pipelining 4 / caching 0

last time

what stalling for jXX, ret would look like

branch prediction

- guess where conditional jump will go
- if wrong — undo guessed instructions
- trick: undo (“squash”) by clearing pipeline registers

logic for stalling/squashing

- stall_X/write disable on pipeline regs: keep old stage
- bubble_X/reset on pipeline regs: reset to default (nop)
- check icodes in pipeline to decide when to stall/forward

stalling involves both stall_X and bubble_X

- repeat decode? keep PC same AND stall_D AND bubble_E

implementing stalling + prediction

need to handle updating PC:

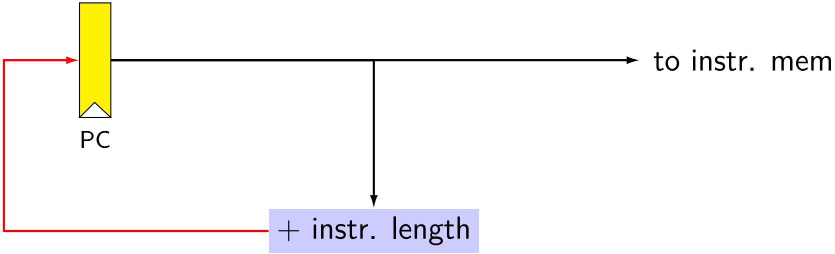
- stalling: retry same PC
- prediction: use predicted PC
- misprediction: correct mispredicted PC

need to updating pipeline registers:

- repeat stage in stall: keep same values
- don't go to next stage in stall: insert nop values
- ignore instructions from misprediction: insert nop values

building the PC update (one possibility)

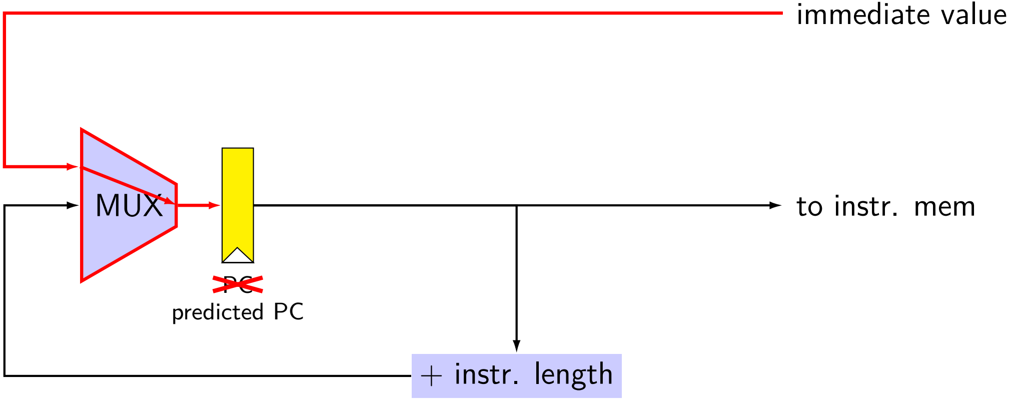
(1) normal case: PC ← PC + instr len



building the PC update (one possibility)

(1) normal case: PC ← PC + instr len

(2) immediate: call/jmp, and *prediction* for cond. jumps

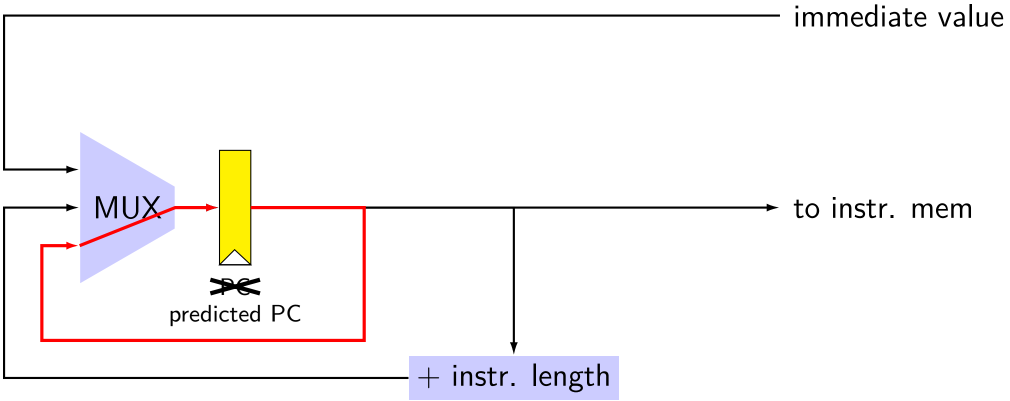


building the PC update (one possibility)

(1) normal case: PC ← PC + instr len

(2) immediate: call/jmp, and *prediction* for cond. jumps

(3) repeat previous PC for stalls (load/use hazard, halt, ret?)



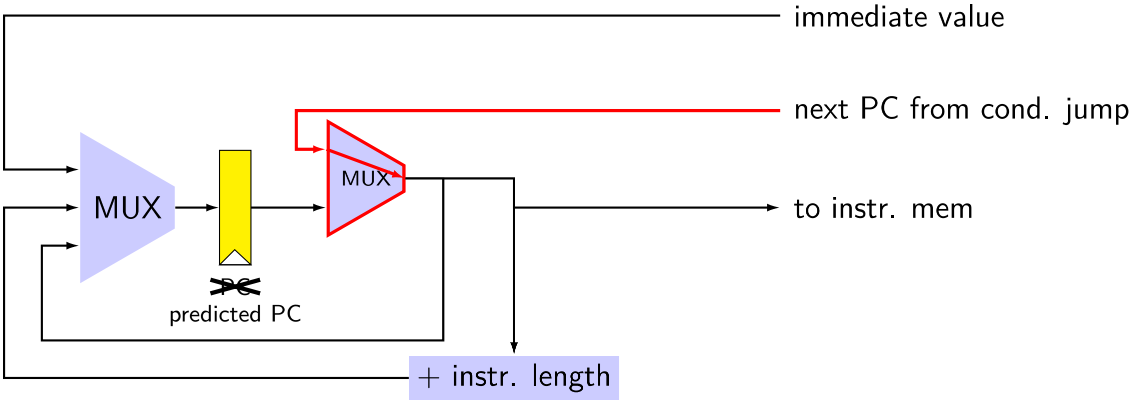
building the PC update (one possibility)

(1) normal case: PC ← PC + instr len

(2) immediate: call/jmp, and *prediction* for cond. jumps

(3) repeat previous PC for stalls (load/use hazard, halt, ret?)

