Pipeline Hazards

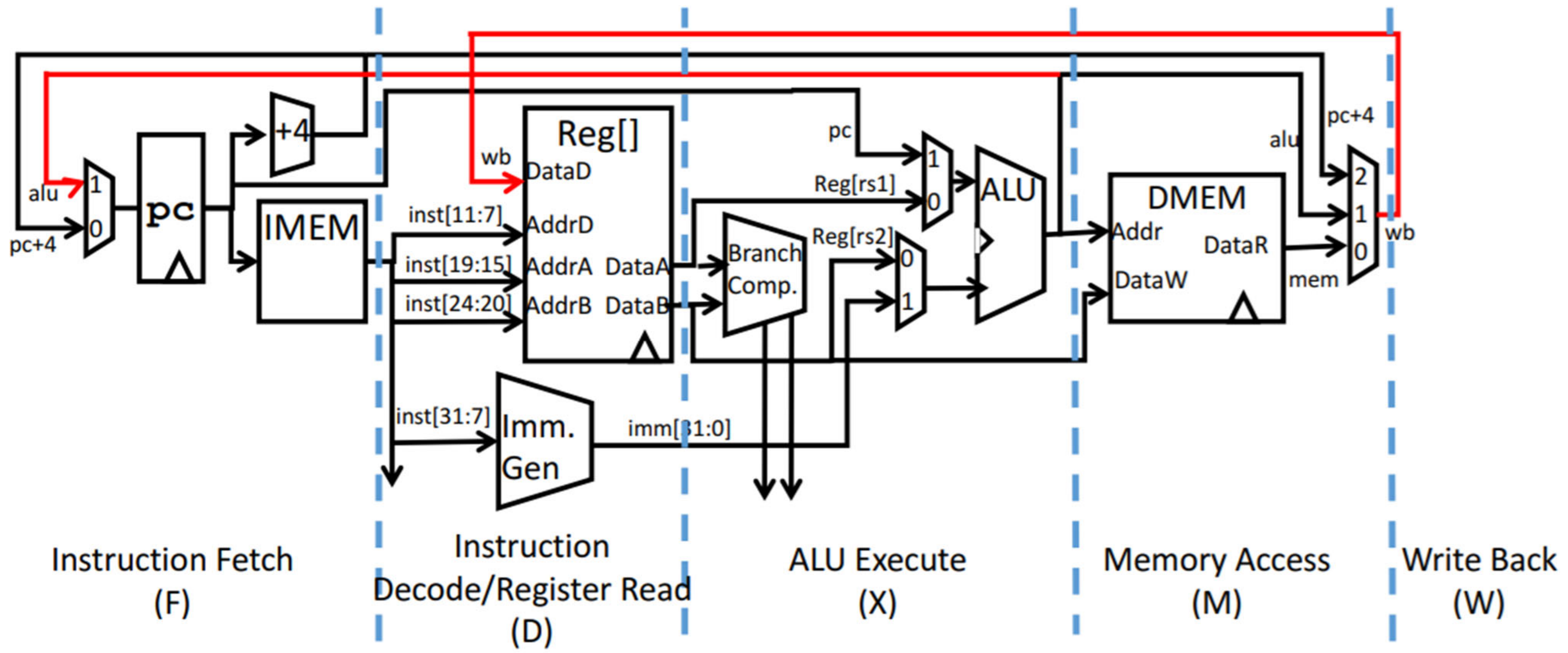
CSE 410

Lecture 11

ISA vs. Implentation

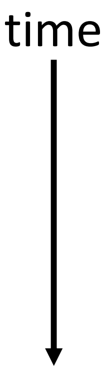
- The ISA defines the operation of the machine as though it were a single cycle implementation
  - All previous instructions have completed before the next one starts
- Pipelining is a mechanism to speed up instruction completion rate
  - The semantics of the ISA have to be respected, though
  - That is, the effect of a pipelined execution of the instructions must be the same as what the ISA expect

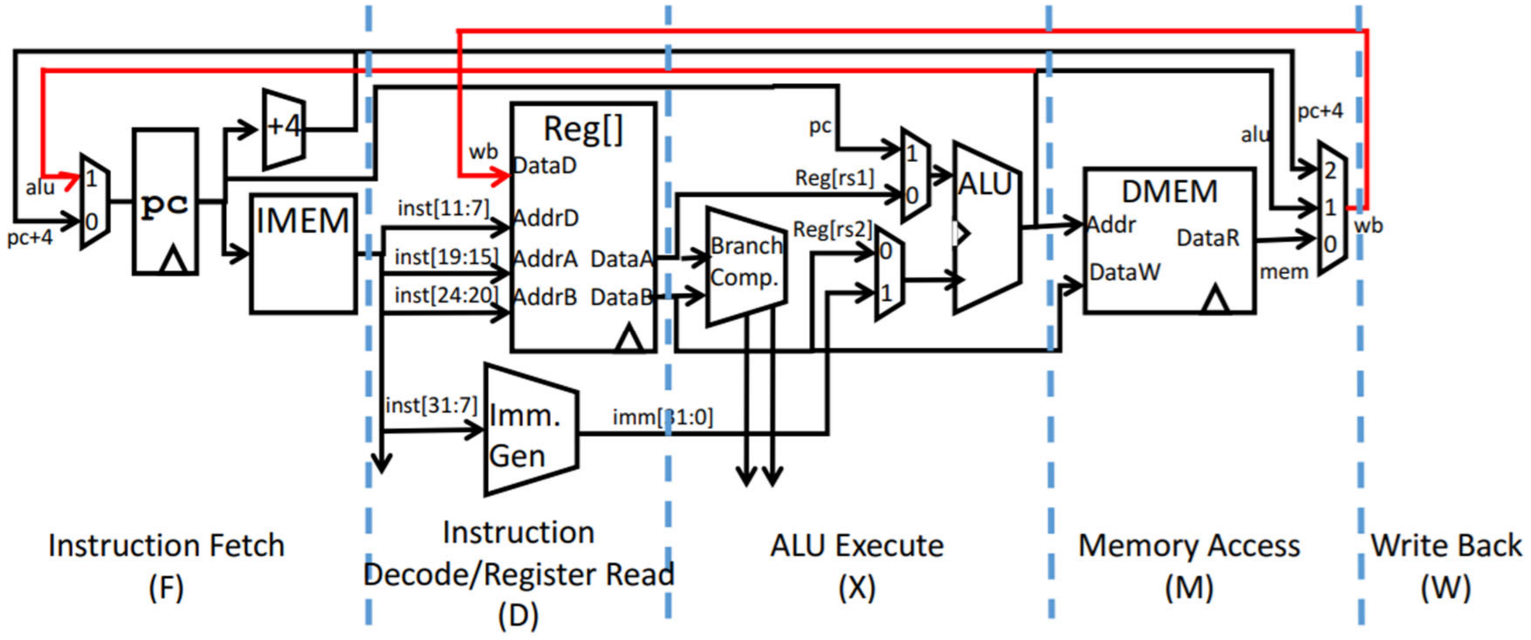
5-Stage Pipeline



Pipelining

add x3, x2, x1
or x6, x5, x4
and x9, x8, x7
addi x11 x10, 1
sub x14, x13, x12





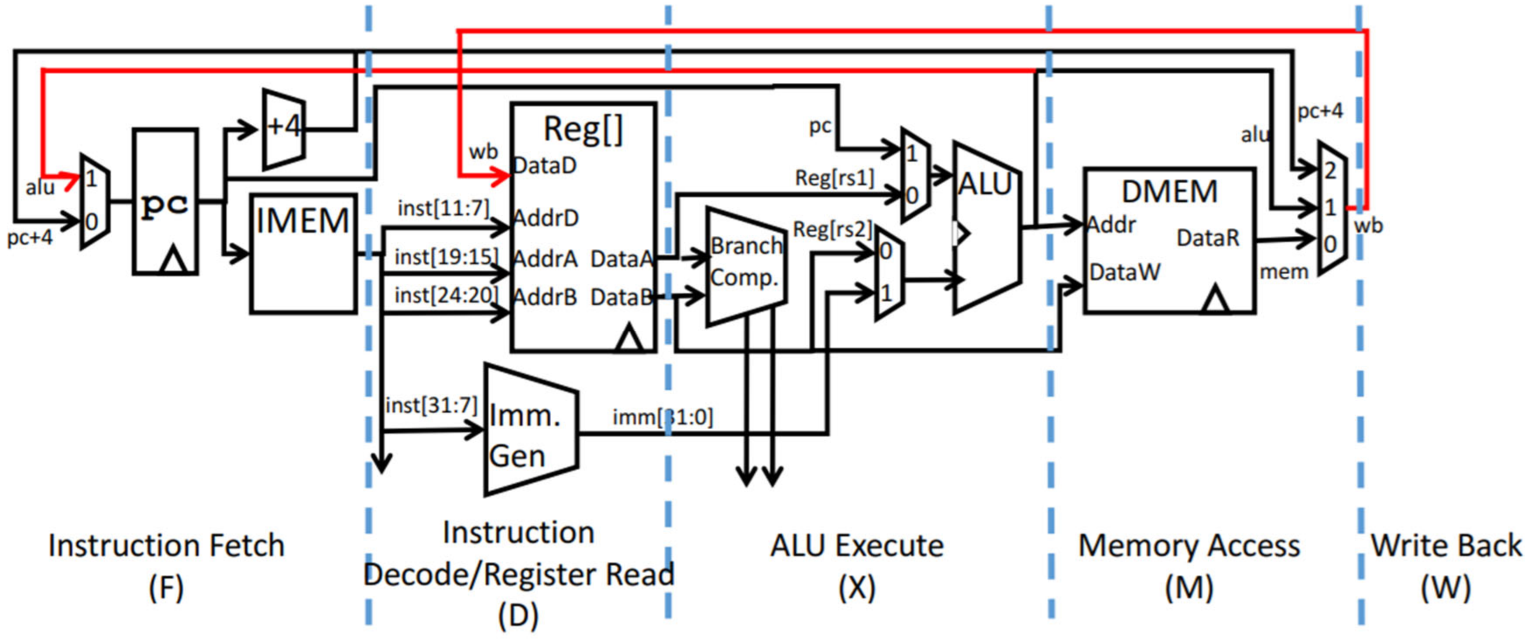
sub x14, x13, x12 addi x11, x10, 1 and x9, x8, x7 or x6, x5, x4 add x3, x2, x1

