Verilog Implementation Report

Computer Organization Project 7

Jaewon Chung, 2015-11832
2019/05/26

1. Introduction

- In this project, I will implement the following:
  - A baseline CPU that communicates with a 2-latency memory
  - A cached CPU that communicates with a 4-latency memory
- From this, I will deeply understand the structure of caches, and analyze its performance relative to the baseline CPU.

2. Design

- Cache structure: Diagram, structure, and algorithm were all provided in the lecture and lab slides.
  - I planned to implement separate caches for instruction and data.
  - The instruction cache prefetches the next block of instructions when it is idle.
  - The data cache employs the write-back / write-allocate cache policy.
- Timing: This I tried to design carefully. Refer to the attached design document.

3. Implementation

- **Instruction cache**
  - Outputs instruction and `InstMemBusy` signal. The `InstMemBusy` signal works like a micro-architectural state; the cache may be busy due to prefetching, but InstMemBusy acts as if the cache serves only the requests from the CPU.
  - Cache hit is serviced combinationally. `InstMemBusy` is not asserted at all.
  - If cache miss occurs, `InstMemBusy` is asserted, and the CPU knows that it should stall.
  - Generally at every `posedge clk`:
    - **If** data is being requested and cache misses: access the memory. `InstCacheCounter` is the memory access latency counter.
    - **Else**, prefetch the next block of the most recently fetched block, if it wasn't already prefetched. `PrefetchCounter` is the memory access latency counter.
  - A corner case: the cache may be prefetching when an instruction read is requested by the CPU.
    - **If** the cache is prefetching the block requested, the `InstCacheCounter` takes over the `PrefetchCounter` and fetch continues.
    - **Else**, `i_readM` is first deasserted. This cancels the prefetch. Then at the next cycle, the cache begins memory access to the correct block. Hence the prefetching scheme may introduce an additional latency of 1 cycle.

- **Data Cache**
  - Outputs data and `DataMemBusy` signal.
  - Read/write hit is serviced combinationally. `DataMemBusy` is not asserted at all.
  - If cache miss occurs, Data`MemBusy` is asserted, and the CPU knows that it should stall.
  - At every `posedge clk`:
    - **If** cache there is a request pending and the cache missed:
      - **If** the evict signal is high, first evict the cache line to the memory.
      - **Else**, this means eviction is done. Fetch the cache line from the memory.
    - **Else**, do nothing.
  - In order to begin memory read immediately after eviction is completed, eviction is handled (i.e. the valid and dirty bit is cleared) at the clock negative edge. Then at the next cycle positive edge, memory read can begin.

4. Discussion

Results and Experiments

	Baseline 1	Baseline 2	Cache 1	Cache 2	Cache 3
I-Mem Latency	2 cycles	4 cycles	1 ~ 6 cycles	1 ~ 6 cycles	1 ~ 7 cycles
D-Mem Latency	2 cycles	4 cycles	4 cycles	1 ~ 10 cycles	1 ~ 10 cycles
I-Cache Hit Ratio	-	-	0.8405	0.8405	0.8919
D-Cache Hit Ratio	-	-	-	0.9756	0.9756
Total Clock #	**2736**	**5180**	**3289**	**2756**	**2578**

- Baseline 1: submitted version
- Baseline 2: only memory latency changed to 4 cycles from Baseline 1
- Cache 1: instruction cache
- Cache 2: instruction cache + data cache
- Cache 3: submitted version. instruction cache with prefetching + data cache

Hit ratio was calculated on every PC or address that requested data to the cache.

Cache 2 didn't really improve performance. This may be because the baseline CPU already had many stalls due to data hazards. That is, increasing the memory latency to 2 in the pipelined baseline CPU wasn't a big performance impact for it, since it had to stall anyway at the ID stage due to data dependency. Hence even if the cache provided instructions immediately on request, the pipeline stalled anyway.

5. Conclusion

- I believe I have achieved every goal specifeid in the introduction.

Timing design

1. Baseline

LWD

Cycles	IF	ID	EX	MEM	WB
1	-				
2	LWD1	x			
3	-	LWD1	x		
4	LWD2	x	LWD1	x	
5	-	LWD2	x	LWD1	x
6	SUB	LWD2	x	LWD1	x
7	-	SUB	LWD2	x	LWD1
8	ADDI	x	SUB	LWD2	x
9	ADDI (in IR)	x	SUB	LWD2	x
10		ADDI	x	SUB	LWD2

Memory Access
- Stalls IF, ID, and EX stage.
- Fetched instruction in the IF stage is kept in the IR register.

