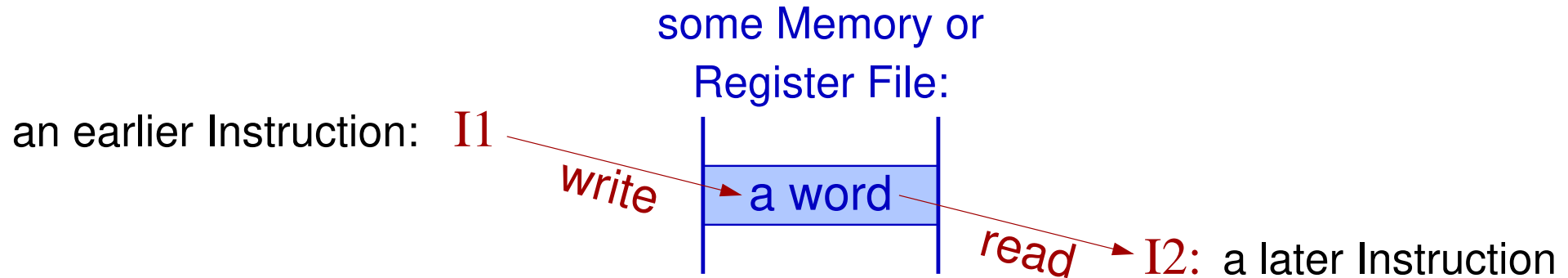
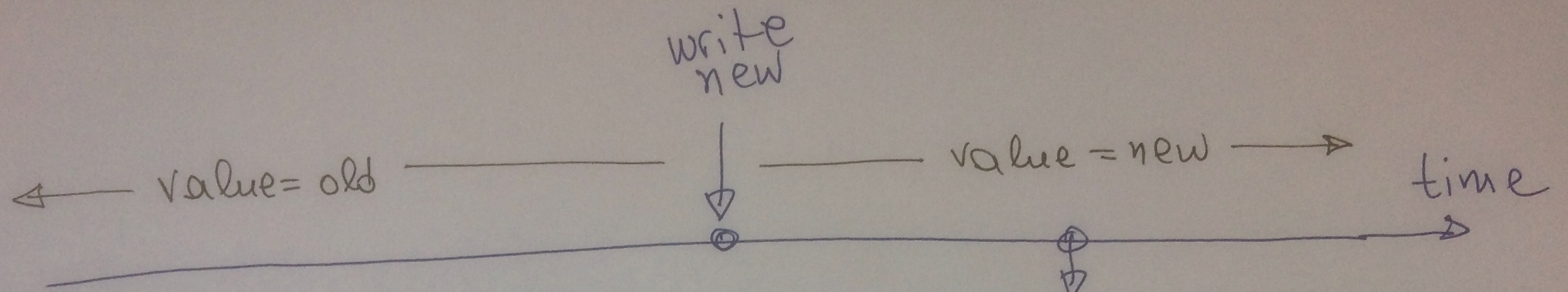


Data Dependences (Hazards) in Pipelines

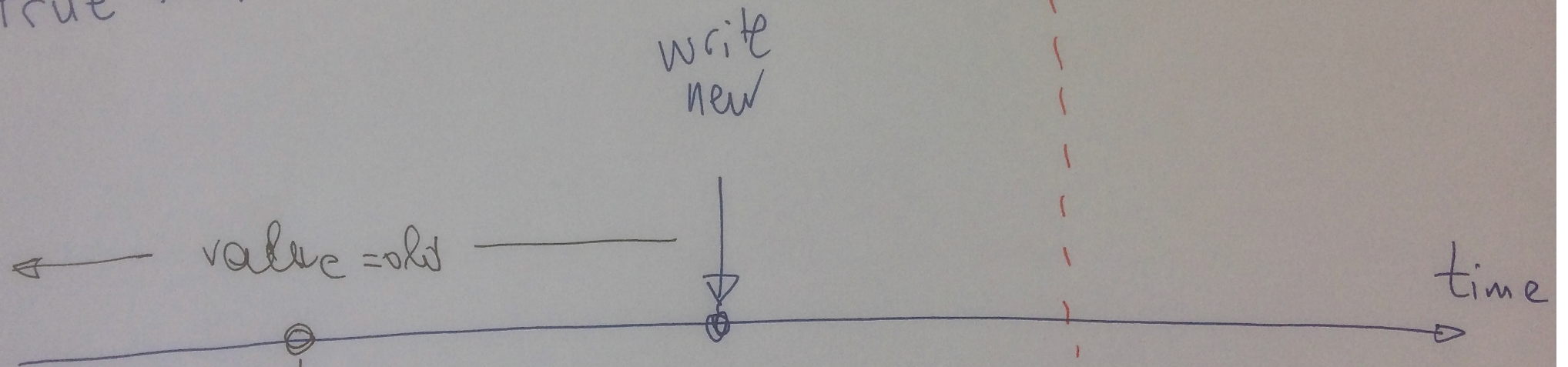


*I2 needs the new data written by I1,
hence must wait for I1 to write –or at least to generate– the new data*

- RAW (Read after Write) – true dependence, as above
- RAR (Read after Read) – not a dependence, can freely reorder the reads
- WAR (Write after Read) – "antidependence": if you want to do the write (I2) early, just keep a copy of the old data and have I1 read that copy
- WAW (Write after Write) – if you want to reorder them, simply abort the write of I1 (if no one reads this word between I1 and I2)



