Changelog

Changes made in this version not seen in first lecture:

- 12 October 2017: slide 10: “extra 4” $\rightarrow$ “extra 3” (ret takes 4 cycles, but stalls for 3)
- 12 October 2017: slide 11: simplify by only considering undoing instruction in fetch/decode, not fetch/decode/execute
- 12 October 2017: slide 15: write ‘subq’ instead of ‘OPq’
- 12 October 2017: slide 23: lines to registers should go to same sides of register
- 12 October 2017: slide 25: correct highlighting of %r9
- 12 October 2017: slide 31: add missing newline in “common for processors to do this” box

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		

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 */ );
```

stalling costs

with only stalling:

extra 3 cycles (total 4) for every `ret`

extra 2 cycles (total 3) for conditional jmp

up to 3 extra cycles for data dependencies

stalling costs

with only stalling:

extra 3 cycles (total 4) for every `ret`

extra 2 cycles (total 3) for conditional jmp

up to 3 extra cycles for data dependencies

can we do better?

stalling costs

with only stalling:

extra 3 cycles (total 4) for every `ret`

can't easily read memory early
might be written in previous in

extra 2 cycles (total 3) for conditional jmp

up to 3 extra cycles for data dependencies

can we do better?

stalling costs

with only stalling:

extra 3 cycles (total 4) for every `ret`

extra 2 cycles (total 3) for conditional jmp

trick: guess and check

up to 3 extra cycles for data dependencies

can we do better?

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:

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				
3	addq [?]	jne	subq (set ZF)		
4	rmmovq [?]	addq [?]	jne (use ZF)	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 [?]	j			
4	rmmovq [?]	addq [?]	jne (use ZF)	OPq	
5	xorq	nothing	nothing	jne (done)	OPq

fetch correct next instruction

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*

* — ignoring data hazards

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

squashing with stall/bubble

time	fetch	decode	execute	memory	writeback
1	subq				
	N	N			
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);

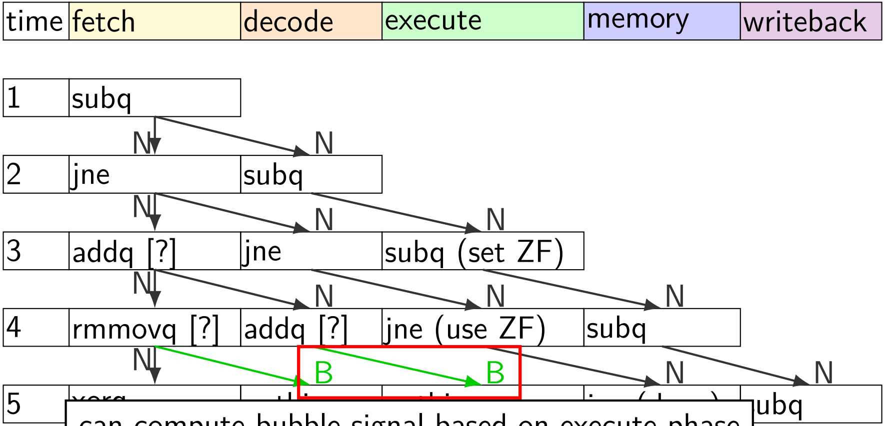
squashing with stall/bubble

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



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 || ...;
```

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);

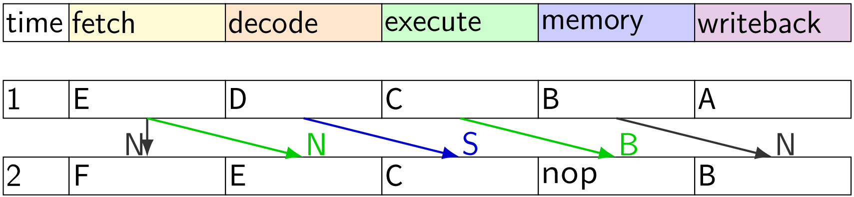
exercise: squash + stall (2)

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)



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

stalling costs

with only stalling:

extra 3 cycles (total 4) for every `ret`

extra 2 cycles (total 3) for conditional jmp

up to 3 extra cycles for data dependencies

trick: use values waiting to get to register file

can we do better?