Data Hazard

Cycles	IF	ID	EX	MEM	WB
1	-				
2	ADD	x			
3	-	ADD	x		
4	SUB	x	ADD	x	
5	-	SUB	x	ADD	x
6	-	SUB	x	x	ADD
7	-	SUB	x	x	x
8	ADDI	x	SUB	x	x
9		ADDI	x	SUB	x

DataHazard asserted during ID stage
- Immediately:
  - i_readM deasserted
- Next posedge clk:
  - IF/ID latching disabled; IF/ID keeps its value
  - ID/EX latch flushed
  - PC Write disabled

Jump Misprediction

- No DataHazard

Cycles	IF	ID	EX	MEM	WB
1	-				
2	JMP	x			
3	-	JMP	x		
4	-	x	JMP	x	
5	ADD	x	x	JMP	x
6		ADD	x	x	JMP

- With DataHazard

Cycles	IF	ID	EX	MEM	WB
1	-				
2	JPR	x			
3	-	JPR (DataHazard)	x		
4	-	JPR (Misprediction)	x	x	
5	-	x	JPR	x	x
6	ADD	x	x	JPR	x
7		ADD	x	x	JPR

JumpMisprediction asserted during ID stage
- Immediately:
  - i_readM deasserted (cancels current fetch if InstMemBusy==1)
  - correct nextPC calculated
- Next posedge clk:
  - IF flushed
  - correct PC latched
  - i_readM asserted
  - InstMemBusy asserted

Branch Misprediction

- Without DataHazard

Cycles	IF	ID	EX	MEM	WB
1	-				
2	BNE	x			
3	-	BNE	x		
4	-	x	BNE	x	
5	-	x	x	BNE	x
6	ADD	x	x	x	BNE
7		ADD	x	x	x

- With DataHazard

Cycles	IF	ID	EX	MEM	WB
1	-				
2	BNE	x			
3	-	BNE (DataHazard)	x		
4	-	BNE	x	x	
5	-	x	BNE (Misprediction)	x	x
6	-	x	x	BNE	x
7	ADD	x	x	x	BNE
8		ADD	x	x	x

BranchMisprediction Asserted during ID stage
- Immediately:
  - i_readM deasserted (cancels current fetch if InstMemBusy==1)
  - correct nextPC calculated (has priority over JumpMisprediction)
- Next posedge clk:
  - IF flushed
  - correct PC latched
  - i_readM asserted
  - InstMemBusy asserted

IF stage

`InstMemCounter`: counts the number of cycles left.
- Branch misprediction / Jump misprediction / Pipeline clean and `InstMemCounter==0`
  => InstMemCounter initialized to `` `LATENCY - 1 ``
- Pipeline clean and `InstMemCounter!=0`
  => InstMemCounter decremented
- When data hazard is asserted, force-set `InstMemCounter` to 0
  When data hazard is deasserted, force-set `InstMemCounter` to `` `LATENCY - 1 ``

`InstMemBusy`: asserted when the instruction memory is accessing the memory

FetchCompletePC: Records the PC most recently fetched successfully

PCWrite: When asserted, latch nextPC to PC.
IFIDWrite: When asserted, latch IF information to IF/ID register.
IFFlush: When asserted, flush IF/ID register.

Fetched instructions are immediately latched into the `IR` register, and fetched data into the `DR` register. Then on the next clock positive edge, they are transfered to `IF_ID_Inst` or `MEM_WB_MemData` if that pipeline register's write is enabled.

ID stage

IDEXWrite: When asserted, latch ID information to ID/EX register.
IDEXFlush: When asserted, flush ID/EX register

EX stage

EXMEMWrite: When asserted, latch EX information of EX/MEM register. Never flush.

MEM stage

DataMemBusy: asserted when the data memory is accessing the memory
DataMemCounter: counts down from (latency - 1) to 0 every clk. We get the data when 1->0.

d_readM: asserted when requesting data to the memory module
d_writeM: asserted when writing data to the memory module
MemCompleteAddr: Records the address most recently read successfully. Flush content after memory write.

