

1 Virtual Memory and Caches

- 1.1 In the caches we have studied so far, the virtual address is translated to a physical address before the cache is accessed, and the physical address supplies the tag, index, and offset. This is a physically indexed, physically tagged (PIPT) cache. Why might this be slow? How could we do better?
- 1.2 Suppose we introduce a *virtual cache* that works entirely in the virtual address space and only translates on a miss. Hits are now fast compared to PIPT. What are some possible problems with this sort of cache? *Hint: what if we have multiple programs running on the same CPU?*
- 1.3 What are some possible fixes for these problems?

2 Virtually Indexed, Physically Tagged Caches

A virtually indexed, physically tagged (VIPT) cache uses the virtual address for the index and part of the physical address as the tag, as in PIPT. We still need the physical address from the TLB or a page walk to compare the tag, but the cache lookup and translation now happen in parallel. Physical tags eliminate homonyms. Can we still have aliasing?

2.1 A direct-mapped, write-back VIPT cache uses the following address bits:

- Block offset: bits 5–0
- Index: bits **12–6** (from the VA)
- Tag: bits **31–12** (from the PA)

The page size is 4 KiB. The page table maps VPN 0x1 $\rightarrow$ PPN 0x7 and VPN 0x2 $\rightarrow$ PPN 0x7. Memory contains $\text{Mem}[0x7040] = 5$, and the cache starts empty. Fill in the table for each access below. What value does access 4 return?

- Load VA 0x1040
- Load VA 0x2040
- Store 9 to VA 0x1040
- Load VA 0x2040

Access	VA	PA	Index	Tag	Hit/Miss	Value
1						
2						
3						
4						

2.2 Now the cache uses these **new** bits, with the same page table, memory contents, and accesses as before:

- Block offset: bits 5–0
- Index: bits **11–6** (from the VA)
- Tag: bits **31–12** (from the PA)

Fill in the table. What value does access 4 return?

Access	VA	PA	Index	Tag	Hit/Miss	Value
1						
2						
3						
4						

2.3 Why did the first VIPT cache design cause aliasing but not the second one? Can you generalize the rule of thumb for VIPT cache designs?

2.4 A VIPT cache has the following parameters:

- Page size: 4 KB
 - Block size: 64 B
 - Cache size: 32 KB
 - Associativity: 2-way
- (a) Can this cache have aliasing?
- (b) If yes, when one physical block is accessed through different virtual addresses, how many different sets could it end up in?
- (c) Keeping the associativity and block size the same, what is the largest this cache can be without aliasing?
- (d) Keeping the cache size at 32 KB, what associativity would remove aliasing?

3 Intro to Out of Order Execution

Consider a small upgrade to the pipeline. We keep IF and ID, but add separate execution paths. Instructions issue in program order, and execution can overlap. Count the issue cycle as the first execution cycle.

Execution path	Operations	Latency
Path 1	Integer ALU / memory	ALU: 1 + MEM: 2
Path 2	FP add	2 cycles
Path 3	FP multiply	4 cycles

- 3.1 Both instructions below have ready operands. Issue A in cycle 1, then B in cycle 2. When does each instruction complete? Which instruction finishes first? Does issuing in program order guarantee completion in order?

A: `fmul f4, f0, f2`
 B: `fadd f8, f6, f10`

Instruction	Issue cycle	Execution cycles	Completes
A: <code>fmul</code>			
B: <code>fadd</code>			

Now consider a simple pipeline with a waiting queue. Loads use the ALU and then MEM. FP operations go directly to WB.

Timing rules:

- F and D take 1 cycle each. Fetch A in cycle 1.
- Dispatch into the queue whenever it has room. Issue whenever operands and an execution path are ready.
- Issue at most 1 instruction per cycle.
- Forward results starting in the following cycle.
- WB accepts 1 result per cycle.
- In order: issue the next instruction, or wait.
- Out of order: issue the oldest ready instruction.

A: `fmul f4, f0, f2`
 B: `fadd f6, f4, f8`
 C: `lw x5, 0(x6)`

- 3.2 Why must B wait? Can C execute during that wait?

- 3.3 Trace the pipeline with **in order** issue. Use f4? for a cycle spent waiting for A's result, and Q for a cycle spent waiting to issue. In which cycle does C write back?

Cycle	1	2	3	4	5	6	7	8	9	10	11
A											
B											
C											

- 3.4 Trace the same code with **out of order** issue. In which cycle does each instruction write back?

Cycle	1	2	3	4	5	6	7	8	9	10	11
A											
B											
C											

4 Hazards and Register Renaming

For each pair below, the earlier instruction is listed first and the later instruction second.

Hazard	Earlier / later access	What must stay correct?
RAW	Write / read	Read the producer's value
WAR	Read / write	Read before a later overwrite
WAW	Write / write	Keep the later write as the final value

- 4.1 Find the RAW, WAR, and WAW relationships in the code below. B waits for A. Could C safely run before B reads f3? What would go wrong if C and D wrote f3 in reverse order?

```
A: fmul f1, f0, f2
B: fadd f0, f3, f1
C: fmul f3, f2, f3
D: fadd f3, f3, f1
```

Type	Instruction pairs	Register

- 4.2 Rewrite the code using the free temporaries f4, f5, f6, and f7. Give each destination a fresh name and update its later reads. Where do the final values of the original f0, f1, and f3 now live? Which dependencies remain?

Old code snippet:

```
A: fmul f1, f0, f2
B: fadd f0, f3, f1
C: fmul f3, f2, f3
D: fadd f3, f3, f1
```

New code snippet:

```
A: fmul __, __, __
B: fadd __, __, __
C: fmul __, __, __
D: fadd __, __, __
```