 1.7: OSD Frame

osd_comp	Component
0	y
1	u
2	v

Table 1.8: OSD Component

1.7 Frame Store

Pixels can be written directly to the frame store, using the same mechanism as OSD writes. By writing pixels to the frame store and afterwards setting the `source_select` field of the trick register (described on page 24) arbitrary bitmaps can be shown.

The only difference between an OSD write and a frame store write is the value of `osd_frame` and/or `osd_comp`. Tables 1.7 and 1.8 list the frame and component codes. Frames 0 and 1 are used for storing I and P frames. Frames 2 and 3 are used for storing B frames. All frames are stored in 4:2:0 format, with u and v frames having half the width and height of the y frame. Note y, u and v values are stored in memory with an offset of 128.

Writes to the frame store are only defined when `horizontal_size` and `vertical_size` have meaningful values. Writes with `osd_frame` 4 are only defined when `osd_comp` is 0.

1.8 Video Modeline

The video timing parameters are:

- `horizontal_resolution`
- `horizontal_sync_start`
- `horizontal_sync_end`
- `horizontal_length`
- `vertical_resolution`
- `vertical_sync_start`

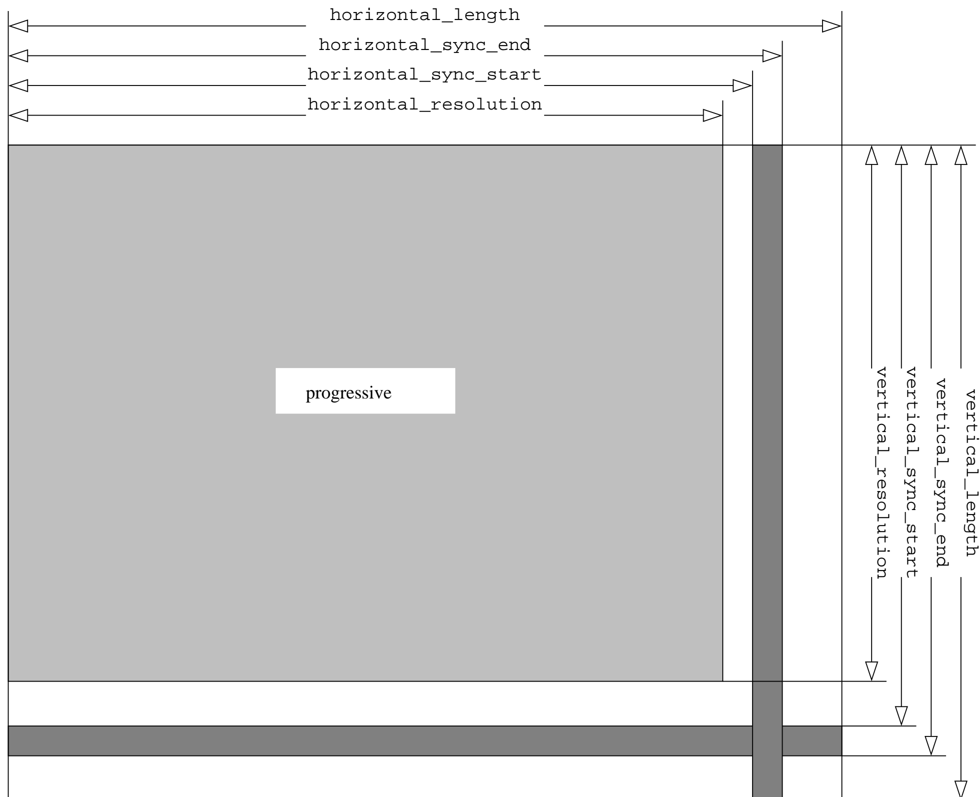

Figure 1.2: Progressive Video

- `vertical_sync_end`
- `vertical_length`
- `horizontal_halfline`
- `interlaced`
- `pixel_repetition`

These parameters can be deduced from the X11 modeline for the display, which is described in the "XFree86 Video Timings HOWTO". Writing to the internal registers which contain the video timing parameters will restart the video synchronization generator.

Two video timing diagrams are shown, one for progressive video (Figure 1.2) and one for interlaced video (Figure 1.3). The diagrams show the picture area (a light grey rectangle), flanked by horizontal sync (a dark grey vertical bar) and vertical sync (a dark grey horizontal bar).

`horizontal_resolution` number of dots per scan line.

