forwarding idea

read wrong value (e.g. from register)

correct value is already computed
- elsewhere in pipeline
- maybe even after old value was read

substitute from wrong value
- using MUX

quiz question: forwarding in IRMOVQ

cycle #	0	1	2	3	4	5	6	7	8
irmovq `$50, %r8`	F	D	E	M	W				
addq `%r11, %r8`		F	D	E	M	W			

quiz question: forwarding in IRMOVQ

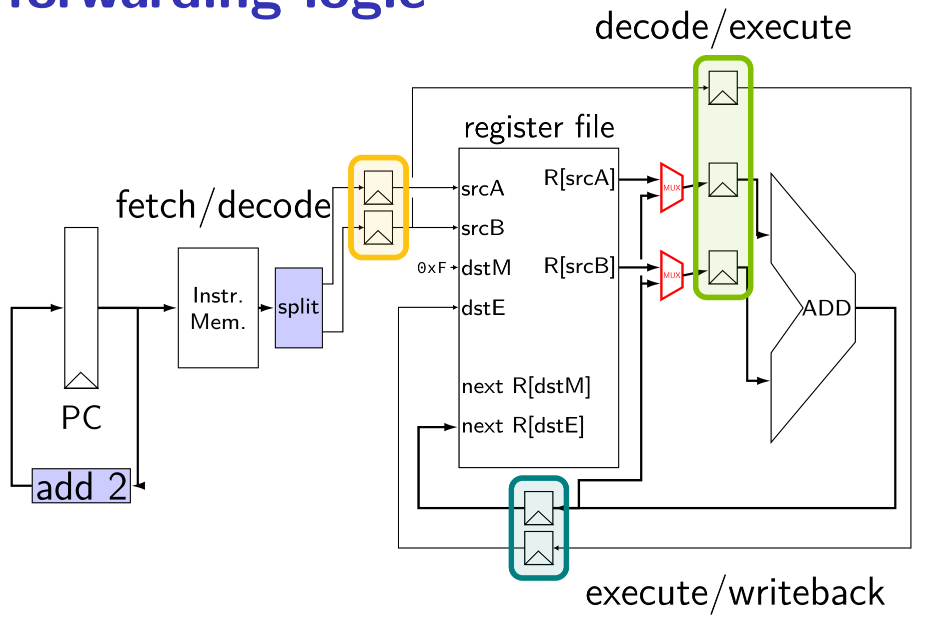
cycle #	0	1	2	3	4	5	6	7	8
irmovq `$50, %r8`	F	D	E	M	W				
addq `%r11, %r8`		F	D	E	M	W			

output of decode/execute regs (`irmovq`)
- (unchanged during execute stage)

input of execute/memory regs (`irmovq`)

input of decode/execute regs (`addq`)

forwarding logic



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

forwarding in HCL

```
register dE {
    valA : 64 = 0;
    dstE : 4 = 0;
};
...
/* was: d_valA = reg_outputA; */
d_valA = [
    reg_srcA == e_dstE : e_valE;
    ...
    1 : reg_outputA;
];
d_dstE = ...;
```

forwarding in HCL

```
register dE {
    valA : 64 = 0;
    dstE : 4 = 0;
};
...
/* was: d_valA = reg_outputA; */
d_valA = [
    reg_srcA == e_dstE : e_valE;
    ...
    1 : reg_outputA;
];
d_dstE = ...;
```

forwarding in HCL

```
register dE {
    valA : 64 = 0;
    dstE : 4 = 0;
};
...
/* was: d_valA = reg_outputA; */
d_valA = [
    reg_srcA == e_dstE : e_valE;
    ...
    1 : reg_outputA;
];
d_dstE = ...;
```

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

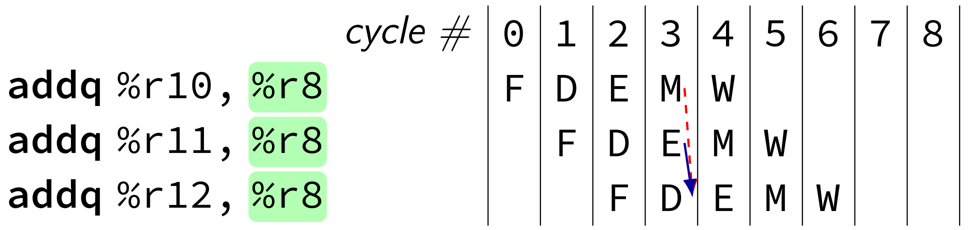
cycle #	0	1	2	3	4	5	6	7	8
mrmovq 0(%rax), %rbx	F	D	E	M	W				
subq %rbx, %rcx		F	D	E	M	W			
subq %rbx, %rcx		F	F	D	E	M	W		

