

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

1

last time

certificates and certificate authorities

- chains of certificates

- trust anchors

- the reality of website “verification”

cryptographic hash functions

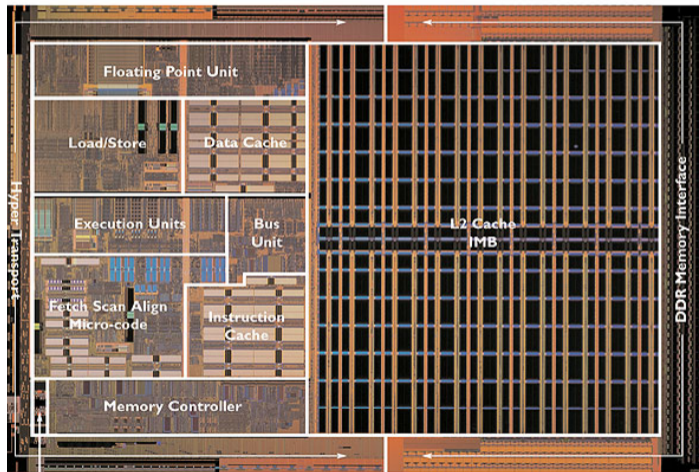
Diffie-Hellman-style key agreement

TLS handshake authenticate + get symmetric keys for connection

denial of service how game is biased for attacker

(briefly) firewalls

2004 CPU



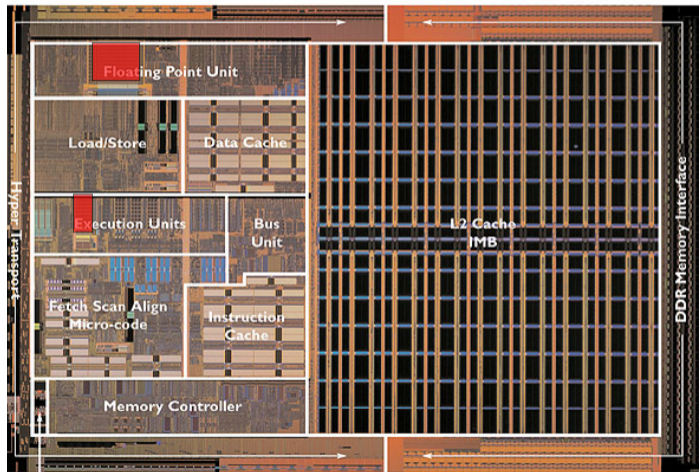
Clock Generator



Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

▲ Registers



Clock Generator



Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

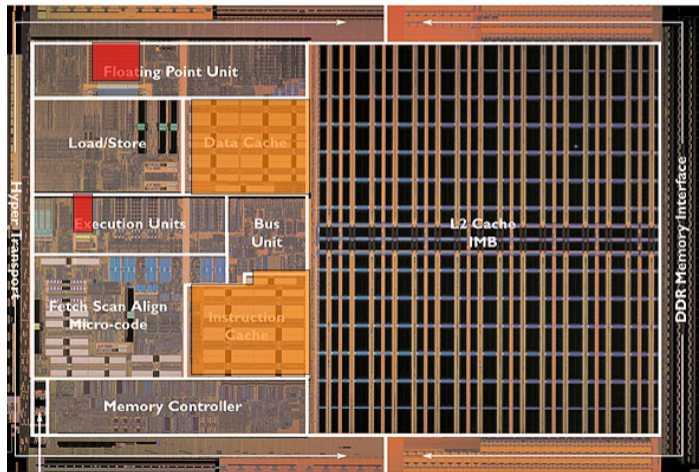


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

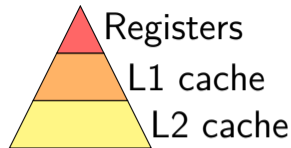
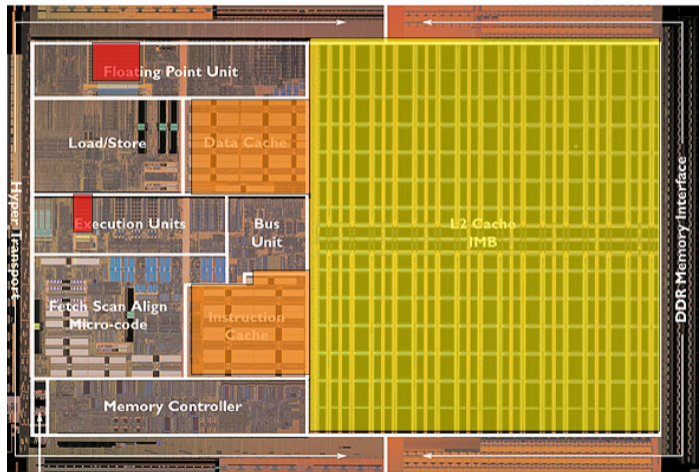


