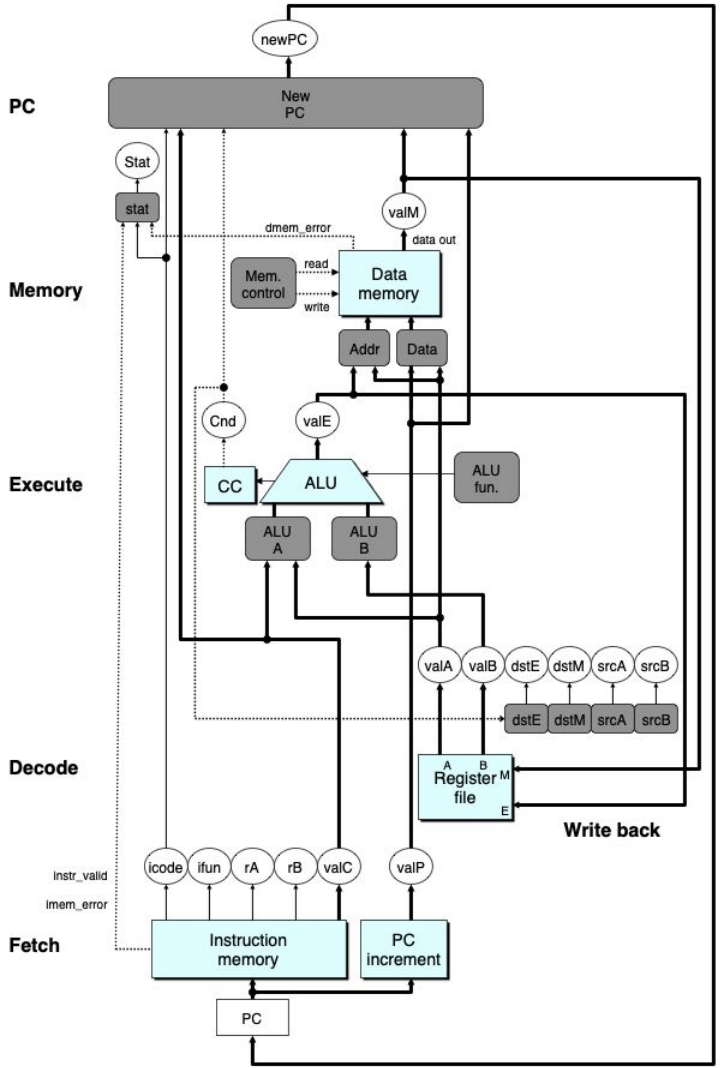
CSCI2021 Lecture 25
Mar 28, 2025

Y86 Execution Trace, Pipeline Basics

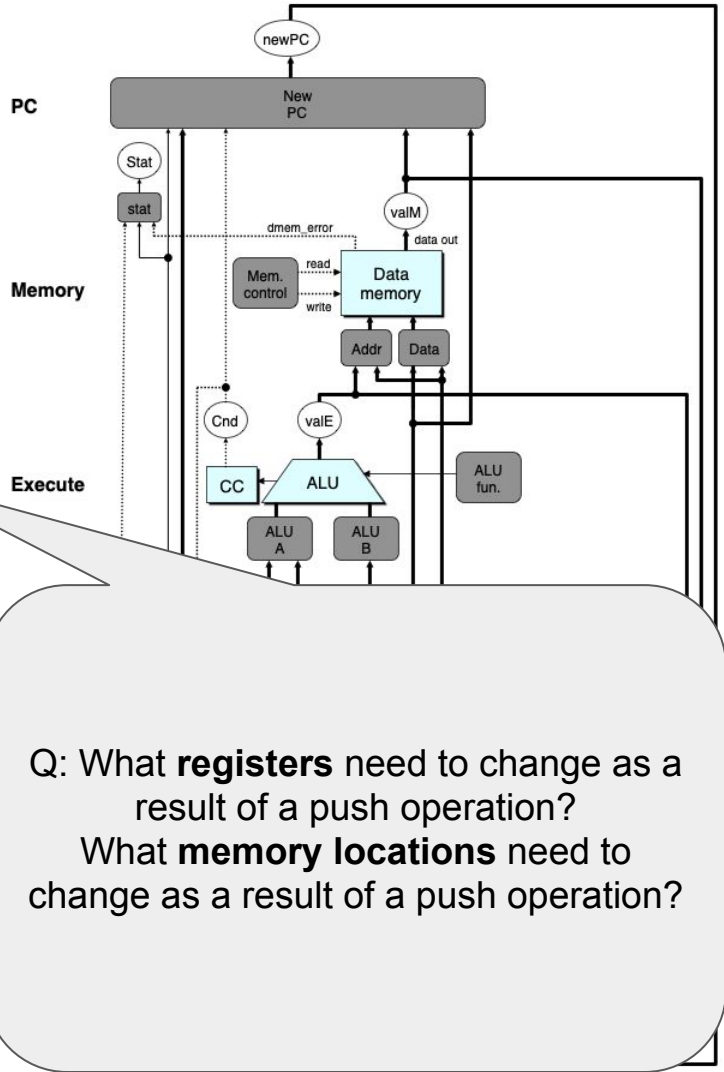
Exercise: Pop and Push

	popq rA	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	
Decode	valA ← R[%rsp] valB ← R[%rsp]	
Execute	valE ← valB + 8	
Memory	valM ← M_8[valA]	
Write back	R[%rsp] ← valE R[rA] ← valM	
PC Update	PC ← valP	



Exercise: Pop and Push

	popq rA	pushq rA
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	
Decode	valA ← R[%rsp] valB ← R[%rsp]	
Execute	valE ← valB + 8	
Memory	valM ← M_8[valA]	
Write back	R[%rsp] ← valE R[rA] ← valM	
PC Update	PC ← valP	

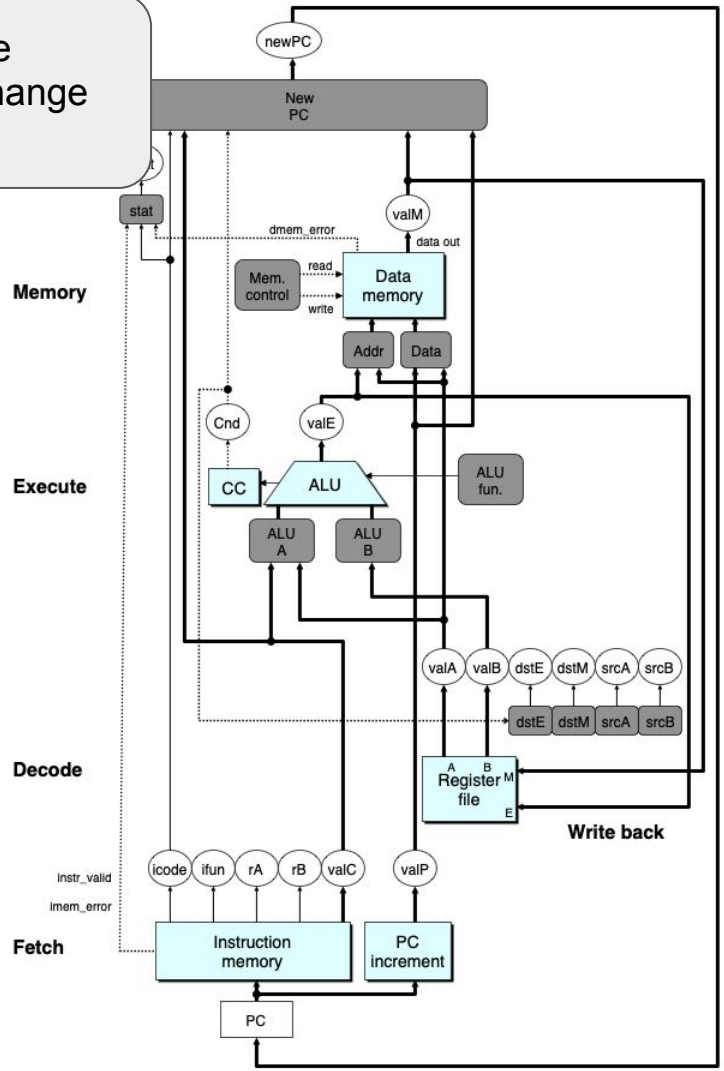


Q: What **registers** need to change as a result of a push operation?
What **memory locations** need to change as a result of a push operation?

Exercise: Pop and

%rsp needs to change
M[R[%rsp] - 8] needs to change
PC needs to change

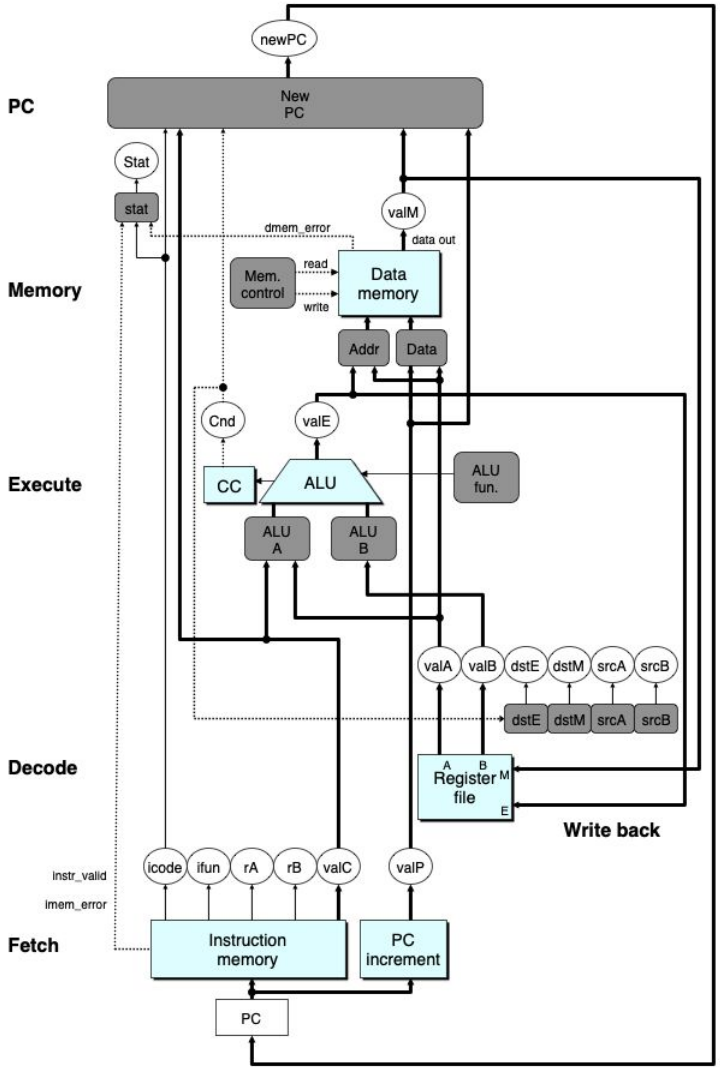
	popq rA	pushq rA
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	
Decode	valA ← R[%rsp] valB ← R[%rsp]	
Execute	valE ← valB + 8	
Memory	valM ← M_8[valA]	
Write back	R[%rsp] ← valE R[rA] ← valM	
PC Update	PC ← valP	