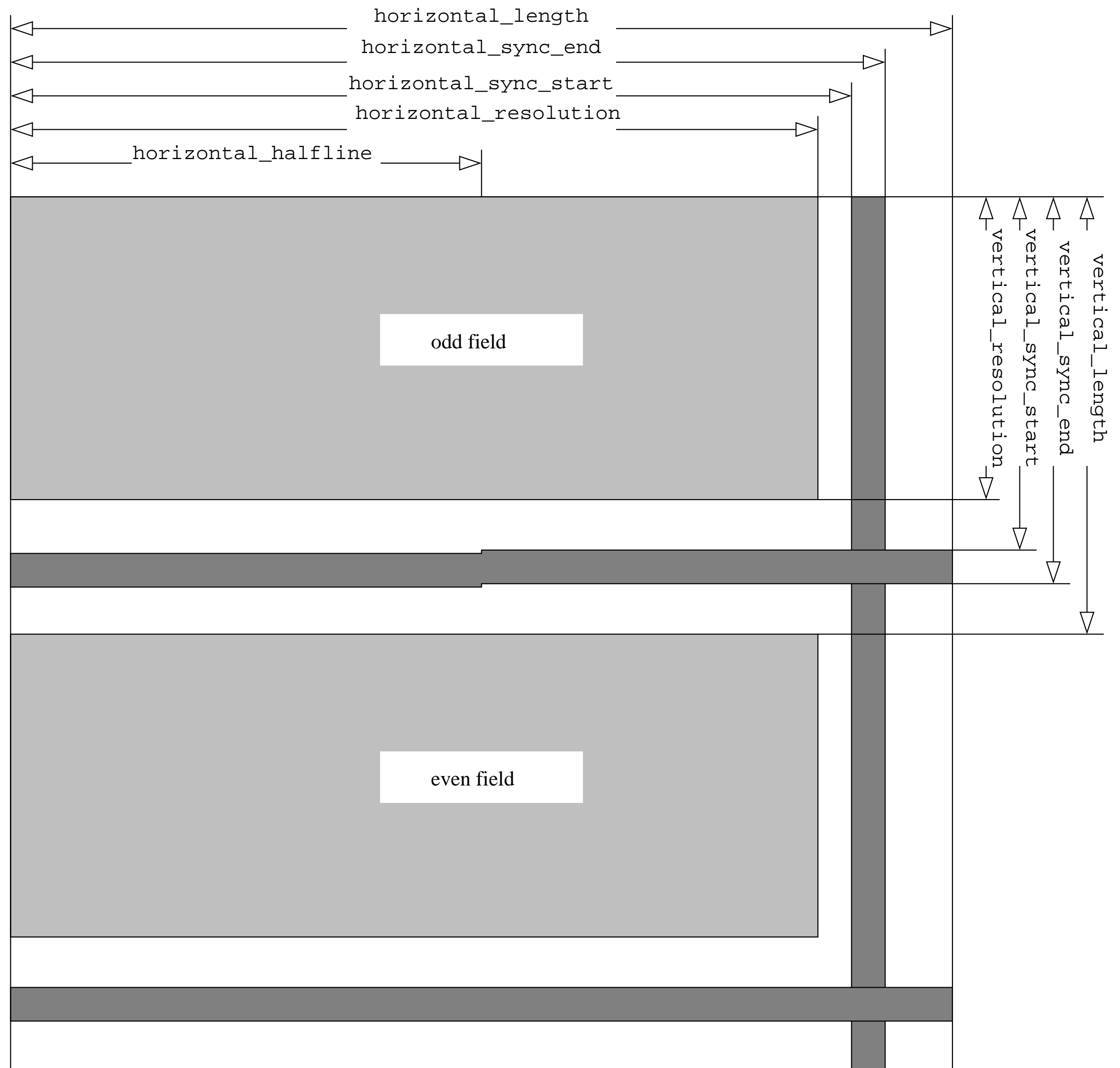

Figure 1.3: Interlaced Video

`horizontal_sync_start` used to specify the horizontal position the horizontal sync pulse begins. The leftmost pixel of a line has position zero.

`horizontal_sync_end` used to specify the horizontal position the horizontal sync pulse ends.

`horizontal_length` total length, in pixels, of one scan line.

`vertical_resolution` number of visible lines per frame (progressive) or field (interlaced).

`vertical_sync_start` used to specify the line number within the frame (progressive) or field (interlaced) the vertical sync pulse begins. The topmost line of a frame or field is line number zero.

`vertical_sync_end` used to specify the line number within the frame (progressive) or field (interlaced) the vertical sync pulse ends.

`horizontal_halfline` used to specify the horizontal position the vertical sync begins on odd fields of interlaced video. Not used in progressive mode.

`vertical_length` total number of lines of a vertical frame (progressive) or field (interlaced).

`clip_display_size` If asserted, the image is clipped to (`display_horizontal_size`, `display_vertical_size`). If not asserted, the image is clipped to (`horizontal_size`, `vertical_size`).

`interlaced` used to specify interlaced output is required. If `interlaced` is asserted, vertical sync is delayed one-half scan line at the end of odd fields.

`pixel_repetition` If `pixel_repetition` is asserted, each pixel is output twice. This can be used if the original dot clock is too low for the transmitter. As an example, suppose valid dot clock rates are 25…165 MHz, but the SDTV video being decoded has a dot clock of only 13.5 MHz. Asserting `pixel_repetition` and doubling dot clock frequency results in a dot clock of 27 MHz, sufficient for SDTV video to be transmitted across the link.

1.9 Interrupts

Three independent conditions may trigger an interrupt: when a picture header is encountered in the bitstream, when frame display ends, and when video resolution or frame rate changes. All three interrupt sources are optional and can be disabled individually.

When `picture_hdr_intr_en` is high and a picture header is encountered in the bitstream, `picture_hdr` is set and the interrupt signal is asserted until the status register is read. If `picture_hdr_intr_en` is low, the interrupt signal is never raised. `picture_hdr`

and picture_hdr_intr_en are 0 on power-up or reset. The picture header interrupt marks the "heartbeat" of the video decoding engine.

When video vertical synchronization begins and frame_end_intr_en is high, frame_end is set and the interrupt signal is asserted until the status register is read. If frame_end_intr_en is low, the interrupt signal is never raised. frame_end and frame_end_intr_en are 0 on power-up or reset. The frame end interrupt marks the "heartbeat" of the video display engine.

When one of horizontal_size, vertical_size, display_horizontal_size, display_vertical_size, progressive_sequence, aspect_ratio_information, frame_rate_code, frame_rate_extension_n, or frame_rate_extension_d changes, and video_ch_intr_en is high, video_ch is set and the interrupt signal is asserted until the status register is read. If video_ch_intr_en is low, the interrupt signal is never raised. video_ch and video_ch_intr_en are 0 on power-up or reset. The video change interrupt marks an abrupt change in the MPEG2 bitstream.

It is suggested that software, when receiving a video change interrupt:

1. Reads the size, display size and frame rate registers.
2. If frame_rate_code, frame_rate_extension_d or frame_rate_extension_n have changed, change dot clock frequency.
3. Calculates a video modeline, either using a look-up table or algebraically, e.g. using the VESA General Timing Formula.
4. Writes the new video modeline parameters to the horizontal, horizontal sync, vertical, vertical sync and video mode registers. This restarts the video synchronization.
5. If horizontal_size or vertical_size have changed and osd_enable is high, rewrite the On-Screen Display.

1.10 Watchdog

The MPEG2 decoder contains a watchdog circuit. The watchdog circuit resets the decoder if the decoder is unresponsive. The decoder is considered unresponsive if the decoder does not accept MPEG2 data for a period of time longer than the watchdog timeout interval. We outline how to configure the watchdog timeout interval, define under which conditions the watchdog circuit activates, and describe what happens when the watchdog timer expires.