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

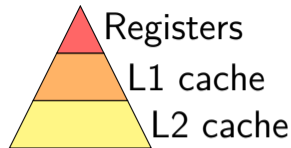
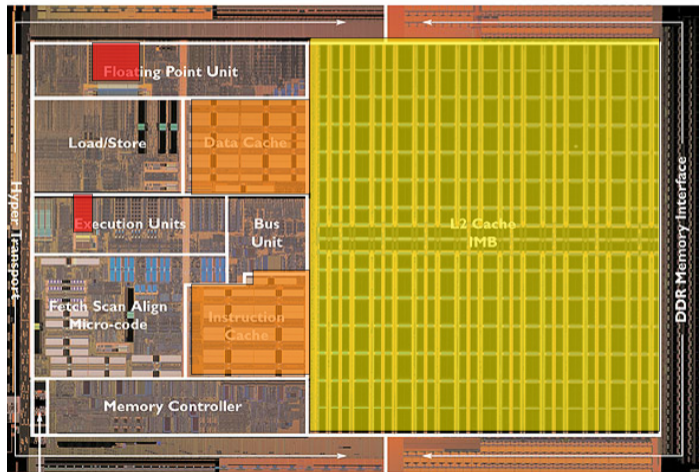
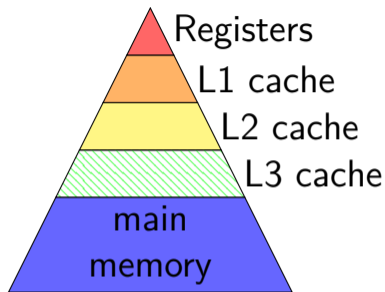
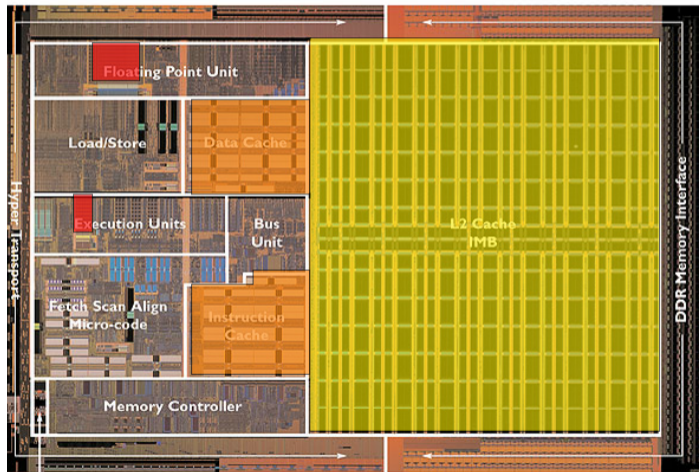
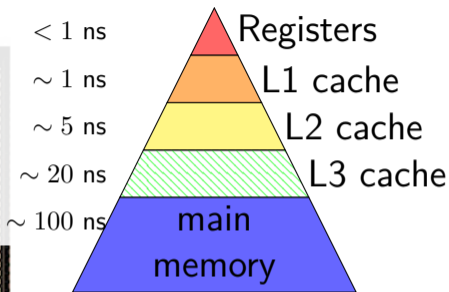
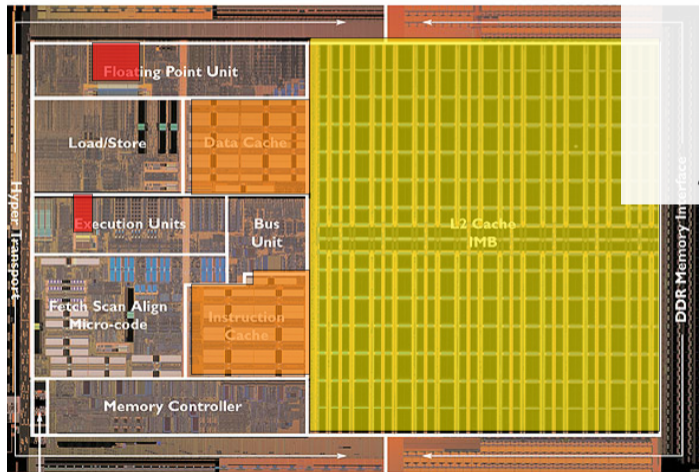


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



2004 CPU

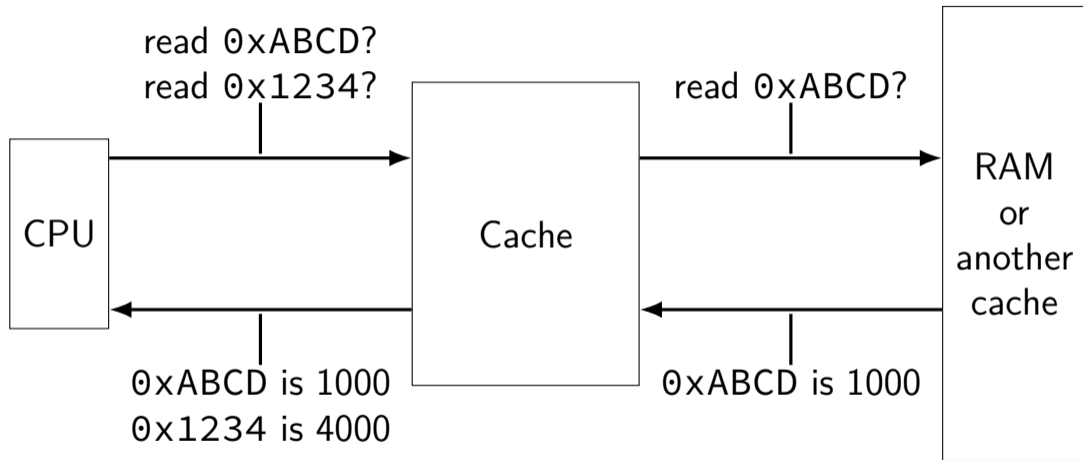


Clock Generator



Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

the place of cache



memory hierarchy goals

performance of the fastest (smallest) memory

hide 100x latency difference? 99+% hit (= value found in cache) rate

capacity of the largest (slowest) memory

memory hierarchy assumptions

temporal locality

“if a value is accessed now, it will be accessed again soon”

 caches should keep **recently accessed values**

spatial locality

“if a value is accessed now, adjacent values will be accessed soon”

 caches should **store adjacent values at the same time**

natural properties of programs — think about loops

locality examples

```
double computeMean(int length, double *values) {  
    double total = 0.0;  
    for (int i = 0; i < length; ++i) {  
        total += values[i];  
    }  
    return total / length;  
}
```

temporal locality: machine code of the loop

spatial locality: machine code of most consecutive instructions

temporal locality: total, i, length accessed repeatedly

spatial locality: values[i+1] accessed after values[i]

building a (direct-mapped) cache

Cache

value
00 00
00 00
00 00
00 00

cache block: 2 bytes

Memory

addresses	bytes
00000-00001	00 11
00010-00011	22 33
00100-00101	55 55
00110-00111	66 77
01000-01001	88 99
01010-01011	AA BB
01100-01101	CC DD
01110-01111	EE FF
10000-10001	F0 F1
...	...

building a (direct-mapped) cache

read byte at 01011?

