Readings

- **Caches**

- **Required**
  - H&H Chapters 8.1-8.3
  - Refresh: P&P Chapter 3.5

- **Recommended**
  - An early cache paper by Maurice Wilkes
Recall: Cache Structure

Address → Tag Store

(is the address in the cache? + bookkeeping)

Hit/miss? → Data Store

(stores memory blocks)

Data
Recall: Set Associativity

- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks

Key idea: Associative memory within the set
+ Accommodates conflicts better (fewer conflict misses)
-- More complex, slower access, larger tag store
What’s In A Tag Store Entry?

- Valid bit
- Tag
- Replacement policy bits

- Dirty bit?
  - Write back vs. write through caches
Handling Writes (I)

- When do we write the modified data in a cache to the next level?
  - **Write through**: At the time the write happens
  - **Write back**: When the block is evicted

- **Write-back**
  + Can combine multiple writes to the same block before eviction
    - Potentially saves bandwidth between cache levels + saves energy
  -- Need a bit in the tag store indicating the block is “dirty/modified”

- **Write-through**
  + Simpler
  + All levels are up to date. **Consistency**: Simpler cache coherence because no need to check close-to-processor caches’ tag stores for presence
  -- More bandwidth intensive; no combining of writes
Handling Writes (II)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

Allocate on write miss
- Can combine writes instead of writing each of them individually to next level
- Simpler because write misses can be treated the same way as read misses
- Requires transfer of the whole cache block

No-allocate
- Conserves cache space if locality of writes is low (potentially better cache hit rate)
What if the processor writes to an entire block over a small amount of time?

Is there any need to bring the block into the cache from memory in the first place?

Why do we not simply write to only a portion of the block, i.e., subblock

- E.g., 4 bytes out of 64 bytes
- Problem: Valid and dirty bits are associated with the entire 64 bytes, not with each individual 4 bytes
Subblocked (Sectored) Caches

- Idea: Divide a block into subblocks (or sectors)
  - Have separate valid and dirty bits for each subblock (sector)
  - Allocate only a subblock (or a subset of subblocks) on a request

++ No need to transfer the entire cache block into the cache
   (A write simply validates and updates a subblock)
++ More freedom in transferring subblocks into the cache (a cache block does not need to be in the cache fully)
   (How many subblocks do you transfer on a read?)

-- More complex design
-- May not exploit spatial locality fully

[Diagram showing cache block structure with valid (v), dirty (d), subblock, and tag bits]
Instruction vs. Data Caches

- **Separate or Unified?**

- **Pros and Cons of Unified:**
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., separate I and D caches)
  - -- Instructions and data can thrash each other (i.e., no guaranteed space for either)
  - -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

- **First level caches are almost always split**
  - Mainly for the last reason above

- **Higher level caches are almost always unified**
Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity; latency is critical
  - Tag store and data store accessed in parallel

- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Tag store and data store accessed serially

- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter (filters some temporal and spatial locality)
    - Management policies are therefore different
Cache Performance
Cache Parameters vs. Miss/Hit Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy
Cache Size

- **Cache size**: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better

- **Too large** a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path

- **Too small** a cache
  - doesn’t exploit temporal locality well
  - useful data replaced often

- **Working set**: the whole set of data the executing application references
  - Within a time interval
Block Size

- Block size is the data that is associated with an address tag, not necessarily the unit of transfer between hierarchies.
  - Sub-blocking: A block divided into multiple pieces (each w/ V/D bits)

- Too small blocks
  - don’t exploit spatial locality well
  - have larger tag overhead

- Too large blocks
  - too few total # of blocks $\rightarrow$ less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high
Large Blocks: Critical-Word and Subblocking

- Large cache blocks can take a long time to fill into the cache
  - fill cache line **critical word first**
  - restart cache access before complete fill

- Large cache blocks can waste bus bandwidth
  - divide a block into subblocks
  - associate separate valid and dirty bits for each subblock
  - **Recall: When is this useful?**

<table>
<thead>
<tr>
<th>v</th>
<th>d</th>
<th>subblock</th>
<th>v</th>
<th>d</th>
<th>subblock</th>
<th>•</th>
<th>•</th>
<th>•</th>
<th>•</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>d</td>
<td>subblock</td>
<td>tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Associativity

- How many blocks can be present in the same index (i.e., set)?

