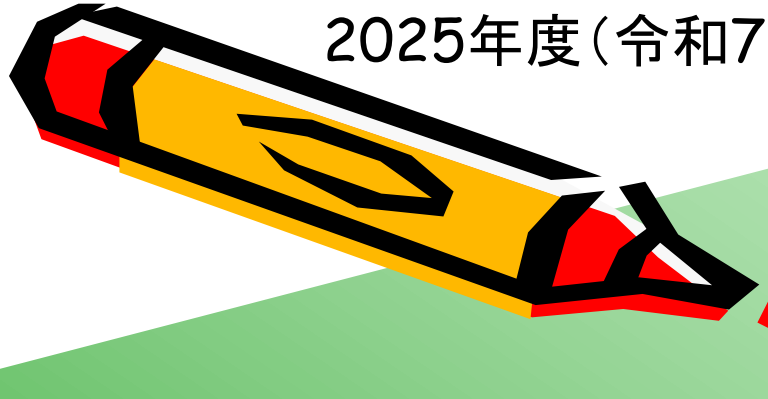


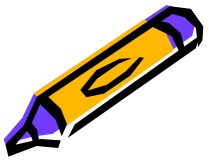
2025年度（令和7年）版

Ver. 2025-10-08a

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

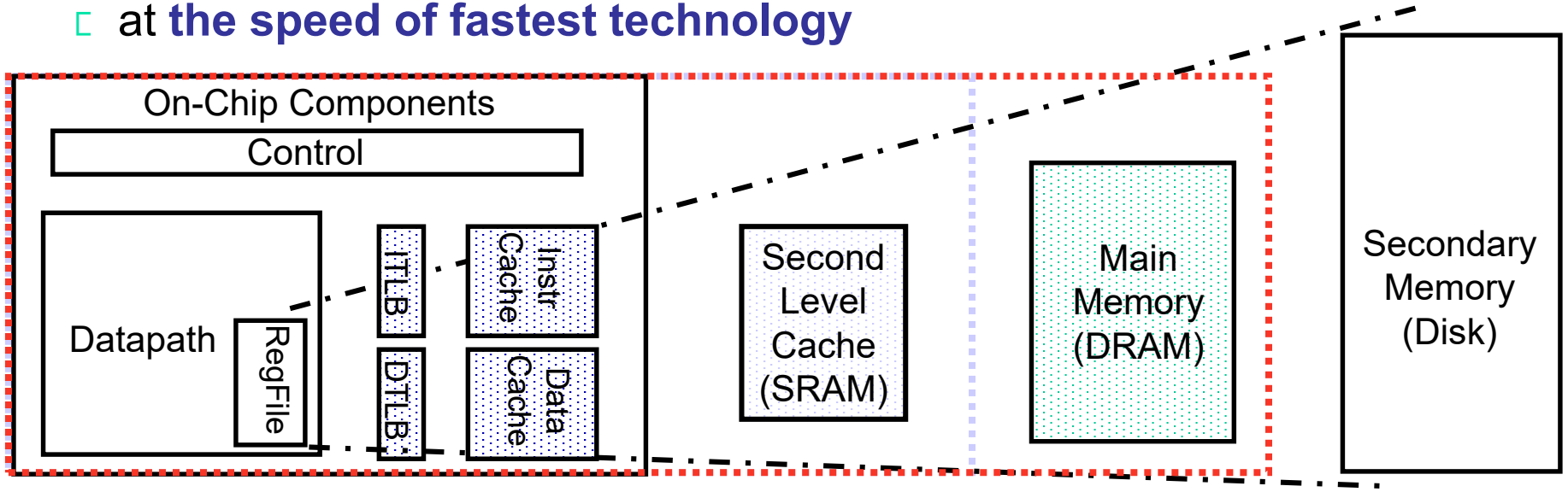


www.arch.cs.titech.ac.jp/lecture/CA/
Tue 13:30-15:10, 15:25-17:05
Fri 13:30-15:10

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

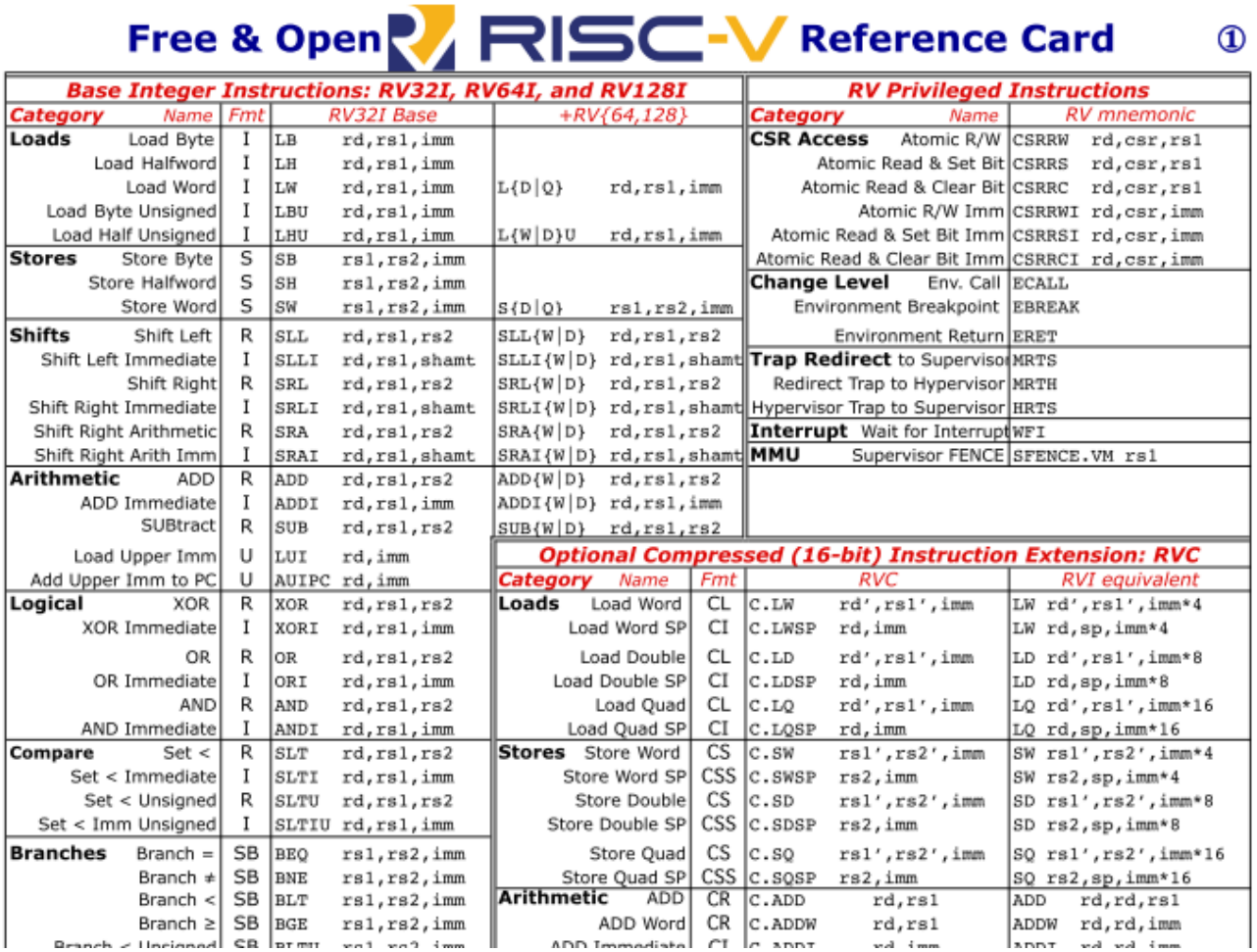
- By taking advantage of **the principle of locality**（局所性）
  - Present **much memory** in **the cheapest technology**
  - at **the speed of fastest technology**



