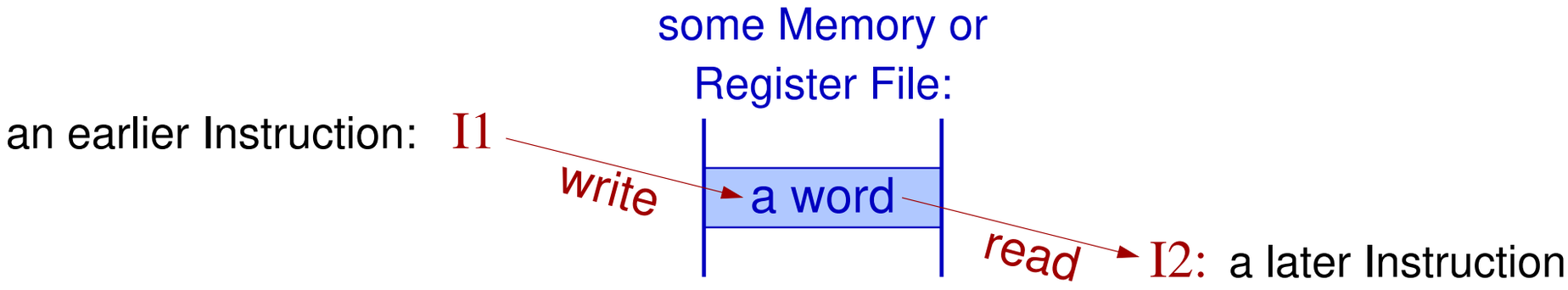
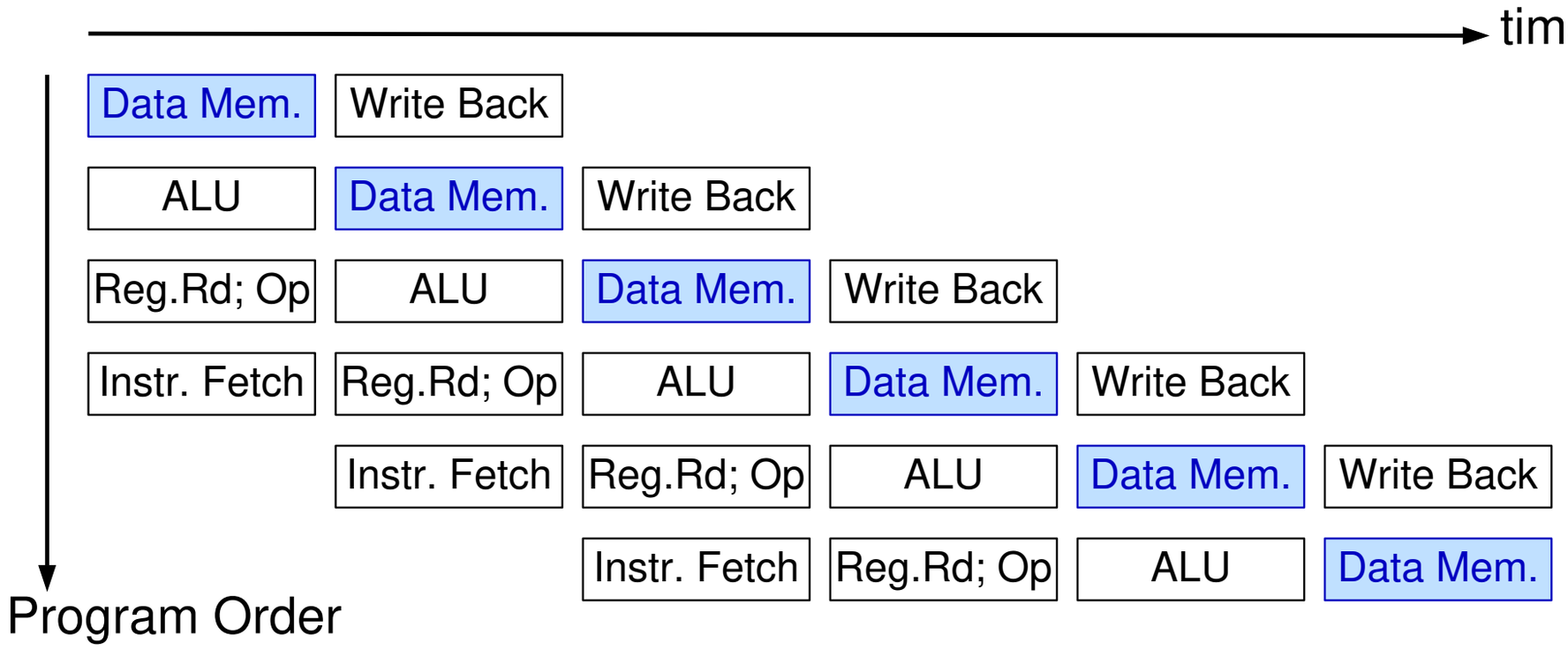
Data Dependences (Hazards) in Pipelines



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

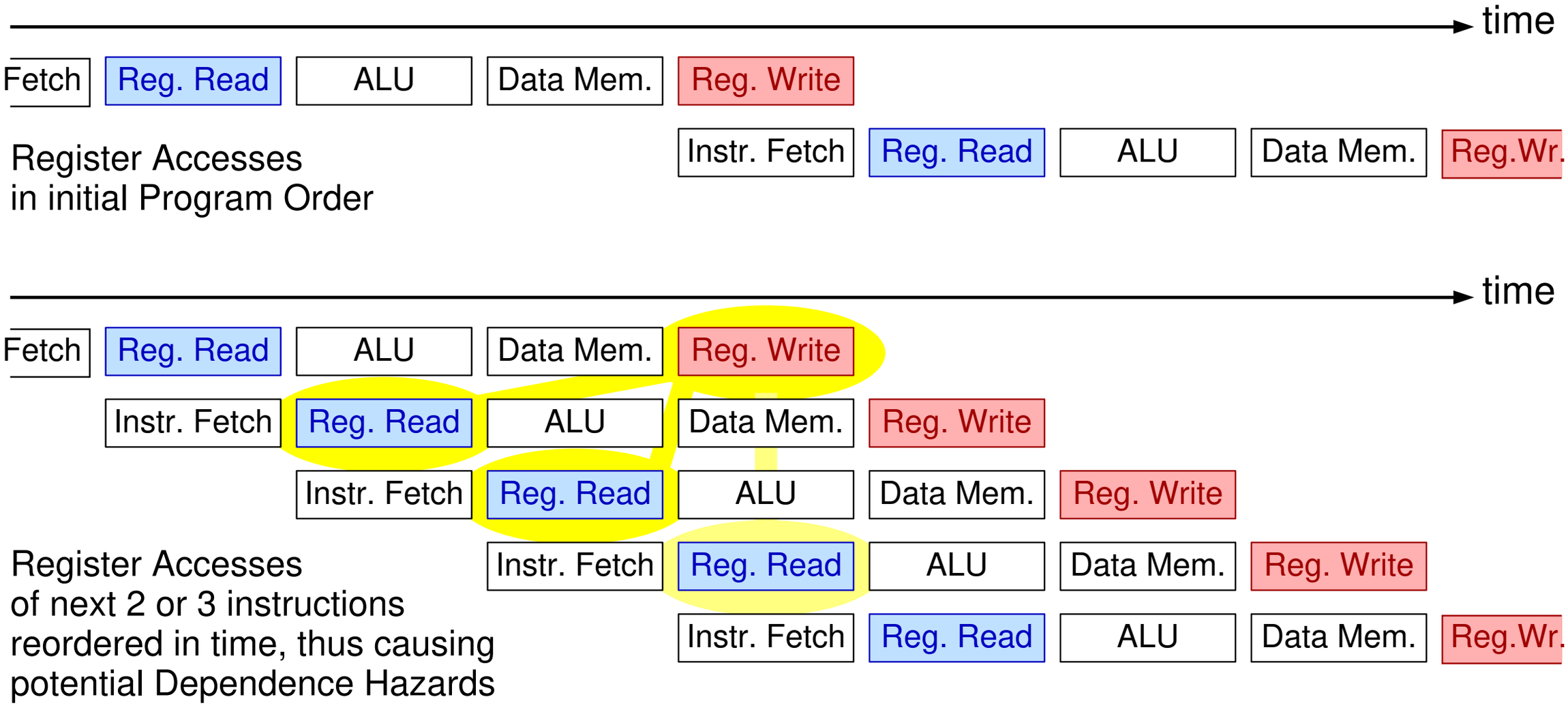
- RAW (Read after Write) – true dependence, as above
- RAR (Read after Read) – not a dependence, can freely reorder the reads
- WAR (Write after Read) – "antidependence": if you want to do the write (I2) early, just keep a copy of the old data and have I1 read that copy
- WAW (Write after Write) – if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

