CS356 Unit 12

Processor Hardware Organization

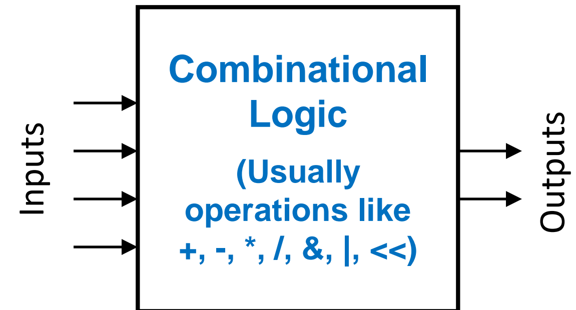
Pipelining

From combinational to sequential logic

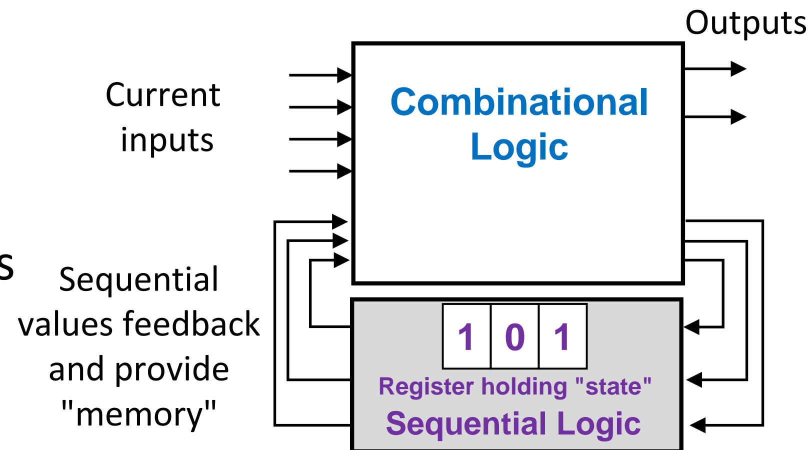
BASIC HW

Logic Circuits

- _______________ **Logic**
  - Performs a specific function (mapping of 2^n input combinations to desired output combinations)
  - **No internal state** or feedback
    - Given a set of inputs, we will always get the same output after some time (propagation) ________
- _______________ **Logic**
  - **Registers**: fundamental building blocks
    - __________ a set of bits for later use
    - Acts like a variable from software
    - Controlled by a "clock" signal



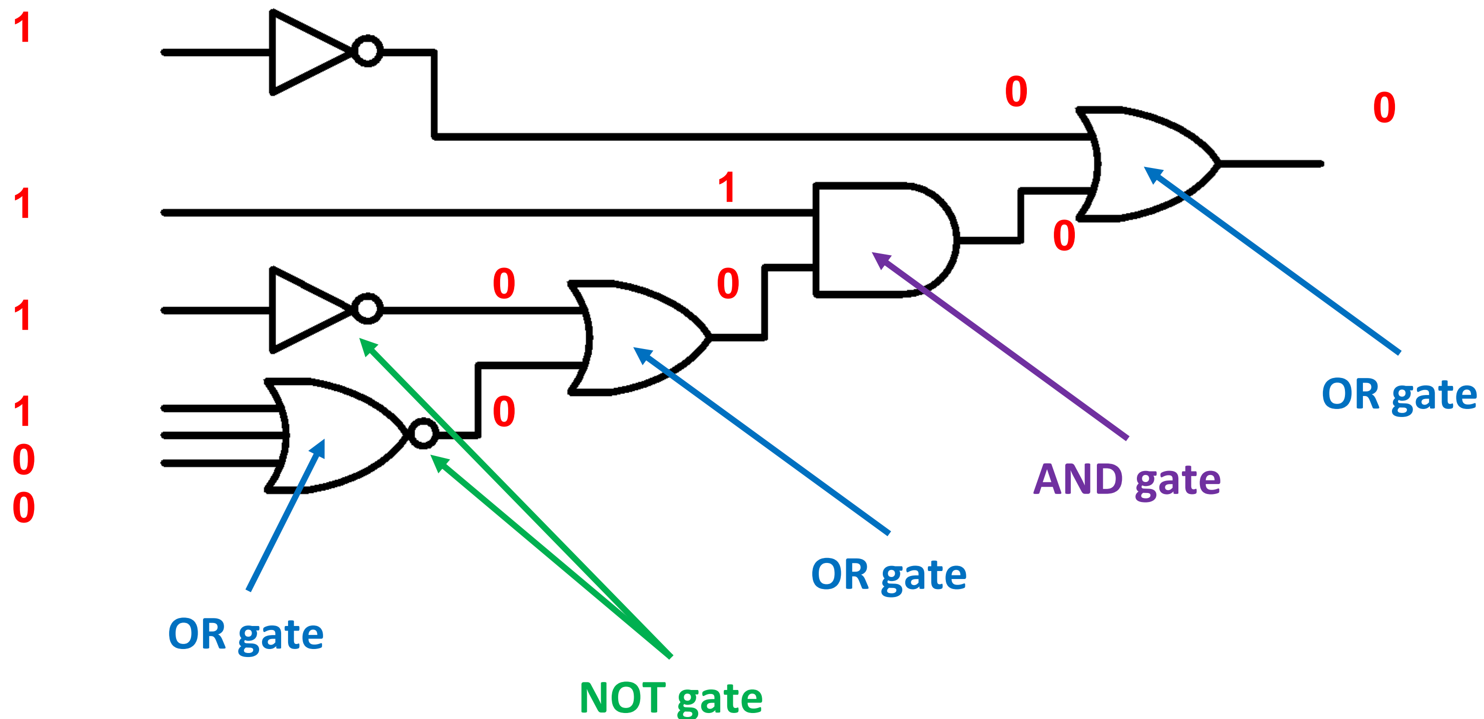
Outputs depend only on current outputs



Outputs depend on current inputs and previous inputs (previous inputs summarized by current state)

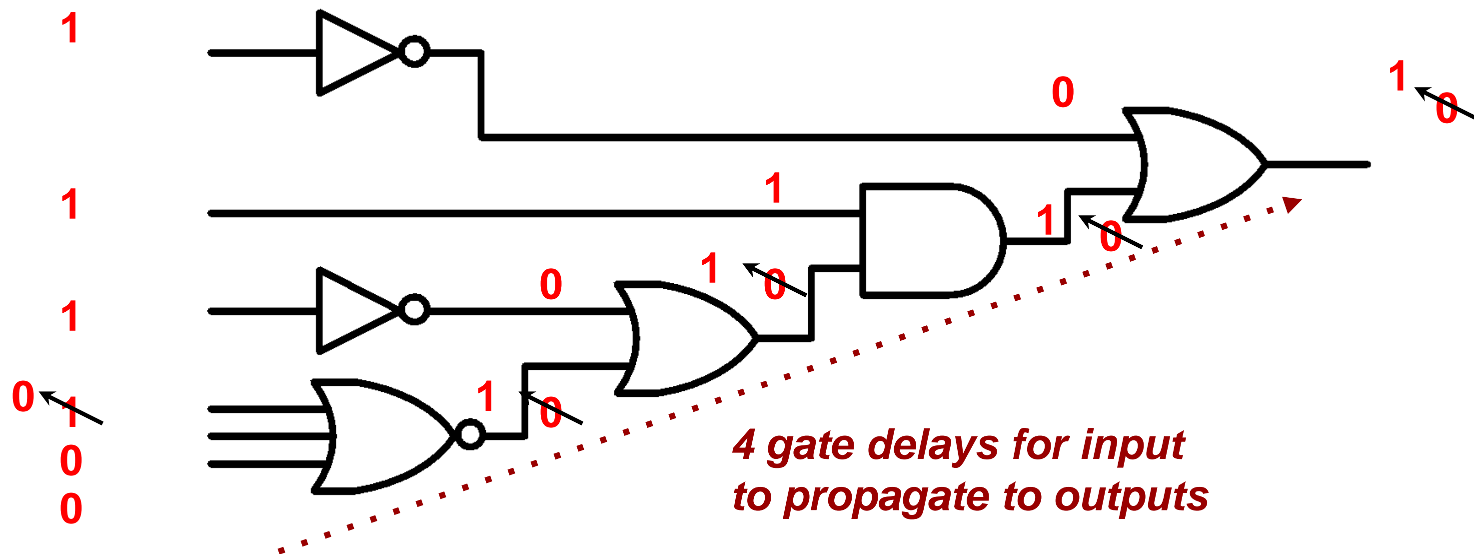
Combinational Logic Gates

- Circuits called **gates** perform logic operations to produce desired outputs from some digital inputs



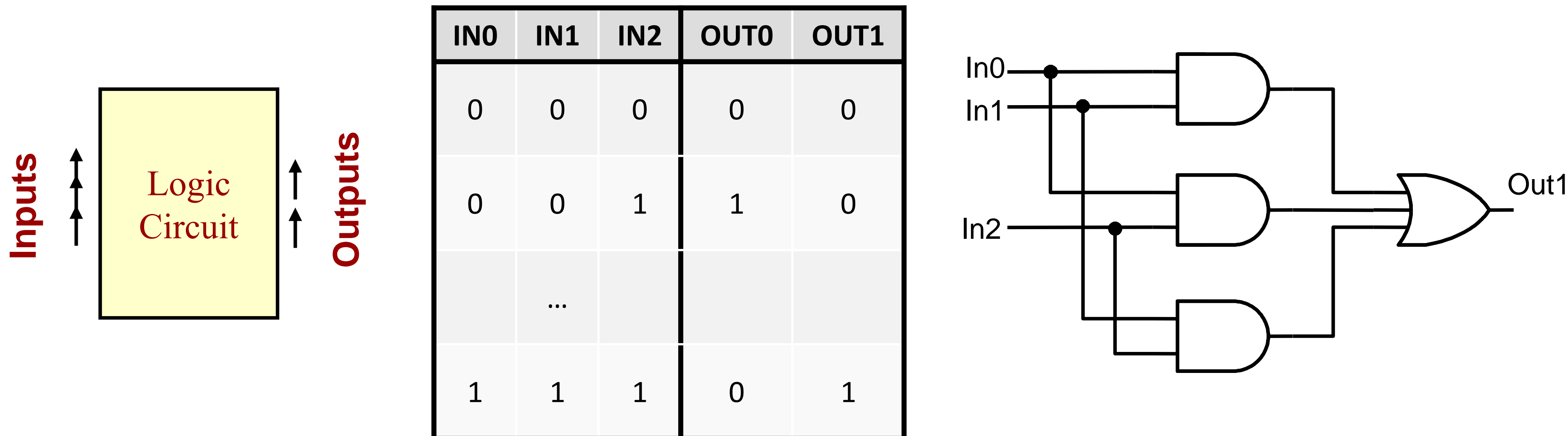
Propagation Delay

- All digital logic circuits have propagation delay
  - Time for output to change when inputs are changed



Combinational Logic Functions

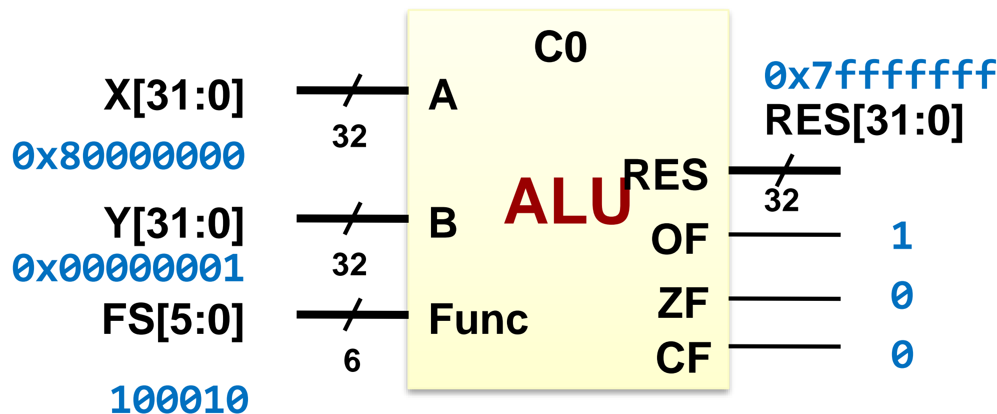
- Map input combinations of *n*-bits to desired *m*-bit output
- Can describe function with a **truth table** and then find its circuit implementation



IN0	IN1	IN2	OUT0	OUT1
0	0	0	0	0
0	0	1	1	0
	…			
1	1	1	0	1

ALU's

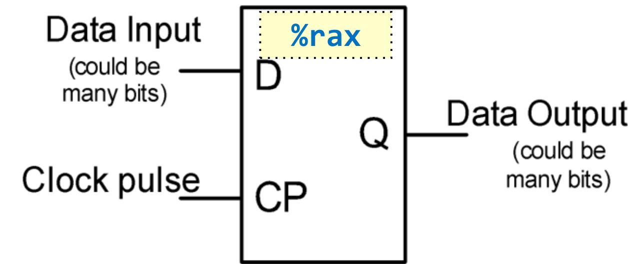
- Perform a selected operation on two input numbers
  - FS[5:0] selects the desired operation



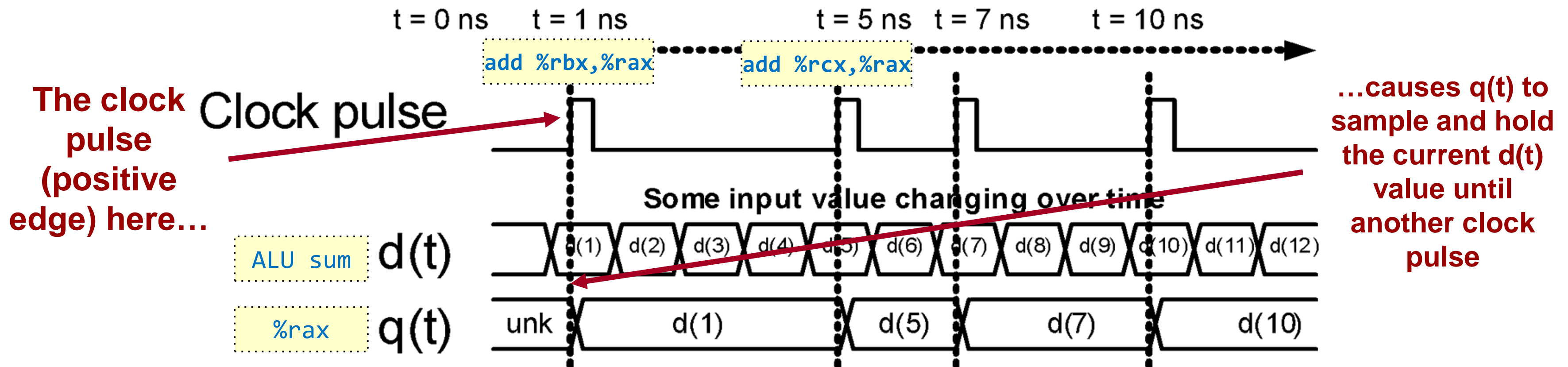
Func. Code	Op.	Func. Code	Op.
00_0000	A SHL B	10_0000	A+B
00_0010	A SHR B	10_0010	A-B
00_0011	A SAR B	…	…
…	…	10_0100	A AND B
01_1000	A * B	10_0101	A OR B
01_1001	A * B (uns.)	10_0110	A XOR B
01_1010	A / B	10_0111	A NOR B
01_1011	A / B (uns.)	…	…
…	…	10_1010	A < B

Sequential Devices (Registers)

- **Registers** ______ the **D input** value when a control input (*clock signal*) transitions from ______ (*clock edge*) and store that value at the **Q output** until the next clock edge
- A register is similar to a _________ in software: at the clock edge, it stores a value for later use.
- We can choose to only clock the register at "_______" times when we want the register to capture a new value (e.g., when it is the ___________ of an instruction)
- **Key Idea**
  Registers ______ data while we operate on those values

