2023年度（令和5年）版

Ver. 2023-10-12a

Course number: CSC.T363

コンピュータアーキテクチャ Computer Architecture

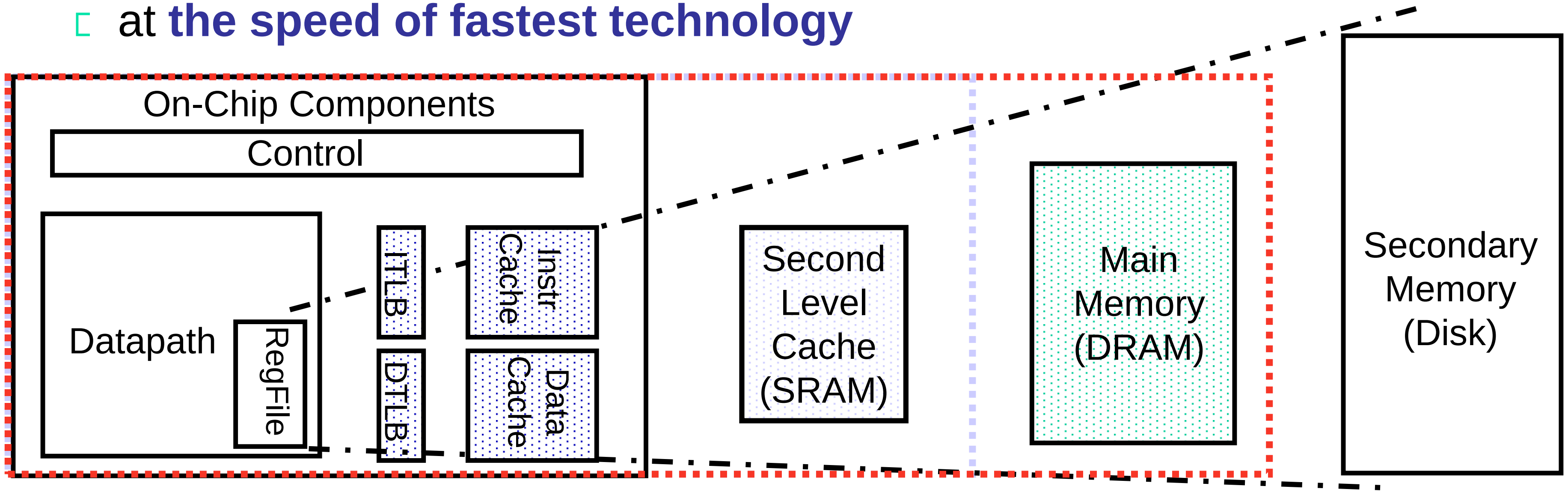
4. キャッシュ：ダイレクトマップ方式 Caches: Direct-Mapped

www.arch.cs.titech.ac.jp/lecture/CA/
Tue 13:30-15:10, 15:25-17:05
Fri 13:30-15:10

吉瀬 謙二　情報工学系
Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

A Typical Memory Hierarchy

- By taking advantage of **the principle of locality** (局所性)
  - Present **much memory** in **the cheapest technology**
  - at **the speed of fastest technology**



Speed (%cycles):	½'s	1's	10's	100's	1,000's
Size (bytes):	100's	K's	10K's	M's	G's to T's
Cost:	highest				lowest

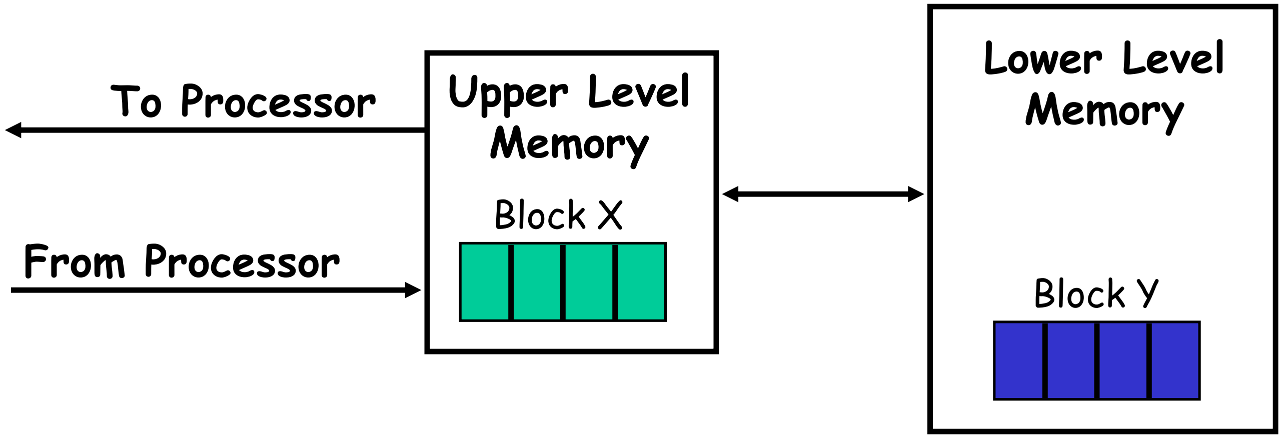
TLB: Translation Lookaside Buffer

パレートの法則

- Vilfredo Federico Damaso Pareto
  - イタリアの経済学者(1948 – 1923)
- パレートの法則
  - 全体の数値の大部分は，全体を構成するうちの一部の要素が生み出している
  - 80:20の法則

The Memory Hierarchy: Why Does it Work?

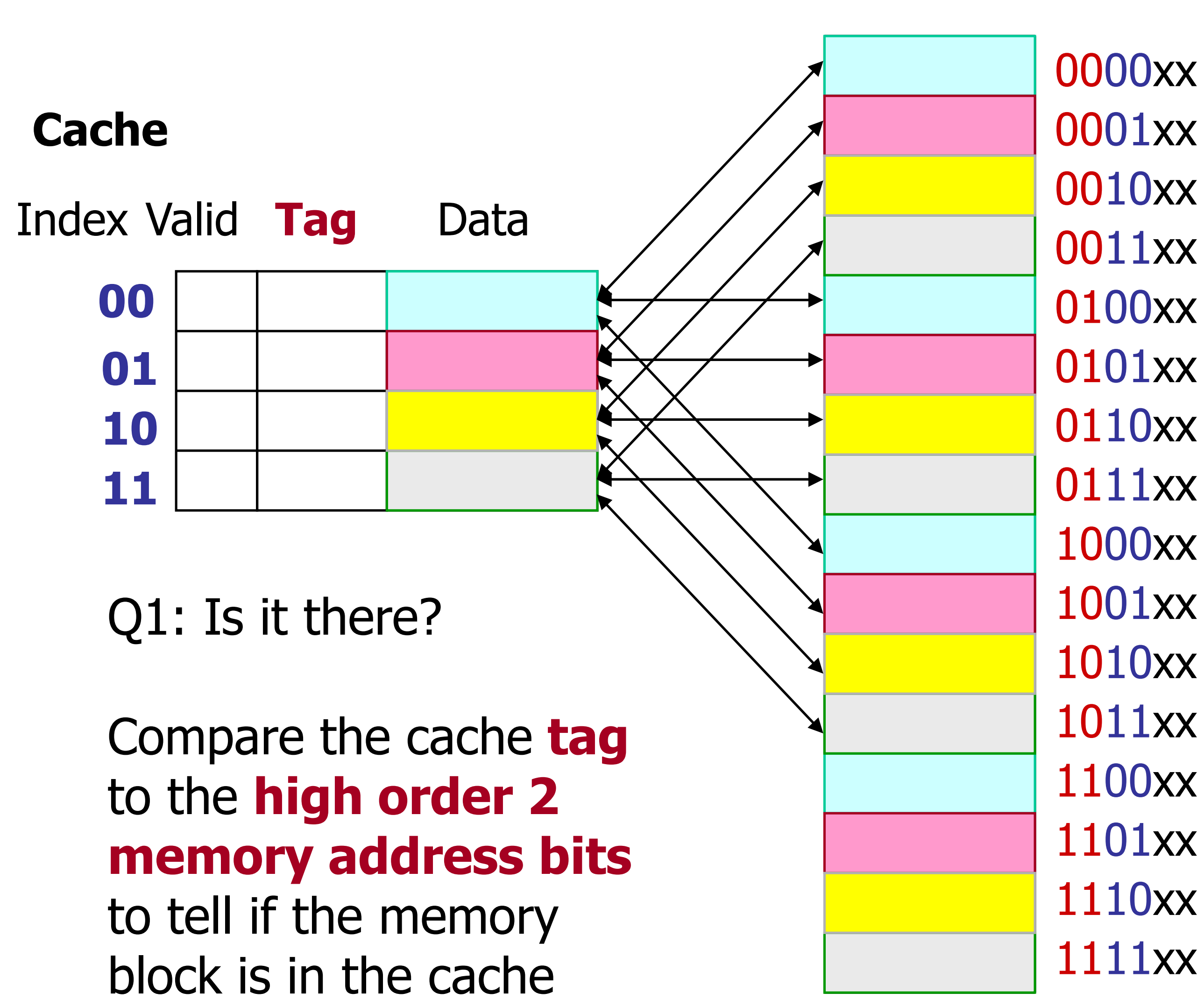
- **Temporal Locality** (時間的局所性, Locality in Time):
  ⇒ Keep **most recently accessed** data items closer to the processor
- **Spatial Locality** (空間的局所性, Locality in Space):
  ⇒ Move blocks consisting of **contiguous words** to the upper levels



