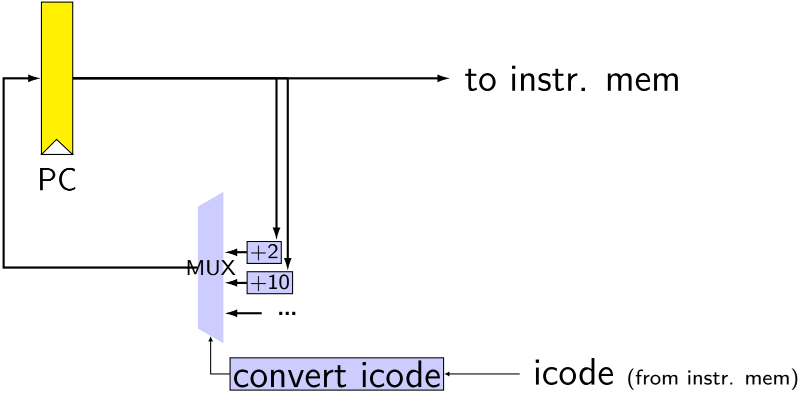
Changelog

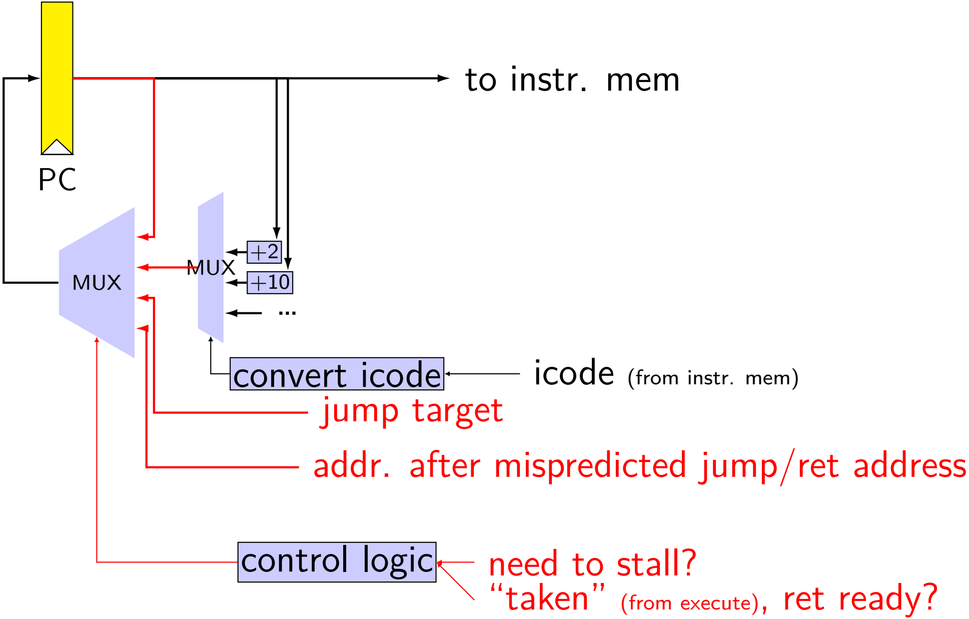
Changes made in this version not seen in first lecture:

- 18 October: move load/use hazard, extra branch prediction slides earlier
- 18 October: HCLRS signals: fixed wrong arrow direction

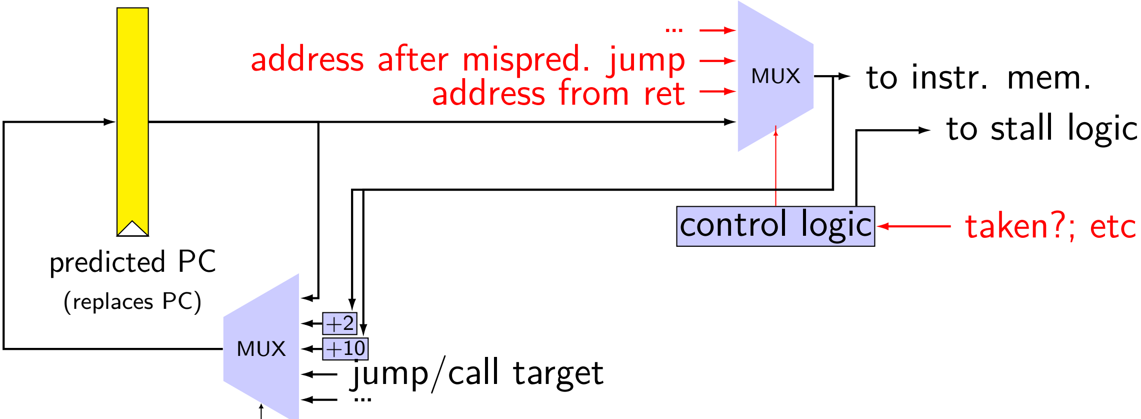
PC update (adding stall)



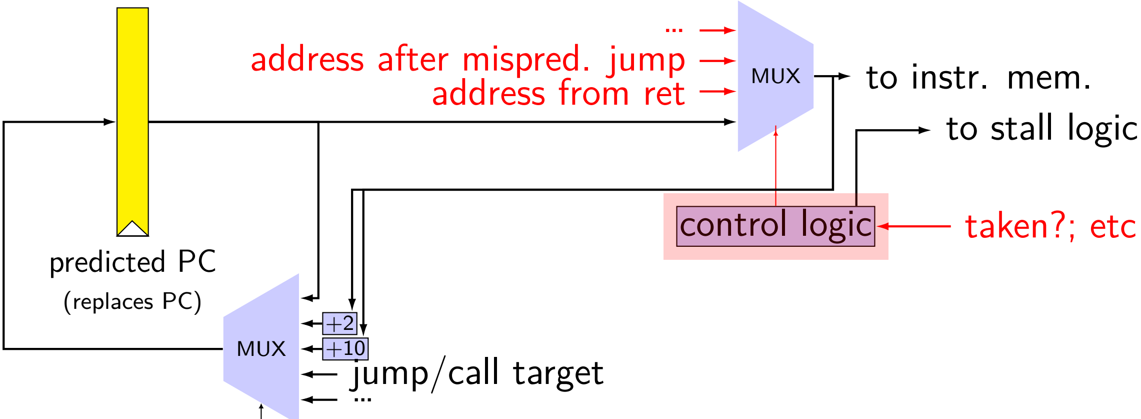
PC update (adding stall)



PC update (rearranged)

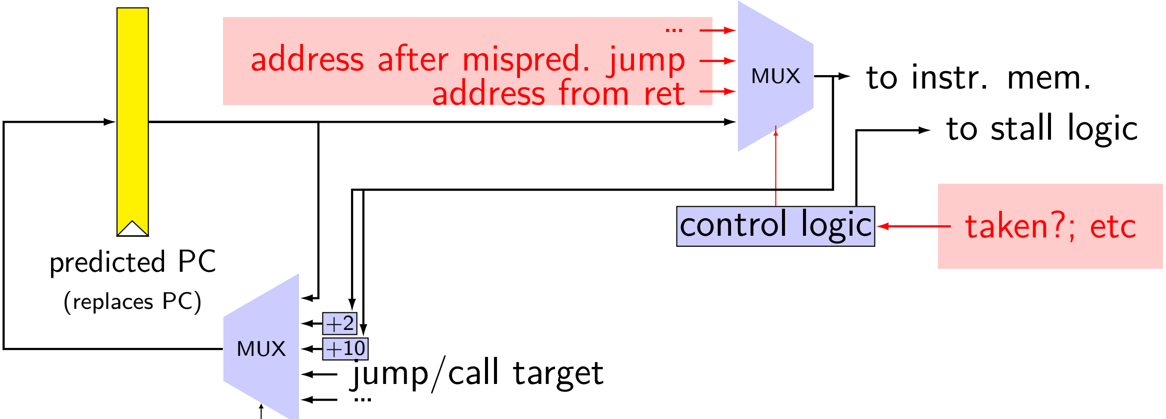


PC update (rearranged)

