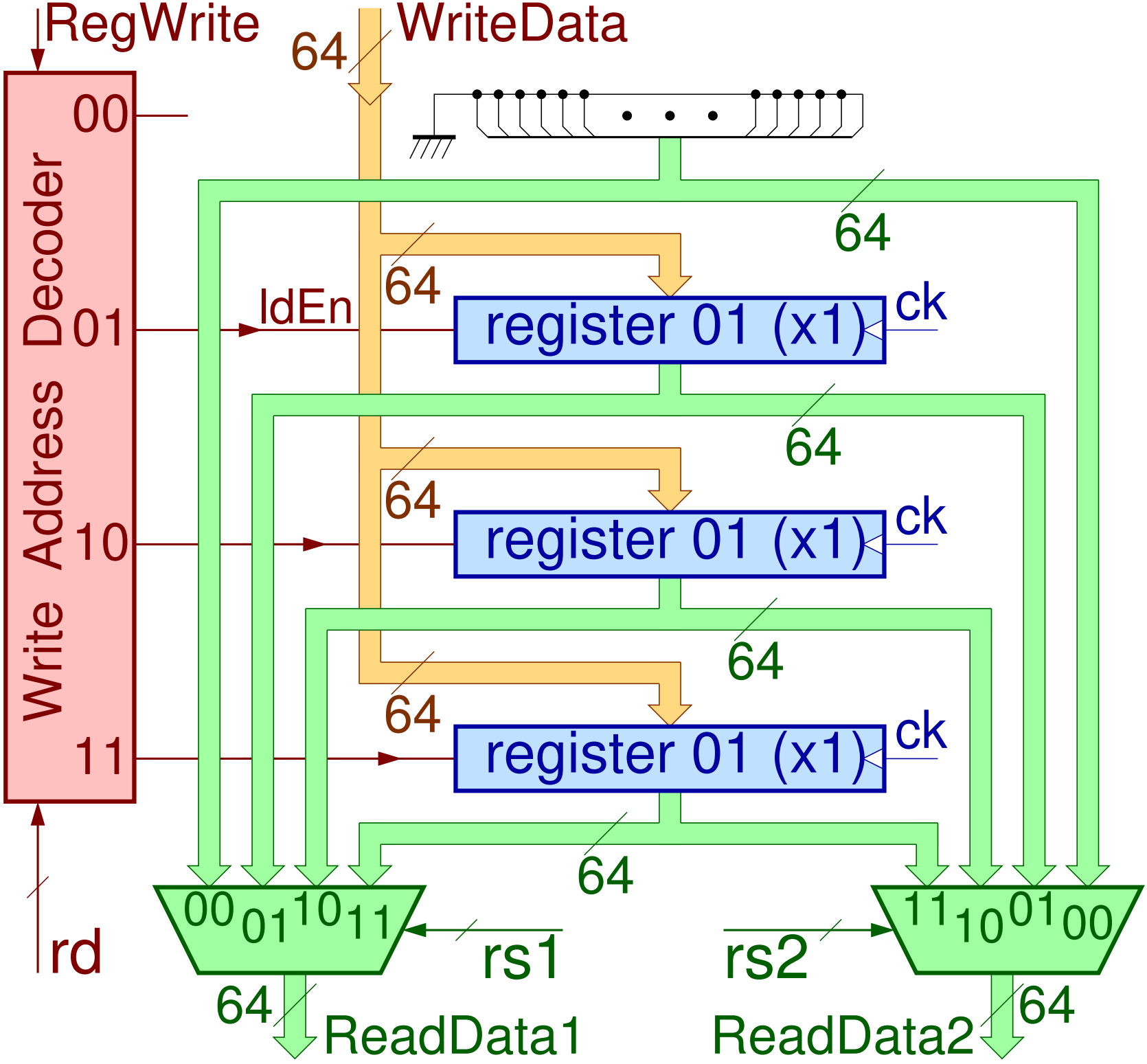
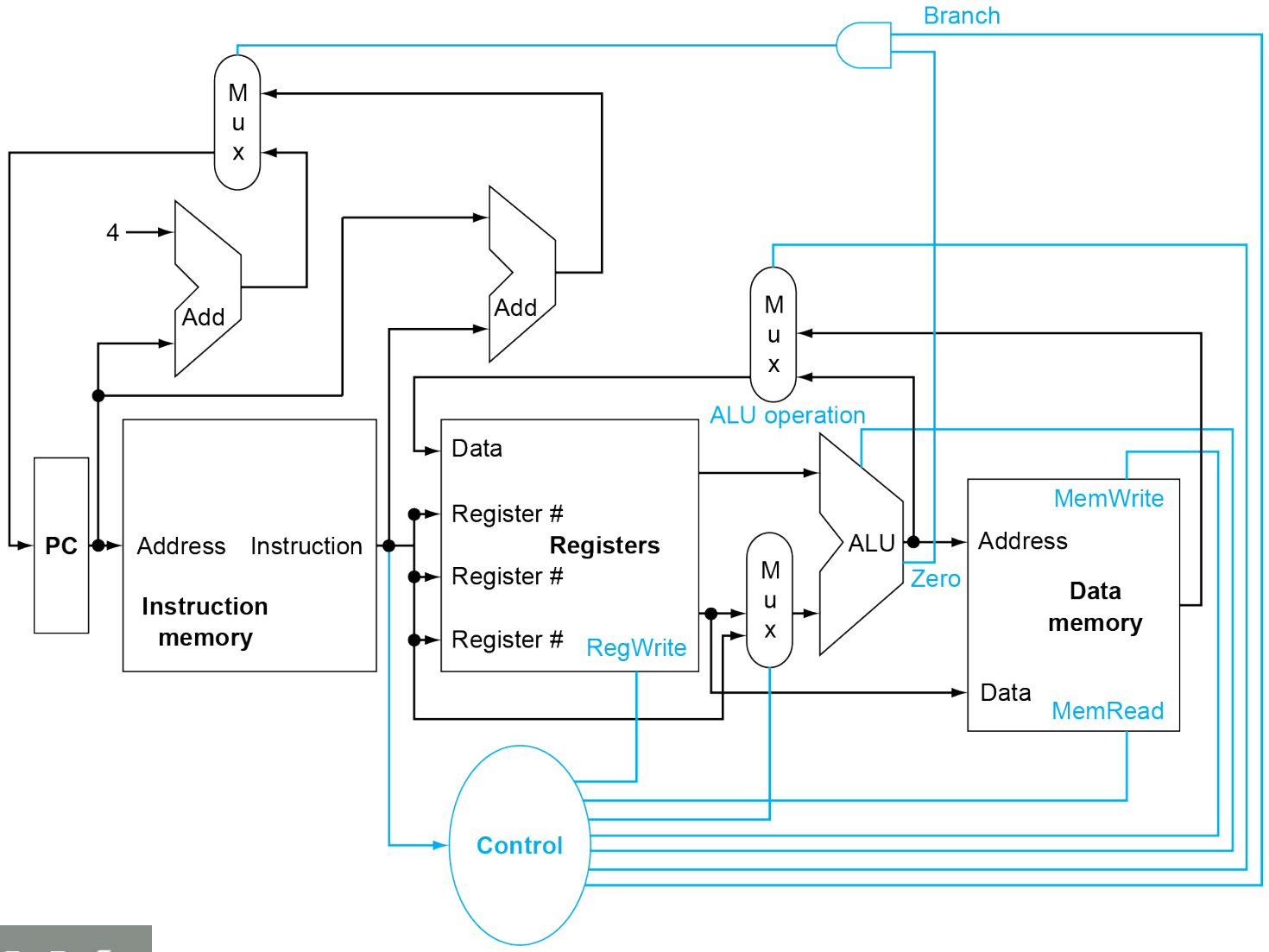
Chapter 4