revisiting data hazards

stalling worked

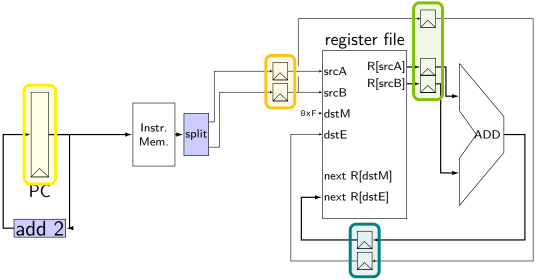
but very unsatisfying — wait 2 extra cycles to use anything?!

observation: value ready before it would be needed

(just not stored in a way that let's us get it)

motivation

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

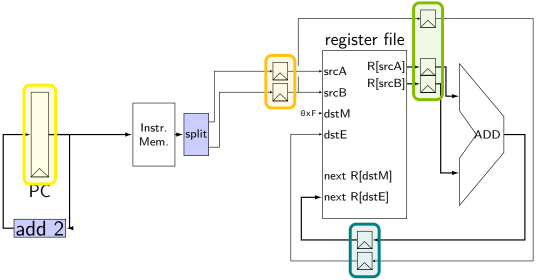


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2		9	8	800	900	9		
3				900	800	8	1700	9
4							1700	8

should be 1700

motivation

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

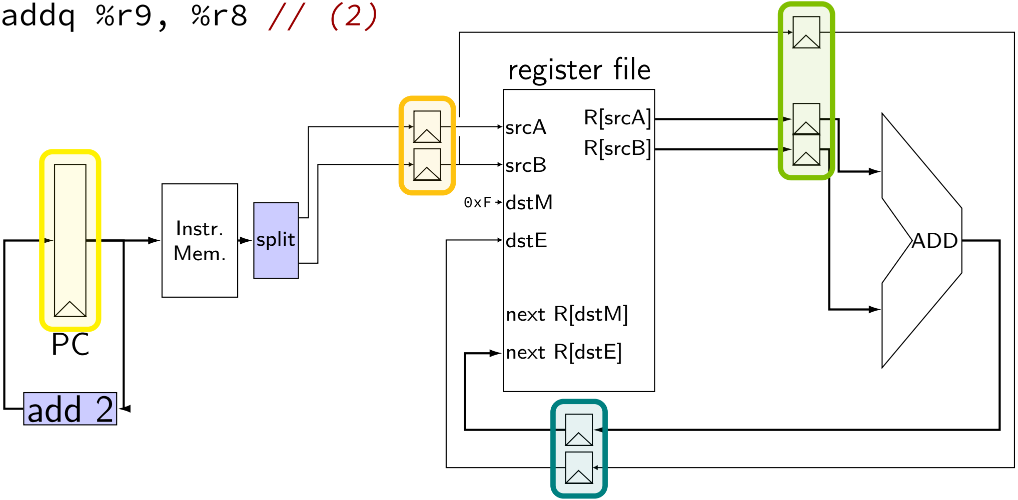


	fetch	fetch/decode		decode/execute			execute/writeback	
cycle	PC	rA	rB	R[srcA]	R[srcB]	dstE	next R[dstE]	dstE
0	0x0							
1	0x2	8	9					
2		9	8	800	900	9		
3				900	800	8	1700	9
4							1700	8

should be 1700

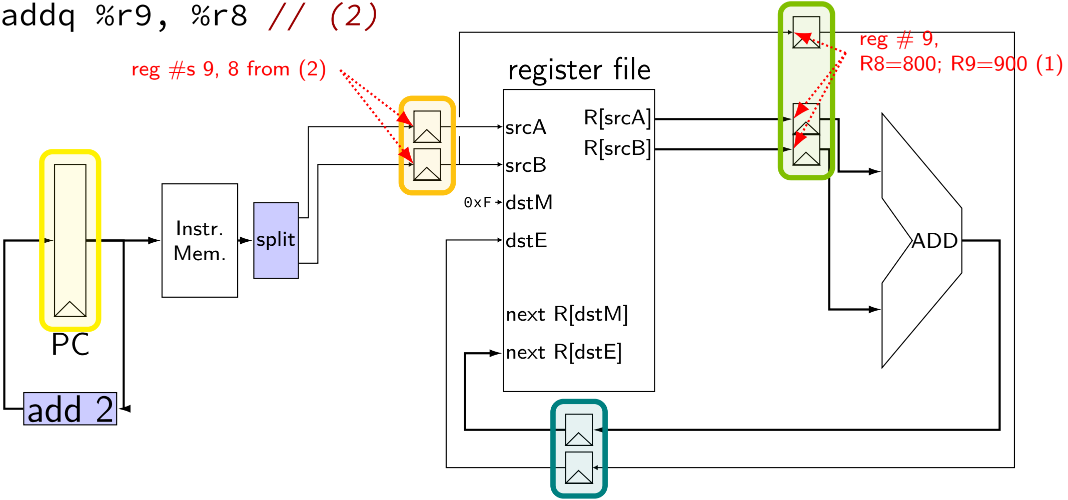
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