Exercise: Pop and Push

	`popq rA`	`pushq rA`
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2
Decode	valA ← R[%rsp] valB ← R[%rsp]	valA ← R[rA] valB ← R[%rsp]
Execute	valE ← valB + 8	valE ← valB + (-8)
Memory	valM ← M_8[valA]	M_8[valE] ← valA
Write back	R[%rsp] ← valE R[rA] ← valM	R[%rsp] ← valE
PC Update	PC ← valP	PC ← valP

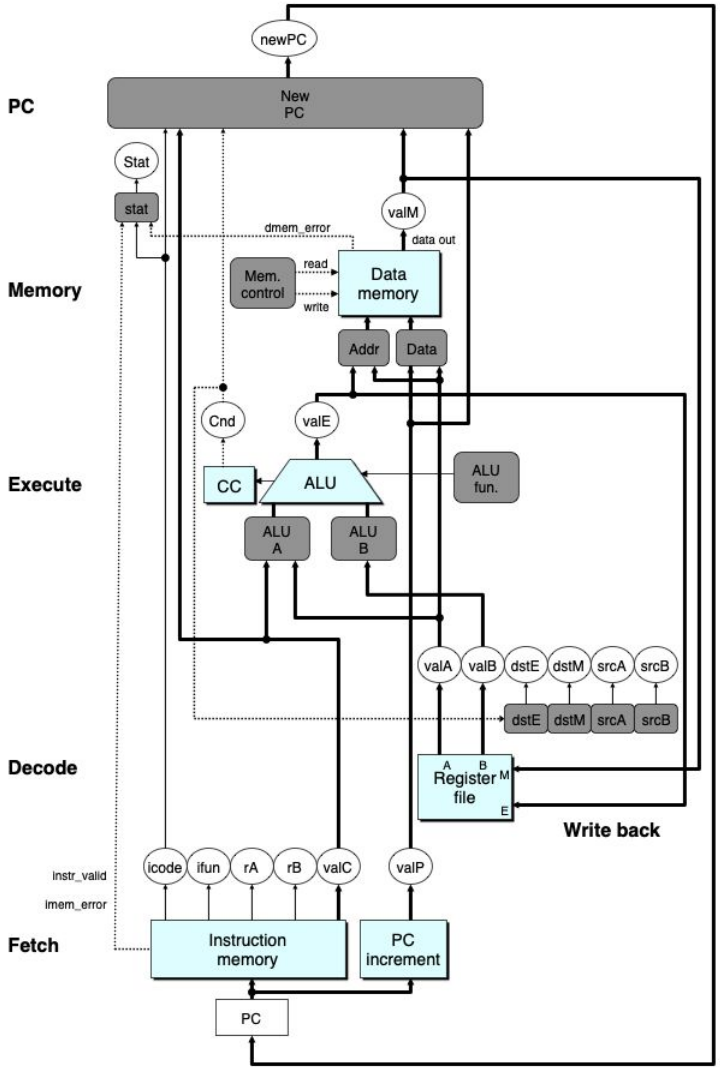
Complete the diagram for pushq rA



Exercise: Ret

	popq rA	ret
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	
Decode	valA ← R[%rsp] valB ← R[%rsp]	
Execute	valE ← valB + 8	
Memory	valM ← M_8[valA]	
Write back	R[%rsp] ← valE R[rA] ← valM	
PC Update	PC ← valP	

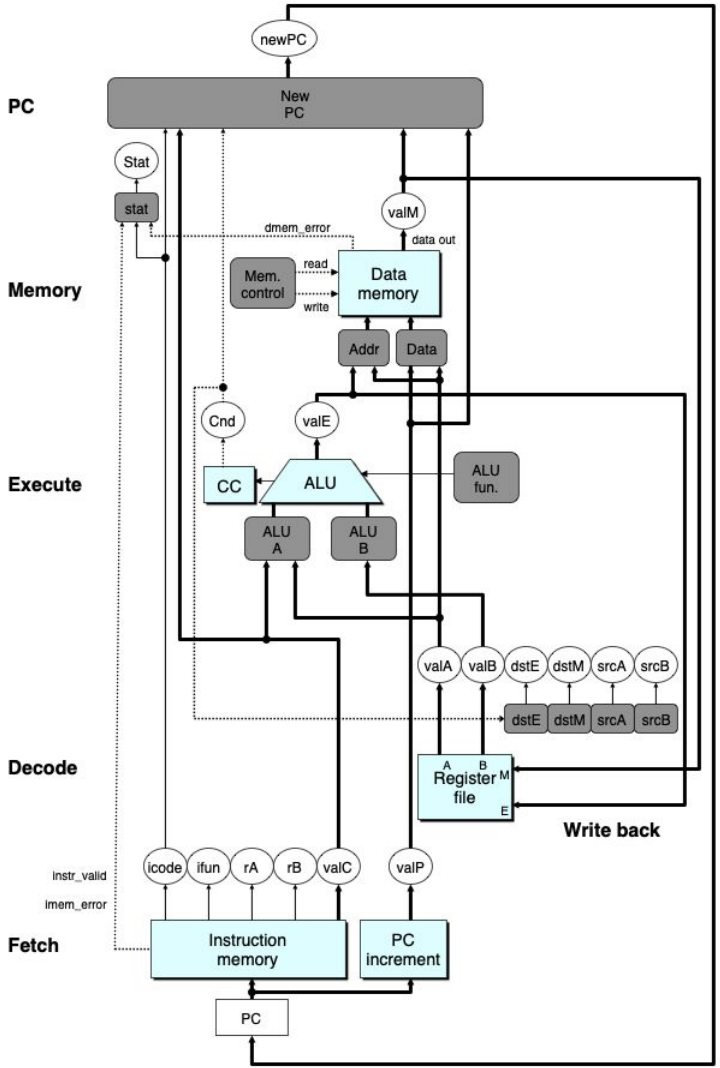
Complete the diagram for ret



Exercise: Ret

	popq rA	ret
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] valP ← PC + 1
Decode	valA ← R[%rsp] valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
Execute	valE ← valB + 8	valE ← valB + 8
Memory	valM ← M_8[valA]	valM ← M_8[valA]
Write back	R[%rsp] ← valE R[rA] ← valM	R[%rsp] ← valE
PC Update	PC ← valP	PC ← valM

Complete the diagram for ret



Exercise: Specifics

	`popq rA`	`popq %rsi`
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← _____ rA:rB ← _____ valP ← _____
Decode	valA ← R[%rsp] valB ← R[%rsp]	valA ← _____ valB ← _____
Execute	valE ← valB + 8	valE ← ____ + 8
Memory	valM ← M_8[valA]	valM ← _____
Write back	R[%rsp] ← valE R[rA] ← valM	R[%rsp] ← _____
PC Update	PC ← valP	PC ← _____

Fill in blanks with specific values

Some memory data:

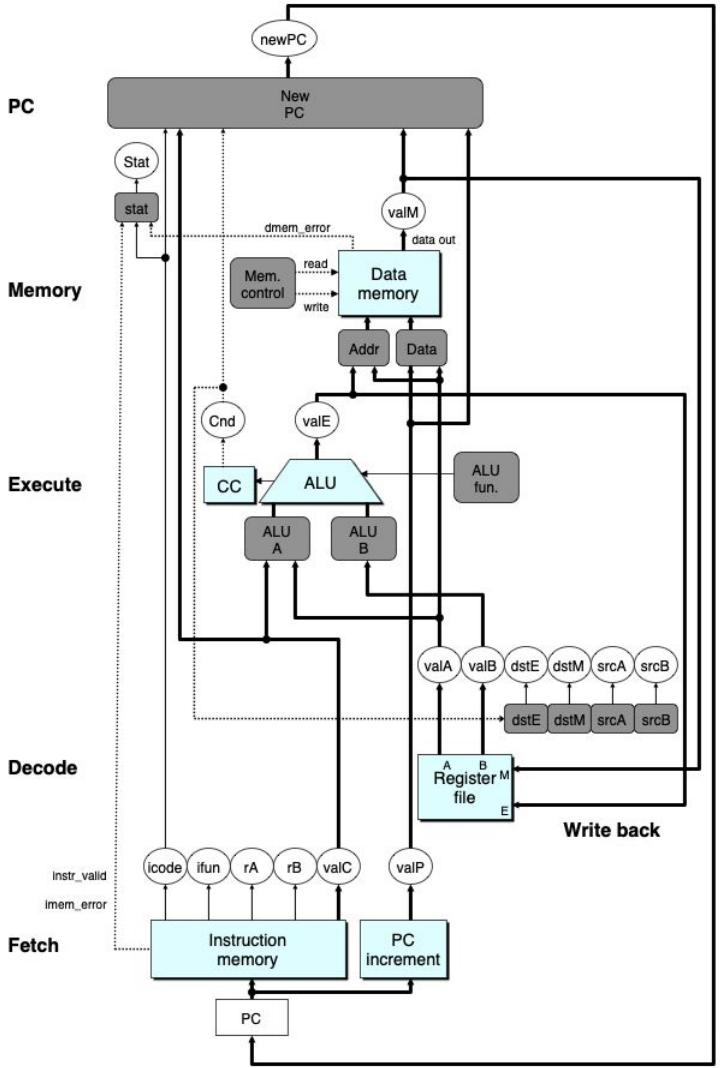
```
0xf018: 12 34 ab cd 77 ff 10 e3
0xf020: f7 ff ff ff ff ff ff ff
0xf028: 6c 01 00 00 00 00 00 00
                ...
0x8010: b0 6f b0 5f 90 10 60 12
0x8018: 80 10 80 00 00 00 00 00
                ...
```