RAW Read-after-Write
true Dependence

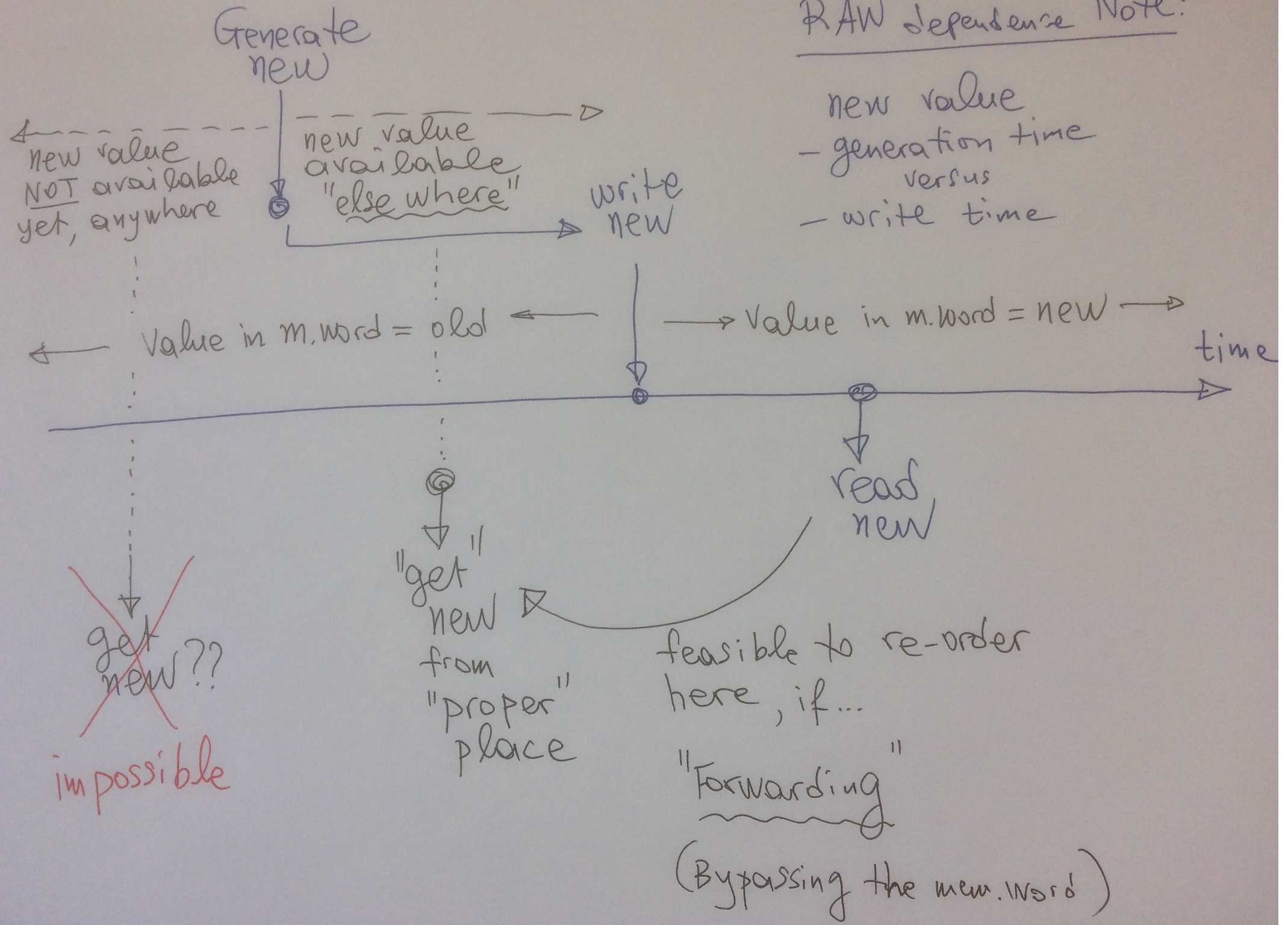


~~read new??~~

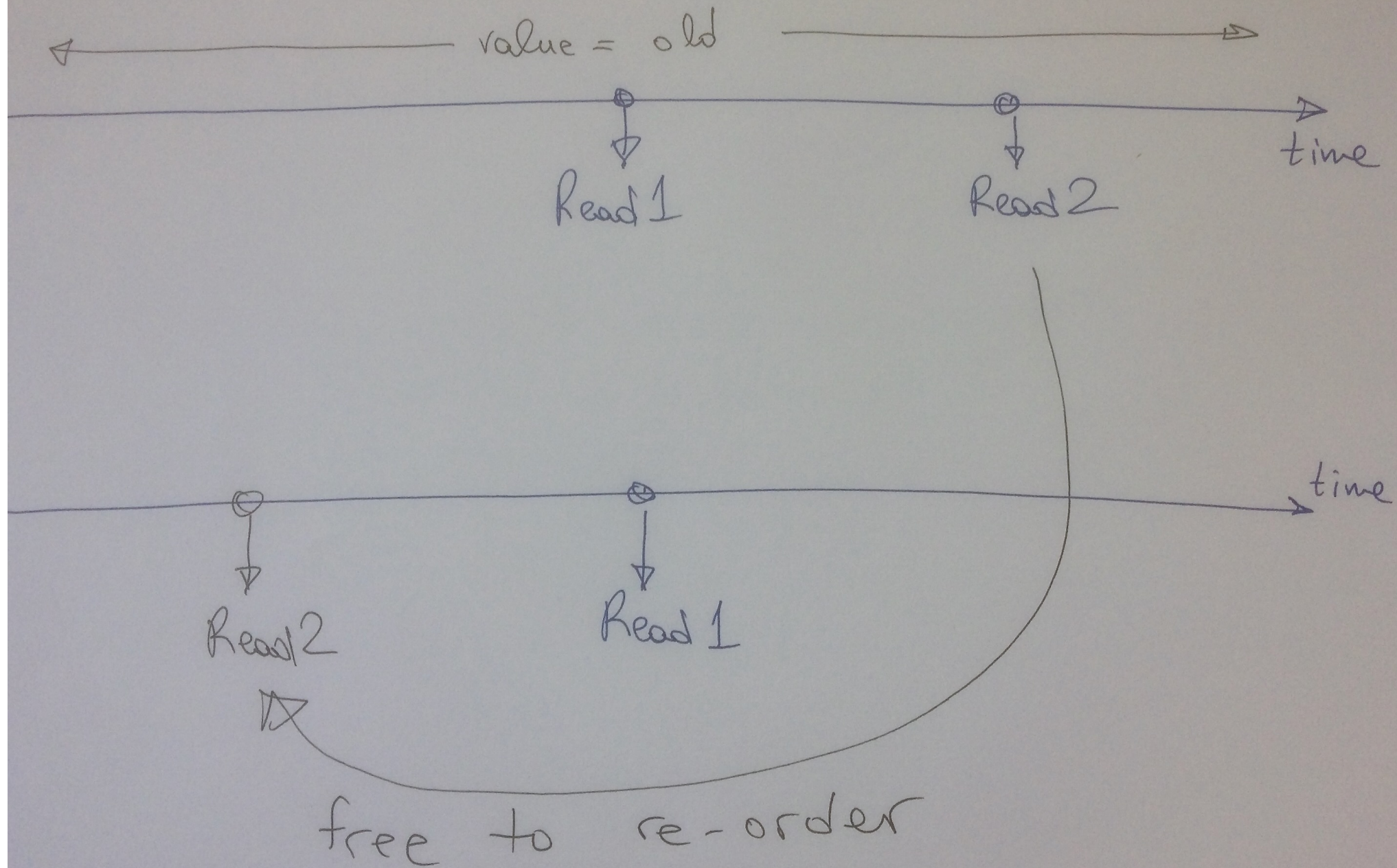
re-order is
NOT possible

RAW Dependence Note:

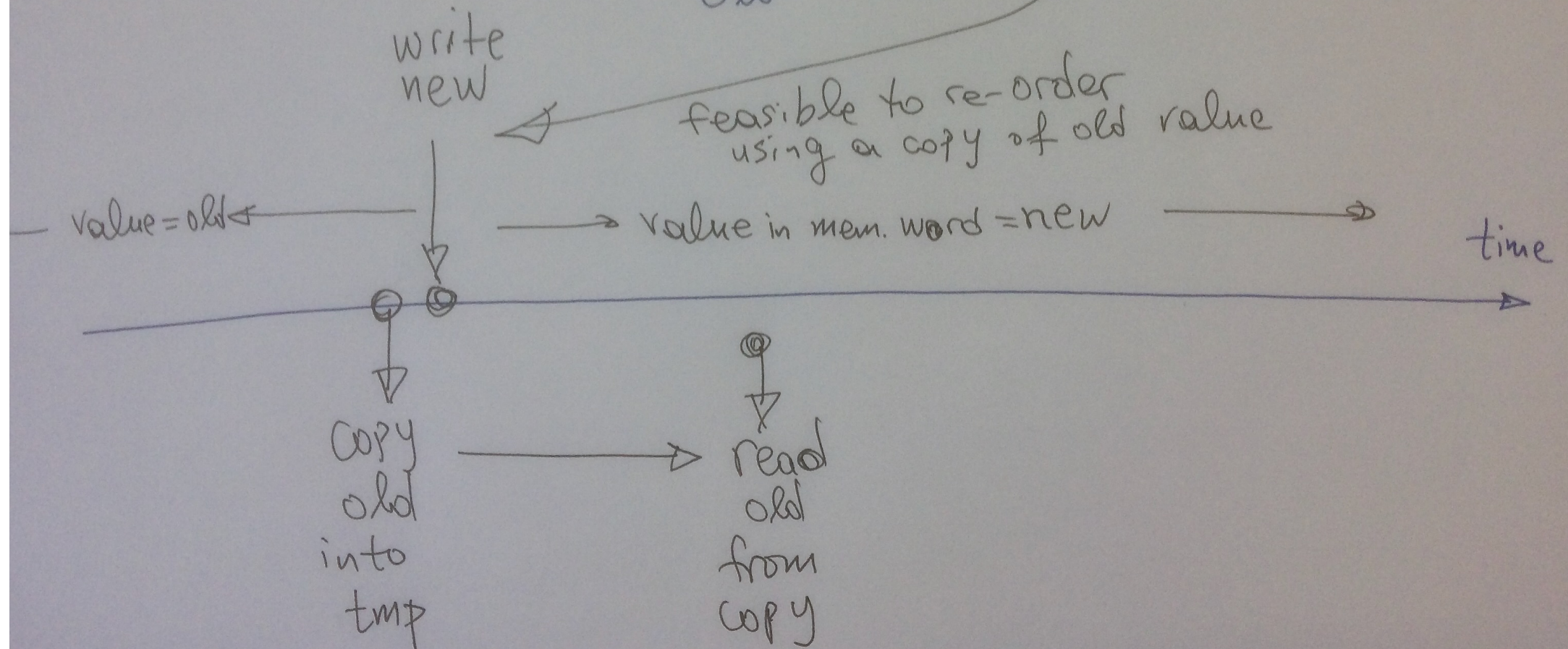
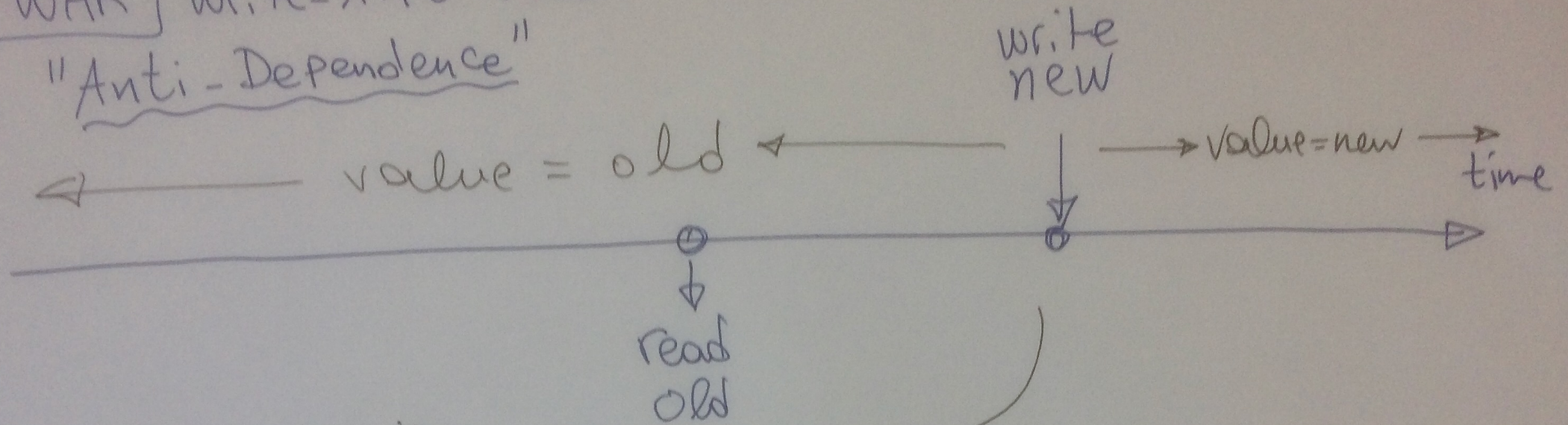
new value
- generation time
versus
- write time



RAR Read-After-Read : NOT a dependence



WAR write-After-Read
"Anti-Dependence"