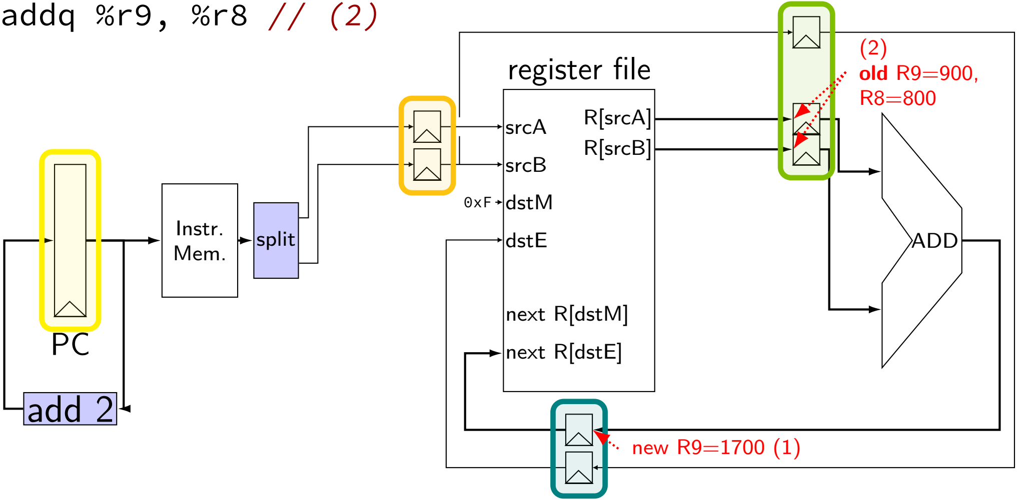
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



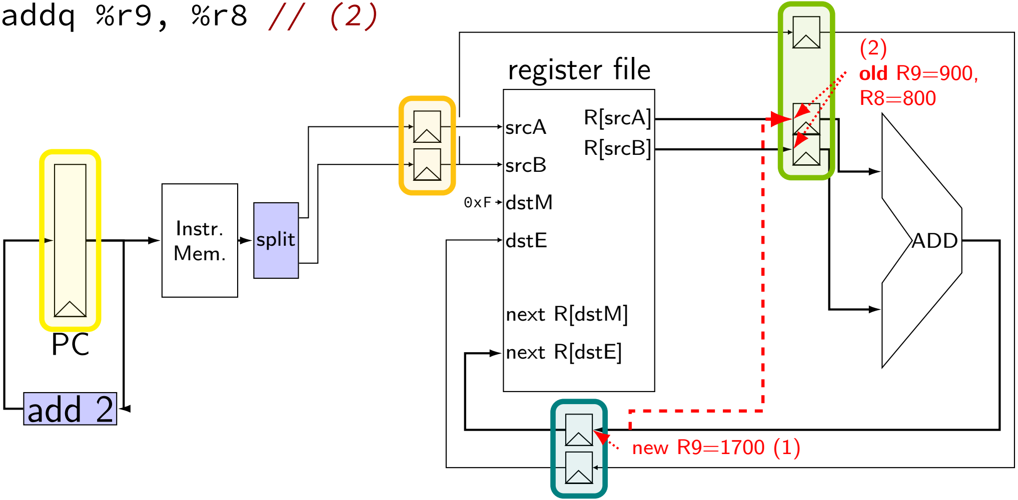
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