Some register data:

```
PC:   0x8010
%rsp: 0xf020
%rsi:    0xa
```

Y86 SEQ: Timing

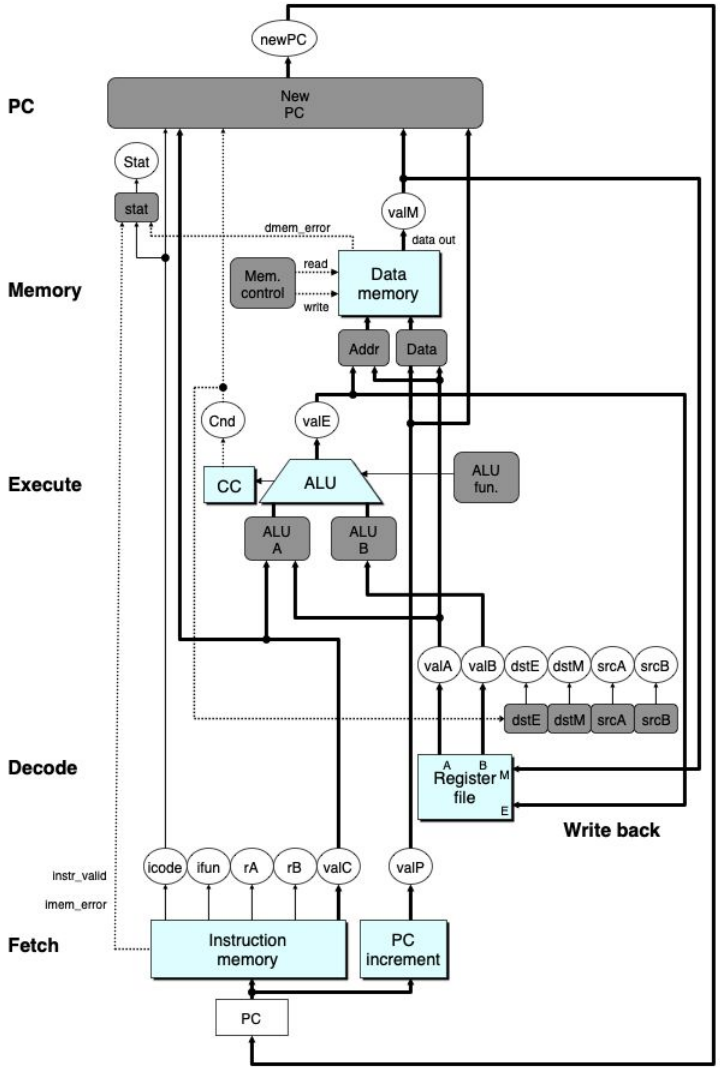
One clocked register (PC)



Y86 SEQ: Timing

One clocked register (PC)

Clock signal rises, whatever value was waiting on input lines to PC is now loaded and stored into PC

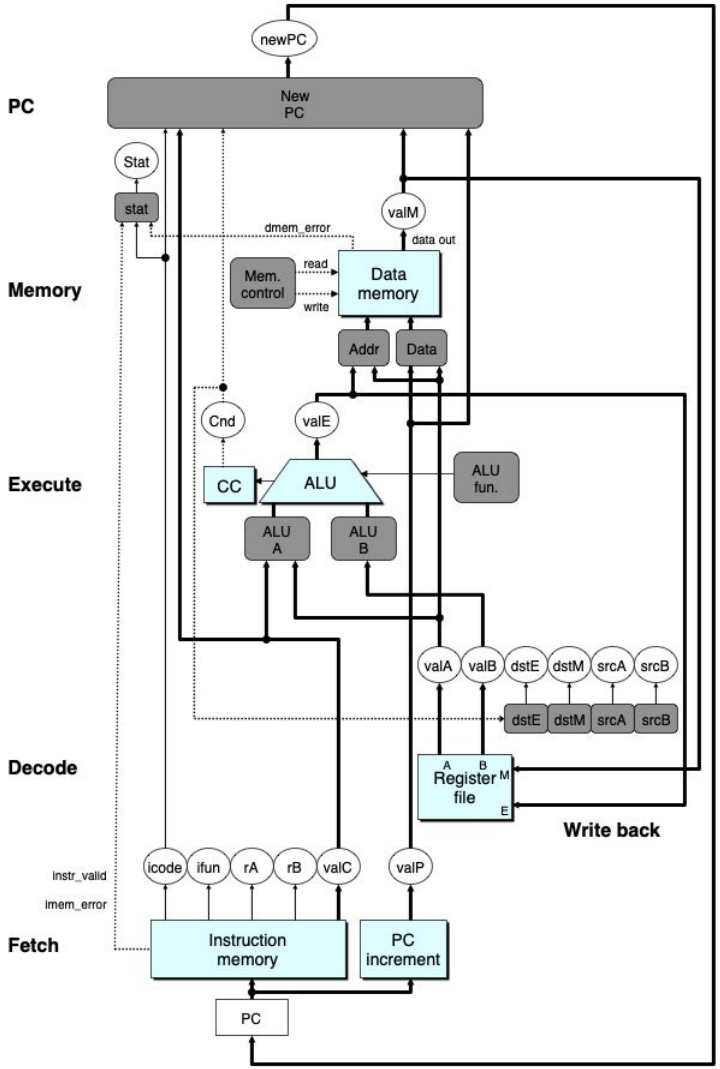


Y86 SEQ: Timing

One clocked register (PC)

Clock signal rises, whatever value was waiting on input lines to PC is now loaded and stored into PC

That address is now fed forward into Fetch stage, then Decode, etc.



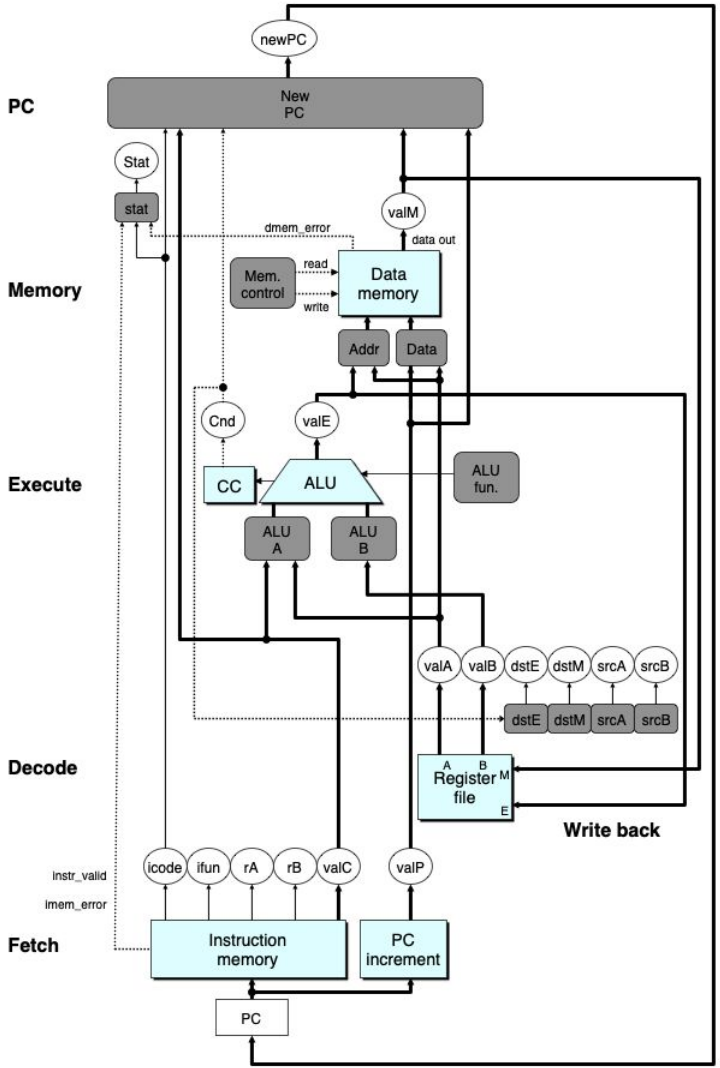
Y86 SEQ: Timing

One clocked register (PC)

Clock signal rises, whatever value was waiting on input lines to PC is now loaded and stored into PC

That address is now fed forward into Fetch stage, then Decode, etc.

Have to wait until all combinational logic in all stages completes before we can safely raise the clock signal again.



Y86 SEQ: Timing

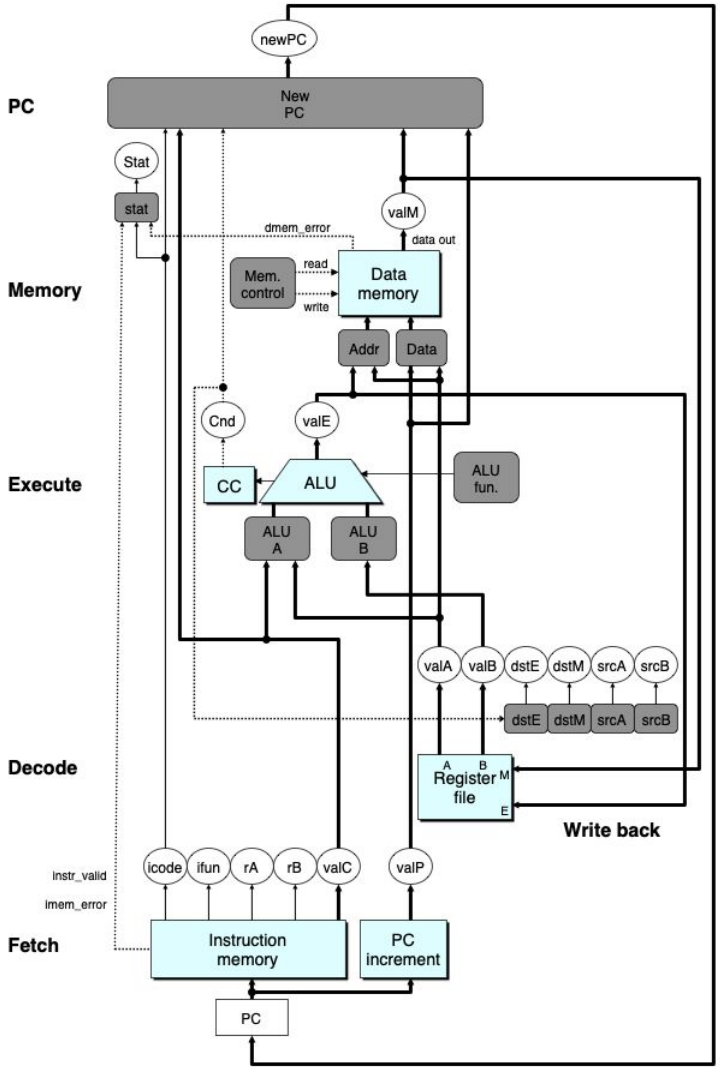
One clocked register (PC)

Clock signal rises, whatever value was waiting on input lines to PC is now loaded and stored into PC

That address is now fed forward into Fetch stage, then Decode, etc.

Have to wait until all combinational logic in all stages completes before we can safely raise the clock signal again.

