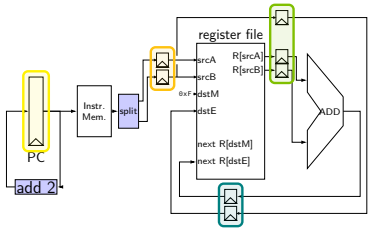
CS3330: PIPE

Last time: data hazard stall

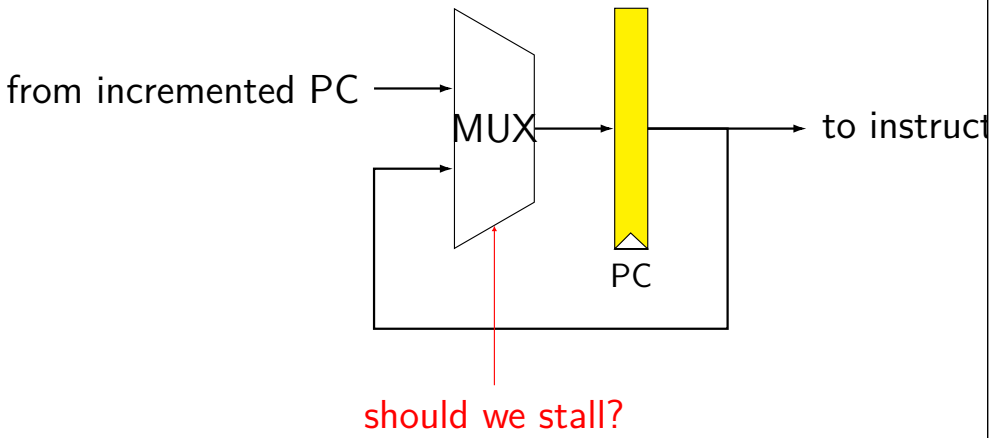
```
// initially %r8 = 800,
//           %r9 = 900, etc.
addq %r8, %r9
// hardware stalls twice
addq %r9, %r8
```



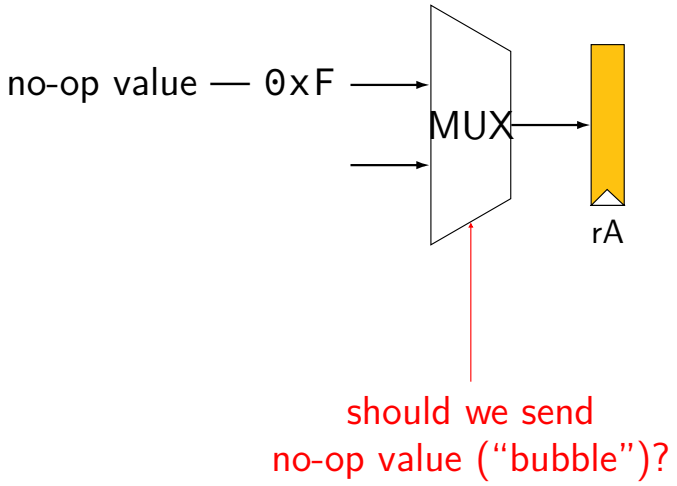
	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2*	8	9					
2	0x2*	F	F	800	900	9		
3	0x2	F	F	---	---	F	1700	9
4		9	8	---	---	F	---	F
5				1700	800	8	---	F
6							2500	8