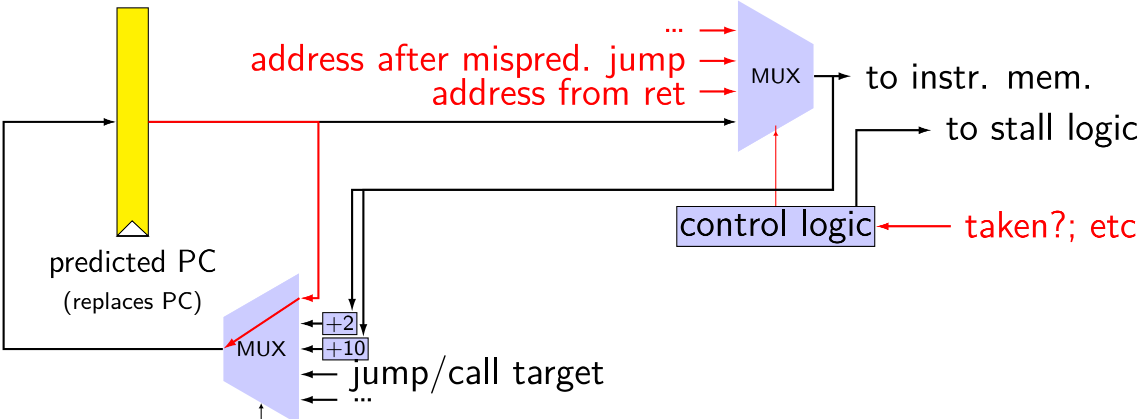


PC update (rearranged)

same logic as before — but happens in next cycle
inputs are from slightly different place…
(e.g. 'taken?' from *execute to memory* registers, not *execute* directly)



PC update (rearranged)



rearranged PC update in HCL

```
/* replacing the PC register: */
register fF {
    predictedPC: 64 = 0;
}

/* actual input to instruction memory */
pc = [
    conditionCodesSaidNotTaken : jumpValP;
        /* from later in pipeline */
    ...
    1: F_predictedPC;
];
```

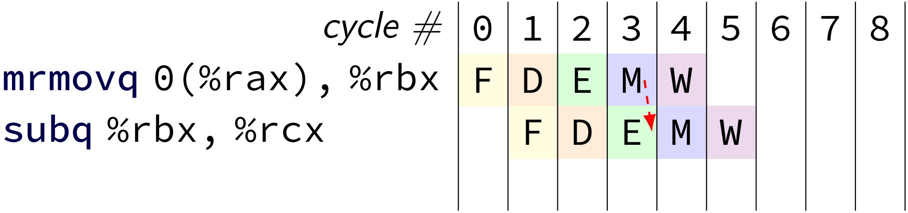
why rearrange PC update?

either works

- correct PC at beginning or end of cycle?
- still some time in cycle to do so...

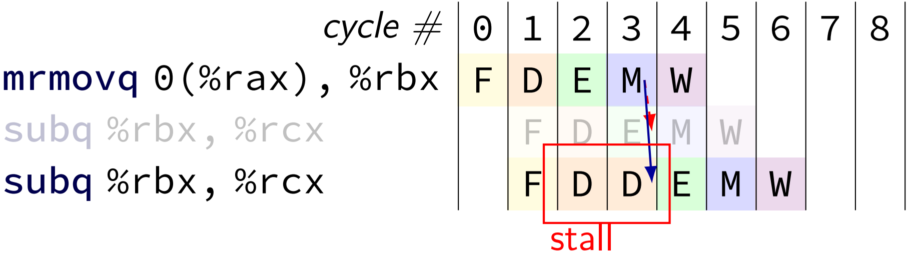
maybe easier to think about branch prediction this way?

unsolved problem



cycle #	0	1	2	3	4	5	6	7	8
`mrmovq 0(%rax), %rbx`	F	D	E	M	W				
`subq %rbx, %rcx`		F	D	E	M	W			

unsolved problem



cycle #	0	1	2	3	4	5	6	7	8
mrmovq 0(%rax), %rbx	F	D	E	M	W				
subq %rbx, %rcx		F	D	E	M	W			
subq %rbx, %rcx		F	D	D	E	M	W		

stall

after forwarding/prediction

where do we still need to stall?

memory output needed in fetch

- `ret` followed by anything

memory output needed in exceute

- `mrmovq` or `popq` + use
  (in immediatelly following instruction)

overall CPU

5 stage pipeline

1 instruction completes every cycle — except hazards

most data hazards: solved by forwarding

load/use hazard: 1 cycle of stalling

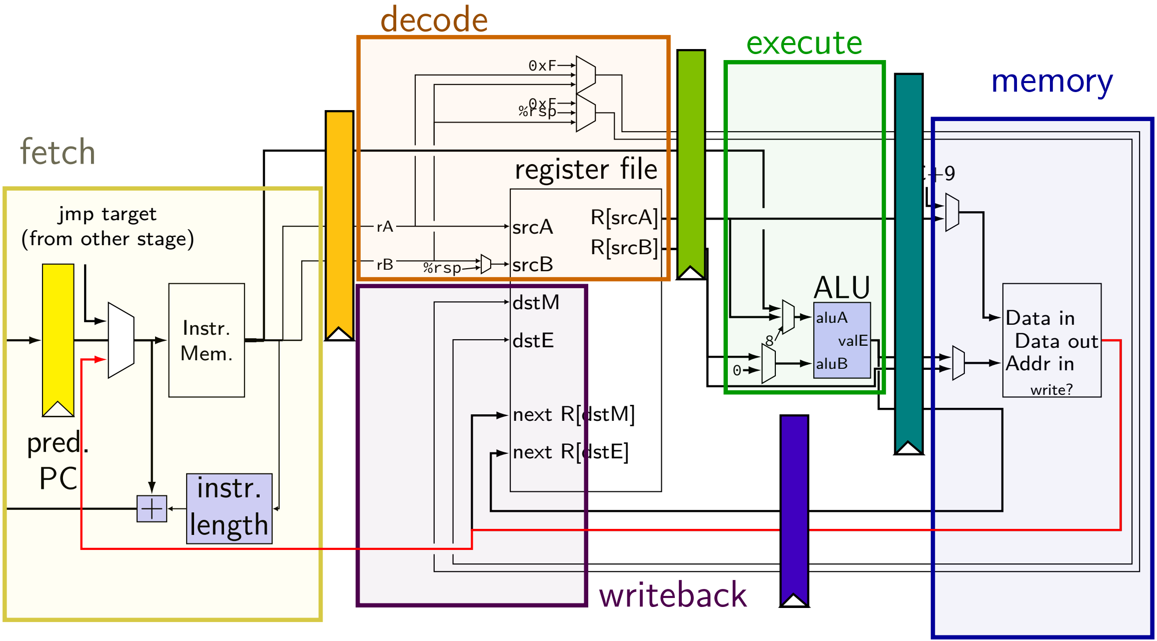
jXX control hazard: branch prediction + squashing

- 2 cycle penalty for misprediction
- (correct misprediction after jXX finishes execute)

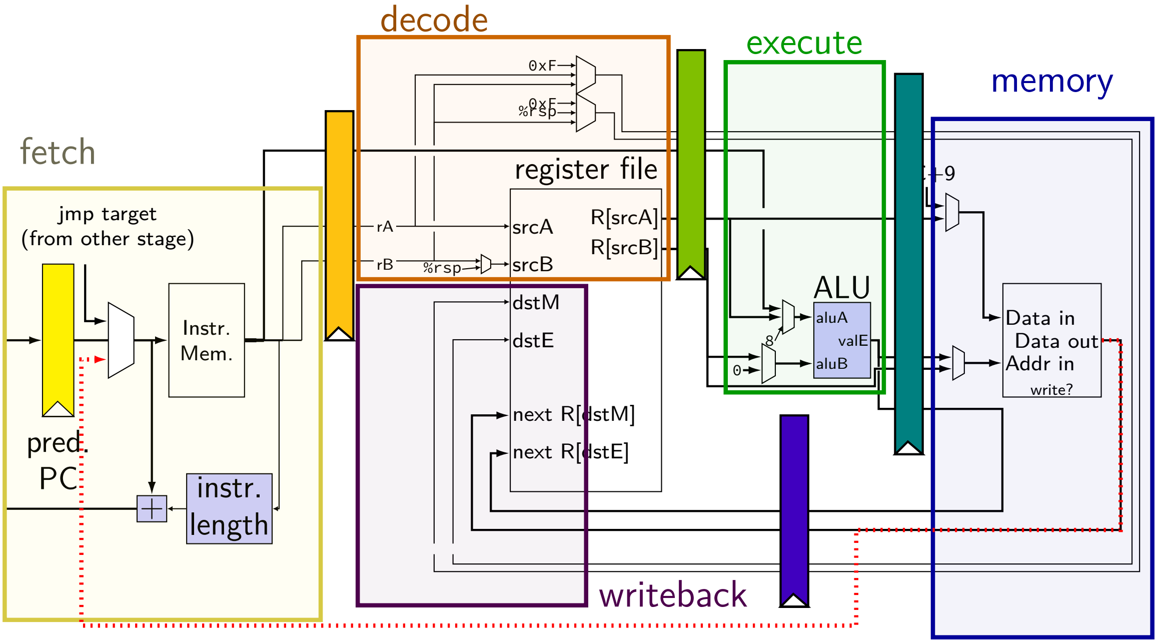
ret control hazard: 3 cycles of stalling