(4) correct for misprediction of conditional jump



building the PC update (one possibility)

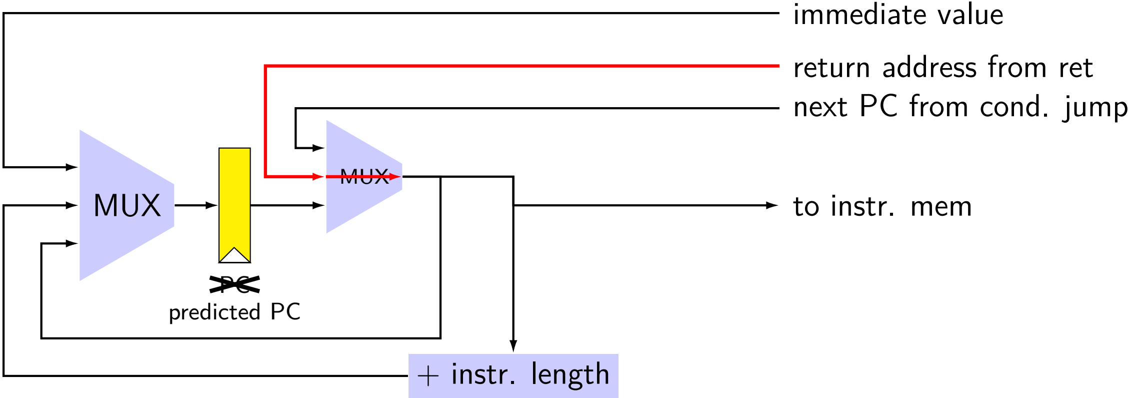
(1) normal case: PC ← PC + instr len

(2) immediate: call/jmp, and *prediction* for cond. jumps

(3) repeat previous PC for stalls (load/use hazard, halt, ret?)

(4) correct for misprediction of conditional jump

(5) correct for missing return address for `ret`



building the PC update (one possibility)

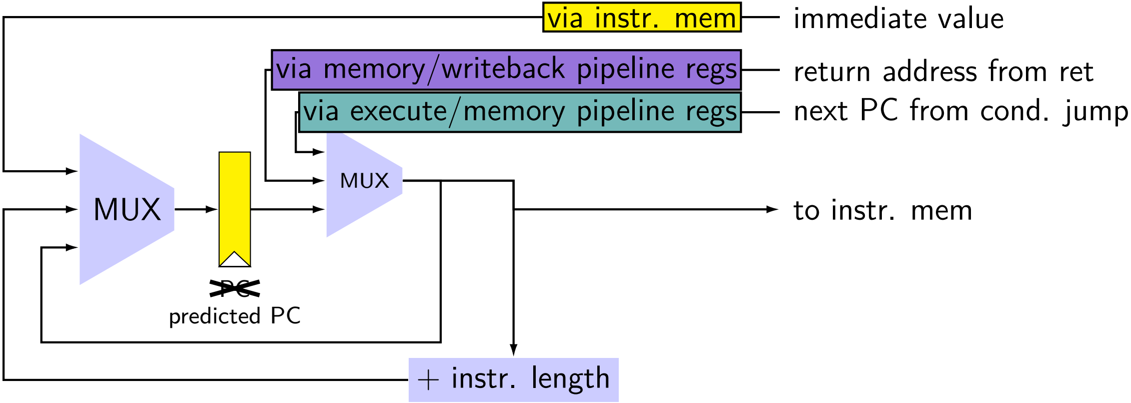
(1) normal case: PC ← PC + instr len

(2) immediate: call/jmp, and *prediction* for cond. jumps

(3) repeat previous PC for stalls (load/use hazard, halt, ret?)

(4) correct for misprediction of conditional jump

(5) correct for missing return address for `ret`



via instr. mem — immediate value

via memory/writeback pipeline regs — return address from ret

via execute/memory pipeline regs — next PC from cond. jump

MUX

MUX

~~PC~~

predicted PC

to instr. mem

+ instr. length

PC update overview

predict based on instruction length + immediate

override prediction with stalling sometimes

correct when prediction is wrong just before fetching

- retrieve corrections from pipeline register outputs for jCC/ret instruction

above is what textbook does

alternative: could instead correct prediction just before setting PC register

- retrieve corrections into PC cycle before corrections used
- moves logic from beginning-of-fetch to end-of-previous-fetch

I think this is more intuitive, but consistency with textbook is less confusing…

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)

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

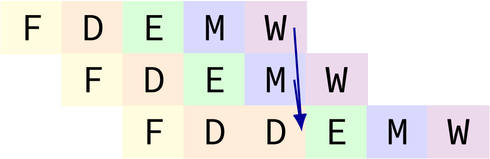
typical performance:

- 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

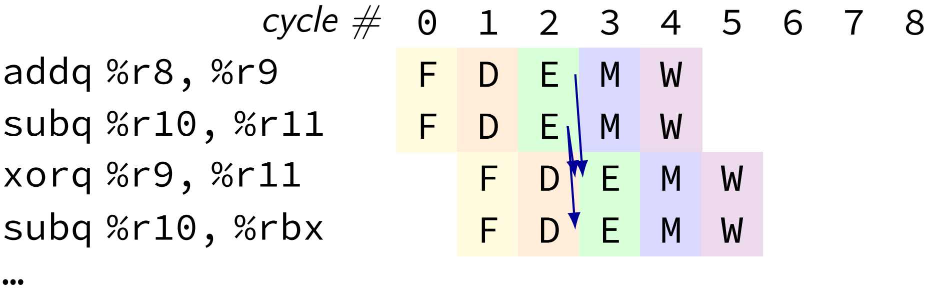
beyond pipelining: out-of-order, multiple issue

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	9	10	11
`mrmovq 0(%rbx), %r8`	F	D	E	M	M	M	W	C				
`subq %r8, %r9`		F					D	E	W	C		
`addq %r10, %r11`			F	D	E	W					C	
`xorq %r12, %r13`				F	D	E	W					C
…												

stalling/misprediction and latency

hazard handling where pipeline latency matters

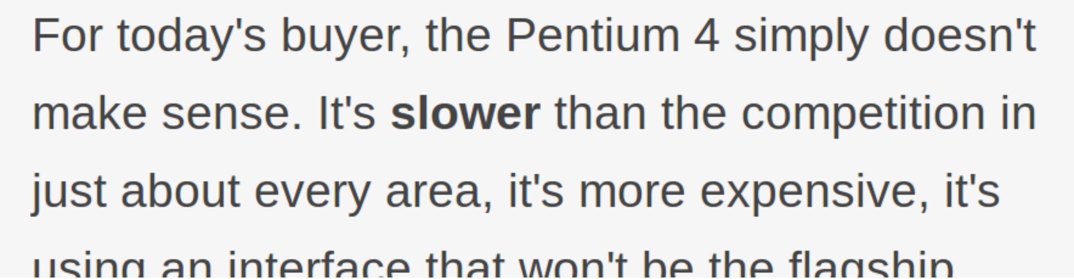
longer pipeline — larger penalty

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 using an interface that won't be the flagship

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

recall: data/instruction memory

model in CPU: one cycle per access

but earlier — had to talk to memory on different chip

can't do that in one cycle

solution: keep copies of part of memory ("cache")

- copy can be accessed quickly
- hope: almost always use copy?

