ELECTRONICS AND COMMUNICATION TECHNOLOGIES: ELECTRONICS SYSTEMS

LM Cyber Security – Fall 2024

Federico Baronti, Luca Crocetti

Dip. Ing. Informazione

Via G. Caruso, 16 – Stanza B-1-09

050 2217581 – federico.baronti@unipi.it

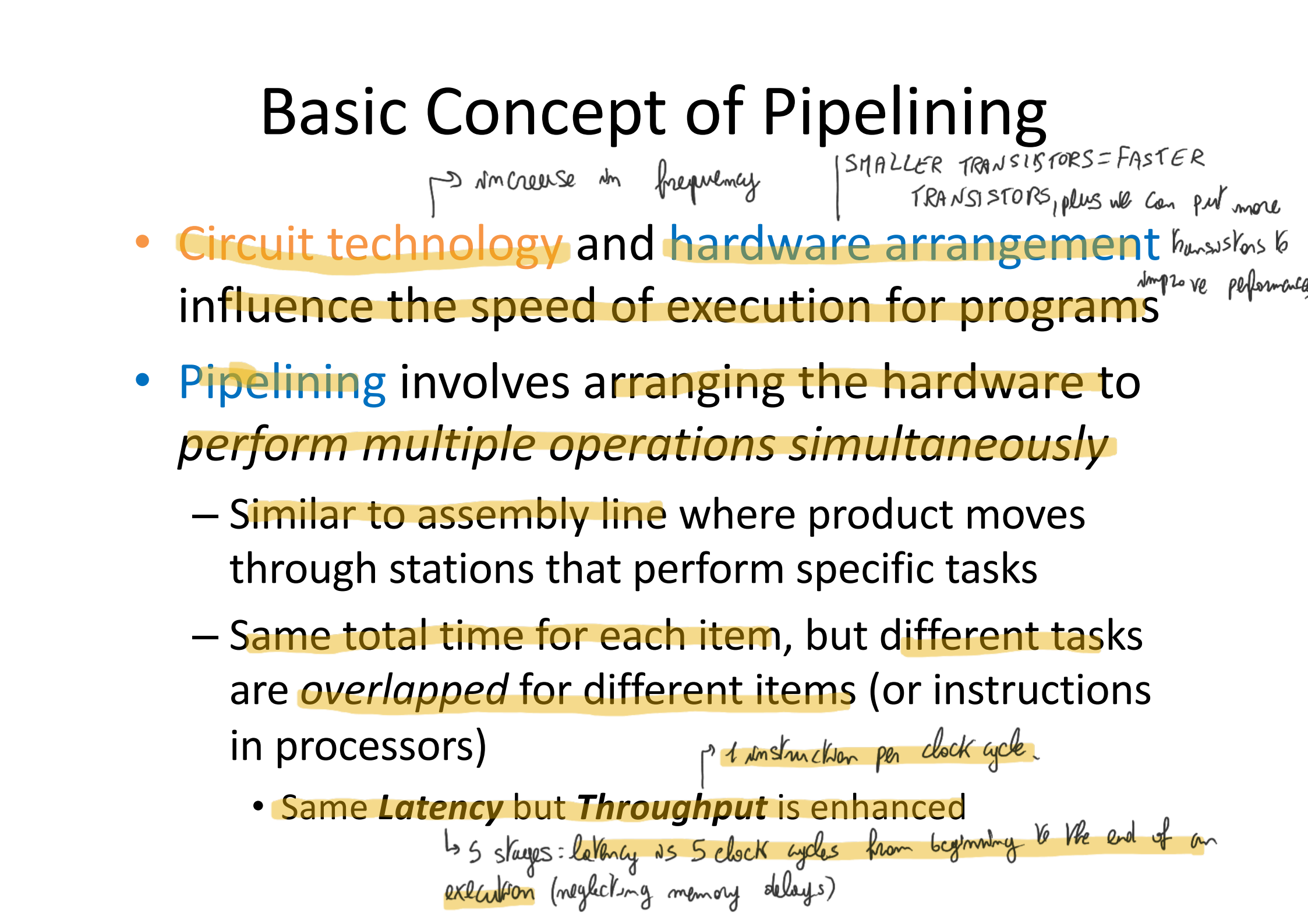
Office hours:

Friday 14-16. Please, contact me in advance before showing up. We can also arrange an appointment remotely on Microsoft Teams.

PIPELINE INTRODUCTION IN THE 5-STAGE CPU ARCHITECTURE

Basic Concept of Pipelining

- Circuit technology and hardware arrangement influence the speed of execution for programs
- Pipelining involves arranging the hardware to *perform multiple operations simultaneously*
  - Similar to assembly line where product moves through stations that perform specific tasks
  - Same total time for each item, but different tasks are *overlapped* for different items (or instructions in processors)
    - Same ***Latency*** but ***Throughput*** is enhanced

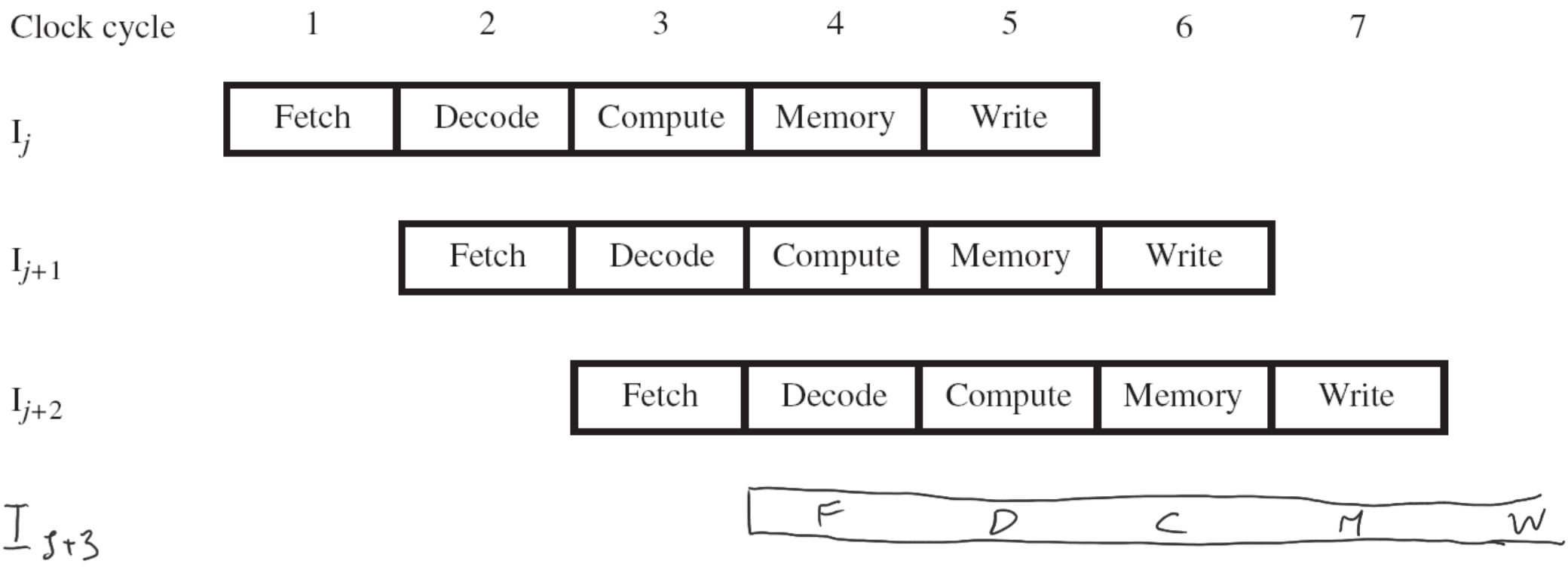


Pipelining in a Computer

- Focus on pipelining of *instruction execution*
- 5-stage organization consists of:
  Fetch, Decode, Compute, Memory, Write
- We've seen how instructions are fetched & executed one at a time with only one stage active in any cycle
- *With pipelining*, multiple stages are active simultaneously for different instructions
- Still 5 cycles to execute, but *rate* reaches 1 instruction per cycle
  - Ideally, one instruction is completed at each cycle

↳ There are some issues that can slow down the pipe: we can get close to 1

“Ideal situation”



Each stage is processing a different instruction.

Pipeline Organization

What do we need to achieve pipelining?

From 1 stage to the following one we need to carry out all the info needed to execute that instruction.

- Use program counter (PC) to fetch instructions
  - A new instruction enters pipeline every cycle
- Carry along instruction-specific information as instructions flow through the different stages
- Use *interstage buffers* to hold this information
  - These buffers incorporate RA, RB, RZ, RM, RY, IR, and PC-Temp registers, we need other bits to also store)
  - The buffers also hold control signal settings

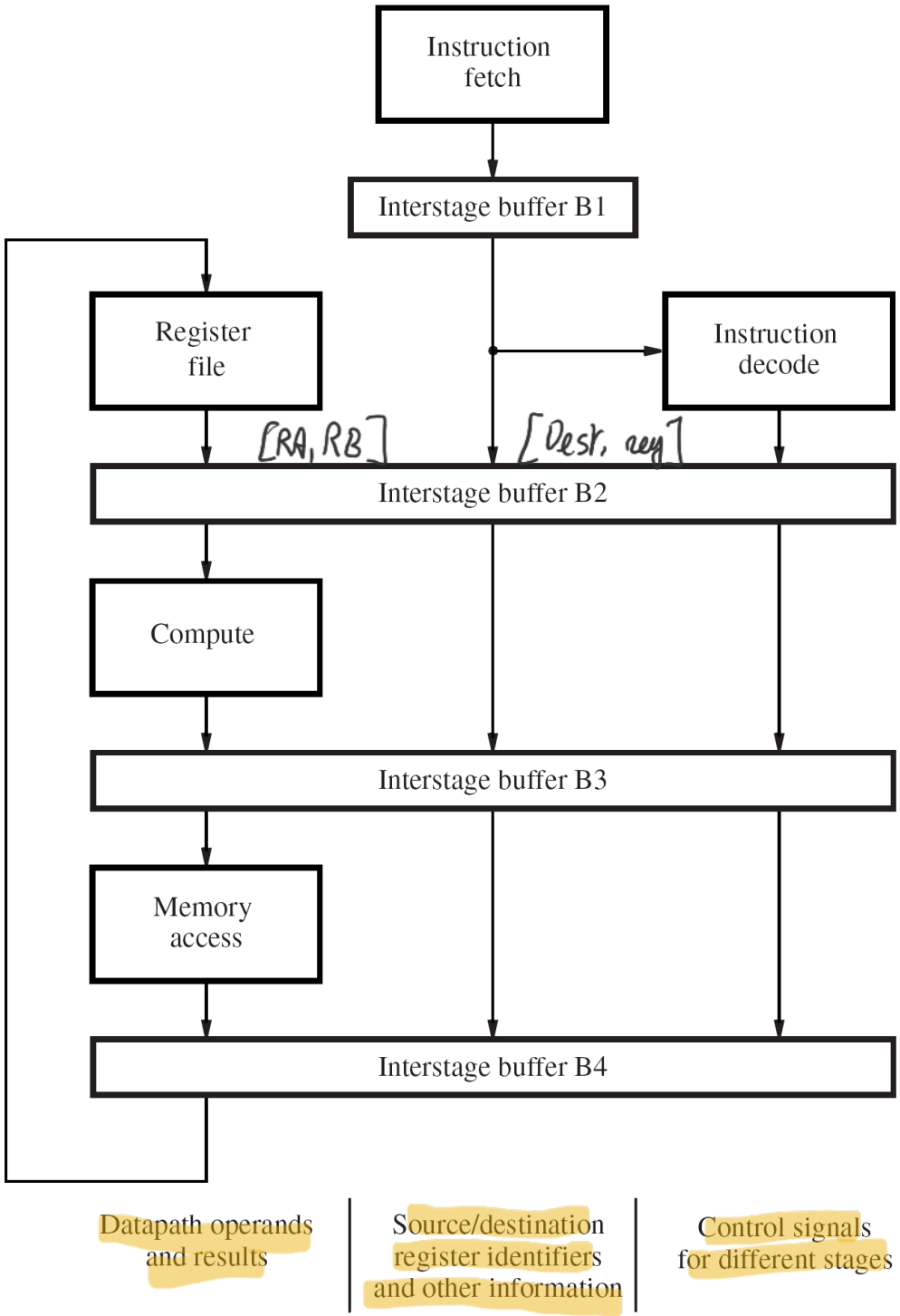
1 Fetch

2 Decode

3 Compute

4 Memory

5 Write



4 intermediate buffers: left hand side, the data path. At every clock cycle the IR changes, what's decoded needs to be carried together with the instr. data.

We need a control circuitry too, though, much more complex than before.

What possible issues there can be?

There are situations in which the pipe has to be blocked though.

1) data dependency

1) Instructions may have as source operands results of the previous instructions.
2) Memory delay for fetch and memory (a miss, for example)
3) During fetch, PC is incremented by 4. This works only in case we do not have instructions that modify the execution of the code! But we have jmps, calls, ret, branch.
EX: branch changes PC in compute, but we already did 2 fetches before that!