stall

multiple forwarding paths

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

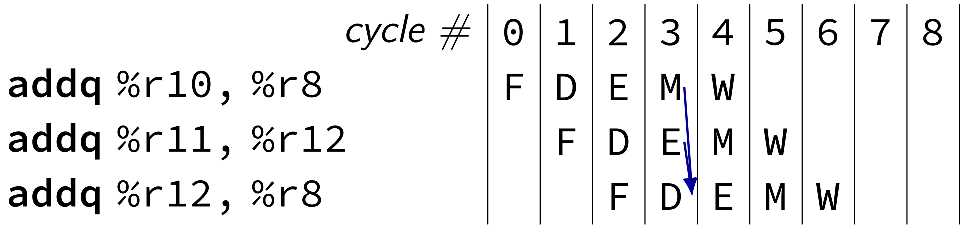


	cycle #	0	1	2	3	4	5	6	7	8
addq %r10, %r8		F	D	E	M	W				
addq %r11, %r8			F	D	E	M	W			
addq %r12, %r8				F	D	E	M	W		

multiple forwarding HCL

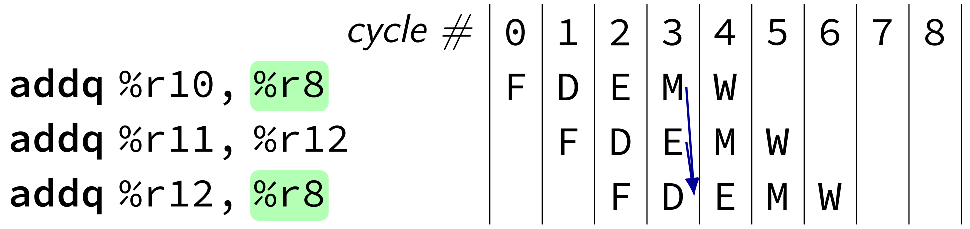
```
d_valA = [
    ...
    reg_srcA == e_dstE : e_valE;
    reg_srcA == m_dstE : m_valE;
    ...
    1 : reg_outputA;
];
```

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		

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

ret control hazard: 3 cycles of stalling

pipelined control costs

how much faster than single-cycle processor?

at most five times faster

depends on hardware details
- does added logic make clock cycle slower?

depends on what programs we run:
- how many mispredicted jumps?
- how many rets?
- how many load/use hazards?