The watchdog timeout interval can be configured by writing watchdog_interval, register 0, bits 15-8.

- writing 0 to watchdog_interval causes the watchdog timer to expire immediately.
- writing a value from 1 to 254, inclusive, to watchdog_interval enables the watchdog circuit.

- writing 255 decimal to `watchdog_interval` disables the watchdog circuit.

The default value of `watchdog_interval` is 127. If `watchdog_interval` has a value from 1 to 254, inclusive, the watchdog timeout is

$$watchdog_timeout = (watchdog_interval + 1).(repeat_frame + 1).2^{18}$$

`clk` clock cycles. `repeat_frame` (Section 1.11) determines the numer of times a decoded video frame is displayed. Each decoded video image is shown `repeat_frame` + 1 times. If a video frame is shown n times, the watchdog timeout is multiplied by n as well. This implies there is no need to adjust the watchdog timer if video is reproduced in slow motion.

The default value of `repeat_frame` is 0. If decoder `clk` frequency is 75 MHz the default watchdog timeout interval is 0.45 seconds.

The watchdog timer starts running when the decoder raises the `busy` signal. If the `busy` signal remains high for longer than the watchdog timeout interval, a reset is generated.

The watchdog timer is reset

- when the global `rst` input signal is driven low
- when the decoder `busy` signal is low
- when the decoder has been halted to show the current frame (`repeat_frame` is 31, *freeze-frame*)
- when the decoder has been halted to show a particular framestore frame (`source_select` is non-zero)
- when the watchdog circuit has been disabled (`watchdog_interval` has been set to 0 or to 255)
- during the first 2^{26} `clk` clock cycles after the watchdog timer expired, or the decoder was reset. This watchdog timer *holdoff* disables the watchdog during system initialisation. If clock frequency is 75 MHz, 2^{26} clock cycles corresponds to 0.89 seconds.

When the watchdog timer expires

- the `watchdog_rst` output pin becomes low during one `clk` clock cycle. The `watchdog_rst` output can be used to reset external hardware, or to generate a processor interrupt.
- the `watchdog_status` bit in the status register is set to 1. Software can detect whether the watchdog timer expired by checking `watchdog_status` in the status register. Reading the status register resets the `watchdog_status` bit back to 0.
- The framestore, On-Screen Display and circular video buffer are filled with zeroes.
- any data in the memory response fifo is discarded.

- `osd_enable` is set to 0. This disables the On-Screen Display, as the On-Screen Display now contains all zeroes.

- configuration data written to the register file is **not** modified when the watchdog expires. In particular, the video timing parameters (Sec. 1.8) remain unchanged.

The `watchdog_rst` output pin can optionally be used to reset external hardware when the watchdog expires. Examples of external hardware are the memory controller and the DVI dot clock generator. Note, however, resetting memory controller and DVI dot clock generator when the watchdog timer expires is optional.

The MPEG2 decoder does not require the external memory controller to be reset when the watchdog timer expires. When the watchdog timer expires, the MPEG2 decoder will write zeroes to all addresses from FRAME_0_Y to VBUF_END (`framestore_request.v`, STATE_CLEAR). When the watchdog timer expires, the MPEG2 decoder will also read and discard any data from the memory response fifo (`framestore_response.v`, STATE_FLUSH). These two actions re-synchronize MPEG2 decoder and external memory controller and bring memory to a known state.

The MPEG2 decoder also does not require the DVI clock generator to be reset when the watchdog expires. When the watchdog timer expires, the video timing parameters (Sec. 1.8) remain unchanged. If the DVI clock frequency remains unchanged when the watchdog timer expires, the decoder will continue with exactly the same video timing.

1.11 Trick mode

The trick mode register provides a toolbox for implementing non-standard playback modes. An example of a non-standard playback mode is slow motion. It is perhaps easiest to visualize trick mode settings as a pipeline (Figure 1.4).

`flush_vbuf` Writing one to `flush_vbuf` clears the incoming video buffer. Flushing the video buffer may be useful when changing channels.

`persistence` If `persistence` is set, and no new decoded image is available at frame start the last decoded image is shown again. If `persistence` is not set, and no new decoded image is available at frame start a blank screen is shown. `persistence` is 1 on power-up or reset.

`source_select` If zero, normal video is shown. Non-zero values allow continuous output of a blank screen, or a specific frame from the frame store, as in table 1.9. `source_select` is 0 on power-up or reset.

`repeat_frame` If zero, each decoded image is shown once. If non-zero, contains the number of times the decoded image will be additionally shown, as in table 1.10. A value of 31 shows the image indefinitely. `repeat_frame` is 0 on power-up or reset.

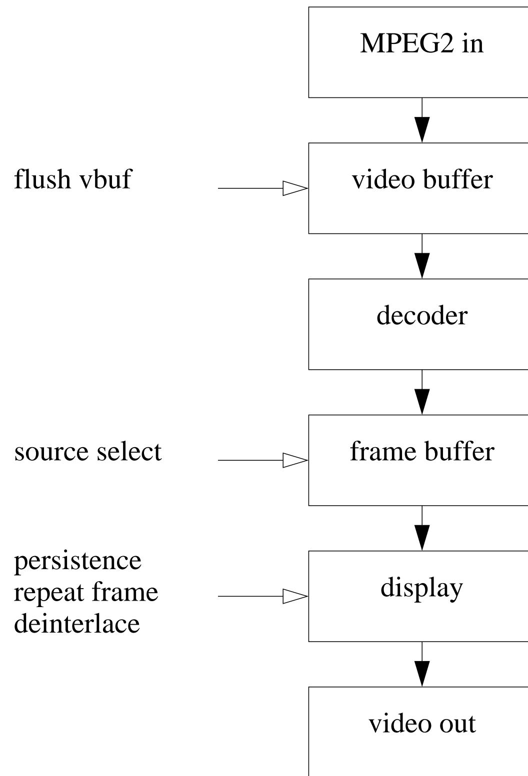

Figure 1.4: Trick mode pipeline

source_select	Frame shown
0	last decoded frame
1	blank screen
4	frame 0
5	frame 1
6	frame 2
7	frame 3

Table 1.9: Source Select

repeat_frame	times shown
0	1
1	2
2	3
...	
30	31
31	forever

Table 1.10: Repeat Frame

`deinterlace` Setting `deinterlace` high forces the decoder to output video as frames, even if the MPEG2 stream is interlaced. This can be used to reproduce interlaced MPEG2 streams on progressive displays. Setting `deinterlace` is not recommended when reproducing a progressive MPEG2 stream on a progressive display. Setting `deinterlace` has no effect if the video modeline specifies interlaced output (`interlaced` set). Note no spatial or temporal interpolation is done ("weaving").

