changelog

Changes since first lecture:

14 October 2021: add quiz review slides

last time

combining forwarding with stalling

forwarding with different pipelines

control hazards

- don't know correct PC value in time

branch prediction

- guess outcome of conditional jump
- correct wrong guess by "squashing" incorrect instructions

quiz Q1

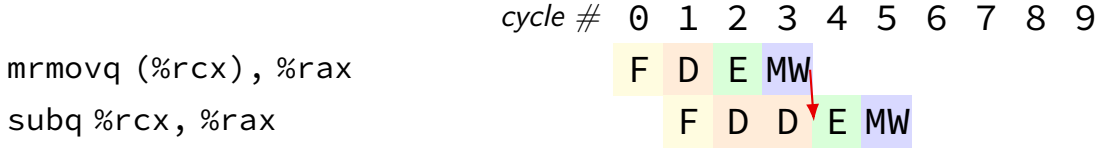
"Which of the following assembly snippets would exercise a data hazard if executed on the processor described above?"

data hazard = data dependency that causes problem with naive pipeline

need to know how instructions divided into stages

if processor handles hazard through stalling or forwarding:
still a hazard

quiz Q3



quiz Q4 (irmovq)

```
addq %r8, %r9
subq %r9, %r10
irmovq $42, %r9
```

forwarding to irmovq?

don't need it — irmovq doesn't read

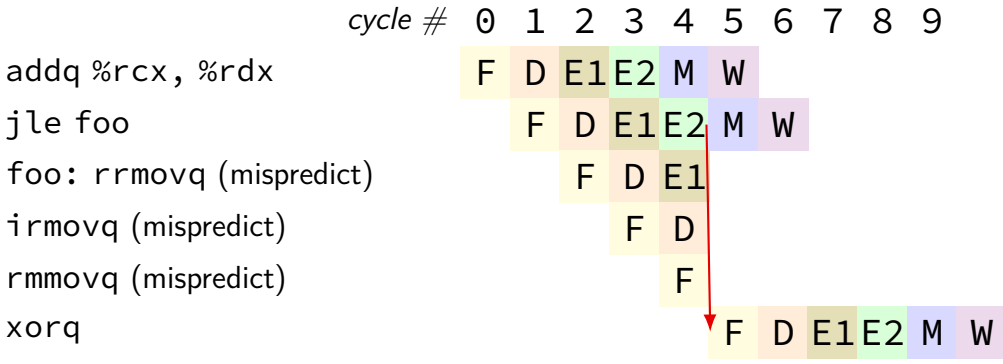
if you wanted to, how would you avoid in HCLRS?

- set reg_srcB to REG_NONE for irmovq (and similar)
- special case on forwarding MUX (not recommended)

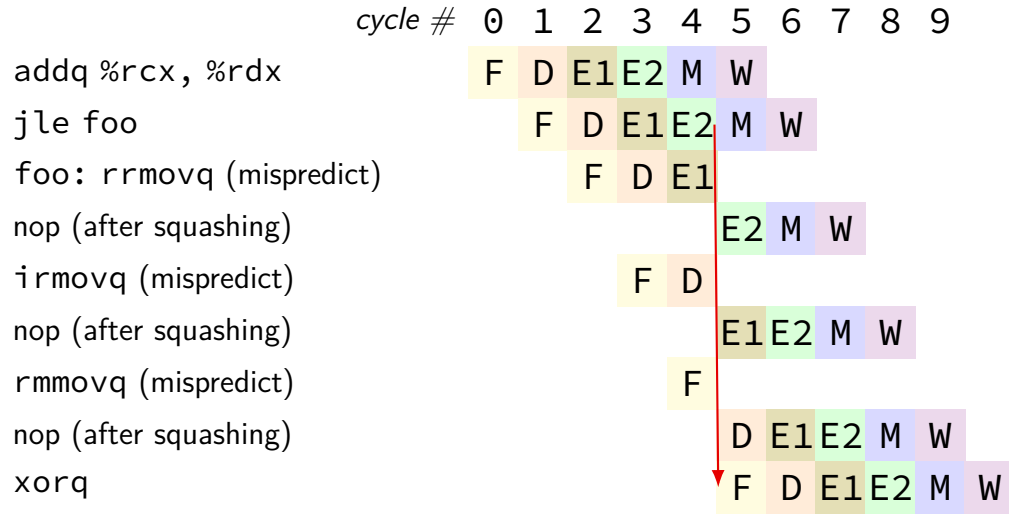
do we need to avoid?

- not in this case, but...
- don't want to end up stalling based on detecting false load/use hazard

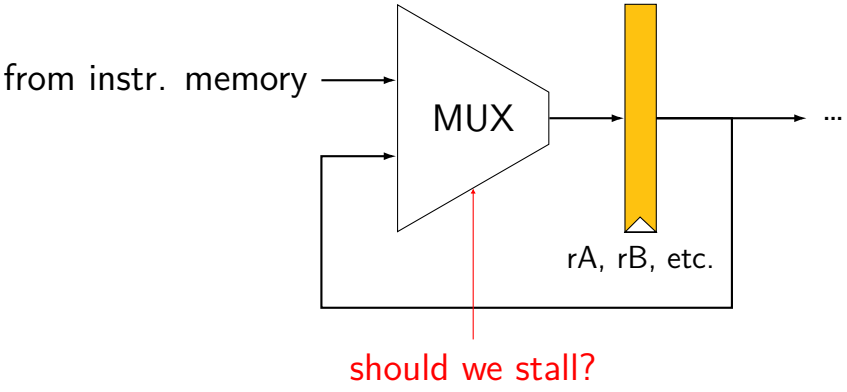
quiz Q5+6


cycle #
0 1 2 3 4 5 6 7 8 9
addq %rcx, %rdx
jle foo
foo: rrmovq (mispredict)
irmovq (mispredict)
rmmovq (mispredict)
xorq
F D E1 E2 M W
F D E1 E2 M W
F D E1
F D
F
F D E1 E2 M W

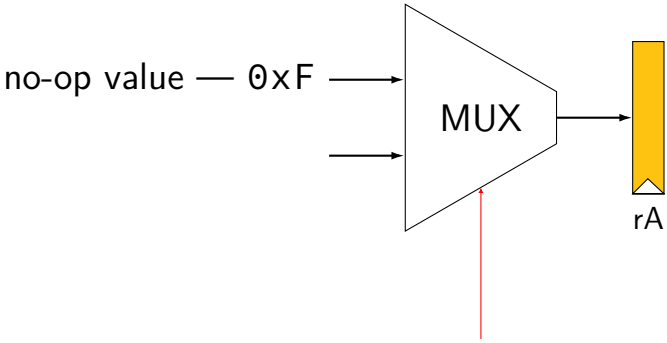
quiz Q5+6 (alt)


cycle #
0 1 2 3 4 5 6 7 8 9
addq %rcx, %rdx
F D E1 E2 M W
jle foo
F D E1 E2 M W
foo: rrmovq (mispredict)
F D E1
nop (after squashing)
E2 M W
irmovq (mispredict)
F D
nop (after squashing)
E1 E2 M W
rmmovq (mispredict)
F
nop (after squashing)
D E1 E2 M W
xorq
F D E1 E2 M W

fetch/decode 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
- pipeline: keep same instruction in this stage next cycle

`bubble_B`: use default value for all registers

- register input ← default value
- pipeline: put no-operation in this stage next cycle

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		???		