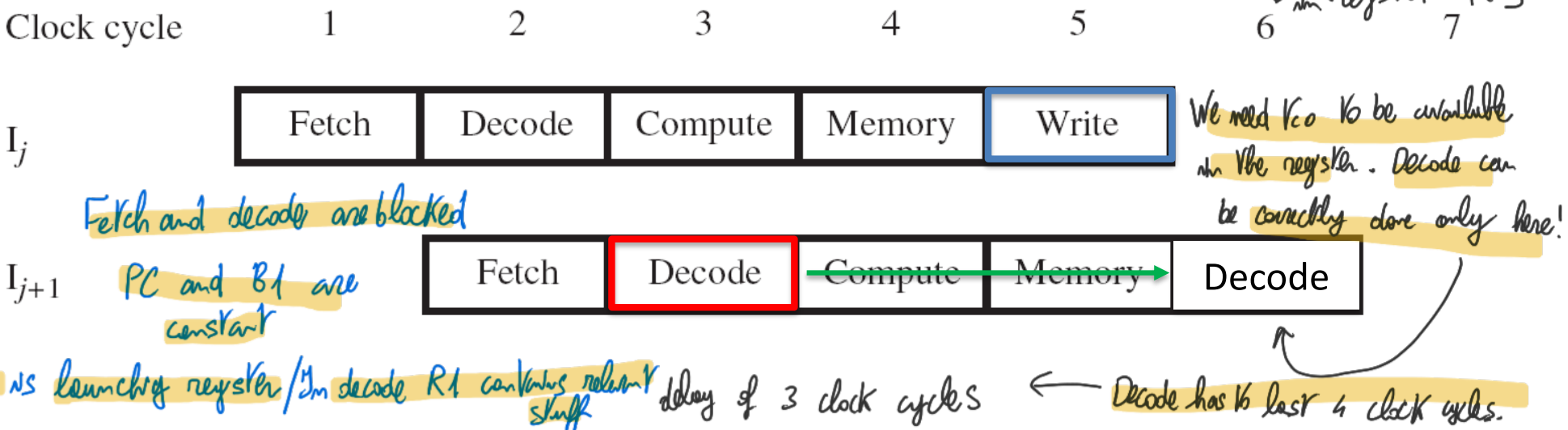
With step 2, we know that we can build a hierarchical structure using caches.

Pipelining Issues

1) Data dependency.
addw R8, R9, 25
subw R9, R8, 24

- Consider two successive instructions I_j and I_{j+1}. Assume that the destination register of I_j matches one of the source registers of I_{j+1}
  - Result of I_j is written to destination at the end of cycle 5
  - But I_{j+1} reads *old* value of register in cycle 3
- Due to pipelining, I_{j+1} **computation is incorrect**. So *stall* (delay) I_{j+1} until I_j writes the new value
- Condition requiring this stall is a ***data hazard***

The result of the instruction is in register R9



Data Dependencies

- Now consider the specific instructions

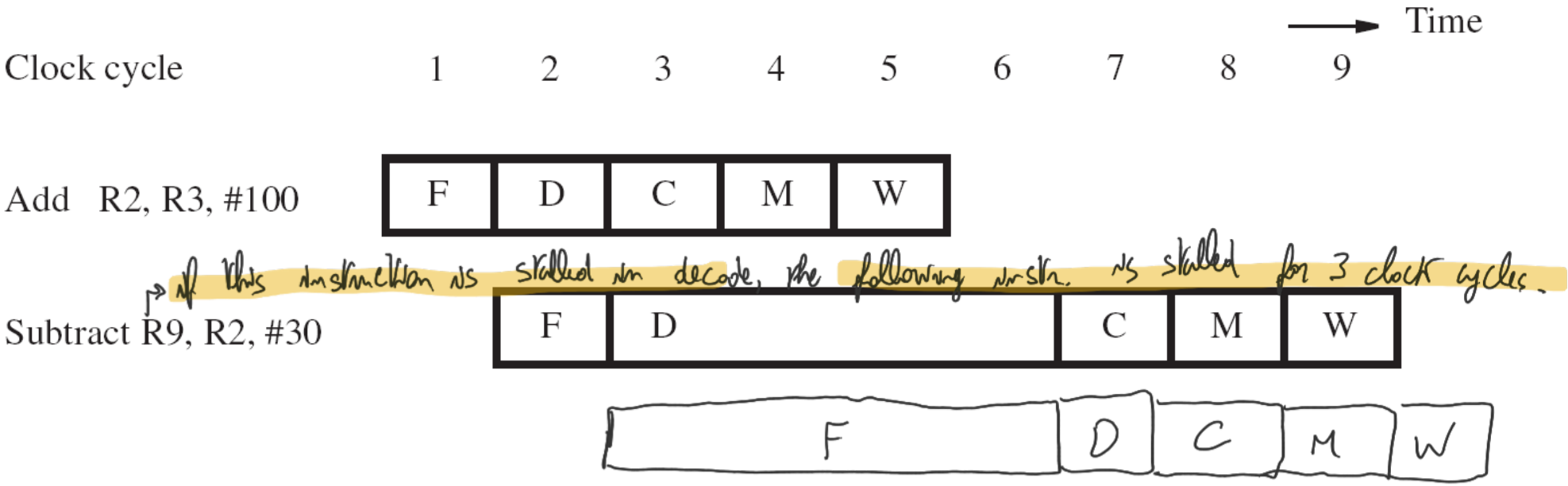
  Add R2, R3, #100

  Subtract R9, R2, #30

- Destination R2 of Add is a source for Subtract
- There is a ***data dependency*** between them because R2 carries data from Add to Subtract
- On *non*-pipelined datapath, result is available in R2 because Add completes before Subtract

Stalling the Pipeline

- With pipelined execution, old value is still in register R2 when Subtract is in Decode stage
- So **stall** Subtract for **3 cycles** in Decode stage
- New value of R2 is then available in cycle 6



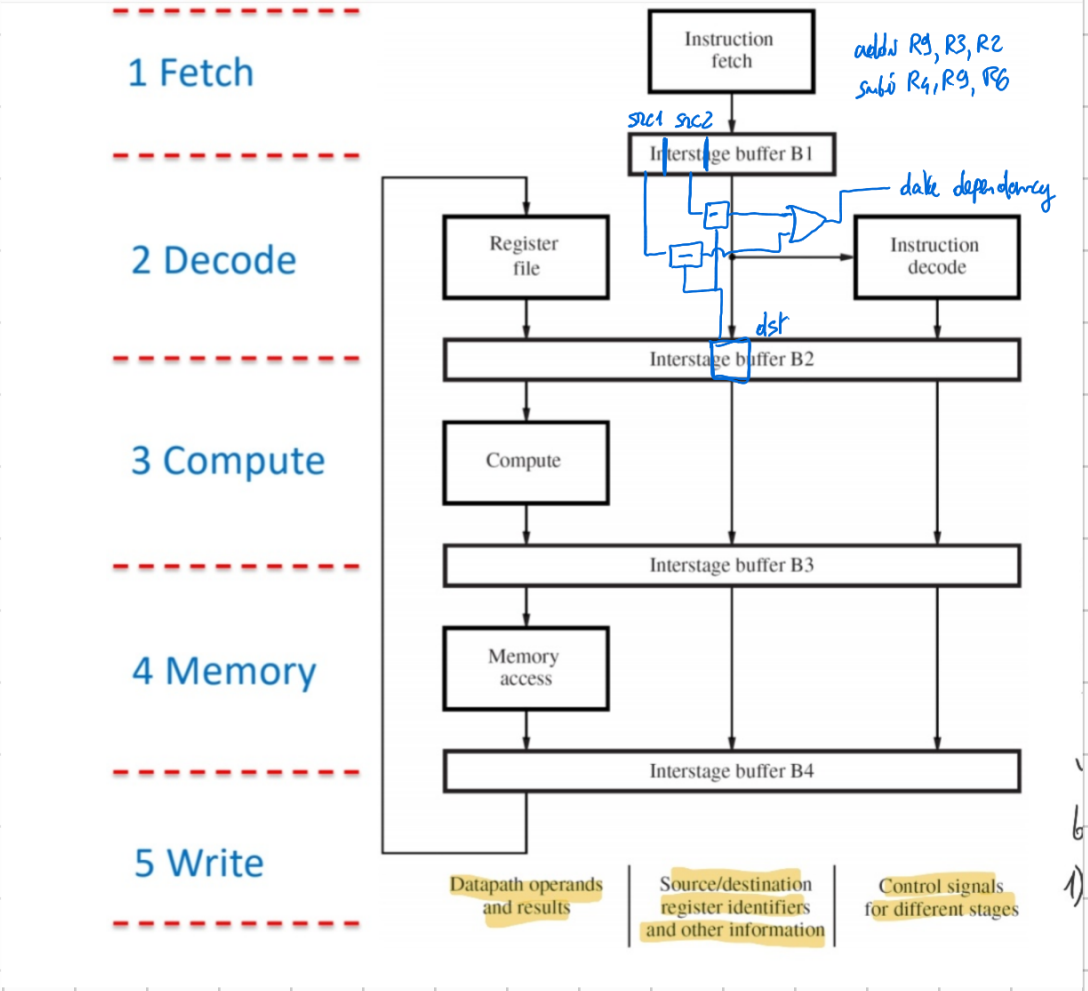
Problem, block the pipe but not all of it! Add bus B to be compared.

Details for Stalling the Pipeline (1)

1st of all detection! We can check the source registers in B1. Are they equal to the destination register in B2? If yes, dependency. But not only B2, also B3, B4!

- Control circuitry must recognize dependency while Subtract is being decoded in cycle 3
- Interstage buffers carry register identifiers for source(s) and destination of instructions
- In cycle 3, compare destination identifier in Compute stage against source(s) in Decode
- R2 matches, so Subtract kept in Decode while Add is allowed to continue normally

NOTE: we consider checking B3, B4 at least unlikely, the compiler can help with this.

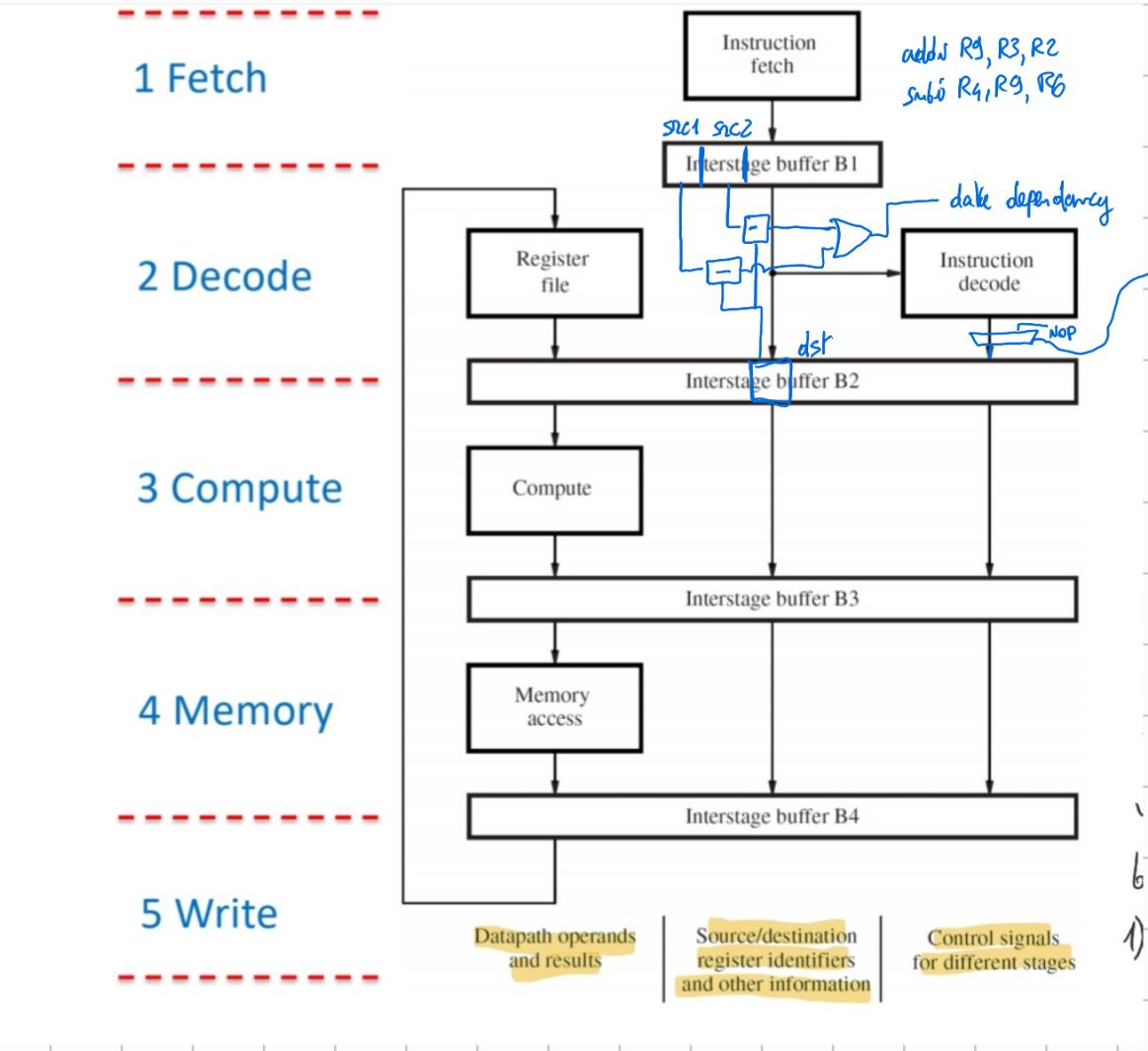


Details for Stalling the Pipeline (2)