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

RISC-V Reference Card



Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Atomic R/W Imm	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level	Env. Call	ECALL
	Environment Breakpoint	EBREAK
	Environment Return	ERET
Trap Redirect	to Supervisor	MRTS
	Redirect Trap to Hypervisor	MRTH
	Hypervisor Trap to Supervisor	HRTS
Interrupt	Wait for Interrupt	WFI
MMU	Supervisor FENCE	SFENCE.VM rs1

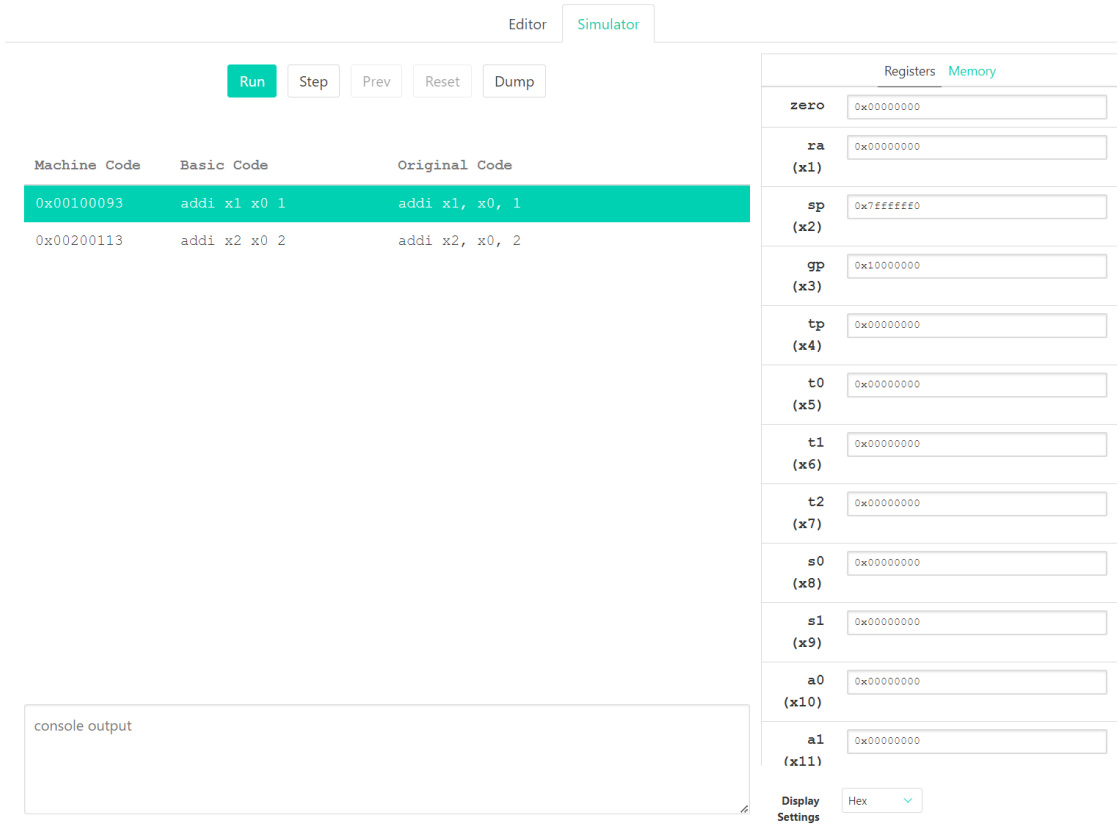
Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Load Double	CL	C.LD rd',rs1',imm	LD rd',rs1',imm*8
	Load Double SP	CI	C.LDSP rd,imm	LD rd,sp,imm*8
	Load Quad	CL	C.LQ rd',rs1',imm	LQ rd',rs1',imm*16
	Load Quad SP	CI	C.LQSP rd,imm	LQ rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Store Double	CS	C.SD rs1',rs2',imm	SD rs1',rs2',imm*8
	Store Double SP	CSS	C.SDSP rs2,imm	SD rs2,sp,imm*8
	Store Quad	CS	C.SQ rs1',rs2',imm	SQ rs1',rs2',imm*16
	Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Word	CR	C.ADDW rd,rs1	ADDW rd,rd,imm
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm

https://www.arch.cs.titech.ac.jp/lecture/CA/RISCVGreenCard.pdf

RISC-V instruction set simulator

- **venus** is a RISC-V instruction set simulator built for education.
  - https://venus.kvakil.me/
  - https://github.com/kvakil/venus



sample sequence 1

```
addi x1, x0, 1
addi x2, x0, 10
add  x3, x1, x2
```

sample sequence 2

```
   addi x1, x0, 1
   addi x2, x0, 10
L: addi x1, x1, 1
   bne x1, x2, L
```

sample sequence 3

```
lui  x1, 0x123
ori  x1, x1, 0x456
sw   x1, 32(x0)
lw   x2, 32(x0)
lb   x3, 32(x0)
lb   x4, 33(x0)
lb   x5, 34(x0)
```

little-endian, big-endian

In a little-endian configuration, multibyte stores write the least-significant register byte at the lowest memory byte address, followed by the other register bytes in ascending order of their significance. Loads similarly transfer the contents of the lesser memory byte addresses to the less-significant register bytes.