R[9] written during cycle 3; read during cycle 4

fetch/fetch logic — advance or not



fetch/decode logic — bubble or not



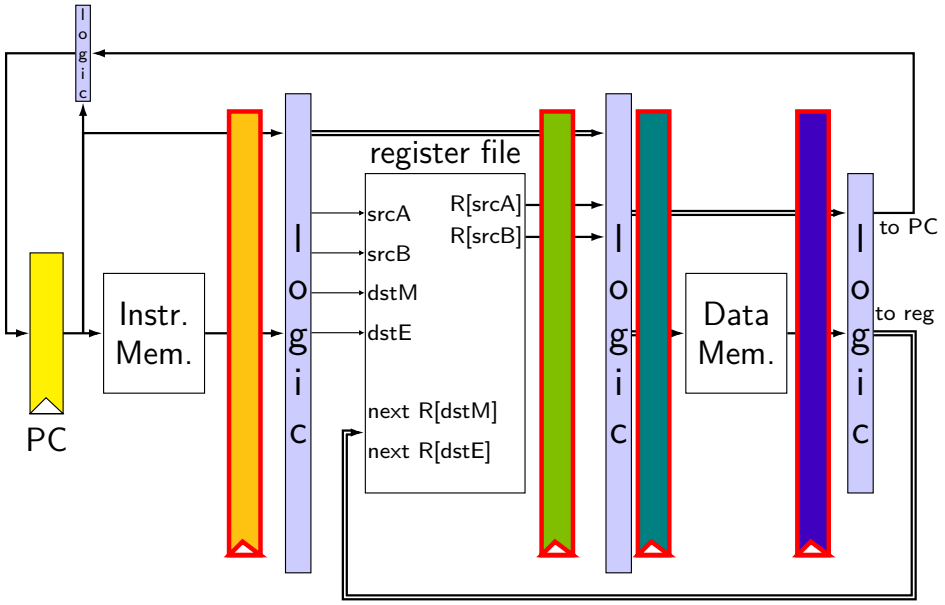
preview: HCL2D shortcuts

HCL2D provides these MUXes for you — for every register bank

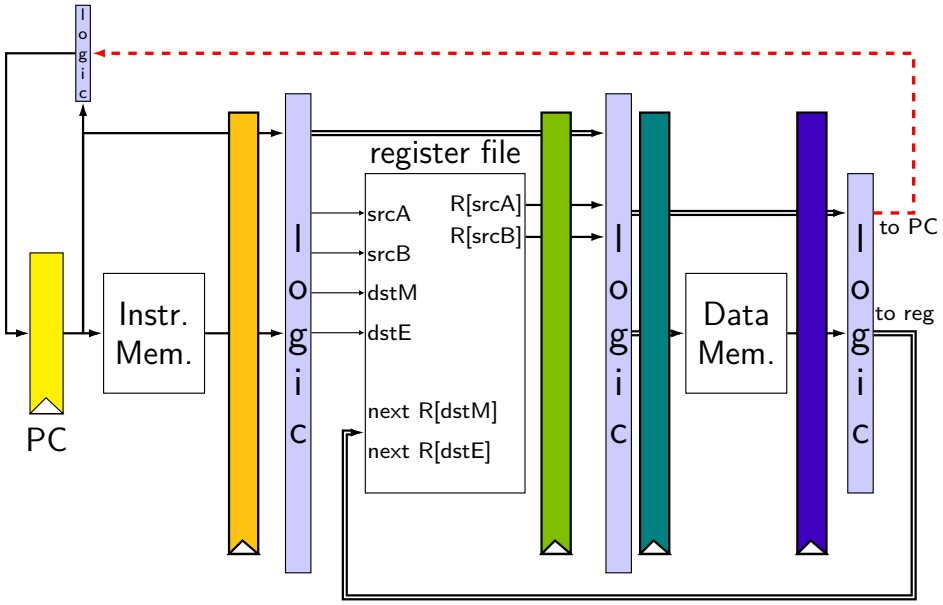
controlled by "bubble" and "stall" signals

more Thursday/next Tuesday

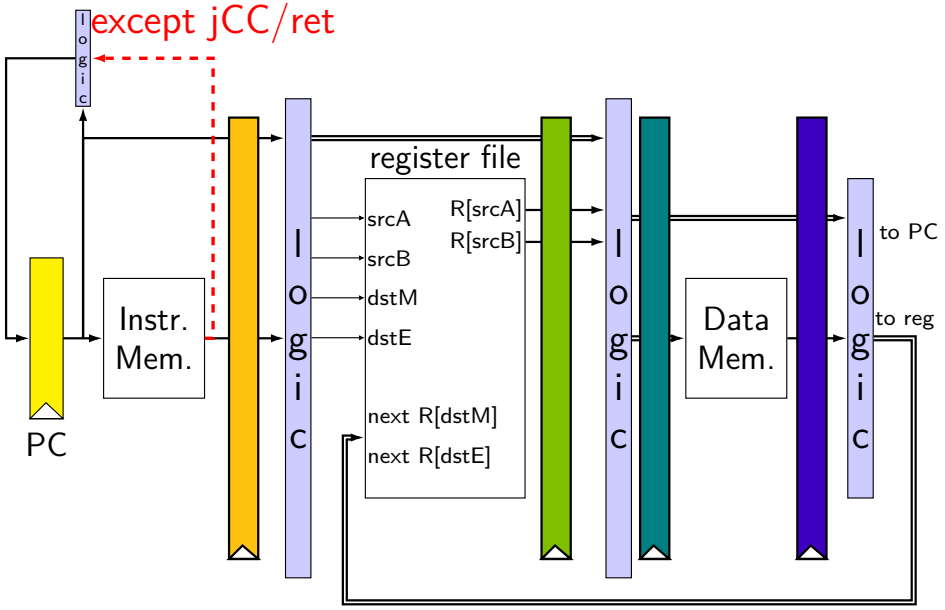
SEQ + pipeline registers



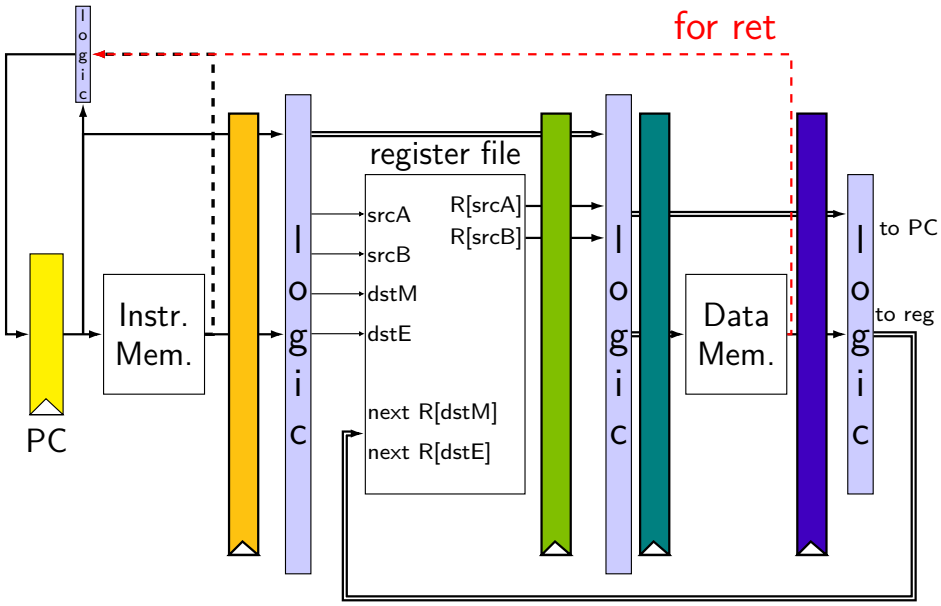
SEQ + pipeline registers



SEQ + pipeline registers



SEQ + pipeline registers



Stages

fetch — instruction memory, *most* PC computation

decode — reading register file

execute — computation, condition code reading and writing

memory — memory read/write

writeback — writing register file, writing Stat register

Stages

fetch — instruction memory, *most* PC computation

decode — reading register file

execute — computation, condition code reading and writing

read/write in same stage avoids data hazards
get value updated for prior instruction