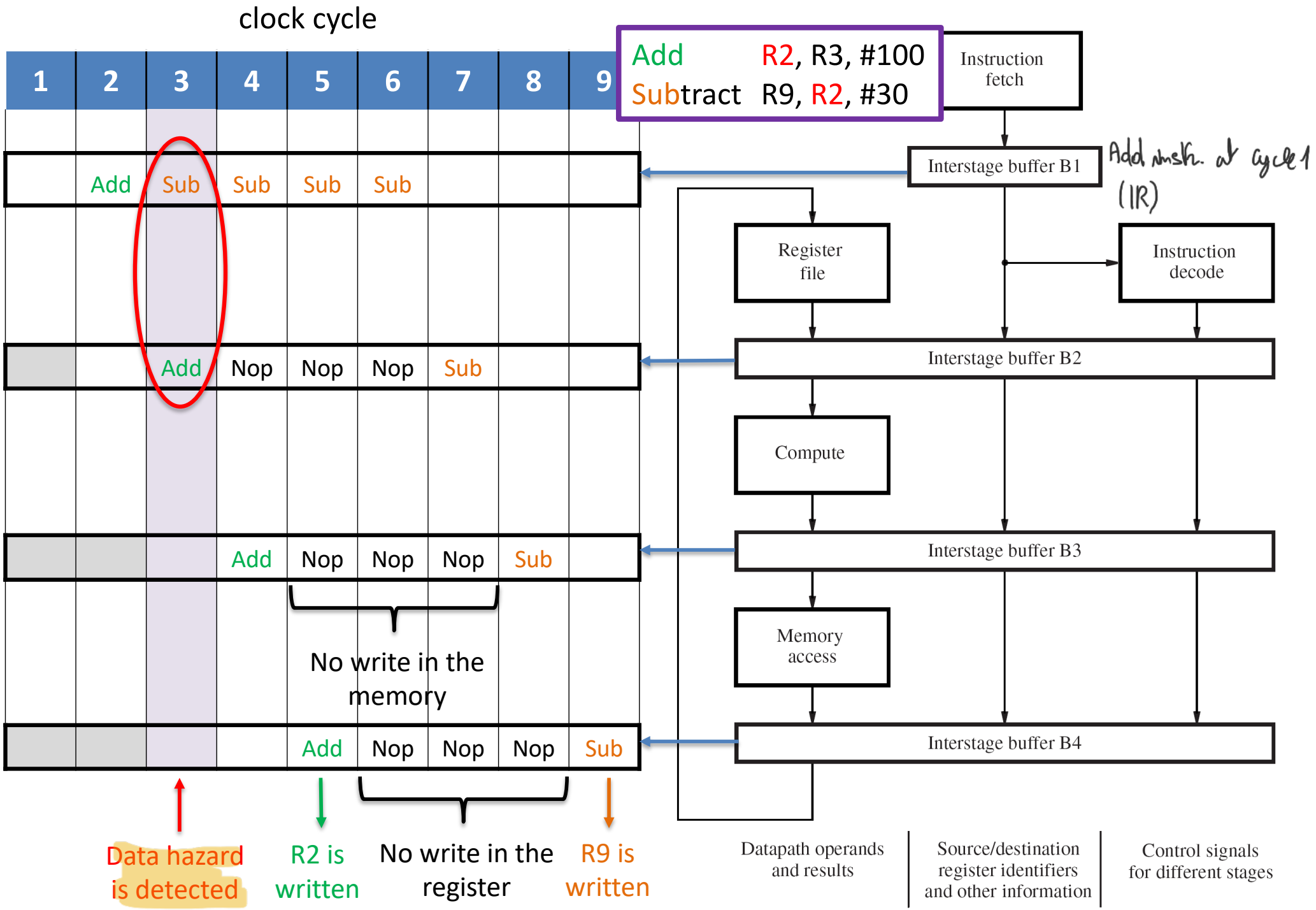
What goes into compute? Subs is blocked, but something will be stored in the buffers! We use NOPs.

- Stall the Subtract instruction for 3 cycles by holding interstage buffer B1 contents steady
- But what happens after Add leaves Compute?
- Control signals are set in cycles 3 to 5 to create an *implicit* NOP (No-operation) in Compute
- NOP control signals in interstage buffer B2 create a cycle of idle time in each later stage
- The idle time from each NOP is called a *bubble*

To address this we need to add something in case of data dependency:



In case we have data dependency, we have a multiplexer that selects a NOP in case of data dependency.


clock cycle
1
2
3
4
5
6
7
8
9
Add R2, R3, #100
Subtract R9, R2, #30
Add Sub Sub Sub Sub
Add Nop Nop Nop Sub
Add Nop Nop Nop Sub
Add Nop Nop Nop Sub
No write in the memory
Instruction fetch
Interstage buffer B1
Add instr. at cycle 1 (IR)
Register file
Instruction decode
Interstage buffer B2
Compute
Interstage buffer B3
Memory access
Interstage buffer B4
Data hazard is detected
R2 is written
No write in the register
R9 is written
Datapath operands and results
Source/destination register identifiers and other information
Control signals for different stages

At cycle 3: sub is in B1, add in B2. We can check here the destination register of add and source registers of sub. In case yes, data dependence is detected. The control circuitry has to block sub in B1. So the program counter cannot save the next instruction because B1 is blocked.

On B2 at clock cycle 4 a NOP is introduced too. At clock cycle 5 the first nop moves to memory. At clock cycle 6, the result of the add in R2.

At 4 a nop is introduced. In cycle 5 the info is saved. At clock cycle 6 we can continue the sub.

Every time you have data dependency, you are wasting 3 clock cycles.

To avoid a data hazard, what can the compiler do? It can manually place 3 NOP instruction to avoid data dependencies.

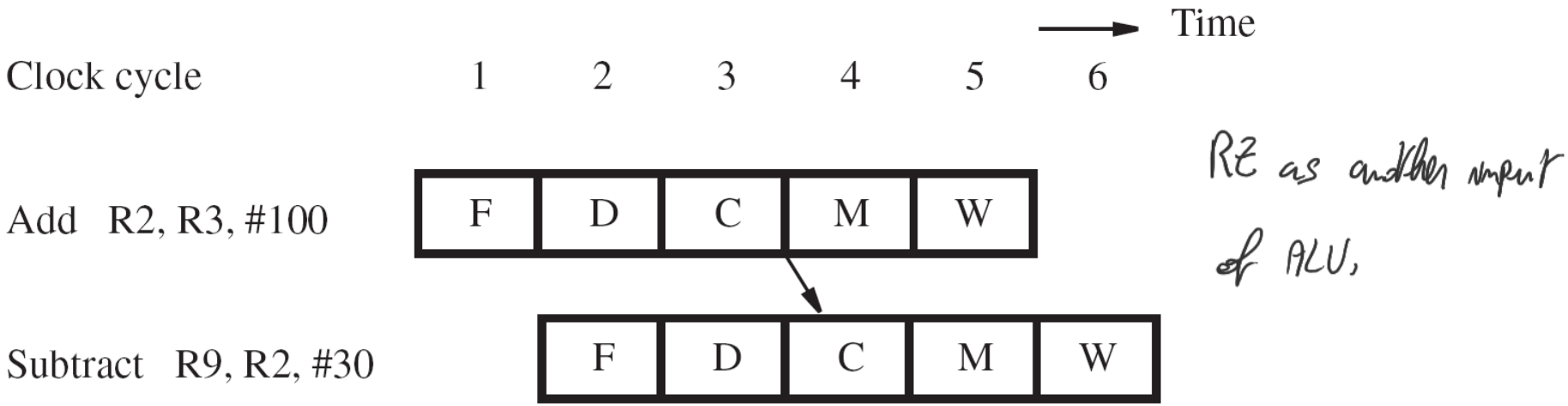
Every time there's a data dependency, 3 NOPs are inserted, then the optimizer takes a nearby instr. with no data dependency and places it instead of a NOP.

In general, HW solution loses 3 clock cycles in case of data dependencies.

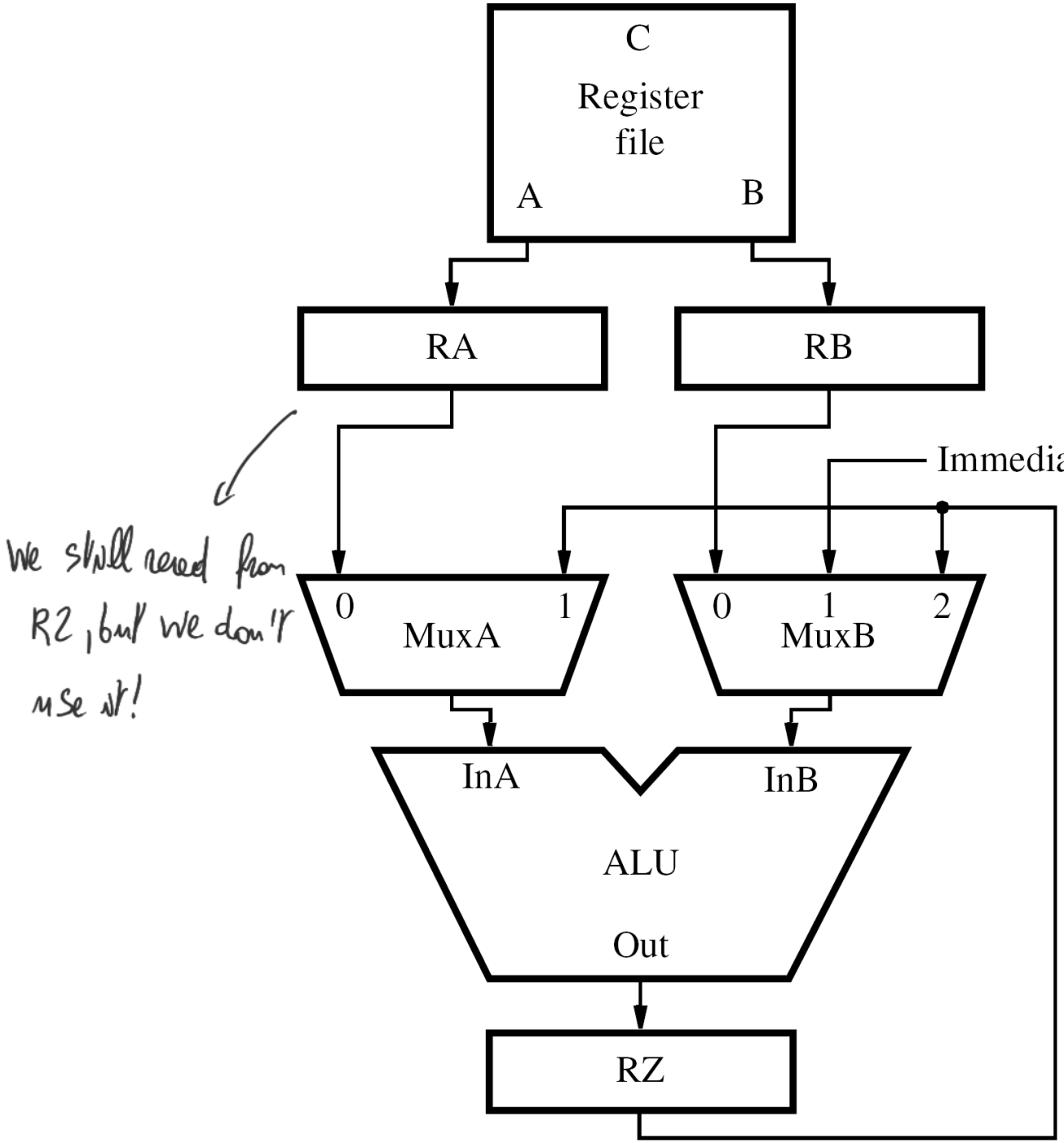
But the result of the adder is ready before! At clock cycle 4 we have R2 as the result of the op.

Operand Forwarding

- Operand forwarding handles dependencies without the penalty of stalling the pipeline
- For the preceding sequence of instructions, new value for R2 is available at end of cycle 3
- *Forward* value to where it is needed in cycle 4



R2 as another input of ALU,



We still need from R2, but we don't use it!

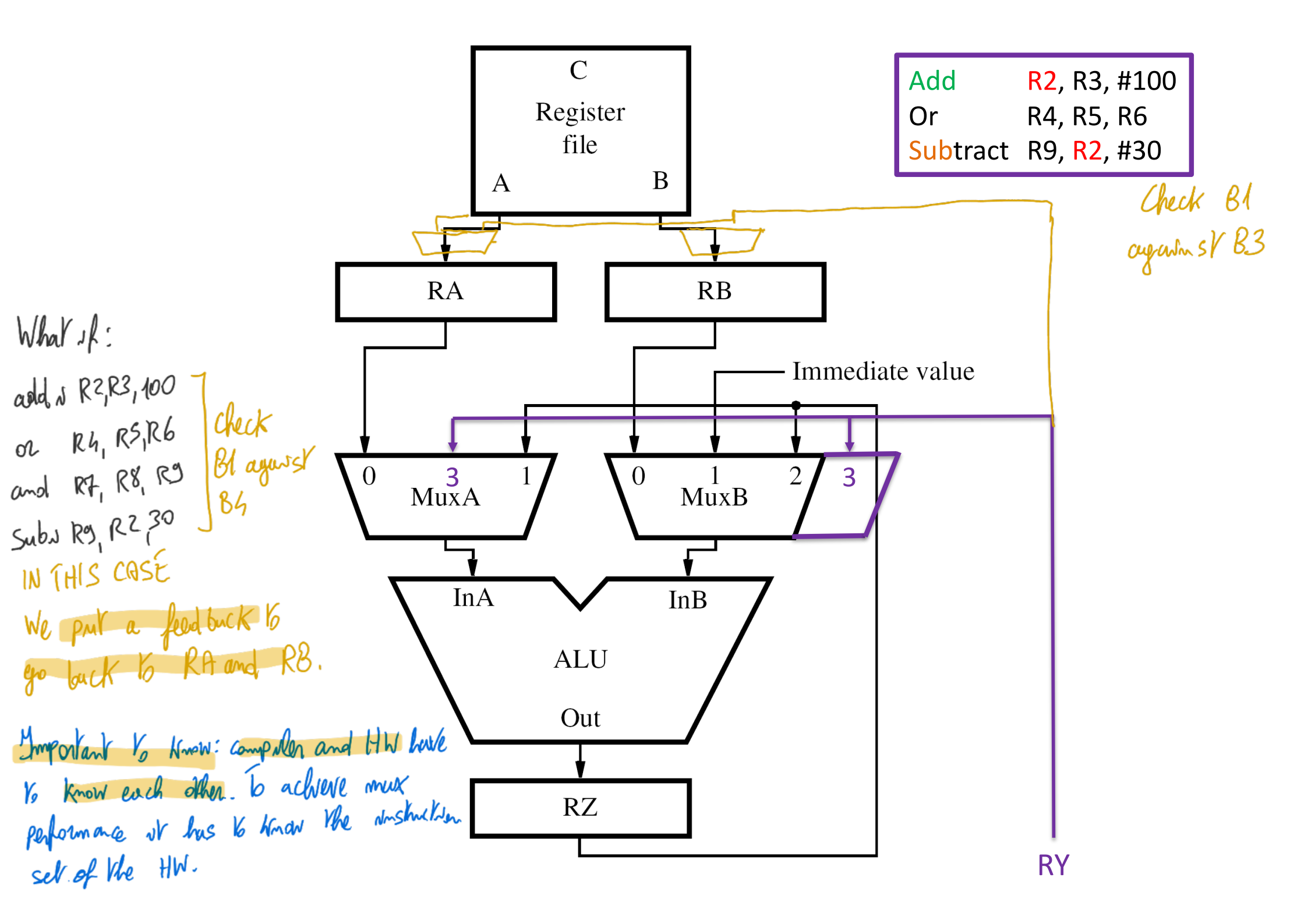
Add	R2, R3, #100
Subtract	R9, R2, #30

PREFERRED SOLUTION, but we could directly forward ALU to another point before RA/RB. From a time point of view, propagation delay can be longer.

