22-cache-p1-notes

Caching: P1

Agenda:

- 0. Re-Orienting
- 1. Memory is Slow
- 2. Caching
- 3. The Basic Mechanism (Edited to include pseudocode of algorithm, as promised in class)
- 4. Flavor 1: Direct-Mapped
- 5. Flavor 2: Fully Associative
- 6. Flavor 3: Set Associative
- 7. Tradeoffs (if time)
- 8. Locality (if time)

Reading: 6.0, 6.1.4, 6.2-6.4

slides.pdf (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/22-cache-p1/slides.pdf)

---

0. Re-Orienting

Computation...all the way down to the laws of physics.

Mapping high-level languages to ISAs

Building datapaths out of combo logic (with some registers too)

Then using sequential logic to get the datapath implementing the ISA

But what about making it **faster?**.

Two main techniques:

- Caching
- Pipelining

---

1. Memory is Slow

Computation

Textbook, p560:

- Register: 0 cycles
- Main memory: 50-200

---

2. Caching

Idea: have some small, fast memory really close.

Textbook says: 1-30 cycles

Q: How can this be a win?

Think of a sample Y86 program (say, the calendar)....

- What if the cache could only hold 4 bytes?
- What if it could hold a 1 kilobyte chunk?
- What if it could two 1 kilobyte chunks?
- What if it could two 256-byte chunks?

Q: Implications?

Two main ones:

- What the machine seems to be doing (from the outside) no longer necessarily matches what it's actually doing
- How long an instruction takes to execute is no longer a simple matter of looking up a number in the manual (or FSM)

Q: Challenges?

- What do we put there?
- How do we organize it?
- What if the cache is full?
- What about writing to memory?
- Costs?

Keeping mind:

- Why, in general, can this be a win?

---

3. The Basic Mechanism

The typical construction:

(Fig 6.27a)

A **line** consists of

- **block** of B bytes
- a **tag**
- a **valid** bit
- (maybe more bookkeeping?)

A **set** consists of E lines

The cache then consists of S sets.

Cache **size** C = B x E x S

B,E,S are usually powers of two

- Why?

Addressing

(Fig 6.27b)

Partition address into tag, set index, offset

Use set index to find the set

Use tag to see if any line in that set matches the address

If so... use offset to find the byte within the block

(Note connection: tag bits + set bits + offset bits = address bits)

The Promised Pseudocode

The basic cache mechanism follows the following algorithm:

```
Suppose the CPU requests the byte at address Atag:Aset:Aoffset

If Set[Aset] contains a line L such that L.valid == TRUE AND L.tag == Atag
     then a cache hit!  return byte L.block[Aoffset]
```

```
Else
     cache miss!
     find an empty line  L' in Set[Aset].      (If no line is empty, then make one
empty by evicting a line from the set)
     Set L'.valid to be TRUE
     Set L'.block to be bytes M[Atag:Aset:0] through M[Atag:Aset:foxes]
     Return byte L'.block[Aoffset]
```

Let's work through some examples!

Practice Problem 6.10

And then... going from some sample addresses to the tag and set index and block offset

---

4. Flavor 1: Direct-Mapped

E = 1. One line per set!

Idea

Work through examples for

```
(S, E, B, m) = (4, 1, 2, 4)
```

Block-at-a-time movement

Conflict misses

Example sequence of instructions from pp602-603

---

5. Flavor 2: Fully Associative

S = 1. Only one set!

Gosh, how do you "address" into the cache?

No more conflict misses!

But.... consider the cost.

6. Flavor 3: Set Associative

(The common approach)

7. Tradeoffs

Which approach works best?

What blocksize is best?

8. Locality

The reason why it works!

Spatial locality

Temporal locality