pipelining (finish) / caching (start)

last time

forwarding cases

- dependency on pipeline
- value needs to be available before stage in which it is used
- (e.g. can't add + read memory in one cycle if execute+memory stages seperate)

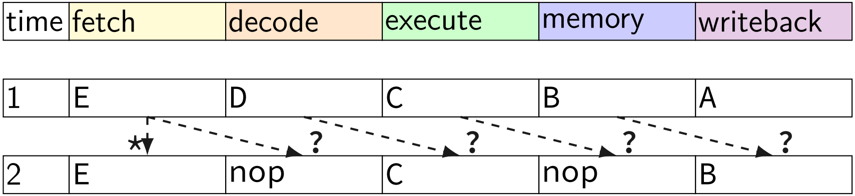
branch prediction — speculative guess about what conditional jump

squashing — undoing guess

pipeline register operations for stall/squashing

- keep same value in next cycle (HCLRS: `stall_X`)
- put no-op values in next cycle (HCLRS: `bubble_X`)

exercise: squash + stall (1)



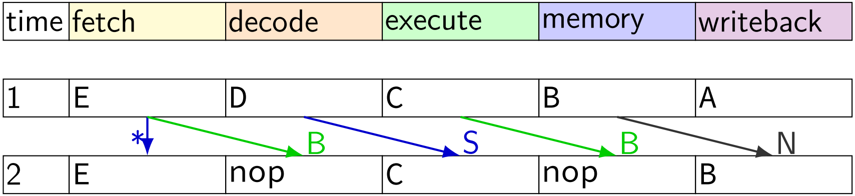
time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	E	nop	C	nop	B

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)

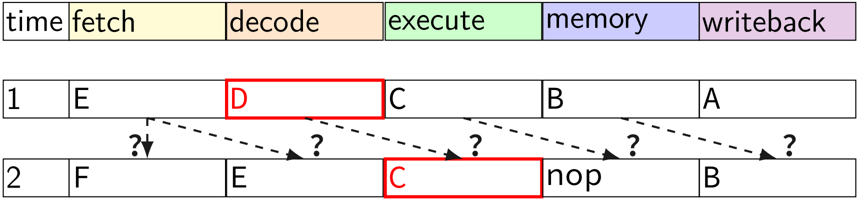


time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	E	nop	C	nop	B

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

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

exercise: squash + stall (2)

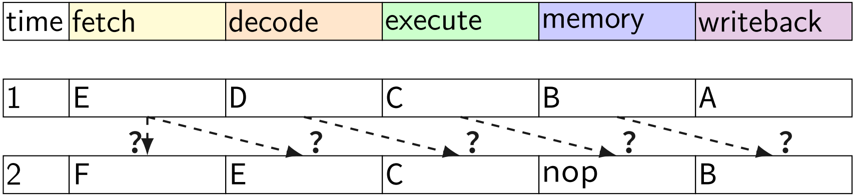


time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	F	E	C	nop	B

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

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

exercise: squash + stall (2)



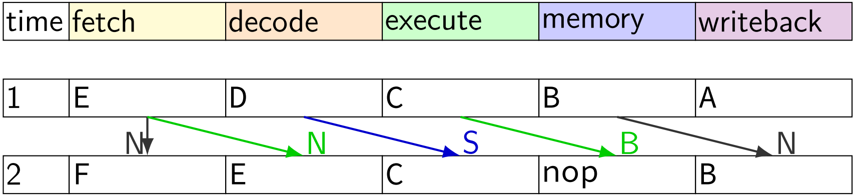
time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	F	E	C	nop	B

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
	N	N	S	B	N
2	F	E	C	nop	B

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

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

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

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

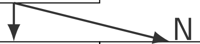
(*and* to replace instruction with a nop)

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

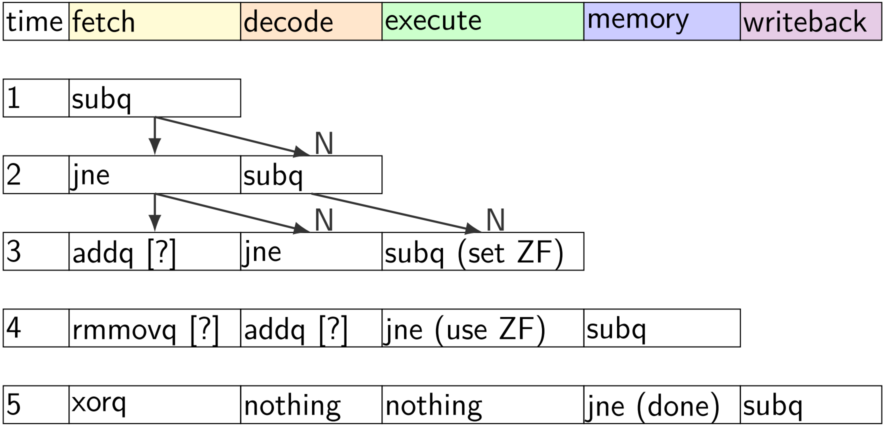


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

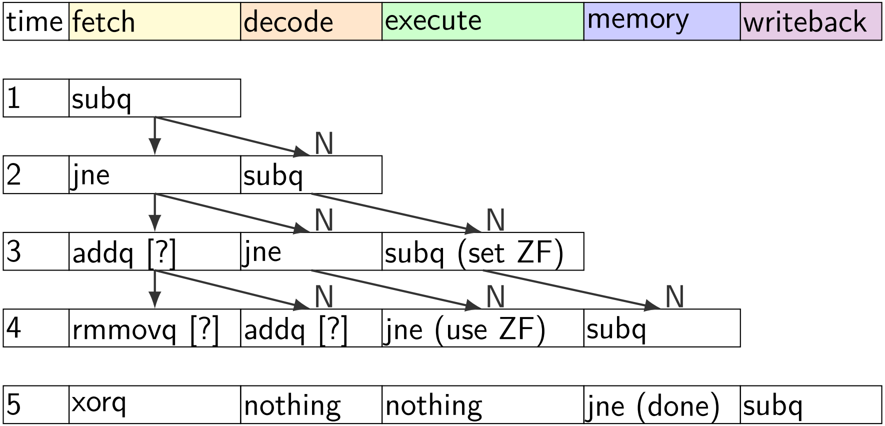


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

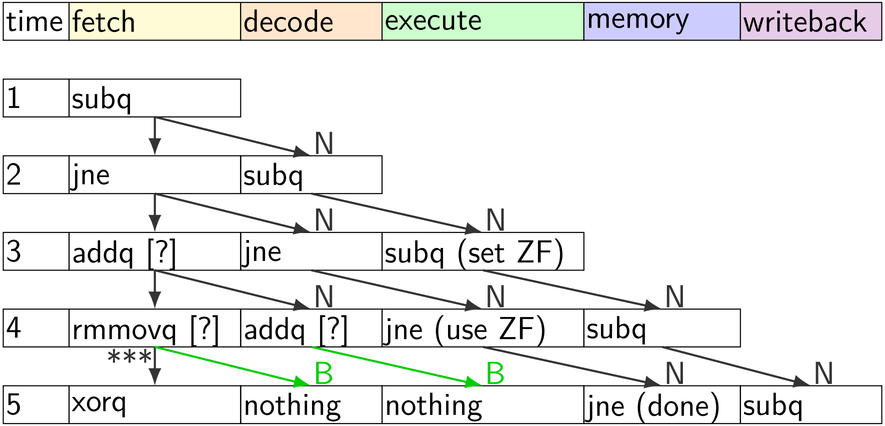


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

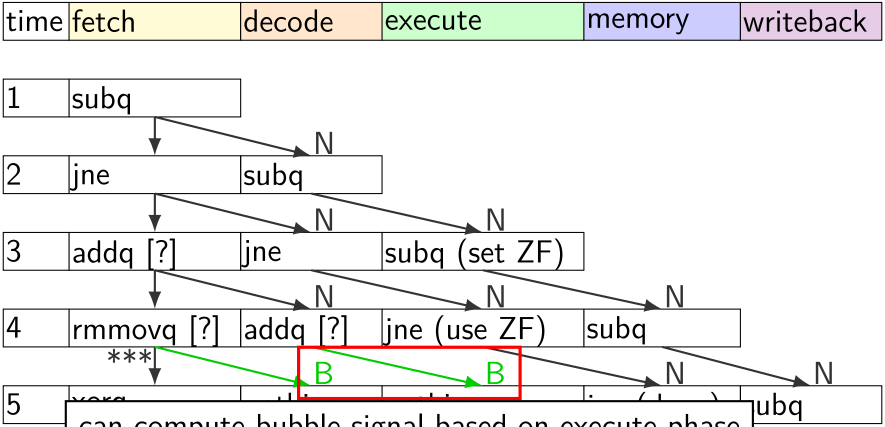


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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				ubq