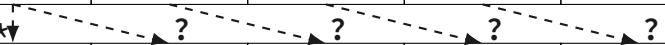
exercise result

```
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)

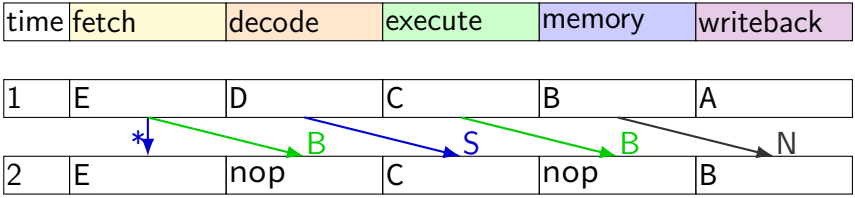
time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	E	nop	C	nop	B



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

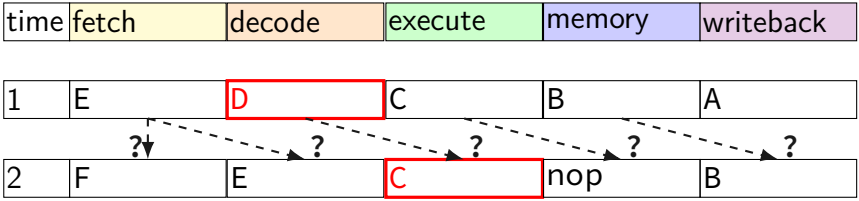
exercise: squash + stall (1)



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

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

exercise: squash + stall (2)

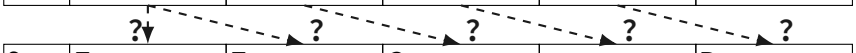


stall (S) = keep old value; normal (N) = use new value

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

exercise: squash + stall (2)

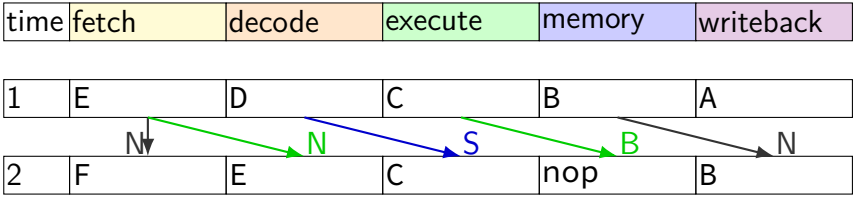
time	fetch	decode	execute	memory	writeback
1	E	D	C	B	A
2	F	E	C	nop	B



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)



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	

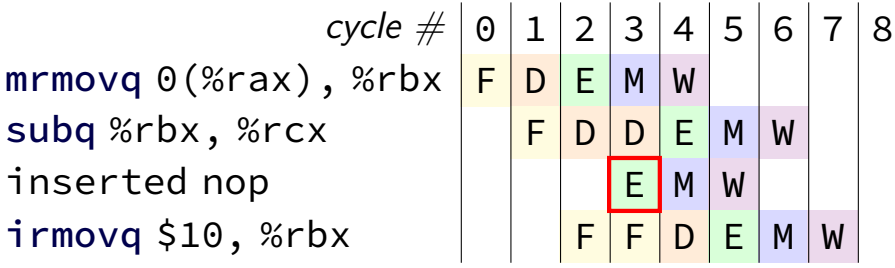
…

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

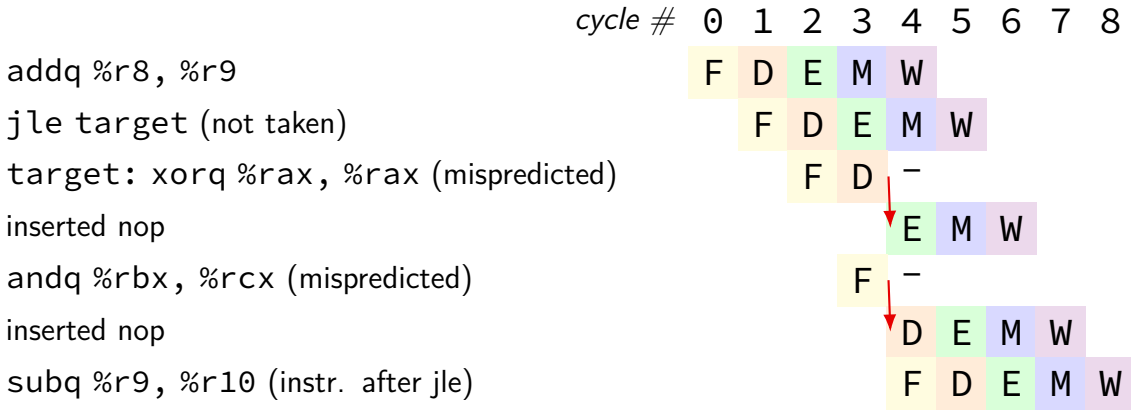


insert nop in cycle 3
during cycle 2:
bubble_E = 1

need way to keep pipeline register unchanged to repeat a stage

(*and* to replace instruction with a nop)

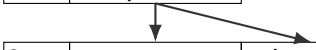
jump misprediction: bubbles



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

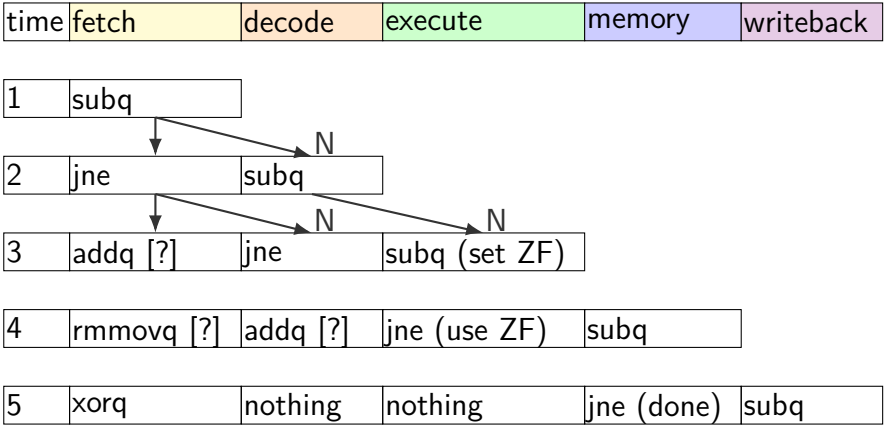
squashing with stall/bubble

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



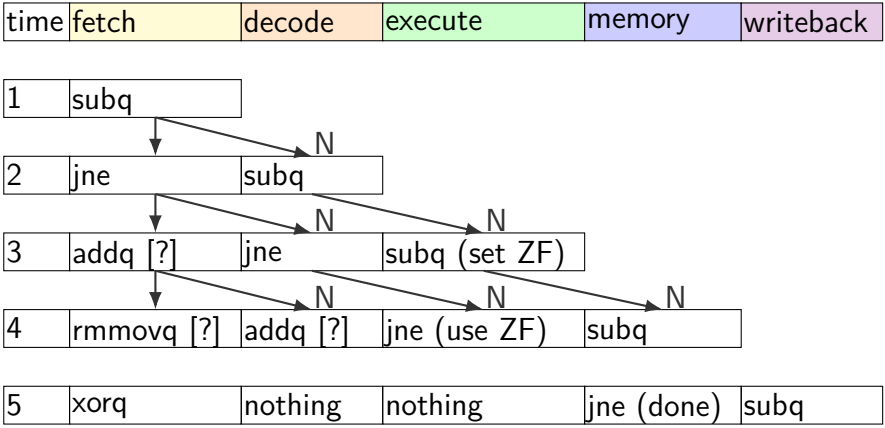
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

squashing with stall/bubble



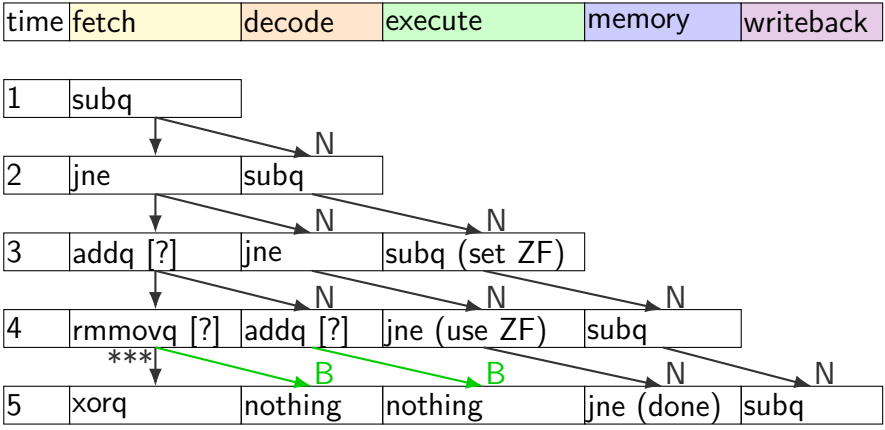
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

squashing with stall/bubble



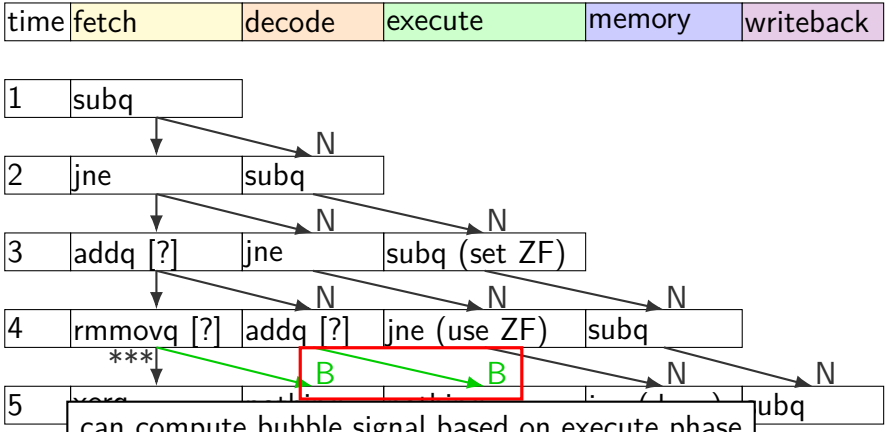
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

squashing with stall/bubble



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

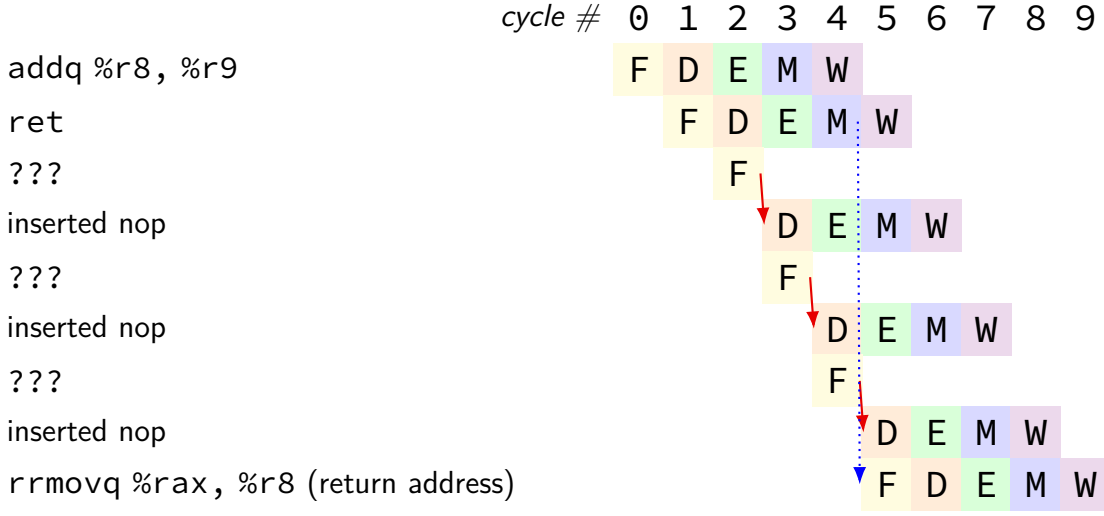
squashing with stall/bubble



squashing HCLRS

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

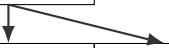
ret bubbles



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

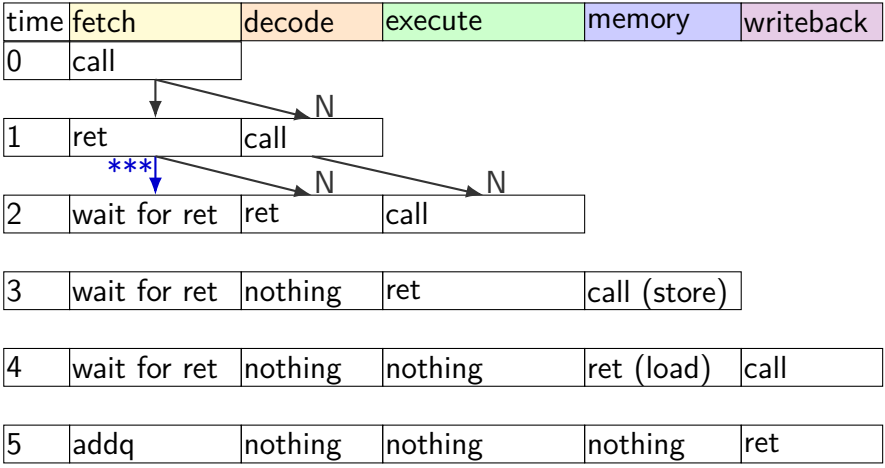
ret stall

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);

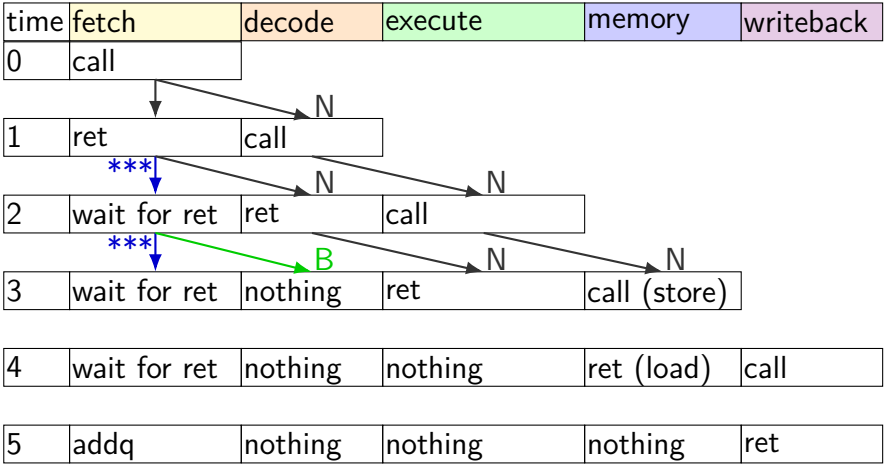
ret stall



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

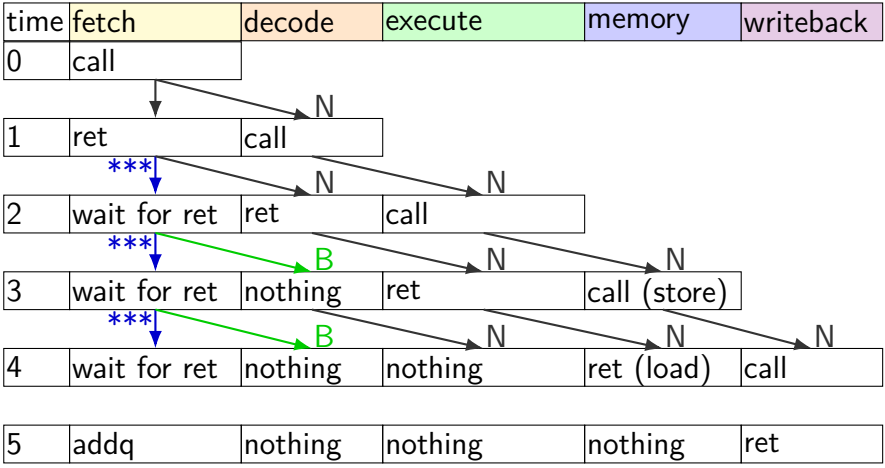
bubble (B) = use default (no-op);

ret stall



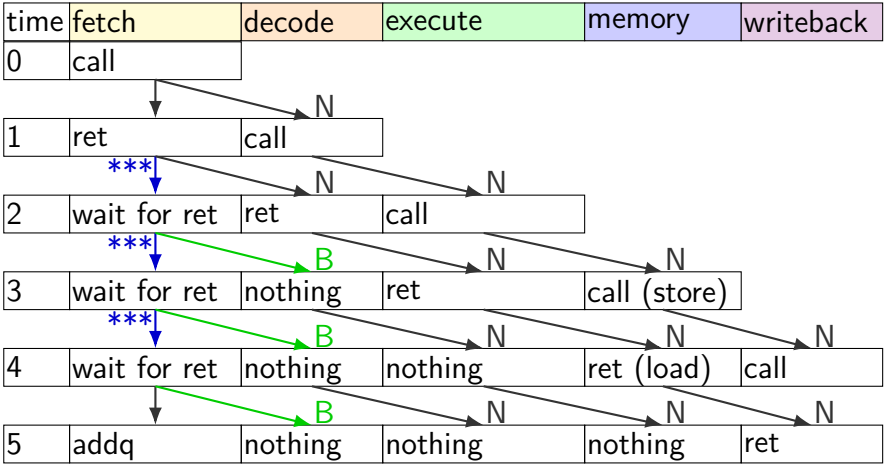
stall (S) = keep old value; normal (N) = use new value
bubble (B) = use default (no-op);

ret stall



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

ret stall



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

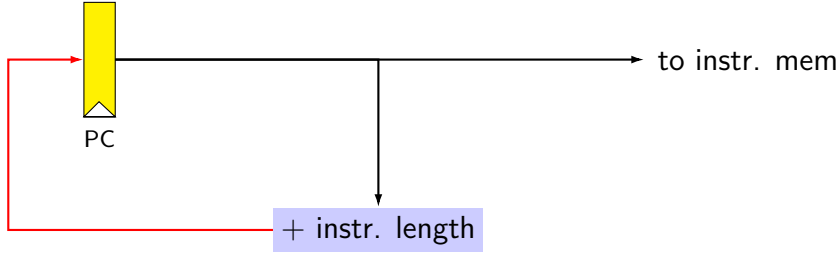
HCLRS bubble example