1.12 Test point

The MPEG2 decoder provides a test point for connecting a logic analyzer. Internally, the decoder contains various test points, only one of which is actually output to the logic analyzer. Which internal test point is output to the logic analyzer is determined by the contents of `testpoint_sel`. The value of bits 0..31 of the test point can also be read by software. While this is no substitute for a logic analyzer, it is recognized that in many cases this may be the only option available.

`testpoint_sel` Used in hardware debugging. Determines which internal test point is multiplexed to the 34-channel logical analyzer test point.

`testpoint` Used in hardware debugging. Provides the current value of bits 0 to 31 of the 34-channel logical analyzer test point.

2 Decoder Sources

Chapter 2 provides an overview of the decoder sources for the hardware engineer who wishes to synthesize or modify the decoder.

2.1 Source Directory Structure

The source files are organized in directories as follows:

`bench/`	`iverilog`	Icarus behavioral simulation, page 33
`doc/`		Documentation
`rtl/`	`mpeg2`	MPEG2 decoder, page 27
`tools/`	`fsmgraph`	Finite state machine graphs, page 38
	`ieee1180`	IEEE1180 IDCT accuracy test, page 41
	`logicport`	Logicport logic analyzer, page 38
	`mpeg2dec`	Reference MPEG2 decoder, page 41
	`streams`	MPEG2 test streams, page 41

A linux system with Icarus Verilog is suggested, but not required, as development environment.

2.2 MPEG2 Decoder

The `rtl/mpeg2` directory contains the sources of the MPEG2 decoder itself. This section describes the changes most likely to be needed when instantiating the decoder: changing default modeline, changing FIFO sizes, choosing dual-ported ram and fifo models, changing memory mapping. In addition, references are provided for the IDCT and bilinear chroma upsampling algorithms.

2.2.1 FIFO sizes

Fifo depth and almost full/almost empty thresholds are defined in `fifo_size.v`. Note setting fifo depths and thresholds to arbitrary values can result in decoder deadlock.

Figure 2.1 shows MPEG2 decoder data flow. Together, `framestore_request`, memory controller and `framestore_response` implement the framestore. Communication with the framestore is through fifos. The incoming MPEG2 stream is written to `vbuf_write_fifo`. `framestore_request` reads the stream from `vbuf_write_fifo` and writes it to the circular video buffer in memory. If `vbuf_read_fifo` is almost empty, `framestore_request` issues memory read requests for the circular video buffer. `framestore_response` receives data from the circular video buffer and writes the data

to vbuf_read_fifo. The net result is vbuf_write_fifo, circular video buffer and vbuf_read_fifo acting as a single, huge fifo.

Variable-length decoding reads the MPEG2 stream from vbuf_read_fifo, and produces motion vectors and run/length codes. Run/length decoding, inverse quantizing, inverse zig-zag and inverse discrete cosine transform (IDCT) read the run/length codes and produce the prediction error. The prediction error is written to predict_err_fifo, one row of eight pixels at a time.

Motion compensation address generation motcomp_addrgen translates the motion vectors into three sets of memory addresses: the addresses where the forward motion compensation pixels can be read, the addresses where the backward motion compensation pixels can be read, and the addresses where the reconstructed pixels can be written. The addresses of the pixels needed for forward and backward motion compensation are written to the fwd_reader and bwd_reader address fifos. The address of the reconstructed pixels is written to the motion compensation destination fifo, dst_fifo. The memory subsystem reads the fwd_reader and bwd_reader address fifos, and writes the pixel values to the fwd_reader and bwd_reader data fifos.

Motion compensation reconstruction motcomp_recon adds pixel values read from forward motion compensation data fifo, backward motion compensation data fifo and prediction error, and writes the result to the address read from the motion compensation destination fifo.

Displaying the video image requires chroma resampling and yuv to rgb conversion. Resampling address generation resample_addrgen scans the reconstructed video image, line by line. The addresses of the pixels are written to the disp_reader address fifo. The memory subsystem reads the addresses from disp_reader address fifo and writes the pixel values to the disp_reader data fifo. resample_dta reads the pixel values from the disp_reader data fifo, while resample_bilinear does the actual bilinear chroma upsampling calculations. After conversion from yuv to rgb, the pixels are written to the pixel queue pixel_queue which adapts between decoder and DVI clocks.

Note the memory tag fifo mem_tag_fifo between framestore_request and framestore_response. For every memory read request, framestore_request writes a tag to the memory tag fifo. The tag identifies the source of the memory read request: circular video buffer, forward and backward motion compensation, or resampling. For every data word received from memory, framestore_response reads a tag from the memory tag fifo, and writes the data word received from memory to the data fifo corresponding to the tag. If the memory tag fifo is almost full, framestore_request stops issuing memory read or write requests. As a result, the number of outstanding memory read requests is always less than or equal to the size of the memory tag fifo.

When modifying fifo_size.v, care should be taken the fifos can never overflow. Note that when framestore_request stops issuing memory read requests, there still may be outstanding memory read requests in the memory request queue. The number of outstanding memory read requests is always smaller than, or equal to, the size of the memory tag fifo. When modifying fifo_size.v, remember fifos which receive data from memory may receive outstanding data, even after framestore_request has stopped sending memory read requests.

Figure 2.1: MPEG2 decoder dataflow

2.2.2 Dual-ported memory and FIFO models

FPGAs typically provide dedicated on-chip fifo's and dual-port RAMs. The designer then has to choose between using vendor-provided FIFOs and dual-port RAMs or writing his own.

