CDL Modules

Generated by Doxygen 1.8.11

Contents

Chapter 1

CDL modules

This repository contains a number of open source CDL modules (either Apache or BSD licensed) that may be freely reused in any project.

Purpose

The purpose of this repository is to provide freely available modules for both teaching and implementation; the repository includes regression suites for most of the designs, which permits some confidence in the solidity of the design.

A second purpose of the repository is to provide for a corpus of open source hardware modules written in CDL, to enable a new version of CDL to be designed; this new version will include use of a hardware IR (intermediate representation) language that permits targeting and optimization for a number of backends (simulation, verification, FPGA, silicon, emulation, and so on).

Module structure

The modules are grouped into separate directories based on function.

- apb

  The APB modules are predominantly simple APB peripherals; timers, GPIO, and other fixed purpose I/O. There is also the *apb_processor* which is a module that can replay APB transactions (reads and writes) from a ROM, and execute simple ALU operations and loop depending on values in an accumulator; this module permits somewhat flexible APB transactions to be performed in hardware without a microcontroller (such as to initialize a PLL or DDR controller, where APB transactions may depend on lock, skew, etc).

- boards

  The boards directory contains subdirectories for various FPGA or other boards, into which various designs have been built. At present this is mainly the Terasic Cyclone V DE1 board, with the Cambridge University Computer Laboratory I/O daughterboard.

- cpu

  The cpu directory contains a 6502 implementation and numerous RISC-V implementations.

  The RISC-V implementations are based around a variety of execution pipelines of varying lengths, with replaceable instruction decoders. This permits a breadth of embedded RISC-V implementations to be created, supporting a range of RV32 ISA sets (such as RV32E, RV32IMAC, etc). The pipelines are customizable with user instruction encodings as well.

- csrs

  The CSR modules (control/status register) map the APB bus to a pipelined request/response bus to fit in to an FPGA (or silicon) without having to use multicycle paths or slow timing in any way. The modules include one that maps APB to the CSR interface; a CSR interface back to APB; and a CSR interface to simple single-cycle read/write accesses.

- input_devices

  At present the input_devices modules consist solely of PS2-keyboard related modules: a PS2 host (to which a keyboard may be connected); and a PS2 keyboard decoder (which maps PS2 keyboard codes to key up/down events for an 8-bit keycode).

- led

  This directory contains useful modules for driving LEDs: LED 7-segment display, from hex digits; a Neopixel (WS2812) driver for any length of Neopixel chain.

- microcomputers

  At present the 'BBC microcomputer' is the only microcomputer design; this is an implementation of the BBC microcomputer model B, with video, ROMs, and floppy disk controller, that runs at 50MHz on a the Cyclone V.

  A 'Picoriscv' implementation is at a conceptual level; it is not clear if this is worth moving forward with outside of the RISC-V CPU directory, at present.

- network

  At present, empty

- serial

- storage

  The storage modules currently consist of only a model of the Intel 8271 floppy disk controller; this was the initial floppy disk controller for the BBC microcomputer. Instead of supporting a physical disk drive, the FD↩C8271 module uses an SRAM to supply track format and sector information, as well as disk data.

- utils

  The utils directory contains some generic utility modules. Firstly there is a generic module to support multiplexing two request/acknowledge buses on to a single request/acknowledge bus. This has to be compiled using CDL options to specify the actual request/acknowledge type bus required for a module - this is the CDL 1.4 method for parametrizing types. This module is used in two *dprintf* multiplexer modules also in the utils directory.

  The dprintf module is an sprintf-like module: it is used for debug printing. The module is supplied with a request - a format string to display with arguments, and an address - and it generates a stream of output bytes (with addresses) that are the output of the formatting. The module supports 7-bit characters (suitable for teletext), and hexadecimal and decimal formatted numbers (with field size).

  The utils also includes a hysteresis_switch module. This module takes an input and generates a filtered output using hysteresis; it may be used, for example, to debounce an input switch.

- video

  The video modules relate to video output, currently. There are two framebuffer modules - one uses a bitmap framebuffer, and the other a teletext framebuffer (where the SRAM contains characters and control codes). These use a framebuffer_timing module, that generates the timing for a video output (vsync, hsync, display area enable, and so on).

  To provide the teletext decoding there is a teletext decoder module; this is used also by the Mullard SAA5050 implementation, used by the BBC microcomputer.

  There is also a Motorola 6845 CRTC controller module, which was a character video timing generator, that was used for all the display modes in the BBC microcomputer.

Documentation progress

Currently documented: apb, csrs, input_devices, utils, led

To do: storage, microcomputers, cpus, serial

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Module Documentation

4.1 apb_master_axi

Files

- file apb_master_axi.cdl

  AXI target to an APB master interface.

4.1.1 Detailed Description

4.1.2 Modules

4.1.2.1 module apb_master_axi::apb_master_axi (clock *aclk,* input bit *areset_n,* input **t_axi_request** *ar,* output bit *awready,* input **t_axi_request** *aw,* output bit *arready,* output bit *wready,* input **t_axi_write_data** *w,* input bit *bready,* output **t_axi_write_response** *b,* input bit *rready,* output **t_axi_read_response** *r,* output **t_apb_request** *apb_request,* input **t_apb_response** *apb_response*)

Parameters

in	*apb_response*	AXI target mapping to an APB master

This is a very simple AXI target that handles a single AXI transaction at any one time; it converts this into an APB read or write transaction, whose completion allows the completion of the AXI transaction.

This module is currently very primitive, and does not checking of transaction input really.

4.2 apb_master_mux

Files

- file apb_target_gpio.cdl

  Simple GPIO target for an APB bus.

4.2.1 Detailed Description

4.2.2 Modules

4.2.2.1 module apb_master_mux::apb_master_mux (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request_0,* output t_apb_response *apb_response_0,* input t_apb_request *apb_request_1,* output t_apb_response *apb_response_1,* output t_apb_request *apb_request,* input t_apb_response *apb_response*)

APB multiplexer - or rather, arbiter and multiplexer

The module takes two APB requests in, and provides a single APB request out.

An APB request from each master is registered, and APB transactions are performed using simple round-robin arbitration.

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*apb_request_0*	APB request to master 0
out	*apb_response_0*	APB response to master 0
in	*apb_request_1*	APB request to master 1
out	*apb_response_1*	APB response to master 1
out	*apb_request*	APB request to targets
in	*apb_response*	APB response from targets

4.3 apb_processor

Files

- file apb_processor.cdl

  Pipelined APB request/response master, driven by a ROM.

4.3.1 Detailed Description

4.3.2 Modules

4.3.2.1 module apb_processor::apb_processor (clock *clk,* input bit *reset_n,* input t_apb_processor_request *apb_processor_request,* output t_apb_processor_response *apb_processor_response,* output t_apb_request *apb_request,* input t_apb_response *apb_response,* output t_apb_rom_request *rom_request,* input bit *rom_data[40]*)

The module is presented with a request to execute a program from the ROM starting at a certain address. It executes the program, and hence a set of APB requests, as required.

The purpose of the module is to permit programmed sequences of APB transactions without a full-fledge microcontroller being needed, even for PLL setup or DDR pin DLL scanning, and so on.

A request to run a program is an *address* with a *valid* bit; if a valid request is presented, it should be held until acknowledge. The module will acknowledge a request using a single cycle *acknowledge* in its `apb_processor_response`. Then the module will start reading the ROM at the given address, executing 'APB program instructions' from the data returned. The ROM is external to this module, and hence the `rom_request` and `rom_data` signals permit a simple synchronous memory to be attached with the program data.

A second request to run a program may be presented while the APB processor module is busy with the previous program request; this is perfectly acceptable, but there will not be an *acknowledge* until the APB processor is ready to start the new program; the new request should be held stable until that point.

The ROM contains the APB program, which is 40 bits of data per instruction - 8 bits of opcode, 32 bits of data, per word. The opcode is in [8;32], and the operand data is in [32;0].

The opcodes fall in to 6 different classes: ALU, branch, set parameter, APB request, wait, finish.

- ALU

  Five ALU ops are supported - OR, BIC, AND, XOR, ADD

- Branch

  Four branch types are supported - always, if acc is zero, if acc is nonzero, and if repeat count is nonzero (with the side effect of decrementing the repeat count)

- Set parameter

  Set parameter can set the APB address, accumulator and repeat count

- Wait

  Wait uses the accumulator, decrementing it once per cycle, to wait before moving on in the program.

- APB request

  APB request can request read, write accumulator, or write using the ROM content as data; these can optionally also auto-increment the address

- Finish

  Complete the program, and permit a new request to be started

The module presents a registered APB request interface out, and accepts an APB response back, including *pready*.

Parameters

in	*clk*	Clock for the CSR interface; a superset of all targets clock
in	*reset_n*	Active low reset
in	*apb_processor_request*	Request from the client to execute from an address in the ROM
out	*apb_processor_response*	Response to the client acknowledging a request
out	*apb_request*	Pipelined csr request interface output
in	*apb_response*	Pipelined csr request interface response
out	*rom_request*	Request to the instruction ROM for reading, with address
in	*rom_data*	Read data back from the ROM with the APB program instruction

4.4 apb_target_de1_cl_inputs

Files

- file apb_target_timer.cdl

  Simple timer target for an APB bus.

4.4.1 Detailed Description

4.4.2 Modules

4.4.2.1 module apb_target_de1_cl_inputs::apb_target_de1_cl_inputs (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* input t_de1_cl_user_inputs *user_inputs*)

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*apb_request*	APB request
out	*apb_response*	APB response
in	*user_inputs*	This module provides an APB target to get the status of inputs for the Cambridge University DE1-SOC daughterboard.

The CL DE1-SOC daughterboard contains a joystick, a diamond of four buttons, two rotary dials, and apparently a temperature alarm and touchpanel intrerrupt (the latter two I have not used as yet).

Two registers are provided. The first is the state register,

Bits	Meaning
31	Inputs changed since last read of state
5;26	zero
25	temperature alarm
24	touchpanel interrupt
6;18	zero
17	right rotary dial is pressed in
16	left rotary dial is pressed in
3;13	zero
12	joystick is being pressed
11	joystick is being pushed right
10	joystick is being pushed left
9	joystick is being pushed down
8	joystick is being pushed up
4;4	zero
3	diamond y (top black button) is being pressed
2	diamond x (left blue button) is being pressed
1	diamond b (right red button) is being pressed
0	diamond a (bottom green button) is being pressed

The second register relates just to the rotary dials; the hardware keeps track of directional impulses to provide an 8-bit 'rotary value' for each dial.

Bits	Meaning
31	Inputs changed since last read of state
13;18	zero
17	right rotary dial is pressed in
16	left rotary dial is pressed in
8;8	right rotary dial position (decremented on anticlockwise, incremented on clockwise)
8;0	left rotary dial position (decremented on anticlockwise, incremented on clockwise)

4.5 apb_target_dprintf

Files

- file apb_target_dprintf.cdl

  Simple target for an APB bus to drive the dprintf request.

4.5.1 Detailed Description

4.5.2 Modules

4.5.2.1 module apb_target_dprintf::apb_target_dprintf (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* output t_dprintf_req_4 *dprintf_req,* input bit *dprintf_ack*)

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*apb_request*	APB request
out	*apb_response*	APB response
in	*dprintf_ack*	Simple Dprintf requester with an APB interface.

A dprintf request is an address and, in this case (a *t_dprintf_req_4*) four 64-bit data words. This is mapped to eight 32-bit data words, with data register 0 mapping to the most significant word of the *dprintf_req* data 0 (so that data register 0 corresponds to the first text displayed as part of the dprintf).

The module provides an address register, which is the address presented in the dprintf request. Usually for a dprintf to a teletext framebuffer, for example, this is the address of the first character of the output within the framebuffer.

The normal operation is to write a number of data registers, starting with register 0, and then to write to the address register *with commit* to invoke the dprintf.

Another method could be to have the address and bulk of the data set up, and then a single write to a *data with commit* to, for example, fill out a 32-bit hex value for display, invoking the dprintf (for example if a dprintf were set up to display 'latest pc %08x', the pc value can be written to the correct data register with commit).

The address register can be read back, in which case it has some status also:

Bits	Meaning
31	dprintf_req valid (i.e. has not been completed by dprintf slave)
15;16	zero
16;0	address for dprintf request.

The top bit of this register is set by a commit and cleared when the dprintf slave acknowledges the dprintf request.

For more details on dprintf requests themselves, see the documentation in utils/src/dprintf

4.6 apb_target_gpio

Files

- file apb_target_gpio.cdl

  Simple GPIO target for an APB bus.

4.6.1 Detailed Description

4.6.2 Modules

4.6.2.1 module apb_target_gpio::apb_target_gpio (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* output bit *gpio_output[16],* output bit *gpio_output_enable[16],* input bit *gpio_input[16],* output bit *gpio_input_event*)

Simple APB interface to a GPIO system.

The module has 16 outputs, each with separate enables which reset to off; it also has 16 inputs, each of which is synced and then edge detected (or other configured event).

This module has four APB-addressable registers:

- OutputControl (0):

  Each output (16 of them) has 2 bits; bit 0 is used for GPIO output 0 *value*, and bit 1 is used for its *enable*. Bits 2 and 3 are used for GPIO output 1, and so on.

- InputStatus (1):

  This register contains the input pin values and the event status for the 16 GPIO inputs. The bottom 16 bits contain the input pin *value* for each of the 16 inputs; the top 16 bits contain the *event* status.

- InputReg0 (2):

  On reads, this register contains the input pin event types for input pins - see the *t_gpio_input_type* for the decode; bits [3;0] are used for GPIO 0, [3;4] for GPIO1, and so on up to [3;28] for GPIO7. On writes, this register writes a *single* GPIO input control: bits[4;0] contain the GPIO input to control; and bits [3;12] contain the event type - which is only written if bit[8] is set; and if bit[9] then the input event is cleared.

- InputReg1 (3):

  This register contains the input pin event types for the top 8 GPIO pins, in a manner identical to InputReg0.

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*apb_request*	APB request
out	*apb_response*	APB response
out	*gpio_output*	GPIO output values, if `gpio_output_enable` is set
out	*gpio_output_enable*	GPIO output enables
in	*gpio_input*	GPIO input pin connections
out	*gpio_input_event*	Driven high when at least one GPIO input event has occurred, for use as an interrupt to a CPU

4.7 apb_target_led_ws2812

Files

- file apb_target_led_ws2812.cdl

  Simple APB target for driving a chain of Neopixels.

4.7.1 Detailed Description

4.7.2 Modules

4.7.2.1 module apb_target_led_ws2812::apb_target_led_ws2812 (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* input bit *divider_400ns_in[8],* output bit *led_chain*)

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*apb_request*	APB request
out	*apb_response*	APB response
in	*divider_400ns_in*	Default value for divider_400ns
out	*led_chain*	A simple module to drive a chain of Neopixel LEDs as an APB target.

The Neopixel LEDs are 24-bit RGB LEDs that are driven through a chain, with a single control pin providing the information for the LEDs.

This module supports up to 16 LEDs in a chain, each with a 24-bit RGB value. The chain must be configured with a clock divider (which is the number of clock ticks required to generate a 400ns, approximately, clock) and the number of LEDs in the chain.

The clock divider resets to 0, but whenever it is zero it resets to the input value *divider_400ns_in*; hence it may be effectively 'hardwired'.

Once configured the LED colors can be individually written to address-mapped registers. The LED chain automatically updates.

The configuration register is:

Bits	Meaning
12;2	zero
4;16	last LED in the chain (0 for one LED, 15 for sixteen LEDs)
8;8	zero
8;0	clock divider to create a 400ns clock enable from system clock

Example divider values

System clock	Period	Divider
50MHz	20ns	19
100MHz	10ns	39
20MHz	50ns	7

4.8 apb_target_ps2_host

Files

- file apb_target_timer.cdl

  Simple timer target for an APB bus.

4.8.1 Detailed Description

4.8.2 Modules

4.8.2.1 module apb_target_ps2_host::apb_target_ps2_host (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* input t_ps2_pins *ps2_in,* output t_ps2_pins *ps2_out*)

This module provides an APB target to read data from a PS2 host interface, particularly for a PS2 keyboard or mouse.

Before use, the clock divider must be set up with a write to the state.

The state register is

Bits	Meaning
16;16	ps2_state.divider_3us (number of ticks to get approx 3us)
15	0
3;12	fifo_wptr
11	0
3;8	fifo_rptr
2;6	0
5	fifo full
4	fifo empty
3	fifo overflow occurred since last read
2	PS2 timeout occurred since last read
1	PS2 protocol error occurred since last read
0	PS2 parity error occurred since last read

The fifo register is:

Bits	Meaning
31	Fifo empty (data invalid)
23; 8	0
8; 0	PS2 Rx data

The usage model is to poll bit 4 of the state register (the fifo empty bit); when this is cleared the fifo register will have at least one received byte; this may then be read.

An alternative is to poll the fifo register itself, and check if the top bit is clear; if so, the bottom eight bits contain valid data. This will set the FIFO underflow bit, on every read when there is no valid receive data.

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*apb_request*	APB request
out	*apb_response*	APB response
in	*ps2_in*	Pin values from the outside
out	*ps2_out*	Pin values to drive - 1 means float high, 0 means pull low

4.9 apb_target_rv_timer

Files

- file apb_target_rv_timer.cdl

  RISC-V compatible timer target for an APB bus.

4.9.1 Detailed Description

4.9.2 Modules

4.9.2.1 module apb_target_rv_timer::apb_target_rv_timer (clock *clk,* input bit *reset_n,* input t_timer_control *timer_control,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* output t_timer_value *timer_value*)

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*timer_control*	Control of the timer
in	*apb_request*	APB request
out	*apb_response*	APB response
out	*timer_value*	RISC-V compatible timer with an APB interface.

This is a monotonically increasing 64-bit timer with a 64-bit comparator.

The timer has a fractional component to permit, for example, a 'nanosecond' timer that is clocked at, say, 600↩ MHz; in this case the timer is ticked every 1.666ns, and so an addition in each cycle of 0xa to a 4-bit fractional component and a 1 integer component. The timer_control has, therfore, a fixed-point adder value with a 4-digit fractional component.

However, this would actually lead to a timer that would be only 99.61% accurate.

Hence a further subfraction capability is supported; this permits a further 1/16th of a nanosecond (or whatever the timer unit is) to be added for M out of every N cycles.

In the case of 600MHz a bonus 1/16th should be added for 2 out of every 3 cycles. This is set using a bonus↩ _subfraction_numer of 0 and a bonus_subfraction_denom of 2 (meaning for 2 out of every 3 cycles add a further 1/16th).

Hence every three cycles the timer will have 0x1.b, 0x1.b, 0x1.a added to it - hence the timer will have gone up by an integer value of 5ns, which is correct for 3 600MHz clock cycles.

If the control values are tied off to zero then the extra fractional logic will be optimized out.

Some example values for 1ns timer values:

Clock	Period	Adder (Int/fraction)	Subfraction Numer/Denom
1GHz	1ns	1 / 0x0	0 / 0
800MHz	1.25ns	1 / 0x4	0 / 0
600MHz	1.66ns	1 / 0xa	0 / 2

The period should be:

subfraction numer/denum=0/x: (Int + fraction/16)

subfraction numer/denum=M/N: (Int + (fraction+(N-M-1)/N)/16)

4.10 apb_target_sram_interface

Files

- file apb_target_sram_interface.cdl

  APB bus target to drive an SRAM read/write request.

4.10.1 Detailed Description

4.10.2 Modules

4.10.2.1 module apb_target_sram_interface::apb_target_sram_interface (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* output bit *sram_ctrl[32],* output t_sram_access_req *sram_access_req,* input t_sram_access_resp *sram_access_resp*)

APB target peripheral that generates SRAM read/write requests

The module maintains a 32-bit SRAM address that is used in the requests, which is a read/write register. There is also a 32-bit control register, that can be used for any purpose by the client.

SRAM requests occur when the data register is accessed; it can be accessed in one of three different ways. Firstly, it may be accessed simply read/write, with either generating the appropriate SRAM request to the address given by the SRAM address register. Secondly, it may be accessed with a post-increment, where the SRAM address register value is used as-is in the request, but it is incremented ready for a subsequent transaction. Thirdly, it may be accessed 'windowed'; in this manner the bottom 7 bits of the APB address are used in conjunction with the top 25 bits of the SRAM address register to generate the address for the SRAM request.

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*apb_request*	APB request
out	*apb_response*	APB response
out	*sram_ctrl*	SRAM control data, for whatever purpose
out	*sram_access_req*	SRAM access request
in	*sram_access_resp*	SRAM access response

4.11 apb_target_timer

Files

- file apb_target_timer.cdl

  Simple timer target for an APB bus.

4.11.1 Detailed Description

4.11.2 Modules

4.11.2.1 module apb_target_timer::apb_target_timer (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* output bit *timer_equalled[3]*)

Simple timer with an APB interface. This is a monotonically increasing 31-bit timer with three 31-bit comparators.

The timers are read/written through the APB interface with the timer value read-only at address 0.

The three comparators are at addresses 4, 5 and 6. When a comparator is written it writes the 31-bit *comparator* value and it clears the *timer's equalled* bit. When a comparator is read it returns the *comparator* value (in bits [31;0]), and it returns the *equalled* status in bit [31] - while atomically clearing it.

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*apb_request*	APB request
out	*apb_response*	APB response
out	*timer_equalled*	One output bit per timer, mirroring the three timer's *equalled* state

4.12 bbc_micro_de1_cl

Files

- file bbc_micro_de1_cl.cdl

  BBC microcomputer with RAMs for the CL DE1 + daughterboard.

4.12.1 Detailed Description

4.12.2 Modules

4.12.2.1 module bbc_micro_de1_cl::bbc_micro_de1_cl (clock *clk,* clock *video_clk,* input bit *reset_n,* input bit *video_locked,* output **t_de1_cl_lcd** *lcd,* input bit *keys[4],* input bit *switches[10],* output bit *leds[10],* input **t_ps2_pins** *ps2_in,* output **t_ps2_pins** *ps2_out,* output bit *led_data_pin,* input **t_de1_cl_inputs_status** *inputs_status,* output **t_de1_cl_inputs_control** *inputs_control*)

Parameters

in	*clk*	50MHz clock from DE1 clock generator
in	*video_clk*	9MHz clock from PLL, derived from 50MHz
in	*reset_n*	hard reset from a pin - a key on DE1
in	*video_locked*	High if video PLL has locked
out	*lcd*	LCD display out to computer lab daughterboard
in	*keys*	DE1 keys
in	*switches*	DE1 switches
out	*leds*	DE1 leds
in	*ps2_in*	PS2 input pins
out	*ps2_out*	PS2 output pin driver open collector
out	*led_data_pin*	DE1 CL daughterboard neopixel LED pin
in	*inputs_status*	DE1 CL daughterboard shifter register etc status
out	*inputs_control*	DE1 CL daughterboard shifter register control

4.13 bbc_micro_de1_cl_bbc

Files

- file bbc_micro_de1_cl_bbc.cdl

  BBC microcomputer with RAMs for the CL DE1 + daughterboard.

4.13.1 Detailed Description

4.13.2 Modules

4.13.2.1 module bbc_micro_de1_cl_bbc::bbc_micro_de1_cl_bbc (clock *clk,* clock *video_clk,* input bit *reset_n,* input bit *bbc_reset_n,* input bit *framebuffer_reset_n,* output t_bbc_clock_control *clock_control,* input t_bbc_keyboard *bbc_keyboard,* output t_video_bus *video_bus,* input t_csr_request *csr_request,* output t_csr_response *csr_response*)

Clock that mirrors 2MHz falling - video data from RAM is valid at this edge, so used by CRTC, SAA5050 latches, SAA5050, vidproc

6502 clock, >=2MHz but extended when accessing 1MHz peripherals

Parameters

in	*clk*	50MHz clock from DE1 clock generator
in	*video_clk*	9MHz clock from PLL, derived from 50MHz
in	*reset_n*	hard reset from a pin - a key on DE1

4.14 bbc_micro_de1_cl_io

Files

- file bbc_micro_de1_cl.cdl

  BBC microcomputer with RAMs for the CL DE1 + daughterboard.

4.14.1 Detailed Description

4.14.2 Modules

4.14.2.1 module bbc_micro_de1_cl_io::bbc_micro_de1_cl_io (clock *clk*, clock *video_clk*, input bit *reset_n*, input bit *bbc_reset_n*, input bit *framebuffer_reset_n*, input bit *keys[4]*, input bit *switches[10]*, input t_bbc_clock_control *clock_control*, output t_bbc_keyboard *bbc_keyboard*, output t_video_bus *video_bus*, output t_csr_request *csr_request*, input t_csr_response *csr_response*, input t_ps2_pins *ps2_in*, output t_ps2_pins *ps2_out*, input t_de1_cl_inputs_status *inputs_status*, output t_de1_cl_inputs_control *inputs_control*, output bit *lcd_source*, output bit *leds[10]*, output bit *led_chain*)

Parameters

in	*clk*	50MHz clock from DE1 clock generator
in	*video_clk*	9MHz clock from PLL, derived from 50MHz
in	*reset_n*	hard reset from a pin - a key on DE1
in	*ps2_in*	PS2 input pins
out	*ps2_out*	PS2 output pin driver open collector
in	*inputs_status*	DE1 CL daughterboard shifter register etc status
out	*inputs_control*	DE1 CL daughterboard shifter register control

4.15 de1_cl_controls

Files

- file bbc_micro_de1_cl.cdl

  BBC microcomputer with RAMs for the CL DE1 + daughterboard.

4.15.1 Detailed Description

4.15.2 Modules

4.15.2.1 module de1_cl_controls::de1_cl_controls (clock *clk,* input bit *reset_n,* output t_de1_cl_inputs_control *inputs_control,* input t_de1_cl_inputs_status *inputs_status,* output t_de1_cl_user_inputs *user_inputs,* input bit *sr_divider[8]*)

This module manages the buttons and other controls on the Cambridge University Computer Laboratory DE1 daughterboard.

A number of input switches are handled through a shift register, which is clocked using the input clock 'clk' divided down by the divider. This is handled by

The rotary encoder switch '318-ENC130175F-12PS' available from Mouser has the following operation:

Clockwise: B disconnects from C when A is disconnected from C

Counter-clockwise: B connects to C when A is disconnected from C

The CL daughterboard for the DE1 has a debounce RC network on the A and B pins, with a RC (probably) of 47us (high), so presumably the encoder is not optical :-).

Parameters

in	*clk*	system clock - not the shift register pin, something faster
in	*reset_n*	async reset
out	*inputs_control*	Signals to the shift register etc on the DE1 CL daughterboard
in	*inputs_status*	Signals from the shift register, rotary encoders, etc on the DE1 CL daughterboard
out	*user_inputs*	
in	*sr_divider*	clock divider to control speed of shift register

4.16 picoriscv_de1_cl

Files

- file picoriscv_de1_cl.cdl

  Pico-RISC-V microcomputer for the CL DE1 + daughterboard.

4.16.1 Detailed Description

4.16.2 Modules

4.16.2.1 module picoriscv_de1_cl::picoriscv_de1_cl (clock *clk,* clock *video_clk,* input bit *reset_n,* input bit *video_locked,* output **t_de1_cl_lcd** *lcd,* input bit *keys[4],* input bit *switches[10],* output bit *leds[10],* input **t_ps2_pins** *ps2_in,* output **t_ps2_pins** *ps2_out,* output bit *led_data_pin,* input **t_de1_cl_inputs_status** *inputs_status,* output **t_de1_cl_inputs_control** *inputs_control*)

Parameters

in	*clk*	50MHz clock from DE1 clock generator
in	*video_clk*	9MHz clock from PLL, derived from 50MHz
in	*reset_n*	hard reset from a pin - a key on DE1
in	*video_locked*	High if video PLL has locked
out	*lcd*	LCD display out to computer lab daughterboard
in	*keys*	DE1 keys
in	*switches*	DE1 switches
out	*leds*	DE1 leds
in	*ps2_in*	PS2 input pins
out	*ps2_out*	PS2 output pin driver open collector
out	*led_data_pin*	DE1 CL daughterboard neopixel LED pin
in	*inputs_status*	DE1 CL daughterboard shifter register etc status
out	*inputs_control*	DE1 CL daughterboard shifter register control

4.17 riscv_adjunct_de1_cl

Files

- file riscv_adjunct_de1_cl.cdl

  RISC-V adjunct for ARM AXI for the CL DE1 + daughterboard.

