How to Build a Cache
COMP 252 - Lecture 2

Antoniu Pop

antoniu.pop@manchester.ac.uk

1 February 2019
Computing technology trends, challenges & [some] solutions

- Rapid increase of processing power
- Slow increase of memory access capability
- Memory wall
- Solutions: Caches, out-of-order execution, prefetching, compiler optimizations, etc.

- No silver bullet...
- Core concepts behind caches
  - What: small, fast memory – close to the CPU
  - Why: Speed imbalance – accessing data can take one, zero, one, zero, zero cycles
  - How: Spatial & Temporal Locality
Computing technology trends, challenges & [some] solutions
  ▶ Rapid increase of processing power
Computing technology trends, challenges & [some] solutions

- Rapid increase of processing power
- Slow increase of memory access capability
Computing technology trends, challenges & [some] solutions

- Rapid increase of processing power
- Slow increase of memory access capability
- Memory wall

Solutions: Caches, out-of-order execution, prefetching, compiler optimizations, etc.

No silver bullet...

Core concepts behind caches

What: small, fast memory – close to the CPU
Why: Speed imbalance – accessing data can take /one.pnum/zero.pnums/one.pnum/zero.pnum/zero.pnum of cycles
How: Spatial & Temporal Locality
Computing technology trends, challenges & [some] solutions

- Rapid increase of processing power
- Slow increase of memory access capability
- Memory wall
- Solutions: Caches, out-of-order execution, prefetching, compiler optimizations, etc.
Computing technology trends, challenges & [some] solutions

- Rapid increase of processing power
- Slow increase of memory access capability
- Memory wall
- Solutions: Caches, out-of-order execution, prefetching, compiler optimizations, etc.
- No silver bullet...
Computing technology trends, challenges & [some] solutions

- Rapid increase of processing power
- Slow increase of memory access capability
- Memory wall
- Solutions: Caches, out-of-order execution, prefetching, compiler optimizations, etc.
- No silver bullet...

Core concepts behind caches
Computing technology trends, challenges & [some] solutions

- Rapid increase of processing power
- Slow increase of memory access capability
- Memory wall
- Solutions: Caches, out-of-order execution, prefetching, compiler optimizations, etc.
- No silver bullet...

Core concepts behind caches

- What: small, fast memory – close to the CPU
Previous Lecture

- **Computing technology trends, challenges & [some] solutions**
  - Rapid increase of processing power
  - Slow increase of memory access capability
  - Memory wall
  - Solutions: Caches, out-of-order execution, prefetching, compiler optimizations, etc.
  - No silver bullet...

- **Core concepts behind caches**
  - What: small, fast memory – close to the CPU
  - Why: Speed imbalance – accessing data can take 10s-100s of cycles
Computing technology trends, challenges & [some] solutions
- Rapid increase of processing power
- Slow increase of memory access capability
- Memory wall
- Solutions: Caches, out-of-order execution, prefetching, compiler optimizations, etc.
- No silver bullet...

Core concepts behind caches
- What: small, fast memory – close to the CPU
- Why: Speed imbalance – accessing data can take 10s-100s of cycles
- How: Spatial & Temporal Locality
Cache Labs: Objectives

Test:
▶ Cache behaviour
▶ Parameters
▶ Limits

Learn:
▶ How the cache works
▶ How memory accesses are handled
▶ What inhibits the cache’s behaviour

Understand:
▶ Importance of caches
▶ How to avoid inefficient patterns
▶ How to write, optimize programs to take advantage of caches
Cache Labs: Objectives

Test:
▶ Cache behaviour
▶ Parameters
▶ Limits

Learn:
▶ How the cache works
▶ How memory accesses are handled
▶ What inhibits the cache's behaviour

Understand:
▶ Importance of caches
▶ How to avoid inefficient patterns
▶ How to write, optimize programs to take advantage of caches
Cache Labs: Objectives

Test:
- Cache behaviour
- Parameters
- Limits

Learn:
- How the cache works
- How memory accesses are handled
- What inhibits the cache’s behaviour
Cache Labs: Objectives