The file `wrappers.v` defines the implementation of all dual-port RAMs and fifos in the design. For each component, two versions are provided: one where read and write port share a common clock; and one where read and write port have independent clocks.

`dpram_sc` dual-ported ram, same clock for read and write ports

`dpram_dc` dual-ported ram, different clock for read and write ports

`fifo_sc` fifo, same clock for read and write ports

`fifo_dc` fifo, different clock for read and write ports

The dual-ported rams are inferred from code in `wrappers.v`. The fifos can be either implemented in Verilog, or instantiated as FPGA primitives, depending upon `wrappers.v`. Following fifo models are available:

`xfifo_sc.v` fifo, same clock for read and write port.

`generic_fifo_sc_b.v` OpenCores generic fifo, different clock for read and write ports.

`xilinx_fifo_sc.v` Xilinx Virtex-5 fifo, same clock for read and write ports. Uses `xilinx_fifo.v`, `xilinx_fifo144.v` and `xilinx_fifo216.v`.

`xilinx_fifo_dc.v` Xilinx Virtex-5 fifo, different clock for read and write ports. Uses `xilinx_fifo.v`, `xilinx_fifo144.v` and `xilinx_fifo216.v`.

`xilinx_fifo_sc.v` and `xilinx_fifo_dc.v` implement fifos using FIFO18, FIFO18_36, FIFO36 or FIFO36_72 Virtex-5 primitives. Table 2.1 lists available data and address widths. If a `xilinx_fifo_sc.v` or a `xilinx_fifo_dc.v` is instantiated with data and/or address widths different from those in Table 2.1, the actual fifo will be larger and/or wider.

2.2.3 Memory mapping

The MPEG2 decoder memory mapping is defined in `rtl/mpeg2/mem_codes.v`. The default memory mapping needs 4 mbyte RAM and is sufficient for SDTV. By defining `MP_AT_HL` an alternative memory mapping can be chosen which requires 16 mbyte RAM and is sufficient for HDTV.

Translation of macroblock addresses to memory addresses is implemented in `rtl/mpeg2/mem_addr.v`. A macroblock address, a signed motion vector (mv_x, mv_y) with halfpixel precision, and an signed offset (delta_x, delta_y) with pixel precision are translated to an address in memory.

Data bits	Address bits	FIFO Depth	Implementation
4	13	8192	FIFO36
4	12	4096	FIFO18
9	12	4096	FIFO36
9	11	2048	FIFO18
18	11	2048	FIFO36
18	10	1024	FIFO18
36	10	1024	FIFO36
36	9	512	FIFO18
72	9	512	FIFO36_72
144	9	512	2 * FIFO36_72
216	9	512	3 * FIFO36_72

Table 2.1: Xilinx FIFO address widths

The macroblock address is assumed to iterate over all allowable values: beginning at zero, incrementing by one, until after the final macroblock the macroblock address is reset to zero. Macroblock address has to be initialized to zero, or an error condition results. Macroblock address changes other than incrementing by one, remaining unchanged or resetting to zero also result in an error condition.

Note the motion vector (mv_x, mv_y) is scaled by a factor two when accessing chrominance as defined in [1, par. 7.6.3.7]. The offset (delta_x, delta_y) remains unchanged when accessing chrominance blocks.

The translation of macroblock addresses and motion vectors to memory addresses in `rtl/mpeg2/mem_addr.v` has to be kept synchronized with the framestore dump task `write_framestore` in `rtl/sim/mem_ctl.v`, else the framestore dumps made during simulation will not accurately represent framestore contents.

Note out-of-range memory accesses are translated to the ADDR_ERR address. If a memory request with address `mem_req_rd_addr` equal to ADDR_ERR occurs during simulation, simulation stops with an error message.

The MPEG2 decoder zeroes out the framestore after system reset or when the watchdog timer expires. The MPEG2 decoder writes zeroes to all addresses from FRAME_0_Y to VBUF_END when the `rst` input pin goes low or when the `watchdog_rst` pin goes low.

2.2.4 Modeline

The default modeline is 800x600 progressive @ 60 Hz (SVGA). The `modeline.v` source contains the modeline parameters, and can be edited to change horizontal and vertical resolution, sync pulse width and position. The default pixel frequency on the ML505 is 38.21 MHz, and is defined in `dotclock_synthesizer.v`. Note `dotclock_synthesizer.v` synthesizes two frequencies, dotclock and dotclock90, equal in frequency but 90 degrees

phase shifted. The frequency synthesized is

$$f_{out} = f_{osc}.r.\frac{DCM_ADV_INST.CLKFX_MULTIPLY}{DCM_ADV_INST.CLKFX_DIVIDE}$$

where f_{osc} is the 100 MHz user clock frequency

$$f_{osc} = 100$$

and

$$r = \frac{PLL_ADV_INST.CLKFBOUT_MULT}{PLL_ADV_INST.CLKOUT1_DIVIDE} = 0.25$$

To change pixel frequency, first calculate the multiplier and divider for the new frequency. Suppose one wishes to synthesize a frequency of 35 MHz:

```
macpro mpeg2ether # ./mpeg2ether --dot_clock 35
dotclock ftarget =  35.00 fout =  35.00 MHz
 multiplier:  7 divider:  5
 high frequency mode: 0 ch7301 lowfreq: 1 ch7301 colorbars: 0
```

A pixel frequency of 35 MHz requires a multiplier of 7 and a divider of 5, with lowfreq asserted. Hence, in `dvi/dotclock.v`:

```
parameter [7:0]
  DEFAULT_DIVIDER       = 8'd4, // Divider minus one, actually
  DEFAULT_MULTIPLIER    = 8'd6; // Multiplier minus one, actually
parameter
  DEFAULT_LOWFREQ       = 1'b1
```

Note the modeline can be configured at any time using the `mpeg2ether` utility; it is only when changing the *default* modeline that modifying the sources is necessary. The `mpeg2ether` utility is explained on page **??**.