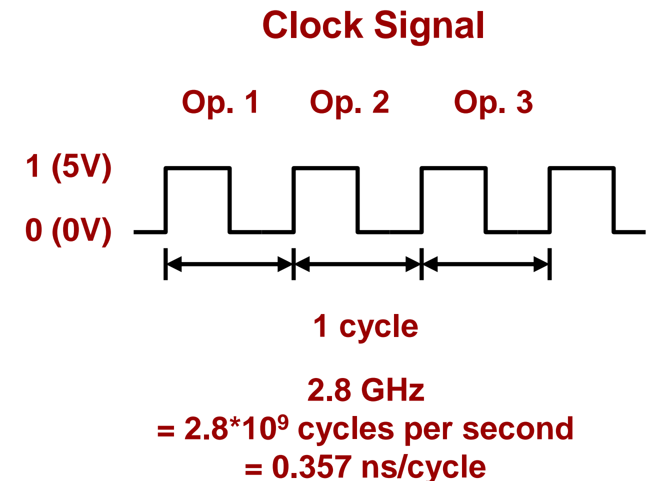


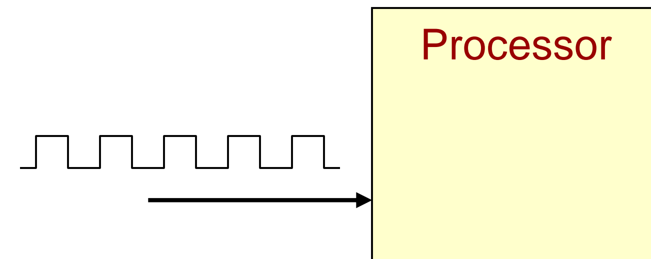
Block Diagram of a Register



Clock Signal

- Alternating high/low voltage pulse train
- Controls the ordering and timing of operations performed in the processor
- 1 cycle is usually measured from rising edge to rising edge
- Clock frequency = # of cycles per second (e.g. 2.8 GHz = 2.8 * 10^9 cycles per second)





Basic HW organization for a simplified instruction set

FROM X86 TO RISC

From CISC to RISC

- **Complex Instruction Set Computers** (CISC) often have instructions that _______ widely in how much _______ they perform and how much ________ they take to execute
  - Fewer instructions are needed for a task
- **Reduced Instruction Set Computers** (RISC) favor instructions that take roughly the same time to execute and follow a common sequence of steps
  - ______ instructions needed, each faster

```
// CISC instruction
movq 0x40(%rdi, %rsi, 4), %rax

// RISC equivalent with 1 memory or ALU
// operation per instruction
mov %rsi, %rbx   # use %rbx as a temp.
shl    2, %rbx   # %rsi * 4
add %rdi, %rbx   # %rdi + (%rsi*4)
add $0x40, %rbx  # 0x40 + %rdi + (%rsi*4)
mov (%rbx), %rax # %rax = *%rbx
```

CISC vs. RISC Equivalents

“Iron Law” of Processor Performance: How RISC can win

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Clock cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Clock cycle}}$$

John Hennessy and David Patterson, ACM Turing Award Lecture, 2017

A RISC Subset of x86

- Split mov instructions that access memory into separate instructions:
  - ld = ____________ from memory
  - st = ____________ to memory
- Limit ld & st instructions to use at most

  ____
  - No ld 0x04(%rdi, %rsi, 4), %rax
    - Too much work
  - At most ld 0x40(%rdi), %rax or
    st %rax, 0x40(%rdi)
- Limit arithmetic & logic instructions to only operate on registers
  - No add (%rsp), %rax since this implicitly accesses (dereferences) memory
  - Only add _______________

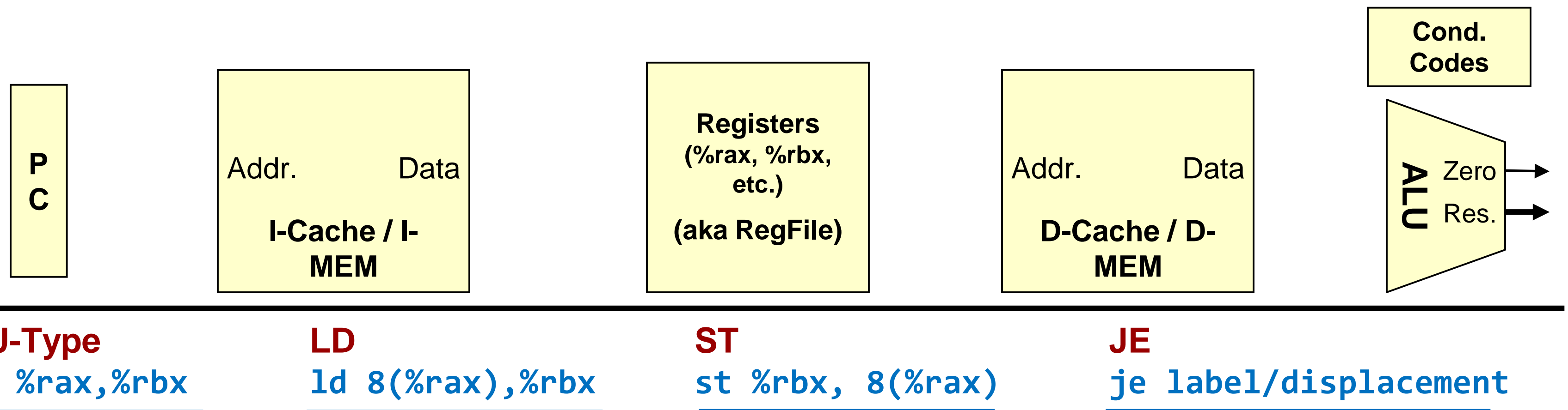
```
// 3 x86 memory read instructions
mov (%rdi), %rax             // 1
mov 0x40(%rdi), %rax         // 2
mov 0x40(%rdi,%rsi), %rax // 3

// Equivalent load sequences
ld 0x0(%rdi), %rax           // 1
ld 0x40(%rdi), %rax          // 2
mov %rsi, %rbx               // 3a
add %rdi, %rbx               // 3b
ld  0x40(%rbx), %rax         // 3c
```

```
// 3 x86 memory write instructions
mov %rax, (%rdi)             // 1
mov %rax, 0x40(%rdi)         // 2
mov %rax, 0x40(%rdi,%rsi) // 3

// Equivalent store sequences
st %rax, 0x0(%rdi)           // 1
st %rax, 0x40(%rdi)          // 2
mov %rsi, %rbx               // 3a
add %rdi, %rbx               // 3b
st  %rax, 0x40(%rbx)         // 3c
```

```
// CISC instruction
add %rax, (%rsp)

// Equivalent RISC sequence with ld / st
ld   0(%rsp), %rbx
add  %rax, %rbx
st   %rbx, 0(%rsp)
```

Developing a Processor Organization

Hardware components used by each instruction type:

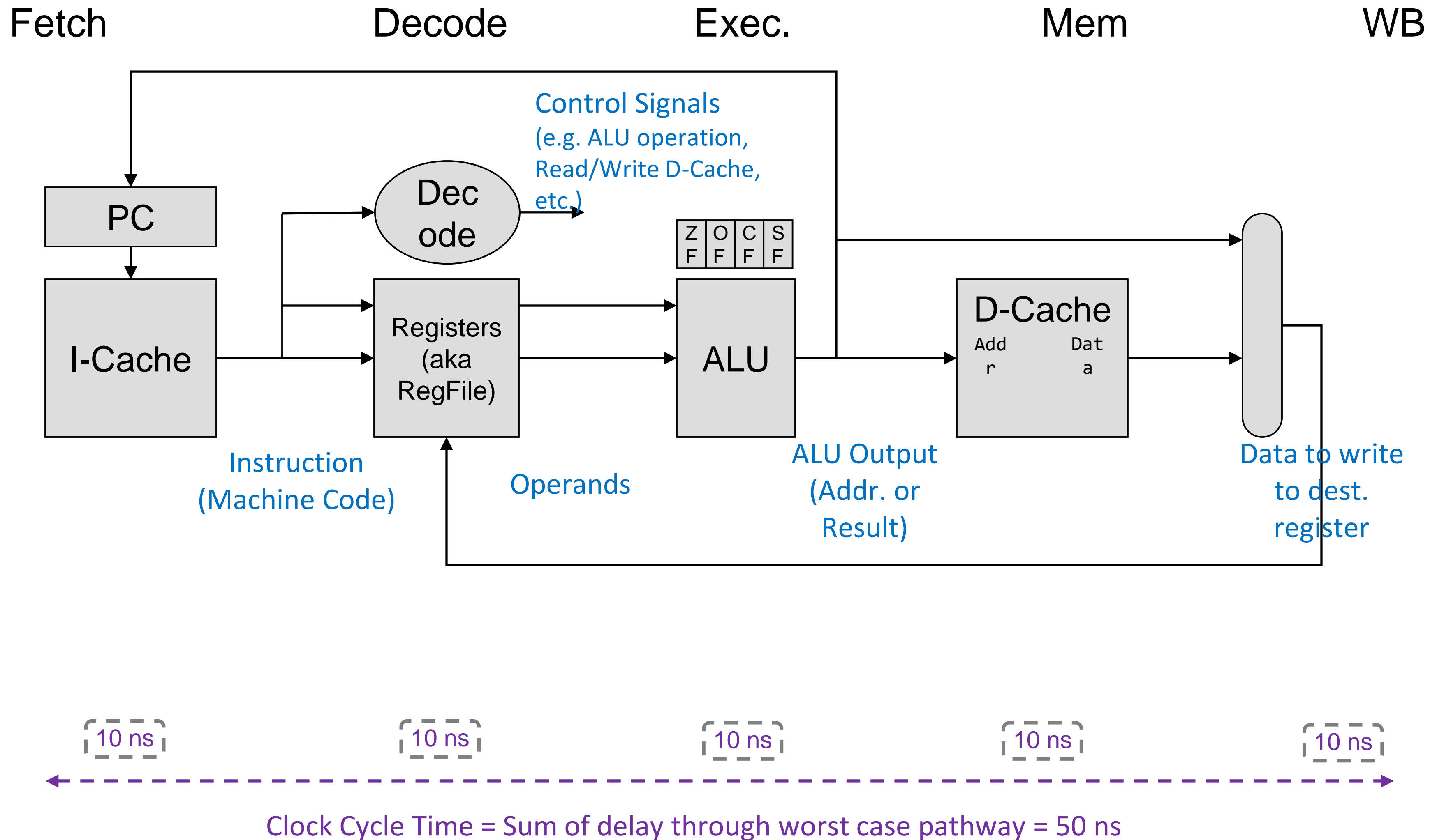


ALU-Type	LD	ST	JE
`add %rax,%rbx`	`ld 8(%rax),%rbx`	`st %rbx, 8(%rax)`	`je label/displacement`

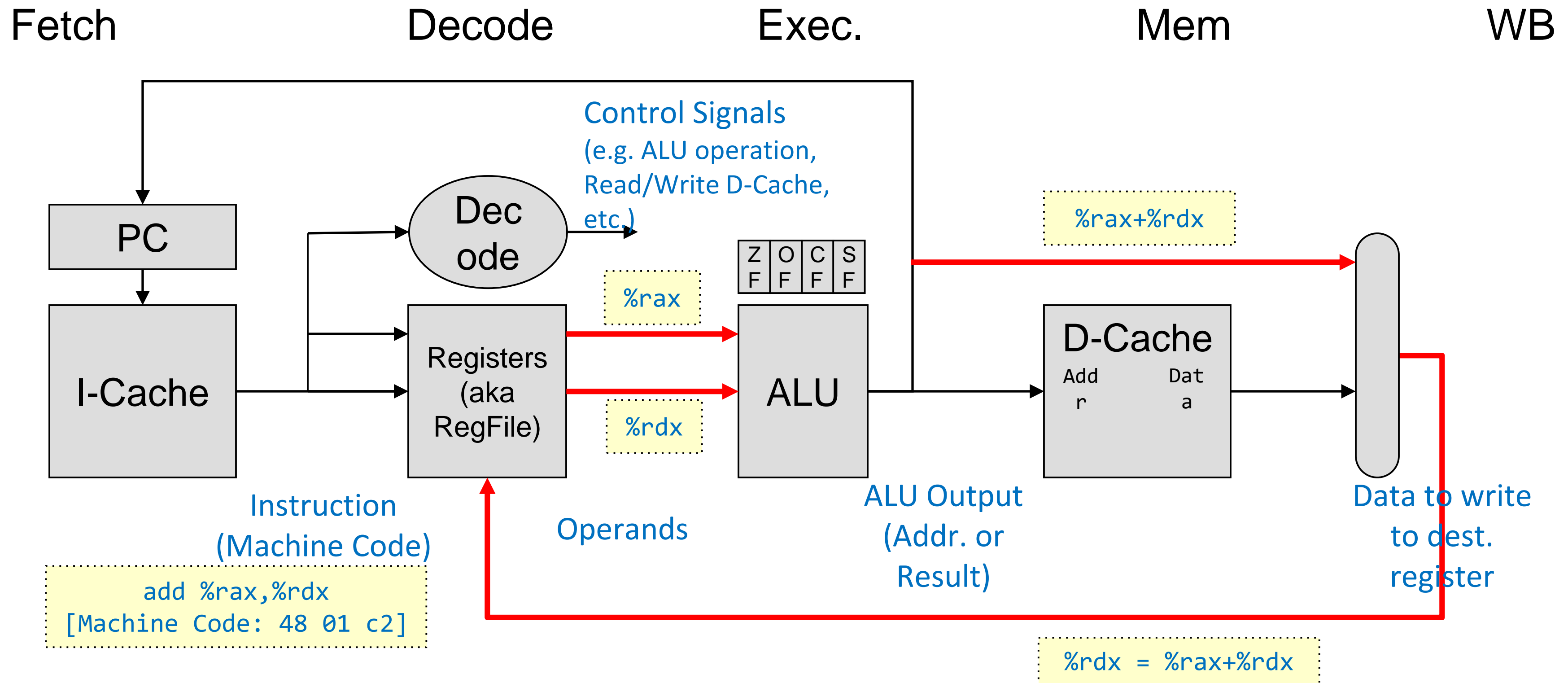
1.
2.
3.
4.
5.
6.