Cache

- Two questions to answer (in hardware):
  - Q1: **How do we know if a data item is in the cache?**
  - Q2: **If it is, how do we find it?**
- **Direct mapped**
  - For each item of data at the lower level, there is **exactly one location** in the cache where it might be - so lots of items at the lower level must **share** locations in the upper level
  - Address mapping:
    (block address) modulo (# of blocks in the cache)
  - First, consider block sizes of **one word**

Caching: A Simple First Example



Two low order bits define the byte in the word (32-b words)

Q1: Is it there?

Compare the cache **tag** to the **high order 2 memory address bits** to tell if the memory block is in the cache

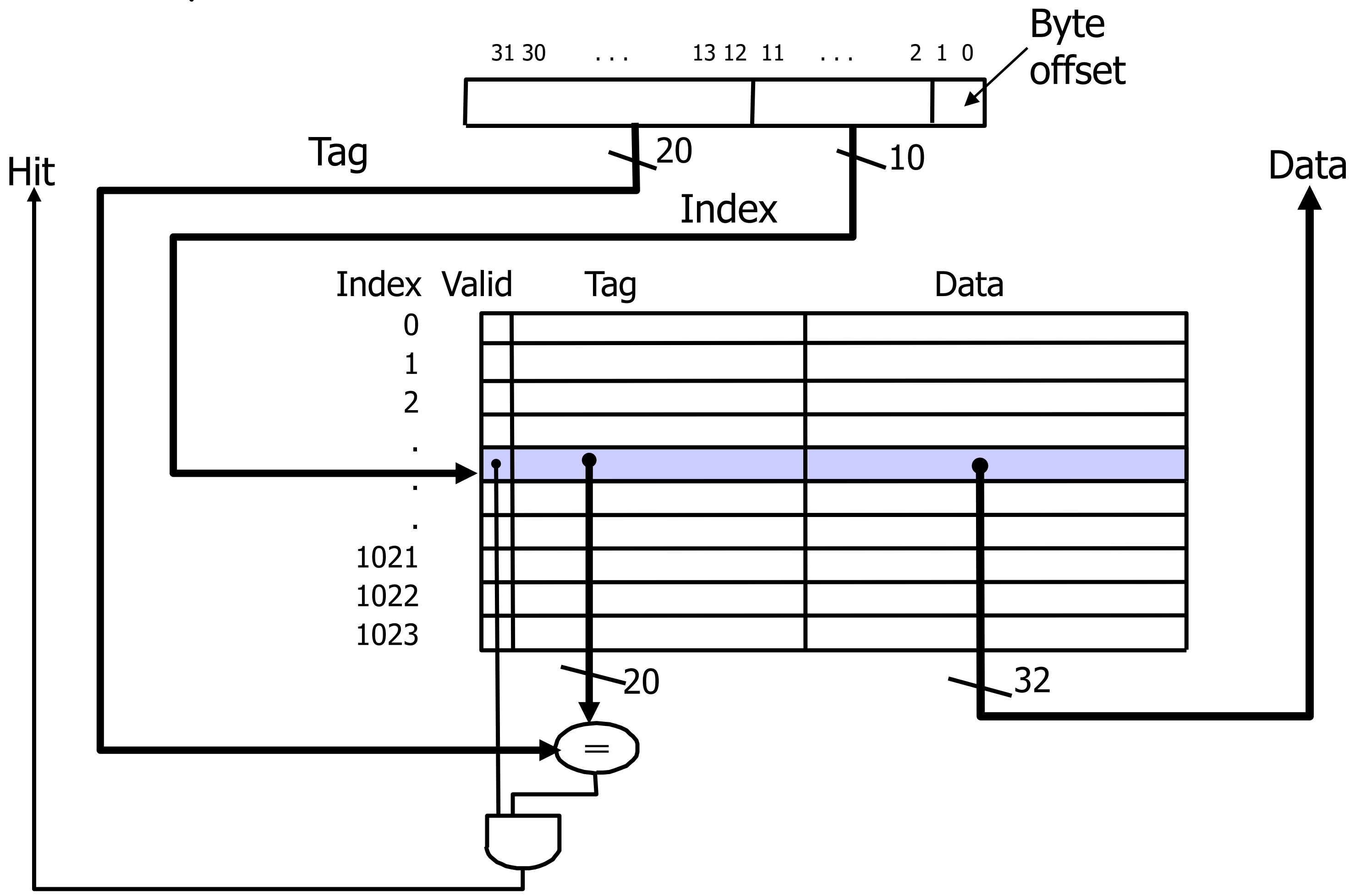
Q2: How do we find it?

Use **next 2 low order memory address bits** – the **index** – to determine which cache block

(block address) modulo (# of blocks in the cache)

MIPS Direct Mapped Cache Example

- One word/block, cache size = 1K words



What kind of locality are we taking advantage of?