2.2.5 Inverse Discrete Cosine Transform

The IDCT algorithm used is described in [4]. A copy of document [4] can be found in the `doc` directory. The IDCT implementation uses 12 18x18 multipliers and two dual-port rams, and can do streaming. Run-length decoding (`rld.v`), inverse quantizing (`iquant.v`, `zigzag_table.v`) and IDCT transform (`idct.v`) all operate at the same speed of one pixel per clock. The IDCT meets the requirements of the former IEEE-1180.

2.2.6 Bilinear chroma upsampling

The chrominance components have half the vertical and half the horizontal resolution of the luminance. To obtain equal chrominance and luminance resolution, bilinear chroma upsampling is used. Bilinear chroma upsampling computes chroma pixel values by vertical and horizontal interpolation. Vertical interpolation implies adding two rows of

Source	Description
`resample.v`	Upsampling top-level file
`resample_addrgen.v`	Generates memory addresses of chroma/lumi rows
`resample_dta.v`	Reads chroma/lumi rows from memory
`resample_bilinear.v`	Performs bilinear upsampling calculations

Table 2.2: Upsampling source files

chroma values with different weights. The chroma row closest to the luma row gets weight 3/4, while the chroma row farthest from the luma row gets weight 1/4. The document `doc/bilinear.pdf` shows the weights used.

Bilinear chroma upsampling is implemented in various source files, as described in Table 2.2.

2.3 Simulation

Behavioral simulation using Icarus Verilog is described. For timing simulation consult your synthesis software.

2.3.1 Icarus Verilog Simulation

Behavioral simulation of the decoder can be performed using Icarus Verilog. The Icarus Verilog testbench in the `bench/iverilog` directory contains the following files:

`testbench.v` Top-level Verilog source; instantiates MPEG2 decoder.

`mem_ctl.v` Simple memory controller, for simulation only.

`Makefile` Makefile to create and run the simulation.

`wrappers.v` Wrapper for dual-port ram and fifos. Implements synchronous fifos using `xfifo_sc.v`, and implements asynchronous fifos as OpenCores `generic_fifo_sc_b.v`.

`generic_dpram.v, generic_fifo_dc.v, generic_fifo_sc_b.v` Opencores generic fifos.

Create the decoder is easy using the accompanying `Makefile`. First, remove any files left over from a previous simulation:

```
koen@macpro ~/xilinx/mpeg2/bench/iverilog $ make clean
rm -f mpeg2 stream.dat testbench.lxt trace framestore_*.ppm tv_out_*.ppm
```

Now create the decoder:

```
koen@macpro ~/xilinx/mpeg2/bench/iverilog $ make
iverilog -D__IVERILOG__ -DMODELINE_SIF -I ../../rtl/mpeg2 -o mpeg2
 testbench.v mem_ctl.v wrappers.v generic_fifo_dc.v
 generic_fifo_sc_b.v generic_dpram.v ../../rtl/mpeg2/mpeg2video.v
../../rtl/mpeg2/vbuf.v ../../rtl/mpeg2/getbits.v
xxd -c 1 ../../tools/streams/stream-susi.mpg |
 cut -d\  -f 2 > stream.dat
```

This executes two commands:

- `iverilog` to compile the Verilog sources to an executable, `mpeg2`.
- `xxd` to convert the binary MPEG2 program stream file `stream.mpg` to an ASCII file `stream.dat`, which the simulator can load.

When compiling the Verilog sources, two Verilog parameters are defined on the command line: `__IVERILOG__` and `MODELINE_SIF`. The first Verilog define, `__IVERILOG__` , is defined only during simulation, and never during synthesis. It is used to enable several run-time checks which only make sense in a simulation environment. The second Verilog define, `MODELINE_SIF`, chooses one of several pre-defined video output formats from `modeline.v`.

Finally, run the newly created executable `mpeg2`:

```
koen@macpro ~/xilinx/mpeg2/bench/iverilog $ make test
IVERILOG_DUMPER=lxt ./mpeg2
LXT info: dumpfile testbench.lxt opened for output.
$readmemh(stream.dat): Not enough words in the read file for
 requested range.
testbench.mem_ctl.write_framestore      dumping framestore to
               framestore_000.ppm @  0.02 ms
testbench.mem_ctl.write_framestore      dumping framestore to
               framestore_001.ppm @  0.02 ms
testbench.mpeg2.motcomp macroblock_address:    0
testbench.mpeg2.motcomp macroblock_address:    1
testbench.mpeg2.motcomp macroblock_address:    2
testbench.mpeg2.motcomp macroblock_address:    3
```

During simulation, the environment variable `IVERILOG_DUMPER=lxt` is set. This instructs the simulator to produce a dumpfile in the more compact `lxt` format, instead of the default `vcd` format.

By default, simulator output includes the macroblock address. This allows easy monitoring of decoder progress.

Each Verilog source file contains a `define DEBUG` statement, which can be uncommented or commented to switch trace output for that particular source file on or off.

During simulation, two kinds of graphics files are written: framestore dumps `framestore_*.ppm` and video captures `tv_out_*.ppm`. The framestore is where the decoder stores already decoded images. These are Portable Pixmap graphics files in ASCII format. Figure 2.2 shows a sample framestore dump.

The framestore consists of four frames and the on-screen display (OSD). The first two frames contain I and P pictures, while the last two frames contain B-pictures. Each frame consists of y (luminance), u and v (chrominance) information, with u and v having half the horizontal and half the vertical resolution of y. In the framestore dump, uninitialized memory is displayed in green. Looking at figure 2.2, one can see that the first three frames of the framestore have already been written; the decoder is halfway through the fourth frame. The On-Screen Display, at the bottom of the framestore dump, has not been initialized yet.