Cache

value
00 00
00 00
00 00
00 00

cache block: 2 bytes

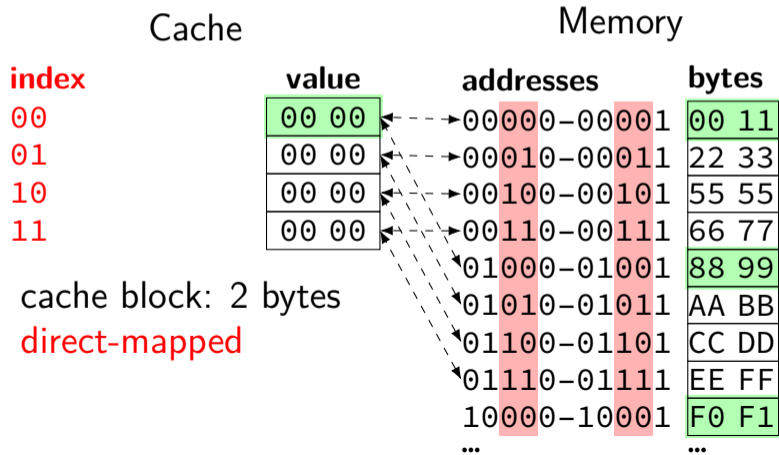
Memory

addresses	bytes
00000-00001	00 11
00010-00011	22 33
00100-00101	55 55
00110-00111	66 77
01000-01001	88 99
01010-01011	AA BB
01100-01101	CC DD
01110-01111	EE FF
10000-10001	F0 F1
...	...

building a (direct-mapped) cache

read byte at 01011?

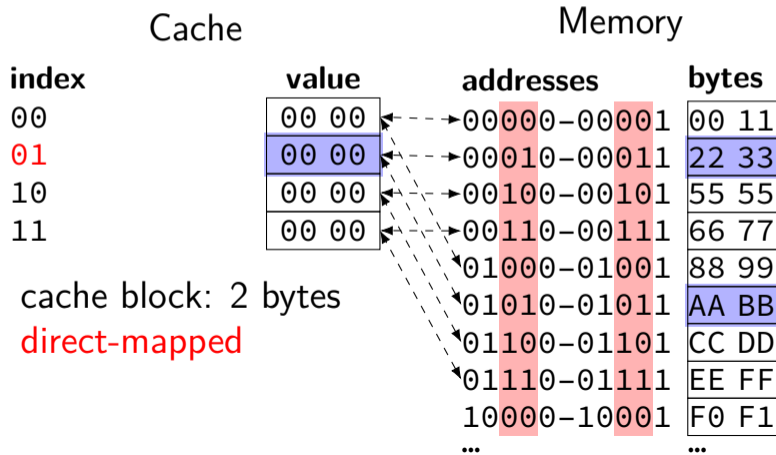
exactly **one place** for each address
spread out what can go in a block



building a (direct-mapped) cache

read byte at 01011?

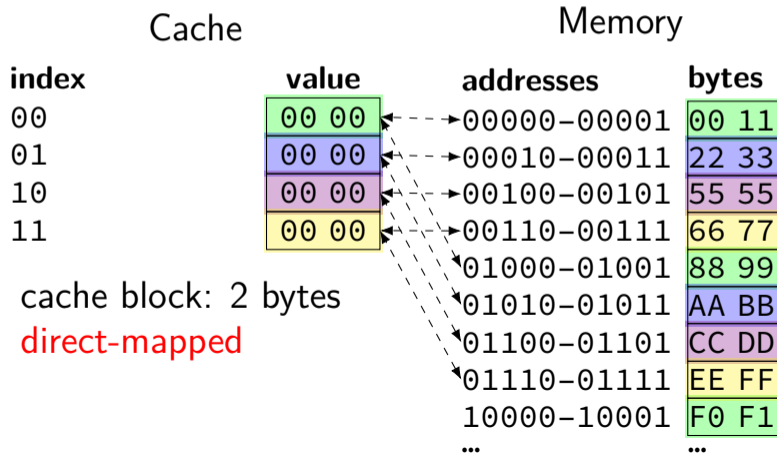
exactly **one place** for each address
spread out what can go in a block



building a (direct-mapped) cache

read byte at 01011?

exactly **one place** for each address
spread out what can go in a block



building a (direct-mapped) cache

read byte at 01011?