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

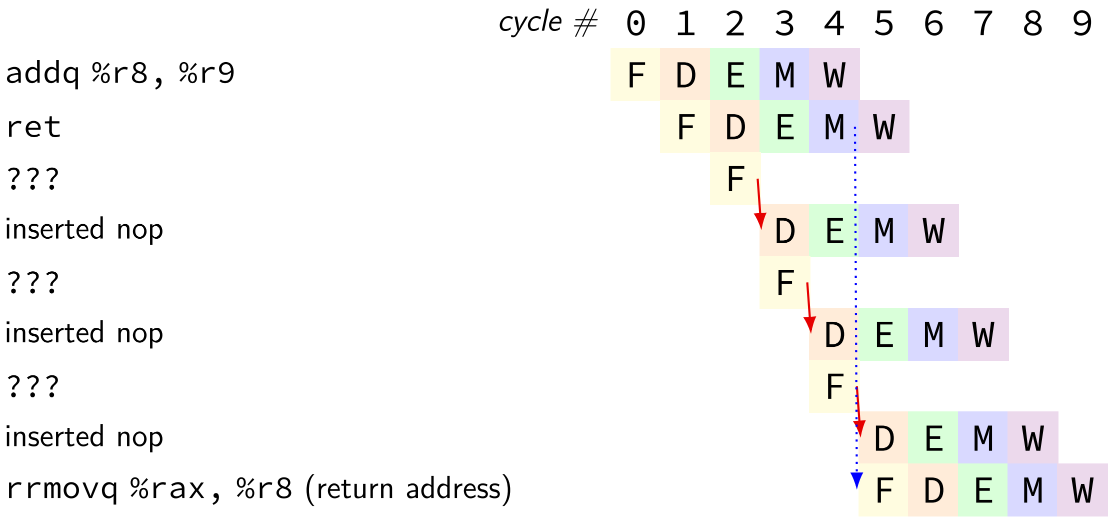
stal ew value

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 || ...;
```

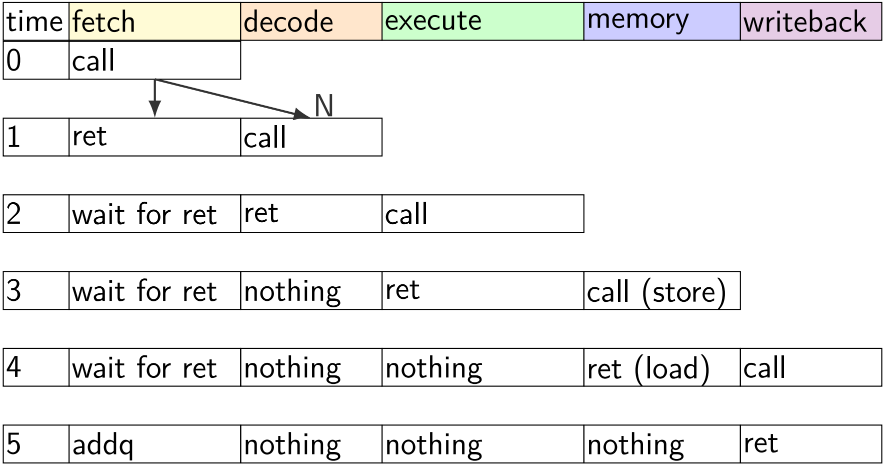
ret bubbles



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”)

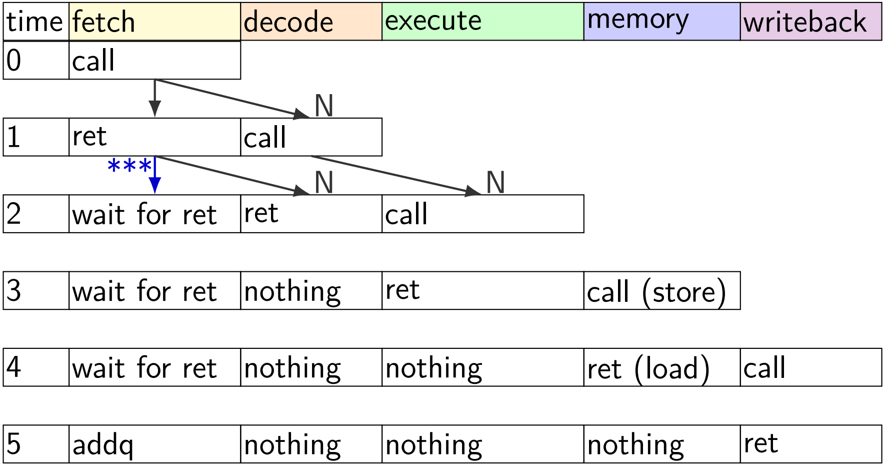
ret stall



time	fetch	decode	execute	memory	writeback
0	call				
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

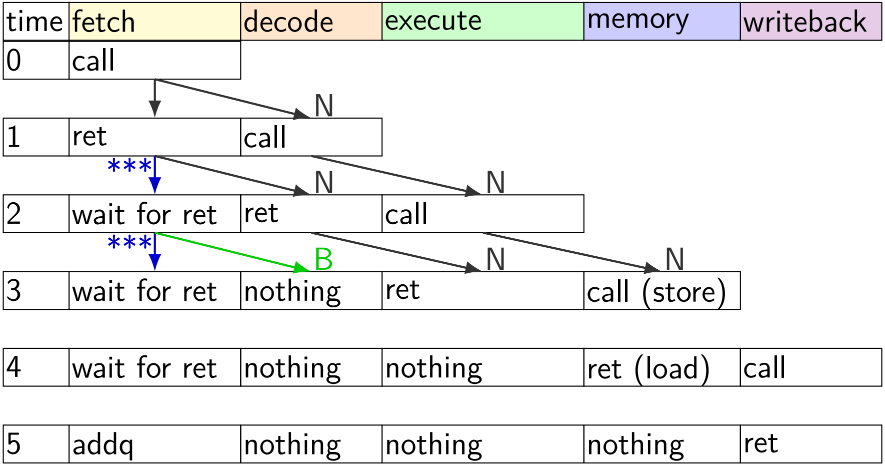


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

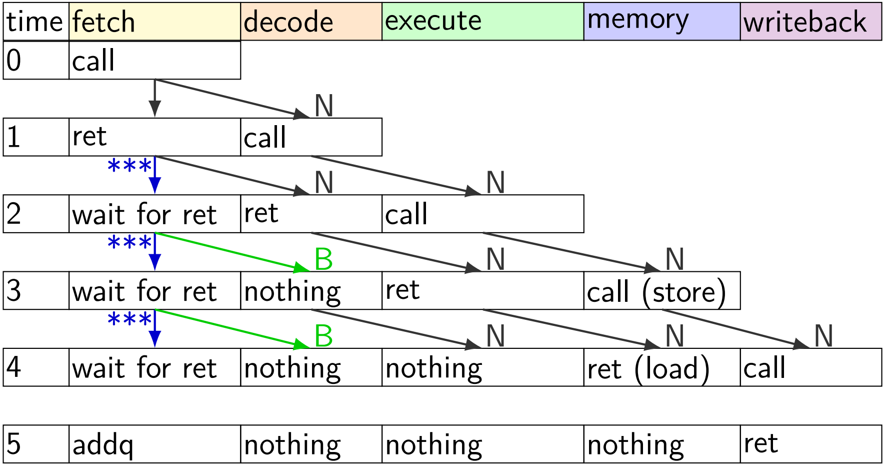


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

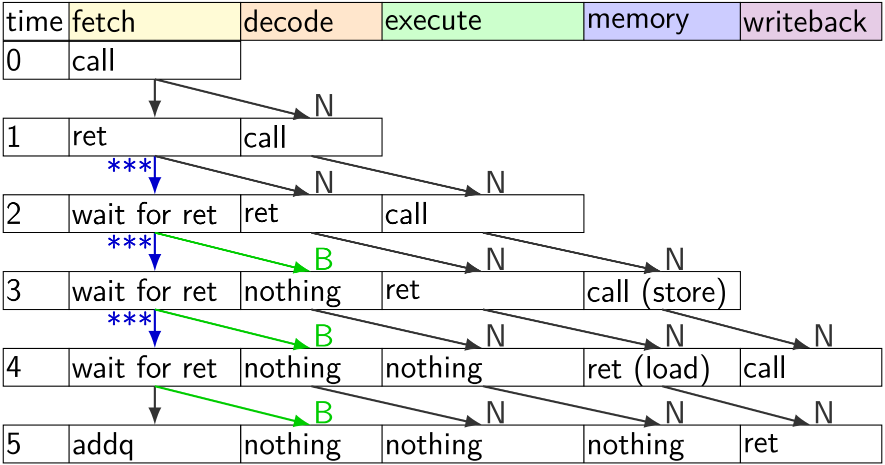


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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);

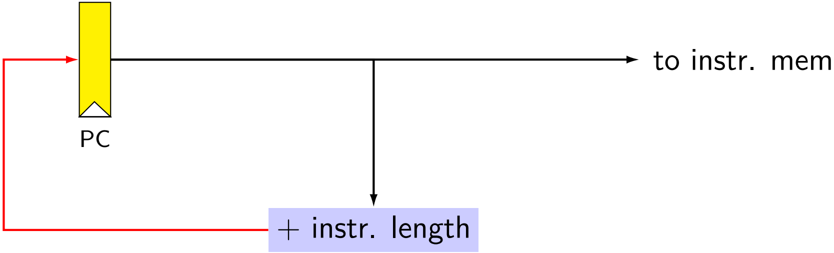
HCLRS bubble example

