Memory subsytem for pipelined RISC-V processor

Ashwin Rajesh (Sr. No 21519)
M.Tech Microelectronics and VLSI Design
Indian Institute of Science, Bangalore

***Abstract*— This report summarises the design of a the memory subsystem for a pipelined RISC-V processor as part of the Processor System Design course by Prof. Kuruvilla Varghese at the Indian Institute of Science, Bengaluru.**

I. Problem Statement

Design and Implement instruction and data caches forthe pipelinedRISC-Vprocessordesigned in lab exercise 2on Digilent BASYS3 FPGA Board.Target Device is Xilinx Artix-7 XC7A35T-ICPG236C (Family Artix-7, Part XC7A35T, Package CPG236, Speed Grade -1).

The instruction cache is 2-way set associative with a block size of four words. The datacache is direct-mapped, write-through and with a block size of four words. The main memory is to be implementedin Block RAM. There is no need to support any burst transferfor filling the cache.

II. Toplevel architecture

The memory subsystem consists of main memory made of a BRAM and separate instruction and data caches. The instruction cache is read-only but data cache needs to handle reads for loads and writes for stores as well. A memory controller manages the arbitration between instruction and data cache reads. Since a write-through scheme is used, all writes to the data memory is immediately forwarded to the main memory.

The caches output enable signals for the memory controller to know when they are requesting for a read and the memory controller responds with a valid signal to the corresponding cache after reading its value from the main memory. The caches need to wait for the memory controller to fetch the data if the other cache has also simultaneously requested for data reads.

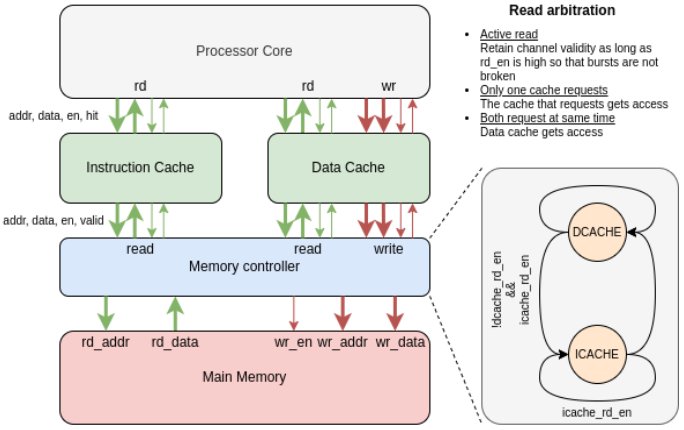


Fig. 1: Memory subsystem

III. Cache blocks

The instruction and data caches were designed in a fully parameterized way. Only the controller is designed separately for each of them. A single “cache block” is first designed. This cache block handles basic reads and writes and abstracts away details of the tag, data and valid arrays. It has an asynchronous read interface where input address is given and the read data is returned along with the hit/miss status. It also has a write interface where write address and write data is given along with a write enable and “write new” signals. The “write new” signal writes the input tag into the tag array and sets the appropriate valid bit in the valid array. The valid and hit status of the write address is also asynchronously read

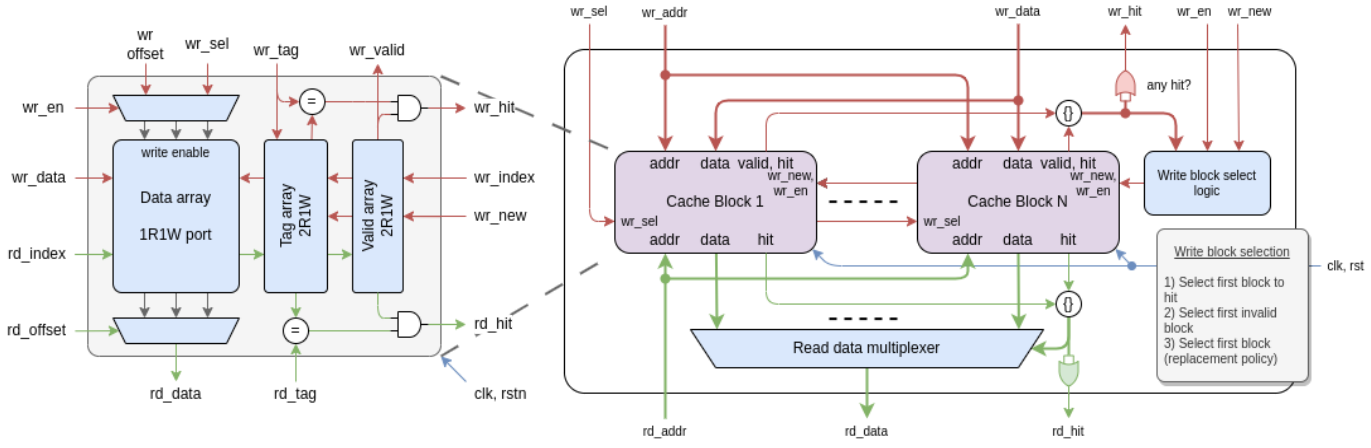


Fig. 2 : Parameterized cache module design

for the cache replacement and write block selection logic.

