EECS150 - Digital Design
Lecture 11 – SRAM (II), Caches

September 29, 2011

Elad Alon
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www-inst.eecs.berkeley.edu/~cs150

Announcements

• Homework #4 due today

• Homework #5 out today
  - Due next Thurs.

• Project checkpoints #1-3 released
  - Checkpoint #1 due next Wed. evening
  - Attend discussion session, start early!

• Next Tues. lecture will be given by Daiwei
  - (Elad will be out of town on Tues. and so office hours will be cancelled)
  - Will schedule additional TA office hours if needed
First-in-first-out (FIFO) Memory

- Used to implement queues.
- Find common use in computers & communication devices.
- Generally, used to “decouple” actions of producer and consumer:
  - Producer can perform many writes without consumer performing any reads (or vice versa).
  - Buffer size is finite, so on average need equal # of reads and writes.
  - Typical uses:
    - interfacing I/O devices, e.g. network interface. Data bursts from network, then processor bursts to memory buffer (or reads one word at a time from interface). Operations not synchronized.
    - Example: Audio output. Processor produces output samples in bursts (during process swap-in time). Audio DAC clocks it out at constant sample rate.

FIFO Interfaces

- Address pointers are used internally to keep next write position and next read position into a dual-port memory.
  - If pointers equal after write → FULL:
  - If pointers equal after read → EMPTY:
Xilinx Virtex5 FIFOs

- Virtex5 BlockRAMS include dedicated circuits for FIFOs.
- Details in User Guide [ug190].
- Takes advantage of separate dual ports and independent ports clocks.

---

Processor Design Considerations (1/2)

- **Register File: Consider distributed RAM (LUT RAM)**
  - Size is close to what is needed: distributed RAM primitive configurations are 32 or 64 bits deep. Extra width is easily achieved by parallel arrangements.
  - LUT-RAM configurations offer multi-porting options - useful for register files.
  - Asynchronous read, might be useful by providing flexibility on where to put register read in the pipeline.

- **Instruction / Data Caches : Consider Block RAM**
  - Higher density, lower cost for large number of bits
  - A single 36kbit Block RAM implements 1K 32-bit words.
  - Configuration stream based initialization, permits a simple “boot strap” procedure.

- **Other Memories in Project? Video?**
- **Main memory will be in external DRAM**
More generally, how does software interface to I/O devices?

*SO-DIMM* stands for small outline dual in-line memory module. SO-DIMMs are often used in systems which have space restrictions such as notebooks.

*DDR2* stands for second generation double data rate. DDR transfers data both on the rising and falling edges of the clock signal.

Why Different Types of Memory?
**Example: Direct-Mapped Cache**

- Reminder: entire address space is larger than size of cache (by definition)

- Need a way to map memory addresses into cache locations
  - And keep track of what memory address actually was
  - \( \rightarrow \) Cache tags

- M-byte cache with N byte blocks:

---

**Ex.: 1KB Direct Mapped Cache with 32B Blocks**

- For an M byte cache:
  - The uppermost \((32 - \log_2(M))\) bits are always the Cache Tag
  - The lowest \(\log_2(N)\) bits are the Byte Select

---

<table>
<thead>
<tr>
<th>Block address</th>
<th>Cache Tag</th>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Example: 0x50</td>
<td>Ex: 0x01</td>
<td>Ex: 0x00</td>
</tr>
<tr>
<td></td>
<td>Stored as part of the cache “state”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>0x50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Valid Bit
- Cache Tag
- Cache Data
- Byte 31
- Byte 1
- Byte 9
- Byte 32
- Byte 33
- Byte 992
Block Size Tradeoff

- In general, larger block sizes take advantage of spatial locality, **BUT:**
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
    - If block size is too big relative to cache size, miss rate will go up
      - Too few cache blocks

- In general, Average Access Time:
  \[ \text{Average Access Time} = \text{Hit Time} \times (1 - \text{Miss Rate}) + \text{Miss Penalty} \times \text{Miss Rate} \]

![Graph showing the tradeoff between block size, miss penalty, and access time](image)

Extreme Example: Single Line

<table>
<thead>
<tr>
<th>Valid Bit</th>
<th>Cache Tag</th>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Byte 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 3</td>
</tr>
</tbody>
</table>

- Cache Size = 4 bytes
  - Only ONE entry in the cache
- If an item is accessed, likely that it will be accessed again soon
  - But it is unlikely that it will be accessed again immediately!!!
  - The next access will likely be a miss
    - Continually loading data into the cache but discard them before they are used again
    - Worst nightmare of a cache designer: Ping Pong Effect

- Conflict Misses are misses caused by:
  - Different memory locations mapped to the same cache index
    - Solution 1: make the cache size bigger
    - Solution 2: Multiple entries for the same Cache Index
Set Associative Cache

- **N-way set associative**: N entries for each Cache Index
  - N direct mapped caches operate in parallel
- **Example: Two-way set associative cache**
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared to the input in parallel
  - Data is selected based on the tag result

**Diagram: Set Associative Cache**

- Valid
- Cache Tag
- Cache Data
- Cache Index
- Cache Block 0
- Cache Data
- Cache Tag
- Valid
- Adr Tag
- Compare
- Set 1
- Mux
- 0 Set 0
- Compare
- JR
- Hit
- Cache Block

Extreme Ex. #2: Fully Associative Cache

- **Fully Associative Cache**
  - Forget about the Cache Index
  - Compare the Cache Tags of all cache entries in parallel
  - Example: Block Size = 32 B blocks, we need (M/32) 27-bit comparators
    - Usually implemented with a Content Addressable Memory (CAM)
- By definition: Conflict Miss = 0 for a fully associative cache

**Diagram: Extreme Ex. #2: Fully Associative Cache**

- Cache Tag (27 bits long)
- Byte Select
- Byte 31
- Byte 1
- Byte 0
- Byte 63
- Byte 33
- Byte 32
Disadvantage of Set Associative Caches

- N-way Set Associative Cache versus Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss decision and set selection

- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.

Summary: Sources of Cache Misses

- Compulsory (cold start or process migration, first reference):
  - first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant

- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity

- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size

- Invalidation: other process (e.g., I/O) updates memory
Which Block to Replace on a Miss?

- Easy for Direct Mapped
  - Don’t really have any choice

- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

<table>
<thead>
<tr>
<th>Associativity:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Rand.</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
What Happens on a Write?

- **Write through**—The information is written to both the block in the cache and to the block in the lower-level memory.
- **Write back**—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no writes of repeated writes
- WT always combined with write buffers so that don’t wait for lower level memory

### Write Buffer for Write Through

- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) $\ll 1 / \text{DRAM write cycle}$
- Memory system designer’s nightmare:
  - Store frequency (w.r.t. time) $> 1 / \text{DRAM write cycle}$
  - Write buffer saturation
• Store frequency (w.r.t. time) > 1 / DRAM write cycle
  – If this condition exists for a long period of time (CPU cycle time too quick and/or too many store instructions in a row):
    • Store buffer will overflow no matter how big you make it
    • CPU Cycle Time <= DRAM Write Cycle Time

• Solution for write buffer saturation:
  – Use a write back cache
  – Install a second level (L2) cache w/write-back:

Write-Miss Policy: Write Allocate vs. Not Allocate

• Assume a 16-bit write to memory location 0x0 causes a miss
  – Do we read in the block?
    • Yes: Write Allocate
    • No: Write Not Allocate