Example Behavior of Direct Mapped Cache

- Consider the main memory word reference string (word addresses) 0 1 2 3 4 3 4 15

Start with an empty cache - all blocks initially marked as not valid

0 miss

Tag	
00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

01 ~~00~~	~~Mem(0)~~ 4
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

15 miss

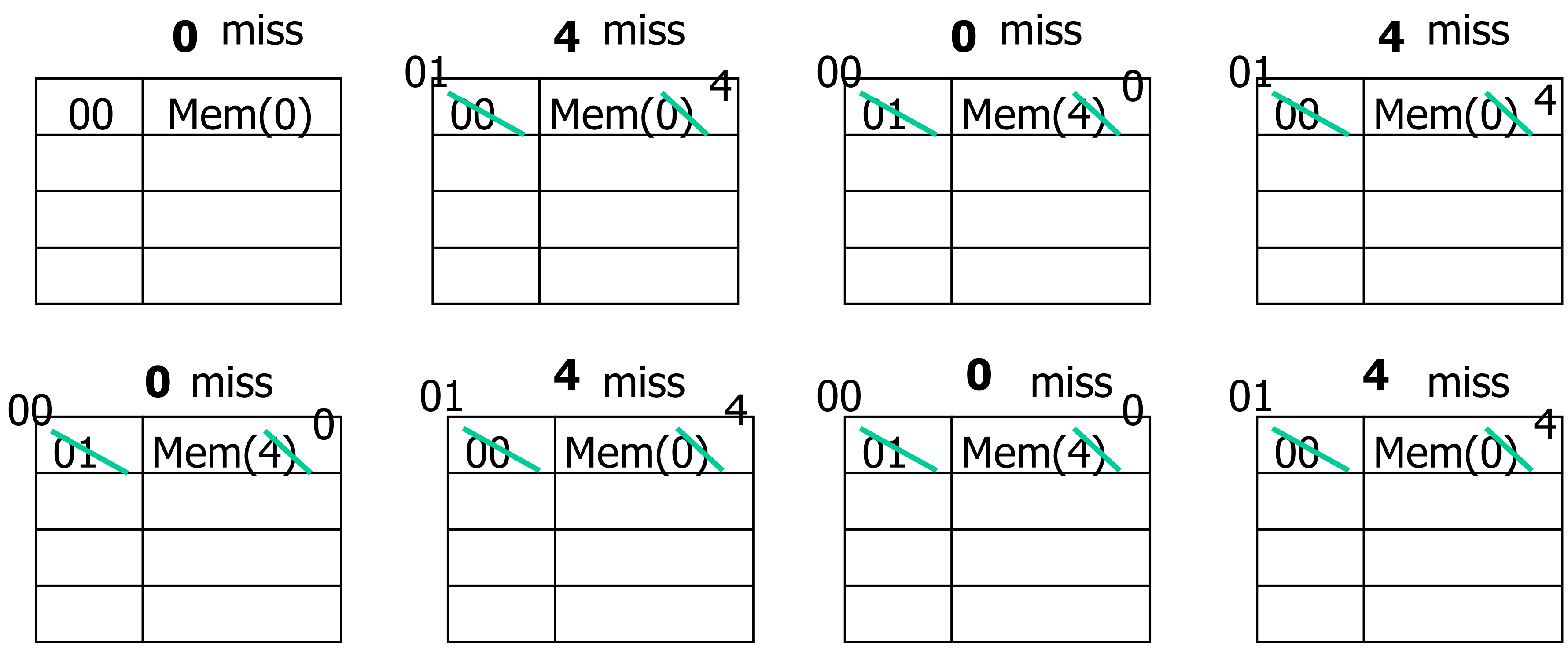
01	Mem(4)
00	Mem(1)
00	Mem(2)
11 ~~00~~	~~Mem(3)~~ 15

- 8 requests, 6 misses

Another Reference String Mapping

- Consider the main memory word reference string

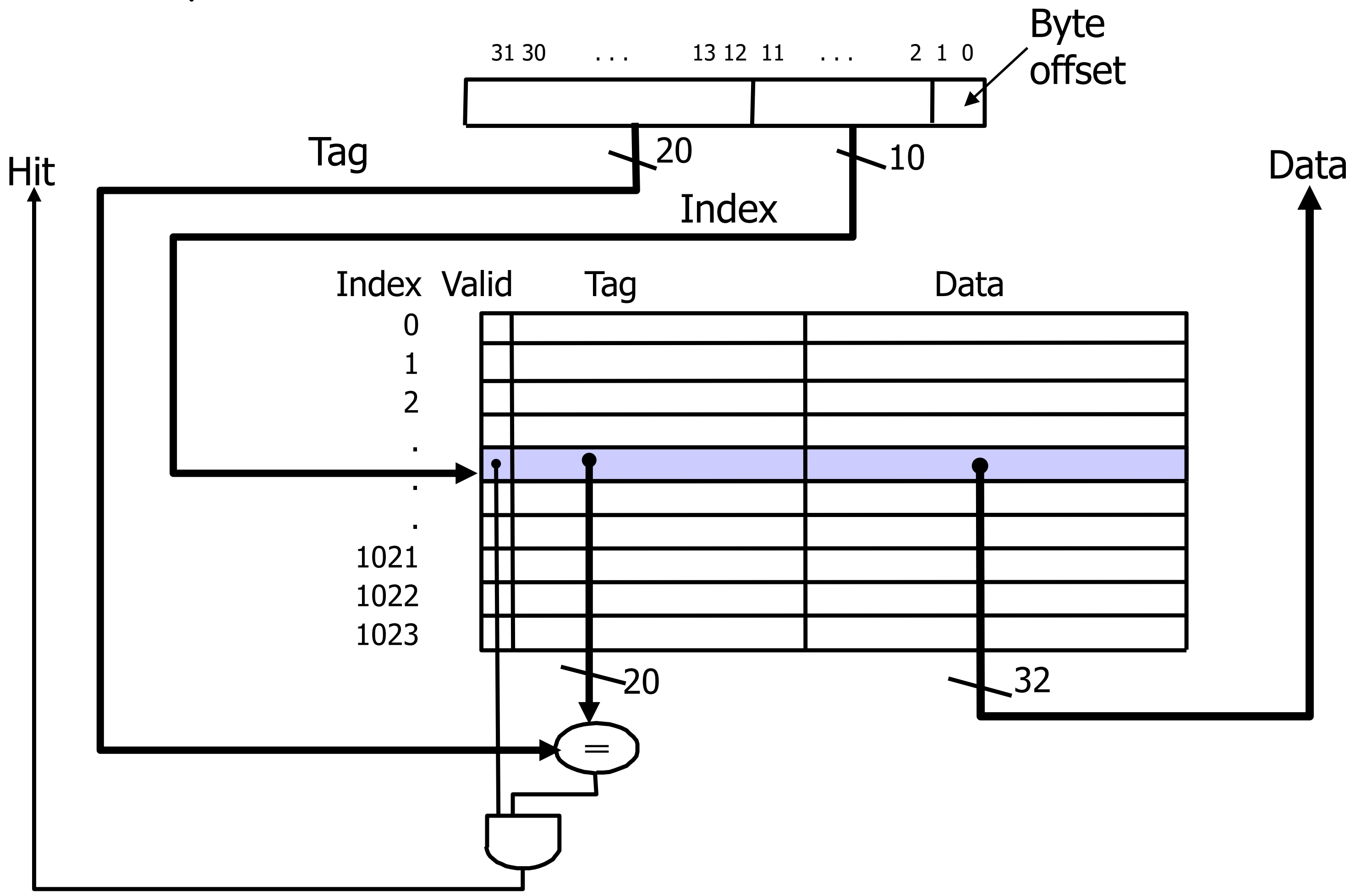
  0 4 0 4 0 4 0 4



- 8 requests, 8 misses
  - Ping pong effect due to **conflict** misses - two memory locations that map into the same cache block