4.17.1 Detailed Description

4.17.2 Modules

4.17.2.1 module riscv_adjunct_de1_cl::riscv_adjunct_de1_cl (clock *clk,* clock *video_clk,* input bit *reset_n,* input bit *video_locked,* output t_de1_cl_lcd *lcd,* input bit *keys[4],* input bit *switches[10],* output bit *leds[10],* input t_ps2_pins *ps2_in,* output t_ps2_pins *ps2_out,* output bit *led_data_pin,* input t_de1_cl_inputs_status *inputs_status,* output t_de1_cl_inputs_control *inputs_control*)

Parameters

in	*clk*	50MHz clock from DE1 clock generator
in	*video_clk*	9MHz clock from PLL, derived from 50MHz
in	*reset_n*	hard reset from a pin - a key on DE1
in	*video_locked*	High if video PLL has locked
out	*lcd*	LCD display out to computer lab daughterboard
in	*keys*	DE1 keys
in	*switches*	DE1 switches
out	*leds*	DE1 leds
in	*ps2_in*	PS2 input pins
out	*ps2_out*	PS2 output pin driver open collector
out	*led_data_pin*	DE1 CL daughterboard neopixel LED pin
in	*inputs_status*	DE1 CL daughterboard shifter register etc status
out	*inputs_control*	DE1 CL daughterboard shifter register control

4.18 riscv_csrs_minimal

Files

- file riscv_csrs_minimal.cdl

  Control/status registers for a minimal RISC-V implementation.

4.18.1 Detailed Description

4.18.2 Modules

4.18.2.1 module riscv_csrs_minimal::riscv_csrs_minimal (clock *clk,* input bit *reset_n,* input t_riscv_irqs *irqs,* input t_riscv_csr_access *csr_access,* input t_riscv_word *csr_write_data,* output t_riscv_csr_data *csr_data,* input t_riscv_csr_controls *csr_controls,* output t_riscv_csrs_minimal *csrs*)

This module implements a minimal set of RISC-V CSRs, as per v2.1 (May 2016) of the RISC-V instruction set manual user level ISA and v1.9.1 of the privilege architecture (Nov 2016), with the exception that MTIME has been removed (as this seems to be the correct thing to do).

The privilege specifcation (v1.10) indicates:

meip is read-only and is derived from the external irq in to this module

mtip is read-only, cleared by writing to the memory-mapped timer comparator

msip is read-write in a memory-mapped register somewhere

Hence the irqs structure must provide these three signals

Minimal CSRs as only machine mode and debug mode are supported. In debug mode every register access is supported. In machine mode then every register EXCEPT access to ??? is supported.

Given machine mode is the only mode supported:

there are no SEI and UEI interrupt pins in to this module

SEIP and UEIP are not supported

STIP and UTIP are not supported

SSIP and USIP are not supported

mstatus.SIE and mstatus.UIE (and previous versions) are hardwired to 0

mstatus.SPP and mstatus.UPP are hardwired to 0

The mip (machine interrupt pending register) therefore is:

{20b0, MEIP, 3b0, MTIP, 3b0, MSIP, 3b0}

The mie (machine interrupt enable register) is:

{20b0, MEIE, 3b0, MTIE, 3b0, MSIE, 3b0}

The instruction to the pipeline to request an interrupt (which is only taken if an instruction is in uncommitted in the execution stage) must be generated using the execution mode and the interrupt enable bits and the interrupt pending bits.

Hence the 'take interrupt' is (mip & mie) != 0 && mstatus.MIE && (current mode >= machine mode) && (current mode != debug mode).

The required priority order is:

external interrupts, software interrupts, timer interrupts

If an instruction has been committed then it may trap, and the trap will occur prior to an interrupt which happens after the commit point. In this case there will be a trap, and the trapped instruction will be fetched, and then an interrupt can be taken.

When an interrupt is taken the following occurs:

MPP <= current execution mode (must be machine mode, as debug mode is not interruptible)

mstatus.MPIE <= mstatus.MIE

Note that WFI should wait independent of mstatus.MIE for (mip & mie) != 0 (given machine mode only) In debug mode WFI should be a NOP.

WFI may always be a NOP. Scratch debug write

write_data

Parameters

in	*clk*	RISC-V clock
in	*reset_n*	Active low reset
in	*irqs*	Interrupts in to the CPU
in	*csr_access*	RISC-V CSR access, combinatorially decoded
in	*csr_write_data*	Write data for the CSR access, later in the cycle than possibly
out	*csr_data*	CSR respone (including read data), from the current *csr_access*
in	*csr_controls*	Control signals to update the CSRs
out	*csrs*	CSR values

4.19 riscv_e32_decode

Files

- file riscv_i32_decode.cdl

  Instruction decoder for RISC-V implementation.

4.19.1 Detailed Description

4.19.2 Modules

4.19.2.1 module riscv_e32_decode::riscv_e32_decode (input t_riscv_i32_inst *instruction,* output t_riscv_i32_decode *idecode,* input t_riscv_config *riscv_config*)

Parameters

in	*riscv_config*	Instruction decoder for RISC-V RV32E instruction set.

This is based on the RISC-V v2.1 specification (hence figure numbers are from that specification)

4.20 riscv_e32c_decode

Files

- file riscv_i32_decode.cdl

  Instruction decoder for RISC-V implementation.

4.20.1 Detailed Description

4.20.2 Modules

4.20.2.1 module riscv_e32c_decode::riscv_e32c_decode (input t_riscv_i32_inst *instruction,* output t_riscv_i32_decode *idecode,* input t_riscv_config *riscv_config*)

Parameters

in	*riscv_config*	Instruction decoder for RISC-V RV32E instruction set.

This is based on the RISC-V v2.1 specification (hence figure numbers are from that specification)

4.21 riscv_i32_alu

Files

- file riscv_i32_alu.cdl

  ALU for i32 RISC-V implementation.

4.21.1 Detailed Description

4.21.2 Modules

4.21.2.1 module riscv_i32_alu::riscv_i32_alu (input t_riscv_i32_decode *idecode,* input t_riscv_word *pc,* input t_riscv_word *rs1,* input t_riscv_word *rs2,* output t_riscv_i32_alu_result *alu_result*)

Parameters

out	*alu_result*	

4.22 riscv_i32_debug

Files

- file riscv_i32_debug.cdl

  RISC-V debug module with APB interface.

4.22.1 Detailed Description

4.22.2 Modules

4.22.2.1 module riscv_i32_debug::riscv_i32_debug (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* output t_riscv_debug_mst *debug_mst,* input t_riscv_debug_tgt *debug_tgt*)

This is a RISC-V debug module designed for the RV32I pipelines in the CDL hardware repo.

It provides the registers defined in the RISC-V Debug specificaiton revision 0.13.

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*apb_request*	APB request
out	*apb_response*	APB response
out	*debug_mst*	Debug master to PDMs
in	*debug_tgt*	Debug target from PDMs

4.23 riscv_i32_decode

Files

- file riscv_i32_decode.cdl

  Instruction decoder for RISC-V implementation.

4.23.1 Detailed Description

4.23.2 Modules

4.23.2.1 module riscv_i32_decode::riscv_i32_decode (input t_riscv_i32_inst *instruction*, output t_riscv_i32_decode *idecode*, input t_riscv_config *riscv_config*)

Parameters

in	*riscv_config*	Instruction decoder for RISC-V I32 instruction set.

This is based on the RISC-V v2.2 specification (hence figure numbers are from that specification)

It provides for the option of RV32E and RV32M.

RV32E indicates an illegal instruction if a register outside 0-15 is accessed.

RV32M provides decode of multiply and divide; if it is not desired then the instructions are decoded as illegal.

4.24 riscv_i32_fetch_debug

Files

- file riscv_i32_fetch_debug.cdl

  Instruction fetch interposer for debug.

4.24.1 Detailed Description

4.24.2 Modules

4.24.2.1 module riscv_i32_fetch_debug::riscv_i32_fetch_debug (input t_riscv_fetch_req *pipeline_ifetch_req*, output t_riscv_fetch_resp *pipeline_ifetch_resp*, input t_riscv_i32_trace *pipeline_trace*, input t_riscv_pipeline_debug_control *debug_control*, output t_riscv_pipeline_debug_control *debug_response*, output t_riscv_fetch_req *ifetch_req*, input t_riscv_fetch_resp *ifetch_resp*)

Parameters

in	*ifetch_resp*	

4.25 riscv_i32_minimal

4.25.1 Detailed Description

4.25.2 Modules

4.25.2.1 module riscv_i32_minimal::riscv_i32_minimal (clock *clk,* input bit *reset_n,* input bit *proc_reset_n,* input t_riscv_irqs *irqs,* output t_riscv_mem_access_req *data_access_req,* input t_riscv_mem_access_resp *data_access_resp,* input t_sram_access_req *sram_access_req,* output t_sram_access_resp *sram_access_resp,* input t_riscv_config *riscv_config,* output t_riscv_i32_trace *trace*)

Parameters

in	*irqs*	Interrupts in to the CPU
out	*trace*	An instantiation of the single stage pipeline RISC-V with RV32I with a single SRAM

Compressed instructions are supported IF i32c_force_disable is 0 and riscv_config.i32c is 1

A single memory is used for instruction and data, at address 0

Any access outside of the bottom 1MB is passed as a request out of this module.

4.26 riscv_i32_minimal_apb

4.26.1 Detailed Description

4.26.2 Modules

4.26.2.1 module riscv_i32_minimal_apb::riscv_i32_minimal_apb (clock *clk,* input bit *reset_n,* input t_riscv_mem_access_req *data_access_req,* output t_riscv_mem_access_resp *data_access_resp,* output t_apb_request *apb_request,* input t_apb_response *apb_response*)

Parameters

in	*apb_response*	

4.27 riscv_i32_muldiv

Files

- file riscv_i32_alu.cdl

  ALU for i32 RISC-V implementation.

4.27.1 Detailed Description

4.27.2 Modules

4.27.2.1 module riscv_i32_muldiv::riscv_i32_muldiv (clock *clk,* input bit *reset_n,* input t_riscv_i32_coproc_controls *coproc_controls,* output t_riscv_i32_coproc_response *coproc_response,* input t_riscv_config *riscv_config*)

Parameters

in	*riscv_config*	

Multiplication:

Consider multiplication of two 3-bit numbers a and b (hence octal)

A straight (unsigned) view of a value X as Xs.Xb is Xb+4∗Xs (Xs is sign bit, Xb remaining bits) A signed view of a value X as Xs.Xb is Xb-4∗Xs Hence one can consider Xsigned = Xunsigned - 8∗Xs

Consider Runsigned = Xunsigned ∗ Yunsigned Then Xsigned ∗ Ysigned = (Xunsigned - 8∗Xs) ∗ (Yunsigned - 8∗↩ Ys) = (Xunsigned∗Yunsigned) + 64∗Xs∗Ys -8∗(Xs∗Yunsigned + Ys∗Xunsigned) (mod 64) = Runsigned -8∗(Xs∗↩ Yunsigned + Ys∗Xunsigned)

Xunsigned ∗ Yunsigned has the following multiplication table:

```
0    1    2    3    4    5    6    7
```

0 0 0 0 0 0 0 0 1 0 1 2 3 4 5 6 7 2 0 2 4 6 10 12 14 16 3 0 3 6 11 14 17 22 25 4 0 4 10 14 20 24 30 34 5 0 5 12 17 24 31 36 43 6 0 6 14 22 30 36 44 52 7 0 7 16 25 34 43 52 61

If both are signed then we have the following correction to add -8∗(Xs∗Yunsigned + Ys∗Xunsigned) (in decimal...)

```
0    1    2    3    4    5    6    7
```

0 0 0 0 0 0 0 0 1 0 0 0 0 -8 -8 -8 -8 2 0 0 0 0 -16 -16 -16 -16 3 0 0 0 0 -24 -24 -24 -24 4 0 -8 -16 -24 -64 -72 -80 -88 5 0 -8 -16 -24 -72 -80 -88 -96 6 0 -8 -16 -24 -80 -88 -96 -104 7 0 -8 -16 -24 -88 -96 -104 -112

And in octal (addition)

```
0    1    2    3    4    5    6    7
```

0 0 0 0 0 0 0 0 1 0 0 0 0 70 70 70 70 2 0 0 0 0 60 60 60 60 3 0 0 0 0 50 50 50 50 4 0 70 60 50 0 70 60 50 5 0 70 60 50 70 60 50 40 6 0 70 60 50 60 50 40 30 7 0 70 60 50 50 40 30 20

If they are both signed (7==-1, 6==-2, 5=–3, 4==-4) we have the following multiplication table:

```
0    1    2    3    4    5    6    7
```

0 0 0 0 0 0 0 0 1 0 1 2 3 74 75 76 77 2 0 2 4 6 70 72 74 76 3 0 3 6 11 64 67 72 75 4 0 74 70 64 20 14 10 4 5 0 75 72 67 14 11 6 3 6 0 76 74 72 10 6 4 2 7 0 77 76 75 4 3 2 1

If the column (X) in unsigned and the row is signed then we have the following correction to add -8∗Ys∗Xunsigned (in decimal...)

```
0    1    2    3    4    5    6    7
```

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 4 0 -8 -16 -24 -32 -40 -48 -56 5 0 -8 -16 -24 -32 -40 -48 -56 6 0 -8 -16 -24 -32 -40 -48 -56 7 0 -8 -16 -24 -32 -40 -48 -56

And in octal (addition)

```
0    1    2    3    4    5    6    7
```

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 4 0 70 60 50 40 30 20 10 5 0 70 60 50 40 30 20 10 6 0 70 60 50 40 30 20 10 7 0 70 60 50 40 30 20 10

Hence the multiplication table:

```
0    1    2    3    4    5    6    7
```

0 0 0 0 0 0 0 0 1 0 1 2 3 4 5 6 7 2 0 2 4 6 10 12 14 16 3 0 3 6 11 14 17 22 25 4 0 74 70 64 60 54 50 44 5 0 75 72 67 64 61 56 53 6 0 76 74 72 70 66 64 62 7 0 77 76 75 74 73 72 71

Hence the multiplication of two 32-bit numbers X and Y, using a 64-bit accumulator A, can be performed by setting A initially to:

0 for unsigned∗unsigned -2^32∗(X[31]?Y) for X signed Y unsigned -2^32∗(Y[31]?X) for Y signed X unsigned -2^32∗(Y[31]?X[31;0] + X[31]?Y[31;0]) for both signed.

The operation of the multiply then requires a 64-bit accumulator

+1 +4 provides 0, 1, 4, 5 (single 35-bit adder) (stage1_0 = 0; stage1_1 = (3b0,in); stage1_4 = (1b0,in,2b0); stage1↩_5 = stage1_1 + stage1_4;)

+1 +4 with optional double provides 0, 2, 8, 10, 1, 3, 9, 11, 4, 6, 12, 14, 5, 7, 13, 15 (one more 36-bit adders) (0 = 0+0; 2=0+stage1_1_dbl; 3=stage1_1+stage1_1_dbl; stage2_add_in_0 = mux(stage1_0, stage1_1, stage1_4, stage1_5) stage2_add_in_1 = mux(stage1_0, stage1_1, stage1_4, stage1_5)<<1

Division

Unsigned division/remainder (column / row)

```
0    1    2    3    4    5    6    7
```

0 7/0 7/0 7/0 7/0 7/0 7/0 7/0 1 0/0 1/0 2/0 3/0 4/0 5/0 6/0 7/0 2 0/0 0/1 1/0 1/1 2/0 2/1 3/0 3/1 3 0/0 0/1 0/2 1/0 1/1 1/2 2/0 2/1 4 0/0 0/1 0/2 0/3 1/0 1/1 1/2 1/3 5 0/0 0/1 0/2 0/3 0/4 1/0 1/1 1/2 6 0/0 0/1 0/2 0/3 0/4 0/5 1/0 1/1 7 0/0 0/1 0/2 0/3 0/4 0/5 0/6 1/0

Signed division/remainder (column / row) (x86 except div by 0) Note: x86, C99 - sign of remainder = sign of dividend

```
0    1    2    3    4    5    6    7
```

0 7/0 7/0 7/0 7/0 7/0 7/0 7/0 7/0 1 0/0 1/0 2/0 3/0 4/0 5/0 6/0 7/0 2 0/0 0/1 1/0 1/1 6/0 7/7 7/0 0/7 3 0/0 0/1 0/2 1/0 7/7 7/0 0/6 0/7 4 0/0 0/1 0/2 0/3 1/0 0/5 0/6 0/7 5 0/0 0/1 0/2 7/0 1/7 1/0 0/6 0/7 6 0/0 0/1 7/0 7/1 2/0 1/7 1/0 0/7 7 0/0 7/0 6/0 5/0 4/0 3/0 2/0 1/0

For positive/positive one can use unsigned division directly For negative/negative one can do -d/-r, and negate the remainder For negative/positive one can do -d/r, then negate the result and the remainder For positive/negative one can do d/-r, then negate the result

So the first cycle of a divide prepares 'positive' d and r and records the signs (as required)

The multiply requires three adders One 34 bit; one 36 bit, one 64 bit. Divide requires compare; it could do 3 compares per cycle, or just one to start with

We have a multiplier register that gets shifted; this can be the divisor that gets shifted

Multiply then occurs with the following states:

Init : adder high 0 is zero or abs(a) if signed and a negative; adder high 1 is zero or abs(b) if signed and b negative; a_reg <= rs1, b_reg <= rs2 Step (until completed) : a_reg = a_reg>>4; mult=a_reg&15; shf=stage; adder is shifter + acc, with carry chain. complete if a_reg&15 will be 0 Result valid: accumulator has result (present top or bottom half)

Divide occurs with the following states:

Init Shift Step (until completed) Result valid (provides result signing stuff)

Hence the design is for a data pipeline with (for multiply):

a_reg - contains multiplier b_reg - contains multiplicand accumulator - contains 64-bit result of multiply

- initialize with 0 for unsigned∗unsigned; -2^32∗(X[31]?Y) for X signed Y unsigned; -2^32∗(Y[31]?X) for Y signed X unsigned; -2^32∗(Y[31]?X[31;0] + X[31]?Y[31;0]) for both signed mult_data = b_reg ∗ bottom 4 bits of a_reg mult_shf = b_reg ∗ bottom 4 bits of a_reg shifted to be in correct position for accumulation (i.e. shift left by 4∗stage) 64-bit adder of accumulator plus mult_shf

Result is accumulator - pick top or bottom 32 bits as required

For divide it becomes:

a_reg - contains abs(a) (if signed) else a (dividend) b_reg - contains -abs(b) (if signed) else -b (divisor) accumulator - bottom 32 bits contains remainder (initialized to a_reg)

- top 32 bits contain quotient (initialized to zero) mult_data = b_reg << bottom 2 bits of stage mult_shf = b_reg shifted to be in correct position for subtraction from remainder 32-bit adder of low accumulator plus mult_shf; if >=0 then must update accumulator low and set bit in accumulator high 32-bit 'set bit N of' of high accumulator to build quotient if

Quotient result is accumulator high, or negated accumulator high if signed and signs of two inputs differ Remainder result is accumulator low, or negated accumulator low if signed and dividend input was negative

4.28 riscv_i32_pipeline_debug

Files

- file riscv_i32_pipeline_debug.cdl

  Low-gate-count RISC-V pipeline debug module.

4.28.1 Detailed Description

4.28.2 Modules

4.28.2.1 module riscv_i32_pipeline_debug::riscv_i32_pipeline_debug (clock *clk,* input bit *reset_n,* input t_riscv_debug_mst *debug_mst,* output t_riscv_debug_tgt *debug_tgt,* output t_riscv_pipeline_debug_control *debug_control,* input t_riscv_pipeline_debug_response *debug_response,* input bit *rv_select[6]*)

Parameters

in	*rv_select*	This is a fully synchronous pipeline debug module.

It is designed to feed data in to a RISC-V pipeline (being merged with instruction fetch responses), and it takes commands and reports out to a RISC-V debug module.

4.29 riscv_i32_trace

Files

- file riscv_i32_trace.cdl

  Instruction trace for RISC-V implementation.

4.29.1 Detailed Description

4.29.2 Modules

4.29.2.1 module riscv_i32_trace::riscv_i32_trace (clock *clk*, input bit *reset_n*, input t_riscv_i32_trace *trace*)

Trace instruction execution PC

pc

branch_taken

branch_target

instr

flow

branch_taken

trap

retire

rfw

rd

data

Parameters

in	*clk*	Clock for the CPU
in	*reset_n*	Active low reset
in	*trace*	Trace signals

4.30 riscv_i32c_decode

Files

- file riscv_i32c_decode.cdl

  Instruction decoder for RISC-V implementation.

4.30.1 Detailed Description

4.30.2 Modules

4.30.2.1 module riscv_i32c_decode::riscv_i32c_decode (input t_riscv_i32_inst *instruction,* output t_riscv_i32_decode *idecode,* input t_riscv_config *riscv_config*)

Parameters

in	*riscv_config*	Instruction decoder for RISC-V I16 instruction set.

This is based on the RISC-V v2.2 specification (hence figure numbers are from that specification)

4.31 riscv_i32c_pipeline

4.31.1 Detailed Description

4.31.2 Modules

4.31.2.1 module riscv_i32c_pipeline::riscv_i32c_pipeline (clock *clk,* input bit *reset_n,* input t_riscv_irqs *irqs,* output t_riscv_mem_access_req *dmem_access_req,* input t_riscv_mem_access_resp *dmem_access_resp,* output t_riscv_fetch_req *ifetch_req,* input t_riscv_fetch_resp *ifetch_resp,* output t_riscv_i32_coproc_controls *coproc_controls,* input t_riscv_i32_coproc_response *coproc_response,* input t_riscv_config *riscv_config,* output t_riscv_i32_trace *trace*)

Parameters

in	*irqs*	Interrupts in to the CPU
out	*trace*	This is just the processor pipeline, using a single stage for execution.

The instruction fetch request for the next cycle is put out just after the ALU stage logic, which may be a long time into the cycle; the fetch data response presents the instruction fetched at the end of the cycle, where it is registered for execution.

The pipeline is then a single stage that takes the fetched instruction, decodes, fetches register values, and executes the ALU stage; determining in half a cycle the next instruction fetch, and in the whole cycle the data memory request, which is valid just before the end

A coprocessor is supported; this may be configured to be disabled, in which case the outputs are driven low and the inputs are coprocessor response is ignored.

A coprocessor can implement, for example, the multiply for i32m (using riscv_i32_muldiv).

4.32 riscv_i32c_pipeline2

Files

- file riscv_simple.cdl

  Very simple RISC-V implementation ported to CDL.

4.32.1 Detailed Description

4.32.2 Modules

4.32.2.1 module riscv_i32c_pipeline2::riscv_i32c_pipeline2 (clock *clk,* input bit *reset_n,* input t_riscv_irqs *irqs,* output t_riscv_mem_access_req *dmem_access_req,* input t_riscv_mem_access_resp *dmem_access_resp,* output t_riscv_fetch_req *ifetch_req,* input t_riscv_fetch_resp *ifetch_resp,* input t_riscv_config *riscv_config,* output t_riscv_i32_trace *trace*)

Parameters

in	*irqs*	Interrupts in to the CPU
out	*trace*	

4.33 riscv_i32c_pipeline3

4.33.1 Detailed Description

4.33.2 Modules

4.33.2.1 module riscv_i32c_pipeline3::riscv_i32c_pipeline3 (clock *clk,* input bit *reset_n,* input **t_riscv_irqs** *irqs,* output **t_riscv_mem_access_req** *dmem_access_req,* input **t_riscv_mem_access_resp** *dmem_access_resp,* output **t_riscv_fetch_req** *ifetch_req,* input **t_riscv_fetch_resp** *ifetch_resp,* output **t_riscv_i32_coproc_controls** *coproc_controls,* input **t_riscv_i32_coproc_response** *coproc_response,* input **t_riscv_config** *riscv_config,* output **t_riscv_i32_trace** *trace*)

Parameters

in	*irqs*	Interrupts in to the CPU
out	*trace*	This is just the processor pipeline, using thress stages for execution.

The decode and RFR is performed in the first stage

The ALU execution (and coprocessor execution) is performed in the second stage

Memory operations are performed in the third stage

Register file is written at the end of the third stage; there is a RFW stage to forward data from RFW back to execution.

Instruction fetch

The instruction fetch request for the next cycle is put out just after the ALU stage logic, which may be a long time into the cycle (althought the design keeps this to a minimum); the fetch data response presents the instruction fetched at the end of the cycle, where it is registered for execution.

The instruction fetch response must then be valid combinatorially based on the instruction fetch request.

Data memory access

The data memory request is presented in the ALU stage, for an access to complete during the memory stage.

To support simple synchronous memory operation the data memory access includes valid write data in the same cycle as the request.

The data memory response is valid one cycle later than a request. This includes a wait signal. The external memory subsystem, therefore, is a two stage pipeline. The wait signal controls whether an access completes, but not if an access can be taken (except indirectly).

Hence external logic must always either register a request or guarantee not to assert wait.

An example implementation of could be dmem_access_resp.wait = fn (access_in_progress); access_can_be_↩ taken = (!access_in_progress.valid) || (!dmem_access_resp.wait); if (access_can_be_taken) { access_in_progress <= dmem_access_req; } }

4.34 riscv_jtag_apb_dm

Files

- file jtag_apb.cdl

  JTAG tap client to APB master module.

4.34.1 Detailed Description

4.34.2 Modules

4.34.2.1 module riscv_jtag_apb_dm::riscv_jtag_apb_dm (clock *jtag_tck,* input bit *reset_n,* input bit *ir[5],* input t_jtag_action *dr_action,* input *bitdr_in[50],* output *bitdr_tdi_mask[50],* output *bitdr_out[50],* clock *apb_clock,* output t_apb_request *apb_request,* input t_apb_response *apb_response*)

JTAG client module that presents an APB master interface

Parameters

in	*jtag_tck*	JTAG TCK signal, used as a clock
in	*reset_n*	Reset that drives all the logic
in	*ir*	JTAG IR which is to be matched against t_jtag_addr
in	*dr_action*	Action to perform with DR (capture or update, else ignore)
in	*bitdr_in*	Data register in; used in update, replaced by dr_out in capture, shift
out	*bitdr_tdi_mask*	One-hot mask indicating which DR bit TDI should replace (depends on IR)
out	*bitdr_out*	Data register out; same as data register in, except during capture when it is replaced by correct data dependent on IR, or shift when it goes right by one
in	*apb_clock*	APB clock signal, asynchronous to JTAG TCK
out	*apb_request*	APB request out
in	*apb_response*	APB response

4.35 riscv_minimal_debug

4.35.1 Detailed Description

4.35.2 Modules

4.35.2.1 module riscv_minimal_debug::riscv_minimal_debug (clock *clk,* input bit *reset_n,* output t_riscv_mem_access_req *dmem_access_req,* input t_riscv_mem_access_resp *dmem_access_resp,* output t_riscv_fetch_req *ifetch_req,* input t_riscv_fetch_resp *ifetch_resp,* input t_riscv_debug_mst *debug_mst,* output t_riscv_debug_tgt *debug_tgt,* input t_riscv_config *riscv_config,* input bit *rv_select[6],* output t_riscv_i32_trace *trace*)

Parameters

in	*debug_mst*	Driven by debug module to all RISC-V cores
out	*debug_tgt*	Wired-or response bus from all RISC-V cores
out	*trace*	

4.36 riscv_simple

Files

- file riscv_simple.cdl

  Very simple RISC-V implementation ported to CDL.

4.36.1 Detailed Description

4.36.2 Modules

4.36.2.1 module riscv_simple::riscv_simple (clock *clk,* input bit *reset_n,* output t_riscv_mem_access_req *dmem_access_req,* input t_riscv_mem_access_resp *dmem_access_resp,* output t_riscv_mem_access_req *imem_access_req,* input t_riscv_mem_access_resp *imem_access_resp,* output t_riscv_i32_trace *trace*)

Parameters

out	*trace*	This processor tries to keep it as simple as possible, with a 3-stage pipeline.

The first stage is instruction fetch; the instruction memory request is put out just before the middle of the cycle, and a memory (running either at 2x the clock speed, or off the negedge of the clock) presents the instruction fetched at the end of the cycle, where it is registered.