Clock cycle must be long enough so that signals propagate through everything



Y86 SEQ: Timing

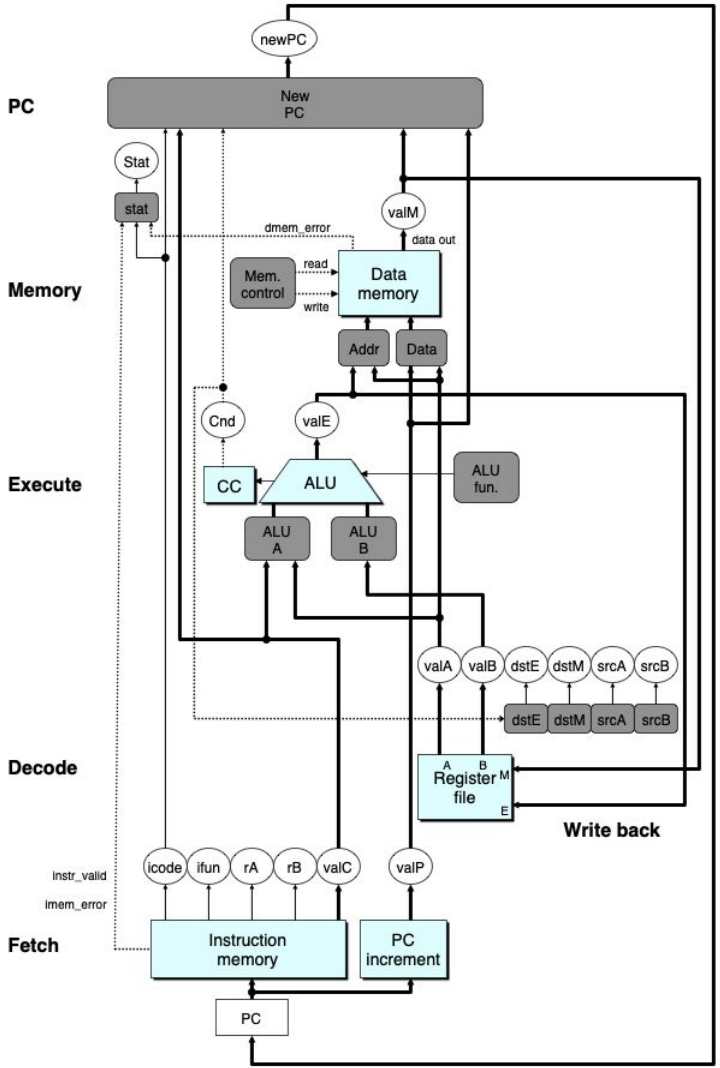
One clocked register (PC)

Clock signal rises, whatever value was waiting on input lines to PC is now loaded and stored into PC

That address is now fed forward into Fetch stage, then Decode, etc.

Have to wait until all combinational logic in all stages completes before we can safely raise the clock signal again.

Clock cycle must be long enough so that signals propagate through everything :(



Pipelining: The Laundry Analogy

Question: Who lives in a situation where you personally do the laundry?

Pipelining: The Laundry Analogy

Question: Who lives in a situation where you personally do the laundry?

Let’s assume we have 3 steps to this process: Wash, Dry, Fold.

Pipelining: The Laundry Analogy

Question: Who lives in a situation where you personally do the laundry?

Let’s assume we have 3 steps to this process: Wash, Dry, Fold. (😂 But let’s pretend)

Pipelining: The Laundry Analogy

Question: Who lives in a situation where you personally do the laundry?

Let’s assume we have 3 steps to this process: Wash, Dry, Fold.

Wash: 30 minutes

Dry: 40 minutes

Fold: 15 minutes

Pipelining: The Laundry Analogy

Question: Who lives in a situation where you personally do the laundry?

Let’s assume we have 3 steps to this process: Wash, Dry, Fold.

Wash: 30 minutes

Dry: 40 minutes

Fold: 15 minutes

How much time does it take to complete 1 load of laundry?

Pipelining: The Laundry Analogy

Question: Who lives in a situation where you personally do the laundry?

Let’s assume we have 3 steps to this process: Wash, Dry, Fold.

Wash: 30 minutes

Dry: 40 minutes

Fold: 15 minutes

How much time does it take to complete 1 load of laundry? 30 + 40 + 15 = 85 minutes

Pipelining: The Laundry Analogy

Question: Who lives in a situation where you personally do the laundry?

Let’s assume we have 3 steps to this process: Wash, Dry, Fold.

Wash: 30 minutes

Dry: 40 minutes

Fold: 15 minutes

How much time does it take to complete 1 load of laundry? 30 + 40 + 15 = 85 minutes

How much time does it take to complete 2 loads of laundry?

Pipelining: The Laundry Analogy

Question: Who lives in a situation where you personally do the laundry?

Let’s assume we have 3 steps to this process: Wash, Dry, Fold.

Wash: 30 minutes

Dry: 40 minutes

Fold: 15 minutes

How much time does it take to complete 1 load of laundry? 30 + 40 + 15 = 85 minutes

How much time does it take to complete 2 loads of laundry?

How much time does it take to complete 3 loads of laundry?

Pipelining: The Laundry Analogy

Question: Who lives in a situation where you personally do the laundry?

Let’s assume we have 3 steps to this process: Wash, Dry, Fold.

Wash: 30 minutes

Dry: 40 minutes

Fold: 15 minutes

How much time does it take to complete 1 load of laundry? 30 + 40 + 15 = 85 minutes

How much time does it take to complete 2 loads of laundry?

How much time does it take to complete 3 loads of laundry?

If I have infinite loads of laundry, how often do I complete a single load?

Pipelining: Terminology

Latency: Time required to complete a single task (i.e. one single load of laundry, or a single instruction)

Pipelining: Terminology

Latency: Time required to complete a single task (i.e. one single load of laundry, or a single instruction)

Throughput: Tasks performed per time unit (usually per second)

Pipelining: Terminology

Latency: Time required to complete a single task (i.e. one single load of laundry, or a single instruction)

Throughput: Tasks performed per time unit (usually per second)

Low latency = good. (The shorter the time to complete a single task, the better.)

Pipelining: Terminology

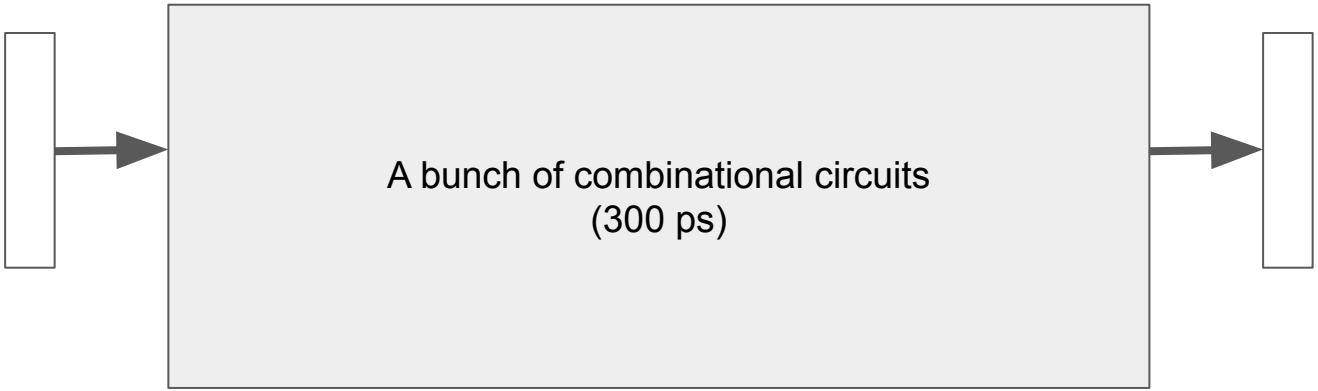
Latency: Time required to complete a single task (i.e. one single load of laundry, or a single instruction)

Throughput: Tasks performed per time unit (usually per second)

Low latency = good. (The shorter the time to complete a single task, the better.)

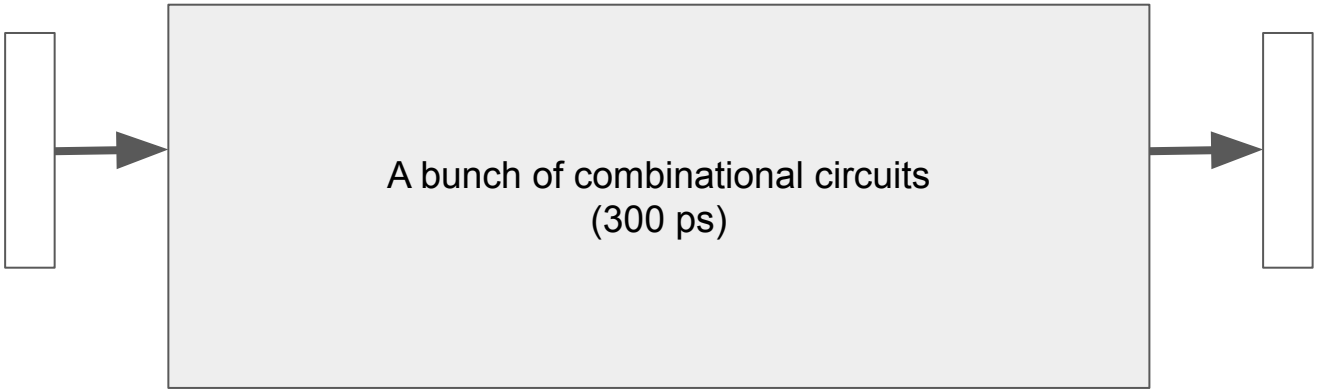
High throughput = better. (The more tasks completed per second, the better. This matters more than low latency)

Pipelining: Combinational Logic



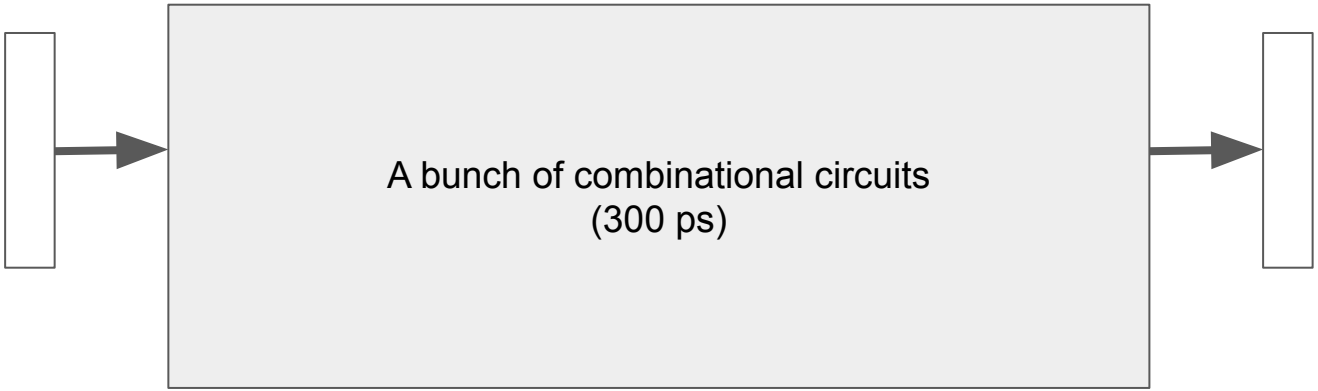
- Suppose that our combinational circuit takes 300 picoseconds