MEMWBWrite: When asserted, latch MEM information to MEM/WB register. Otherwise, flush it.

WB stage

No special control. Everything just passes through.

TestBench

num_inst: counts unique instructions that enter the IF stage. Increment whenever IFIDWrite is asserted on clock positive edge.

2. Cache (write-back, write-allocate)

Instruction cache hit
Data cache read hit

Cycles	IF	ID	EX	MEM	WB
1	LWD				
2	ADD	LWD			
3	SUB	ADD	LWD		
4	ADDI	SUB	ADD	LWD	
5		ADDI	SUB	ADD	LWD

Instruction cache hit
Data cache miss (6 cycle on ordinary fetch, 10 cycles on evict & fetch)

Cycles	IF	ID	EX	MEM	WB
1	LWD				
2	ADD	LWD			
3	SUB	ADD	LWD		
4	ADDI (in IR)	SUB	ADD	LWD	
5	ADDI (in IR)	SUB	ADD	LWD	
6	ADDI (in IR)	SUB	ADD	LWD	
7	ADDI (in IR)	SUB	ADD	LWD	
8	ADDI (in IR)	SUB	ADD	LWD	
9	ADDI (in IR)	SUB	ADD	LWD	
10	NOT	ADDI	SUB	ADD	LWD
11		NOT	ADDI	SUB	ADD

While data memory is fetching, instruction fetch is done in cycle 4.
From then on, `FetchCompletePC!=PC` keeps `i_readC` low, preventing the instruction cache from fetching the same PC again.

Instruction cache miss (always 6 cycle delay)
Data cache read miss (6 cycle delay or 10 cycle delay)

1) Data cache 6 cycle delay

Cycles	IF	ID	EX	MEM	WB
1	-	SUB	ADDI	LWD	
...	-	SUB	ADDI	LWD	x
6	NOT	SUB	ADDI	LWD	x
7		NOT	SUB	ADDI	LWD

2) Data cache 10 cycle delay (same situation as instruction cash hit, data cache miss)

Cycles	IF	ID	EX	MEM	WB
1	-	SUB	ADDI	LWD	
...	-	SUB	ADDI	LWD	x
6	NOT (in IR)	SUB	ADDI	LWD	x
7	NOT (in IR)	SUB	ADDI	LWD	x
8	NOT (in IR)	SUB	ADDI	LWD	x
9	NOT (in IR)	SUB	ADDI	LWD	x
10	NOT (in IR)	SUB	ADDI	LWD	x
11	ADD	NOT	SUB	ADDI	LWD
12		ADD	NOT	SUB	ADDI

Instruction cache miss (always 6 cycle delay)
Data cache hit

Cycles	IF	ID	EX	MEM	WB
1	-	BNE	ADDI	LWD	
...	-	x	BNE	ADDI	LWD
6	NOT	x	x	BNE	ADI
7		NOT	x	x	BNE

Data Hazard

Cycles	IF	ID	EX	MEM	WB
1	ADD	ADI			
2		ADD	ADI		
3		ADD	x	ADI	
4		ADD	x	x	ADI
5	SUB	ADD	x	x	x
6		SUB	ADD	x	x

Cycles	IF	ID	EX	MEM	WB
1	JPR	ADI			
2		JPR (DataHazard)	ADI		
3		JPR (DataHazard)	x	ADI	
4		JPR (DataHazard)	x	x	ADI
5	SUB	JPR (Misprediction)	x	x	x
6		SUB	ADD	x	x

Jump misprediction during instruction cache hit

Cycles	IF	ID	EX	MEM	WB
1	JMP				
2	ADD (wrong)	JMP			
3	SUB (correct)	x	JMP		
4		SUB	x	JMP	

Cycles	IF	ID	EX	MEM	WB
1	JMP				
2	ADD (wrong)	JMP			
3	- correct PC	x	JMP		
4	- correct PC	x	x	JMP	
5	- correct PC	x	x	x	JMP
...	- correct PC	x	x	x	x
8	NOT	x	x	x	x
9		NOT	x	x	x

Jump misprediction during instruction cache miss

Cycles	IF	ID	EX	MEM	WB
1	JMP	ADD			
2	- wrong PC	JMP	ADD		
3	- correct PC	x	JMP	ADD	