MIPS Direct Mapped Cache Example

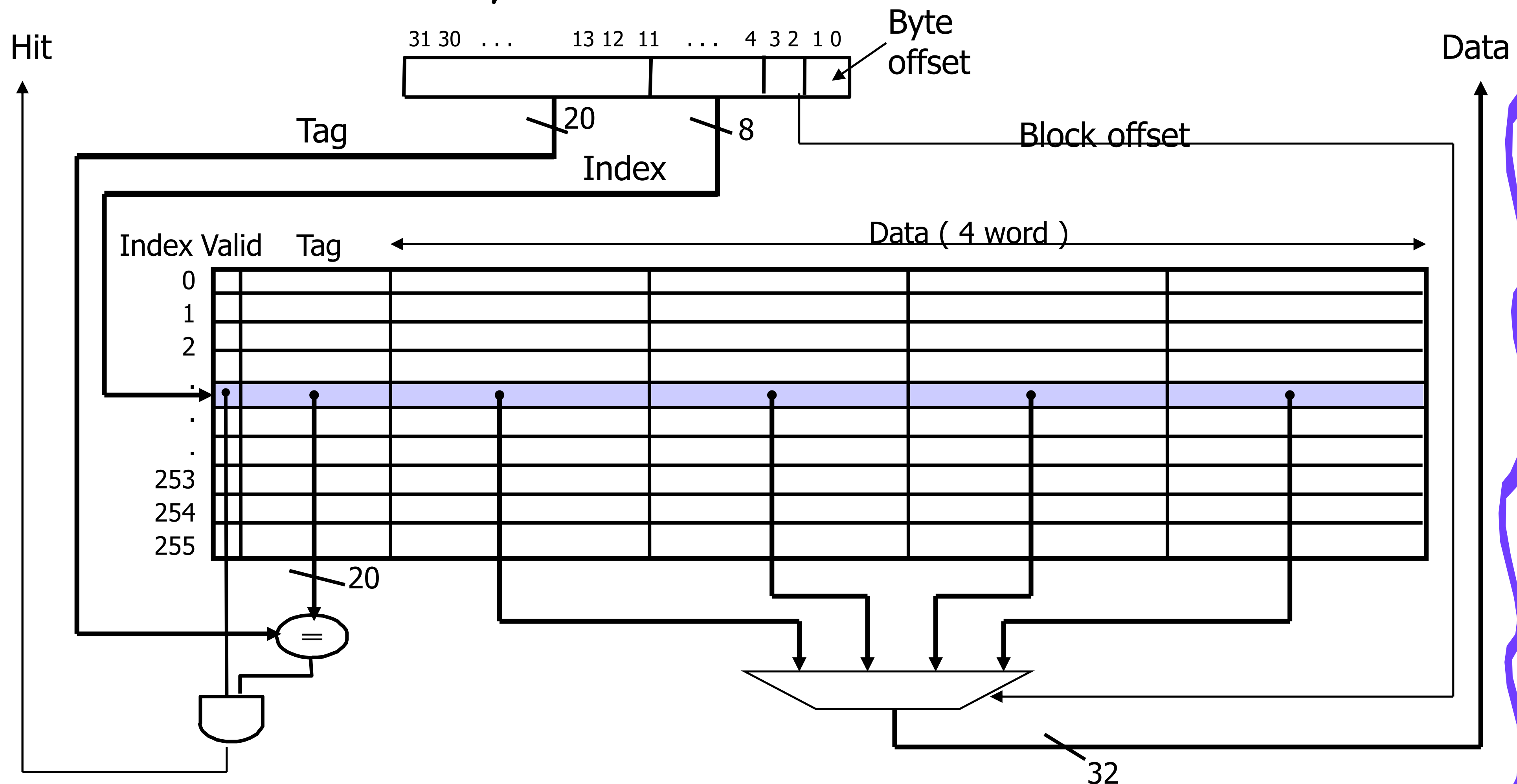
- One word/block, cache size = 1K words



What kind of locality are we taking advantage of?

Multiword Block Direct Mapped Cache

- Four words/block, cache size = 1K words



What kind of locality are we taking advantage of?

Taking Advantage of **Spatial Locality**

- Let cache block hold more than one word

0 1 2 3 4 3 4 15

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

4 miss

01 ~~00~~	~~Mem(1)~~ 5	~~Mem(0)~~ 4
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

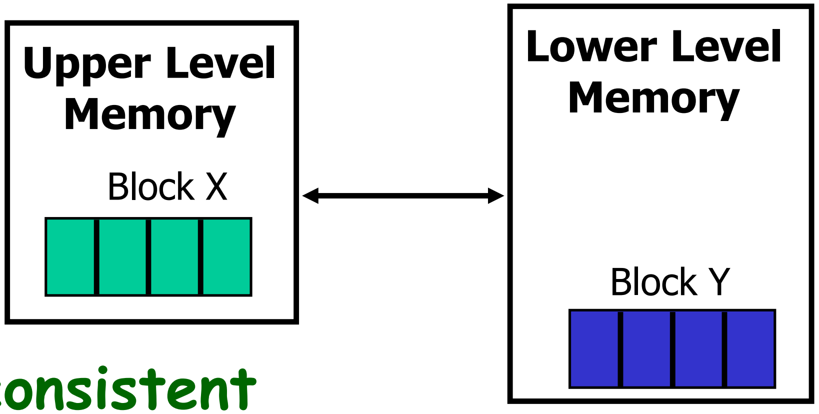
15 miss

01	Mem(5)	Mem(4)
11 ~~00~~	~~Mem(3)~~ 15	~~Mem(2)~~ 14

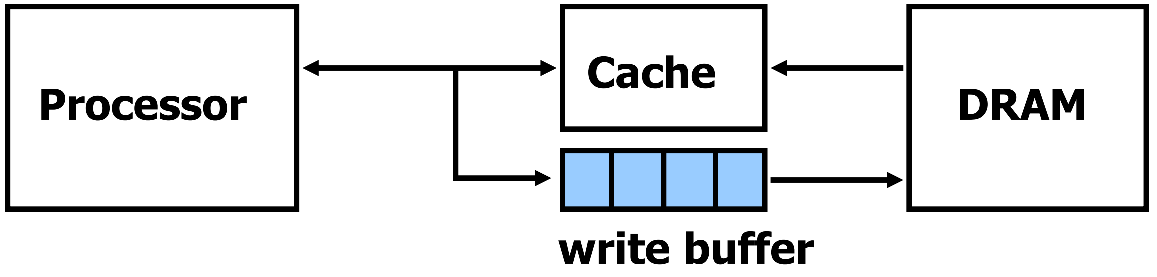
- 8 requests, 4 misses

Handling Cache Hits (Miss is the next issue)

- **Read hits (I$ and D$)**
  - this is what we want!
- **Write hits (D$ only)**
  - allow cache and memory to be **inconsistent**
    - write the data only into the cache block (**write-back**)
    - need a **dirty** bit for each data cache block to tell if it needs to be written back to memory when it is evicted
  - require the cache and memory to be **consistent**
    - always write the data into both the cache block and the next level in the memory hierarchy (**write-through**) so don't need a dirty bit
    - writes run at the speed of the next level in the memory hierarchy – **so slow**! – or can use a **write buffer**, so only have to stall if the write buffer is full