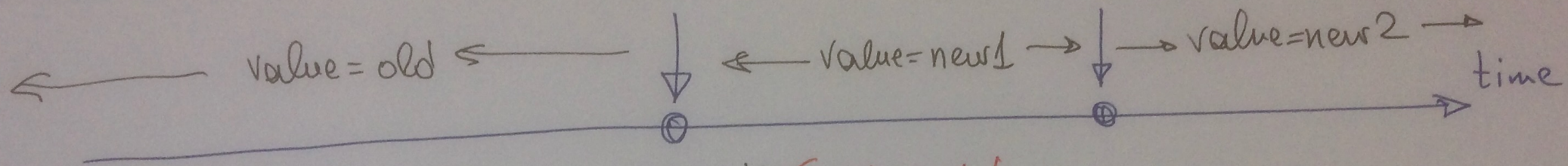


WAW

Write-After-Write

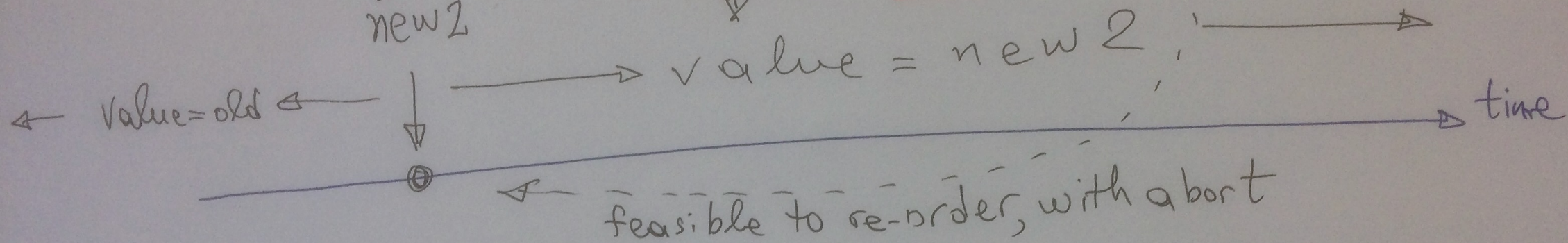
write
new1

write
new2



~~write new1~~ Abort!

write
new2

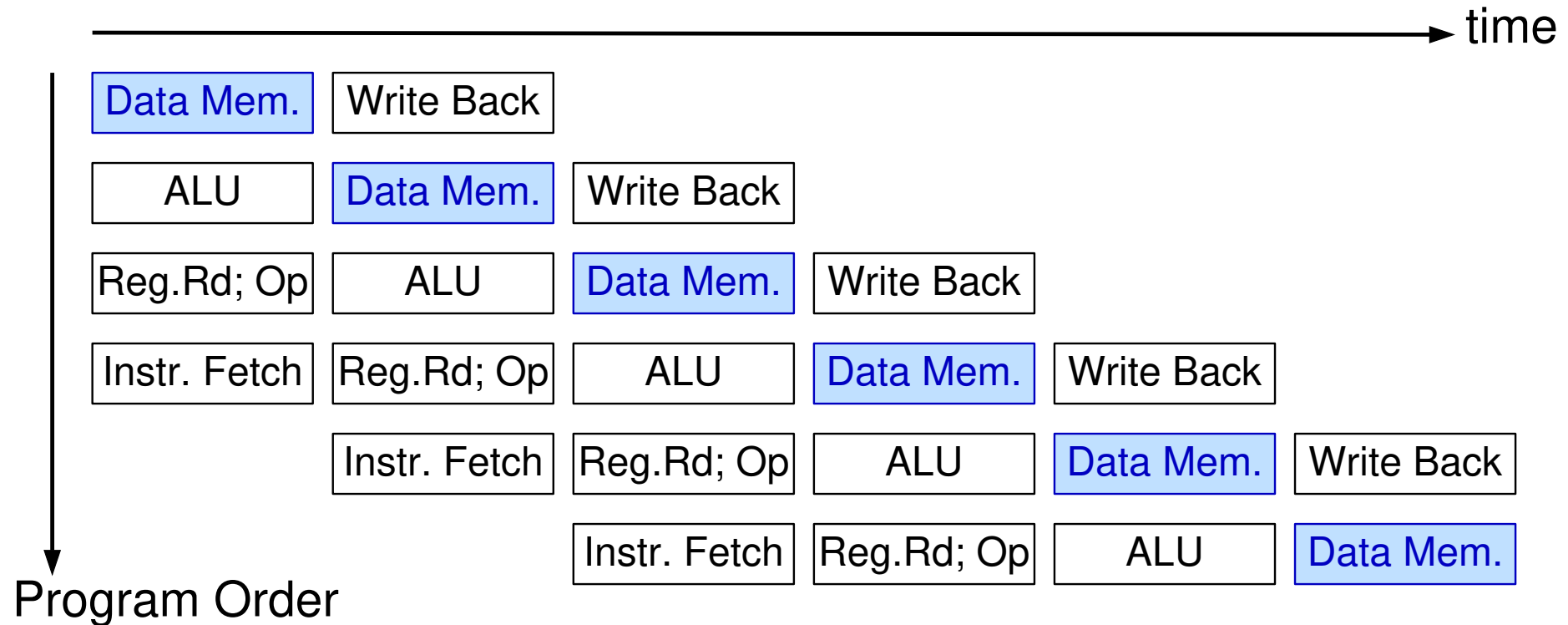


Γιατί να εμφανιστεί WAW (χωρίς ερμηνεία Read)? Άξενος ω;;

- π.χ. το write new2 μπορεί να γίνει πέρα σε if

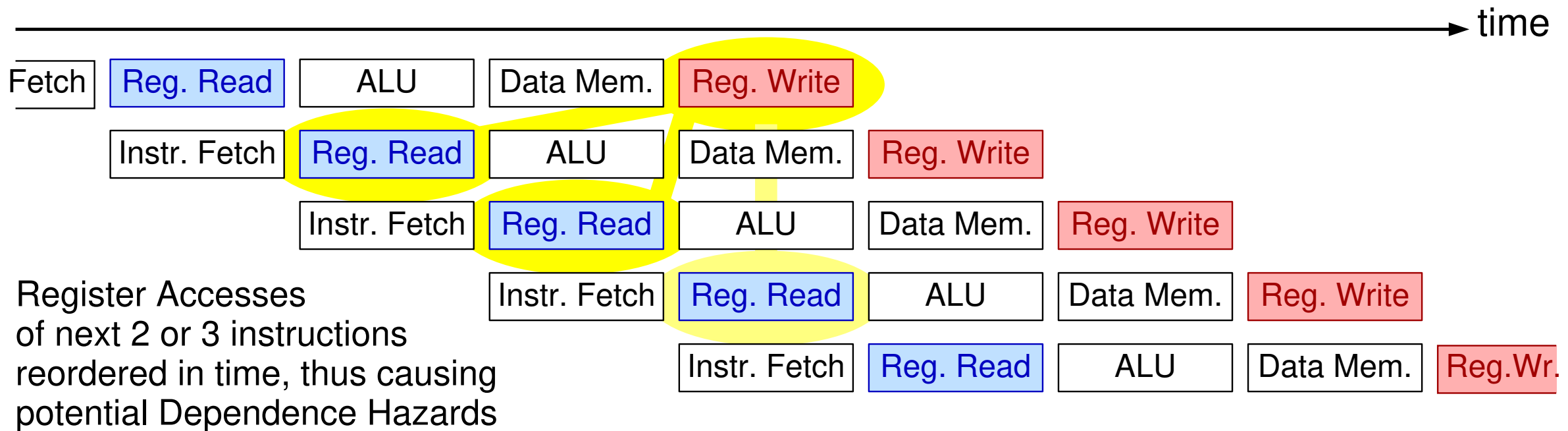
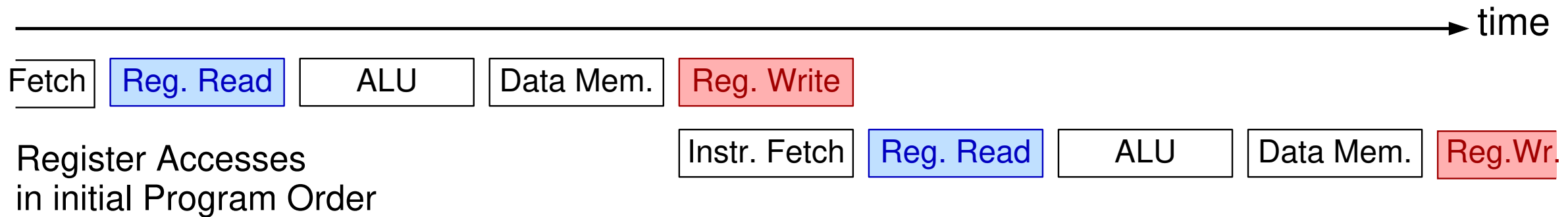
- π.χ. writes πέρα δύο pointers, *P, *Q ... και δεν ξέρουμε
 εάν $P = Q$

No Memory Data Hazards in our simple Pipeline



- Data Memory accesses are performed ‘in–order’ in our simple pipeline, i.e. are not reordered relative to what the program specifies, thus, no dependences of memory word accesses are ever violated

Register Accesses reordered, with Pipelining



- For each instruction that writes a destination register, if the next 2 or 3 instructions read that same register, i.e. need its result, we have to do something about it...


```

60: ld  x10, 40(x1)
64: sub x11, x2, x3
68: add x12, x3, x4
72: sd  x13, 48(x1)
76: add x14, x11, x6

```

x1	100	x10	10 14
x2	200	x11	110 168
x3	32	x12	120
x4	400	x13	130

Memory:

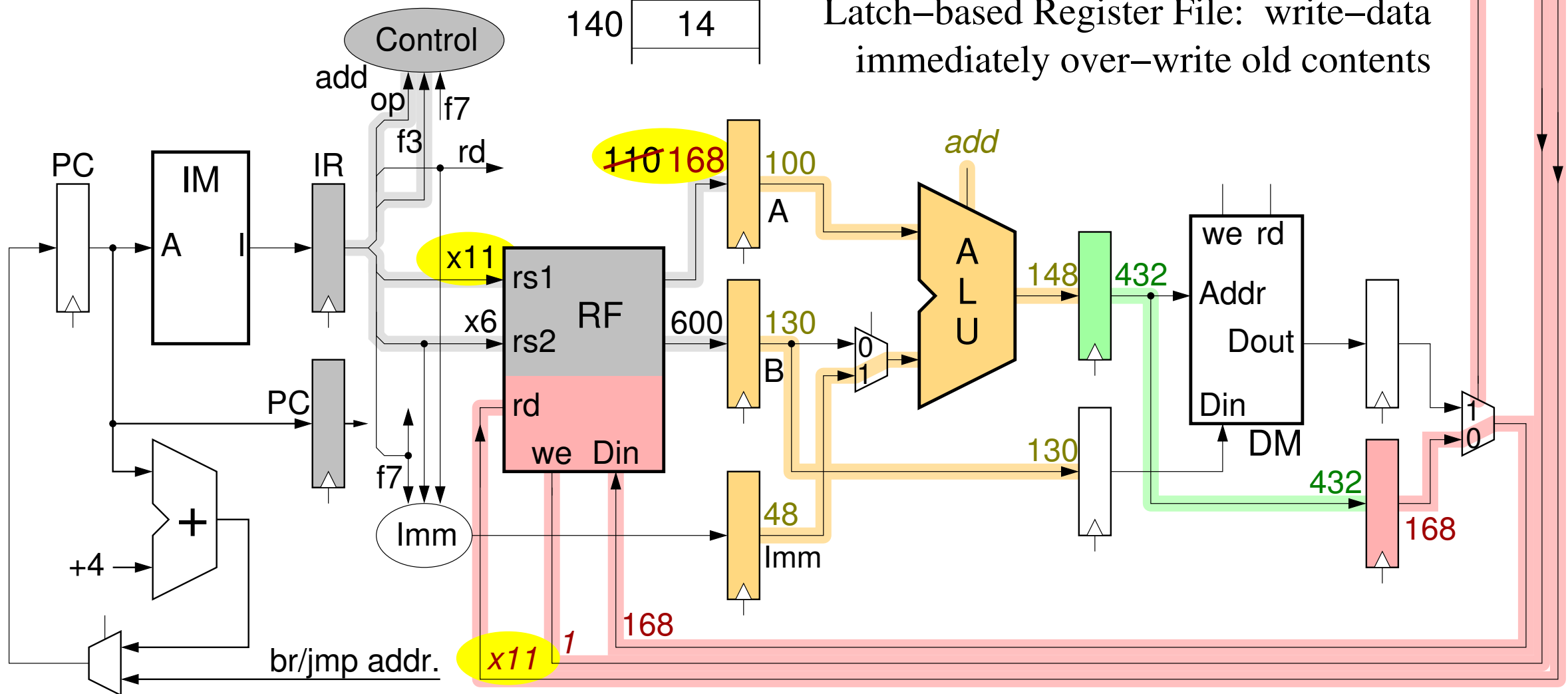
140	14
-----	----

Instruction Distance 3

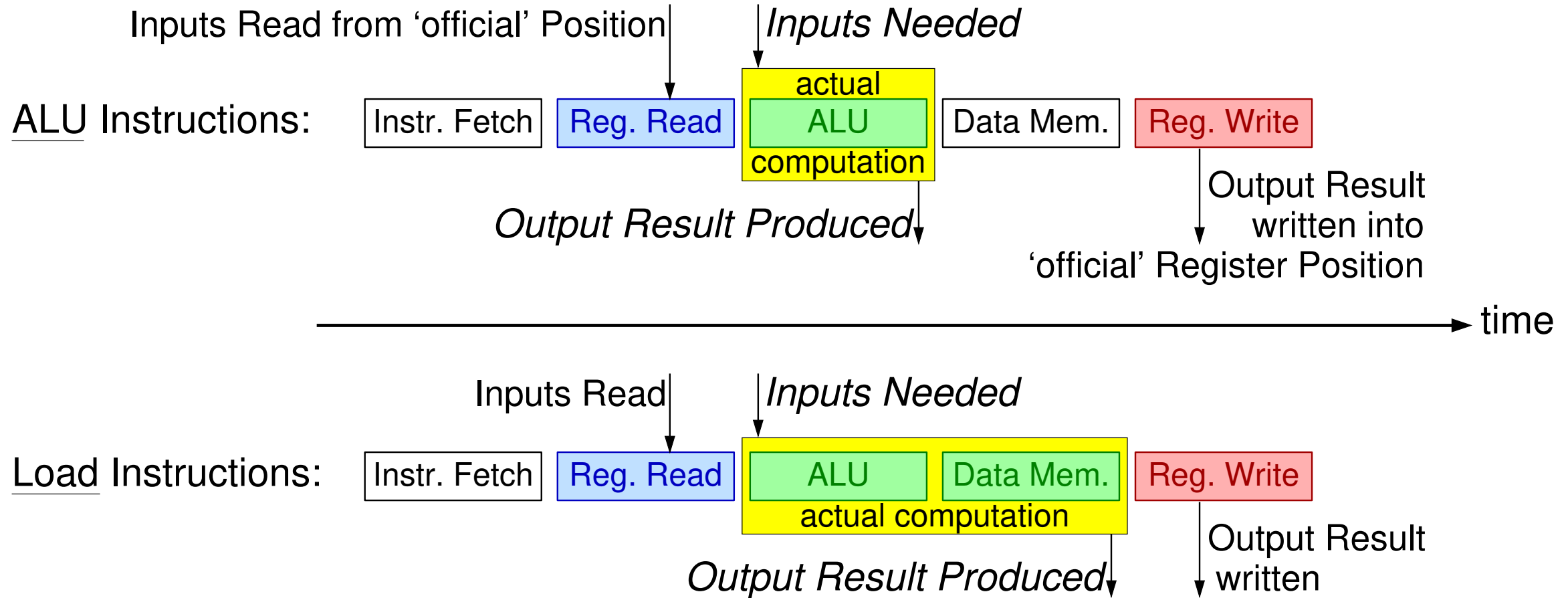
*nothing special
needed,
just latch-based RF*

(Cycle 6)

Latch-based Register File: write-data
immediately over-write old contents



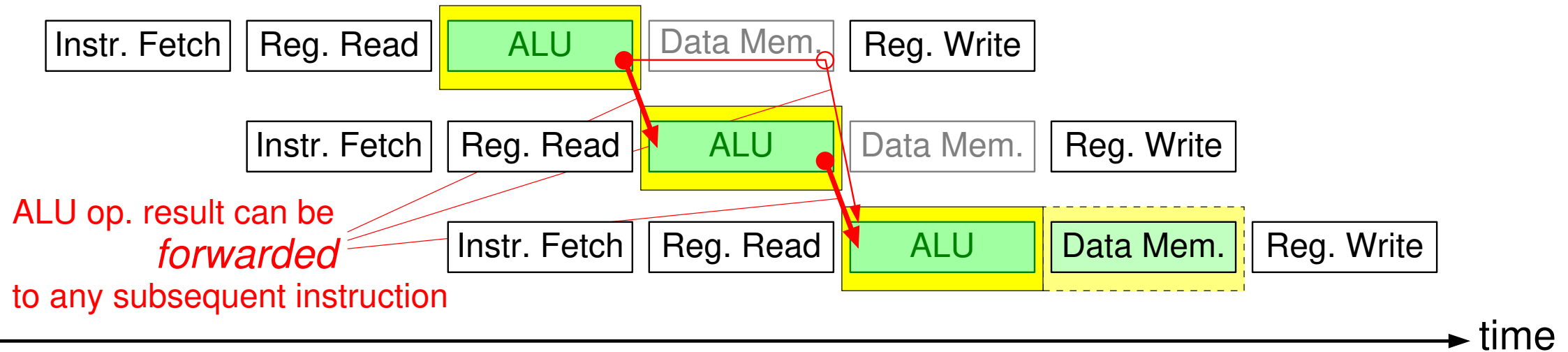
Actual Need–Produce Time vs. from/in–Register Time



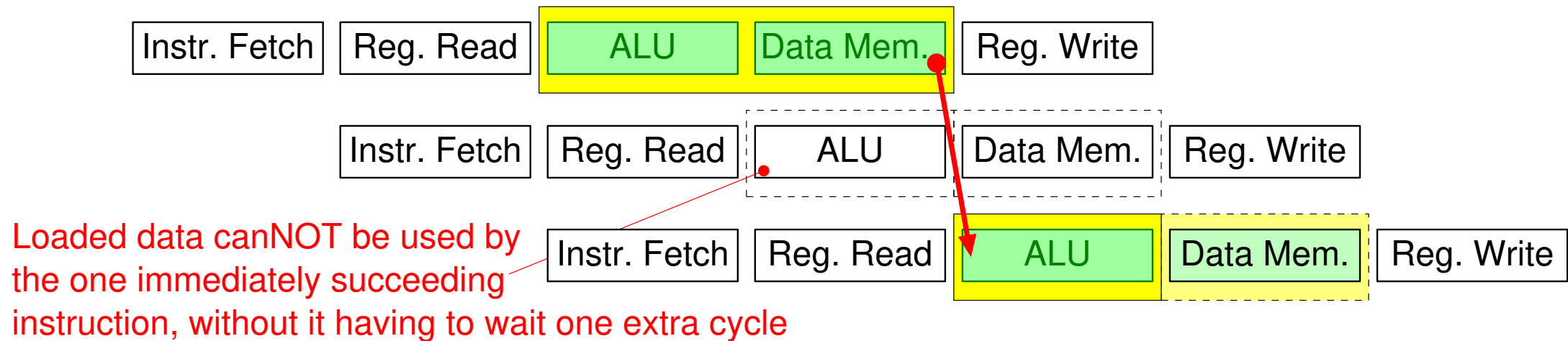
- All we care about is actual Results 'Forwarded' from Producer to Consumer instruction
- We can 'Bypass' the 'official' loop through the Register File for immediate–use Results