writeback — writing register file, writing Stat register

Stages

fetch — instruction memory, *most* PC computation

decode — reading register file

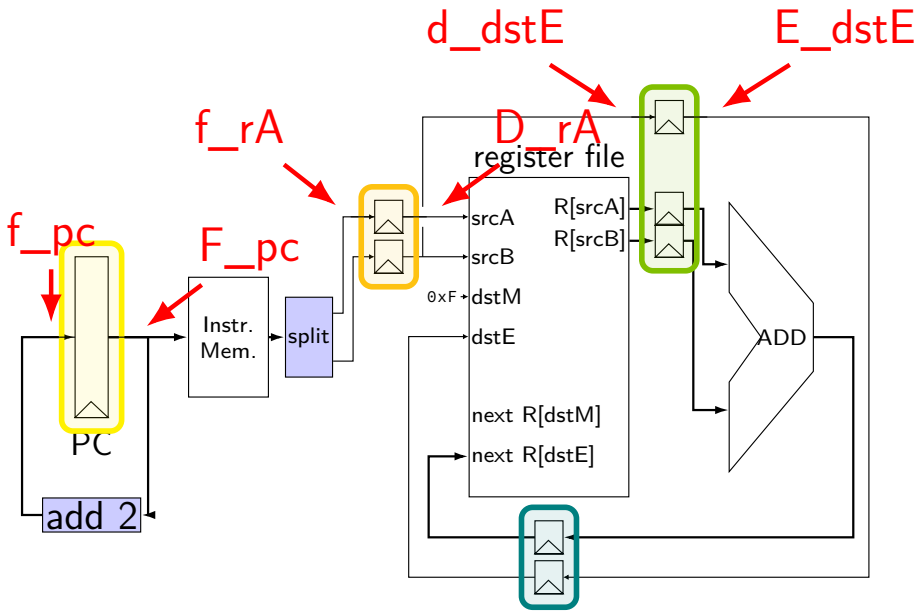
execute — computation, condition code reading and writing

memory — memory read/write

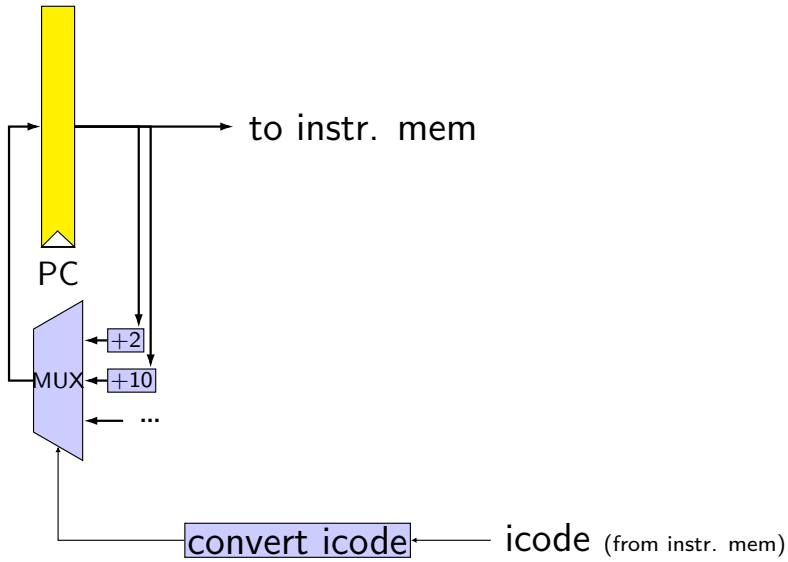
writeback — writing register file, writing Stat register

don't want to halt until everything else is don

pipeline register naming convention



normal PC update: logic

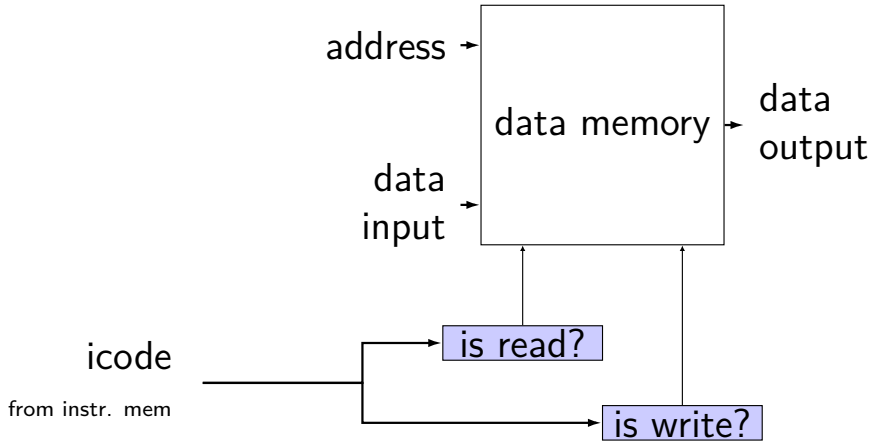


simple PC update: code

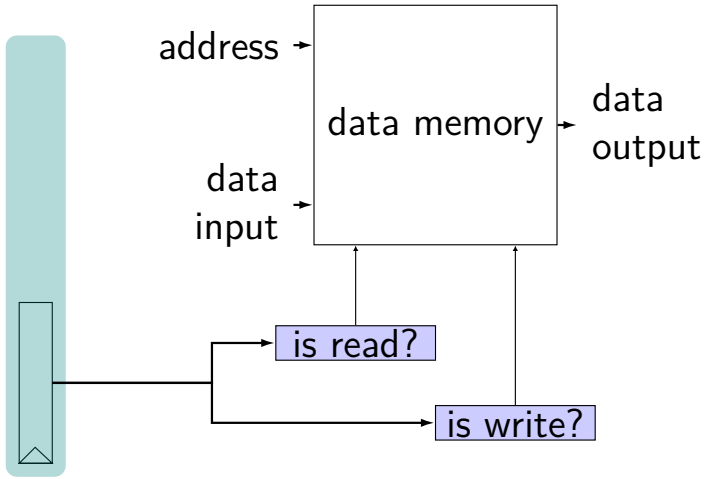
last week's lab...

```
icode = i10bytes[0..4];
f_pc = [
    icode == ADD || ...: F_pc + 2;
    icode == IRMOVQ || ...: F_pc + 10;
    ...
];
```