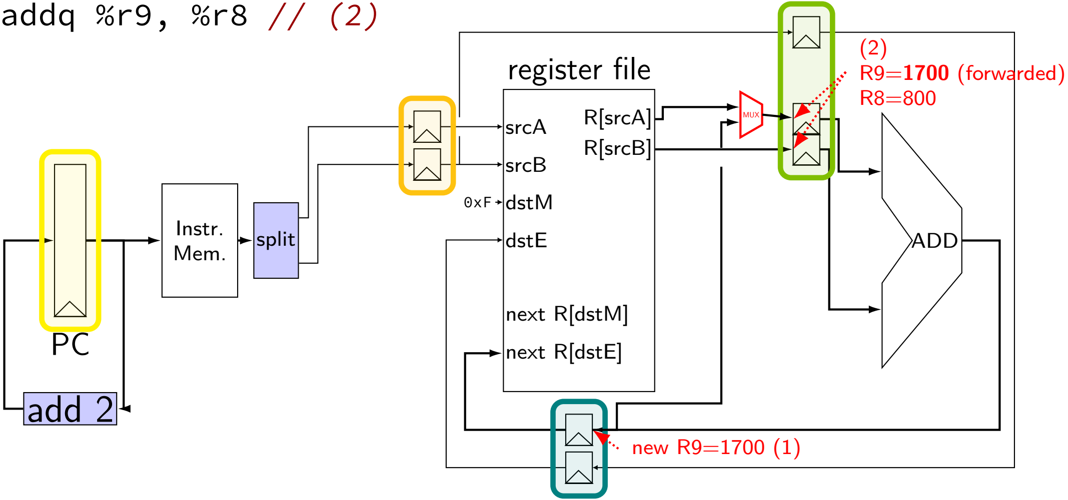
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



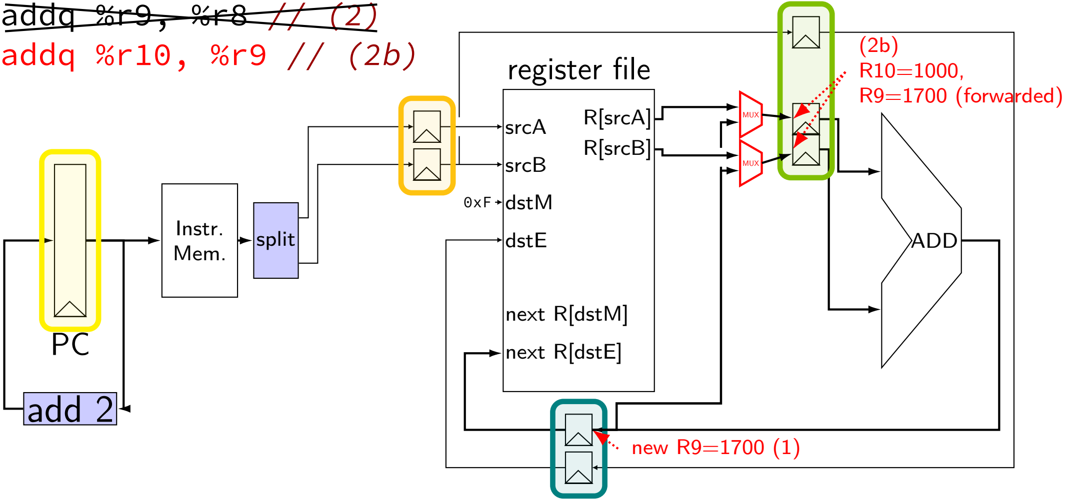
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



