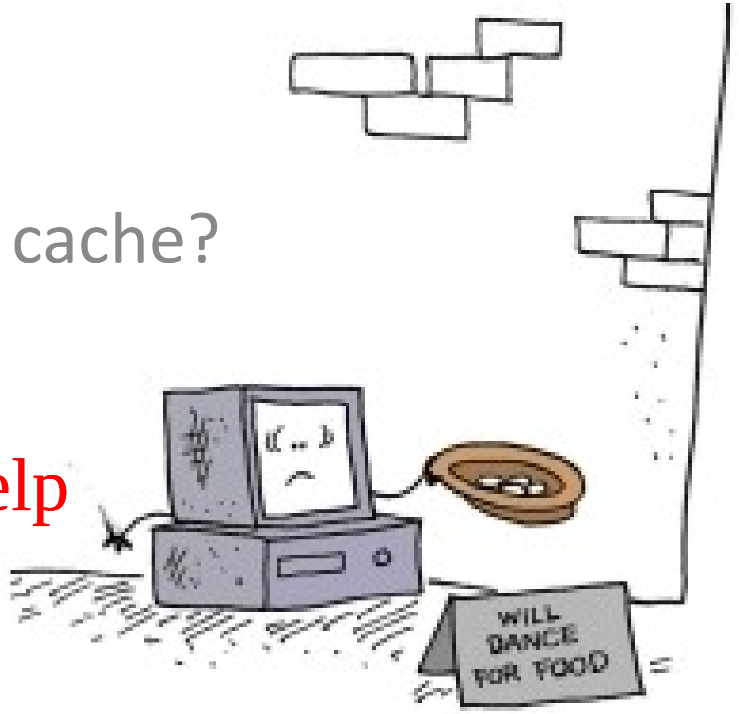


Low on cache?

Caches, Part I

Slides from Patterson's 61C, Fall 2001

Outline

- Memory Hierarchy
- Direct-Mapped Cache
- Types of Cache Misses
- A (long) detailed example

Memory Hierarchy (1/4)

- Processor
  - executes programs
  - runs on order of nanoseconds to picoseconds
  - needs to access code and data for programs: where are these?
- Disk
  - HUGE capacity (virtually limitless)
  - VERY slow: runs on order of milliseconds
  - so how do we account for this gap?

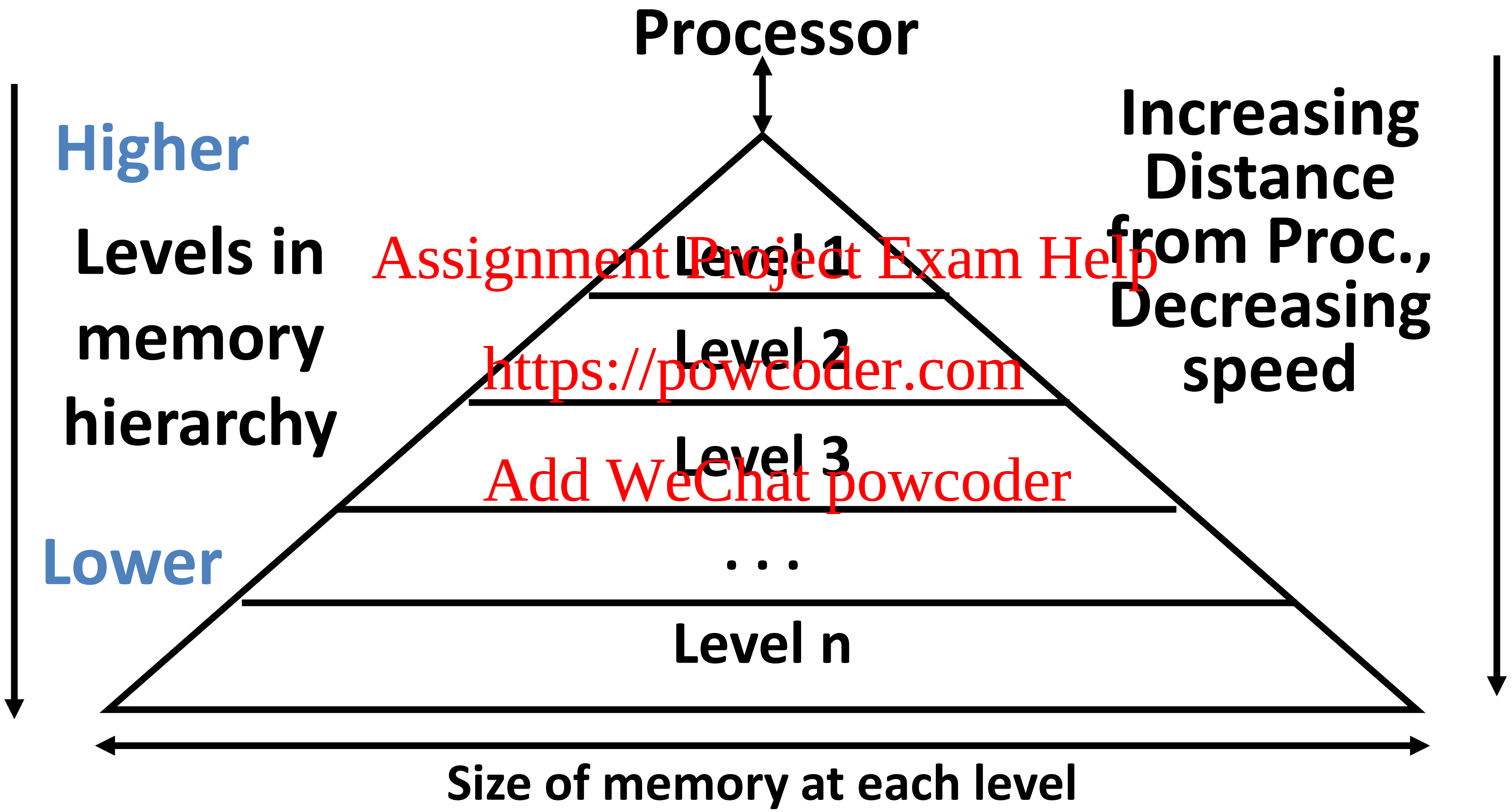
Slides from Patterson's 61C, Fall 2001

Memory Hierarchy (2/4)

- Memory (DRAM)
  - smaller than disk (not limitless capacity)
  - contains subset of data on disk: basically portions of programs that are currently being run
  - much faster than disk: memory accesses don't slow down processor quite as much
  - Problem: memory is still too slow (hundreds of nanoseconds)
  - Solution: add more layers (caches)

Slides from Patterson's 61C, Fall 2001

Memory Hierarchy (3/4)


Processor
Higher
Levels in memory hierarchy
Lower
Level 1
Level 2
Level 3
. . .
Level n
Increasing Distance from Proc., Decreasing speed
Size of memory at each level