Write Buffer for Write-Through Caching

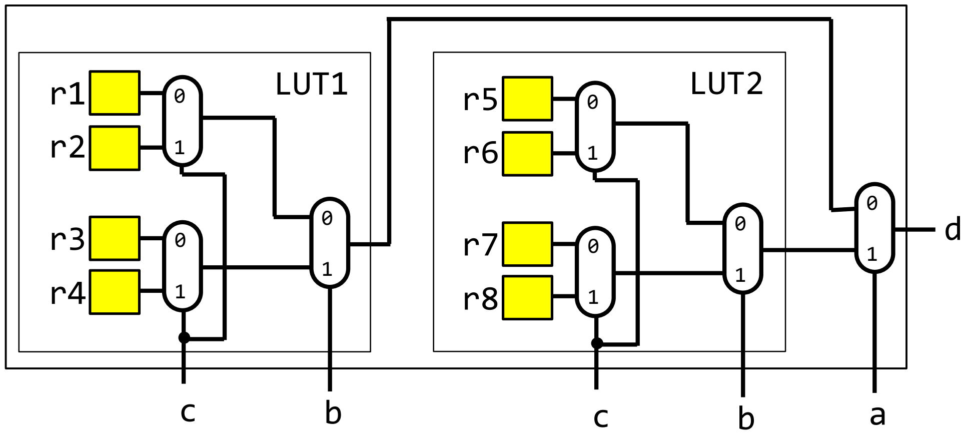


- **Write buffer** between the cache and main memory
    - Processor: writes data into the cache and the write buffer
    - **Memory controller**: writes contents of the write buffer to memory
- The write buffer is just a **FIFO**
    - Typical number of entries: 4
    - Works fine if store frequency is low
- Memory system designer's nightmare, Write buffer **saturation**
    - One solution is to use a write-back cache; another is to use an L2 cache

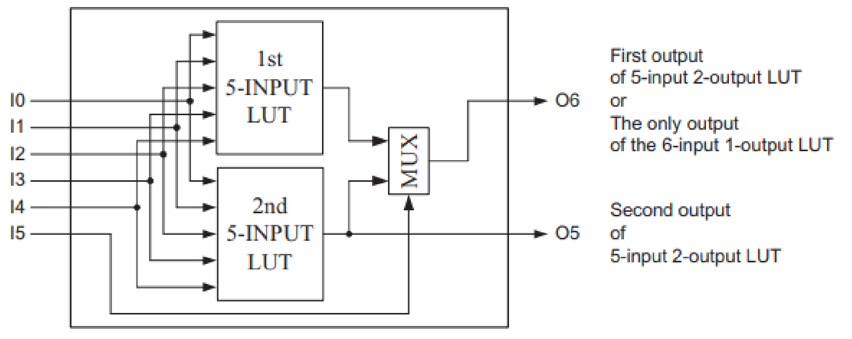
Handling Cache **Misses**

- **Read misses (I$ and D$)**
  - stall the entire pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume
- **Write misses (D$ only)**
  - Write allocate
    - (a) write the word into the cache updating both the tag and data, no need to check for cache hit, no need to stall
    - **(b) stall** the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache, write the word from the processor to the cache, then let the pipeline resume
  - No-write allocate – skip the cache write and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn't full;
    must invalidate **the cache block** since it will be **inconsistent**

ルックアップテーブル (Lookup Table, LUT)



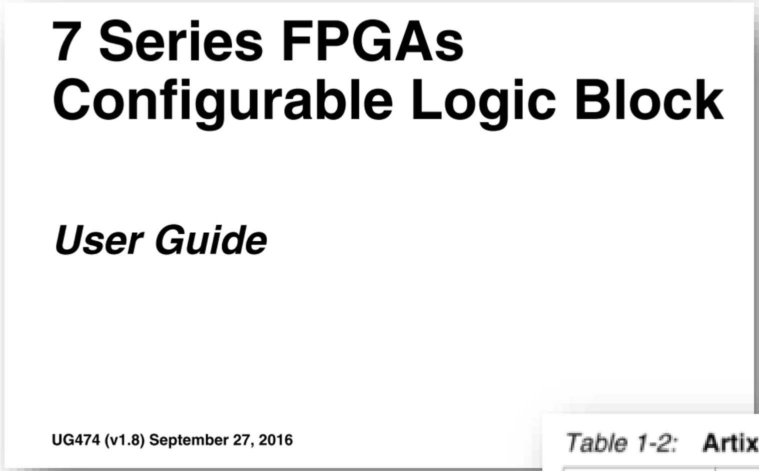
2個の2入力のLUTで3入力のLUTを構成



I0
I1
I2
I3
I4
I5
1st 5-INPUT LUT
2nd 5-INPUT LUT
MUX
O6
O5
First output of 5-input 2-output LUT or The only output of the 6-input 1-output LUT
Second output of 5-input 2-output LUT

Fig. 3. Choice between 1-output 6-input LUT and 2-output 5-input LUT in Xilinx FPGA devices.

Xilinx 7 Series FPGA Configuration Logic Block (CLB)

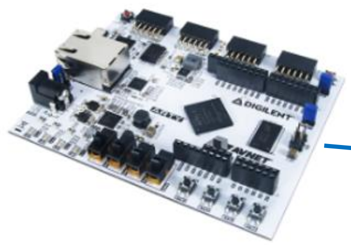


7 Series FPGAs
Configurable Logic Block

User Guide

UG474 (v1.8) September 27, 2016

Slices = SLICEL + SLICEM
Distributed RAM (bit) = SLICEM * 256



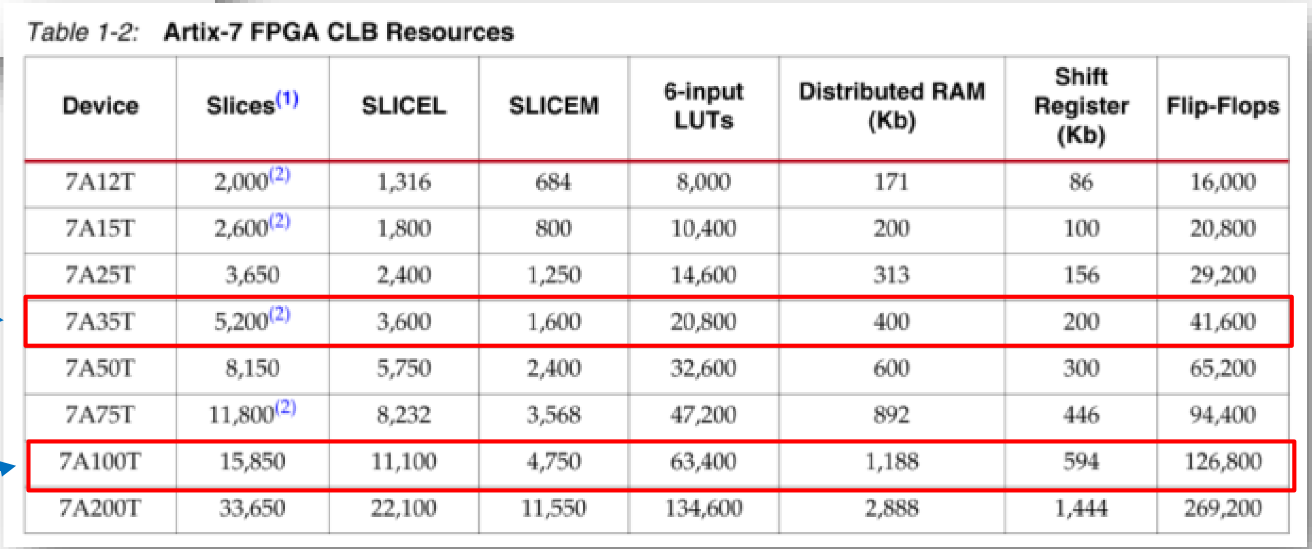


Table 1-2: **Artix-7 FPGA CLB Resources**

Device	Slices[1]	SLICEL	SLICEM	6-input LUTs	Distributed RAM (Kb)	Shift Register (Kb)	Flip-Flops
7A12T	2,000[2]	1,316	684	8,000	171	86	16,000
7A15T	2,600[2]	1,800	800	10,400	200	100	20,800
7A25T	3,650	2,400	1,250	14,600	313	156	29,200
7A35T	5,200[2]	3,600	1,600	20,800	400	200	41,600
7A50T	8,150	5,750	2,400	32,600	600	300	65,200
7A75T	11,800[2]	8,232	3,568	47,200	892	446	94,400
7A100T	15,850	11,100	4,750	63,400	1,188	594	126,800
7A200T	33,650	22,100	11,550	134,600	2,888	1,444	269,200

