# CDL Modules

Generated by Doxygen 1.8.6

Sun Mar 5 2017 17:12:19

# Contents

# Chapter 1

# Main Page

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; timer, GPIO. 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.

- cpus

- 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.

- 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_processor

### Files

- file apb_processor.cdl

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

### 4.1.1 Detailed Description

### 4.1.2 Modules

#### 4.1.2.1 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 |

## 4.2 apb_target_gpio

### 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_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 |

## 4.3 apb_target_timer

### Files

- file apb_target_timer.cdl

  *Simple timer target for an APB bus.*

### 4.3.1 Detailed Description

### 4.3.2 Modules

#### 4.3.2.1 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 timer 0 at address 0, timer 1 at address 1, and so on. When a timer is written it writes the 31-bit *comparator* value and it clears the *timer's equalled* bit. When a timer 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 |

## 4.4 bbc_micro_de1_cl

### Files

- file bbc_micro_de1_cl.cdl

  *BBC microcomputer with RAMs for the CL DE1 + daughterboard.*

### 4.4.1 Detailed Description

### 4.4.2 Modules

#### 4.4.2.1 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 |

## 4.5 bbc_micro_de1_cl_bbc

### Files

- file bbc_micro_de1_cl_bbc.cdl

  *BBC microcomputer with RAMs for the CL DE1 + daughterboard.*

### 4.5.1 Detailed Description

### 4.5.2 Modules

#### 4.5.2.1 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 |

# 4.6 bbc_micro_de1_cl_io

## Files

- file bbc_micro_de1_cl.cdl

  *BBC microcomputer with RAMs for the CL DE1 + daughterboard.*

### 4.6.1 Detailed Description

### 4.6.2 Modules

#### 4.6.2.1 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 |

## 4.7 de1_cl_controls

### Files

- file bbc_micro_de1_cl.cdl

  *BBC microcomputer with RAMs for the CL DE1 + daughterboard.*

### 4.7.1 Detailed Description

### 4.7.2 Modules

#### 4.7.2.1 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 |

## 4.8 cpu6502

### Files

- file cpu6502.cdl

  *CDL implementation of 6502 CPU core.*

### 4.8.1 Detailed Description

### 4.8.2 Modules

#### 4.8.2.1 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 |
| out | *data_out* | In real 6502, valid at end of phi2 with data to write |

| | | |
|---|---|---|
| in | *data_in* | Captured at the end of phi2 (rising clock in here) |