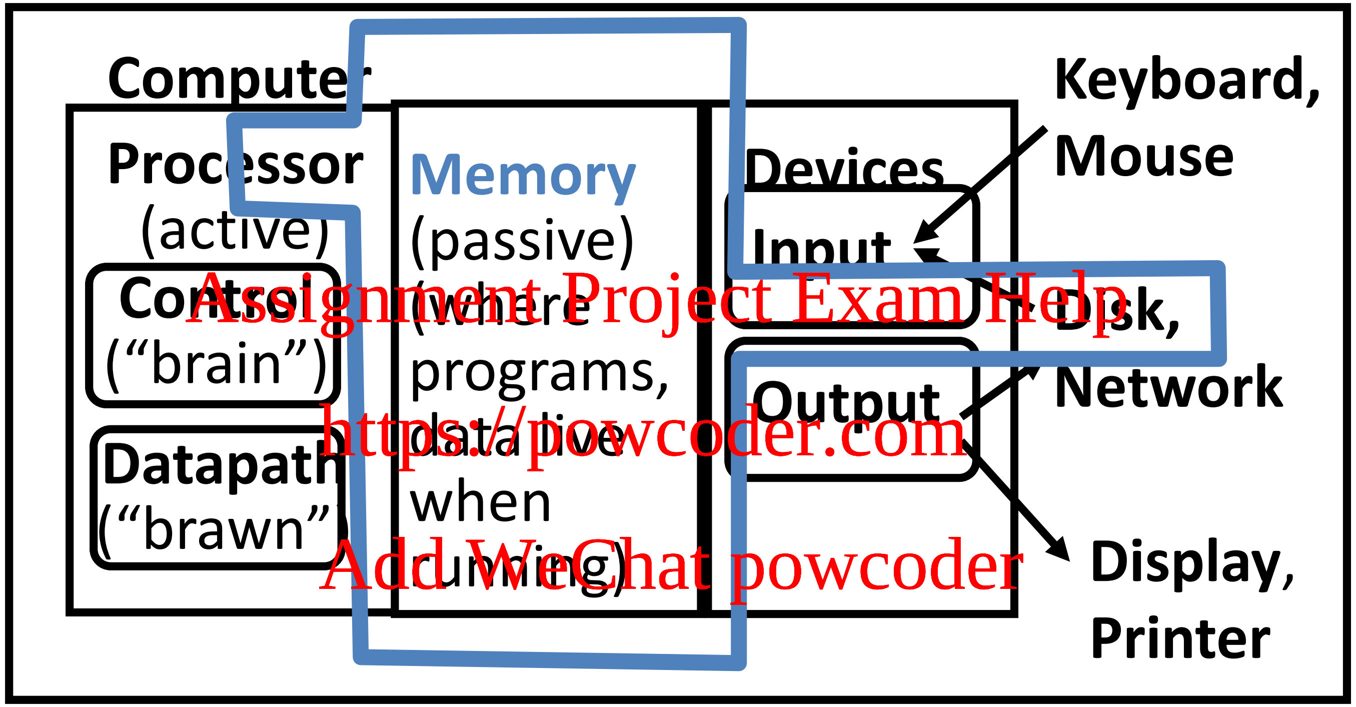
As we move to deeper levels the latency goes up and price per bit goes down.

Memory Hierarchy (4/4)

- If level is closer to Processor, it must…
  - Be smaller
  - Be faster
  - Contain a subset (most recently used data) of lower levels beneath it (i.e., levels farther from processor)
  - Contain all the data in higher levels above it (i.e., levels closer to processor)
- Lowest Level (usually disk) contains all available data
- Is there another level lower than disk?

Slides from Patterson's 61C, Fall 2001

Memory Hierarchy



- Purpose:
  - Faster access to large memory from processor

Slides from Patterson's 61C, Fall 2001

Memory Hierarchy Analogy: Library (1/2)

- You (the processor) are writing a term paper at a table in Schulich

- Schulich Library is equivalent to disk

  - essentially limitless capacity

  - very slow to retrieve a book
- Table is memory
  - smaller capacity: means you must return book when table fills up
  - easier and faster to find a book there once you've already retrieved it

Memory Hierarchy Analogy: Library (2/2)

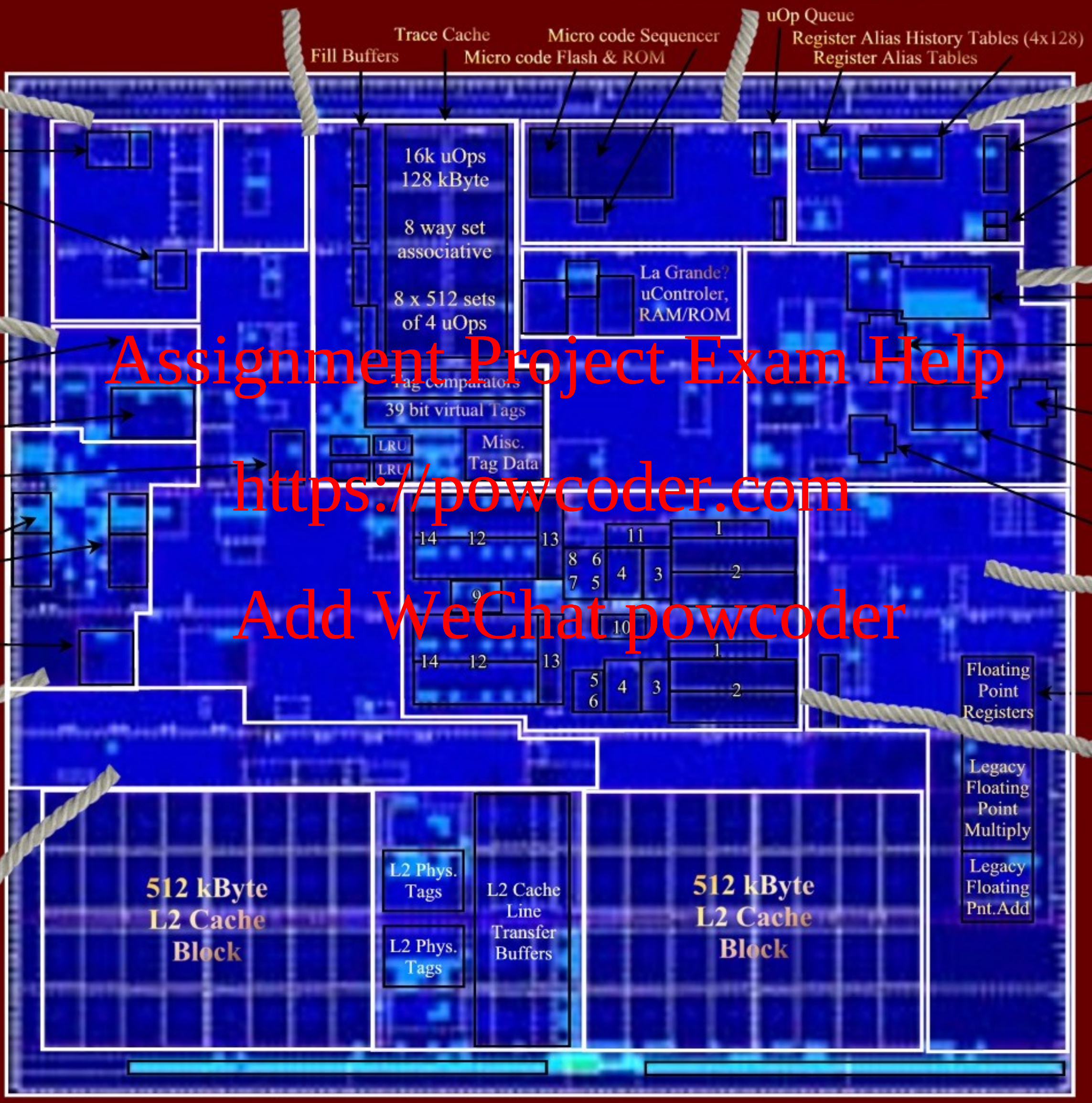
- Open books on table are cache
  - smaller capacity: can have very few open books fit on table; again, when table fills up, you must close a book
  - much, much faster to retrieve data
- Illusion created: whole library open on the tabletop
  - Keep as many recently used books open on table as possible since likely to use again
  - Also keep as many books on table as possible, since faster than going to library

Memory Hierarchy Basis

- Disk contains everything
- When Processor needs something, bring it into to all lower levels of memory
- Cache contains copies of data in memory that are being used
- Memory contains copies of data on disk that are being used
- Entire idea is based on ***Temporal Locality***: if we use it now, we'll want to use it again soon (a Big Idea)

Slides from Patterson's 61C, Fall 2001

Intel Pentium 5 Prescott



Trace Cache Access, next Address Predict

Trace Cache Branch Prediction Table (BTB), 1024 entries.