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...

Instruction Distance 3

```
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
x2	200
x3	32
x4	400

x10	~~10~~ 14
x11	~~110~~ 168
x12	120
x13	130

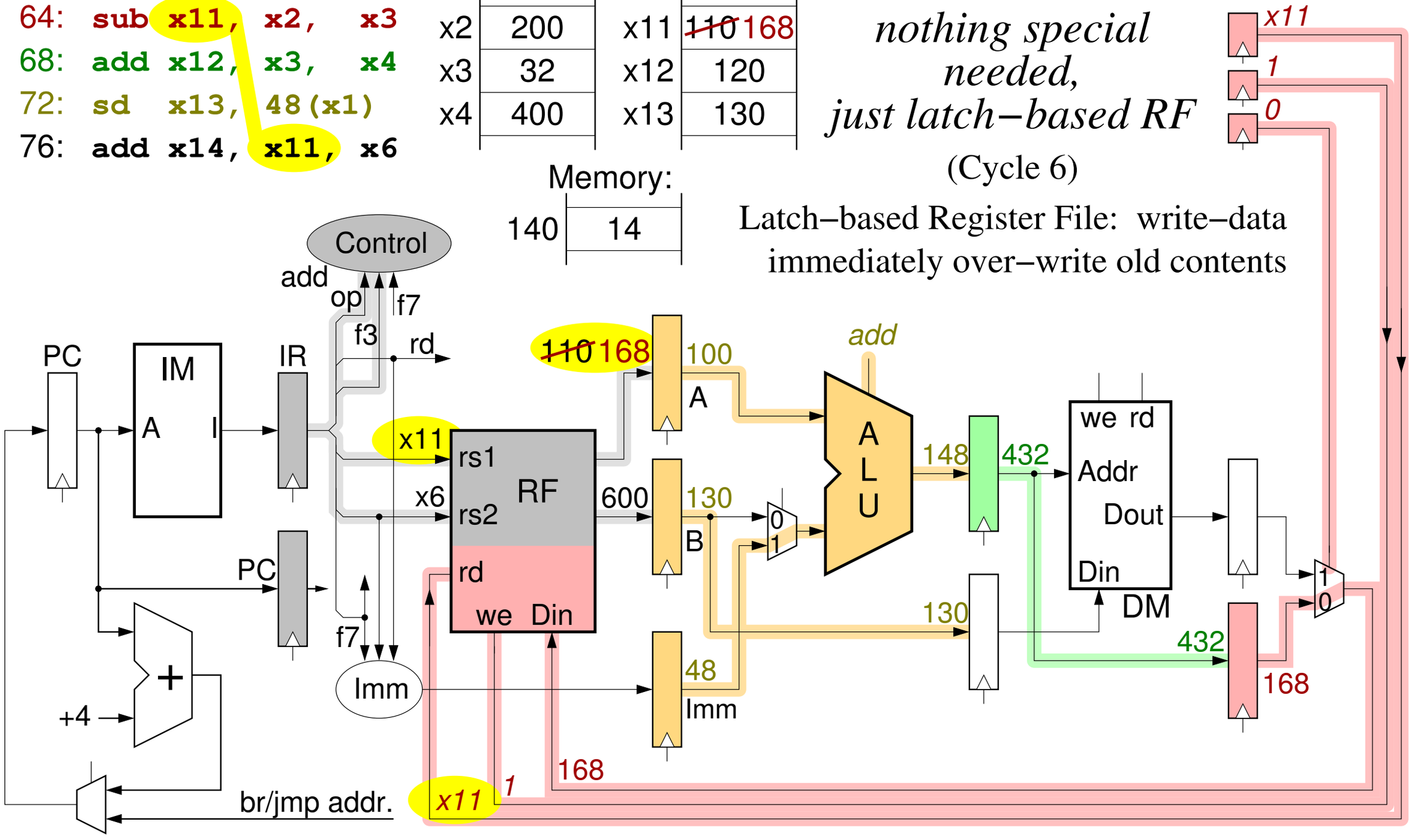
Memory:

140	14

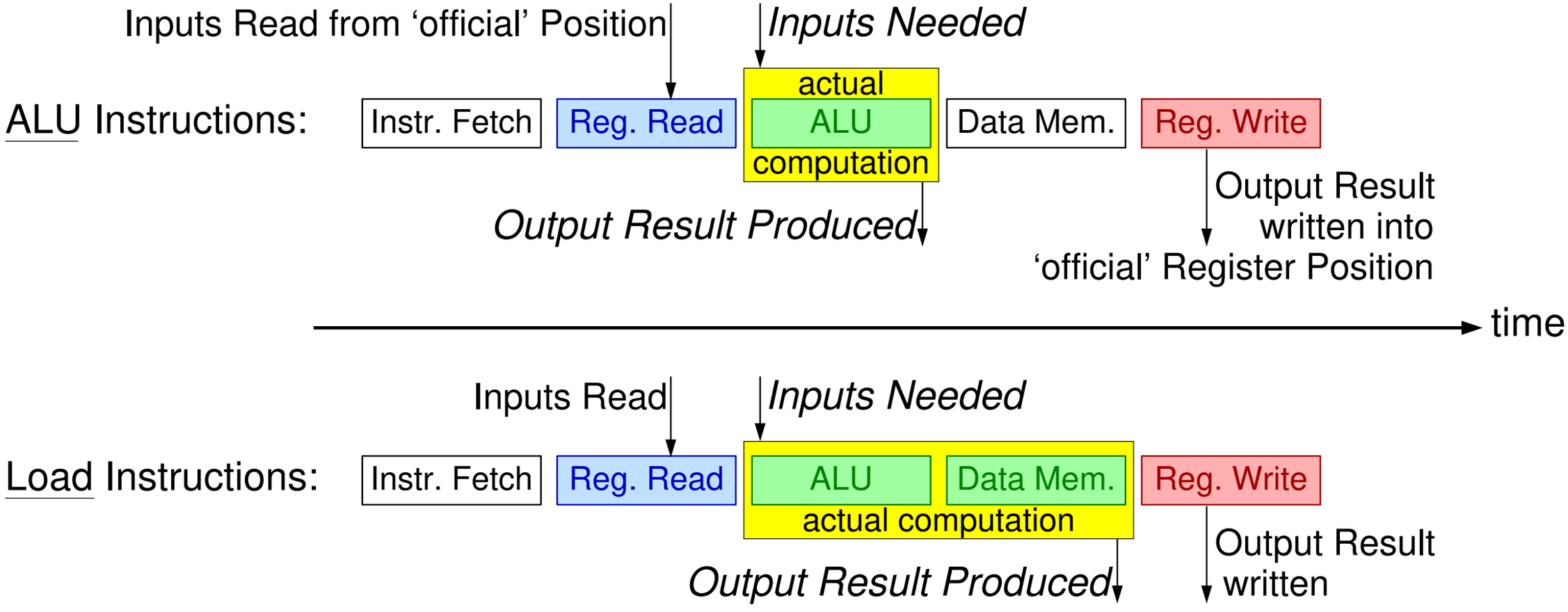
nothing special needed, just latch–based RF

(Cycle 6)

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



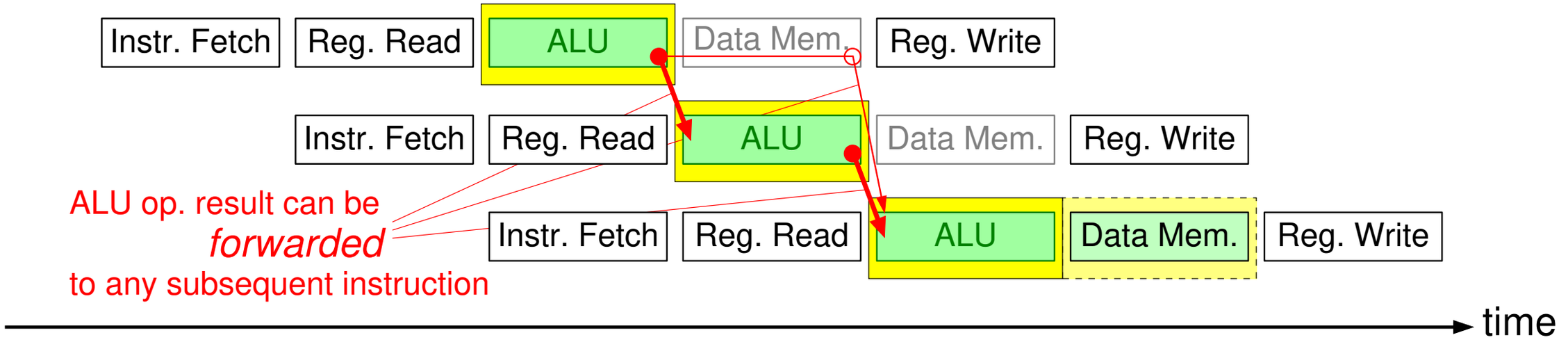
Actual Need–Produce Time vs. from/in–Register Time



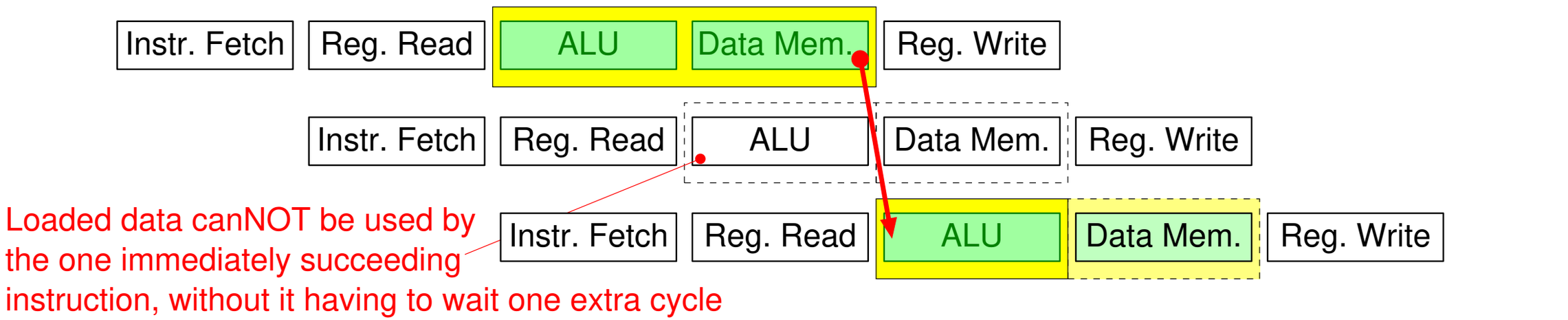
- All we care about is actual Results 'Forwarded' from Producer to Consumer instruction
- We can 'Bypass' the 'official' loop through the Register File for immediate–use Results

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

Forward to Distance 2

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