What happens if instr. are NOT consecutive?

E.g.

Add	R2, R3, #100
Or	R4, R5, R6
Subtract	R9, R2, #30

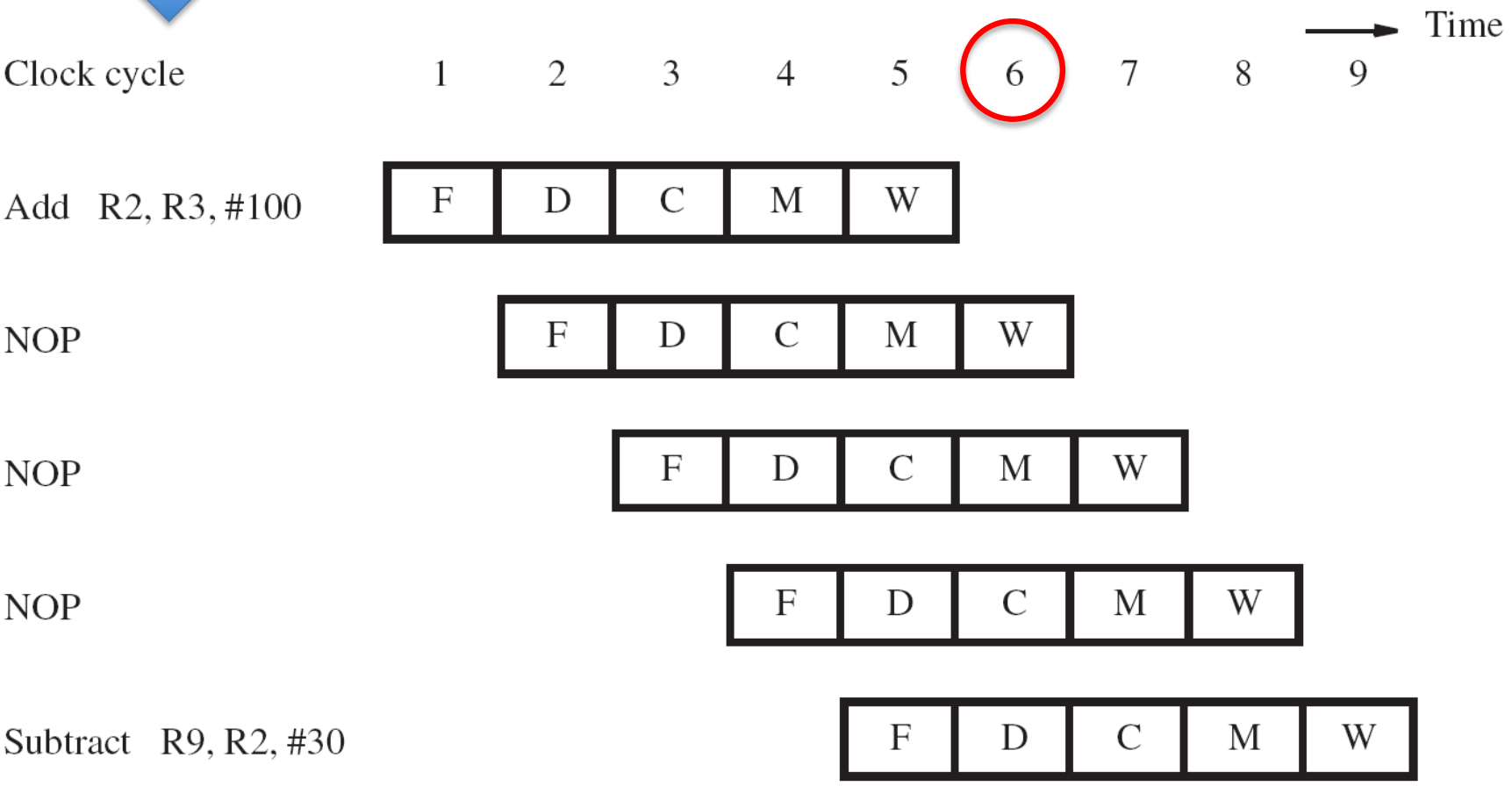

C
Register
file
A
B
RA
RB
Immediate value
0
3
1
MuxA
0
1
2
3
MuxB
InA
InB
ALU
Out
RZ
RY
Add R2, R3, #100
Or R4, R5, R6
Subtract R9, R2, #30
Check B1 against B3
What if:
add, R2, R3, 100
or R4, R5, R6
and R7, R8, R9
Subv R9, R2, 30
Check B1 against B4
IN THIS CASE
We put a feedback to go back to RA and RB.
Important to know: compiler and HW have to know each other. To achieve max performance it has to know the instruction set of the HW.

Software Handling of Dependencies

- Compiler can generate **& analyze** instructions
- Data dependencies are evident from registers
- Compiler puts **three *explicit* NOP** instructions between instructions having a dependency
- Delay ensures new value available in register but causes total execution time to increase
- Compiler can *optimize* by moving instructions into NOP slots (if data dependencies permit)
  - Leading to a smaller code size and shorter execution time

```
Add       R2, R3, #100
Subtract  R9, R2, #30
```

Compiler

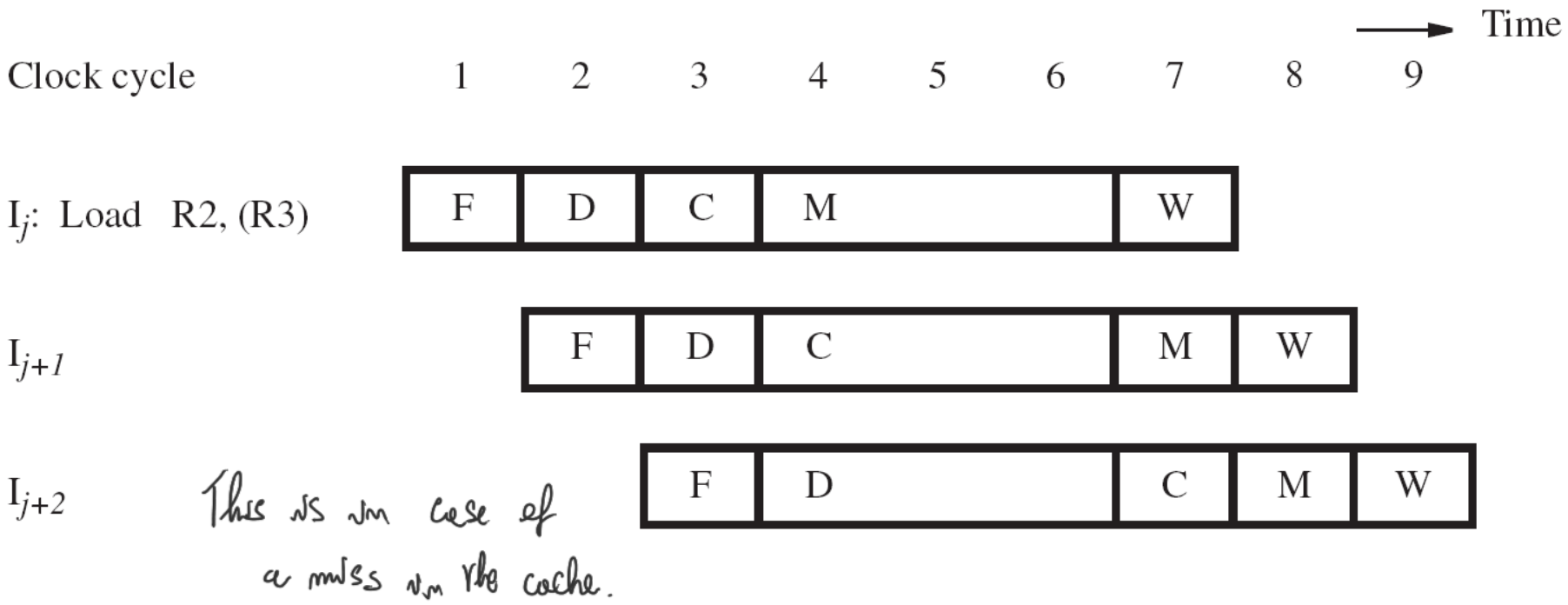


Memory Delays (1)

- Memory delays can also cause pipeline stalls
- A cache memory holds instructions and data from the main memory and is faster to access
- With cache, typical access time is one cycle
- But a cache *miss* requires accessing slower main memory with a much longer delay
- In pipeline, memory delay for one instruction causes subsequent instructions to be delayed

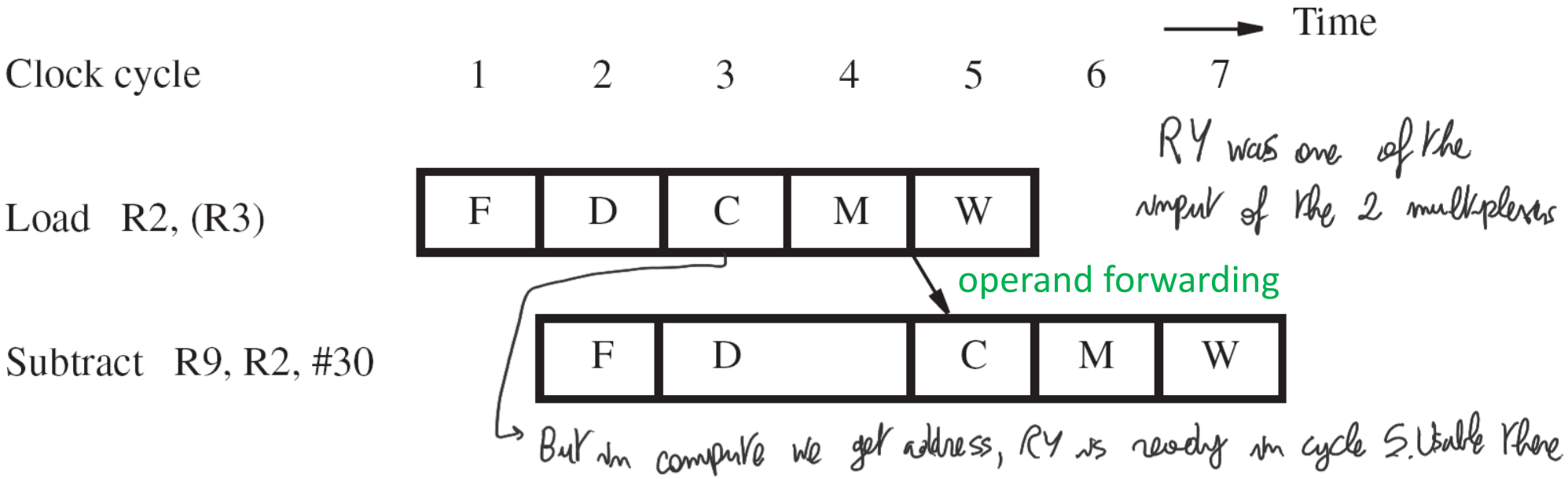
Memory Delays (2)

- Example considering a **data** memory access time of 3 clock cycles



Memory Delays (3)

- Even with a cache *hit*, a Load instruction may cause a short delay due to a data dependency
- One-cycle stall required for correct value to be forwarded to instruction needing that value
- Optimize with useful instruction to fill delay

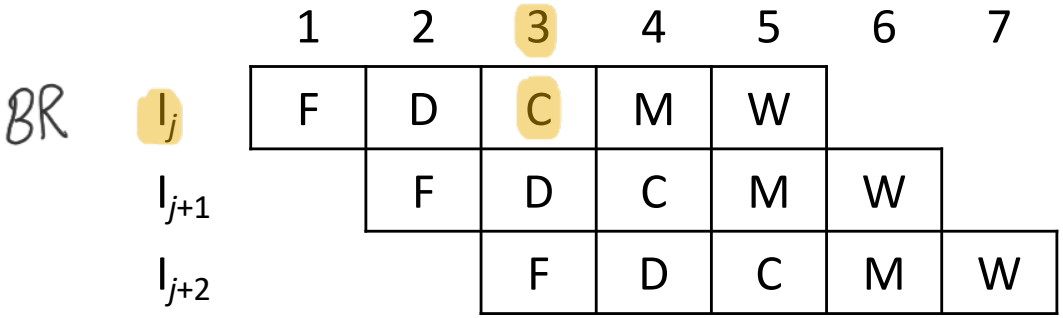


At clock cycle 4 what is ready is an effective address: Rz contains effective address; Ry contains the data read in cycle 5 only. So this data dependancy with operand forwarding technique only reduces the delay to 1 clock cycle.

In case of a MISS, all the block will need to be transfered to the cache. Might take a while.

