Pipelining 4 / Caches 1

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

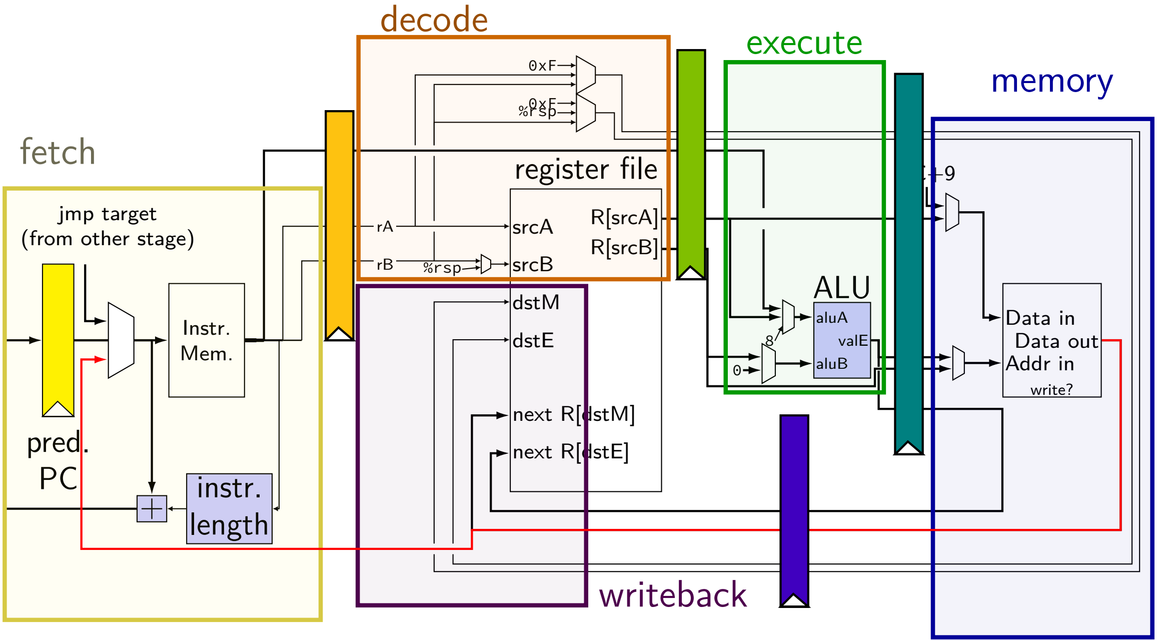
jXX control hazard: branch prediction + squashing

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

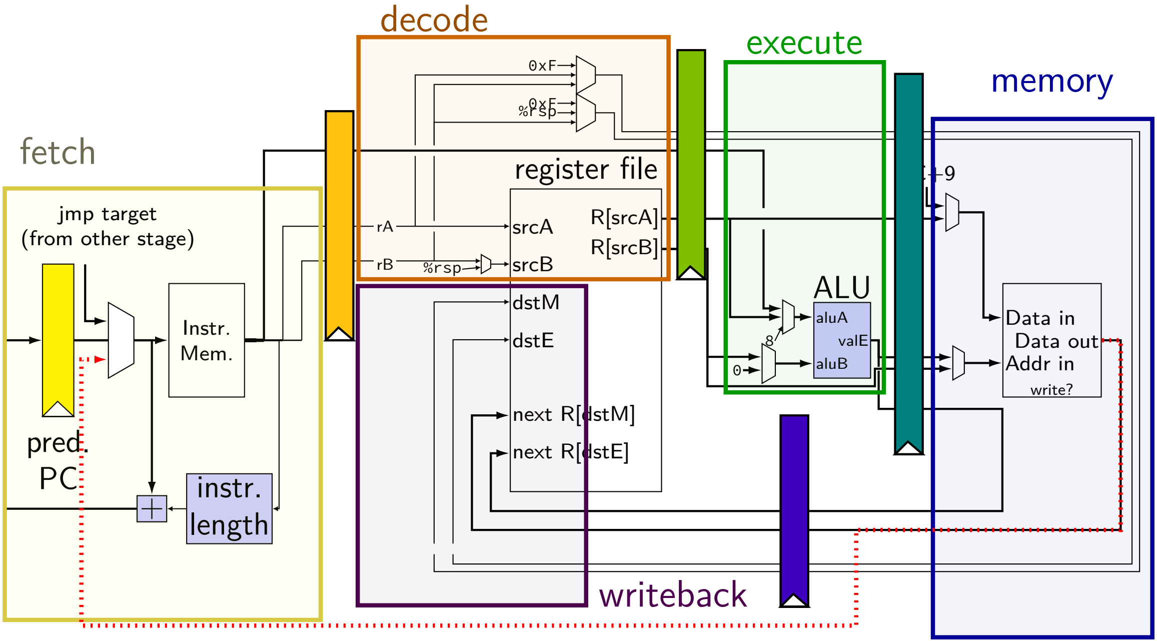
ret control hazard: 3 cycles of stalling

- (fetch next instruction after ret finishes memory)