ALU result to next I; Load result to next-after-next I

from ALU Instructions:



from Load Instruction:



- ALU instructions never stall the pipeline, but Load instructions will do so when immediately followed by a dependent instruction

```

60: ld  x10, 40(x1)
64: sub x11, x2, x3
68: add x12, x3, x4
72: sd  x13, 48(x11)
76: add x14, x5, x6

```

x1	100	x10	10 14
x2	200	x11	110 168
x3	32	x12	120
x4	400	x13	130

Memory:

140	14
-----	----

Forward to Distance 2

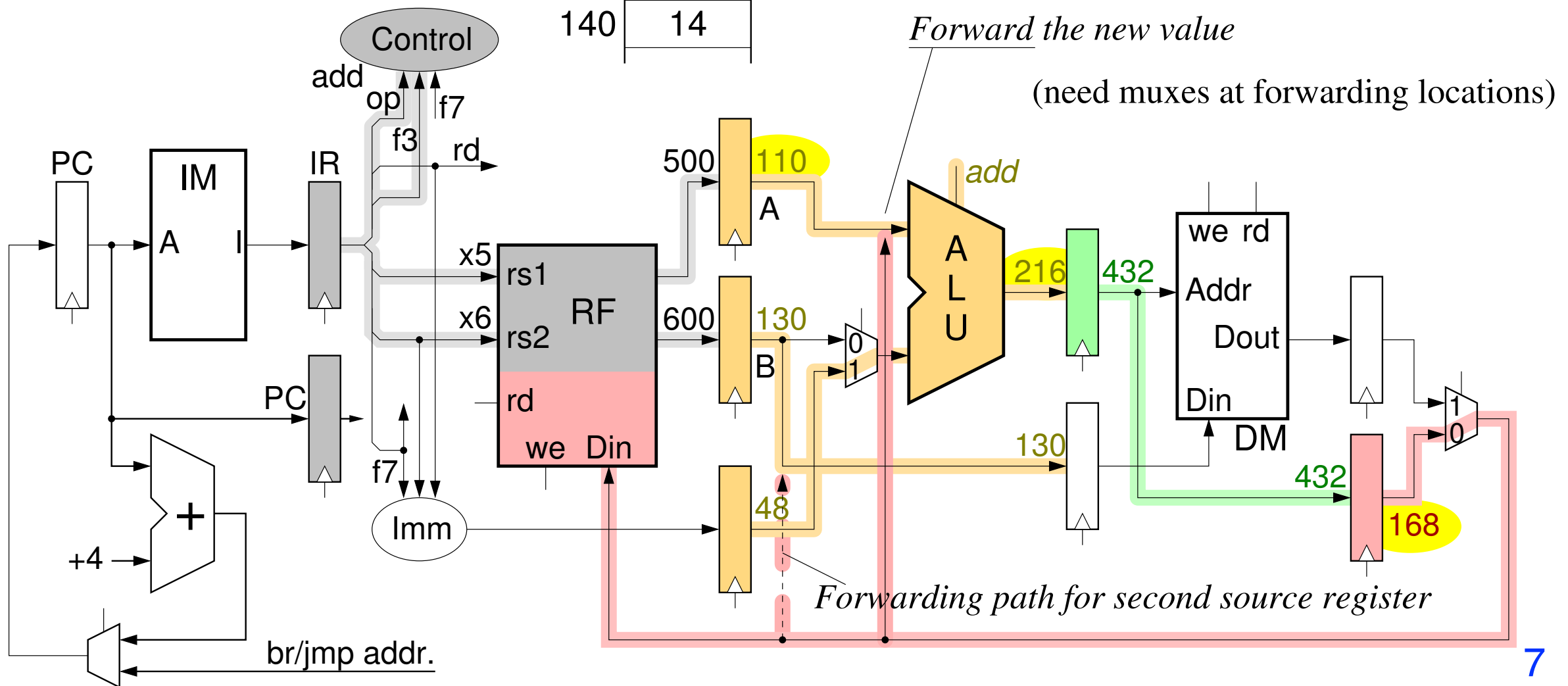
e.g. from ALU op. (Cycle 6)

Erroneous old value coming from stage 2

Correct new value available, in another place

Forward the new value

(need muxes at forwarding locations)




```

60: ld  x10, 40(x1)
64: sub x11, x2, x3
68: add x12, x11, x4
72: sd  x13, 48(x1)
76: add x14, x5, x6

```

x1	100	x10	10 14
x2	200	x11	110 168
x3	32	x12	120
x4	400	x13	130

Memory:

140	14
-----	----

Forward to Distance 1

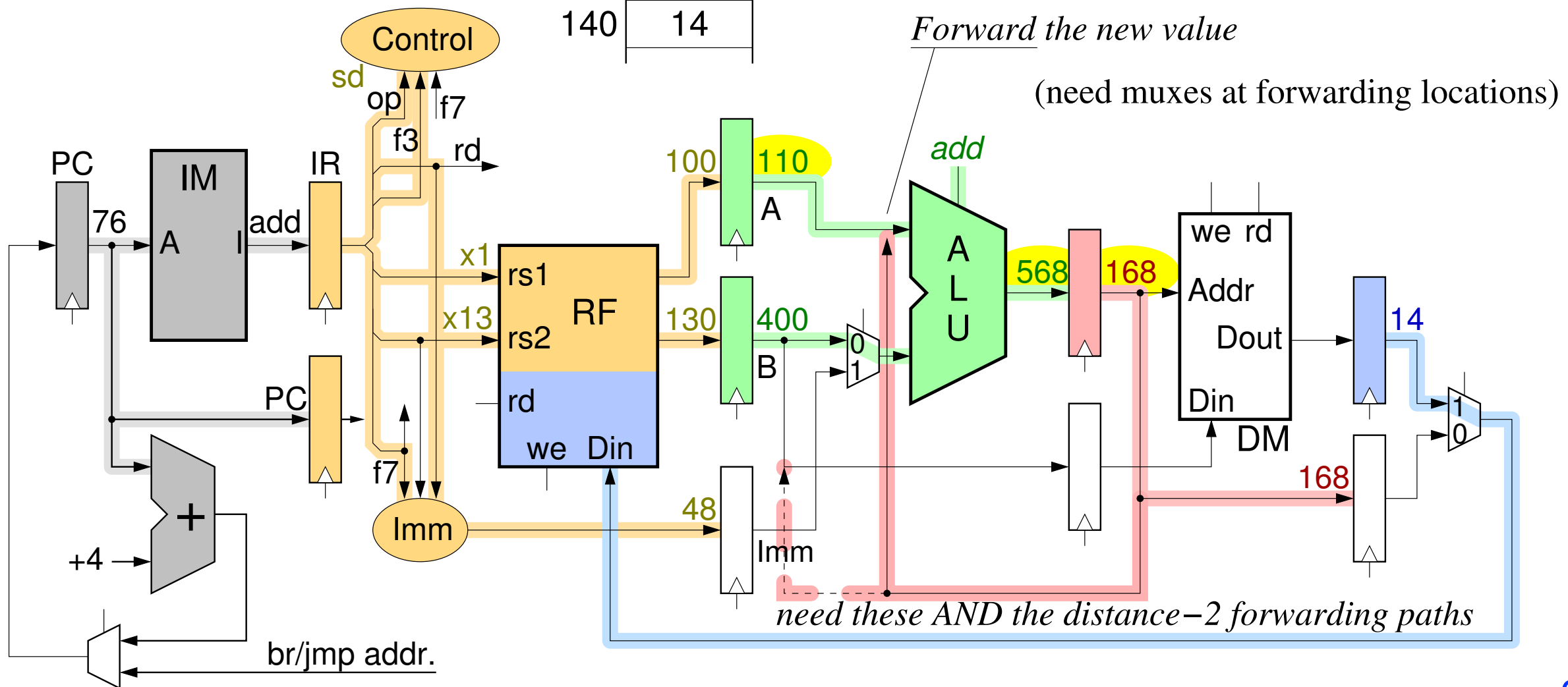
from ALU op. ONLY (Cycle 5)

Erroneous old value coming from stage 2

Correct new value available, in another place

Forward the new value

(need muxes at forwarding locations)



60: ld x10, 40(x1)