Return Stacks (4 x16 entries)

Trace Cache next IP's (4x)

Instruction Decoder

Up to 4 decoded uOps/cycle out. (from max. one x86 instr/cycle) Instructions with more than four are handled by Micro Sequencer

Raw Instruction Bytes in

Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total

Instruction TLB's 128 entry, fully associative for 4k and 4M pages. In: Virtual address [47:12] Out: Physical address [39:12] + 2 page level bits

Instruction Fetch from L2 cache and Branch Prediction

Front Side Bus Interface, 533..800 MHz

Instruction Trace Cache

Execution Pipeline Start

Buffer Allocation & Register Rename

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

uOp Schedulers

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)

FP Move Scheduler: (8x8 dependency matrix)

Load / Store Linear Address Collision History Table

Load / Store uOp Scheduler: (8x8 dependency matrix)

FP, MMX, SSE1..3

Floating Point, MMX, SSE1..3 Renamed Register File 256 entries of 128 bit.

Integer Execution Core

(1) uOp Dispatch unit & Replay Buffer
Dispatches up to 6 uOps / cycle

(2) Integer Renamed Register File
256 entries of 32 bit (+ 6 status flags)
12 read ports and six write ports

(3) Databus switch & Bypasses to and
from the Integer Register File.

(4) Flags, Write Back

(5) Double Pumped ALU 0

(6) Double Pumped ALU 1

(7) Load Address Generator Unit

(8) Store Address Generator Unit

(9) Load Buffer (96 entries)

(10) Store Buffer (48 entries)

(11) ROB Reorder Buffer 4x64 entries

(12) 16 kByte Level 1 Data cache
four way set associative. 1R/1W

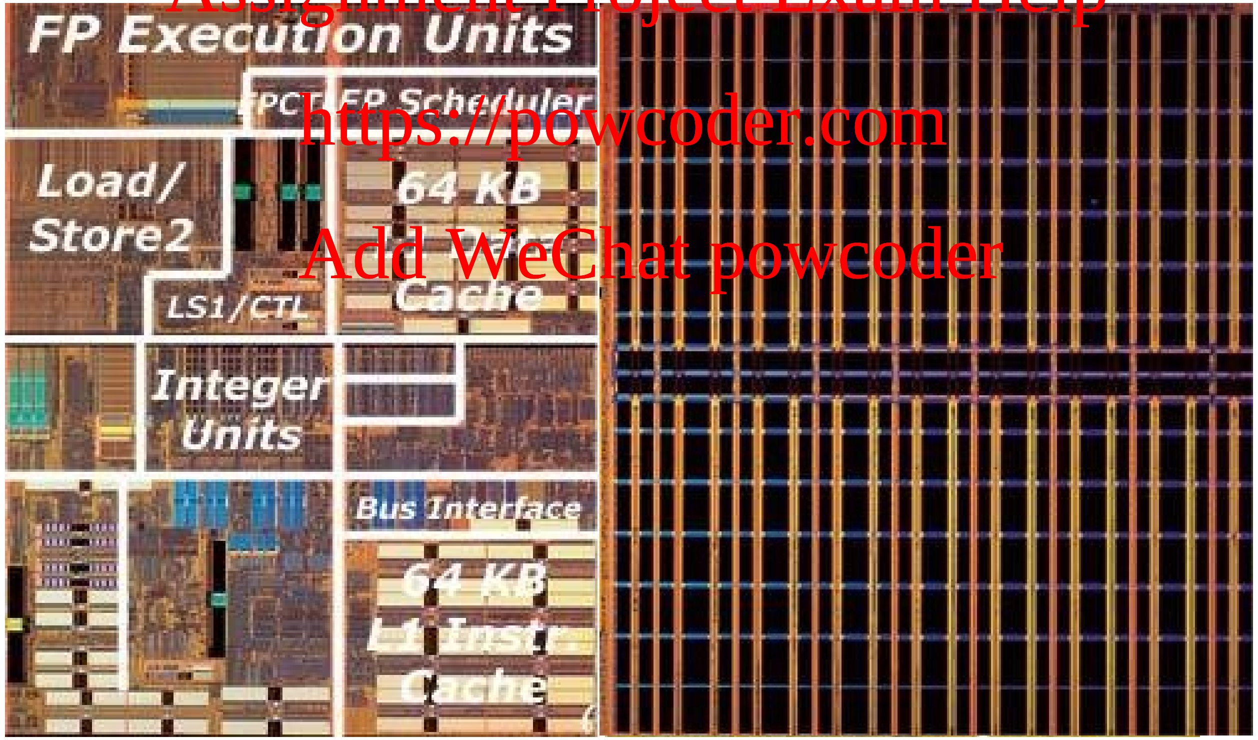
(13) Databus multiplexing

(14) Cache Line Read / Write Transferbuffers and
256 bit wide bus to and from L2 cache

April 19, 2003 www.chip-architect.com

Athlon XP-64 Core

- The greatest share of the surface (over 50 percent) is taken up by the 1 MB L2 cache.