Test:
▶ Cache behaviour
▶ Parameters
▶ Limits

Learn:
▶ How the cache works
▶ How memory accesses are handled
▶ What inhibits the cache’s behaviour

Understand:
▶ Importance of caches
▶ How to avoid inefficient patterns
▶ How to write, optimize programs to take advantage of caches
Today’s Lecture – Learning Objectives

To understand

▶ how cache is logically structured
Today’s Lecture – Learning Objectives

To understand

- how cache is logically structured
- how cache operates
  - CPU reads
  - CPU writes
Today’s Lecture – Learning Objectives

To understand
- how cache is logically structured
- how cache operates
  - CPU reads
  - CPU writes
- key cache performance parameters
Today’s Lecture – Learning Objectives

To understand

▶ how cache is logically structured
▶ how cache operates
  ▶ CPU reads
  ▶ CPU writes
▶ key cache performance parameters
▶ making cache more efficient
Cache Functionality

- The CPU gives the cache a full (e.g. 32 bit) address to access some data
Cache Functionality

- The CPU gives the cache a full (e.g. 32 bit) address to access some data

- The cache contains a (small) selection of values from main memory
  - Each such value in cache comes from a particular main memory location
Cache Functionality

▶ The CPU gives the cache a full (e.g. 32 bit) address to access some data

▶ The cache contains a (small) selection of values from main memory
  ▶ Each such value in cache comes from a particular main memory location

▶ How do we find the data?
▶ It may or may not be in the cache
Fully Associative Cache (1)

Address

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data if the address is found
### Fully Associative Cache (1)

- **Address**: 
  - Data if the address is found

- **Cache**: 
  - Small
  - Only holds a few memory values (32k?)

#### Diagram:
- **Address**
- **Data**

- Data if the address is found
Fully Associative Cache (1)

- Cache is small
- Only holds a few memory values (32k ?)
- Cache stores both addresses and data
Fully Associative Cache (1)

- Cache is small
- Only holds a few memory values (32k ?)
- Cache stores both addresses and data

Hardware compares input address with all stored addresses
Parallel lookup of (key/value) pairs

Data if the address is found
If address is found
  - this is a “cache hit”
  - data is read (or written)
  - no need to access main RAM memory – fast
Fully Associative Cache (2)

- If address is found
  - this is a “cache hit”
  - data is read (or written)
  - no need to access main RAM memory – fast

- If address is not found
  - this is a “cache miss”
  - must go to main RAM memory – slow
Fully Associative Cache (2)

- If address is found
  - this is a “cache hit”
  - data is read (or written)
  - no need to access main RAM memory – fast

- If address is not found
  - this is a “cache miss”
  - must go to main RAM memory – slow

- But how does data get into the cache?
Fully Associative Cache (2)

- If address is found
  - this is a “cache hit”
  - data is read (or written)
  - no need to access main RAM memory – fast

- If address is not found
  - this is a “cache miss”
  - must go to main RAM memory – slow

- But how does data get into the cache?

- If we have to go to main memory, should we just leave the cache as it was?
Locality

Caches rely on locality

- **Temporal Locality**: Things used will be (likely) used again soon, e.g., instructions and data in loops.
- **Spatial Locality**: Things close together (adjacent addresses) in store are often used together, e.g., instructions or arrays.
Locality

Caches rely on locality

▶ Temporal Locality
  ▶ things when used will be (likely) used again soon
  ▶ e.g. instructions and data in loops
Locality

Caches rely on locality

- Temporal Locality
  - things when used will be (likely) used again soon
  - e.g. instructions and data in loops

- Spatial locality
  - things close together (adjacent addresses) in store are often used together
  - e.g. instructions or arrays
Cache Hit Rate

- Fraction of cache accesses which “hit” in cache

Hit rates for instructions usually better than for data due to more regular access patterns and more locality.
Cache Hit Rate

- Fraction of cache accesses which “hit” in cache
- Need > 98% to hide 50 : 1 ratio of memory speed to instruction speed
Cache Hit Rate