The second stage takes the fetched instruction, decodes, fetches register values, and executes the ALU stage; determining in half a cycle the next instruction fetch, and in the whole cycle the data memory request, which is valid just before the end

Mem, CPU , imem_req.7 , imem_resp.9 , ifetch.0, decode.2, RF rd.5 , Exec , dmem_req.9 , dmem_resp.9 , RFW 0 , 0 , fetch A , X , , , , , , , 1 , 0 , - , inst A , , , , , , , 2 , 1 , fetch B , X , inst A , inst A , inst A , inst A, inst A , , 3 , 1 , , inst B , , , , , , inst A , inst A

4.37 cpu6502

Files

- file cpu6502.cdl

  CDL implementation of 6502 CPU core.

4.37.1 Detailed Description

4.37.2 Modules

4.37.2.1 module cpu6502::cpu6502 (clock *clk,* input bit *reset_n,* input bit *ready,* input bit *irq_n,* input bit *nmi_n,* output bit *ba,* output bit *address[16],* output bit *read_not_write,* output bit *data_out[8],* input bit *data_in[8]*)

Clock control logic - phase 0 is always one tick, phase 1 can be extended for reads by 'ready'

Decode 'ir' register (and other state, but not microsequencer)

Data path - drive buses, perform shift, inc/dec, ALU operations

Decimal flag set

Instruction started

pc

ir

acc

x

y

z

n

c

v

i

sp

Parameters

in	*clk*	Clock, rising edge is start of phi1, end of phi2 - the phi1/phi2 boundary is not required
in	*ready*	Stops processor during current instruction. Does not stop a write phase. Address bus reflects current address being read. Stops the phase 2 from happening.
in	*irq_n*	Active low interrupt in
in	*nmi_n*	Active low non-maskable interrupt in
out	*ba*	Goes high during phase 2 if ready was low in phase 1 if read_not_write is 1, to permit someone else to use the memory bus
out	*address*	In real 6502, changes during phi 1 with address to read or write
out	*read_not_write*	In real 6502, changes during phi 1 with whether to read or write

4.38 csr_master_apb

Files

- file csr_master_apb.cdl

  Pipelined CSR request/response master, driven by an APB.

4.38.1 Detailed Description

4.38.2 Modules

4.38.2.1 module csr_master_apb::csr_master_apb (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* input t_csr_response *csr_response,* output t_csr_request *csr_request*)

The documentation of the CSR interface itself is in other files (at this time, csr_target_csr.cdl).

This module drives a CSR interface in response to an incoming APB interface; it is an APB target presenting a CSR master interface. Its purpose is to permit an extension of an APB bus through a CSR target pipelined chain, hence providing for a timing-friendly CSR interface in an FPGA or ASIC.

The APB has a 32-bit `paddr` field, which is presented as 16 bits of CSR select and 16 bits of CSR address on the CSR interface. There is no timeout in this module on the CSR interface, so accesses to CSRs that have no responder on the bus will hang the module.

It is therefore wise to add a CSR target that detects very long transactions, and which responds by acknowledging them, to the CSR chain.

Parameters

in	*clk*	Clock for the APB and CSR interface; must be a superset of all targets clock
in	*reset_n*	Active low reset
in	*apb_request*	APB request from master
out	*apb_response*	APB response to master
in	*csr_response*	Pipelined csr request interface response
out	*csr_request*	Pipelined csr request interface output

4.39 csr_target_apb

Files

- file csr_target_apb.cdl

  Pipelined CSR request/response interface to APB slave interface.

4.39.1 Detailed Description

4.39.2 Modules

4.39.2.1 module csr_target_apb::csr_target_apb (clock *clk,* input bit *reset_n,* input t_csr_request *csr_request,* output t_csr_response *csr_response,* output t_apb_request *apb_request,* input t_apb_response *apb_response,* input bit *csr_select[16]*)

The documentation of the pipelined CSR interface itself is in other files (at this time, csr_target_csr.cdl).

This module provides a CSR target interface, and drives out an APB master request bus. It can therefore be used at the 'leaf' end of a CSR interface tree, to access standard APB peripherals.

The module must be told which `csr_select` it should be listening for on the CSR target interface; it converts any read or write to an APB master request (with top 16 bits of *paddr* zeroed) to the APB request. Hence the APB target attached to this module is accessed by CSR requests with the select set to `csr_select`.

The module is lightweight, effectively being a registered end-point on the CSR interface and a registered APB request.

Parameters

`in`	*clk*	Clock for the CSR interface, possibly gated version of master CSR clock
`in`	*reset_n*	Active low reset
`in`	*csr_request*	Pipelined csr request interface input
`out`	*csr_response*	Pipelined csr request interface response
`out`	*apb_request*	APB request to target
`in`	*apb_response*	APB response from target
`in`	*csr_select*	Hard-wired select value for the client

4.40 csr_target_csr

Files

- file csr_target_csr.cdl

  Pipelined CSR request/response interface to simple CSR read/write.

4.40.1 Detailed Description

4.40.2 Modules

4.40.2.1 module csr_target_csr::csr_target_csr (clock *clk,* input bit *reset_n,* input t_csr_request *csr_request,* output t_csr_response *csr_response,* output t_csr_access *csr_access,* input t_csr_access_data *csr_access_data,* input bit *csr_select[16]*)

This CSR interface is designed to provide a simple CSR access (select, read/write, address, data) to a client from a pipelined request from a master.

The initial design motiviation was to permit a pipelined CSR access from a master to a number of targets, to run off a single fast clock in an FPGA. This requires registering the read data in response to access requests, and registering the request to the targets; the simplest variant being a fixed latency master-to-target and a fixed latency target-to-master. The current design uses a valid/acknowledgement system to replace the fixed latency.

A valid request is received, and if it matches the *csr_select* field then the request is acknowledged. Since the master is a fair distance away, and the *valid* signal will not be removed until an *ack* is seen, the handshake is effectively: valid low, ack low; valid high, ack low; valid high, ack high; valid high, ack low; valid low, ack low.

Hence a valid request starts with valid high in, and ack out low. If this matches the select, then this interface responds with a single cycle of ack high, and the CSR access is performed.

The clock for the client must be based on the same clock as the master. However, it may be a derived clock - in which case the ack will appear to the master to be more than one clock cycle long. The master must manage this, by removing valid when it sees the ack, and waiting until it sees ack is low before starting another transaction.

Read transactions have a further stage, though, compared to writes. A read transaction will follow an 'ack' with a 'read_data_valid' cycle; if a master performs a read then the handshake will be: valid low, ack low; valid high, ack low; valid high, ack high (one target cycle); valid high, ack low, read_data_valid high (one target cycle); valid low, ack low.

In this case the master must again wait until it sees read_data_valid high and then low before starting a new transaction, to allow the target to use a derived clock.

Parameters

in	*clk*	Clock for the CSR interface, possibly gated version of master CSR clock
in	*reset_n*	Active low reset
in	*csr_request*	Pipelined csr request interface input
out	*csr_response*	Pipelined csr request interface response
out	*csr_access*	Registered CSR access request to client
in	*csr_access_data*	Read data valid combinatorially based on csr_access
in	*csr_select*	Hard-wired select value for the client

4.41 csr_target_timeout

Files

- file csr_target_timeout.cdl

  Timeout target to auto-complete CSR transactions on a timeout.

4.41.1 Detailed Description

4.41.2 Modules

4.41.2.1 module csr_target_timeout::csr_target_timeout (clock *clk,* input bit *reset_n,* input t_csr_request *csr_request,* output t_csr_response *csr_response,* input bit *csr_timeout[16]*)

This module provides a CSR target interface which never directly responds to a request, but which will complete a read or write if the request stays for a specified period of time.

This permits any transaction to be attempted by a CSR interface master, even if no target decodes the transaction. Such transactions will be handled by this module.

Parameters

in	*clk*	Clock for the CSR interface, possibly gated version of master CSR clock
in	*reset_n*	Active low reset
in	*csr_request*	Pipelined csr request interface input
out	*csr_response*	Pipelined csr request interface response
in	*csr_timeout*	Number of cycles to wait for until auto-acknowledging a request

4.42 hps_fpga_debug

Files

- file apb_master_axi.cdl

  AXI target to an APB master interface.

4.42.1 Detailed Description

4.42.2 Modules

4.42.2.1 module hps_fpga_debug::hps_fpga_debug (clock *clk,* input bit *reset_n,* clock *lw_axi_clock_clk,* input **t_axi_request** *lw_axi_ar,* output bit *lw_axi_arready,* input **t_axi_request** *lw_axi_aw,* output bit *lw_axi_awready,* output bit *lw_axi_wready,* input **t_axi_write_data** *lw_axi_w,* input bit *lw_axi_bready,* output **t_axi_write_response** *lw_axi_b,* input bit *lw_axi_rready,* output **t_axi_read_response** *lw_axi_r,* input **t_de1_cl_inputs_status** *de1_cl_inputs_status,* output **t_de1_cl_inputs_control** *de1_cl_inputs_control,* output bit *de1_cl_led_data_pin,* clock *de1_cl_lcd_clock,* input bit *de1_cl_lcd_reset_n,* output **t_de1_cl_lcd** *de1_cl_lcd,* output **t_de1_leds** *de1_leds,* input **t_ps2_pins** *de1_ps2_in,* output **t_ps2_pins** *de1_ps2_out,* input **t_ps2_pins** *de1_ps2b_in,* output **t_ps2_pins** *de1_ps2b_out,* clock *de1_vga_clock,* input bit *de1_vga_reset_n,* output **t_adv7123** *de1_vga,* input bit *de1_keys[4],* input bit *de1_switches[10],* input bit *de1_irda_rxd,* output bit *de1_irda_txd*)

4.43 ps2_host

Files

- file ps2_host.cdl

  PS2 interface for keyboard or mouse.

4.43.1 Detailed Description

4.43.2 Modules

4.43.2.1 module ps2_host::ps2_host (clock *clk,* input bit *reset_n,* input t_ps2_pins *ps2_in,* output t_ps2_pins *ps2_out,* output t_ps2_rx_data *ps2_rx_data,* input bit *divider[16]*)

As a PS2 host, to receive data from the slave (the first target for the design), the module:

1. Looks for clock falling
2. If data is low, then assume this is a start bit. Set timeout timer.
3. Wait for clock falling. Clock in data bit 0
4. Wait for clock falling. Clock in data bit 1
5. Wait for clock falling. Clock in data bit 2
6. Wait for clock falling. Clock in data bit 3
7. Wait for clock falling. Clock in data bit 4
8. Wait for clock falling. Clock in data bit 5
9. Wait for clock falling. Clock in data bit 6
10. Wait for clock falling. Clock in data bit 7
11. Wait for clock falling. Clock in parity bit.
12. Wait for clock falling. Clock in stop bit.
13. Wait for clock high.
14. Validate data (stop bit 1, parity correct)

If a timeout timer expires, which could happen if the framing is bad, then an abort can be taken.

Parameters

`in`	*clk*	Clock
`in`	*reset_n*	Active low reset
`in`	*ps2_in*	Pin values from the outside
`out`	*ps2_out*	Pin values to drive - 1 means float high, 0 means pull low
`out`	*ps2_rx_data*	PS2 receive data from the device, in parallel
`in`	*divider*	Clock divider input to generate approx 3us from `clk`

4.44 ps2_host_keyboard

Files

- file ps2_host_keyboard.cdl

  PS2 interface converter for keyboard as host.

4.44.1 Detailed Description

4.44.2 Modules

4.44.2.1 module ps2_host_keyboard::ps2_host_keyboard (clock *clk,* input bit *reset_n,* input t_ps2_rx_data *ps2_rx_data,* output t_ps2_key_state *ps2_key*)

Module to convert from PS2 receive data, from a host PS2 receive module, in to keyboard data (up/down, extended key).

An incoming valid byte helps build the result. An 0xe0 sets the `extended` bit. A 0xf0 sets the `released` bit. The rest set the `key` field, and `valid` out. `valid` is made in to a single cycle pulse.

Parameters

`in`	*clk*	Clock
`in`	*reset_n*	Active low reset
`in`	*ps2_rx_data*	Receive data from a ps2_host module
`out`	*ps2_key*	PS2 key decoded

4.45 jtag_apb

Files

- file jtag_apb.cdl

  JTAG tap client to APB master module.

4.45.1 Detailed Description

4.45.2 Modules

4.45.2.1 module jtag_apb::jtag_apb (clock *jtag_tck,* input bit *reset_n,* input bit *ir[5],* input t_jtag_action *dr_action,* input *bitdr_in[50],* output *bitdr_tdi_mask[50],* output *bitdr_out[50],* clock *apb_clock,* output t_apb_request *apb_request,* input t_apb_response *apb_response*)

JTAG client module that presents an APB master interface

Parameters

in	*jtag_tck*	JTAG TCK signal, used as a clock
in	*reset_n*	Reset that drives all the logic
in	*ir*	JTAG IR which is to be matched against t_jtag_addr
in	*dr_action*	Action to perform with DR (capture or update, else ignore)
in	*bitdr_in*	Data register in; used in update, replaced by dr_out in capture, shift
out	*bitdr_tdi_mask*	One-hot mask indicating which DR bit TDI should replace (depends on IR)
out	*bitdr_out*	Data register out; same as data register in, except during capture when it is replaced by correct data dependent on IR, or shift when it goes right by one
in	*apb_clock*	APB clock signal, asynchronous to JTAG TCK
out	*apb_request*	APB request out
in	*apb_response*	APB response

4.46 jtag_tap

Files

- file jtag_tap.cdl

  JTAG tap controller module.

4.46.1 Detailed Description

4.46.2 Modules

4.46.2.1 module jtag_tap::jtag_tap (clock *jtag_tck,* input bit *reset_n,* input t_jtag *jtag,* output bit *tdo,* output bit *ir[ir_length],* output t_jtag_action *dr_action,* output bit *dr_in[dr_length],* input bit *dr_tdi_mask[dr_length],* input bit *dr_out[dr_length]*)

JTAG TAP controller module that basically implements the JTAG state machine, holds an IR, and interacts with a client to capture and update data.

Parameters

in	*jtag_tck*	JTAG TCK
in	*reset_n*	Reste for all the logic
in	*jtag*	JTAG inputs
out	*tdo*	JTAG TDO pin
out	*ir*	IR register to be used by client
out	*dr_action*	DR action (capture, update, shift, or none)
out	*dr_in*	DR to be fed to client
in	*dr_tdi_mask*	One-hot mask indicating where TDI should be inserted (based on DR length, based on IR)
in	*dr_out*	DR from client (from client data if capture, shifted if shift)

4.47 led_seven_segment

Files

- file led_seven_segment.cdl

  Simple module to support 7-segment hex display.

4.47.1 Detailed Description

4.47.2 Modules

4.47.2.1 module led_seven_segment::led_seven_segment (input bit *hex[4],* output bit *leds[7]*)

Simple module to map a hex value to the LEDs required to make the appropriate symbol in a 7-segment display.

The module combinatorially takes in a hex value, and drives out 7 LED values.

Parameters

in	*hex*	Hexadecimal to display on 7-segment LED
out	*leds*	1 for LED on, 0 for LED off, for segments a-g in bits 0-7

4.48 led_ws2812_chain

Files

- file led_ws2812_chain.cdl

  'Neopixel' LED chain driver module

4.48.1 Detailed Description

4.48.2 Modules

4.48.2.1 module led_ws2812_chain::led_ws2812_chain (clock *clk,* input bit *reset_n,* input bit *divider_400ns[8],* output t_led_ws2812_request *led_request,* input t_led_ws2812_data *led_data,* output bit *led_chain*)

A chain of any length of Neopixel LEDs can be driven by this module

The interface is a request/data interface; this module presents a *ready* request to the client, which then presents a valid 24-bit RGB data value. When the module takes the data it removes its *ready* request. The client keeps supplying data in response to the *ready* requests.

To terminate the chain the client supplies data with a *last* indication asserted.

To ease implementation of the client, the request includes a *first* indicator and an *led_number* indicator - effectively a client can read a register file based on *led_number* and drive *valid* when the data is valid, and *last* if led_number matches the end of the register file.

This module copes with all of the requirements of the Neopixel chain, and it takes a constant clock input. To provide the correct frequency of data pin toggling to the Neopixels a clock divider value must be supplied, with the approximate number of clock ticks that make up 400ns (ideally 408ns).

The Neopixel WS2812 LED chains use a serial data stream with encoded clock to provide data to the LEDs.

If the LED chain data is held low for >50us then the stream performs a 'load to LEDs' - this transfers the serial data already loaded in to the LEDs to the actual LED drivers themselves.

Before loading the LEDs the chain should be fed data. The data is fed using a high/low data pulse per bit. The ratio high/low provides the data bit value.

A high/low of 1:2 provids a zero bit; a high/low of 1:2 provides a one bit. The total bit time should be 1.25us. Hence this logic requires a 1.25/3us, or roughly 400ns clock generator. This is performed using a clock divider and a user-supplied divide value, which will depend on the input clock frequency. For example, if the input clock frequency is 50MHz, which is a period of 20ns, then the divider should be set to 20.

The data is provided to the LEDs green, red then blue, most significant bit first, with 8 bits for each component.

The logic uses a simple state machine; when it is idle it will have no data in hand, and need data to feed in to the LED stream. At this point it requests a valid first LED data. When valid data is received into a buffer the state machine transitions to the data-in-hand state; it remains there until the data transmitter takes the data, when it either requests more data (as per idle), or if the last LED data was provided by the client, it moves to requests an LED load, and it waits in loading state until that completes. At this point it transitions back to idle, and the process restarts.

When there is valid LED data in the internal buffer the data transmitter can start; the data is transferred to the shift register, and it is driven out by the data transmitter to the LED chain one bit at a time.

Parameters

in	*clk*	System clock - not the pin clock, which is derived from this
in	*reset_n*	Active low reset
in	*divider_400ns*	clock divider value to provide for generating a pulse every 400ns based on clk
out	*led_request*	LED data request, to get data from the next LED to light
in	*led_data*	LED data, for the requested led
out	*led_chain*	Data pin for LED chain, modulated by this module to drive LED settings

4.49 bbc_csr_interface

4.49.1 Detailed Description

4.49.2 Modules

4.49.2.1 module bbc_csr_interface::bbc_csr_interface (clock *clk,* input bit *reset_n,* input t_bbc_csr_request *csr_request,* output t_bbc_csr_response *csr_response,* output t_bbc_csr_access *csr_access,* input t_bbc_csr_access_data *csr_read_data,* input bit *csr_select[16]*)

This CSR interface is designed to provide a simple CSR access (select, read/write, address, data) to a client from a pipelined request from a master.

The initial design motiviation was to permit a pipelined CSR access from a master to a number of targets, to run off a single fast clock in an FPGA. This requires registering the read data in response to access requests, and registering the request to the targets; the simplest variant being a fixed latency master-to-target and a fixed latency target-to-master. The current design uses a valid/acknowledgement system to replace the fixed latency.

A valid request is received, and if it matches the field then the request is acknowledged. Since the master is a fair distance away, and the signal will not be removed until an is seen, the handshake is effectively: valid low, ack low; valid high, ack low; valid high, ack high; valid high, ack low; valid low, ack low.

Hence a valid request starts with valid high in, and ack out low. If this matches the select, then this interface responds with a single cycle of ack high, and the CSR access is performed.

The clock for the client must be based on the same clock as the master. However, it may be a derived clock - in which case the ack will appear to the master to be more than one clock cycle long. The master must manage this, by removing valid when it sees the ack, and waiting until it sees ack is low before starting another transaction.

Read transactions have a further stage, though, compared to writes. A read transaction will follow an 'ack' with a 'read_data_valid' cycle; if a master performs a read then the handshake will be: valid low, ack low; valid high, ack low; valid high, ack high (one target cycle); valid high, ack low, read_data_valid high (one target cycle); valid low, ack low.

In this case the master must again wait until it sees read_data_valid high and then low before starting a new transaction, to allow the target to use a derived clock.

Parameters

in	*clk*	Clock for the CSR interface
in	*csr_request*	Pipelined csr request interface input
out	*csr_response*	Pipelined csr request interface response
out	*csr_access*	Registered CSR access request to client
in	*csr_read_data*	Read data valid combinatorially based on csr_access
in	*csr_select*	Hard-wired select value for the client

4.50 bbc_display_sram

Files

- file bbc_display_sram.cdl

  BBC micro display to SRAM write interface module.

4.50.1 Detailed Description

4.50.2 Modules

4.50.2.1 module bbc_display_sram::bbc_display_sram (clock *clk,* input bit *reset_n,* input t_bbc_display *display,* output t_bbc_display_sram_write *sram_write,* input t_csr_request *csr_request,* output t_csr_response *csr_response*)

Parameters

in	*clk*	Clock running at 2MHz
out	*csr_response*	This module mimics a monitor attached to the BBC video output, generating a stream of SRAM write requests as pixels are driven by the video output signals.

A regular video stream (from the BBC micro) runs at 2MHz with either 6 or 8 pixels per tick. On the BBC micro this is a pixel clock of either 16MHz or 12MHz.

The 't_bbc_display' indicates 1, 2, 4, 6 or 8 pixels per clock - but the interpretation here is for either 6 or 8 - since 1, 2 and 4 'pixels per clock' is the internal BBC pixels, which have been replicated on the bus. This should probably be fixed rather than explained.

The module is designed with a display input stage that manages vsync and hsync, and which then handles the 'back porch' for both vertical and horizontal blanking. The 'back porch' is the number of pixel clocks or scanlines that should not be captured following the detection of hsync/vsync respectively.

The display input stage then combines the input pixel data with the blanking for back porches to produce a validated pixel stream for the second stage of the module. Coupled to this are restart frame/line indicators.

For interlaced capture (which most monitors would be) the vsync will occur at different points in a line for even and odd fields. Even fields are SRAM addresses 0, 2, 4 (in 'line' terms), and odd fields are SRAM address 1, 3, 5 (again in 'line' terms). So the display input stage determines if a vsync corresponds to an odd or an even field.

The second stage is the SRAM data collation stage. This gathers the valid pixels from the display input stage into a shift register, and when 16 pixels are ready to be written they are passed to the SRAM write output stage. This SRAM data collation stage manages the SRAM addresses, resetting to the base address on a frame restart (plus a a single line of an odd field, interlaced), and incrementing the address on every write. A fixed number of SRAM writes is permitted per line (to set the captured display width). A fixed number of scanlines is permitted per frame (field).

Note that at the end of a line, for interlaced frames, the SRAM address is moved down by a line too, so that even fields do write to even 'lines' in SRAM, and odd lines just to the odd 'lines'.

4.51 bbc_floppy_sram

Files

- file bbc_floppy_sram.cdl

  BBC micro floppy to SRAM read/write interface module.

4.51.1 Detailed Description

4.51.2 Modules

4.51.2.1 module bbc_floppy_sram::bbc_floppy_sram (clock *clk,* input bit *reset_n,* input **t_bbc_floppy_op** *floppy_op,* output **t_bbc_floppy_response** *floppy_response,* output **t_bbc_floppy_sram_request** *sram_request,* input **t_bbc_floppy_sram_response** *sram_response,* input **t_csr_request** *csr_request,* output **t_csr_response** *csr_response*)

Parameters

in	*clk*	Clock running at 2MHz
out	*csr_response*	This module provides an SRAM-fakeout of a set of floppy disks, tied to the BBC micro floppy disc controller.

4.52 bbc_keyboard_csr

Files

- file bbc_display_sram.cdl

  BBC micro display to SRAM write interface module.

4.52.1 Detailed Description

4.52.2 Modules

4.52.2.1 module bbc_keyboard_csr::bbc_keyboard_csr (clock *clk*, input bit *reset_n*, output t_bbc_keyboard *keyboard*, input bit *keyboard_reset_n*, input t_csr_request *csr_request*, output t_csr_response *csr_response*)

Parameters

in	*clk*	Clock running at 2MHz
out	*csr_response*	This module provides a keyboard source from CSR writes

4.53 bbc_keyboard_ps2

Files

- file bbc_keyboard_ps2.cdl

  BBC micro keyboard from PS2 keys.

4.53.1 Detailed Description

4.53.2 Modules

4.53.2.1 module bbc_keyboard_ps2::bbc_keyboard_ps2 (clock *clk,* input bit *reset_n,* input t_ps2_key_state *ps2_key,* output t_bbc_keyboard *keyboard*)

Parameters

in	*clk*	Clock of PS2 keyboard
out	*keyboard*	This module provides a BBC keyboard source from a PS2 keyboard, using a mapping ROM

4.54 bbc_micro

Files

- file bbc_micro.cdl

  BBC microcomputer implementation module.

4.54.1 Detailed Description

4.54.2 Modules

4.54.2.1 module bbc_micro::bbc_micro (clock *clk,* input t_bbc_clock_control *clock_control,* output t_bbc_clock_status *clock_status,* input bit *reset_n,* input t_bbc_keyboard *keyboard,* output t_bbc_display *display,* output bit *keyboard_reset_n,* output t_bbc_floppy_op *floppy_op,* input t_bbc_floppy_response *floppy_response,* input t_bbc_micro_sram_request *host_sram_request,* output t_bbc_micro_sram_response *host_sram_response*)

Clock that mirrors 2MHz falling - video data from RAM is valid at this edge, so used by CRTC, SAA5050 latches, SAA5050, vidproc

Clock that mirrors 1MHzE rising - 1MHz system clock - used by keyboard and SAA5050

Clock that mirrors 1MHzE falling, end of 1MHz CPU bus cycle, used by 6522, 6850, 6845, some latches

6502 clock, >=2MHz but extended when accessing 1MHz peripherals

Parameters

in	*clk*	Clock at least at '4MHz' - CPU runs at least half of this

4.55 bbc_micro_clocking

Files

- file bbc_micro_clocking.cdl

  BBC microcomputer clock generation module.

4.55.1 Detailed Description

4.55.2 Modules

4.55.2.1 module bbc_micro_clocking::bbc_micro_clocking (clock *clk,* input bit *reset_n,* input t_bbc_clock_status *clock_status,* output t_bbc_clock_control *clock_control,* input t_csr_request *csr_request,* output t_csr_response *csr_response*)

Parameters

in	*clk*	4MHz clock in as a minimum

4.56 bbc_micro_keyboard

Files

- file bbc_micro_keyboard.cdl

  BBC microcomputer keyboard module.

4.56.1 Detailed Description

4.56.2 Modules

4.56.2.1 module bbc_micro_keyboard::bbc_micro_keyboard (clock *clk,* input bit *reset_n,* output bit *reset_out_n,* input bit *keyboard_enable_n,* input bit *column_select[4],* input bit *row_select[3],* output bit *key_in_column_pressed,* output bit *selected_key_pressed,* input **t_bbc_keyboard** *bbc_keyboard*)

Parameters

out	*reset_out_n*	From the Break key
in	*keyboard_enable_n*	Asserted to make keyboard detection operate
in	*column_select*	Wired to pa[4;0], and indicates which column of the keyboard matrix to access
in	*row_select*	Wired to pa[3;4], and indicates which row of the keyboard matrix to access
out	*key_in_column_pressed*	Wired to CA2, asserted if keyboard_enable_n and a key is pressed in the specified column (other than row 0)
out	*selected_key_pressed*	Asserted if keyboard_enable_n is asserted and the selected key is pressed

4.57 bbc_micro_rams

Files

- file bbc_micro_rams.cdl

  BBC microcomputer RAMs module.

4.57.1 Detailed Description

4.57.2 Modules

4.57.2.1 module bbc_micro_rams::bbc_micro_rams (clock *clk*, input bit *reset_n*, input t_bbc_clock_control *clock_control*, input t_bbc_micro_sram_request *host_sram_request*, output t_bbc_micro_sram_response *host_sram_response*, input t_bbc_display_sram_write *display_sram_write*, input t_bbc_floppy_sram_request *floppy_sram_request*, output t_bbc_floppy_sram_response *floppy_sram_response*, output t_bbc_micro_sram_request *bbc_micro_host_sram_request*, input t_bbc_micro_sram_response *bbc_micro_host_sram_response*)

Parameters

in	*clk*	4MHz clock in as a minimum

4.58 bbc_micro_with_rams

Files

- file bbc_micro_with_rams.cdl

  BBC microcomputer with RAMs module.

4.58.1 Detailed Description

4.58.2 Modules

4.58.2.1 module bbc_micro_with_rams::bbc_micro_with_rams (clock *clk,* clock *video_clk,* input bit *reset_n,* input t_csr_request *csr_request,* output t_csr_response *csr_response,* input t_bbc_micro_sram_request *host_sram_request,* output t_bbc_micro_sram_response *host_sram_response,* output t_bbc_display_sram_write *display_sram_write,* output t_video_bus *video_bus*)

Clock that mirrors 2MHz falling - video data from RAM is valid at this edge, so used by CRTC, SAA5050 latches, SAA5050, vidproc

6502 clock, >=2MHz but extended when accessing 1MHz peripherals

Parameters

in	*clk*	4MHz clock in as a minimum

4.59 bbc_vidproc

Files

- file bbc_vidproc.cdl

  BBC microcomputer video ULA CDL implementation.