- Larger associativity
  - lower miss rate (reduced conflicts)
  - higher hit latency and area cost (plus diminishing returns)

- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches

- Is power of 2 associativity required?
Classification of Cache Misses

- **Compulsory miss**
  - first reference to an address (block) always results in a miss
  - subsequent references should hit unless the cache block is displaced for the reasons below

- **Capacity miss**
  - cache is too small to hold everything needed
  - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity

- **Conflict miss**
  - defined as any miss that is neither a compulsory nor a capacity miss
How to Reduce Each Miss Type

- **Compulsory**
  - Caching cannot help
  - Prefetching can: Anticipate which blocks will be needed soon

- **Conflict**
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Better, randomized indexing
    - Software hints?

- **Capacity**
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set and computation such that each “computation phase” fits in cache
How to Improve Cache Performance

- Three fundamental goals

- Reducing miss rate
  - Caveat: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted

- Reducing miss latency or miss cost

- Reducing hit latency or hit cost

- The above three together affect performance
Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches

- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches
Software Approaches for Higher Hit Rate

- Restructuring data access patterns
- Restructuring data layout
- Loop interchange
- Data structure separation/merging
- Blocking
- ...
Restructuring Data Access Patterns (I)

- **Idea:** Restructure data layout or data access patterns
- **Example:** If column-major
  - $x[i+1,j]$ follows $x[i,j]$ in memory
  - $x[i,j+1]$ is far away from $x[i,j]$

This is called **loop interchange**

Other optimizations can also increase hit rate
  - Loop fusion, array merging, ...
Restructuring Data Access Patterns (II)

- **Blocking**
  - Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache
  - Avoids cache conflicts between different chunks of computation
  - Essentially: *Divide the working set so that each piece fits in the cache*

- Also called **Tiling**
Restructuring Data Layout (I)

- Pointer based traversal (e.g., of a linked list)
- Assume a huge linked list (1B nodes) and unique keys

- Why does the code on the left have poor cache hit rate?
  - “Other fields” occupy most of the cache line even though rarely accessed!

```c
struct Node {
    struct Node* next;
    int key;
    char[256] name;
    char[256] school;
};

while (node) {
    if (node->key == input-key) {
        // access other fields of node
    }
    node = node->next;
}
```
Restructuring Data Layout (II)

- **Idea:** separate frequently-used fields of a data structure and pack them into a separate data structure

- **Who should do this?**
  - Programmer
  - Compiler
  - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently used?

```c
struct Node {
    struct Node* next;
    int key;
    struct Node-data* node-data;
}

struct Node-data {
    char [256] name;
    char [256] school;
}

while (node) {
    if (node->key == input-key) {
        // access node->node-data
    }
    node = node->next;
}
```
Multi-Core Issues in Caching
Caches in a Multi-Core System
Caches in Multi-Core Systems

- Cache efficiency becomes even more important in a multi-core/multi-threaded system
  - Memory bandwidth is at premium
  - Cache space is a limited resource across cores/threads

- How do we design the caches in a multi-core system?

- Many decisions
  - Shared vs. private caches
  - How to maximize performance of the entire system?
  - How to provide QoS to different threads in a shared cache?
  - Should cache management algorithms be aware of threads?
  - How should space be allocated to threads in a shared cache?
Private vs. Shared Caches

- **Private** cache: Cache belongs to one core (a shared block can be in multiple caches)
- **Shared** cache: Cache is shared by multiple cores
Resource Sharing Concept and Advantages

- **Idea:** Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
  - Example resources: functional units, pipeline, caches, buses, memory

- **Why?**
  - Resource sharing improves utilization/efficiency $\rightarrow$ throughput
    - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data
  - Reduces communication latency
    - For example, data shared between multiple threads can be kept in the same cache in multithreaded processors
  - Compatible with the shared memory programming model
Resource Sharing Disadvantages

- Resource sharing results in **contention for resources**
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it

- Sometimes reduces each or some thread’s performance
  - Thread performance can be worse than when it is run alone

- **Eliminates performance isolation** → inconsistent performance across runs
  - Thread performance depends on co-executing threads

- Uncontrolled (free-for-all) sharing **degrades QoS**
  - Causes unfairness, starvation

Need to efficiently and fairly utilize shared resources
Private vs. Shared Caches

- **Private** cache: Cache belongs to one core (a shared block can be in multiple caches)
- **Shared** cache: Cache is shared by multiple cores
Shared Caches Between Cores

- **Advantages:**
  - High effective capacity
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
    - If one core does not utilize some space, another core can
  - Easier to maintain coherence (a cache block is in a single location)