During simulation, by default, the framestore is dumped whenever a new frame begins; and every 200 macroblocks. As a framestore dump is a graphics file in ASCII format, one can also look at the file using standard text file utilities. These are the first 12 lines of a sample framestore dump:

```
koen@macpro ~/xilinx/mpeg2/bench/iverilog $ head -12 framestore_0001.ppm
P3
# mpeg2 framestore dump @ 11.81 ms
# frame number 2
# horizontal_size 352
# vertical_size 288
# display_horizontal_size 0
# display_vertical_size 0
# mb_width 22
# mb_height 18
# picture_structure frame picture
# chroma_format 420
352 2618 255
255 255 255 255 255 255 255 255 255 255 255 255
 255 255 255 255 255 255 255 255 255 255 255 255
```

The header of the framestore dump contains information about decoder status at the moment of the dump.

Figure 2.3 shows video capture file `tv_out_0000.ppm`.Horizontal sync is displayed as a vertical black stripe, to the right of the image. Vertical sync is displayed as a horizontal black stripe, below the image area. Blanking is displayed in a dark grey. The position of picture, horizontal sync and vertical sync in figure 2.3 is as defined in figure 1.2. As with the framestore dumps, one can look at `tv_out_0000.ppm` using standard text utilities.

```
koen@macpro ~/xilinx/mpeg2/bench/iverilog $ head -10 tv_out_0000.ppm
```

Figure 2.2: Framestore dump

Figure 2.3: Video output capture

```
P3
# picture 1 @ 10.73 ms
# horizontal resolution 352 sync_start 381 sync_end 388 length 458
# vertical resolution 288 sync_start 295 sync_end 298 length 315
# interlaced 0 halfline 175
459 316 255
0 0 0
0 77 0
3 0 3
2 0 2
```

The header of the video capture file contains information about the video modeline at the moment of video capture.

To end the simulation, go to the window where iverilog is running and type `ctrl-c finish`. The simulator will finish writing `trace` and `testbench.lxt` files, and return control to the command prompt.

The binary file `testbench.lxt` is a log of all wire and register changes which occurred during simulation. `testbench.lxt` can be displayed using vcd viewers such as `gtkwave`.

```
koen@macpro ~/xilinx/mpeg2/bench/iverilog $ gtkwave testbench.lxt &
```

Once `testbench.lxt` file has been loaded in `gtkwave`, internal decoder wires and registers can be displayed as waveforms.

2.3.2 Conformance Tests

The `bench/conformance` directory contains a testbench for the ISO/IEC 13818-4 MPEG2 conformance tests. The testbench assumes the ISO/IEC 13818-4 conformance test bit-

streams are available on your system. The ISO/IEC 13818-4 MPEG2 Conformance test bitstreams for Main Profile @ Main Level can be downloaded from the ISO web site using the `tools/streams/retrieve` script.

Typing `make clean test` in the `bench/conformance` directory simulates all MP@ML conformance test bitstreams. Table 2.3 summarizes test results.

When running the compatibility tests, note the decoder is not MPEG1-compatible, and does not decode MPEG1 streams. The MPEG2 decoder decodes MPEG2 4:2:0 program streams only.

2.4 Tools

The `tools` directory contains various utilities and tools used during decoder development and test.

2.4.1 Logic Analyzer

On the Xilinx ML505, the MPEG2 decoder testpoint has been broken out to the Xilinx Generic Interface (XGI) . The test point selection can be done using the GPIO DIP switches. If the ML505 is held so the LCD can be read, the GPIO DIP switches are at the bottom right of the board. GPIO DIP switches are numbered 1 to 8, from left to right.

If GPIO DIP switch 3 is off, test point selection is made by writing to register 15 decimal, REG_WR_TESTPOINT. If GPIO DIP switch 3 is on, test point selection is made by dip switches 5 to 8. GPIO DIP switch 5 is MSB, GPIO DIP switch 8 is LSB.

Verify the probing has been enabled in `probe.v`. Note that, as one adds test points, routing and timing closure becomes more and more difficult. Only define those test points you need.

The Intronix Logicport is a small USB-based logic analyzer. It has 34 channels, two of which can be used as clock inputs, and does state analysis at up to 200 MHz. The MPEG2 decoder on the ML505 runs at 75 MHz, with a typical dot clock of 27 MHz, well within the capabilities of the Logicport logic analyzer. Probing the memory controller at 200 MHz, however, is borderline. To be on the safe side, when probing the memory controller with the Logicport, lower memory clock to 125 MHz .

A small two-layer adapter board has been designed to connect the Intronix Logicport to the Xilinx ML505. Board layout can be downloaded from `http://www.kdvelectronics.eu/probe_adapter/probe_adapter.html`.

The `tools/logicport` directory contains Logicport configuration files for the test points defined in `probe.v`. Note configuration files can be read and waveforms displayed by Logicport software even if no analyzer is present.

2.4.2 Finite State Machine Graphs

The MPEG2 decoder uses Finite State Machines throughout; no embedded processors or microcontrollers are used. Verifying the correctness of the Finite State Machines is

Test bitstream	Profile and level	Remarks
tcela/tcela-16-matrices	11172-2	Fail (MPEG1 stream)
tcela/tcela-18-d-pict	11172-2	Fail (MPEG1 stream)
compcore/ccm1	11172-2	Fail (MPEG1 stream)
tcela/tcela-19-wide	11172-2	Fail (MPEG1 stream)
toshiba/toshiba_DPall-0	SP@ML	
nokia/nokia6_dual	SP@ML	
nokia/nokia6_dual60	SP@ML	
nokia/nokia_7	SP@ML	
tcela/tcela-14-bff-dp	SP@ML	
ibm/ibm-bw-v3	SP@ML	
tcela/tcela-8-fp-dp	SP@ML	
tcela/tcela-9-fp-dp	SP@ML	1 bit off
mei/MEI.stream16v2	SP@ML	Fail (MPEG1 stream)
mei/MEI.stream16.long	SP@ML	Fail (MPEG1 stream)
ntr/ntr_skipped_v3	SP@ML	
teracom/teracom_vlc4	SP@ML	
tcela/tcela-15-stuffing	SP@ML	
tcela/tcela-17-dots	SP@ML	
gi/gi4	MP@ML	
gi/gi6	MP@ML	
gi/gi_from_tape	MP@ML	
gi/gi7	MP@ML	
gi/gi_9	MP@ML	
ti/TI_cl_2	MP@ML	
tceh/tceh_conf2	MP@ML	
mei/mei.2conftest.4f	MP@ML	
mei/mei.2conftest.60f.new	MP@ML	
tek/Tek-5.2	MP@ML	
tek/Tek-5-long	MP@ML	
tcela/tcela-6-slices	MP@ML	
tcela/tcela-7-slices	MP@ML	
sony/sony-ct1	MP@ML	
sony/sony-ct2	MP@ML	
sony/sony-ct3	MP@ML	
sony/sony-ct4	MP@ML	
att/att_mismatch	MP@ML	
teracom/teracom_vlc4	MP@ML	
ccett/mcp10ccett	MP@ML	
lep/bits_conf_lep_11	MP@ML	
hhi/hhi_burst_short	MP@ML	
hhi/hhi_burst_long	MP@ML	
tcela/tcela-10-killer	MP@ML	