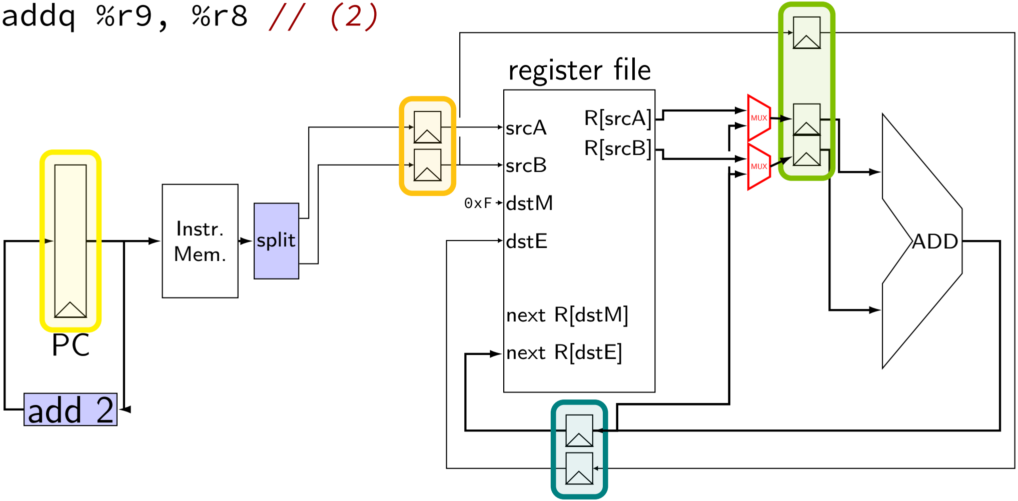
forwarding

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
addq %r10, %r9 // (2b)
```



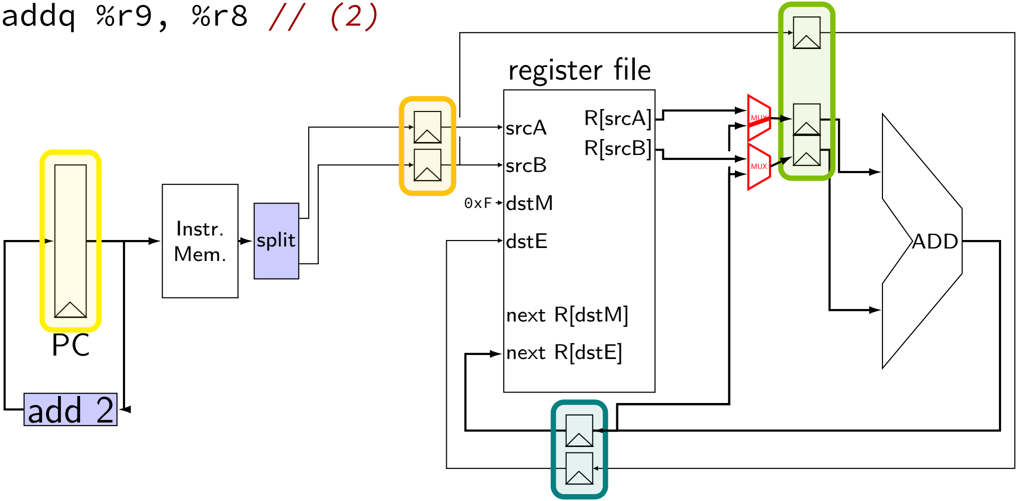
forwarding: MUX conditions

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



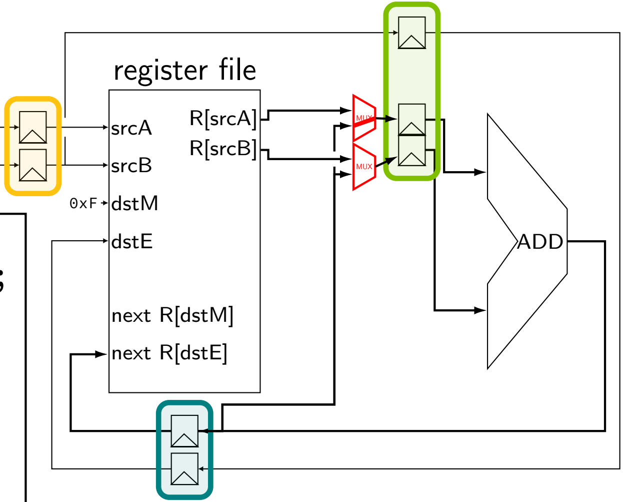
forwarding: MUX conditions

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



forwarding: MUX conditions

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



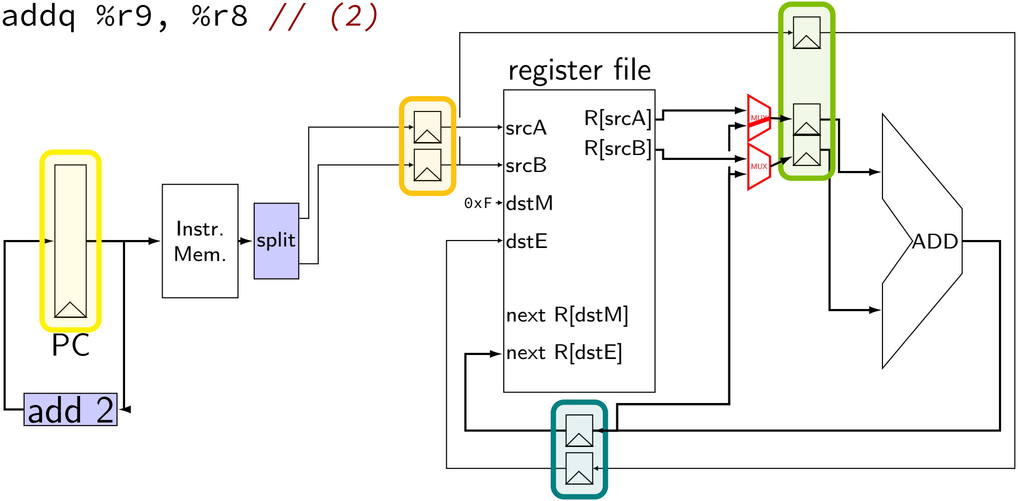