- **Disadvantages**
  - Slower access (cache not tightly coupled with the core)
  - Cores incur **conflict misses due to other cores’ accesses**
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)
Cache Coherence
Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?
The Cache Coherence Problem

ld r2, x

Interconnection Network

P1

P2

Main Memory

x 1000

1000
The Cache Coherence Problem

Id r2, x

P1

1000

Interconnection Network

P2

Id r2, x

1000

Main Memory
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1

ld r2, x

Interconnection Network

Main Memory
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1

ld r2, x
Should NOT load 1000
ld r5, x

ld r2, x
add r1, r2, r4
st x, r1

Interconnection Network

Main Memory
Cache Examples:
For You to Study
Cache Terminology

- **Capacity** ($C$):
  - the number of data bytes a cache stores

- **Block size** ($b$):
  - bytes of data brought into cache at once

- **Number of blocks** ($B = C/b$):
  - number of blocks in cache: $B = C/b$

- **Degree of associativity** ($N$):
  - number of blocks in a set

- **Number of sets** ($S = B/N$):
  - each memory address maps to exactly one cache set
How is data found?

- Cache organized into $S$ sets
- Each memory address maps to exactly one set

- Caches categorized by number of blocks in a set:
  - Direct mapped: 1 block per set
  - $N$-way set associative: $N$ blocks per set
  - Fully associative: all cache blocks are in a single set

- Examine each organization for a cache with:
  - Capacity ($C = 8$ words)
  - Block size ($b = 1$ word)
  - So, number of blocks ($B = 8$)
Direct Mapped Cache

Address

11...11111000
11...11111000
11...11110100
11...11111000
11...11101100
11...11101000
11...11001000
11...11000000
00...00100100
00...00100000
00...00011100
00...00011000
00...00010100
00...00010000
00...00001100
00...00001000
00...00000100
00...00000010
00...00000000

mem[0xFF...FC]
mem[0xFF...F8]
mem[0xFF...F4]
mem[0xFF...F0]
mem[0xFF...EC]
mem[0xFF...E8]
mem[0xFF...E4]
mem[0xFF...E0]
mem[0x00...1C]
mem[0x00...20]
mem[0x00...1C]
mem[0x00...18]
mem[0x00...14]
mem[0x00...10]
mem[0x00...0C]
mem[0x00...08]
mem[0x00...04]
mem[0x00...00]

Set Number

7 (111)
6 (110)
5 (101)
4 (100)
3 (011)
2 (010)
1 (001)
0 (000)

2^{30} Word Main Memory

2^{3} Word Cache
Direct Mapped Cache Hardware

- Memory Address
- Tag
- Set
- Byte Offset
- Data
- Hit
- V Tag
- Data
- 8-entry x (1+27+32)-bit SRAM
Direct Mapped Cache Performance

# MIPS assembly code

```
addi $t0, $0, 5
loop:  beq $t0, $0, done
lw    $t1, 0x4($0)
lw    $t2, 0x8($0)
lw    $t3, 0x8($0)
addi $t0, $t0, -1
j     loop
done:
```

Miss Rate =

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td>00...00</td>
<td>mem[0x00...08]</td>
</tr>
<tr>
<td>0</td>
<td>00...00</td>
<td>mem[0x00...0C]</td>
</tr>
</tbody>
</table>

Set 7 (111)
Set 6 (110)
Set 5 (101)
Set 4 (100)
Set 3 (011)
Set 2 (010)
Set 1 (001)
Set 0 (000)

Memory Address

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>00...00</td>
<td>001</td>
<td>00</td>
</tr>
</tbody>
</table>
Direct Mapped Cache Performance

```
# MIPS assembly code
addi $t0, $0, 5
loop:  beq $t0, $0, done
lw   $t1, 0x4($0)
lw   $t2, 0xC($0)
lw   $t3, 0x8($0)
addi $t0, $t0, -1
j    loop
done:
```

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td>00...00</td>
<td>mem[0x00...0C]</td>
</tr>
<tr>
<td>0</td>
<td>00...00</td>
<td>mem[0x00...08]</td>
</tr>
<tr>
<td>1</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
</tr>
</tbody>
</table>

Set 7: (111)  
Set 6: (110)  
Set 5: (101)  
Set 4: (100)  
Set 3: (011)  
Set 2: (010)  
Set 1: (001)  
Set 0: (000)  

Miss Rate = 3/15 = 20%

Temporal Locality Compulsory Misses
Direct Mapped Cache: Conflict

```
# MIPS assembly code
addi $t0, $0, 5
loop: beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0x24($0)
addi $t0, $t0, -1
j loop
done:
```