Pipelining: Combinational Logic



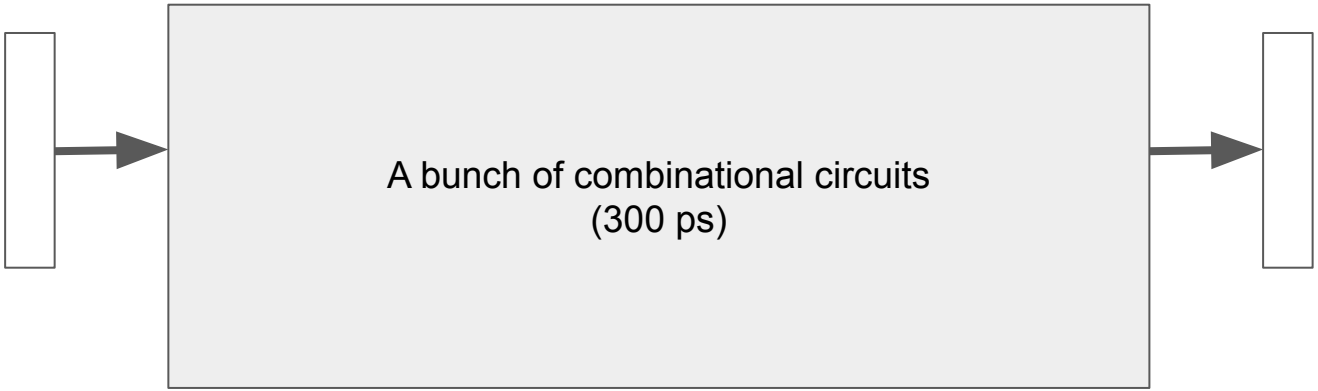
- Suppose that our combinational circuit takes 300 picoseconds
- Additional 20 picoseconds to save result in register

Pipelining: Combinational Logic



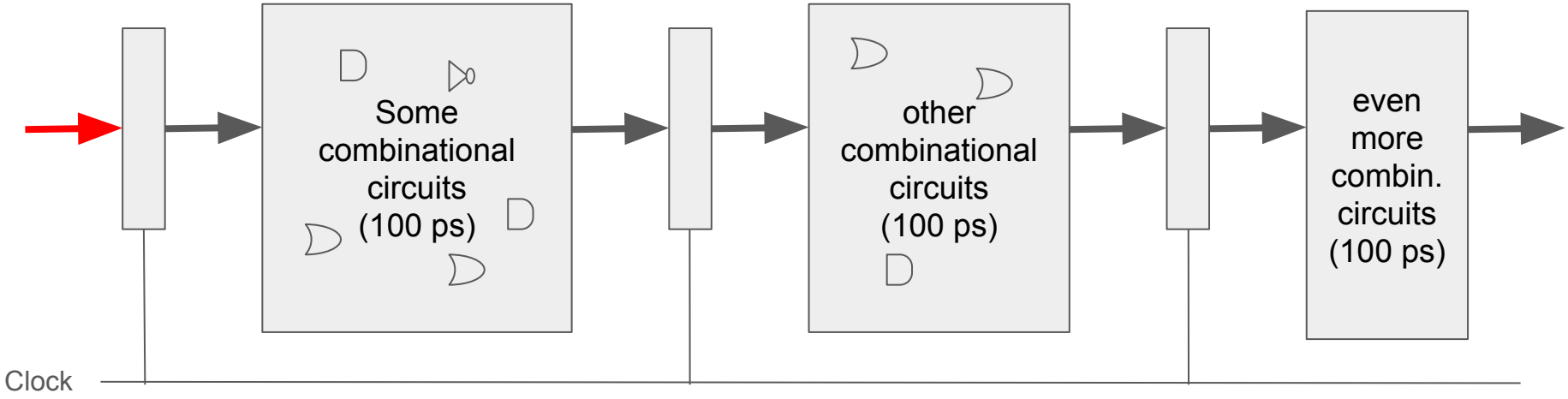
- Suppose that our combinational circuit takes 300 picoseconds
- Additional 20 picoseconds to save result in register
- Clock cycle of 320 picoseconds

Pipelining: Combinational Logic



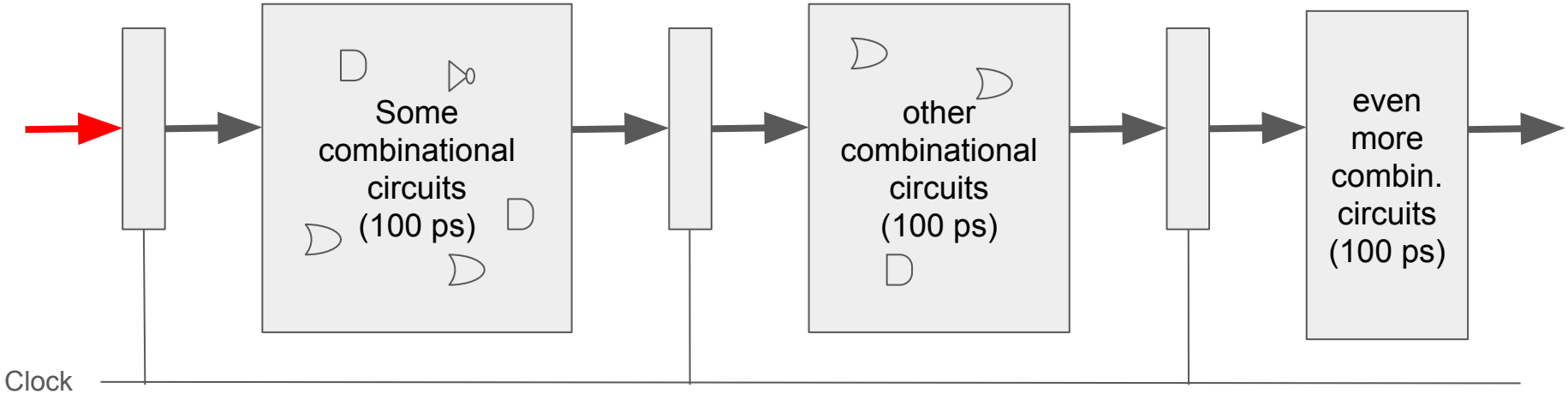
- Suppose that our combinational circuit takes 300 picoseconds
- Additional 20 picoseconds to save result in register
- Clock cycle of 320 picoseconds
- Latency of 320 picoseconds
- Throughput of 1 instruction / 320 picoseconds ~ 3.125 instr/nanosec

Pipelining: What if we add clocked registers?



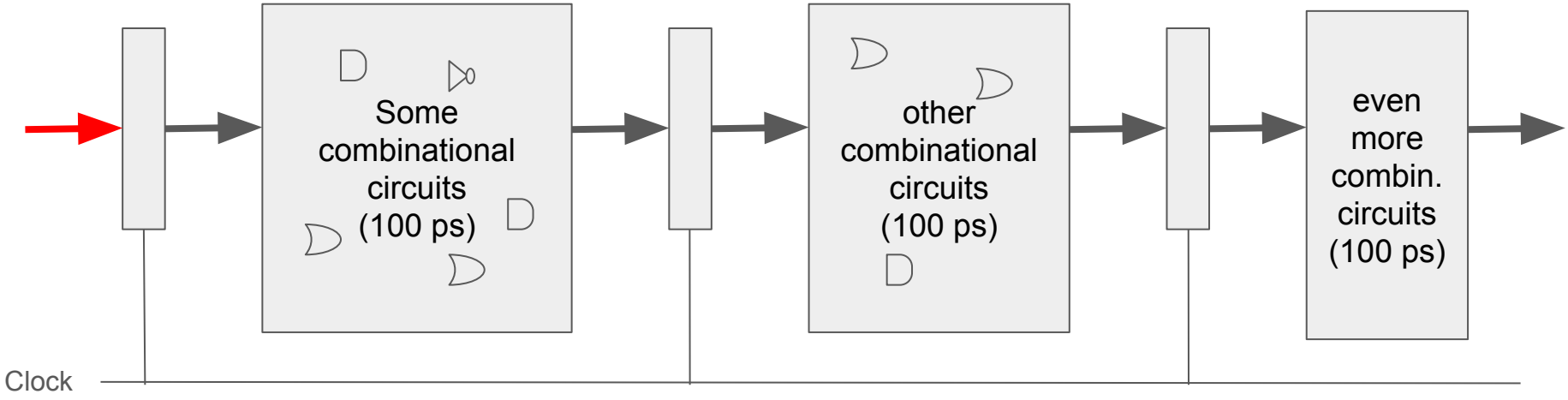
- Divide logic into 3 smaller chunks with 100 picosecond time for each

Pipelining: What if we add clocked registers?



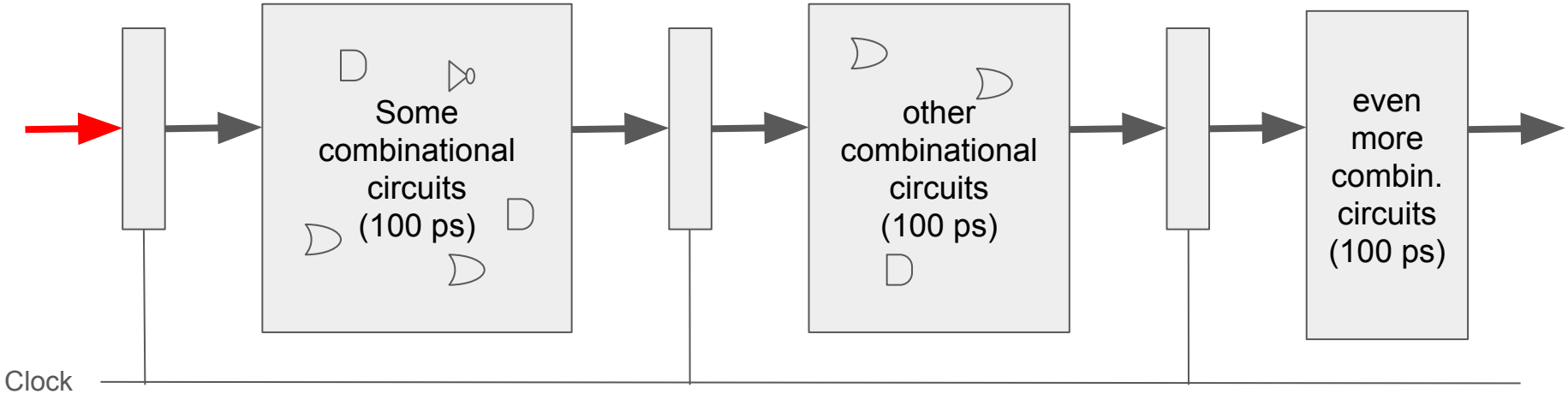
- Divide logic into 3 smaller chunks with 100 picosecond time for each
- Clocked registers still 20 picoseconds

Pipelining: What if we add clocked registers?