- Fraction of cache accesses which “hit” in cache
- Need > 98% to hide 50 : 1 ratio of memory speed to instruction speed
- Hit rates for instructions usually better than for data
  - more regular access patterns
  - more locality
What to do on a Cache Miss

- Temporal locality says it is a good idea to put recently used data in the cache

Memory reads
- Put newly read value into the cache (just the value required?)
- But cache may be already full
- Need to choose a location to reject (replacement policy)
What to do on a Cache Miss

- Temporal locality says it is a good idea to put recently used data in the cache

- Memory reads
  - Put newly read value into the cache (just the value required?)
What to do on a Cache Miss

- Temporal locality says it is a good idea to put recently used data in the cache

- Memory reads
  - Put newly read value into the cache (just the value required?)
  - But cache may be already full
What to do on a Cache Miss

- Temporal locality says it is a good idea to put recently used data in the cache

- Memory reads
  - Put newly read value into the cache (just the value required?)
  - But cache may be already full
  - Need to choose a location to reject (replacement policy)
Cache Replacement Policy

- Least Recently Used (LRU)
  - makes sense
  - but expensive to implement (in hardware)
Cache Replacement Policy

- Least Recently Used (LRU)
  - makes sense
  - but expensive to implement (in hardware)

- Round Robin (or cyclic)
  - cycle round locations
  - least recently fetched from memory
Cache Replacement Policy

- Least Recently Used (LRU)
  - makes sense
  - but expensive to implement (in hardware)

- Round Robin (or cyclic)
  - cycle round locations
  - least recently fetched from memory

- Random
  - easy to implement
  - not as bad as it might seem
Memory Writes are slightly more complex than reads.

- Cache hit
  - Update value in cache
  - But what about value in RAM?
  - If we write to RAM every time it will be slow
  - Does RAM need updating every time?
Cache Write Strategy

- Memory Writes are slightly more complex than reads
- Must do an address comparison first to see if address is in cache
Cache Write Strategy

- Memory Writes are slightly more complex than reads
- Must do an address comparison first to see if address is in cache
- Cache hit
  - Update value in cache
Cache Write Strategy

- Memory Writes are slightly more complex than reads
- Must do an address comparison first to see if address is in cache
- Cache hit
  - Update value in cache
  - But what about value in RAM?
Cache Write Strategy

▶ Memory Writes are slightly more complex than reads
▶ Must do an address comparison first to see if address is in cache
▶ Cache hit
  ▶ Update value in cache
  ▶ But what about value in RAM?
  ▶ If we write to RAM every time it will be slow
Cache Write Strategy

- Memory Writes are slightly more complex than reads
- Must do an address comparison first to see if address is in cache
- Cache hit
  - Update value in cache
  - But what about value in RAM?
  - If we write to RAM every time it will be slow
  - Does RAM need updating every time?
Cache Hit Write Strategy

- **Write Through**
  - Every cache write is also done to memory
  - Write through is buffered i.e. processor doesn't wait for it to finish (but multiple writes could back up)

- **Copy Back**
  - Write is only done to cache (mark as "dirty")
  - RAM is updated when "dirty" cache entry is replaced (or cache is flushed e.g. on process switch)
Cache Hit Write Strategy

- Write Through
  - Every cache write is also done to memory

- Write Through with buffers
  - Write through is buffered i.e. processor doesn’t wait for it to finish (but multiple writes could back up)

- Copy Back
  - Write is only done to cache (mark as “dirty”)
  - RAM is updated when “dirty” cache entry is replaced (or cache is flushed e.g. on process switch)
Cache Hit Write Strategy

- Write Through
  - Every cache write is also done to memory

- Write Through with buffers
  - Write through is buffered i.e processor doesn’t wait for it to finish (but multiple writes could back up)
Cache Hit Write Strategy

- **Write Through**
  - Every cache write is also done to memory

- **Write Through with buffers**
  - Write through is buffered i.e processor doesn’t wait for it to finish (but multiple writes could back up)