```
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
x2	200
x3	32
x4	400

x10	~~10~~ 14
x11	~~110~~ 168
x12	120
x13	130

Memory:

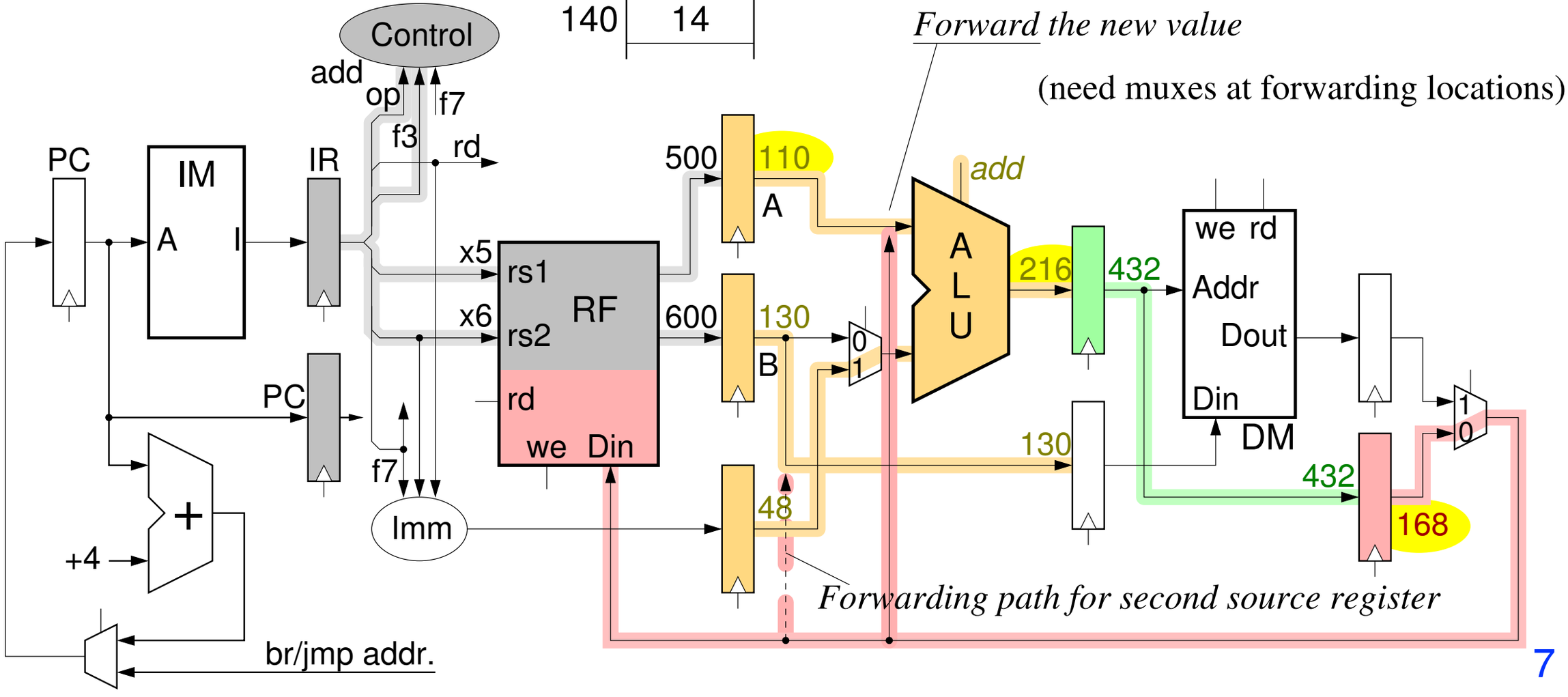
140	14

Erroneous old value coming from stage 2

Correct new value available, in another place

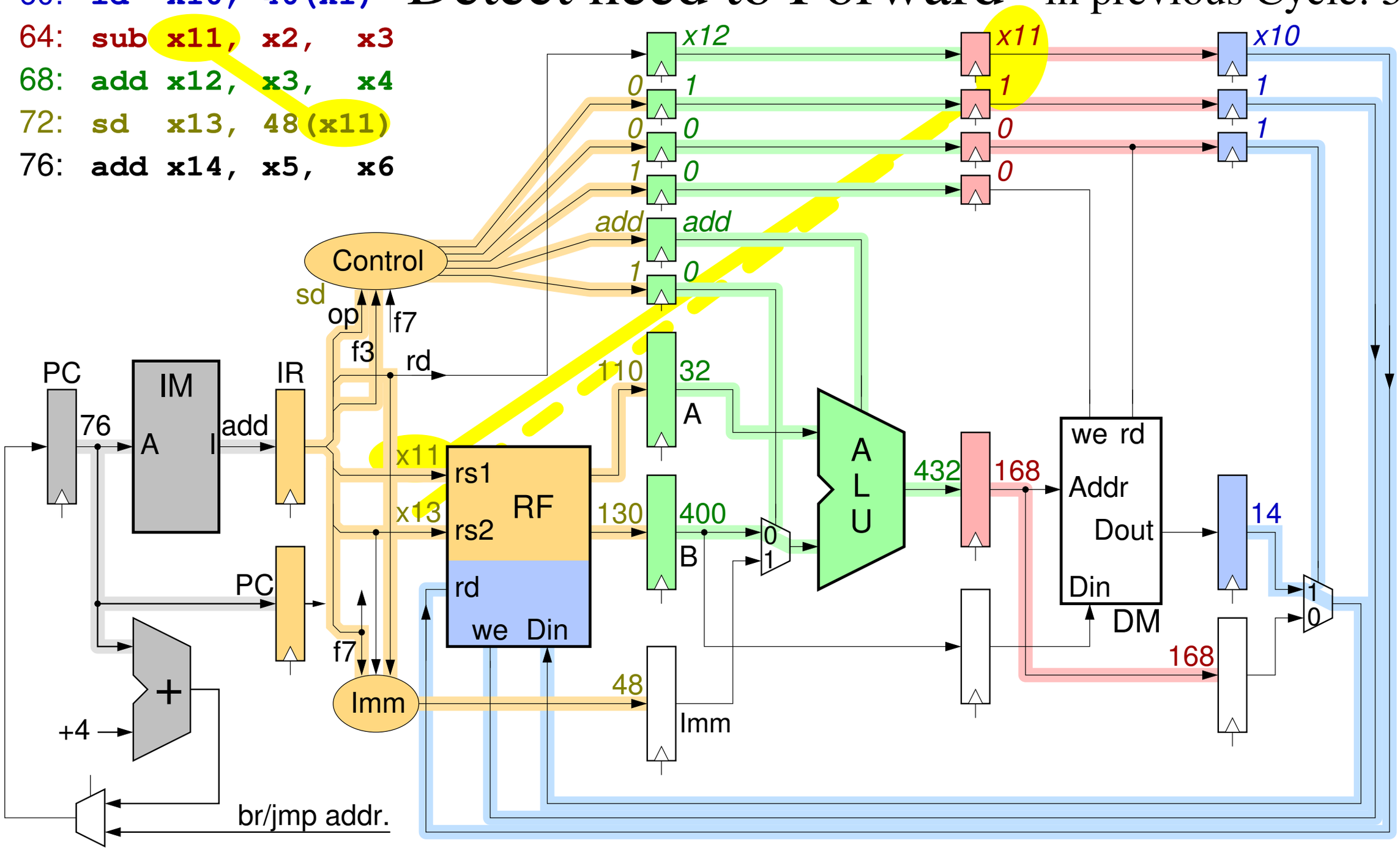
Forward the new value

(need muxes at forwarding locations)



Detect need to Forward – in previous Cycle: 5

```
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
```