Hazards – Something goes wrong

time

```
add   x3, x2, x1
or    x6, x5, x4
and   x9, x8, x7
addi  x11, ~~x10~~ x9, 1
sub   x14, x13, x12
```

now



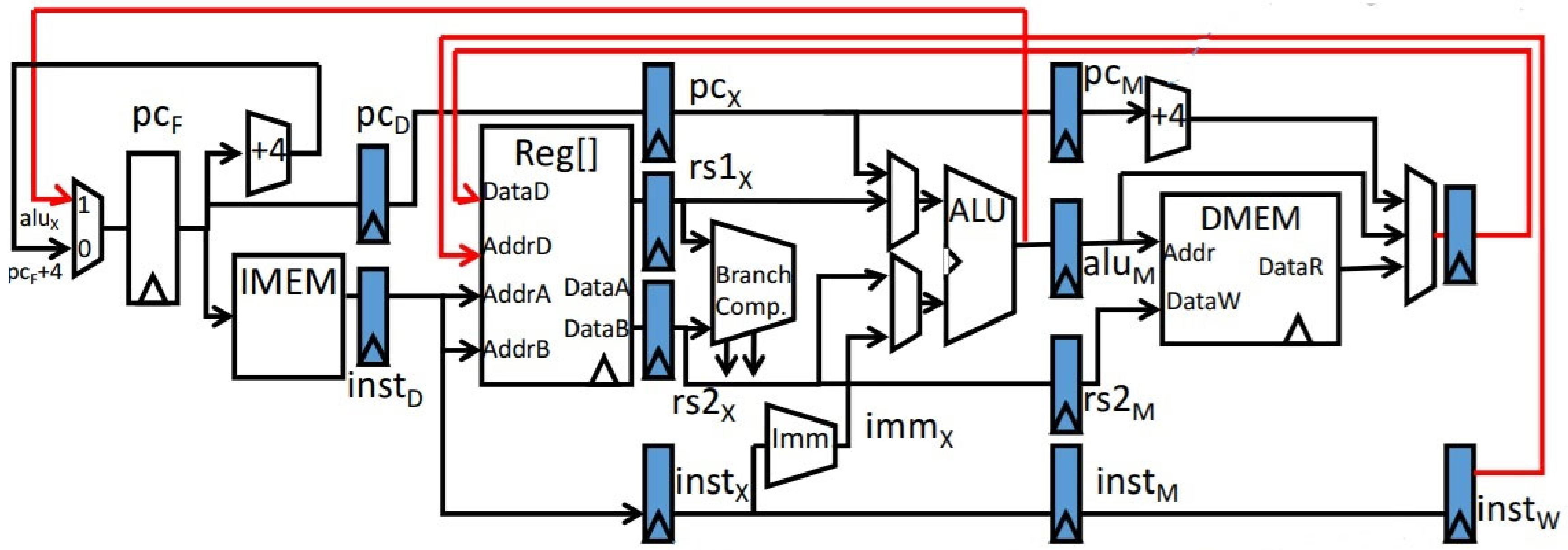
sub x14, x13, x12 | addi x11, x9, 1 | and x9, x8, x7 | or x6, x5, x4 | add x3, x2, x1

Data Hazard

- Later instruction reads a value written by an earlier instruction, but... value not yet written to register

- Instruction N reads register K
- Instruction N-1 or N-2 or N-3 writes register K
- According to the ISA, the write should have occured before the read
- Instruction N should get the value that written by the earlier instruction
- If we’re not careful, in a pipelined implementation it won’t

Data Hazards



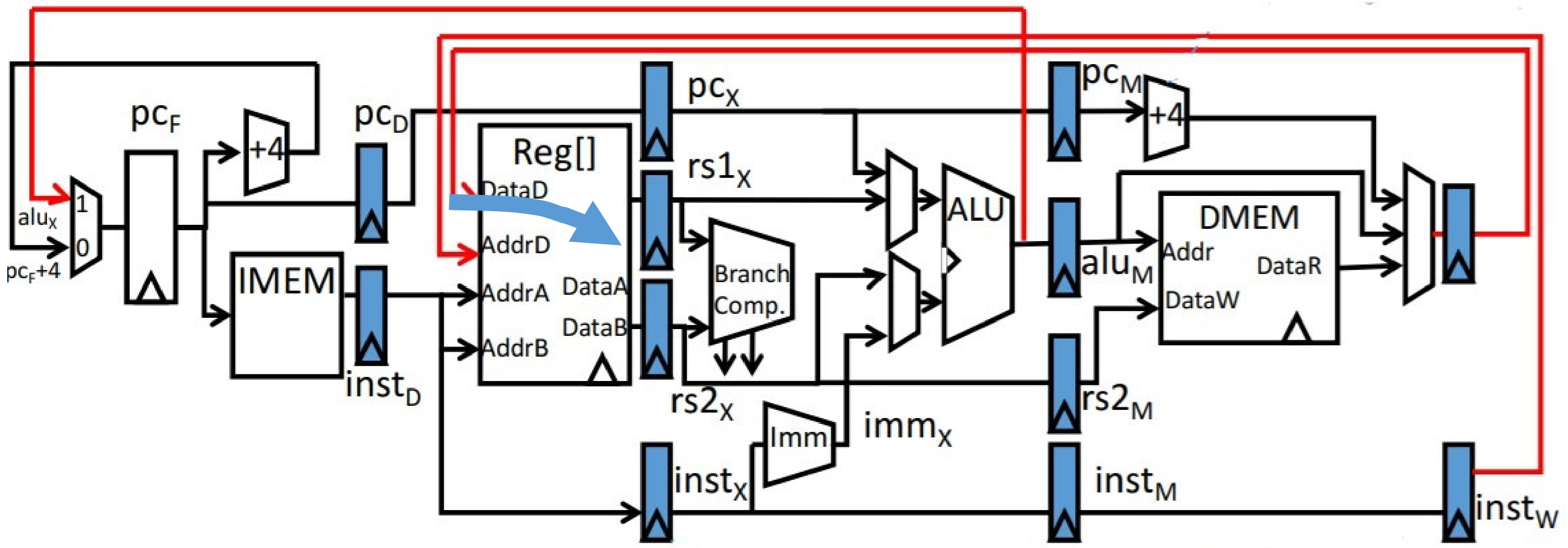
add x6, **x5**, x4

add **x5**, x4, x3

In pipelining, this is called a data hazard.

In general, this is called a data dependence, or a true dependence, or a read-after-write (RAW) dependence.

Data Hazards at Distance 3



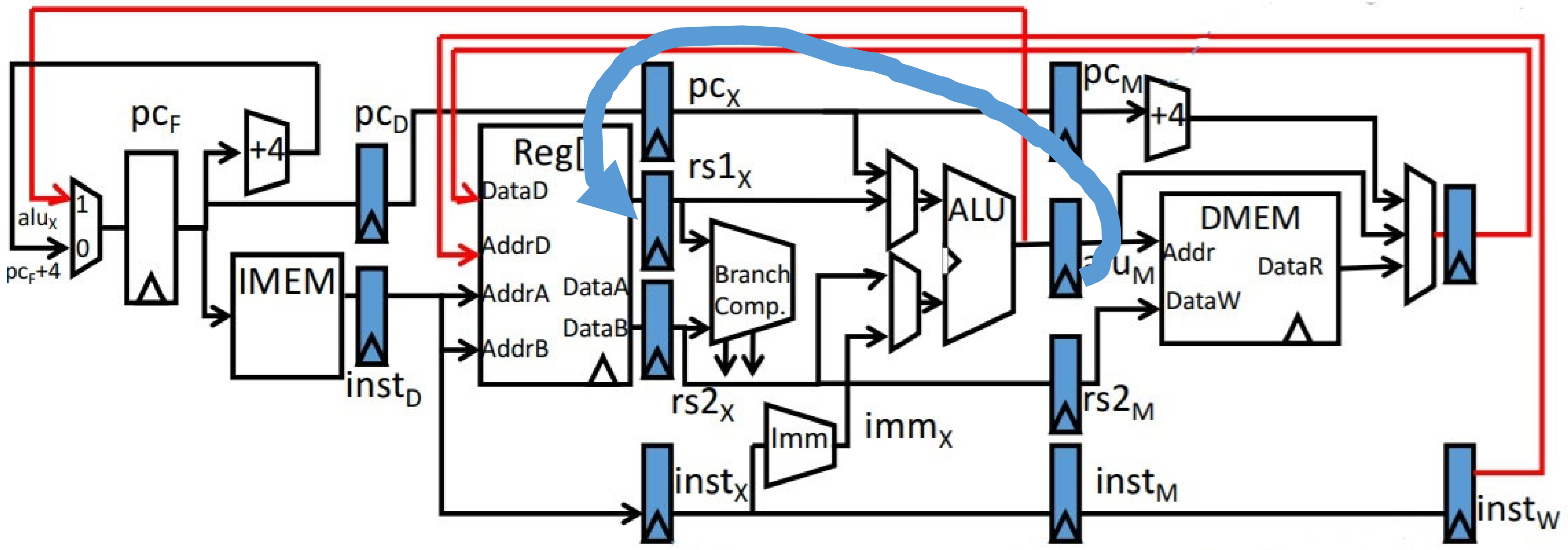
add x6, **x5**, x4

add **x5**, x4, x3

Solution

- design the register file so that if one of the read registers will be written this cycle, send the new value to the output pipeline register
- control gets more complicated

Data Hazards at Distance 2

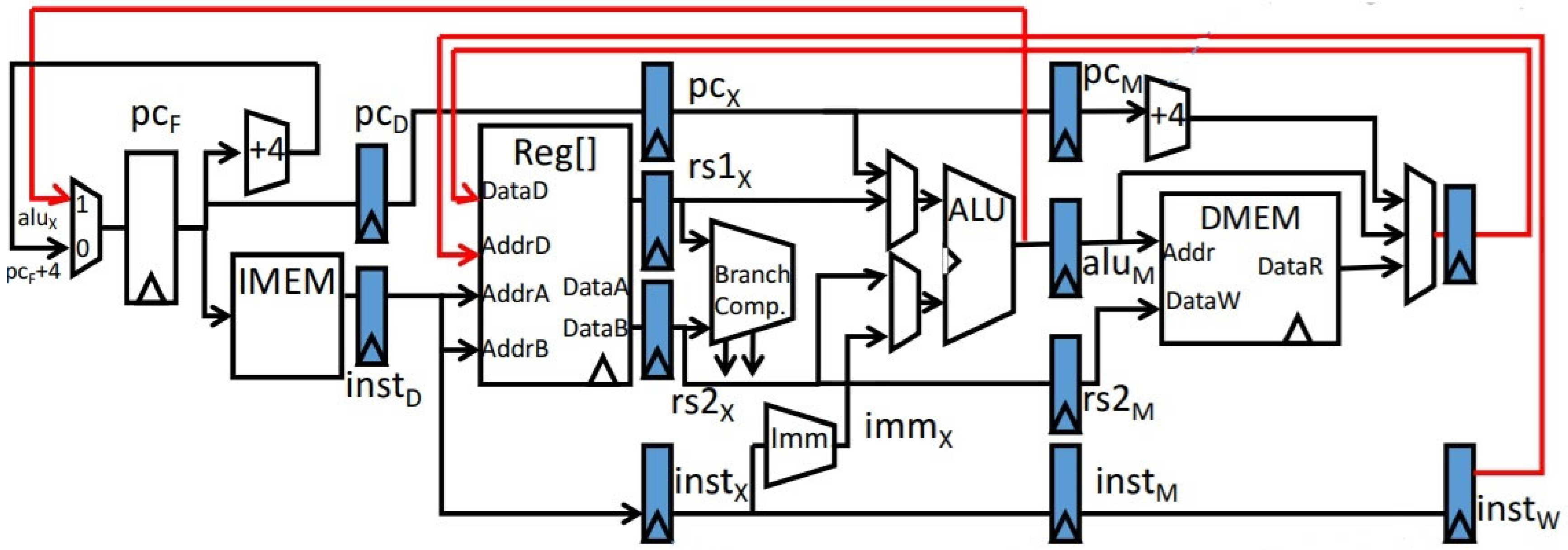


add x6, **x5**, x4   add **x5**, x4, x3

Solution

- If not a lw instruction, “forwarding”
  - The value that will be written has already been produce and is in the ALU pipeline register
  - Feed it back to the rs1 or rs2 pipeline register and use it, rather than the value fetched from the register
  - control gets more complicated