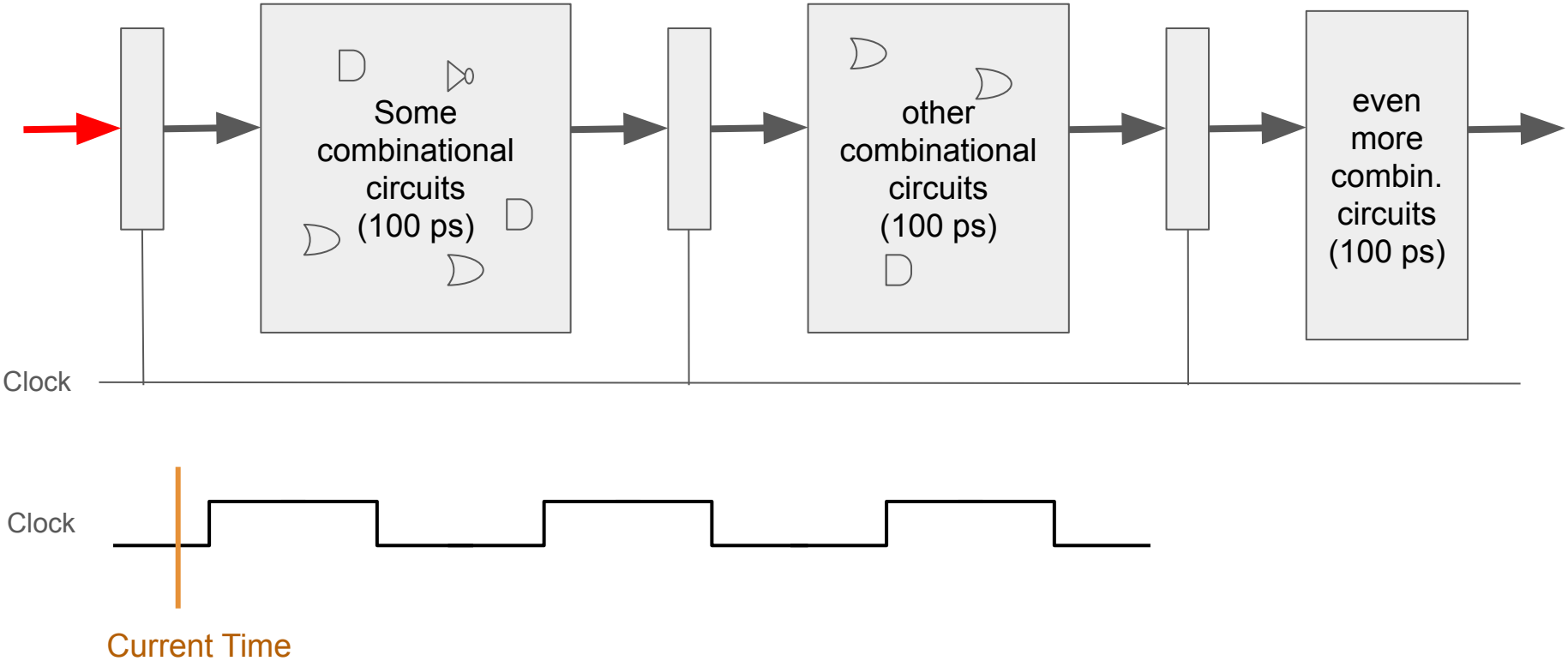
- Divide logic into 3 smaller chunks with 100 picosecond time for each
- Clocked registers still 20 picoseconds
- Can begin a new operation as soon as previous one completes first stage

Pipelining: What if we add clocked registers?

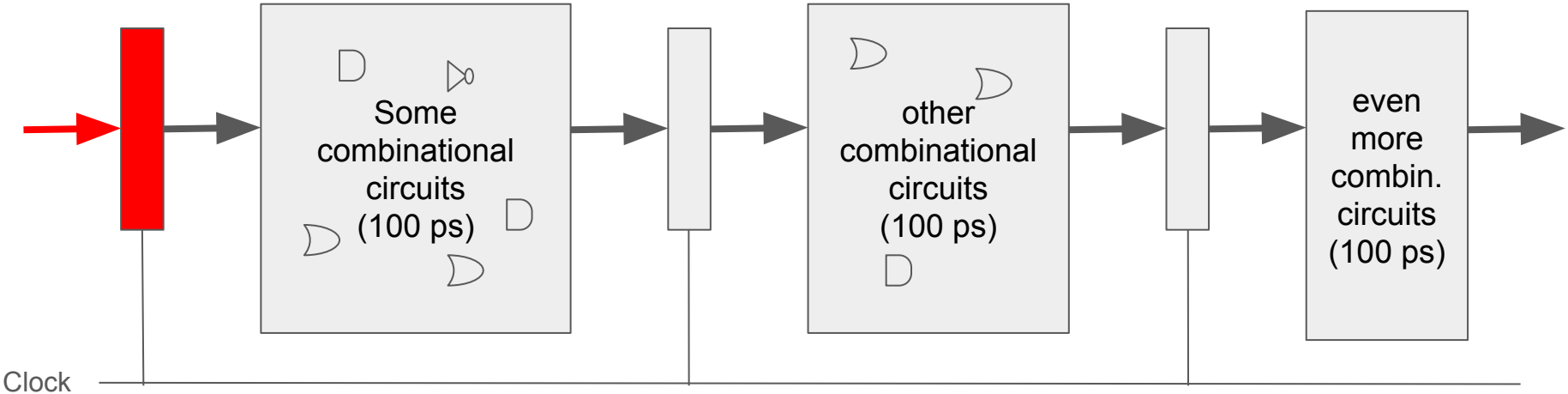


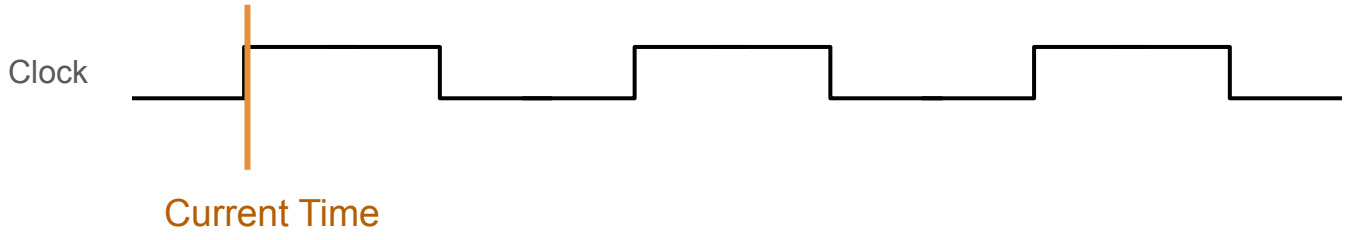
- Divide logic into 3 smaller chunks with 100 picosecond time for each
- Clocked registers still 20 picoseconds
- Can begin a new operation as soon as previous one completes first stage
- Latency will be higher - 360 picoseconds. What about throughput?

Pipelining: What if we add clocked registers?

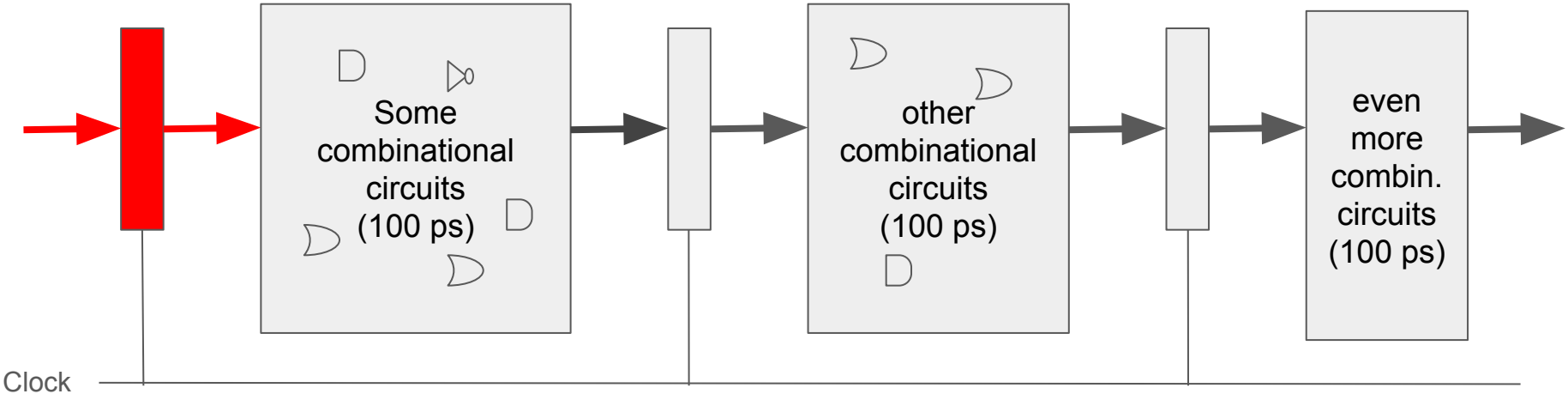


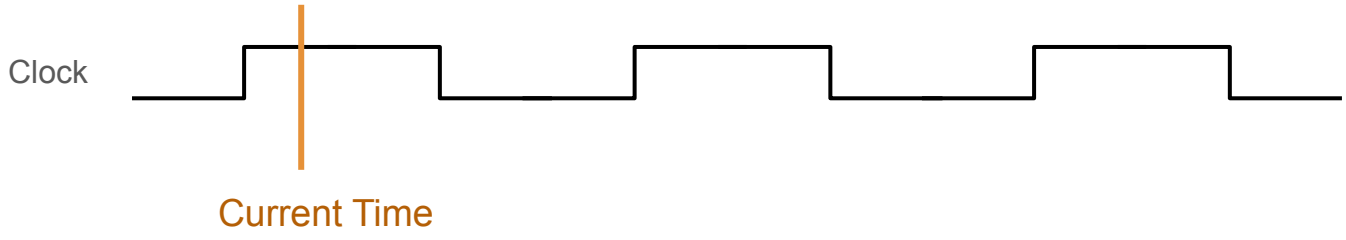
Pipelining: What if we add clocked registers?