```
d_valA= [
   condition : e_valE;
   1 : reg_outputA;
];
```

What could condition be?

a. `W_rA == reg_srcA`
b. `W_dstE == reg_srcA`
c. `e_dstE == reg_srcA`
d. `d_rB == reg_srcA`
e. something else

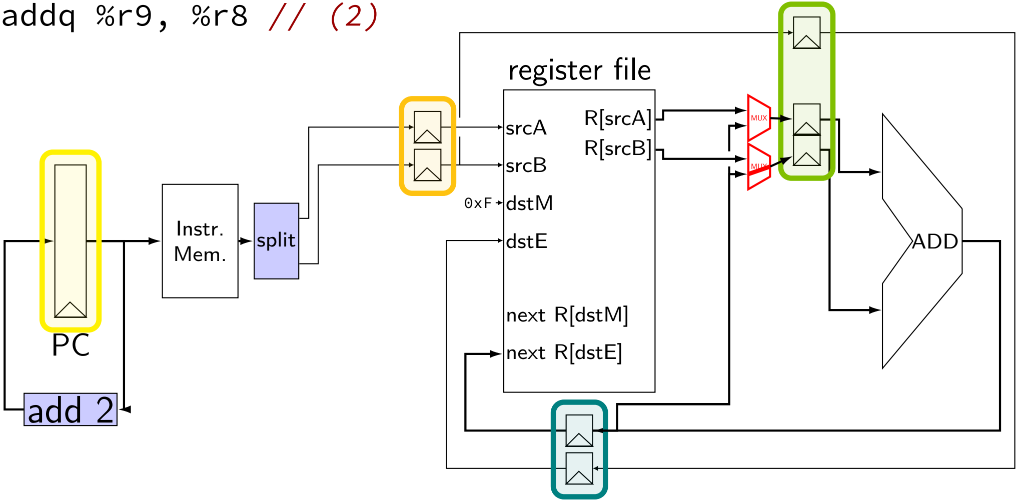
forwarding: MUX conditions

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



forwarding: MUX conditions

```
addq %r8, %r9 // (1)
addq %r9, %r8 // (2)
```



some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`mrmovq 4(%r11), %r10`			F	D	E	M	W		
`rmmovq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`mrmovq 4(%r11), %r10`			F	D	E	M	W		
`rmmovq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`mrmovq 4(%r11), %r10`			F	D	E	M	W		
`rmmovq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`mrmovq 4(%r11), %r10`			F	D	E	M	W		
`rmmovq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`mrmovq 4(%r11), %r10`			F	D	E	M	W		
`rmmovq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`mrmovq 4(%r11), %r10`			F	D	E	M	W		
`rmmovq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

	cycle # 0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`mrmovq 4(%r11), %r10`			F	D	E	M	W		
`rmmovq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`mrmovq 4(%r11), %r10`			F	D	E	M	W		
`rmmovq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

some forwarding paths

cycle #	0	1	2	3	4	5	6	7	8
`addq %r8, %r9`	F	D	E	M	W				
`subq %r9, %r11`		F	D	E	M	W			
`mrmovq 4(%r11), %r10`			F	D	E	M	W		
`rmmovq %r9, 8(%r11)`				F	D	E	M	W	
`xorq %r10, %r9`					F	D	E	M	W

multiple forwarding paths (1)

	cycle #	0	1	2	3	4	5	6	7	8
addq `%r10, %r8`		F	D	E	M	W				
addq `%r11, %r8`			F	D	E	M	W			
addq `%r12, %r8`				F	D	E	M	W		

multiple forwarding paths (1)

	cycle #	0	1	2	3	4	5	6	7	8
`addq %r10, %r8`		F	D	E	M	W				
`addq %r11, %r8`			F	D	E	M	W			
`addq %r12, %r8`				F	D	E	M	W		

multiple forwarding HCL (1)