2004 CPU

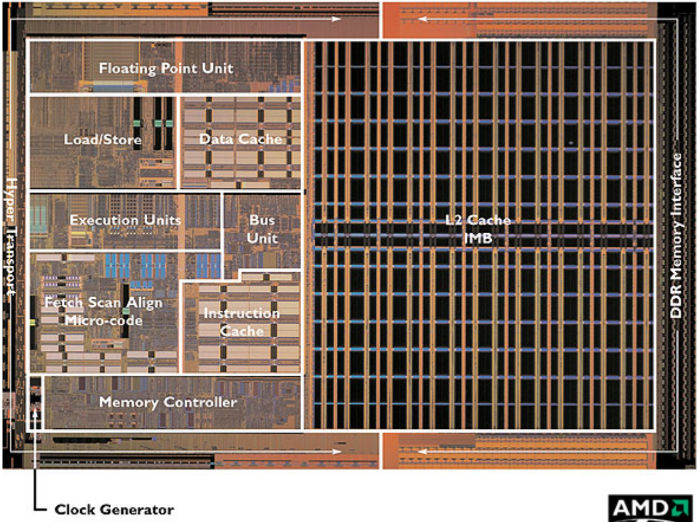


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

▲ Registers

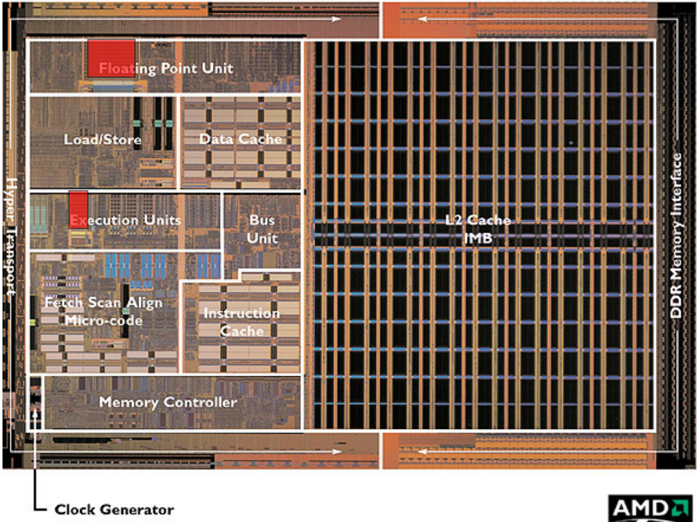
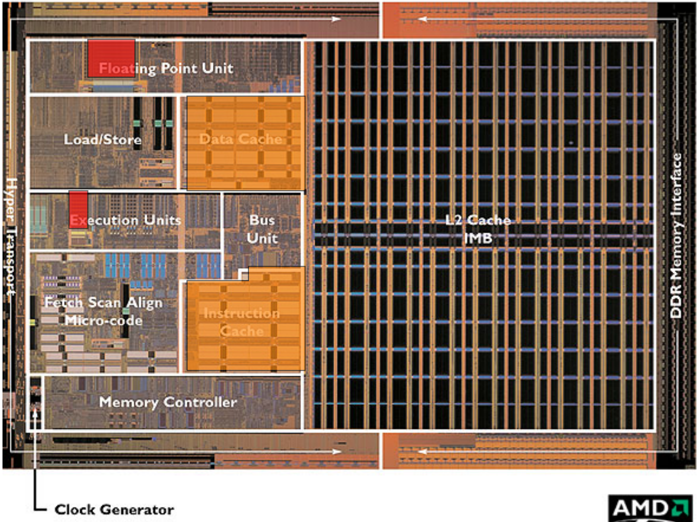
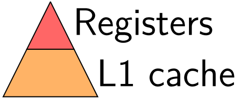


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



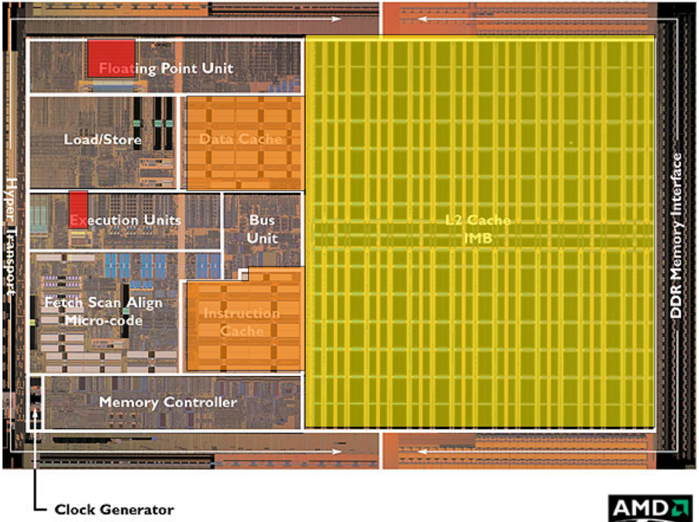


Registers

L1 cache

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

2004 CPU



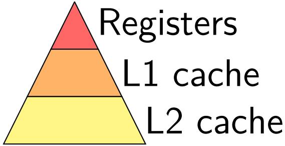
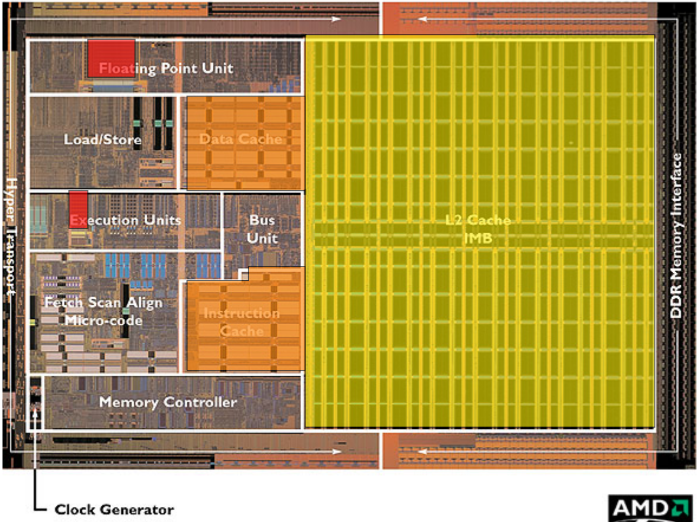


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



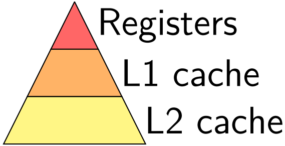
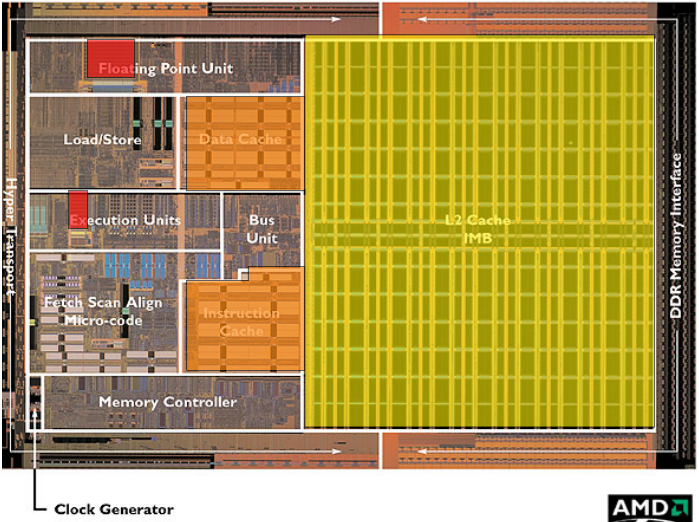