The Processor


RegWrite
64
WriteData
Write Address Decoder
00
01
10
11
ldEn
register 01 (x1)
ck
register 01 (x1)
ck
register 01 (x1)
ck
64
rd
00 01 10 11
rs1
rs2
11 10 01 00
ReadData1
ReadData2

Control & Datapath

Single (long) cycle per instruction



Pipelined Datapath & Control Operation without data or control dependencies, yet

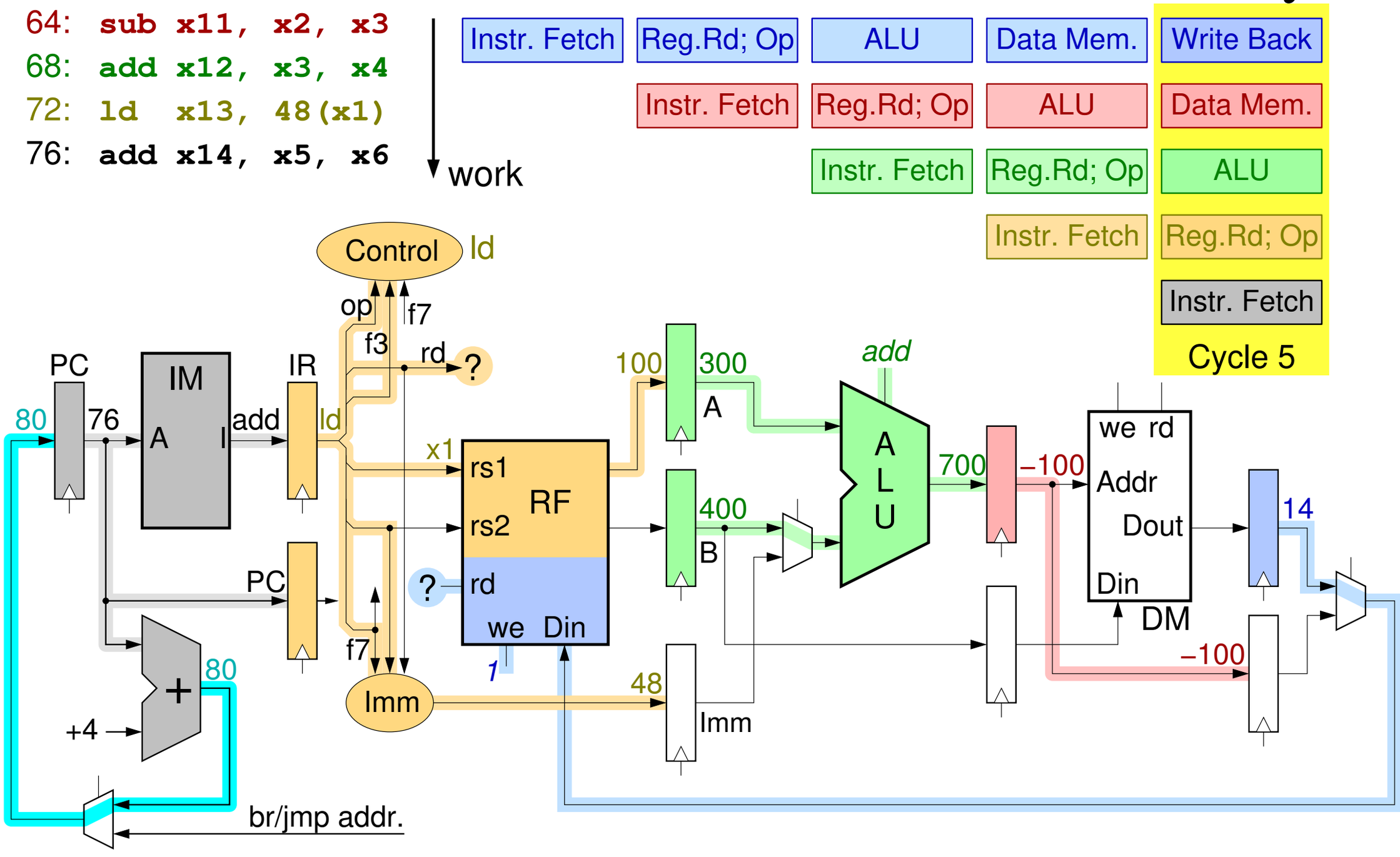
University of Crete
Dept. of Computer Science
CS–225 (HY–225)
Computer ORganization
Spring 2020 semester

Slides for §9.3 – 9.5

§9.3 Pipelined Datapath Operation
§9.4 Control for the Pipelined Datapath
§9.5 Graphical representation: time–work

