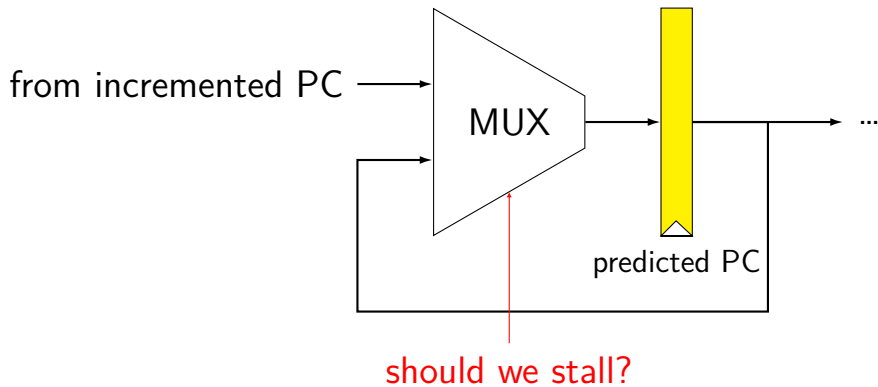
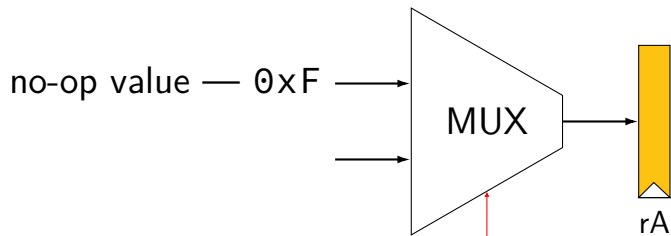


fetch/fetch logic — advance or not



fetch/decode logic — bubble or not



should we send
no-op value (“bubble”)?

HCLRS signals

```
register aB {  
    ...  
}
```

HCLRS: every register bank has these MUXes built-in

`stall_B`: keep **old value** for all registers
register input → register output

`bubble_B`: use **default value** for all registers
register input → default value

exercise

```
register aB {  
    value : 8 = 0xFF;  
}  
...
```

stall: keep old value bubble: store default value
--

time	a_value	B_value	stall_B	bubble_B
0	0x01	0xFF	0	0
1	0x02	???	1	0
2	0x03	???	0	0
3	0x04	???	0	1
4	0x05	???	0	0
5	0x06	???	0	0
6	0x07	???	1	0
7	0x08	???	1	0
8		???		

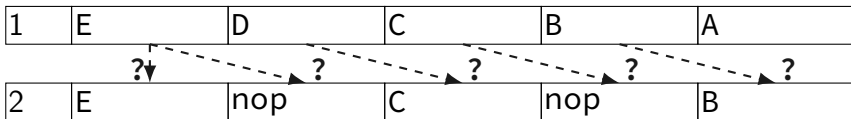
exercise result

```
register aB {  
    value : 8 = 0xFF;  
}  
...
```

time	a_value	B_value	stall_B	bubble_B
0	0x01	0xFF	0	0
1	0x02	0x01	1	0
2	0x03	0x01	0	0
3	0x04	0x03	0	1
4	0x05	0xFF	0	0
5	0x06	0x05	0	0
6	0x07	0x06	1	0
7	0x08	0x06	1	0
8		0x06		

exercise: squash + stall (1)

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

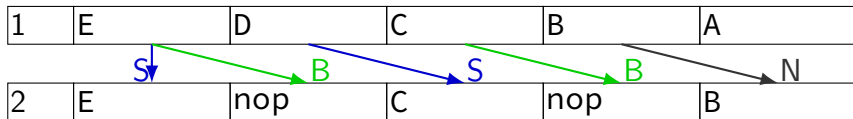


stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

exercise: what are the ?s
write down your answers,
then compare with your neighbors

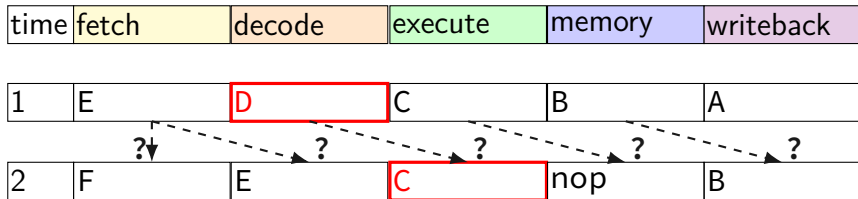
exercise: squash + stall (1)

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------



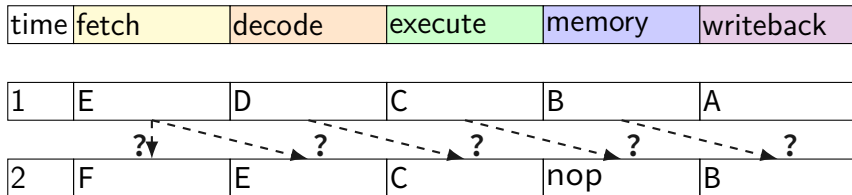
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

exercise: squash + stall (2)



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

exercise: squash + stall (2)

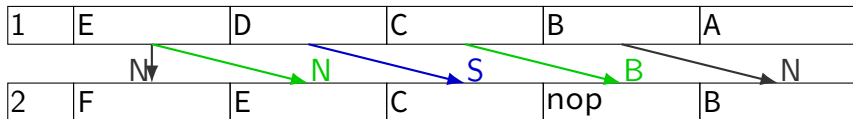


stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

exercise: what are the ?s
write down your answers,
then compare with your neighbors

exercise: squash + stall (2)

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

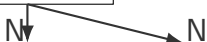


stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

ret stall

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

0	call
---	------



1	ret	call
---	-----	------

2	wait for ret	ret	call
---	--------------	-----	------

3	wait for ret	nothing	ret	call (store)
---	--------------	---------	-----	--------------

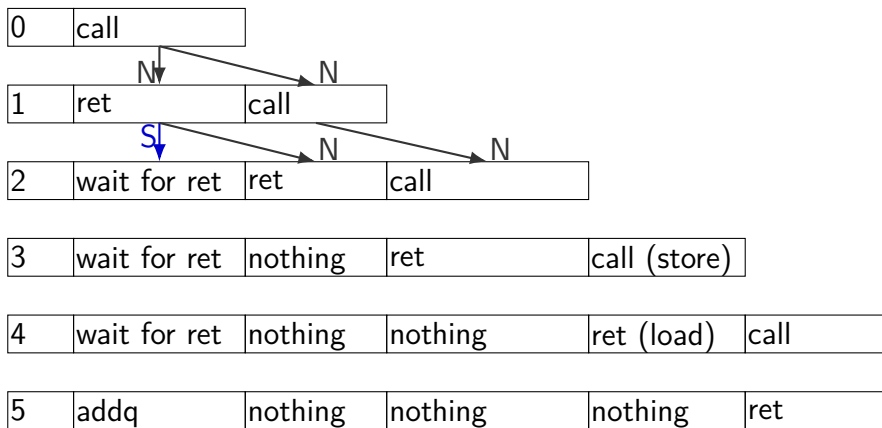
4	wait for ret	nothing	nothing	ret (load)	call
---	--------------	---------	---------	------------	------