- **Copy Back**
  - Write is only done to cache (mark as “dirty”)
  - RAM is updated when “dirty” cache entry is replaced (or cache is flushed e.g. on process switch)
Cache Miss Write Strategy

Cache miss on write

- Write Allocate
- Find a location (reject an entry if necessary)
- Assign cache location and write value
- Write through back to RAM
- Or rely on copy back later

- Write Around (or Write no-Allocate)
- Just write to RAM
- Subsequent read will cache it if necessary
Cache Miss Write Strategy

Cache miss on write

- Write Allocate
  - Find a location (reject an entry if necessary)
  - Assign cache location and write value

- Write Through back to RAM

- Or rely on copy back later

- Write Around (or Write no-Allocate)
  - Just write to RAM

  Subsequent read will cache it if necessary
Cache Miss Write Strategy

Cache miss on write

- Write Allocate
  - Find a location (reject an entry if necessary)
  - Assign cache location and write value
  - Write through back to RAM
  - Or rely on copy back later
Cache Miss Write Strategy

Cache miss on write
▶ Write Allocate
  ▶ Find a location (reject an entry if necessary)
  ▶ Assign cache location and write value
  ▶ Write through back to RAM
  ▶ Or rely on copy back later

▶ Write Around (or Write no-Allocate)
  ▶ Just write to RAM
Cache Miss Write Strategy

Cache miss on write

- **Write Allocate**
  - Find a location (reject an entry if necessary)
  - Assign cache location and write value
  - Write through back to RAM
  - Or rely on copy back later

- **Write Around (or Write no-Allocate)**
  - Just write to RAM
  - Subsequent read will cache it if necessary
Overall Cache Write Strategy

- Fastest is Write Allocate / Copy Back
Overall Cache Write Strategy

- Fastest is Write Allocate / Copy Back
- But cache & main memory are not “coherent”
  - Can have different values until the cache line is written back
Overall Cache Write Strategy

- Fastest is Write Allocate / Copy Back
- But cache & main memory are not “coherent”
  - Can have different values until the cache line is written back
- Does this matter?
Overall Cache Write Strategy

▶ Fastest is Write Allocate / Copy Back

▶ But cache & main memory are not “coherent”
  ▶ Can have different values until the cache line is written back

▶ Does this matter?
  ▶ Autonomous I/O devices
Overall Cache Write Strategy

- Fastest is Write Allocate / Copy Back
- But cache & main memory are not “coherent”
  - Can have different values until the cache line is written back
- Does this matter?
  - Autonomous I/O devices
  - Exceptions
Overall Cache Write Strategy

- Fastest is Write Allocate / Copy Back
- But cache & main memory are not “coherent”
  - Can have different values until the cache line is written back
- Does this matter?
  - Autonomous I/O devices
  - Exceptions
  - Multi-processors
Overall Cache Write Strategy

- Fastest is Write Allocate / Copy Back
- But cache & main memory are not “coherent”
  - Can have different values until the cache line is written back
- Does this matter?
  - Autonomous I/O devices
  - Exceptions
  - Multi-processors
- May need special handling (later)
Each cache line is:

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
</table>

- If the address is e.g. `/three.pnum/two.pnum` bit-byte address, logically the data is one byte.
- In practice, we hold more data per address, e.g. a `/four.pnum` byte (or `/three.pnum/two.pnum` bit) word.
- The stored address needed is then only `/three.pnum/zero.pnum` bits.
- In real practice (later), we would hold even more data per address.

- Ef/uniFB01ciency?
- How many bits are effectively used to store data?
Cache Data Width

Each cache line is:

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
</table>

- If address is e.g. 32 bit byte address, logically data is one byte

Effective Space Utilization?