Slides from Patterson's 61C, Fall 2001

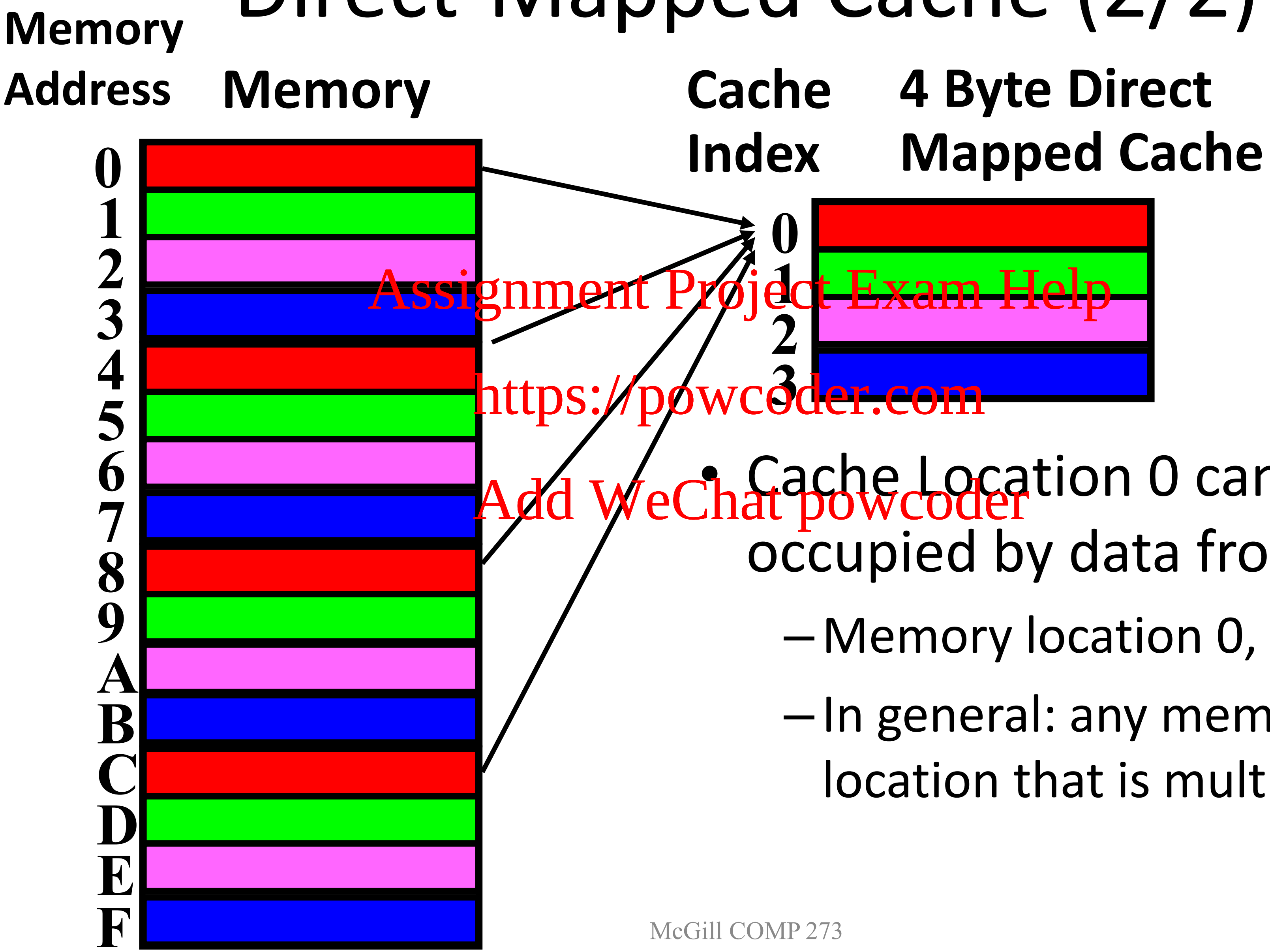
Cache Design

- How do we organize cache?
- Where does each memory address map to?
  - Remember that cache is subset of memory, so multiple memory addresses map to the same cache location.
- How do we know which elements are in cache?
- How do we quickly locate them?

Direct-Mapped Cache (1/2)

- In a ***direct-mapped cache***, each memory address is associated with one possible ***block*** within the cache
  - Therefore, we only need to look in a single location in the cache to see if the data exists in the cache
  - A *block* is the unit of transfer between cache and memory

Direct-Mapped Cache (2/2)



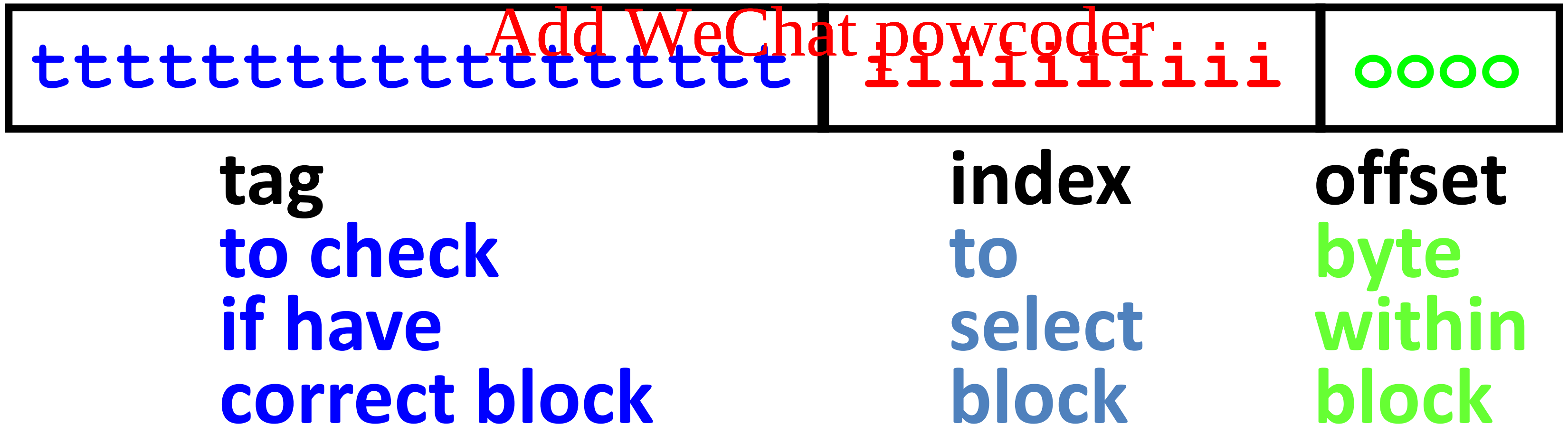
- Cache Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ...
  - In general: any memory location that is multiple of 4

Issues with Direct-Mapped

1 Since multiple memory addresses map to same cache index, how do we tell which one is in there?

2 What if we have a block size > 1 byte?

- **Solution**: divide memory address into three fields



Direct-Mapped Cache Terminology

- All fields are read as unsigned integers.
- The ***Index***: specifies the cache index (which "row" of the cache we should look in)
- The ***Offset***: once we've found correct block, specifies which byte within the block we want
- The ***Tag***: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

Direct-Mapped Cache Example

- Suppose we have a direct-mapped **16KB cache** with **4 word blocks**.
- Determine the size of the tag, index and offset fields if we're using a 32-bit architecture.

Direct-Mapped Cache Example

- Offset
  - – need to specify correct byte within a block
  - – block contains

    4 words = 16 bytes = 2^4 bytes
  - – need **4 bits** to specify correct byte

Direct-Mapped Cache Example

- Index
  - need to specify correct row in cache
  - cache contains 16 KB = $2^4 2^{10}$ = 2^{14} bytes
    block contains 2^4 bytes (4 words)

# rows/cache = # blocks/cache (there's one block/row)

$$= \frac{\text{bytes/cache}}{\text{bytes/row}}$$

$$= \frac{2^{14}\ \text{bytes/cache}}{2^{4}\ \text{bytes/row}}$$

$$= 2^{10}\ \text{rows/cache}$$

- need ***10 bits*** to specify this many rows

Direct-Mapped Cache Example

- Tag
  - used remaining bits as tag
  - tag length = memory address bits minus offset bits minus index bits

    = 32 - 4 - 10 bits

    = 18 bits
  - so the tag is leftmost ***18 bits*** of memory address

Accessing data in a direct mapped cache

- Example: 16KB, direct-mapped, 4 word blocks
- Read 4 addresses

  0x00000014
  0x0000001C
  0x00000034
  0x00008014
- Memory values on right:
  - Let us only consider cache and memory levels of hierarchy

Memory

Address (hex)	Value of Word
...	...
00000010	a
00000014	b
00000018	c
0000001C	d
...	...
00000030	e
00000034	f
00000038	g
0000003C	h
...	...
00008010	i
00008014	j
00008018	k
0000801C	l
...	...