- (fetch next instruction after ret finishes memory)

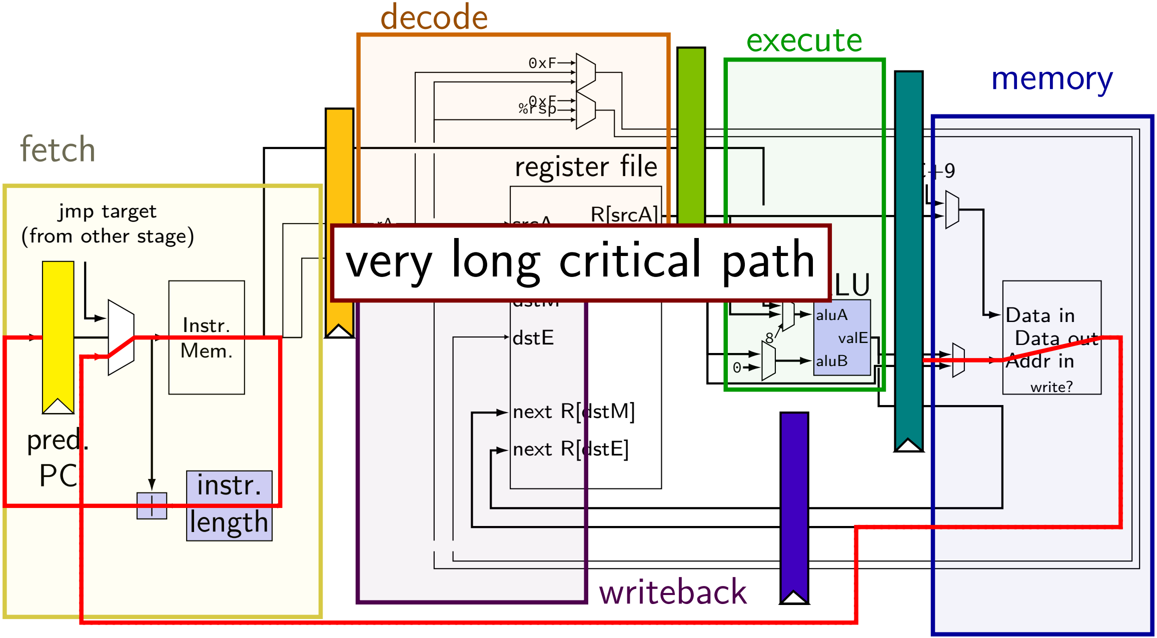
ret paths


decode
execute
memory
fetch
writeback
0xF
0xF
%rsp
register file
jmp target
(from other stage)
rA
rB
%rsp
srcA
srcB
R[srcA]
R[srcB]
+9
Instr.
Mem.
ALU
aluA
valE
aluB
8
0
Data in
Data out
Addr in
write?
dstM
dstE
next R[dstM]
next R[dstE]
pred.
PC
instr.
length

ret paths

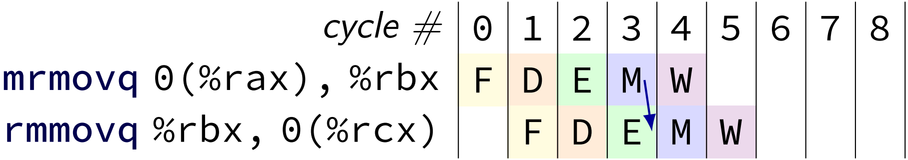

decode
execute
memory
fetch
0xF
0xF
%rsp
register file
jmp target
(from other stage)
rA
srcA
R[srcA]
rB
%rsp
srcB
R[srcB]
+9
Instr.
Mem.
dstM
dstE
ALU
aluA
valE
aluB
8
0
Data in
Data out
Addr in
write?
pred.
PC
next R[dstM]
next R[dstE]
instr.
length
writeback

ret paths



very long critical path

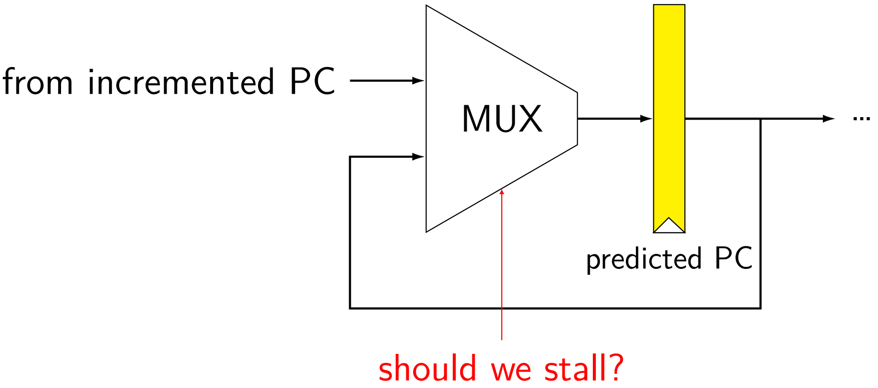
solveable problem



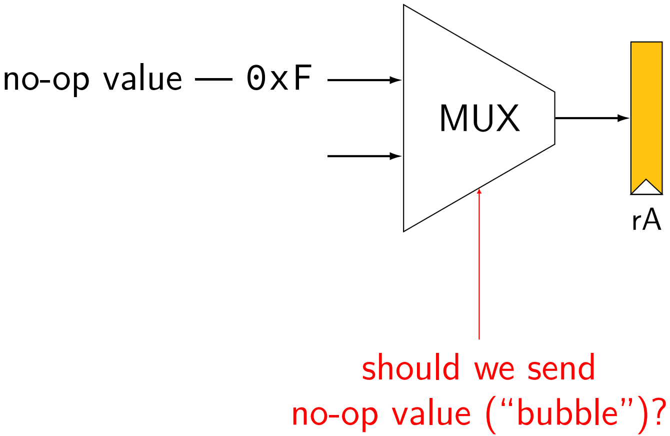
cycle #	0	1	2	3	4	5	6	7	8
mrmovq `0(%rax), %rbx`	F	D	E	M	W				
rmmovq `%rbx, 0(%rcx)`		F	D	E	M	W			

common for real processors to do this
but our textbook only forwards to the end of decode

fetch/fetch logic — advance or not



fetch/decode logic — bubble or not



no-op value — `0xF`

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

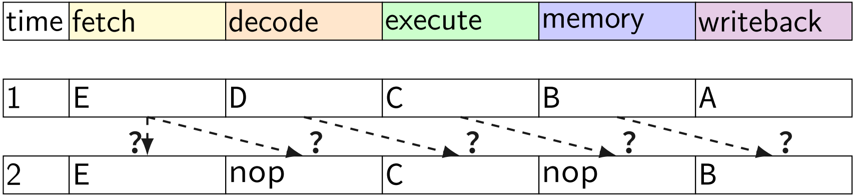
time	a_value	B_value	stall_B	bubble_B
0	0x01	0xFF	0	0
1	0x02	???	1	0
2	0x03	???	0	0
3	0x04	???	0	1
4	0x05	???	0	0
5	0x06	???	0	0
6	0x07	???	1	0
7	0x08	???	1	0
8		???		

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)



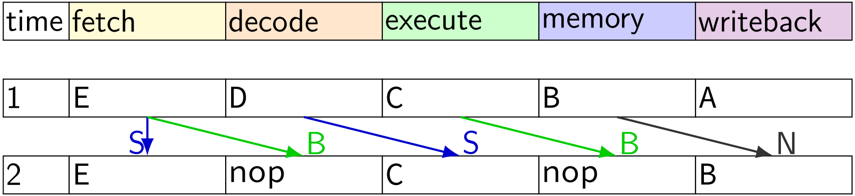
time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	E	nop	C	nop	B

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)

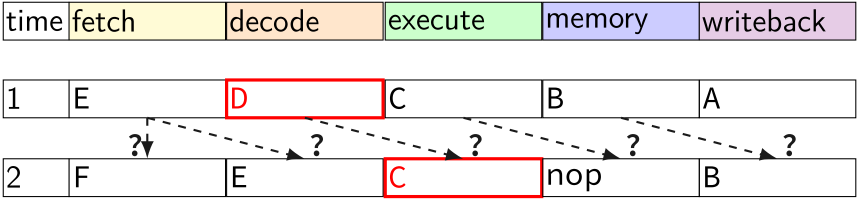


time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	E	nop	C	nop	B

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

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

exercise: squash + stall (2)