Forward to Distance 1

from ALU op. ONLY (Cycle 5)

```
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
x2	200
x3	32
x4	400

x10	~~10~~ 14
x11	~~110~~ 168
x12	120
x13	130

Memory:

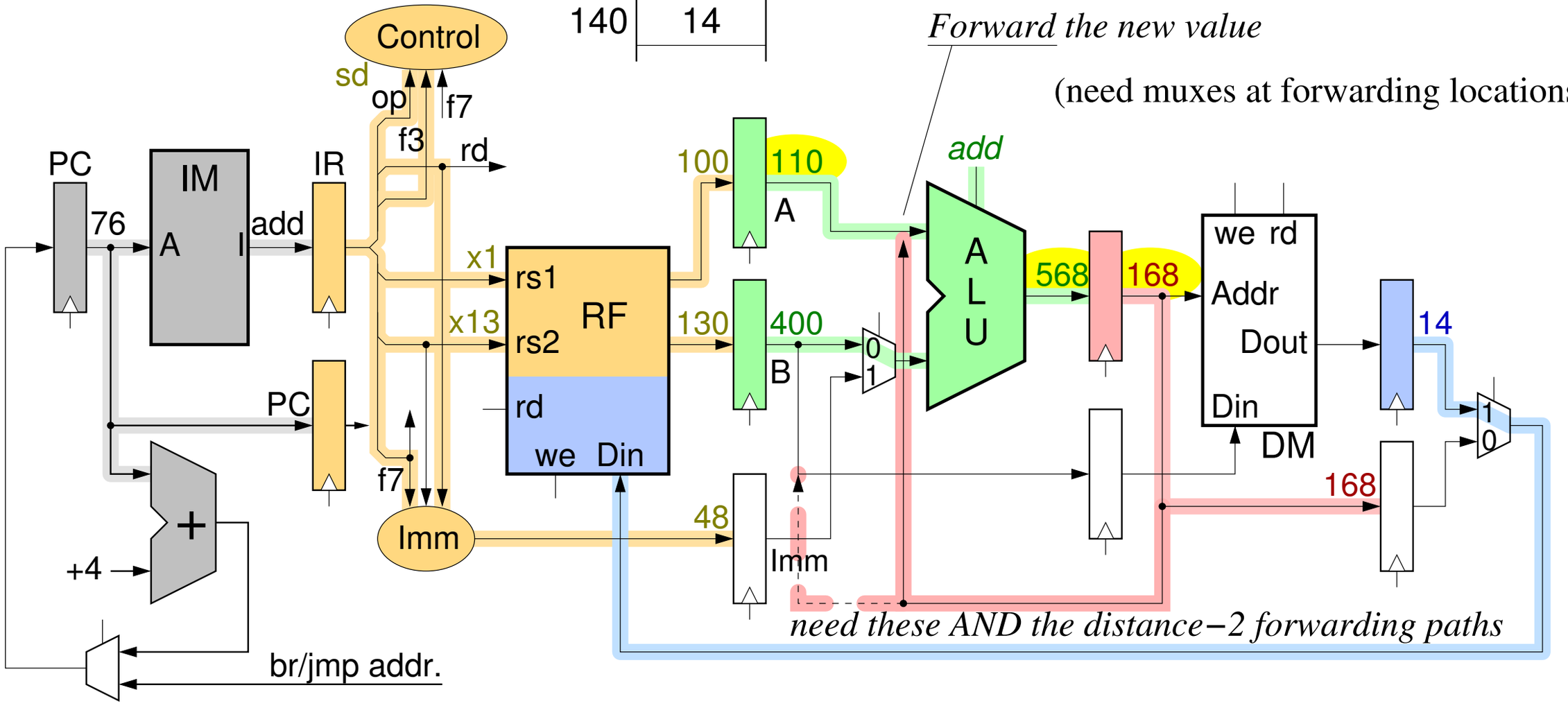
140	14

Erroneous old value coming from stage 2