Branch Delays

- Ideal pipelining: fetch each new instruction while the previous instruction is being decoded



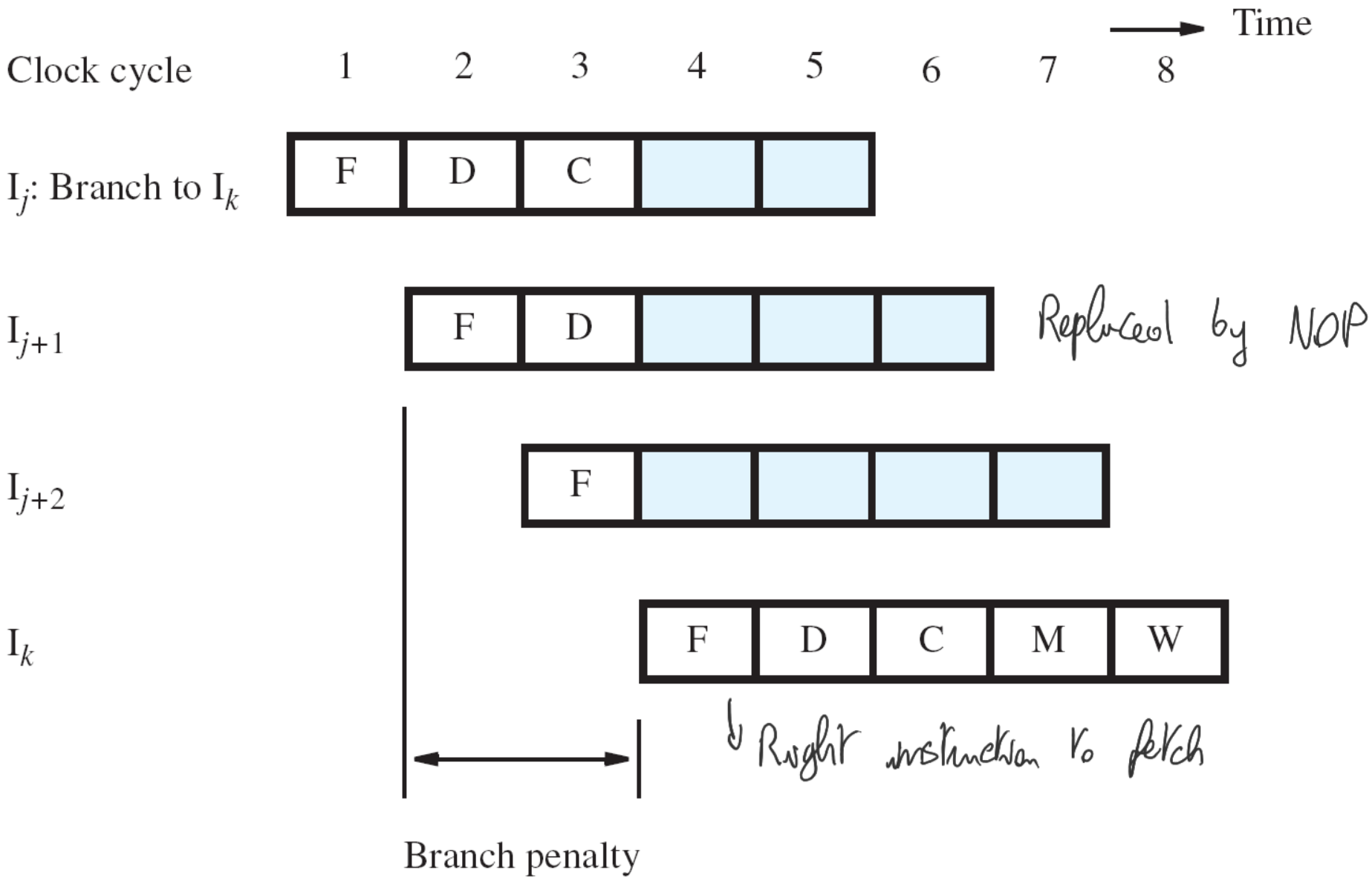
- Branch instructions may alter execution sequence, but they must be processed to know the effect
- Any delay for determining branch outcome may lead to an increase in total execution time
- Techniques to mitigate this effect are desired
- Understand branch behavior to find solutions

Discarding instr: instead of moving on the instr., a NOP is replaced.
Control signals of I_{J+1}, I_{J+2} are replaced by NOP instruction: Bubble = control signal related to No Action

Unconditional Branches (1)

- Consider instructions I_j, I_{j+1}, I_{j+2} in sequence
  - I_j is an unconditional branch with target I_k
- The *target address*, calculated using **offset** (which is available after the Decode stage) and [PC]+4, is known after the Compute stage
- In pipeline, *target* I_k is known for I_j in cycle 4, but instructions I_{j+1} , I_{j+2} fetched in cycles 2 & 3
- *Target* I_k should have followed I_j immediately, so discard I_{j+1} , I_{j+2} and incur into a two-cycle *penalty*

Unconditional Branches (2)



Effect of branch penalty

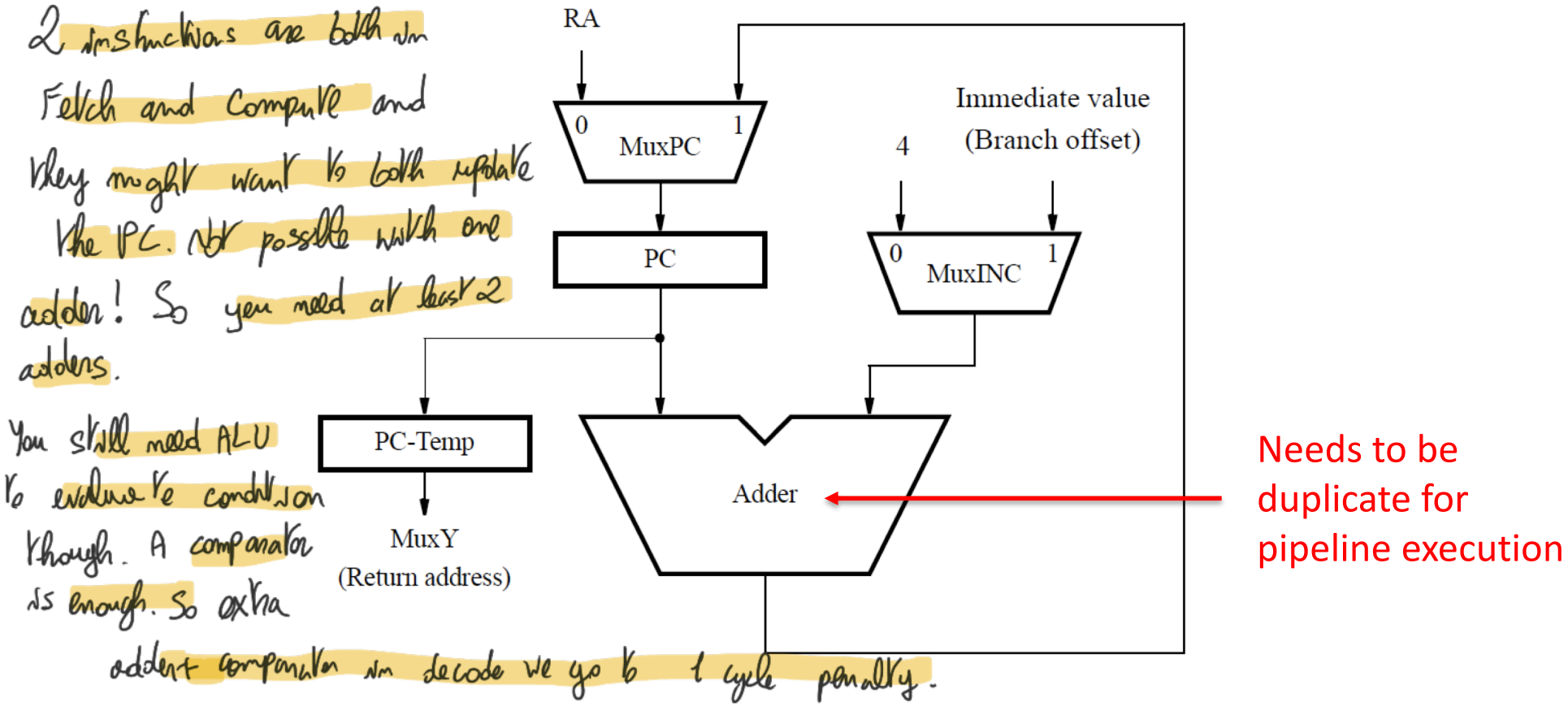
- Ideally, 1 instr. per *cycle* w/ pipelining
- Branch instr. are executed frequently
  - Roughly p_{BR} = 20 % of the instr. executed by the processor (may be significantly larger than the number of branch instr. in the code, because of loops)
  - A 2-cycle branch penalty increases the average **number of cycles *S* per instr.** by 40 %
    - $S = (1 - p_{BR}) \times 1 + p_{BR} \times (1 + 2) = 1 + 2 \times p_{BR} = 1.4$
    - $S = 1 + \delta_{BR}$*, $\delta_{BR} = 2 \times p_{BR} = 0.4$
  - Things are a little bit better, as not all the conditional branches will be taken

* δ_{BR} = degradation of performance

So, if offset is positive, we make one kind of prediction. Otherwise another kind.

Reducing the Branch Penalty (1)

- In pipeline, adder for PC in the Instruction Address Generator block is used every cycle, so it cannot calculate the branch target address



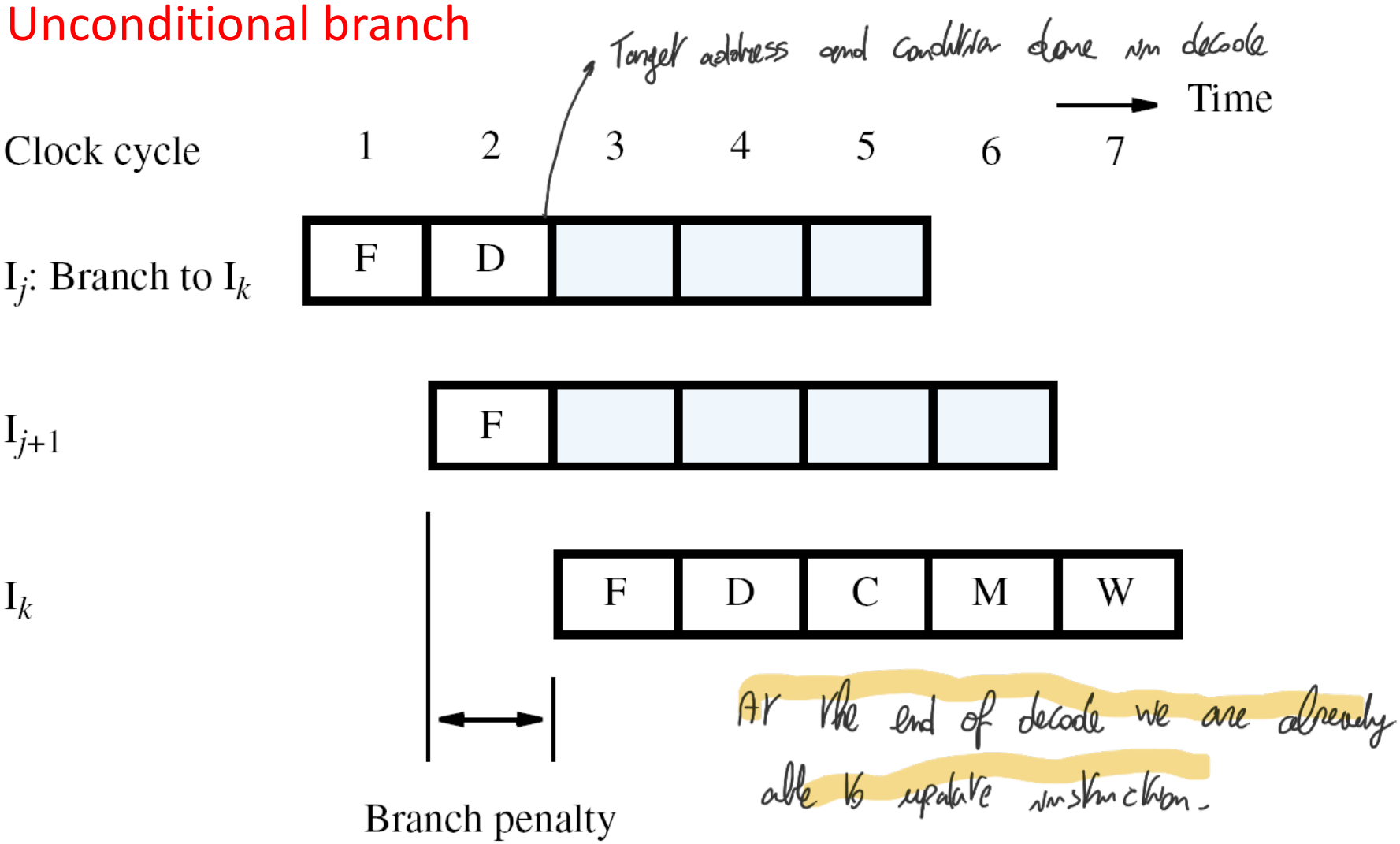
Reducing the Branch Penalty (2)

- In pipeline, adder for PC in the Instruction Address Generator block is used every cycle, so it cannot calculate the branch target address
- So introduce a second adder just for branches and place this second adder in the Decode stage to enable earlier determination of target address
- For previous example, now only I_{j+1} is fetched
- Only one instruction needs to be discarded
- The branch penalty for **UNCONDITIONAL** branches is reduced to one cycle

We don't have symmetrical behavior: if condition is false, no problem.