5	addq	nothing	nothing	nothing	ret
---	------	---------	---------	---------	-----

stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

ret stall

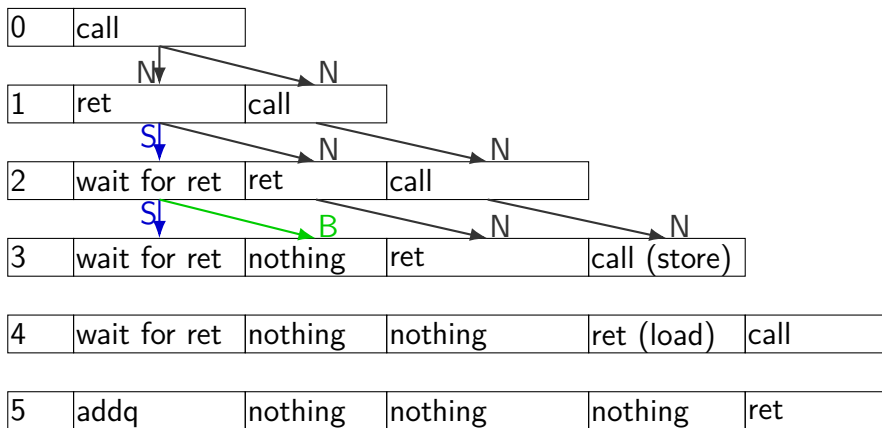
time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

ret stall

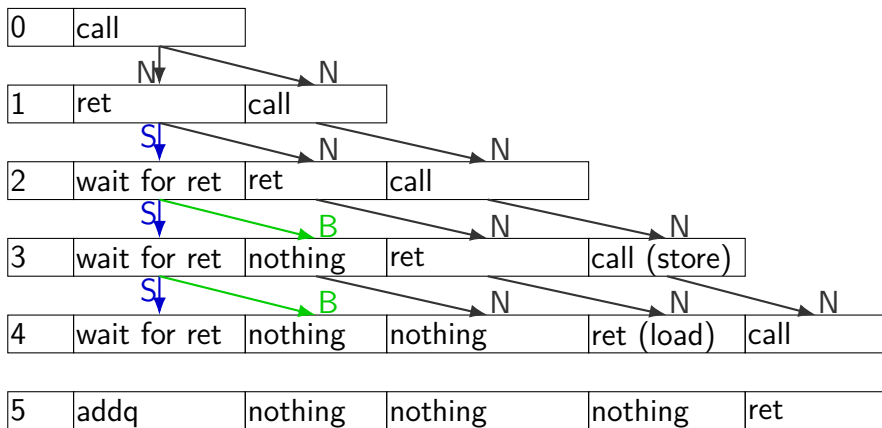
time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

ret stall

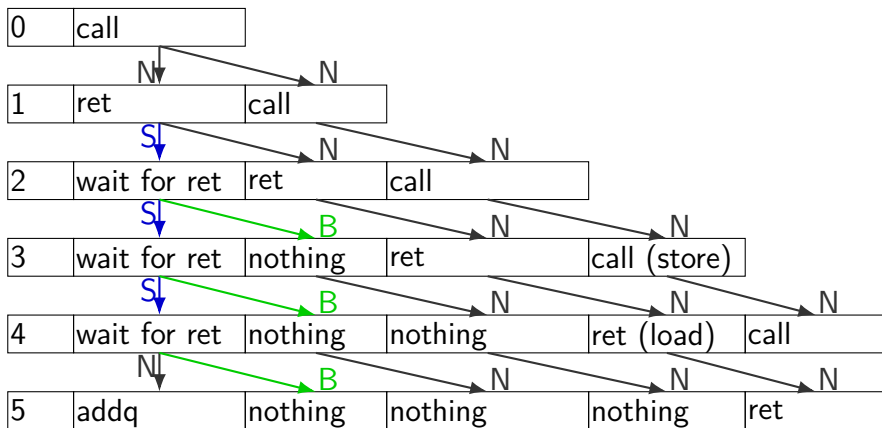
time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

ret stall

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------



stall (S) = keep old value; normal (N) = use new value
 bubble (B) = use default (no-op);

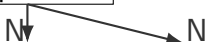
HCLRS bubble example

```
register fD {  
    icode : 4 = NOP;  
    rA   : 4 = REG_NONE;  
    rB   : 4 = REG_NONE;  
    ...  
};  
wire need_ret_bubble : 1;  
need_ret_bubble = ( D_icode == RET ||  
                   E_icode == RET ||  
                   M_icode == RET );  
  
bubble_D = ( need_ret_bubble ||  
             ... /* other cases */ );
```


squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------

1	subq
---	------



2	jne	subq
---	-----	------

3	addq [?]	jne	subq (set ZF)
---	----------	-----	---------------

4	rmmovq [?]	addq [?]	jne (use ZF)	subq
---	------------	----------	--------------	------

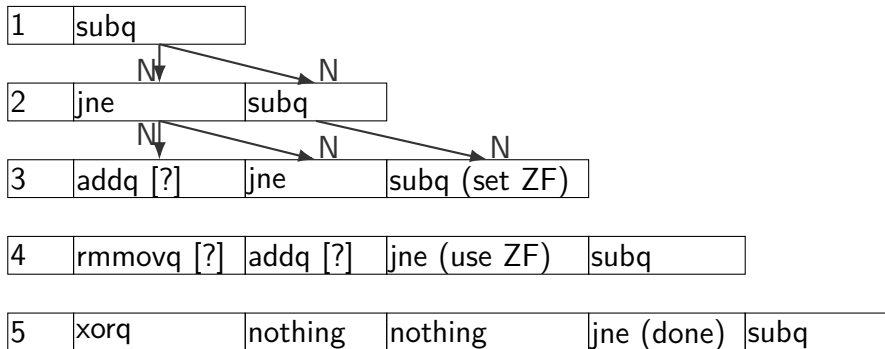
5	xorq	nothing	nothing	jne (done)	subq
---	------	---------	---------	------------	------

stall (S) = keep old value; normal (N) = use new value

bubble (B) = use default (no-op);

squashing with stall/bubble

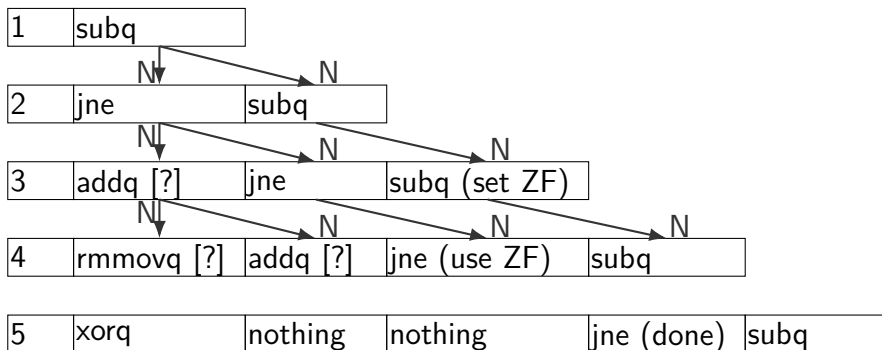
time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