In a big-endian configuration, multibyte stores write the most-significant register byte at the lowest memory byte address, followed by the other register bytes in descending order of their significance. Loads similarly transfer the contents of the greater memory byte addresses to the less-significant register bytes.

パレートの法則

- Vilfredo Federico Damaso Pareto
  - イタリアの経済学者(1848 – 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

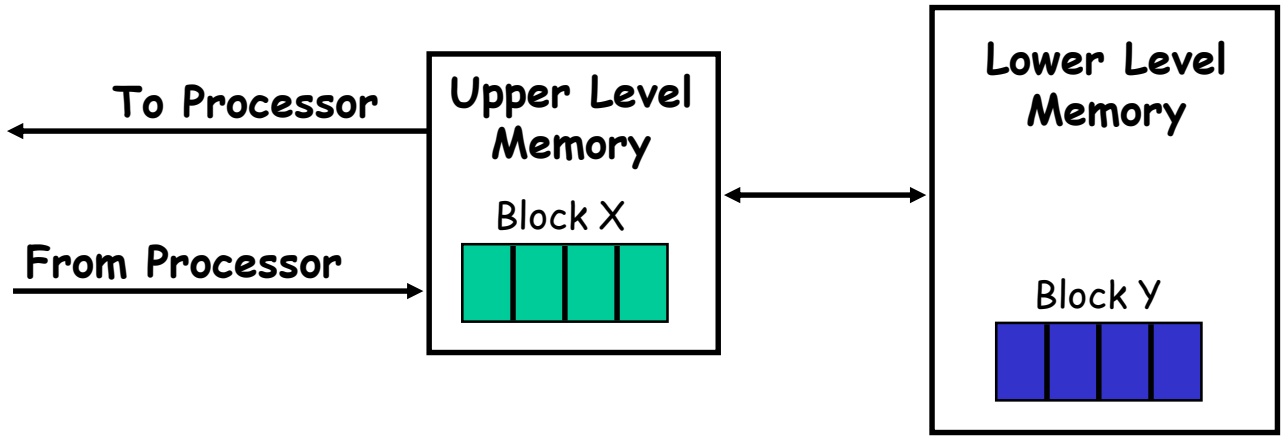
The Memory Hierarchy: Why Does it Work?

- **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):

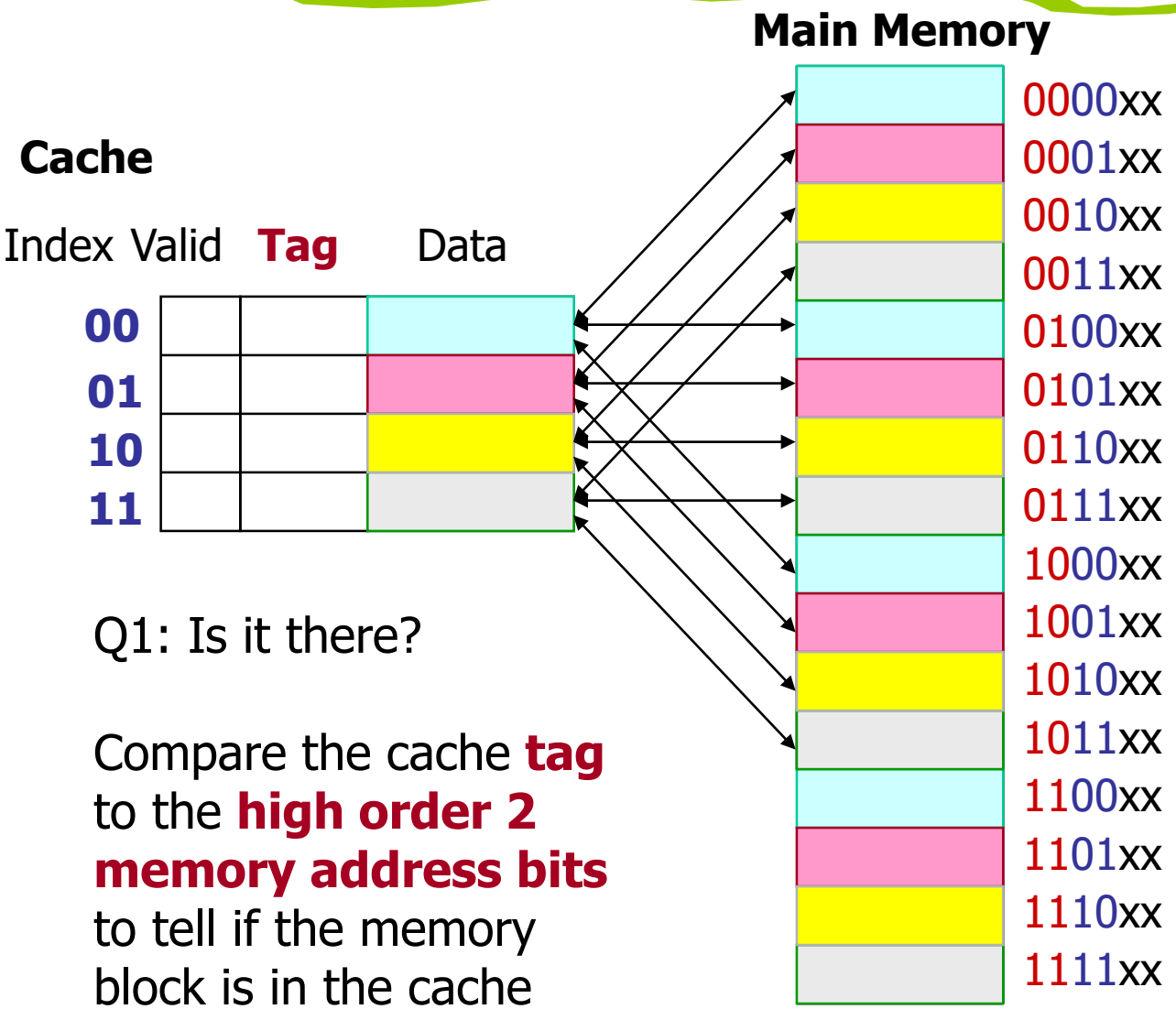
  ⇒ Move blocks consisting of **contiguous words** to the upper levels



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 (32bit word)