N such “blocks” are used to create an N-way set associative cache module. This module further abstracts away details of choosing which line in a set to write to and multiplexing from different lines in a set. It has a read multiplexer and write enable logic that selects which “block” the data will be written to and where the data is read from (depending on read hit signals). This cache module is then used in the instruction and data caches along with some control circuits.

IV. Instruction Cache

The instruction cache is a read-only cache memory. The controller handles read misses. If there is a read miss, the controller starts a counter that stores the word offset. The counter increments every time the memory interface gives a valid signal, indicating a successful read. The address decremented by one is given to the cache write port. It is decremented by one because of the two cycle BRAM access latency. The controller has an idle and read state. It comes out of the read state when the counter overflows beyond the offset size, indicating that the whole cache line was read. The cache is 2-way set associative and has a total of 16 cache lines.

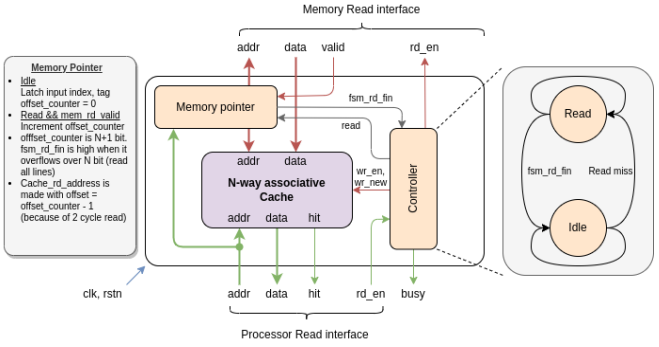


Fig. 3 : Instruction Cache

The processor gets a hit signal and a busy signal from the cache. The busy signal indicates that the cache is busy fetching a cache line from memory. The processor needs to ensure that it fetches instructions only when the hit signal is high. Else, it inserts bubbles in the pipeline.

V. Data Cache

The data cache is read-write memory to handle both loads and stores. The FSM has an additional write state which is used only if the memory expresses that it cannot presently handle any writes. This will be useful if the BRAM is to be replaced with another level of cache.

For handling writes, the cache directly forwards writes to the memory in the same cycle, since it is a write-through cache. If the memory is unable to write which is expressed by lowering its write_rdy signal, the FSM goes into the write state and stays there till it is able to write. If it a write miss, the FSM goes into the read state and reads a cache line into the cache. This will contain the updated values since the read happens after a write. The cache has 16 cache lines.

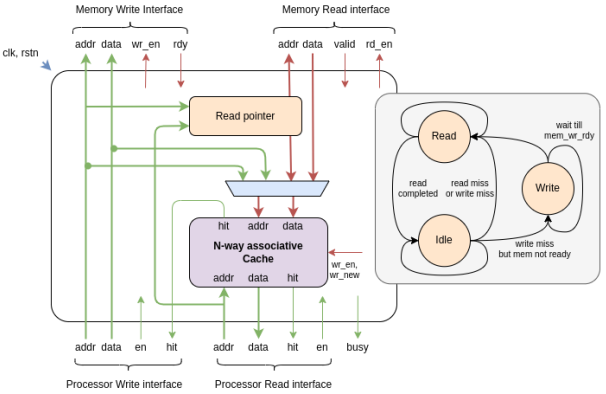


Fig. 4 : Data cache

The processor gets hit signal for both reads and writes and a busy signal. The processor should proceed with a load only after the read hit signal has gone high and with a store only after the write hit signal has gone high. Else, it should stall the whole pipeline. Inserting bubbles in the memory stage is not enough because the forwarding logic in the execute stage will not be able to forward values from instructions in the writeback or after the writeback stage when those instructions are flushed out by the pipeline bubbles.

VI. Processor Pipeline

In the processor pipeline, we need to add more stall conditions. Instruction cache misses will result in fetch stage bubbles and data cache write or read misses will result in pipeline stalls. Apart from this, the read alignment logic will move from the writeback stage to the memory stage since the datacache performs asynchronous reads.