squashing with stall/bubble

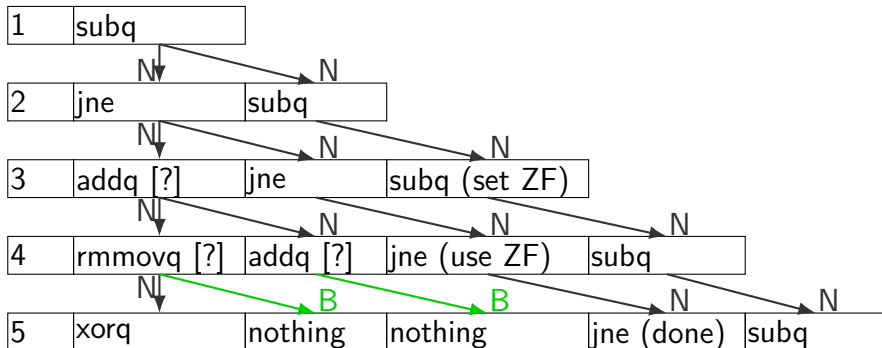
time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

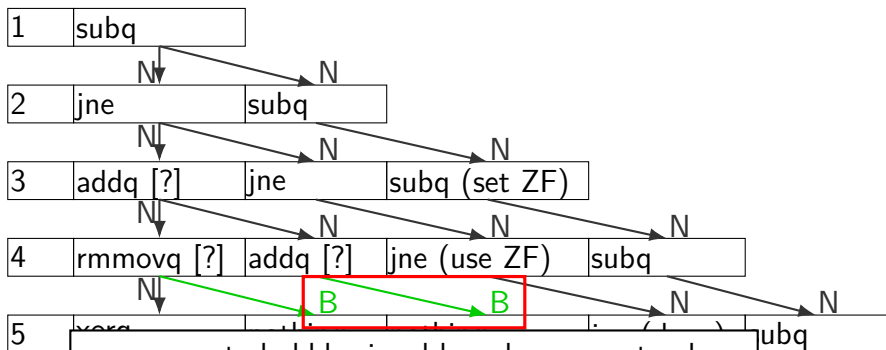
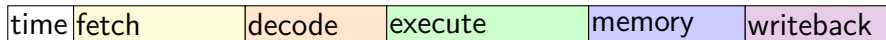
squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
------	-------	--------	---------	--------	-----------



stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

squashing with stall/bubble



can compute bubble signal based on execute phase
won't even start CC write for addq
new value

bubble (B) = use default (no-op);

squashing HCLRS

```
just_detected_mispredict =  
    e_icode == JXX && !branchTaken;  
bubble_D = just_detected_mispredict || ...;  
bubble_E = just_detected_mispredict || ...;
```

missing pieces

multi-cycle memories

beyond pipelining: out-of-order, multiple issue

missing pieces

multi-cycle memories

beyond pipelining: out-of-order, multiple issue

multi-cycle memories

ideal case for memories: single-cycle

achieved with **caches** (next topic)

fast access to small number of things

typical performance:

90+% of the time: single-cycle

sometimes many cycles (3–400+)

variable speed memories

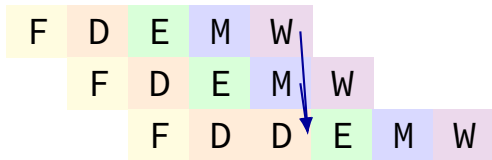
cycle # 0 1 2 3 4 5 6 7 8

memory is fast: (cache "hit"; recently accessed?)

`mrmovq 0(%rbx), %r8`

`mrmovq 0(%rcx), %r9`

`addq %r8, %r9`



memory is slow: (cache "miss")

`mrmovq 0(%rbx), %r8`

`mrmovq 0(%rcx), %r9`

`addq %r8, %r9`

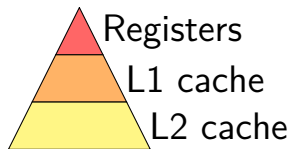
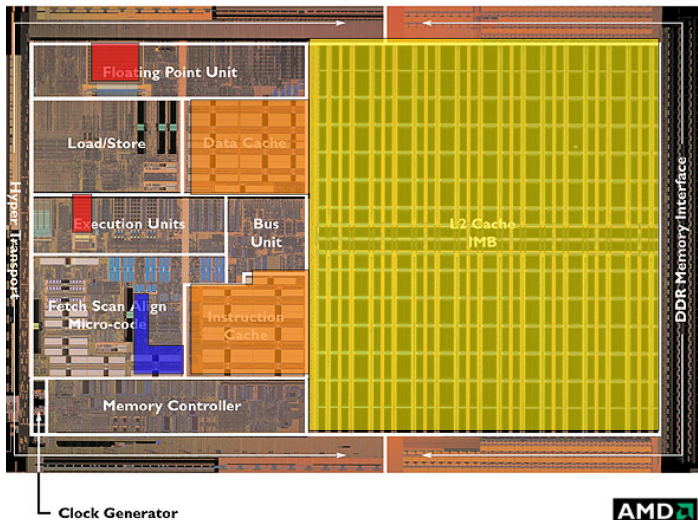



missing pieces

multi-cycle memories

beyond pipelining: out-of-order, multiple issue

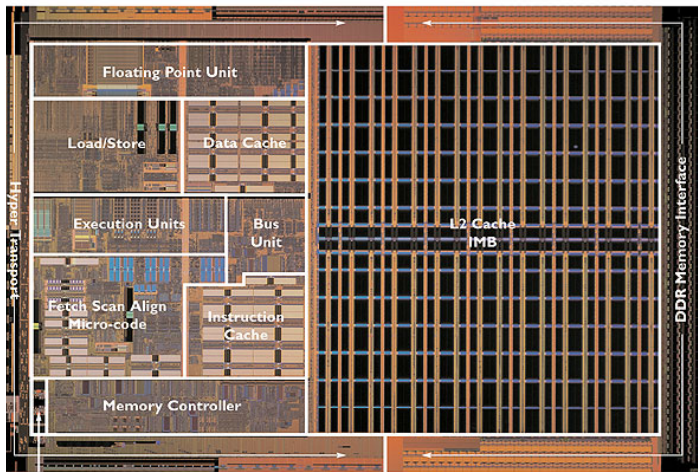
2004 CPU



 Branch Prediction (approximate)