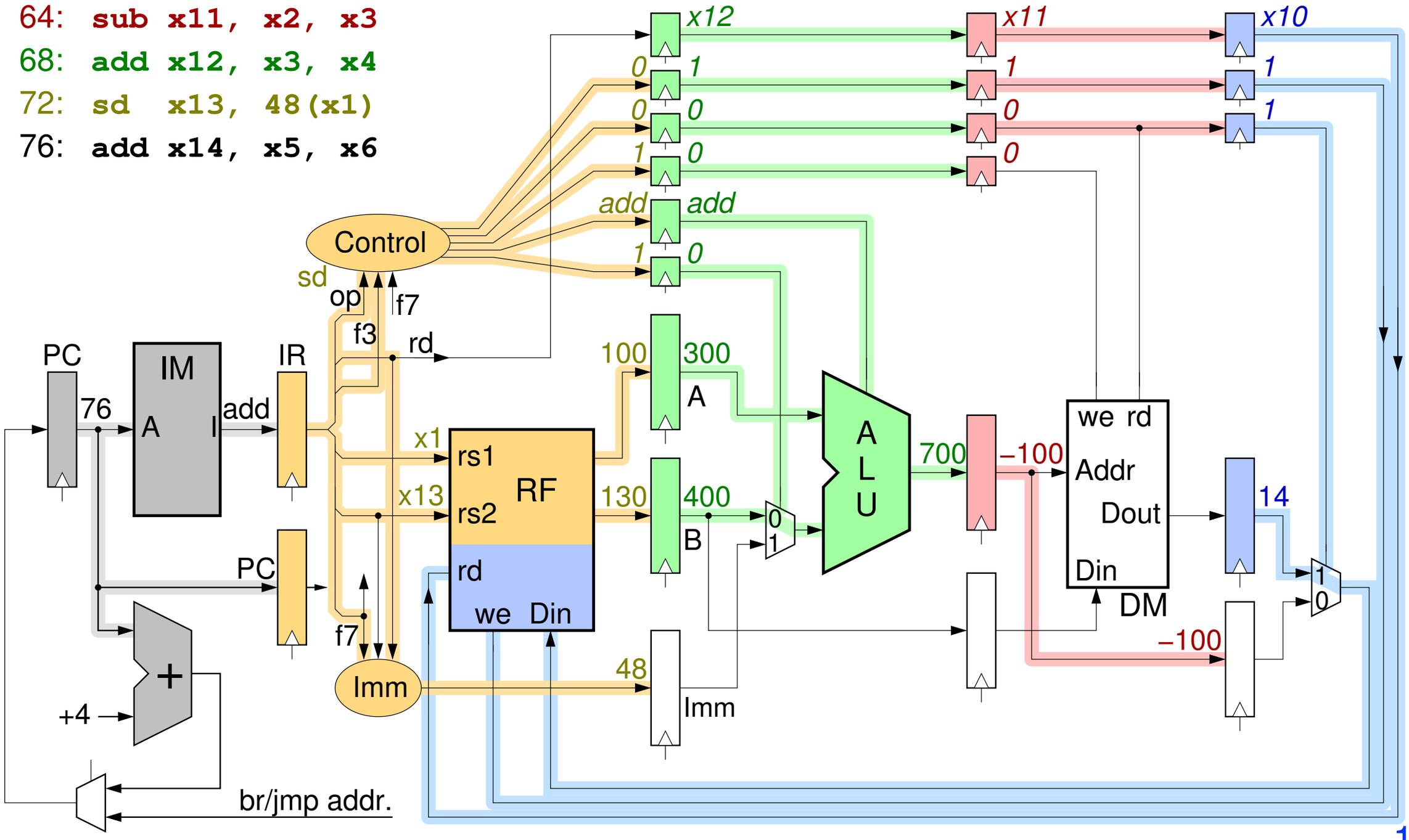
Cycle 5

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

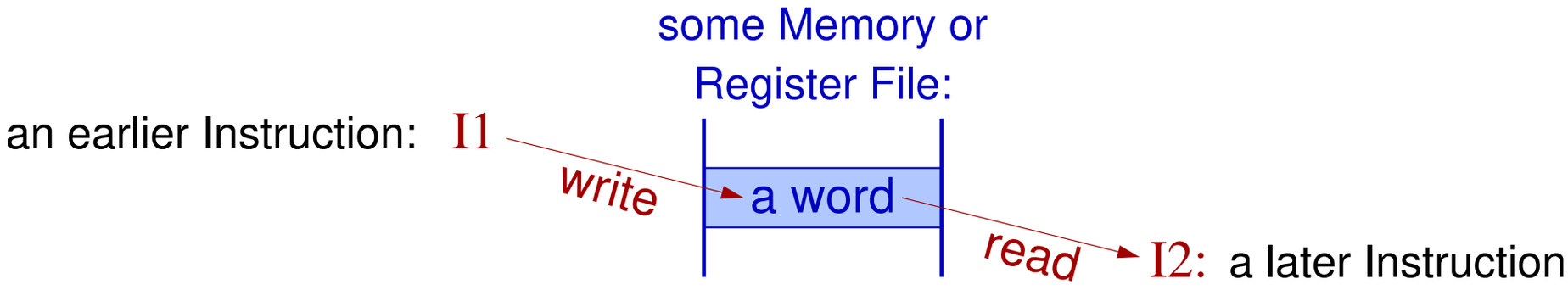


Cycle 5

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



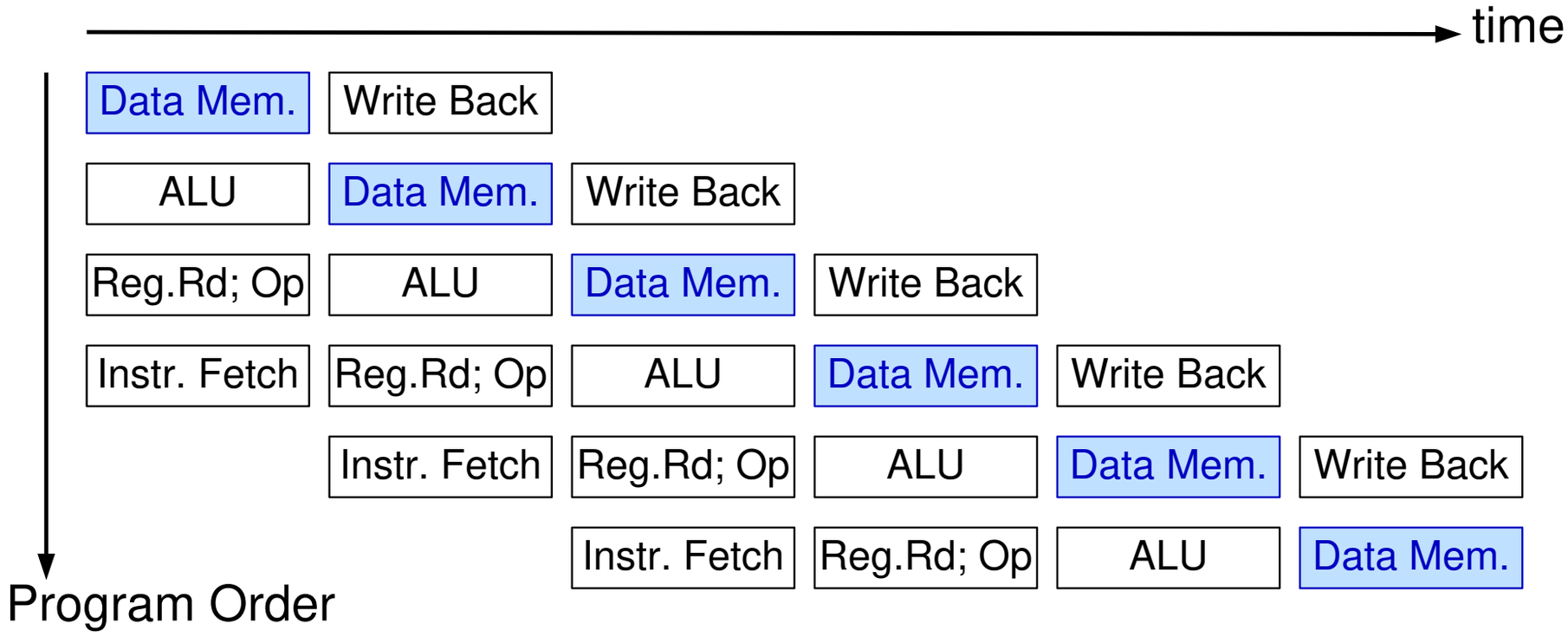
Data Dependences (Hazards) in Pipelines



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

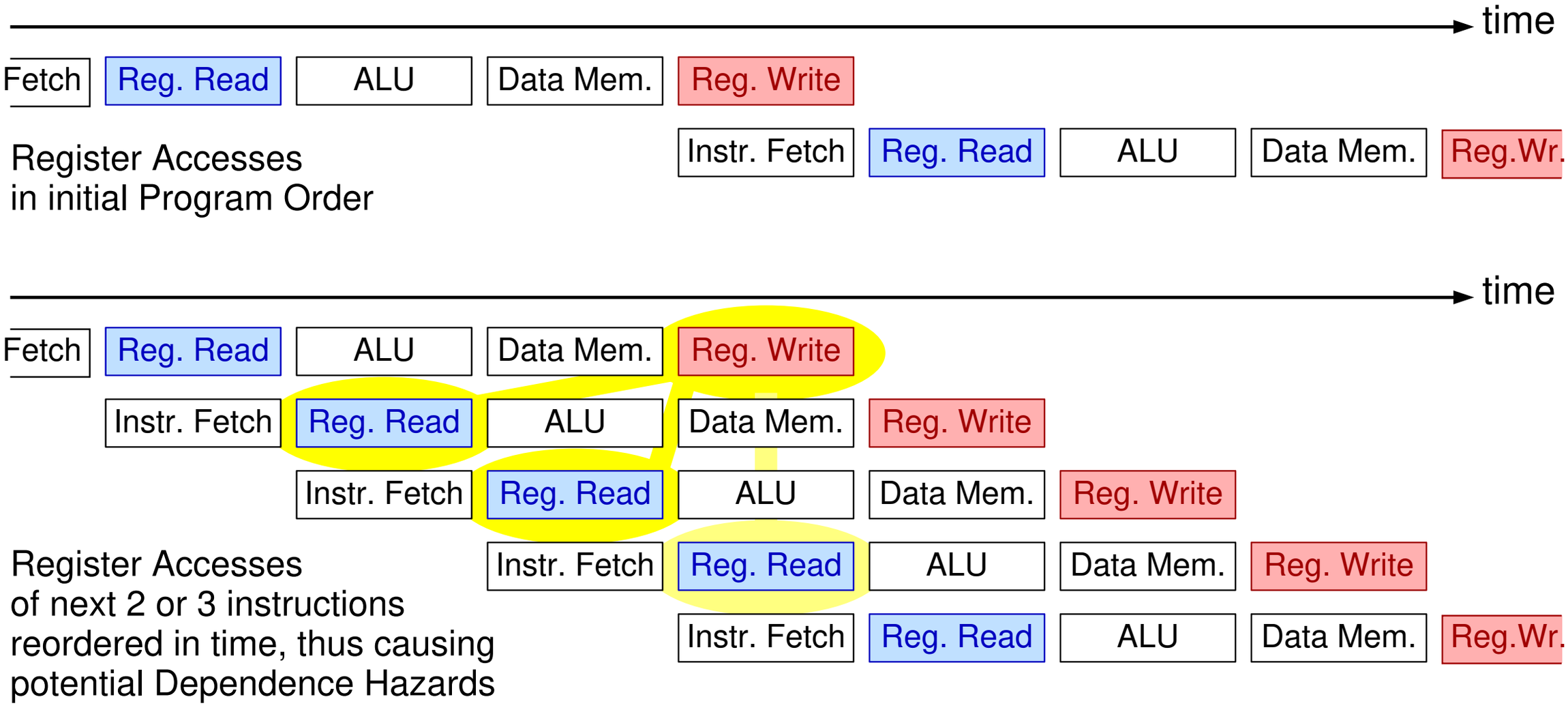
- RAW (Read after Write) – true dependence, as above
- RAR (Read after Read) – not a dependence, can freely reorder the reads