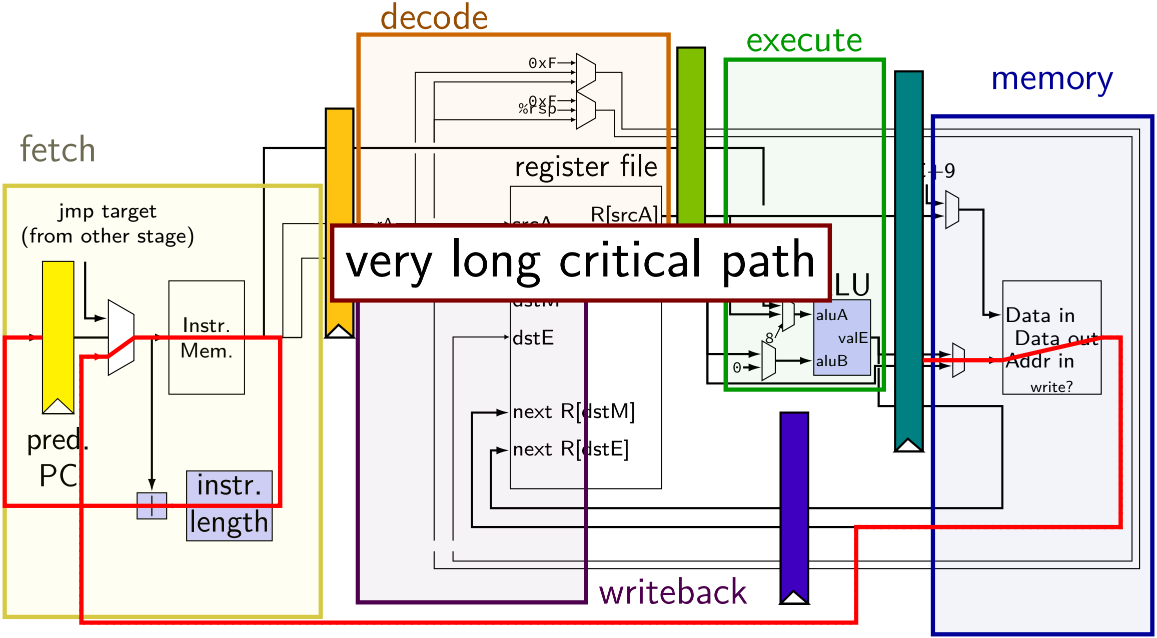
ret paths


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

ret paths

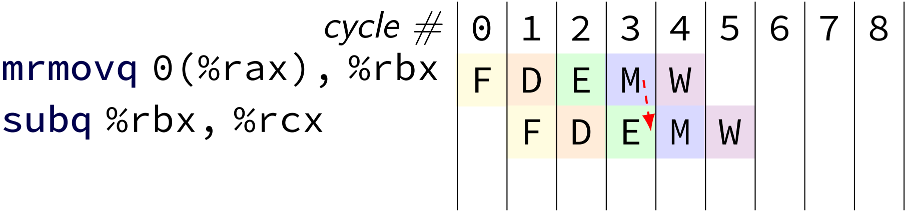

decode
execute
memory
fetch
writeback
0xF
0xF
%rsp
register file
jmp target
(from other stage)
rA
rB
%rsp
srcA
srcB
R[srcA]
R[srcB]
dstM
dstE
next R[dstM]
next R[dstE]
Instr.
Mem.
pred.
PC
instr.
length
+9
ALU
aluA
valE
aluB
8
0
Data in
Data out
Addr in
write?

ret paths



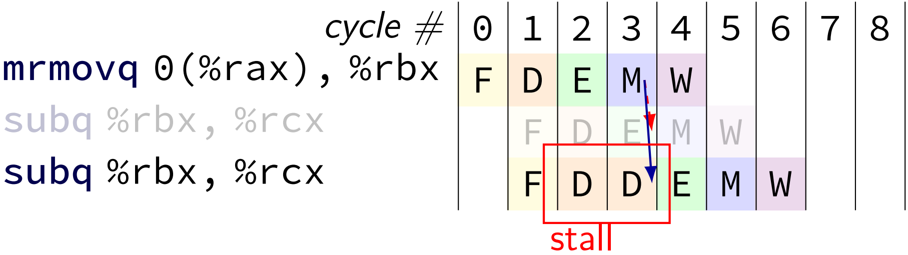
very long critical path

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			

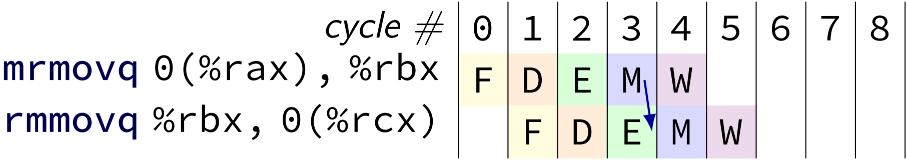
unsolved problem



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

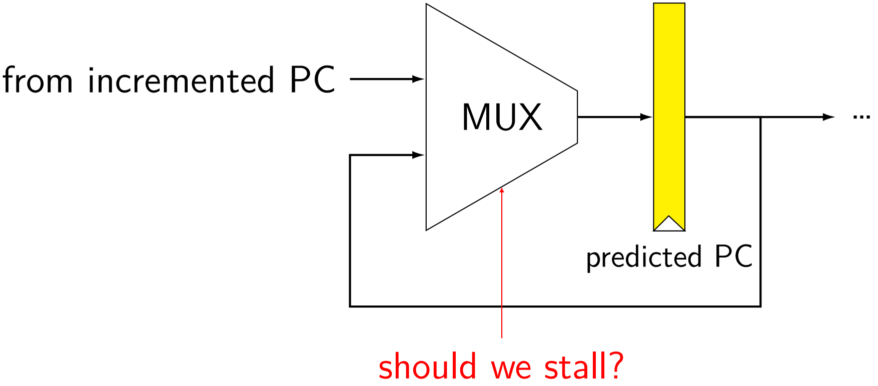
solveable problem



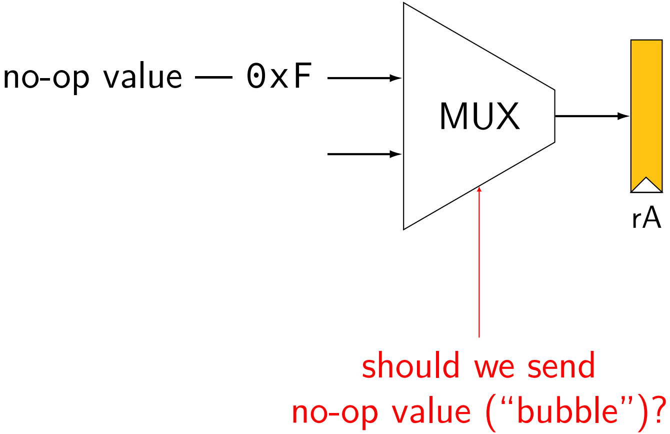
cycle #	0	1	2	3	4	5	6	7	8
mrmovq 0(%rax), %rbx	F	D	E	M	W				
rmmovq %rbx, 0(%rcx)		F	D	E	M	W			

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



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

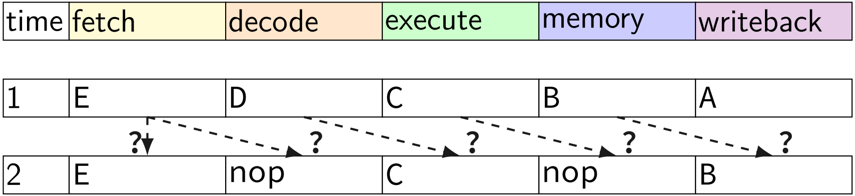
time	a_value	B_value	stall_B	bubble_B
0	0x01	0xFF	0	0
1	0x02	???	1	0
2	0x03	???	0	0
3	0x04	???	0	1
4	0x05	???	0	0
5	0x06	???	0	0
6	0x07	???	1	0
7	0x08	???	1	0
8		???		