Xilinx 7 Series Configuration Logic Block (CLB)

SLICEM

SLICEL

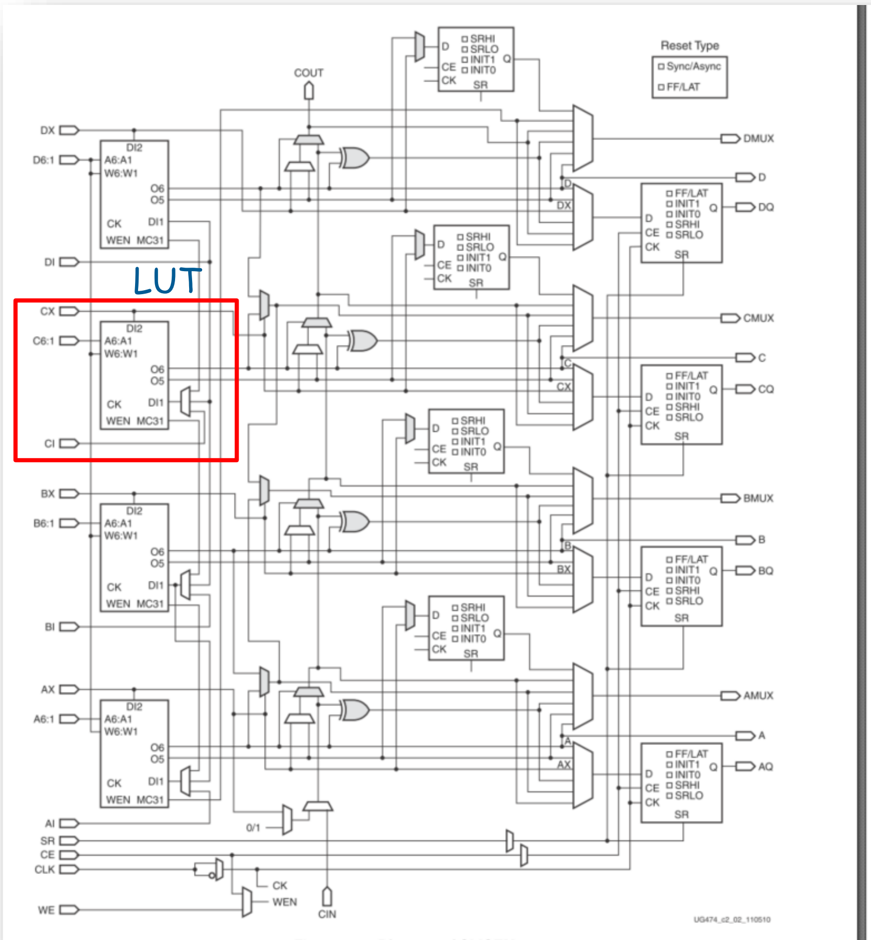
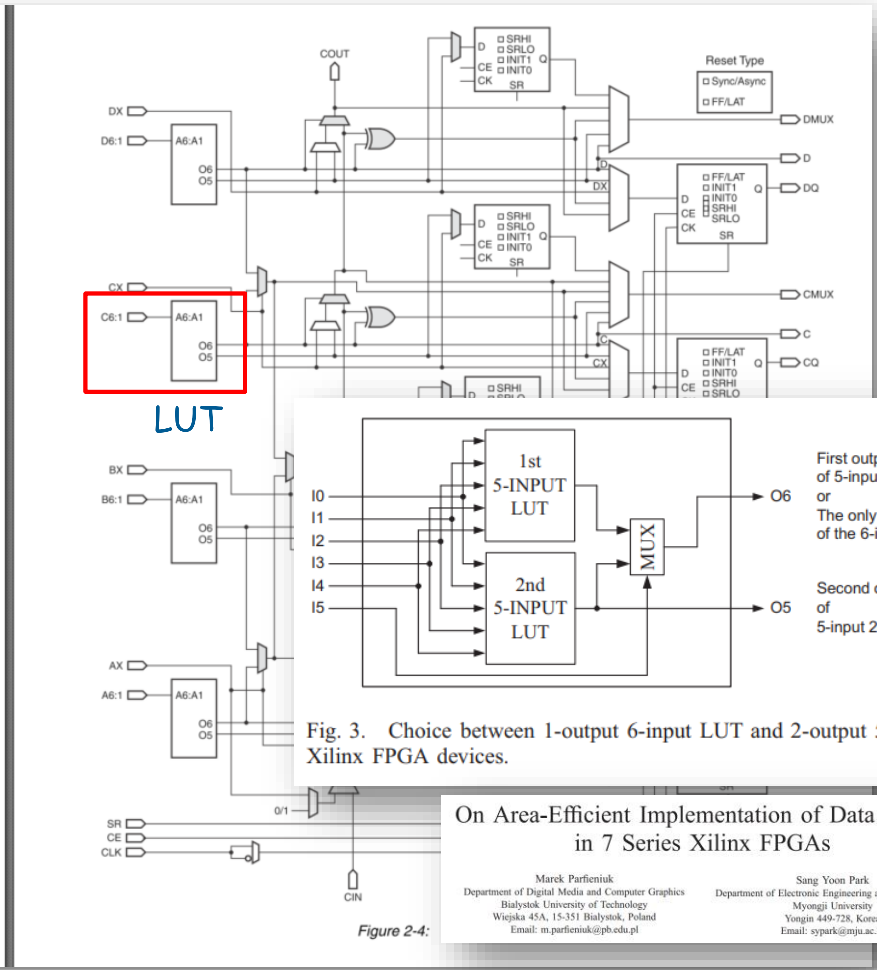


Figure 2-3: **Diagram of SLICEM**



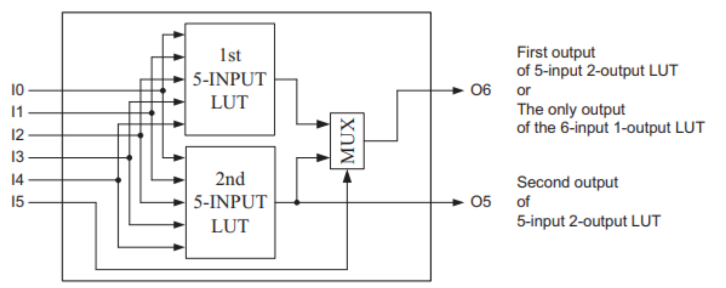

I0
I1
I2
I3
I4
I5
1st 5-INPUT LUT
2nd 5-INPUT LUT
MUX
O6
O5
First output of 5-input 2-output LUT or The only output of the 6-input 1-output LUT
Second output of 5-input 2-output LUT

Fig. 3. Choice between 1-output 6-input LUT and 2-output 5-input LUT in Xilinx FPGA devices.