VII. Results

The design was implemented and tested on the basys3 board for generating numbers from the fibonacci sequence at a maximum clock frequency of 75MHz. The worst negative slack was 0.640ns giving a maximum frequency of **78.78MHz**. About 15.67% of LUTs, 13.59% of flip flops and 4% of BRAM resources were used for implementing the design.

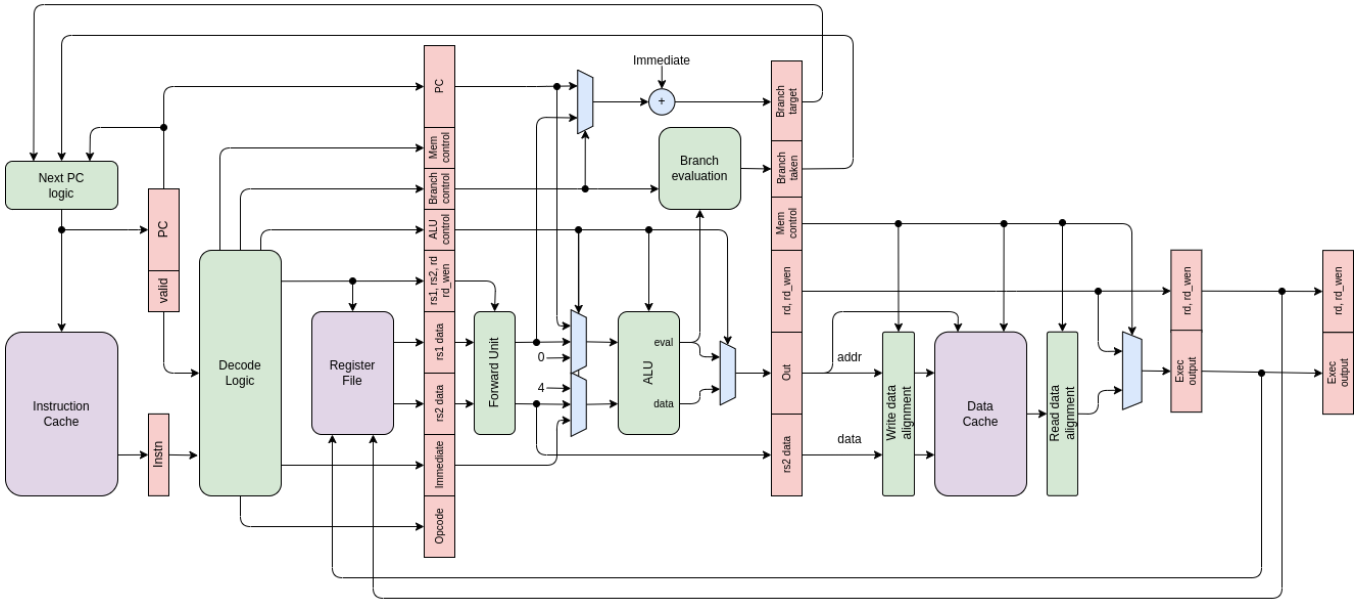
VIII. References

[1] The RISC-V Instruction Set Manua, *Andrew Waterman, SiFive inc*

[2] Digital Design and Computer Architecture, RISC-V Edition, *Sarah L Harris, David Money Harris*

[3] Computer Organization and Design "The Hardware/Software Interface: RISC-V Edition" *David A. Patterson, John L. Hennessy*

APPENDIX



Pipelined RISC-V datapath with cache

Design Timing Summary

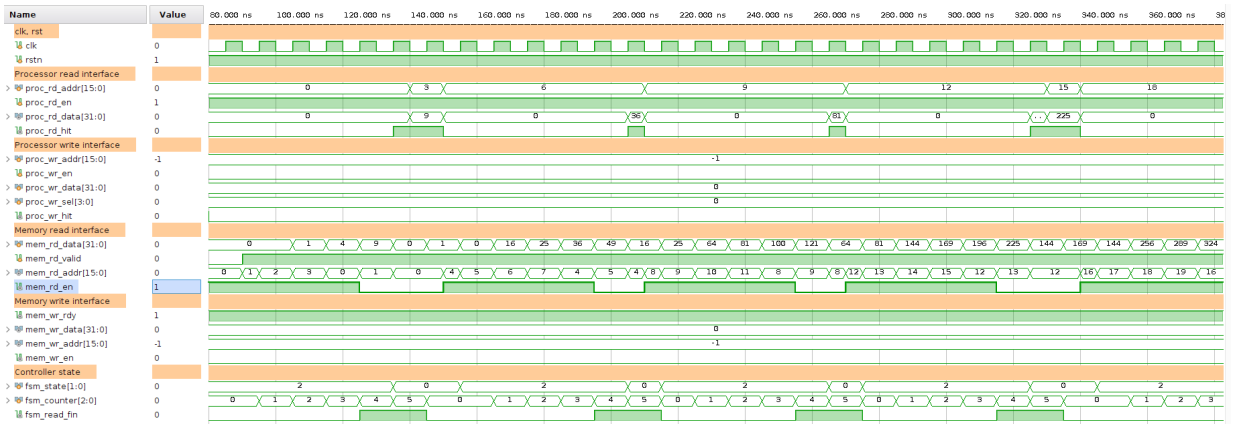
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.117 ns	Worst Hold Slack (WHS):	0.055 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4097	Total Number of Endpoints:	4097	Total Number of Endpoints:	1454