```
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)

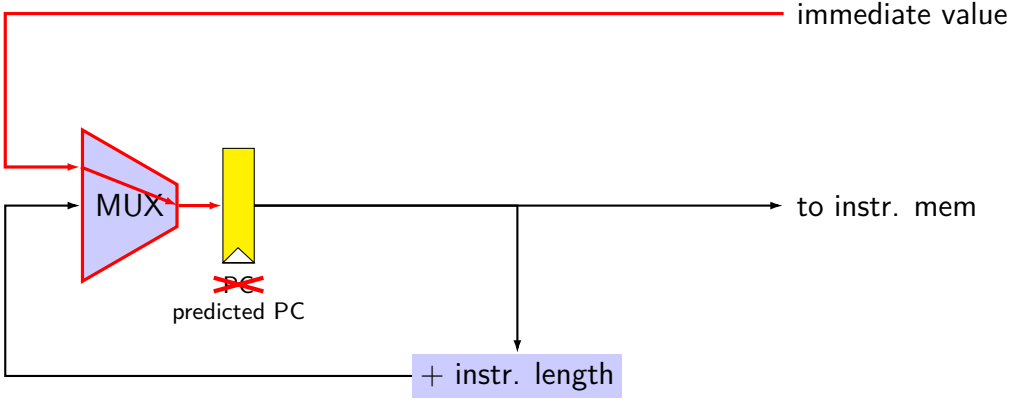
(1) normal case: PC $\leftarrow$ PC + instr len



building the PC update (one possibility)