Slides from Patterson's 61C, Fall 2001

Accessing data in a direct mapped cache

- 4 Addresses:

  0x00000014, 0x0000001C, 0x00000034, 0x00008014

- 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

Tag	Index	Offset
000000000000000000	0000000001	0100
000000000000000000	0000000001	1100
000000000000000000	0000000011	0100
000000000000000010	0000000001	0100

Accessing data in a direct mapped cache

- Lets go through accessing some data in this cache
  - 16KB, direct-mapped, 4 word blocks
- Will see 3 types of events:
- cache miss: nothing in cache in appropriate block, so fetch from memory
- cache hit: cache block is valid and contains proper address, so read desired word
- cache miss, block replacement: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory

16 KB Direct Mapped Cache, 16B blocks

Example Block

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0		🟦	🟦	🟦	🟦
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

***Valid bit*:** determines if anything is stored in that row (when computer initially turned on, all entries are invalid)

Read 0x00000014

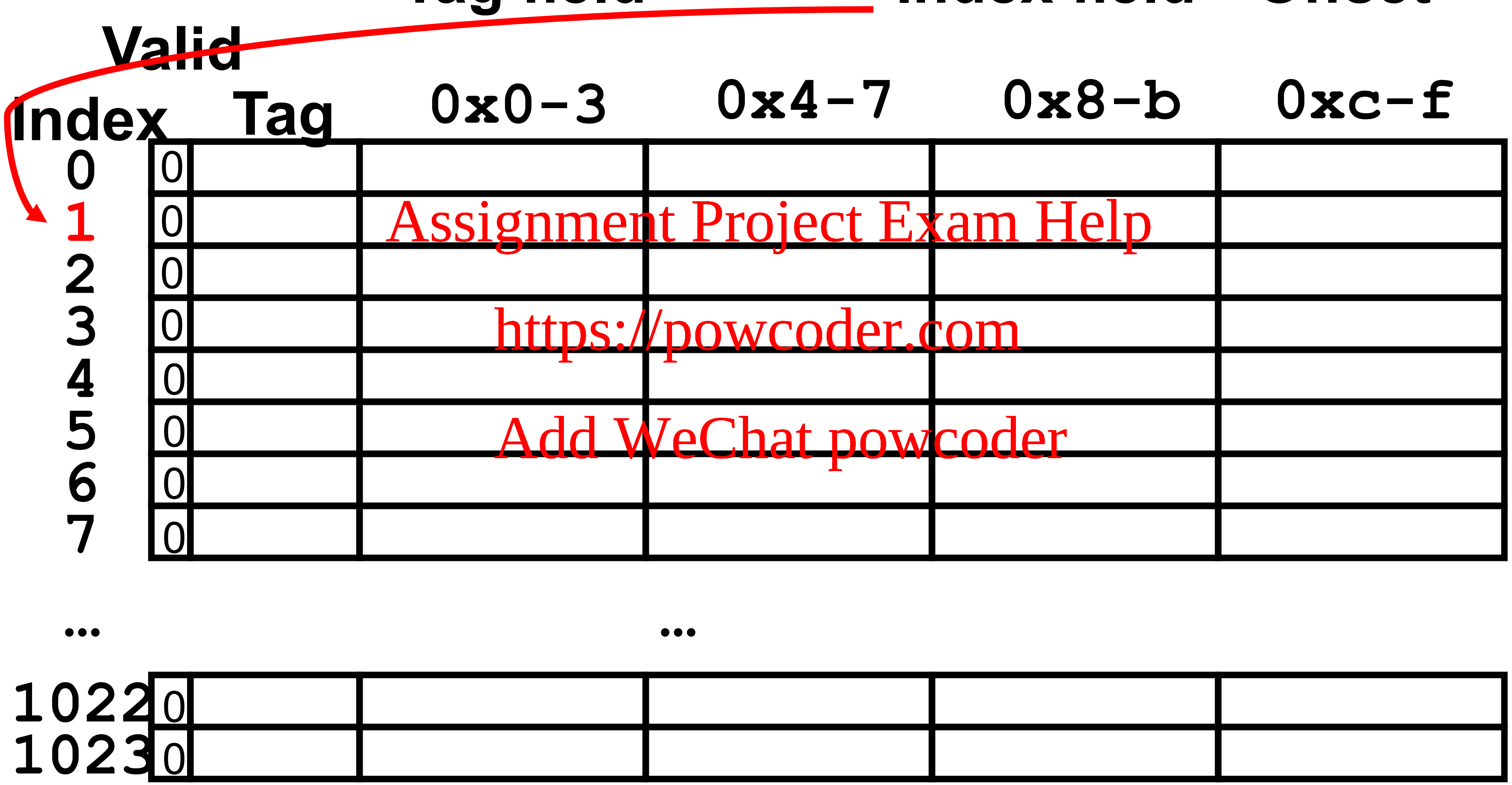
```
000000000000000000  0000000001  0100
     Tag field      Index field Offset
```

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

So we read block 1 (0000000001)

000000000000000000 0000000001 0100

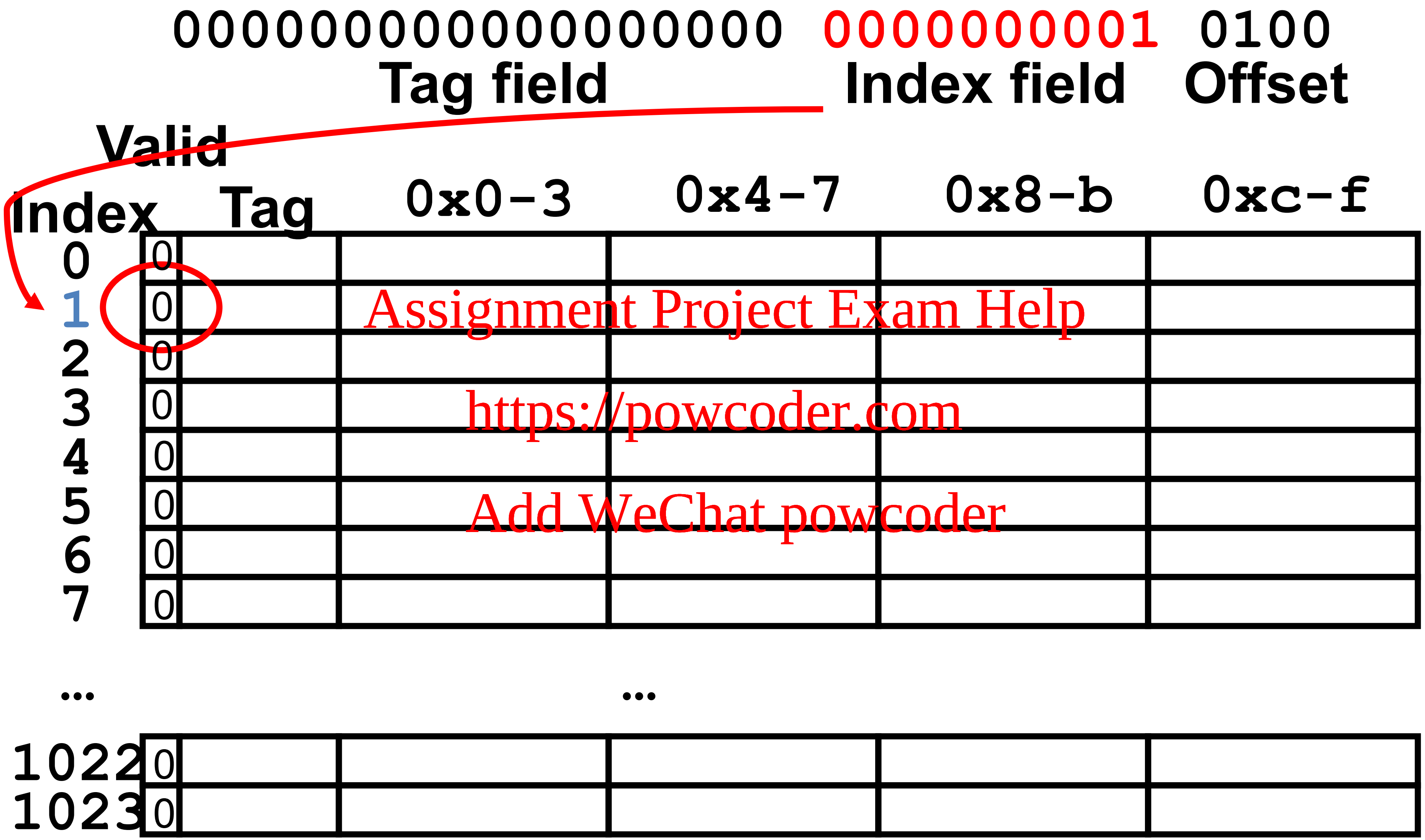
Tag field Index field Offset



Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Slides from Patterson's 61C, Fall 2001