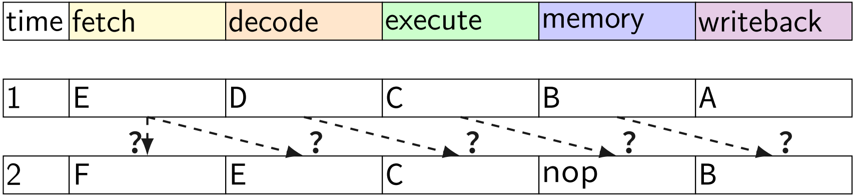


time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	F	E	C	nop	B

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

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

exercise: squash + stall (2)



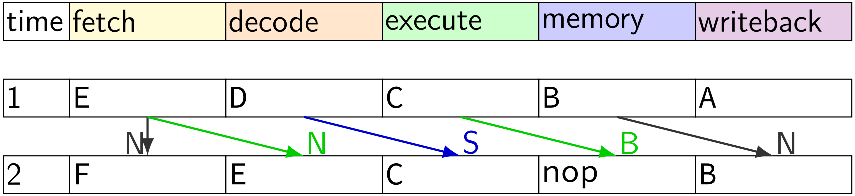
time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	F	E	C	nop	B

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)



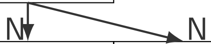
time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	F	E	C	nop	B

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

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

ret stall

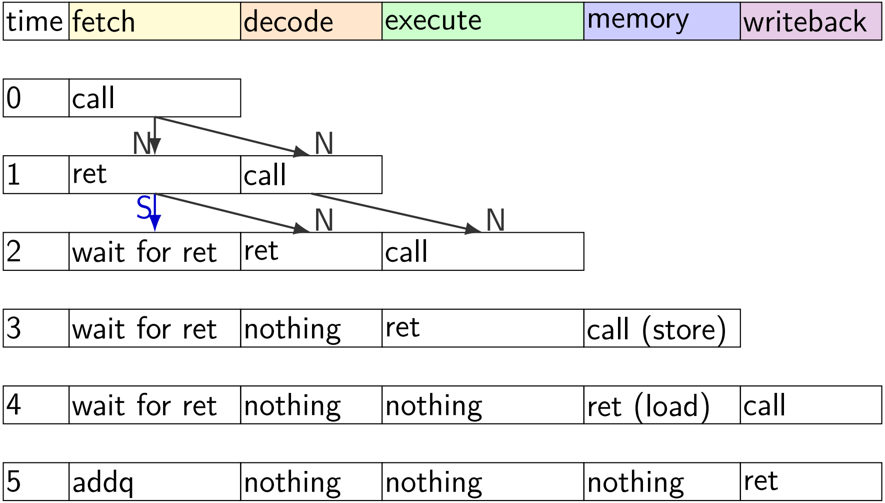
time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret



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

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

ret stall

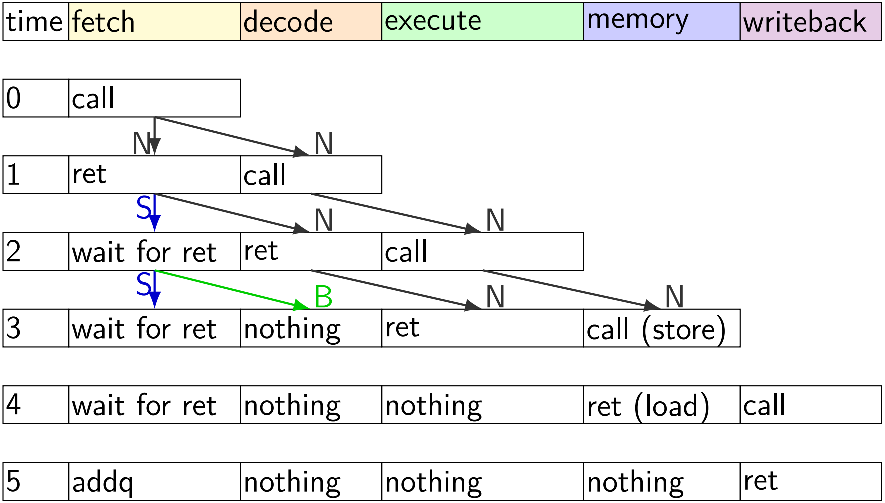


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

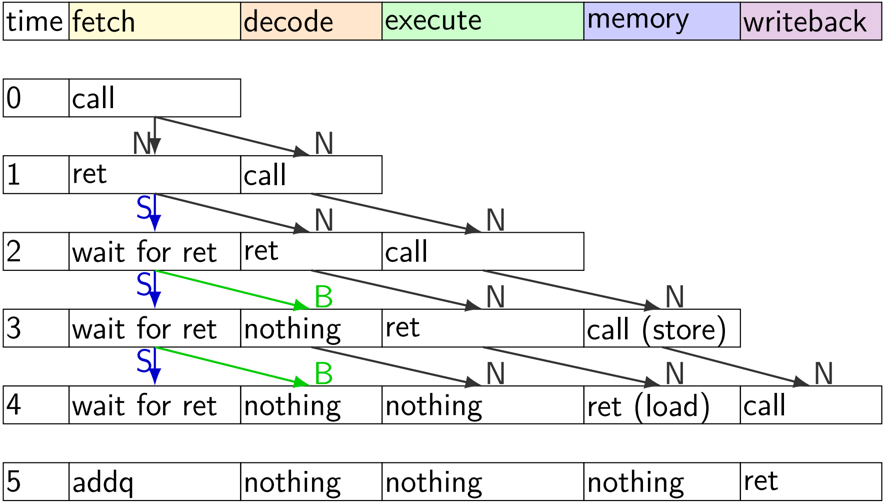


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

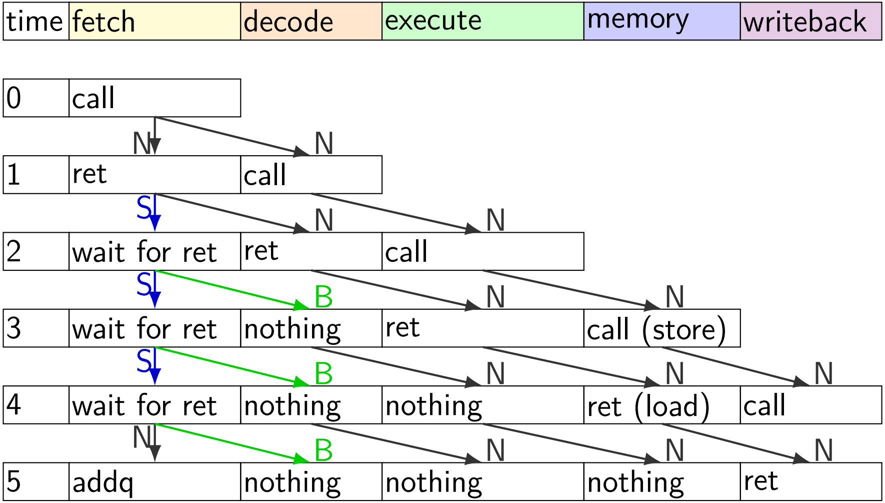


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall



time	fetch	decode	execute	memory	writeback
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

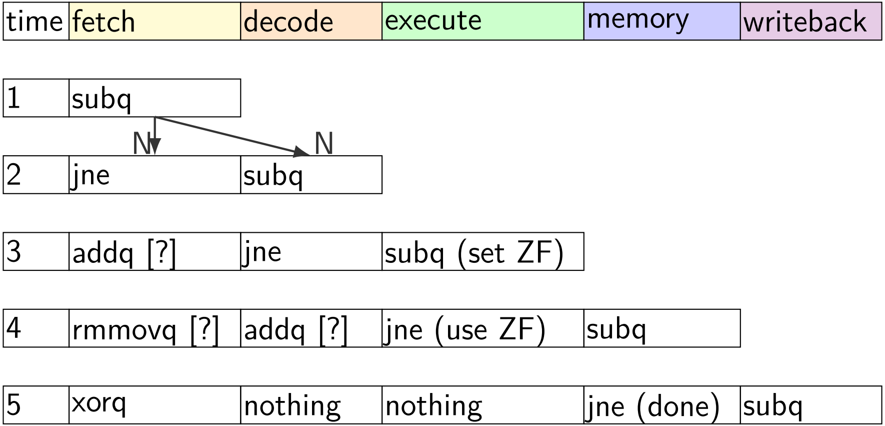
bubble (B) = use default (no-op);