(1) normal case: PC $\leftarrow$ PC + instr len

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

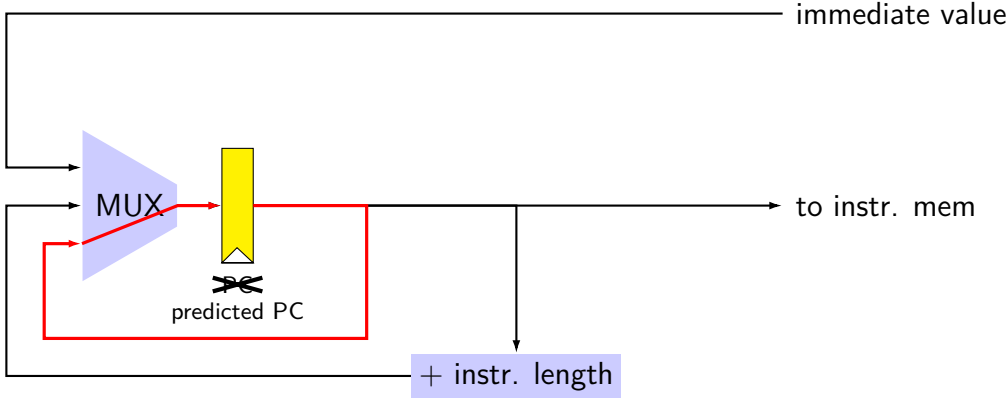


building the PC update (one possibility)

(1) normal case: PC $\leftarrow$ PC + instr len

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

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



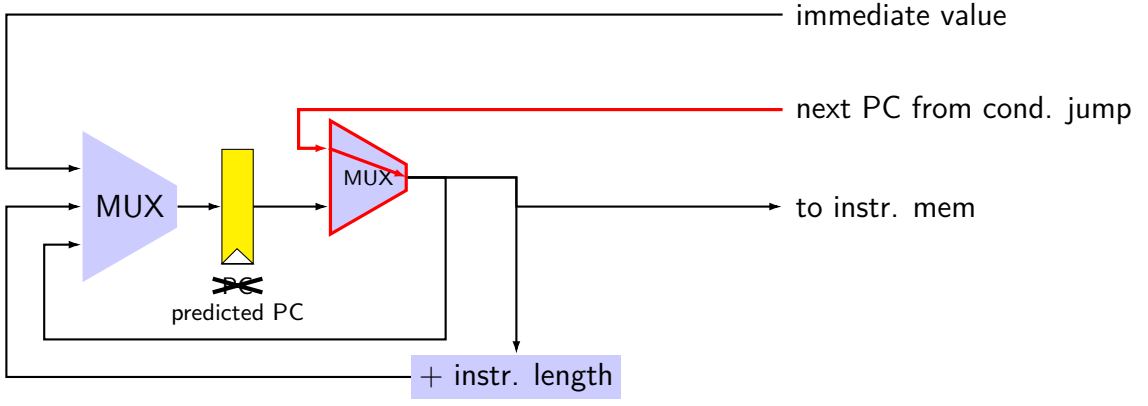
building the PC update (one possibility)

(1) normal case: PC $\leftarrow$ 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)

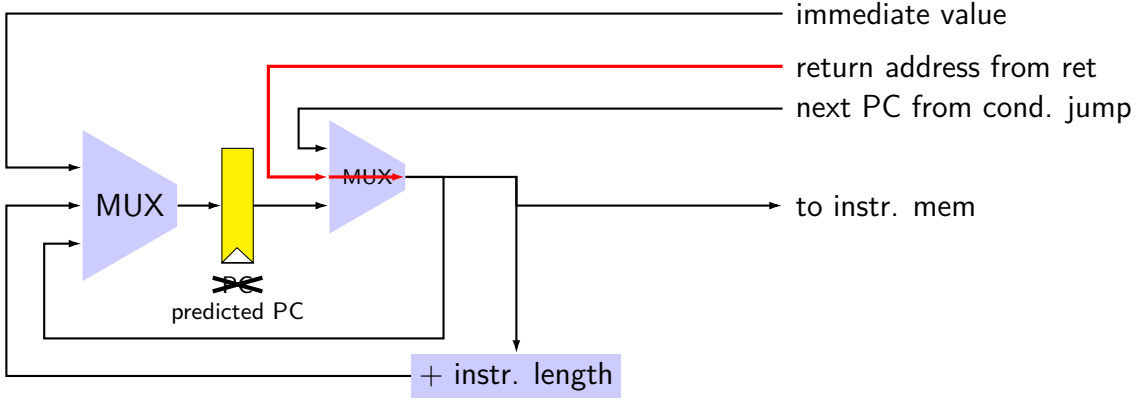
(1) normal case: PC $\leftarrow$ PC + instr len

(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)

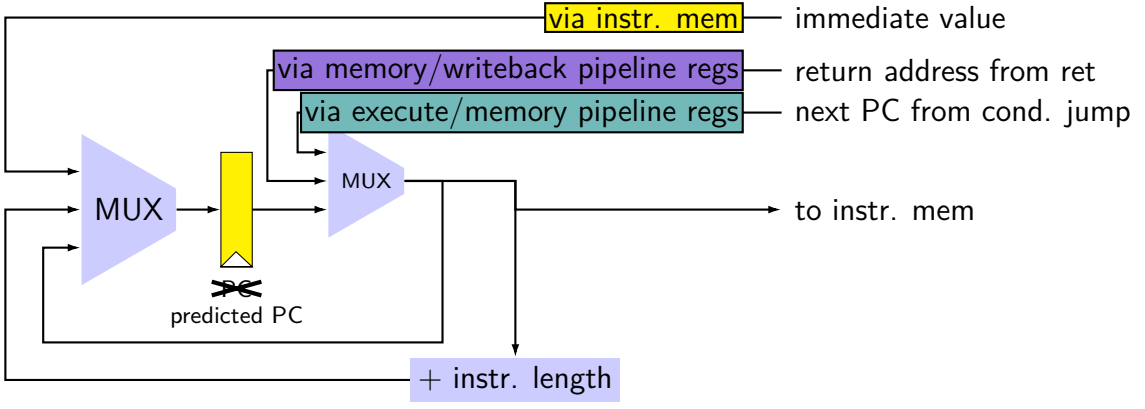
(1) normal case: PC $\leftarrow$ PC + instr len

(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

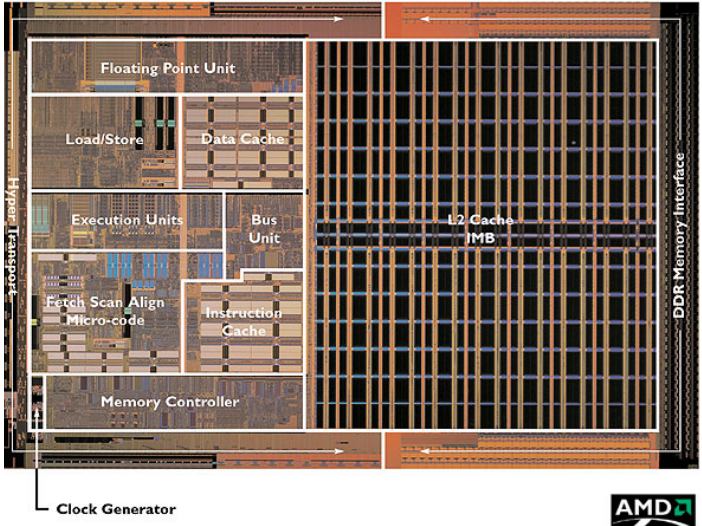
but earlier — had to talk to memory on different chip

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?