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)



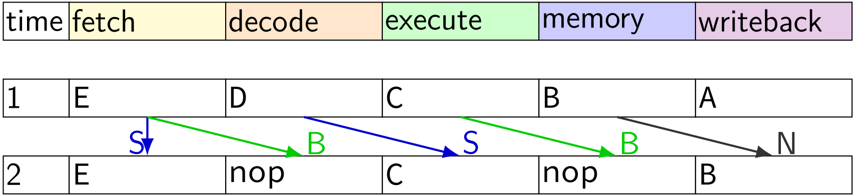
time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	E	nop	C	nop	B

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)

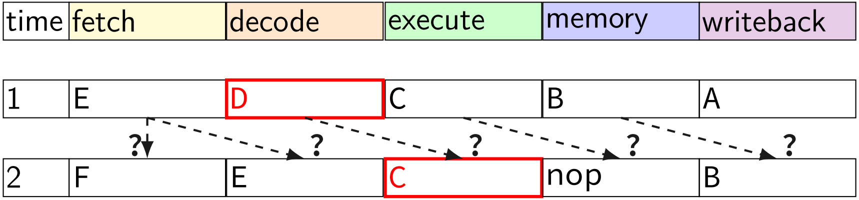


time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	E	nop	C	nop	B

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

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

exercise: squash + stall (2)

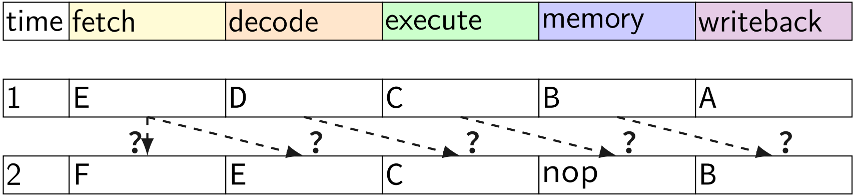


time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	F	E	C	nop	B

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

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

exercise: squash + stall (2)



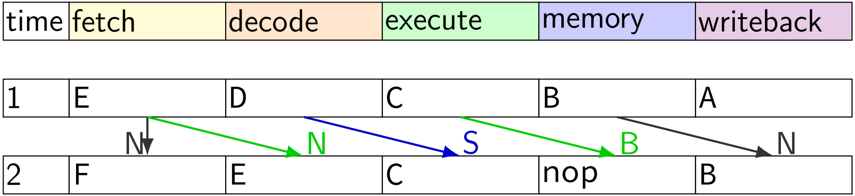
time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
	?	?	?	?	?
2	F	E	C	nop	B

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)