Correct new value available, in another place

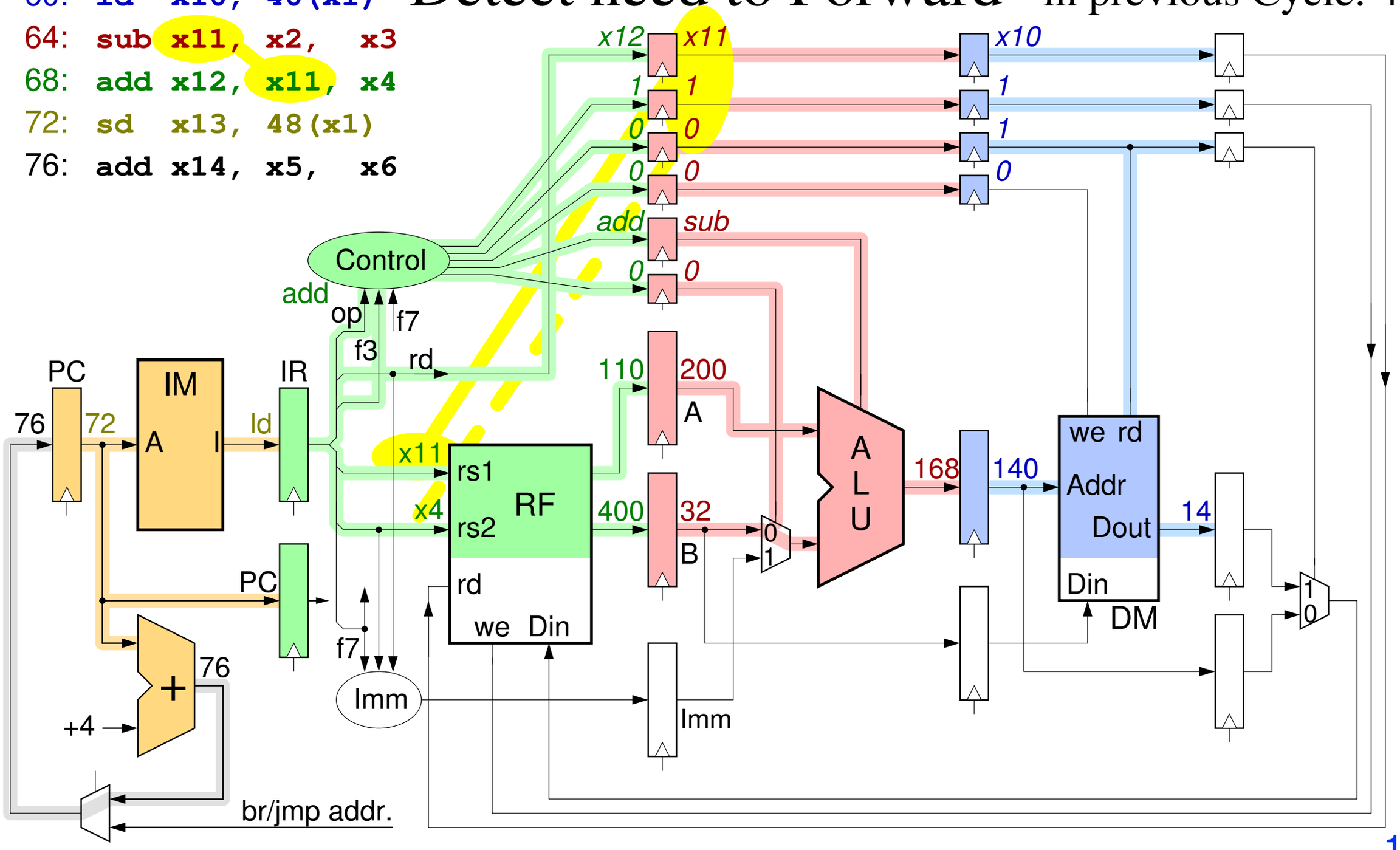
Forward the new value

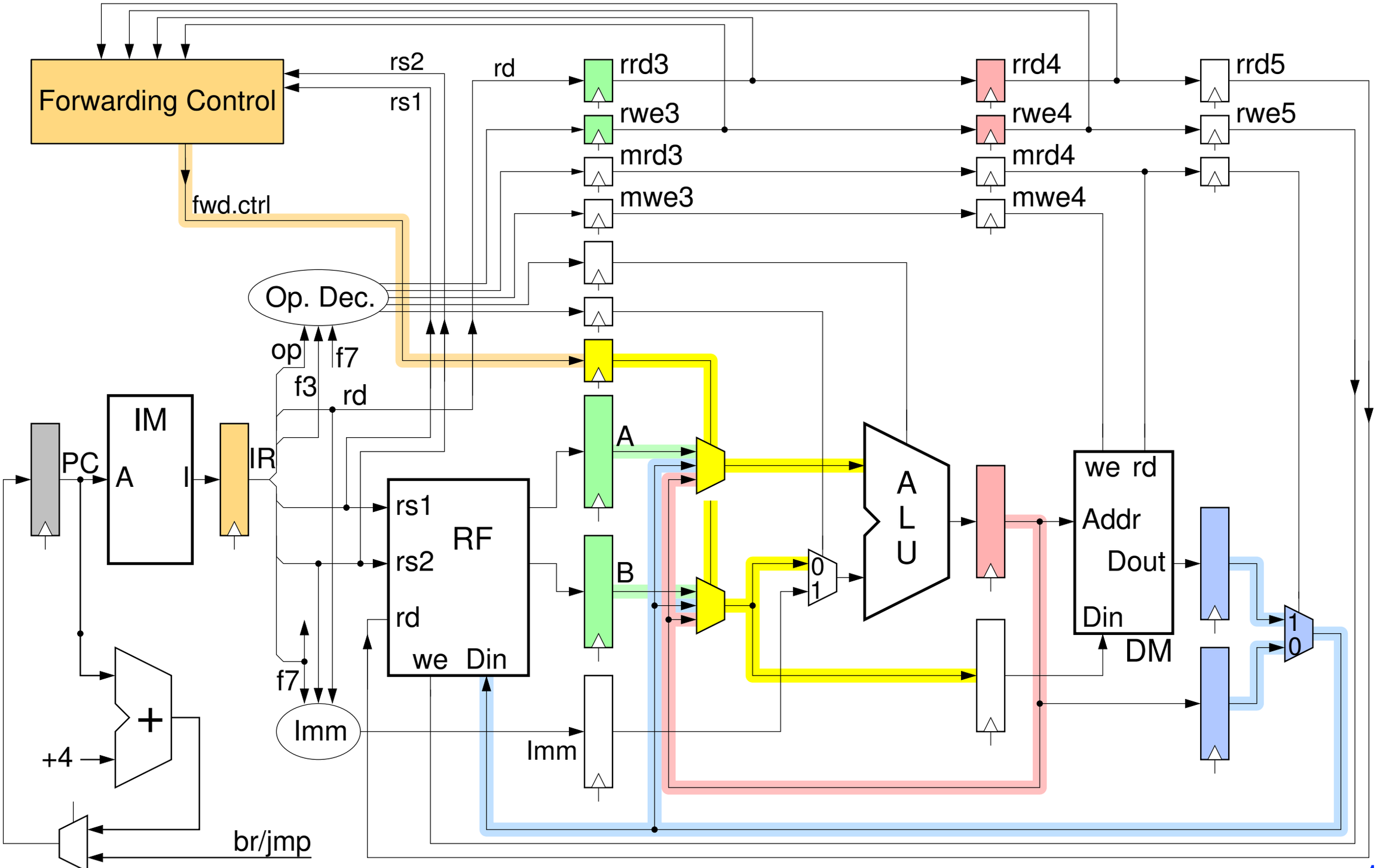
(need muxes at forwarding locations)



Detect need to Forward – in previous Cycle: 4