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



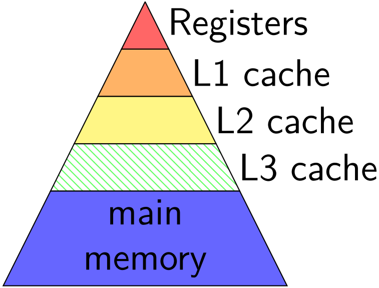
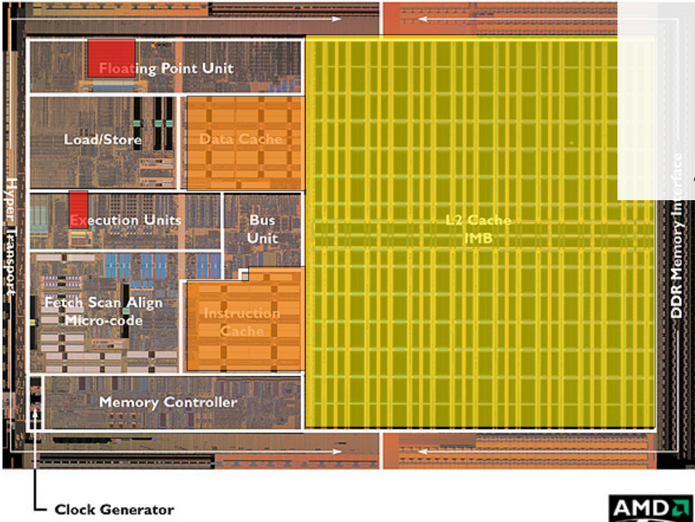


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



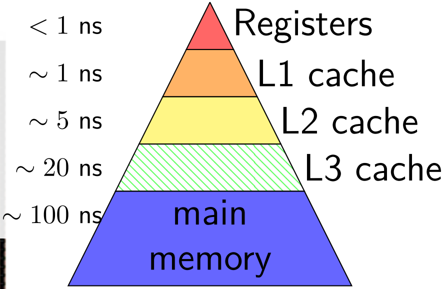
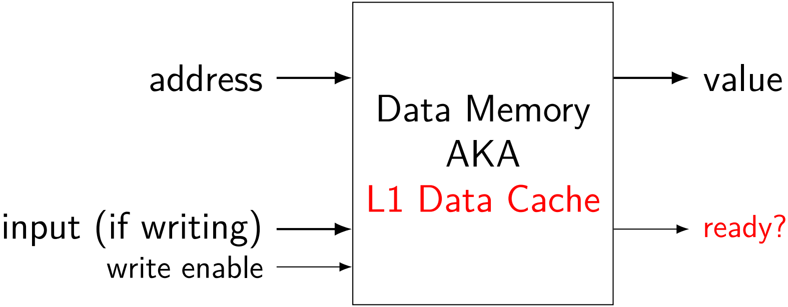
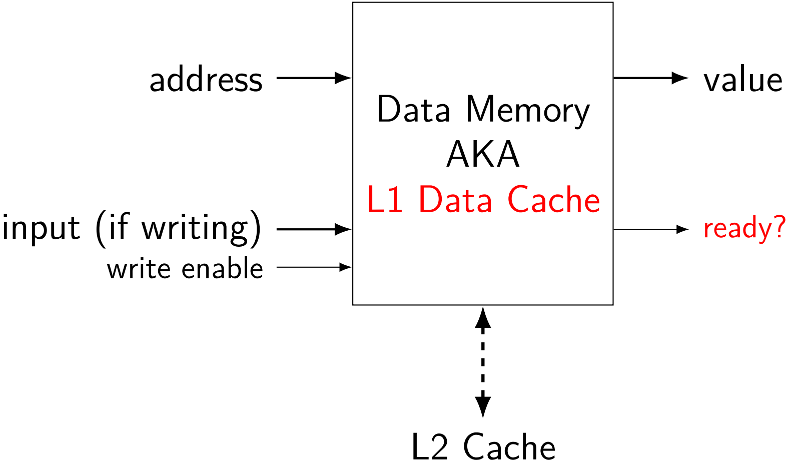


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

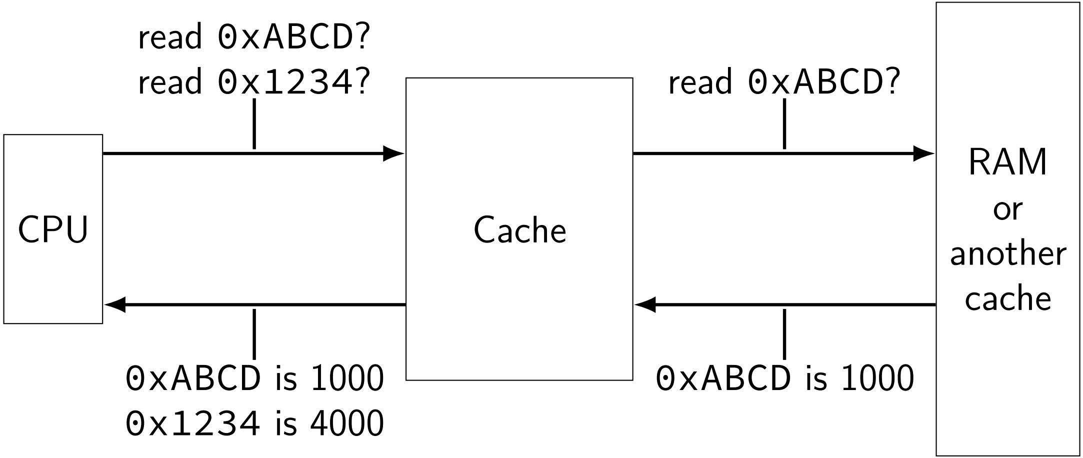
cache: real memory



cache: real memory



the place of cache


CPU

read 0xABCD?
read 0x1234?

0xABCD is 1000
0x1234 is 4000

Cache

read 0xABCD?