USC Viterbi School of Engineering

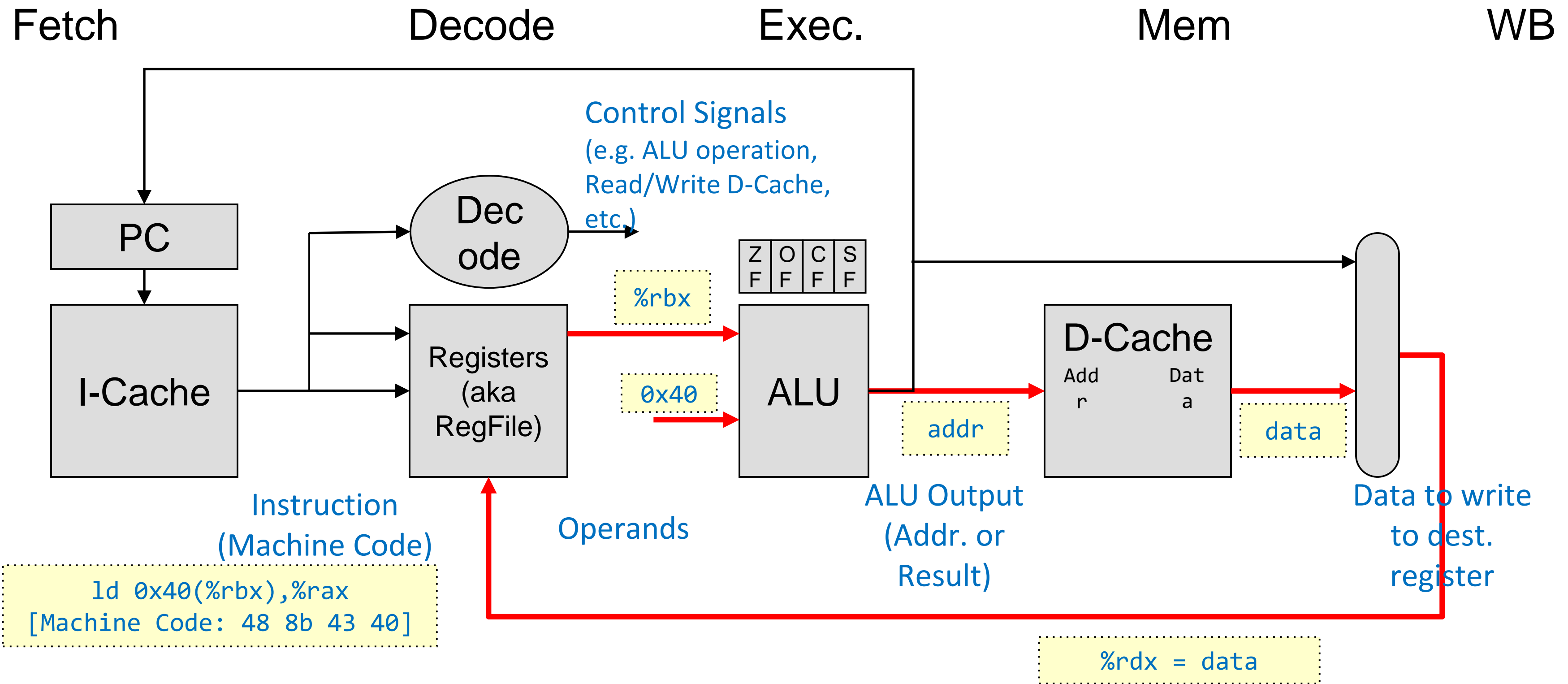
Processor Block Diagram



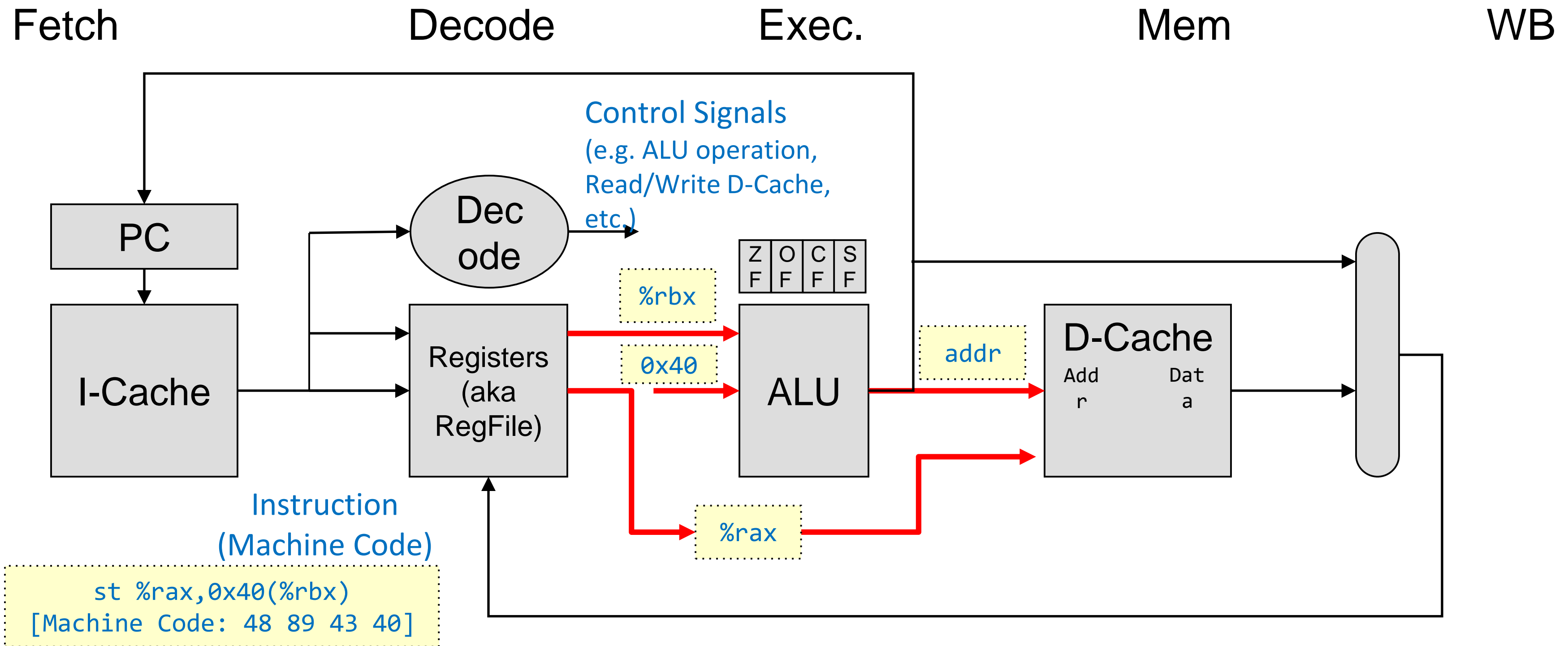
Processor Execution (add)



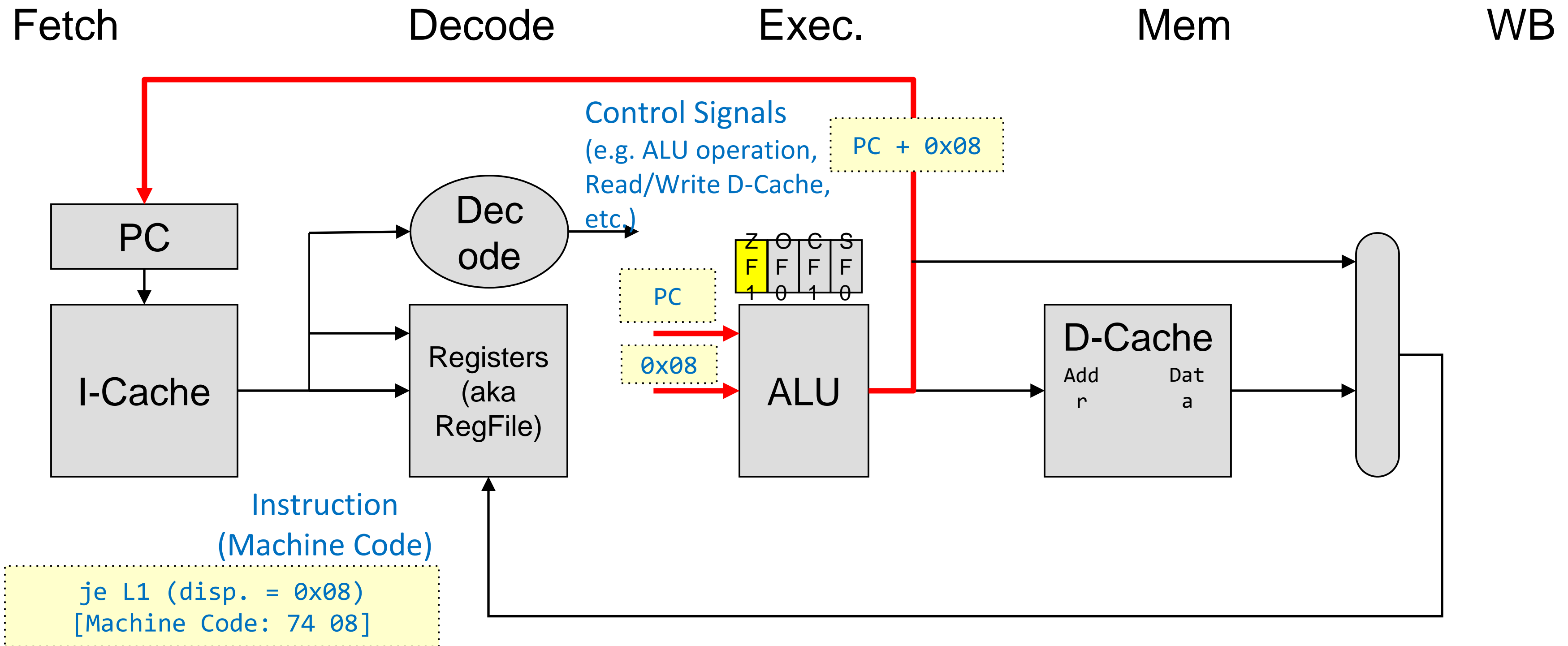
Processor Execution (ld)



Processor Execution (st)



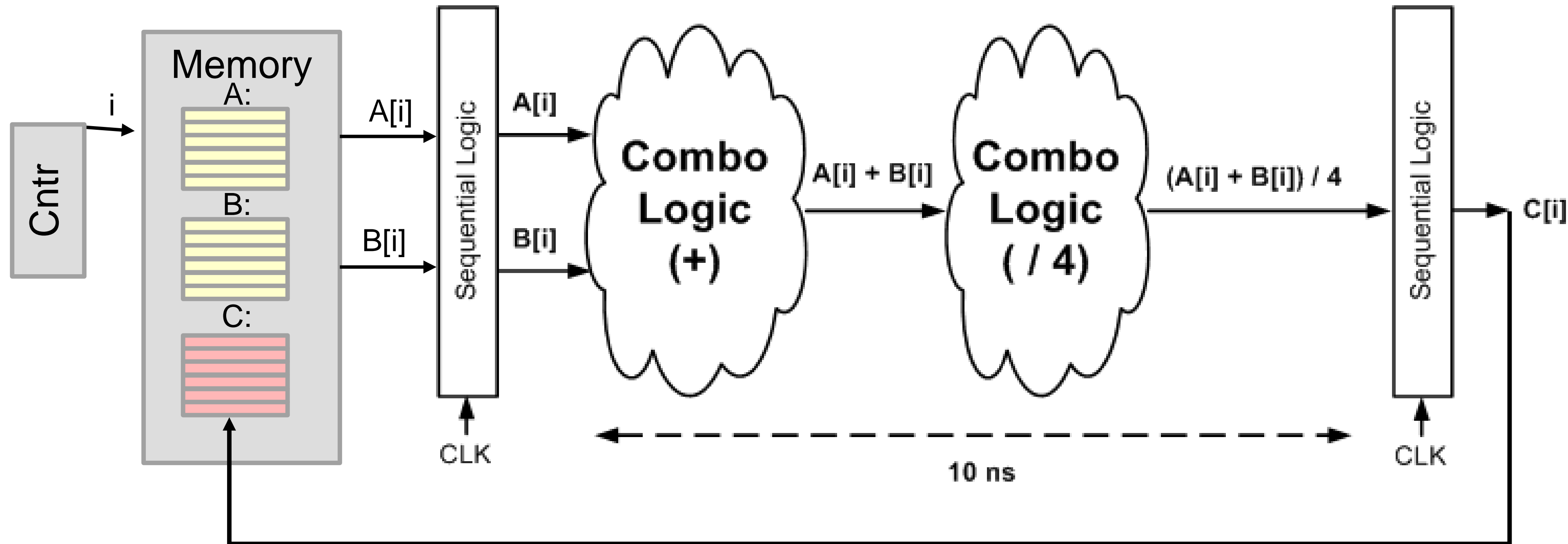
Processor Execution (je)



PIPELINING

Example

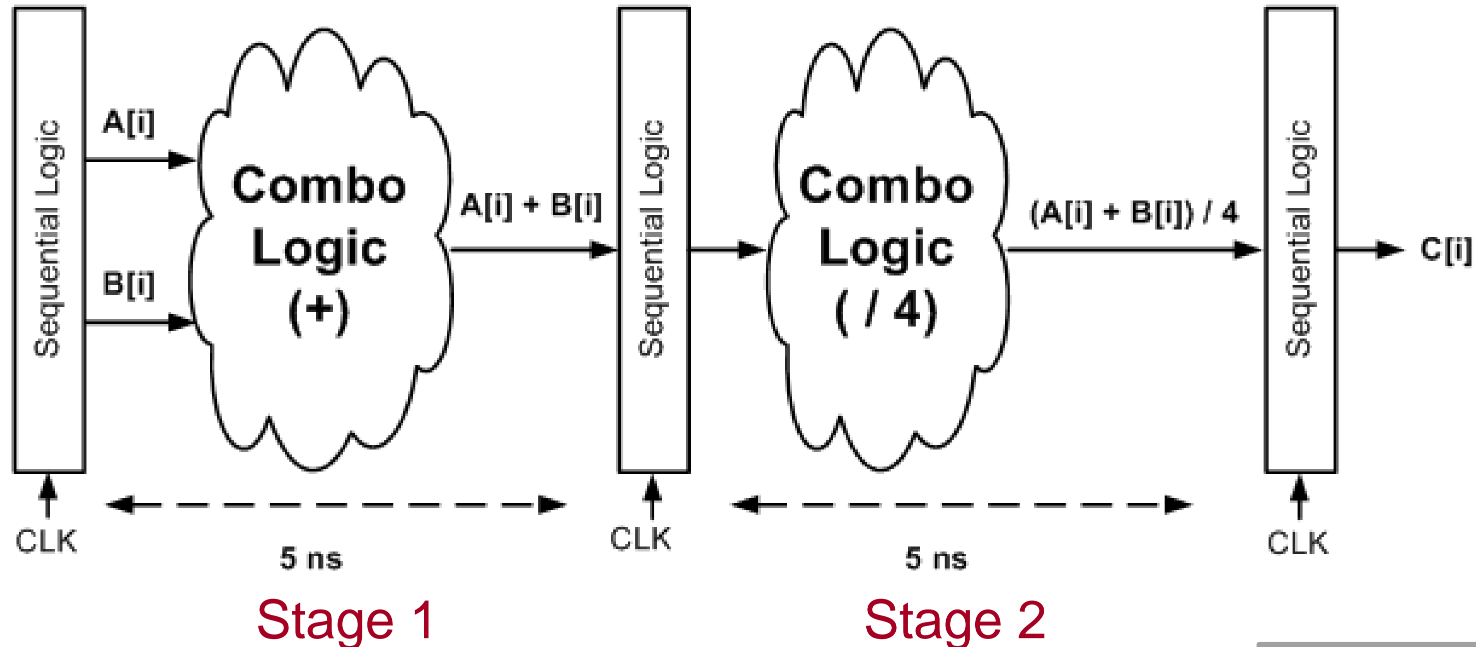
```
for(i=0; i < 100; i++)
   C[i] = (A[i] + B[i]) / 4;
```



10 ns per input set = 1000 ns total

Pipelining Example

```
for(i=0; i < 100; i++)
  C[i] = (A[i] + B[i]) / 4;
```

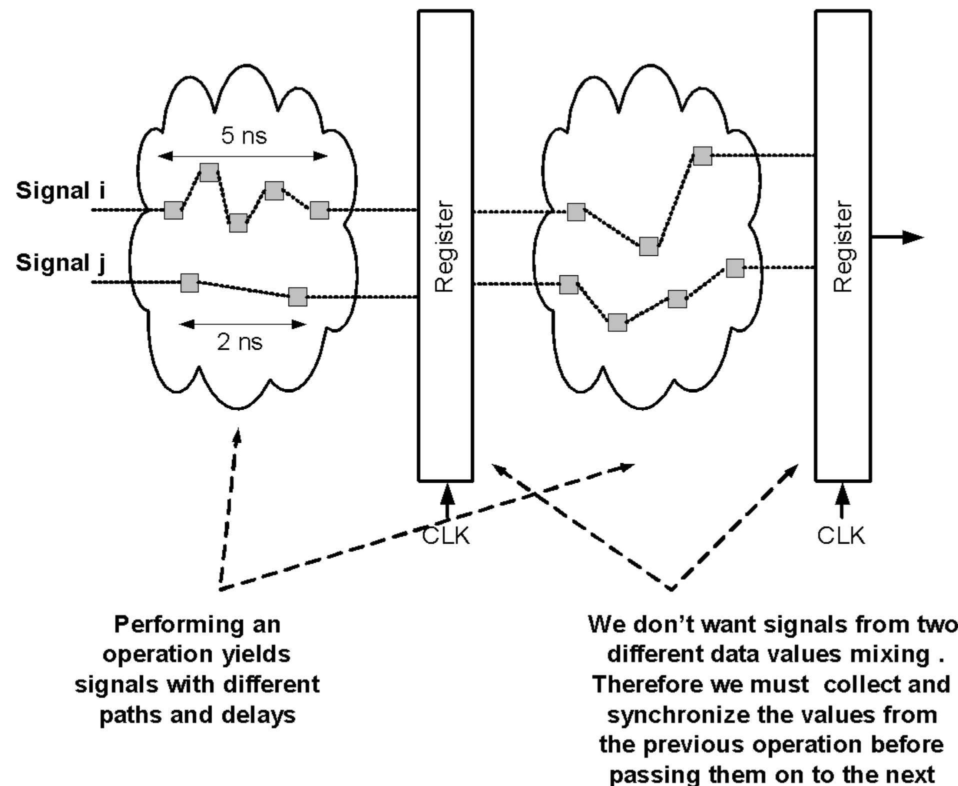


	Stage 1	Stage 2
Clock 0	A[0] + B[0]	
Clock 1	A[1] + B[1]	(A[0] + B[0]) / 4
Clock 2	A[2] + B[2]	(A[1] + B[1]) / 4

Pipelining refers to insertion of registers to split combinational logic into smaller stages that can be overlapped in time (i.e., create an assembly line)