Data Hazards at Distance 2

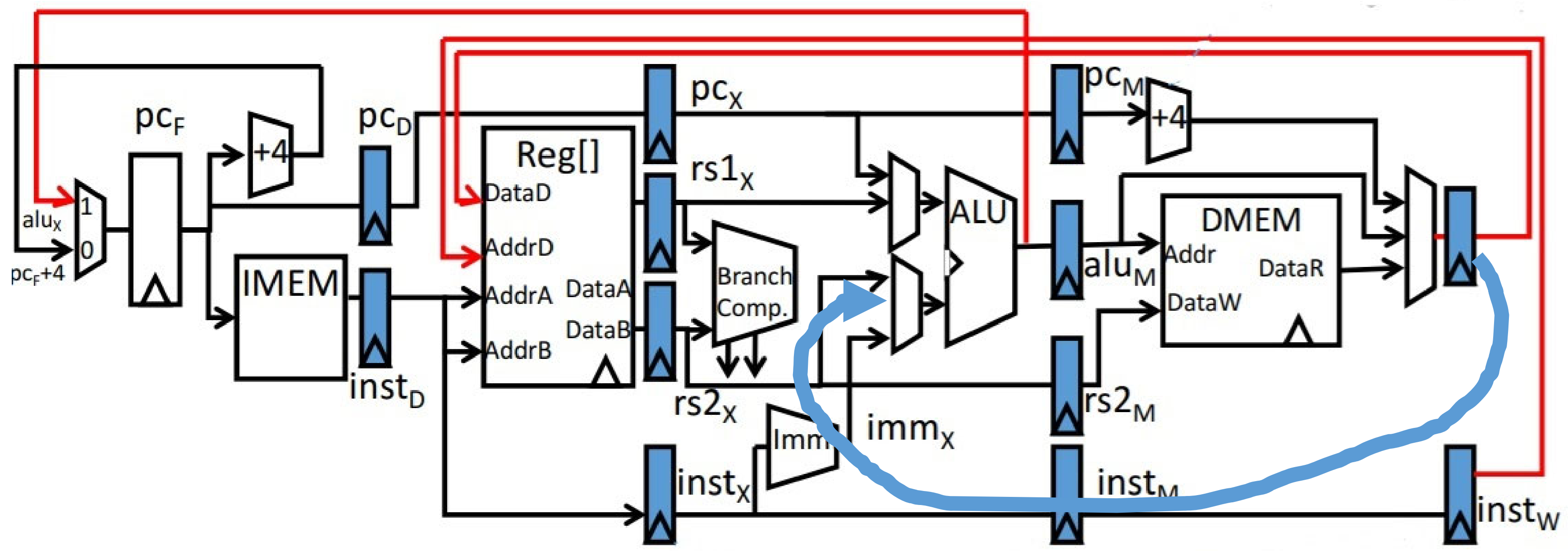


add x6, **x5**, x4   lw **x5**, 12(x3)

Solution

- If a lw instruction
  - the needed data won't be available until the end of this cycle
  - We don't want to make the cycle time longer to wait for it to be produced and then feed it back
  - Result: we put a wrong value into the rs1x or rs2x pipeline register
- BUT, the value is available when the add instruction moves to the ALU

Data Hazards at Distance 2



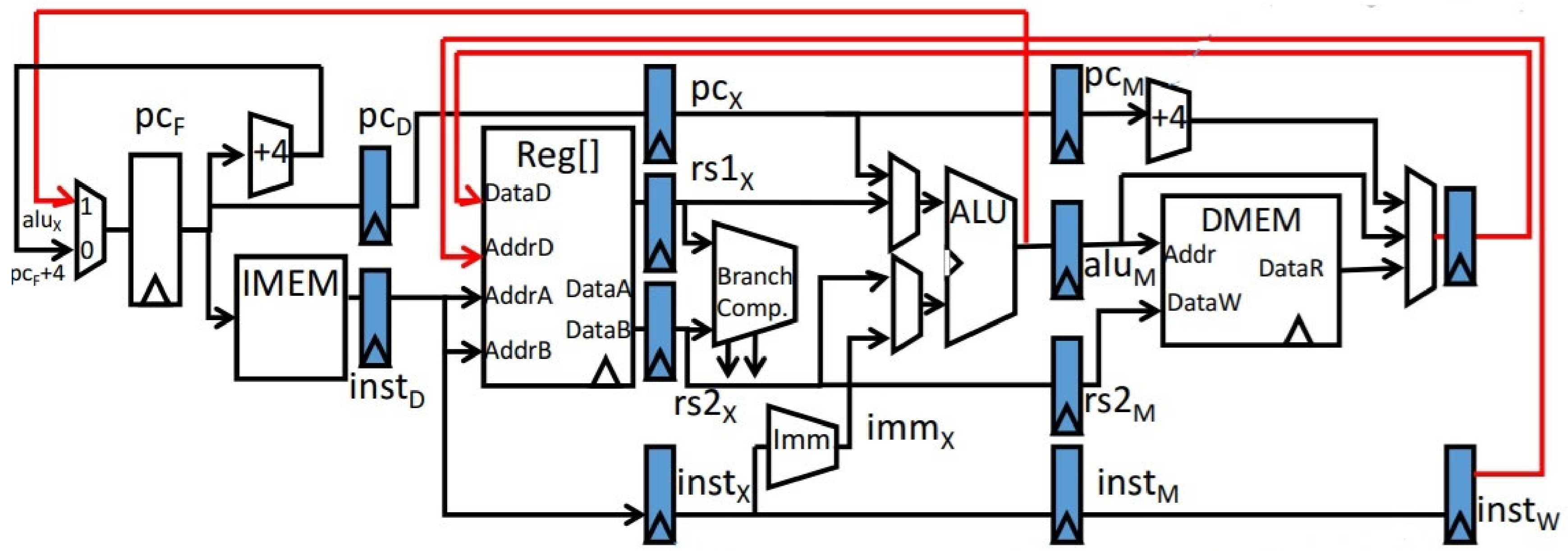
add x6, **x5**, x4

lw **x5**, 12(x3)

Solution

- ...
- BUT, the value is available when the add instruction moves to the ALU
  - control keeps getting more complicated!

Data Hazards at Distance 1

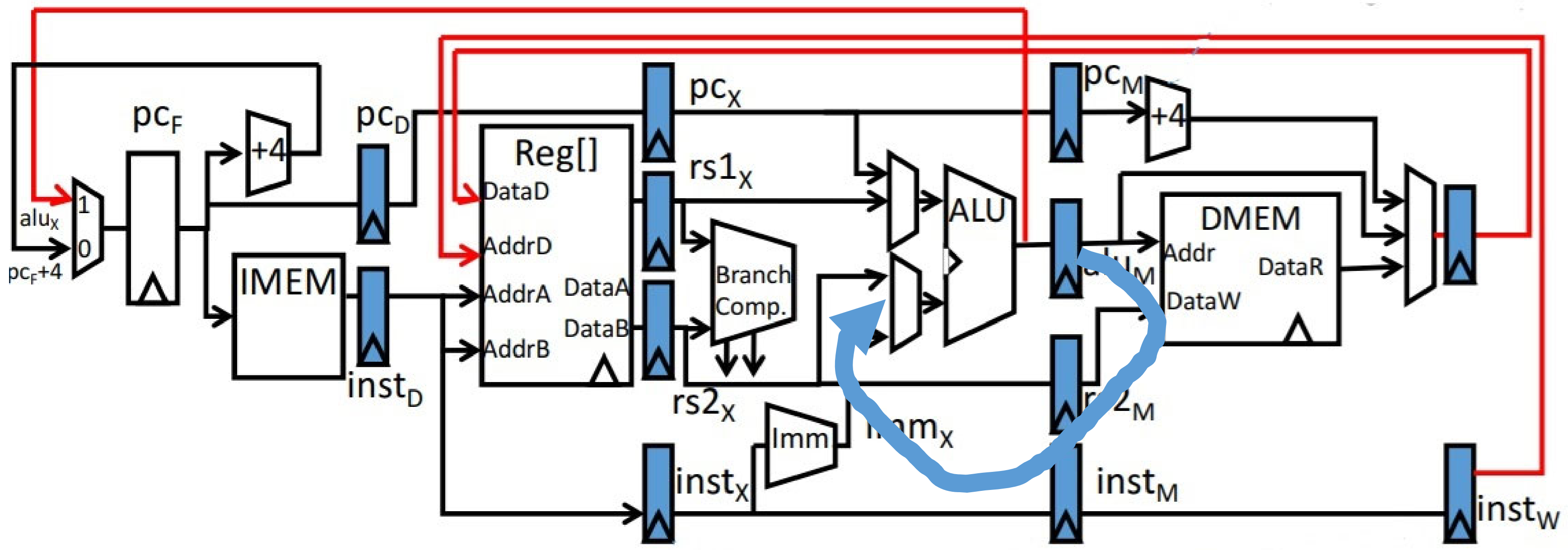


add x6, **x5**, x4   add **x5**, x4, x3

Solution

- If not a lw instruction, "forwarding"
    - Needed value isn't available during when instructions are at these locations in pipeline
    - But...