- WAR (Write after Read) – "antidependence": if you want to do the write (I2) early, just keep a copy of the old data and have I1 read that copy
- WAW (Write after Write) – if 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



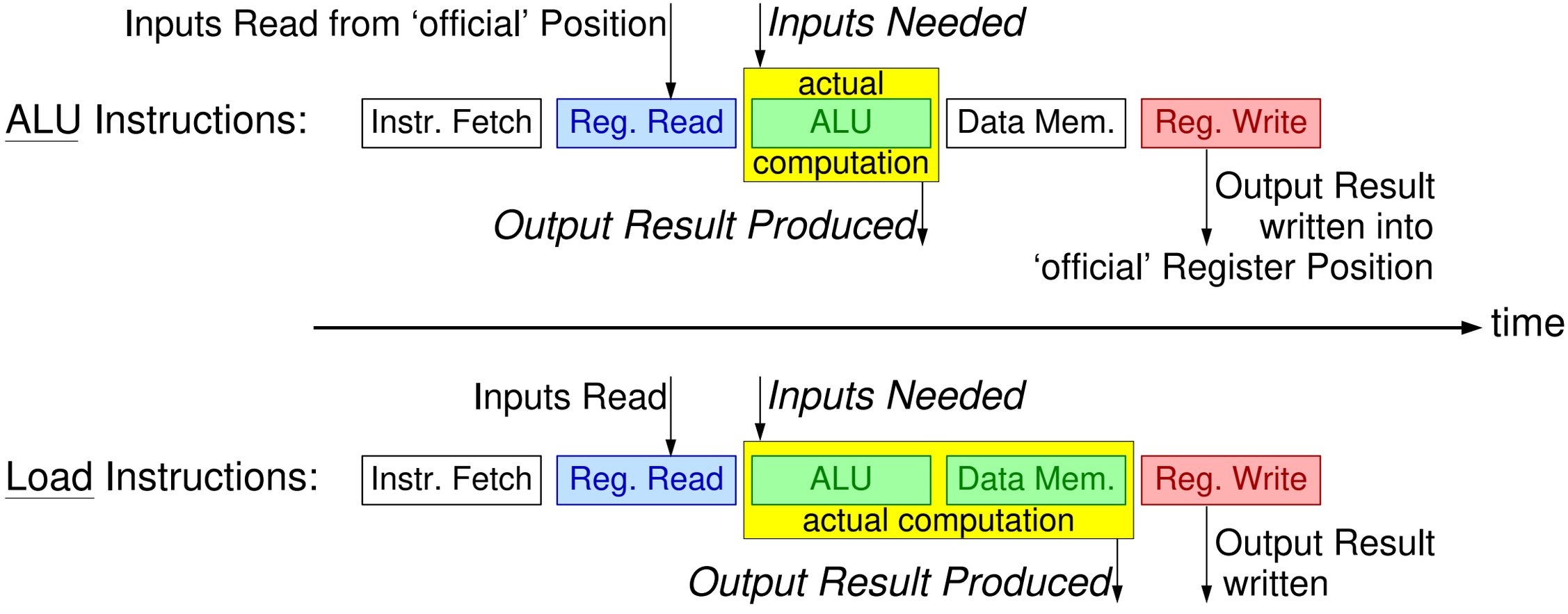
- Data Memory accesses are performed ‘in–order’ in our simple pipeline,
  i.e. are not reordered relative to what the program specifies,
  thus, no dependences of memory word accesses are ever violated

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...

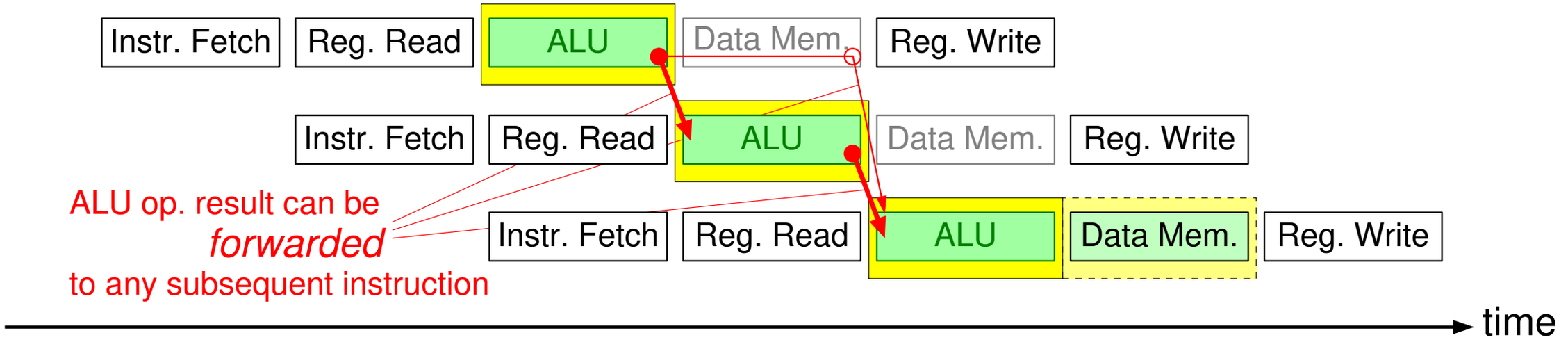
Actual Need–Produce Time vs. from/in–Register Time