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

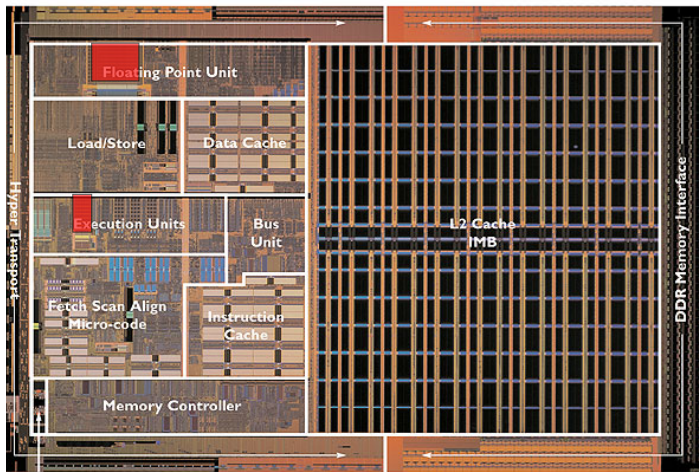
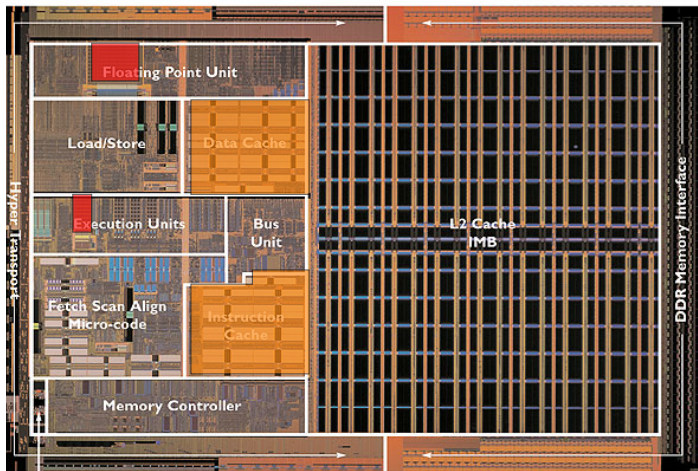


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

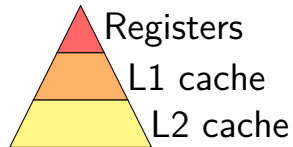
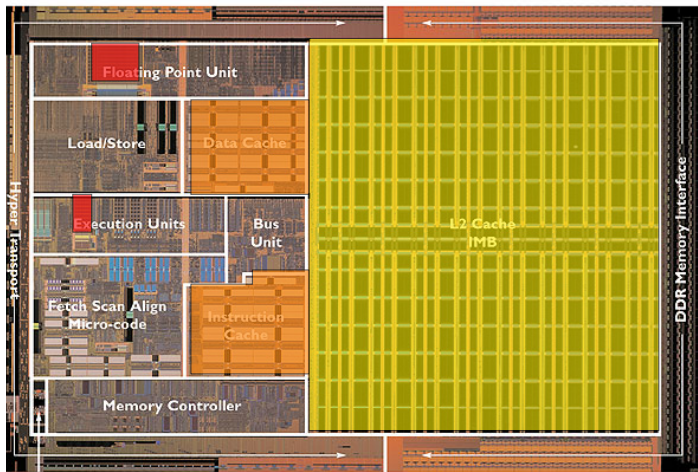


Clock Generator



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

2004 CPU

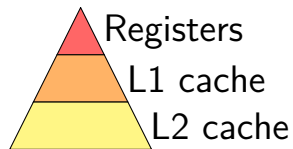
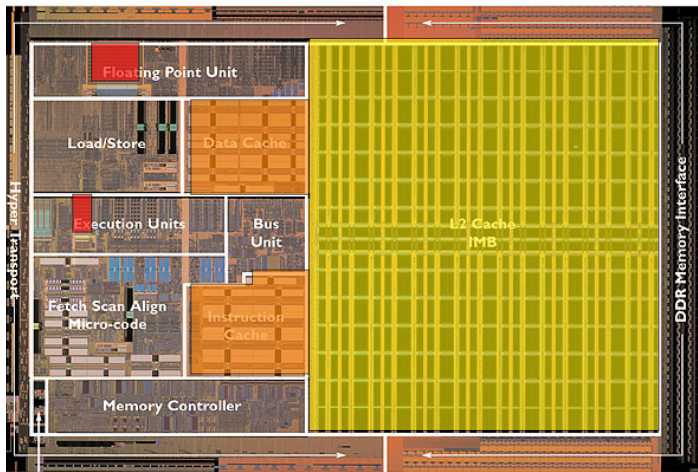


Clock Generator



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

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

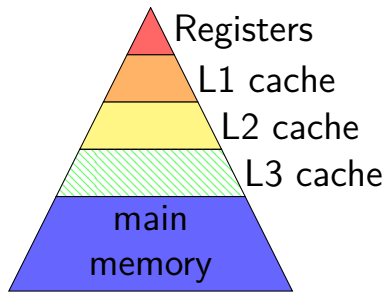
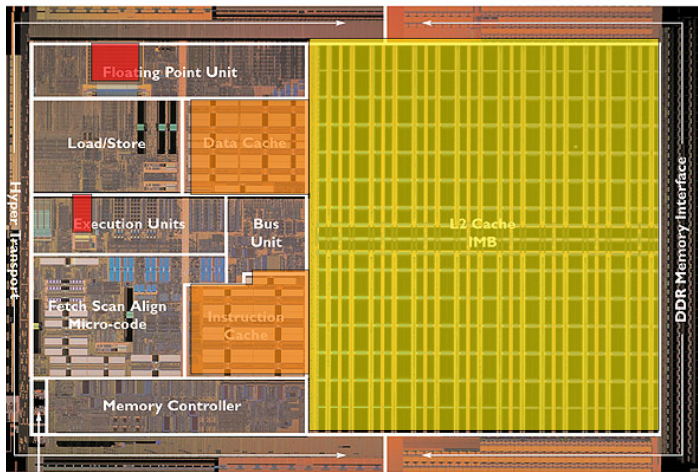