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

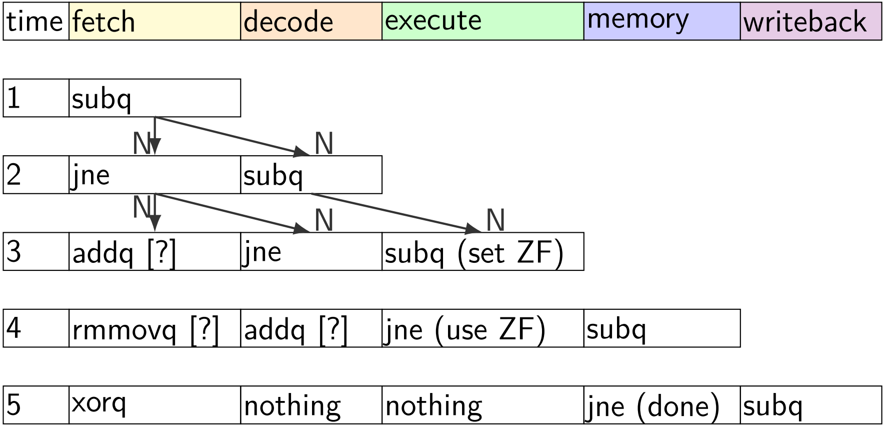


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

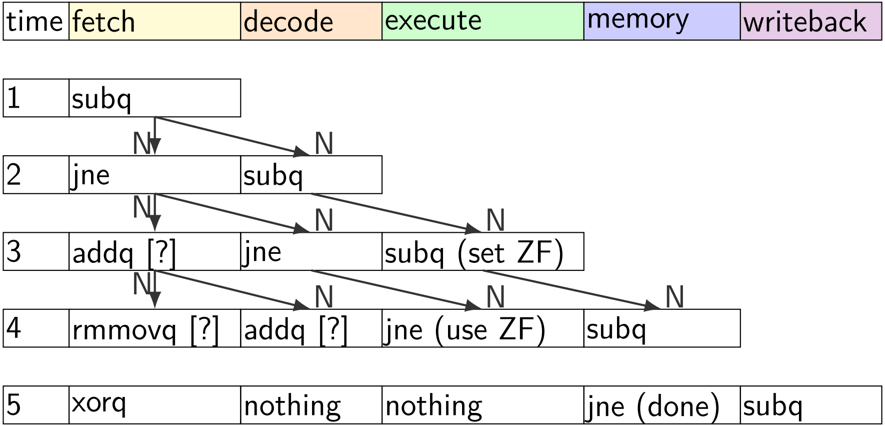


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

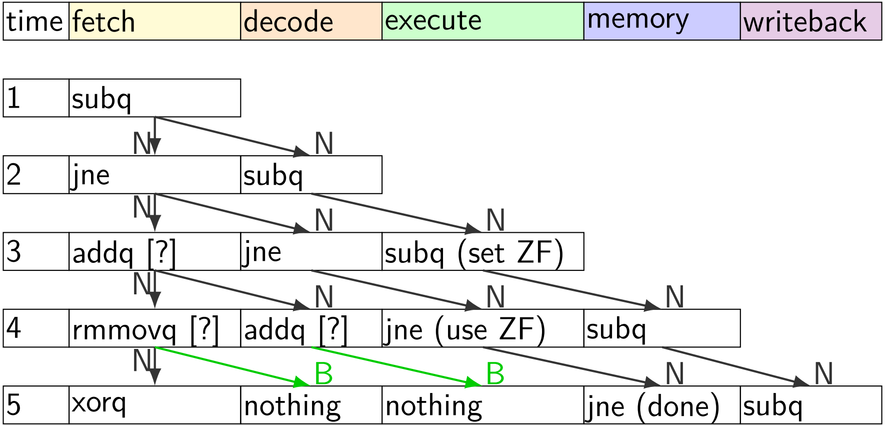


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

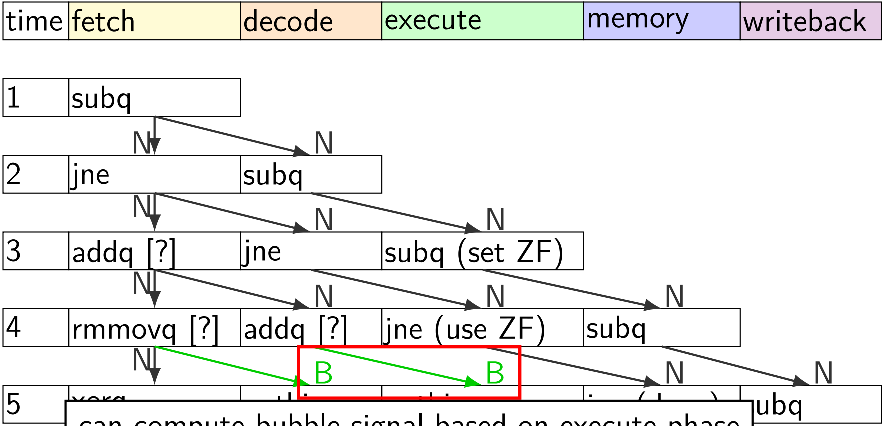


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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
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

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

stal ew 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 || ...;
```

better branch prediction

forward (target $>$ PC) not taken; backward taken

intuition: loops:

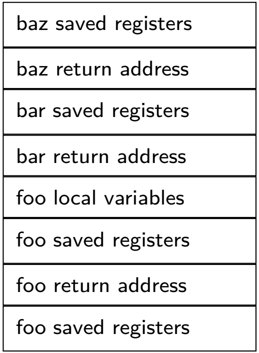
```
LOOP: ...
      ...
      je LOOP