Cache		Memory	
index	valid	value	addresses bytes
00	0	00 00	00000-00001 1
01	0	00 00	00010-00011 22 33
10	0	00 00	00100-00101 55 55
11	0	00 00	00110-00111 66 77
			01000-01001 88 99
			01010-01011 AA BB
			01100-01101 CC DD
			01110-01111 EE FF
			10000-10001 F0 F1
			...

cache block: 2 bytes
direct-mapped

is this even a value?

need extra bit to know

building a (direct-mapped) cache

read byte at 01011?

invalid, fetch

Cache

index	valid	value
00	0	00 00
01	1	AA BB
10	0	00 00
11	0	00 00

cache block: 2 bytes

direct-mapped

Memory

addresses	bytes
00000-00001	00 11
00010-00011	22 33
00100-00101	55 55
00110-00111	66 77
01000-01001	88 99
01010-01011	AA BB
01100-01101	CC DD
01110-01111	EE FF
10000-10001	F0 F1

...

...

building a (direct-mapped) cache

read byte at 01011?

invalid, fetch

Cache				Memory	
index	valid	tag	value	addresses	bytes
00	0	00	00 00	00000-00001	00 11
01	1	01	AA BB	00010-00011	22 33
10	0	00	00 00	00100-00101	55 55
11	0			00110-00111	66 77
				01000-01001	88 99
				01010-01011	AA BB
				01100-01101	CC DD
				01110-01111	EE FF
				10000-10001	F0 F1
			

value from 01010 or 00010?

need tag to know

cache block: 2 bytes
direct-mapped

building a (direct-mapped) cache

read byte at 01011?

invalid, fetch

Cache

index	valid	tag	value
00	0	00	00 00
01	1	01	AA BB
10	0	00	00 00
11	0	00	00 00

cache block: 2 bytes

direct-mapped

Memory

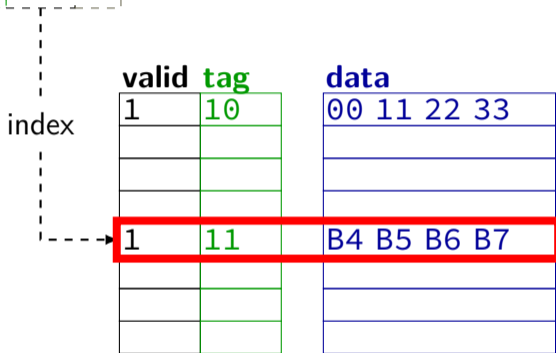
addresses	bytes
00000-00001	00 11
00010-00011	22 33
00100-00101	55 55
00110-00111	66 77
01000-01001	88 99
01010-01011	AA BB
01100-01101	CC DD
01110-01111	EE FF
10000-10001	F0 F1

...

...

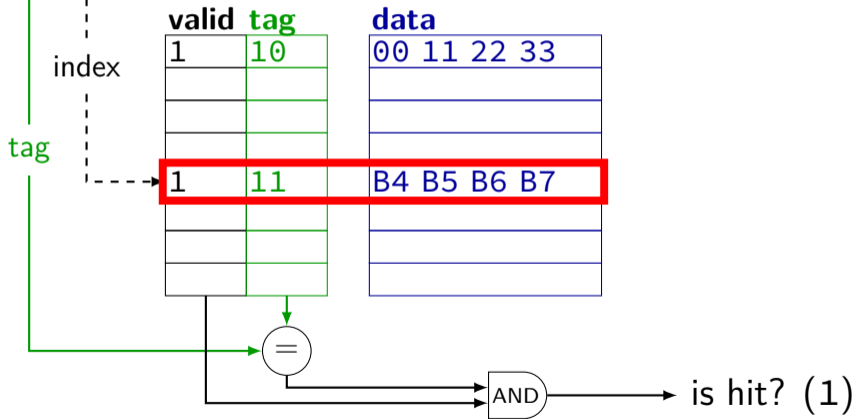
cache operation (read)

0b1110010

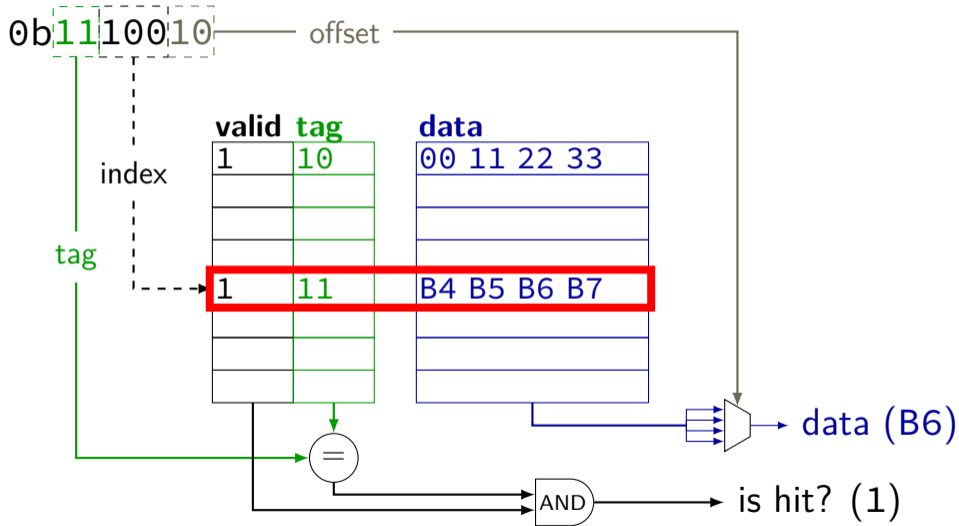


cache operation (read)

0b1110010



cache operation (read)



terminology

row = set

preview: change how much is in a row

Tag-Index-Offset (TIO)

address 001111 (stores value 0xFF)

cache tag index offset

2 byte blocks, 4 sets

2 byte blocks, 8 sets

4 byte blocks, 2 sets

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	----
11	1	001	EE FF

4 byte blocks, 2 sets

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	----
011	0	--	----
100	0	--	----
101	1	00	AA BB
110	0	--	----
111	1	00	EE FF