```
/* decode output: valA */
d_valA = [
    ...
    reg_srcA == e_dstE : e_valE;
        /* forward from end of execute */

    reg_srcA == m_dstE : m_valE;
        /* forward from end of memory */


    ...
    1 : reg_outputA;
];
```

multiple forwarding paths (2)

cycle #	0	1	2	3	4	5	6	7	8
`addq %r10, %r8`	F	D	E	M	W				
`addq %r11, %r12`		F	D	E	M	W			
`addq %r12, %r8`			F	D	E	M	W		

multiple forwarding paths (2)

	cycle #	0	1	2	3	4	5	6	7	8
addq %r10, %r8		F	D	E	M	W				
addq %r11, %r12			F	D	E	M	W			
addq %r12, %r8				F	D	E	M	W		

multiple forwarding paths (2)

cycle #	0	1	2	3	4	5	6	7	8
`addq %r10, %r8`	F	D	E	M	W				
`addq %r11, %r12`		F	D	E	M	W			
`addq %r12, %r8`			F	D	E	M	W		

multiple forwarding HCL (2)

```
d_valA = [
    ...
    reg_srcA == e_dstE : e_valE;
    ...
    1 : reg_outputA;
];
...
d_valB = [
    ...
    reg_srcB == m_dstE : m_valE;
    ...
    1 : reg_outputA;
];
```

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

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)

hazards versus dependencies

dependency — X needs result of instruction Y?

hazard — will it not work in some pipeline?

- before extra work is done to "resolve" hazards
- like forwarding or stalling or branch prediction

ex.: dependencies and hazards (1)

```
addq    %rax, %rbx
subq    %rax, %rcx
irmovq  $100, %rcx
addq    %rcx, %r10
addq    %rbx, %r10
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (1)

```
addq    %rax,  %rbx
subq    %rax,  %rcx
irmovq  $100,  %rcx
addq    %rcx,  %r10
addq    %rbx,  %r10
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (1)

```
addq    %rax,  %rbx
subq    %rax,  %rcx
irmovq  $100,  %rcx
addq    %rcx,  %r10
addq    %rbx,  %r10
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (1)

```
addq    %rax, %rbx
subq    %rax, %rcx
irmovq  $100, %rcx
addq    %rcx, %r10
addq    %rbx, %r10
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (2)

```
        mrmovq   0(%rax)  %rbx
        addq     %rbx     %rcx
        jne      foo
foo:    addq     %rcx     %rdx
        mrmovq   (%rdx)   %rcx
```

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                    // 4 stage   // 5 stage
addq %rax, %r8      //           // W
subq %rax, %r9      // W         // M
xorq %rax, %r10     // EM        // E
andq %r8,  %r11     // D         // D
```

pipeline with different hazards

example: 4-stage pipeline:
fetch/decode/execute+memory/writeback

```
                        // 4 stage    // 5 stage
addq %rax, %r8          //            // W
subq %rax, %r9          // W          // M
xorq %rax, %r10         // EM         // E
andq %r8,  %r11         // D          // D
```

addq/andq is hazard with 5-stage pipeline

addq/andq is **not** a hazard with 4-stage pipeline

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

result only available after second execute stage

where does forwarding, stalls occur?

cycle #	0	1	2	3	4	5	6	7	8
addq %rcx, %r9	F	D	E1	E2	M	W			
addq %r9, %rbx									
addq %rax, %r9									
rmmovq %r9, (%rbx)									

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8
addq %rcx, %r9	F	D	E1	E2	M	W			
addq %r9, %rbx									
addq %rax, %r9									
rmmovq %r9, (%rbx)									