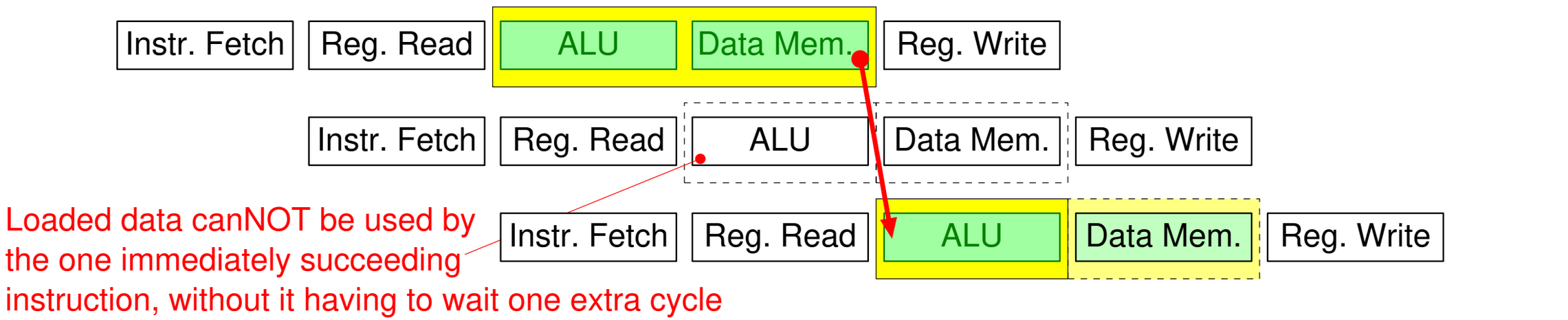
- All we care about is actual Results 'Forwarded' from Producer to Consumer instruction
- We can 'Bypass' the 'official' loop through the Register File for immediate–use Results

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

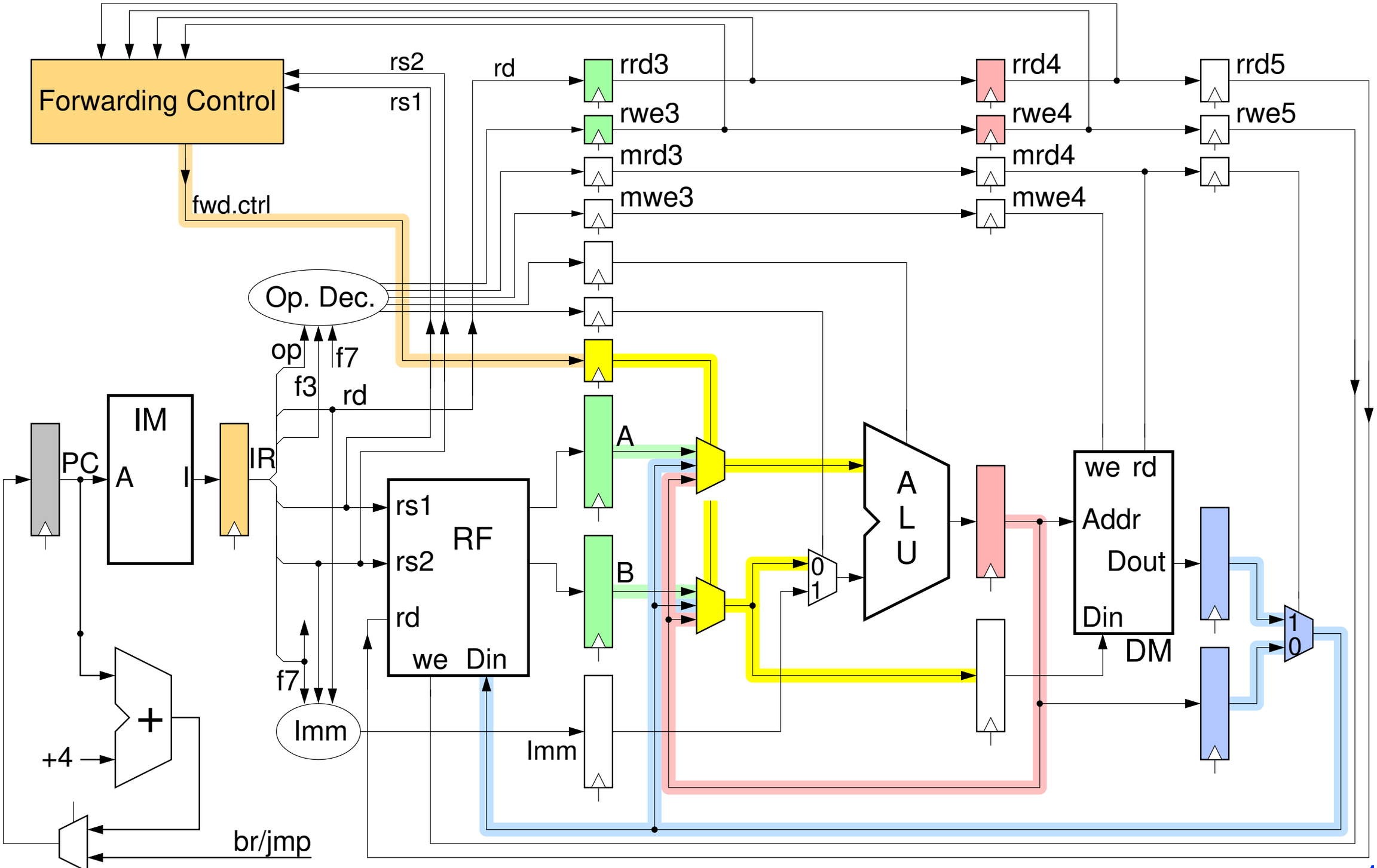
from ALU Instructions:



from Load Instruction:

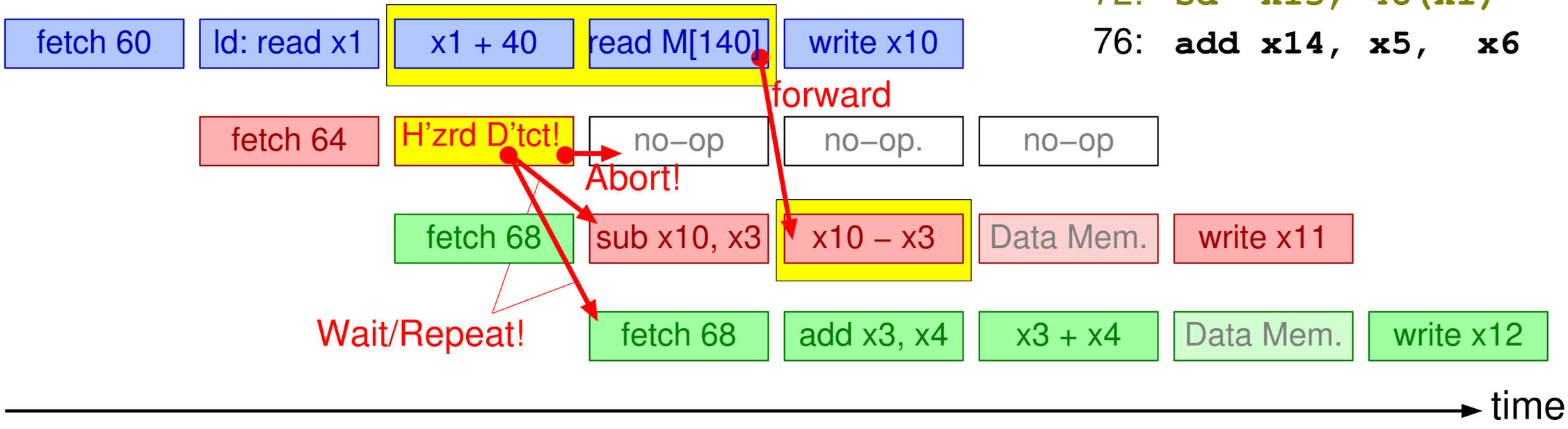


- ALU instructions never stall the pipeline, but Load instructions will do so when immediately followed by a dependent instruction

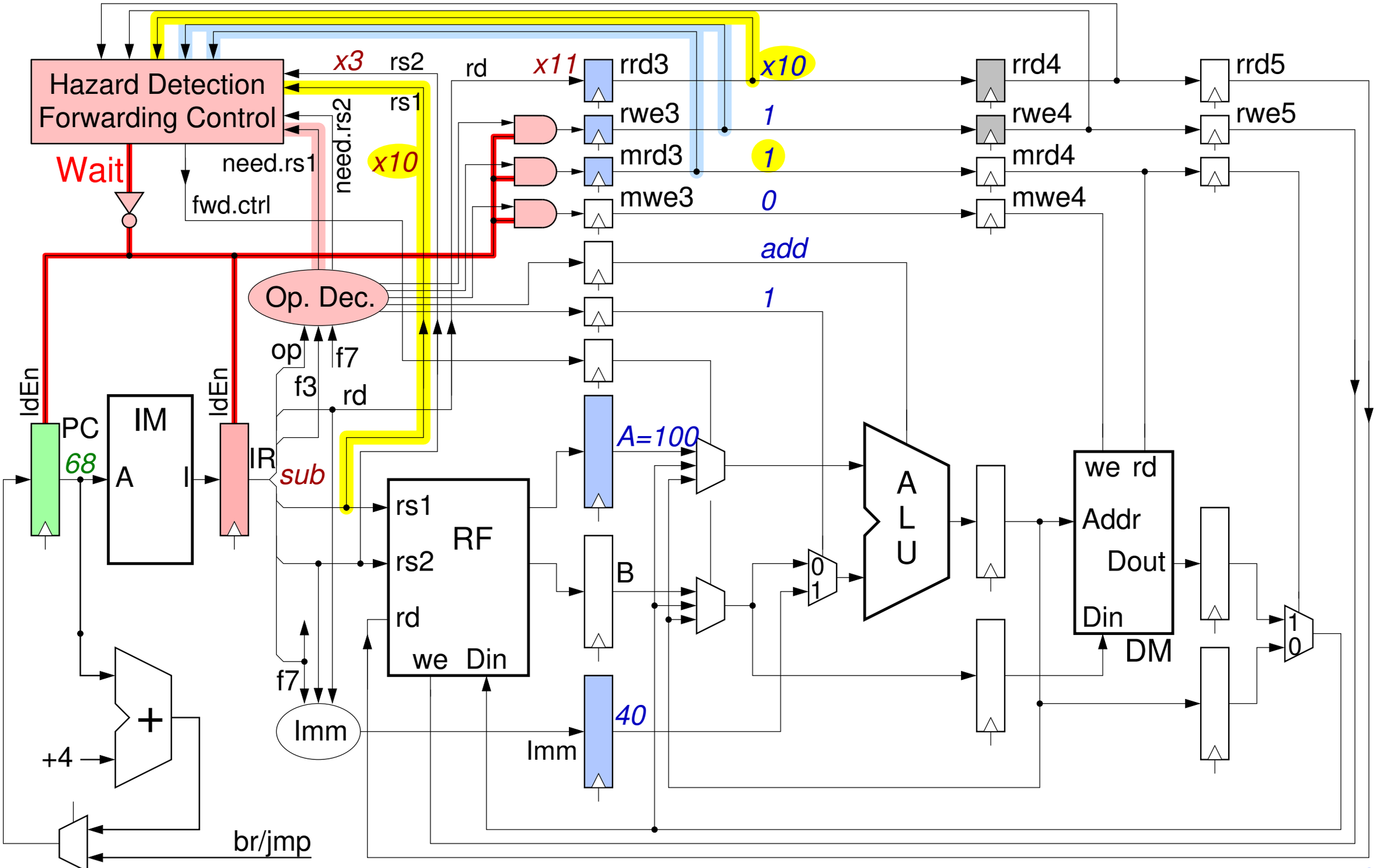

Forwarding Control
rs2
rs1
rd
fwd.ctrl
rrd3
rwe3
mrd3
mwe3
rrd4
rwe4
mrd4
mwe4
rrd5
rwe5
Op. Dec.
op
f3
f7
rd
IM
PC
A
I
IR
rs1
rs2
rd
RF
we Din
A
B
Imm
f7
Imm
+4
+
br/jmp
0
1
A
L
U
we rd
Addr
Dout
Din
DM
1
0