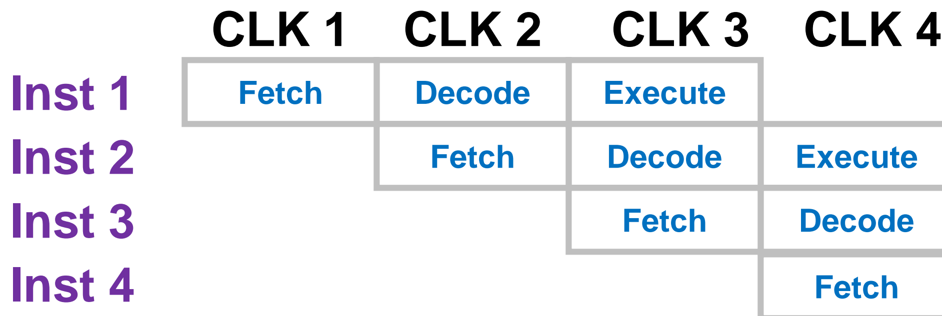
Need for Registers

- Provides separation between combinational functions
  - Without registers, fast signals could “catch-up” to data values in the next operation stage

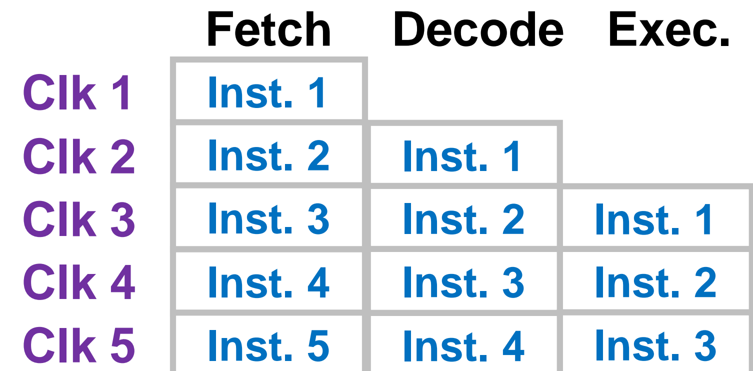


Processors & Pipelines

- Overlaps execution of multiple instructions
- Natural breakdown into stages
  - ______, _______, _______, _______, _______
- Fetch an instruction, while decoding another, while executing another



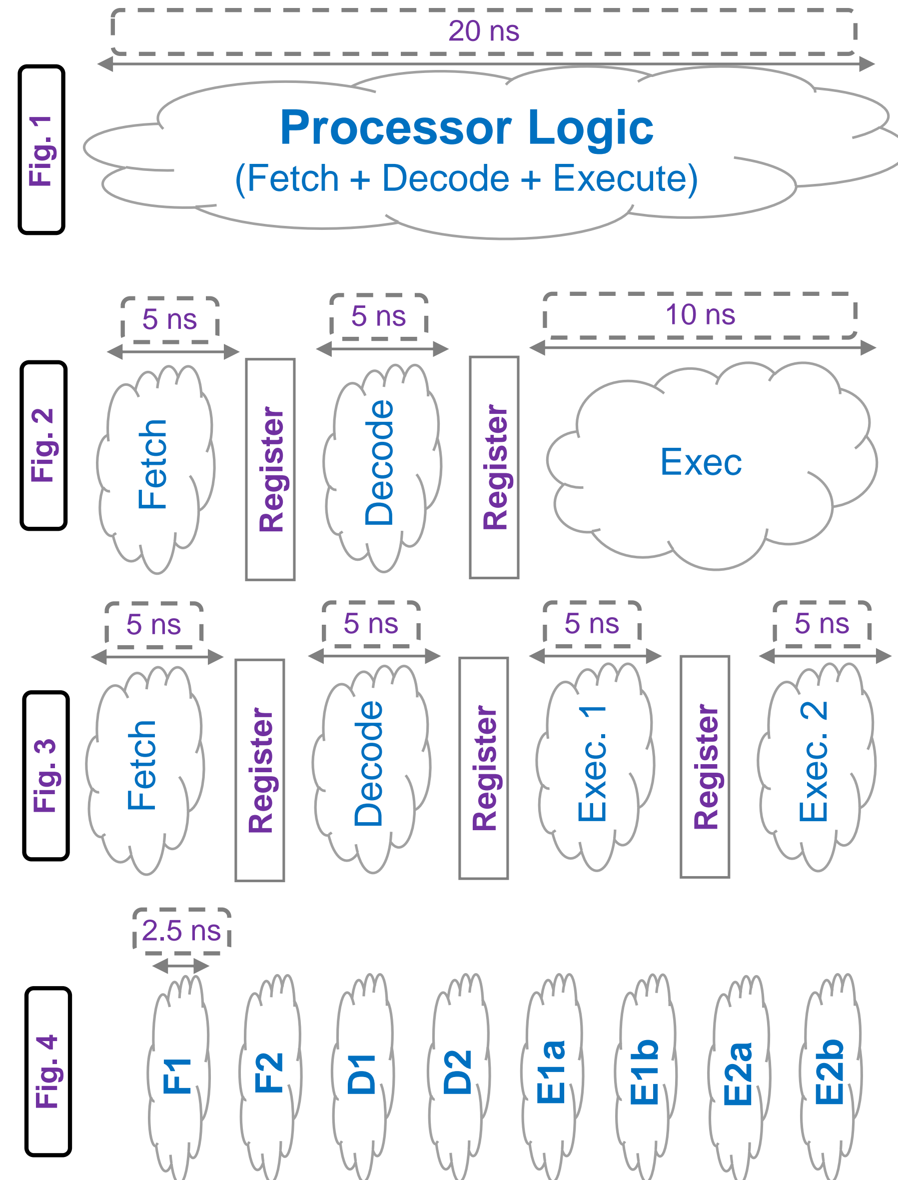
Pipelining (Instruction View)



Pipelining (Stage View)

Balancing Pipeline Stages

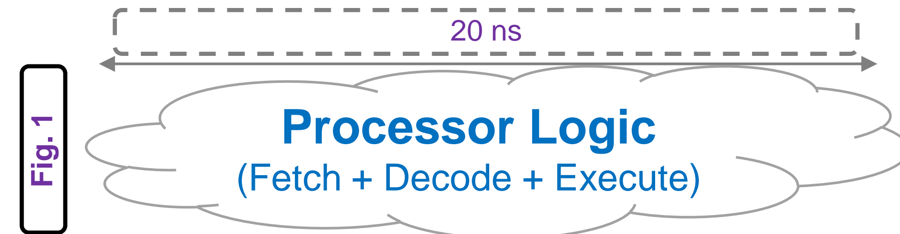
- Clock period must equal the *LONGEST* delay from register to register
- Fig. 1: If total logic delay is 20ns => 50MHz
  - Throughput: 1 instruc. / 20 ns
- Fig. 2: Unbalanced stage delays limit the clock speed to the slowest stage (worst case)
  - Throughput: 1 instruc. / 10 ns => 100MHz
- Fig. 3: Better to split into more, balanced stages
  - Throughput: 1 instruc. / 5 ns => 200MHz
- Fig. 4: Are more stages better
  - Ideally: 2x stages => 2x throughput
  - Throughput: 1 instruc. / 2.5 ns => 400MHz
  - **Each register adds extra delay so at some point deeper pipelines don't pay off**



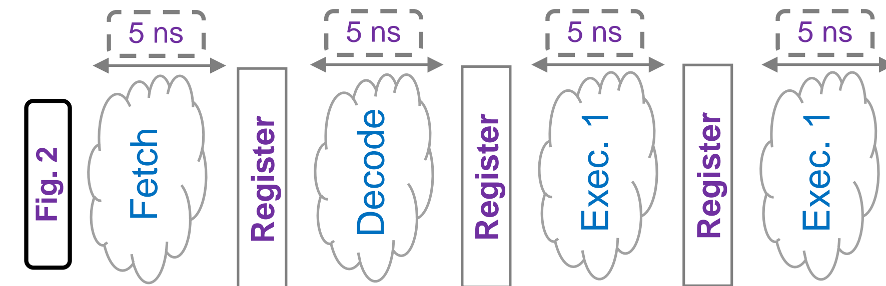
Balancing Pipeline Stages

Main Points:

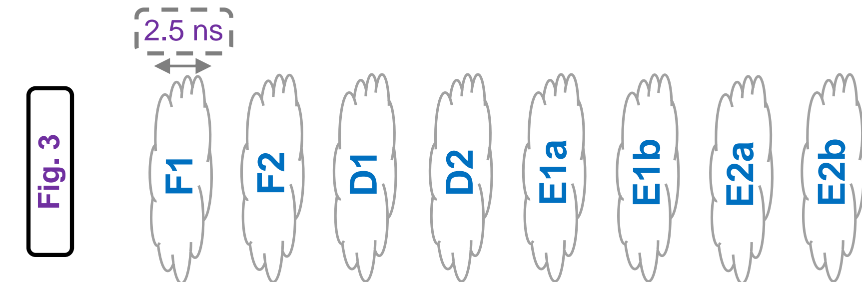
- **Latency** of any single instruction is ___________
- **Throughput** and thus overall program performance can be dramatically __________
  - Ideally K stage pipeline will lead to throughput increase by a factor of ____
  - Reality is splitting stages adds some ________ and thus hits a point of diminishing returns



Non-pipelined
(Latency = 20ns, Throughput = 1x)

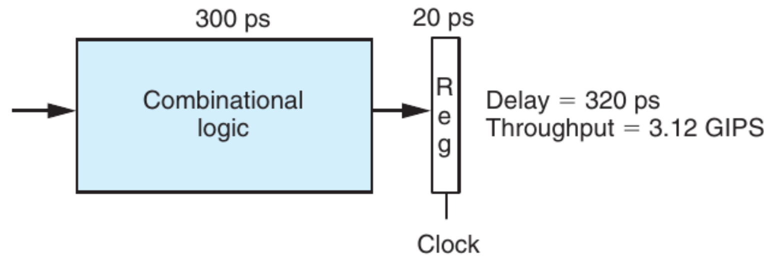


4 Stage Pipeline
(Latency = 20ns, Throughput = 4x)



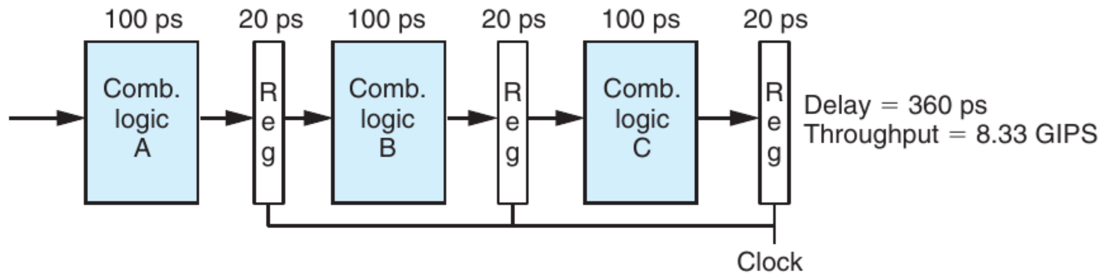
8 Stage Pipeline
(Latency = 20ns, Throughput = 8x)

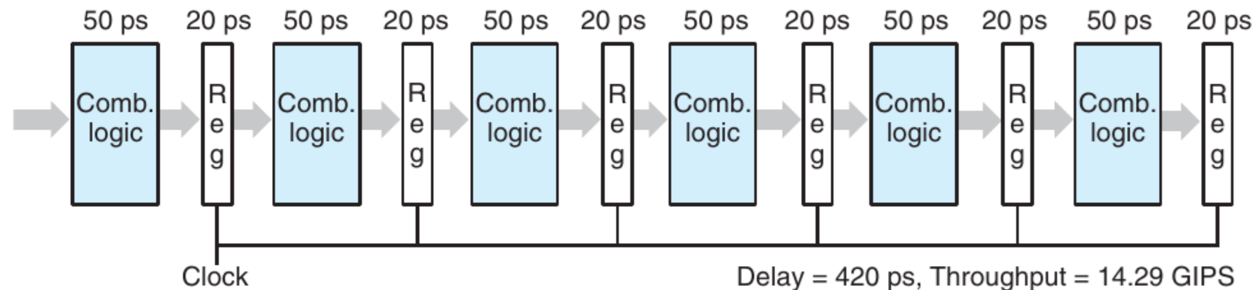
Throughput and Latency



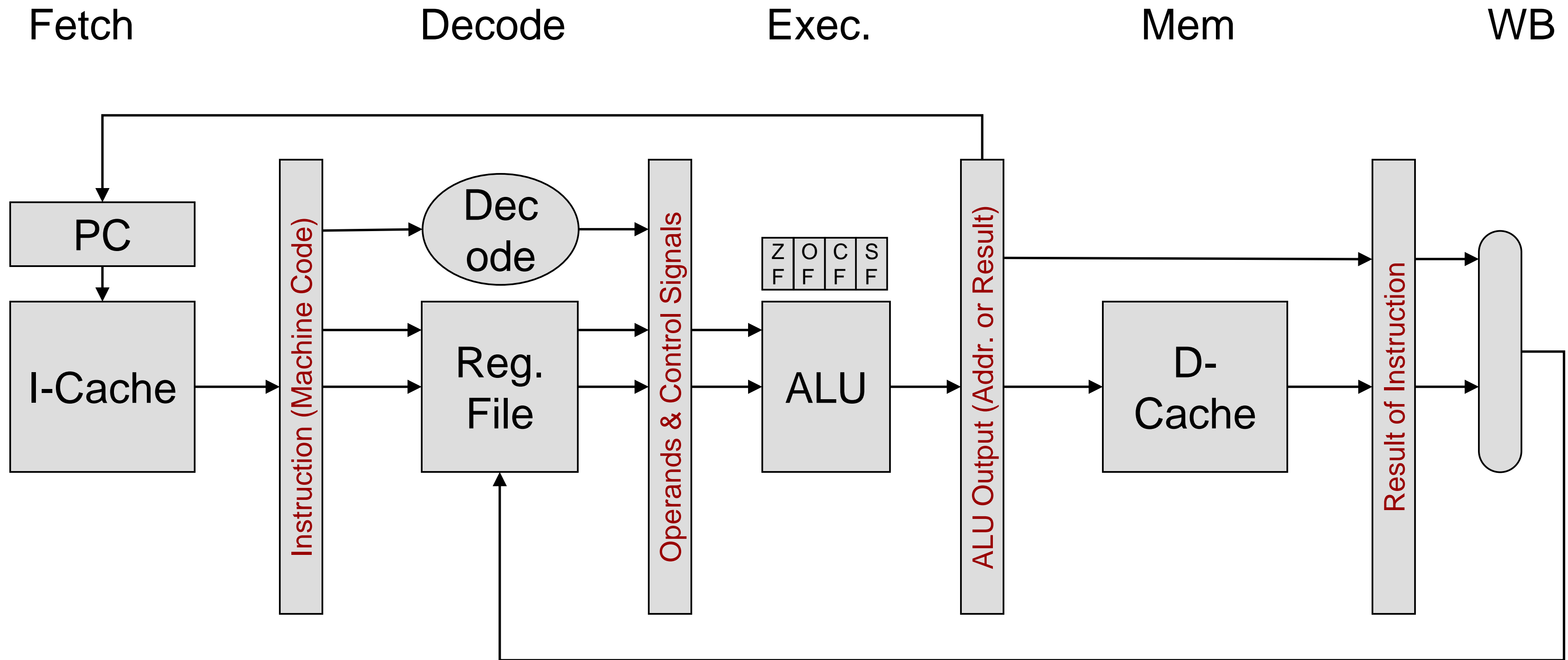
n	clock (ps)	tput (GIPS)
1	320	3.125
2	170	5.882
3	120	8.333
4	95	10.526
5	80	12.500
6	70	14.286

```
clock = 300/n + 20
tput  = 1/clock
delay = n*clock
```





5-Stage Pipeline



Pipelining

- Let's see how a sequence of instructions can be executed

Instruction
`ld 0x40(%rbx),%rax`
`add %rcx,%rdx`
`je L1`

Sample Sequence - 1

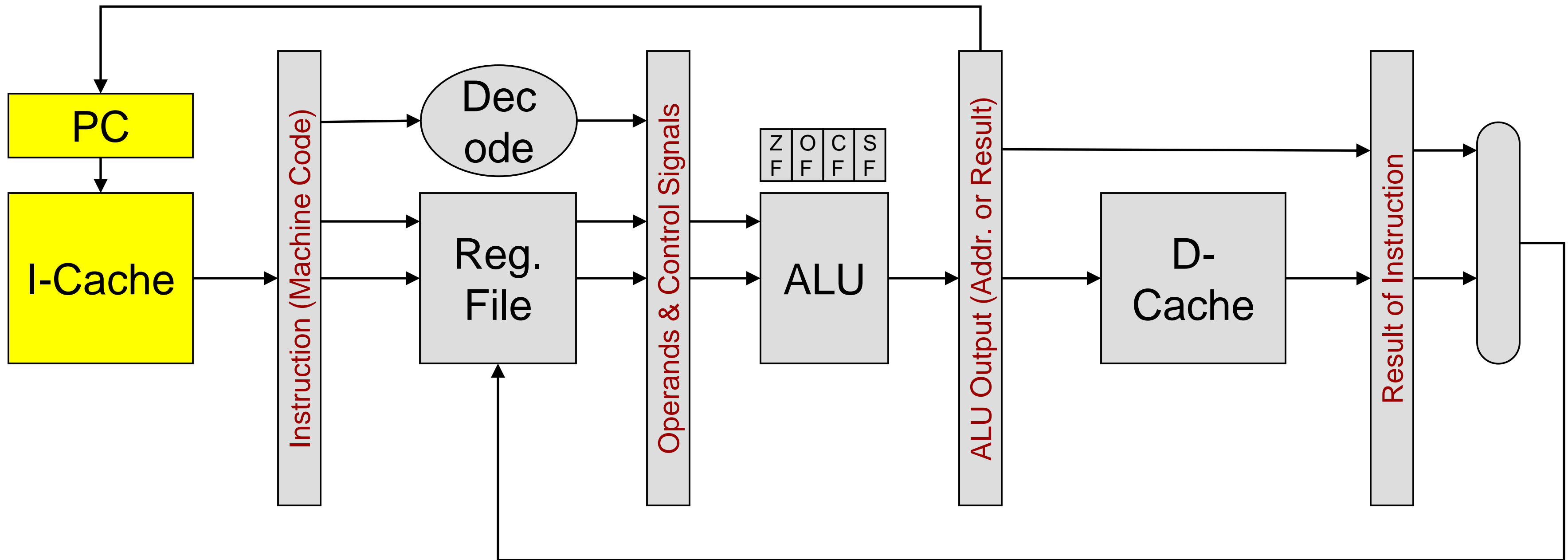
Fetch
(LD)

Decode

Exec.