```
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
```




Forwarding Control
rs2
rs1
rd
fwd.ctrl
rrd3
rwe3
mrd3
mwe3
rrd4
rwe4
mrd4
mwe4
rrd5
rwe5
Op. Dec.
op
f3
f7
rd
IM
PC
A
I
IR
rs1
rs2
rd
RF
we Din
A
B
Imm
f7
Imm
+4
+
br/jmp
0
1
A
L
U
we rd
Addr
Dout
Din
DM
1
0

Forwarding Control Logic

Assuming that the instruction in stage3 is NOT a Load

- Define *Match(rs, rd)* as follows:

  (rs == rd != x0) 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:

```
add t0, s1, s2
add t0, t0, s3
add t0, t0, s4
```

Forward to Distance 2

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

```
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
x2	200
x3	32
x4	400

x10	~~10~~ 14
x11	~~110~~ 168
x12	120
x13	130

Memory:

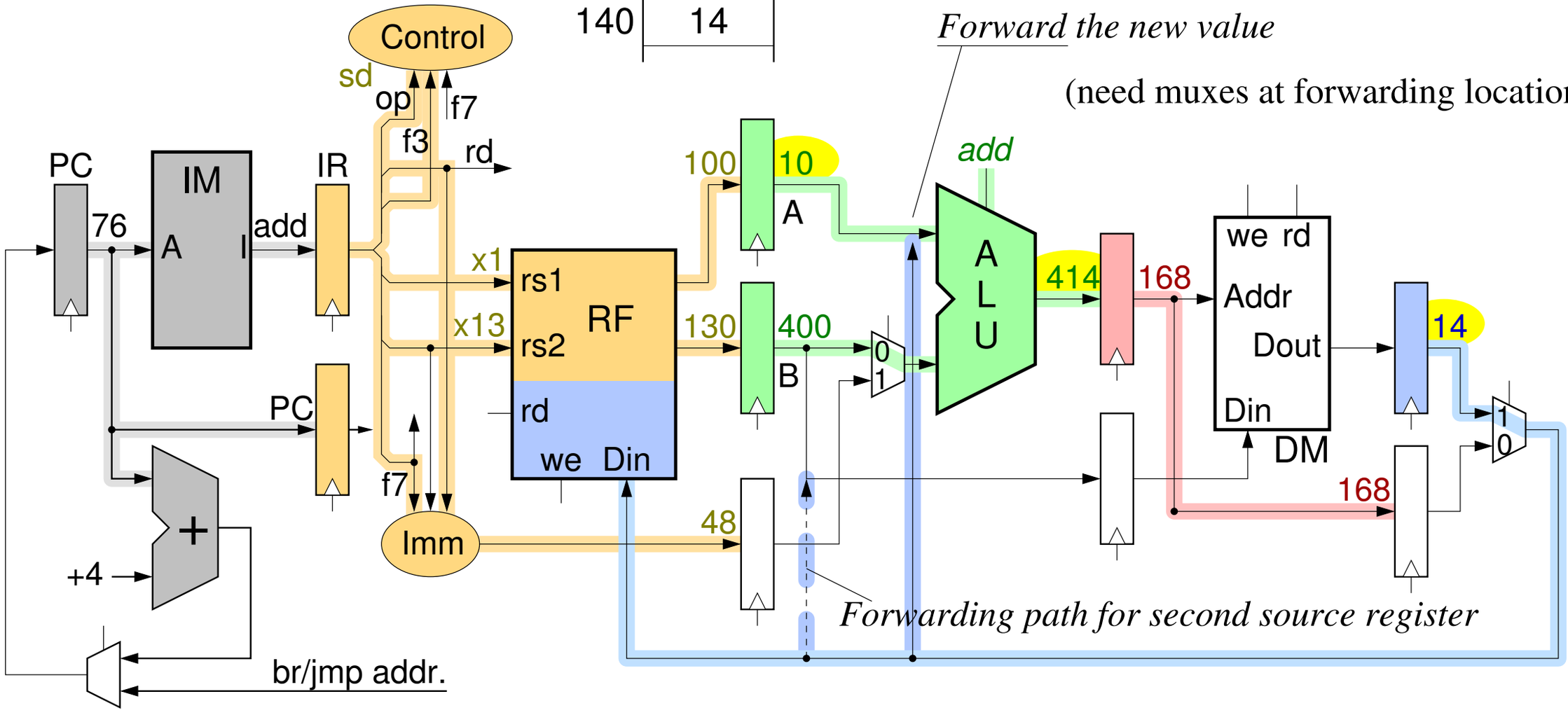
140	14

Erroneous old value coming from stage 2

Correct new value available, in another place

Forward the new value

(need muxes at forwarding locations)