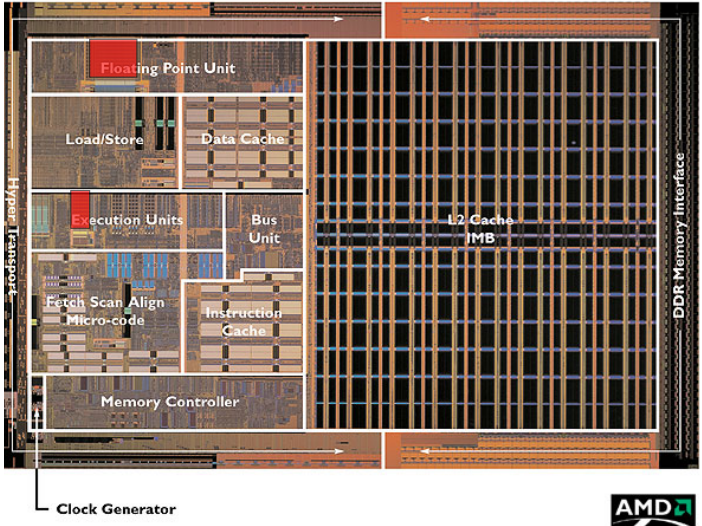
2004 CPU



AMD 64 Opteron

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

2004 CPU



Registers

AMD 64 Opteron

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

2004 CPU

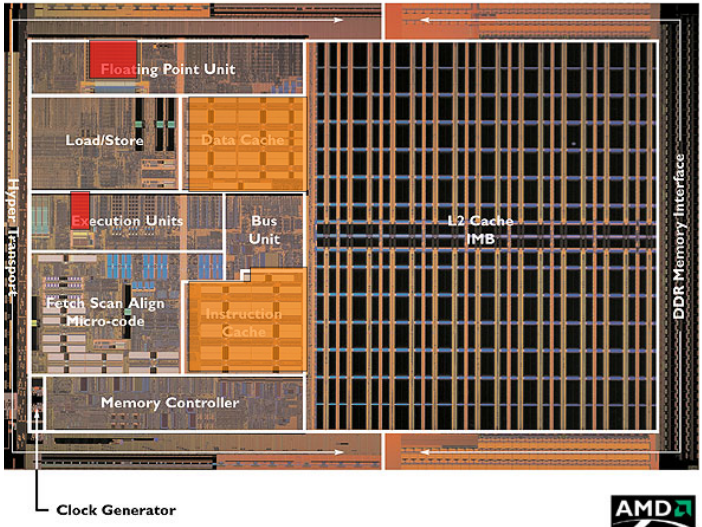
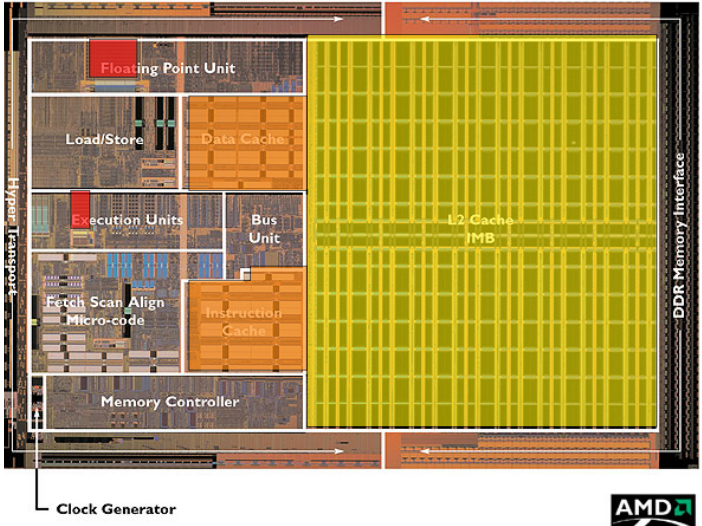
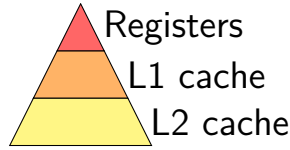


Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

2004 CPU

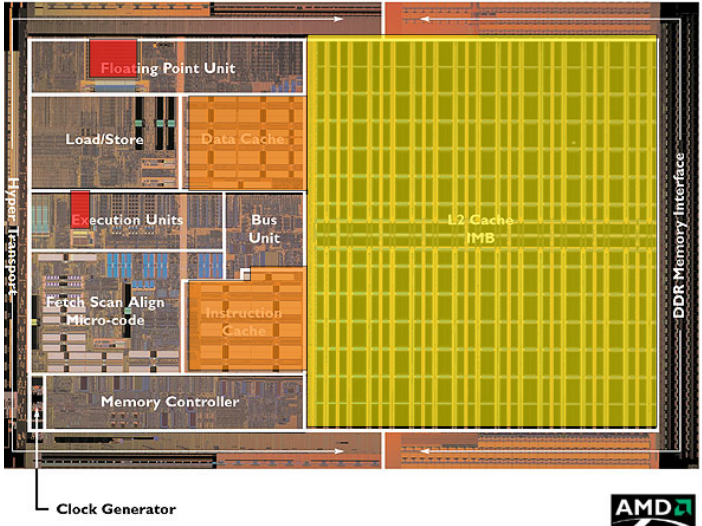


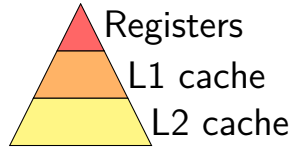


AMD 64 Opteron

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

2004 CPU

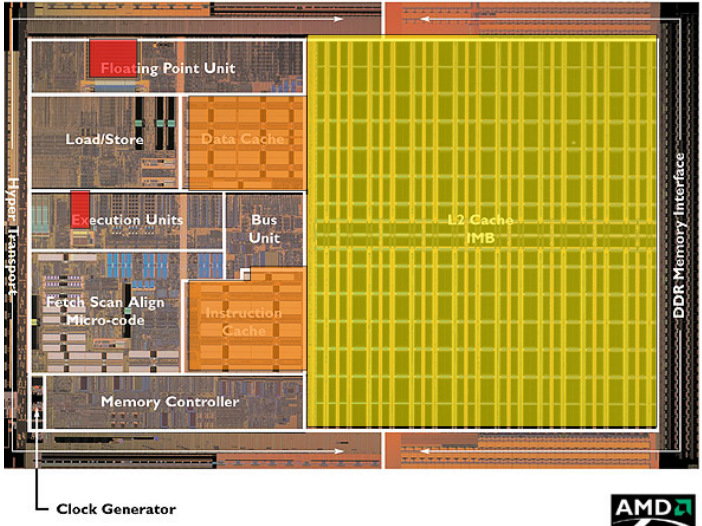


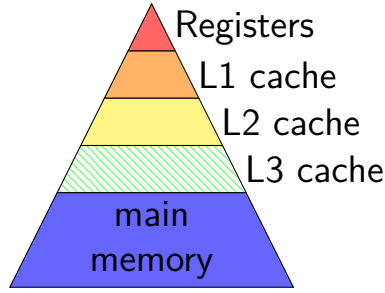


AMD 64 Opteron

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

2004 CPU

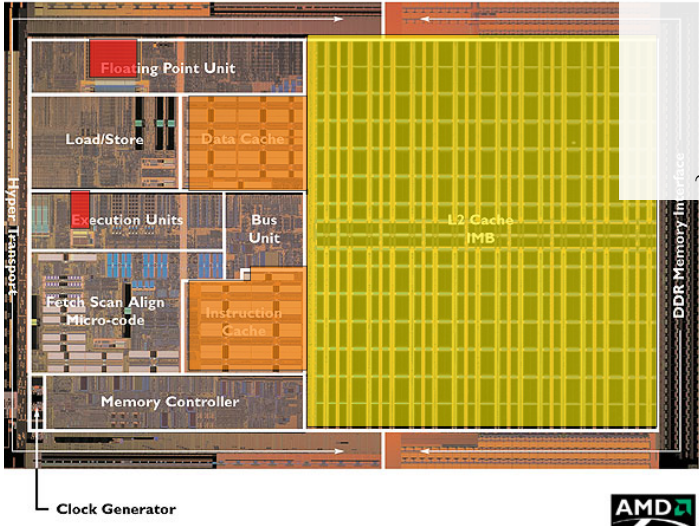


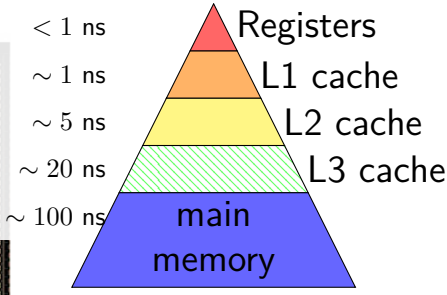


AMD 64 Opteron

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

2004 CPU

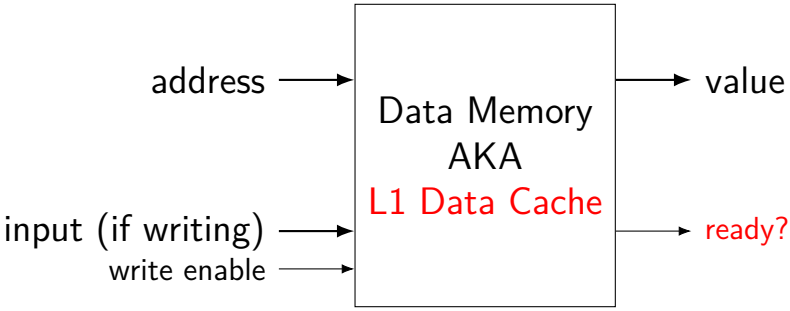




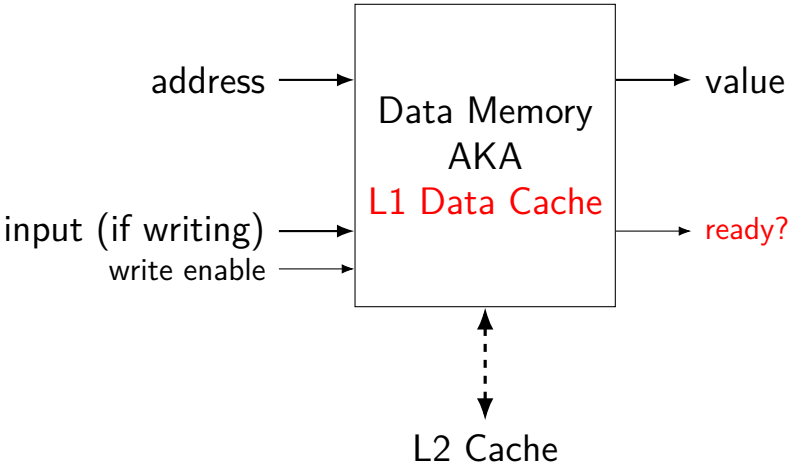
AMD 64 Opteron

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

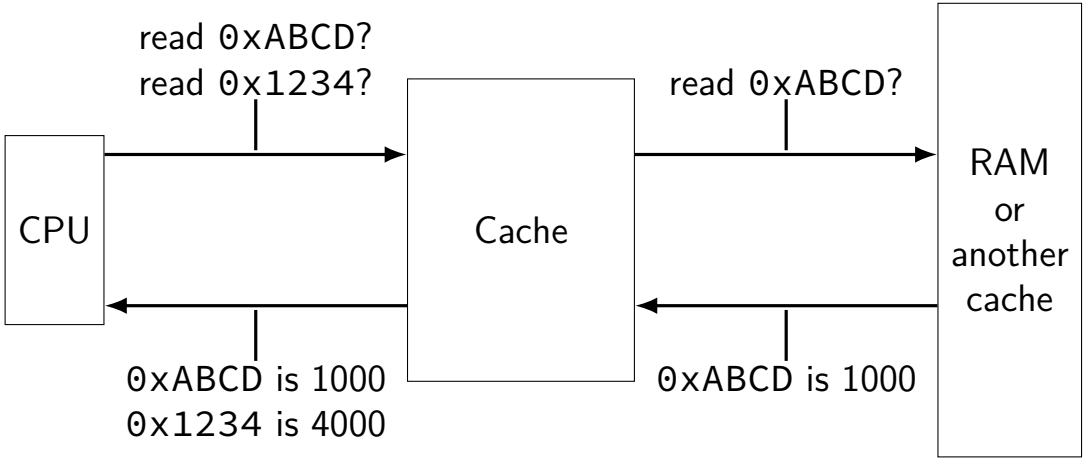
cache: real memory



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

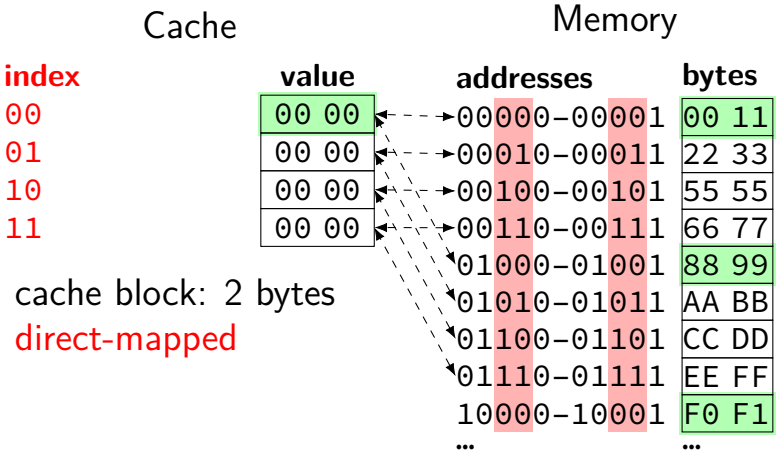
Memory

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?

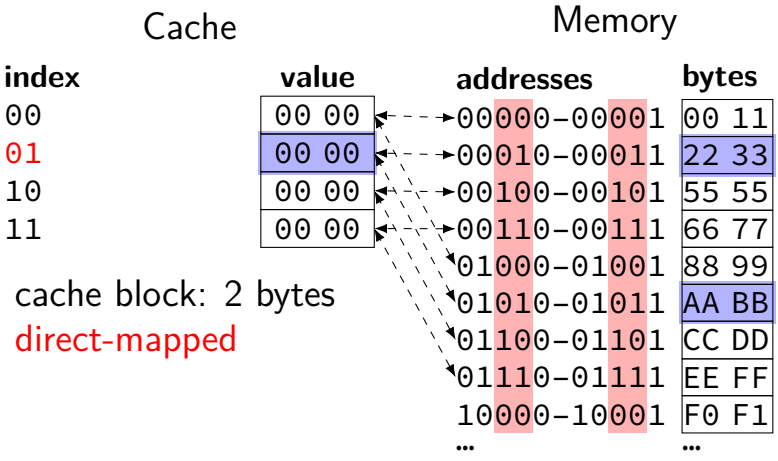
exactly one place for each address
spread out what can go in a block



building a (direct-mapped) cache

read byte at 01011?

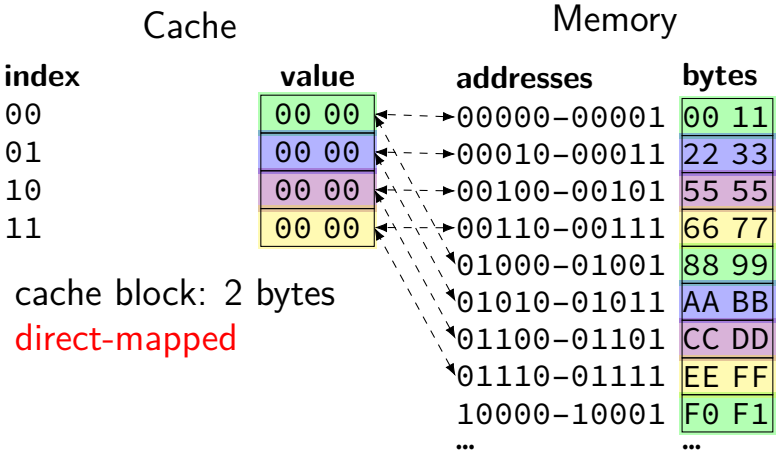
exactly one place for each address
spread out what can go in a block



building a (direct-mapped) cache

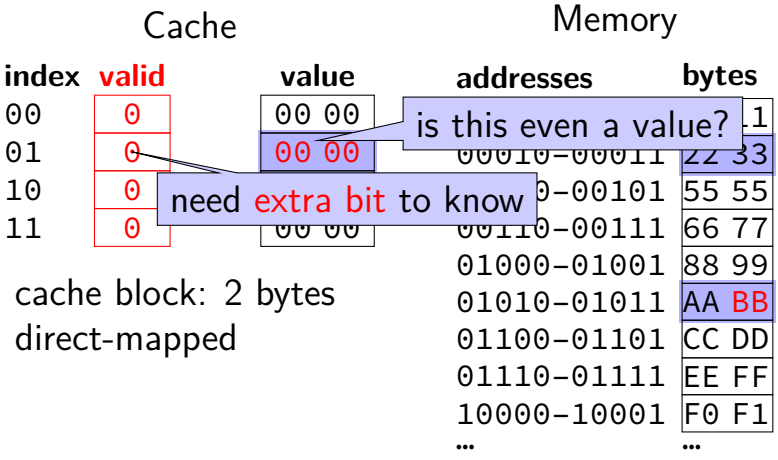
read byte at 01011?

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



building a (direct-mapped) cache

read byte at 01011?



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

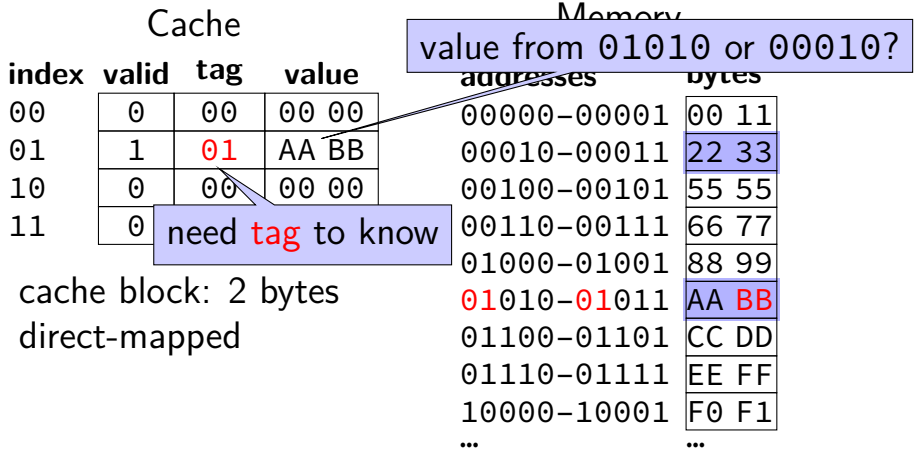
Memory

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 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