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
```

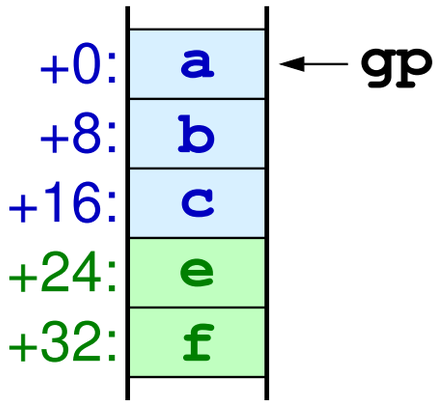


- The instruction immediately after a Load wants to use the load'ed data
- Impossible without losing one cycle:
  force this instruction to wait (repeat itself on the next cycle)
- Simple, in–order Pipeline: the next instruction has to wait too!


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
40
Imm
Imm
f7
rs1
rs2
rd
RF
we Din
0
1
A
L
U
we rd
Addr
Dout
Din
DM
1
0
+
+4
br/jmp

Instruction Scheduling

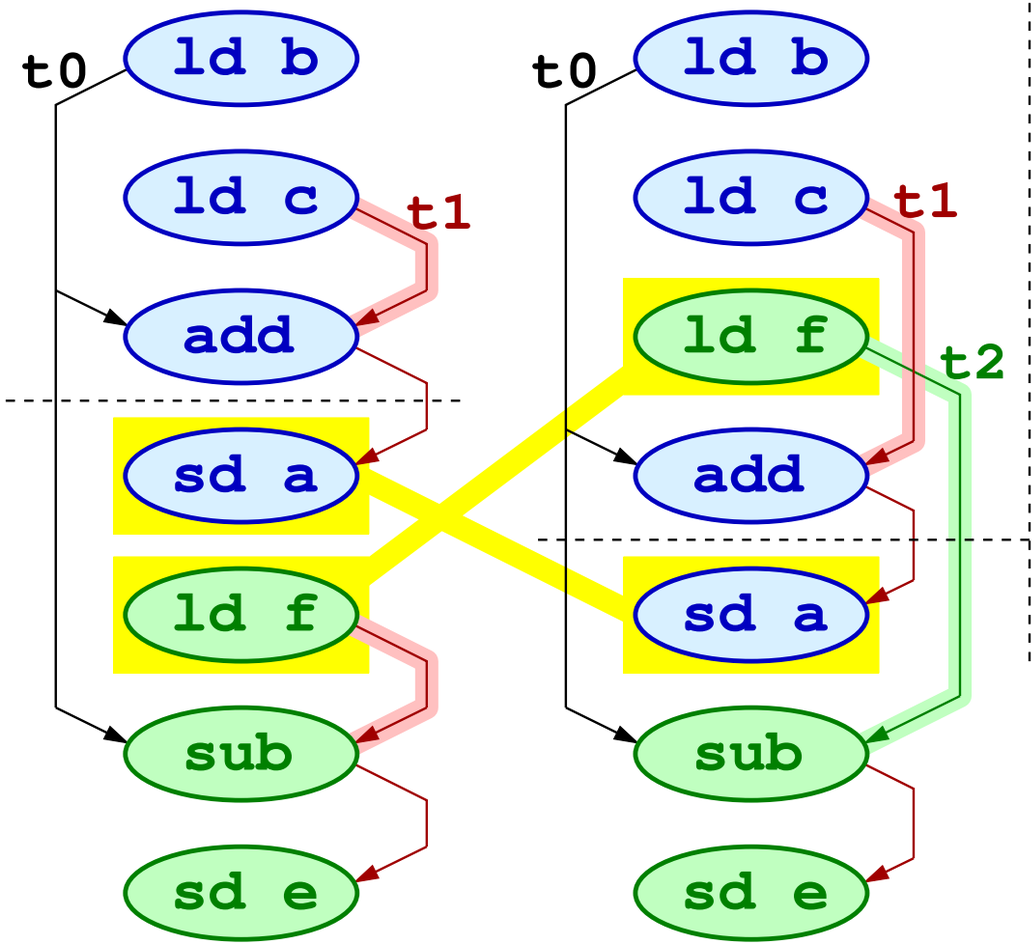
```
a = b + c;
e = b - f;
```




```
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

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

```
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)

Control Dependences (branch/jump) in Pipelines

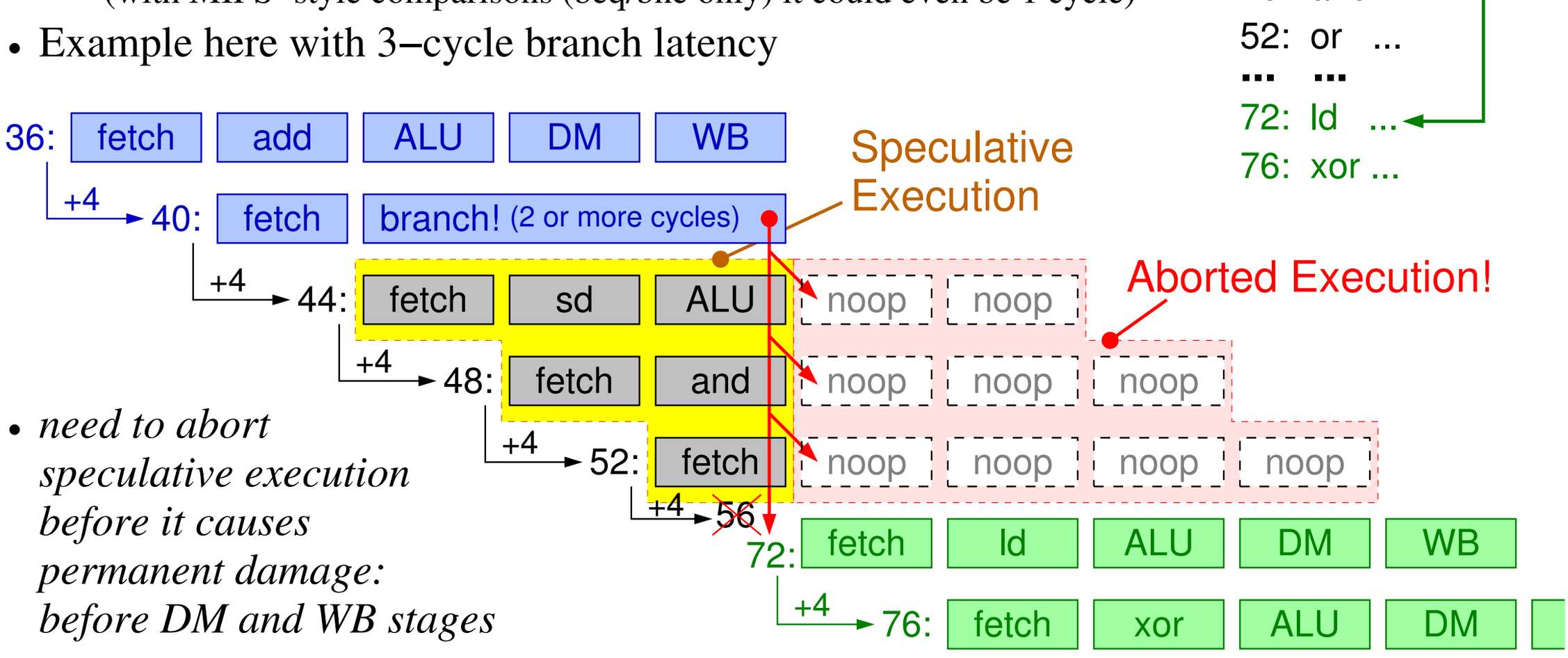
- 'Data Dependence' = next instruction uses data (register/memory) from previous
- 'Control Dependence' = which is the next instruction depends on the previous
- Control Dependences arise from 'Control Transfer Instructions (CTI)'
- Control Transfer Instructions (CTI) are: Jump and Branch Instructions
- 'Jumps' are Unconditional CTI's: they always transfer control
- 'Branches' are Conditional CTI's: whether or not they transfer control depends on the result of a data comparison that they have to perform

Statistics (rough numbers, in a majority of programs, but NOT always so):

- Branches are about 15–16% of all ('dynamically') executed instructions in a program
  - about 2/3 of executed branches are 'taken' (successful) = ~10% of all instr.
  - about 1/3 of executed branches are not taken (unsuccessful) = ~5% of all instr.
  - most backwards branches appear in loops, and they are about 90% taken
- Jumps are about 4–5% of all executed instructions in a program
  - procedure calls are about 1%, and returns another ~1%, of all executed instr.

Branch Taken example

- In modern processors, branch latency is quite long
- In our simple pipeline, branch latency is 2 cycles (read registers; compare)
  (with MIPS–style comparisons (beq/bne only) it could even be 1 cycle)
- Example here with 3–cycle branch latency



- *need to abort speculative execution before it causes permanent damage: before DM and WB stages*

- In this example, each taken branch causes the loss of 3 extra clock cycles
- About 2/3 of all executed branches are taken, so this is a heavy loss

Branch Target Buffer (BTB)

- A small table – a cache, like a hash table – containing pairs of (instruction) addresses for which there is statistical evidence that their next–PC is something other than PC+4

