APB

Free ware, 2015
Fen Logic Ltd.

Description

The ARM APB interface is used by many IP providers. In this directory you will find various APB related code examples. The docs directory has a copy of the ARM APB AMBA specification.

apb_regs:

A basic example how to read status signals and generate control signals from the APB bus. It has read/write bits, read status bit and static read bits.

apb_regs1:

A small variation on the apb_regs code where read-only and read-write bits are combined in the same address.

apb_wrtsetclr.v:
An example of *write-bits-to-set* and *write-bits-to-clear* code.

apb_pulse.v:
An example how to generate a pulse which is high for one apb-clock cycle. All pulses generated here appear in the cycle ***after*** the APB access has completed.

apb_fifo.v:
An example how to access a synchronous FFO from the APB bus. It shows how to write to a FIFO, read from a FIFO and return the FIFO status signals. It also uses the 'pulse' code to generate a FIFO clear pulse.

apb_levirq.v:
Example of dealing with level interrupts. The module has four incoming interrupts a control register (for interrupt enables) and a status register (for incoming interrupt status bits and pending interrupts status).

apb_edgeirq.v:
Variant of the above. This module detects, stores and deals with interrupt pulses. The module has four incoming interrupts a control register (for interrupt enables) and a status register (for incoming interrupt status bits and pending interrupts status).

apb_decode.v:
This module is an address decoder which splits the APB bus in several regions. For details see the section about address decoder below.

apb_fastdecode.v:
Variant of the above. This decoder has lex flexibility in addressable regions, but is much smaller and faster.

apb_bus.v:
This is a behavioural model of the APB bus which can generate APB read and write cycles. It is extensively used in the test-benches which test all the above mentioned code.

Write-to-set/clear

Write-to-set and write-to-clear register are used mostly in multi-threaded/ multi -tasking applications. It allows independent (or a-synchronous) pieces of software to share one control register. The alternative is that the user must use semaphores to change a register which is very expensive in terms of software cycles and resources.

A write-to-set register sets only those bits high which the user is writing as high. Assume a register with the value 0xAA00FF00. Performing a write-to-set on that register with the value 0x55555555 will set all even bits. The odd bits will remain unchanged. Thus the result will be 0xFF55FF55.

A write-to-clear register clears only those bits high which the user is writing as high. Assume a register with the value 0xAA00FF00. Performing a write-to-clear on that register with the value 0x55555555 will clear all even bits. The odd bits will remain unchanged. Thus the result will be 0xAA005500.
*Beware that where the user writes a **one**, the result in the register bit will be a **zero!***

Interrupts

The interrupt modules have the following register mapping:

Address offset	Description	Type	Notes
0x00	Control	R/W	Has the four interrupt enable bits
0x04	Status	RO	Has eight interrupt status bits
0x04	Clear	WO	Interrupt clear bits (apb_edgeirq.v only)

The control register for both the apb_levirq and apb_edgeirq has the following organisation:

0x0: Reset = 0x00000000	
Bits	**Description**
31:4	Unused/reserved
3	Enable for interrupt_request[3] 0 : Interrupt is disabled 1 : Interrupt is enabled
2	Enable for interrupt_request[2] 0 : Interrupt is disabled 1 : Interrupt is enabled
1	Enable for interrupt_request[1] 0 : Interrupt is disabled 1 : Interrupt is enabled
0	Enable for interrupt_request[0] 0 : Interrupt is disabled 1 : Interrupt is enabled

The status register of the apb_levirq has the following organisation:

0x4 (Read Only): Reset = 0x00000000	
Bits	**Description**
31:8	Unused/reserved
7	interrupt pending[3] : interrup request[3] is 1 & interrupt_enable[3] is 1 0 : interrupt_pending[3] is low 1 : interrupt_pending [3] is high (generating an interrupt)
6	interrupt pending[2] : interrup request[2] is 1 & interrupt_enable[2] is 1 0 : interrupt_pending[2] is low 1 : interrupt_pending [2] is high (generating an interrupt)
5	interrupt pending[1] : interrup request[1] is 1 & interrupt_enable[1] is 1 0 : interrupt_pending[1] is low 1 : interrupt_pending [1] is high (generating an interrupt)
4	interrupt pending[0] : interrup request[0] is 1 & interrupt_enable[0] is 1 0 : interrupt_pending[0] is low 1 : interrupt_pending [0] is high (generating an interrupt)
3	Status of interrupt_request[3] input 0 : interrupt_request[3] is low 1 : interrupt_request[3] is high
2	Status of interrupt_request[2] input 0 : interrupt_request[2] is low 1 : interrupt_request[2] is high
1	Status of interrupt_request[1] input 0 : interrupt_request[1] is low 1 : interrupt_request[1] is high
0	Status of interrupt_request[0] input 0 : interrupt_request[0] is low 1 : interrupt_request[0] is high

The status register of the apb_edgeirq has the following organisation:

0x4 (Read, Write clear): Reset = 0x00000000	
Bits	**Description**
31:8	Unused/reserved
7	interrupt pending[3] : edge_seen[3] is 1 & interrupt_enable[3] is 1 0 : interrupt_pending[3] is low 1 : interrupt_pending [3] is high (generating an interrupt)
6	interrupt pending[2] : edge_seen[2] is 1 & interrupt_enable[2] is 1 0 : interrupt_pending[2] is low 1 : interrupt_pending [2] is high (generating an interrupt)
5	interrupt pending[1] : edge_seen[1] is 1 & interrupt_enable[1] is 1 0 : interrupt_pending[1] is low 1 : interrupt_pending [1] is high (generating an interrupt)
4	interrupt pending[0] : edge_seen[0] is 1 & interrupt_enable[0] is 1 0 : interrupt_pending[0] is low 1 : interrupt_pending [0] is high (generating an interrupt)
3	Read: Status of edge_seen[3] register bit 0 : edge_seen[3] is low 1 : edge_seen[3] is high Write : Change edge_seen[3] register bit 0 : edge_seen[3] remains unchanged 1 : edge_seen[3] is cleared
2	Read: Status of edge_seen[2] register bit 0 : edge_seen[2] is low 1 : edge_seen[2] is high Write : Change edge_seen[2] register bit 0 : edge_seen[2] remains unchanged 1 : edge_seen[2] is cleared
1	Read: Status of edge_seen[1] register bit 0 : edge_seen[1] is low 1 : edge_seen[1] is high Write : Change edge_seen[1] register bit 0 : edge_seen[1] remains unchanged 1 : edge_seen[1] is cleared
0	Read: Status of edge_seen[0] register bit 0 : edge_seen[0] is low 1 : edge_seen[0] is high Write : Change edge_seen[0] register bit 0 : edge_seen[0] remains unchanged 1 : edge_seen[0] is cleared

Decoders

The decoder modules splits the APB bus in a number of address regions. It does this by:

- Splitting the psel, making one for each region
- Multiplexing the result signals: prdata, pready, pslverr.

Each module has a PORTS parameter which specifies how many regions the bus should be split into.

Additionally each module has a TOP_DEFAULT parameter.

- If that is set to zero all addresses which are out of the specified address range generate a **`pslverr`**.
- If that is set to one all addresses which are out of the specified address range are handled by the highest port.

prdata.

Unfortunately interfaces or arrays of ports are not supported by most simulation let alone synthesis tools. Thus the prdata return path is made up from one very wide port:

```
input [PORTS*32-1:0] prdata,
```

You have to connect all return data path as one big concatenation:

```
  .prdata { <port n bus>,... <port 0 bus> },
```

My preferred method dealing with that is to use an array and connect that up:

```
wire [31:0] m_prdata [0:3];
...
.m_prdata ( {m_prdata[3],
             m_prdata[2],
             m_prdata[1],
             m_prdata[0]}
          ),
```

For the full code have a look at the test benches.

Pass through

The module has ***a lot*** of signals which are passed through. Some companies don't like that. But as the full source code is available you can re-write them.

apb_decode.v

The **apb_decode.v** module has two parameters to specify the outgoing address range:
BOTREGION : Sets the start address of port[0] (psel[0])
REGION : Sets the size of each port.
The table below shows how this splits the address map for a 4 port module:

Port	Lowest address (Inclusive)	Highest address (Exclusive)
0	BOTREGION	BOTREGION+REGION
1	BOTREGION+REGION	BOTREGION+2*REGION
2	BOTREGION+2*REGION	BOTREGION+3*REGION
3	BOTREGION+3*REGION	BOTREGION+4*REGION

Thus port[0] starts at address BOTREGION and ends just below BOTREGION+REGION.

apb_fastdecode.v

The **apb_fastdecode.v** module use significant less logic and is thus much faster than the normal decoder. It is limited in other ways:

- An address region can only be a power of two.
- The first address region always starts at zero.
- It does NOT do a full address decode thus ports can appear multiple mapped.

It has one parameter to specify the outgoing address range:
MS_SLVADR : Set the MS address bit going to a port.
The table below shows how this splits the address map for a 4 port module for two different values of MS_SLVADR:

Port	Lowest address SLVADR=10	Highest address SLVADR=10	Lowest address SLVADR=15	Highest address SLVADR=15
0	0x0000_0000	0x0000_07FC	0x0000_0000	0x0000_FFFC
1	0x0000_0800	0x0000_0FFC	0x0001_0000	0x0001_FFFC
2	0x0000_1000	0x0000_17FC	0x0002_0000	0x0002_FFFC
3	0x0000_1800	0x0000_1FFC	0x0003_0000	0x0003_FFFC