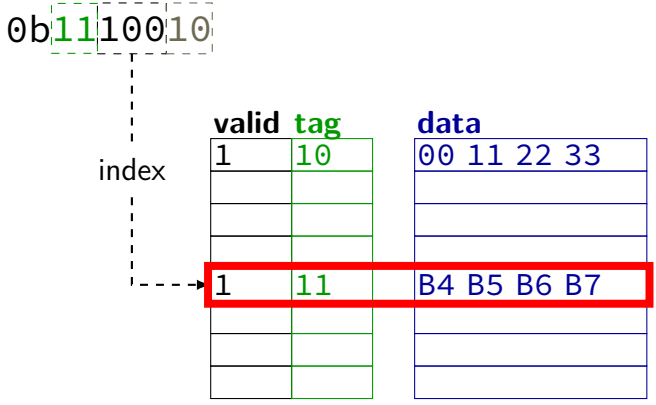
cache block: 2 bytes

direct-mapped

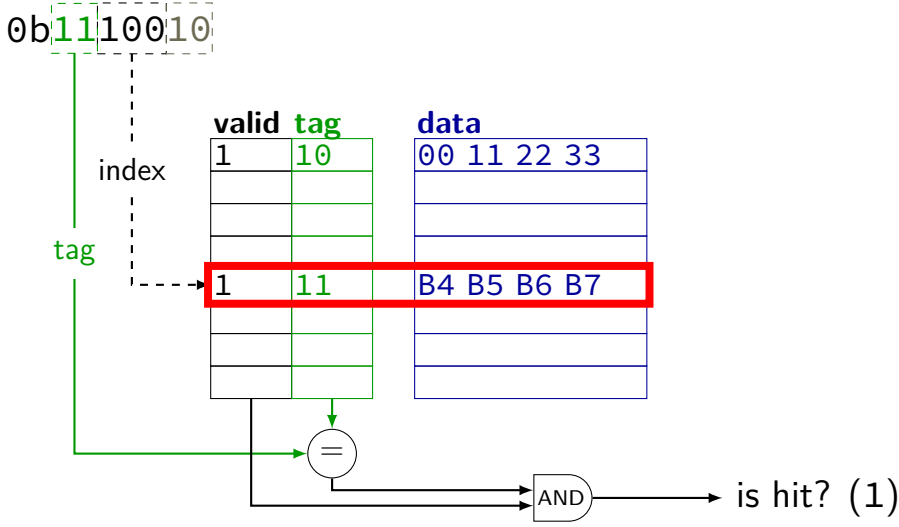
Memory

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
…	…

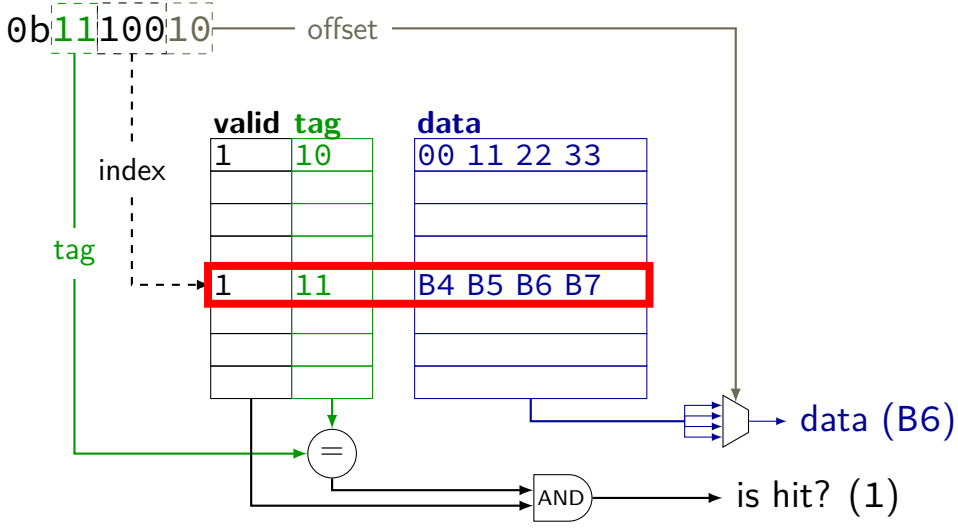
cache operation (read)



cache operation (read)

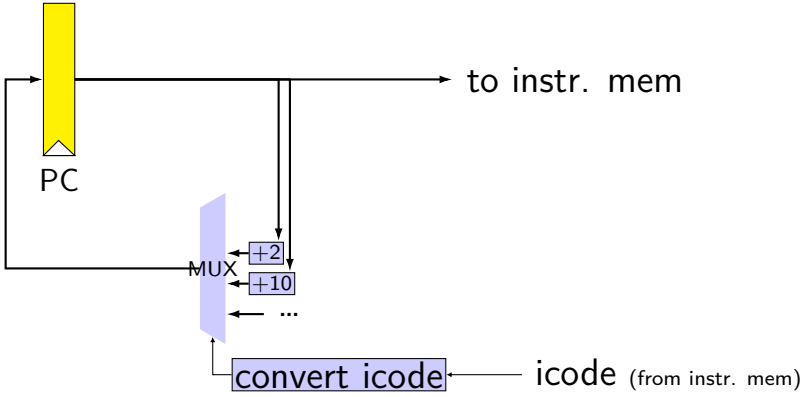


cache operation (read)

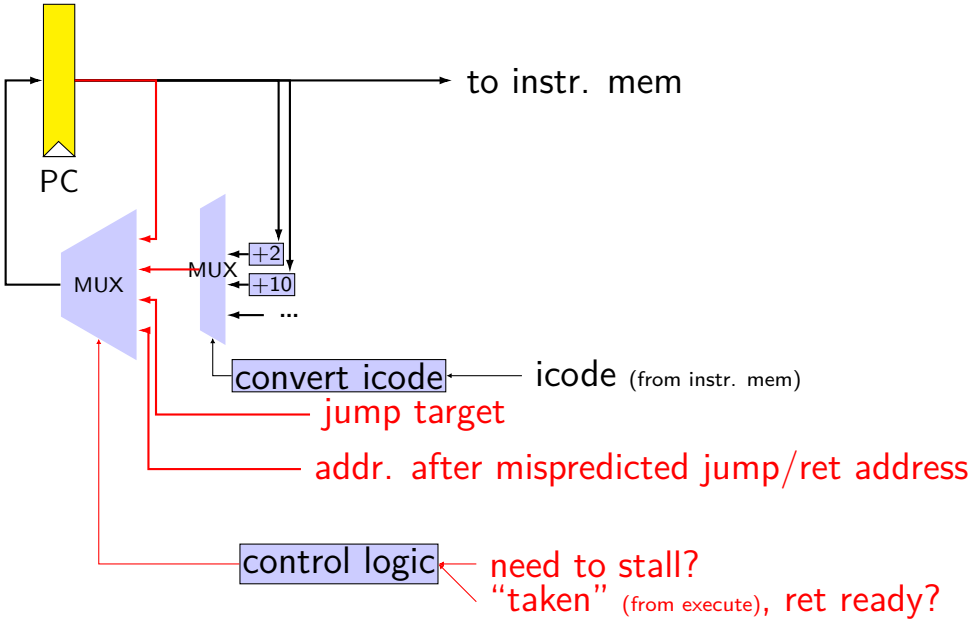


backup slides

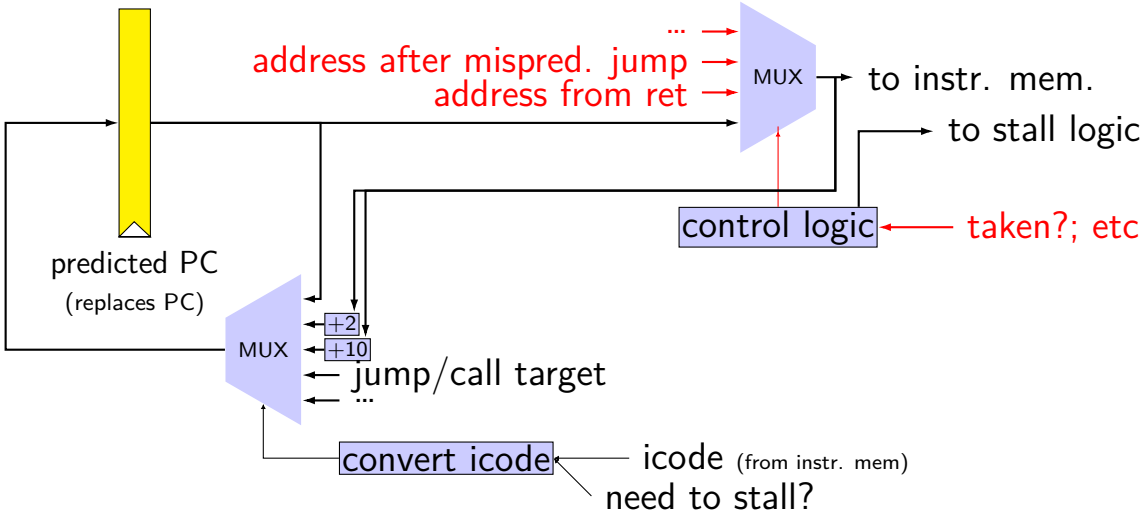
PC update (adding prediction, stall)