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
```

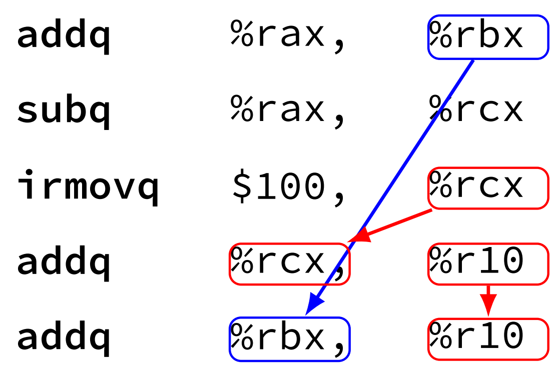
where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

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

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									

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	

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

exercise: different pipeline

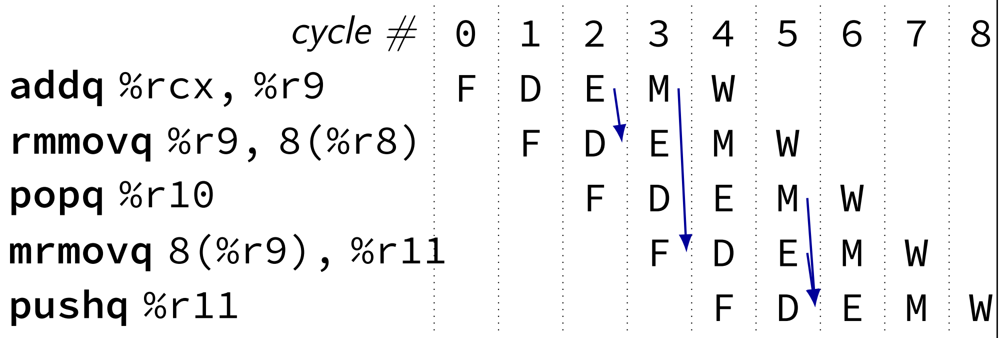
split execute into two stages: F/D/E1/E2/M/W

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

exercise: forwarding paths

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

exercise: forwarding paths



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

exercise: forwarding paths

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

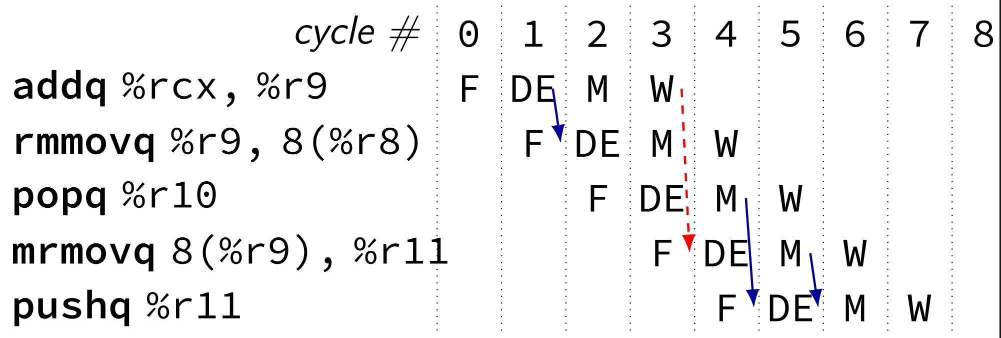
exercise: forwarding paths (alt pipe)

suppose four-stage pipeline:
fetch/decode+execute/memory/writeback

cycle #	0	1	2	3	4	5	6	7	8
addq %rcx, %r9	F	DE	M	W					
rmmovq %r9, 8(%r8)		F	DE	M	W				
popq %r10			F	DE	M	W			
mrmovq 8(%r9), %r11				F	DE	M	W		
pushq %r11					F	DE	M	W	

exercise: forwarding paths (alt pipe)

suppose four-stage pipeline:
fetch/decode+execute/memory/writeback



cycle #	0	1	2	3	4	5	6	7	8
addq %rcx, %r9	F	DE	M	W					
rmmovq %r9, 8(%r8)		F	DE	M	W				
popq %r10			F	DE	M	W			
mrmovq 8(%r9), %r11				F	DE	M	W		
pushq %r11					F	DE	M	W	

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

ret control hazard: 3 cycles of stalling

pipelined control costs

how much faster than single-cycle processor?

at most five times faster

depends on HW details:
- how expensive is forwarding logic? (new MUXes on critical path)
- how well balanced are the stages?

depends on what programs we run:
- how many mispredicted jumps?
- how many rets?
- how many load/use hazards?

HCL2D pipeline registers

```
register xF {
    pc : 64 = 0;
};
/* Fetch+PC Update*/
register fD {
    rA : 4 = REG_NONE; rB : 4 = REG_NONE;
};
/* Decode */
register dE {
    valA : 64 = 0; valB : 64 = E; dstE : 4 = REG_NONE;
}
/* Execute */
register eW {
    valE : 64 = 0; dstE : 4 = REG_NONE;
}
/* Writeback */
```

HCL2D: Fetch/Decode

unpipelined

```
/* Fetch+PC Update*/
pc = F_pc;
x_pc = pc + 2;
rA = i10bytes[12..16];
rB = i10bytes[8..12];
/* Decode */
reg_srcA = rA;
reg_srcB = rB;
dstE = rB;
valA = reg_outputA;
valB = reg_outputB;
```

pipelined

```
/* Fetch+PC Update*/
pc = F_pc;
x_pc = pc + 2;
f_rA = i10bytes[12..16];
f_rB = i10bytes[8..12];
/* Decode */
reg_srcA = D_rA;
reg_srcB = D_rB;
dstE = D_rB;
d_valA = reg_outputA;
d_valB = reg_outputB;
```

HCL2D pipelining debugging: intro

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

- notably: big drain on TA time

HCL2D pipeline debugging (1)

draw a picture of the state of the instructions

get −d output

- redirect to a file
- `cpu.exe -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

- remove anything not involved in the error
- find a **minimal** test case
- don't spend time looking at irrelevant instructions

draw the pipeline stages

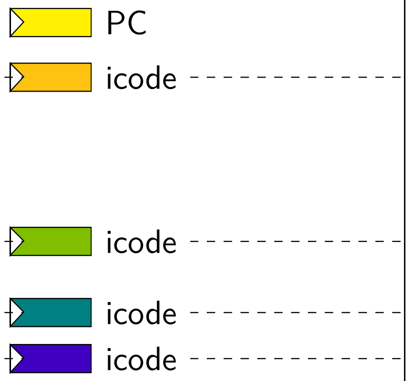
- what instructions are in fetch/decode/etc. when

HCL2D addq unpipelined