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

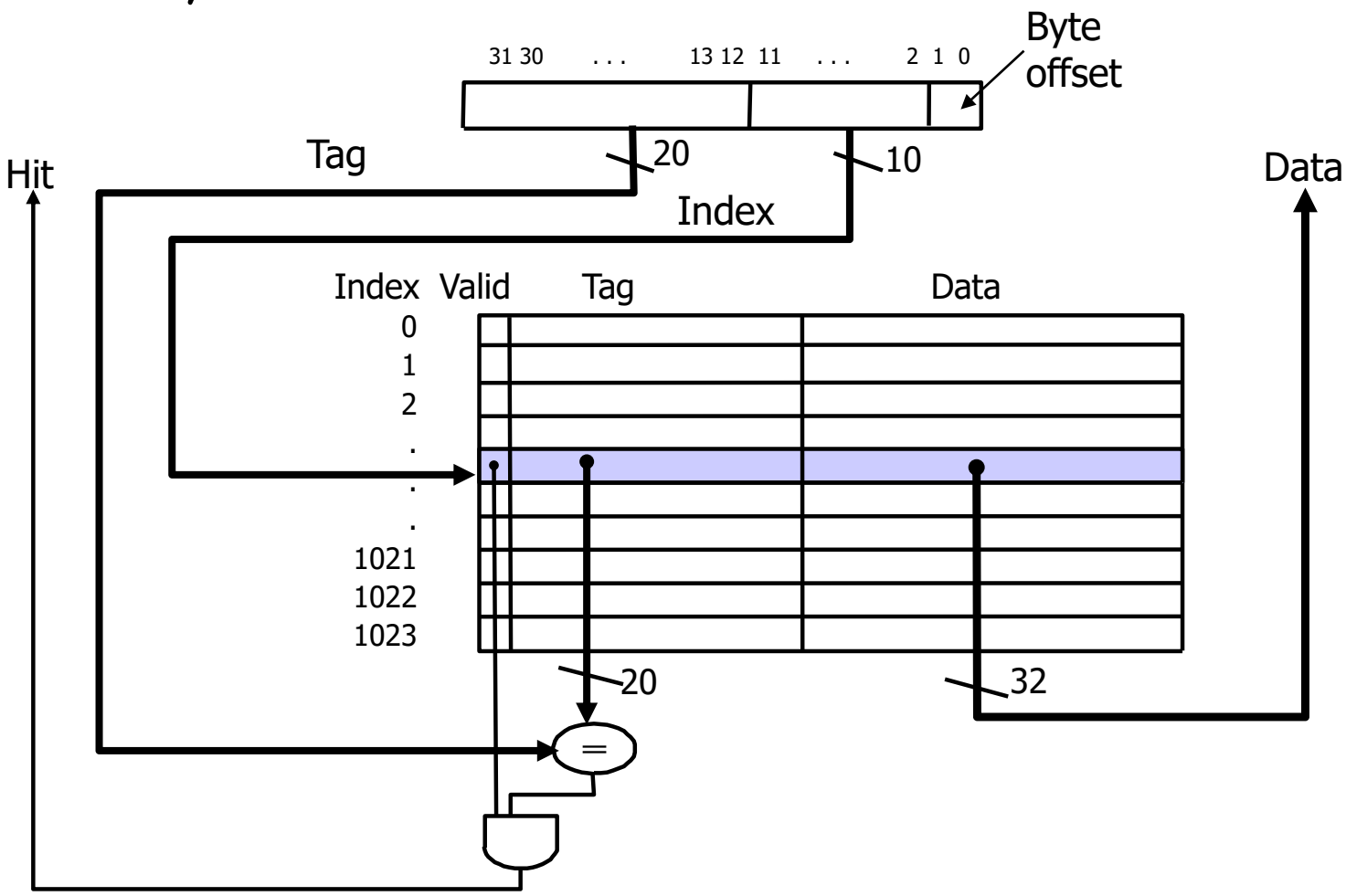
Q2: How do we find it?

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

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

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

Tag

0 miss

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~~ 4

~~00~~	Mem(0)
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

11 ~~00~~ ~~Mem(3)~~ 15

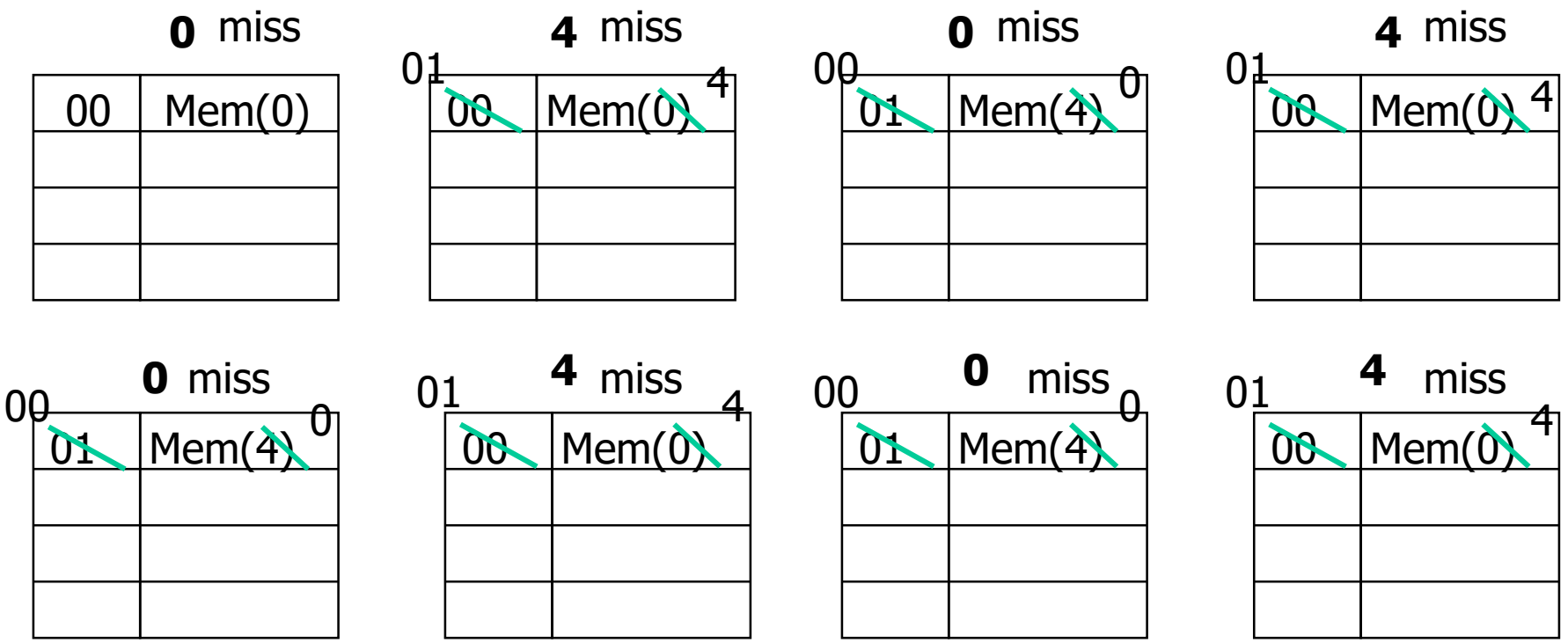
01	Mem(4)
00	Mem(1)
00	Mem(2)
~~00~~	~~Mem(3)~~