```
36: add ...
40: beq ..., goto72
44: sd  ...
48: and ...
52: or  ...
... ...
72: ld  ...
76: xor ...
```

PC of a jump or branch–likely instruction;

Target PC to which this instruction usually went, in the past.

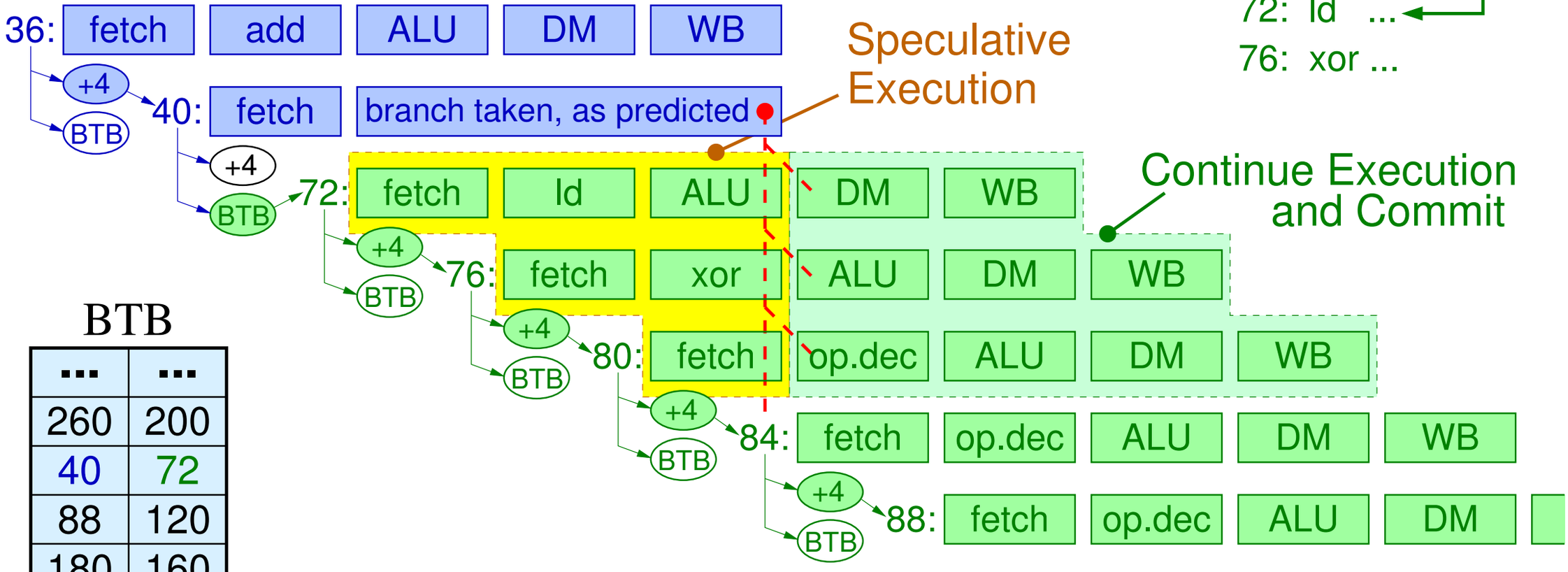
...	...
260	200
40	72
88	120
180	160
...	...

- A 'best approximation' – not necessarily correct information
- Branches that are believed not–taken are NOT entered into the BTB
- Like IM –the Instruction Cache– this will oftentimes 'overflow':
  old pairs are removed to make room for more recent ones
- May be complemented with a small hardware stack:
  – on every call (jal ra,...), push the return address;
  – on every return (jr ra), pop an address and predict jumpin to that one

- *In parallel with each Fetch, search the fetched instruction's PC value in the BTB*

When the BTB prediction is Correct

- When a matching BTB entry is found, use its Prediction;
  else, fetch from PC+4



BTB

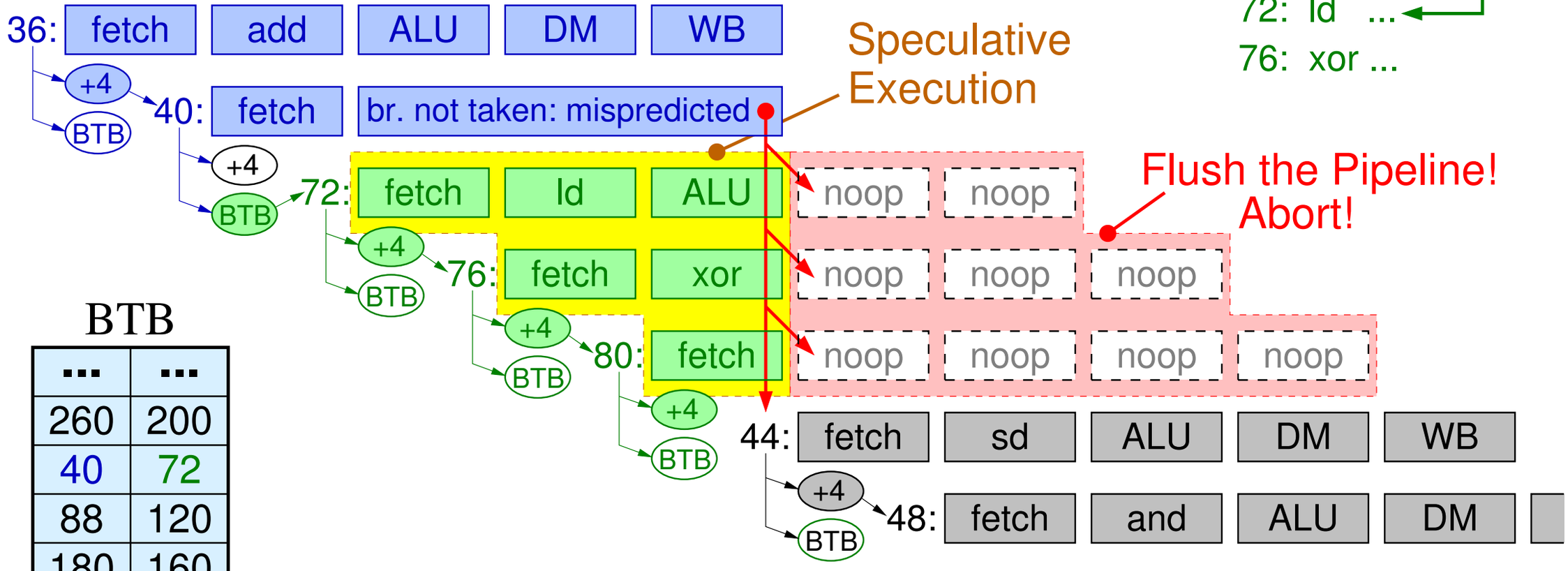
...	...
260	200
40	72
88	120
180	160
...	...

- *When Prediction is Correct, NO extra clock cycles are lost!*

When the BTB prediction is Wrong

- Prediction says: After fetching from 40, fetch from 72
- But this time, the branch ends up going the other way: to 44

```
36: add ...
40: beq ..., goto72
44: sd  ...
48: and ...
52: or  ...
... ...
72: ld  ...
76: xor ...
```



BTB

...	...
260	200
40	72
88	120
180	160
...	...

- When Mispredicted, branches cost 3 extra clock cycles in this pipeline

Relative Performance

- Define Performance = 1/Execution Time
- "X is n time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time_B / Execution Time_A = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Performance Summary

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

  - Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

Average Access Time - Caches

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$