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8
`addq %rcx, %r9`	F	D	E1	E2	M	W			
`addq %r9, %rbx`		F	D	E1	E2	M	W		
`addq %rax, %r9`			F	D	E1	E2	M	W	
`rmmovq %r9, (%rbx)`				F	D	E1	E2	M	W

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8	
`addq %rcx, %r9`	F	D	E1	E2	M	W				
`addq %r9, %rbx`		F	D	E1	E2	M	W			
`addq %r9, %rbx`		F	D	D	E1	E2	M	W		
`addq %rax, %r9`			F	D	E1	E2	M	W		
`addq %rax, %r9`			F	F	D	E1	E2	M	W	
`rmmovq %r9, (%rbx)`				F	D	E1	E2	M	W	
`rmmovq %r9, (%rbx)`					F	D	E1	E2	M	W

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8	
addq %rcx, %r9	F	D	E1	E2	M	W				
addq %r9, %rbx		F	D	E1	E2	M	W			
addq %r9, %rbx		F	D	D	E1	E2	M	W		
addq %rax, %r9			F	D	E1	E2	M	W		
addq %rax, %r9			F	F	D	E1	E2	M	W	
rmmovq %r9, (%rbx)				F	D	E1	E2	M	W	
rmmovq %r9, (%rbx)					F	D	E1	E2	M	W

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

cycle #	0	1	2	3	4	5	6	7	8	
addq %rcx, %r9	F	D	E1	E2	M	W				
addq %r9, %rbx		F	D	E1	E2	M	W			
addq %r9, %rbx		F	D	D	E1	E2	M	W		
addq %rax, %r9			F	D	E1	E2	M	W		
addq %rax, %r9			F	F	D	E1	E2	M	W	
rmmovq %r9, (%rbx)				F	D	E1	E2	M	W	
rmmovq %r9, (%rbx)					F	D	E1	E2	M	W

missing pieces

multi-cycle memories

beyond pipelining: out-of-order, multiple issue

missing pieces

multi-cycle memories

beyond pipelining: out-of-order, multiple issue

multi-cycle memories

ideal case for memories: single-cycle

achieved with caches (next topic)

- fast access to small number of things

typical performance:

- 90+% of the time: single-cycle

sometimes many cycles (3–400+)

variable speed memories

memory is fast: (cache "hit"; recently accessed?)

cycle #	0	1	2	3	4	5	6	7	8
`mrmovq 0(%rbx), %r8`	F	D	E	M	W				
`mrmovq 0(%rcx), %r9`		F	D	E	M	W			
`addq %r8, %r9`			F	D	D	E	M	W	

memory is slow: (cache "miss")

cycle #	0	1	2	3	4	5	6	7	8	9	10	11
`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:

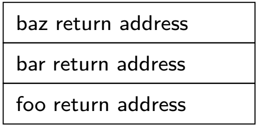
```
LOOP: ...
      ...
      je LOOP

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

stack in memory



(partial?) stack in CPU registers

prediction before fetch

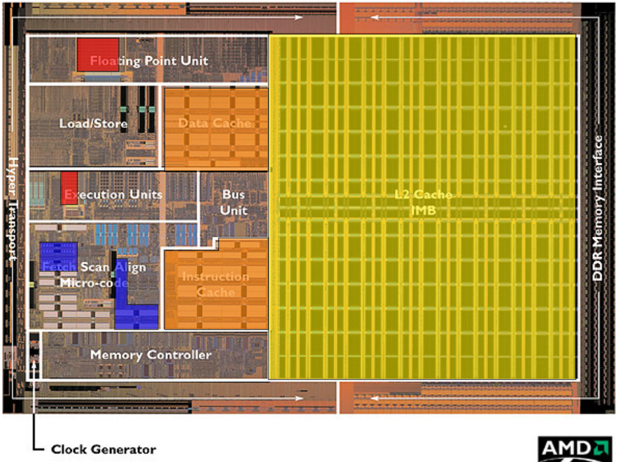
real processors can take multiple cycles to read instruction memory

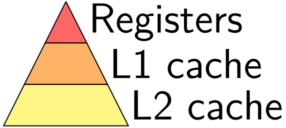
predict branches before reading their opcodes

how — more extra data structures

- tables of recent branches (often many kilobytes)

2004 CPU





Branch Prediction (approximate)

Image: approx 2004 AMD press image of Opteron die

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