No valid data



```
000000000000000000  0000000001  0100
     Tag field      Index field Offset
```

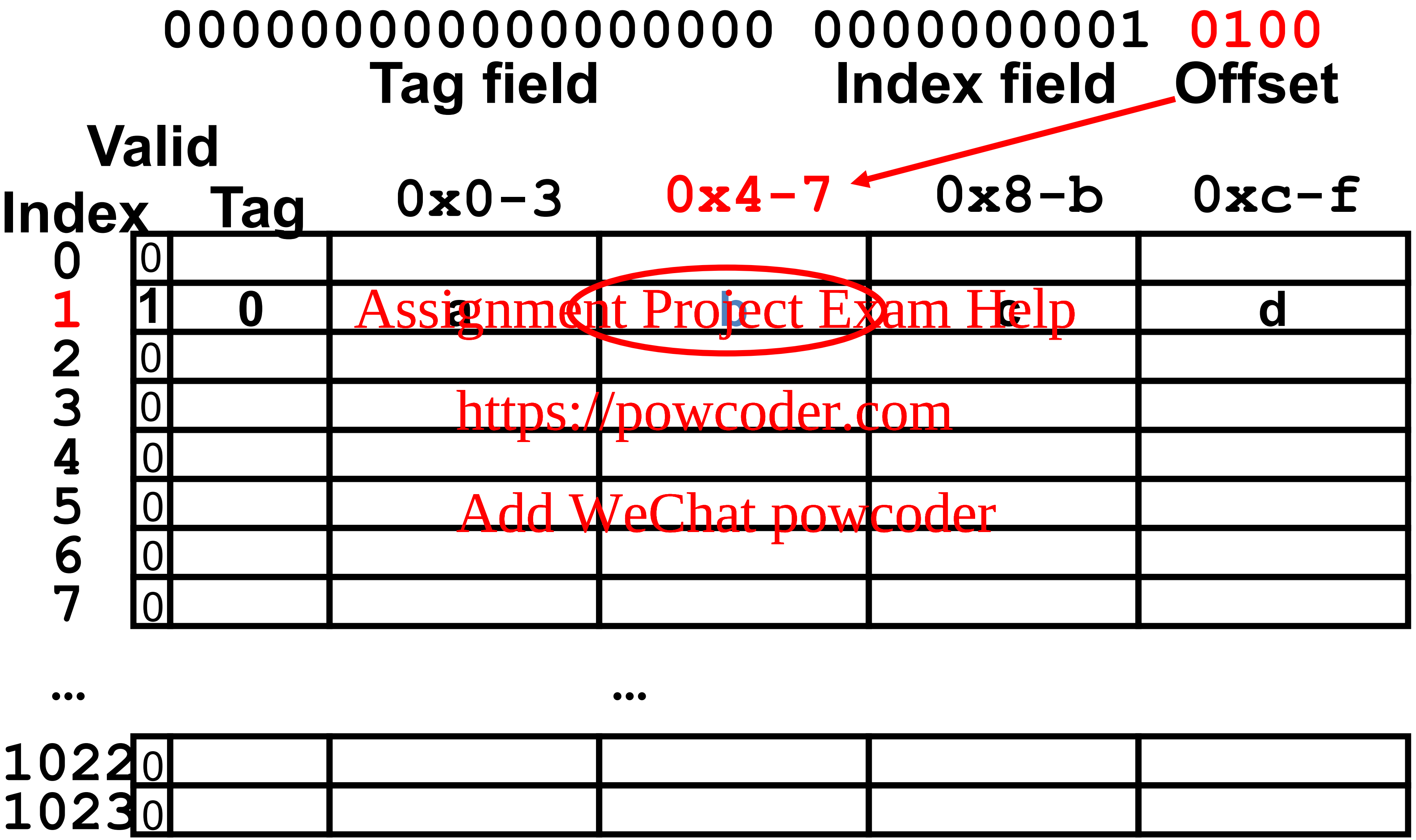
Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

So load that data into cache, setting tag, valid

000000000000000000	0000000001	0100
Tag field	**Index field**	**Offset**

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Read from cache at offset, return word **b**