LOOP: ...
      jne SKIP_LOOP
      ...
      jmp LOOP
SKIP_LOOP:
```

predicting ret: extra copy of stack

predicting ret — stack in processor registers

different than real stack/out of room? just slower



baz saved registers

baz return address

bar saved registers

bar return address

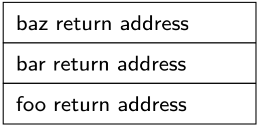
foo local variables

foo saved registers

foo return address

foo saved registers

stack in memory



baz return address

bar return address

foo return address

(partial?) stack
in CPU registers

prediction before fetch

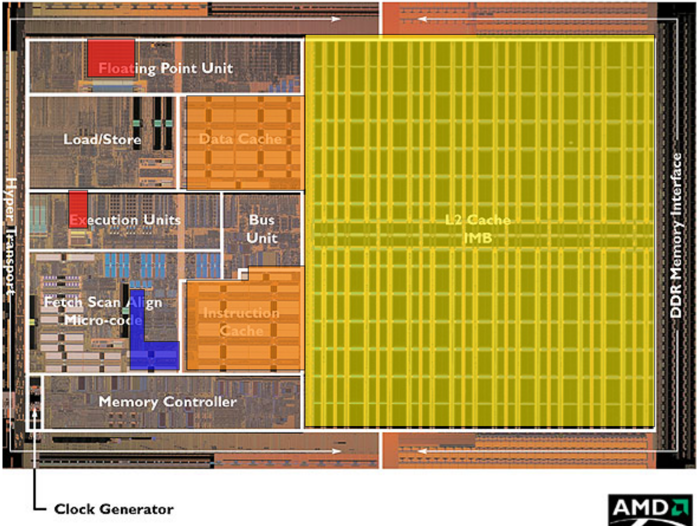
real processors can take multiple cycles to read instruction memory

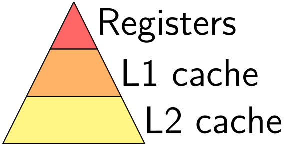
predict branches before reading their opcodes

how — more extra data structures

- tables of recent branches (often many kilobytes)

2004 CPU





🟦 Branch Prediction (approximate)

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

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

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

cycle #	0	1	2	3	4	5	6	7	8
`mrmovq 0(%rbx), %r8`	F	D	E	M	W				
`mrmovq 0(%rcx), %r9`		F	D	E	M	W			
`addq %r8, %r9`			F	D	D	E	M	W	

memory is slow: (cache "miss")

cycle #	0	1	2	3	4	5	6	7	8			
`mrmovq 0(%rbx), %r8`	F	D	E	M	M	M	M	M	W			
`mrmovq 0(%rcx), %r9`		F	D	E	E	E	E	E	M	M	M	M
`addq %r8, %r9`			F	D	D	D	D	D	D	D	D	D

missing pieces

multi-cycle memories

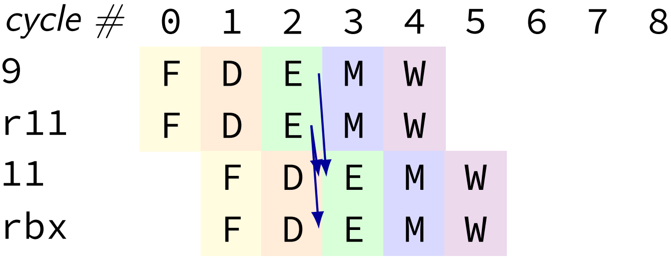
beyond pipelining: out-of-order, multiple issue

beyond pipelining: multiple issue

start more than one instruction/cycle

multiple parallel pipelines; many-input/output register file

hazard handling much more complex



cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r10, %r11`	F	D	E	M	W				
`xorq %r9, %r11`		F	D	E	M	W			
`subq %r10, %rbx`		F	D	E	M	W			

…

beyond pipelining: out-of-order

find later instructions to do instead of stalling

lists of available instructions in pipeline registers

- take any instruction with available values

provide illusion that work is still done in order

- much more complicated hazard handling logic

cycle #	0	1	2	3	4	5	6	7	8		
`mrmovq 0(%rbx), %r8`	F	D	E	M	M	M	W				
`subq %r8, %r9`		F					D	E	W		
`addq %r10, %r11`			F	D	E					W	
`xorq %r12, %r13`				F	D	E					W
…											

stalling/misprediction and latency

hazard handling where pipeline latency matters

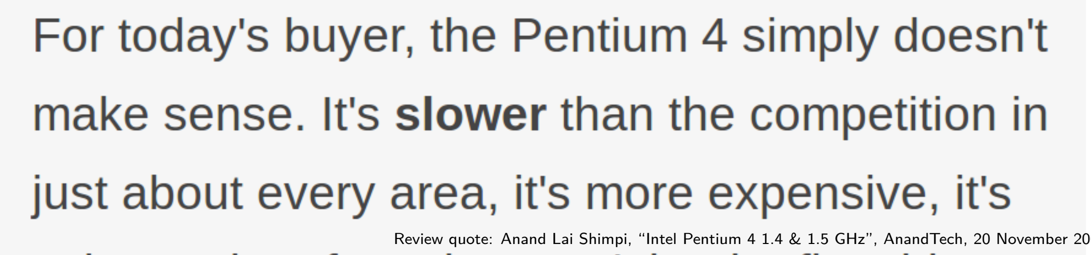
longer pipeline — larger penalty

part of Intel's Pentium 4 problem (c. 2000)

on release: 50% higher clock rate, 2-3x pipeline stages of competitors

out-of-order, multiple issue processor

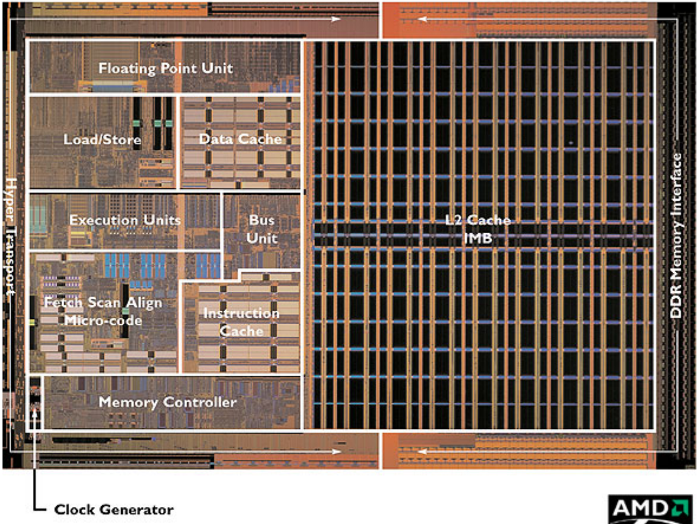
first-generation review quote:



For today's buyer, the Pentium 4 simply doesn't make sense. It's **slower** than the competition in just about every area, it's more expensive, it's

Review quote: Anand Lai Shimpi, "Intel Pentium 4 1.4 & 1.5 GHz", AnandTech, 20 November 2000

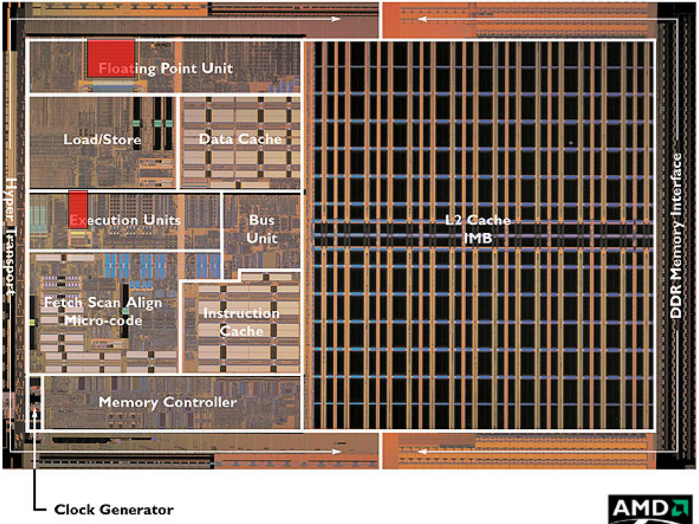
2004 CPU



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

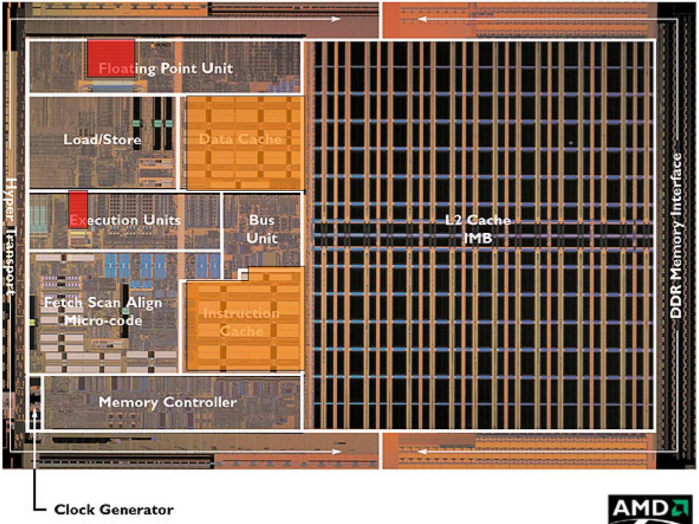
2004 CPU

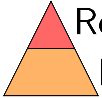
🔺 Registers



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

2004 CPU

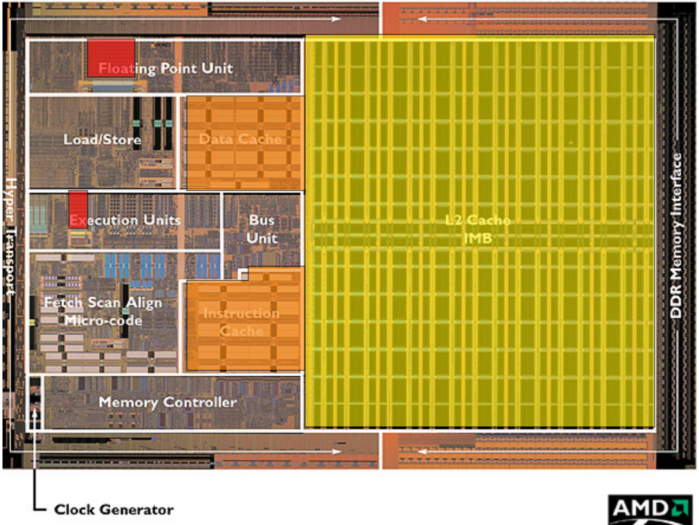


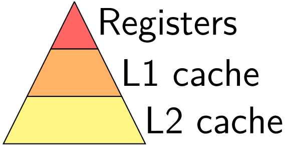

Registers

L1 cache

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

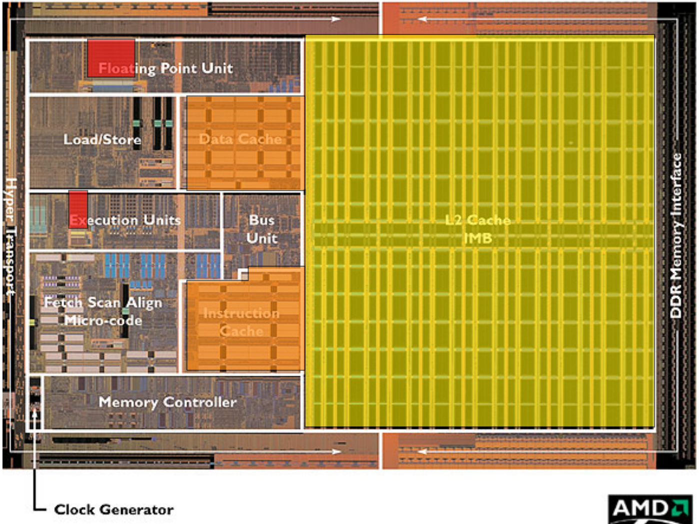
2004 CPU

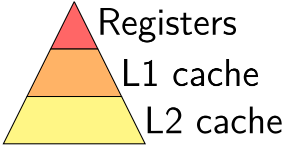




age: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



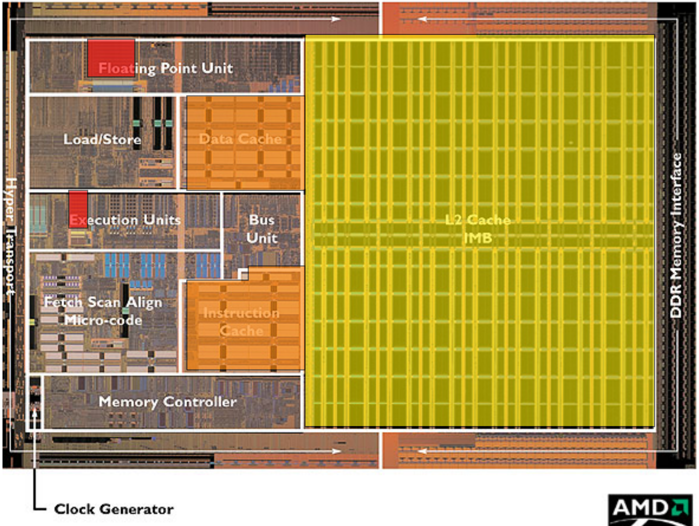

Registers

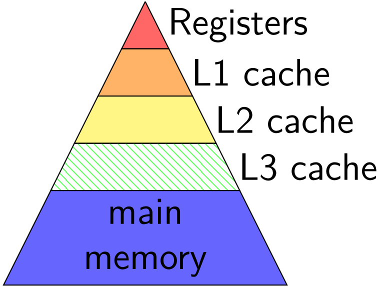
L1 cache

L2 cache