Data Hazards at Distance 1

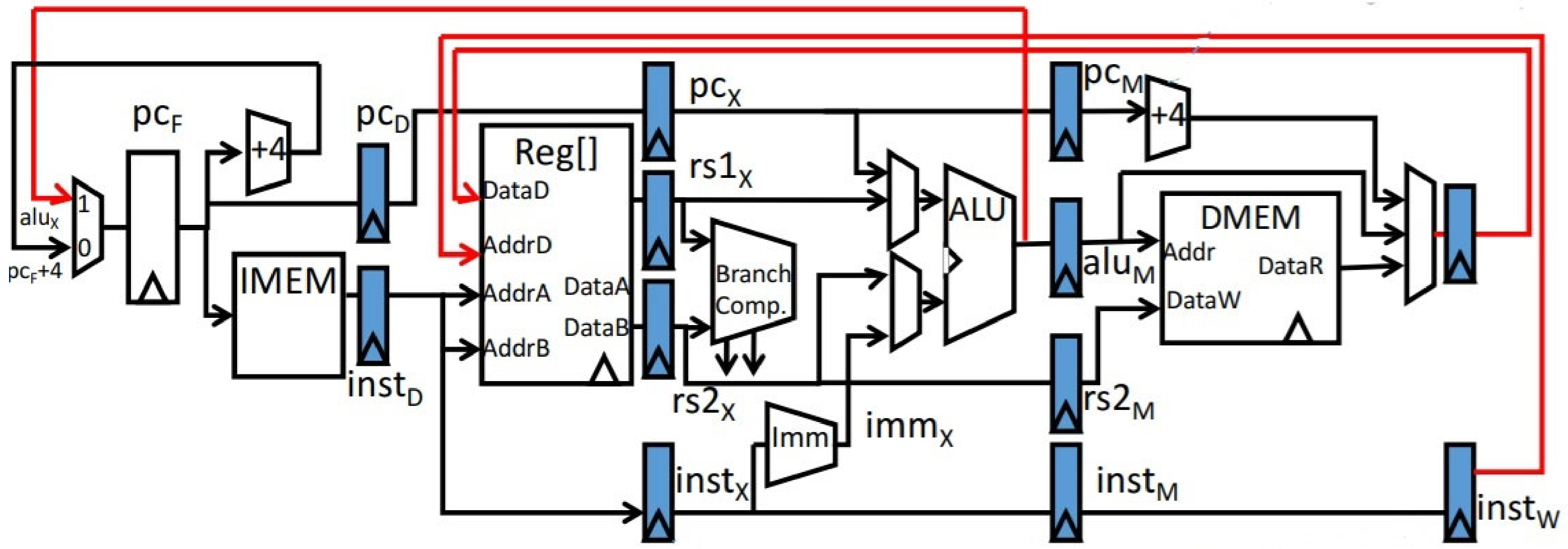


add x6, **x5**, x4 add **x5**, x4, x3

Solution

- If not a lw instruction, "forwarding"
  - Needed value is in ALU pipeline register at start of cycle when it is needed
  - (After this, I'm going to stop repeating that "control gets more complicated")

Data Hazards at Distance 1

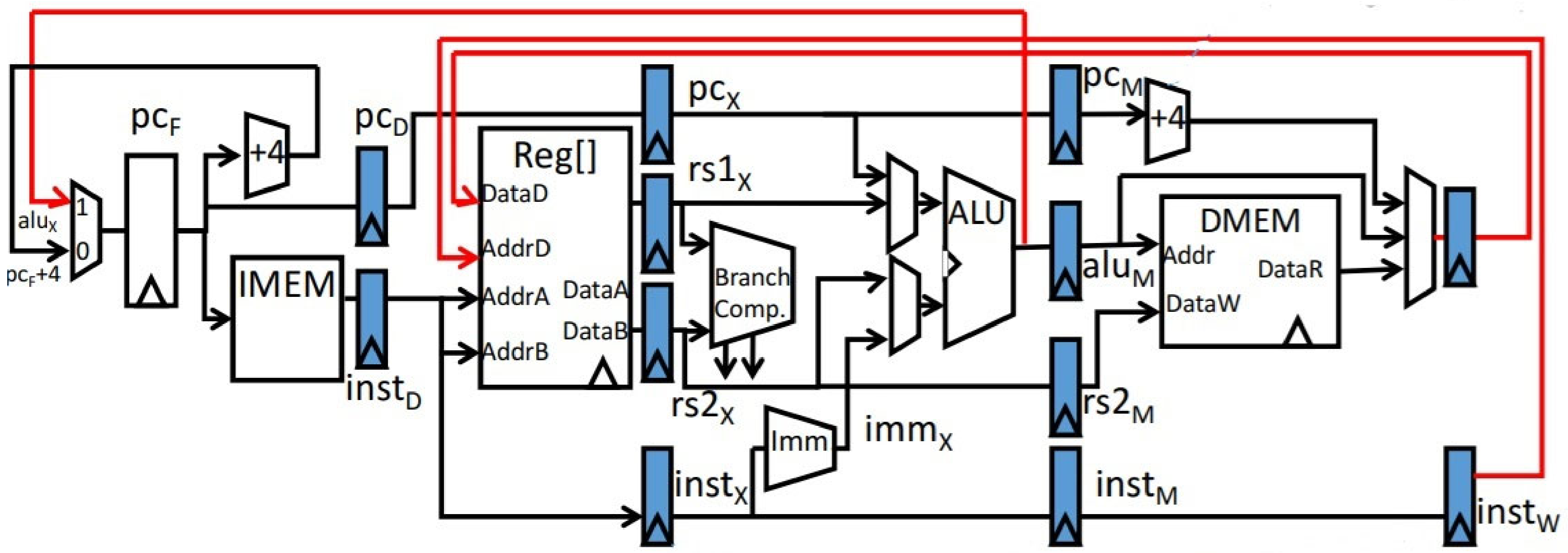


add x6, **x5**, x4   lw **x5**, 12(x3)

Solution

- If a lw instruction, need value isn't available at this stage of processing

Data Hazards at Distance 1

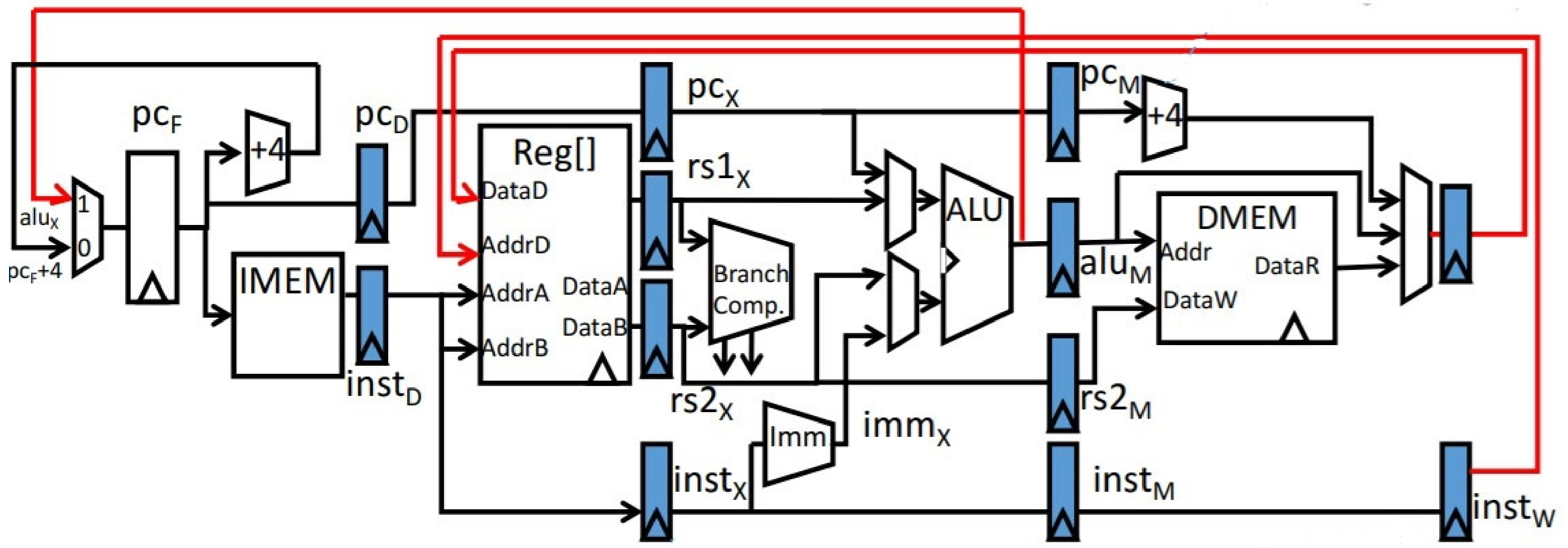


add x6, **x5**, x4 lw **x5**, 12(x3)

Solution

- If the instruction producing the value is a lw, the value isn't available at (the beginning of) this stage

Data Hazards at Distance 1



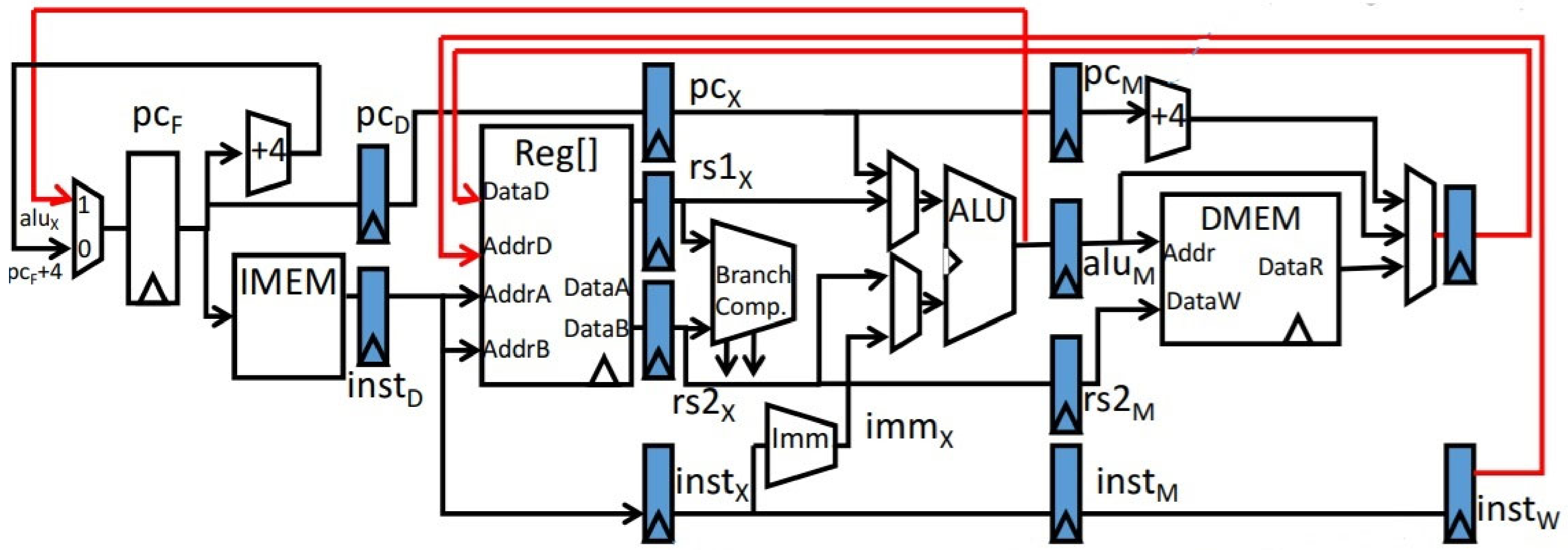
add x6, **x5**, x4 lw **x5**, 12(x3)

Solution

- By the time the value is available, it’s too late
  - Instruction has used ALU but with incorrect input values
  - It won’t have another chance to use the ALU

S

Data Hazards <u>at Distance 1</u>



add x6, **x5**, x4 nop (add x0, x0, x0) lw **x5**, 12(x3)