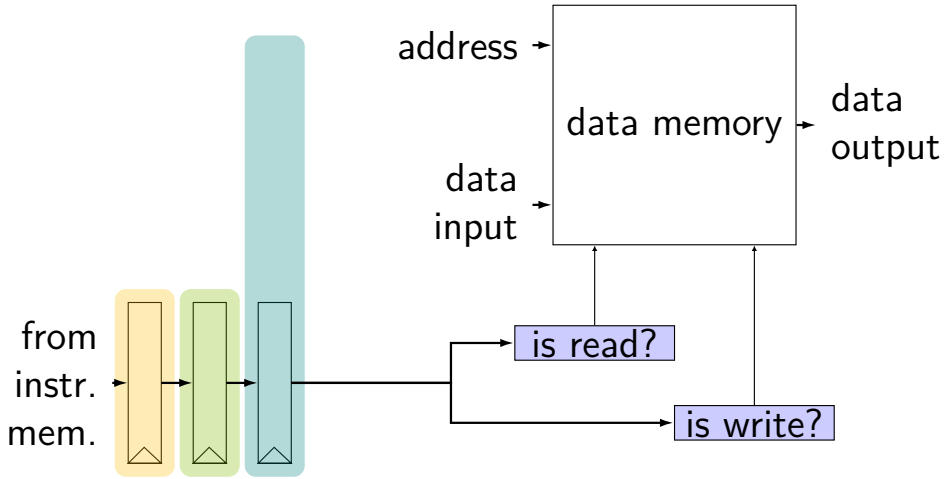

memory read/write logic



memory read/write logic



memory read/write logic



memory read/write: SEQ code

```
icode = i10bytes[4..8];
mem_readbit = [
    icode == MRMOVQ || ...: 1;
    0;
];
```


memory read/write: PIPE code

```
f_icode = i10bytes[4..8];
register fD { /* and dE and eM and mW */
    icode : 4 = NOP;
}
d_icode = D_icode
...
e_icode = E_icode;
mem_readbit = [
    M_icode == IMRMOVQ || ...: 1;
    0;
];
```

memory read/write: PIPE code

```
f_icode = i10bytes[4..8];
register fD { /* and dE and eM and mW */
    icode : 4 = NOP;
}
d_icode = D_icode
...
e_icode = E_icode;
mem_readbit = [
    M_icode == IMRMOVQ || ...: 1;
    0;
];
```


in general

will always pass icode in pipeline registers

control logic (often not drawn) will use it

examples:

- register number selection
- ALU input selection
- stalling

coding pipeline stages

use only prior stage's outputs

e.g. decode stage: get from fetch (`D_icode`, ...)

set only inputs for next stage

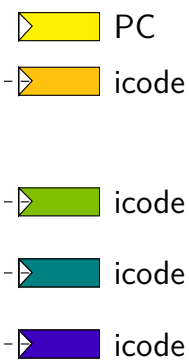
e.g. decode stage: send to execute (`d_icode`, ...)

two exceptions (share between instructions):

- what instruction to run next?
- data and control hazards

pushq pipeline registers

stage	pushq rA
fetch	icode : ifun $\leftarrow M_1$[PC] valP $\leftarrow$ PC $+ 2$
PC update	PC $\leftarrow$ valP
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[\%rsp]$
execute	valE $\leftarrow$ valB $- 8$
memory	$M[$ valE $] \leftarrow$ valA
write back	



pushq pipeline registers

stage	pushq rA	
fetch	icode : ifun $\leftarrow M_1$[PC] valP $\leftarrow$ PC $+ 2$	PC
PC update	PC $\leftarrow$ valP	icode, rA, rB
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[\%rsp]$	icode
execute	valE $\leftarrow$ valB $- 8$	icode
memory	$M[$ valE $] \leftarrow$ valA	icode
write back		

pushq pipeline registers

stage	pushq rA	
fetch	icode : ifun $\leftarrow M_1$[PC] valP $\leftarrow$ PC $+ 2$	PC
PC update	PC $\leftarrow$ valP	icode, rA, rB
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[\%rsp]$	icode, valA, valB
execute	valE $\leftarrow$ valB $- 8$	icode valA
memory	$M[$ valE $] \leftarrow$ valA	icode
write back		

pushq pipeline registers

stage	pushq rA	
fetch	icode : ifun $\leftarrow M_1$[PC] valP $\leftarrow$ PC $+ 2$	PC
PC update	PC $\leftarrow$ valP	icode, rA, rB
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[\%rsp]$	icode, valA, valB
execute	valE $\leftarrow$ valB $- 8$	icode, valA, valE
memory	$M[$ valE $] \leftarrow$ valA	icode
write back		

addq pipeline registers

stage	addq rA, rB	
fetch	icode : ifun $\leftarrow M_1$[PC] valP $\leftarrow$ PC $+ 2$	PC
PC update	PC $\leftarrow$ valP	icode
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$	icode
execute	valE $\leftarrow$ valB $+$ valB	icode
memory		icode
write back	$R[$ rB $] \leftarrow$ valE	

addq pipeline registers

stage	`addq rA, rB`	
fetch	icode : ifun $\leftarrow M_1$[PC] valP $\leftarrow$ PC $+ 2$	PC
PC update	PC $\leftarrow$ valP	icode, rA, rB
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$	
		icode, rB
execute	valE $\leftarrow$ valB $+$ valB	
		icode, rB
memory		icode, rB
write back	$R[$ rB $] \leftarrow$ valE	

addq pipeline registers

stage	`addq rA, rB`	
fetch	icode : ifun $\leftarrow M_1$[PC] valP $\leftarrow$ PC $+ 2$	PC
PC update	PC $\leftarrow$ valP	icode, rA, rB
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$	
		icode, rB, valA, valB
execute	valE $\leftarrow$ valB $+$ valB	
		icode, rB
memory		icode, rB
write back	$R[$ rB $] \leftarrow$ valE	

addq pipeline registers

stage	`addq rA, rB`	
fetch	icode : ifun $\leftarrow M_1$[PC] valP $\leftarrow$ PC $+ 2$	PC
PC update	PC $\leftarrow$ valP	icode, rA, rB
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$	
		icode, rB, valA, valB
execute	valE $\leftarrow$ valB $+$ valB	
		icode, rB, valE
memory		icode, rB, valE
write back	$R[$ rB $] \leftarrow$ valE	