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

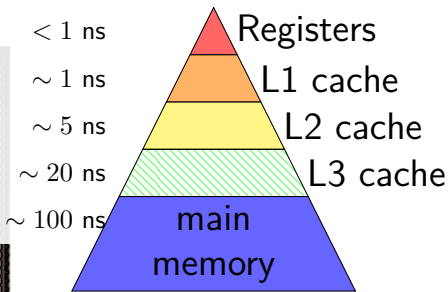
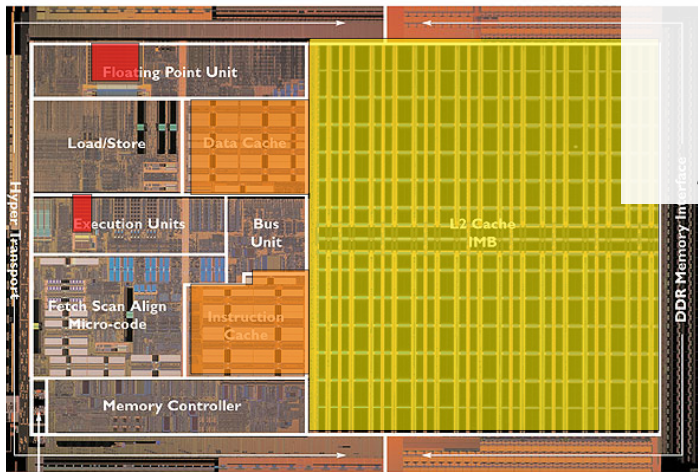
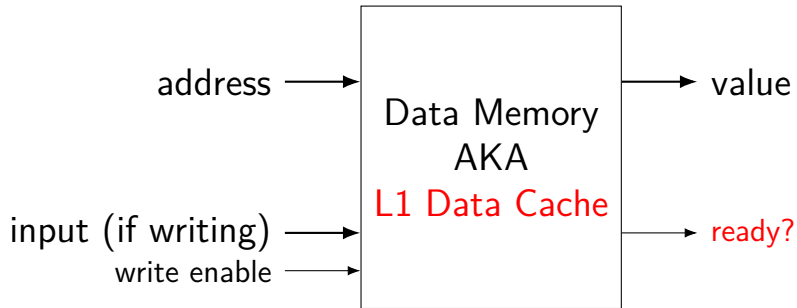
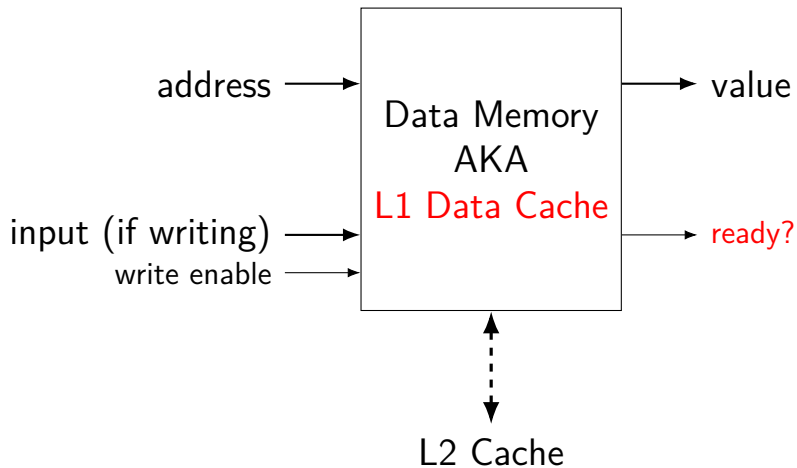


Image: approx 2004 AMD press image of Opteron die; approx register location via chip-architect.org (Hans de Vries)

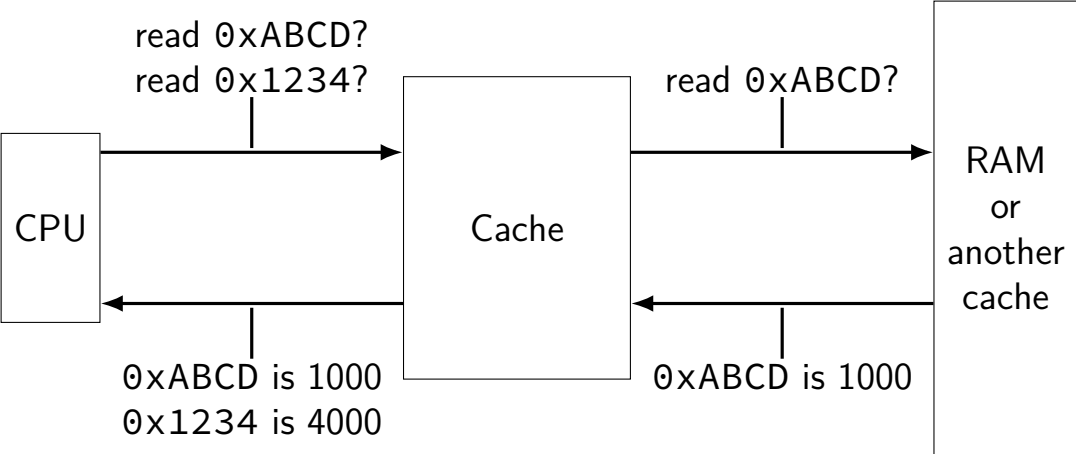
cache: real memory



cache: real memory



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]

Caching

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

Cache

Memory

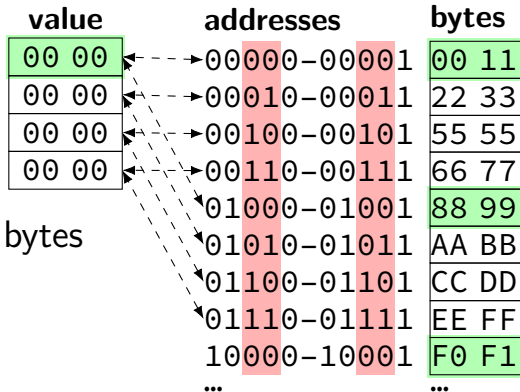
index

00

01

10

11



cache block: 2 bytes

direct-mapped

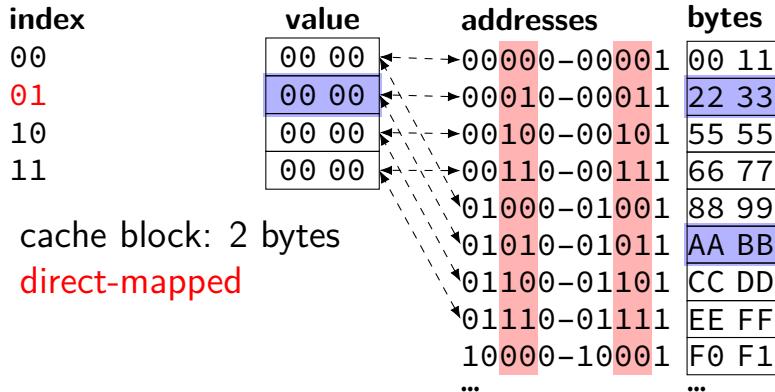
building a (direct-mapped) cache

read byte at 01011?

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

Cache

Memory



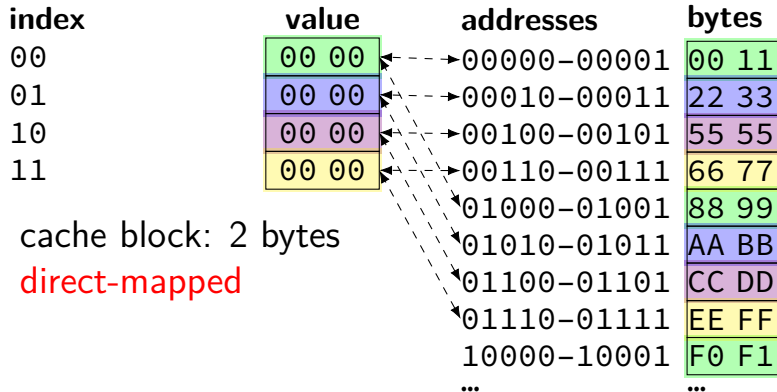
building a (direct-mapped) cache

read byte at 01011?