Solution

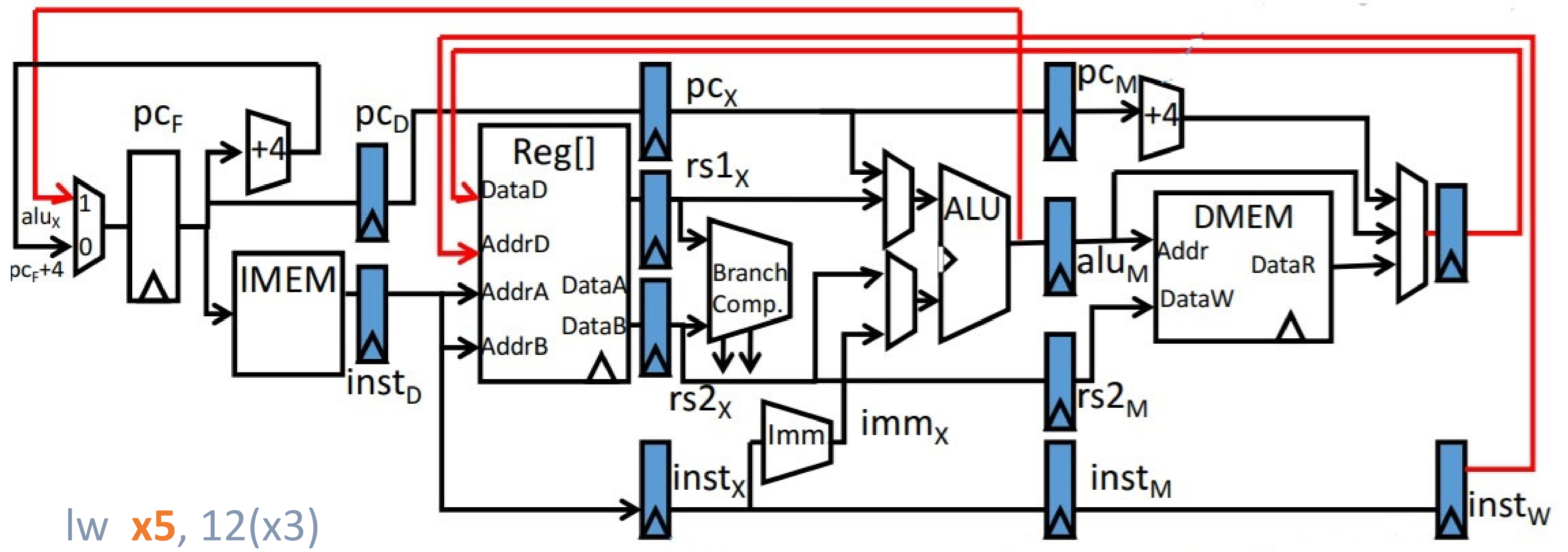
- If there is "a bubble" between the two instructions, then we can resolve the dependence by forwarding (as it's now distance two)
- Note: the bubble uses the pipeline but doesn't do anything useful
  - Bubbles slow down the pipeline

Where Do Bubbles Come From?

- In early RISC architectures, it was the programmer’s responsibility to explicitly code them
  - It was a programmer error to try to use the value produced by a lw instruction in the immediately following instruction
  - You had to explicitly code a nop instruction
    - The hardware didn’t detect it if you didn’t, you just got wrong results
  - Of course, “the programmer” is a compiler, so it’s not such a big deal to have to insert NOPs where needed

- In RISC-V, things have advanced and building complicated control isn’t such a big deal
  - The hardware inserts the NOP whenever needed

Data Hazards at Distance 1

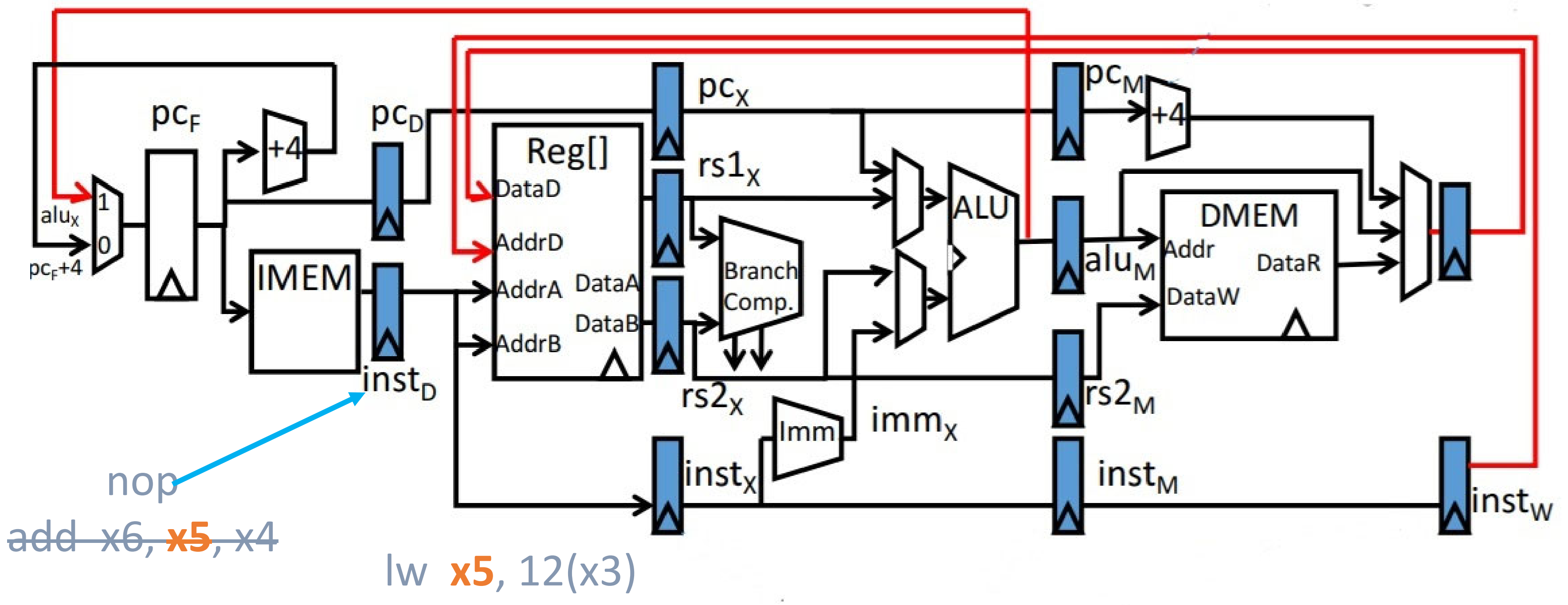


lw **x5**, 12(x3)

Bubble Solution

- When control fetches the lw instruction, it remembers that in the inst_D pipeline register

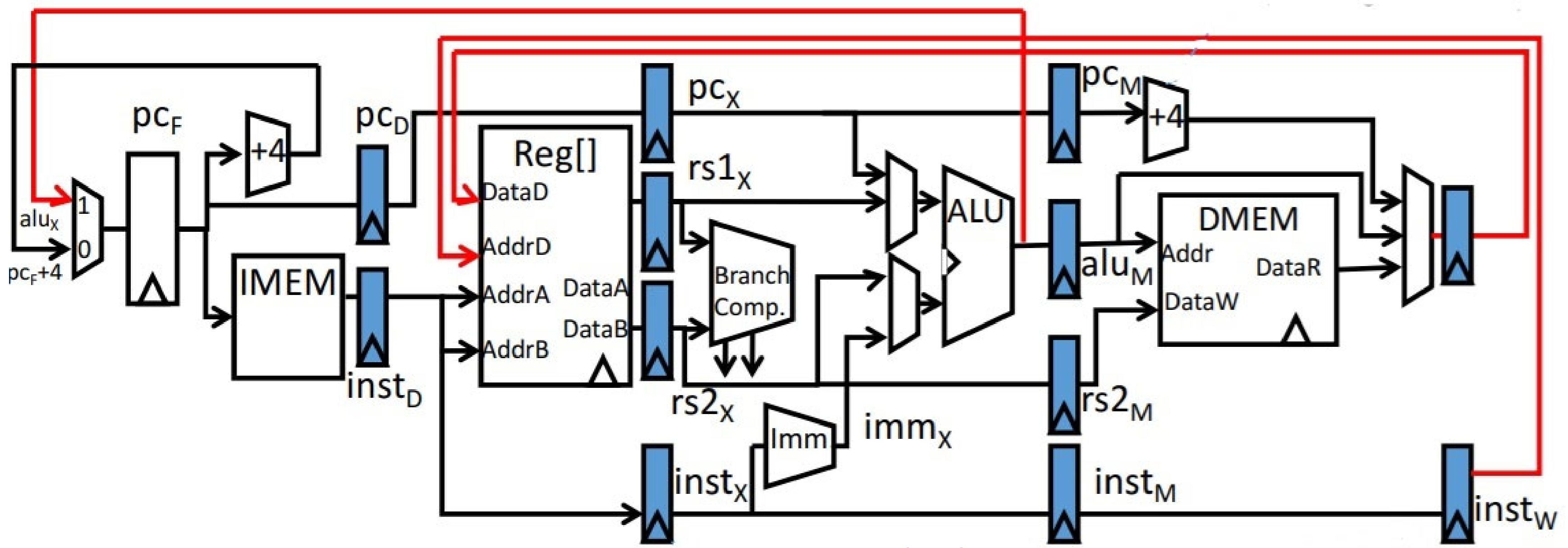
Data Hazards at Distance 1



Bubble Solution

- When control fetches the next instruction, it detects that it reads a register that is written by the previous instruction, and that the previous instruction was a lw
- So, it
  - injects a nop into the inst$_D$ pipeline register, rather than the add instruction
  - disables updating of the PC
    - (Yes, control keeps getting more complicated)