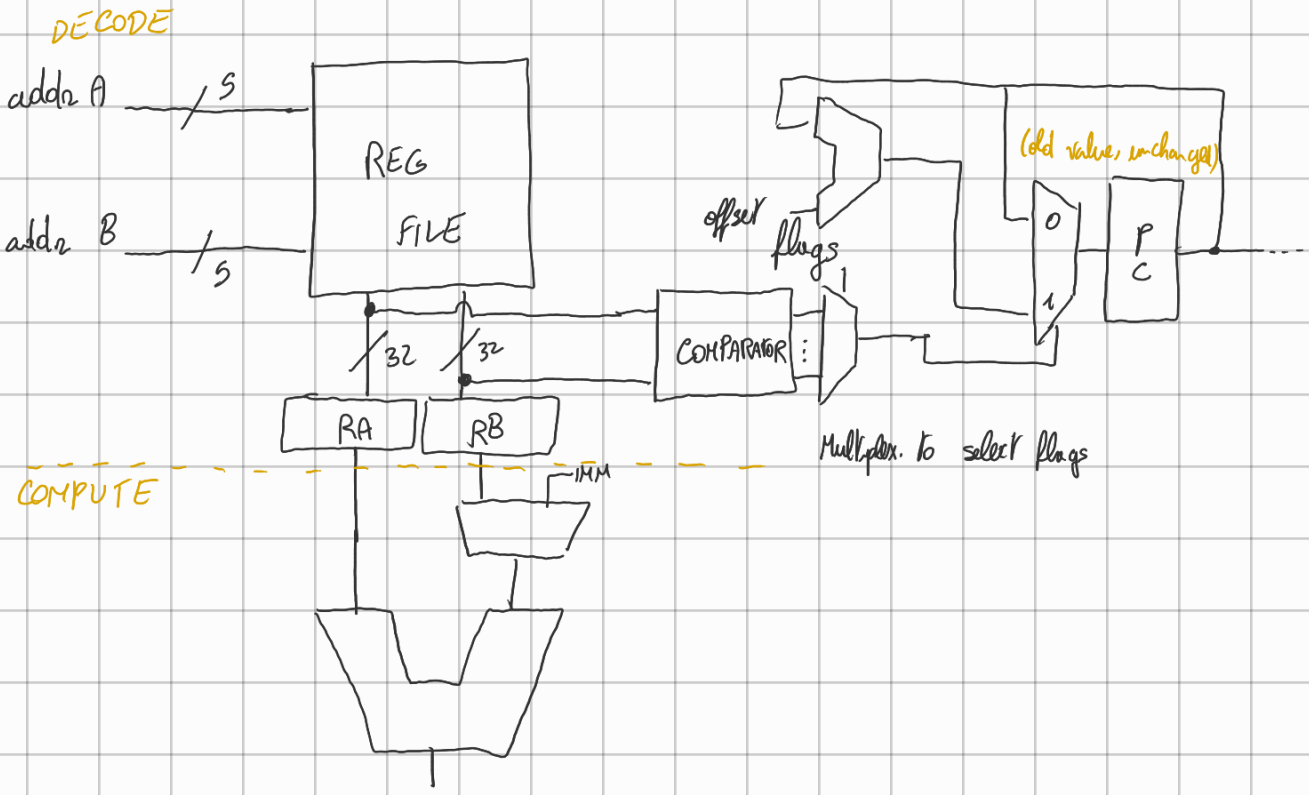
Reducing the Branch Penalty (3)

Unconditional branch



Conditional Branches

- Consider a conditional branch instruction:
  - Branch_if_[R5]=[R6] LOOP
- Requires not only target address calculation, but also requires comparison for condition
- In the 5-stage architecture, ALU performs the comparison
- Target address now calculated in Decode stage
- To maintain one-cycle penalty, a comparator just for branches must be inserted in Decode stage
  - Higher hardware complexity


DECODE
addr A
5
addr B
5
REG
FILE
32
32
RA
RB
IMM
COMPUTE
COMPARATOR
offset
flags
1
0
1
(old value, unchanged)
PC
Multiplex. to select flags

This is alternative to branch prediction.

The Branch Delay Slot (1)

- Let both branch decision and target address be determined in Decode stage of pipeline
- Instruction immediately following a branch is always fetched, regardless of branch decision
- That next instruction is discarded with penalty, except when conditional branch is not taken
  - Non-uniform behaviour between the two cases
- The location immediately following the branch is called the *branch delay slot*

Trivial solution: place a NOP after every branch.
Compiler could take instructions before the branch and put it after it, but avoid data dependency

If compiler can place it after the branch, advantage.

1) Start from HW arrangement in decode. Then we organise HW in order not to discard instr. after the branch. Every fetch is executed. Complexity moved from HW to SW (compiler): first compiler adds a NOP after every branch, then we try to find instruction to substitute.

Here, NOP is always there and if the optimizer can optimize, good. Before, if condition was not met, no loosing of clock cycle and worst case we lose it.

The Branch Delay Slot (2)

- Instead of conditionally discarding instruction in delay slot, ***always*** let it complete execution
- Let compiler find an instruction *before* branch to move into slot, if data dependencies permit
  - Called ***delayed branching*** due to reordering
- If useful instruction put in slot, penalty is *zero*
- If not possible, insert explicit NOP in delay slot for one-cycle penalty, whether or not taken
- Benefits of delayed branching depends on the possibility for the compiler to fill the branch delay slot (this happens for more than 70 % of the cases in many programs)

=> after branch, you execute either NOP or useful instr. and then the longer instr

• COND. and TARGET ADDRESS COMPUTED IN DECODE STAGE

① Discard next instr. when branch taken

$\delta_{BR}^{①} = 1 \times p_{BR} \times p_{BRTAKEN}$

1 = delay; p_{BR} = frequency; $p_{BRTAKEN}$ = probability br. taken

② Delayed branching

$\delta_{BR}^{②} = 1 \times p_{BR} \times p_{BR-NOP}$

1 = delay; p_{BR-NOP} = branch followed by nop.

We need to compare $p_{BRTAKEN}$ and p_{BR-NOP}

Branches come from loops and conditional statements. In case of loops, the $p_{BRTAKEN}$ is quite high for loops.

If the compiler is good enough to make p_{BR-NOP}, delayed branches may outperform the first solution. Plus, sol. 2 is simpler!

For branches coming from loops, ② could be a good solution.

	Add	R7, R8, R9
	Branch_if_[R3]=0	TARGET
	I_{j+1}	
	⋮	
TARGET:	I_k	

(a) Original sequence of instructions containing a conditional branch instruction

	Branch_if_[R3]=0	TARGET
	Add	R7, R8, R9
	I_{j+1}	
	⋮	
TARGET:	I_k	

(b) Placing the Add instruction in the branch delay slot where it is always executed

Branch Prediction

Assume we are at ①.
We want to reduce ② CBR penalty. Approach: try to fetch right instruction

- A branch is decided in Decode stage (cycle 2) while following instruction is *always* fetched
- Following instruction may require discarding (or with delayed branching, it may be a NOP)
- Instead of discarding the *following* instruction, can we anticipate the ***actual*** next instruction?
- Two aims: (a) *predict* the branch decision
  (b) use prediction *earlier* in cycle 1

↳ But you need this at fetch level! So in HW you need a branch target buffer that recalls where branches are in the memory.

So you update Program counter before knowing real the instruction!

1. Implementing predictor algorithm 2. Implement predictor in fetch phase

Static: for a given branch we always make same prediction.

Dynamic: depends on the previous result

Static Branch Prediction

- Simplest approach: assume branch *not* taken
  - Penalty if prediction disproved during Decode
- If branches are "random", accuracy is 50%
- **But a branch at end of a loop is usually taken**
  - So for **backward** branch, always predict *taken*
  - Instead, always predict *not taken* for **forward** branch
- Expect higher accuracy for this special case, but what about accuracy for other branches?
  - For the last iteration of loops, the static prediction is wrong

DO-WHILE

WHILE

positive offset

Branch Taken: We Jump out

I expect higher accuracies for loop, but what about condition?

STATIC can work quite well for loop.

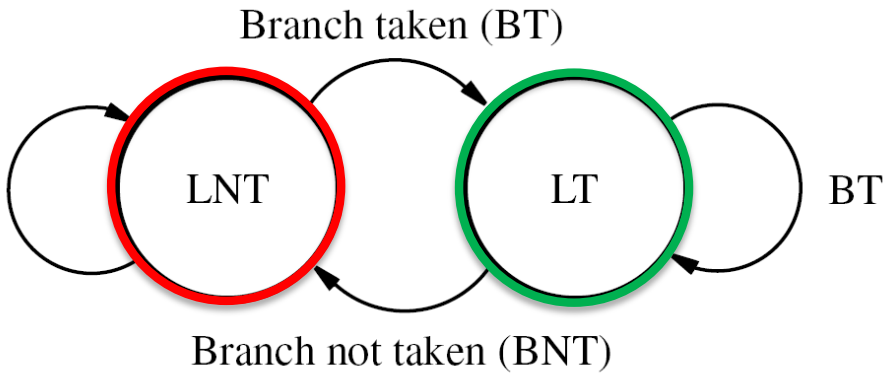
Static prediction assumes loop is never exited. So prediction is correct until loop ends.

Dynamic Branch Prediction

every branch needs its own state machine

- Idea: track branch decisions during execution for *dynamic* prediction to improve accuracy
- Simplest approach: use most recent outcome for likely taken (LT) or likely not-taken (LNT)
- For branch at end of loop, we mispredict in last pass, and in first pass if loop is *re-entered*
- Avoid misprediction for loop re-entry with four states (ST, LT, LNT, SNT) for strongly/likely
- Must be wrong *twice* to change prediction

Last iteration and first iteration of the same loop when reentered.