`0000000000000000000 0000000001 0100`

Tag field **Index field** **Offset**

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read 0x0000001C

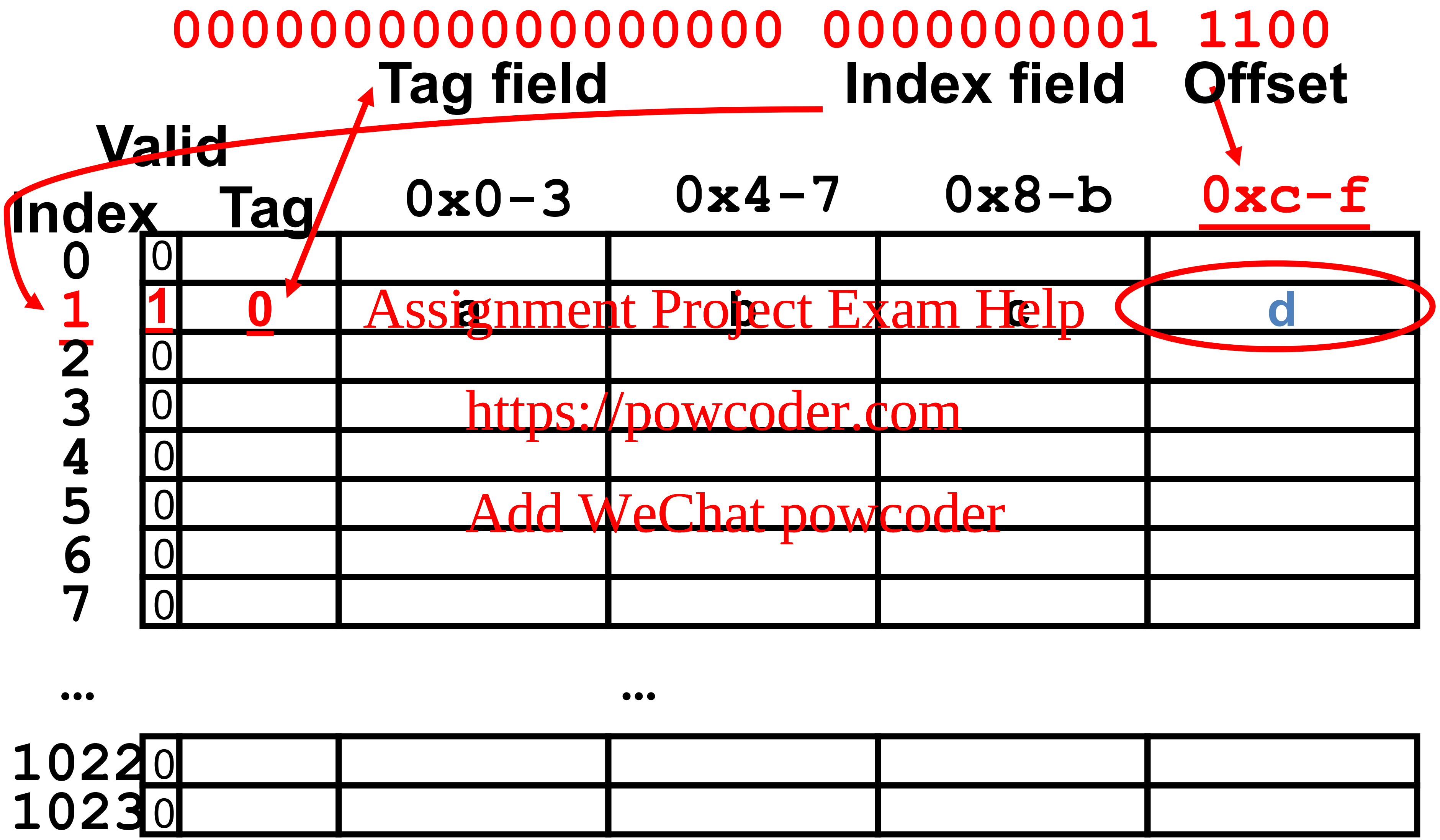
000000000000000000 0000000001 1100

Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Slides from Patterson's 61C, Fall 2001

Data valid, tag OK, so read offset return word **d**



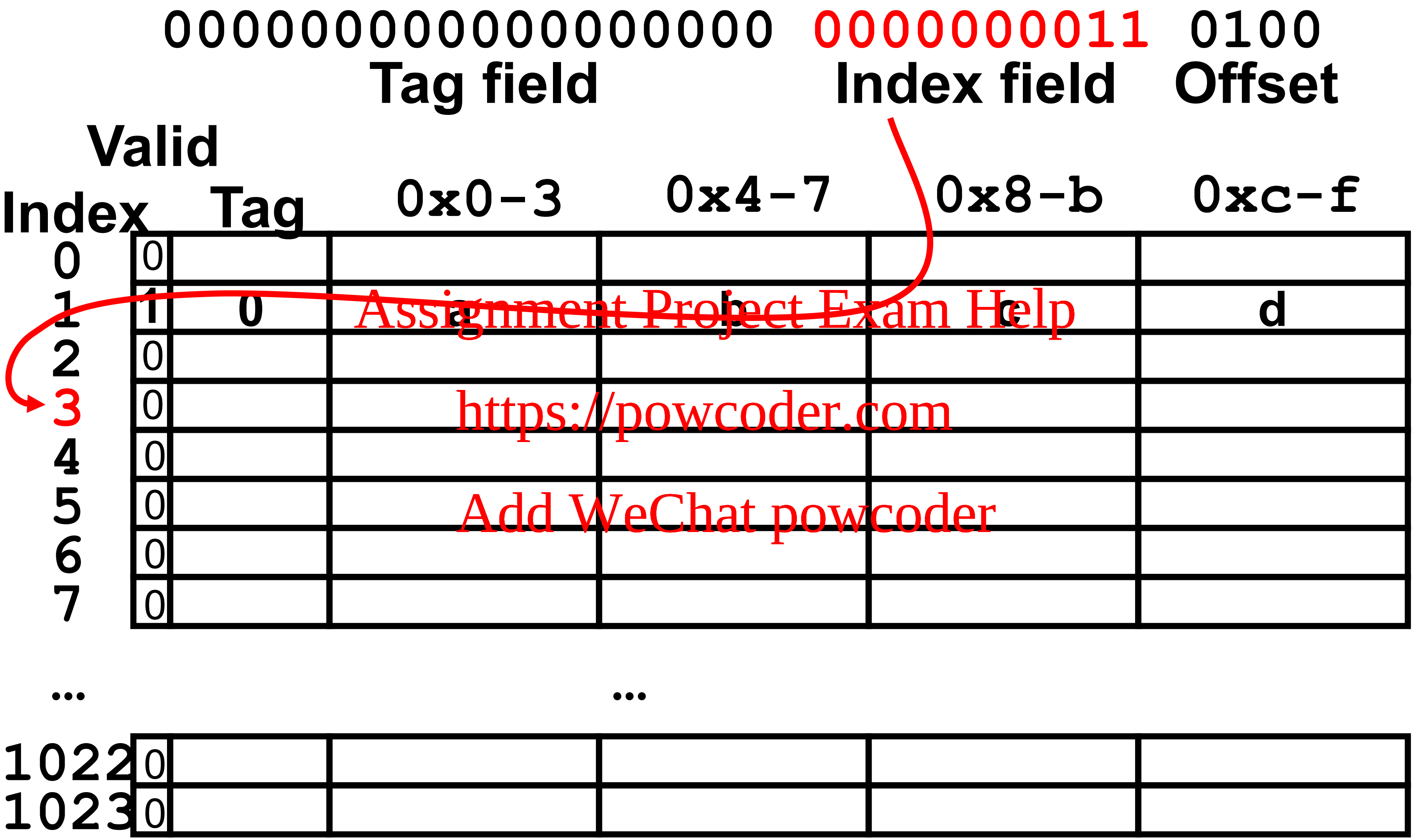
Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read 0x00000034

000000000000000000	0000000011	0100
Tag field	Index field	Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

So read block 3



`000000000000000000 0000000011 0100`

Tag field | Index field | Offset

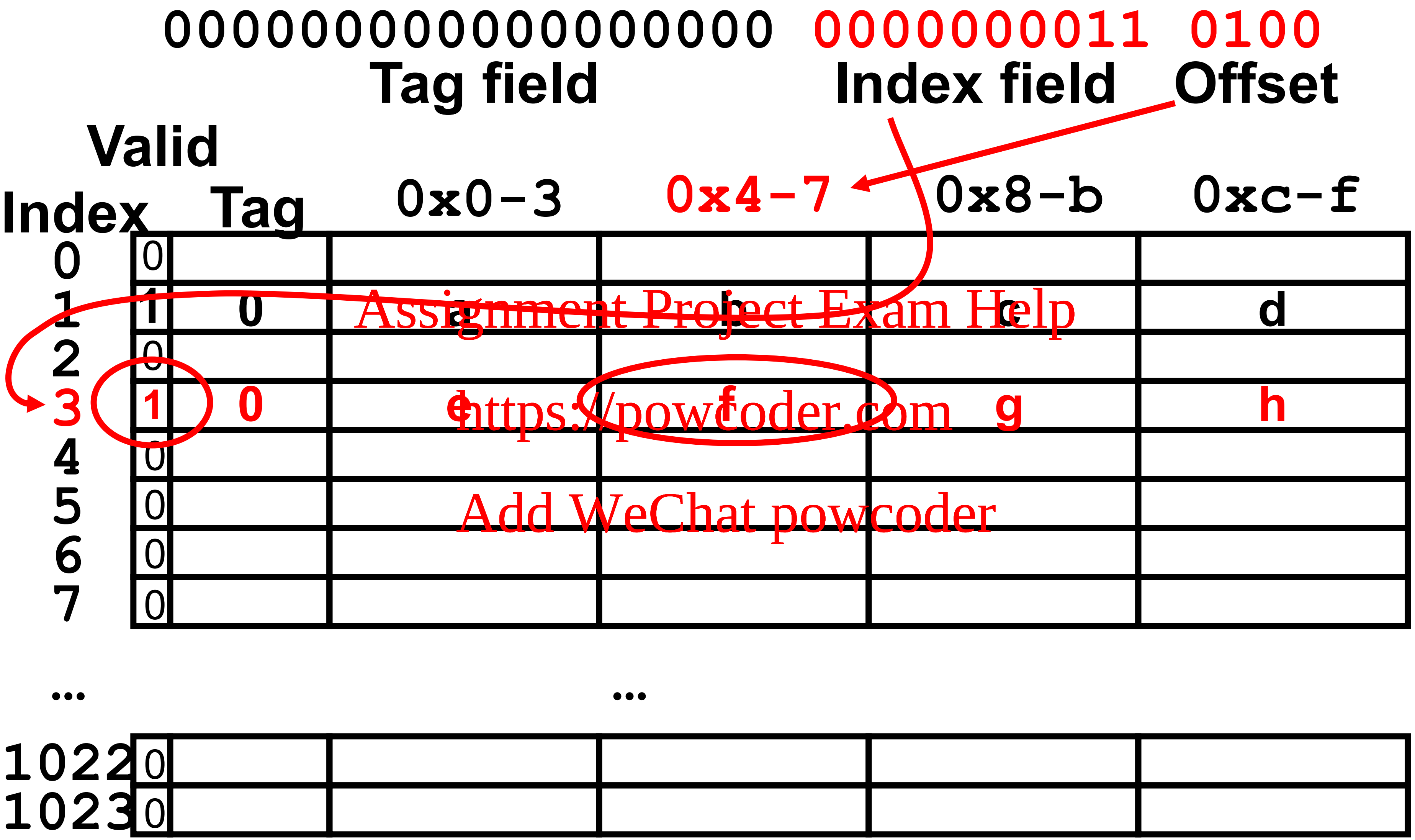
Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