addq pipeline registers

stage	`addq rA, rB`	
fetch	icode : ifun $\leftarrow M_1$[PC] valP $\leftarrow$ PC $+ 2$	PC
PC update	PC $\leftarrow$ valP	icode, rA, rB
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$ dstE $\leftarrow$ rB	
		icode, dstE, valA, valB
execute	valE $\leftarrow$ valB $+$ valB	
		icode, dstE, valE
memory		icode, dstE, valE
write back	$R[$ dstE $] \leftarrow$ valE	

addq pipeline registers

stage	addq rA, rB
fetch	icode : ifun $\leftarrow M_1$[PC] valP $\leftarrow$ PC $+ 2$
PC update	PC $\leftarrow$ valP
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$ dstE $\leftarrow$ rB
execute	valE $\leftarrow$ valB $+$ valB
write back	$R[$ dstE $] \leftarrow$ valE

PC

icode, rA, rB

icode, dstE, valA, valB

, dstE, valE

, dstE, valE

redundant with rB + icode
but will make handling data hazards easier

computing the PC

conditional jmp — instruction and condition codes

ret — from memory

otherwise — instruction only

computing the PC

conditional jmp — instruction and condition codes

ret — from memory

otherwise — instruction only

can be done in fetch stage entirely

computing the PC

conditional jmp — instruction and condition codes

ret — from memory

otherwise — instruction only

must wait till memory stage
worst case: ret immediately follows memory write

computing the PC

conditional jmp — instruction and condition codes

ret — from memory

otherwise — in

must wait till execute stage
worst case: previous instruction sets CCs

conditional jmp (w/ stalling)

```
        subq %r8, %r8
        je label

label:  irmovq ...
```

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

conditional jmp (w/ stalling)

```
        subq %r8, %r8
        je label

label:  irmovq ...
```

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

conditional jmp (w/ stalling)

```
        subq %r8, %r8
        je label

label:  irmovq ...
```

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

ZF sent via register

conditional jmp (w/ stalling)

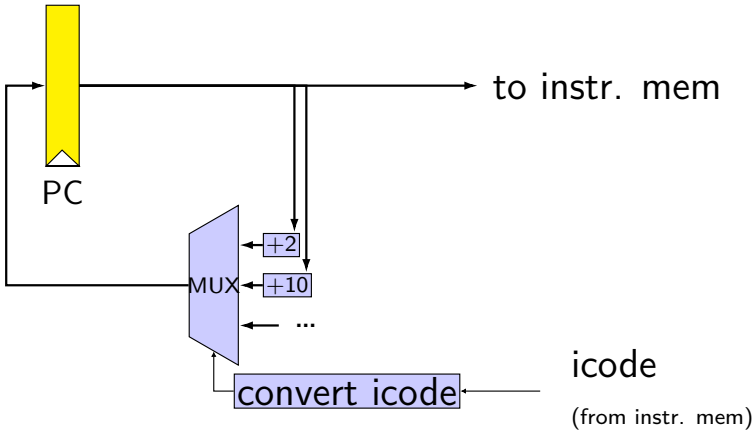
```
        subq %r8, %r8
        je label

label:  irmovq ...
```

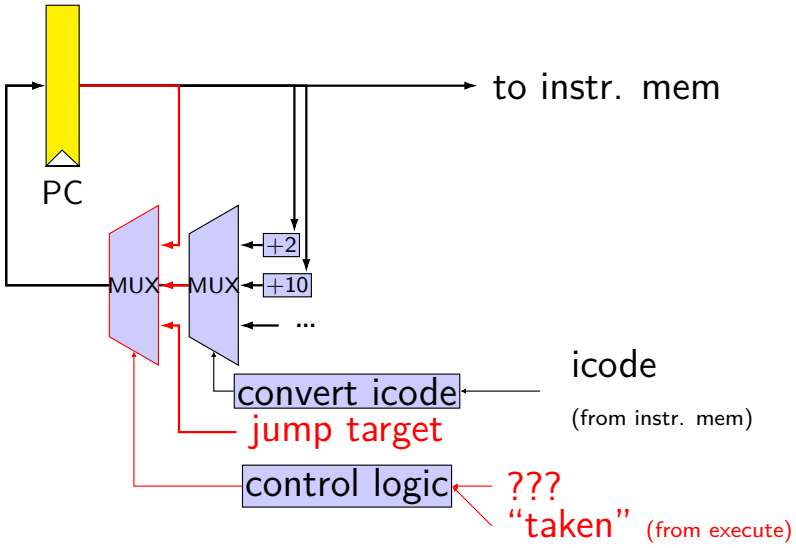
time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

"taken" sent from execute to fetch

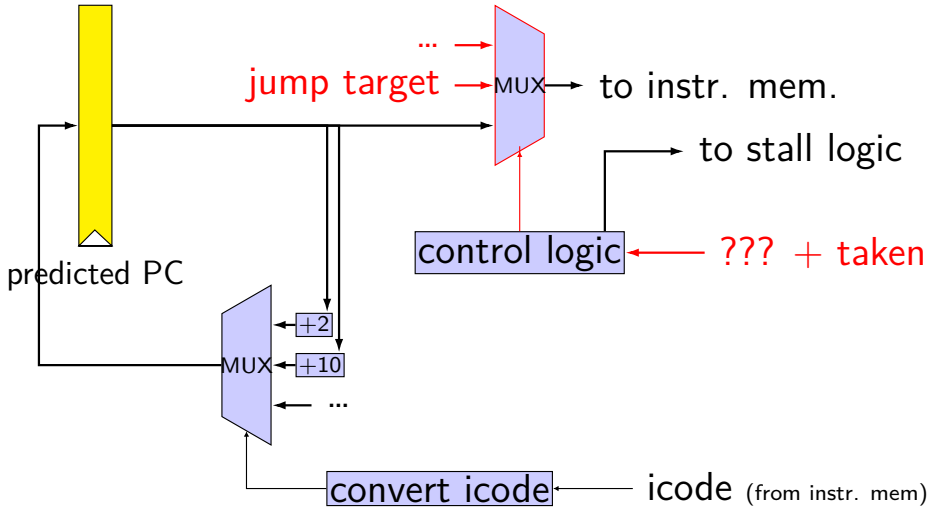
PC update (revised)



