ECE 436 Laboratory 5

CACHE DESIGN

Description:
Before designing a cache for your Jackal processor, you must explore the design space to find the cache size and organization that gives you the lowest CPI with the smallest cost. You can use the Block RAM for your cache storage at no cost, but you must consider the datapath and control structures that will be part of your processor's area.

Given that CPI and cache performance are program dependent, you must consider a variety of programs to find the overall best cache design (remember: optimize the common). For this lab, you are welcome to do this manually. That is, you can take multiple test programs and walk through their execution, tracking the cache hit rate. If you assume some cache hit and miss access times, you can determine the CPI for the overall program execution for various cache sizes and organizations.

For those of you slightly more daring, you can explore a cache simulation program called Dinero. Information detailing this very useful tool is at:
http://www.cs.wisc.edu/~markhill/DineroIV/
The program takes in memory trace files and simulates their performance on various cache organizations. It enables you to simulate very large programs on multiple cache configurations very quickly.

For either approach, you must evaluate the CPI for at least three cache sizes (with a maximum total cache size of 64 bytes with at most 8 lines of between 2 and 8 bytes each, not including tag, dirty, or valid bits) or organizations on at least three of the original test programs (posted in a .zip file on the toolkit website). You can choose the cache sizes and organizations to try, but make an educated guess what might be the good options. Organizations options include data (instruction, data, unified), associativity, replacement policy, block size, coherency policy (i.e. write-through or write-back for data caches), etc.

When you have selected a good overall cache design, implement it in VHDL. You may use either the Block RAM or the regular CLB structures. Remember to include as necessary tag bits, valid bits, dirty bits, and the datapath structures for the various comparisons.

Good luck!

Deliverables:
We will be expecting the following items in the laboratory report:
- A graph plotting cache sizes/organizations against CPI for each testbench.
- An argument for and a description of the cache design you implemented, including relevant VHDL and schematic files. Include the CLB area of your cache design (i.e. the area that is not in the Block RAM).

For the first bullet, clearly state you assumptions. For example:
- How many cycles does it take each type of instruction to execute?
- How many cycles does it take to access main memory?
- How many cycles does it take to access the cache?
- What is the cache miss penalty (i.e. in addition to the main memory access time)?

Instantiating Block RAM on the Spartan-II FPGA

VINU VIJAY KUMAR (VV6V@VIRGINIA.EDU)

The Block RAM primitives available on the Spartan-II FPGA can be used for building efficient on-chip memory structures. All reads and writes to Block RAMs take a single clock cycle. While the memories can be made dual-ported, we discuss only single-port memories here.

Block RAM memories are instantiated by including the *spblockram.vhd* module (posted on the Toolkit site and reproduced below for reference) as a new source in your project.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity spblockram is
 port (clk : in std_logic;
       we  : in std_logic;
       a   : in std_logic_vector(4 downto 0);
       di  : in std_logic_vector(15 downto 0);
       do  : out std_logic_vector(15 downto 0));
end spblockram;

architecture syn of spblockram is

 type ram_type is array (31 downto 0) of std_logic_vector (15 downto 0);
 signal RAM : ram_type;
 signal read_a : std_logic_vector(4 downto 0);

 begin
 process (clk)
   begin
       if (clk'event and clk = '1') then
              if (we = '1') then
                     RAM(conv_integer(a)) <= di;
              end if;
              read_a <= a;
       end if;
 end process;
  do <= RAM(conv_integer(read_a));
 end syn;
```

When synthesized, the module shown above becomes a memory with 32 words, each of width 16-bits. The code can be modified to build memories of different sizes by changing the size of the address vector *a* (i.e. size 4 downto 0 in the code above). The *read_a* signal also needs to be changed accordingly.

When organized as 16-bit data width memories, each block RAM can have a maximum depth of 256 words. The XC2S100 has 10 of these blocks. If the instantiated memory depth is greater than 256 words, then the synthesis tool automatically combines blocks to build the memory required. For example, if a memory depth of 1024 words is instantiated, the tool combines four blocks to build the memory.