Dist. 1 dep. on LOAD

```
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
x2	200
x3	32
x4	400

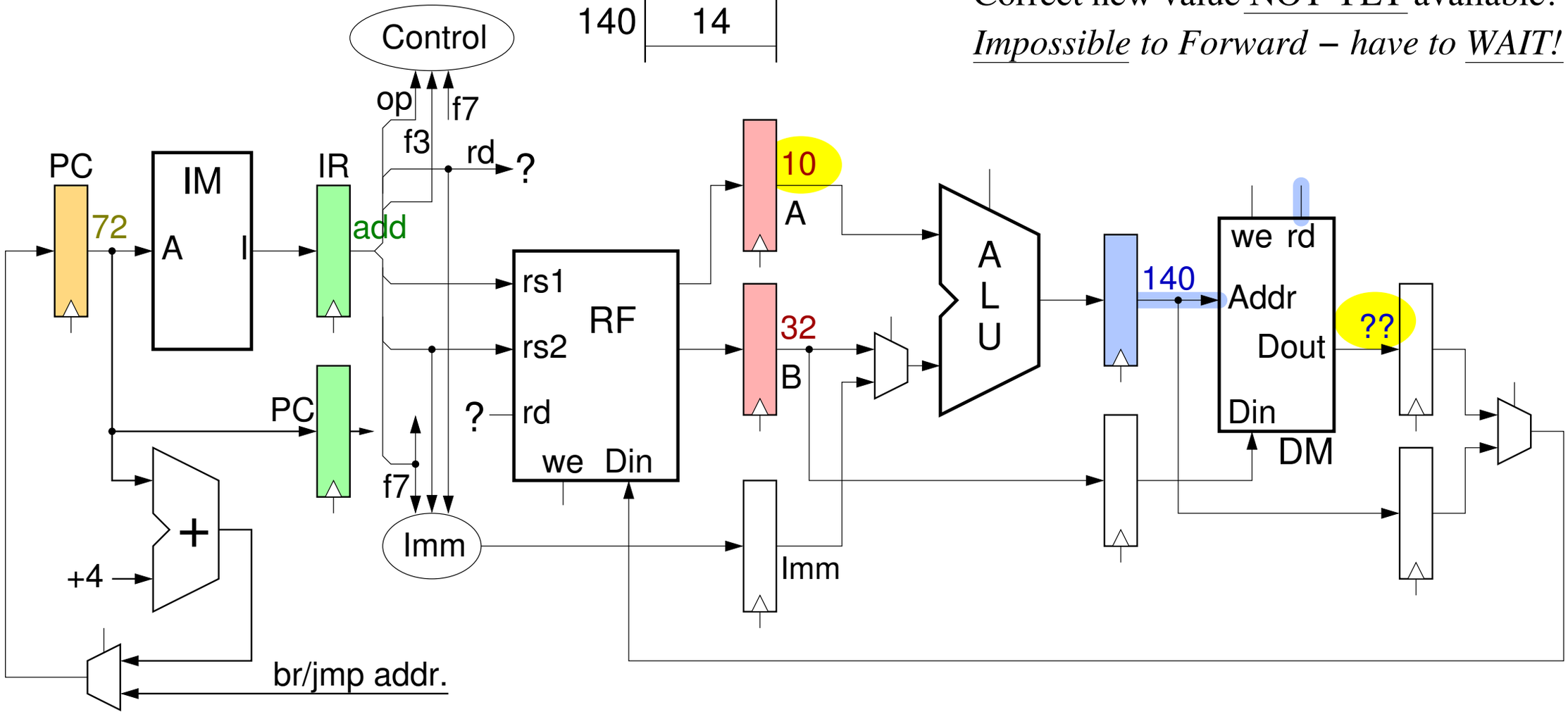
x10	~~10~~ ?
x11	~~110~~ ?
x12	120
x13	130

Memory:

140	14

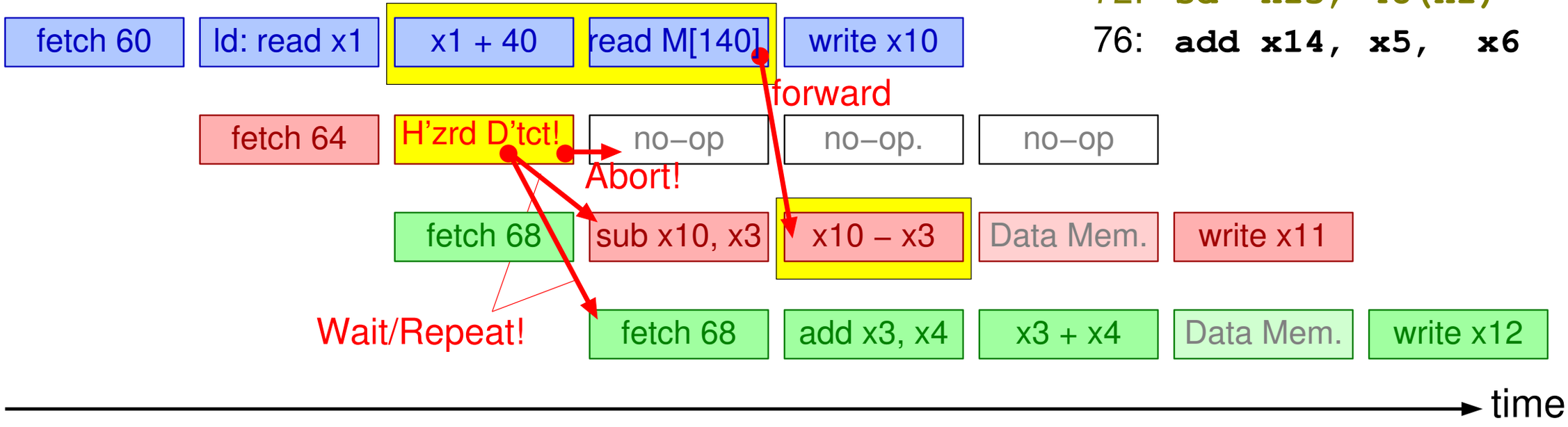
Dependence on immed. preceding Load
(start of Cycle 4)

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

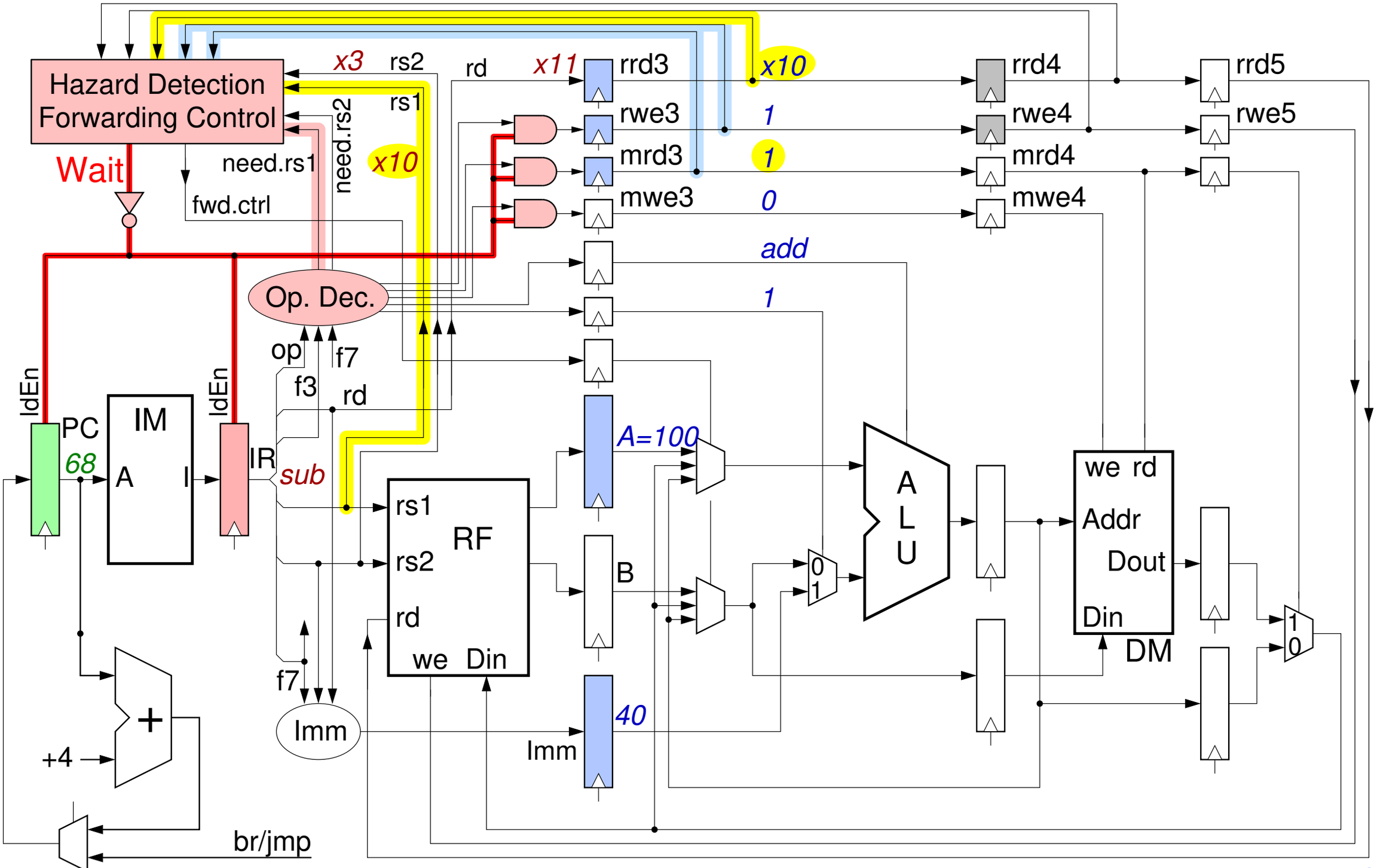


Distance 1 dependence on LOAD: Wait!