Data Hazards at Distance 1

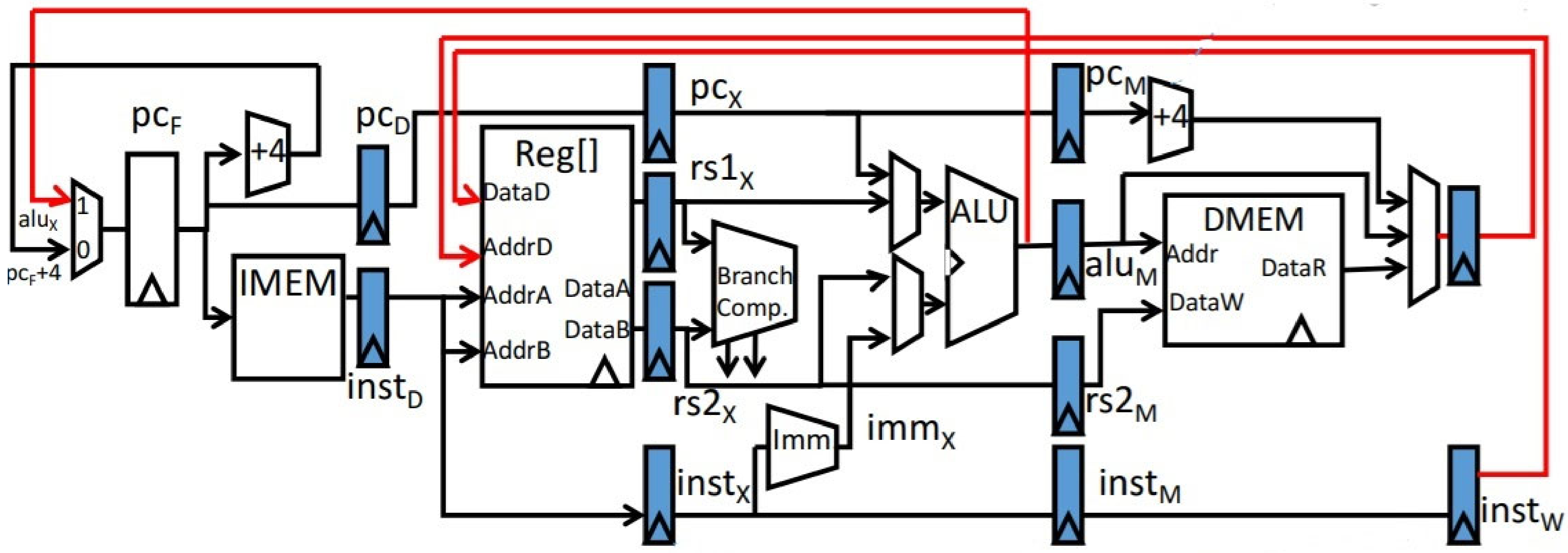


add x6, **x5**, x4   nop   lw **x5**, 12(x3)

Bubble Solution

- Because the PC wasn’t updated last cycle, this next cycle the same add instruction is fetched
- There is no dependence between the add and the nop, so control “dispatches” the add instruction

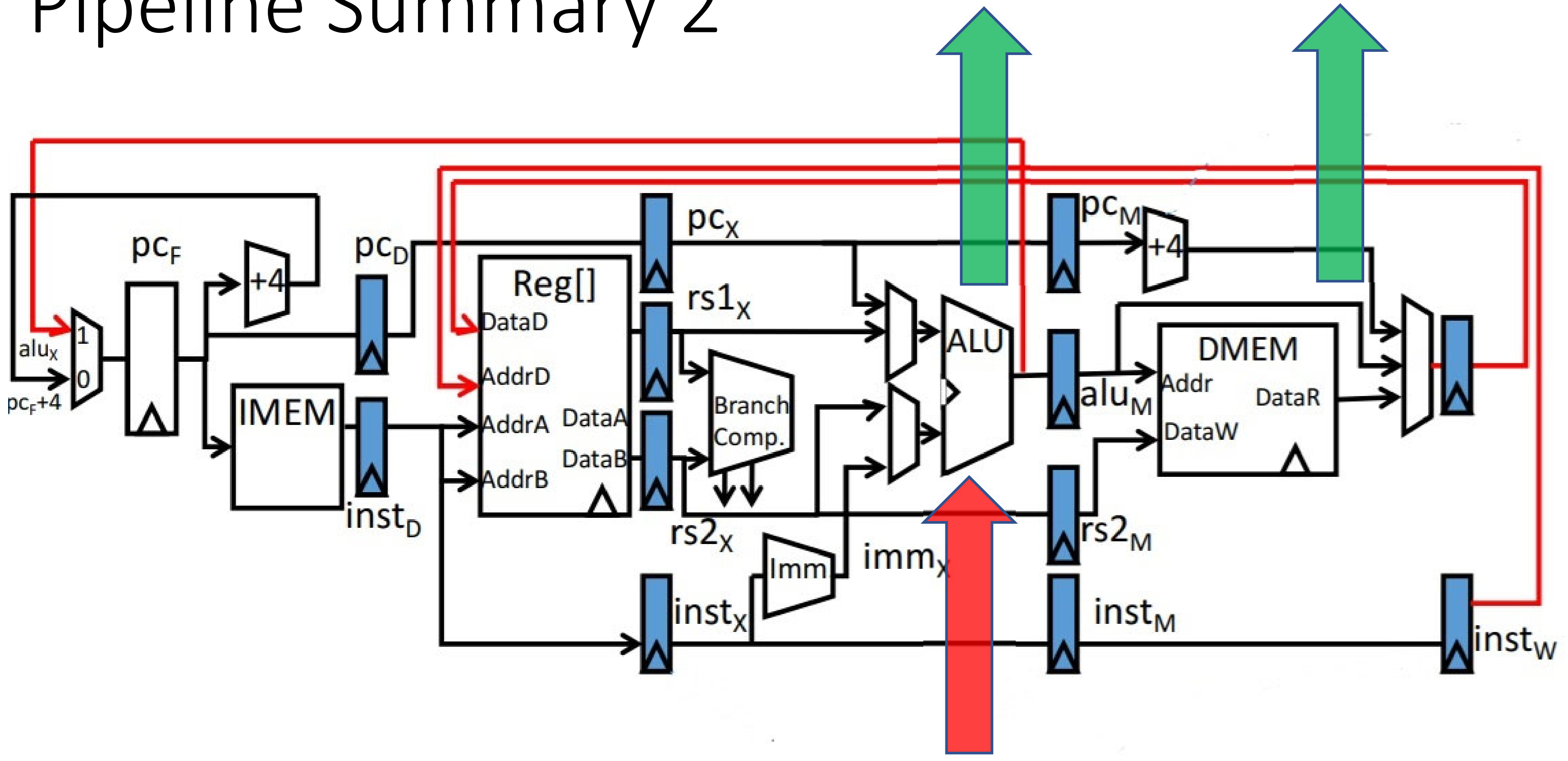
Pipeline Summary 1



add x6, **x5**, x4 nop nop nop add **x5**, x4, x3

Without forwarding, the instruction reading a value written to a register must be at least four instructions behind the instruction that writes the register.

Pipeline Summary 2



• With forwarding, the use of a value must occur on a cycle later than the production of the value

Pipeline Summary 3

- Without forwarding, how many cycles does it take to issue these instruction sequences?

  -
    ```
    addi    x4, x0, 10
    addi    x5, x0, 20
    add     x5, x5, x4
    ```