0xABCD is 1000

RAM
or
another
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

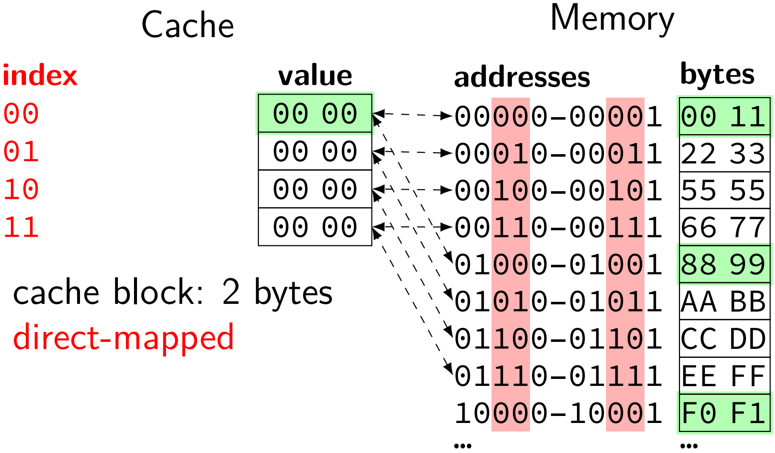
Memory

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
…	…

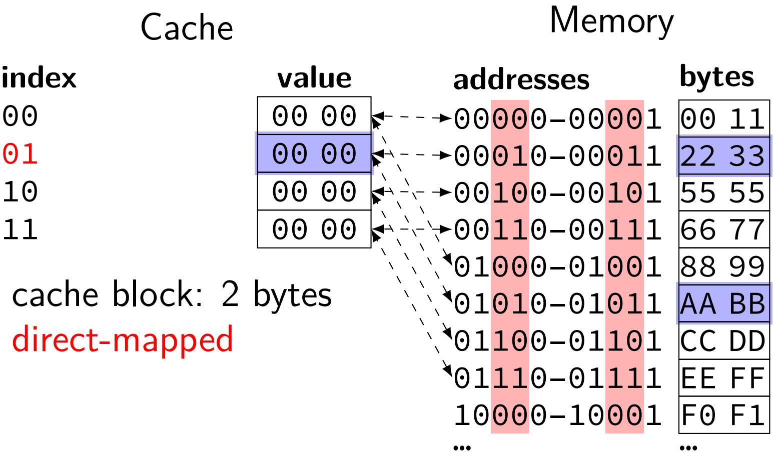
cache block: 2 bytes

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
…	…

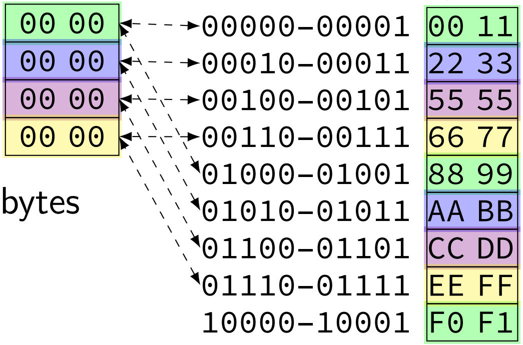
cache block: 2 bytes

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

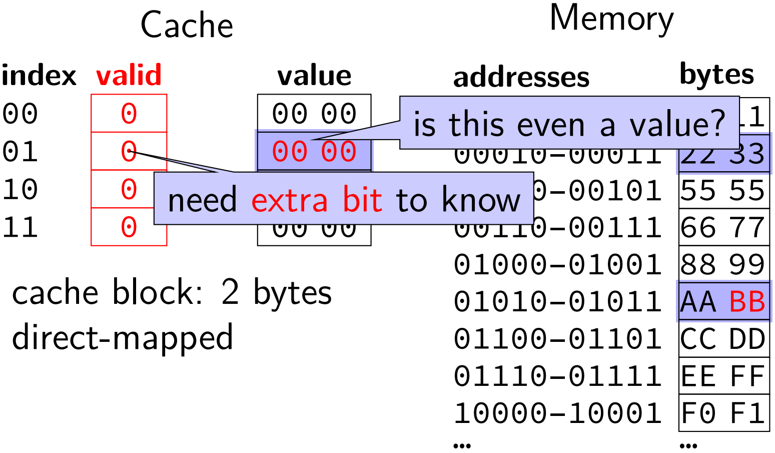
addresses	bytes
00000–00001	00 11
00010–00011	22 33
00100–00101	55 55
00110–00111	66 77
01000–01001	88 99
01010–01011	AA BB
01100–01101	CC DD
01110–01111	EE FF
10000–10001	F0 F1
…	…

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

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

cache block: 2 bytes

direct-mapped

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

value from 01010 or 00010?

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?

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

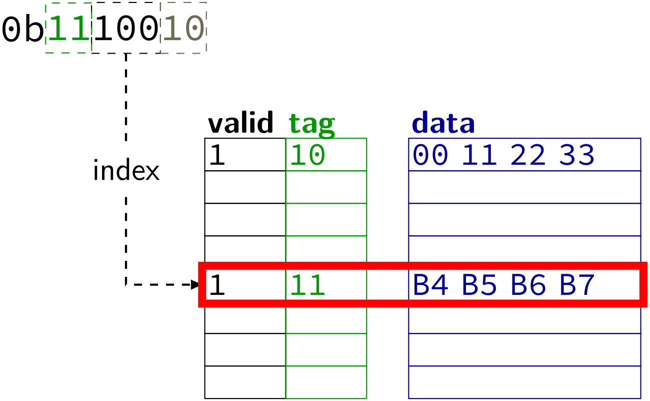
cache block: 2 bytes

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
...	...

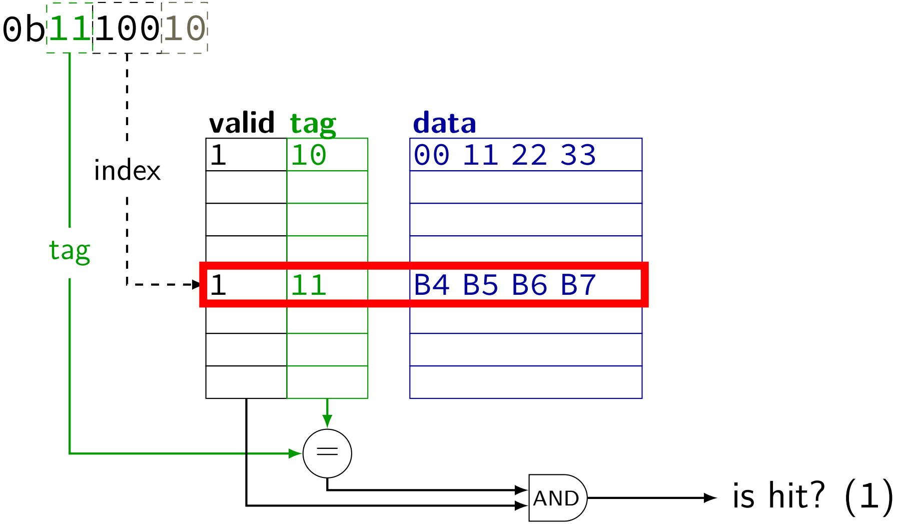
cache operation (read)



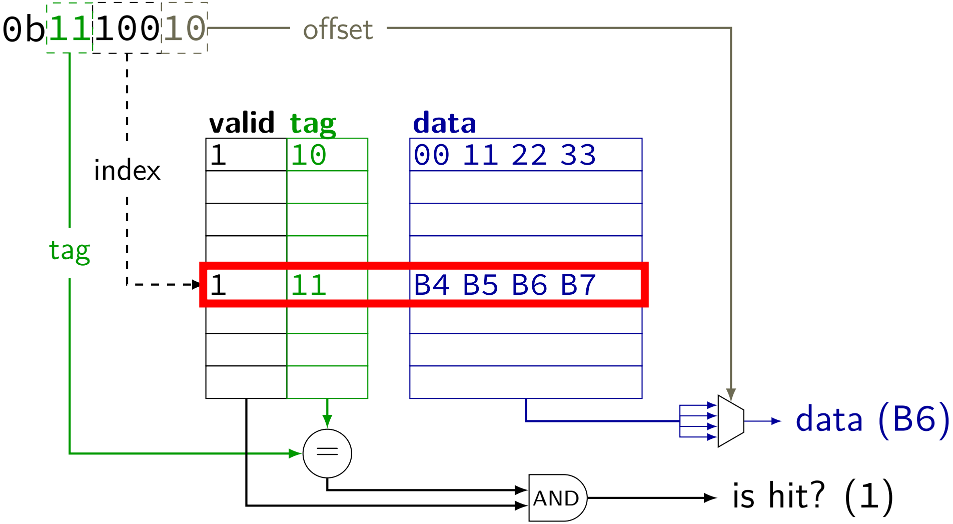
`0b11100 10`

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

cache operation (read)



cache operation (read)



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

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

$2 = 2^1$ bytes in block

1 bit to say which byte

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
			00 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	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	--	-- --
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	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
			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	000	00 11 22 33
1	1	001	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	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	--	-- --
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	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 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)