- 8 requests, 6 misses

Another Reference String Mapping

- Consider the main memory word reference string

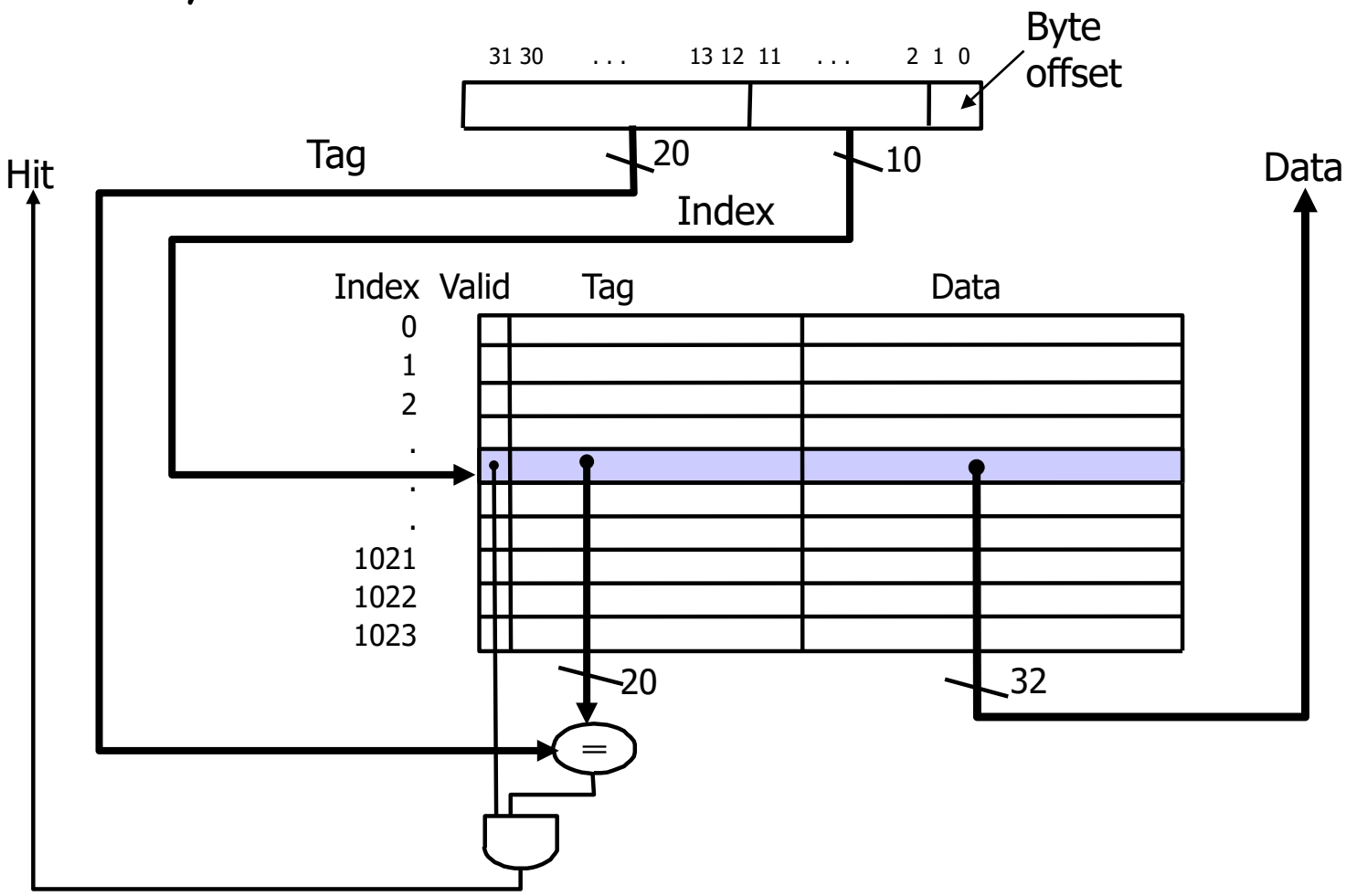
0 4 0 4 0 4 0 4



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

Direct Mapped Cache Example

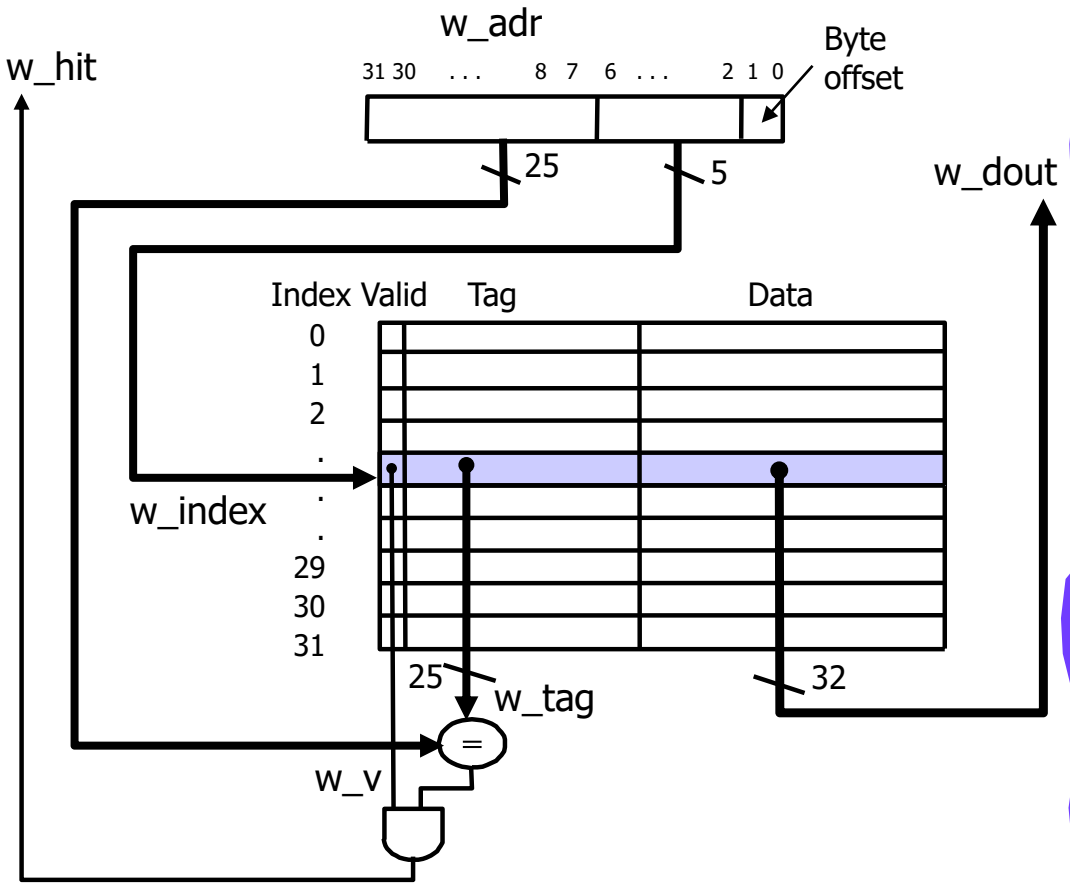
- One word/block, cache size = 1K words



What kind of locality are we taking advantage of?

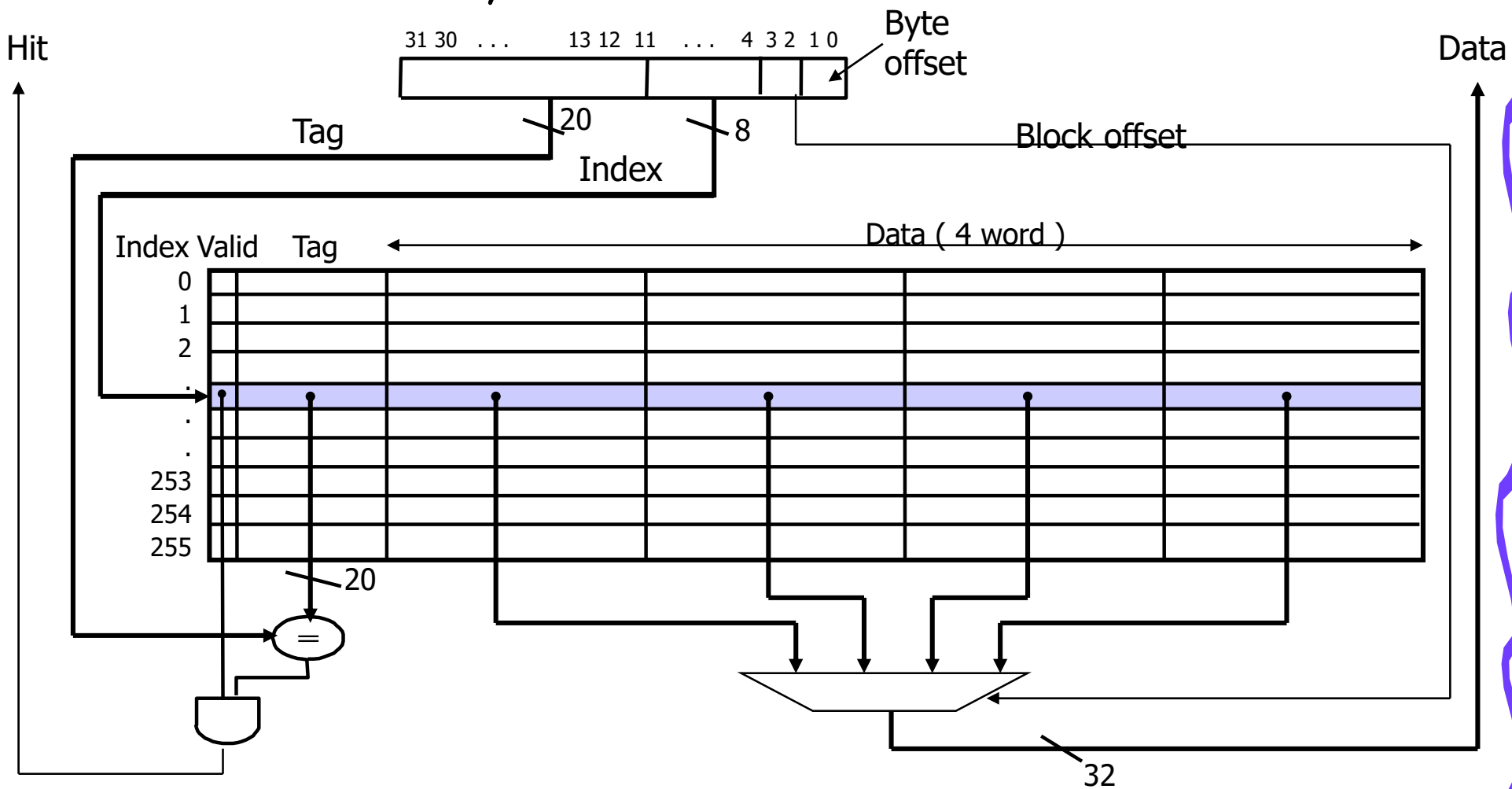
Direct Mapped Cache Example

```
module m_cache_direct_mapped_32 (
  input  wire        w_clk,
  input  wire        w_we,
  input  wire [31:0] w_adr,
  input  wire [4:0]  w_wadr,
  input  wire [57:0] w_wd,
  output wire        w_hit,
  output wire [31:0] w_dout
);

  reg [57:0] mem [0:31];
  integer i; initial for (i=0; i<32; i=i+1) mem[i] = 0;

  wire [4:0]  w_index = w_adr[6:2];
  wire        w_v;
  wire [24:0] w_tag;
  assign {w_v, w_tag, w_dout} = mem[w_index];
  assign w_hit = w_v & (w_adr[31:7]==w_tag);

  always @(posedge w_clk) if (w_we) mem[w_wadr] <= w_wd;
endmodule
```