```
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 */ );
```

building the PC update (one possibility)

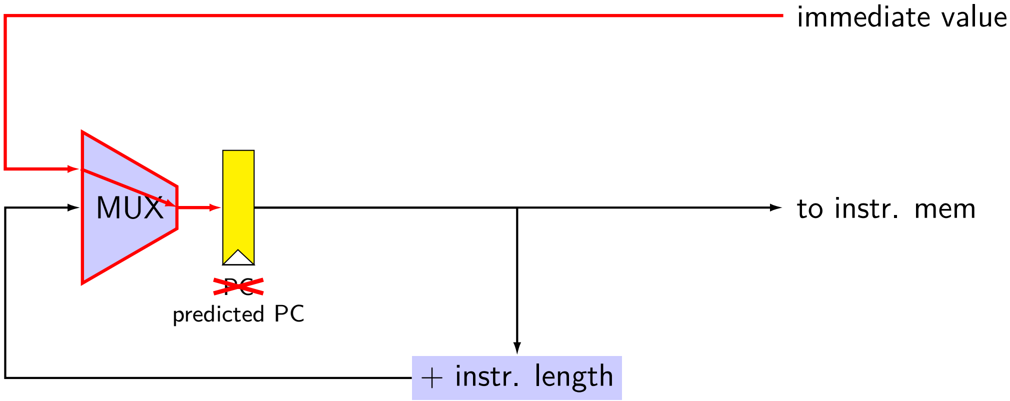
(1) normal case: PC ← PC + instr len



building the PC update (one possibility)

(1) normal case: PC ← PC + instr len

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

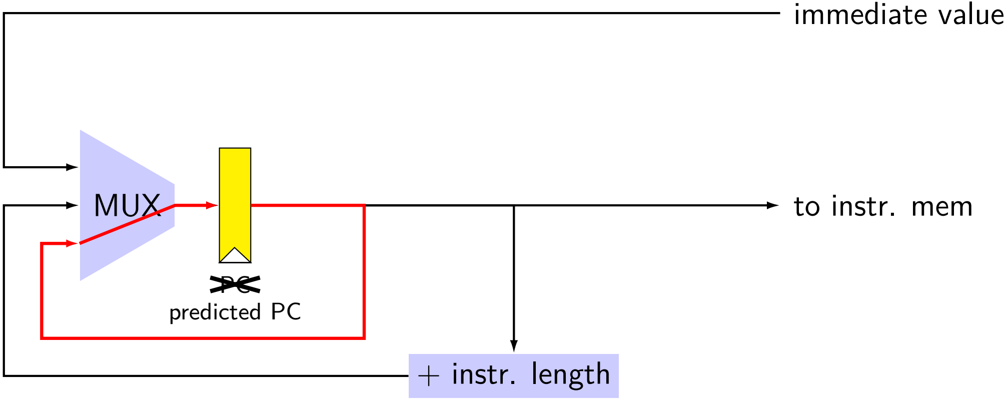


building the PC update (one possibility)

(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?)



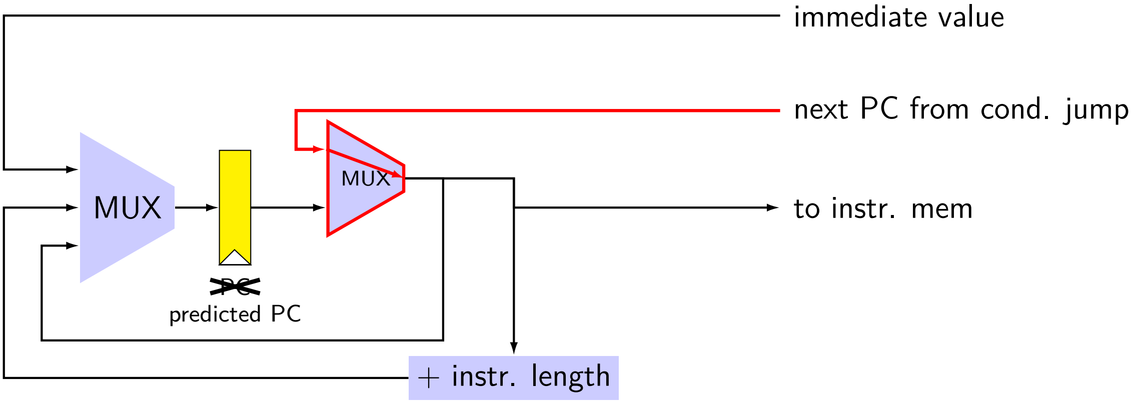
building the PC update (one possibility)

(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)

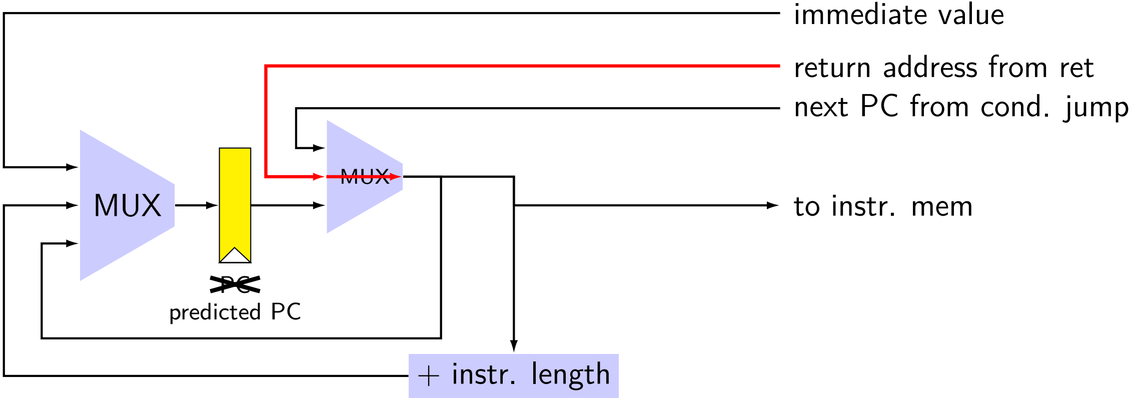
(1) normal case: PC ← PC + instr len

(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)

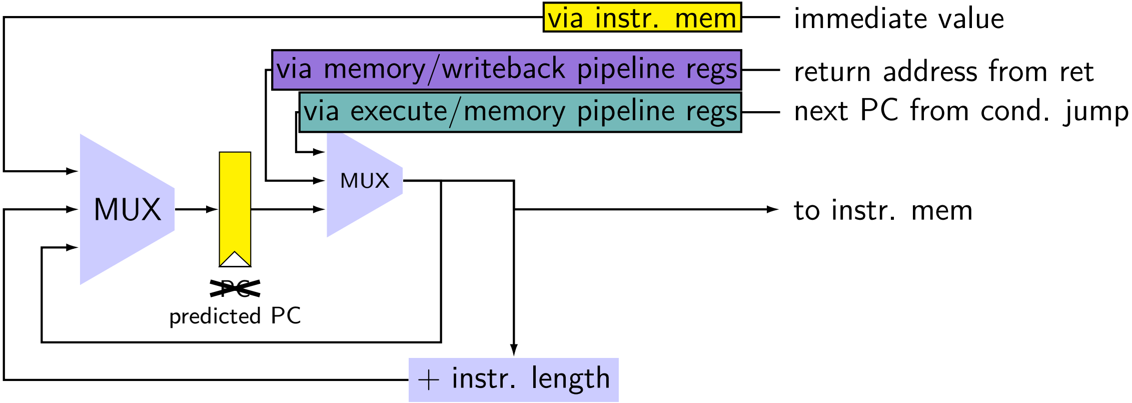
(1) normal case: PC ← PC + instr len

(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`



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)

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

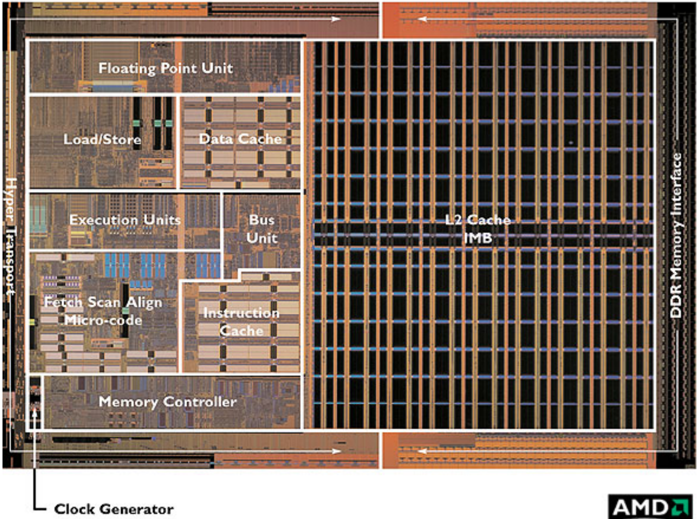
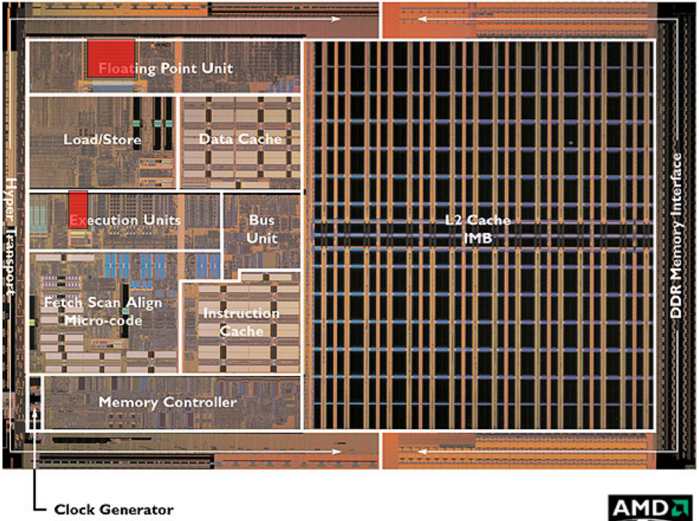


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

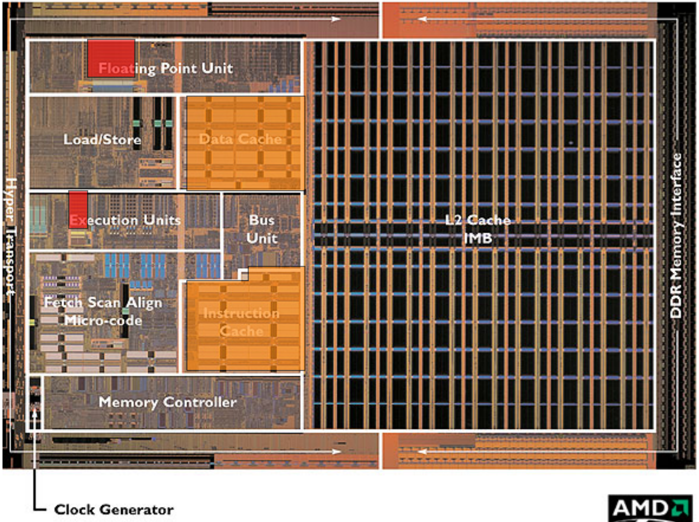