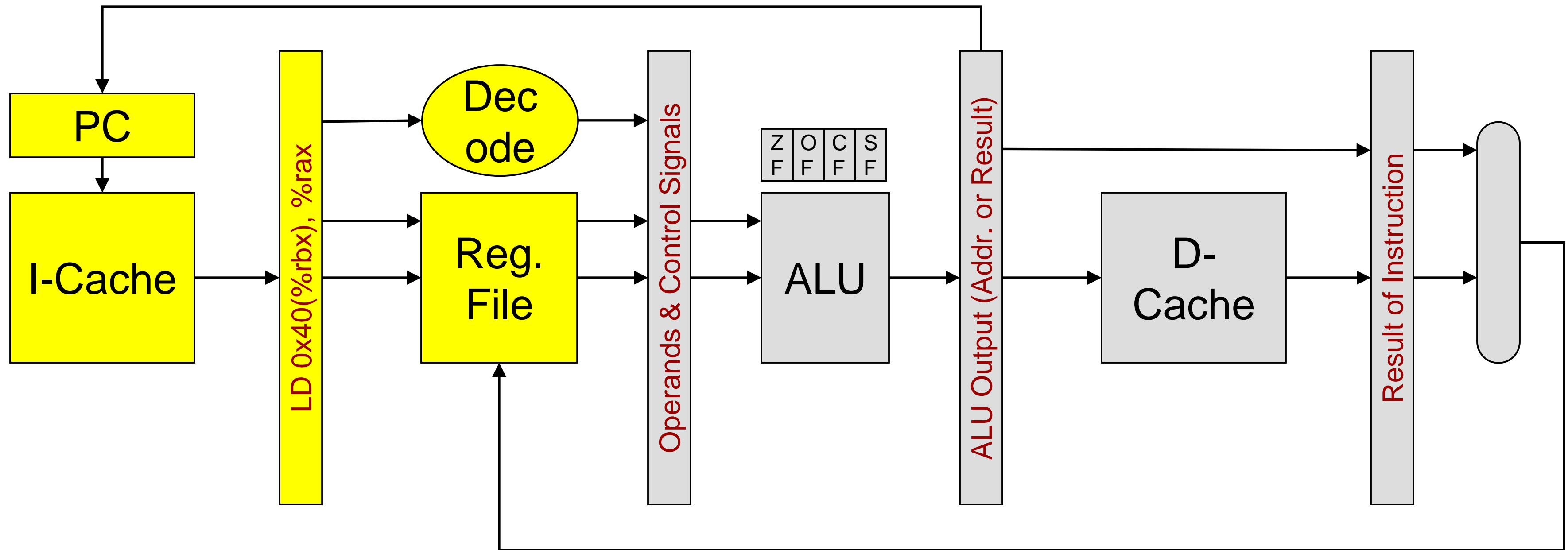
Mem

WB

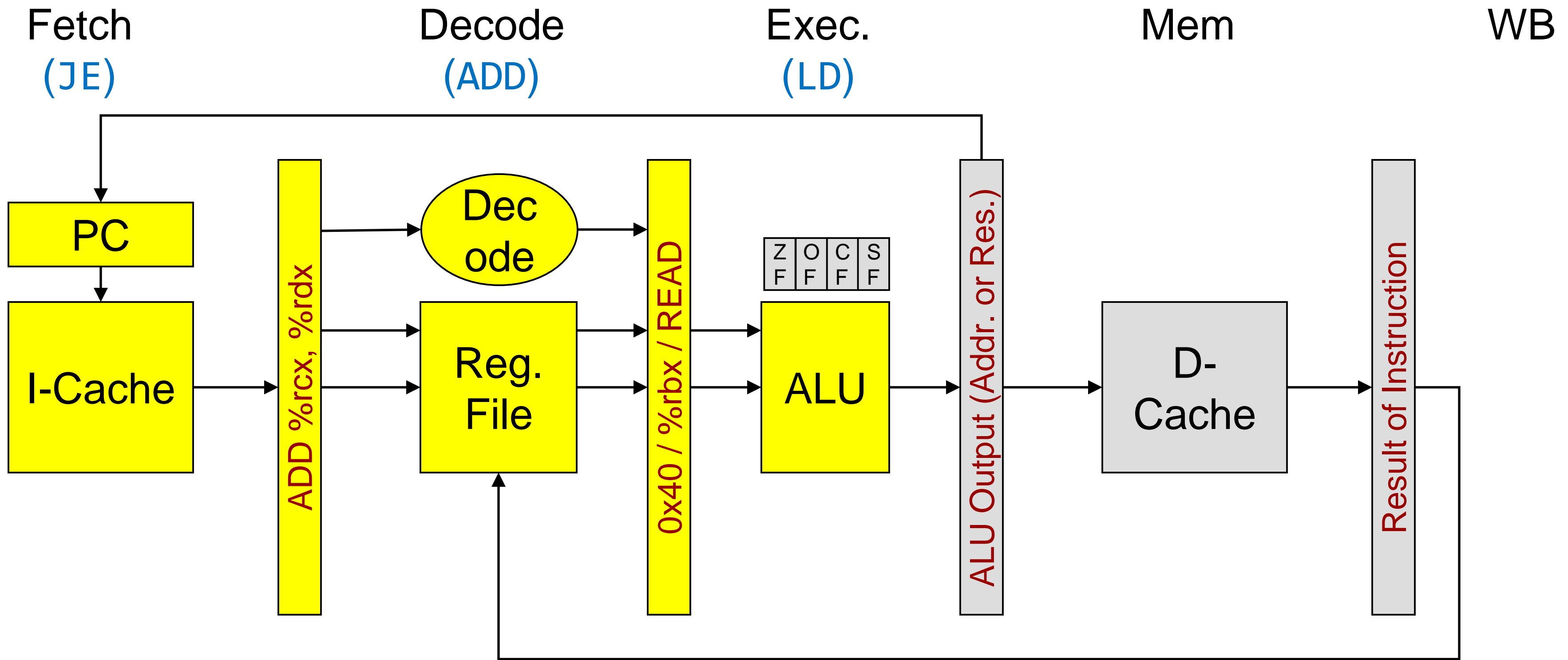


Fetch LD

Sample Sequence - 2



Sample Sequence - 3

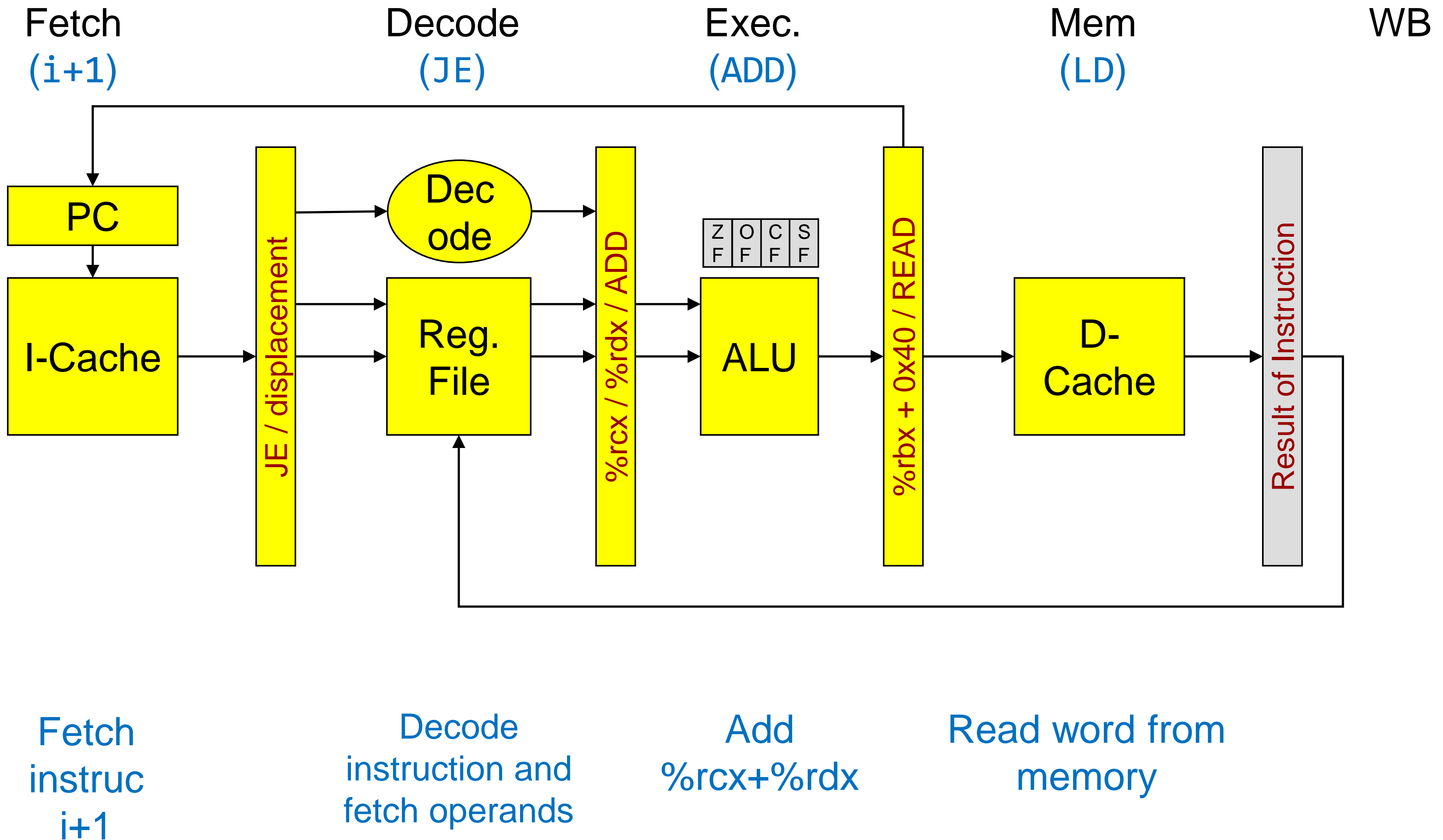


Fetch JE

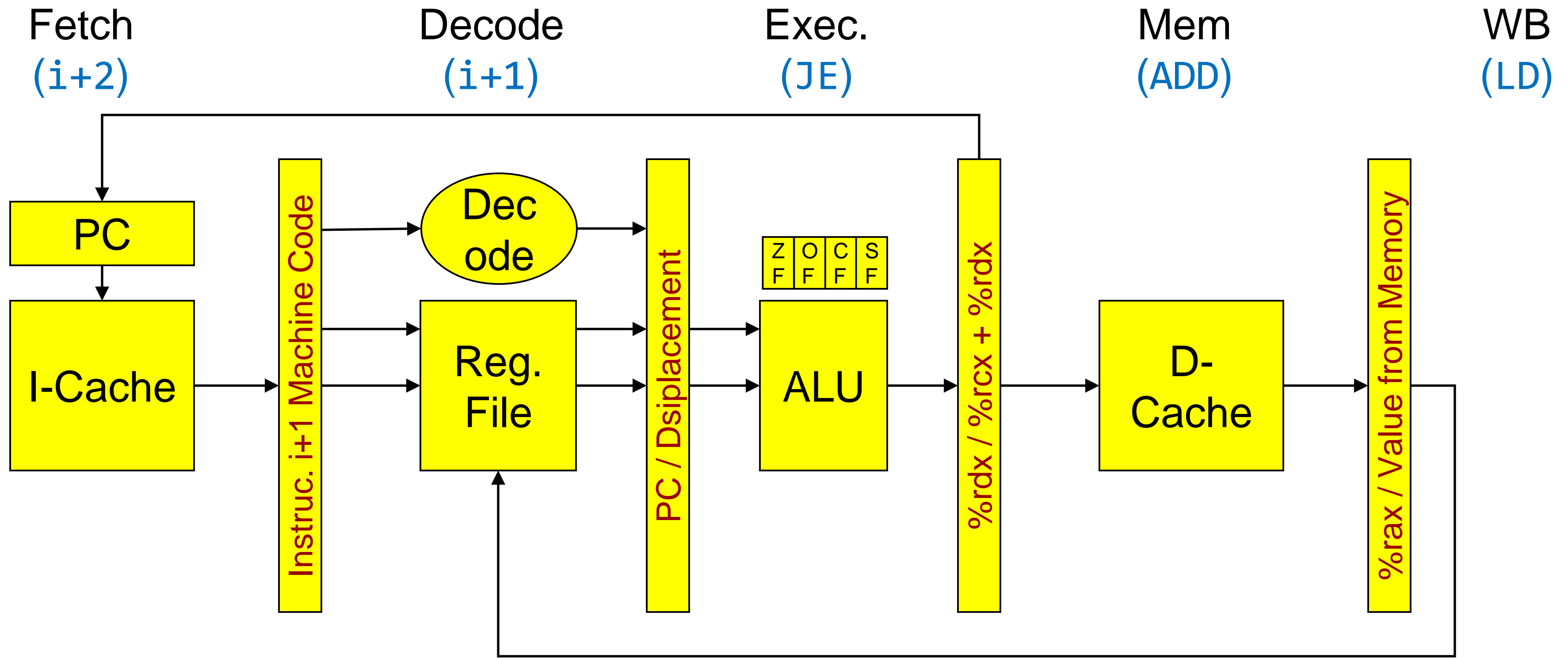
Decode instruction and fetch operands

Add displacement 0x40 to %rbx

Sample Sequence - 4



Sample Sequence - 5



Fetch next instruc i+2

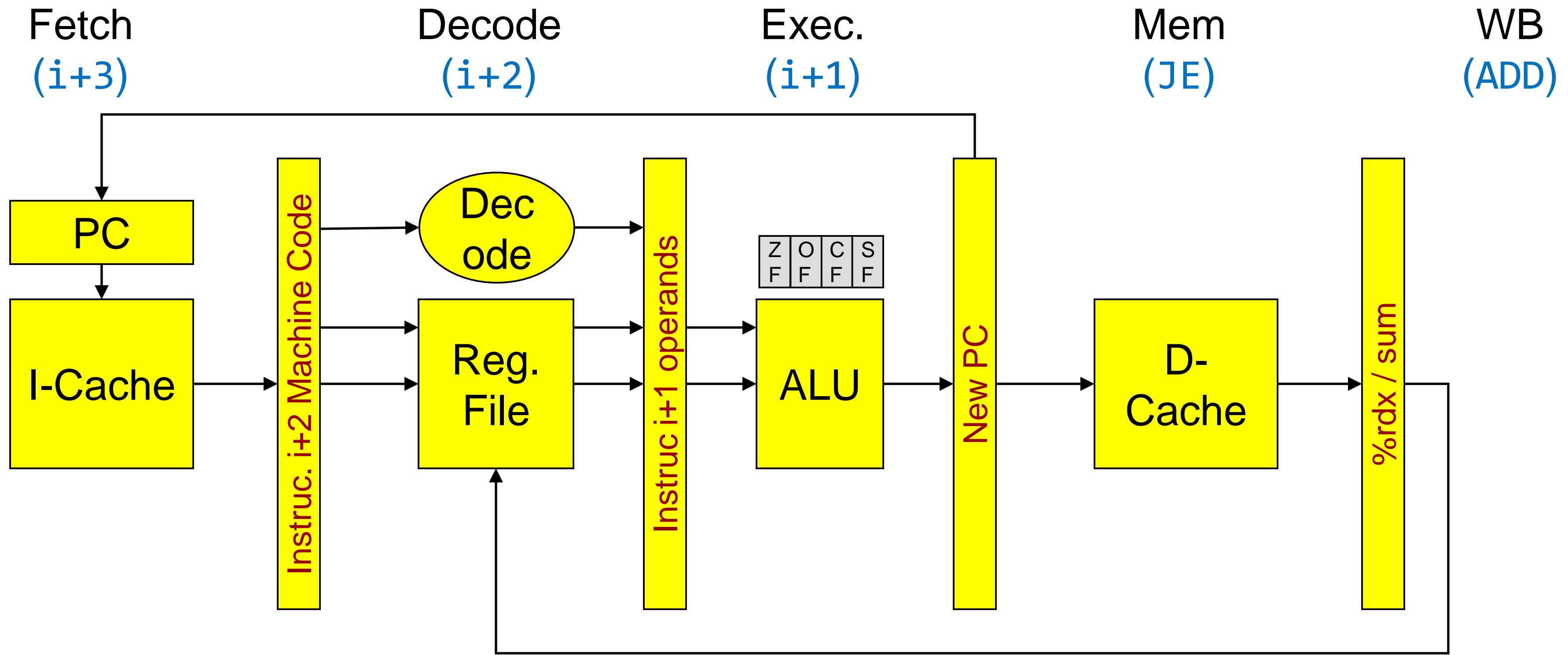
Decode instruction i+1 and fetch operands

Check if condition is true, add displ.