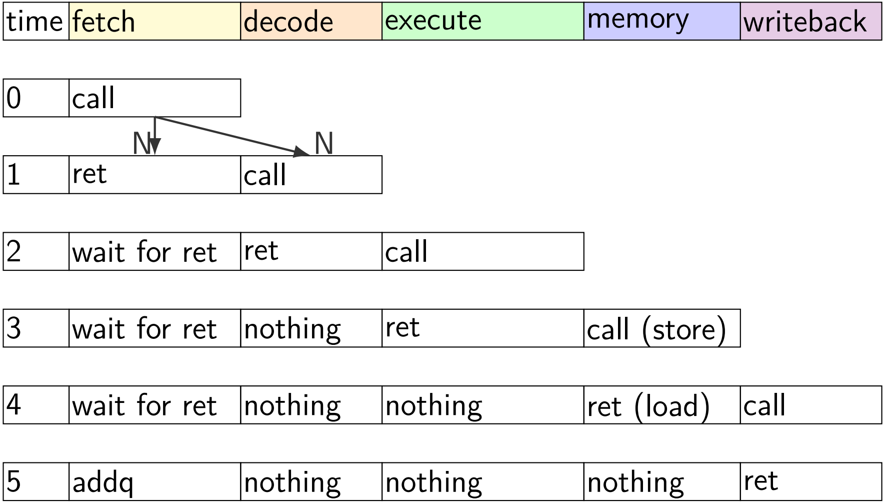


time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	F	E	C	nop	B

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

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

ret stall

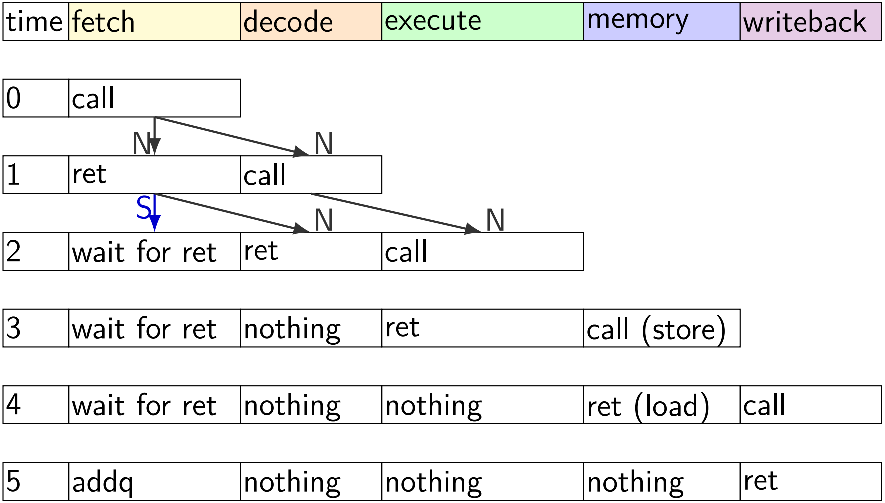


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

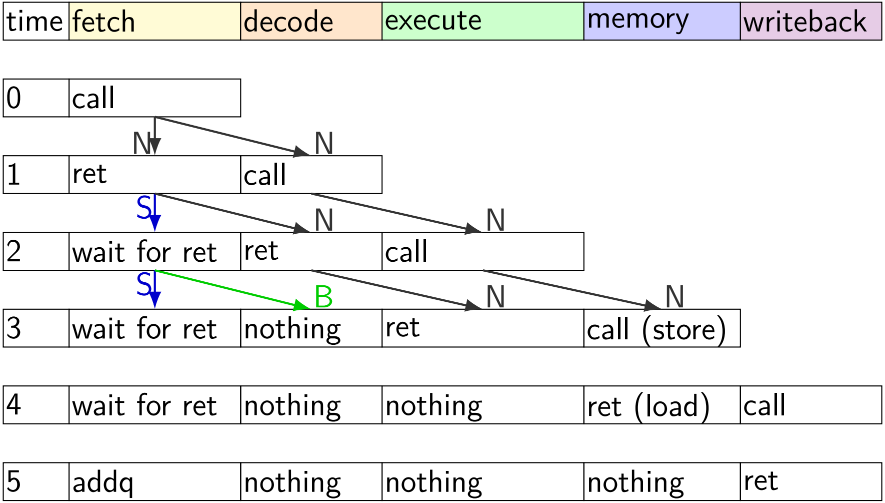
ret stall



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

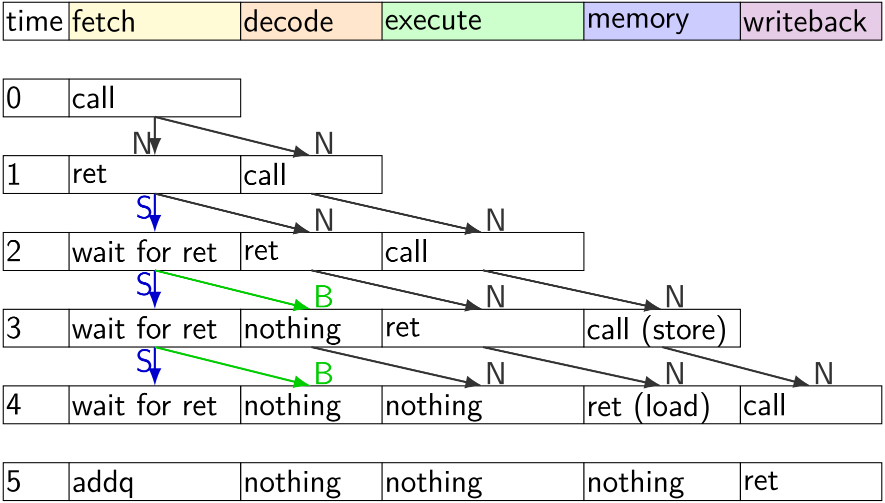


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

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

ret stall

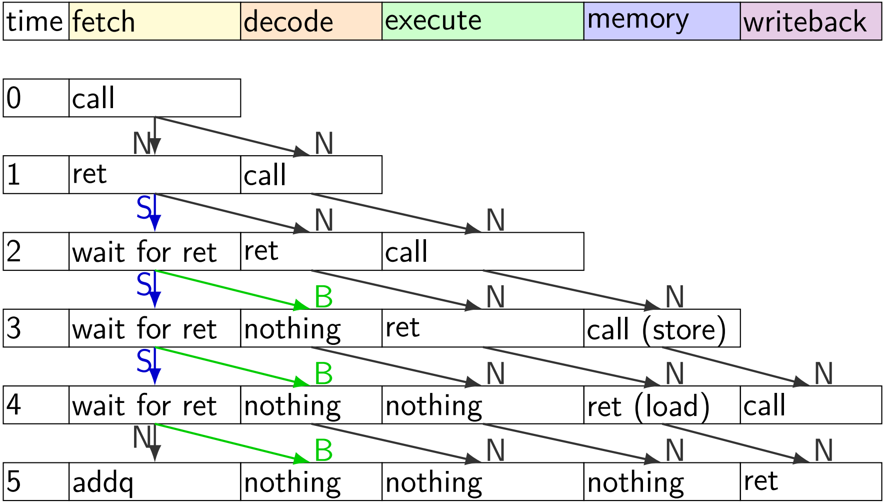


time	fetch	decode	execute	memory	writeback
0	call				
1	ret	call			
2	wait for ret	ret	call		
3	wait for ret	nothing	ret	call (store)	
4	wait for ret	nothing	nothing	ret (load)	call
5	addq	nothing	nothing	nothing	ret

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

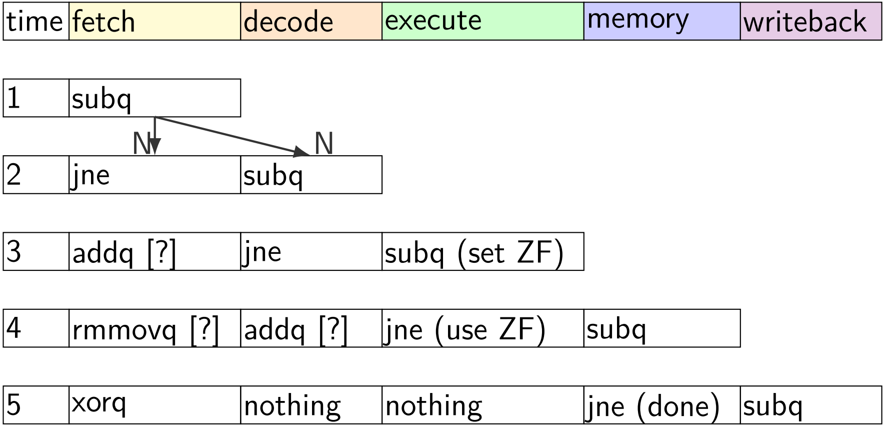
bubble (B) = use default (no-op);