2004 CPU



▲ Registers

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

2004 CPU

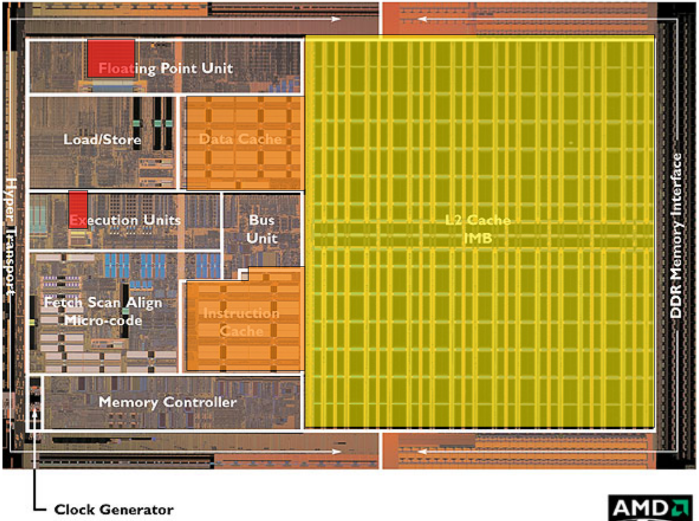


Registers

L1 cache

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

2004 CPU



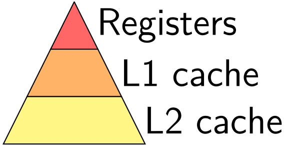
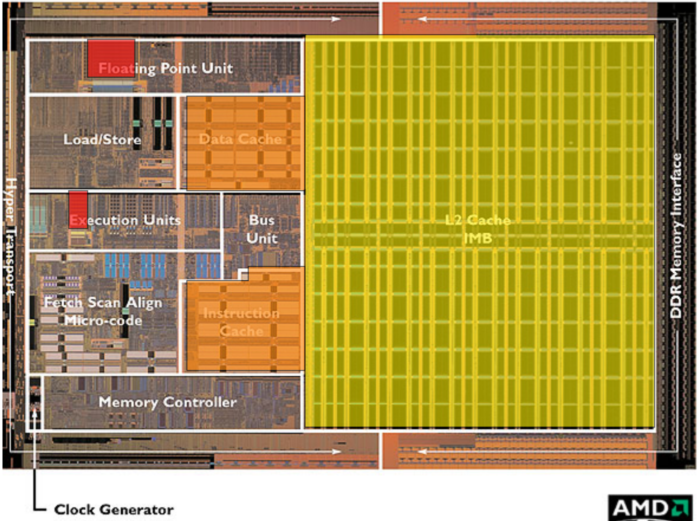
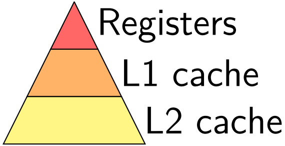


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



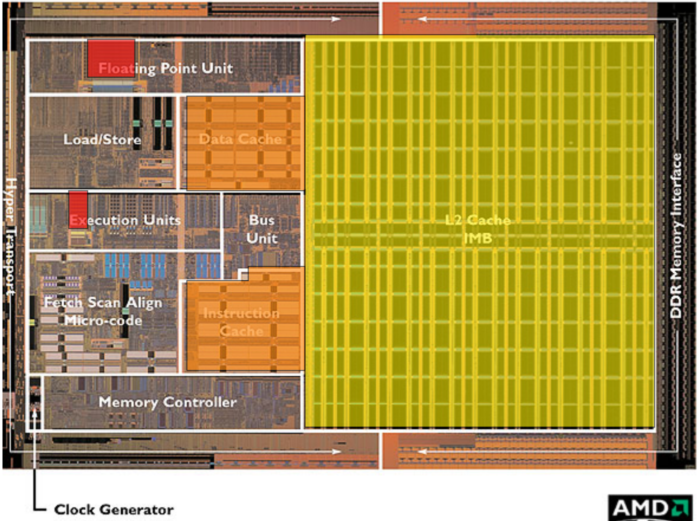

Registers

L1 cache

L2 cache

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

2004 CPU



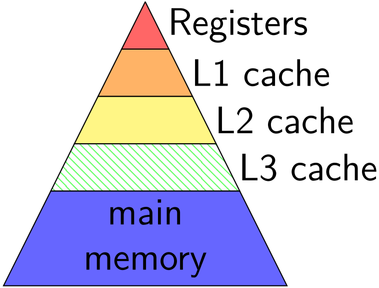
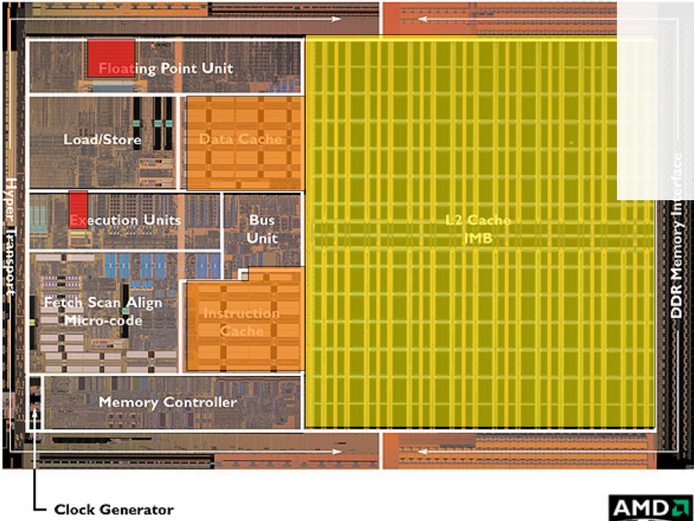
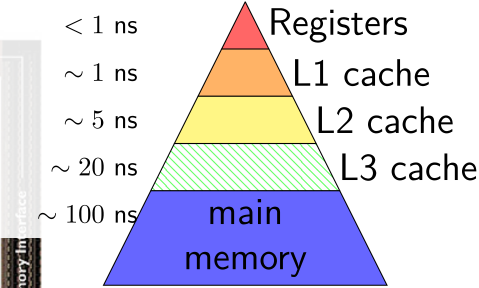


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU





< 1 ns Registers

~ 1 ns L1 cache

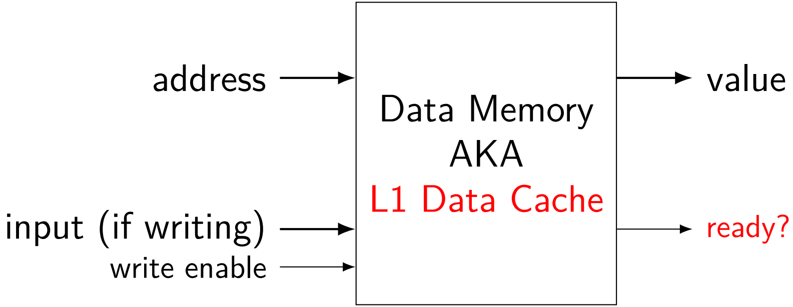
~ 5 ns L2 cache

~ 20 ns L3 cache

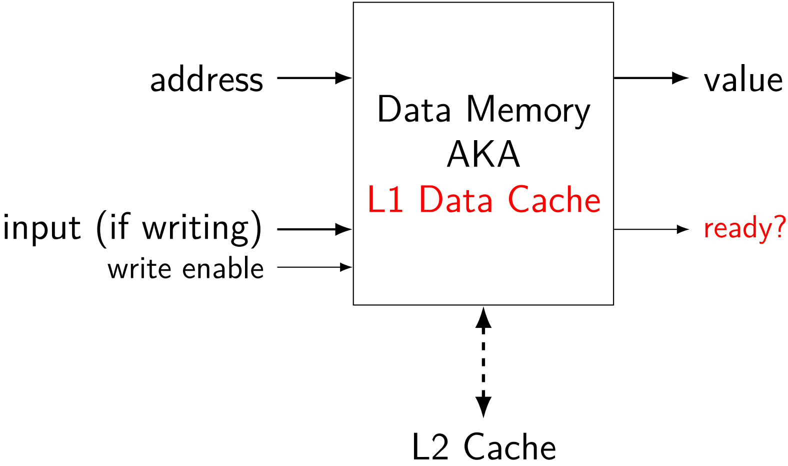
~ 100 ns main memory

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

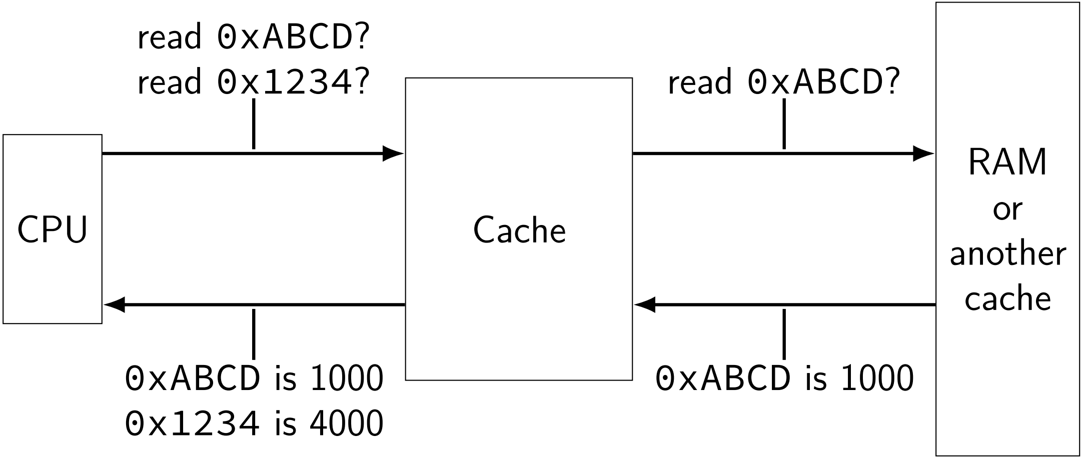
cache: real memory



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]`

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

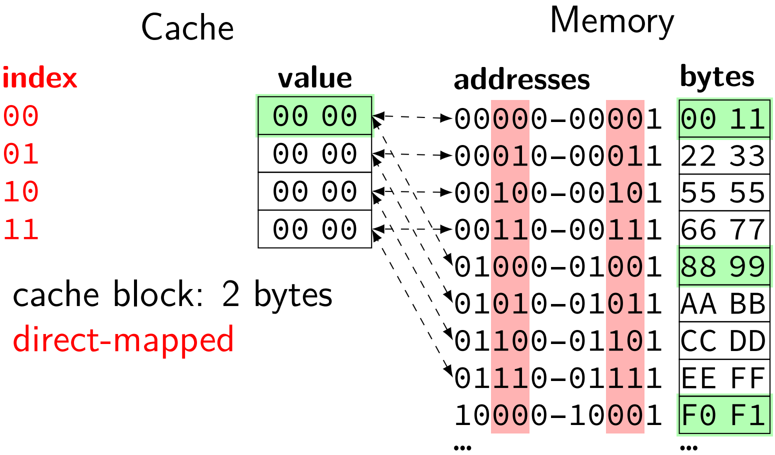
Memory