## 4.9 csr_master_apb

### Files

- file csr_master_apb.cdl

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

### 4.9.1 Detailed Description

### 4.9.2 Modules

#### 4.9.2.1 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 |

## 4.10 csr_target_apb

### Files

- file csr_target_apb.cdl

  *Pipelined CSR request/response interface to APB slave interface.*

### 4.10.1 Detailed Description

### 4.10.2 Modules

#### 4.10.2.1 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 |

## 4.11 csr_target_csr

### Files

- file csr_target_csr.cdl

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

### 4.11.1 Detailed Description

### 4.11.2 Modules

#### 4.11.2.1 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 |

## 4.12 csr_target_timeout

### Files

- file csr_target_timeout.cdl

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

### 4.12.1 Detailed Description

### 4.12.2 Modules

#### 4.12.2.1 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 |

## 4.13 ps2_host

### Files

- file ps2_host.cdl

  *PS2 interface for keyboard or mouse.*

### 4.13.1 Detailed Description

### 4.13.2 Modules

#### 4.13.2.1 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 |

## 4.14 ps2_host_keyboard

### Files

- file ps2_host_keyboard.cdl

  *PS2 interface converter for keyboard as host.*

### 4.14.1 Detailed Description

### 4.14.2 Modules

#### 4.14.2.1 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 |

## 4.15 led_seven_segment

### Files

- file led_seven_segment.cdl

  *Simple module to support 7-segment hex display.*

### 4.15.1 Detailed Description

### 4.15.2 Modules

#### 4.15.2.1 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 |

## 4.16 led_ws2812_chain

### Files

- file led_ws2812_chain.cdl

  *'Neopixel' LED chain driver module*

### 4.16.1 Detailed Description

### 4.16.2 Modules

#### 4.16.2.1 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 |

## 4.17 bbc_csr_interface

### 4.17.1 Detailed Description

### 4.17.2 Modules

#### 4.17.2.1 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 |

## 4.18 bbc_display_sram

### Files

- file bbc_display_sram.cdl

  *BBC micro display to SRAM write interface module.*

### 4.18.1 Detailed Description

### 4.18.2 Modules

#### 4.18.2.1 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'.

## 4.19 bbc_floppy_sram

### Files

- file bbc_floppy_sram.cdl

  *BBC micro floppy to SRAM read/write interface module.*

### 4.19.1 Detailed Description

### 4.19.2 Modules

#### 4.19.2.1 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. |

## 4.20 bbc_keyboard_csr

### Files

- file bbc_display_sram.cdl

  *BBC micro display to SRAM write interface module.*

### 4.20.1 Detailed Description

### 4.20.2 Modules

#### 4.20.2.1 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 |

## 4.21 bbc_keyboard_ps2

### Files

- file bbc_keyboard_ps2.cdl

  *BBC micro keyboard from PS2 keys.*

### 4.21.1 Detailed Description

### 4.21.2 Modules

#### 4.21.2.1 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 |

## 4.22 bbc_micro

### Files

- file bbc_micro.cdl

  *BBC microcomputer implementation module.*

### 4.22.1 Detailed Description

### 4.22.2 Modules

#### 4.22.2.1 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 |

## 4.23 bbc_micro_clocking

### Files

- file bbc_micro_clocking.cdl

  *BBC microcomputer clock generation module.*

### 4.23.1 Detailed Description

### 4.23.2 Modules

#### 4.23.2.1 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 |

## 4.24 bbc_micro_keyboard

### Files

- file bbc_micro_keyboard.cdl

  *BBC microcomputer keyboard module.*

### 4.24.1 Detailed Description

### 4.24.2 Modules

#### 4.24.2.1 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 |

## 4.25 bbc_micro_rams

### Files

- file bbc_micro_rams.cdl

  *BBC microcomputer RAMs module.*

### 4.25.1 Detailed Description

### 4.25.2 Modules

#### 4.25.2.1 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 |

## 4.26 bbc_micro_with_rams

### Files

- file bbc_micro_with_rams.cdl

  *BBC microcomputer with RAMs module.*

### 4.26.1 Detailed Description

### 4.26.2 Modules

#### 4.26.2.1 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 |

## 4.27 bbc_vidproc

### Files

- file bbc_vidproc.cdl

  *BBC microcomputer video ULA CDL implementation.*

### 4.27.1 Detailed Description

### 4.27.2 Modules

#### 4.27.2.1 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 |

## 4.28 acia6850

### Files

- file acia6850.cdl

  *6850 async communications chip CDL implementation*

### 4.28.1 Detailed Description

### 4.28.2 Modules

#### 4.28.2.1 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 |

## 4.29 via6522

### Files

- file via6522.cdl

  *CDL implementation of a 6522 versatile interface adaptor (VIA)*

### 4.29.1 Detailed Description

### 4.29.2 Modules

#### 4.29.2.1 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 |

## 4.30 fdc8271

### Files

- file fdc8271.cdl

  *CDL implementation of 8271 FDC.*

### 4.30.1 Detailed Description

### 4.30.2 Modules

#### 4.30.2.1 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 |

# 4.31 dprintf

## Files

- file dprintf.cdl

  *Debug text formatter.*

### 4.31.1 Detailed Description

### 4.31.2 Modules

#### 4.31.2.1 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 154 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 |