On Area-Efficient Implementation of Data Delays in 7 Series Xilinx FPGAs

Marek Parfieniuk
Department of Digital Media and Computer Graphics
Bialystok University of Technology
Wiejska 45A, 15-351 Bialystok, Poland
Email: m.parfieniuk@pb.edu.pl

Sang Yoon Park
Department of Electronic Engineering and MPEES-ARC
Myongji University
Yongin 449-728, Korea
Email: sypark@mju.ac.kr

Distributed RAM

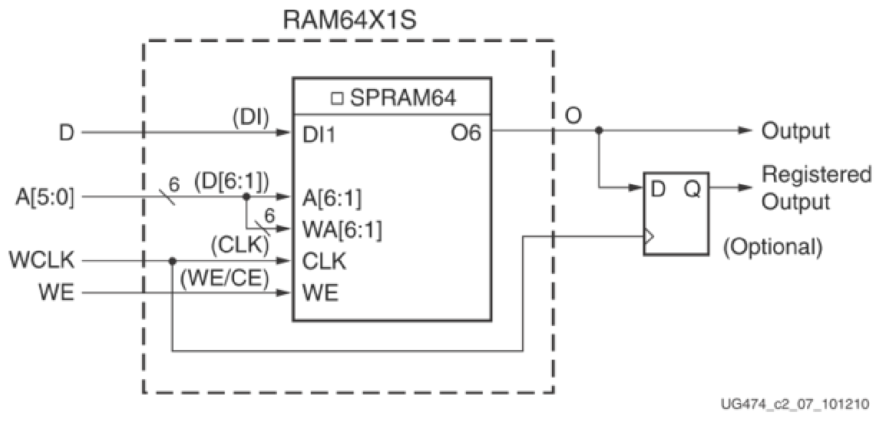


Figure 2-8: **64 X 1 Single Port Distributed RAM (RAM64X1S)**

```
module m_RAM64X1S (clk, a, d, we, dout);
  input wire clk;
  input wire [5:0] a;
  input wire d, we;
  output wire dout;

  reg [0:0] mem [0:63];
  assign dout = mem[a];
  always @(posedge clk) if(we) mem[a] <= d;
endmodule
```

LUTRAM = 1

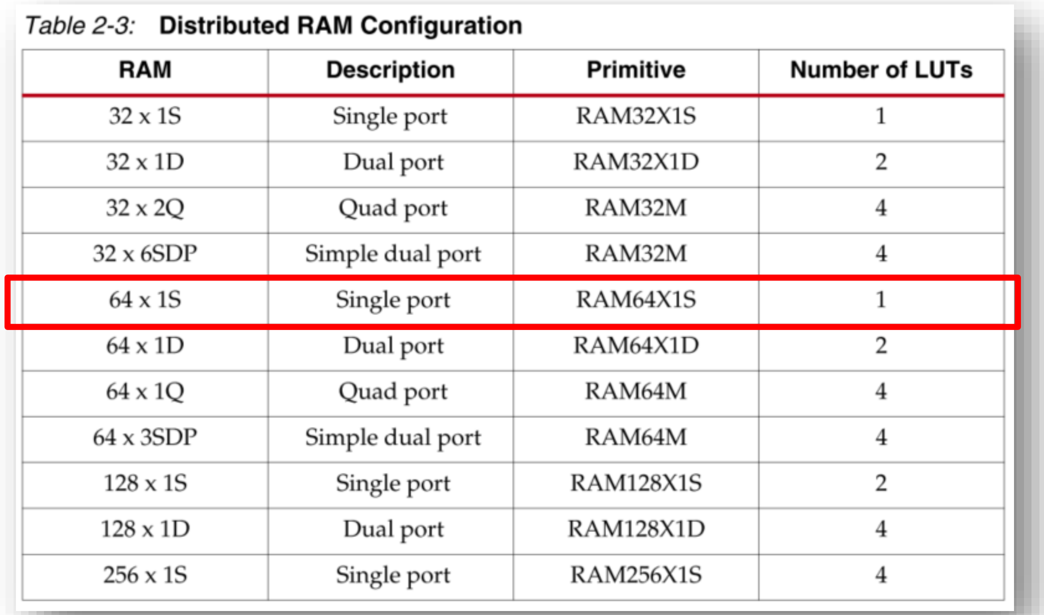
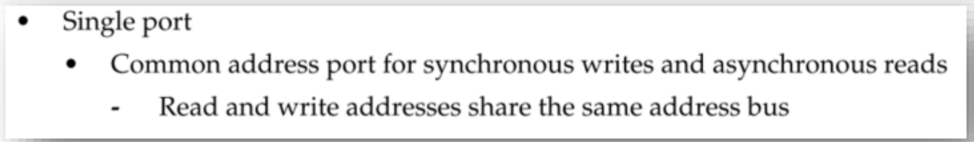


Table 2-3: **Distributed RAM Configuration**

RAM	Description	Primitive	Number of LUTs
32 x 1S	Single port	RAM32X1S	1
32 x 1D	Dual port	RAM32X1D	2
32 x 2Q	Quad port	RAM32M	4
32 x 6SDP	Simple dual port	RAM32M	4
64 x 1S	Single port	RAM64X1S	1
64 x 1D	Dual port	RAM64X1D	2
64 x 1Q	Quad port	RAM64M	4
64 x 3SDP	Simple dual port	RAM64M	4
128 x 1S	Single port	RAM128X1S	2
128 x 1D	Dual port	RAM128X1D	4
256 x 1S	Single port	RAM256X1S	4



- Single port
  - Common address port for synchronous writes and asynchronous reads
    - Read and write addresses share the same address bus