Tag-Index-Offset (TIO)

address 001111 (stores value 0xFF)

cache tag index offset

2 byte blocks, 4 sets

1

2 byte blocks, 8 sets

1

4 byte blocks, 2 sets

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	----
11	1	001	EE FF

4 byte blocks, 2 sets

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	000	00 11
001	1	001	F1 F2
010	0	--	----
011	0	--	----
100	0	--	----
101	1	00	AA BB
110	0	--	----
111	1	00	EE FF

2 = 2¹ bytes in block
1 bit to say which byte

Tag-Index-Offset (TIO)

address 001111 (stores value 0xFF)

cache tag index offset

2 byte blocks, 4 sets 1

2 byte blocks, 8 sets 1

4 byte blocks, 2 sets 11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0		
11	1		

4 byte

4 = 2² bytes in block
2 bits to say which byte

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
	0	--	-- --
	0	--	-- --
	0	--	-- --
	1	00	AA BB
	0	--	-- --
	1	00	EE FF

Tag-Index-Offset (TIO)

address 001111 (stores value 0xFF)

cache tag index offset

2 byte blocks, 4 sets

11 1

2 byte blocks, 8 sets

1

4 byte blocks, 2 sets

1 11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	----
11	1	001	EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
			F1 F2

100	0	--	----
101	1	00	AA BB
110	0	--	----
111	1	00	EE FF

$2^2 = 4$ sets

2 bits to index set

4 byte blocks, 2 sets

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE FF

Tag-Index-Offset (TIO)

address 001111 (stores value 0xFF)

cache	tag	index	offset
2 byte blocks, 4 sets	11	1	
2 byte blocks, 8 sets	111	1	
4 byte blocks, 2 sets	1		11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	----
11	1		

$2^3 = 8$ sets
3 bits to index set

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	----
011	0	--	----
100	0	--	----
101	1	00	AA BB
110	0	--	----
111	1	00	EE FF

Tag-Index-Offset (TIO)

address 001111 (stores value 0xFF)

cache	tag	index	offset
2 byte blocks, 4 sets		11	1
2 byte blocks, 8 sets		111	1
4 byte blocks, 2 sets		1	11

2 byte blocks, 4 sets

index	valid	tag	value
00	1	000	00 11
01	1	001	AA BB
10	0	--	----
11	1	001	EE FF

4 byte blocks, 2 sets

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	----
011	0	--	----
100	0	--	----
101	0	--	----
110	0	--	----
111	1	00	EE FF

$2^1 = 2$ sets
1 bit to index set

Tag-Index-Offset (TIO)

address 001111 (stores value 0xFF)

cache	tag	index	offset
2 byte blocks, 4 sets	001	11	1
2 byte blocks, 8 sets	00	111	1
4 byte blocks, 2 sets	001	1	11

tag — whatever is left over

00	1	000	00 11
01	1	001	AA BB
10	0	--	----
11	1	001	EE FF

4 byte blocks, 2 sets

index	valid	tag	value
0	1	000	00 11 22 33
1	1	001	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
000	1	00	00 11
001	1	01	F1 F2
010	0	--	----
011	0	--	----
100	0	--	----
101	1	00	AA BB
110	0	--	----
111	1	00	EE FF

TIO: exercise

64-byte blocks, 128 set cache

stores $64 \times 128 = 8192$ bytes (of data)

if addresses 32-bits, then how many tag/index/offset bits?

which bytes are stored in the same block as byte from 0x1037?

- A. byte from 0x1011
- B. byte from 0x1021
- C. byte from 0x1035
- D. byte from 0x1041

Tag-Index-Offset formulas (direct-mapped)

(formulas derivable from prior slides)

$S = 2^s$ number of sets

s (set) index bits

$B = 2^b$ block size

b (block) offset bits

m memory addresses bits

$t = m - (s + b)$ tag bits

$C = B \times S$ cache size (if direct-mapped)

Tag-Index-Offset formulas (direct-mapped)

(formulas derivable from prior slides)

$S = 2^s$ number of sets

s (set) index bits

$B = 2^b$ block size

b (block) offset bits

m memory addresses bits

$t = m - (s + b)$ tag bits

$C = B \times S$ **cache size** (if direct-mapped)

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

index	valid	tag	value
00	0		
01	0		
10	0		
11	0		

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

index	valid	tag	value
00	0		
01	0		
10	0		
11	0		

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00	0		
01	0		
10	0		
11	0		

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	miss
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00	1	000000	mem[0x00] mem[0x01]
01	0		
10	0		
11	0		

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00	1	000000	mem[0x00] mem[0x01]
01	0		
10	0		
11	0		

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00	1	000000	mem[0x00] mem[0x01]
01	1	011000	mem[0x62] mem[0x63]
10	0		
11	0		

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00	1	01100	mem[0x60] mem[0x61]
01	1	01100	mem[0x62] mem[0x63]
10	0		
11	0		

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	
01100100 (64)	

tag index offset

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00	1	01100	mem[0x60] mem[0x61]
01	1	01100	mem[0x62] mem[0x63]
10	0		
11	0		

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	miss
01100100 (64)	

tag index offset

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	0		
11	0		

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	miss
01100100 (64)	miss

tag index offset

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00	1	000000	mem[0x00] mem[0x01]
01	1	011000	mem[0x62] mem[0x63]
10	1	011000	mem[0x64] mem[0x65]
11	0		

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	miss
01100100 (64)	miss

tag index offset

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] mem[0x65]
11	0		

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

example access pattern (1)