```
wire rA : 4, rB : 4, dstE : 4;
wire valA : 64, valB : 64, valE : 64;
register xF {
    pc : 64 = 0;
};
/* Fetch+PC Update*/
pc = F_pc; x_pc = pc + 2;
rA = i10bytes[12..16]; rB = i10bytes[8..12];
/* Decode */
reg_srcA = rA; reg_srcB = rB; dstE = rB;
valA = reg_outputA; valB = reg_outputB;
/* Execute */
valE = valA + valB;
/* Writeback */
reg_dstE = dstE; reg_inputE = valE;
```

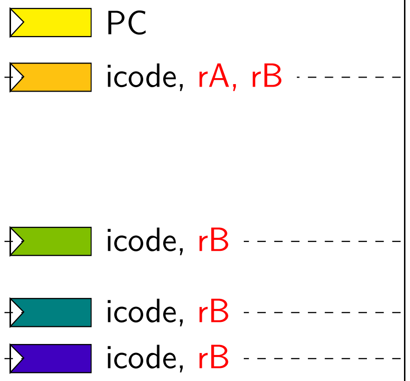
addq pipeline registers

stage	addq rA, rB
fetch	icode : ifun $\leftarrow M_1[\text{PC}]$ rA : rB $\leftarrow M_1[\text{PC}+1]$ valP $\leftarrow$ PC $+ 2$
PC update	PC $\leftarrow$ valP
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$
execute	valE $\leftarrow$ valB $+$ valB
memory	
write back	$R[$ rB $] \leftarrow$ valE



addq pipeline registers

stage	addq rA, rB
fetch	icode : ifun $\leftarrow M_1[\text{PC}]$ rA : rB $\leftarrow M_1[\text{PC}+1]$ valP $\leftarrow$ PC $+ 2$
PC update	PC $\leftarrow$ valP
decode	valA $\leftarrow R[$ rA $]$ valB $\leftarrow R[$ rB $]$
execute	valE $\leftarrow$ valB $+$ valB
memory	
write back	$R[$ rB $] \leftarrow$ valE



addq pipeline registers

stage	`addq rA, rB`
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC+1] valP $\leftarrow$ PC + 2
PC update	PC $\leftarrow$ valP
decode	valA $\leftarrow R$[rA] valB $\leftarrow R$[rB]
execute	valE $\leftarrow$ valB + valB
memory	
write back	R[rB] $\leftarrow$ valE

- 🟨 PC
- 🟧 icode, rA, rB
- 🟩 icode, rB, valA, valB
- 🟦 icode, rB
- 🟪 icode, rB

addq pipeline registers

stage	`addq rA, rB`
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC+1] valP $\leftarrow$ PC + 2
PC update	PC $\leftarrow$ valP
decode	valA $\leftarrow R$[rA] valB $\leftarrow R$[rB]
execute	valE $\leftarrow$ valB + valB
memory	
write back	R[rB] $\leftarrow$ valE

- 🟨 PC
- 🟧 icode, rA, rB
- 🟩 icode, rB, valA, valB
- 🟦 icode, rB, valE
- 🟪 icode, rB, valE

addq pipeline registers

stage	`addq rA, rB`
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC+1] 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
memory	
write back	R[dstE] $\leftarrow$ valE

- 🟨 PC
- 🟧 icode, rA, rB
- 🟩 icode, dstE, valA, valB
- 🟦 icode, dstE, valE
- 🟪 icode, dstE, valE

addq pipeline registers

stage	`addq rA, rB`
fetch	icode : ifun $\leftarrow M_1$[PC] rA : rB $\leftarrow M_1$[PC+1] 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
memory	
write back	R[dstE] $\leftarrow$ valE

- 🟨 PC
- 🟧 icode, rA, rB
- 🟩 icode, dstE, valA, valB
- 🟦 icode, dstE, valE
- 🟪 icode, dstE, valE

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