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

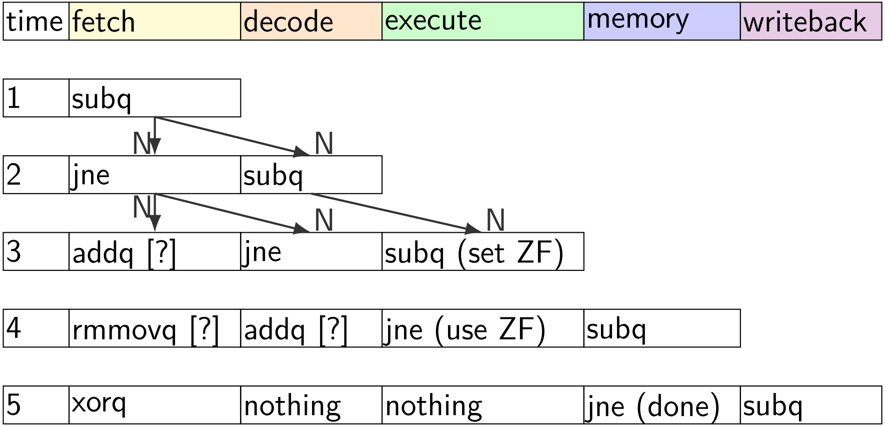


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

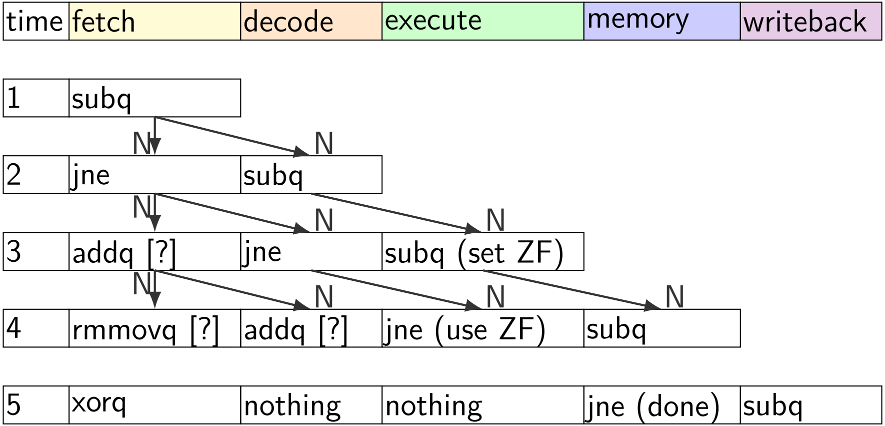


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

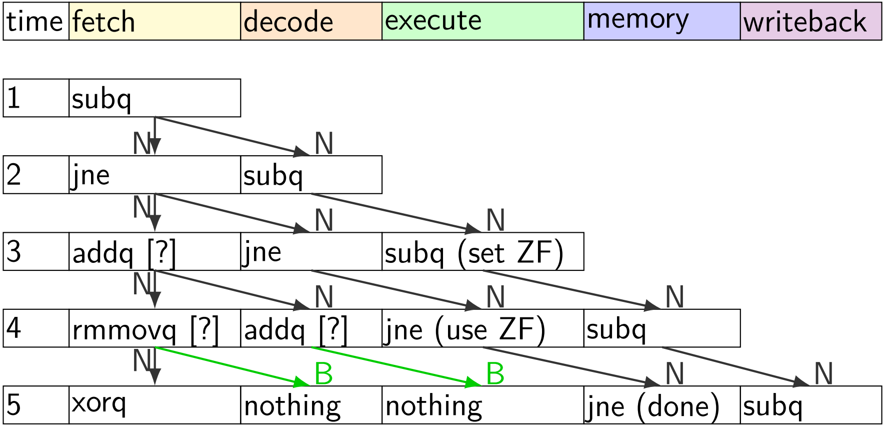


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

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

squashing with stall/bubble

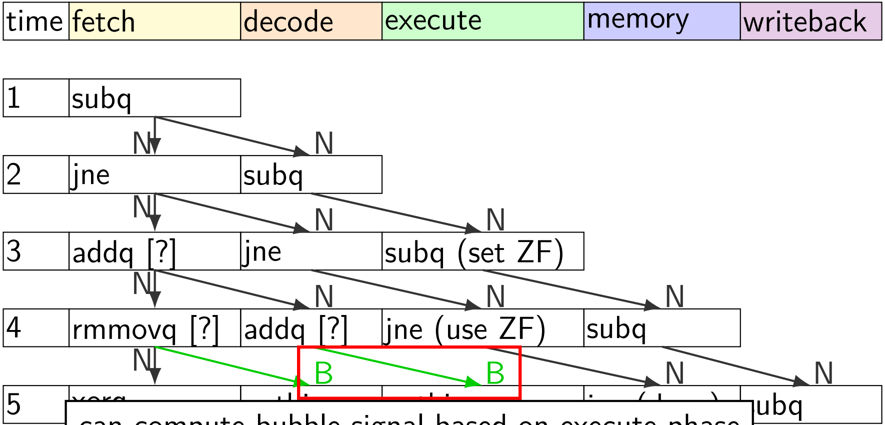


time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	subq	
5	xorq	nothing	nothing	jne (done)	subq

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

missing pieces

multi-cycle memories

beyond pipelining: out-of-order, multiple issue

missing pieces

multi-cycle memories

beyond pipelining: out-of-order, multiple issue

multi-cycle memories

ideal case for memories: single-cycle

achieved with caches (next topic)

- fast access to small number of things

typical performance:

- 90+% of the time: single-cycle

sometimes many cycles (3–400+)

variable speed memories

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")