4.59.1 Detailed Description

4.59.2 Modules

4.59.2.1 module bbc_vidproc::bbc_vidproc (clock *clk_cpu,* clock *clk_2MHz_video,* input bit *reset_n,* input bit *chip_select_n,* input bit *address,* input bit *cpu_data_in[8],* input bit *pixel_data_in[8],* input bit *disen,* input bit *invert_n,* input bit *cursor,* input bit *saa5050_red[6],* input bit *saa5050_green[6],* input bit *saa5050_blue[6],* output bit *crtc_clock_enable,* output bit *red[8],* output bit *green[8],* output bit *blue[8],* output t_bbc_pixels_per_clock *pixels_valid_per_clock*)

Parameters

in	*clk_cpu*	2MHz bus clock
in	*clk_2MHz_video*	2MHz video
in	*reset_n*	Not present on the chip, but required for the model - power up reset
in	*chip_select_n*	Active low chip select
in	*address*	Valid with chip select
in	*cpu_data_in*	Data in (from CPU)
in	*pixel_data_in*	Data in (from SRAM)
in	*disen*	Asserted by CRTC if black output required (e.g. during sync)
in	*invert_n*	Asserted (low) if the output should be inverted (post-disen probably)
in	*cursor*	Asserted for first character of a cursor
in	*saa5050_red*	3 pixels in at 2MHz, red component, from teletext
in	*saa5050_green*	3 pixels in at 2MHz, green component, from teletext
in	*saa5050_blue*	3 pixels out at 2MHz, blue component, from teletext
out	*crtc_clock_enable*	High for 2MHz, toggles for 1MHz - the 'character clock' - used also to determine when the shift register is loaded
out	*red*	8 pixels out at 2MHz, red component
out	*green*	8 pixels out at 2MHz, green component
out	*blue*	8 pixels out at 2MHz, blue component

4.60 picoriscv

Files

- file picoriscv.cdl

  Pico-RISC-V microcomputer implementation module.

4.60.1 Detailed Description

4.60.2 Modules

4.60.2.1 module picoriscv::picoriscv (clock *clk,* input bit *reset_n,* clock *video_clk,* input bit *video_reset_n,* output t_video_bus *video_bus,* input t_prv_keyboard *keyboard,* input t_csr_request *csr_request,* output t_csr_response *csr_response*)

Parameters

in	*clk*	Clock, divided down for CPU
in	*reset_n*	Active low reset
in	*video_clk*	Video clock, independent of CPU clock
in	*video_reset_n*	Active low reset

4.61 picoriscv_clocking

Files

- file picoriscv_clocking.cdl

  Clock control for pico-RISC-V microcomputer.

4.61.1 Detailed Description

4.61.2 Modules

4.61.2.1 module picoriscv_clocking::picoriscv_clocking (clock *clk,* input bit *reset_n,* input **t_prv_clock_status** *clock_status,* output **t_prv_mem_control** *mem_control,* output **t_prv_clock_control** *clock_control,* input **t_csr_request** *csr_request,* output **t_csr_response** *csr_response*)

Parameters

out	*csr_response*	This module controls the clocking of a Pico-risc-V microcomputer

4.62 acia6850

Files

- file acia6850.cdl

  6850 async communications chip CDL implementation

4.62.1 Detailed Description

4.62.2 Modules

4.62.2.1 module acia6850::acia6850 (clock *clk,* input bit *reset_n,* input bit *read_not_write,* input bit *chip_select[2],* input bit *chip_select_n,* input bit *address,* input bit *data_in[8],* output bit *data_out[8],* output bit *irq_n,* input bit *tx_clk,* input bit *rx_clk,* output bit *txd,* input bit *cts,* input bit *rxd,* output bit *rts,* input bit *dcd*)

Parameters

in	*clk*	Clock that rises when the 'enable' of the 6850 completes - but a real clock for this model
in	*read_not_write*	Indicates a read transaction if asserted and chip selected
in	*chip_select*	Active high chip select
in	*chip_select_n*	Active low chip select
in	*address*	Changes during phase 1 (phi[0] high) with address to read or write
in	*data_in*	Data in (from CPU)
out	*data_out*	Read data out (to CPU)
out	*irq_n*	Active low interrupt
in	*tx_clk*	Clock used for transmit data - must be really about at most quarter the speed of clk
in	*rx_clk*	Clock used for receive data - must be really about at most quarter the speed of clk

4.63 via6522

Files

- file via6522.cdl

  CDL implementation of a 6522 versatile interface adaptor (VIA)

4.63.1 Detailed Description

4.63.2 Modules

4.63.2.1 module via6522::via6522 (clock *clk,* clock *clk_io,* input bit *reset_n,* input bit *read_not_write,* input bit *chip_select,* input bit *chip_select_n,* input bit *address[4],* input bit *data_in[8],* output bit *data_out[8],* output bit *irq_n,* input bit *ca1,* input bit *ca2_in,* output bit *ca2_out,* output bit *pa_out[8],* input bit *pa_in[8],* input bit *cb1,* input bit *cb2_in,* output bit *cb2_out,* output bit *pb_out[8],* input bit *pb_in[8]*)

Control registers

Parameters

in	*clk*	1MHz clock rising when bus cycle finishes
in	*clk_io*	1MHz clock rising when I/O should be captured - can be antiphase to clk
in	*read_not_write*	Indicates a read transaction if asserted and chip selected
in	*chip_select*	Active high chip select
in	*chip_select_n*	Active low chip select
in	*address*	Changes during phase 1 (phi[0] high) with address to read or write
in	*data_in*	Data in (from CPU)
out	*data_out*	Read data out (to CPU)
out	*irq_n*	Active low interrupt
in	*ca1*	Port a control 1 in
in	*ca2_in*	Port a control 2 in
out	*ca2_out*	Port a control 2 out
out	*pa_out*	Port a data out
in	*pa_in*	Port a data in
in	*cb1*	Port b control 1 in
in	*cb2_in*	Port b control 2 in
out	*cb2_out*	Port b control 2 out
out	*pb_out*	Port b data out
in	*pb_in*	Port b data in

4.64 fdc8271

Files

- file fdc8271.cdl

  CDL implementation of 8271 FDC.

4.64.1 Detailed Description

4.64.2 Modules

4.64.2.1 module fdc8271::fdc8271 (clock *clk,* input bit *reset_n,* input bit *chip_select_n,* input bit *read_n,* input bit *write_n,* input bit *address[2],* input bit *data_in[8],* output bit *data_out[8],* output bit *irq_n,* output bit *data_req,* input bit *data_ack_n,* output bit *select[2],* input bit *ready[2],* output bit *fault_reset,* output bit *write_enable,* output bit *seek_step,* output bit *direction,* output bit *load_head,* output bit *low_current,* input bit *track_0_n,* input bit *write_protect_n,* input bit *index_n,* output **t_bbc_floppy_op** *bbc_floppy_op,* input **t_bbc_floppy_response** *bbc_floppy_response*)

Diskettes had a standard format, with up to 80 (or so) tracks, each with a fixed layout

Each track would 'start' at the index marker, with a sync gap; the track then contained 'N' sectors each with an ID, a sync gap, data and another sync gap.

At the end there would be a final gap - but not a sync gap. A sync gap is 8hff's followed by six 8h00's. The final gap is all 8hff.

What this means is that effectively a track is 1's until the first sector Each sector is 48 0's, then a sector ID (which starts with a 1) followed by 1's with about 48 0's separating the ID from the sector data. The sector data starts with a single marker byte (starting with a 1) followed by the data and a CRC, followed by 1's.

The 48 0's may not always be 48, and the 1's may vary too - these are effectively start/stop regions, which can be encroached on by variations in disk speed.

Bytes on the disk are stored with a high clock pulse every 4us, and a low or high data pulse in the middle (i.e. after 2us).

Each bit on the disk is normally 4us, and a track is notionally 8∗5,208 bits, so 166,656us (basically 1/6th of a second). This is because the disk spins at 360rpm, 6rps. At 4us/bit, a byte is read a 32us/byte - this is the NMI response time.

Note that the index markers have gapped clocks, to identify them properly - but the data is guaranteed to have ones, so the disk will not have just zeros for the index markers.

A simple implementation of a disk system might just support the sector data with a fixed number of sectors. However, for more flexibility (and perhaps ease of hardware implementation here) an alternative approach can be taken. This is to have a sector descriptor memory, as well as a disk data memory.

The sector descriptor memory is indexed by track and sector number. Supporting up to 16 sectors per track means that a sector descriptor memory is addressed by {track[7;0], sector[4;0]} - an 11-bit address. The sector descriptor memory is indexed by physical sector - i.e. the position on the track. The sector descriptor memory contains the sector's logical sector number.

The sector descriptor (64 bits) must contain:

bit indicating it is valid (so that a max number of sectors <16 can be used) start address in disk data memory of sector data (excluding the bottom 7 bits must be zero, since sectors are always a multiple of 128 bytes)

id address mark for sector (8 bits) (8hFE, clock pattern 8hc7) track number (7 bits) head number (1 bit) sector number (4 bits) sector length (2 bits, 0=>128, 1=>256, 2=>512, 3=>1024) disk data address mark (8 bits) (8hFB valid, 8hf8 deleted data) (in the sector data itself on the disk) bit indicating ID has bad CRC bit indicating data has bad CRC

A disk descriptor then needs a base address of sector descriptor data, a base address of disk data memory, a number of tracks. For emulation purposes, the disk descriptor also includes details on how realistic the timing should be. It should also have a 'valid' bit (0 if disk not loaded), and a 'write protect' bit

The NMI code is: 7 (NMI brk) 3 0xd00 PHA 4 0xd01 LDA &FE28 ;; FDC Status/Command 2 0xd04 AND #&1F 2 0xd06 CMP #&03 2 0xd08 BNE LBCBA ... error handling 4 0xd0a LDA &FE2B ;; FDC Data 4 0xd0d STA &FFFF ;; Replaced with destination address 6 0xd10 INC 0xD0E 3 0xd13 BNE 0xd18

- 0xd15 INC 0xD0F 4 0xd18 PLA 6 0xd19 RTI

Total of 47 cycles, or 23.5us per byte for data transfers

```
On a real drive, each track has 5208 bytes.
  index mark
  post-index gap, 32 bytes (26 0xff, 6 0x00 sync)
  Numsectors * { id field, 7 bytes; post-id field gap 17 bytes (11 0xff, 6 0x00 sync); data field (n bytes); p
  trailing gap (40 0xff, 6 0x00 sync)

id field is:
  id address mark
  track address (00-74, officially in 8271)
  head address (0 or 1)
  sector address (01-26)
  sector length (0=>128, 1=>256, 2=>512)
  2 bytes CRC

data field is:
  data address mark
  N bytes data
  2 byte CRC
```

Command

command

Parameter

parameter

Write action

action

Status read

status

Result read

result

Seek track

track

Seek sector id

track

sector

Load head

track

Find index

track

Read id

track

Read data

track

sector

Parameters

in	*clk*	
in	*reset_n*	8271 has an active high reset, but...
in	*chip_select_n*	Active low chip select
in	*read_n*	Indicates a read transaction if asserted and chip selected
in	*write_n*	Indicates a write transaction if asserted and chip selected
in	*address*	Address of register being accessed
in	*data_in*	Data in (from CPU)
out	*data_out*	Read data out (to CPU)
out	*irq_n*	Was INT on the 8271, but that means something else now; active low interrupt
out	*data_req*	
in	*data_ack_n*	
out	*select*	drive select
in	*ready*	drive ready
out	*fault_reset*	
out	*write_enable*	High if the drive should write data
out	*seek_step*	High if the drive should step
out	*direction*	Direction of step
out	*load_head*	Enable drive head
out	*low_current*	Asserted for track>=43
in	*track_0_n*	Asserted low if the selected drive is on track 0
in	*write_protect_n*	Asserted low if the selected drive is write-protected
in	*index_n*	Asserted low if the selected drive photodiode indicates start of track
out	*bbc_floppy_op*	Model drive operation, including write data
in	*bbc_floppy_response*	Parallel data read, specific to the model

4.65 dprintf

Files

- file dprintf.cdl

  Debug text formatter.

4.65.1 Detailed Description

4.65.2 Modules

4.65.2.1 module dprintf::dprintf (clock *clk,* input bit *reset_n,* input t_dprintf_req_4 *dprintf_req,* output bit *dprintf_ack,* output t_dprintf_byte *dprintf_byte*)

This module that takes an input debug request and converts it in to a stream of bytes. The debug request is similar to a 'printf' string, in that it allows formatted data.

A request is effectively a bytestream with an SRAM address. The byte stream consists of ASCII characters plus potentially 'video control' characters - all in the range 1 to 127, plus control codes of 0 or 128 to 255.

The code 0 is just skipped; it allows for simple alignment of data in the dprintf request.

A code of 128 to 191 is a zero-padded hex format field. The encoding is 8h10xxssss; x is unused, and the size *ss* is 0-f, indicating 1 to 16 following nybbles are data (msb first). The data follows in the succeeding bytes.

A code of 192 to 254 is a space-padded decimal format field. The The encoding is 8h11ppppss; the *size* is 0-3 for 1 to 4 bytes of data, in the succeeding bytes. The *padding* (pppp) is zero for no padding; 1 forces the string to be at least 2 characters long (prepadded with space if required); 2 is pad to 3 characters, and so on. The maximum padding is to a ten character output (pppp of 9).

A code of 255 terminates the string.

Parameters

in	*clk*	Clock for data in and display SRAM write out
in	*dprintf_req*	Debug printf request
out	*dprintf_ack*	Debug printf acknowledge
out	*dprintf_byte*	Byte to output

4.66 dprintf_2_mux

4.67 dprintf_4_mux

4.68 generic_valid_ack_mux

Files

- file generic_valid_ack_mux.cdl

  A generic valid/ack multiplexer to combine buses with valid/ack protocol.

4.68.1 Detailed Description

4.68.2 Modules

4.68.2.1 module generic_valid_ack_mux::generic_valid_ack_mux (clock *clk,* input bit *reset_n,* input gt_generic_valid_req *req_a,* input gt_generic_valid_req *req_b,* output bit *ack_a,* output bit *ack_b,* output gt_generic_valid_req *req,* input bit *ack*)

Generic multiplexer for two identical requesters (with a valid signal each), to arbitrate for an output request, with a response with an 'ack' signal.

This module may be used with a different type (using type remapping) to generate a specific multiplexer for two validated requests, which have just an ack in response (e.g. the teletext dprintf requests).

The module registers its output request; it remembers which requester it consumed from last, and will preferentially consue from the other port next - hence supplying some degree of fairness.

When its output is not valid, or is being acknowledged, it may take a new request from one of the two requesting masters, using the desired priority. It will also then acknowledge that requester.

If its output is valid and is not acknowledged, then it will not consumer another request.

Parameters

in	*clk*	Clock for logic
in	*reset_n*	Active low reset
in	*req_a*	Request from upstream 'A' port, which must have a valid bit
in	*req_b*	Request from upstream 'B' port, which must have a valid bit
out	*ack_a*	Acknowledge to upstream 'A' port
out	*ack_b*	Acknowledge to upstream 'B' port
out	*req*	Request out downstream, which must have a valid bit
in	*ack*	Acknowledge from downstream

4.69 hysteresis_switch

Files

- file hysteresis_switch.cdl

  A hystersis detector using counter pairs.

4.69.1 Detailed Description

4.69.2 Modules

4.69.2.1 module hysteresis_switch::hysteresis_switch (clock *clk,* input bit *reset_n,* input bit *clk_enable,* input bit *input_value,* output bit *output_value,* input bit *filter_period[16],* input bit *filter_level[16]*)

CDL implementation of a module that takes an input signal and notionally keeps a count of cycles that the input is low, and cycles that the input is high; using these counters it makes a decision on the real value of the output, using hysteresis.

Since infinite history is not sensible and the counters cannot run indefinitely without overflow anyway, the counters divide by 2 on a configurable divider (effectively filtering the input stream).

The two notional counters are *cycles_low* and *cycles_high*.

To switch to a 'high' output from a current 'low' output requires the *cycles_high* - *cycles_low* to be greater than half of the filter period.

To switch to a 'low' output from a current 'high' output requires the *cycles_high* - *cycles_low* to be less than minus half of the filter period.

Hence a n+1 bit difference would need to be maintained for *cycles_high* and *cycles_low*. This difference would increase by 1 if the input is high, and decrease by 1 if the input is low.

Hence an actual implementation can maintain an up/down counter *cycles_diff*, which is divided by 2 every filter period, and which is incremented on input of 1, and decremented on input of 0.

When the output is low and the *cycles_diff* is $>$ half the filter period the output shifts to high.

When the output is high and the *cycles_diff* is $<$ -half the filter period the output shifts to low. Clock for all the logic, based on an enable in

Parameters

in	*clk*	Clock for the module
in	*reset_n*	Active low reset
in	*clk_enable*	Assert to enable the internal clock; this permits I/O switches to easily use a slower clock
in	*input_value*	Input pin level, to apply hysteresis to
out	*output_value*	Output level, after hysteresis
in	*filter_period*	Period over which to filter the input - the larger the value, the longer it takes to switch, but the more glitches are removed
in	*filter_level*	Value to exceed to switch output levels - the larger the value, the larger the hysteresis; must be less than 2$*$filter_period

4.70 crtc6845

Files

- file crtc6845.cdl

  CDL implementation of 6845 CRTC.

4.70.1 Detailed Description

4.70.2 Modules

4.70.2.1 module crtc6845::crtc6845 (clock *clk_2MHz,* clock *clk_1MHz,* input bit *reset_n,* output bit *ma[14],* output bit *ra[5],* input bit *read_not_write,* input bit *chip_select_n,* input bit *rs,* input bit *data_in[8],* output bit *data_out[8],* input bit *lpstb_n,* input bit *crtc_clock_enable,* output bit *de,* output bit *cursor,* output bit *hsync,* output bit *vsync*)

This is an implementation of the Motorola 6845 CRTC, which was used in the BBC microcomputer for sync and video memory address generation.

Parameters

in	*clk_2MHz*	2MHz clock that runs the memory interface and video sync output
in	*clk_1MHz*	Clock that rises when the 'enable' of the 6845 completes - but a real clock for this model - used for the CPU interface
in	*reset_n*	Active low reset
out	*ma*	Memory address
out	*ra*	Row address
in	*read_not_write*	Indicates a read transaction if asserted and chip selected
in	*chip_select_n*	Active low chip select
in	*rs*	Register select - address line really
in	*data_in*	Data in (from CPU) for writing
out	*data_out*	Data out (to CPU) for reading
in	*lpstb_n*	Light pen strobe input, used to capture the memory address of the display when the CRT passes it; not much use nowadays
in	*crtc_clock_enable*	An enable for clk_2MHz for the character clock - on the real chip this is actually a clock
out	*de*	Display enable output, asserted during horizontal display when vertical display is also permitted
out	*cursor*	Driven when the cursor is configured and the cursor address is matched
out	*hsync*	Horizontal sync strobe, of configurable position and width
out	*vsync*	Vertical sync strobe, of configurable position and width

4.71 framebuffer

Files

- file framebuffer.cdl

  Framebuffer module with separate display and video sides.

4.71.1 Detailed Description

4.71.2 Modules

4.71.2.1 module framebuffer::framebuffer (clock *csr_clk,* clock *sram_clk,* clock *video_clk,* input bit *reset_n,* input t_bbc_display_sram_write *display_sram_write,* output t_video_bus *video_bus,* input t_csr_request *csr_request,* output t_csr_response *csr_response,* input bit *csr_select[16]*)

Parameters

in	*csr_clk*	Clock for CSR reads/writes
in	*sram_clk*	SRAM write clock, with frame buffer data
in	*video_clk*	Video clock, used to generate vsync, hsync, data out, etc
in	*csr_select*	This is a module that takes SRAM writes into a framebuffer, and includes a mapping to a dual-port SRAM (write on one side, read on the other), where the video side drives out vsync, hsync, data enable and pixel data.

The video side is asynchronous to the SRAM write side.

Video timing is handled by a *framebuffer_timing* module, which generates the synchronization signals and display enable. This module must be configure correctly for the display size and porches.

The video output start at a programmable base address in SRAM; moving down a line adds a programmable amount to the address in SRAM.

The module generates output pixel data from a shift register and a data buffer that fill from an internal dual-port SRAM, using the video timing.

The SRAM is filled with SRAM write requests, using a different clock to the video generation.

The current implementation is 1bpp RGB, with 16 pixels per SRAM word Bottom 16 SRAM data bits [16; 0] are red, bit[15] leftmost Next 16 SRAM data bits [16; 16] are green, bit[31] leftmost Top 16 SRAM data bits [16; 132] are blue, bit[47] leftmost

4.72 framebuffer_teletext

Files

- file framebuffer_teletext.cdl

  Teletext framebuffer module with separate write and video sides.

4.72.1 Detailed Description

4.72.2 Modules

4.72.2.1 module framebuffer_teletext::framebuffer_teletext (clock *csr_clk,* clock *sram_clk,* clock *video_clk,* input bit *reset_n,* input **t_bbc_display_sram_write** *display_sram_write,* output **t_video_bus** *video_bus,* input bit *csr_select_in[16],* input **t_csr_request** *csr_request,* output **t_csr_response** *csr_response*)

Parameters

in	*csr_clk*	Clock for CSR reads/writes
in	*sram_clk*	SRAM write clock, with frame buffer data
in	*video_clk*	Video clock, used to generate vsync, hsync, data out, etc
out	*csr_response*	This module provides a teletext framebuffer of configurable size.

Each character is 12x10 pixels - horizontally smoothed, but not vertically.

This module incorporates a *framebuffer_timing* module which should be configured with the correct display size and porches; the csr_select used for this is 1 more than that used for this module itself.

The module includes a CSR target with two registers: framebuffer start address (register 0), and SRAM words per line (register 1). The words-per-line register should be initialized to the number of characters in a line (i.e. displayed pixels/12). The framebuffer start register can be used to have multiple framebuffers, or to provide for scrolling, or it can be held at 0.

4.73 framebuffer_timing

Files

- file framebuffer_timing.cdl

  Framebuffer timing module to create sync and display signals.

4.73.1 Detailed Description

4.73.2 Modules

4.73.2.1 module framebuffer_timing::framebuffer_timing (clock *csr_clk,* clock *video_clk,* input bit *reset_n,* output t_video_timing *video_timing,* input t_csr_request *csr_request,* output t_csr_response *csr_response,* input bit *csr_select[16]*)

This module generates v_sync, h_sync and display_enable for a framebuffer, using configurable timings. The synchronization signals are active for a single cycle, and v_sync is asserted simulationeously with h_sync for the start of a frame.

The timing is controlled by three registers: display size, horizontal porches, and vertical porches.

Each line has hsync, porch, display lines, porch; each frame similarly.

The module that uses this must manage the synchronization signals required for the particular video standard (the length and polarity of synchronization pulses are varied in standards); the timing module provides the ability to generate framebuffer addresses and pixels reliably, rather than the actual video output synchronization signals.

Display size register:

Bits	Meaning
6;26	0
10;16	Vertical lines of display -1
6;10	0
10;0	Horizontal pixels of display -1

Horizontal porches register:

Bits	Meaning
6;26	0
10;16	pixels after display prior to hsync -1
6;10	0
10;0	pixels including and after hsync -1 prior to display

Vertical porches register:

Bits	Meaning
6;26	0
10;16	lines after display prior to vsync -1

Bits	Meaning
6;10	0
10;0	lines including and after vsync -1 prior to display

Parameters

in	*csr_clk*	Clock for CSR reads/writes
in	*video_clk*	Video clock, used to generate vsync, hsync, data out, etc
in	*reset_n*	Active low reset
out	*video_timing*	Video timing outputs
in	*csr_request*	Pipelined CSR request interface to control the module
out	*csr_response*	Pipelined CSR response interface to control the module
in	*csr_select*	CSR select value to target this module on the CSR interface

4.74 saa5050

Files

- file saa5050.cdl

  CDL implementation of Mullard SAA5050.

4.74.1 Detailed Description

4.74.2 Modules

4.74.2.1 module saa5050::saa5050 (clock *clk_2MHz,* input bit *clk_1MHz_enable,* input bit *reset_n,* input bit *superimpose_n,* input bit *data_n,* input bit *data_in[7],* input bit *dlim,* input bit *glr,* input bit *dew,* input bit *crs,* input bit *bcs_n,* output bit *tlc_n,* input bit *lose,* input bit *de,* input bit *po,* output bit *red[6],* output bit *green[6],* output bit *blue[6],* output bit *blan,* input **t_bbc_micro_sram_request** *host_sram_request*)

This module instantiates the *teletext* module to provide a teletext decoder that is compatible with the SAA5050 as it is used in the BBC microcomputer (i.e. some features of the chip are not supported, such as superimpose).

Parameters

in	*clk_2MHz*	Supposedly 6MHz pixel clock (TR6), except we use 2MHz and deliver 3 pixels per tick; rising edge should be coincident with clk_1MHz edges
in	*clk_1MHz_enable*	Clock enable high for clk_2MHz when the SAA's 1MHz would normally tick
in	*reset_n*	Active low reset
in	*superimpose_n*	Not implemented
in	*data_n*	Serial data in, not implemented
in	*data_in*	Parallel character data in
in	*dlim*	Not implemented, clocks serial data in somehow
in	*glr*	General line reset, can be tied to hsync - assert once per line before data comes in
in	*dew*	Data entry window - used to determine flashing rate and resets the ROM decoders - can be tied to vsync
in	*crs*	Character rounding select - drive high on even interlace fields to enable use of rounded character data (kinda indicates 'half line')
in	*bcs_n*	Assert (low) to enable double-height characters (?)
out	*tlc_n*	Asserted (low) when double-height characters occur (?)
in	*lose*	Load output shift register enable - must be low before start of character data in a scanline, rising with (or one tick earlier?) the data; changes off falling F1, rising clk_1MHz
in	*de*	Display enable
in	*po*	Picture on
out	*red*	Red pixels out, 6 per 2MHz clock tick
out	*green*	Green pixels out, 6 per 2MHz clock tick
out	*blue*	Blue pixels out, 6 per 2MHz clock tick
out	*blan*	Not implemented
in	*host_sram_request*	Write only, writes on clk_2MHz rising, acknowledge must be handled by supermodule

4.75 teletext

Files

- file teletext.cdl

  CDL implementation of a teletext decoder.

4.75.1 Detailed Description

4.75.2 Modules

4.75.2.1 module teletext::teletext (clock *clk,* input bit *reset_n,* input t_teletext_character *character,* input t_teletext_timings *timings,* output t_teletext_rom_access *rom_access,* input bit *rom_data[45],* output t_teletext_pixels *pixels*)

This is an implementaion of the core of a presentation level 1.0 teletext decoder, for arbitrary sized teletext output displays.

The output is supplied at 12 pixels per clock (one character width) The input is a byte of per clock of character data.

The implementation does not currently support double width or double size characters - they are not presentation level 1.0 features.

Teletext characters are displayed from a 12x20 grid. The ROM characters have two background rows, and then are displayed with 2 background pixels on the left, and then 10 pixels from the ROM The ROM is actually 5x9, and it is doubled to 10x18.

The type of pixel doubling is controlled with the *timings* input. It can be pure doubling, or smoothed. Some outputs may not want to use the doubling, for which the best approach is to request only even scanlines (in the *timings*) and to not smoothe, and then to select alternate pixel color values from the output bus.