No valid data

00000000000000000	0000000011	0100
Tag field	Index field	Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Load that cache block, return word **f**



```
000000000000000000 0000000011 0100
```

Tag field **Index field** **Offset**

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read 0x00008014

000000000000000010 0000000001 0100

Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Slides from Patterson's 61C, Fall 2001

So we read block 1, Data is Valid

000000000000000000010	0000000001	0100
Tag field	Index field	Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Cache Block 1 Tag does not match (0 != 2)

Tag field	Index field	Offset
000000000000000010	0000000001	0100

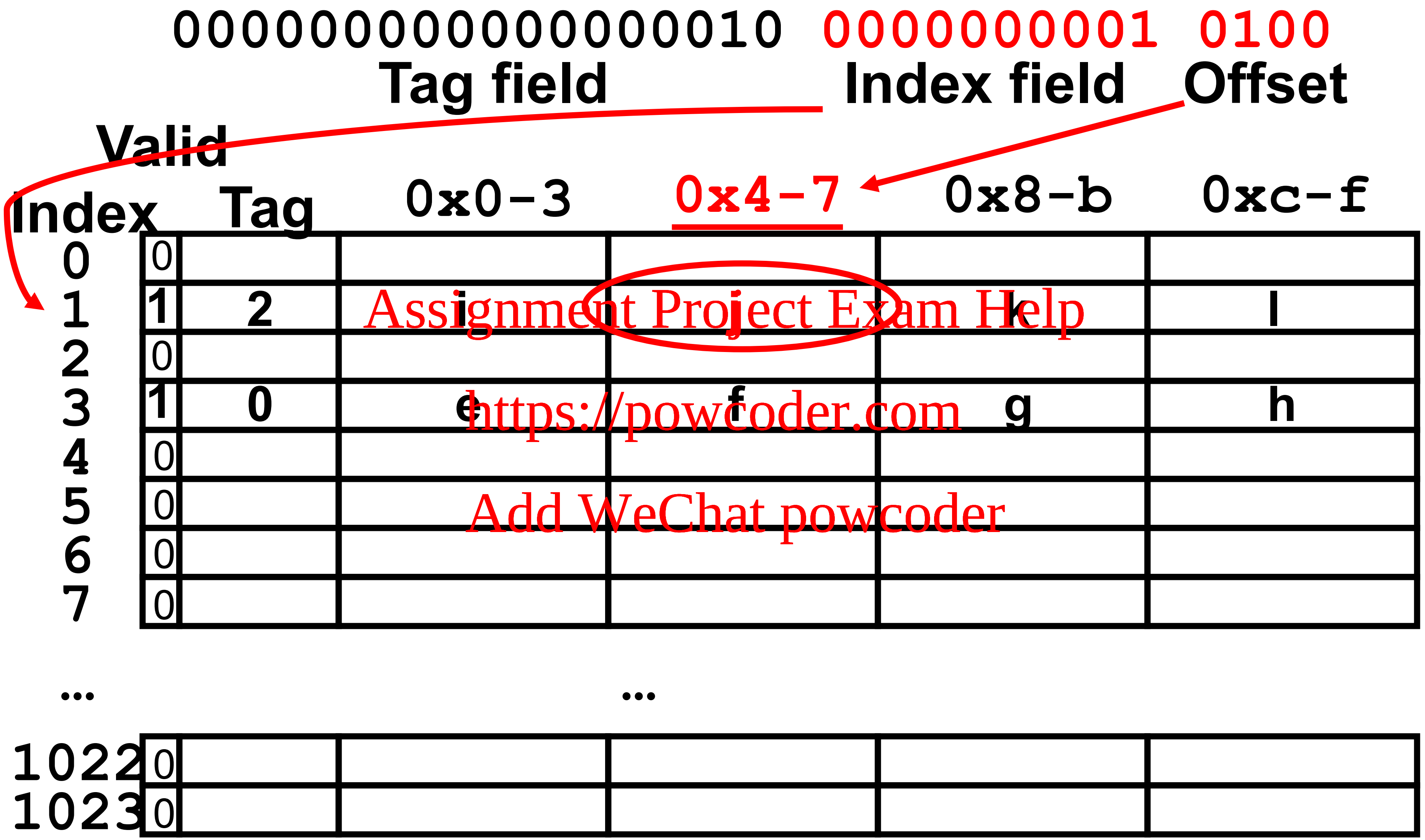
Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Miss, replace block 1 with new data & tag

000000000000000010	0000000001	0100
Tag field	**Index field**	**Offset**

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

And return word **j**



000000000000000010 0000000001 0100

Tag field **Index field** **Offset**

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Slides from Patterson's 61C, Fall 2001

Do an example yourself. What happens?

- Cache: Hit, Miss, Miss with replace ?
  Values returned: a ,b, c, d, e, ..., k, l ?
- Read address 0x00000030 ?
  `000000000000000000 0000000011 0000`
- Read address 0x0000001c ?
  `000000000000000000 0000000001 1100`

Cache

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		

Answers

- 0x00000030 a hit

  Index = 3, Tag matches,
  Offset = 0, value = e
- 0x0000001c a miss

  Index = 1, Tag mismatch, so
  replace from memory,
  Offset = 0xc, value = d
- Therefore, returned values are:
  - 0x00000030 = e
  - 0x0000001c = d

Memory

Address	Value of Word
...	...
00000010	a
00000014	b
00000018	c
0000001c	d
...	...
00000030	e
00000034	f
00000038	g
0000003c	h
...	...
00008010	i
00008014	j
00008018	k
0000801c	l
...	...

"And in Conclusion…"

- We would like to have the capacity of disk at the speed of the processor: unfortunately this is not feasible.
- So we create a memory hierarchy:
  - each successively higher level contains "most used" data from next lower level
  - exploits temporal locality and spatial locality
  - do the common case fast, worry less about the exceptions (design principle of MIPS)
- Locality of reference is a Big Idea

Slides from Patterson's 61C, Fall 2001

Review and More Information

- Sections 5.1 - 5.3 of textbook