`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

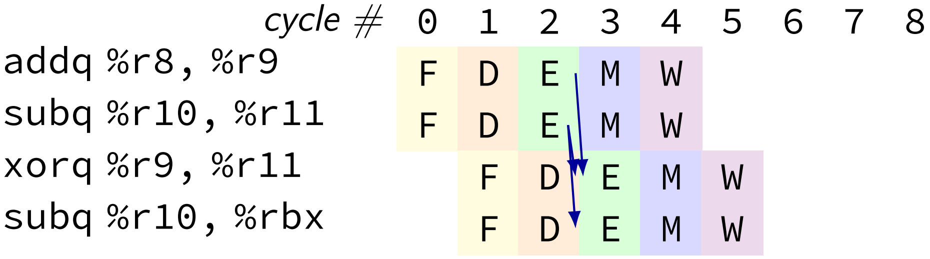
beyond pipelining: out-of-order, multiple issue

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
…											

better branch prediction

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

intuition: loops:

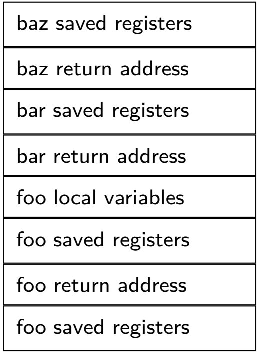
```
LOOP: ...
      ...
      je LOOP

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

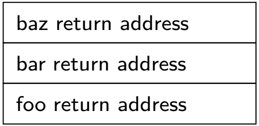
foo local variables

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

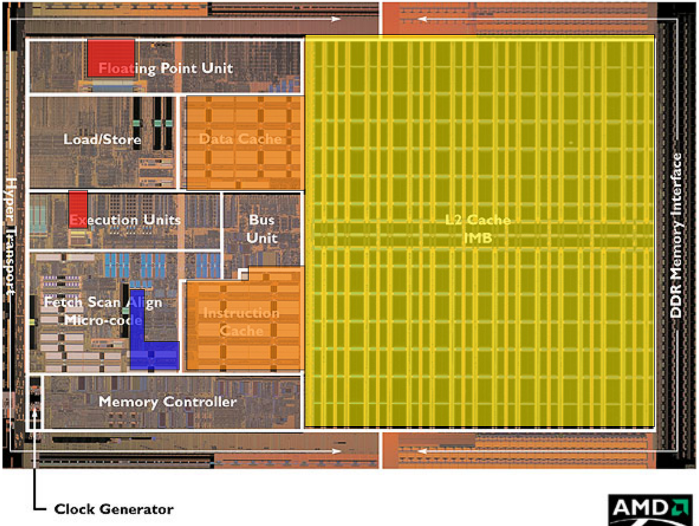
real processors can take multiple cycles to read instruction memory

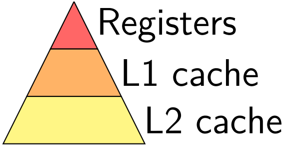
predict branches before reading their opcodes

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)

stalling/misprediction and latency

hazard handling where pipeline latency matters

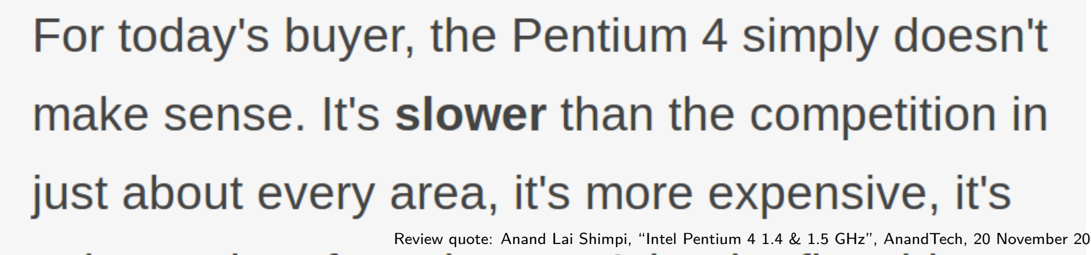
longer pipeline — larger penalty

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

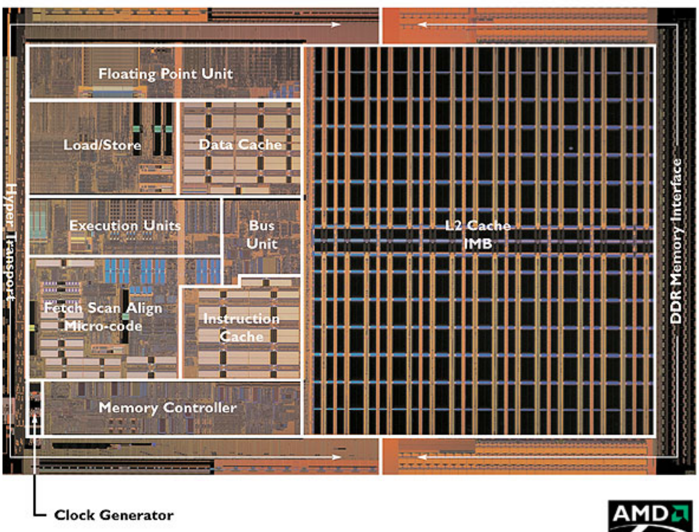
first-generation review quote:



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

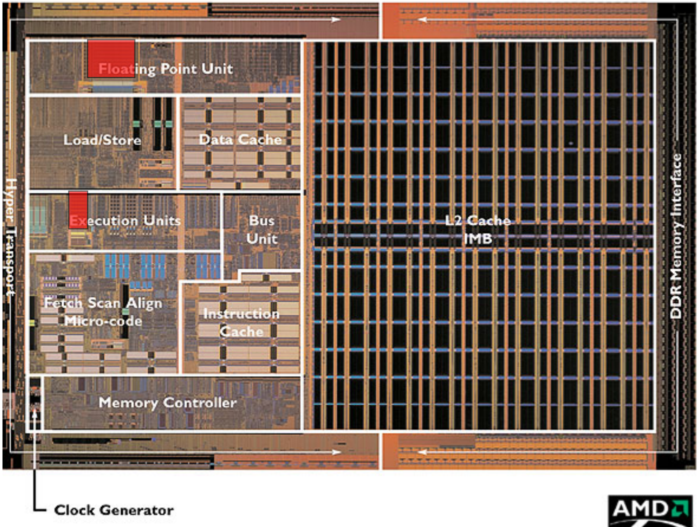
2004 CPU



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

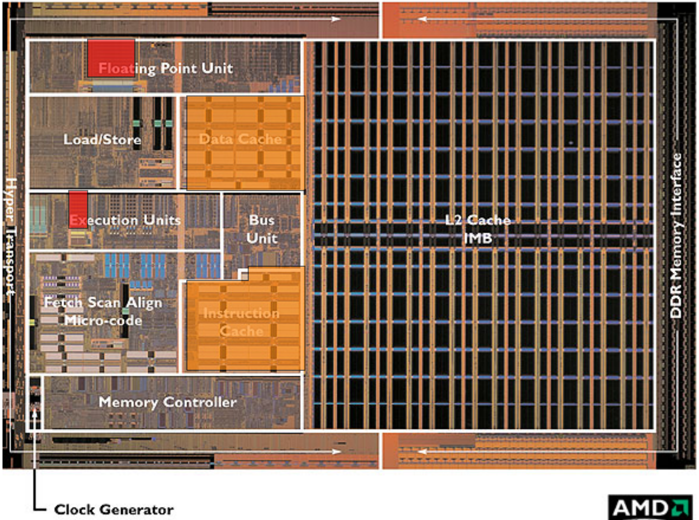
2004 CPU

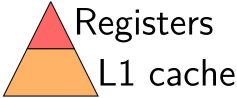
🔺 Registers



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

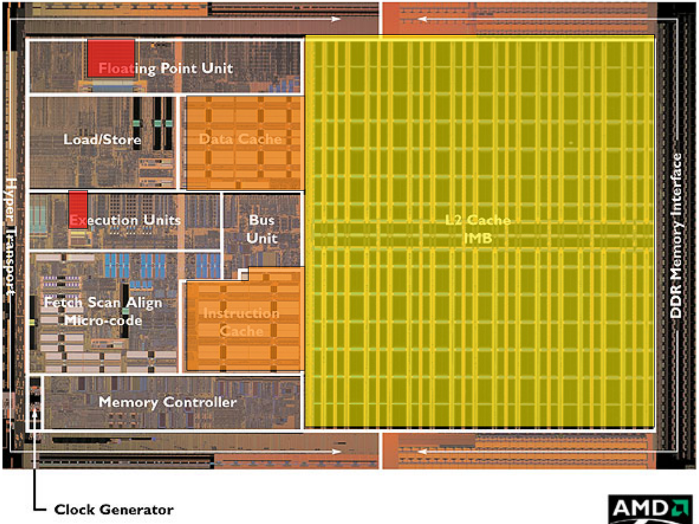
2004 CPU

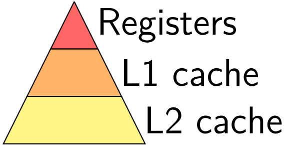




age: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU



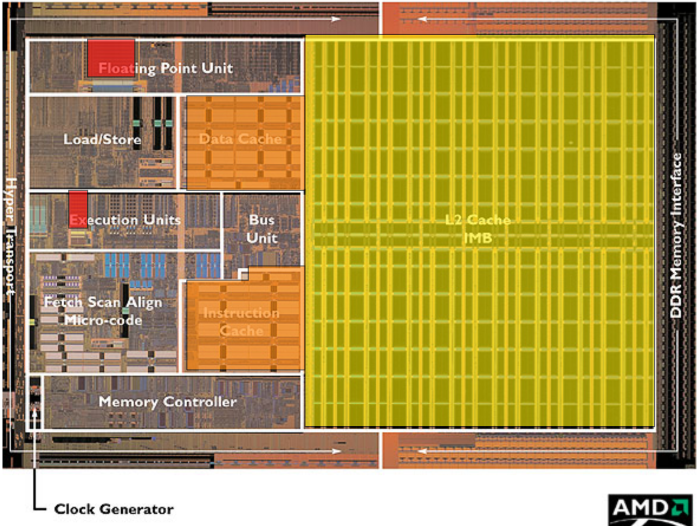

Registers

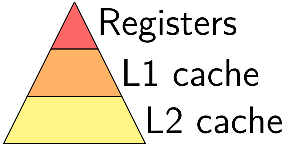
L1 cache

L2 cache

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

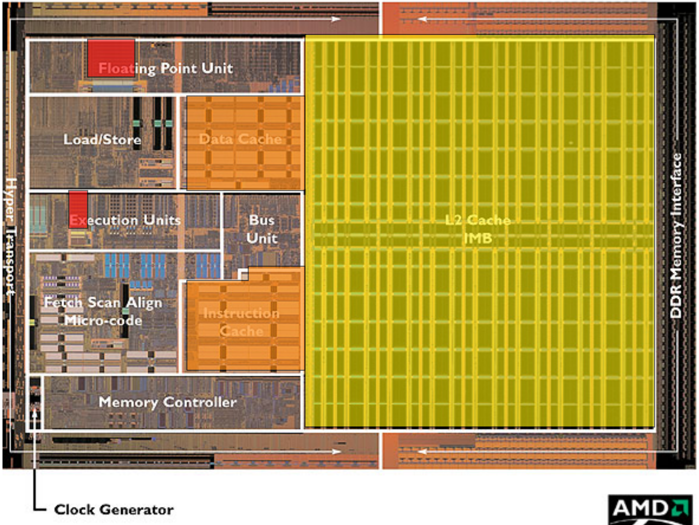
2004 CPU

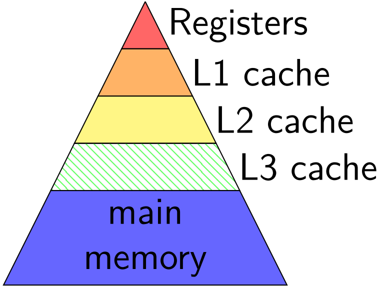




age: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU




Registers

L1 cache

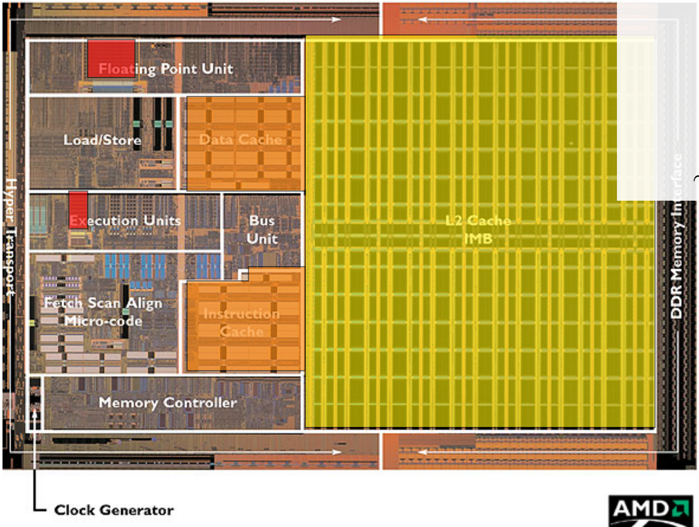
L2 cache

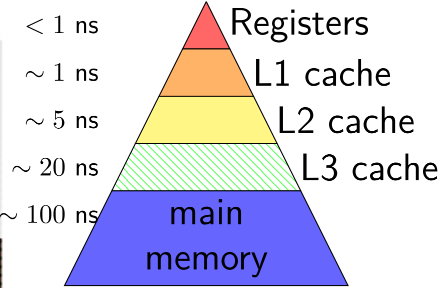
L3 cache

main memory

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

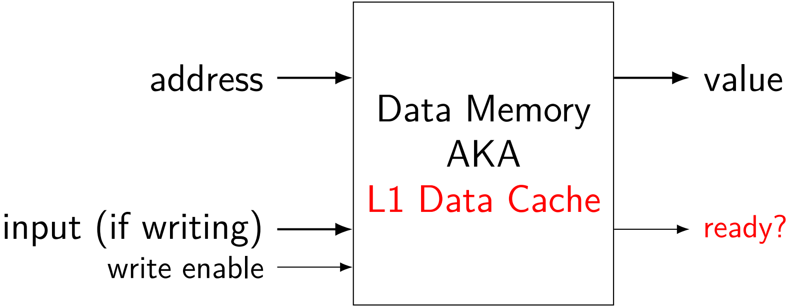
2004 CPU



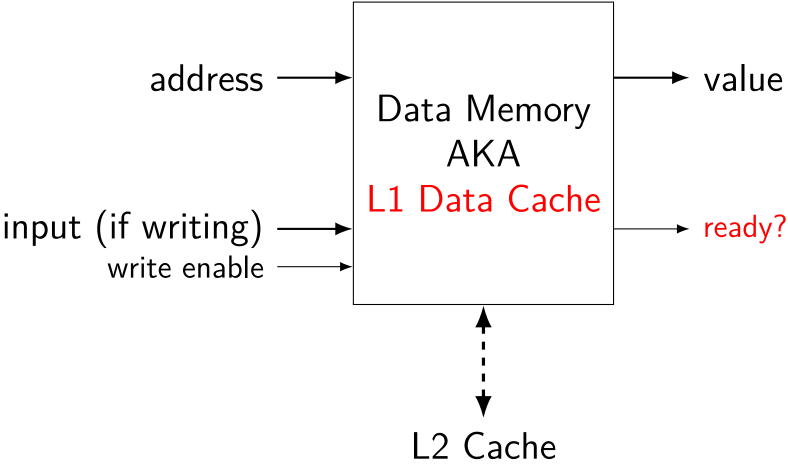