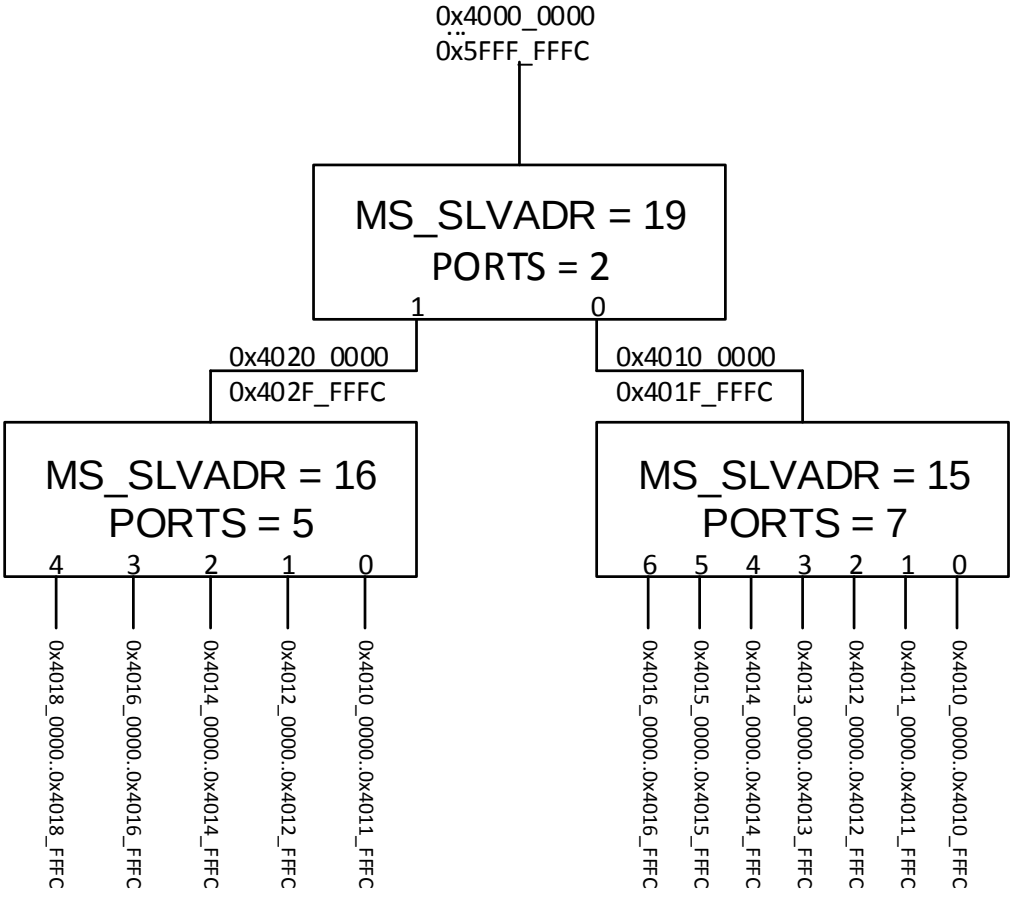
If there is no pre-decoder which limits the incoming s_psel this will repeat:

Port	Lowest address SLVADR=10	Highest address SLVADR=10	Lowest address SLVADR=15	Highest address SLVADR=15
0	0x0000_2000	0x0000_27FC	0x0004_0000	0x0004_FFFC
1	0x0000_2800	0x0000_2FFC	0x0005_0000	0x0005_FFFC
2	0x0000_3000	0x0000_37FC	0x0006_0000	0x0006_FFFC
3	0x0000_3800	0x0000_3FFC	0x0007_0000	0x0007_FFFC
0	0x0000_4000	0x0000_47FC	0x0008_0000	0x0008_FFFC
	etc.			

It is common to split the address map in regions and sub regions.

Thus the first fast decoder can use MS_SLVADR = 19 splitting the APB bus in regions of 1 Mbytes.

Then each of those can have a fast decoder using MS_SLVADR = 15 / 16 splitting each 1 Mbyte region in sub regions of 64 / 128Kbytes.



APB_BUS

The **apb_bus.v** module is a behaviour model which generates APB read and write cycles. The apb address bus has been limited to 16 bits which is sufficient for most test-benches.

Parameters

The **apb_bus**.v module has two parameters:

- CLKPER : The clock period of the apb **`pclk`** signal.
- TIMEOUT : The number of clock cycles the model waits for **`pready`** to go high. If not the code time-out and stops the simulation with an error message.

Tasks

The module has a number of internal tasks which the user can call to perform reads or writes. For example usage just look at the test benches provided which test each of the APB example modules.

Beware that the read and write task 'post' a read or write request and return immediately after the post was successful. *Thus if the tasks returns, the read or write bus cycle still has to happen.* If the bus is busy with a previous cycle, the tasks waits until that cycle has started and then post the request and returns.

write task.

```
task write;
input [15:0] address;
input [31:0] data;
```

Post a write request.
Example usage: **`apb_bus0.write(16'h0000,32'h00400001);`**

read task.

```
task read;
input [15:0] address;
input [31:0] data;
```

Post a read request and can perform a read data check. The '**`data`**' argument is what is expected to be read. Bits which are unknown or do not need to be checked can be set to 'x'. Thus to ignore all the data use: 32'hxxxxxxxx for data.
Example usage: **`apb_bus0.read(16'h0000,32'hxx40xxx1);`**
If the read data does not match the module gives an error message. it also increments n internal **`errors`** counter. e.g.

```
if (apb_bus0.errors!=0)
   $display("%m: Found errors");
```

delay task.

```
task delay;
input integer cycles;
```

This task waits until the APB bus has finished and then waits a number of cycles.
`cycles` must >=1