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

cache block: 2 bytes

direct-mapped

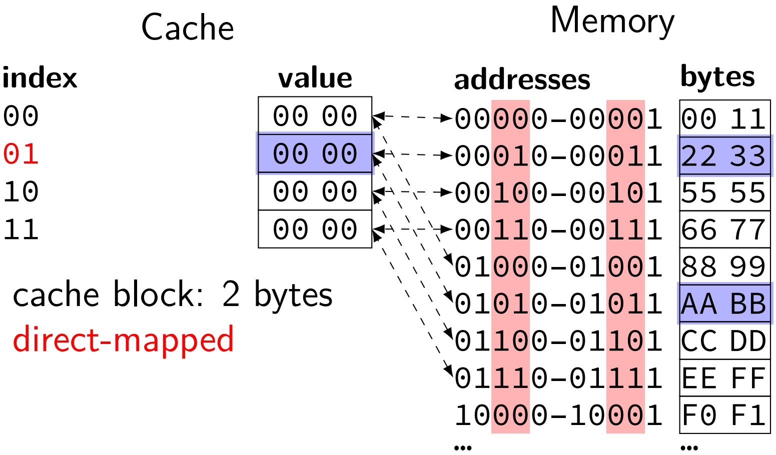
Memory

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?

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
…	…

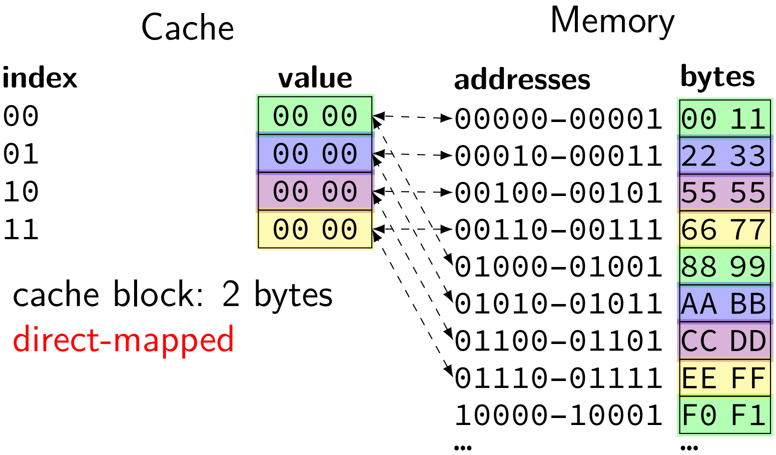
cache block: 2 bytes

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

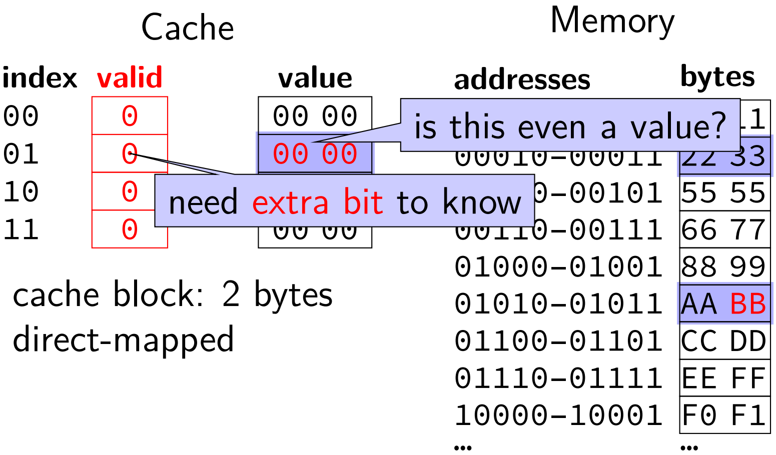
addresses	bytes
00000–00001	00 11
00010–00011	22 33
00100–00101	55 55
00110–00111	66 77
01000–01001	88 99
01010–01011	AA BB
01100–01101	CC DD
01110–01111	EE FF
10000–10001	F0 F1
…	…

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

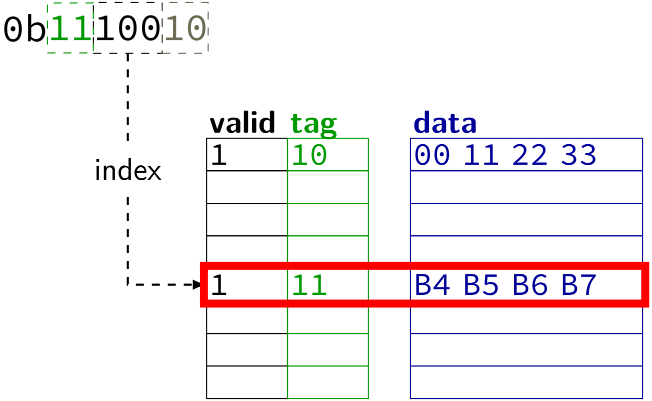
cache block: 2 bytes

direct-mapped

Memory

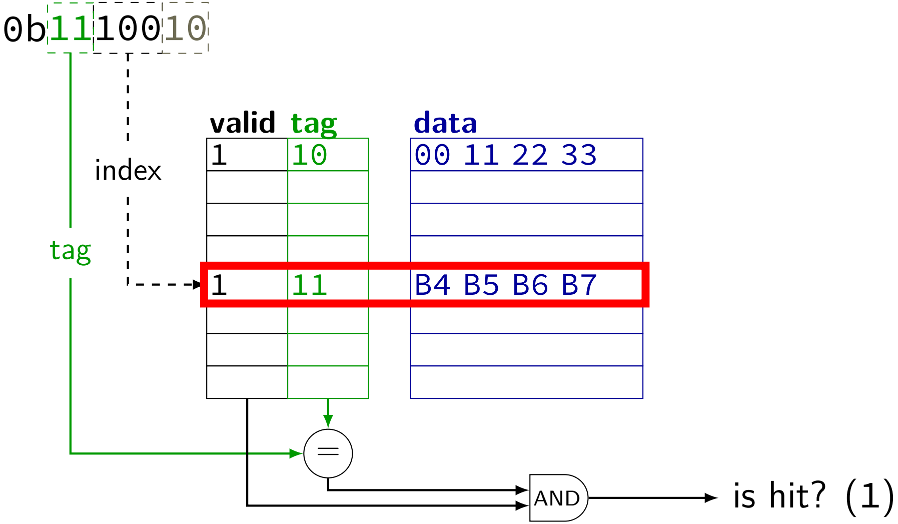
addresses	bytes
00000–00001	00 11
00010–00011	22 33
00100–00101	55 55
00110–00111	66 77
01000–01001	88 99
01010–01011	AA BB
01100–01101	CC DD
01110–01111	EE FF
10000–10001	F0 F1
…	…

cache operation (read)



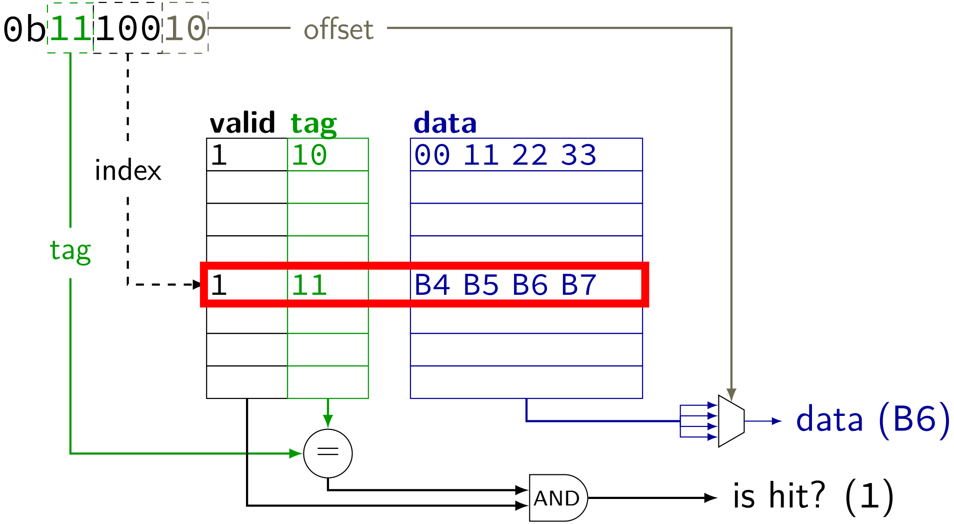
valid	tag	data
1	10	00 11 22 33
1	11	B4 B5 B6 B7

cache operation (read)



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

cache operation (read)