2 byte blocks, 4 sets

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	miss
01100100 (64)	miss

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] mem[0x65]
11	0		

miss caused by conflict

tag index offset

$B = 2 = 2^b$ byte block size

$b = 1$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

$m = 8$ bit addresses

$t = m - (s + b) = 5$ tag bits

exercise

4 byte blocks, 4 sets

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

index	valid	tag	value
00			
01			
10			
11			

exercise

4 byte blocks, 4 sets

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

index	valid	tag	value
00			
01			
10			
11			

how is the 8-bit address 61 (01100001) split up into tag/index/offset?

b block offset bits;

$B = 2^b$ byte block size;

s set index bits; $S = 2^s$ sets ;

$t = m - (s + b)$ tag bits (leftover)

exercise

4 byte blocks, 4 sets

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

index	valid	tag	value
00			
01			
10			
11			

$B = 4 = 2^b$ byte block size

$b = 2$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

$m = 8$ bit addresses

$t = m - (s + b) = 4$ tag bits

exercise

4 byte blocks, 4 sets

address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

$B = 4 = 2^b$ byte block size

$b = 2$ (block) offset bits

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

index	valid	tag	value
00			
01			
10			
11			

$m = 8$ bit addresses

$t = m - (s + b) = 4$ tag bits

exercise

4 byte blocks, 4 sets

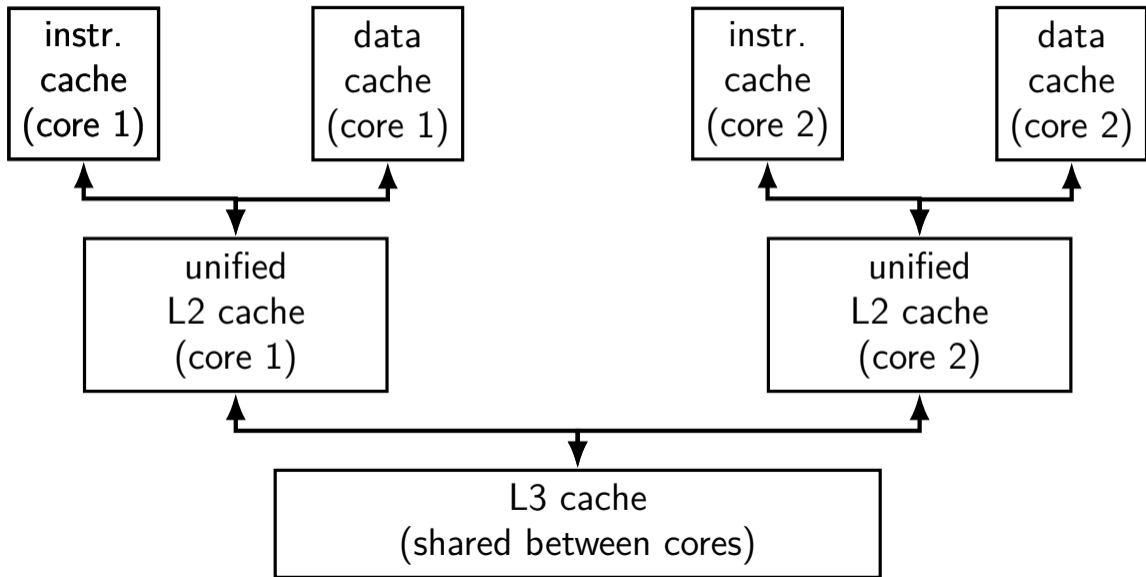
address (hex)	result
00000000 (00)	
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

tag index offset

index	valid	tag	value
00			
01			
10			
11			

exercise: which accesses are hits?

split caches; multiple cores



hierarchy and instruction/data caches

typically separate data and instruction caches for L1

(almost) never going to read instructions as data or vice-versa

avoids instructions evicting data and vice-versa

can optimize instruction cache for different access pattern

easier to build fast caches: that handles less accesses at a time

cache accesses and C code (1)

```
int scaleFactor;  
  
int scaleByFactor(int value) {  
    return value * scaleFactor;  
}
```

```
scaleByFactor:  
    movl scaleFactor, %eax  
    imull %edi, %eax  
    ret
```

exercice: what data cache accesses does this function do?

cache accesses and C code (1)

```
int scaleFactor;  
  
int scaleByFactor(int value) {  
    return value * scaleFactor;  
}
```

```
scaleByFactor:  
    movl scaleFactor, %eax  
    imull %edi, %eax  
    ret
```

exercice: what data cache accesses does this function do?

- 4-byte read of scaleFactor
- 8-byte read of return address

possible scaleFactor use

```
for (int i = 0; i < size; ++i) {  
    array[i] = scaleByFactor(array[i]);  
}
```

misses and code (2)

scaleByFactor:

```
movl scaleFactor, %eax
imull %edi, %eax
ret
```

suppose each time this is called in the loop:

return address located at address 0x7fffffe43b8

scaleFactor located at address 0x6bc3a0

with direct-mapped 32KB cache w/64 B blocks, what is their:

	return address	scaleFactor
tag		
index		
offset		

misses and code (2)

scaleByFactor:

```
movl scaleFactor, %eax
imull %edi, %eax
ret
```

suppose each time this is called in the loop:

return address located at address 0x7ffffffe43b8

scaleFactor located at address 0x6bc3a0

with direct-mapped 32KB cache w/64 B blocks, what is their:

	return address	scaleFactor
tag	0xffffffffc	0xd7
index	0x10e	0x10e
offset	0x38	0x20

misses and code (2)

scaleByFactor:

```
movl scaleFactor, %eax
imull %edi, %eax
ret
```

suppose each time this is called in the loop:

return address located at address 0x7ffffffe43b8

scaleFactor located at address 0x6bc3a0

with direct-mapped 32KB cache w/64 B blocks, what is their:

	return address	scaleFactor
tag	0xffffffffc	0xd7
index	0x10e	0x10e
offset	0x38	0x20

conflict miss coincidences?