Doubling

Doubling without smoothing can be achieved be true doubling of pixels

A simple smoothing can be performed for a pixel depending on its NSEW neighbors:

```
  |NN|
  |NN|
WW|ab|EE
WW|cd|EE
  |SS|
  |SS|
```

a is filled if the pixel is filled itself, or if N&W

b is filled if the pixel is filled itself, or if N&E

c is filled if the pixel is filled itself, or if S&W

d is filled if the pixel is filled itself, or if S&E

Hence one would get:

```
|..|**|**|**|..|
|..|**|**|**|..|
|---------------
|**|..|..|**|**|
|**|..|..|**|**|
|---------------
|..|**|..|..|**|
|..|**|..|..|**|
```

smoothed to:

```
|..|**|**|**|..|
|.*|**|**|**|*.|
|---------------
|**|*.|.*|**|**|
|**|*.|..|**|**|
|---------------
|.*|**|..|.*|**|
|..|**|..|..|**|
```

Or, without intervening lines:

```
|..******..|
|..******..|
|**....****|
|**....****|
|..**....**|
|..**....**|
```

smoothed to:

```
|..******..|
|.********.|
|***..*****|
|***...****|
|.***...***|
|..**....**|
```

So for even scanlines ('a' and 'b') the smoother needs row n and row n-1.

a is set if n[x] or n[x-left]&(n-1)[x]

b is set if n[x] or n[x-right]&(n-1)[x]

For odd scanlines ('c' and 'd') the smoother needs row n and row n+1.

c is set if n[x] or n[x-left]&(n+1)[x]

d is set if n[x] or n[x-right]&(n+1)[x]

This method has the unfortunate impact of smoothing two crossing lines, such as a plus:

```
|....**....|      |....**....|
|....**....|      |....**....|
|....**....|      |....**....|
|....**....|      |...****...|
|**********|      |**********|
|**********|      |**********|
|....**....|      |...****...|
|....**....|      |....**....|
|....**....|      |....**....|
|....**....|      |....**....|
```

Hence a better smoothing can be performed for a pixel depending on all its neighbors:

```
|NW|NN|NE|
|  |NN|  |
|WW|ab|EE|
|WW|cd|EE|
|  |SS|  |
|SW|SS|SE|
```

a is filled if the pixel is filled itself, or if (N&W) but not NW

b is filled if the pixel is filled itself, or if (N&E) but not NE

c is filled if the pixel is filled itself, or if (S&W) but not SW

d is filled if the pixel is filled itself, or if (S&E) but not SE

Hence one would get:

```
|..|**|**|**|..|
|..|**|**|**|..|
|---------------
|**|..|..|**|**|
|**|..|..|**|**|
|---------------
|..|**|..|..|**|
|..|**|..|..|**|
```

smoothed to:

```
|..|**|**|**|..|
|.*|**|**|**|..|
|---------------
|**|*.|..|**|**|
|**|*.|..|**|**|
|---------------
|.*|**|..|..|**|
|..|**|..|..|**|
```

Or, without intervening lines:

```
|..*******..|
|..*******..|
|**.....****|
|**.....****|
|..**.....**|
|..**.....**|
```

smoothed to:

```
|..*******..|
|.********..|
|***...****|
|***...****|
|.***.....**|
|..***.....**|
```

So for even scanlines ('a' and 'b') the smoother needs row n and row n-1.

a is set if n[x] or (n[x-left]&(n-1)[x]) &$\sim$ (n-1)[x-left]

b is set if n[x] or (n[x-right]&(n-1)[x]) &$\sim$ (n-1)[x-right]

For odd scanlines ('c' and 'd') the smoother needs row n and row n+1.

c is set if n[x] or (n[x-left]&(n+1)[x]) &$\sim$ (n+1)[x-left]

d is set if n[x] or (n[x-right]&(n+1)[x]) &$\sim$ (n+1)[x-left]

Graphics

Graphics characters are 6 blobs on a 6x10 grid (contiguous, separated):

```
|000111| |.00.11|
|000111| |.00.11|
|000111| |......|
|222333| |.22.33|
|222333| |.22.33|
|222333| |.22.33|
|222333| |......|
|444555| |.44.55|
|444555| |.44.55|
|444555| |......|
```

Parameters

in	*clk*	Character clock
in	*reset_n*	Active low reset
in	*character*	Parallel character data in, with valid signal
in	*timings*	Timings for the scanline, row, etc
out	*rom_access*	Teletext ROM access, registered output
in	*rom_data*	Teletext ROM data, valid in cycle after rom_access
out	*pixels*	Output pixels, three clock ticks delayed from valid data in

Chapter 5

Header Files

5.1 cdl/inc/apb.h File Reference

Types for the APB bus.

5.1.1 Detailed Description

Types for the APB bus.

Header file for the types for an APB bus, but no modules

Data Structures

- struct t_apb_request
- struct t_apb_response
- struct t_apb_processor_response
- struct t_apb_processor_request
- struct t_apb_rom_request

5.1.2 Data Structure Documentation

5.1.2.1 struct t_apb_request

Data Fields

bit[32]	paddr	
bit	penable	
bit	psel	
bit[32]	pwdata	
bit	pwrite	

5.1.2.2 struct t_apb_response

Data Fields

bit	perr	
bit[32]	prdata	
bit	pready	

5.1.2.3 struct t_apb_processor_response

Data Fields

bit	acknowledge	
bit	rom_busy	

5.1.2.4 struct t_apb_processor_request

Data Fields

bit[16]	address	
bit	valid	

5.1.2.5 struct t_apb_rom_request

Data Fields

bit[16]	address	
bit	enable	

5.2 cdl/inc/apb_peripherals.h File Reference

Modules of various simple APB peripherals.

5.2.1 Detailed Description

Modules of various simple APB peripherals.

Header file for the modules for some very simple APB peripherals

5.2.2 Modules

5.2.2.1 module apb_master_axi (clock *aclk,* input bit *areset_n,* input **t_axi_request** *ar,* output bit *awready,* input **t_axi_request** *aw,* output bit *arready,* output bit *wready,* input **t_axi_write_data** *w,* input bit *bready,* output **t_axi_write_response** *b,* input bit *rready,* output **t_axi_read_response** *r,* output **t_apb_request** *apb_request,* input **t_apb_response** *apb_response*)

5.2.2.2 module apb_master_mux (clock *clk,* input bit *reset_n,* input **t_apb_request** *apb_request_0,* output **t_apb_response** *apb_response_0,* input **t_apb_request** *apb_request_1,* output **t_apb_response** *apb_response_1,* output **t_apb_request** *apb_request,* input **t_apb_response** *apb_response*)

Parameters

in	*clk*	System clock
in	*reset_n*	Active low reset
in	*apb_request_0*	APB request to master 0
out	*apb_response_0*	APB response to master 0
in	*apb_request_1*	APB request to master 1
out	*apb_response_1*	APB response to master 1
out	*apb_request*	APB request to targets
in	*apb_response*	APB response from targets

5.2.2.3 module apb_processor (clock *clk,* input bit *reset_n,* input **t_apb_processor_request** *apb_processor_request,* output **t_apb_processor_response** *apb_processor_response,* output **t_apb_request** *apb_request,* input **t_apb_response** *apb_response,* output **t_apb_rom_request** *rom_request,* input bit *rom_data[40]*)

Parameters

clk	Clock for the CSR interface; a superset of all targets clock
apb_request	Pipelined csr request interface output
apb_response	Pipelined csr request interface response

5.2.2.4 module apb_target_de1_cl_inputs (clock *clk,* input bit *reset_n,* input **t_apb_request** *apb_request,* output **t_apb_response** *apb_response,* input **t_de1_cl_user_inputs** *user_inputs*)

Parameters

clk	System clock
reset_n	Active low reset
apb_request	APB request
apb_response	APB response

5.2.2.5 module apb_target_dprintf (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* output t_dprintf_req_4 *dprintf_req,* input bit *dprintf_ack*)

Parameters

clk	System clock
reset_n	Active low reset
apb_request	APB request
apb_response	APB response

5.2.2.6 module apb_target_gpio (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* output bit *gpio_output[16],* output bit *gpio_output_enable[16],* input bit *gpio_input[16],* output bit *gpio_input_event*)

Parameters

clk	System clock
reset_n	Active low reset
apb_request	APB request
apb_response	APB response

5.2.2.7 module apb_target_led_ws2812 (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* input bit *divider_400ns_in[8],* output bit *led_chain*)

Parameters

clk	System clock
reset_n	Active low reset
apb_request	APB request
apb_response	APB response
divider_400ns↩ _in	Default value for divider_400ns

5.2.2.8 module apb_target_ps2_host (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* input t_ps2_pins *ps2_in,* output t_ps2_pins *ps2_out*)

Parameters

clk	System clock
reset_n	Active low reset
apb_request	APB request

Parameters

apb_response	APB response
ps2_in	Pin values from the outside
ps2_out	Pin values to drive - 1 means float high, 0 means pull low

5.2.2.9 module apb_target_rv_timer (clock *clk,* input bit *reset_n,* input **t_timer_control** *timer_control,* input **t_apb_request** *apb_request,* output **t_apb_response** *apb_response,* output **t_timer_value** *timer_value*)

Parameters

clk	System clock
reset_n	Active low reset
timer_control	Control of the timer
apb_request	APB request
apb_response	APB response

5.2.2.10 module apb_target_sram_interface (clock *clk,* input bit *reset_n,* input **t_apb_request** *apb_request,* output **t_apb_response** *apb_response,* output bit *sram_ctrl[32],* output **t_sram_access_req** *sram_access_req,* input **t_sram_access_resp** *sram_access_resp*)

Parameters

clk	System clock
reset_n	Active low reset
apb_request	APB request
apb_response	APB response
sram_ctrl	SRAM control for whatever purpose
sram_access_req	SRAM access request
sram_access_resp	SRAM access response

5.2.2.11 module apb_target_timer (clock *clk,* input bit *reset_n,* input **t_apb_request** *apb_request,* output **t_apb_response** *apb_response,* output bit *timer_equalled[3]*)

Parameters

clk	System clock
reset_n	Active low reset
apb_request	APB request
apb_response	APB response

5.3 cdl/inc/axi.h File Reference

Types for the AXI bus.

5.3.1 Detailed Description

Types for the AXI bus.

Header file for the types for an AXI bus, but no modules

Data Structures

- struct t_axi_request
- struct t_axi_write_data
- struct t_axi_write_response
- struct t_axi_read_response

Enumerations

- enum t_axi_burst {
  axi_burst_fixed = 0,
  axi_burst_incr = 1,
  axi_burst_wrap = 2,
  axi_burst_reserved = 3 }
- enum t_axi_size {
  axi_size_1 = 0,
  axi_size_2 = 1,
  axi_size_4 = 2,
  axi_size_8 = 3,
  axi_size_16 = 4,
  axi_size_32 = 5,
  axi_size_64 = 6,
  axi_size_128 = 7 }
- enum t_axi_resp {
  axi_resp_okay = 0,
  axi_resp_exokay = 1,
  axi_resp_slverr = 2,
  axi_resp_decerr = 3 }

5.3.2 Data Structure Documentation

5.3.2.1 struct t_axi_request

Data Fields

bit[32]	addr	
t_axi_burst	burst	
bit[4]	cache	
bit[12]	id	
bit[4]	len	
bit[2]	lock	
bit[3]	prot	
bit[4]	qos	
bit[4]	region	
t_axi_size	size	
bit[4]	user	
bit	valid	

5.3.2.2 struct t_axi_write_data

Data Fields

bit[32]	data	
bit[12]	id	
bit	last	
bit[4]	strb	
bit[4]	user	
bit	valid	

5.3.2.3 struct t_axi_write_response

Data Fields

bit[12]	id	
t_axi_resp	resp	
bit[4]	user	
bit	valid	

5.3.2.4 struct t_axi_read_response

This structure is used to store read response

Data Fields

bit[32]	data	
bit[12]	id	
bit	last	
t_axi_resp	resp	
bit[4]	user	
bit	valid	

5.3.3 Enumeration Type Documentation

5.3.3.1 enum t_axi_burst

Enumerator

axi_burst_fixed

axi_burst_incr

axi_burst_wrap

axi_burst_reserved

5.3.3.2 enum t_axi_resp

Enumerator

axi_resp_okay

axi_resp_exokay

axi_resp_slverr

axi_resp_decerr

5.3.3.3 enum t_axi_size

Enumerator

axi_size_1

axi_size_2

axi_size_4

axi_size_8

axi_size_16

axi_size_32

axi_size_64

axi_size_128

5.3.4 Modules

5.3.4.1 module axi_master (clock *aclk,* input bit *areset_n,* output t_axi_request *ar,* input bit *awready,* output t_axi_request *aw,* input bit *arready,* input bit *wready,* output t_axi_write_data *w,* output bit *bready,* input t_axi_write_response *b,* output bit *rready,* input t_axi_read_response *r*)

5.4 cdl/inc/bbc_micro_types.h File Reference

BBC micro types header file for CDL.

5.4.1 Detailed Description

BBC micro types header file for CDL.

Header file for the types shared by more than one CDL module for the BBC micro implementation

Data Structures

- struct t_bbc_keyboard
- struct t_bbc_display
- struct t_bbc_display_sram_write
- struct t_bbc_floppy_sector_id
- struct t_bbc_floppy_op
- struct t_bbc_floppy_response
- struct t_bbc_floppy_sram_request
- struct t_bbc_floppy_sram_response
- struct t_bbc_clock_control
- struct t_bbc_clock_status
- struct t_bbc_micro_sram_request
- struct t_bbc_micro_sram_response

Enumerations

- enum t_bbc_pixels_per_clock {
  bbc_ppc_1,
  bbc_ppc_2,
  bbc_ppc_4,
  bbc_ppc_6,
  bbc_ppc_8 }
- enum t_bbc_csr_select {
  bbc_csr_select_clocks = 0,
  bbc_csr_select_display = 1,
  bbc_csr_select_floppy = 2,
  bbc_csr_select_keyboard = 3,
  bbc_csr_select_framebuffer = 4 }
- enum t_bbc_sram_select {
  bbc_sram_select_micro = 0,
  bbc_sram_select_display = 1,
  bbc_sram_select_floppy = 2,
  bbc_sram_select_cpu = 16,
  bbc_sram_select_cpu_ram_0 = 16,
  bbc_sram_select_cpu_ram_1 = 17,
  bbc_sram_select_cpu_os = 18,
  bbc_sram_select_cpu_teletext = 20,
  bbc_sram_select_cpu_rom_0 = 24,
  bbc_sram_select_cpu_rom_1 = 25,
  bbc_sram_select_cpu_rom_2 = 26,
  bbc_sram_select_cpu_rom_3 = 27 }

5.4.2 Data Structure Documentation

5.4.2.1 struct t_bbc_keyboard

The BBC keyboard consists of a keyboard matrix with ten columns of eight rows of keys. The columns can be individually powered, and then the eight rows can be read as a byte to see which of the column's keys is pressed. There is an additional 'Break' key that is independent of the keyboard matrix, that provides a reset signal to the motherboard.

This structure is used to pass the keyboard state in to the BBC micro implementation - since an ASIC of FPGA does not contain a physical keyboard, the key pressed information needs to be conveyed over a bus from outside. This structure permits this.

Data Fields

bit[64]	keys_down_cols_0_to↩ _7	
bit[16]	keys_down_cols_8_to↩ _9	
bit	reset_pressed	

5.4.2.2 struct t_bbc_display

The BBC micro output from its video ULA is separate red, green and blue pixel data, and sync signals. This structure conveys this information out of the BBC implementation, plus the number of pixels per clock, so that the display interface may be clocked at 2MHz. For modes where there are fewer than 8 pixels per clock, the red, green and blue data is replicated throughout the bus - so the only real need for pixels_per_clock is to indicate if the pixel clock rate is 12MHz or 16MHz (it is 12MHz if bbc_ppc_6)

Data Fields

bit[8]	blue	
bit	clock_enable	
bit[8]	green	
bit	hsync	
t_bbc_pixels_per_clock	pixels_per_clock	
bit[8]	red	
bit	vsync	

5.4.2.3 struct t_bbc_display_sram_write

To ease implementation of display framebuffers in target hardware there is a CDL module supplied called 'bbc_↩ display_sram'. This module converts from a t_bbc_display structure to a 3bpp frame buffer (RGB per pixel). The output from this module is therefore a stream of SRAM write transactions, each of 16 pixels.

The module itself is configured through a CSR request interface to set the base address of the frame buffer (amongst other things).

This bus does not have an equivalent 'response' bus; there is no way to back-pressure the BBC video subsystem, hence no way to back-pressure the display SRAM writes.

Data Fields

bit[16]	address	
bit[48]	data	
bit	enable	

5.4.2.4 struct t_bbc_floppy_sector_id

This structure is used in the request and response to a floppy drive from the FDC (floppy disc controller), for the ID read/written to a sector.

Each sector on a floppy has a descriptor that includes byte fields for the head, logical sector number, and the head/sector length and, and a CRC - and the sector data has its own CRC.

This structure fits into 32 bits, so a 32-bit wide SRAM can store this data.

Data Fields

bit	bad_crc	
bit	bad_data_crc	
bit	deleted_data	
bit	head	
bit[2]	sector_length	
bit[6]	sector_number	
bit[7]	track	

5.4.2.5 struct t_bbc_floppy_op

The floppy op structure is used to convey a floppy operation from the FDC to the floppy drive; it is effectively an internal set of signals that are driven inside the FDC to the floppy controller, which converts them to analog data or other control signals to the floppy drive interface.

The structure has no 'valid' signal - it is valid on every clock tick. However, control signals are required to toggle on and toggle off - it is the 'rising edge' of step_out, step_in, next_id, read_data_enable, etc that cause those to occur.

step_out and step_in are mutually exclusive; step_out moves the head out towards the outer rim of the disc, which is where track 0 is.

next_id is asserted if the drive should read the next sector ID (in reality waiting for the disc to spin round until a sector id descriptors is decoded from the surface) from the disc. In response to this, some time later, a floppy response with a valid sector_id should be presented.

read_data_enable is asserted if the next word (32 bits) of sector data should be read from the disc surface. This should only be asserted after a 'next_id', or after a previous 'read_data_enable'. After a 'next_id' it causes the first data word of the sector for which the sector id was returned; otherwise it continues data from that sector.

write_data_enable and write_data are not currently used. They should be used to write the data after a 'next_id' has been asserted, at 32 bits per write.

write_sector_id_enable and sector_id are not currently used. They should be used to write the sector id data for a sector. This is generally done on a floppy disc controller only when formatting a track, and so in fact may never be implemented (if formatting is assumed to be hard as opposed to soft).

Data Fields

bit	next_id	
bit	read_data_enable	
t_bbc_floppy_sector_id	sector_id	
bit	step_in	
bit	step_out	
bit[32]	write_data	
bit	write_data_enable	
bit	write_sector_id_enable	

5.4.2.6 struct t_bbc_floppy_response

The floppy response structure conveys data back from the floppy drive interface to the FDC in response to the floppy operation.

sector_id_valid is asserted for a single clock tick in conjunction with valid sector_id data in response to a 'next_id' rising edge in the floppy operation; this may occur any number of clock ticks after the request, and in the intervening period no other requests are permitted.

read_data_valid is asserted for a single clock tick in conjunction with valid read_data in response to a 'read_data_enable' floppy operation; this may occur any number of clock ticks after the request, and in the intervening period no other requests are permitted.

index is asserted if the latest sector_id is the first physical sector of the track - i.e. if the 'index hole' on the floppy disc is at that point. On a real floppy disc the index hole need not be anywhere near an actual valid sector data field, but for the emulation the index value is valid for the whole of the period from one sector_id_valid to the next.

track_zero is asserted if the current track is track zero. This becomes asserted when the drive is 'stepped out' to the outermost track (i.e. the physical track number is decremented to 0).

disk_ready is asserted if there is a floppy in the drive.

write_protect is asserted if the floppy in the drive has a write protect tab on it.

Data Fields

bit	disk_ready	
bit	index	
bit[32]	read_data	
bit	read_data_valid	
t_bbc_floppy_sector_id	sector_id	
bit	sector_id_valid	
bit	track_zero	
bit	write_protect	

5.4.2.7 struct t_bbc_floppy_sram_request

To implement the floppy drive there is a CDL implementation which takes floppy operations and converts them to SRAM reads (and writes); this is a standard SRAM access request interface.

Data Fields

bit[20]	address	
bit	enable	
bit	read_not_write	
bit[32]	write_data	

5.4.2.8 struct t_bbc_floppy_sram_response

The CDL implementation for the floppy drive uses this as a response

- ack asserts to acknowledge a read or write request, and valid read data is returned with data_valid.

Data Fields

bit	ack	
bit[32]	read_data	
bit	read_data_valid	

5.4.2.9 struct t_bbc_clock_control

This structure conveys clock gating and reset information to the BBC micro CDL implementation and various peripherals and other logic. Other modules require it to determine when to clock: for example, the floppy disc controller clocks on the CPU clock, so the interface from this module to its SRAM also clocks at the same edges (i.e. clk gated by enable_cpu).

Data Fields

bit[4]	debug	
bit	enable_1MHz_falling	Asserted if the rising edge of 'clk' should also be a falling '1MHz' clock edge
bit	enable_1MHz_rising	Asserted if the rising edge of 'clk' should also be a rising '1MHz' clock edge
bit	enable_2MHz_video	Asserted if the rising edge of 'clk' should also be a rising video '2MHz' clock edge
bit	enable_cpu	Asserted if the rising edge of 'clk' should also be a rising CPU clock edge
bit[2]	phi	Phase of BBC 6502 clock operation - in a real BBC micro this comes from the CPU
bit	reset_cpu	Asserted if the CPU should be reset, controlled by a CSR register
bit	will_enable_2MHz_video	Asserted if 'enable_2MHz_video' will be asserted in the next 'clk' period

5.4.2.10 struct t_bbc_clock_status

This structure conveys information in to the clock control module from the BBC micro - the real BBC micro has complex management of the CPU and hence system bus clock based on whether a 1MHz peripheral I/O space is being accessed or not.

Data Fields

bit	cpu_1MHz_access	Asserted by the BBC micro if a 1MHz peripheral is being accessed - this the CPU clock enables to align with the 1MHz clock enables

5.4.2.11 struct t_bbc_micro_sram_request

This structure is used to enable writing and reading any SRAM within a CDL implementation; it is a bus that can be pipelined arbitrarily (both in request and response), and it may be split amongst multiple targets (hence it can be set up as a pipelined tree, with the master at the root).

The protocol is for the master to assert valid with the required request on the bus. The master must wait for an 'ack' from a target to reach it, when it may then remove the 'valid' (for at least one cycle). If the request has been a read, then the master must also wait for 'read_data_valid' - which may occur in the same cycle as the 'ack'.

Before issuing another SRAM transaction the master must wait for 'ack' to go low.

A target receiving a valid request should compare the 'select' lines with the SRAMs that it services, and assert 'ack' if it can handle the request. It then performs the transaction, and returns any read data with the 'read_data_valid' signal asserted. In every cycle that it does not have valid read_data the read_data and read_data_valid must be 0.

The target may wait for valid to be deasserted before deasserting 'ack' (if it had been the selected target).

Data Fields

bit[24]	address	Constant during 'valid', indicates address in SRAM should be accessed.
bit	read_enable	Constant during 'valid', indicates if a read access is required. Exclusive with write_enable
bit[8]	select	Constant during 'valid', indicates which SRAM should be accessed. Usually one of t_bbc_sram_select
bit	valid	Asserted to indicate that an SRAM request is valid
bit[64]	write_data	Constant during 'valid', contains data to be written to SRAM (if write_enable is asserted) - ignored otherwise.
bit	write_enable	Constant during 'valid', indicates if a write access is required. Exclusive with read_enable

5.4.2.12 struct t_bbc_micro_sram_response

This structure conveys back towards the host the acknowledgement and any SRAM read data in response to a BBC micro SRAM read/write request.

Data Fields

bit	ack	Asserted to indicate that a SRAM request has been taken - held high until valid is deasserted

Data Fields

bit[64]	read_data	Read data from an SRAM request, valid with read_data_valid, zero in all other cycles
bit	read_data_valid	Asserted when the read data from an SRAM request is valid

5.4.3 Enumeration Type Documentation

5.4.3.1 enum t_bbc_csr_select

This enumeration matches the C, and it is used to select the CSR target (the 'select' field of csr_request's).

Enumerator

bbc_csr_select_clocks

bbc_csr_select_display

bbc_csr_select_floppy

bbc_csr_select_keyboard

bbc_csr_select_framebuffer

5.4.3.2 enum t_bbc_pixels_per_clock

The BBC micro operates with a variable speed pixel clock - it can be 12MHz or 16MHz. Furthermore, for some graphics 'modes' the number of real pixels per clock tick drops as pixels are replicated, to enable pixel information to be used for color selection. Hence 8 pixel per clock at 2MHz is 2 colors for 16Mpps, whereas 2 pixels per clock at 2MHz indicates 16Mpps where each pixel is replicated 4 times over, and can be of 2^4=16 different colors. Mode 2 uses bbc_ppc_2; modes 1 and 5 use bbc_ppc_4; modes 0, 3, 4 and 6 use bbc_ppc_8. Note that modes 0-3 run with 640 base pixels at 16MHz, hence 40us of pixel data per row.

For teletext mode the pixel rate is officially 12Mpps, as the teletext characters are 12 pixels wide and there are 40 characters per screen (hence roughly 480 pixels wide, and at 12Mpps that is 40us).

Enumerator

bbc_ppc_1

bbc_ppc_2

bbc_ppc_4

bbc_ppc_6

bbc_ppc_8

5.4.3.3 enum t_bbc_sram_select

This enumeration matches the C, and it is used to select the SRAM target for host SRAM transactions

Enumerator

bbc_sram_select_micro

bbc_sram_select_display

bbc_sram_select_floppy

bbc_sram_select_cpu

bbc_sram_select_cpu_ram_0

bbc_sram_select_cpu_ram_1

bbc_sram_select_cpu_os

bbc_sram_select_cpu_teletext

bbc_sram_select_cpu_rom_0

bbc_sram_select_cpu_rom_1

bbc_sram_select_cpu_rom_2

bbc_sram_select_cpu_rom_3

5.5 cdl/inc/bbc_submodules.h File Reference

BBC micro CDL submodules.

5.5.1 Detailed Description

BBC micro CDL submodules.