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

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

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

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	???	???	1
2 byte blocks, 8 sets	???	???	1
4 byte blocks, 2 sets	???	???	11

2 byte blocks, 4 sets

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

4 byte

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

$4 = 2^2$ bytes in block

2 bits to say which byte

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		

$2^3 = 8$ sets

3 bits to index set

4

index	vali		
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	--	-- --
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)

backup slides

stalling/misprediction and latency

hazard handling where pipeline latency matters

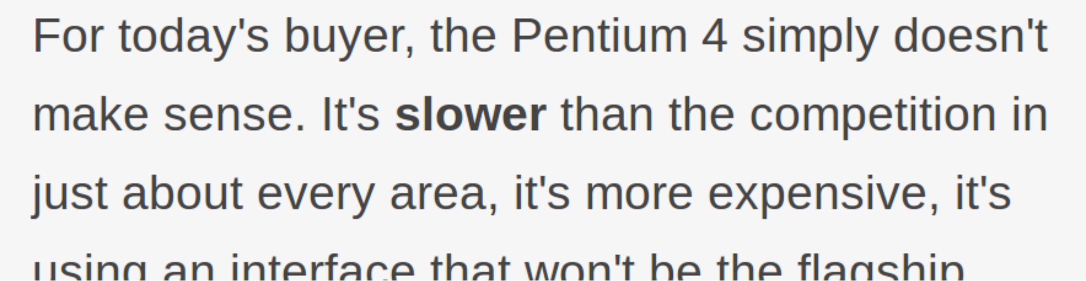
longer pipeline — larger penalty

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

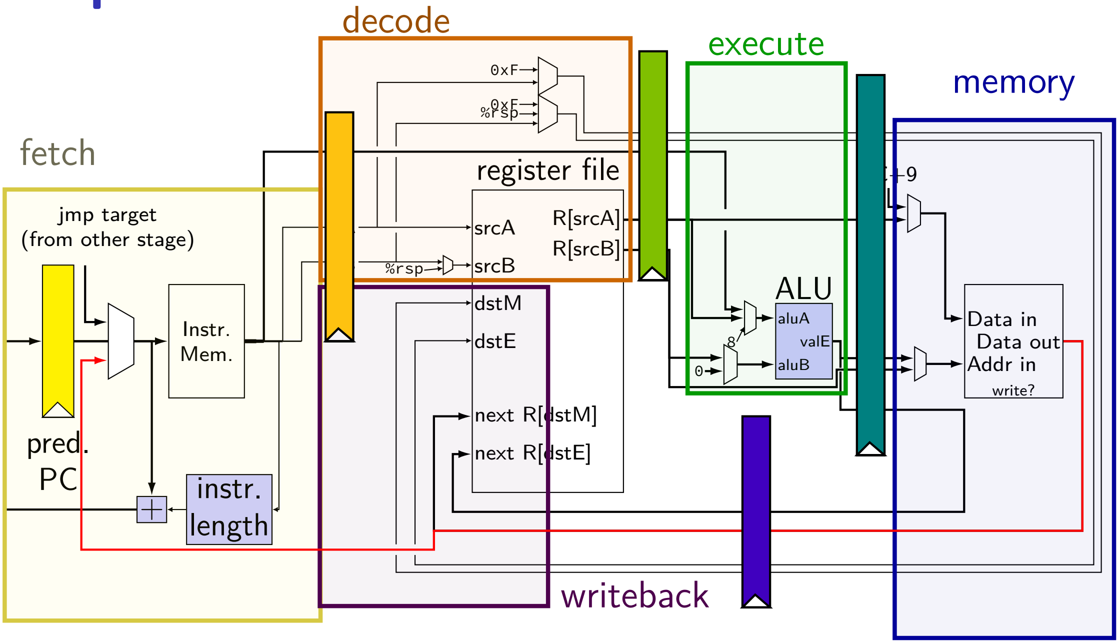
first-generation review quote:



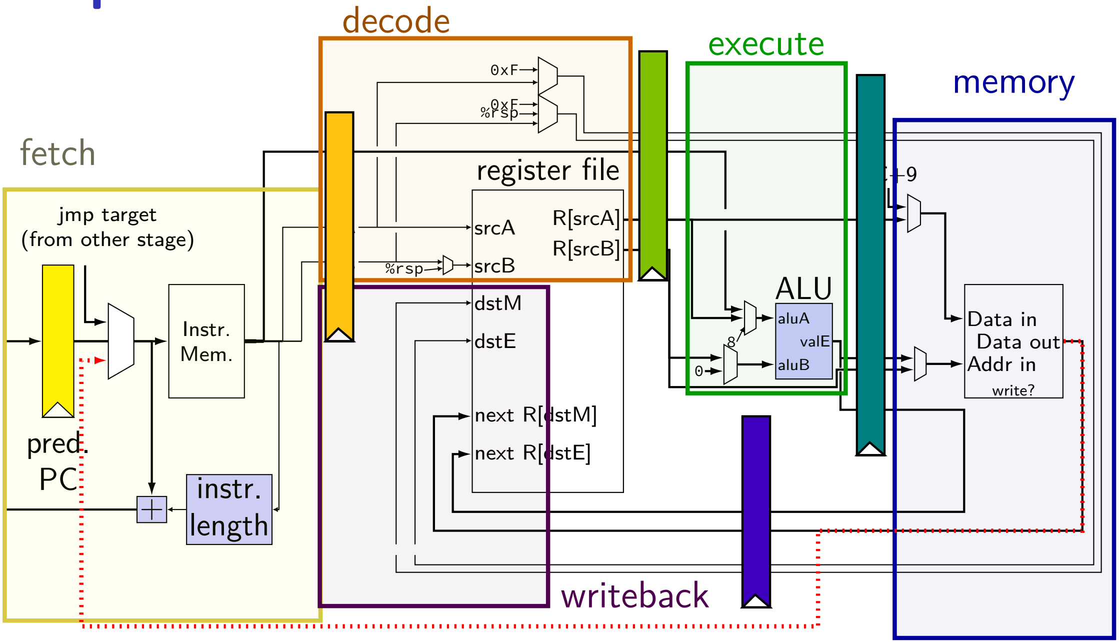
For today's buyer, the Pentium 4 simply doesn't make sense. It's **slower** than the competition in just about every area, it's more expensive, it's using an interface that won't be the flagship