Table 2.3: Conformance Test Suite

important. Finite state machine transition graphs are created from Verilog source files as a means of visually inspecting and verifying source correctness. The `mkfsmgraph` Perl script in `tools/fsmgraph` assumes the comment `/* next state logic */` marks the beginning of a `case` statement in an `always` block, used to select the next state, and that all states begin with STATE_ :

```
/* next state logic */
always @*
  case (state)
    STATE_INIT: if (first_pixel_read) next = STATE_WAIT;
                else next = STATE_INIT;
    ...
    default next = STATE_INIT
  endcase
/* state */
always @(posedge clk)
  if(~rst) state <= STATE_INIT;
  else state <= next;
```

The `mkfsmgraph` tool parses the Verilog source files using the following algorithm:

- read the Verilog file until the comment `/* next state logic */` is found
- take the first `always` block after the `/* next state logic */` comment
- any word beginning with `STATE_` is assumed to represent a FSM state.
- if the character following the FSM state is a colon (:) the state is a graph node.
- if the character following the FSM state is a semicolon (;) the state is the end point of a state transition.
- if the character following the FSM state is neither a colon (:) nor a semicolon (;) the state is not added to the graph.

The resulting graph is written to standard output in gml format. Graph layout software uDrawGraph from the University of Bremen, Germany, is then used to produce a visually appealing graph.

No attempt has been made to write a script capable of parsing arbitrary Verilog sources. The Verilog sources have been written so the script can parse them.

The graph of the variable length-decoding FSM `vld.v` has been simplified further by removing all transitions to STATE_NEXT_START_CODE and STATE_ERROR. Nodes which transition to STATE_NEXT_START_CODE are drawn with double border. Removing transitions to STATE_NEXT_START_CODE and STATE_ERROR produces a graph with much less visual clutter. A large format version of the FMS graph of `vld.v` can be found in `doc/vld-poster.pdf`. It is suggested to become familiar with the graph before significantly modifying `vld.v`.

2.4.3 IEEE-1180 IDCT Accuracy Test

`idct.v` has been tested to comply with the former IEEE-1180, the actual ISO/IEC 23002-1 [2]. The testbench can be found in the `tools/ieee1180` directory. Test results can be found in the file `ieee-1180-results`. Test results indicate the idct implementation is IEEE-1180 compliant.

2.4.4 Reference software decoder

The directory `tools/mpeg2dec` contains the MPEG2 reference decoder, modified to provide extensive logging and to regularly write the framebuffers to file. A sample run could be:

```
koen@macpro ~/xilinx/mpeg2/tools $ mkdir run
koen@macpro ~/xilinx/mpeg2/tools $ cd run
koen@macpro ~/xilinx/mpeg2/tools/run $ ../mpeg2dec/mpeg2decode
 -r -v9 -t -o0 'dump_%d_out_%c' -b ../streams/tcela-17.mpg > log
saving dump_0_out_f.y.ppm
saving dump_0_out_f.u.ppm
saving dump_0_out_f.v.ppm
saving dump_0_forward_ref_frm.y.ppm
saving dump_0_forward_ref_frm.u.ppm
saving dump_0_forward_ref_frm.v.ppm
saving dump_0_backward_ref_frm.y.ppm
saving dump_0_backward_ref_frm.u.ppm
saving dump_0_backward_ref_frm.v.ppm
saving dump_0_auxframe.y.ppm
saving dump_0_auxframe.u.ppm
saving dump_0_auxframe.v.ppm
saving dump_1_out_f.y.ppm
saving dump_1_out_f.u.ppm
...
```

The `log` file contains detailed information about the execution of the MPEG2 decoding algorithm, while the `.ppm` files contain framestore dumps, using separate graphics files for each y, u and v component.

2.4.5 MPEG2 Test Streams

The `tools/streams` directory contains some sample MPEG2 program streams, useful during testing. The `retrieve` script in the `tools/streams` directory can be used to download the ISO/IEC 13818-4 conformance test bitstreams from the ISO web site[1].

[1] ISO/IEC 13818-4 test bitstreams, `http://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_13818-4_2004_Confo`

Bibliography

[1] ITU-T Recommendation H.262 “Information technology - Generic coding of moving pictures and associated audio information: Video”, 2000. Also published as ISO/IEC International Standard 13818-2.

[2] ISO/IEC International Standard 23002-1 “Information technology - MPEG video technologies - Part 1: Accuracy requirements for implementation of integer-output 8x8 inverse discrete cosine transform”, 2006.

[3] “Architecture and Bus-Arbitration Schemes for MPEG-2 Video Decoder", Jui-Hua Li and Nam Ling, IEEE Transactions on Circuits and Systems for Video Technology, Vol. 9, No. 5, August 1999, p.727-736.

[4] “Systematic approach of Fixed Point 8x8 IDCT and DCT Design and Implementation", Ci-Xun Zhang , Jing Wang , Lu Yu, Institute of Information and Communication Engineering, Zhejiang University, Hangzhou, China, 310027.

[5] “Virtex-5 FPGA User Guide”, Xilinx UG190 (v3.2), December 11, 2007.

[6] “ML505/506 MIG Design Creation Using ISE 9.2i SP3, MIG 2.0 and ChipScope Pro 9.2i”, Xilinx, December 2007.