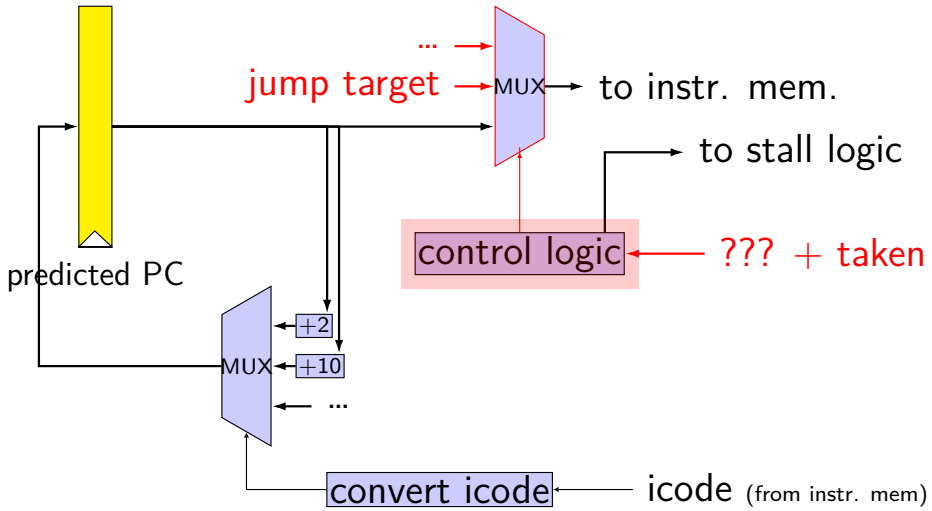
PC update (revised)



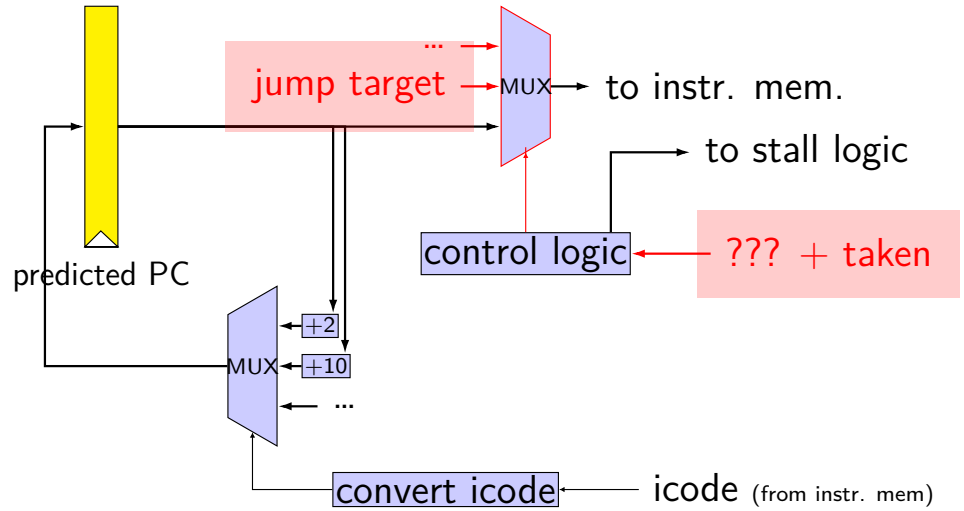
PC update (rearranged)



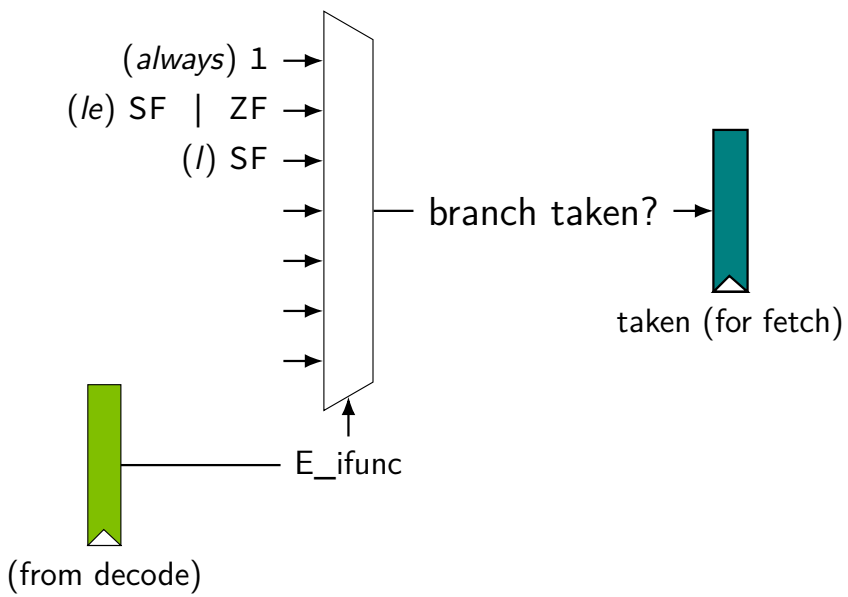
PC update (rearranged)



PC update (rearranged)



"taken" signal (in execute stage)



jCC state tracking

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

jCC state tracking

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

f_icode = JXX

jCC state tracking

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

D_icode = JXX

jCC state tracking

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)		
5	irmovq	nothing	nothing	jCC (done)	OPq

E_icode = JXX

jCC state tracking

time	fetch	decode	execute	memory	writeback
1	OPq				
2	jCC	OPq			
3	wait for jCC	jCC	OPq (set ZF)		
4	wait for jCC	nothing	jCC (use ZF)	OPq	
5	irmovq	nothing	nothing	jCC (done)	OPq