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

Cache

Memory



building a (direct-mapped) cache

read byte at 01011?

Cache

Memory

index	valid	value	addresses	bytes
00	0	00 00		1
01	0	00 00	00010-00011	22 33
10	0		0-00101	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
...	...

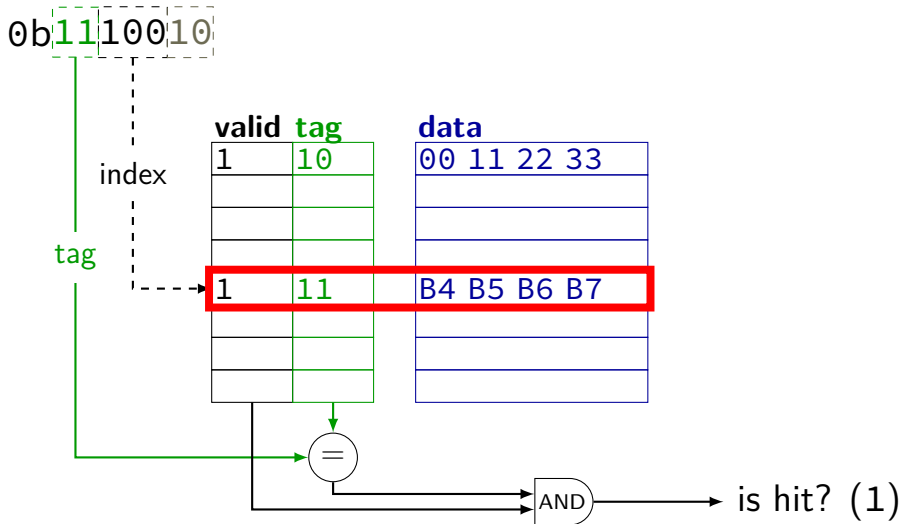
cache operation (read)

0b1110010

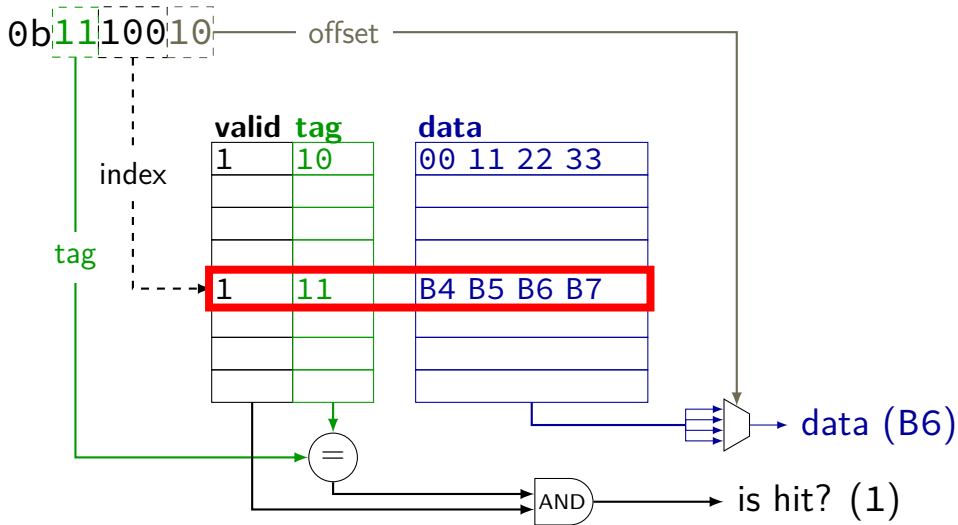
index

valid	tag	data
1	10	00 11 22 33
1	11	B4 B5 B6 B7

cache operation (read)



cache operation (read)



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	00	00 11 22 33
1	1	01	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	00	00 11 22 33
1	1	01	CC DD EE FF

2 byte blocks, 8 sets

index	valid	tag	value
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 bytes

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

index	valid	tag	value
0	1	00	00 11 22 33
1	1	01	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	???		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	00	00 11 22 33
1	1	01	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	00	00 11 22 33
1	1	01	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	00	00 11 22 33
1	1	01	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	00	00 11 22 33
1	1	01	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 formulas (direct-mapped only)

m memory addresses bits (Y86-64: 64)

$S = 2^s$ number of sets

s (set) index bits

$B = 2^b$ block size

b (block) offset 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		

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

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

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

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

$b = 1$ (block) offset bits

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

example access pattern (1)

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

2 byte blocks, 4 sets

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

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

$b = 1$ (block) offset bits

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

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

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

$b = 1$ (block) offset bits

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

example access pattern (1)

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

2 byte blocks, 4 sets

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

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

$b = 1$ (block) offset bits

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

$m = 8$ bit addresses

$S = 4 = 2^s$ 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		

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

$b = 1$ (block) offset bits

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

$m = 8$ bit addresses

$S = 4 = 2^s$ 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		

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

$b = 1$ (block) offset bits

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

$m = 8$ bit addresses

$S = 4 = 2^s$ 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		

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

$b = 1$ (block) offset bits

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

$m = 8$ bit addresses

$S = 4 = 2^s$ 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		

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

$b = 1$ (block) offset bits

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

$m = 8$ bit addresses

$S = 4 = 2^s$ 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		

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

$b = 1$ (block) offset bits

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

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

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

miss caused by conflict

exercise

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

4 byte blocks, 4 sets

index	valid	tag	value
00			
01			
10			
11			

exercise

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

4 byte blocks, 4 sets

index	valid	tag	value
00			
01			
10			
11			

how is the 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

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

4 byte blocks, 4 sets

index	valid	tag	value
00			
01			
10			
11			

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 2$ (block) offset bits

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

exercise

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

tag index offset

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 2$ (block) offset bits

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

4 byte blocks, 4 sets

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)	

tag index offset

index	valid	tag	value
00			
01			
10			
11			

exercise: how many accesses are hits?