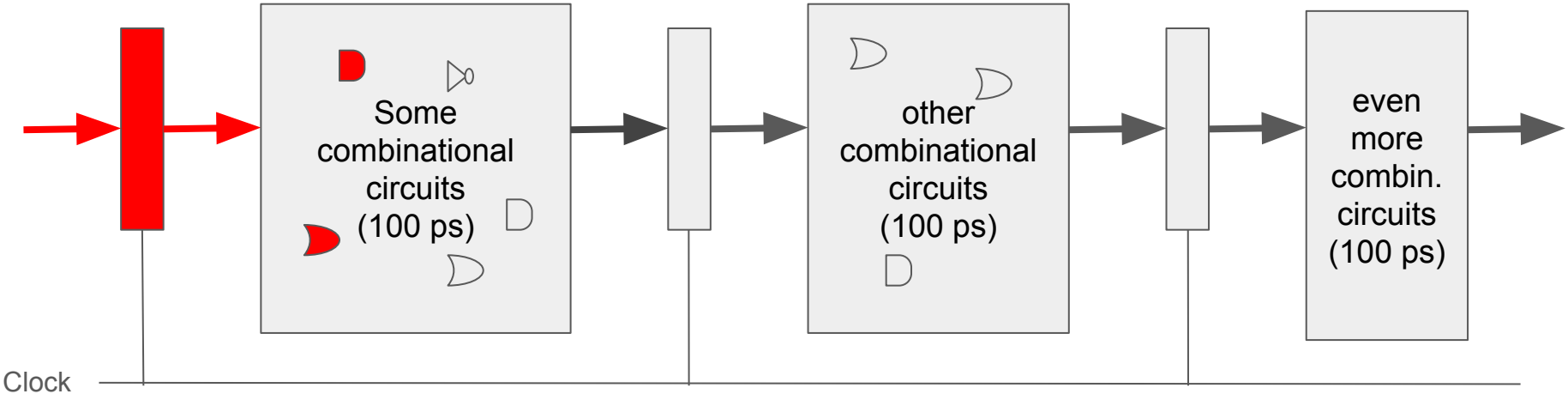


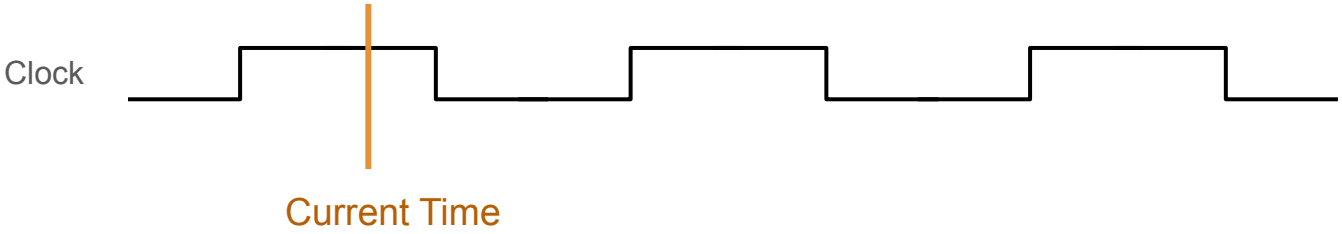
Pipelining: What if we add clocked registers?



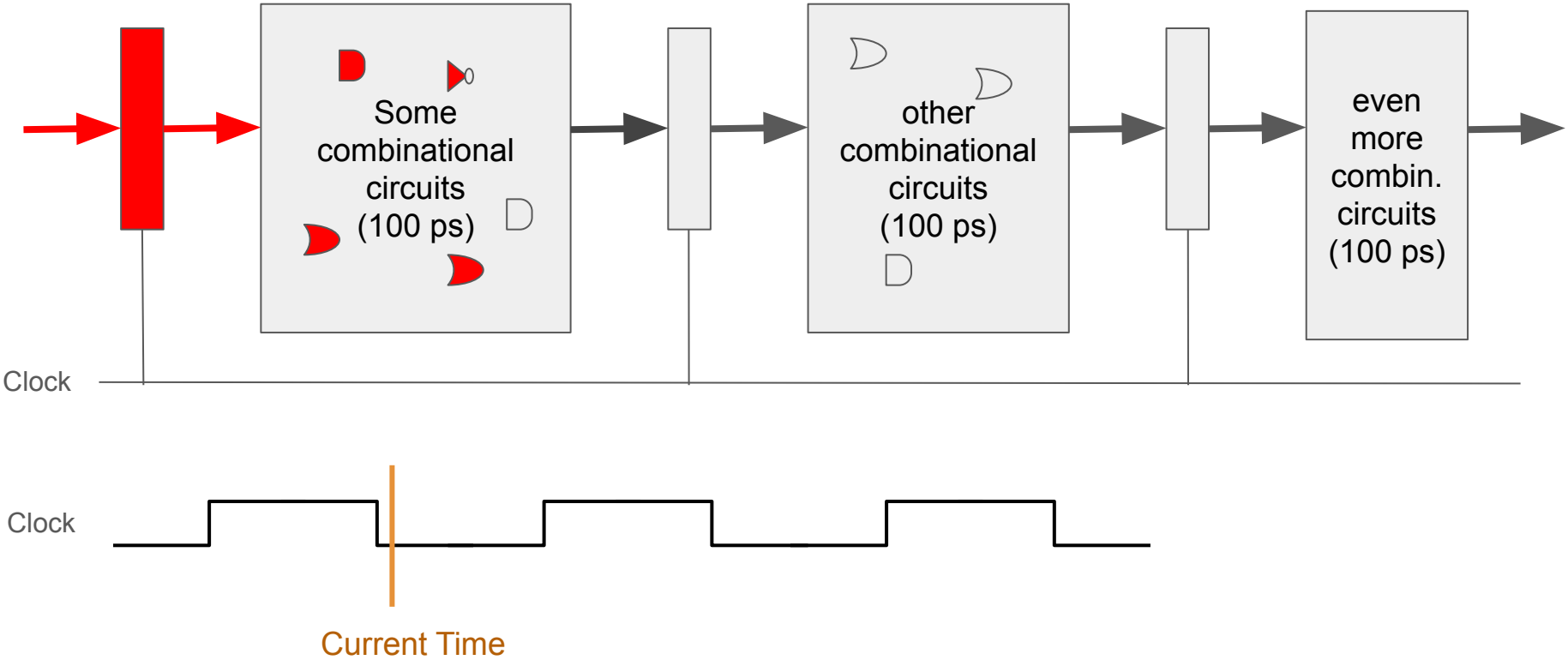


Pipelining: What if we add clocked registers?

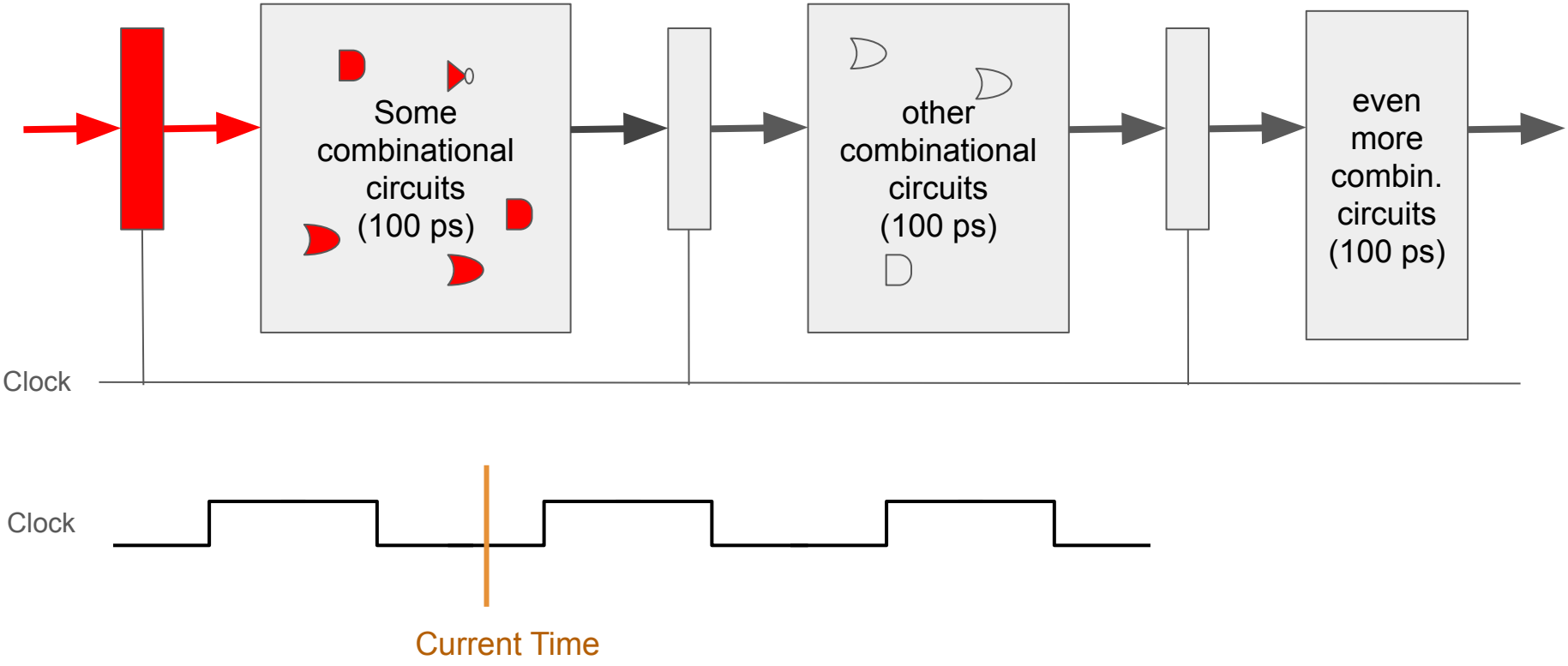




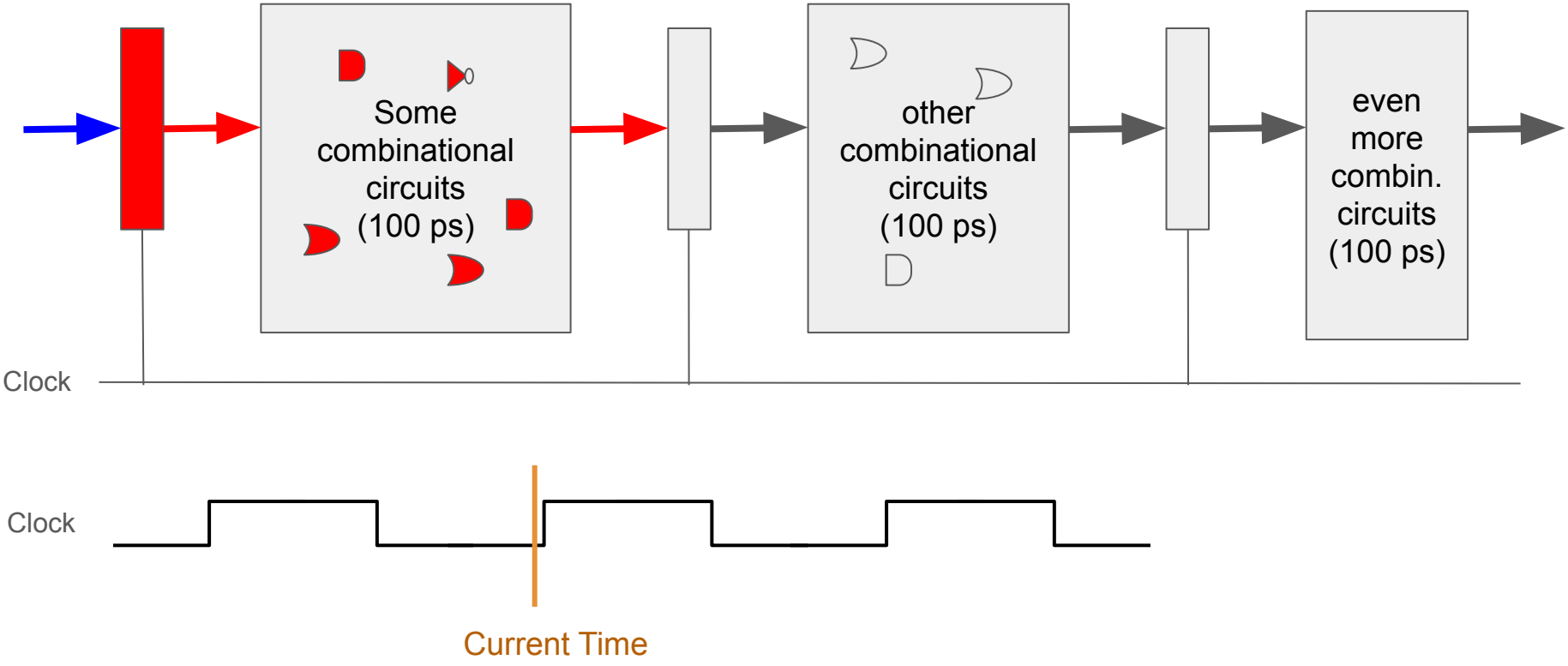
Pipelining: What if we add clocked registers?