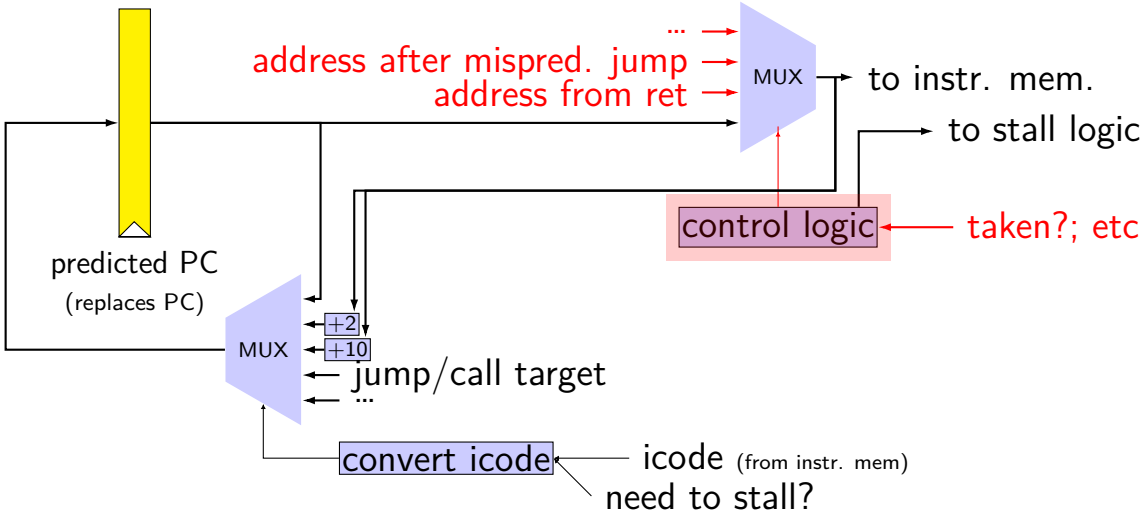
PC update (adding prediction, stall)



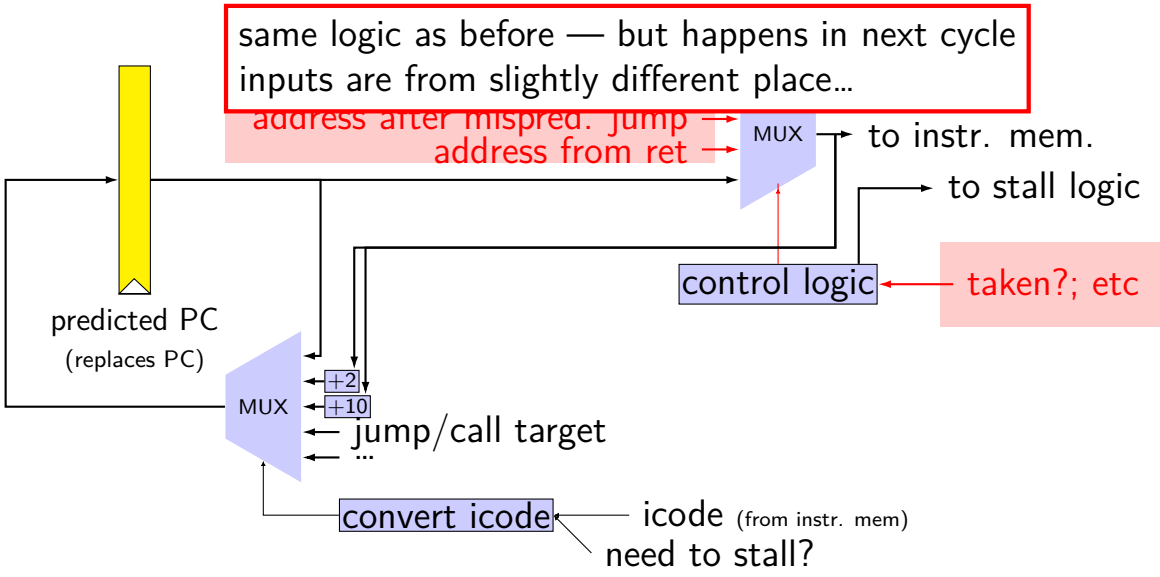
PC update (rearranged)



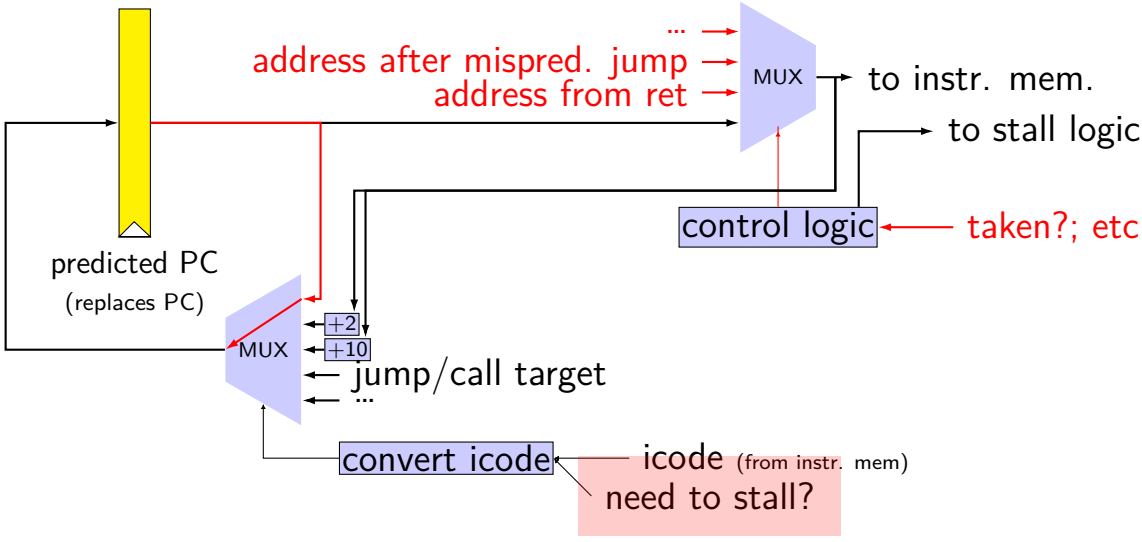
PC update (rearranged)



PC update (rearranged)



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;
];
```

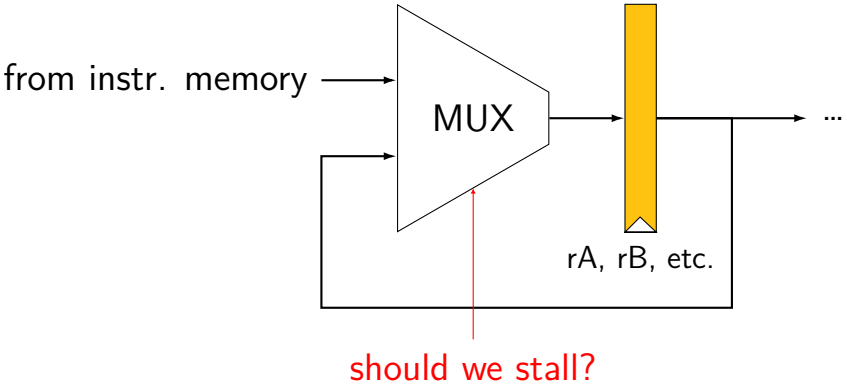
why rearrange PC update?

either works

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

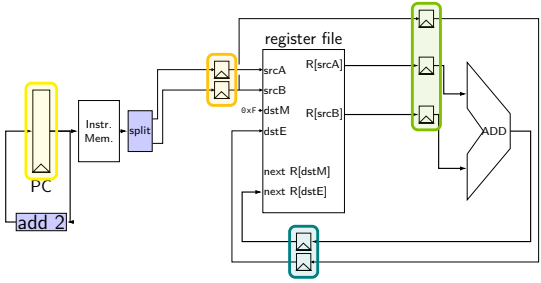
maybe easier to think about branch prediction this way?

fetch/decode logic — advance or not



exercise

path	time
add 2	50 ps
instruction memory	200 ps
register file read	125 ps
add	100 ps
register file write	125 ps



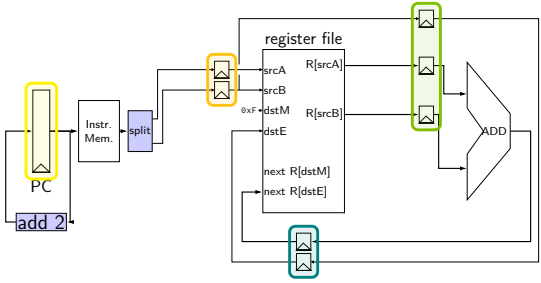
pipeline register delay: 10ps

how will throughput improve if we double the speed of the instruction memory?

A. 2.00x **B.** 1.70x to 1.99x
C. 1.60x to 1.69x **D.** 1.50x to 1.59x
E. less than 1.50x

exercise

path	time
add 2	50 ps
instruction memory	200 ps
register file read	125 ps
add	100 ps
register file write	125 ps



pipeline register delay: 10ps

how will throughput improve if we double the speed of the instruction memory?

A. 2.00x **B.** 1.70x to 1.99x
C. 1.60x to 1.69x **D.** 1.50x to 1.59x
E. less than 1.50x

$$\frac{1}{135} \div \frac{1}{210} = 1.56\text{x} \text{ — D}$$