```
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
```



- 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
Forwarding Control
Wait
need.rs1
fwd.ctrl
need.rs2
x3
rs2
rs1
x10
rd
x11
rrd3
rwe3
mrd3
mwe3
x10
1
1
0
add
1
rrd4
rwe4
mrd4
mwe4
rrd5
rwe5
Op. Dec.
op
f3
f7
rd
IdEn
PC
68
IM
A
I
IdEn
IR
sub
A=100
B
rs1
rs2
rd
RF
we Din
f7
Imm
Imm
40
+4
+
br/jmp
0
1
A
L
U
we rd
Addr
Dout
Din
DM
1
0

Hazard Detection Logic

- *Wait =*

  (mrd3) AND (Match(rs1, rrd3)) AND (need_rs1)

  OR

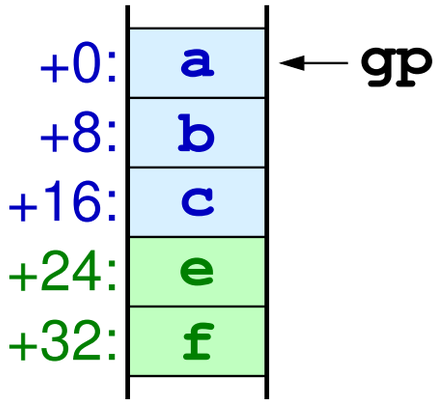
  (mrd3) AND (Match(rs2, rrd3)) AND (need_rs2)

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

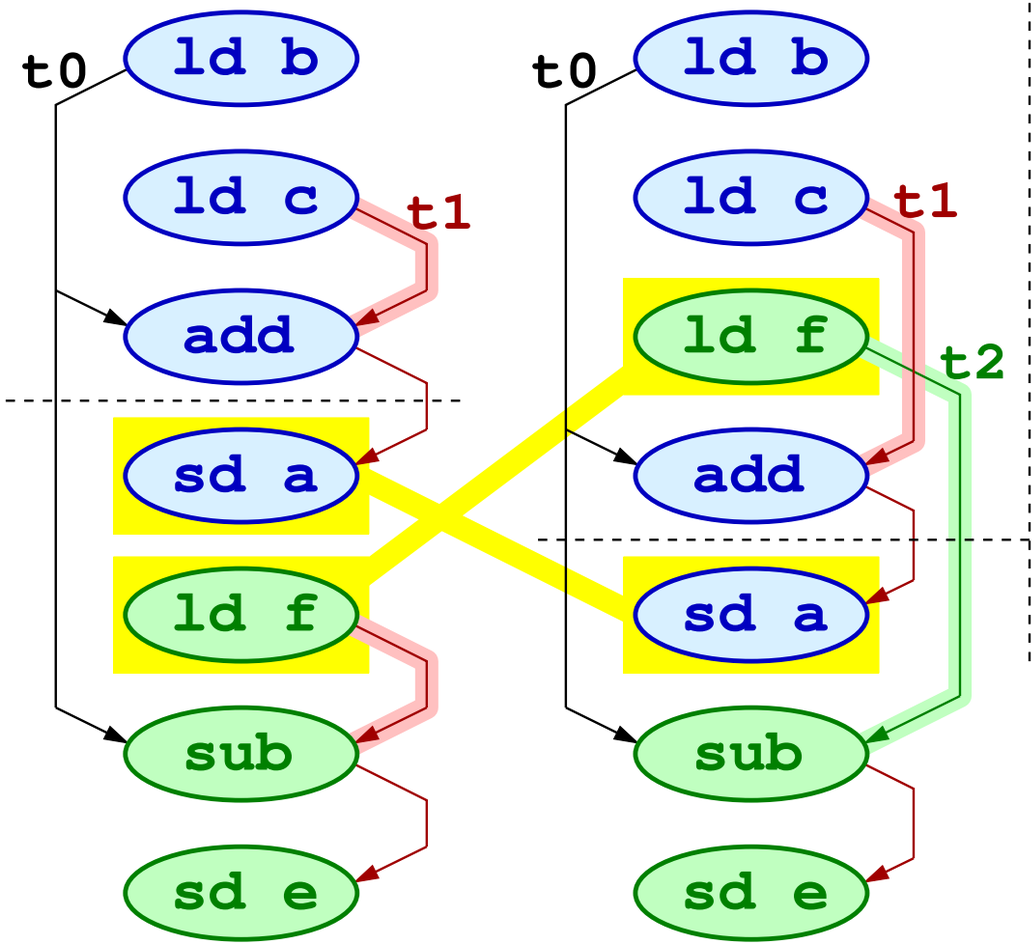
Instruction Scheduling

```
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



This is 'Static' Scheduling, at Compile Time

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

What if the program is?:
RAW dependence?

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

- Does the compiler know for sure if i!=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)