exercise

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

tag index offset

4 byte blocks, 4 sets

index	valid	tag	value
00	1	0000	mem[0x00] - mem[0x03]
01	0		
10			
11			

exercise

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

tag index offset

4 byte blocks, 4 sets

index	valid	tag	value
00	1	0000	mem[0x00] - mem[0x03]
01	0		
10			
11			

exercise

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

tag index offset

4 byte blocks, 4 sets

index	valid	tag	value
00	1	0110	mem[0x60] - mem[0x63]
01	0		
10			
11			

exercise

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

tag index offset

4 byte blocks, 4 sets

index	valid	tag	value
00	1	0110	mem[0x60] - mem[0x63]
01	0		
10			
11			

exercise

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

tag index offset

4 byte blocks, 4 sets

index	valid	tag	value
00	1	0110	mem[0x60] - mem[0x63]
01	0		
10			
11			

exercise

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

tag index offset

4 byte blocks, 4 sets

index	valid	tag	value
00	1	0000	mem[0x00] - mem[0x03]
01	0		
10			
11			

exercise

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

tag index offset

4 byte blocks, 4 sets

index	valid	tag	value
00	1	0000	mem[0x00] - mem[0x03]
01	1	0110	mem[0x64] - mem[0x67]
10			
11			

exercise

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

tag index offset

4 byte blocks, 4 sets

index	valid	tag	value
00	1	0000	mem[0x00] - mem[0x03]
01	1	0110	mem[0x64] - mem[0x67]
10			
11			

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

$m = 8$ bit addresses

$S = 4 = 2^s$ sets

$s = 2$ (set) index bits

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

$b = 1$ (block) offset bits

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

miss caused by conflict

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

multiple places to put values with same index
avoid conflict misses

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0		set 0	0		
1	0		set 1	0		

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

way 0

way 1

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

$m = 8$ bit addresses

$S = 2 = 2^s$ sets

$s = 1$ (set) index bits

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

$b = 1$ (block) offset bits

$t = m - (s + b) = 6$ tag bits

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	0			0		

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

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	0			0		

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

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	1	011000	mem[0x62] mem[0x63]	0		

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

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

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

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

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

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

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

tag index offset

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

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

needs to replace block in set 0!

tag index offset

adding associativity

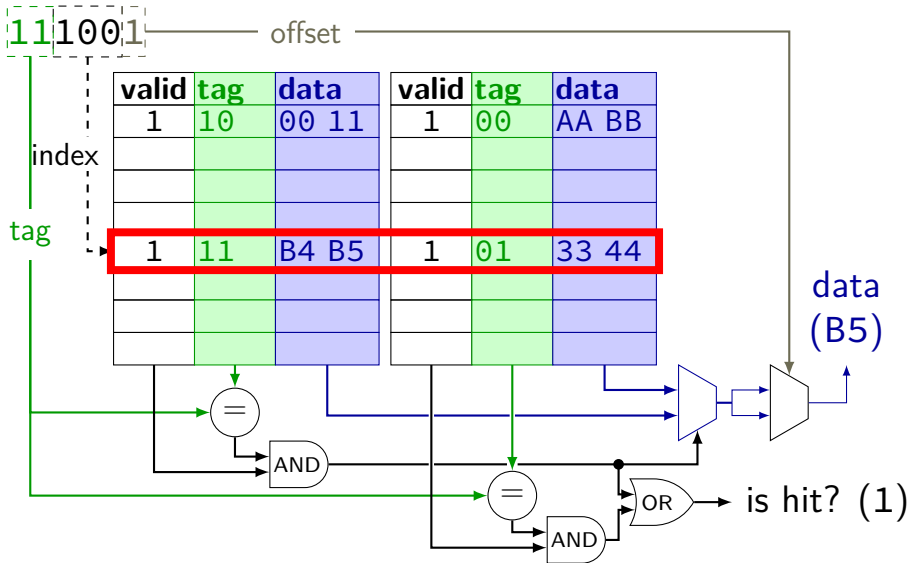
2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

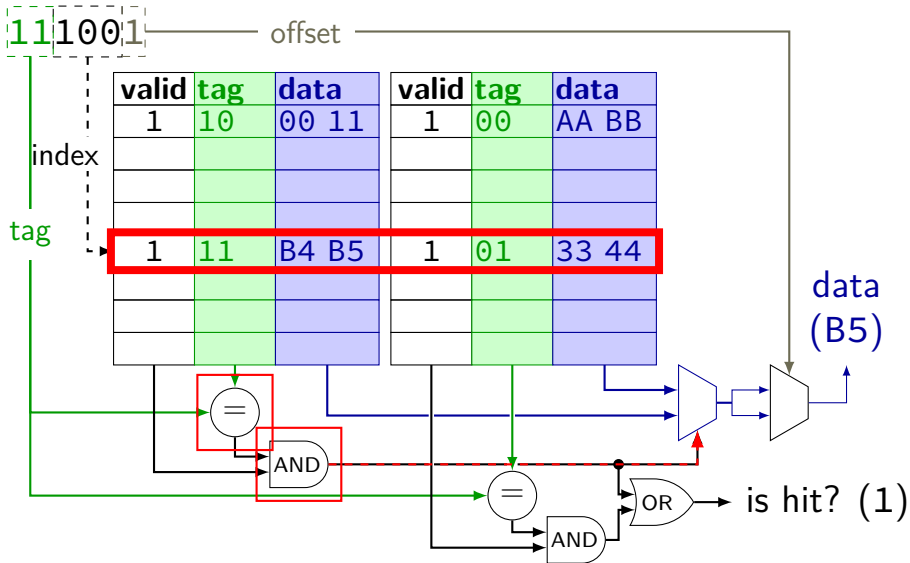
address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

tag index offset

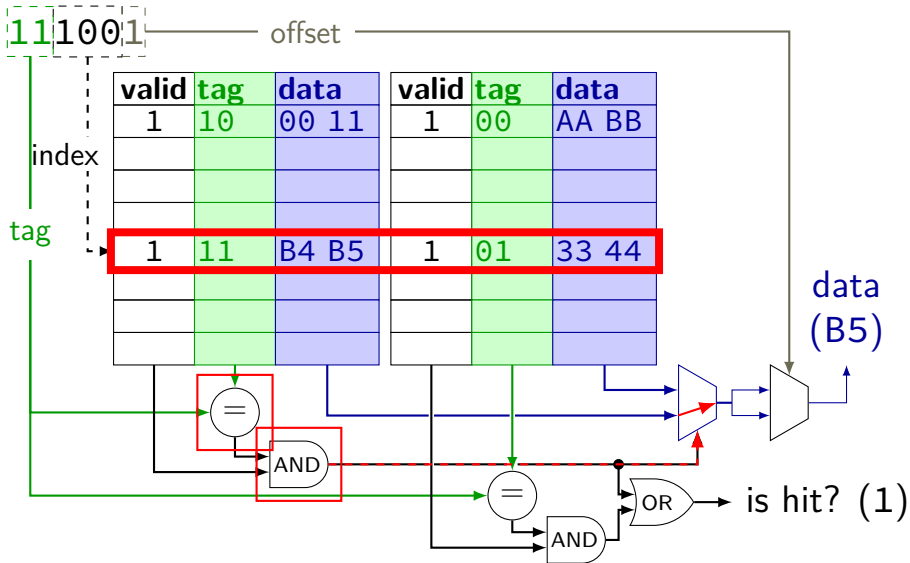
cache operation (associative)



cache operation (associative)



cache operation (associative)



associative lookup possibilities

none of the blocks for the index are valid

none of the valid blocks for the index match the tag
something else is stored there

one of the blocks for the index is valid and matches the tag

associativity terminology

direct-mapped — one block per set

***E*-way set associative** — *E* blocks per set
E ways in the cache

fully associative — one set total (everything in one set)

Tag-Index-Offset formulas (complete)

m memory addresses bits (Y86-64: 64)

E number of blocks per set (“ways”)

$S = 2^s$ number of sets

s (set) index bits

$B = 2^b$ block size

b (block) offset bits

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

$C = B \times S \times E$ cache size (excluding metadata)

Tag-Index-Offset exercise

m	memory addresses bits (Y86-64: 64)
E	number of blocks per set (“ways”)
$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
$t = m - (s + b)$	tag bits
$C = B \times S \times E$	cache size (excluding metadata)

My desktop:

L1 Data Cache: 32 KB, 8 blocks/set, 64 byte blocks

L2 Cache: 256 KB, 4 blocks/set, 64 byte blocks

L3 Cache: 8 MB, 16 blocks/set, 64 byte blocks

Divide the address 0x34567 into **tag**, **index**, **offset** for each cache.