Just pass sum to next stage

Write word to %rax

Sample Sequence - 6



Fetch next instruc i+3

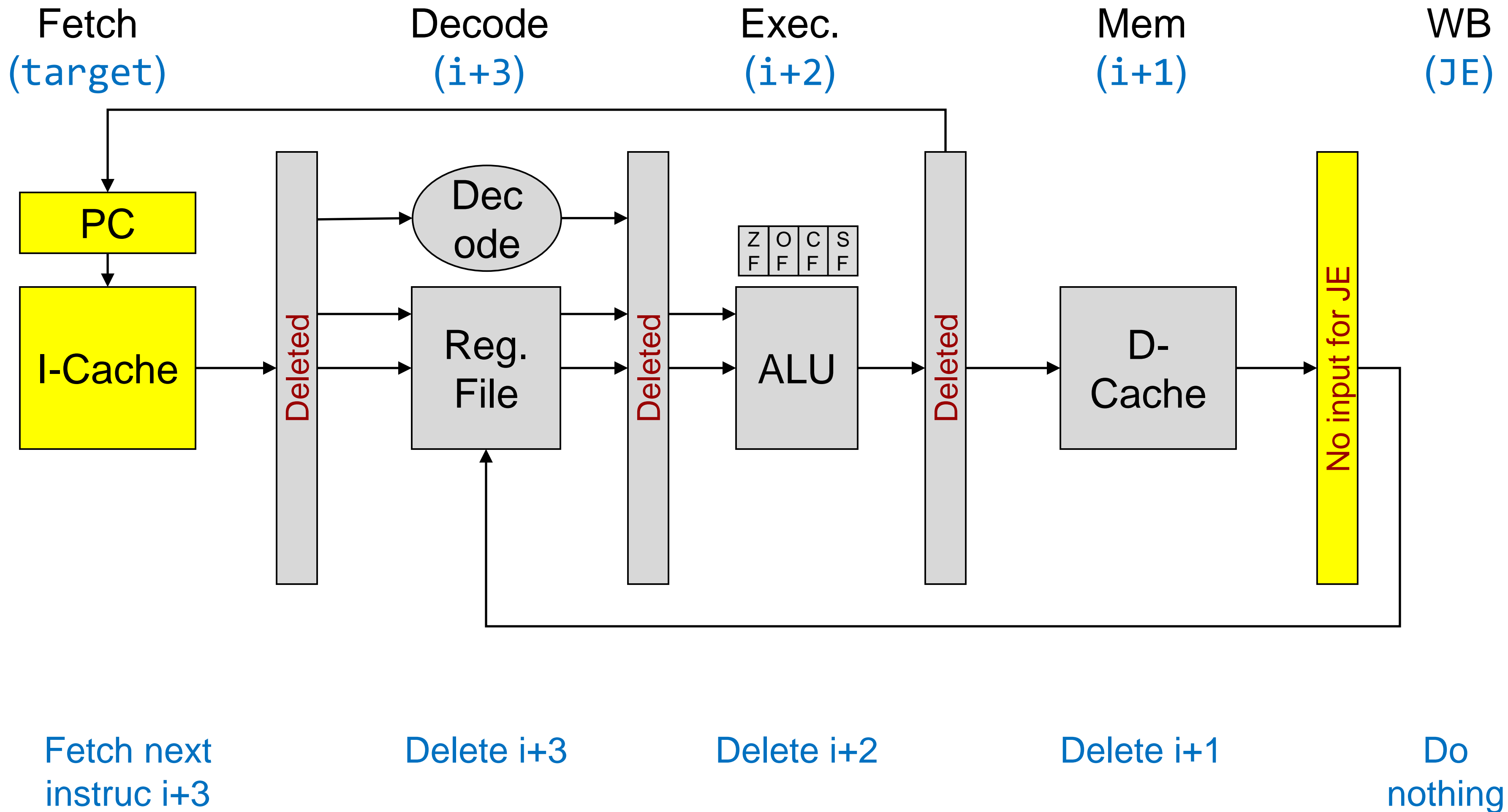
Decode instruction i+2 and fetch operands

Use the ALU

Update PC

Write word to %rdx

Sample Sequence - 7



Problems from overlapping instruction execution…

HAZARDS

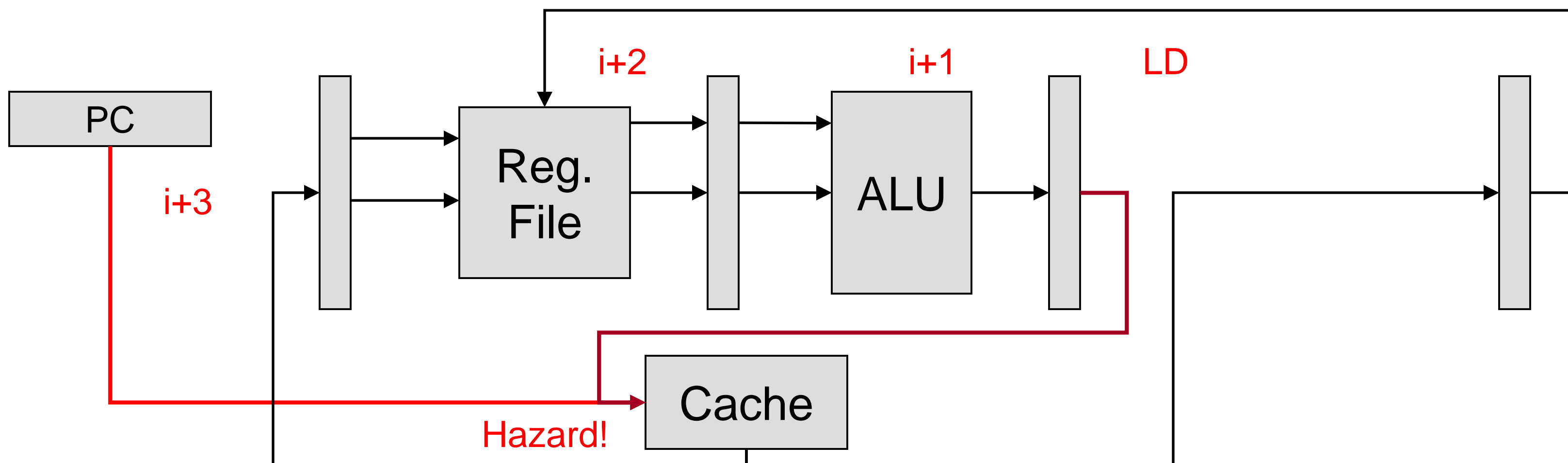
Hazards

Hazards prevent parallel or ____________________ execution!

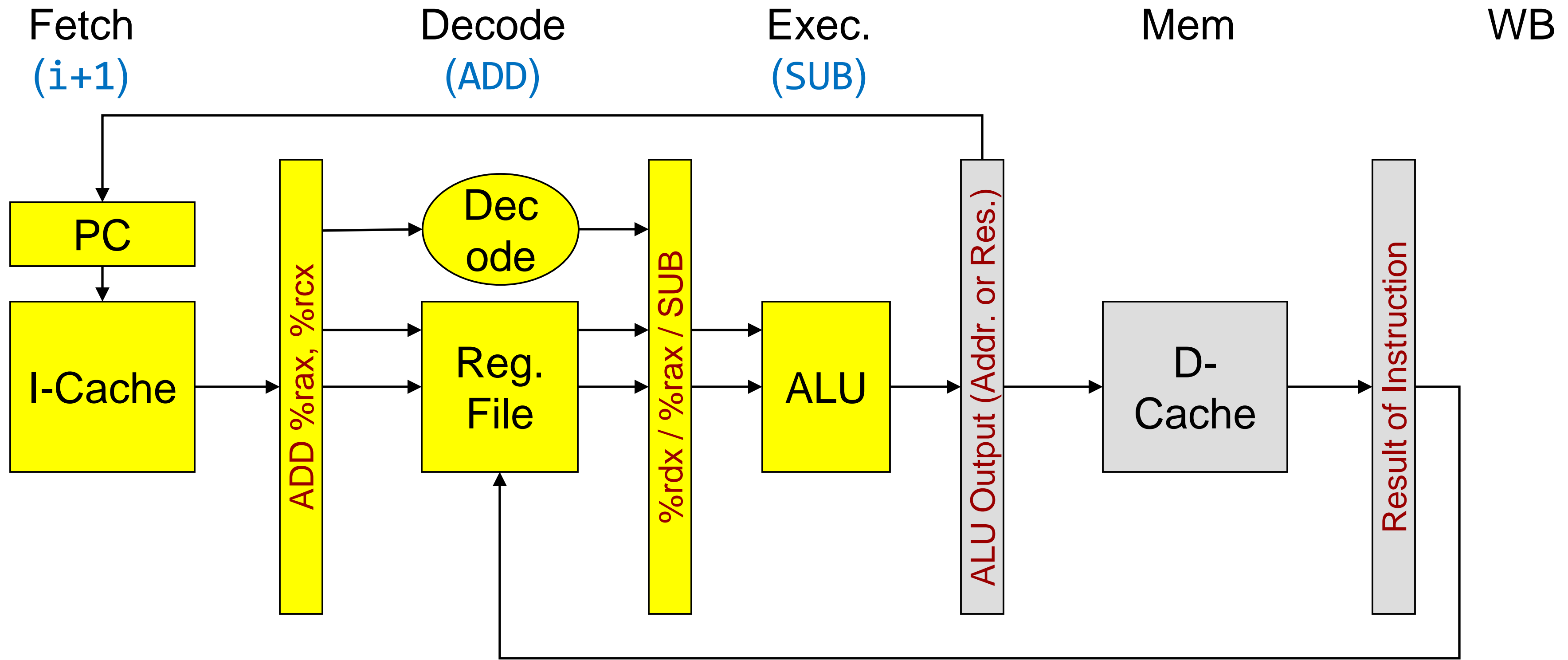
- ___________________ **Hazards**
  - Problem: We don't know what instruction __________ but we need to
  - Examples: ______________________________ calls
- **Data Hazards** / Data Dependencies
  - Problem: When a later instruction needs data from a ___________ instruction
  - Examples:
    - `sub %rdx,%rax`
    - `add %rax,%rcx`
- __________________ **Hazards**
  - Problem: Due to limited _______________, the HW doesn't support overlapping a certain _________________ of instructions
  - Examples: See next slides

Structural Hazards

- Example structural hazard: A single cache rather than separate instruction & data caches
  - Structural hazard any time an instruction needs to perform a data access (i.e. `ld` or `st`) since we always want to fetch a new instruction each clock cycle



Data Hazard - 1

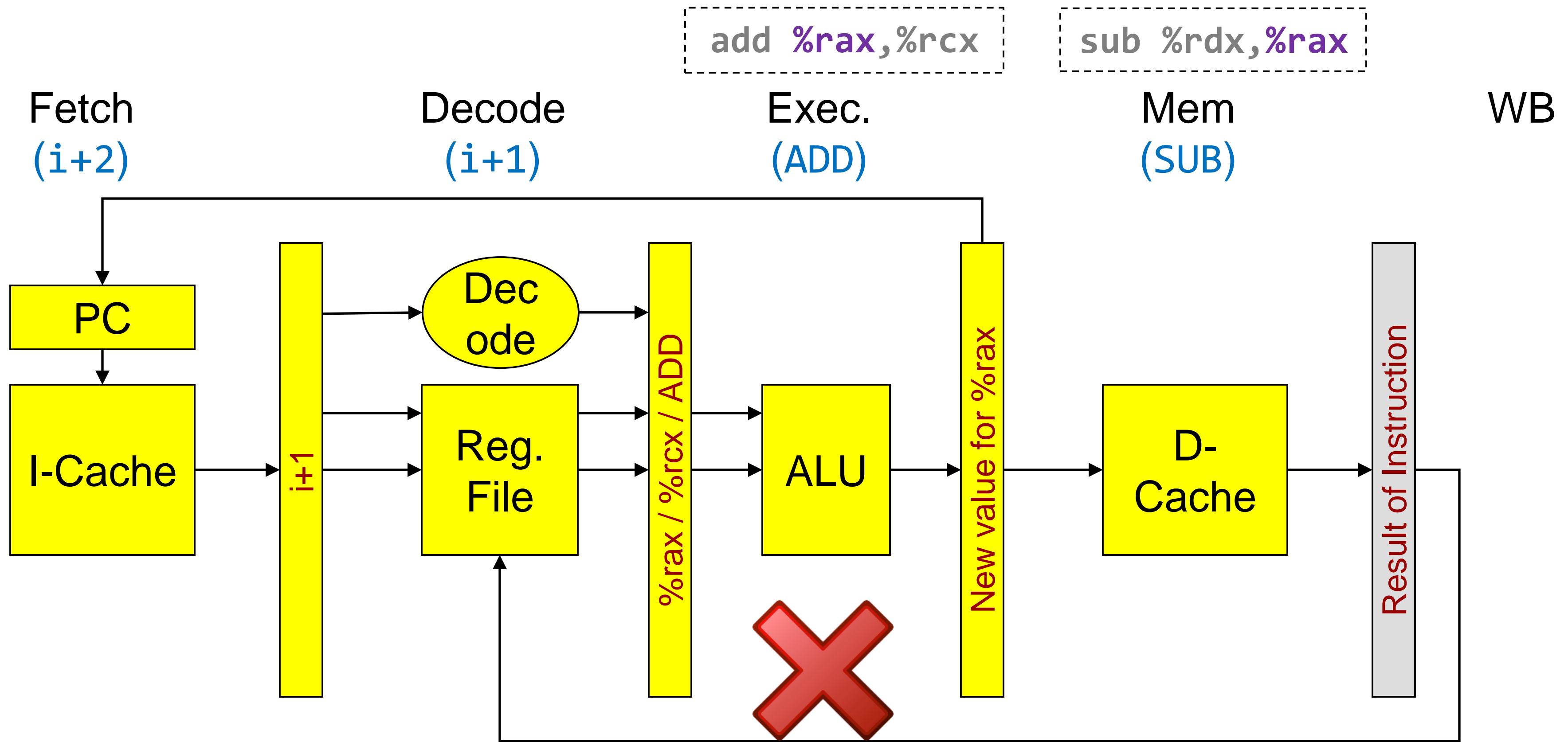


Fetch i+1

Decode and get register operands (Do we get the desired %rax value?)