M_icode = JXX

PC update

```
pc_for_imem = [
    M_icode == JXX && M_branchTaken:
        jumpTarget;
    M_icode == JXX && !M_branchTaken:
        predictedPC;
    ...;
]
```


ret

```
        call empty
        addq %r8, %r9

empty:  ret
```

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

ret

```
        call empty
        addq %r8, %r9

empty:  ret
```

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

return address stored here

ret

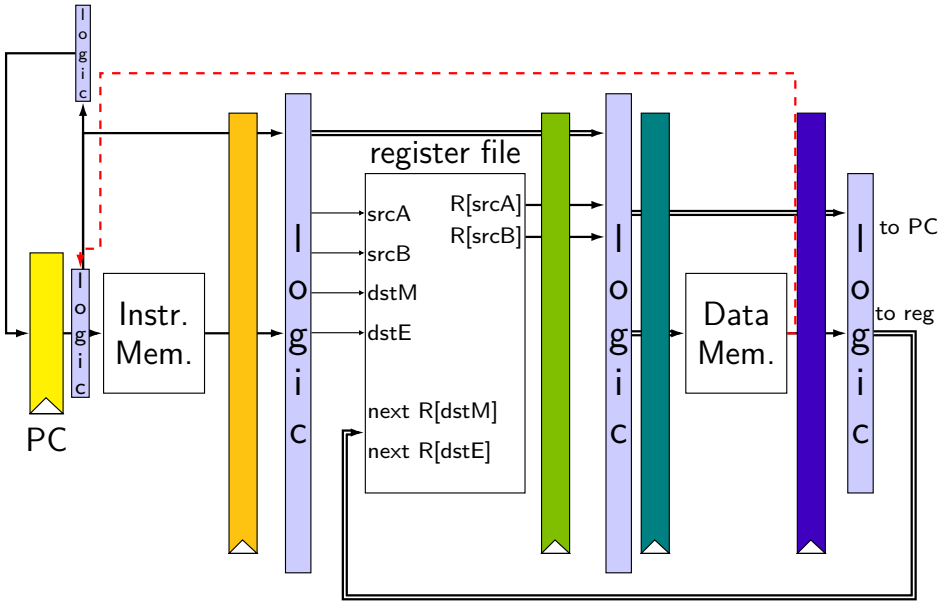
```
        call empty
        addq %r8, %r9

empty:  ret
```

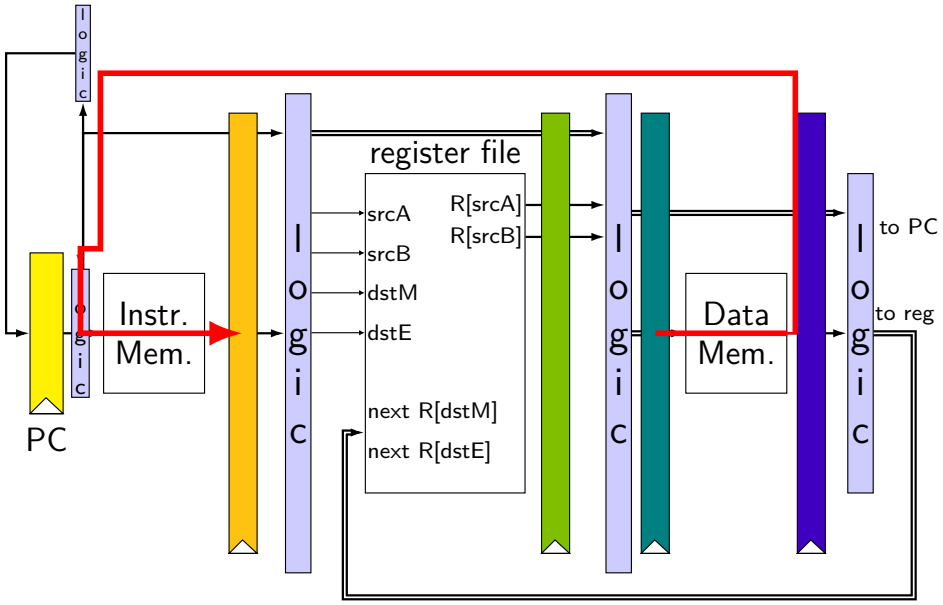
time	fetch	decode	execute	memory	writeback
1	call				
2	ret	call			
3	wait for ret	ret	call		
4	wait for ret	nothing	ret	call (store)	
5	wait for ret	nothing	nothing	ret (load)	call
6	addq			nothing	ret

why not start addq here?