obviously I set that up to have the same index

have to use exactly the right amount of stack space...

but one of the reasons we'll want something better than
direct-mapped cache

C and cache misses (warmup 1)

```
int array[4];
```

```
...
```

```
int even_sum = 0, odd_sum = 0;
```

```
even_sum += array[0];
```

```
odd_sum += array[1];
```

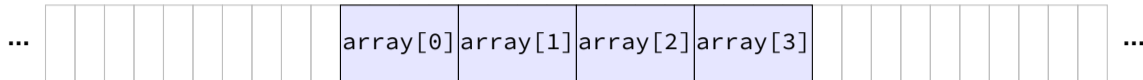
```
even_sum += array[2];
```

```
odd_sum += array[3];
```

Assume everything but array is kept in registers (and the compiler does not do anything funny).

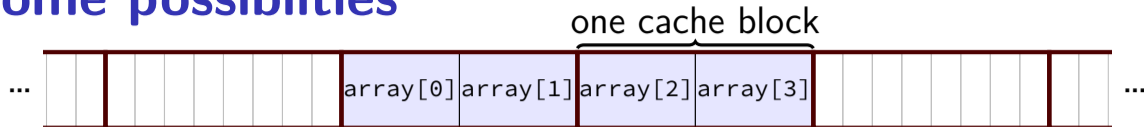
How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

some possibilities



Q1: how do cache blocks correspond to array elements?
not enough information provided!

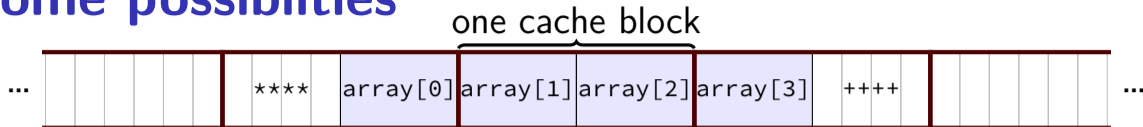
some possibilities



if array[0] starts at beginning of a cache block...
array split across two cache blocks

memory access	cache contents afterwards
—	(empty)
read array[0] (miss)	{array[0], array[1]}
read array[1] (hit)	{array[0], array[1]}
read array[2] (miss)	{array[2], array[3]}
read array[3] (hit)	{array[2], array[3]}

some possibilities

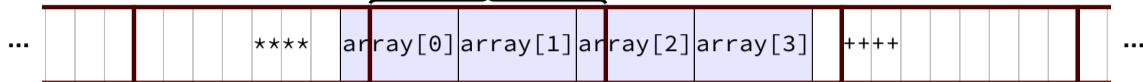


if `array[0]` starts right in the middle of a cache block
array split across three cache blocks

memory access	cache contents afterwards
—	(empty)
read <code>array[0]</code> (miss)	{****, <code>array[0]</code> }
read <code>array[1]</code> (miss)	{ <code>array[1]</code> , <code>array[2]</code> }
read <code>array[2]</code> (hit)	{ <code>array[1]</code> , <code>array[2]</code> }
read <code>array[3]</code> (miss)	{ <code>array[3]</code> , +++++}

some possibilities

one cache block



if `array[0]` starts at an odd place in a cache block,
need to read two cache blocks to get most array elements

memory access	cache contents afterwards
—	(empty)
read <code>array[0]</code> byte 0 (miss)	{ <code>****</code> , <code>array[0]</code> byte 0 }
read <code>array[0]</code> byte 1-3 (miss)	{ <code>array[0]</code> byte 1-3, <code>array[2]</code> , <code>array[3]</code> byte 0 }
read <code>array[1]</code> (hit)	{ <code>array[0]</code> byte 1-3, <code>array[2]</code> , <code>array[3]</code> byte 0 }
read <code>array[2]</code> byte 0 (hit)	{ <code>array[0]</code> byte 1-3, <code>array[2]</code> , <code>array[3]</code> byte 0 }
read <code>array[2]</code> byte 1-3 (miss)	{ part of <code>array[2]</code> , <code>array[3]</code> , <code>++++</code> }
read <code>array[3]</code> (hit)	{ part of <code>array[2]</code> , <code>array[3]</code> , <code>++++</code> }

aside: alignment

compilers and malloc/new implementations usually try **align** values

align = make address be multiple of something

most important reason: don't cross cache block boundaries

C and cache misses (warmup 2)

```
int array[4];  
int even_sum = 0, odd_sum = 0;  
even_sum += array[0];  
even_sum += array[2];  
odd_sum += array[1];  
odd_sum += array[3];
```

Assume everything but array is kept in registers (and the compiler does not do anything funny).

Assume array[0] at beginning of cache block.

How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

C and cache misses (warmup 3)

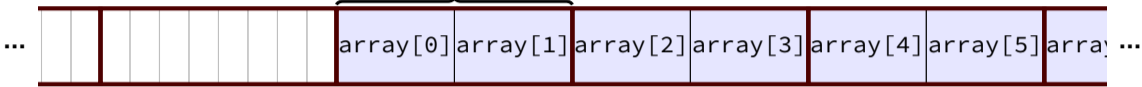
```
int array[8];  
...  
int even_sum = 0, odd_sum = 0;  
even_sum += array[0];  
odd_sum += array[1];  
even_sum += array[2];  
odd_sum += array[3];  
even_sum += array[4];  
odd_sum += array[5];  
even_sum += array[6];  
odd_sum += array[7];
```

Assume everything but array is kept in registers (and the compiler does not do anything funny), and array[0] at beginning of cache block.