Perform %rax-%rdx

Data Hazard - 2



Fetch i+2

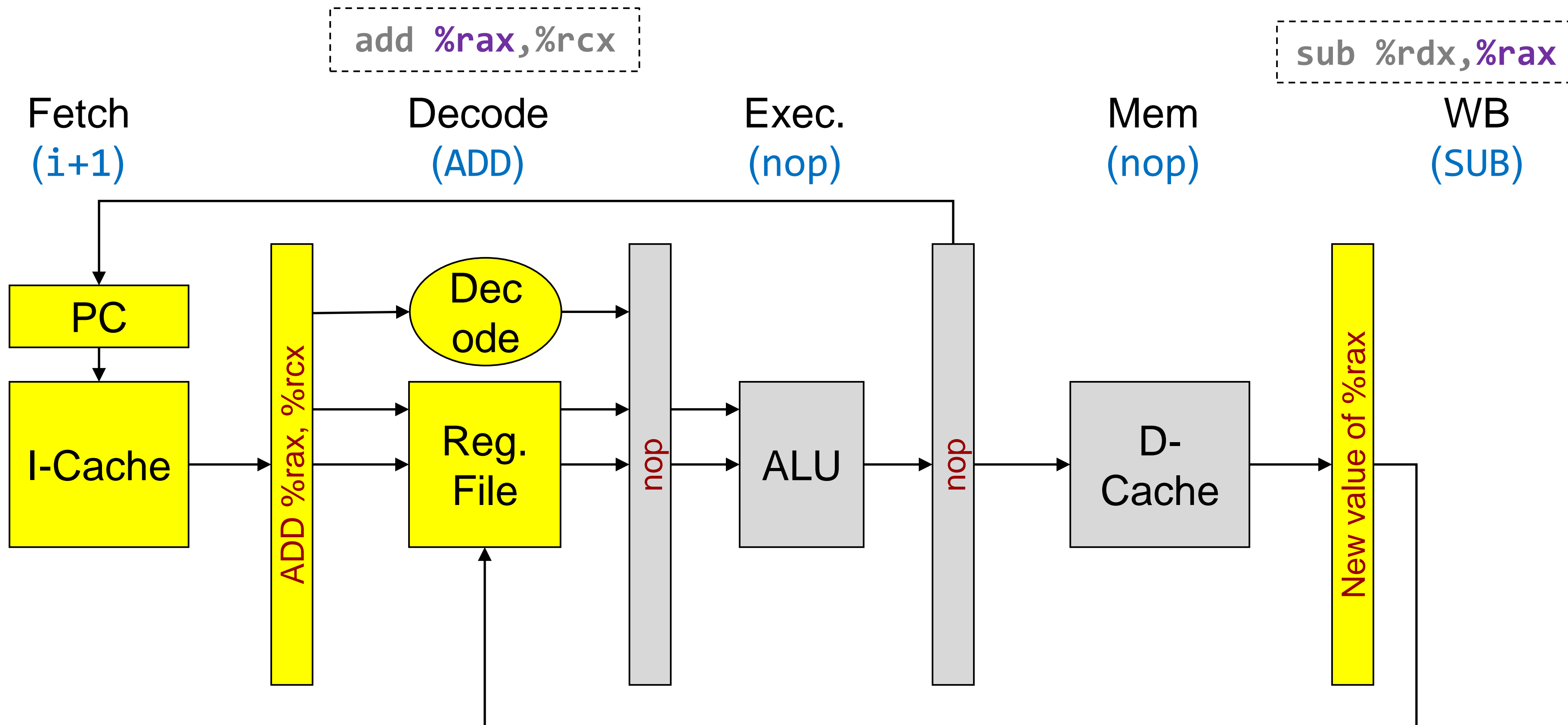
Decode i+1

Perform %rax+%rcx using the wrong value!

New value for %rax has not been written back yet

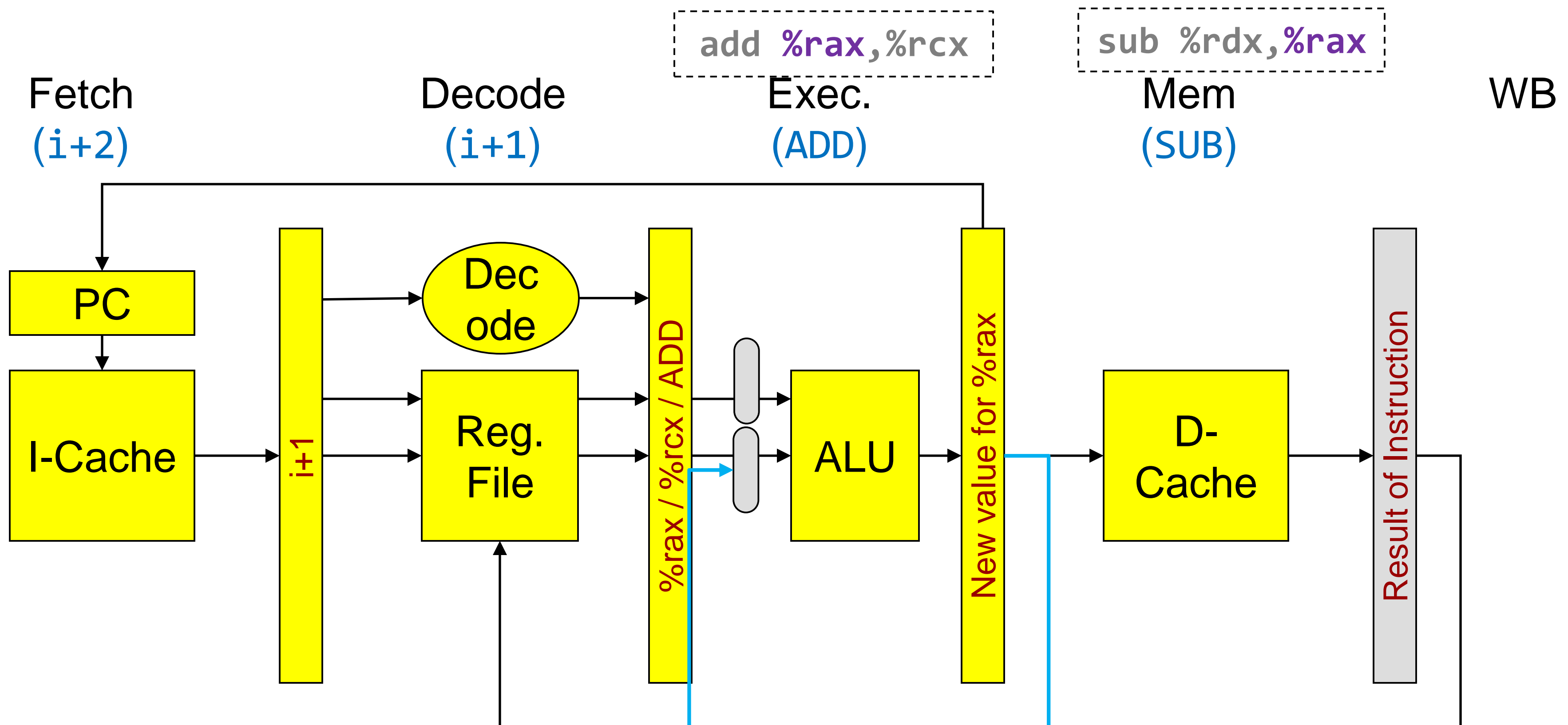
Stalling

- Solution 1: _________ the ADD instruction in the DECODE stage and insert ________ into the pipeline until the new value of the needed register is present at the cost of lower performance



Forwarding

- Solution 2: Create new hardware paths to hand-off (________) the data from the __________ instruction in the pipeline to the _________________ instruction

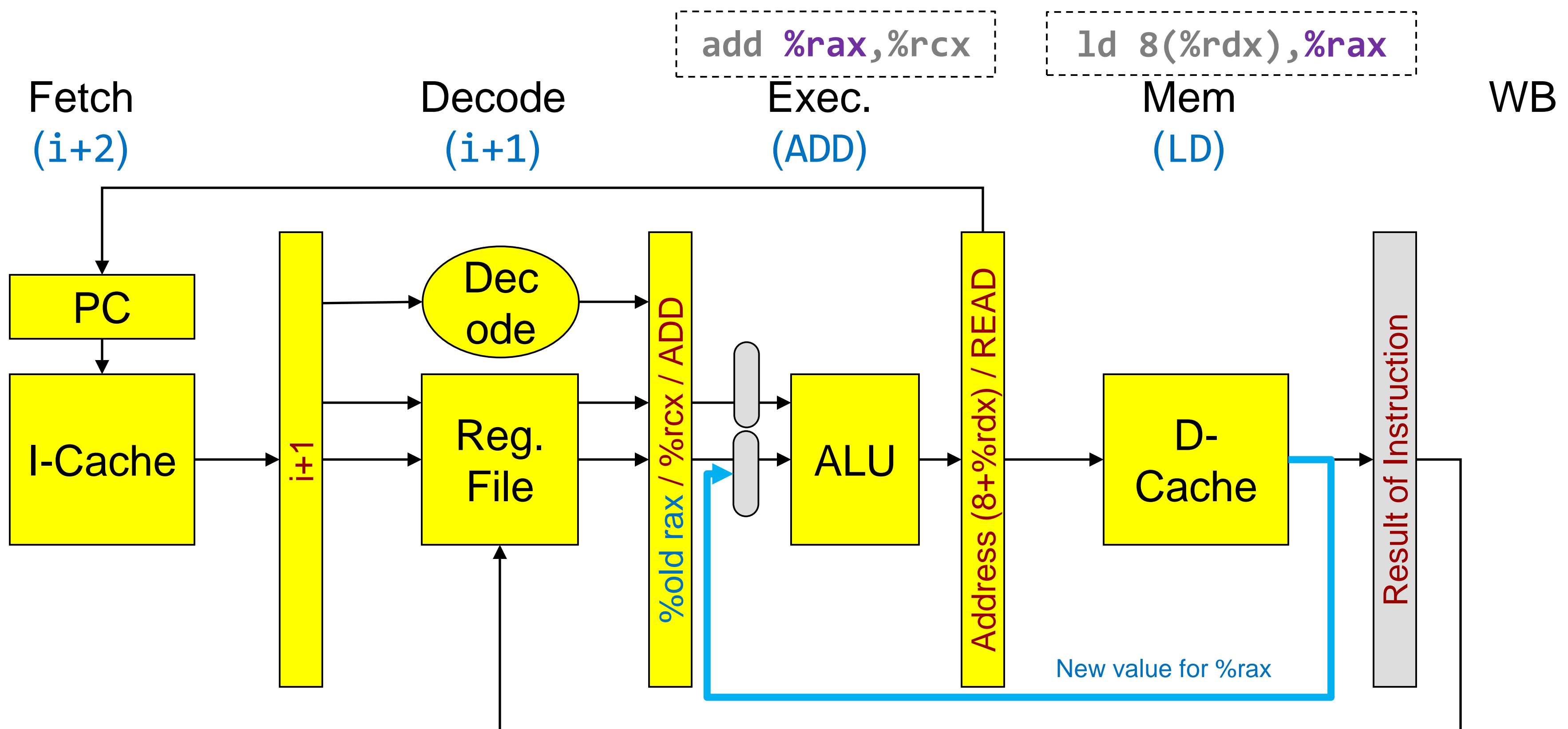


Solving Data Hazards

- **Key Point:** Data dependencies (i.e., instructions needing values produced by earlier ones) limit performance
- Forwarding solves many of the data hazards (data dependencies) that exist
  - It allows instructions to continue to flow through the pipeline without the need to stall and waste time
  - The cost is additional hardware and added complexity
- Even forwarding cannot solve all the issues
  - A structural hazard still exists when a _____________ a value needed by the next instruction

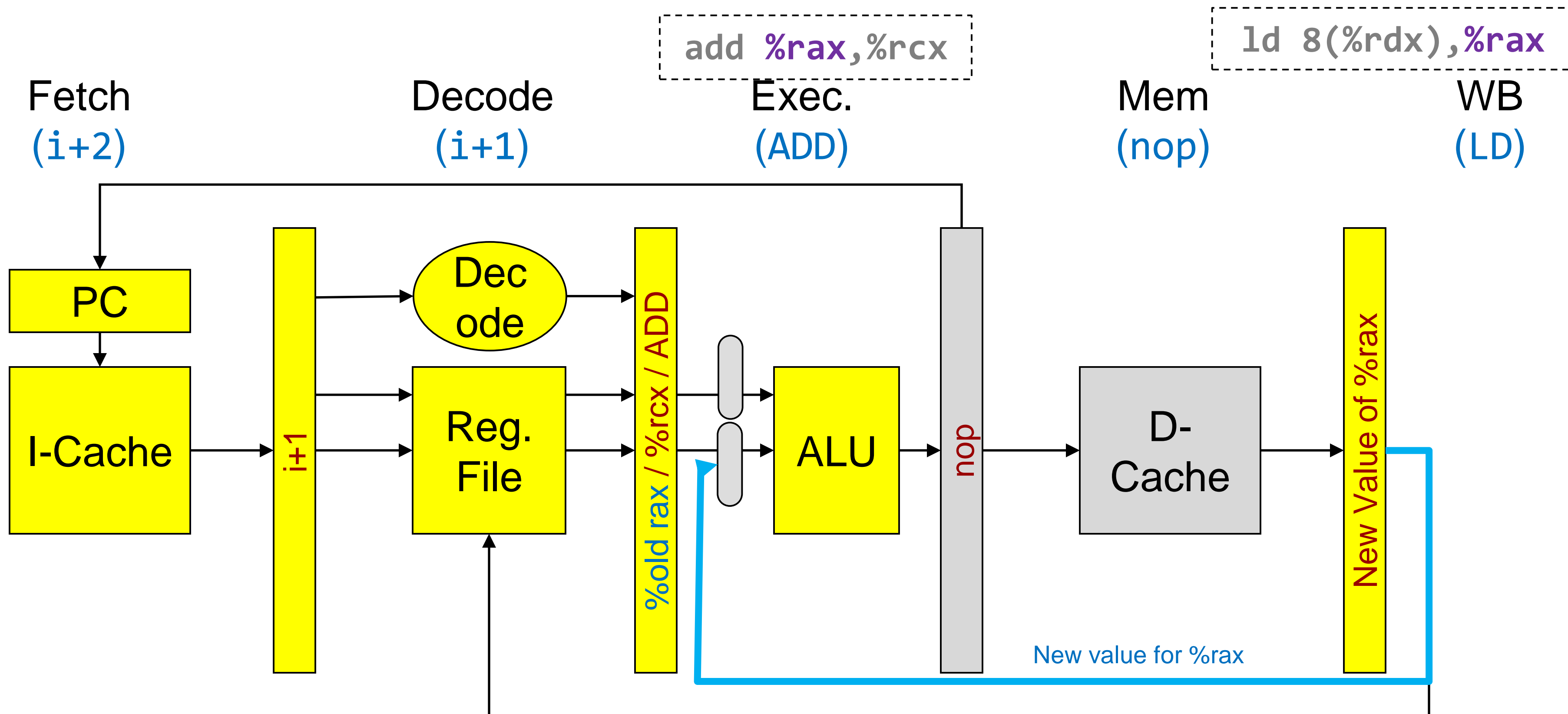
LD + Dependent Instruction Hazard

- Even forwarding cannot prevent the need to stall when a **ld** instruction produces a value needed by the instruction behind it
  - Would require performing ____________ worth of work in only a single cycle



