Computer Architecture Project 2

Pipelined CPU with L1 Data Cache Implementation

R E P O R T

B03902001 資工三 駱定暄
B03902027 資工三 王冠鈞
B03902039 資工三 施秉志

- Coding Enviroment

  1. System: 217 Workstation (Linux)
  2. Editor: Vim
  3. Collaborate via Github

- Module Implementation / Modification Details

  Most of the modules are reused without modification from the previous project. Some of the modules are newly added into this project, but no need to modify (e.g. the memory part of the cache). The following are the main modules that are modified in this project.

  1. CPU (CPU.v): As the figure in page 6 of the spec, we replaced data memory with a data cache, which may output stall signals to PC and all the latches. Most of the latches are wired with the stall output directly, except for PC and IFID. Since they may receive a stall signal from either the HDU or the cache, the two signals will first pass an OR gate, and then pass the OR'd result into the stall signal inputs. In other words, a stall signal from either source will make the PC and IFID stall, but only the cache will decide that whether other latches will stall or not.
  2. Cache Controller (dcache_top.v): This is the main part of this project. See the next section.
  3. Pipeline Latches (IDEX.v, EXMEM.v, MEMWB.v): Add stall signal inputs (Stall_i), which are directly from the cache. The latches will stall, by keeping the previous values, when the stall signal in on.

- Detailed Implemenatation of The Cache Controller

  The cache is the major part in this project. The cache from the sample file consists of 3 counterparts: the cache controller (dcache_top.v), the SRAM for tags (dcache_tag_sram.v), and the SRAM for data (dacahe_data_sram.v). Both the two SRAMs have been completed, while the cache controller needs more implementation. The total (added) implementation consists of 4 parts: the tag comparator, read data, write data and the state controller. Now we'll introduce the detailed implementation of these parts.

  1. Tag Comparator: Check if the block is valid and if the tag bits are the same. If they are, enable the block entry to for reading or writing.
  2. Read Data: Get data with a size of one word (4 bytes, 32 bits) according to the offset given.
  3. Write Data: Write data with a size of one word (4 bytes, 32 bits) according to the offset given.
  4. Controller: Control the SRAM and the data memory according to the current state.

Current State	STATE_MISS	STATE_MISS	STATE_READMISS	STATE_READMISS	STATE_READMISSOK	STATE_WRITEBACK	STATE_WRITEBACK
is dirty	true	false	-	-	-	-	-
is ack'ed	-	-	true	false	-	true	false
mem_enable	1	1	0	-	0	1	-
mem_write	1	0	0	-	0	0	-
write_back	1	0	0	-	0	0	-
cache_we	0	0	1	-	0	0	-
(new) state	STATE_WRITEBACK	STATE_READMISS	STATE_READMISSOK	STATE_READMIS	STATE_IDLE	STATE_READMISS	STATE_WRITEBACK
description	a.	b.	c.	c.	d.	e.	e.

  a. When it gets a (read / write) miss and the block is dirty, the current data must write back and replace it with new data.
  b. Different from a., since it is not dirty, the cache doesn't need to write it back.

c. When the state is read miss, if the cache gets an ACK from the memory, enable the cache to write the memory data to cahce block. Otherwise, keep waiting.
d. After the read miss is handled, set all signals to 0 and return to idle state.
e. When the cache is writing back, if an ACK is got from the memory, go to read miss state. Otherwise, keep waiting.

- Testing Method

1. Simulator: iverilog
2. Wave Viewer: gtkwave (for debug purpose)

We debugged our CPU using the instructions from project 1 & 2.

- Task Distribution

駱定暄：stall mechanism, dcache controller
王冠鈞：companion, report
施秉志：companion, report