TIO: exercise

64-byte blocks, 128 set cache

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

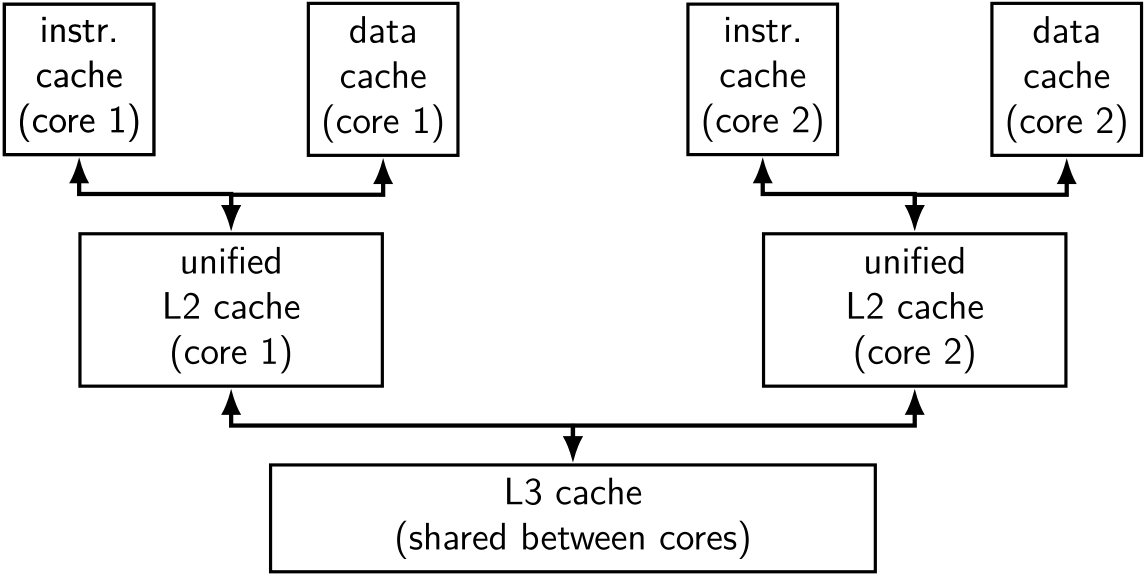
if addresses 32-bits, then how many tag/index/offset bits?

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`

backup slides

split caches; multiple cores



instr. cache (core 1)

data cache (core 1)

instr. cache (core 2)

data cache (core 2)

unified L2 cache (core 1)

unified L2 cache (core 2)

L3 cache (shared between 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

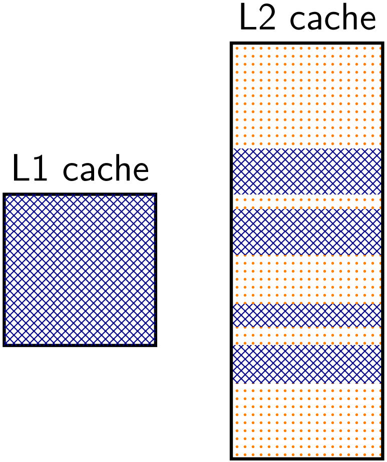
can optimize instruction cache for different access pattern

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

inclusive versus exclusive

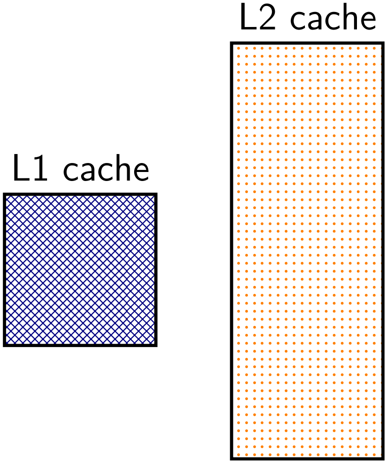
L2 inclusive of L1

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



L2 exclusive of L1

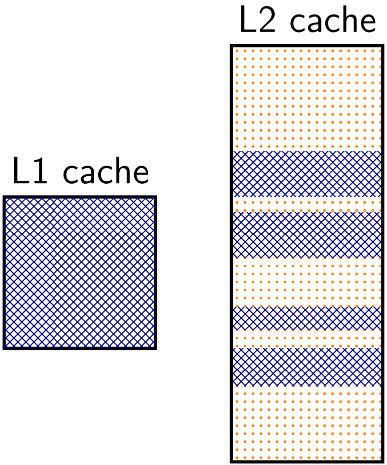
L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2



inclusive versus exclusive

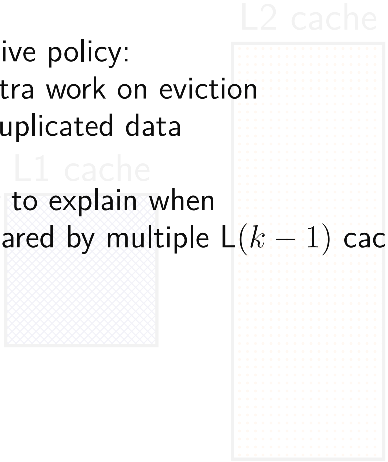
L2 inclusive of L1

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



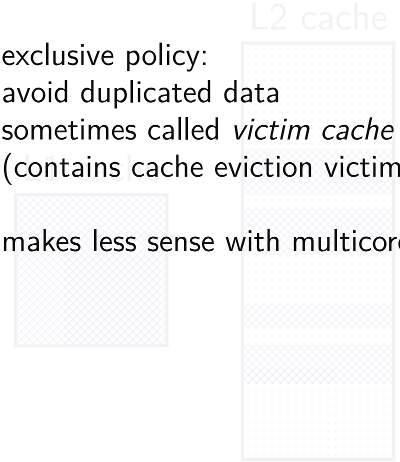
inclusive policy:
no extra work on eviction
but duplicated data

easier to explain when
Lk 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

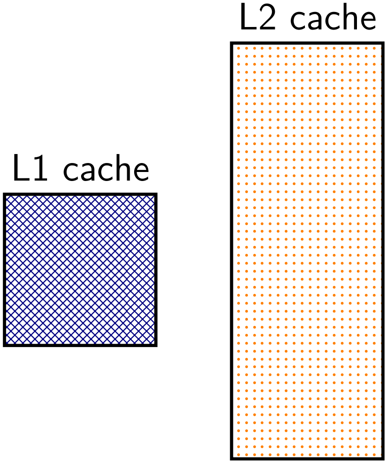


exclusive policy:
avoid duplicated data
sometimes called *victim cache*
(contains cache eviction victims)

makes less sense with multicore

L2 exclusive of L1

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



stalling: bubbles + stall

cycle #	0	1	2	3	4	5	6	7	8
`mrmovq 0(%rax), %rbx`	F	D	E	M	W				
`subq %rbx, %rcx`		F	D	D	E	M	W		
inserted nop				E	M	W			
`irmovq $10, %rbx`			F	F	D	E	M	W	
…									

need way to keep pipeline register unchanged to repeat a stage

(*and* to replace instruction with a nop)

stalling: bubbles + stall

cycle #	0	1	2	3	4	5	6	7	8
`mrmovq 0(%rax), %rbx`	F	D	E	M	W				
`subq %rbx, %rcx`		F	D	D	E	M	W		
`inserted nop`				E	M	W			
`irmovq $10, %rbx`			F	F	D	E	M	W	
...									