Header file for the modules required for the BBC micro CDL implementation. It should probably be tidied up to put the SRAMs separately, and to start to pull together a cpu/peripheral CDL module header file of its own. But it will do for now

5.5.2 Modules

5.5.2.1 module acia6850 (clock *clk,* input bit *reset_n,* input bit *read_not_write,* input bit *chip_select[2],* input bit *chip_select_n,* input bit *address,* input bit *data_in[8],* output bit *data_out[8],* output bit *irq_n,* input bit *tx_clk,* input bit *rx_clk,* output bit *txd,* input bit *cts,* input bit *rxd,* output bit *rts,* input bit *dcd*)

Parameters

clk	Clock that rises when the 'enable' of the 6850 completes - but a real clock for this model
read_not_write	Indicates a read transaction if asserted and chip selected
chip_select	Active high chip select
chip_select_n	Active low chip select
address	Changes during phase 1 (phi[0] high) with address to read or write
data_in	Data in (from CPU)
data_out	Read data out (to CPU)
irq_n	Active low interrupt
tx_clk	Clock used for transmit data - must be really about at most quarter the speed of clk
rx_clk	Clock used for receive data - must be really about at most quarter the speed of clk

5.5.2.2 module bbc_display (clock *clk,* input **t_bbc_display_sram_write** *display_sram_write,* input **t_bbc_floppy_sram_request** *floppy_sram_request,* output **t_bbc_keyboard** *keyboard,* output bit *reset_n,* output **t_bbc_floppy_sram_response** *floppy_sram_response*)

Parameters

clk	Clock running at 2MHz

5.5.2.3 module bbc_display_sram (clock *clk,* input bit *reset_n,* input **t_bbc_display** *display,* output **t_bbc_display_sram_write** *sram_write,* input **t_csr_request** *csr_request,* output **t_csr_response** *csr_response*)

Parameters

clk	Clock running at 2MHz

5.5.2.4 module bbc_floppy_sram (clock *clk,* input bit *reset_n,* input **t_bbc_floppy_op** *floppy_op,* output **t_bbc_floppy_response** *floppy_response,* output **t_bbc_floppy_sram_request** *sram_request,* input **t_bbc_floppy_sram_response** *sram_response,* input **t_csr_request** *csr_request,* output **t_csr_response** *csr_response*)

Parameters

clk	Clock running at 2MHz

5.5.2.5 module bbc_keyboard_csr (clock *clk,* input bit *reset_n,* output **t_bbc_keyboard** *keyboard,* input bit *keyboard_reset_n,* input **t_csr_request** *csr_request,* output **t_csr_response** *csr_response*)

Parameters

clk	Clock running at 2MHz

5.5.2.6 module bbc_keyboard_ps2 (clock *clk,* input bit *reset_n,* input **t_ps2_key_state** *ps2_key,* output **t_bbc_keyboard** *keyboard*)

Parameters

clk	Clock of PS2 keyboard

5.5.2.7 module bbc_micro (clock *clk,* input t_bbc_clock_control *clock_control,* output t_bbc_clock_status *clock_status,* input bit *reset_n,* input t_bbc_keyboard *keyboard,* output t_bbc_display *display,* output bit *keyboard_reset_n,* output t_bbc_floppy_op *floppy_op,* input t_bbc_floppy_response *floppy_response,* input t_bbc_micro_sram_request *host_sram_request,* output t_bbc_micro_sram_response *host_sram_response*)

Parameters

clk	Clock at least at '4MHz' - CPU runs at least half of this

5.5.2.8 module bbc_micro_clocking (clock *clk,* input bit *reset_n,* input t_bbc_clock_status *clock_status,* output t_bbc_clock_control *clock_control,* input t_csr_request *csr_request,* output t_csr_response *csr_response*)

Parameters

clk	4MHz clock in as a minimum

5.5.2.9 module bbc_micro_keyboard (clock *clk,* input bit *reset_n,* output bit *reset_out_n,* input bit *keyboard_enable_n,* input bit *column_select[4],* input bit *row_select[3],* output bit *key_in_column_pressed,* output bit *selected_key_pressed,* input t_bbc_keyboard *bbc_keyboard*)

Parameters

reset_out_n	From the Break key
keyboard_enable_n	Asserted to make keyboard detection operate
column_select	Wired to pa[4;0], and indicates which column of the keyboard matrix to access
row_select	Wired to pa[3;4], and indicates which row of the keyboard matrix to access
key_in_column_pressed	Wired to CA2, asserted if keyboard_enable_n and a key is pressed in the specified column (other than row 0)
selected_key_pressed	Asserted if keyboard_enable_n is asserted and the selected key is pressed

5.5.2.10 module bbc_micro_rams (clock *clk,* input bit *reset_n,* input t_bbc_clock_control *clock_control,* input t_bbc_micro_sram_request *host_sram_request,* output t_bbc_micro_sram_response *host_sram_response,* input t_bbc_display_sram_write *display_sram_write,* input t_bbc_floppy_sram_request *floppy_sram_request,* output t_bbc_floppy_sram_response *floppy_sram_response,* output t_bbc_micro_sram_request *bbc_micro_host_sram_request,* input t_bbc_micro_sram_response *bbc_micro_host_sram_response*)

Parameters

clk	4MHz clock in as a minimum

5.5.2.11 module bbc_vidproc (clock *clk_cpu,* clock *clk_2MHz_video,* input bit *reset_n,* input bit *chip_select_n,* input bit *address,* input bit *cpu_data_in[8],* input bit *pixel_data_in[8],* input bit *disen,* input bit *invert_n,* input bit *cursor,* input bit *saa5050_red[6],* input bit *saa5050_green[6],* input bit *saa5050_blue[6],* output bit *crtc_clock_enable,* output bit *red[8],* output bit *green[8],* output bit *blue[8],* output **t_bbc_pixels_per_clock** *pixels_valid_per_clock*)

Parameters

in	*clk_cpu*	Output on real chip in a sense (2MHz out somewhat)
in	*clk_2MHz_video*	Output on real chip, 2MHz video clock
in	*reset_n*	Not present on the chip, but required for the model - power up reset
in	*chip_select_n*	Active low chip select
in	*address*	Valid with chip select
in	*cpu_data_in*	Data in (from CPU) - was combined with pixel_data_in in BBC micro to save pins
in	*pixel_data_in*	Data in (from RAM) - was combined with cpu_data_in in BBC micro to save pins
in	*disen*	Asserted by CRTC if black output required (e.g. during sync)
in	*invert_n*	Asserted (low) if the output should be inverted (post-disen probably)
in	*cursor*	Asserted for first character of a cursor
in	*saa5050_red*	3 pixels in at 2MHz, red component, from teletext
in	*saa5050_green*	3 pixels in at 2MHz, green component, from teletext
in	*saa5050_blue*	3 pixels out at 2MHz, blue component, from teletext
out	*crtc_clock_enable*	High for 2MHz, toggles for 1MHz - the 'character clock' - used also to determine when the shift register is loaded
out	*red*	8 pixels out at 2MHz, red component
out	*green*	8 pixels out at 2MHz, green component
out	*blue*	8 pixels out at 2MHz, blue component

5.5.2.12 module cpu6502 (clock *clk,* input bit *reset_n,* input bit *ready,* input bit *irq_n,* input bit *nmi_n,* output bit *ba,* output bit *address[16],* output bit *read_not_write,* output bit *data_out[8],* input bit *data_in[8]*)

Parameters

clk	Clock, rising edge is start of phi1, end of phi2 - the phi1/phi2 boundary is not required
ready	Stops processor during current instruction. Does not stop a write phase. Address bus reflects current address being read. Stops the phase 2 from happening.
irq_n	Active low interrupt in
nmi_n	Active low non-maskable interrupt in
ba	Goes high during phase 2 if ready was low in phase 1 if read_not_write is 1, to permit someone else to use the memory bus
address	In real 6502, changes during phi 1 with address to read or write
read_not_write	In real 6502, changes during phi 1 with whether to read or write
data_out	In real 6502, valid at end of phi2 with data to write
data_in	Captured at the end of phi2 (rising clock in here)

5.5.2.13 module crtc6845 (clock *clk_2MHz,* clock *clk_1MHz,* input bit *reset_n,* output bit *ma[14],* output bit *ra[5],* input bit *read_not_write,* input bit *chip_select_n,* input bit *rs,* input bit *data_in[8],* output bit *data_out[8],* input bit *lpstb_n,* input bit *crtc_clock_enable,* output bit *de,* output bit *cursor,* output bit *hsync,* output bit *vsync*)

Parameters

clk_1MHz	Clock that rises when the 'enable' of the 6845 completes - but a real clock for this model
ma	Memory address
ra	Row address
read_not_write	Indicates a read transaction if asserted and chip selected
chip_select_n	Active low chip select
rs	Register select - address line really
data_in	Data in (from CPU)
data_out	Data out (to CPU)
lpstb_n	Light pen strobe
crtc_clock_enable	Not on the real chip - really CLK - the character clock - but this is an enable for clk_2MHz

5.5.2.14 module fdc8271 (clock *clk,* input bit *reset_n,* input bit *chip_select_n,* input bit *read_n,* input bit *write_n,* input bit *address[2],* input bit *data_in[8],* output bit *data_out[8],* output bit *irq_n,* output bit *data_req,* input bit *data_ack_n,* output bit *select[2],* input bit *ready[2],* output bit *fault_reset,* output bit *write_enable,* output bit *seek_step,* output bit *direction,* output bit *load_head,* output bit *low_current,* input bit *track_0_n,* input bit *write_protect_n,* input bit *index_n,* output **t_bbc_floppy_op *bbc_floppy_op,* input **t_bbc_floppy_response** *bbc_floppy_response*)**

Parameters

reset_n	8271 has an active high reset, but...
chip_select_n	Active low chip select
read_n	Indicates a read transaction if asserted and chip selected
write_n	Indicates a write transaction if asserted and chip selected
address	Address of register being accessed
data_in	Data in (from CPU)
data_out	Read data out (to CPU)
irq_n	Was INT on the 8271, but that means something else now; active low interrupt
select	drive select
ready	drive ready
write_enable	High if the drive should write data
seek_step	High if the drive should step
direction	Direction of step
load_head	Enable drive head
low_current	Asserted for track>=43
track_0_n	Asserted low if the selected drive is on track 0
write_protect_n	Asserted low if the selected drive is write-protected
index_n	Asserted low if the selected drive photodiode indicates start of track
bbc_floppy_op	Model drive operation, including write data
bbc_floppy_response	Parallel data read, specific to the model

5.5.2.15 module saa5050 (clock *clk_2MHz,* input bit *clk_1MHz_enable,* input bit *reset_n,* input bit *superimpose_n,* input bit *data_n,* input bit *data_in[7],* input bit *dlim,* input bit *glr,* input bit *dew,* input bit *crs,* input bit *bcs_n,* output bit *tlc_n,* input bit *lose,* input bit *de,* input bit *po,* output bit *red[6],* output bit *green[6],* output bit *blue[6],* output bit *blan,* input **t_bbc_micro_sram_request *host_sram_request*)**

Parameters

clk_2MHz	Supposedly 6MHz pixel clock (TR6), except we use 2MHz and deliver 3 pixels per tick; rising edge should be coincident with clk_1MHz edges
clk_1MHz_enable	Clock enable high for clk_2MHz when the SAA's 1MHz would normally tick
superimpose_n	Not implemented
data_n	Serial data in, not implemented
data_in	Parallel data in
dlim	clocks serial data in somehow (datasheet is dreadful...)
glr	General line reset - can be tied to hsync - assert once per line before data comes in
dew	Data entry window - used to determine flashing rate and resets the ROM decoders - can be tied to vsync
crs	Character rounding select - drive high on even interlace fields to enable use of rounded character data (kinda indicates 'half line')
bcs_n	Assert (low) to enable double-height characters (?)
tlc_n	Asserted (low) when double-height characters occur (?)
lose	Load output shift register enable - must be low before start of character data in a scanline, rising with (or one tick earlier?) the data; changes off falling F1, rising clk_1MHz
de	Display enable
po	Picture on
host_sram_request	Write only, writes on clk_2MHz rising, acknowledge must be handled by supermodule

5.5.2.16 module via6522 (clock *clk,* clock *clk_io,* input bit *reset_n,* input bit *read_not_write,* input bit *chip_select,* input bit *chip_select_n,* input bit *address[4],* input bit *data_in[8],* output bit *data_out[8],* output bit *irq_n,* input bit *ca1,* input bit *ca2_in,* output bit *ca2_out,* output bit *pa_out[8],* input bit *pa_in[8],* input bit *cb1,* input bit *cb2_in,* output bit *cb2_out,* output bit *pb_out[8],* input bit *pb_in[8]*)

Parameters

clk	1MHz clock rising when bus cycle finishes
clk_io	1MHz clock rising when I/O should be captured - can be antiphase to clk
read_not_write	Indicates a read transaction if asserted and chip selected
chip_select	Active high chip select
chip_select_n	Active low chip select
address	Changes during phase 1 (phi[0] high) with address to read or write
data_in	Data in (from CPU)
data_out	Read data out (to CPU)
irq_n	Active low interrupt
ca1	Port a control 1 in
ca2_in	Port a control 2 in
ca2_out	Port a control 2 out
pa_out	Port a data out
pa_in	Port a data in
cb1	Port b control 1 in
cb2_in	Port b control 2 in
cb2_out	Port b control 2 out
pb_out	Port b data out
pb_in	Port b data in

5.6 cdl/inc/csr_interface.h File Reference

Types and modules for the CSR interface.

5.6.1 Detailed Description

Types and modules for the CSR interface.

Copyright

(C) 2016-2017, Gavin J Stark. All rights reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/↩LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Header file for the types and modules in the pipelined CSR interface, including APB target to CSR master, CSR target to APB master, and CSR target to simple CSR access.

Data Structures

- struct t_csr_request
- struct t_csr_response
- struct t_csr_access

Typedefs

- typedef bit[32] t_csr_access_data

5.6.2 Data Structure Documentation

5.6.2.1 struct t_csr_request

This is the request structure for the pipelined CSR interface.

A valid request has *valid* asserted; this must remain asserted until *acknowledge* is seen in response; another request must not be driven until *acknowledge* is seen to be low.

A valid request has read_not_write (1 for read, 0 for write); select (a 16-bit field) and address (a 16-bit field).

For write requests the data is up to 64 bits - although many registers are shorter.

For read responses a valid request will return a *read_data_valid* signal with valid *read_data*.

This structure should be driven by:

csr_master_apb In response to an APB, this masters the CSR pipelined interface

This structure should terminate (as leaves) in one or more of:

csr_target_csr Provides a t_csr_access to a target

csr_target_apb Provides an APB interface to a target

csr_target_timeout Automatically times out transactions if the bus hangs for a while

Data Fields

bit[16]	address	
bit[32]	data	
bit	read_not_write	
bit[16]	select	
bit	valid	

5.6.2.2 struct t_csr_response

This is the response structure returning from a target on the CSR bus system back to the master. The 'ack' signal is asserted by a target from the point that the request is detected as valid and serviceable (i.e. a valid request with matching select) until the access is performed. The valid signal should be held high until an acknowledge is seen; it should then be taken low for at least one clock tick.

The CSR response from more than one target may be wire-ored together, and pipeline stages may be added as required for timing.

Data Fields

bit	acknowledge	
bit[32]	read_data	
bit	read_data_error	
bit	read_data_valid	

5.6.2.3 struct t_csr_access

To simplify design of CSR targets the 'csr_interface' module converts a t_csr_request/t_csr_response interface into this simple CSR access request interface. Doing this hides the complexity of the shared, pipelined CSR request/response bus from the targets, and ensures consistent operation of targets.

This access bus has signals that are valid for a single cycle. The access requested must be performed in that cycle. Read data for the access must be provided in the cycle of the request (combinatorially on 'address').

Data Fields

bit[16]	address	
bit[32]	data	
bit	read_not_write	
bit	valid	

5.6.3 Typedef Documentation

5.6.3.1 typedef bit [32] t_csr_access_data

This type conveys a response (in the same cycle as a valid CSR access request) to the csr_interface for a target using that module.

5.6.4 Modules

5.6.4.1 module csr_master_apb (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* input t_csr_response *csr_response,* output t_csr_request *csr_request*)

Parameters

clk	Clock for the CSR interface; a superset of all targets clock
reset_n	Active low reset
apb_request	APB request from master
apb_response	APB response to master
csr_response	Pipelined csr request interface response
csr_request	Pipelined csr request interface output

5.6.4.2 module csr_target_apb (clock *clk,* input bit *reset_n,* input t_csr_request *csr_request,* output t_csr_response *csr_response,* output t_apb_request *apb_request,* input t_apb_response *apb_response,* input bit *csr_select[16]*)

Parameters

in	*clk*	Clock for the CSR interface, possibly gated version of master CSR clock
in	*reset_n*	Active low reset
in	*csr_request*	Pipelined csr request interface input
out	*csr_response*	Pipelined csr request interface response
out	*apb_request*	APB request to target
in	*apb_response*	APB response from target
in	*csr_select*	Hard-wired select value for the client

5.6.4.3 module csr_target_csr (clock *clk,* input bit *reset_n,* input t_csr_request *csr_request,* output t_csr_response *csr_response,* output t_csr_access *csr_access,* input t_csr_access_data *csr_access_data,* input bit *csr_select[16]*)

Parameters

clk	Clock for the CSR interface, possibly gated version of master CSR clock
reset_n	Active low reset
csr_request	Pipelined csr request interface input
csr_response	Pipelined csr request interface response
csr_access	Registered CSR access request to client
csr_access_data	Read data valid combinatorially based on csr_access
csr_select	Hard-wired select value for the client

5.6.4.4 module csr_target_timeout (clock *clk,* input bit *reset_n,* input t_csr_request *csr_request,* output t_csr_response *csr_response,* input bit *csr_timeout[16]*)

Parameters

clk	Clock for the CSR interface, possibly gated version of master CSR clock

Parameters

reset_n	Active low reset
csr_request	Pipelined csr request interface input
csr_response	Pipelined csr request interface response
csr_timeout	Number of cycles to wait for until auto-acknowledging a request

5.7 cdl/inc/de1_cl.h File Reference

Input file for DE1 cl inputs and boards.

5.7.1 Detailed Description

Input file for DE1 cl inputs and boards.

Header file for the types and CDL modules for input devices

Header file for the types and CDL modules for input devices

From Martin Hinner's collection of data:

Timing values (display + (right border / front porch) + sync pulse + (back porch / left border))

640x480 @ 60Hz: 25.175MHz clk 640+16+ 96+48 = 800 480+10+2+33 = 525 -syncs 800x600 @ 60Hz: 40MHz clk 800+40+128+88 =1076 600+ 1+4+23 = 628 +syncs 1024x768 @ 60Hz: 65MHz clk 1024+24+136+160=1344 768+ 3+6+29 = 806 -syncs 1280x1024 @ 60Hz: 108MHz clk 1280+48+112+248=1688 1024+ 1+3+38 =1066 +syncs 1600x1200 @ 60Hz: 162MHz clk 1600+64+192+304=2160 1200+ 1+3+46 =1250 +syncs

Data Structures

- struct t_de1_cl_inputs_control
- struct t_rotary_motion_inputs
- struct t_de1_cl_inputs_status
- struct t_de1_cl_diamond
- struct t_de1_cl_joystick
- struct t_de1_cl_shift_register
- struct t_de1_cl_rotary
- struct t_de1_cl_user_inputs
- struct t_de1_cl_lcd
- struct t_de1_cl_shift_register_control
- struct t_de1_leds

5.7.2 Data Structure Documentation

5.7.2.1 struct t_de1_cl_inputs_control

Data Fields

bit	sr_clock	Not really a clock in the FPGA, but a signal toggled by the design
bit	sr_shift	Asserted high for rising sr_clock to shift the shift register; if low, the shift register is loaded from the pins

5.7.2.2 struct t_rotary_motion_inputs

Data Fields

bit	direction_pin	
bit	transition_pin	

5.7.2.3 struct t_de1_cl_inputs_status

Data Fields

t_rotary_motion_inputs	left_rotary	
t_rotary_motion_inputs	right_rotary	
bit	sr_data	Shift register output data

5.7.2.4 struct t_de1_cl_diamond

Data Fields

bit	a	
bit	b	
bit	x	
bit	y	

5.7.2.5 struct t_de1_cl_joystick

Data Fields

bit	c	
bit	d	
bit	l	
bit	r	
bit	u	

5.7.2.6 struct t_de1_cl_shift_register

Data Fields

bit	diall_click	
bit	dialr_click	
t_de1_cl_diamond	diamond	
t_de1_cl_joystick	joystick	
bit	temperature_alarm	
bit	touchpanel_irq	

5.7.2.7 struct t_de1_cl_rotary

Data Fields

bit	direction	
bit	direction_pulse	
bit	pressed	

5.7.2.8 struct t_de1_cl_user_inputs

Data Fields

t_de1_cl_diamond	diamond	
t_de1_cl_joystick	joystick	
t_de1_cl_rotary	left_dial	
t_de1_cl_rotary	right_dial	
bit	temperature_alarm	
bit	touchpanel_irq	
bit	updated_switches	Asserted if diamond, joystick, touchpanel_irq, temperature_alarm, and dial pressed bits have been updated

5.7.2.9 struct t_de1_cl_lcd

Data Fields

bit	backlight	
bit[6]	blue	

Data Fields

bit	display_enable	
bit[7]	green	
bit	hsync_n	
bit[6]	red	
bit	vsync_n	

5.7.2.10 struct t_de1_cl_shift_register_control

Data Fields

bit	sr_clock	
bit	sr_shift	

5.7.2.11 struct t_de1_leds

Data Fields

bit[7]	h0	
bit[7]	h1	
bit[7]	h2	
bit[7]	h3	
bit[7]	h4	
bit[7]	h5	
bit[10]	leds	

5.7.3 Modules

5.7.3.1 module bbc_micro_de1_cl_bbc (clock *clk,* clock *video_clk,* input bit *reset_n,* input bit *bbc_reset_n,* input bit *framebuffer_reset_n,* output **t_bbc_clock_control** *clock_control,* input **t_bbc_keyboard** *bbc_keyboard,* output **t_video_bus** *video_bus,* input **t_csr_request** *csr_request,* output **t_csr_response** *csr_response*)

Parameters

clk	50MHz clock from DE1 clock generator
video_clk	9MHz clock from PLL, derived from 50MHz
reset_n	hard reset from a pin - a key on DE1

5.7.3.2 module bbc_micro_de1_cl_io (clock *clk,* clock *video_clk,* input bit *reset_n,* input bit *bbc_reset_n,* input bit *framebuffer_reset_n,* input bit *keys[4],* input bit *switches[10],* input **t_bbc_clock_control** *clock_control,* output **t_bbc_keyboard** *bbc_keyboard,* output **t_video_bus** *video_bus,* output **t_csr_request** *csr_request,* input **t_csr_response** *csr_response,* input **t_ps2_pins** *ps2_in,* output **t_ps2_pins** *ps2_out,* input **t_de1_cl_inputs_status** *inputs_status,* output **t_de1_cl_inputs_control** *inputs_control,* output bit *leds[10],* output bit *lcd_source,* output bit *led_chain*)

Parameters

clk	50MHz clock from DE1 clock generator
video_clk	9MHz clock from PLL, derived from 50MHz
reset_n	hard reset from a pin - a key on DE1
ps2_in	PS2 input pins
ps2_out	PS2 output pin driver open collector
inputs_status	DE1 CL daughterboard shifter register etc status
inputs_control	DE1 CL daughterboard shifter register control

5.7.3.3 module de1_cl_controls (clock *clk,* input bit *reset_n,* output t_de1_cl_inputs_control *inputs_control,* input t_de1_cl_inputs_status *inputs_status,* output t_de1_cl_user_inputs *user_inputs,* input bit *sr_divider[8]*)

Parameters

in	*clk*	system clock - not the shift register pin, something faster
in	*reset_n*	async reset
out	*inputs_control*	Signals to the shift register etc on the DE1 CL daughterboard
in	*inputs_status*	Signals from the shift register, rotary encoders, etc on the DE1 CL daughterboard
out	*user_inputs*	
in	*sr_divider*	clock divider to control speed of shift register

5.8 cdl/inc/dprintf.h File Reference

Data Structures

- struct t_dprintf_req_4
- struct t_dprintf_req_2
- struct t_dprintf_resp
- struct t_dprintf_byte

5.8.1 Data Structure Documentation

5.8.1.1 struct t_dprintf_req_4

Data Fields

bit[16]	address	
bit[64]	data_0	
bit[64]	data_1	
bit[64]	data_2	
bit[64]	data_3	
bit	valid	

5.8.1.2 struct t_dprintf_req_2

Data Fields

bit[16]	address	
bit[64]	data_0	
bit[64]	data_1	
bit	valid	

5.8.1.3 struct t_dprintf_resp

Data Fields

bit	ack	

5.8.1.4 struct t_dprintf_byte

Data Fields

bit[16]	address	
bit[8]	data	
bit	valid	

5.9 cdl/inc/dprintf_modules.h File Reference

5.9.1 Modules

5.9.1.1 module dprintf (clock *clk,* input bit *reset_n,* input **t_dprintf_req_4** *dprintf_req,* output bit *dprintf_ack,* output **t_dprintf_byte** *dprintf_byte*)

Parameters

in	*clk*	Clock for data in and display SRAM write out
in	*dprintf_req*	Debug printf request
out	*dprintf_ack*	Debug printf acknowledge
out	*dprintf_byte*	Byte to output

5.9.1.2 module dprintf_2_mux (clock *clk,* input bit *reset_n,* input **t_dprintf_req_2** *req_a,* input **t_dprintf_req_2** *req_b,* output bit *ack_a,* output bit *ack_b,* output **t_dprintf_req_2** *req,* input bit *ack*)

5.9.1.3 module dprintf_4_mux (clock *clk,* input bit *reset_n,* input **t_dprintf_req_4** *req_a,* input **t_dprintf_req_4** *req_b,* output bit *ack_a,* output bit *ack_b,* output **t_dprintf_req_4** *req,* input bit *ack*)

5.10 cdl/inc/framebuffer.h File Reference

Framebuffer CDL types and submodules.

5.10.1 Detailed Description

Framebuffer CDL types and submodules.

Copyright

Header file for framebuffer modules for VGA, LCD panel, both bitmapped and teletext.

Data Structures

- struct t_video_timing

5.10.2 Data Structure Documentation

5.10.2.1 struct t_video_timing

Data Fields

bit	display_enable	Asserted if pixels should be presented to the output (i.e. outside the front and back porches both horizontally and vertically)
bit	display_required	Asserted for scanlines being displayed, up to the end of the horizontal displayed area - permits prefetching of pixel data
bit	h_sync	Asserted for a single clock at the start of every scanline
bit	v_displaying	Asserted for a scanline if the scanline will display data
bit	v_frame_last_line	Asserted if
bit	v_sync	Asserted for the whole of the first scanline or a frame
bit	will_display_enable	Asserted if the next clock will have *display_enable* asserted
bit	will_h_sync	Asserted if the next clock will be an *h_sync*

5.10.3 Modules

5.10.3.1 module framebuffer (clock *csr_clk,* clock *sram_clk,* clock *video_clk,* input bit *reset_n,* input t_bbc_display_sram_write *display_sram_write,* output t_video_bus *video_bus,* input t_csr_request *csr_request,* output t_csr_response *csr_response,* input bit *csr_select[16]*)

Parameters

csr_clk	Clock for CSR reads/writes
sram_clk	SRAM write clock, with frame buffer data
video_clk	Video clock, used to generate vsync, hsync, data out, etc
csr_select	CSR select value to target this module on the CSR interface

5.10.3.2 module framebuffer_teletext (clock *csr_clk,* clock *sram_clk,* clock *video_clk,* input bit *reset_n,* input **t_bbc_display_sram_write** *display_sram_write,* output **t_video_bus** *video_bus,* input bit *csr_select_in[16],* input **t_csr_request** *csr_request,* output **t_csr_response** *csr_response*)

Parameters

csr_clk	Clock for CSR reads/writes
sram_clk	SRAM write clock, with frame buffer data
video_clk	Video clock, used to generate vsync, hsync, data out, etc
csr_select_in	Tie to zero for default

5.10.3.3 module framebuffer_timing (clock *csr_clk,* clock *video_clk,* input bit *reset_n,* output **t_video_timing** *video_timing,* input **t_csr_request** *csr_request,* output **t_csr_response** *csr_response,* input bit *csr_select[16]*)

Parameters

in	*csr_clk*	Clock for CSR reads/writes
in	*video_clk*	Video clock, used to generate vsync, hsync, data out, etc
in	*reset_n*	Active low reset
out	*video_timing*	Video timing outputs
in	*csr_request*	Pipelined CSR request interface to control the module
out	*csr_response*	Pipelined CSR response interface to control the module
in	*csr_select*	CSR select value to target this module on the CSR interface

5.11 cdl/inc/hps.h File Reference

5.11.1 Modules

5.11.1.1 module hps_fpga_generic (clock *clk,* input bit *reset_n,* clock *lw_axi_clock_clk,* input **t_axi_request** *lw_axi_ar,* output bit *lw_axi_arready,* input **t_axi_request** *lw_axi_aw,* output bit *lw_axi_awready,* output bit *lw_axi_wready,* input **t_axi_write_data** *lw_axi_w,* input bit *lw_axi_bready,* output **t_axi_write_response** *lw_axi_b,* input bit *lw_axi_rready,* output **t_axi_read_response** *lw_axi_r,* input **t_de1_cl_inputs_status** *de1_cl_inputs_status,* output **t_de1_cl_inputs_control** *de1_cl_inputs_control,* output bit *de1_cl_led_data_pin,* clock *de1_cl_lcd_clock,* input bit *de1_cl_lcd_reset_n,* output **t_de1_cl_lcd** *de1_cl_lcd,* output **t_de1_leds** *de1_leds,* input **t_ps2_pins** *de1_ps2_in,* output **t_ps2_pins** *de1_ps2_out,* input **t_ps2_pins** *de1_ps2b_in,* output **t_ps2_pins** *de1_ps2b_out,* clock *de1_vga_clock,* input bit *de1_vga_reset_n,* output **t_adv7123** *de1_vga,* input bit *de1_keys[4],* input bit *de1_switches[10],* input bit *de1_irda_rxd,* output bit *de1_irda_txd*)

5.12 cdl/inc/input_devices.h File Reference

Input device header file for CDL modules.

5.12.1 Detailed Description

Input device header file for CDL modules.