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

~~00~~ 01	~~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

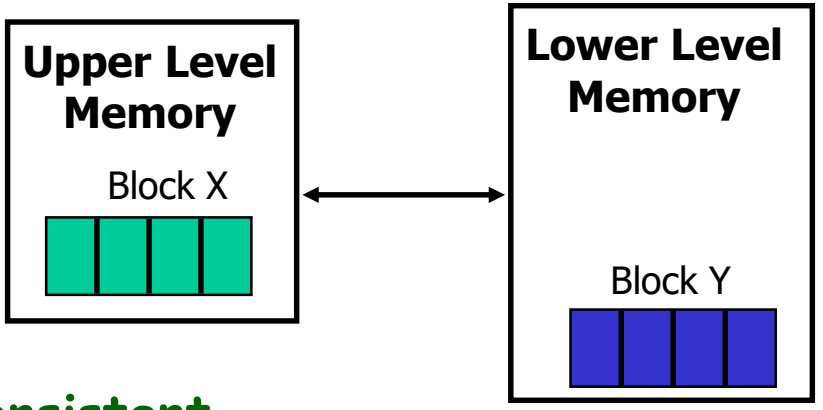
01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

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

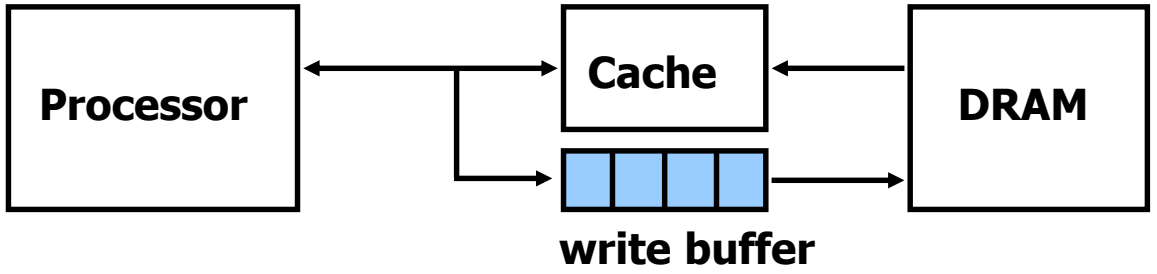
- 8 requests, 4 misses

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

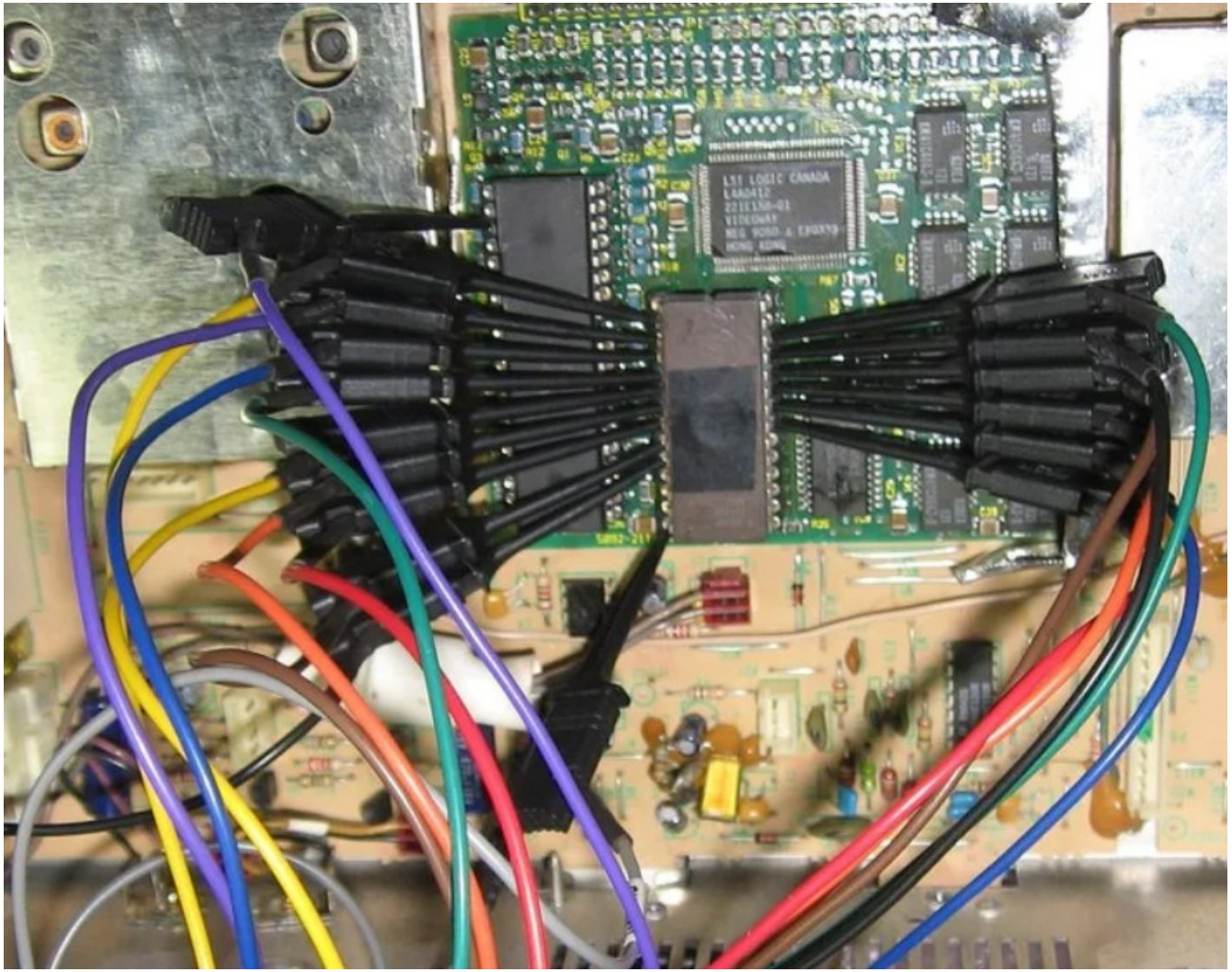


- **Write buffer** between the cache and main memory
  - Processor: writes data into the cache and the write buffer
  - **Memory controller**: writes contents of the write buffer to memory
- The write buffer is just a **FIFO**
  - Typical number of entries: 4
  - Works fine if store frequency is low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