```
0:  addi  x4, 0, 10
1:  addi  x5, 0, 20
2:  nop
3:  nop
4:  nop
5:  add   x5, x5, x4
```

- How many with forwarding?

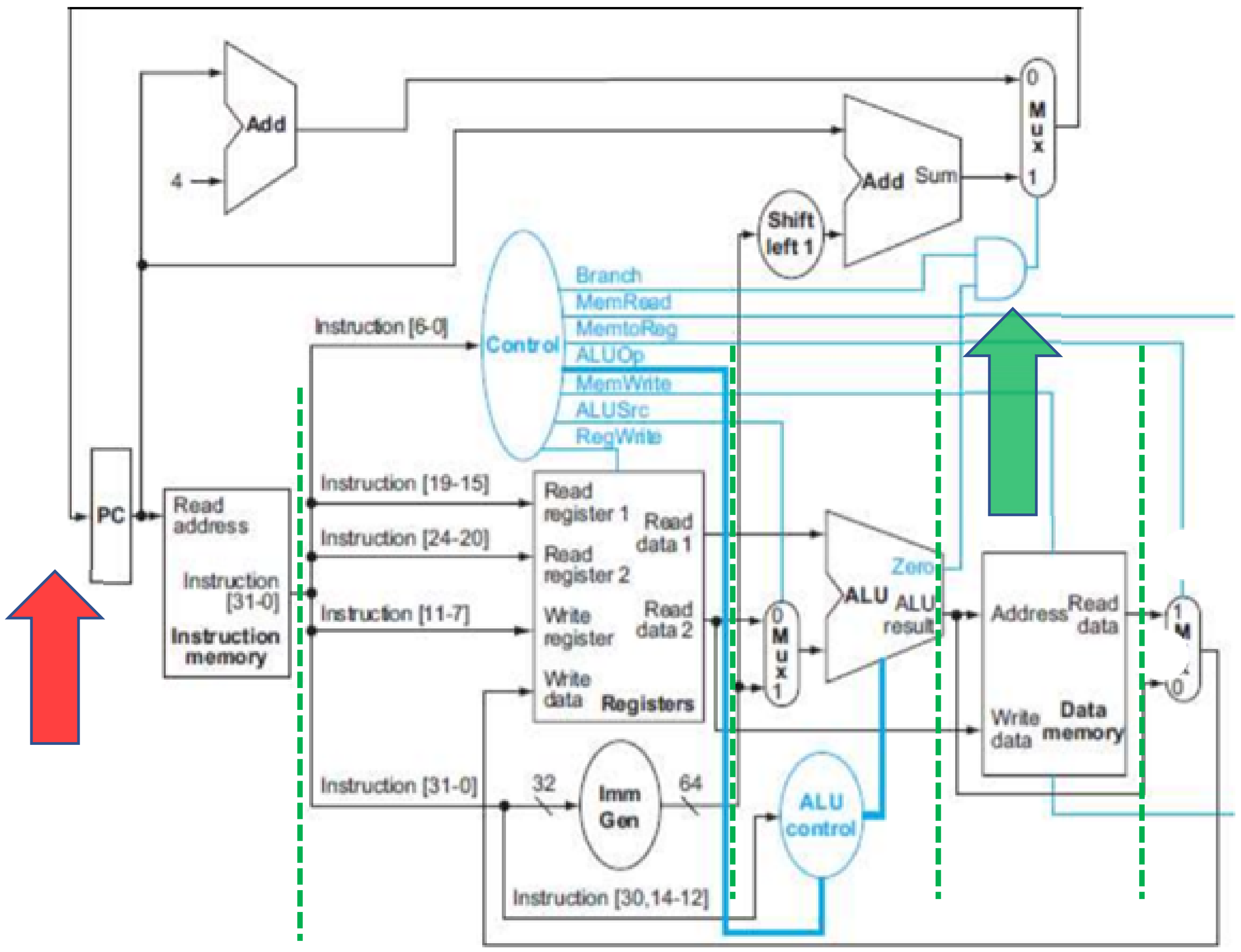
```
0: addi x4, 0, 10
1: addi x5, 0, 20
2: add x5, x5, x4
```

Control Hazards

- Control hazards result because we don’t know whether or not a branch will be taken for two cycles after they’re fetched
  - What instruction should we fetch in the cycle after we fetch the branch?
    - The next sequential instruction (as if the branch weren’t taken)
    - The instruction at the target address (as if the branch were taken)
  - Note that the branch needs to get to the ALU stage to compute the target address, so maybe we have only one choice?
- Control hazards are data hazards on the PC

Control Hazards

What value should the PC be set to during this cycle?

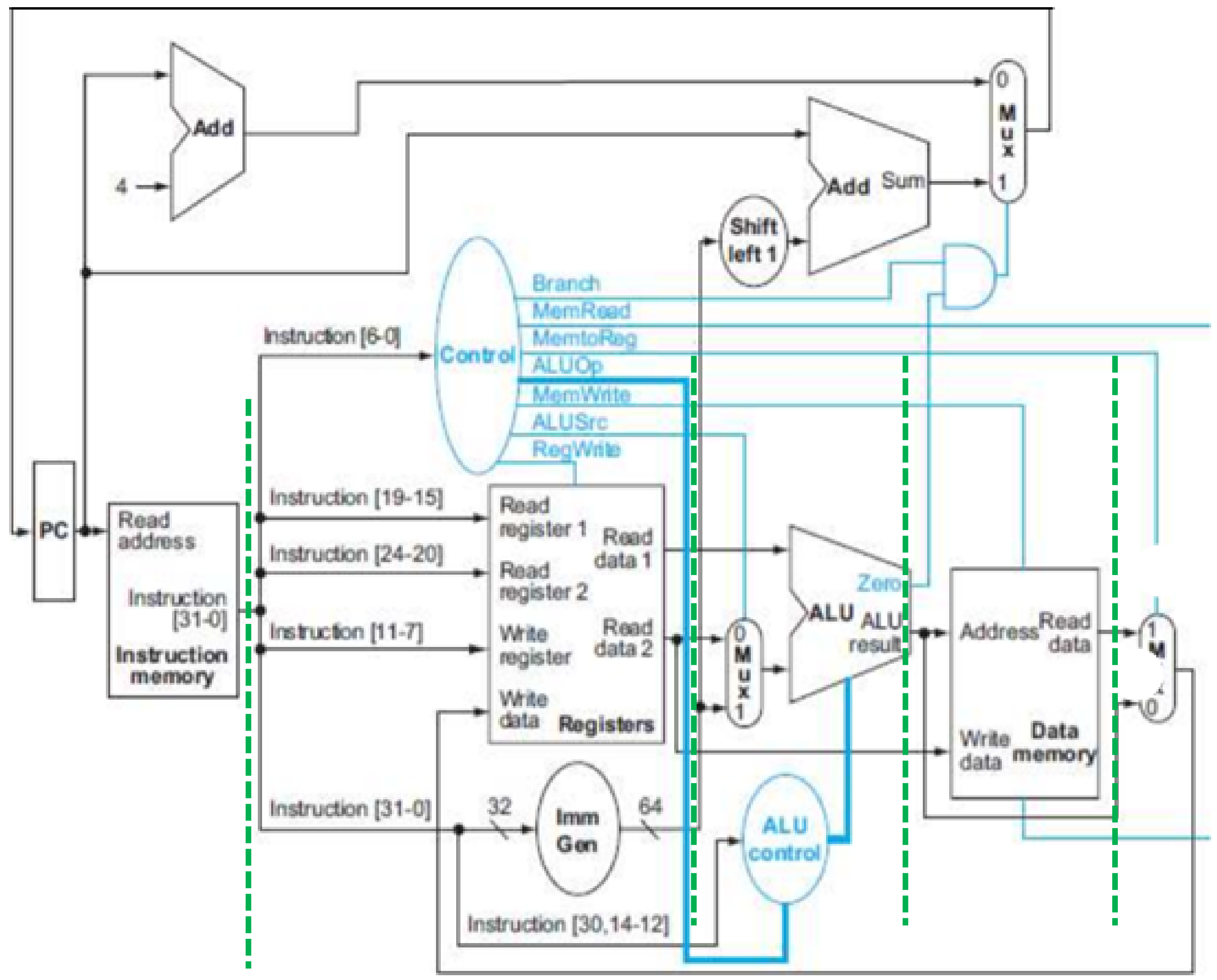


beq x4, x5, loop
add x5, x4, x3
add x8, x7, x6
...

beq x4, x5, -20

Control Hazards

At the ALU stage we can resolve whether the branch is taken or not, and what the target address is. But what two instructions should have been issued after the branch?



beq x4, x5, loop
add x5, x4, x3
add x8, x7, x6
...

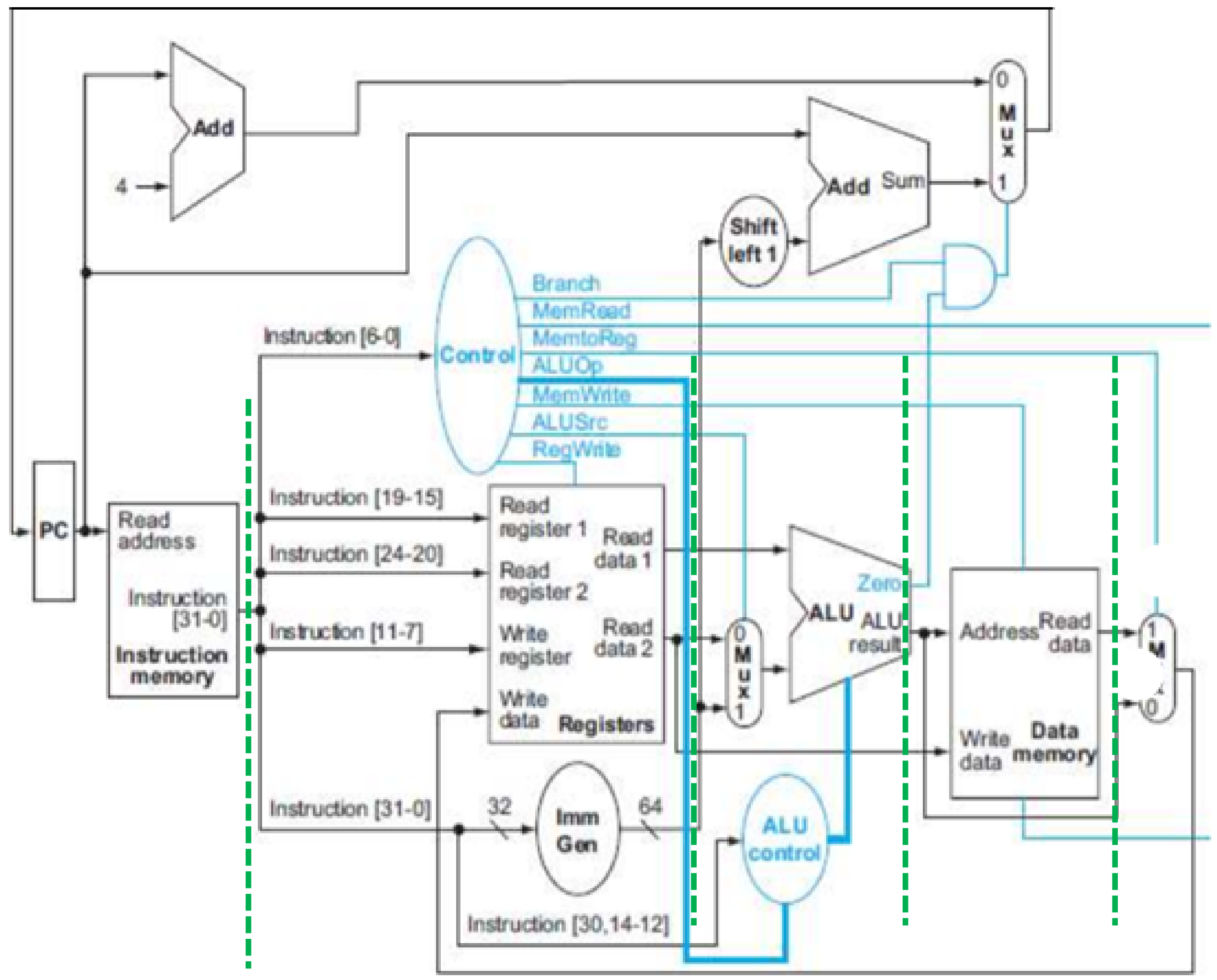
?

?

beq x4, x5, -20

Control Hazards – Pessimistic

Control notices branch and inserts two nops.



beq x4, x5, loop
add x5, x4, x3
add x8, x7, x6
…

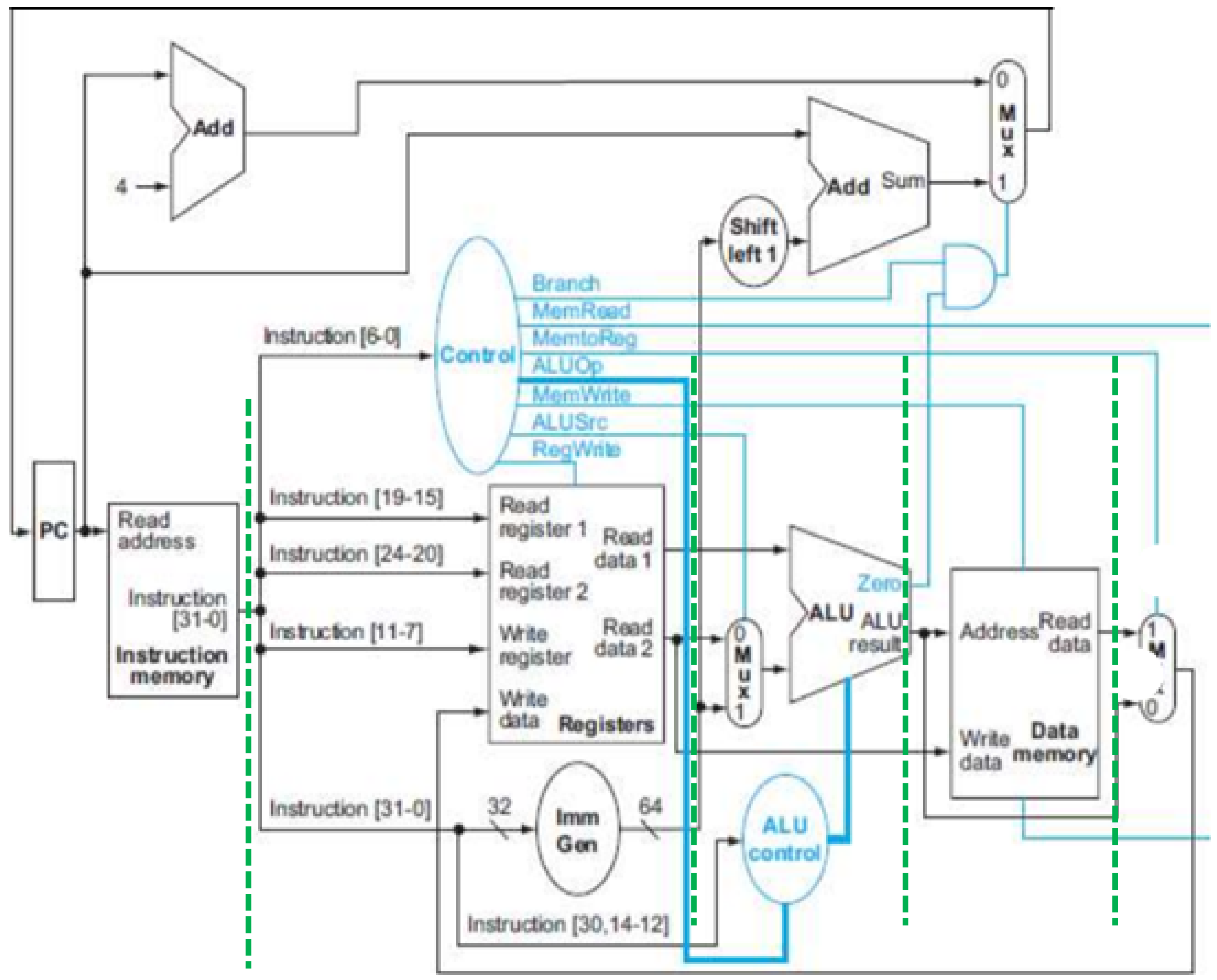
nop

nop

beq x4, x5, -20

Control Hazards – Speculative

Control fetches instructions consecutively from memory.



beq x4, x5, loop
add x5, x4, x3
add x8, x7, x6
...

If at this stage branch is taken, turn trailing instructions to nops (in pipeline registers).

If branch isn't taken, great!

add x8, x7, x6

add x5, x4, x3

beq x4, x5, -20

Real Machines – Branch Prediction

- Control maintains a table something like this:

PC	Next address
0x40180	0x442C0
0x3822C	0x38210
…	…

- Each row corresponds to a branch instruction in memory
- The PC column identifies which branch it is
- The “next address” column is a prediction
  - For instance, when a branch is taken, put update table with branch target address; when not taken, update table with next sequential address.
- Real machines use much more sophisticated schemes

Another Pipelining Summary

- Pipelining is one approach to parallelism
  - Parallelism is the key to “going faster”
- The cycle time in a single cycle implementation has to be the worst case delay through the entire data path
- The cycle time in an N stage pipeline has to be the worst case delay through any one stage
  - So, ideally can get up to an N-fold increase in speed\
- So, why not make a 300-stage pipeline?

Limits to Pipelining

- Because of imbalances in the stages and overheads writing pipeline registers, when you double the number of stages you probably don’t have the cycle time
- Hazards result in bubbles
  - The more stages, the larger the number of bubbles needed
  - The benefits of more stages are limited by the bubbles
- More stages require more pipeline registers and more control, and those take area and power

Mild Lessons for Software

- Hardware
  - Actual performance is much more complicated than just instruction count
  - Actual performance is much more complicated than clock rate
- Software
  - Long sequences of consecutive instructions go fastest, so...
  - Try to avoid branching!
    - Some processors have had “conditional instructions” – they were nops (no operation) unless the last comparison instruction evaluated to true
      - Why?

Beyond Pipelining: Instruction Level Parallelism

- Imagine that the processor has many ALUs, possibly many memory interfaces, an instruction fetch unit that can fetch more than one instruction at a time, and very sophisticated control
- It fetches a bunch of instructions at once and tries to complete their execution in the shortest time possible

