MEMORIES

Free ware, 2017
Fen Logic Ltd.

Description

The memories directory contains a set of various memory models. All models are synchronous memories. A-synchronous memories are rarely found these days. All memories have parameters to control the width and depth. At the moment the memories only support a depth which is a power of two. Most models will map on Xilinx embedded memory macros. As usual the code comes with some basic test-benches which I used to verify the memories. In this case the test-benches are NOT self-checking.

spm:
Single ported memory.

spmbe:
Single ported memory with byte write enables.

tpm.v:
Two port memory. This is a dual-ported memory with one read and one write port. Two port memories are slightly smaller then dual-ported memories but not much. Not all silicon manufacturers offer two-port memories.

bpm.v:
Dual ported memory. This is a memory with two independent ports both can perform reads or writes.

async_read_memory.v:
(Added 12 dec 2017)
This is a dual ported memory with a synchronous write. But the read is performed a-synchronously. Thus new data will come out as soon as the read address changes. Even if no clock edge is present. The module can be synthesized for FPGA but will use a lot of resources. It is mainly provided for behavioural code, e.g. test benches.

Parameters

All memories have parameters to control the data width and depth.

- WIDTH : The data width in bits.
  or
- BYTES : The data width in bytes (used in **spmbe** only).
- L2D: The memory depth in Log-2 data units.
  L2D=8 gives a memory with 2^8 = 256 entries. This also specifies the address width as having L2D bits.
- CLEAR: The memory contents is set to all zeros at start-up.
  This does NOT reflect the real behaviour and as such I don not recommend you use it for real embedded memories. But it is ideal for many behavioural test-bench memories.
- FILE: The name of the file to load the memory with at start-up. If you do not need/want a file set the parameters to an empty string: **""**. The code uses $readmemh(...) to load the file. Beware that any path of the file must be relative to the simulation location. Not the location of the module!

The following parameters are only present in the **tpm** and **dpm** memory models. For the details see the section 'Clashes' further on.

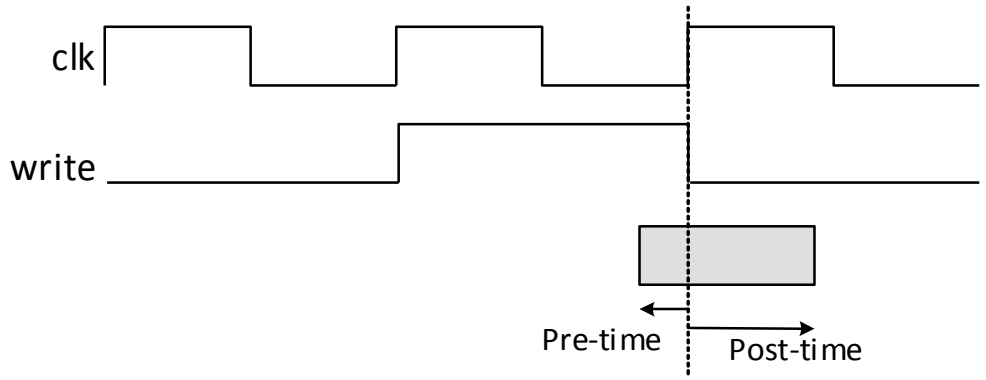
- CHECK: If set to 1 enables write clash checking.
- PRETIME: Pre-rising clock check time.
- POSTTIME: Post-rising clock check time.

Clashes

A dual-port or two port memory has the potential of clashes between the two ports. A clash may occur when a port is being written (from either side). The data will need time to change. A read from the location can result in any mix of the old and new data. A write of the same location from two ports, especially with different data, is also a recipe for disaster.

The **tpm** and **dpm** memories have a built-in check mechanism for this. The checks are switched on when the CHECK parameter is set to 1.

There are two parameters which specify the time where no other read or write access to that location is allowed: PRETIME and POSTTIME. The following figure depicts how they work.



- PRETIME specifies a 'critical period' before the rising clock edge.
- POSTTIME specifies a 'critical period' after the rising clock edge.

The times can be set to zero but negative values are not allowed.

This behaviour is only an approximation of the real behaviour. For final system checks you should always use the manufacturer memory models with all timing value applied (e.g. Use SDF timing annotation)

File format

This is just a reminder how the file format for the $readmemh()..task should look like.

- You can have comments in the file. This fact this is often forgotten but allows you to specify what the file is inside the file itself.
- You can specify an address with the '@' symbol. The address is always relative to the start of the file. If no address is specified loading start with the first (0) location
- Data must be in hexadecimal.

Example:

```
// Example memory load file
@64
00
01
02
```

SET

I have kept the memory models as simple as possible. A useful task I often add to the memories is the 'set' task. It allows pre-loading a memory with a regular pattern. This is very useful for generating test data.

```
//
// Fill the memory with an incrementing pattern
// using start=0 and increment=0 clears the memory
//
task set;
input reg[WIDTH-1:0] start,increment;
integer m;
begin
   for (m=0; m< DEPTH; m=m+1)
   begin
      memory[m] = start;
      start = start + increment;
   end
end
endtask // set
```

For those less familiar with using tasks inside module: you can call the task using the module name. Example:

```
spm #(.WIDTH(16),
      .L2D(12)
      )
     spm_demo (...

initial
begin
   spm_demo.set(0,16'h0001); // Fill with 0,1,2,...
end
```