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

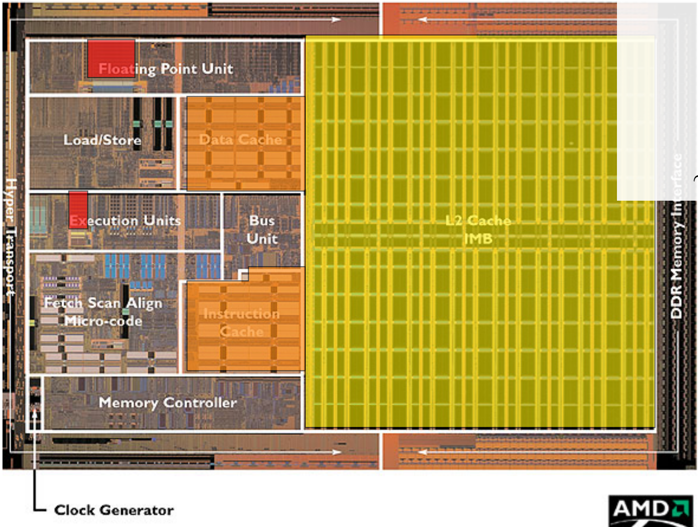
2004 CPU

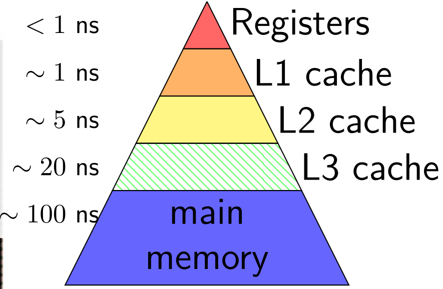



Registers
L1 cache
L2 cache
L3 cache
main memory

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

2004 CPU





< 1 ns Registers

~ 1 ns L1 cache

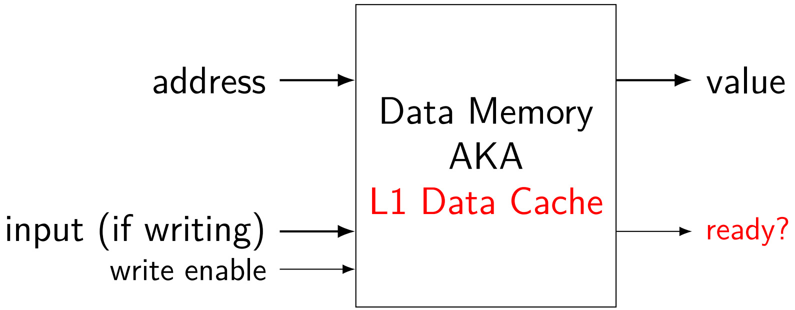
~ 5 ns L2 cache

~ 20 ns L3 cache

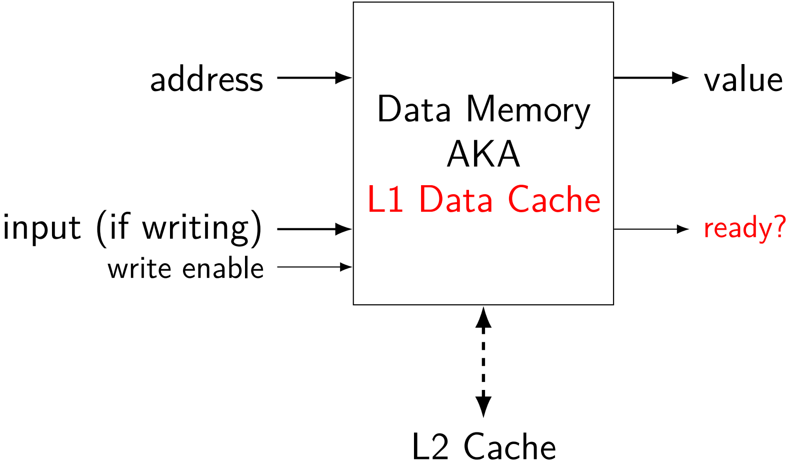
~ 100 ns main memory

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

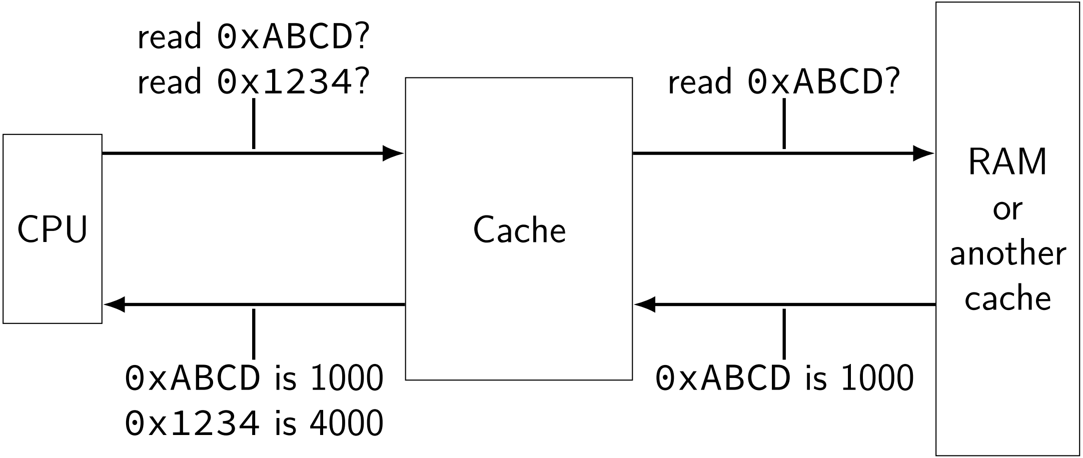
cache: real memory



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]`

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

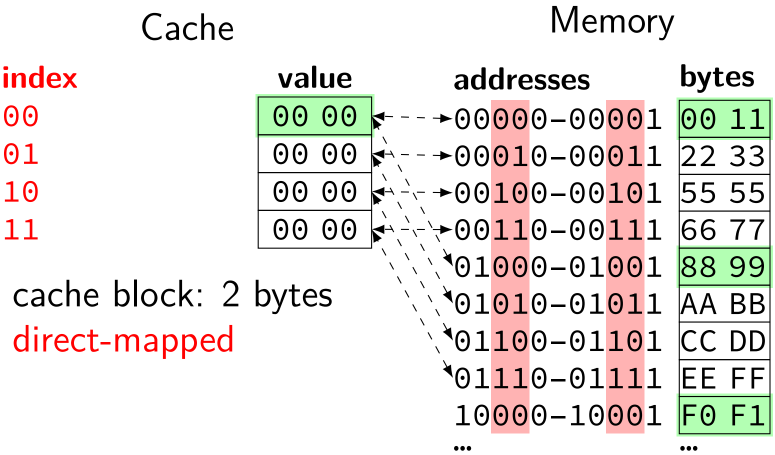
Memory

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

index	value
00	00 00
01	00 00
10	00 00
11	00 00

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
…	…

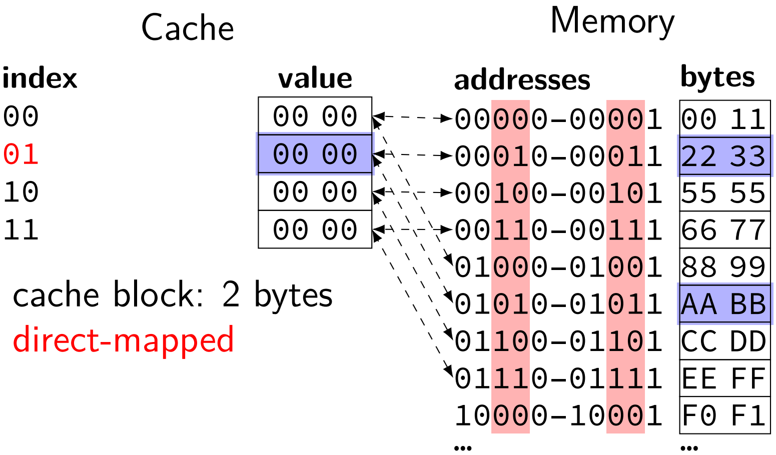
cache block: 2 bytes

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

index	value
00	00 00
01	00 00
10	00 00
11	00 00

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 block: 2 bytes

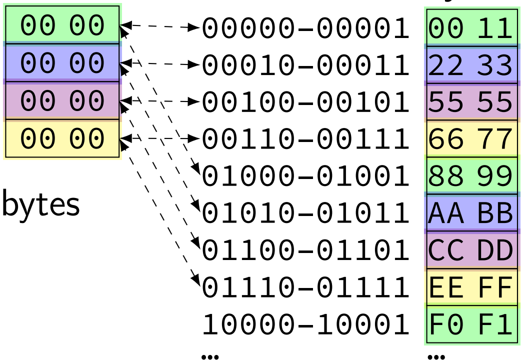
direct-mapped

building a (direct-mapped) cache

read byte at 01011?

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

Cache



index	value
00	00 00
01	00 00
10	00 00
11	00 00

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 block: 2 bytes

direct-mapped

building a (direct-mapped) cache

read byte at 01011?

Cache

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

cache block: 2 bytes

direct-mapped

Memory

addresses	bytes
	1
00010–00011	22 33
0–00101	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
...	...

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

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

need tag to know

cache block: 2 bytes
direct-mapped

Memory

value from 01010 or 00010?

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

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

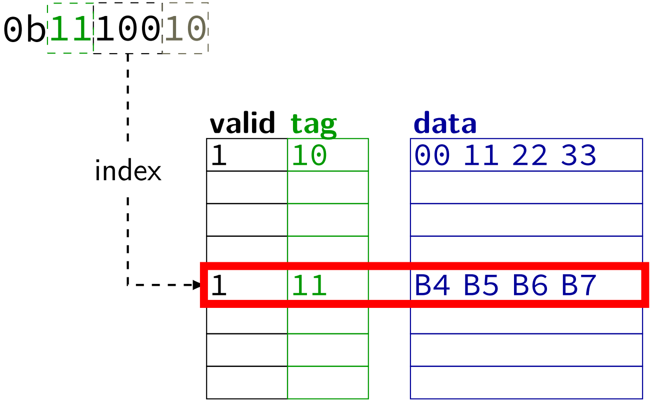
cache block: 2 bytes

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
...	...

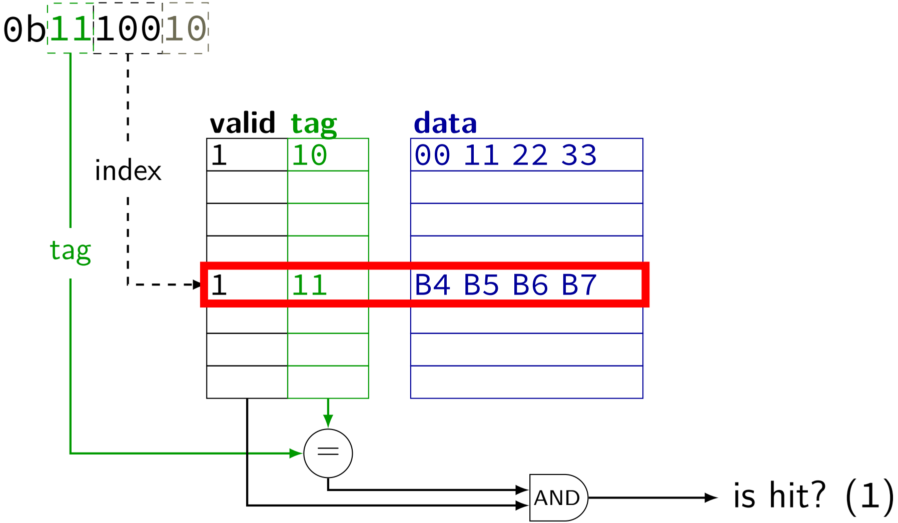
cache operation (read)



0b1110010

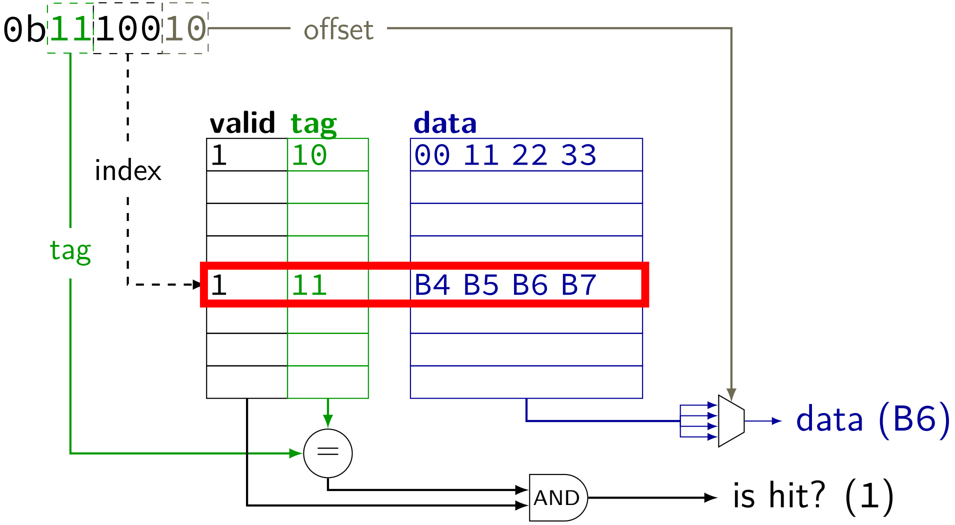
valid	tag	data
1	10	00 11 22 33
1	11	B4 B5 B6 B7

cache operation (read)



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

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

$2 = 2^1$ bytes in block

1 bit to say which byte

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
			0 11
			F1 F2
			-- --
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	???	???	11

2 byte blocks, 4 sets

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

$4 = 2^2$ bytes in block

2 bits to say which byte

4 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
	0	--	-- --
	0	--	-- --
	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

$2^2 = 4$ 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

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

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		

4

index	vali	tag	value
0	1	00	00 11 22 33
1	1	01	CC DD EE FF

$2^3 = 8$ sets

3 bits to index set

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	--	-- --
01			--
10			--
10			BB
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 addreses 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)

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

2 byte blocks, 4 sets

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

example access pattern (1)

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

2 byte blocks, 4 sets

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)