Header file for the types and CDL modules for input devices

Header file for the types and CDL modules for input devices

Data Structures

- struct t_ps2_pins
- struct t_ps2_rx_data
- struct t_ps2_key_state

5.12.2 Data Structure Documentation

5.12.2.1 struct t_ps2_pins

Data Fields

bit	clk	
bit	data	

5.12.2.2 struct t_ps2_rx_data

Data Fields

bit[8]	data	
bit	parity_error	
bit	protocol_error	
bit	timeout	
bit	valid	

5.12.2.3 struct t_ps2_key_state

Data Fields

bit	extended	
bit[8]	key_number	
bit	release	
bit	valid	

5.12.3 Modules

5.12.3.1 module ps2_host (clock *clk,* input bit *reset_n,* input **t_ps2_pins** *ps2_in,* output **t_ps2_pins** *ps2_out,* output **t_ps2_rx_data** *ps2_rx_data,* input bit *divider[16]*)

Parameters

in	*clk*	Clock
in	*ps2_in*	Pin values from the outside
out	*ps2_out*	Pin values to drive - 1 means float high, 0 means pull low

5.12.3.2 module ps2_host_keyboard (clock *clk,* input bit *reset_n,* input **t_ps2_rx_data** *ps2_rx_data,* output **t_ps2_key_state** *ps2_key*)

Parameters

clk	Clock

5.13 cdl/inc/jtag.h File Reference

Data Structures

- struct t_jtag

Enumerations

- enum t_jtag_action {
  action_idle,
  action_capture,
  action_shift,
  action_update }

5.13.1 Data Structure Documentation

5.13.1.1 struct t_jtag

Data Fields

bit	ntrst	
bit	tdi	
bit	tms	

5.13.2 Enumeration Type Documentation

5.13.2.1 enum t_jtag_action

Enumerator

action_idle

action_capture

action_shift

action_update

5.13.3 Modules

5.13.3.1 module jtag_apb (clock *jtag_tck,* input bit *reset_n,* input *bitir[5],* input t_jtag_action *dr_action,* input *bitdr_in[50],* output *bitdr_tdi_mask[50],* output *bitdr_out[50],* clock *apb_clock,* output t_apb_request *apb_request,* input t_apb_response *apb_response*)

5.13.3.2 module jtag_tap (clock *jtag_tck,* input bit *reset_n,* input t_jtag *jtag,* output bit *tdo,* output *bitir[5],* output t_jtag_action *dr_action,* output *bitdr_in[50],* input *bitdr_tdi_mask[50],* input *bitdr_out[50]*)

5.14 cdl/inc/leds.h File Reference

Constants, types and modules for various LED drivers.

5.14.1 Detailed Description

Constants, types and modules for various LED drivers.

Copyright

Header file for the types and modules controlling LEDs, including Neopixel chains.

Data Structures

- struct t_led_ws2812_data
- struct t_led_ws2812_request

Variables

- constant bit[16] led_seven_seg_hex_a = 16b_1101011111101101
- constant bit[16] led_seven_seg_hex_b = 16b_0010011110011111
- constant bit[16] led_seven_seg_hex_c = 16b_0010111111111011
- constant bit[16] led_seven_seg_hex_d = 16b_0111101101101101
- constant bit[16] led_seven_seg_hex_e = 16b_1111110101000101
- constant bit[16] led_seven_seg_hex_f = 16b_1101111101110001
- constant bit[16] led_seven_seg_hex_g = 16b_1110111101111100

5.14.2 Data Structure Documentation

5.14.2.1 struct t_led_ws2812_data

Data Fields

bit[8]	blue	The 8 bit blue component for the LED to display
bit[8]	green	The 8 bit green component for the LED to display
bit	last	Assert if the LED data is for the last LED in the chain
bit[8]	red	The 8 bit red component for the LED to display
bit	valid	Assert if the LED data supplied in this structure is valid

5.14.2.2 struct t_led_ws2812_request

Data Fields

bit	first	If requesting LED data, then the first LED of the stream should be provided; indicates *led_number* is 0
bit[8]	led_number	Number of LED data required, so that a client can use a switch statement or register file or array, for example
bit	ready	Active high signal indicating if LED data is required; ignore *ready* if the response has *valid* asserted

5.14.3 Modules

5.14.3.1 module led_seven_segment (input bit *hex[4],* output bit *leds[7]*)

Parameters

in	*hex*	Hexadecimal to display on 7-segment LED
out	*leds*	1 for LED on, 0 for LED off, for segments a-g in bits 0-7

5.14.3.2 module led_ws2812_chain (clock *clk,* input bit *reset_n,* input bit *divider_400ns[8],* output t_led_ws2812_request *led_request,* input t_led_ws2812_data *led_data,* output bit *led_chain*)

Parameters

clk	system clock - not the pin clock
reset_n	async reset
divider_400ns	clock divider value to provide for generating a pulse every 400ns based on clk
led_request	LED data request
led_data	LED data, for the requested led
led_chain	Data in pin for LED chain

5.14.4 Variable Documentation

5.14.4.1 constant bit [16] led_seven_seg_hex_a = 16b_1101011111101101

5.14.4.2 constant bit [16] led_seven_seg_hex_b = 16b_0010011110011111

5.14.4.3 constant bit [16] led_seven_seg_hex_c = 16b_0010111111111011

5.14.4.4 constant bit [16] led_seven_seg_hex_d = 16b_0111101101101101

5.14.4.5 constant bit [16] led_seven_seg_hex_e = 16b_1111110101000101

5.14.4.6 constant bit [16] led_seven_seg_hex_f = 16b_1101111101110001

5.14.4.7 constant bit [16] led_seven_seg_hex_g = 16b_1110111101111100

5.15 cdl/inc/picoriscv.h File Reference

Module that makes up a Pico-RISC-V microcomputer.

5.15.1 Detailed Description

Module that makes up a Pico-RISC-V microcomputer.

Copyright

5.15.2 Modules

5.15.2.1 module picoriscv (clock *clk,* input bit *reset_n,* clock *video_clk,* input bit *video_reset_n,* output t_video_bus *video_bus,* input t_prv_keyboard *keyboard,* input t_csr_request *csr_request,* output t_csr_response *csr_response*)

Parameters

in	*clk*	Clock, divided down for CPU
in	*reset_n*	Active low reset
in	*video_clk*	Video clock, independent of CPU clock
in	*video_reset_n*	Active low reset

5.16 cdl/inc/picoriscv_submodules.h File Reference

Modules that make up a Pico-RISC-V microcomputer.

5.16.1 Detailed Description

Modules that make up a Pico-RISC-V microcomputer.

Copyright

5.16.2 Modules

5.16.2.1 module picoriscv_clocking (clock *clk*, input bit *reset_n*, input t_prv_clock_status *clock_status*, output t_prv_mem_control *mem_control*, output t_prv_clock_control *clock_control*, input t_csr_request *csr_request*, output t_csr_response *csr_response*)

5.17 cdl/inc/picoriscv_types.h File Reference

Data Structures

- struct t_prv_mem_control
- struct t_prv_clock_control
- struct t_prv_clock_status
- struct t_prv_keyboard

Enumerations

- enum t_prv_csr_select { prv_csr_select_clocks =0 }

5.17.1 Data Structure Documentation

5.17.1.1 struct t_prv_mem_control

This structure conveys memory management to the Pico-RISC-V CDL implementation and various peripherals and other logic

Data Fields

bit	dmem_request	
bit	dmem_set_reg	
bit	ifetch_request	
bit	ifetch_set_reg	
bit	ifetch_use_reg	
bit	io_enable	

5.17.1.2 struct t_prv_clock_control

This structure conveys clock gating and reset information to the Pico-RISC-V CDL implementation and various peripherals and other logic

Data Fields

bit[4]	debug	
bit	riscv_clk_enable	

5.17.1.3 struct t_prv_clock_status

This structure conveys information in to the clock control module from the RISC-V.

Data Fields

bit	dmem_read_enable	
bit	dmem_write_enable	
bit	imem_request	
bit	io_ready	
bit	io_request	

5.17.1.4 struct t_prv_keyboard

Data Fields

bit[64]	keys_low	

5.17.2 Enumeration Type Documentation

5.17.2.1 enum t_prv_csr_select

Enumerator

prv_csr_select_clocks

5.18 cdl/inc/riscv.h File Reference

Header file for RISC-V implementations.

5.18.1 Detailed Description

Header file for RISC-V implementations.

Copyright

Data Structures

- struct t_riscv_mem_access_req
- struct t_riscv_mem_access_resp
- struct t_riscv_irqs
- struct t_riscv_fetch_req
- struct t_riscv_fetch_resp
- struct t_riscv_config
- struct t_riscv_debug_mst
- struct t_riscv_debug_tgt
- struct t_riscv_pipeline_debug_control
- struct t_riscv_pipeline_debug_response

Typedefs

- typedef bit[32] t_riscv_word
- typedef bit[2] t_riscv_fetch_tag
- typedef bit t_riscv_debug_resp

Enumerations

- enum t_riscv_mode {
  rv_mode_user = 3b000,
  rv_mode_supervisor = 3b001,
  rv_mode_machine = 3b011,
  rv_mode_debug = 3b111 }
- enum t_riscv_debug_op {
  rv_debug_set_requests,
  rv_debug_read,
  rv_debug_write,
  rv_debug_acknowledge,
  rv_debug_execute,
  rv_debug_execute_progbuf }

Variables

- constant integer RISCV_DATA_ADDR_WIDTH = 14
- constant integer RISCV_INSTR_ADDR_WIDTH = 14

5.18.2 Data Structure Documentation

5.18.2.1 struct t_riscv_mem_access_req

Data Fields

bit[32]	address	
bit[4]	byte_enable	
bit	read_enable	
bit[32]	write_data	
bit	write_enable	

5.18.2.2 struct t_riscv_mem_access_resp

Data Fields

bit[32]	read_data	Data returned from reading the requested address
bit	wait	Valid in the same cycle as read_data

5.18.2.3 struct t_riscv_irqs

Data Fields

bit	meip	
bit	msip	
bit	mtip	
bit	nmi	
bit	seip	
bit[64]	time	Global time concept; may be tied low if user time CSR is not required
bit	ueip	

5.18.2.4 struct t_riscv_fetch_req

Data Fields

bit[32]	address	
bit	flush	
t_riscv_mode	mode	
bit	sequential	
bit	valid	

5.18.2.5 struct t_riscv_fetch_resp

Data Fields

bit[32]	data	
bit	debug	Needs to permit register read/write encoding, break after execution, break before execution, execution mode, breakpoint-in-hardware-not-software; force-debug-subroutine-trap-before-execution
bit	error	
t_riscv_mode	mode	
t_riscv_fetch_tag	tag	
bit	valid	

5.18.2.6 struct t_riscv_config

Data Fields

bit	coproc_disable	
bit	e32	
bit	i32c	
bit	i32m	
bit	i32m_fuse	
bit	unaligned_mem	

5.18.2.7 struct t_riscv_debug_mst

Data Fields

bit[16]	arg	Argument for debug op
t_riscv_word	data	Data for writing or instruction execution
bit[6]	mask	PDM attention mask (mask && id)==(mask&&select) -> drive attention on next cycle
t_riscv_debug_op	op	Operation for selected PDM to perform
bit[6]	select	PDM to select
bit	valid	Asserted if op is valid; has no effect on mask and attention

5.18.2.8 struct t_riscv_debug_tgt

Data Fields

bit	attention	Asserted by a PDM if it has unacknowledged halt, breakpoint hit, resumption
t_riscv_word	data	Data from a completed transaction; 0 otherwise
bit	halted	Asserted by a PDM if it is selected and halted since last ack; 0 otherwise
bit	hit_breakpoint	Asserted by a PDM if it is selected and has hit breakpoint since lask ack; 0 otherwise
bit	op_was_none	Asserted if the response is not valid
t_riscv_debug_resp	resp	Response from a requested op - only one op should be requested for each response

Data Fields

bit	resumed	Asserted by a PDM if it is selected and has resumed since last ack; 0 otherwise
bit[6]	selected	Number of the PDM driving, or 0 if not driving the bus
bit	valid	Asserted by a PDM if driving the bus

5.18.2.9 struct t_riscv_pipeline_debug_control

Data Fields

t_riscv_word	data	Data from a completed transaction; 0 otherwise
bit	fetch_dret	
bit	halt_request	
bit	kill_fetch	
bit	valid	

5.18.2.10 struct t_riscv_pipeline_debug_response

Data Fields

bit	exec_dret	
bit	exec_halting	
bit	exec_valid	

5.18.3 Typedef Documentation

5.18.3.1 typedef bit t_riscv_debug_resp

5.18.3.2 typedef bit [2] t_riscv_fetch_tag

5.18.3.3 typedef bit [32] t_riscv_word

5.18.4 Enumeration Type Documentation

5.18.4.1 enum t_riscv_debug_op

Enumerator

rv_debug_set_requests Set request bits for halt, resume, step (args[0..2])

rv_debug_read Request read of a GPR/CSR

rv_debug_write Request write of a GPR/CSR

rv_debug_acknowledge Acknowledge halt, breakpoint hit, status; removes attention signal

rv_debug_execute Execute instruction provided resumption of execution at dpc and in mode dcsr.prv

rv_debug_execute_progbuf Execute instruction at 'progbuf' address X (if it is a jump and link it will return)

5.18.4.2 enum t_riscv_mode

Enumerator

rv_mode_user

rv_mode_supervisor

rv_mode_machine

rv_mode_debug

5.18.5 Variable Documentation

5.18.5.1 constant integer RISCV_DATA_ADDR_WIDTH = 14

5.18.5.2 constant integer RISCV_INSTR_ADDR_WIDTH = 14

5.19 cdl/inc/riscv_internal_types.h File Reference

Data Structures

- struct t_riscv_csr_access
- struct t_riscv_csr_data
- struct t_riscv_csr_controls
- struct t_riscv_csr_dcsr
- struct t_riscv_csr_mstatus
- struct t_riscv_csr_mip
- struct t_riscv_csr_mie
- struct t_riscv_csrs_minimal
- struct t_riscv_i32_inst
- struct t_riscv_i32_decode_ext
- struct t_riscv_i32_decode
- struct t_riscv_i32_alu_result
- struct t_riscv_i32_coproc_controls
- struct t_riscv_i32_coproc_response
- struct t_riscv_i32_trace

Enumerations

- enum t_riscv_abi {
  riscv_abi_zero = 0,
  riscv_abi_link = 1,
  riscv_abi_sp = 2 }

- enum t_riscv_opc_rv32 {
  riscv_opc_load = 0,
  riscv_opc_load_fp = 1,
  riscv_opc_custom_0 = 2,
  riscv_opc_misc_mem = 3,
  riscv_opc_op_imm = 4,
  riscv_opc_auipc = 5,
  riscv_opc_op_imm32 = 6,
  riscv_opc_store = 8,
  riscv_opc_store_fp = 9,
  riscv_opc_custom_1 = 10,
  riscv_opc_amo = 11,
  riscv_opc_op = 12,
  riscv_opc_lui = 13,
  riscv_opc_op32 = 14,
  riscv_opc_madd = 16,
  riscv_opc_msub = 17,
  riscv_opc_nmsub = 18,
  riscv_opc_nmadd = 19,
  riscv_opc_op_fp = 20,
  riscv_opc_resvd_0 = 21,
  riscv_opc_custom_2 = 22,
  riscv_opc_branch = 24,
  riscv_opc_jalr = 25,
  riscv_opc_resvd_1 = 26,
  riscv_opc_jal = 27,
  riscv_opc_system = 28,
  riscv_opc_resvd_2 = 29,
  riscv_opc_custom_3 = 30 }
- enum t_riscv_opc_rv32c {
  riscv_opcc0_addi4spn = 0,
  riscv_opcc0_lw = 2,
  riscv_opcc0_sw = 6,
  riscv_opcc1_addi = 0,
  riscv_opcc1_jal = 1,
  riscv_opcc1_li = 2,
  riscv_opcc1_lui = 3,
  riscv_opcc1_arith = 4,
  riscv_opcc1_j = 5,
  riscv_opcc1_beqz = 6,
  riscv_opcc1_bnez = 7,
  riscv_opcc2_slli = 0,
  riscv_opcc2_lwsp = 2,
  riscv_opcc2_misc_alu = 4,
  riscv_opcc2_swsp = 6 }
- enum t_riscv_system_f12 {
  riscv_f12_ecall = 12h0,
  riscv_f12_ebreak = 12h1,
  riscv_f12_mret = 12h302,
  riscv_f12_mwfi = 12h105 }
- enum t_riscv_f3_alu {
  riscv_f3_addsub = 0,
  riscv_f3_sll = 1,
  riscv_f3_slt = 2,
  riscv_f3_sltu = 3,
  riscv_f3_xor = 4,
  riscv_f3_srlsra = 5,
  riscv_f3_or = 6,

riscv_f3_and = 7 }

- enum t_riscv_f3_muldiv {
  riscv_f3_mul = 0,
  riscv_f3_mulh = 1,
  riscv_f3_mulhsu = 2,
  riscv_f3_mulhu = 3,
  riscv_f3_div = 4,
  riscv_f3_divu = 5,
  riscv_f3_rem = 6,
  riscv_f3_remu = 7 }
- enum t_riscv_f3_branch {
  riscv_f3_beq = 0,
  riscv_f3_bne = 1,
  riscv_f3_blt = 4,
  riscv_f3_bge = 5,
  riscv_f3_bltu = 6,
  riscv_f3_bgeu = 7 }
- enum t_riscv_f3_load {
  riscv_f3_lb = 0,
  riscv_f3_lh = 1,
  riscv_f3_lw = 2,
  riscv_f3_lbu = 4,
  riscv_f3_lhu = 5 }
- enum t_riscv_f3_store {
  riscv_f3_sb = 0,
  riscv_f3_sh = 1,
  riscv_f3_sw = 2 }
- enum t_riscv_f3_misc_mem {
  riscv_f3_fence = 0,
  riscv_f3_fence_i = 1 }
- enum t_riscv_f3_system {
  riscv_f3_privileged = 0,
  riscv_f3_csrrw = 1,
  riscv_f3_csrrs = 2,
  riscv_f3_csrrc = 3,
  riscv_f3_csrrwi = 5,
  riscv_f3_csrrsi = 6,
  riscv_f3_csrrci = 7 }
- enum t_riscv_mem_width {
  mw_byte,
  mw_half,
  mw_word }
- enum t_riscv_op {
  riscv_op_branch,
  riscv_op_jal,
  riscv_op_jalr,
  riscv_op_system,
  riscv_op_csr,
  riscv_op_misc_mem,
  riscv_op_load,
  riscv_op_store,
  riscv_op_alu,
  riscv_op_muldiv,
  riscv_op_auipc,
  riscv_op_lui,
  riscv_op_ext,
  riscv_op_illegal }

- enum t_riscv_subop {
  riscv_subop_valid =0,
  riscv_subop_illegal = 0xf,
  riscv_subop_beq =0,
  riscv_subop_bne =1,
  riscv_subop_blt =2,
  riscv_subop_bge =3,
  riscv_subop_bltu =4,
  riscv_subop_bgeu =5,
  riscv_subop_add = 0,
  riscv_subop_sub = 0+8,
  riscv_subop_sll = 1,
  riscv_subop_slt = 2,
  riscv_subop_sltu = 3,
  riscv_subop_xor = 4,
  riscv_subop_srl = 5,
  riscv_subop_sra = 5+8,
  riscv_subop_or = 6,
  riscv_subop_and = 7,
  riscv_subop_mull = 0,
  riscv_subop_mulhss = 1,
  riscv_subop_mulhsu = 2,
  riscv_subop_mulhu = 3,
  riscv_subop_divs = 4,
  riscv_subop_divu = 5,
  riscv_subop_rems = 6,
  riscv_subop_remu = 7,
  riscv_subop_lb = 0,
  riscv_subop_lh = 1,
  riscv_subop_lw = 2,
  riscv_subop_lbu = 4,
  riscv_subop_lhu = 5,
  riscv_subop_sb = 0,
  riscv_subop_sh = 1,
  riscv_subop_sw = 2,
  riscv_subop_ecall = 0,
  riscv_subop_ebreak = 1,
  riscv_subop_mret = 2,
  riscv_subop_mwfi = 3,
  riscv_subop_fence = 0,
  riscv_subop_fence_i = 1,
  riscv_subop_csrrw = 1,
  riscv_subop_csrrs = 2,
  riscv_subop_csrrc = 3 }
- enum t_riscv_mcause {
  riscv_mcause_instruction_misaligned = 0,
  riscv_mcause_instruction_fault = 1,
  riscv_mcause_illegal_instruction = 2,
  riscv_mcause_breakpoint = 3,
  riscv_mcause_load_misaligned = 4,
  riscv_mcause_load_fault = 5,
  riscv_mcause_store_misaligned = 6,
  riscv_mcause_store_fault = 7,
  riscv_mcause_uecall = 8,
  riscv_mcause_secall = 9,
  riscv_mcause_hecall = 10,
  riscv_mcause_mecall = 11 }
- enum t_riscv_trap_cause {

riscv_trap_cause_instruction_misaligned = 0,
riscv_trap_cause_instruction_fault = 1,
riscv_trap_cause_illegal_instruction = 2,
riscv_trap_cause_breakpoint = 3,
riscv_trap_cause_load_misaligned = 4,
riscv_trap_cause_load_fault = 5,
riscv_trap_cause_store_misaligned = 6,
riscv_trap_cause_store_fault = 7,
riscv_trap_cause_uecall = 8,
riscv_trap_cause_secall = 9,
riscv_trap_cause_hecall = 10,
riscv_trap_cause_mecall = 11 }

- enum t_riscv_csr_access_type {
  riscv_csr_access_none = 0,
  riscv_csr_access_write = 1,
  riscv_csr_access_read = 2,
  riscv_csr_access_rw = 3,
  riscv_csr_access_rs = 6,
  riscv_csr_access_rc = 7 }

- enum t_riscv_csr_addr {

CSR_ADDR_READWRITE_MASK = 12hc00,
CSR_ADDR_READ_WRITE_A = 12h000,
CSR_ADDR_READ_WRITE_B = 12h400,
CSR_ADDR_READ_WRITE_C = 12h800,
CSR_ADDR_READ_ONLY = 12hC00,
CSR_ADDR_MODE_MASK = 12h300,
CSR_ADDR_USER_MODE = 12h000,
CSR_ADDR_SUPERVISOR_MODE = 12h100,
CSR_ADDR_HYPERVISOR_MODE = 12h200,
CSR_ADDR_MACHINE_MODE = 12h300,
CSR_ADDR_USTATUS = 12h000,
CSR_ADDR_UIE = 12h004,
CSR_ADDR_UTVEC = 12h005,
CSR_ADDR_USCRATCH = 12h040,
CSR_ADDR_UEPC = 12h041,
CSR_ADDR_UCAUSE = 12h042,
CSR_ADDR_UTVAL = 12h043,
CSR_ADDR_UIP = 12h044,
CSR_ADDR_CYCLE = 12hC00,
CSR_ADDR_TIME = 12hC01,
CSR_ADDR_INSTRET = 12hC02,
CSR_ADDR_CYCLEH = 12hC80,
CSR_ADDR_TIMEH = 12hC81,
CSR_ADDR_INSTRETH = 12hC82,
CSR_ADDR_SSTATUS = 12h100,
CSR_ADDR_SEDELEG = 12h102,
CSR_ADDR_SIDELEG = 12h103,
CSR_ADDR_SIE = 12h104,
CSR_ADDR_STVEC = 12h105,
CSR_ADDR_SCOUNTEREN = 12h106,
CSR_ADDR_SSCRATCH = 12h140,
CSR_ADDR_SEPC = 12h141,
CSR_ADDR_SCAUSE = 12h142,
CSR_ADDR_SBADADDR = 12h143,
CSR_ADDR_SIP = 12h144,
CSR_ADDR_SPTBR = 12h180,
CSR_ADDR_MSTATUS = 12h300,
CSR_ADDR_MISA = 12h301,
CSR_ADDR_MEDELEG = 12h302,
CSR_ADDR_MIDELEG = 12h303,
CSR_ADDR_MIE = 12h304,
CSR_ADDR_MTVEC = 12h305,
CSR_ADDR_MCOUNTEREN = 12h306,
CSR_ADDR_MSCRATCH = 12h340,
CSR_ADDR_MEPC = 12h341,
CSR_ADDR_MCAUSE = 12h342,
CSR_ADDR_MTVAL = 12h343,
CSR_ADDR_MIP = 12h344,
CSR_ADDR_MCYCLE = 12hB00,
CSR_ADDR_MINSTRET = 12hB02,
CSR_ADDR_MCYCLEH = 12hB80,
CSR_ADDR_MINSTRETH = 12hB82,
CSR_ADDR_MVENDORID = 12hF11,
CSR_ADDR_MARCHID = 12hF12,
CSR_ADDR_MIMPID = 12hF13,
CSR_ADDR_MHARTID = 12hF14,
CSR_ADDR_DSCRATCH = 12h7B2 }

Variables

- constant integer riscv_i32_ones = 0
- constant integer riscv_i32_opc = 2
- constant integer riscv_i32_rd = 7
- constant integer riscv_i32_f3 = 12
- constant integer riscv_i32_rs1 = 15
- constant integer riscv_i32_rs2 = 20
- constant integer riscv_i32_f7 = 25
- constant integer riscv_i32_f12 = 20

5.19.1 Data Structure Documentation

5.19.1.1 struct t_riscv_csr_access

Data Fields

t_riscv_csr_access_type	access	
bit[12]	address	

5.19.1.2 struct t_riscv_csr_data

Data Fields

bit	illegal_access	
bit[4]	interrupt_cause	From table 3.6 in RV priv space 1.10
t_riscv_mode	interrupt_mode	Mode to enter if take_interrupt is asserted
t_riscv_word	read_data	
bit	take_interrupt	

5.19.1.3 struct t_riscv_csr_controls

Data Fields

t_riscv_mode	exec_mode	Mode of instruction in the execution stage
bit	interrupt	
bit	retire	
bit	timer_clear	
bit	timer_inc	
bit	timer_load	
bit[64]	timer_value	
bit	trap	
t_riscv_trap_cause	trap_cause	
bit[32]	trap_pc	
bit[32]	trap_value	

5.19.1.4 struct t_riscv_csr_dcsr

Data Fields

bit[3]	cause	1=ebreak, 2=trigger module, 3=debugger request, 4=single step as step was set
bit	ebreakm	make ebreak instructions in machine mode enter debug mode
bit	ebreaks	make ebreak instructions in system mode enter debug mode
bit	ebreaku	make ebreak instructions in user mode enter debug mode
bit	mprven	if clear ignore mstatus.mprv when in debug mode
bit	nmip	asserted if an NMI is pending for the hart
bit[2]	prv	mode of execution prior to entry to debug mode, and to return to on dret
bit	step	when set enter debug mode after current instruction completes
bit	stepie	set to enable interrupts during stepping (may be hardwired to 0)
bit	stopcount	set to stop cycle and instret incrementing on instructions executed in debug mode
bit	stoptime	set to disable incrementing of hart-local timers when in debug mode
bit[4]	xdebug_ver	4 for conformant debug support, 0 otherwise

5.19.1.5 struct t_riscv_csr_mstatus

Data Fields

bit[2]	fs	
bit	mie	
bit	mpie	
bit[2]	mpp	
bit	mprv	
bit	mxr	
bit	sd	
bit	sie	
bit	spie	
bit	spp	
bit	sum	
bit	tsr	
bit	tvm	
bit	tw	
bit	uie	
bit	upie	
bit[2]	xs	

5.19.1.6 struct t_riscv_csr_mip

Data Fields

bit	meip	Machine-external interrupt pending, mirroring the input pin
bit	msip	Machine system interrupt pending, set by memory-mapped register if supported
bit	mtip	Machine timer interrupt pending, set by memory-mapped machine timer comparator meeting mtime
bit	seip	System-external interrupt pending, mirroring the input pin
bit	ssip	System software interrupt pending, set by software
bit	stip	System timer interrupt pending, set by software
bit	ueip	User-external interrupt pending, mirroring the input pin
bit	usip	User software interrupt pending, set by software

Data Fields

bit	utip	User timer interrupt pending, set by software

5.19.1.7 struct t_riscv_csr_mie

Data Fields

bit	meie	Enable for machine-external interrupt pending
bit	msie	Enable for machine system interrupt pending
bit	mtie	Enable for machine timer interrupt pending
bit	seie	Enable for system-external interrupt pending
bit	ssie	Enable for system software interrupt pending
bit	stie	Enable for system timer interrupt pending
bit	ueie	Enable for user-external interrupt pending
bit	usie	Enable for user software interrupt pending
bit	utie	Enable for user timer interrupt pending

5.19.1.8 struct t_riscv_csrs_minimal

Data Fields

bit[64]	cycles	Number of cycles since reset
bit[64]	instret	Number of instructions retired
bit[32]	mcause	Cause of last exception
bit[32]	mepc	PC at last exception
bit[32]	mscratch	Scratch register for exception routines
bit[32]	mtval	Value associated with last exception
bit[32]	mtvec	Trap vector, can be hardwired or writable
bit[64]	time	Mirror of irqs.time - may be tied to 0 if only machine mode is supported