## 4.32 dprintf_2_mux

## 4.33 dprintf_4_mux

## 4.34 generic_valid_ack_mux

### Files

- file generic_valid_ack_mux.cdl

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

### 4.34.1 Detailed Description

### 4.34.2 Modules

#### 4.34.2.1 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 |

## 4.35 hysteresis_switch

### Files

- file hysteresis_switch.cdl

  *A hystersis detector using counter pairs.*

### 4.35.1 Detailed Description

### 4.35.2 Modules

#### 4.35.2.1 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 |

## 4.36 crtc6845

### Files

- file crtc6845.cdl

  *CDL implementation of 6845 CRTC.*

### 4.36.1 Detailed Description

### 4.36.2 Modules

#### 4.36.2.1 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 |

# 4.37 framebuffer

## Files

- file framebuffer.cdl

  *Framebuffer module with separate display and video sides.*

### 4.37.1 Detailed Description

### 4.37.2 Modules

#### 4.37.2.1 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.

The video output side has a programmable horizontal period that starts with hsync high for one clock, and then has a programmable back porch, followed by a programmable number of pixels (with data out enabled only if on the correct vertical portion of the display), followed by a programmable front porch, repeating.

The video output side has a programmable vertical period that is in units of horizontal period; it starts with vsync high for one horizontal period, and then has a programmable front porch, followed by a programmable number of displayed lined, followed by a programmable front porch, repeating.

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 framebuffer uses a *framebuffer_timing* module to generate video sync signals and other controls.

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.

## 4.38 framebuffer_teletext

### Files

- file framebuffer_teletext.cdl

  *Teletext framebuffer module with separate write and video sides.*

### 4.38.1 Detailed Description

### 4.38.2 Modules

#### 4.38.2.1 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 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* | |

## 4.39 framebuffer_timing

### Files

- file framebuffer_timing.cdl

  *Framebuffer timing module to create sync and display signals.*

### 4.39.1 Detailed Description

### 4.39.2 Modules

#### 4.39.2.1 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.

**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 |

## 4.40 saa5050

### Files

- file saa5050.cdl

  *CDL implementation of Mullard SAA5050.*

### 4.40.1 Detailed Description

### 4.40.2 Modules

#### 4.40.2.1 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 |

## 4.41 teletext

### Files

- file teletext.cdl

  *CDL implementation of a teletext decoder.*

### 4.41.1 Detailed Description

### 4.41.2 Modules

#### 4.41.2.1 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]) &∼ (n-1)[x-left]

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

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

### 5.1.3 Modules

#### 5.1.3.1 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**

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

## 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. 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 modules for some very simple APB peripherals

### 5.2.2 Modules

#### 5.2.2.1 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 |

#### 5.2.2.2 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**

| | | |
|---|---|---|
| in | *clk* | System clock |
| in | *reset_n* | Active low reset |
| in | *apb_request* | APB request |
| out | *apb_response* | APB response |

## 5.3 cdl/inc/bbc_micro_types.h File Reference

BBC micro types header file for CDL.

### 5.3.1 Detailed Description

BBC micro types header file for CDL. 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 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
- struct t_video_bus

### 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 }

## 5.3.2 Data Structure Documentation

#### 5.3.2.1 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 | |

#### 5.3.2.2 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 | |

#### 5.3.2.3 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 | |

#### 5.3.2.4 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 | |

#### 5.3.2.5 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 | |

#### 5.3.2.6 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 | |

#### 5.3.2.7 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 | |

#### 5.3.2.8 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 | |

#### 5.3.2.9 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_2M-Hz_video | Asserted if 'enable_2MHz_video' will be asserted in the next 'clk' period |

#### 5.3.2.10 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 |

#### 5.3.2.11 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 |

#### 5.3.2.12 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 |
| 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 |

#### 5.3.2.13 struct t_video_bus

**Data Fields**

| | | |
|---|---|---|
| bit[8] | blue | |
| bit | display_enable | |
| bit[8] | green | |
| bit | hsync | |
| bit[8] | red | |
| bit | vsync | |

### 5.3.3 Enumeration Type Documentation

#### 5.3.3.1 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***

#### 5.3.3.2 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***

#### 5.3.3.3 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***

## 5.4 cdl/inc/bbc_submodules.h File Reference

BBC micro CDL submodules.