Mispredicted jump stalled due to data cahce miss

Cycles	IF	ID	EX	MEM	WB
1	JMP	ADD	LWD		
2	- wrong PC	JMP	ADD	LWD	
3	- correct PC	JMP	ADD	LWD	
4	- correct PC	JMP	ADD	LWD	x
5	- correct PC	JMP	ADD	LWD	x
6	- correct PC	JMP	ADD	LWD	x
7	- correct PC	JMP	ADD	LWD	x
8	NOT	x	JMP	ADD	LWD
9		NOT	x	JMP	ADD
10		x	NOT	x	BNE

`PC` doesn't change, but `JumpMisprediction` is asserted the whole time, which keeps re-initializing `InstMemCounter` to LATENCY - 1 every cycle. To solve this problem, we should memoize the last ID_EX_PC that caused a control hazard. On match, we do not initialize `InstMemCounter`.

Branch Misprediction during instruction cache hit

Cycles	IF	ID	EX	MEM	WB
1	BNE				
2	ADD (wrong)	BNE			
3	SUB (wrong)	ADD	BNE		
4	ADDI (correct)	x	x	BNE	
5		ADDI	x	x	BNE

Cycles	IF	ID	EX	MEM	WB
1	BNE				
2	ADD (wrong)	BNE			
3	SUB (wrong)	ADD	BNE		
4	- correct PC	x	x	BNE	
5	- correct PC	x	x	x	BNE
...	- correct PC	x	x	x	x
9	NOT	x	x	x	x
10		NOT	x	x	x

`num_inst` must be decremented by 1. For this, keep a register that remembers whether an instruction was latched into the ID stage last cycle (latch InstMemBusy==0 at posedge clk).

Branch Misprediction during instruction cache miss

Cycles	IF	ID	EX	MEM	WB
1	- wrong PC	BNE	ADDI	SUB	
2	- wrong PC	x	BNE	ADDI	SUB
3	- correct PC	x	x	BNE	ADDI
4	- correct PC	x	x	x	BNE
5	- correct PC	x	x	x	x
6	- correct PC	x	x	x	x
7	- correct PC	x	x	x	x
8	NOT	x	x	x	x
9		NOT	x	x	x

Mispredicted branch instruction stalled due to data memory miss

Cycles	IF	ID	EX	MEM	WB
1	BNE	LWD			
2	- wrong PC	BNE	LWD		
3	- wrong PC	x	BNE	LWD	
4	- correct PC	x	BNE	LWD	x
5	- correct PC	x	BNE	LWD	x
6	- correct PC	x	BNE	LWD	x
7	- correct PC	x	BNE	LWD	x
8	- correct PC	x	BNE	LWD	x
9	NOT	x	x	BNE	LWD
10		NOT	x	x	BNE

Cycles	IF	ID	EX	MEM	WB
1	- wrong PC	x	LWD		
2	- wrong PC	BNE	x	LWD	
3	- wrong PC	x	BNE	LWD	x
4	NOT (in IR)	x	BNE	LWD	x
5	NOT (in IR)	x	BNE	LWD	x
6	NOT (in IR)	x	BNE	LWD	x
7	NOT (in IR)	x	BNE	LWD	x
8		NOT	x	BNE	LWD

`PC` doesn't change, but `BranchMisprediction` is asserted the whole time, which keeps re-initializing `InstMemCounter` to LATENCY - 1 every cycle. To solve this problem, we should memoize the last ID_EX_PC that caused a control hazard. On match, we do not initialize `InstMemCounter`. Here, BranchMisprediction must be prioritized.

IF stage

i_readC
- signals instruction fetch to instruction cache
- deasserted immediately after i_data is latched into IR
- deasserted on BranchMisprediction / JumpMisprediction => cancels current fetch

FetchCompletePC: latches most recently fetched PC

LastCycleFetch: asserted when an instruction was transfered into the ID stage last cycle.

MEM stage

d_readC: signals data read from data cache
d_writeC: signals data write to data cache

ReadCompletePC: latches most recently fetched address

TestBench

On clock posedge, if `IFIDWrite` is enabled, increment `num_inst` by 1.
On `BranchMisprediction` posedge, if `LastCycleFetch` is asserted, decrement `num_inst` by 1.