keep same instruction in cycle 3
during cycle 2:
stall_D = 1
stall_F = 1 or extra f_pc MUX

need way to keep pipeline register unchanged to repeat a stage

(*and* to replace instruction with a nop)

stalling: bubbles + stall

cycle #	0	1	2	3	4	5	6	7	8
`mrmovq` `0(%rax), %rbx`	F	D	E	M	W				
`subq` `%rbx, %rcx`		F	D	D	E	M	W		
`inserted nop`				E	M	W			
`irmovq` `$10, %rbx`			F	F	D	E	M	W	
…									

insert nop in cycle 3
during cycle 2:
bubble_E = 1

need way to keep pipeline register unchanged to repeat a stage

(*and* to replace instruction with a nop)

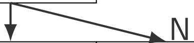
jump misprediction: bubbles

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`jle target` (not taken)		F	D	E	M	W			
`target: xorq %rax, %rax` (mispredicted)			F	D	–				
inserted nop					E	M	W		
`andq %rbx, %rcx` (mispredicted)				F	–				
inserted nop					D	E	M	W	
`subq %r9, %r10` (instr. after jle)					F	D	E	M	W

need option: replace instruction with nop ("bubble")

squashing with stall/bubble

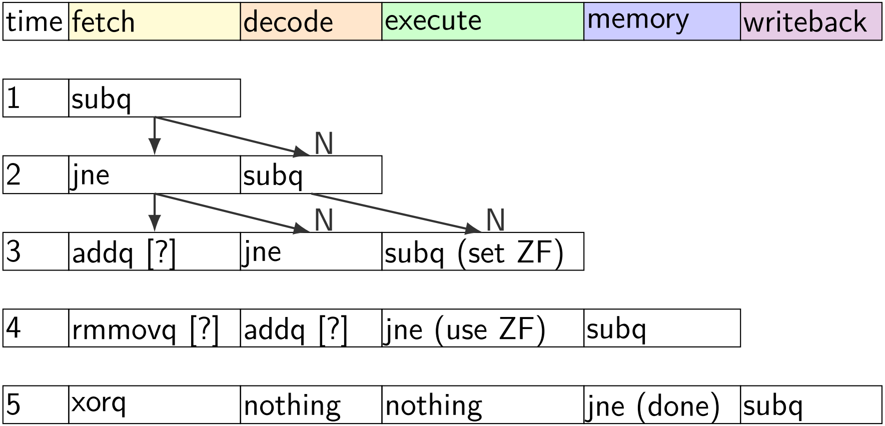
time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq



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

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

squashing with stall/bubble

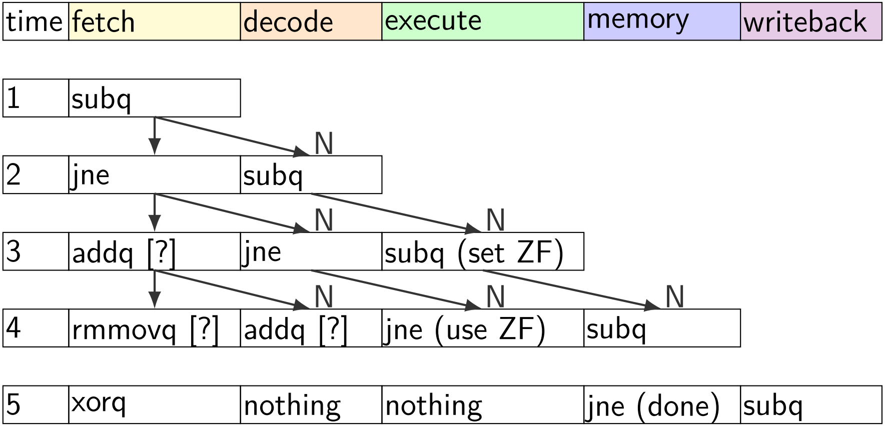


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

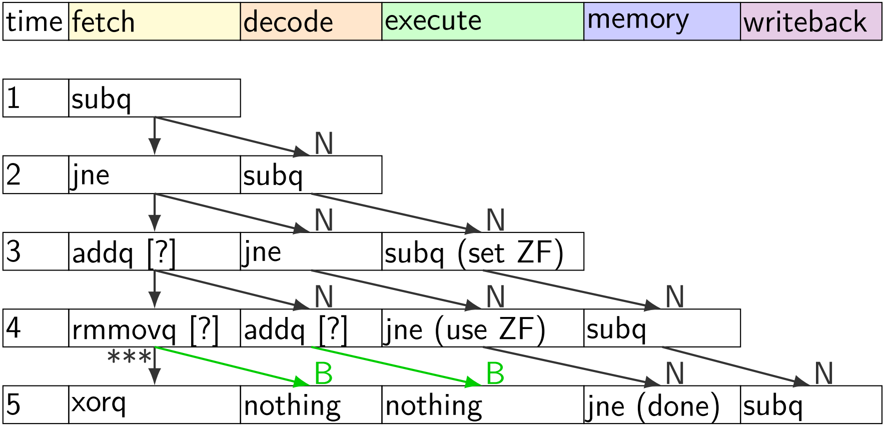


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

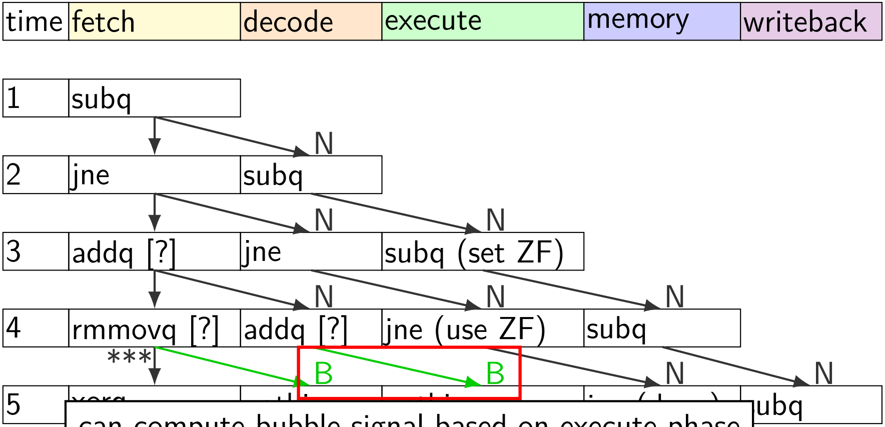


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

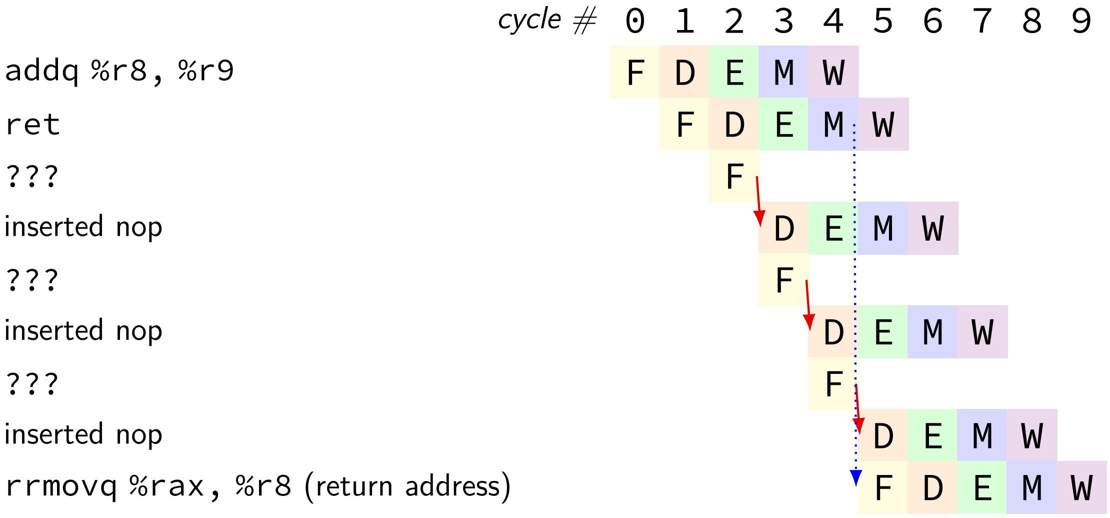
stal ew value

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

squashing HCLRS

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

ret bubbles

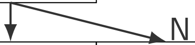


	cycle # 0	1	2	3	4	5	6	7	8	9
`addq %r8, %r9`	F	D	E	M	W					
`ret`		F	D	E	M	W				
`???`			F							
inserted nop				D	E	M	W			
`???`				F						
inserted nop					D	E	M	W		
`???`					F					
inserted nop						D	E	M	W	
`rrmovq %rax, %r8` (return address)						F	D	E	M	W

need option: replace instruction with nop (“bubble”)

ret stall

time	fetch	decode	execute	memory	writeback
0	call				

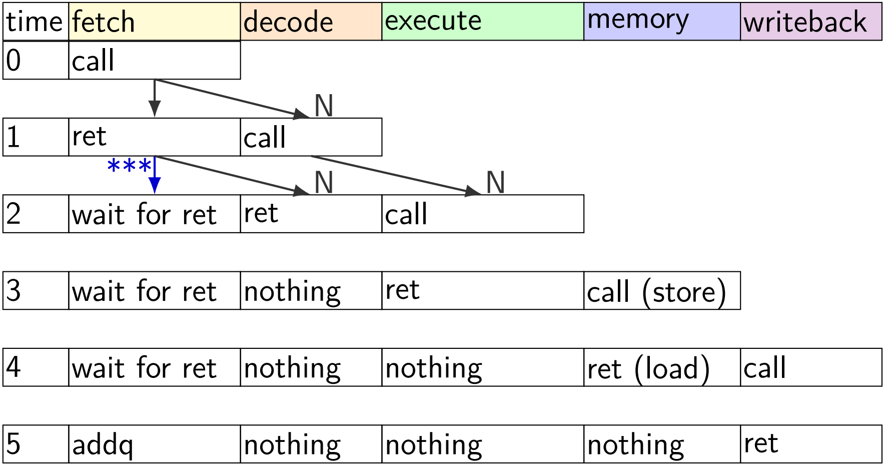


time	fetch	decode	execute	memory	writeback
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

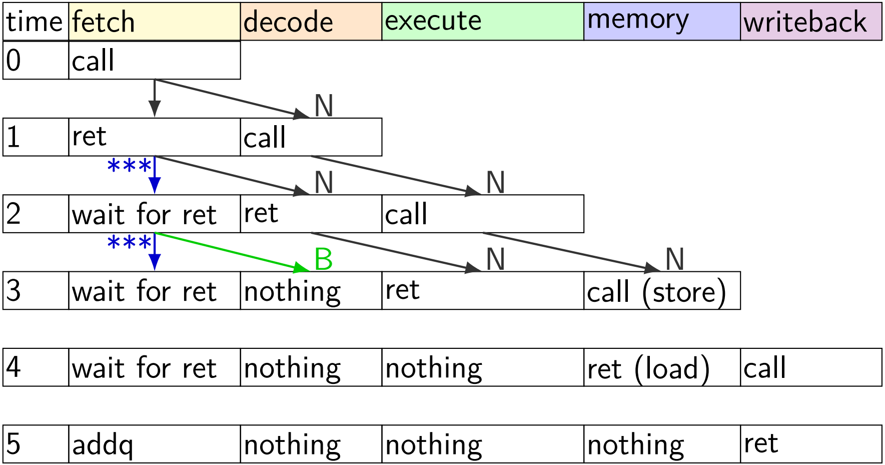


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

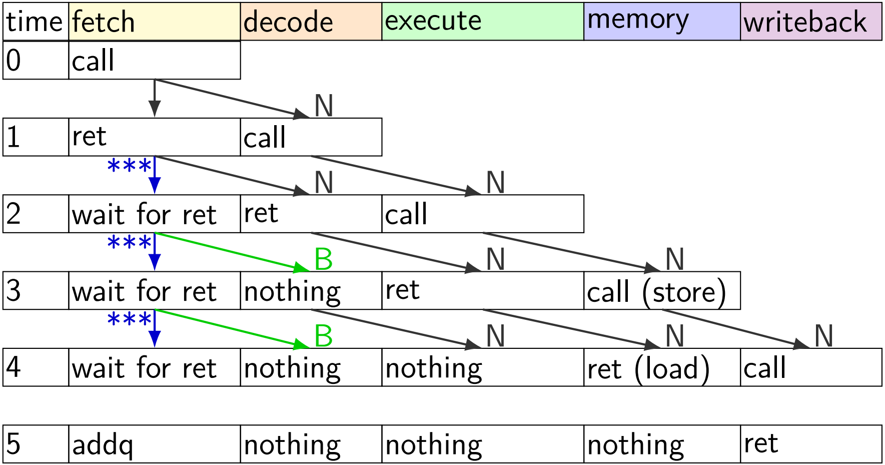


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

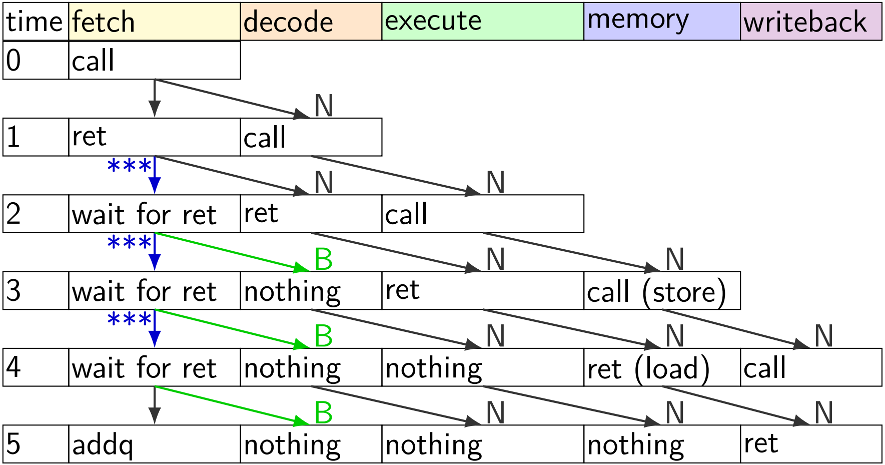


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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 */ );
```