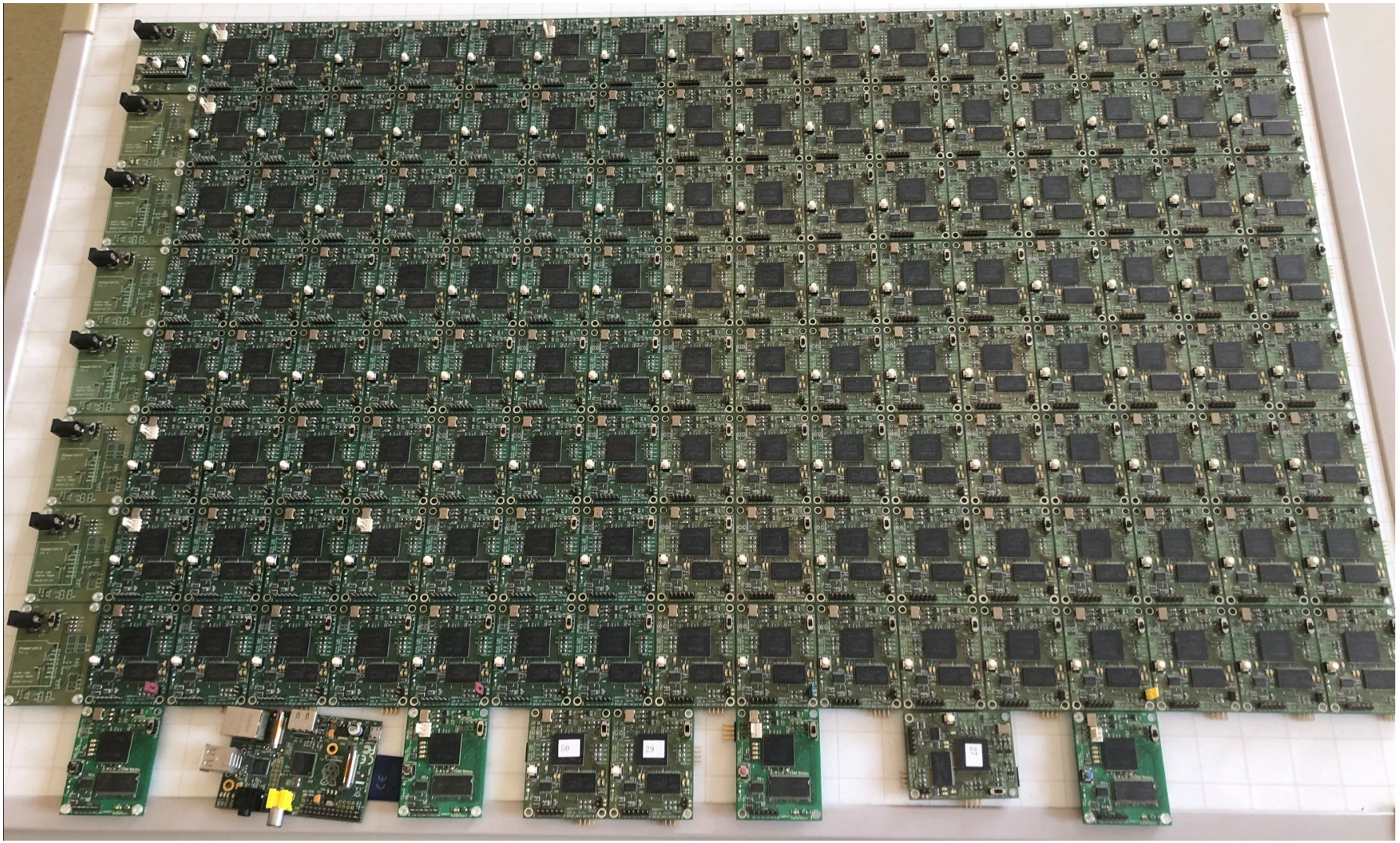
Handling Cache **Misses**

- **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) single-word block: write the word into the cache updating both the tag and data, no need to check for cache hit, no need to stall
    - (b) multi-word block: **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 is not full

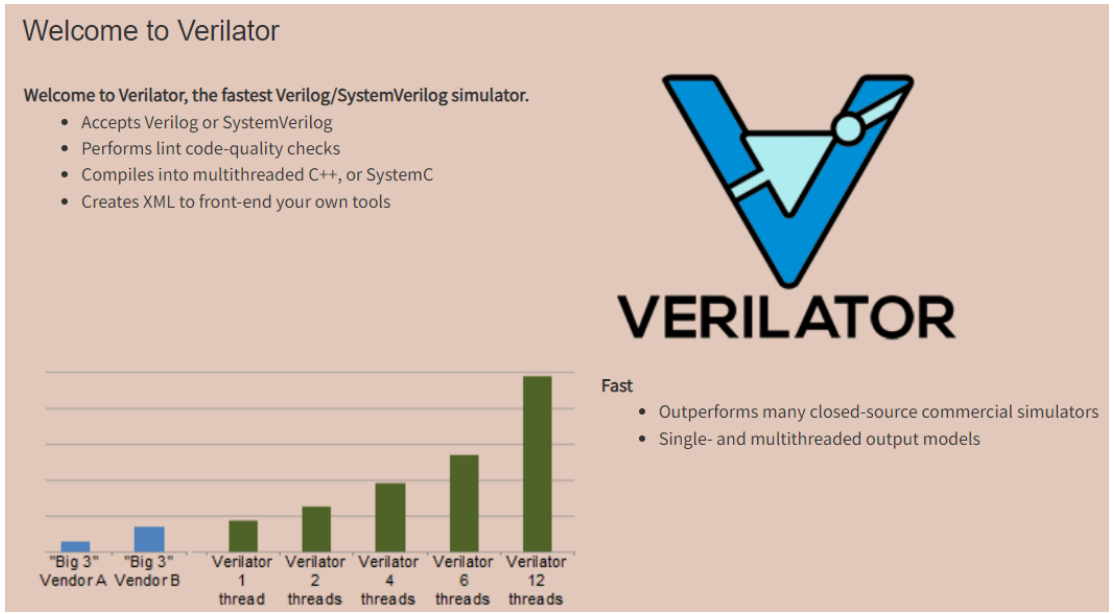
Hardware debug



ScalableCore system



Verilator, the fastest Verilog/SystemVerilog simulator



https://www.veripool.org/verilator/

code001.v

```
module main ();
    initial begin
        $write("hello, world¥n");
        $finish();
    end
endmodule
```

```
$ verilator --binary code001.v
$ obj_dir/Vcode001

hello, world
- code001.v:8: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.026 2024-06-15
- Verilator: $finish at 1ps; walltime 0.001 s; speed 0.000 s/s
- Verilator: cpu 0.000 s on 1 threads; alloced 121 MB
```