All user specified timing constraints are met.

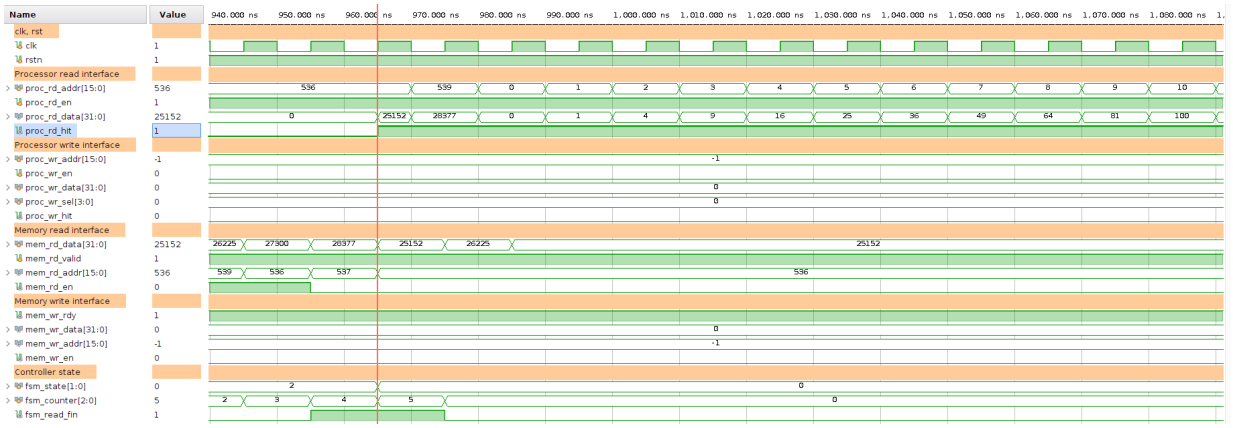
Timing report summary

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
N toplevel	3259	5652	773	96	2587	3225	34	2	14	2
btnc_dbnc (debouncer)	9	21	0	0	10	9	0	0	0	0
btnu_dbnc (debouncer_0)	11	21	0	0	12	11	0	0	0	0
clkdiv_inst (clk_wiz_0)	0	0	0	0	0	0	0	0	0	2
proc_inst (rv32i)	3191	5593	773	96	2563	3157	34	2	0	0
alu (alu)	240	0	0	0	118	240	0	0	0	0
dcache_inst (data_cache)	831	2082	257	96	1007	821	10	0	0	0
icache_inst (instn_cache)	752	2082	260	0	932	728	24	0	0	0
mem_inst (block_mem)	49	11	0	0	28	49	0	2	0	0
reg_file (regfile)	944	992	256	0	533	944	0	0	0	0
seven_seg_out_inst (seven_segment_out)	49	17	0	0	21	49	0	0	0	0