How many bits are effectively used to store data?
Each cache line is:

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
</table>

- If address is e.g. 32 bit byte address, logically data is one byte.
- In practice we hold more data per address:
  - e.g. a 4 byte (32 bit) word
  - stored address needed is then only 30 bits.
Cache Data Width

Each cache line is:

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
</table>

- If address is e.g. 32 bit byte address, logically data is one byte
- In practice we hold more data per address
  - e.g a 4 byte (32 bit) word
  - stored address needed is then only 30 bits
- In real practice (later) we would hold even more data per address
Each cache line is:

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
</table>

- If address is e.g. 32 bit byte address, logically data is one byte
- In practice we hold more data per address
  - e.g a 4 byte (32 bit) word
  - stored address needed is then only 30 bits
- In real practice (later) we would hold even more data per address
- Efficiency?
  - How many bits are effectively used to store data?
Real Cache Implementations

- Fully associative cache is logically what we want
Real Cache Implementations

- Fully associative cache is logically what we want
  - But expensive!
Real Cache Implementations

- Fully associative cache is logically what we want
  - But expensive!
  - Need to store full (e.g., 32 bit) addresses
Real Cache Implementations

- Fully associative cache is logically what we want
  - But expensive!
  - Need to store full (e.g., 32 bit) addresses
  - Hardware comparison is expensive (logic) and uses a lot of power
Real Cache Implementations

- Fully associative cache is logically what we want
  - But expensive!
  - Need to store full (e.g., 32 bit) addresses
  - Hardware comparison is expensive (logic) and uses a lot of power

- It is possible to achieve most of the functionality using ordinary small and fast (usually static) RAM memory in special ways
Real Cache Implementations

- Fully associative cache is logically what we want
  - But expensive!
  - Need to store full (e.g., 32 bit) addresses
  - Hardware comparison is expensive (logic) and uses a lot of power

- It is possible to achieve most of the functionality using ordinary small and fast (usually static) RAM memory in special ways

- Few real processor caches are fully associative
Direct Mapped Cache (1)

- Uses standard RAM to implement the functionality of an ideal cache (with some limitations)
Uses standard RAM to implement the functionality of an ideal cache (with some limitations)

Usually uses static RAM – although more expensive than DRAM, is faster
Direct Mapped Cache (1)

- Uses standard RAM to implement the functionality of an ideal cache (with some limitations)

- Usually uses static RAM – although more expensive than DRAM, is faster

- But overall is considerably cheaper to implement than fully associative
Direct Mapped Cache (2)

Address divided into two parts

- Address divided into two parts
- Index (e.g., /one.pnum/five.pnum) used to address (/three.pnum/two.pnum) RAMs directly
- Rest of address, the tag, is stored and compared with incoming tag
- If same (hit), data is read
- If different (miss), data is not used

Diagram:
- Tag RAM
- Data RAM
- Compare
- Hit / miss
- Data
- Address
Direct Mapped Cache (2)

Address divided into two parts

- index (e.g., 15 bits) used to address (32k) RAMs directly
Direct Mapped Cache (2)

Address divided into two parts

- index (e.g., 15 bits) used to address (32k) RAMs directly
- Rest of address, the tag, is stored and compared with incoming tag
Direct Mapped Cache (2)

Address divided into two parts

- Index (e.g., 15 bits) used to address (32k) RAMs directly
- Rest of address, the tag, is stored and compared with incoming tag

- If same (hit) data is read
Direct Mapped Cache (2)

Address divided into two parts
- index (e.g., 15 bits) used to address (32k) RAMs directly
- Rest of address, the tag, is stored and compared with incoming tag

- If same (hit) data is read
- If different (miss) data is not used
Direct Mapped Cache (3)

- Logically, the RAM is divided into blocks of size equal to cache size.
Direct Mapped Cache (3)

- Logically, the RAM is divided in blocks of size equal to cache size
- Each memory location has an assigned cache entry
Direct Mapped Cache (3)

- Logically, the RAM is divided in blocks of size equal to cache size
- Each memory location has an assigned cache entry
- Multiple RAM location compete for the same cache location
Direct Mapped Cache (3)

- Logically, the RAM is divided in blocks of size equal to cache size
- Each memory location has an assigned cache entry
- Multiple RAM location compete for the same cache location
Direct Mapped Cache (4)

- Is really just a hash table implemented in hardware
Direct Mapped Cache (4)