### 5.4.1 Detailed Description

BBC micro CDL submodules. 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 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

### 5.4.2 Modules

#### 5.4.2.1 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 |

#### 5.4.2.2 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 |

#### 5.4.2.3 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 |

#### 5.4.2.4 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 |

#### 5.4.2.5 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 |

#### 5.4.2.6 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 |

#### 5.4.2.7 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 |

#### 5.4.2.8 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 |

#### 5.4.2.9 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 |

#### 5.4.2.10 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 |

#### 5.4.2.11 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 |

#### 5.4.2.12 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) |

#### 5.4.2.13 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 |

#### 5.4.2.14 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 |

#### 5.4.2.15 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 |

#### 5.4.2.16 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 |

## 5.5 cdl/inc/csr_interface.h File Reference

Types and modules for the CSR interface.

### 5.5.1 Detailed Description

Types and modules for the CSR interface.

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

## 5.5.2 Data Structure Documentation

#### 5.5.2.1 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 | |

#### 5.5.2.2 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 | |

#### 5.5.2.3 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 | |

### 5.5.3 Typedef Documentation

#### 5.5.3.1 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.

### 5.5.4 Modules

#### 5.5.4.1 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 |

#### 5.5.4.2 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 |

#### 5.5.4.3 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 |

#### 5.5.4.4 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 |
| *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 |

## 5.6 cdl/inc/de1_cl.h File Reference

Input file for DE1 cl inputs and boards.

### 5.6.1 Detailed Description

Input file for DE1 cl inputs and boards. 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

Header file for the types and CDL modules for input devices

### 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

## 5.6.2 Data Structure Documentation

#### 5.6.2.1 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 |

#### 5.6.2.2 struct t_rotary_motion_inputs

**Data Fields**

| | | |
|---|---|---|
| bit | direction_pin | |
| bit | transition_pin | |

#### 5.6.2.3 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 |

#### 5.6.2.4 struct t_de1_cl_diamond

Data Fields

| | | |
|---|---|---|
| bit | a | |
| bit | b | |
| bit | x | |
| bit | y | |

#### 5.6.2.5 struct t_de1_cl_joystick

Data Fields

| | | |
|---|---|---|
| bit | c | |
| bit | d | |
| bit | l | |
| bit | r | |
| bit | u | |

#### 5.6.2.6 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 | |

#### 5.6.2.7 struct t_de1_cl_rotary

Data Fields

| | | |
|---|---|---|
| bit | direction | |
| bit | direction_pulse | |
| bit | pressed | |

#### 5.6.2.8 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_-<br>switches | |

#### 5.6.2.9 struct t_de1_cl_lcd

**Data Fields**

| | | |
|---|---|---|
| bit | backlight | |
| bit[6] | blue | |
| bit | display_enable | |
| bit[7] | green | |
| bit | hsync_n | |
| bit[6] | red | |
| bit | vsync_n | |

#### 5.6.2.10 struct t_de1_cl_shift_register_control

**Data Fields**

| | | |
|---|---|---|
| bit | sr_clock | |
| bit | sr_shift | |

### 5.6.3 Modules

**5.6.3.1 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 |

**5.6.3.2 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 |

#### 5.6.3.3 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 |

## 5.7 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

### 5.7.1 Data Structure Documentation

#### 5.7.1.1 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 | |

#### 5.7.1.2 struct t_dprintf_req_2

**Data Fields**

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

#### 5.7.1.3 struct t_dprintf_resp

**Data Fields**

| | | |
|---|---|---|
| bit | ack | |

#### 5.7.1.4 struct t_dprintf_byte

**Data Fields**

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

### 5.7.2 Modules

#### 5.7.2.1 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 |

#### 5.7.2.2 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* )

#### 5.7.2.3 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* )

## 5.8 cdl/inc/framebuffer.h File Reference

Framebuffer CDL types and submodules.