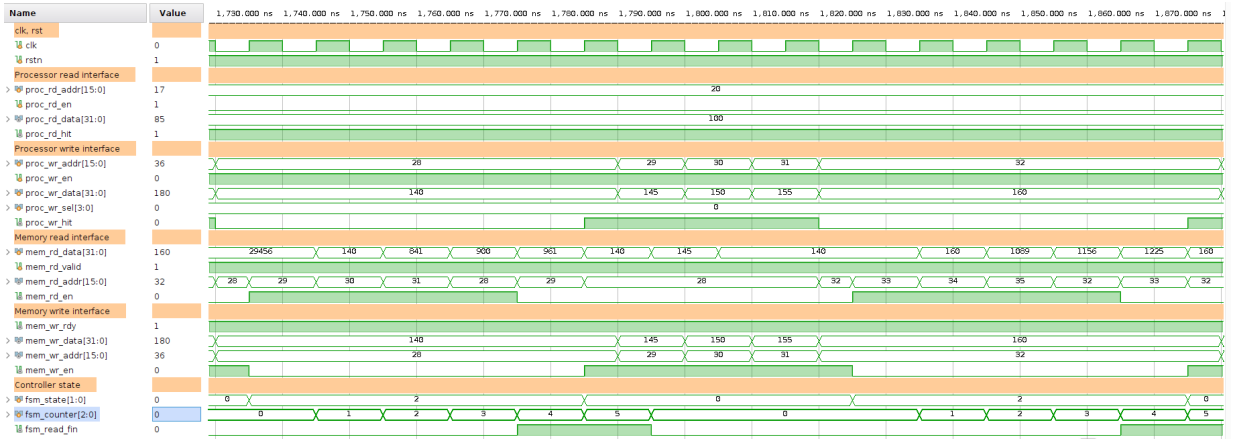
Utilization report summary



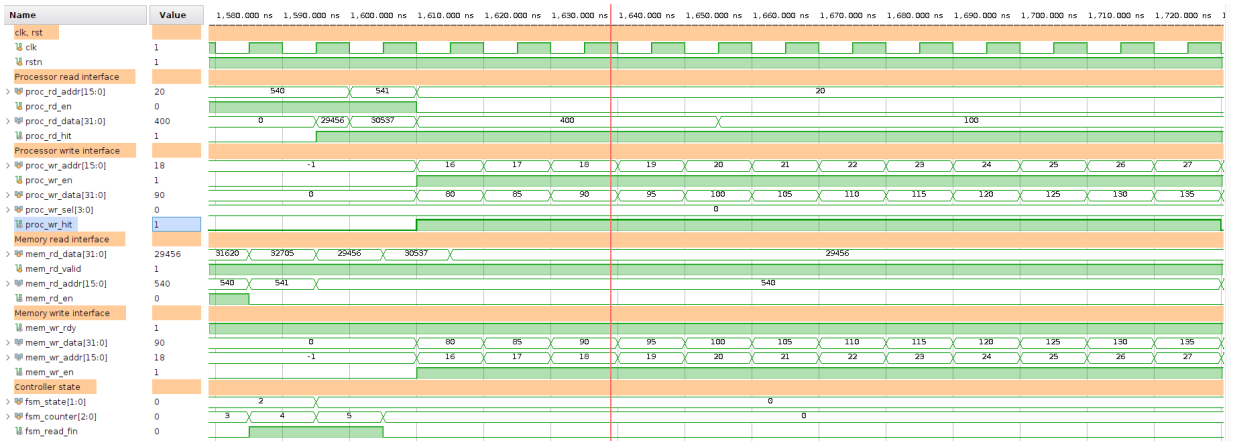
Data cache read misses (instruction cache reads work identically)



Data cache read hits (instruction cache reads work identically)



Data cache write misses



Data cache write hits

```c
void delay();
register int i asm ("s11");
register int d asm ("s10");

asm ("addi sp, s0, 1024");

void delay(int);

int main() {
    int i_prev = 0;
    i = 1;
    d = 10000000;
    while(1) {
         delay(d);
         i      = i + i_prev;
         i_prev = i - i_prev;
     }
}

void delay(int delay) {
     for(int i = 0; i < delay; i++);
}
```

C code used for testing the processor (fibonacci)

```
addi sp, s0, 1024

main:
  addi sp,sp,-32
  sw ra,28(sp)
  sw s0,24(sp)
  addi s0,sp,32
  sw zero,-20(s0)
  li s11,1
  li a5,9998336
  addi s10,a5,1664
.L2:
  mv a5,s10
  mv a0,a5
  call _Z5delayi
  mv a4,s11
  lw a5,-20(s0)
  add a5,a4,a5
  mv s11,a5
  mv a4,s11
  lw a5,-20(s0)
  sub a5,a4,a5
  sw a5,-20(s0)
  j .L2

_Z5delayi:
  addi sp,sp,-48
  sw s0,44(sp)
  addi s0,sp,48
  sw a0,-36(s0)
  sw zero,-20(s0)
.L5:
  lw a4,-20(s0)
  lw a5,-36(s0)
  bge a4,a5,.L6
  lw a5,-20(s0)
  addi a5,a5,1
  sw a5,-20(s0)
  j .L5
.L6:
  nop
  lw s0,44(sp)
  addi sp,sp,48
  jr ra
```

Assembly code used
(generated with gcc, has some pseudo instructions)