Distributed RAM

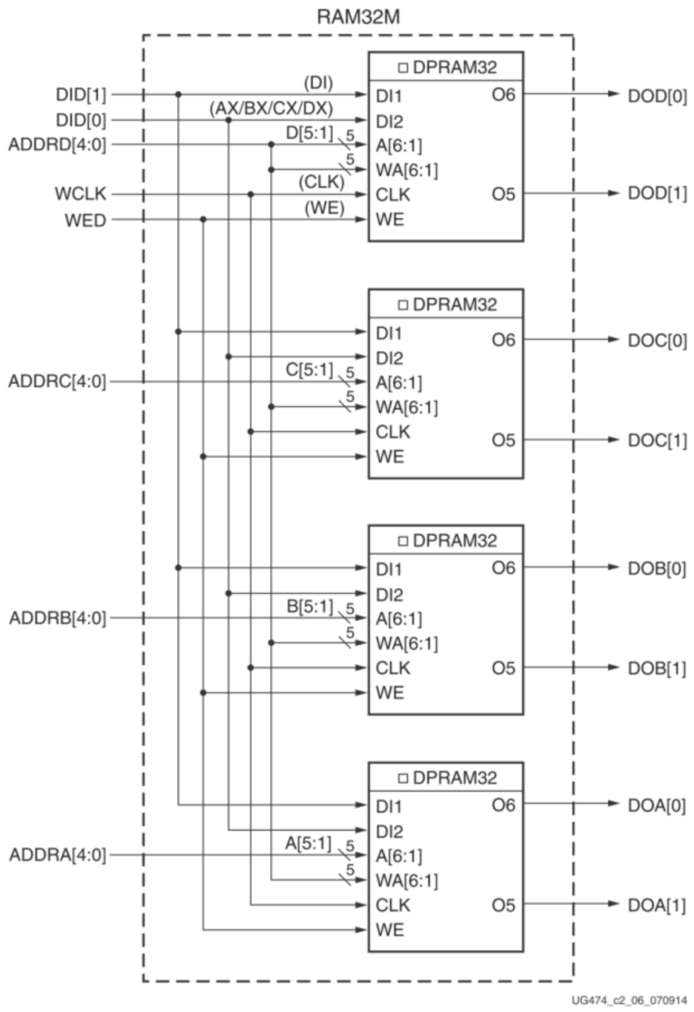


Figure 2-6: **32 X 2 Quad Port Distributed RAM (RAM32M)**

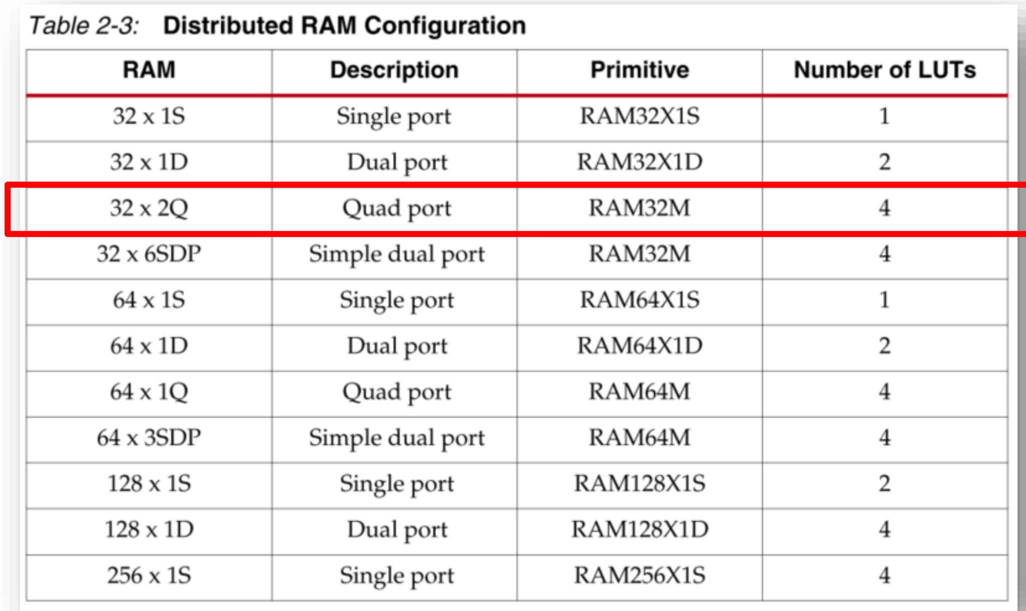


Table 2-3: **Distributed RAM Configuration**

RAM	Description	Primitive	Number of LUTs
32 x 1S	Single port	RAM32X1S	1
32 x 1D	Dual port	RAM32X1D	2
32 x 2Q	Quad port	RAM32M	4
32 x 6SDP	Simple dual port	RAM32M	4
64 x 1S	Single port	RAM64X1S	1
64 x 1D	Dual port	RAM64X1D	2
64 x 1Q	Quad port	RAM64M	4
64 x 3SDP	Simple dual port	RAM64M	4
128 x 1S	Single port	RAM128X1S	2
128 x 1D	Dual port	RAM128X1D	4
256 x 1S	Single port	RAM256X1S	4

- Quad port
  - One port for synchronous writes and asynchronous reads
  - Three ports for asynchronous reads

Distributed RAM

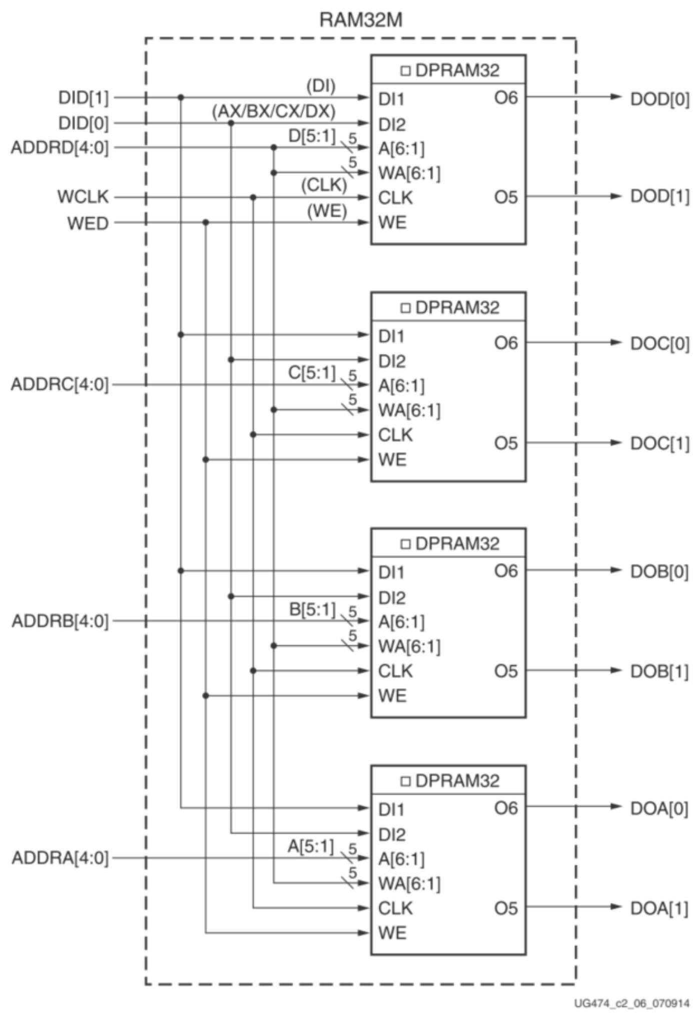
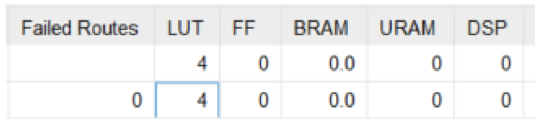


Figure 2-6: **32 X 2 Quad Port Distributed RAM (RAM32M)**

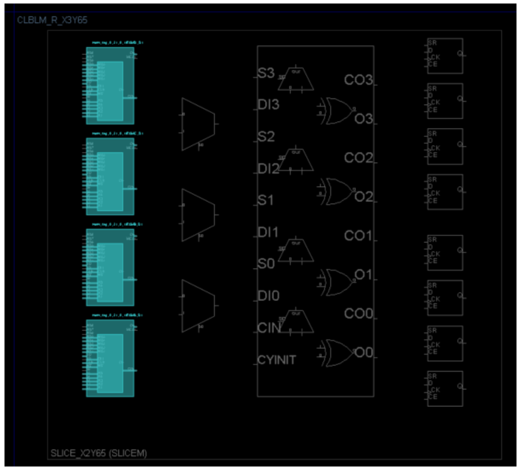
```
module m_RAM32M_Q (clk, a1, a2, a3, a4, d, we, dout1, dout2, dout3, dout4);
  input wire clk;
  input wire [4:0] a1, a2, a3, a4;
  input wire [1:0] d;
  input wire we;
  output wire [1:0] dout1, dout2, dout3, dout4;

  reg [1:0] mem [0:31];
  assign dout1 = mem[a1];
  assign dout2 = mem[a2];
  assign dout3 = mem[a3];
  assign dout4 = mem[a4];
  always @(posedge clk) if(we) mem[a1] <= d;
endmodule
```



Failed Routes	LUT	FF	BRAM	URAM	DSP
	4	0	0.0	0	0
0	4	0	0.0	0	0

LUTRAM = 4


CLBLM_R_X3Y65
S3
DI3
S2
DI2
S1
DI1
S0
DI0
CIN
CYINIT
CO3
O3
CO2
O2
CO1
O1
CO0
O0
SLICE_X2Y65 (SLICEM)