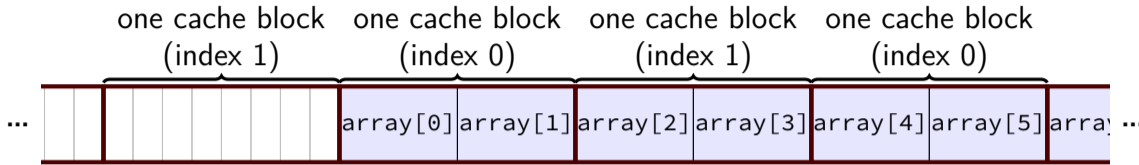
How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

exercise solution

one cache block
(index 0)



exercise solution



exercise solution

one cache block (index 1) one cache block (index 0) one cache block (index 1) one cache block (index 0)

memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}

exercise solution

one cache block one cache block one cache block one cache block

observation: what happens in set 0 doesn't affect set 1
when evaluating set 0 accesses,
can ignore non-set 0 accesses/content

—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}

exercise solution

one cache block one cache block one cache block one cache block

observation: what happens in set 0 doesn't affect set 1
when evaluating set 0 accesses,
can ignore non-set 0 accesses/content

—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}

exercise solution

one cache block (index 1) one cache block (index 0) one cache block (index 1) one cache block (index 0)

...																				...	
memory access																					
—																					
read array[0] (miss)																					
read array[1] (hit)																					
read array[2] (miss)																					
read array[3] (hit)																					
read array[4] (miss)																					
read array[5] (hit)																					
read array[6] (miss)																					
read array[7] (hit)																					

exercise solution

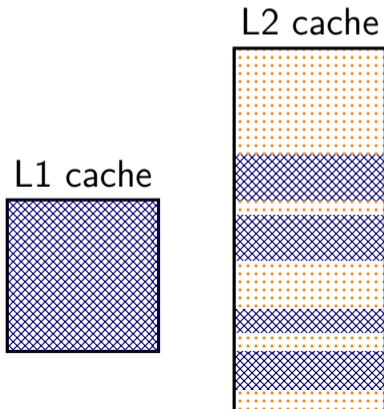
	one cache block (index 1)	one cache block (index 0)	one cache block (index 1)	one cache block (index 0)
...	array[0] array[1]		array[2] array[3]	array[4] array[5] array[6] array[7] ...
memory access	set 0 afterwards		set 1 afterwards	
—	(empty)		(empty)	
read array[0] (miss)	{array[0], array[1]}		(empty)	
read array[1] (hit)	{array[0], array[1]}		(empty)	
read array[2] (miss)	{array[0], array[1]}		{array[2], array[3]}	
read array[3] (hit)	{array[0], array[1]}		{array[2], array[3]}	
read array[4] (miss)	{array[4], array[5]}		{array[2], array[3]}	
read array[5] (hit)	{array[4], array[5]}		{array[2], array[3]}	
read array[6] (miss)	{array[4], array[5]}		{array[6], array[7]}	
read array[7] (hit)	{array[4], array[5]}		{array[6], array[7]}	

backup slides

inclusive versus exclusive

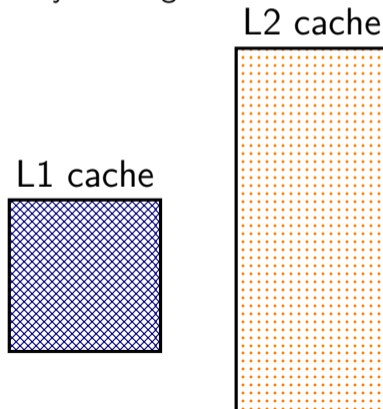
L2 inclusive of L1

everything in L1 cache duplicated in L2
adding to L1 also adds to L2



L2 exclusive of L1

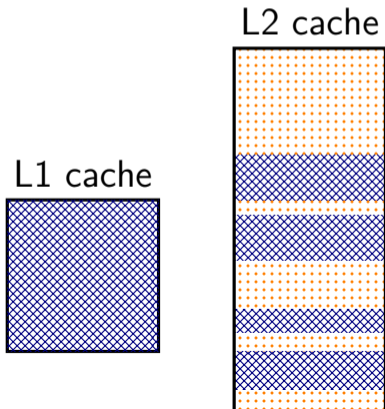
L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2



inclusive versus exclusive

L2 inclusive of L1

everything in L1 cache duplicated in L2
adding to L1 also adds to L2



L2 exclusive of L1

L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2

inclusive policy:

no extra work on eviction
but duplicated data

easier to explain when

L_k shared by multiple $L(k-1)$ caches?

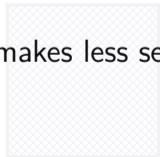
inclusive versus exclusive

L2 inclusive of L1

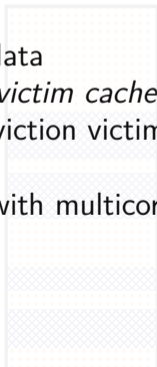
everything in L1 cache duplicated in L2
adding to L1 also adds to L2

exclusive policy:
avoid duplicated data
sometimes called *victim cache*
(contains cache eviction victims)

makes less sense with multicore



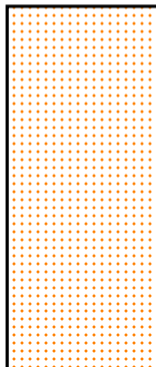
L2 cache



L2 exclusive of L1

L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2

L2 cache



L1 cache