Miss Rate =
### Direct Mapped Cache: Conflict

#### MIPS assembly code

```plaintext
# MIPS assembly code
addi $t0, $0, 5
loop:  beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0x24($0)
addi $t0, $t0, -1
j  loop
done:
```

**Miss Rate** = \( \frac{10}{10} \) = 100%

**Conflict Misses**
N-Way Set Associative Cache
N-way Set Associative Performance

```mips
# MIPS assembly code

addi $t0, $0, 5
loop:
  beq $t0, $0, done
lw $t1, 0x4($0)
lw $t2, 0x24($0)
addi $t0, $t0, -1
j loop
done:

Miss Rate =
```

<table>
<thead>
<tr>
<th>Way 1</th>
<th>Tag</th>
<th>Data</th>
<th>Way 0</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00...10</td>
<td>mem[0x00...24]</td>
<td>1</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 3  
Set 2  
Set 1  
Set 0
### N-way Set Associative Performance

#### MIPS assembly code

```assembly
loop:
    addi $t0, $0, 5
    beq  $t0, $0, done
    lw   $t1, 0x4($0)
    lw   $t2, 0x24($0)
    addi $t0, $t0, -1
    j    loop

done:
```

**Miss Rate = 2/10 = 20%**

Associativity reduces conflict misses

<table>
<thead>
<tr>
<th>Way 1</th>
<th>Way 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Tag</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00...10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set 3</th>
<th>Set 2</th>
<th>Set 1</th>
<th>Set 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem[0x00...24]</td>
<td>mem[0x00...04]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fully Associative Cache

- No conflict misses
- Expensive to build
Spatial Locality?

- Increase block size:
  - Block size, $b = 4$ words
  - $C = 8$ words
  - Direct mapped (1 block per set)
  - Number of blocks, $B = C/b = 8/4 = 2$
Direct Mapped Cache Performance

```
addi $t0, $0, 5
loop: 
  beq $t0, $0, done
  lw $t1, 0x4($0)
  lw $t2, 0xC($0)
  lw $t3, 0x8($0)
  addi $t0, $t0, -1
  j loop
done:
```

**Miss Rate =**
Direct Mapped Cache Performance

```
addi $t0, $0, 5

loop:
  beq $t0, $0, done
  lw  $t1, 0x4($0)
  lw  $t2, 0xC($0)
  lw  $t3, 0x8($0)
  addi $t0, $t0, -1
  j   loop

done:
```

Miss Rate = 1/15
= 6.67%

Larger blocks reduce compulsory misses through spatial locality
Cache Organization Recap

- Main Parameters
  - Capacity: $C$
  - Block size: $b$
  - Number of blocks in cache: $B = C/b$
  - Number of blocks in a set: $N$
  - Number of Sets: $S = B/N$

<table>
<thead>
<tr>
<th>Organization</th>
<th>Number of Ways (N)</th>
<th>Number of Sets (S = B/N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Mapped</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>N-Way Set Associative</td>
<td>$1 &lt; N &lt; B$</td>
<td>B / N</td>
</tr>
<tr>
<td>Fully Associative</td>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>
Capacity Misses

- Cache is too small to hold all data of interest at one time
  - If the cache is full and program tries to access data X that is not in cache, cache must evict data Y to make room for X
  - **Capacity miss** occurs if program then tries to access Y again
  - X will be placed in a particular set based on its address

- In a **direct mapped** cache, there is only one place to put X

- In an **associative cache**, there are multiple ways where X could go in the set.

- How to choose Y to minimize chance of needing it again?
  - Least recently used (LRU) replacement: the least recently used block in a set is evicted when the cache is full.
Types of Misses

- **Compulsory**: first time data is accessed
- **Capacity**: cache too small to hold all data of interest
- **Conflict**: data of interest maps to same location in cache
- **Miss penalty**: time it takes to retrieve a block from lower level of hierarchy
LRU Replacement

# MIPS assembly

```
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 (11)</td>
</tr>
<tr>
<td>2 (10)</td>
</tr>
<tr>
<td>1 (01)</td>
</tr>
<tr>
<td>0 (00)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 (11)</td>
</tr>
<tr>
<td>2 (10)</td>
</tr>
<tr>
<td>1 (01)</td>
</tr>
<tr>
<td>0 (00)</td>
</tr>
</tbody>
</table>
LRU Replacement

# MIPS assembly

```mips
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

<table>
<thead>
<tr>
<th>Way 1</th>
<th>Way 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)

<table>
<thead>
<tr>
<th>Way 1</th>
<th>Way 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)