LD + Dependent Instruction Hazard

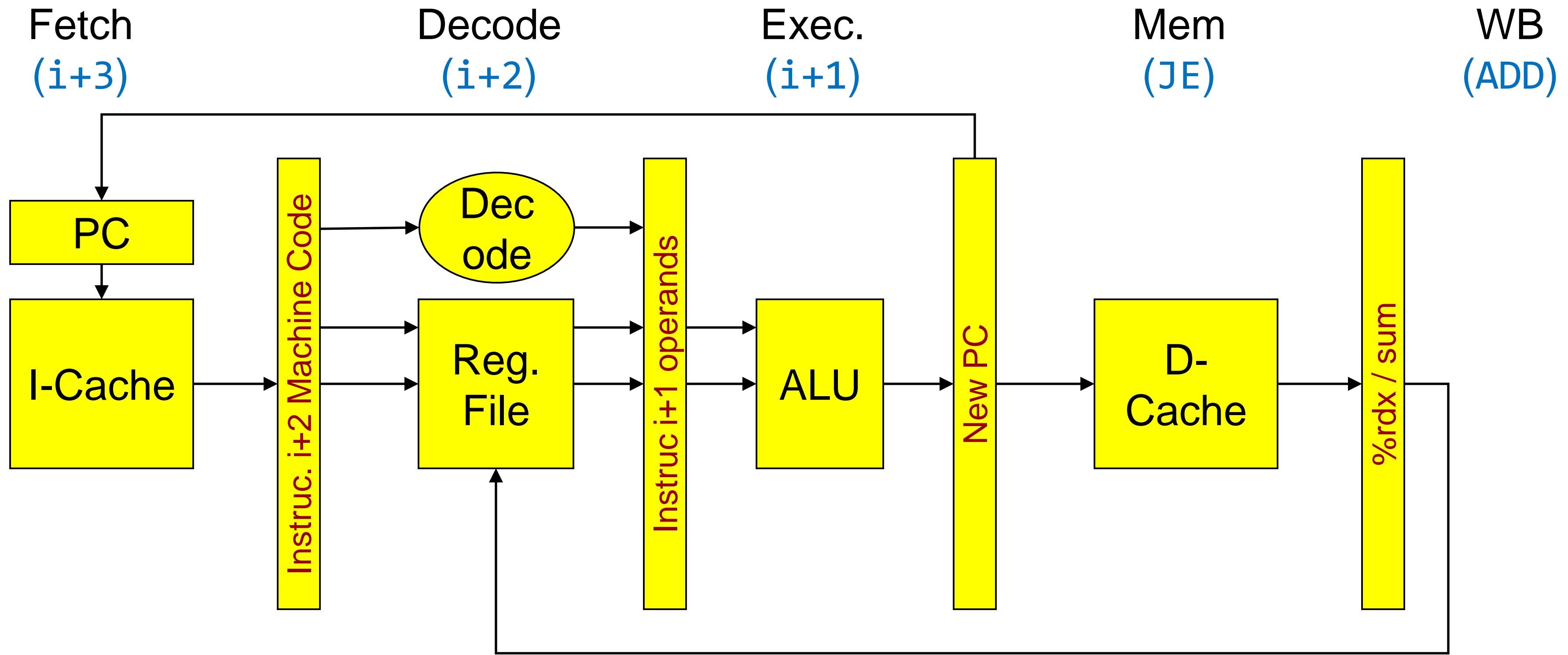
- We would need to introduce ____ stall cycle (nop) into the pipeline to get the timing correct
- Keep this in mind as we move through the next slides



Control Hazards

- Branches/Jumps require us to know
  - ___________ we want to jump to (aka branch/jump target location)… really just the new value of the PC
  - If we _______________ branch or not (checking the jump _____________)
- **Problem:** We often don't know those values until deep in the pipeline and thus we are not sure what instructions should be fetched in the interim
  - Requires us to ___________________ unwanted instructions and waste time

Control Hazard - 1



Fetch next instruc i+3

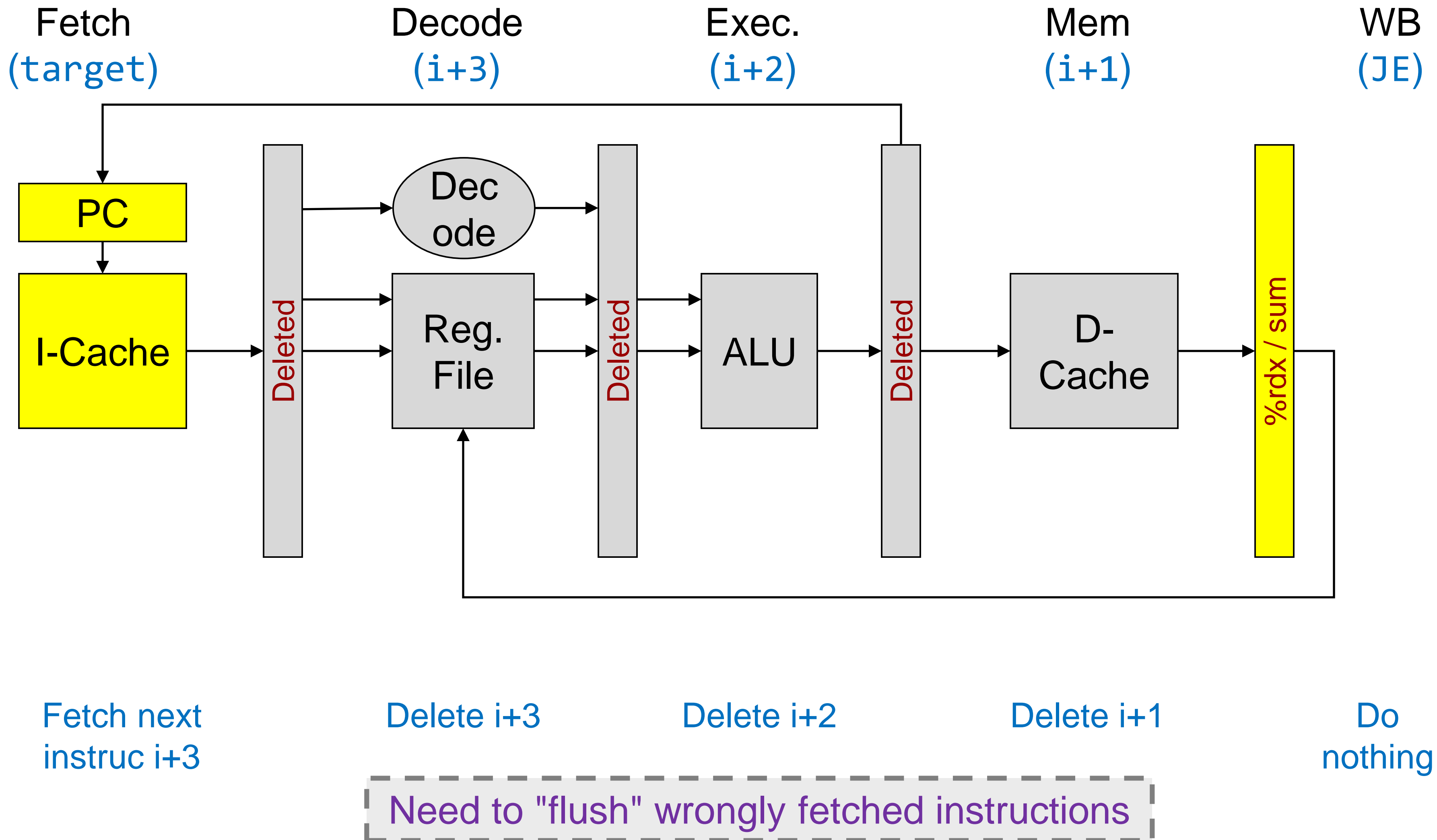
Decode instruction i+2 and fetch operands

Use the ALU

Update PC

Write word to %rdx

Control Hazard - 2



Need to "flush" wrongly fetched instructions

Enlisting the help of the compiler

A FIRST LOOK: CODE REORDERING

Two Sides of the Coin

- If the hardware has some problems it just can't solve, can software (i.e., the compiler) help?
  - __________
- Compilers can ______________ instructions to take best advantage of the processor (pipeline) organization
- Identify the dependencies that will incur stalls and slow performance
  - _________________
  - Jump instructions

```
void sum(int *data, int n, int x) {
  for(int i=0; i < n; i++){
    data[i] += x;
  }
}
```

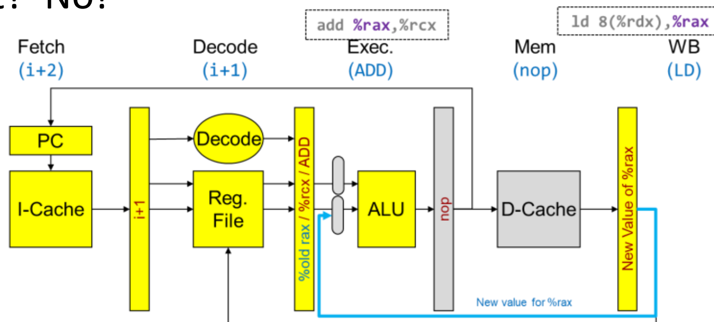
```
sum:
  mov    $0x0,%ecx
L1:
  cmp    %esi,%ecx
  jge    L2
  ld     0(%rdi), %eax
  add    %edx, %eax
  st     %eax, 0(%rdi)
  add    $4, %rdi
  add    $1, %ecx
  j      L1
L2:
  retq
```

C code and its assembly translation

USC Viterbi
School of Engineering

How Can the Compiler Help

- Compilers are written with general parsing and semantic representation front ends but ____________________ backends that generate code optimized for a particular processor
- **Q:** How could the compiler help improve pipelined performance while still maintaining the external behavior that the high level code indicates
- **A:** By finding _______________ instructions and reordering the code
  - Could we have moved any other instruction into that slot? No!




```
sum:
  mov    $0x0,%ecx
L1:
  cmp    %esi,%ecx
  jge    L2
  ld     0(%rdi), %eax
   stall/nop
  add    %edx, %eax
  st     %eax, 0(%rdi)
  add    $4, %rdi
  add    $1, %ecx
  j      L1
L2:
  retq
```

Original Code (incurring 1 stall cycle)

```
sum:
  mov    $0x0,%ecx
L1:
  cmp    %esi,%ecx
  jge    L2
  ld     0(%rdi), %eax
  add    $1, %ecx
  add    %edx, %eax
  st     %eax, 0(%rdi)
  add    $4, %rdi
  j      L1
L2:
  retq
```

Updated Code (w/ Compiler reordering)

Taken or Not Taken: Branch Behavior

- When a conditional jump/branch is
  - True, we say it is __________
  - False, we say it is __________
- Currently our pipeline will fetch sequentially and then potentially flush if the branch is taken
  - Effectively, our pipeline "_________" that each branch is ________________
- The `j L1` instruction is always __________ and thus will incur wasted clock cycles each time it is executed
- Most of the time the `jge L2` will be _________________ and perform well

```
sum:
   mov    $0x0,%ecx
L1:
   cmp    %esi,%ecx
   jge    L2
   ld     0(%rdi), %eax
   add    $1, %ecx
   add    %edx, %eax
   st     %eax, 0(%rdi)
   add    $4, %rdi
   j      L1
L2:
   retq
```

T

NT

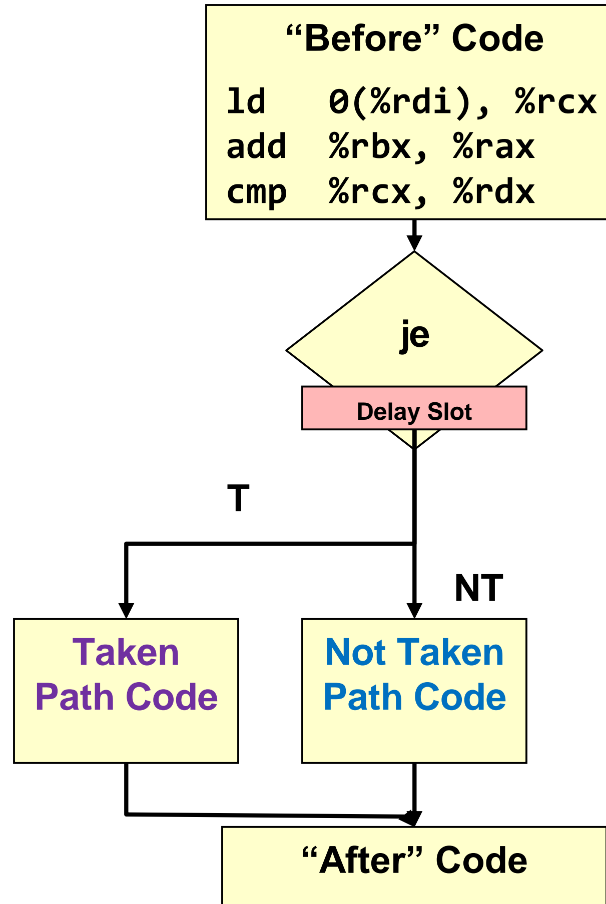
Branch Delay Slots

- **Problem:** After a jump/branch we fetch instructions that we are not sure should be executed
- **Idea:** Find an instruction(s) that should ___________ be executed (independent of whether branch is taken or not), move those instructions to directly ________ the branch, and have HW just let them be executed (not flushed) no matter what the branch outcome is
- **Branch delay slots** = __________________ that the HW will always execute (not flush) after a jump/branch instruction

Branch Delay Slot Example

```
ld    0(%rdi), %rcx
add   %rbx, %rax
cmp   %rcx, %rdx
je    NEXT
delay slot instruc.
NOT TAKEN CODE
…
NEXT:
TAKEN CODE
```

Assume a single instruction delay slot



Flowchart perspective of the delay slot

```
ld    0(%rdi), %rcx
cmp   %rcx, %rdx
je    NEXT
add   %rbx, %rax      Delay Slot
NOT TAKEN CODE
…
NEXT:
TAKEN CODE
```

Move an ALWAYS executed instruction down into the delay slot and let it execute no matter what

Implementing Branch Delay Slots

- HW will define the number of branch delay slots (usually a small number…1 or 2)
- Compiler will be responsible for ________________ instructions to fill the delay slots
  - Must find instructions that the branch does NOT DEPEND on
  - If no instructions can be rearranged, can always insert 'nop' and just waste those cycles

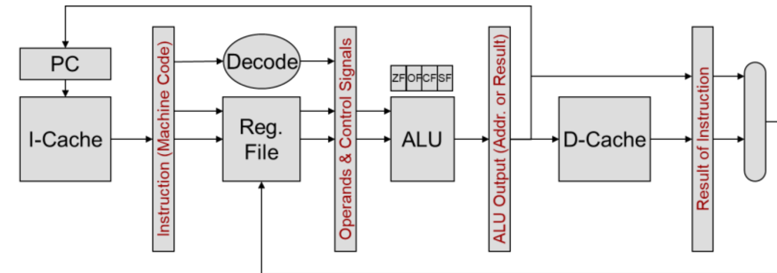
```
ld    0(%rdi), %rcx
add   %rbx, %rax
cmp   %rcx, %rdx
je    NEXT
delay slot instruc.
```

Cannot move 'ld' into delay slot because je needs the %rcx value generated by it

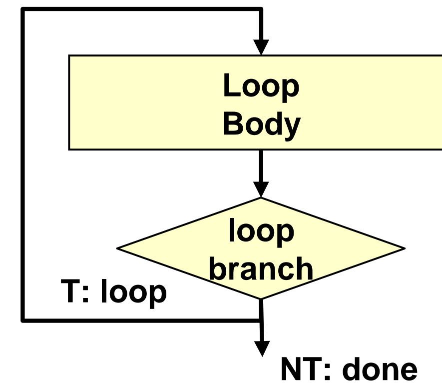
```
ld    0(%rdi), %rcx
add   %rbx, %rax
cmp   %rcx, %rax
je    NEXT
delay slot instruc.
```

If no instruction can be found a 'nop' can be inserted by the compiler

A Look Ahead: Branch Prediction

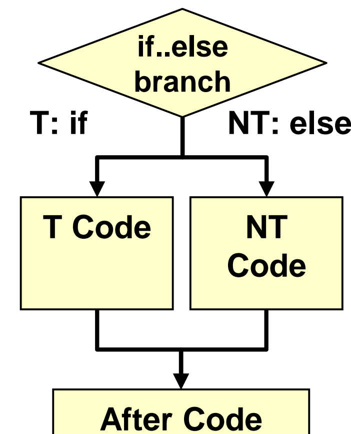


- Currently our pipeline assumes Not Taken and fetches down the sequential path after a jump/branch
- Could we build a pipeline that could predict taken?
  - __________! Location to jump to (branch target) not known until ___________________
- But suppose we could overcome those problems, would we even know how to predict the outcome of a jump/branch before actually looking at the condition codes deeper in the pipeline?
- We could allow a __________ prediction *per instruction* (give a _________ with the branch that indicates T or NT)
- We could allow _____________ prediction *per instruction* (use its ______________ history)



Loops

High probability of being Taken. Prediction can be static.



If Statements

May exhibit data dependent behavior. Prediction may need to be dynamic.

Summary 1

- Pipelining is an effective and important technique to improve the throughput of a processor
- Overlapping execution creates hazards which lead to stalls or wasted cycles
  - Data, Control, Structural
  - More hardware can be investigated to attempt to mitigate the stalls (e.g. forwarding)
- The compiler can help reorder code to avoid stalls and perform useful work (e.g. delay slots)