- Is really just a hash table implemented in hardware
- Index / Tag could be selected from address in many ways
Direct Mapped Cache (4)

- Is really just a hash table implemented in hardware
- Index / Tag could be selected from address in many ways
- Most other aspects of operation are identical to fully associative
Direct Mapped Cache (4)

- Is really just a hash table implemented in hardware
- Index / Tag could be selected from address in many ways
- Most other aspects of operation are identical to fully associative
- Except for replacement policy
Direct Mapped Replacement

- New incoming tag/data can only be stored in one place - dictated by index
Direct Mapped Replacement

- New incoming tag/data can only be stored in one place - dictated by index

- Using LSBs as index exploits principle of spatial locality to minimize displacing recently used data
Direct Mapped Replacement

- New incoming tag/data can only be stored in one place - dictated by index

- Using LSBs as index exploits principle of spatial locality to minimize displacing recently used data

- Much cheaper than associative (and faster?) but lower “hit rate” (due to inflexible replacement strategy)
A compromise!

A set associative cache is simply a small number (e.g. /two.pnum or /four.pnum) of direct mapped caches operating in parallel. If one matches, we have a hit and select the appropriate data.
Set Associative Caches (1)

- A compromise!

- A set associative cache is simply a small number (e.g. 2 or 4) of direct mapped caches operating in parallel
Set Associative Caches (1)

- A compromise!

- A set associative cache is simply a small number (e.g. 2 or 4) of direct mapped caches operating in parallel

- If one matches, we have a hit and select the appropriate data.
E.g., 2-way set associative cache

Tag RAM

Data RAM

Tag Index

Compare

address

Tag RAM

Data RAM

Tag Index

Compare

address
Set Associative Caches (2)

- Now replacement strategy can be a bit more flexible
Set Associative Caches (2)

- Now replacement strategy can be a bit more flexible

- E.g. in a 4 way, we can choose any one of 4 - using LRU, cyclic etc.
Set Associative Caches (2)

- Now replacement strategy can be a bit more flexible

- E.g. in a 4 way, we can choose any one of 4 - using LRU, cyclic etc.

- Now (e.g.) 4 entries all with the same index but different tag can be stored. Less “interference” than direct mapped.
Hit rate gets better with increasing number of ways
Set Associative Caches (3)

▶ Hit rate gets better with increasing number of ways

▶ Obviously higher no. of ways gets more expensive
Set Associative Caches (3)

- Hit rate gets better with increasing number of ways
- Obviously higher no. of ways gets more expensive
- In fact N location fully associative cache is the same as a N way set associative cache!
Example

Cache 0

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
</tbody>
</table>

Cache 1

<table>
<thead>
<tr>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100</td>
<td>00101</td>
<td>00110</td>
<td>00111</td>
</tr>
<tr>
<td>01000</td>
<td>01001</td>
<td>01010</td>
<td>01011</td>
</tr>
<tr>
<td>01100</td>
<td>01101</td>
<td>01110</td>
<td>01111</td>
</tr>
<tr>
<td>10000</td>
<td>10001</td>
<td>10010</td>
<td>10011</td>
</tr>
<tr>
<td>10100</td>
<td>10101</td>
<td>10110</td>
<td>10111</td>
</tr>
<tr>
<td>11000</td>
<td>11001</td>
<td>11010</td>
<td>11011</td>
</tr>
<tr>
<td>11100</td>
<td>11101</td>
<td>11110</td>
<td>11111</td>
</tr>
</tbody>
</table>

RAM

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111
Example
Example

Cache 0 Cache 1

RAM

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
10000
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
11110
11111
Example
Example
Example

Cache 0
00
A

01
C

10
E

11
G

Cache 1
00
B

01
D

10
F

11
H

RAM
0000
0001
0010
0011
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Example

<table>
<thead>
<tr>
<th>RAM</th>
<th>Cache 0</th>
<th>Cache 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>00001</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>00010</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>00011</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>00100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Antoniu Pop – How to Build a Cache
Example