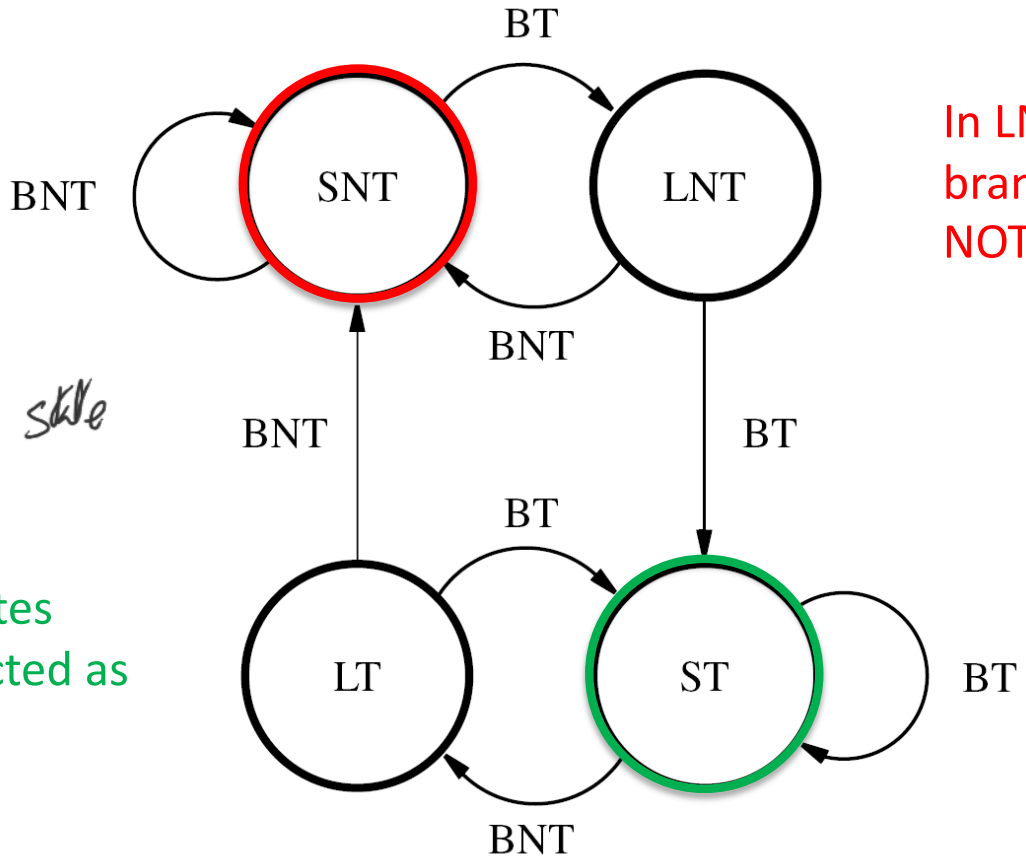


(a) A 2-state algorithm

MOORE MODEL

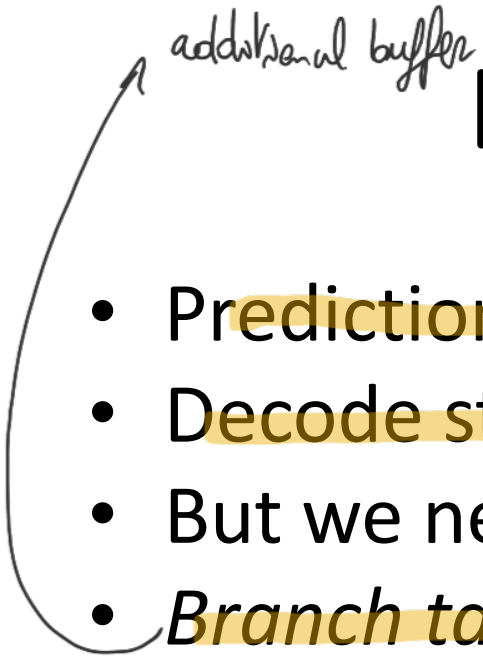
Prediction depends only on state

In LT and ST states branch is predicted as TAKEN



In LNT and SNT states branch is predicted NOT TAKEN

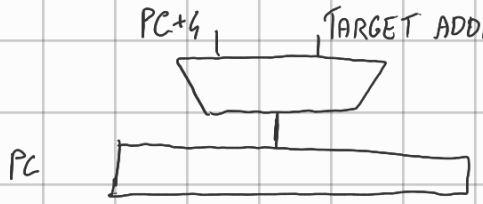
(b) A 4-state algorithm



Branch Target Buffer

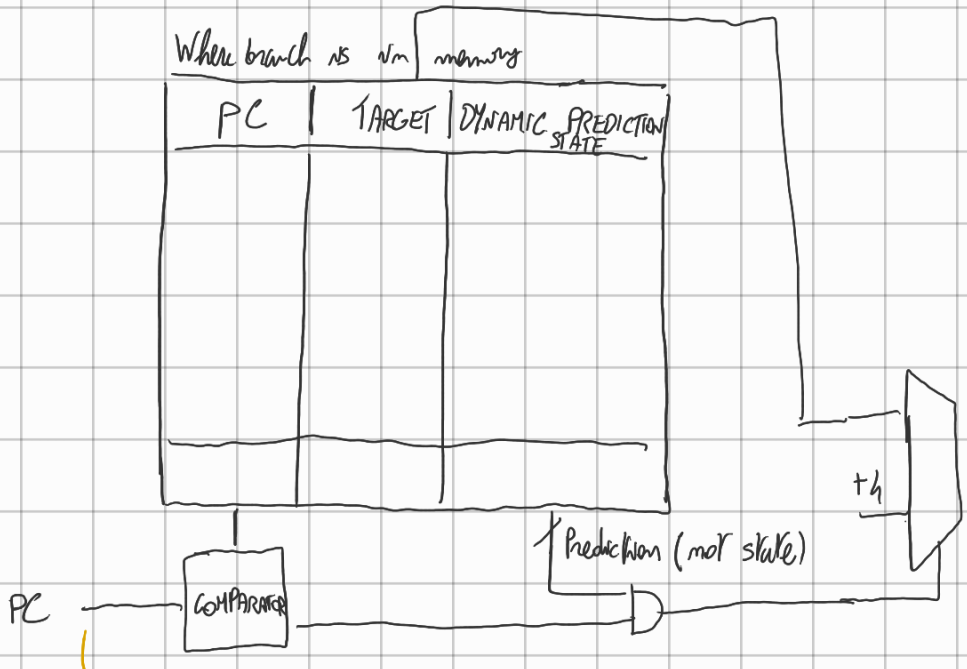
- Prediction only provides a presumed *decision*
- Decode stage computes *target* in cycle 2
- But we need target (and prediction) in cycle 1
- *Branch target buffer* stores target address and history from *last* execution of each branch. Each element of the buffer contains: **the address of the branch instr., the state of the branch prediction alg and the branch target address**
- In cycle 1, use branch instruction address to look up target and use history for prediction
- Fetch in cycle 2 using prediction; if mispredict detected during Decode, correct it in cycle 3

BRANCH TARGET BUFFER, placed in the FETCH stage



In fetch we already need to make a prediction.

So, BRANCH TARGET BUFFER:



Is there a branch? If yes, and pred. is true

You don't know if all branches can stay in the buffer. Every time a branch is executed, a row in the buffer is added.

What if prediction is wrong? Decode stage computes condition. So decision is used to update state machine. If full, buffer has oldest row replaced.

Again, implement is not huge, is it really worth it?

WITH THIS, IN CASE OF CALL, JMP, RET, no problems.

Performance Evaluation

- For a **non-pipelined proc**., the instr. throughput (#instr. per second) is:
  - $F_{np} = R/S$, where R is the clock freq. and S is the average number of cycles to execute one instr. For the 5-stage architecture, $S = 5$ assuming that every memory access can be performed in one cycle (no cache misses).
- For a **pipelined proc.**, throughput F is increased by instr. execution overlapping. Ideally, S can be reduced to 1 (one instr. per cycle). This implies no pipeline stalls.
  - How can we quantify the effect of pipeline stalls and branch penalties on achieved instr. throughput?

Effect of Stalls

- Let's consider a proc. w/ *operand forwarding* (in hardware). Stalls occur when data dependency is related to a Load instr., which causes a 1-cycle stall
  - E.g. if freq. of Load instr. p_{LD} = 25 %, freq. of data dependency after a Load $p_{LD\text{-}dep}$ = 40 %, then throughput F is reduced to:
    - $F = R/(1 + \delta_{stall})$ where $\delta_{stall} = 1 x p_{LD}^{*} x p_{LD\text{-}dep} = 0.1$
    - Thus, $F = R/1.1 = 0.91R$
  - The compiler can improve performance by trying to reduce $p_{LD\text{-}dep}$

→ in general, parameters estimated or obtained in a simulation

↳ quite high: if there's a load probably soon after we use that data.

↳ in 40% compiler can't find instruction to put in between.

* Amount of clock cycles in which pipe is stalled x p. of the event occurring

Effect of Branch Penalty

- Let's consider a proc. w/ *branch decision and branch target address evaluation in the Decode stage*. When branch is mispredicted, there is 1-cycle penalty.
  - E.g. if freq. of *branch* instr. p_{BR} = 20 %, prediction accuracy $p_{BR\text{-}pred}$ = 90 %, then throughput *F* is reduced to:
    - $F = R/(1 + \delta_{BR\_penalty})$ where $\delta_{BR\_penalty} = 1 x p_{BR} x (1- p_{BR\text{-}pred}) = 0.02$
    - Thus, $F = R/1.02 = 0.98R$
  - $\delta_{BR\_penalty}$ adds to δ_{stall}

We need to copy a whole block to do that.

Effect of Cache Misses

- When a cache miss occurs, there is a penalty due to the access of a slower memory which stalls the pipeline for N_{miss} cycles.
  - E.g. if freq. of *cache* misses during fetch $p_{miss\text{-}fetch}$ = 5 %*, freq. of *cache* misses during mem access $p_{miss\text{-}mem}$ = 10 %, freq. of Load and Store instr. $p_{LD\text{-}ST}$ = 30 %, N_{miss} = 15, then throughput *F* is reduced to:
    - $F = R/(1 + \delta_{cache\text{-}miss})$ where $\delta_{cache\text{-}miss} = N_{miss}\,(p_{miss\text{-}fetch} + p_{LD\text{-}ST} \times p_{miss\text{-}mem})$ $= 15\text{x}(0.05 + 0.03) = 1.2$
    - Thus $F = R/2.2 = 0.45R$
  - $\delta_{cache\text{-}miss}$ adds to $\delta_{BR\_penalty}$ and δ_{stall}.
  - Thus, overall $F = R/2.32 = 0.42R$ **Cache misses are the dominant factor.**

*This because memory code is executed sequentially

Number of Pipeline Stages n

If n is increased, more stages. We can expect max clock frequency to increase up to a certain point. But $\delta_{PENALTY}$ also increases. We expect more data dependencies (more stages in pipe) and for branches, because actual address calculated further on

$$F = \frac{R}{1 + \delta_{penalty}}, \; R \text{ clock frequency}$$

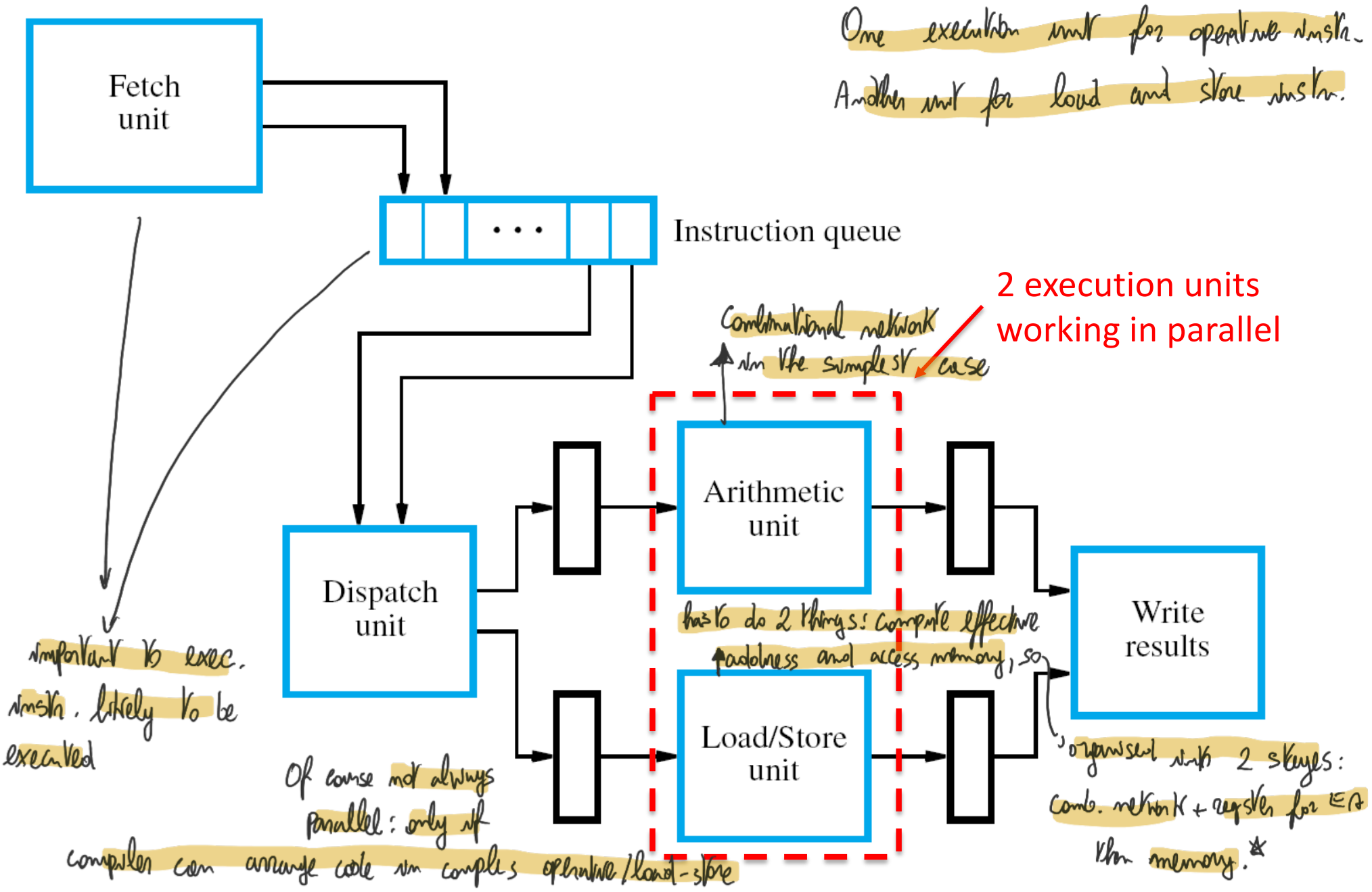
- R increases with n ($R \propto n$, if n is low)
- However, also $\delta_{penalty}$ increases with n because of higher probability of stalls, later branch decisions, higher cycle penalty,…
- Choose n so that ALU determines R (the other stages need similar times) OPTIMAL VALUE
  - To increase R further, pipeline also ALU
  - Up to 20 stages to have R in the order of GHz

If we want to do more, pipelining has given benefit. What's another approach that can be explored? You can do things in parallel (really)

Superscalar Operation

- Introduce **multiple execution units** to enable *multiple instruction issue* for **higher than 1** instr./cycle throughput
- This organization is for a **superscalar processor**
  - An "elaborate" *fetch unit* brings 2 or more instructions into an instruction queue in every cycle
  - A *dispatch unit* takes 2 or more instructions from the head of the instr. queue in every cycle, decodes them, sends them to the appropriate execution units
  - A *completion unit* writes results to registers

Superscalar Processor (1)



* Split because it simplifies the delay.

Superscalar Processor (2)

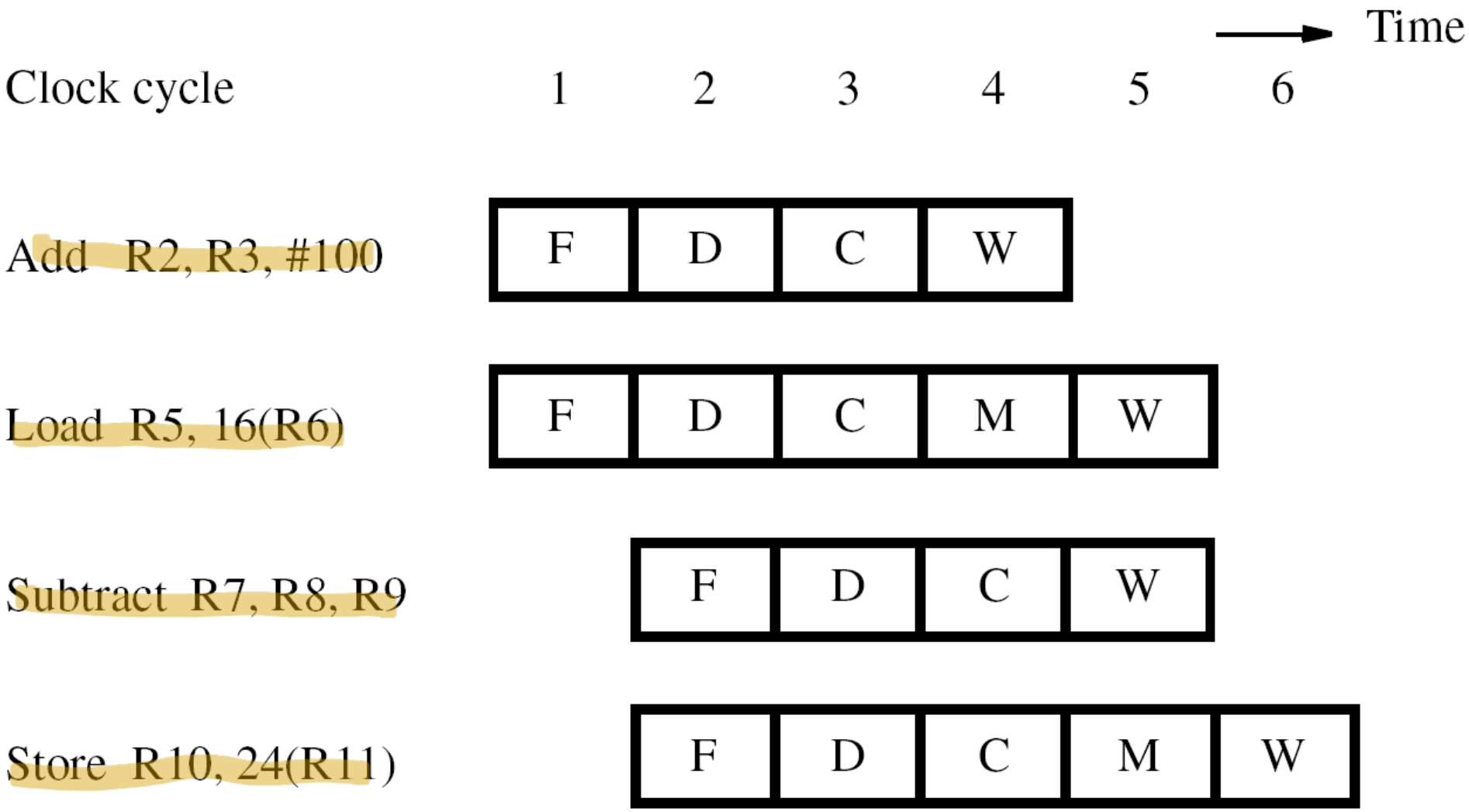
- Minimum superscalar arrangement consists of a **Load/Store** unit and an **Arithmetic** unit
  - Because of Index mode address calculation, Load/Store unit has a two-stage pipeline
  - Arithmetic unit usually has one stage

Arithmetic instructions take less than load/store in terms of # of clock cycles.