5.19.1.9 struct t_riscv_i32_inst

Data Fields

bit[32]	data	
t_riscv_mode	mode	

5.19.1.10 struct t_riscv_i32_decode_ext

Data Fields

bit	dummy	

5.19.1.11 struct t_riscv_i32_decode

Data Fields

t_riscv_csr_access	csr_access	CSR access if valid and legal
t_riscv_i32_decode_ext	ext	extended decode, not used by the main pipeline
bit	illegal	asserted if an illegal opcode
bit[32]	immediate	Immediate value decoded from the instruction
bit[5]	immediate_shift	Immediate shift value decoded from the instruction
bit	immediate_valid	Asserted if immediate data is valid (generally used instead of source register 2)
bit	is_compressed	asserted if from an i32-c decode, clear otherwise (effects link register)
bit	memory_read_unsigned	if a memory read (op is riscv_opc_load), this indicates an unsigned read; otherwise ignored
t_riscv_mem_width	memory_width	ignored unless *memory_read* or *memory_write*; indicates size of memory transfer
t_riscv_op	op	Operation class of the instruction
bit[5]	rd	Destination register that is written by the instruction
bit	rd_written	Asserted if Rd is written to (hence also Rd will be non-zero)
bit	requires_machine_mode	Indicates that in non-machine-mode the instruction is illlegal
bit[5]	rs1	Source register 1 that is required by the instruction
bit	rs1_valid	Asserted if rs1 is valid; if deasserted then rs1 is not used
bit[5]	rs2	Source register 2 that is required by the instruction
bit	rs2_valid	Asserted if rs2 is valid; if deasserted then rs2 is not used
t_riscv_subop	subop	Subclass of the operation class

5.19.1.12 struct t_riscv_i32_alu_result

Data Fields

t_riscv_word	arith_result	Use for mem_address
bit	branch_condition_met	
t_riscv_word	branch_target	
t_riscv_csr_access	csr_access	
t_riscv_word	result	Result of ALU operation, dependent on subop

5.19.1.13 struct t_riscv_i32_coproc_controls

Data Fields

bit	alu_cannot_complete	Late in cycle: If asserted, alu cannot complete because it is still working on its operation
bit	alu_cannot_start	Late in cycle: If asserted, alu_idecode may be valid but rs1/rs2 are not; once deasserted it remains deasserted until a new ALU instruction starts
bit	alu_flush_pipeline	Late in cycle: If asserted, flush everything prior to alu; will only be asserted during a cycle if first cycle if ALU instruction - or if alu_cannot_start
t_riscv_word	alu_rs1	Early in cycle (after some muxes)

Data Fields

t_riscv_word	alu_rs2	Early in cycle (after some muxes)
t_riscv_i32_decode	dec_idecode	Mid-cycle: Idecode for the next cycle
bit	dec_idecode_valid	Mid-cycle: validates dec_idecode
bit	dec_to_alu_blocked	Late in the cycle: if set, ALU will not take decode; note that ALU flush overpowers this

5.19.1.14 struct t_riscv_i32_coproc_response

Data Fields

bit	cannot_complete	Early in cycle: if deasserted the module is performing a calculation that has not produced a valid result yet (feeds back in to controls alu_cannot_complete)
bit	cannot_start	If asserted, block start of the ALU stage - the instruction is then tried again in the next cycle, but can be interrupted
t_riscv_word	result	
bit	result_valid	Early in cycle, if asserted then coproc overcomes the ALU result

5.19.1.15 struct t_riscv_i32_trace

Data Fields

bit	branch_taken	Asserted if a branch is being taken
bit[32]	branch_target	Target of branch if being taken
bit[32]	instr_pc	Program counter of the instruction
bit	instr_valid	
t_riscv_i32_inst	instruction	Instruction word being decoded
t_riscv_word	rfw_data	Result of ALU/memory operation for the instruction
bit	rfw_data_valid	
bit[5]	rfw_rd	
bit	rfw_retire	Asserted if an instruction is being retired
bit	trap	

5.19.2 Enumeration Type Documentation

5.19.2.1 enum t_riscv_abi

Enumerator

riscv_abi_zero

riscv_abi_link

riscv_abi_sp

5.19.2.2 enum t_riscv_csr_access_type

Enumerator

riscv_csr_access_none

riscv_csr_access_write

riscv_csr_access_read

riscv_csr_access_rw

riscv_csr_access_rs

riscv_csr_access_rc

5.19.2.3 enum t_riscv_csr_addr

Enumerator

CSR_ADDR_READWRITE_MASK

CSR_ADDR_READ_WRITE_A

CSR_ADDR_READ_WRITE_B

CSR_ADDR_READ_WRITE_C

CSR_ADDR_READ_ONLY

CSR_ADDR_MODE_MASK

CSR_ADDR_USER_MODE

CSR_ADDR_SUPERVISOR_MODE

CSR_ADDR_HYPERVISOR_MODE

CSR_ADDR_MACHINE_MODE

CSR_ADDR_USTATUS

CSR_ADDR_UIE

CSR_ADDR_UTVEC

CSR_ADDR_USCRATCH

CSR_ADDR_UEPC

CSR_ADDR_UCAUSE

CSR_ADDR_UTVAL

CSR_ADDR_UIP

CSR_ADDR_CYCLE

CSR_ADDR_TIME

CSR_ADDR_INSTRET

CSR_ADDR_CYCLEH

CSR_ADDR_TIMEH

CSR_ADDR_INSTRETH

CSR_ADDR_SSTATUS

CSR_ADDR_SEDELEG

CSR_ADDR_SIDELEG

CSR_ADDR_SIE

CSR_ADDR_STVEC

CSR_ADDR_SCOUNTEREN

CSR_ADDR_SSCRATCH

CSR_ADDR_SEPC

CSR_ADDR_SCAUSE

CSR_ADDR_SBADADDR

CSR_ADDR_SIP

CSR_ADDR_SPTBR

CSR_ADDR_MSTATUS

CSR_ADDR_MISA

CSR_ADDR_MEDELEG

CSR_ADDR_MIDELEG

CSR_ADDR_MIE

CSR_ADDR_MTVEC

CSR_ADDR_MCOUNTEREN

CSR_ADDR_MSCRATCH

CSR_ADDR_MEPC

CSR_ADDR_MCAUSE

CSR_ADDR_MTVAL

CSR_ADDR_MIP

CSR_ADDR_MCYCLE

CSR_ADDR_MINSTRET

CSR_ADDR_MCYCLEH

CSR_ADDR_MINSTRETH

CSR_ADDR_MVENDORID

CSR_ADDR_MARCHID

CSR_ADDR_MIMPID

CSR_ADDR_MHARTID

CSR_ADDR_DSCRATCH

5.19.2.4 enum t_riscv_f3_alu

Enumerator

riscv_f3_addsub

riscv_f3_sll

riscv_f3_slt

riscv_f3_sltu

riscv_f3_xor

riscv_f3_srlsra

riscv_f3_or

riscv_f3_and

5.19.2.5 enum t_riscv_f3_branch

Enumerator

riscv_f3_beq
riscv_f3_bne
riscv_f3_blt
riscv_f3_bge
riscv_f3_bltu
riscv_f3_bgeu

5.19.2.6 enum t_riscv_f3_load

Enumerator

riscv_f3_lb
riscv_f3_lh
riscv_f3_lw
riscv_f3_lbu
riscv_f3_lhu

5.19.2.7 enum t_riscv_f3_misc_mem

Enumerator

riscv_f3_fence
riscv_f3_fence_i

5.19.2.8 enum t_riscv_f3_muldiv

Enumerator

riscv_f3_mul
riscv_f3_mulh
riscv_f3_mulhsu
riscv_f3_mulhu
riscv_f3_div
riscv_f3_divu
riscv_f3_rem
riscv_f3_remu

5.19.2.9 enum t_riscv_f3_store

Enumerator

riscv_f3_sb
riscv_f3_sh
riscv_f3_sw

5.19.2.10 enum t_riscv_f3_system

Enumerator

riscv_f3_privileged

riscv_f3_csrrw

riscv_f3_csrrs

riscv_f3_csrrc

riscv_f3_csrrwi

riscv_f3_csrrsi

riscv_f3_csrrci

5.19.2.11 enum t_riscv_mcause

Enumerator

riscv_mcause_instruction_misaligned

riscv_mcause_instruction_fault

riscv_mcause_illegal_instruction

riscv_mcause_breakpoint

riscv_mcause_load_misaligned

riscv_mcause_load_fault

riscv_mcause_store_misaligned

riscv_mcause_store_fault

riscv_mcause_uecall

riscv_mcause_secall

riscv_mcause_hecall

riscv_mcause_mecall

5.19.2.12 enum t_riscv_mem_width

Enumerator

mw_byte

mw_half

mw_word

5.19.2.13 enum t_riscv_op

Enumerator

riscv_op_branch
riscv_op_jal
riscv_op_jalr
riscv_op_system
riscv_op_csr
riscv_op_misc_mem
riscv_op_load
riscv_op_store
riscv_op_alu
riscv_op_muldiv
riscv_op_auipc
riscv_op_lui
riscv_op_ext
riscv_op_illegal

5.19.2.14 enum t_riscv_opc_rv32

Enumerator

riscv_opc_load
riscv_opc_load_fp
riscv_opc_custom_0
riscv_opc_misc_mem
riscv_opc_op_imm
riscv_opc_auipc
riscv_opc_op_imm32
riscv_opc_store
riscv_opc_store_fp
riscv_opc_custom_1
riscv_opc_amo
riscv_opc_op
riscv_opc_lui
riscv_opc_op32
riscv_opc_madd
riscv_opc_msub
riscv_opc_nmsub
riscv_opc_nmadd
riscv_opc_op_fp
riscv_opc_resvd_0
riscv_opc_custom_2
riscv_opc_branch
riscv_opc_jalr
riscv_opc_resvd_1
riscv_opc_jal
riscv_opc_system
riscv_opc_resvd_2
riscv_opc_custom_3

5.19.2.15 enum t_riscv_opc_rv32c

Enumerator

riscv_opcc0_addi4spn

riscv_opcc0_lw

riscv_opcc0_sw

riscv_opcc1_addi

riscv_opcc1_jal

riscv_opcc1_li

riscv_opcc1_lui

riscv_opcc1_arith

riscv_opcc1_j

riscv_opcc1_beqz

riscv_opcc1_bnez

riscv_opcc2_slli

riscv_opcc2_lwsp

riscv_opcc2_misc_alu

riscv_opcc2_swsp

5.19.2.16 enum t_riscv_subop

Enumerator

riscv_subop_valid

riscv_subop_illegal

riscv_subop_beq

riscv_subop_bne

riscv_subop_blt

riscv_subop_bge

riscv_subop_bltu

riscv_subop_bgeu

riscv_subop_add

riscv_subop_sub

riscv_subop_sll

riscv_subop_slt

riscv_subop_sltu

riscv_subop_xor

riscv_subop_srl

riscv_subop_sra

riscv_subop_or

riscv_subop_and

riscv_subop_mull

riscv_subop_mulhss

riscv_subop_mulhsu

riscv_subop_mulhu

riscv_subop_divs
riscv_subop_divu
riscv_subop_rems
riscv_subop_remu
riscv_subop_lb
riscv_subop_lh
riscv_subop_lw
riscv_subop_lbu
riscv_subop_lhu
riscv_subop_sb
riscv_subop_sh
riscv_subop_sw
riscv_subop_ecall
riscv_subop_ebreak
riscv_subop_mret
riscv_subop_mwfi
riscv_subop_fence
riscv_subop_fence_i
riscv_subop_csrrw
riscv_subop_csrrs
riscv_subop_csrrc

5.19.2.17 enum t_riscv_system_f12

Enumerator

riscv_f12_ecall
riscv_f12_ebreak
riscv_f12_mret
riscv_f12_mwfi

5.19.2.18 enum t_riscv_trap_cause

Enumerator

riscv_trap_cause_instruction_misaligned
riscv_trap_cause_instruction_fault
riscv_trap_cause_illegal_instruction
riscv_trap_cause_breakpoint
riscv_trap_cause_load_misaligned
riscv_trap_cause_load_fault
riscv_trap_cause_store_misaligned
riscv_trap_cause_store_fault
riscv_trap_cause_uecall
riscv_trap_cause_secall
riscv_trap_cause_hecall
riscv_trap_cause_mecall

5.19.3 Variable Documentation

5.19.3.1 constant integer riscv_i32_f12 = 20

5.19.3.2 constant integer riscv_i32_f3 = 12

5.19.3.3 constant integer riscv_i32_f7 = 25

5.19.3.4 constant integer riscv_i32_ones = 0

5.19.3.5 constant integer riscv_i32_opc = 2

5.19.3.6 constant integer riscv_i32_rd = 7

5.19.3.7 constant integer riscv_i32_rs1 = 15

5.19.3.8 constant integer riscv_i32_rs2 = 20

5.20 cdl/inc/riscv_modules.h File Reference

Header file for RISC-V implementations.

5.20.1 Detailed Description

Header file for RISC-V implementations.

Copyright

5.20.2 Modules

5.20.2.1 module riscv_i32_debug (clock *clk,* input bit *reset_n,* input t_apb_request *apb_request,* output t_apb_response *apb_response,* output t_riscv_debug_mst *debug_mst,* input t_riscv_debug_tgt *debug_tgt*)

Parameters

clk	System clock
reset_n	Active low reset
apb_request	APB request
apb_response	APB response
debug_mst	Debug master to PDMs
debug_tgt	Debug target from PDMs

5.20.2.2 module riscv_i32_ifetch_debug (input t_riscv_fetch_req *pipeline_ifetch_req,* output t_riscv_fetch_resp *pipeline_ifetch_resp,* input t_riscv_i32_trace *pipeline_trace,* input t_riscv_pipeline_debug_control *debug_control,* output t_riscv_pipeline_debug_response *debug_response,* output t_riscv_fetch_req *ifetch_req,* input t_riscv_fetch_resp *ifetch_resp*)

5.20.2.3 module riscv_i32_minimal (clock *clk,* input bit *reset_n,* input bit *proc_reset_n,* input t_riscv_irqs *irqs,* output t_riscv_mem_access_req *data_access_req,* input t_riscv_mem_access_resp *data_access_resp,* input t_sram_access_req *sram_access_req,* output t_sram_access_resp *sram_access_resp,* input t_riscv_config *riscv_config,* output t_riscv_i32_trace *trace*)

Parameters

in	*irqs*	Interrupts in to the CPU

5.20.2.4 module riscv_i32_minimal_apb (clock *clk,* input bit *reset_n,* input t_riscv_mem_access_req *data_access_req,* output t_riscv_mem_access_resp *data_access_resp,* output t_apb_request *apb_request,* input t_apb_response *apb_response*)

5.20.2.5 module riscv_i32_pipeline_debug (clock *clk,* input bit *reset_n,* input t_riscv_debug_mst *debug_mst,* output t_riscv_debug_tgt *debug_tgt,* output t_riscv_pipeline_debug_control *debug_control,* input t_riscv_pipeline_debug_response *debug_response,* input bit *rv_select[6]*)

5.20.2.6 module riscv_i32_trace (clock *clk,* input bit *reset_n,* input t_riscv_i32_trace *trace*)

Parameters

clk	Clock for the CPU
reset_n	Active low reset
trace	Trace signals

5.20.2.7 module riscv_i32c_pipeline (clock *clk,* input bit *reset_n,* input t_riscv_irqs *irqs,* output t_riscv_fetch_req *ifetch_req,* input t_riscv_fetch_resp *ifetch_resp,* output t_riscv_mem_access_req *dmem_access_req,* input t_riscv_mem_access_resp *dmem_access_resp,* output t_riscv_i32_coproc_controls *coproc_controls,* input t_riscv_i32_coproc_response *coproc_response,* input t_riscv_config *riscv_config,* output t_riscv_i32_trace *trace*)

Parameters

irqs	Interrupts in to the CPU

5.20.2.8 module riscv_i32c_pipeline3 (clock *clk,* input bit *reset_n,* input t_riscv_irqs *irqs,* output t_riscv_fetch_req *ifetch_req,* input t_riscv_fetch_resp *ifetch_resp,* output t_riscv_mem_access_req *dmem_access_req,* input t_riscv_mem_access_resp *dmem_access_resp,* output t_riscv_i32_coproc_controls *coproc_controls,* input t_riscv_i32_coproc_response *coproc_response,* input t_riscv_config *riscv_config,* output t_riscv_i32_trace *trace*)

Parameters

irqs	Interrupts in to the CPU

5.20.2.9 module riscv_jtag_apb_dm (clock *jtag_tck,* input bit *reset_n,* input bit *ir[5],* input t_jtag_action *dr_action,* input *bitdr_in[50],* output *bitdr_tdi_mask[50],* output *bitdr_out[50],* clock *apb_clock,* output t_apb_request *apb_request,* input t_apb_response *apb_response*)

Parameters

jtag_tck	JTAG TCK signal, used as a clock
reset_n	Reset that drives all the logic
ir	JTAG IR which is to be matched against t_jtag_addr
dr_action	Action to perform with DR (capture or update, else ignore)
bitdr_in	Data register in; used in update, replaced by dr_out in capture, shift
bitdr_tdi_mask	One-hot mask indicating which DR bit TDI should replace (depends on IR)
bitdr_out	Data register out; same as data register in, except during capture when it is replaced by correct data dependent on IR, or shift when it goes right by one
apb_clock	APB clock signal, asynchronous to JTAG TCK
apb_request	APB request out
apb_response	APB response

5.21 cdl/inc/riscv_submodules.h File Reference

5.21.1 Modules

5.21.1.1 module riscv_csrs_minimal (clock *clk,* input bit *reset_n,* input t_riscv_irqs *irqs,* input t_riscv_csr_access *csr_access,* input t_riscv_word *csr_write_data,* output t_riscv_csr_data *csr_data,* input t_riscv_csr_controls *csr_controls,* output t_riscv_csrs_minimal *csrs*)

Parameters

clk	RISC-V clock
reset_n	Active low reset
irqs	Interrupts in to the CPU
csr_access	RISC-V CSR access, combinatorially decoded
csr_write_data	Write data for the CSR access, later in the cycle than possibly
csr_data	CSR response (including take interrupt and read data), from the current *csr_access*
csr_controls	Control signals to update the CSRs
csrs	CSR values

5.21.1.2 module riscv_e32_decode (input t_riscv_i32_inst *instruction,* output t_riscv_i32_decode *idecode,* input t_riscv_config *riscv_config*)

5.21.1.3 module riscv_e32c_decode (input t_riscv_i32_inst *instruction,* output t_riscv_i32_decode *idecode,* input t_riscv_config *riscv_config*)

5.21.1.4 module riscv_i32_alu (input t_riscv_i32_decode *idecode,* input t_riscv_word *pc,* input t_riscv_word *rs1,* input t_riscv_word *rs2,* output t_riscv_i32_alu_result *alu_result*)

5.21.1.5 module riscv_i32_decode (input t_riscv_i32_inst *instruction,* output t_riscv_i32_decode *idecode,* input t_riscv_config *riscv_config*)

5.21.1.6 module riscv_i32_muldiv (clock *clk,* input bit *reset_n,* input t_riscv_i32_coproc_controls *coproc_controls,* output t_riscv_i32_coproc_response *coproc_response,* input t_riscv_config *riscv_config*)

5.21.1.7 module riscv_i32c_decode (input t_riscv_i32_inst *instruction,* output t_riscv_i32_decode *idecode,* input t_riscv_config *riscv_config*)

5.22 cdl/inc/srams.h File Reference

SRAM modules used by all the modules.

5.22.1 Detailed Description

SRAM modules used by all the modules.

Data Structures

- struct t_sram_access_req
- struct t_sram_access_resp

5.22.2 Data Structure Documentation

5.22.2.1 struct t_sram_access_req

Data Fields

bit[32]	address	
bit[8]	byte_enable	
bit[4]	id	
bit	read_not_write	
bit	valid	
bit[64]	write_data	

5.22.2.2 struct t_sram_access_resp

Data Fields

bit	ack	
bit[64]	data	
bit[4]	id	
bit	valid	

5.22.3 Modules

5.22.3.1 module se_sram_mrw_2_16384x48 (clock *sram_clock_0,* input bit *select_0,* input bit *address_0[14],* input bit *read_not_write_0,* input bit *write_data_0[48],* output bit *data_out_0[48],* clock *sram_clock_1,* input bit *select_1,* input bit *address_1[14],* input bit *read_not_write_1,* input bit *write_data_1[48],* output bit *data_out_1[48]*)

5.22.3.2 module se_sram_mrw_2_16384x8 (clock *sram_clock_0,* input bit *select_0,* input bit *address_0[14],* input bit *read_not_write_0,* input bit *write_data_0[8],* output bit *data_out_0[8],* clock *sram_clock_1,* input bit *select_1,* input bit *address_1[14],* input bit *read_not_write_1,* input bit *write_data_1[8],* output bit *data_out_1[8]*)

5.22.3.3 module se_sram_srw_128x45 (clock *sram_clock,* input bit *select,* input bit *address[7],* input bit *read_not_write,* input bit *write_data[45],* output bit *data_out[45]*)

5.22.3.4 module se_sram_srw_128x64 (clock *sram_clock,* input bit *select,* input bit *address[7],* input bit *read_not_write,* input bit *write_enable,* input bit *write_data[64],* output bit *data_out[64]*)

5.22.3.5 module se_sram_srw_16384x32 (clock *sram_clock,* input bit *select,* input bit *address[14],* input bit *write_enable,* input bit *read_not_write,* input bit *write_data[32],* output bit *data_out[32]*)

5.22.3.6 module se_sram_srw_16384x32_we8 (clock *sram_clock,* input bit *select,* input bit *address[14],* input bit *read_not_write,* input bit *write_enable[4],* input bit *write_data[32],* output bit *data_out[32]*)

5.22.3.7 module se_sram_srw_16384x40 (clock *sram_clock,* input bit *select,* input bit *address[14],* input bit *read_not_write,* input bit *write_data[40],* output bit *data_out[40]*)

5.22.3.8 module se_sram_srw_16384x8 (clock *sram_clock,* input bit *select,* input bit *address[14],* input bit *read_not_write,* input bit *write_enable,* input bit *write_data[8],* output bit *data_out[8]*)

5.22.3.9 module se_sram_srw_256x40 (clock *sram_clock,* input bit *select,* input bit *address[8],* input bit *read_not_write,* input bit *write_data[40],* output bit *data_out[40]*)

5.22.3.10 module se_sram_srw_256x7 (clock *sram_clock,* input bit *select,* input bit *address[8],* input bit *read_not_write,* input bit *write_data[7],* output bit *data_out[7]*)

5.22.3.11 module se_sram_srw_32768x32 (clock *sram_clock,* input bit *select,* input bit *address[15],* input bit *read_not_write,* input bit *write_enable,* input bit *write_data[32],* output bit *data_out[32]*)

5.22.3.12 module se_sram_srw_32768x64 (clock *sram_clock,* input bit *select,* input bit *address[15],* input bit *read_not_write,* input bit *write_enable,* input bit *write_data[64],* output bit *data_out[64]*)

5.22.3.13 module se_sram_srw_65536x32 (clock *sram_clock,* input bit *select,* input bit *address[16],* input bit *read_not_write,* input bit *write_enable,* input bit *write_data[32],* output bit *data_out[32]*)

5.22.3.14 module se_sram_srw_65536x8 (clock *sram_clock,* input bit *select,* input bit *address[16],* input bit *read_not_write,* input bit *write_enable,* input bit *write_data[8],* output bit *data_out[8]*)

5.23 cdl/inc/teletext.h File Reference

Data Structures

- struct t_teletext_timings
- struct t_teletext_character
- struct t_teletext_rom_access
- struct t_teletext_pixels

Enumerations

- enum t_teletext_vertical_interpolation {
  tvi_all_scanlines,
  tvi_even_scanlines,
  tvi_odd_scanlines }

5.23.1 Data Structure Documentation

5.23.1.1 struct t_teletext_timings

Data Fields

bit	end_of_scanline	Asserted if end of scanline
bit	first_scanline_of_row	Asserted if first scanline of row; not required if module's internal timing is trusted
t_teletext_vertical_interpolation	interpolate_vertical	Asserted if vertical interpolation is desired
bit	restart_frame	Asserted if restarting the frame (resets all teletext character state)
bit	smoothe	Asserted if interpolation is desired

5.23.1.2 struct t_teletext_character

Data Fields

bit[7]	character	
bit	valid	

5.23.1.3 struct t_teletext_rom_access

Data Fields

bit[7]	address	
bit	select	

5.23.1.4 struct t_teletext_pixels

Data Fields

bit[12]	blue	
bit[12]	green	
bit	last_scanline	Asserted with a pixel to indicate it is on the last scanline of the row
bit[12]	red	
bit	valid	Asserted to indicate that the red, green and blue are valid; asserted three ticks after a valid character in

5.23.2 Enumeration Type Documentation

5.23.2.1 enum t_teletext_vertical_interpolation

Enumerator

tvi_all_scanlines For twenty scanline output characters

tvi_even_scanlines Only output scanlines 0, 2, 4, ... 18 - for even interlace fields, or for 10-scanline displays

tvi_odd_scanlines For twenty scanline output characters, but only outputing scanlines 1, 3, 5, ... 19 - for odd interlace fields

5.23.3 Modules

5.23.3.1 module teletext (clock *clk,* input bit *reset_n,* input t_teletext_character *character,* input t_teletext_timings *timings,* output t_teletext_rom_access *rom_access,* input bit *rom_data[45],* output t_teletext_pixels *pixels*)

Parameters

in	*clk*	Character clock
in	*character*	Parallel character data in, with valid signal
in	*timings*	Timings for the scanline, row, etc
out	*rom_access*	Teletext ROM access
in	*rom_data*	Teletext ROM data, valid in cycle after rom_access
out	*pixels*	Output pixels, two clock ticks delayed from clk in

5.24 cdl/inc/timer.h File Reference

Timer types header file for CDL modules.

5.24.1 Detailed Description

Timer types header file for CDL modules.

Header file for the types for timers

Data Structures

- struct t_timer_control
- struct t_timer_value

5.24.2 Data Structure Documentation

5.24.2.1 struct t_timer_control

Data Fields

bit	block_writes	Assert to block writes to the timer counter
bit[8]	bonus_subfraction_denom	(d-1) in fractional n/16d increment per cycle; If zero then no subfractional add
bit[8]	bonus_subfraction_numer	(n-1) in fractional n/16d increment per cycle
bit	enable_counter	Assert to enable the timer counter; otherwise it holds its value
bit[4]	fractional_adder	f in fraction f/16 to add per cycle
bit[8]	integer_adder	integer amount to add to timer counter per cycle
bit	reset_counter	Assert to reset the timer counter to 0; this takes precedence over enable_counter

5.24.2.2 struct t_timer_value

Data Fields

bit	irq	Asserted if comparator >= timer value
bit[64]	value	64-bit timer value, reflecting the value in the timer counter

5.25 cdl/inc/utils.h File Reference

Header file for utilities.

5.25.1 Detailed Description

Header file for utilities.

Header file for the types and CDL modules for generic utilities

5.25.2 Modules

5.25.2.1 module hysteresis_switch (clock *clk,* input bit *reset_n,* input bit *clk_enable,* input bit *input_value,* output bit *output_value,* input bit *filter_period[16],* input bit *filter_level[16]*)

Parameters

in	*clk_enable*	Assert to enable the internal clock; this permits I/O switches to easily use a slower clock
in	*filter_period*	Period over which to filter the input - the larger the value, the longer it takes to switch, but the more glitches are removed
in	*filter_level*	Value to exceed to switch output levels - the larger the value, the larger the hysteresis; must be less than filter_period

5.26 cdl/inc/video.h File Reference

Data Structures

- struct t_video_bus
- struct t_adv7123

5.26.1 Data Structure Documentation

5.26.1.1 struct t_video_bus

Data Fields

bit[8]	blue	
bit	display_enable	Asserted after end of back porch for display pixels/lines
bit[8]	green	
bit	hsync	Asserted for one cycle at start of back porch
bit[8]	red	
bit	vsync	Asserted for one whole line at start of back porch of frame, simultaneous with hsync

5.26.1.2 struct t_adv7123

Data Fields

bit	blank↩ _n	Drive high for RGB out to come from red/green/blue; low for RGB to be blanking level
bit[10]	blue	Blue data, latched on rising clock
bit[10]	green	Green data, latched on rising clock
bit	hs	Hsync asserted for n cycles to start retrace and the end of a line
bit[10]	red	Red data, latched on rising clock
bit	sync↩ _n	Composite sync, drive low if sync-on-green is not required, should only be driven low in blanking
bit	vs	Vsync asserted for n horizontal lines

5.27 cdl/README.md File Reference

Index