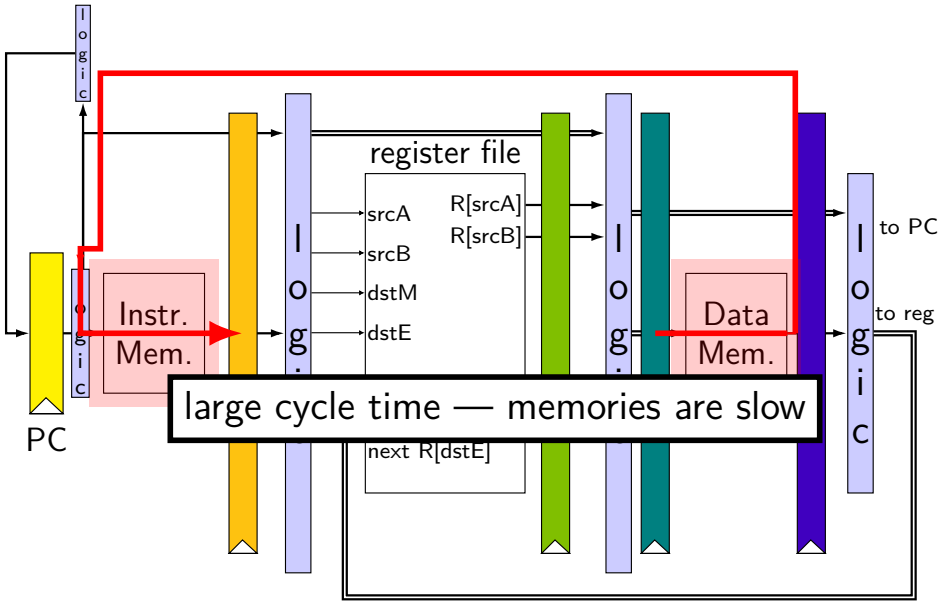
why not memory to PC



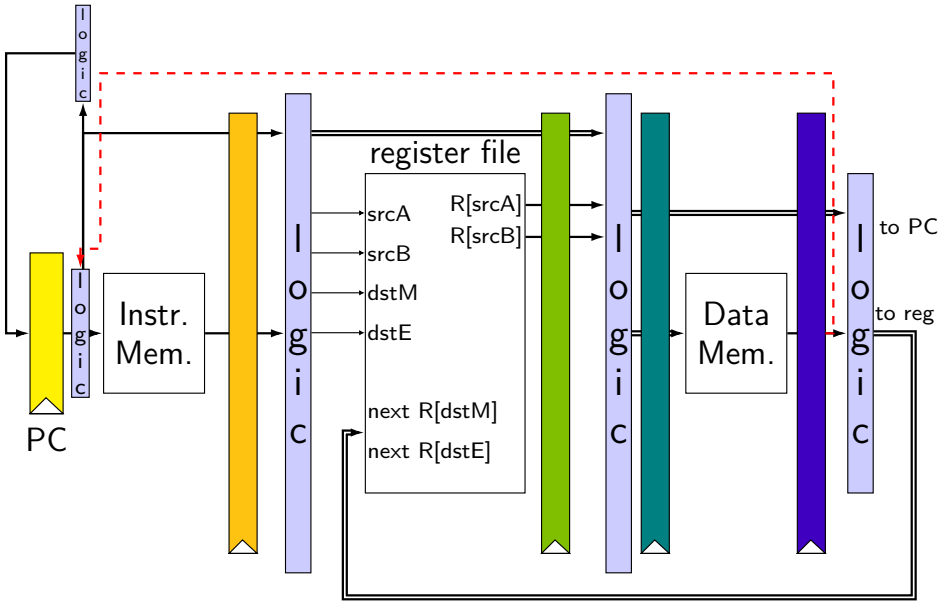
why not memory to PC



why not memory to PC



ret wiring



when do instructions change things?

... other than pipeline registers/PC:

stage	changes
fetch	(none)
decode	(none)
execute	condition codes
memory	memory writes
writeback	register writes/stat changes

when do instructions change things?

... other than pipeline registers/PC:

stage	changes
fetch	(none)
decode	(none)
execute	condition codes
memory	memory writes
writeback	register writes/stat changes

to "undo" instruction during fetch/decode/execute:

- suppress condition code update (if any)
- forget everything in pipeline registers

making guesses

```
        subq    %rcx, %rax
        jne     LABEL
        xorq    %r10, %r11
        xorq    %r12, %r13

LABEL:  addq    %r8, %r9
        rmmovq  %r10, 0(%r11)
```

speculate: jne will goto LABEL

right: 2 cycles faster!

wrong: forget before execute finishes

jXX: speculating right

```
        subq %r8, %r8
        jne LABEL
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
        irmovq $1, %r11
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	irmovq	rmmovq	addq	jne (done)	OPq

jXX: speculating right

```
        subq %r8, %r8
        jne LABEL
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
        irmovq $1, %r11
```

time	fetch	decode	execute	memory	writeback
1	subq				
2	jne	subq			
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	j		
5	irmovq	rmmovq	addq	jne (done)	OPq

were waiting/nothing

jXX: speculating wrong

```
        subq %r8, %r8
        jne LABEL
        xorq %r10, %r11
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
```

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

jXX: speculating wrong

```
        subq %r8, %r8
        jne LABEL
        xorq %r10, %r11
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
```

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

"squash" wrong guesses

jXX: speculating wrong

```
        subq %r8, %r8
        jne LABEL
        xorq %r10, %r11
        ...

LABEL:  addq %r8, %r9
        rmmovq %r10, 0(%r11)
```

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

fetch correct next instruction

jXX control logic

branch prediction simplifies — no stalling logic

... but extra logic to "squash" mispredicted instructions

jCC: assume taken?

book's choice: guess all jXXs are taken

empirical observation: most are

intuition:

```
        irmovq $100, %rax
        irmovq $1, %rbx
LOOP:   ...
        subq %rbx, %rax
        je LOOP // taken 99% of the time
```


Performance

hypothetical instruction mix

kind	portion	cycles (predict)	cycles (stall)
not-taken jXX	3%	3	3
taken jXX	5%	1	3
ret	1%	4	4
others	91%	1*	1*

predict: $3 \times .03 + 1 \times .05 + 4 \times .01 + 1 \times .91 =$ 1.09 cycles/instr.

stall: $3 \times .03 + 3 \times .05 + 4 \times .01 + 1 \times .91 =$ 1.19 cylces/instr.

* — ignoring data hazards

stalling/misprediction and latency

case 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

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

better branch prediction

forward (target > PC) not taken, backward (target < PC) taken

intuition: loops:

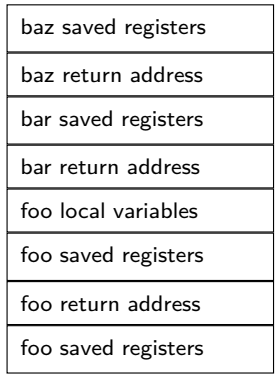
```
LOOP: ...
      ...
      je LOOP

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

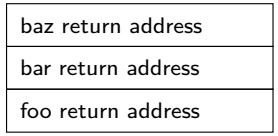

predicting ret: extra copy of stack

predicting ret — stack in processor registers

different than real stack/out of room? just slower



stack in memory



(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

summary

fetch/decode/execute/memory/writeback

add pipeline registers

normal next PC logic in fetch

branch prediction for jXX

- assume taken; verify before following 'execute' finishes

stalling for ret

slower due to branch misprediction and stalling

next two classes

handling data hazards — mostly without stalling

details of stall/cancel logic

pipelined addq