```
add   x5, x4, x3
addi  x4, x3, 10
add   x6, x5, x2
addi  x7, x0, 20
add   x4, x7, x2
```

- What is “the shortest time possible”?

Beyond Pipelining: Instruction Level Parallelism

- When an instruction is eligible for execution is limited by the "dependences" it has
- There are four of them, three of which restrict parallel execution
- RAW – Read-after-write (aka true dependence or dataflow dependence)

```
    addi   x10, x8, x9
    add    x12, x10, x11
```

  - *For simplicity, assume each instruction takes one cycle to complete once issued*
- WAR – Write-after-read

```
    add    x12, x10, x11
    addi   x10, x8, x9
```

- WAW – Write-after-write

```
    add    x10, x12, x11
    addi   x10, x8, x9
```

- RAR – Read-after-read

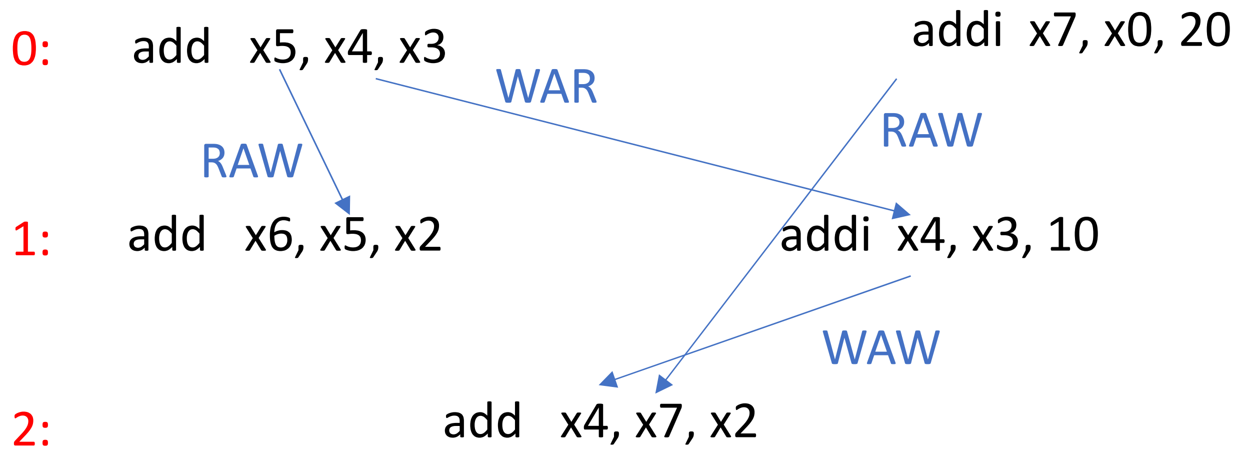
```
    add     x10, x8, x9
    add     x11, x7, x9
```

  - *These two instructions can be executed during the same cycle*

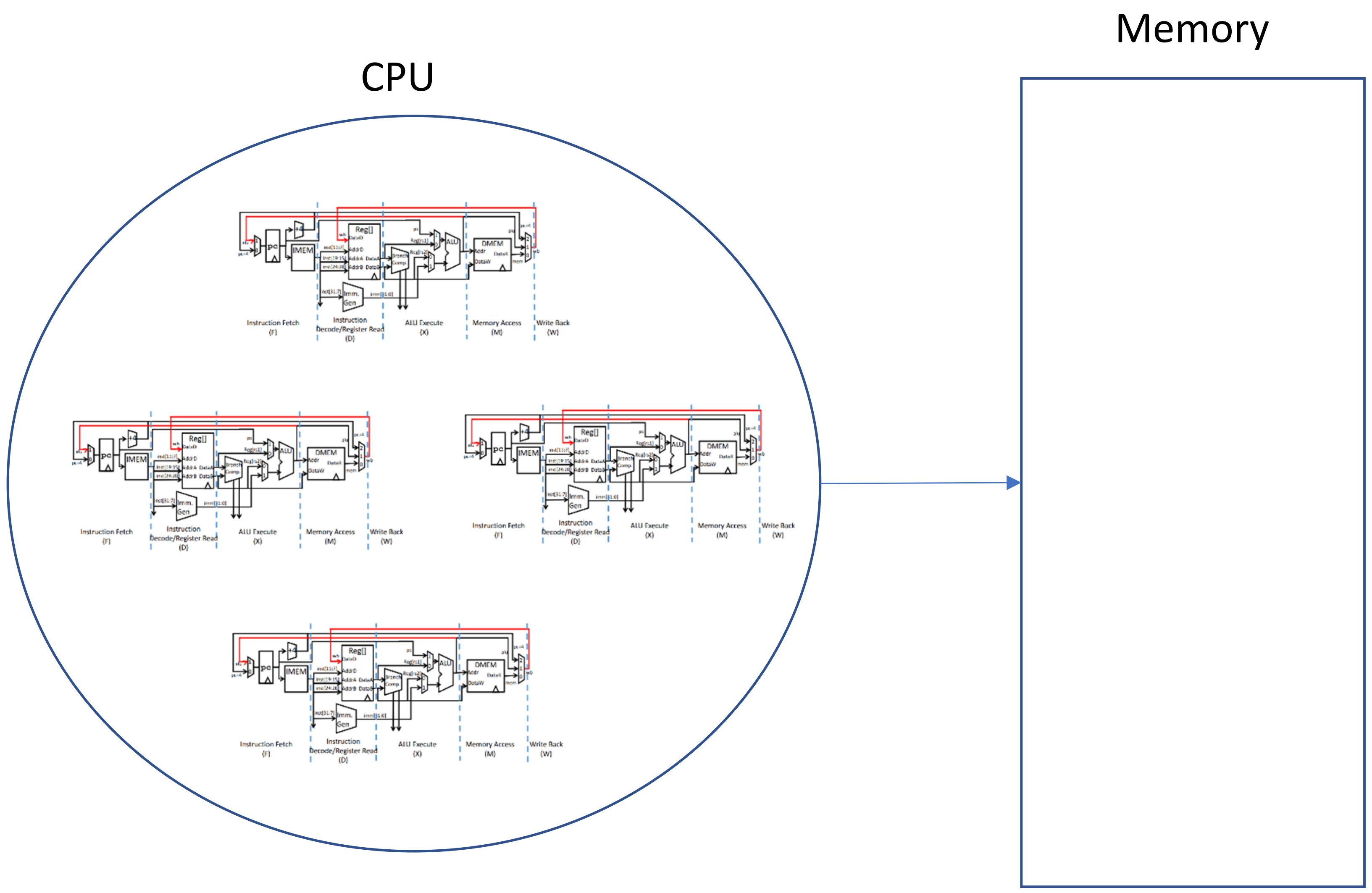
Dependence Graph

What is "the shortest time possible" execution (given unlimited hardware)?

```
add   x5, x4, x3
addi  x4, x3, 10
add   x6, x5, x2
addi  x7, x0, 20
add   x4, x7, x2
```



Hardware Parallelism: Cores



Hardware Parallelism: Cores

- One way to make a CPU implementation faster is to increase its clock rate
  - 1 GHz vs 4.8 GHz
- The downside of that is that the power consumed goes up with the square of the frequency, and...
- The downside of that is heat – the amount of power that must be dissipated in the small area of the chip

- That led to multicore processors
  - Instead of one core going at 16 GHz have 4 cores going at 4 GHz

- Pro: Potentially lots of instructions per second at manageable heat
- The instructions of each running program can use only one core
  - If you have only one program you want to run, you get only single core performance
  - You "increase exploitable parallelism" by pushing the job up a level. The user needs to run many programs at once.

Hardware Parallelism: Hardware Threads

- When we parallelize the execution of each program on a single core by pipelining, performance is limited by both data and control hazards
- When we parallelize the execution of each program by using more advanced control and set of hardware components and doing dynamic data dependence at the instruction level, we end up having a lot of hardware components that sit unused
- Why not use them?
  - Can’t find enough ILP (instruction level parallelism) in a single program
- Idea: Run more than one program on the core at a time
  - The instructions from one program are always completely independent of those from a different program
    - There are no RAW, WAR, or WAW dependences
- We’ll see this idea (increase utilization of hardware by running many independent programs at once) again in the OS portion of the course