### 5.8.1 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

### 5.8.2 Data Structure Documentation

#### 5.8.2.1 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_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* |

### 5.8.3 Modules

#### 5.8.3.1 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 |

#### 5.8.3.2 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 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 |

#### 5.8.3.3 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 |

## 5.9 cdl/inc/input_devices.h File Reference

Input device header file for CDL modules.

### 5.9.1 Detailed Description

Input device header file for CDL modules. 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 CDL modules for input devices

### Data Structures

- struct t_ps2_pins
- struct t_ps2_rx_data
- struct t_ps2_key_state

### 5.9.2 Data Structure Documentation

#### 5.9.2.1 struct t_ps2_pins

**Data Fields**

| | | |
|---|---|---|
| bit | clk | |
| bit | data | |

#### 5.9.2.2 struct t_ps2_rx_data

**Data Fields**

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

#### 5.9.2.3 struct t_ps2_key_state

**Data Fields**

| | | |
|---|---|---|
| bit | extended | |
| bit[8] | key_number | |
| bit | release | |
| bit | valid | |

### 5.9.3 Modules

#### 5.9.3.1 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 |

#### 5.9.3.2 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 |

## 5.10 cdl/inc/leds.h File Reference

Constants, types and modules for various LED drivers.

### 5.10.1 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

## 5.10.2 Data Structure Documentation

#### 5.10.2.1 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 |

#### 5.10.2.2 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 |

## 5.10.3 Modules

#### 5.10.3.1 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 |

#### 5.10.3.2 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 |

### 5.10.4 Variable Documentation

#### 5.10.4.1 constant bit [16] led_seven_seg_hex_a = 16b_1101011111101101

#### 5.10.4.2 constant bit [16] led_seven_seg_hex_b = 16b_0010011110011111

#### 5.10.4.3 constant bit [16] led_seven_seg_hex_c = 16b_0010111111111011

#### 5.10.4.4 constant bit [16] led_seven_seg_hex_d = 16b_0111101101101101

#### 5.10.4.5 constant bit [16] led_seven_seg_hex_e = 16b_1111110101000101

#### 5.10.4.6 constant bit [16] led_seven_seg_hex_f = 16b_1101111101110001

#### 5.10.4.7 constant bit [16] led_seven_seg_hex_g = 16b_1110111101111100

## 5.11 cdl/inc/srams.h File Reference

SRAM modules used by all the modules.

### 5.11.1 Detailed Description

SRAM modules used by all the modules. 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.

### 5.11.2 Modules

#### 5.11.2.1 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]* )

#### 5.11.2.2 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]* )

#### 5.11.2.3 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]* )

#### 5.11.2.4 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]* )

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

5.11.2.6 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]* )

5.11.2.7 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]* )

5.11.2.8 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]* )

5.11.2.9 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]* )

5.11.2.10 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]* )

5.11.2.11 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]* )

5.11.2.12 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]* )

5.11.2.13 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]* )

## 5.12 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 }

### 5.12.1 Data Structure Documentation

#### 5.12.1.1 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 |

#### 5.12.1.2 struct t_teletext_character

**Data Fields**

| | | |
|---|---|---|
| bit[7] | character | |
| bit | valid | |

#### 5.12.1.3 struct t_teletext_rom_access

**Data Fields**

| | | |
|---|---|---|
| bit[7] | address | |
| bit | select | |

#### 5.12.1.4 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 |

### 5.12.2 Enumeration Type Documentation

#### 5.12.2.1 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

### 5.12.3 Modules

#### 5.12.3.1 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 |

## 5.13 cdl/inc/utils.h File Reference

Header file for utilities.

### 5.13.1 Detailed Description

Header file for utilities. 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 CDL modules for generic utilities

### 5.13.2 Modules

#### 5.13.2.1 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 |

## 5.14 cdl/README.md File Reference

# Index