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

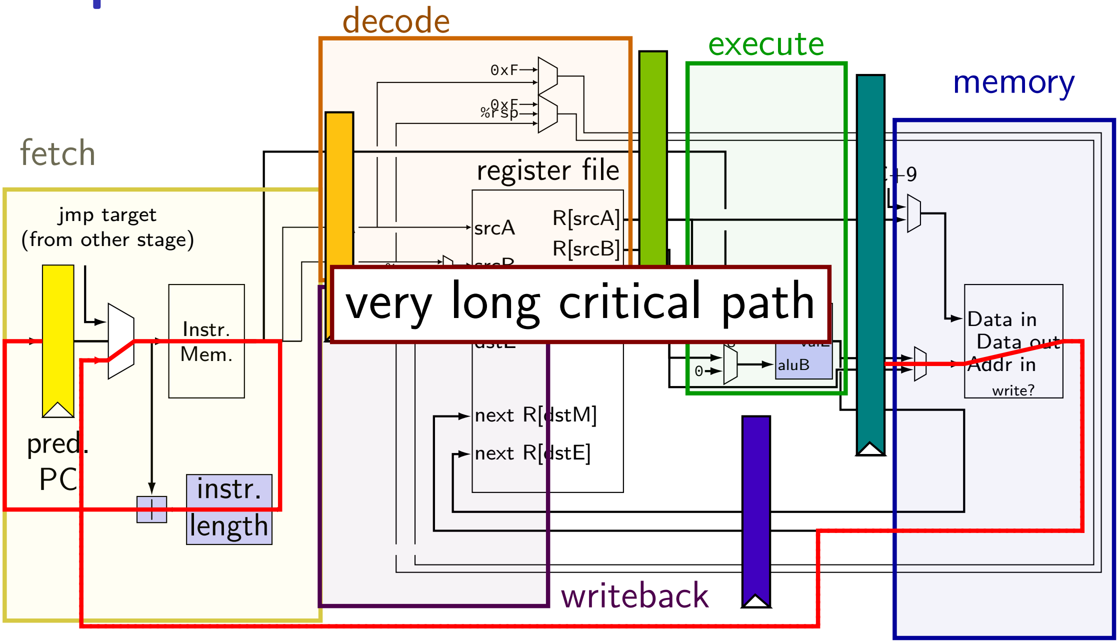
ret paths


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

ret paths

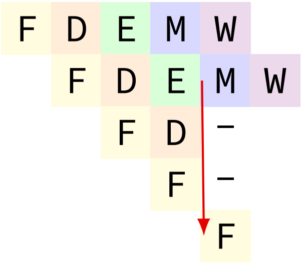

decode
execute
memory
fetch
0xF
0xF
%rsp
register file
jmp target
(from other stage)
srcA
srcB
R[srcA]
R[srcB]
%rsp
Instr.
Mem.
dstM
dstE
ALU
aluA
valE
aluB
8
0
+9
Data in
Data out
Addr in
write?
next R[dstM]
next R[dstE]
pred.
PC
instr.
length
writeback

ret paths



very long critical path

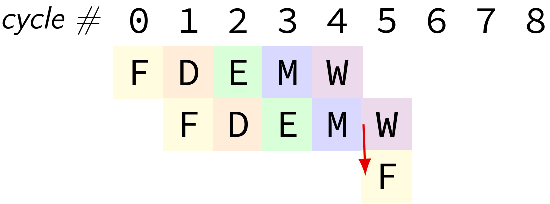
PC update: jmp misprediction



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	–				
`andq %rbx, %rcx` (mispredicted)				F	–				
`subq %r9, %r10` (instr. after jle)					F				

memory stage of jump → fetch of corrected instr.

PC update: ret



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

writeback stage of ret → fetch of corrected instr.

better branch prediction

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

intuition: loops:

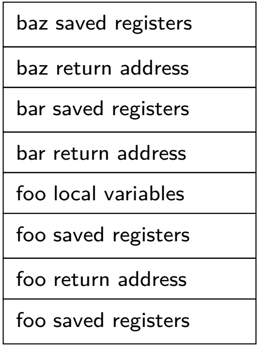
```
LOOP: ...
      ...
      je LOOP