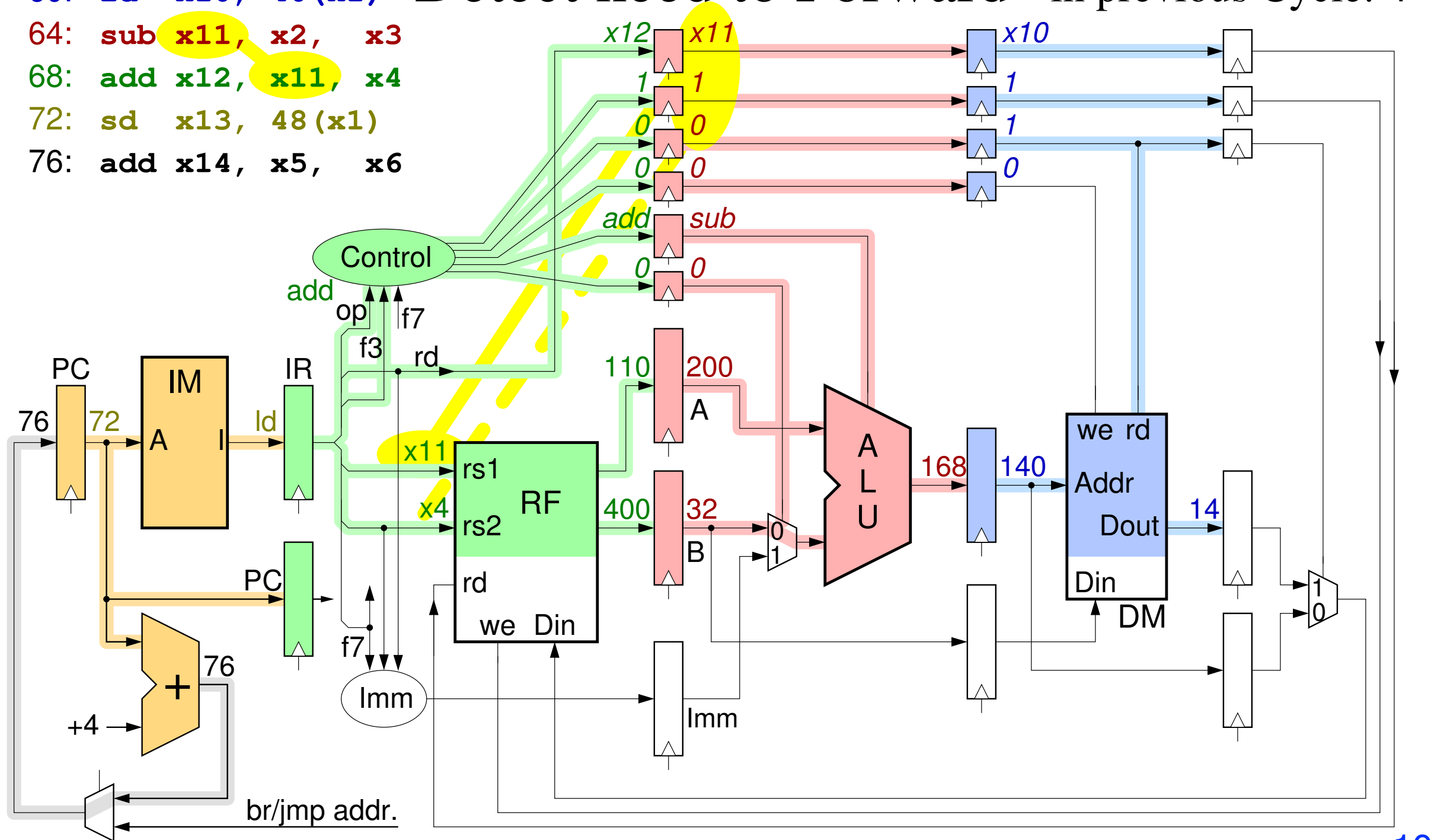
64: sub x11, x2, x3

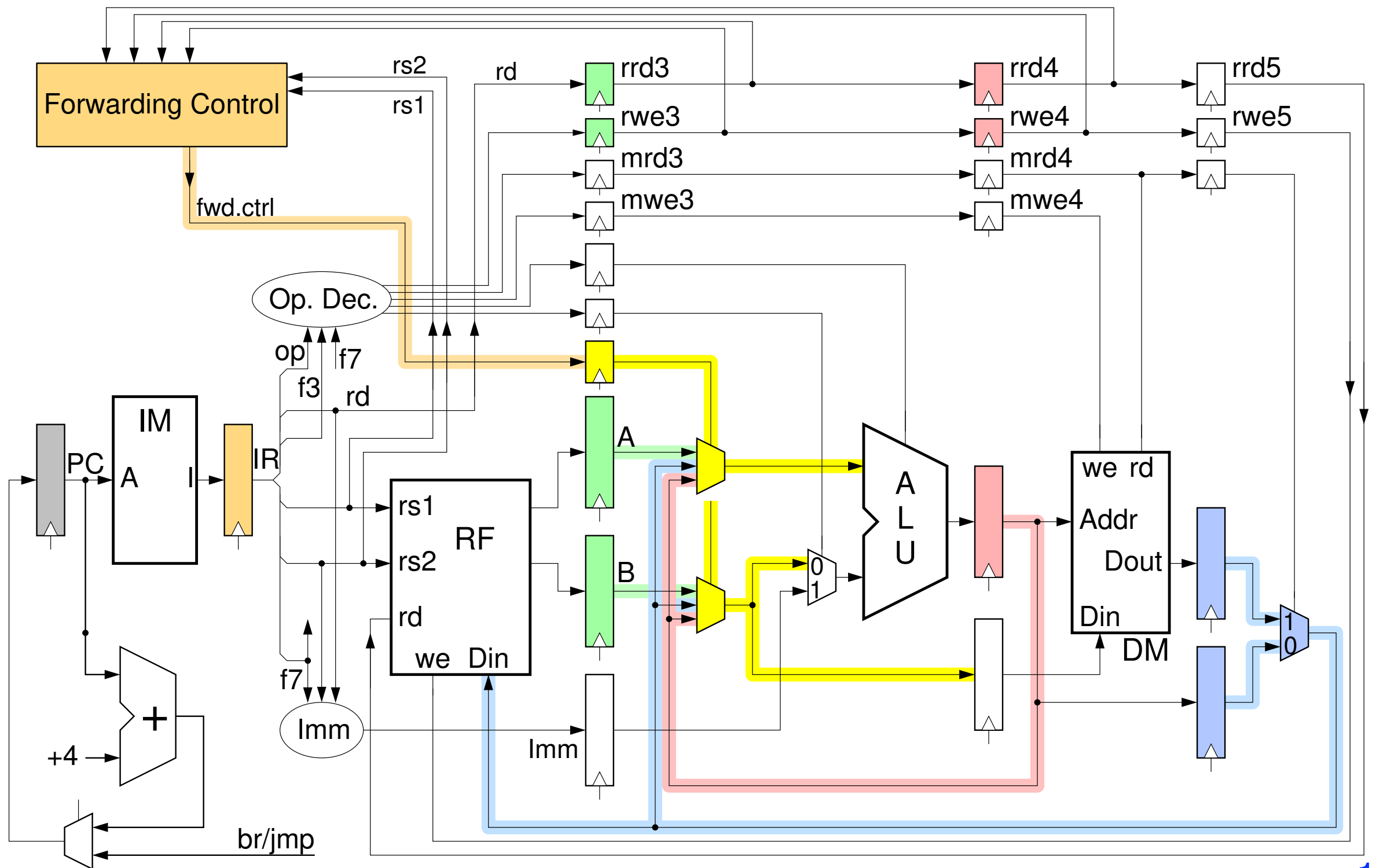
68: add x12, x11, x4

72: sd x13, 48(x1)

76: add x14, x5, x6

Detect need to Forward – in previous Cycle: 4



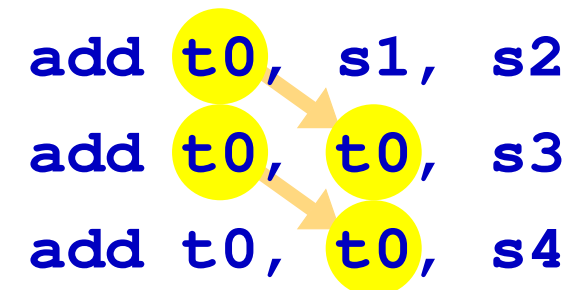


Forwarding Control Logic

Assuming that the instruction in stage3 is NOT a Load

- Define *Match(rs, rd)* as follows:
 $(rs == rd \neq x0) \text{ AND } (rd.writeEnable == ON)$
- if (*Match(rs1, rrd3)*) then { *prepare_to_forward_from_stage4_to_A* }
 else if (*Match(rs1, rrd4)*) then { *prepare_to_forward_from_stage5_to_A* }
 else { *no_forwarding_to_A_will_be_needed* }
- if (*Match(rs2, rrd3)*) then { *prepare_to_forward_from_stage4_to_B* }
 else if (*Match(rs2, rrd4)*) then { *prepare_to_forward_from_stage5_to_B* }
 else { *no_forwarding_to_B_will_be_needed* }

*The stage3 instruction is more recent than the stage4 one,
hence the stage3 result has PRIORITY in forwarding:*




```

60: ld x10, 40(x1)
64: sub x11, x2, x3
68: add x12, x10, x4
72: sd x13, 48(x1)
76: add x14, x5, x6

```

x1	100	x10	10 14
x2	200	x11	110 168
x3	32	x12	120
x4	400	x13	130

Memory:

140	14
-----	----

Forward to Distance 2

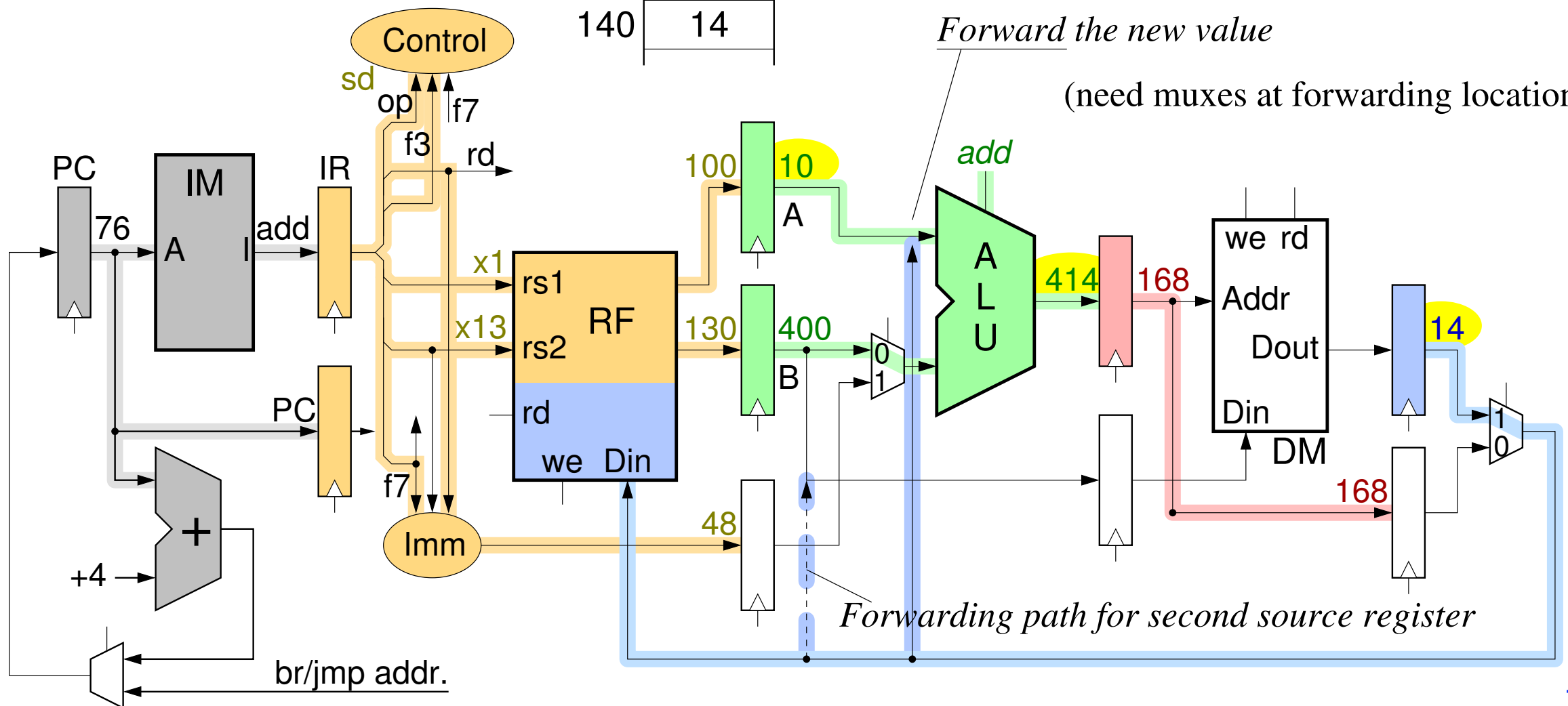
e.g. from Load Instr. (Cycle 5)

Erroneous old value coming from stage 2

Correct new value available, in another place

Forward the new value

(need muxes at forwarding locations)



```
60: ld x10, 40(x1)
64: sub x11, x10, x3
68: add x12, x3, x4
72: sd x13, 48(x1)
76: add x14, x5, x6
```

x1	100	x10	10 ?
x2	200	x11	110 ?
x3	32	x12	120
x4	400	x13	130

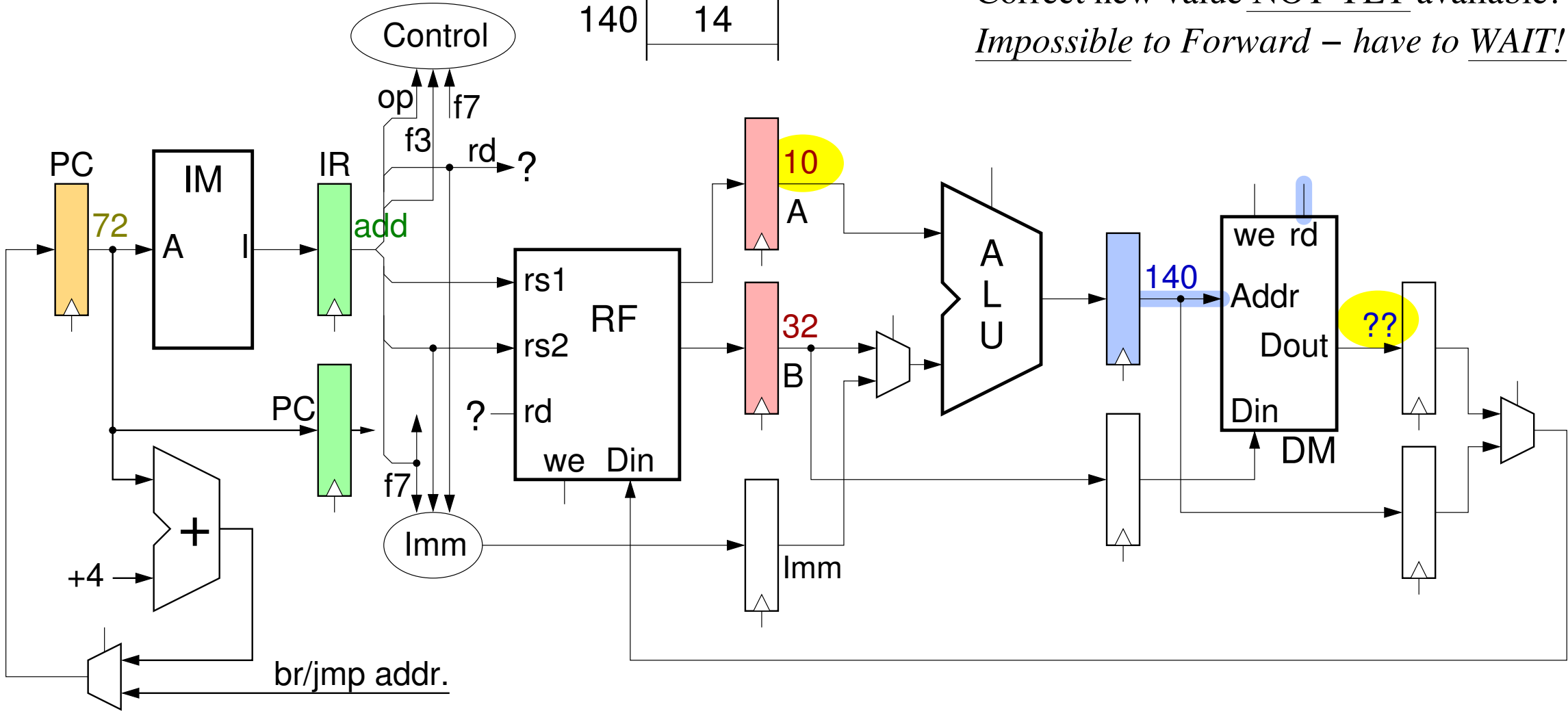
Memory:

140	14
-----	----

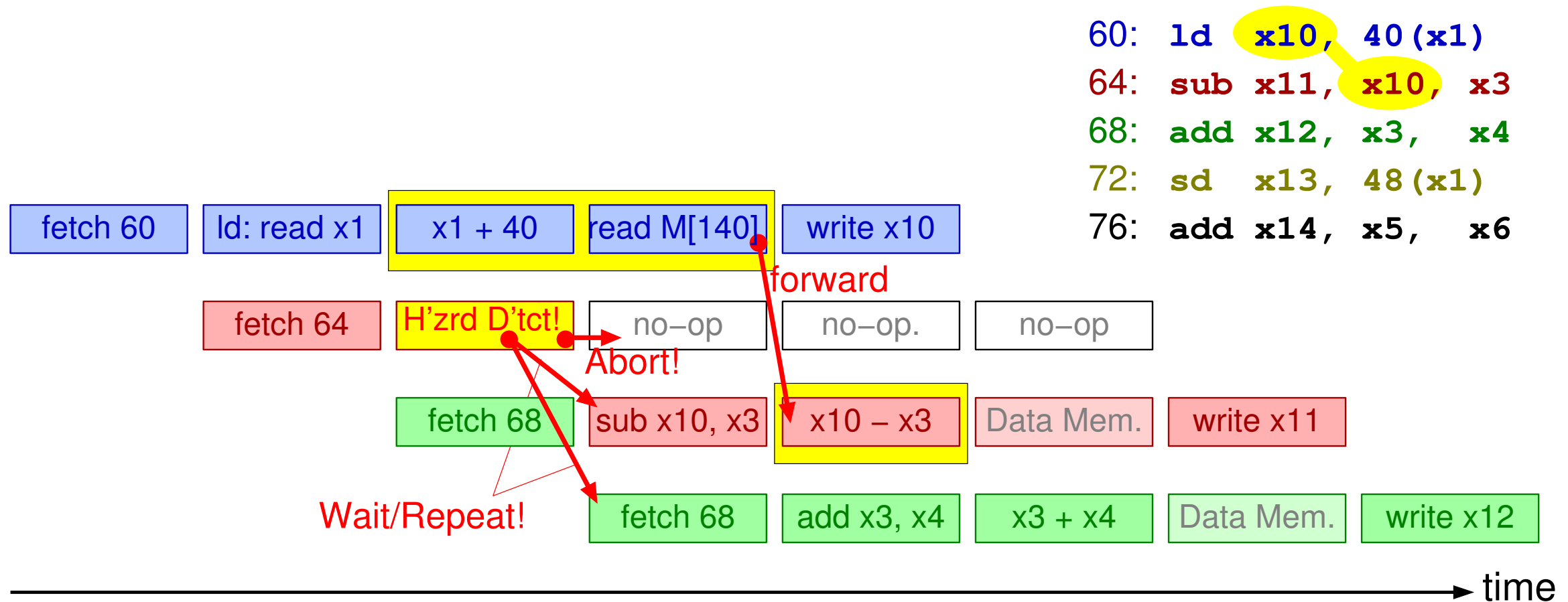
Dist. 1 dep. on LOAD

Dependence on immed. preceding Load
(start of Cycle 4)