age: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

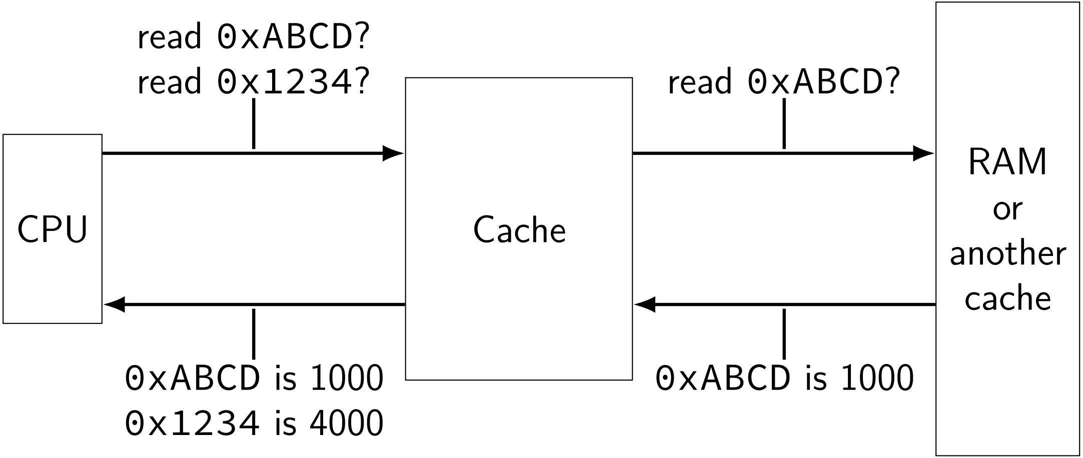
cache: real memory



cache: real memory



the place of cache



read 0xABCD?
read 0x1234?

read 0xABCD?

0xABCD is 1000
0x1234 is 4000

0xABCD is 1000

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]`

backup slides

HCLRS pipelining debugging: intro

debugging pipelines is consistently one of the biggest sources of difficulty in this class

HCL2D pipeline debugging (1)

draw a picture of the state of the instructions

get −d output

- redirect to a file

```
./hclrs file.hcl -d input.yo >output.txt
```

check each stage of the broken instruction

expect forwarding/hazard-handling problems

HCL2D pipeline debugging (2)

write assembly — not just supplied test cases

- write `file.ys` — `make file.yo` to assemble
- remove anything not involved in the error
- find a **minimal** test case
- don't spend time looking at irrelevant instructions

draw the pipeline stages

- what instructions are in fetch/decode/etc. when?
- what values should be in pipelien registers when?
- what forwarding should happen?