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

2 byte blocks, 4 sets

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)

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

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
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)

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

tag index offset

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
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)

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

tag index offset

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		

$m = 8$ 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)

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

tag index offset

2 byte blocks, 4 sets

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

$S = 4 = 2^s$ 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)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	
01100100 (64)	

tag index offset

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

$S = 4 = 2^s$ 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)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	miss
01100100 (64)	

tag index offset

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

$S = 4 = 2^s$ 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)

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

2 byte blocks, 4 sets

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

$S = 4 = 2^s$ 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)

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

2 byte blocks, 4 sets

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
$S = 4 = 2^s$ 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)

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

miss caused by conflict

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] 65]
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

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

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

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

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?

example access pattern (1)

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

2 byte blocks, 4 sets

index	valid	tag	value
00	1	00000	mem[0x00] mem[0x01]
01	1	01100	mem[0x62] mem[0x63]
10	1	01100	mem[0x64] 65]
11	0		

miss caused by conflict

$m = 8$ 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

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 addreses 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 addreses 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)	$B = 64$Byte
	$B = 2^b$ (b: block offset bits)

T-I-O exercise: L1

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

T-I-O exercise: L1

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

T-I-O exercise: L1

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

	3	4	5	6	7
`0x34567`:	`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)		

	3	4	5	6	7
`0x34567`:	`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)	

	3	4	5	6	7
`0x34567`:	`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)		

	3	4	5	6	7
0x34567:	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)	

	3	4	5	6	7
0x34567:	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)		

	3	4	5	6	7
`0x34567`:	`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)		

	3	4	5	6	7
0x34567:	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

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-associtive (1-set) cache of the same size would have done better

capacity — cache was not big enough