T-I-O exercise: L1

quantity

value for L1

block size (given)

$B = 64\text{Byte}$

$B = 2^b$ (b : block offset bits)

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b : block offset bits)
block offset bits	$b = 6$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b : block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b : block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
	$S = \frac{C}{B \times E}$ (S : number of sets)

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b : block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
	$S = \frac{C}{B \times E}$ (S : number of sets)
number of sets	$S = \frac{32\text{KB}}{64\text{Byte} \times 8} = 64$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b : block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
	$S = \frac{C}{B \times E}$ (S : number of sets)
number of sets	$S = \frac{32\text{KB}}{64\text{Byte} \times 8} = 64$
	$S = 2^s$ (s : set index bits)
set index bits	$s = \log_2(64) = 6$

T-I-O results

	L1	L2	L3
sets	64	1024	8192
block offset bits	6	6	6
set index bits	6	10	13
tag bits		(the rest)	

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567:	3	4	5	6	7
	0011	0100	0101	0110	0111

bits 0-5 (all offsets): 100111 = 0x27

T-I-O: splitting

	L1	L2	L3		
block offset bits	6	6	6		
set index bits	6	10	13		
tag bits	(the rest)				
0x34567:	3	4	5	6	7
	0011	0100	0101	0110	0111
bits 0-5 (all offsets):	100111 = 0x27				

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): 100111 = 0x27

L1:

bits 6-11 (L1 set): 01 0101 = 0x15

bits 12- (L1 tag): 0x34

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): 100111 = 0x27

L1:

bits 6-11 (L1 set): 01 0101 = 0x15

bits 12- (L1 tag): 0x34

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): 100111 = 0x27

L2:

bits 6-15 (set for L2): 01 0001 0101 = 0x115

bits 16-: 0x3

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567:	3	4	5	6	7
	0011	0100	0101	0110	0111

bits 0-5 (all offsets): 100111 = 0x27

L2:

bits 6-15 (set for L2): 01 0001 0101 = 0x115

bits 16-: 0x3

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

0x34567: 3 4 5 6 7
 0011 0100 0101 0110 0111

bits 0-5 (all offsets): 100111 = 0x27

L3:

bits 6-18 (set for L3): 0 1101 0001 0101 = 0xD15

bits 18-: 0x0

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	0			0			
1	0			0			

address (hex)	hit?
00000000 (00)	
00000001 (01)	
00001010 (0A)	
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	0			0			
1	0			0			

address (hex)	hit?
00000000 (00)	
00000001 (01)	
00001010 (0A)	
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

how is the address 21 (00100001) 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, 2 sets

index	V	tag	value	V	tag	value	LRU
0	0			0			
1	0			0			

address (hex)	hit?
00000000 (00)	
00000001 (01)	
00001010 (0A)	
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

tag index offset

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	0			0			
1	0			0			

address (hex)	hit?
00000000 (00)	
00000001 (01)	
00001010 (0A)	
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

tag index offset

exercise: how many accesses are hits?
what is the final state of the cache?

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	1	000000	M[0x00] M[0x01] M[0x02] M[0x03]	0			way 1
1	0			0			

address (hex)	hit?
00000000 (00)	miss
00000001 (01)	
00001010 (0A)	
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

tag index offset

exercise: how many accesses are hits?
what is the final state of the cache?

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	1	000000	M[0x00] M[0x01] M[0x02] M[0x03]	1	000001	M[0x08] M[0x09] M[0x0A] M[0x0B]	way 0
1	0			0			

address (hex)	hit?
00000000 (00)	miss
00000001 (01)	hit
00001010 (0A)	miss
00100001 (21)	
00001100 (0C)	
00000011 (02)	
00100011 (23)	

tag index offset

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	1	00100	M[0x20] M[0x21] M[0x22] M[0x23]	1	00001	M[0x08] M[0x09] M[0x0A] M[0x0B]	way 1
1	0			0			

address (hex)	hit?
00000000 (00)	miss
00000001 (01)	hit
00001010 (0A)	miss
00100001 (21)	miss
00001100 (0C)	miss
00000011 (02)	
00100011 (23)	

tag index offset

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	1	00100	M[0x20] M[0x21] M[0x22] M[0x23]	1	00000	M[0x00] M[0x01] M[0x02] M[0x03]	way 0
1	1	00000	M[0x0C] M[0x0D] M[0x0E] M[0x0F]	0			way 1

address (hex)	hit?
00000000 (00)	miss
00000001 (01)	hit
00001010 (0A)	miss
00100001 (21)	miss
00001100 (0C)	miss
00000011 (02)	miss
00100011 (23)	

tag index offset

exercise

4 byte blocks, 2 sets

index	V	tag	value	V	tag	value	LRU
0	1	00100	M[0x20] M[0x21] M[0x22] M[0x23]	1	00000	M[0x00] M[0x01] M[0x02] M[0x03]	way 1
1	1	00000	M[0x0C] M[0x0D] M[0x0E] M[0x0F]	0			way 1

address (hex)	hit?
00000000 (00)	miss
00000001 (01)	hit
00001010 (0A)	miss
00100001 (21)	miss
00001100 (0C)	miss
00000011 (02)	miss
00100011 (23)	hit

tag index offset

backup slides

cache miss types

compulsory (or *cold*) — **first time** accessing something
doesn't matter how big/flexible the cache is

conflict — sets aren't big/flexible enough
a fully-associative (1-set) cache of the same size would have done better

capacity — cache was not big enough

replacement policies

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
000	
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

how to decide where to insert 0x64?

replacement policies

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value	LRU
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]	1
1	1	011000	mem[0x62] mem[0x63]	0			1

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

track which block was read least recently updated on **every access**

example replacement policies

least recently used and approximations

take advantage of **temporal locality**

exact: $\lceil \log_2(E!) \rceil$ bits per set for E -way cache

good approximations: E to $2E$ bits

first-in, first-out

counter per set — where to replace next

(pseudo-)random

no extra information!