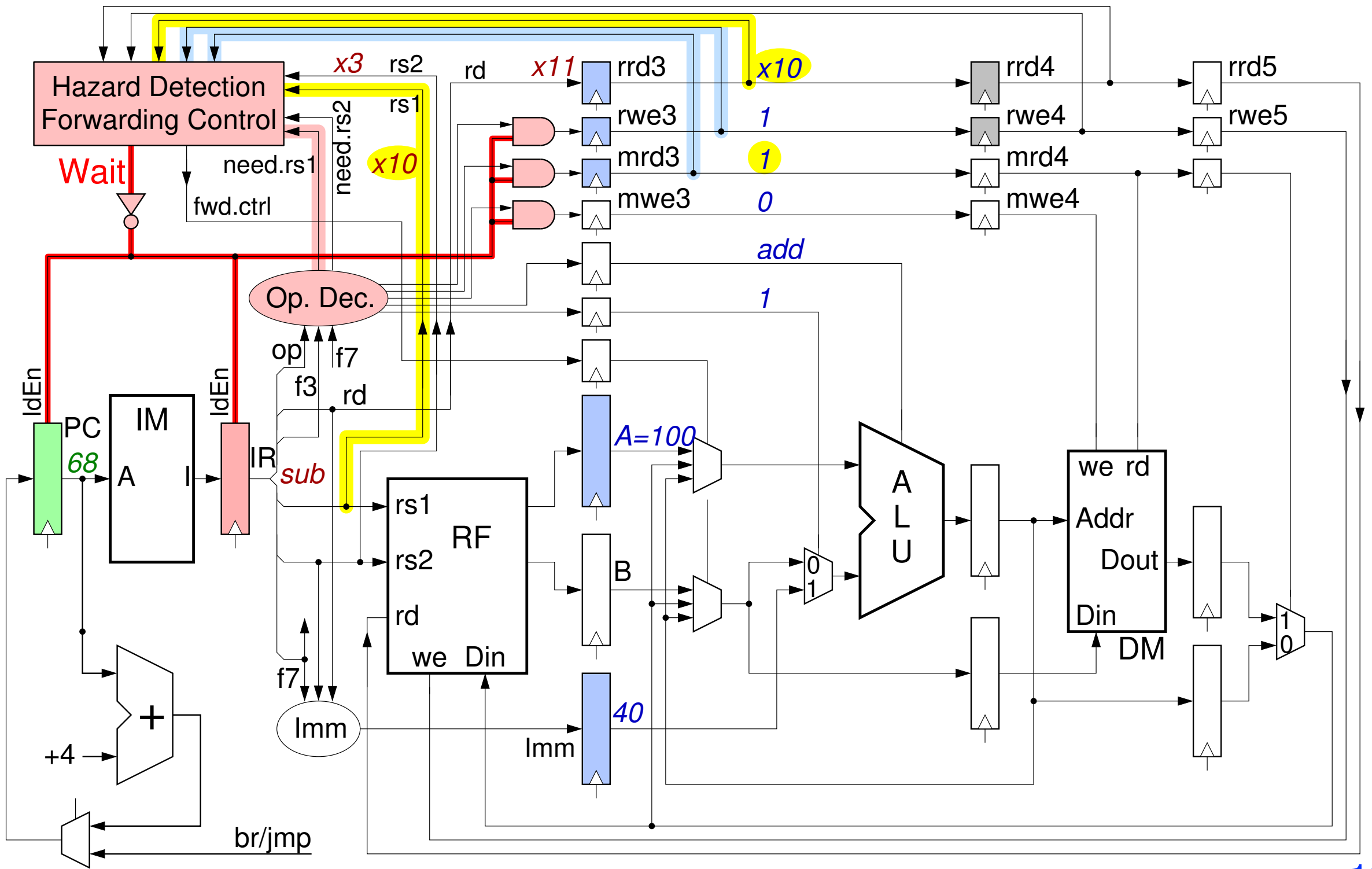
Erroneous old value coming from stage 2
Correct new value NOT YET available!
Impossible to Forward – have to WAIT!



Distance 1 dependence on LOAD: Wait!



- The instruction immediately after a Load wants to use the load'ed data
- Impossible without losing one cycle:
 force this instruction to wait (repeat itself on the next cycle)
- Simple, in-order Pipeline: the next instruction has to wait too!



Hazard Detection Logic

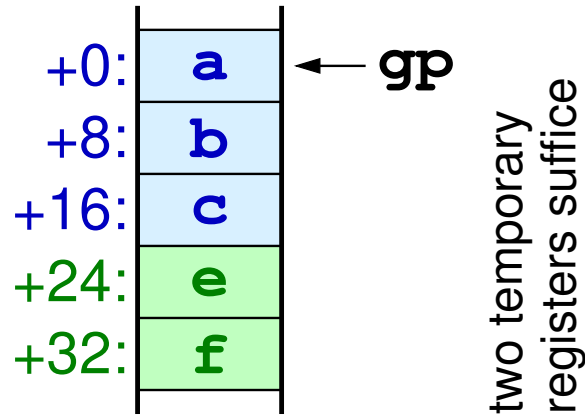
- *Wait* =
 $(\text{mrd3}) \text{ AND } (\text{Match}(\text{rs1}, \text{rrd3})) \text{ AND } (\text{need_rs1})$
OR
 $(\text{mrd3}) \text{ AND } (\text{Match}(\text{rs2}, \text{rrd3})) \text{ AND } (\text{need_rs2})$

i.e. the previous instruction is a Load and I need the register that it will write

- Not all instructions need the register that happens to be specified by their rs1 field, when that field is non-existent for them, e.g. J/U instructions like 'lui' (load upper immediate) or 'jal' (jump and link)
- Several instructions do not need the register that happens to be specified by their rs2 field: besides those with J/U format, the ones with I-format also do not have an rs2 field –most notably: addi and load

Instruction Scheduling

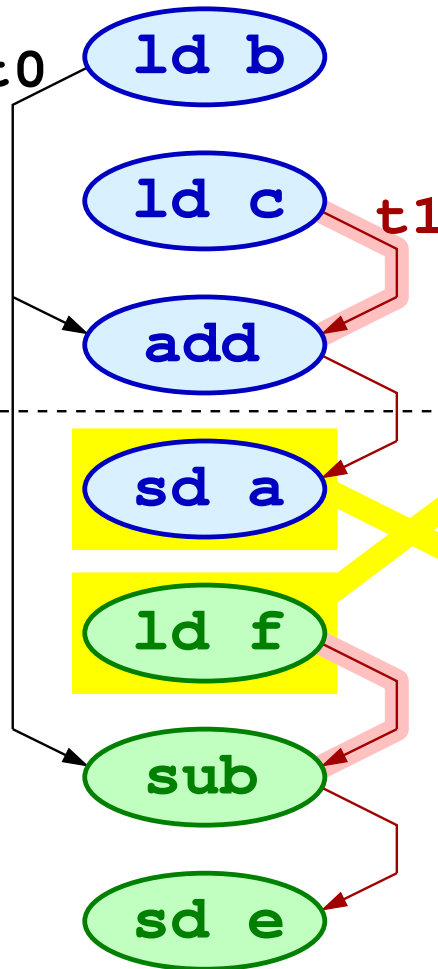
```
a = b + c;
e = b - f;
```



```
ld t0, 8(gp)
ld t1, 16(gp)
add t1, t0, t1
sd t1, 0(gp)
ld t1, 32(gp)
sub t1, t0, t1
sd t1, 24(gp)
```

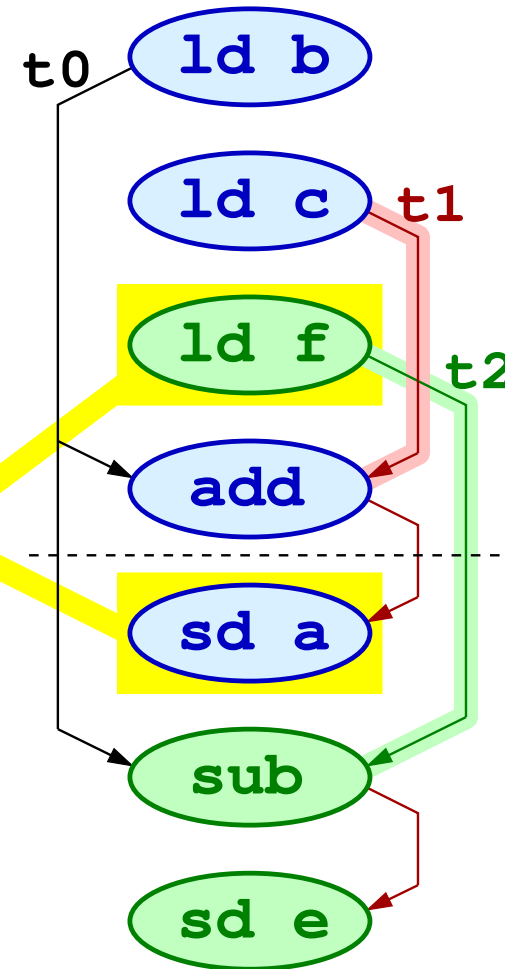
2 extra clock cycles lost

What if the program is?:
RAW dependence?



This is 'Static' Scheduling, at Compile Time

```
a[i] = b + c;
e = b - a[j];
```



the more things you have
'up in the air' (in parallel),
the more temporary
registers you need
in order to 'name'
those 'pending' values

```
ld t0, 8(gp)
ld t1, 16(gp)
ld t2, 32(gp)
add t1, t0, t1
sd t1, 0(gp)
sub t1, t0, t2
sd t1, 24(gp)
```

No extra clock cycle lost

- Does the compiler know for sure if $i \neq j$ (OK to reorder sd–ld) or $i == j$ (fwd in reg.)?
- If unknown to compiler, static sch. impossible
=> dynamic scheduling at runtime (ooo pipe)