- For two execution units, how many operands?
  - Up to 4 inputs, so register file has 4 read ports
  - Up to 2 results, so also need 2 write ports (and methods to prevent writing to the same reg.)

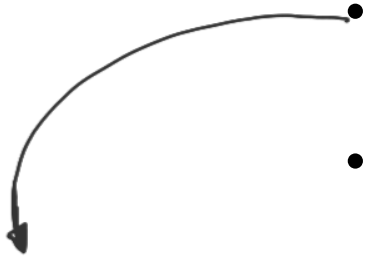
So instructions may overpass each other!

Superscalar Proc. Instr. Exec. Example



Branches and Data Dependencies (1)

- With no branches or data dependencies, interleave arithmetic & memory instructions to obtain maximum throughput (2 instr. per cycle)
- But branches do occur and must be handled
  - Branches processed entirely by fetch unit to determine which instructions enter queue
  - Fetch unit uses prediction for all branches
    - Necessary because decisions may need values produced by other instructions in progress
    - Stalling the fetch unit can significantly reduce throughput and is not acceptable



Instructions are in many cases executed till almost the end but we have to do something before it updates memory or registers. We need to be sure if instr. has to be executed before writing the memory.

For data dependencies we need HW to recognise what value for source register has not been evaluated yet.

In case of dependency, we park the instruction somewhere and let the next ones to execute. Still, we have a finite # of reservation stations, so still a stall preserv. Then when data is calculated, you need a restart signal, with the value that is ready and

Branches and Data Dependencies (2)

- *Speculative execution*: results of instructions not committed until prediction is confirmed
- Requires extra hardware to track speculation and to recover in the event of misprediction
- For data dependencies between instructions, the execution units have *reservation stations*
  - They buffer register identifiers and operands for dispatched instructions awaiting execution
  - Broadcast results for stations to capture & use

Out-of-Order Execution

- With instructions buffered at execution units, should execution reflect original sequencing?
- If two instructions have no dependencies, there are no actual ordering constraints
- This enables *out-of-order execution*, but then leads to *imprecise exceptions*
- For *precise exceptions*, results must strictly be committed in original order. This requires additional hardware

Exceptions might happen. But ISR is called after some subsequent instructions might have been executed. The status of the register is different which is a problem.

Solution: problem arises when status of processor changed after instr. caused exc. To avoid that, software to write the registers following the same order of the code. So instructions are committed in order.

You need a memory for the order of code. Then something to place results before they are written. So temp registers.

Execution Completion

- To commit results in original program order, superscalar processors use 2 techniques:
  - *Register renaming* uses temporary registers to hold new data before it is safe to commit them in the register file
  - *Reorder buffer* in commitment unit is where dispatched instructions are placed strictly in the program order
    - Update the actual destination register only for instruction at the head of *reorder buffer* queue*
    - Ensures instructions *retired* in original order

* Contains instr. in the same order of their fetch.

Dispatch Operation

- Dispatch of instruction proceeds only when all needed resources available (temp. register, space in reservation station & reorder buffer)
- If instruction has some but not all resources, should a subsequent instruction proceed?
  - Decisions must avoid *deadlock* conditions Very complex (two instructions need each other's resources)
  - More complex, so easier to use original order, particularly with more than 2 execution units

RISC: every instr. in 1 word; only 2 kinds of memory accessing instr.;

Pipelining in CISC Processors

- Load/Store architecture simplifies pipelining; influenced development of RISC processors
- CISC processors introduce complications from instructions with multiple memory operands and side effects (autoincrement, cond. codes)
- But existing CISC architectures later pipelined (with more effort) after development of RISC
- Examples: Freescale ColdFire and Intel IA-32

Concluding Remarks

- Pipelining overlaps activities for 1 instr./cycle
- Combine it with multiple instruction issue in superscalar processors for +1 instr./cycle
- Potential performance gains depend on:
  - Instruction set characteristics
  - Design of pipeline hardware
  - Ability of compilers to optimize code
- Interaction of these aspects is a key factor

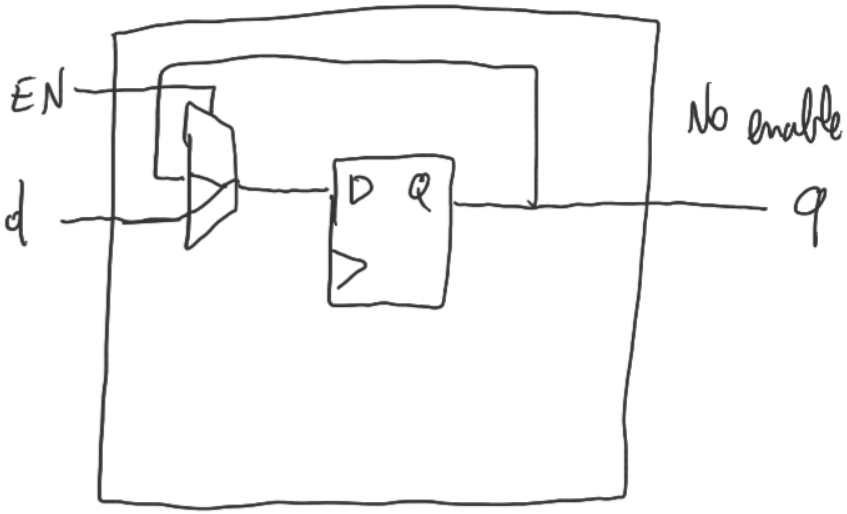
Remember EN on PC! Quando fare call R

Set of D-positive edge triggered FF with the same clock signal

References

- C. Hamacher, Z. Vranesic, S. Zaky, N. Manjikian "Computer Organization and Embedded Systems," *McGraw-Hill International Edition*
  - Chapter VI: Pipelining

DFF with enable: (EN active high)



No enable = less area.

NOTE: NO ACTION
on WRITE = write signal on RF is disabled.

Complementary Metal Oxide Semiconductor Field effect transistor.

CONNECTION BY LABEL

SYMBOLS

THIS NODE IS CONNECTED TO THE MINUS TERMINAL OF POWER SUPPLY.

NegativeMOS: carriers in channel are electrons and you need to attract negative charges from the gate. ↳ REMEMBER SUBSTRATE IS P. So you need to attract the minority of charges.

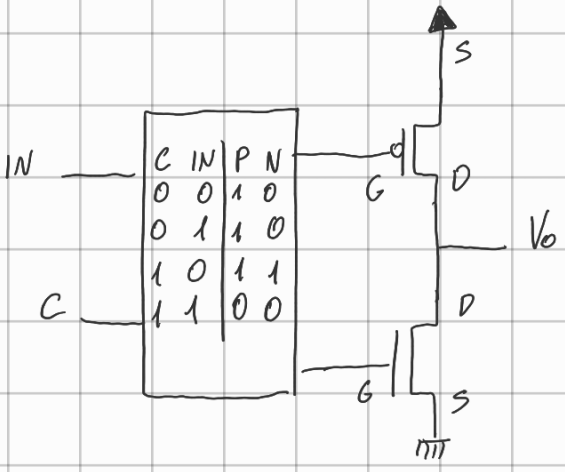
NEED TO REMEMBER THIS:

V_{IN}	NMOS	PMOS	V_{out}
0	OFF	ON	1
1	ON	OFF	0

$V_{GSp} = V_{IN} - V_{cc} \Rightarrow 0 - V_{cc} < V_{Tp} < 0$ ✓

NAND: 2 NMOS in series, 2 PMOS in parallel. NOR: is the opposite.

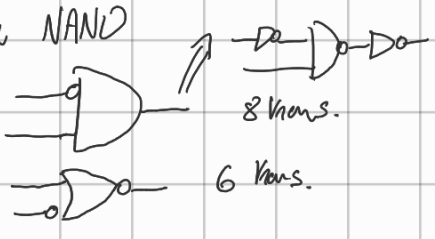
TRI STATE BUFFER:

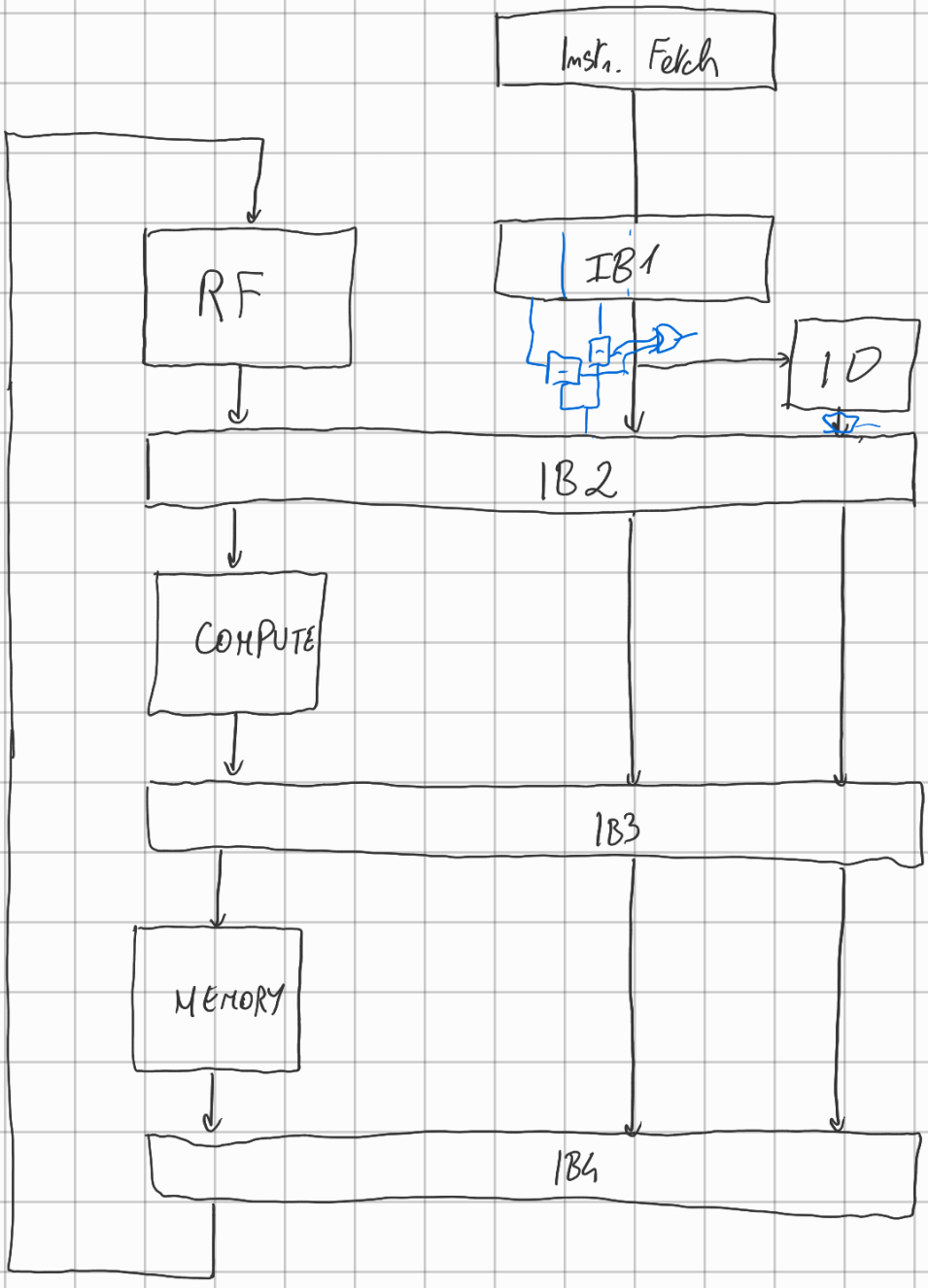


When control is 0, both transistors are off. ⇒ P is a NAND

N is

OR:





	1	2	3	4	5	6	7	8
I_S	F	D	C	M	W			
I_{S+1}		F	D	D	D	D	C	M	W
I_{S+2}								

1	2	3	4	5	6
add	sub	sub	sub	sub	

1	2	3	4	5	6
	add	nop	nop	nop	sub

1	2	3	4	5	6
		add	nop	nop	nop

1	2	3	4	5	6
			add	nop	nop

Power dissipation: static (even 20-30% for leakage) + dynamic

FOR LEAKAGES ideally 0 because we off one of the two.

Dynamically: on the output we can model load as capacitor. That needs to change and discharge. When input changes, there's interval time in which transistors are on.

But even if inputs change abruptly, we have switching power, $f \cdot C \cdot V_{cc}^2 \cdot a$, with

a = activity factor $\in [0, 1]$ This is sum up for every node.