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

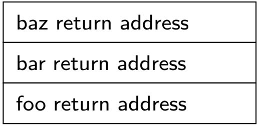
foo local variables

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

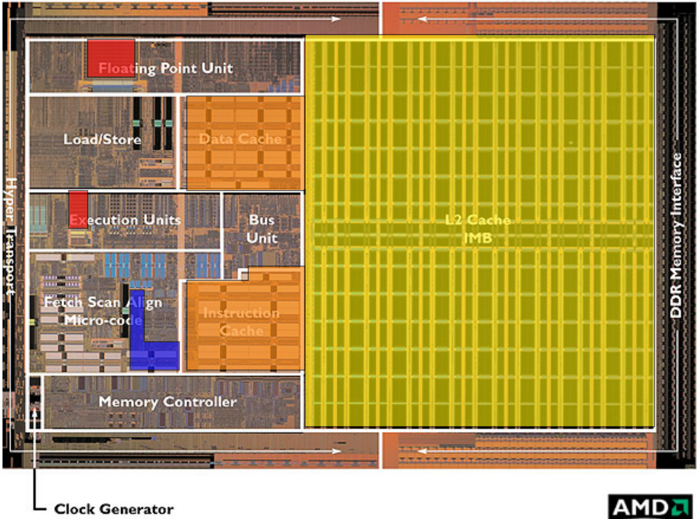
real processors can take multiple cycles to read instruction memory

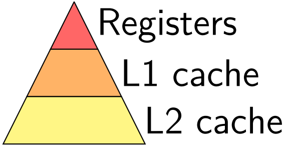
predict branches before reading their opcodes

how — more extra data structures

- tables of recent branches (often many kilobytes)

2004 CPU




Registers

L1 cache

L2 cache

🟦 Branch Prediction (approximate)

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