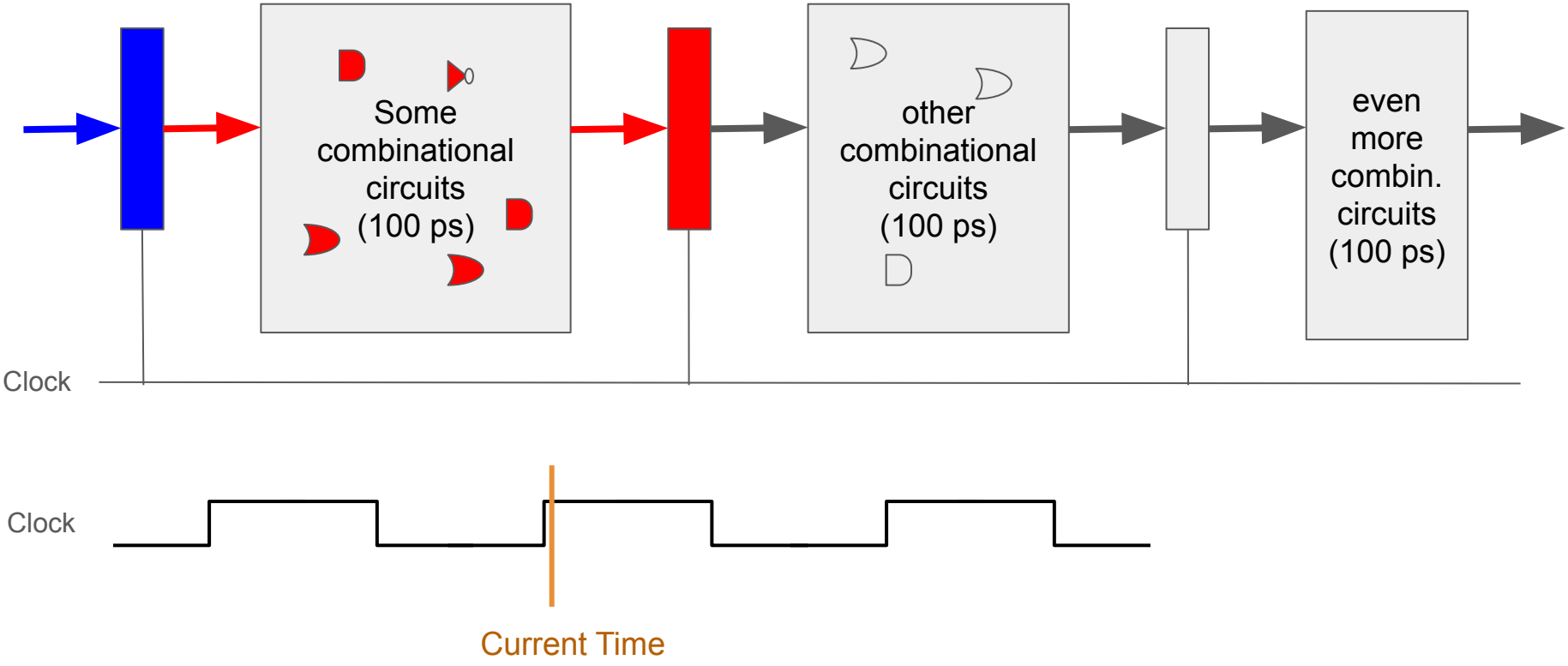
Pipelining: What if we add clocked registers?



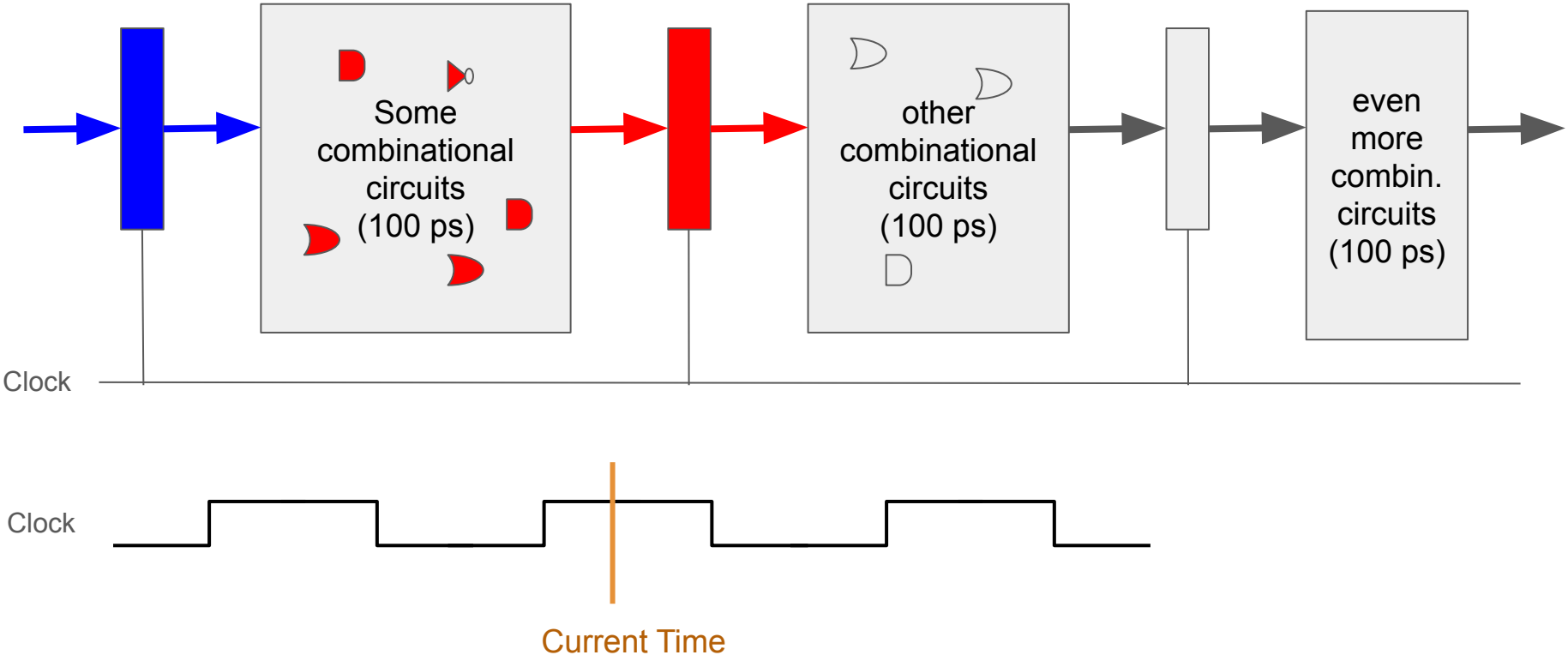
Pipelining: What if we add clocked registers?



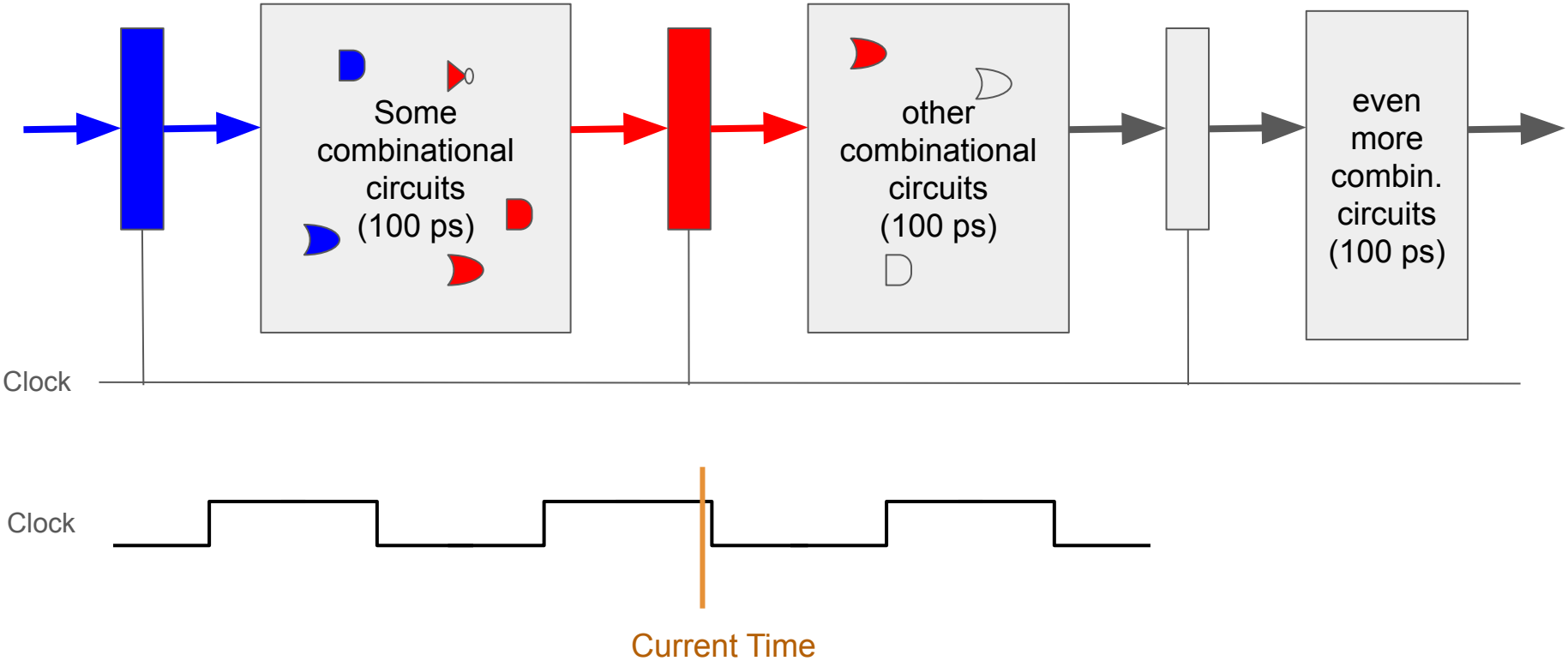
Pipelining: What if we add clocked registers?



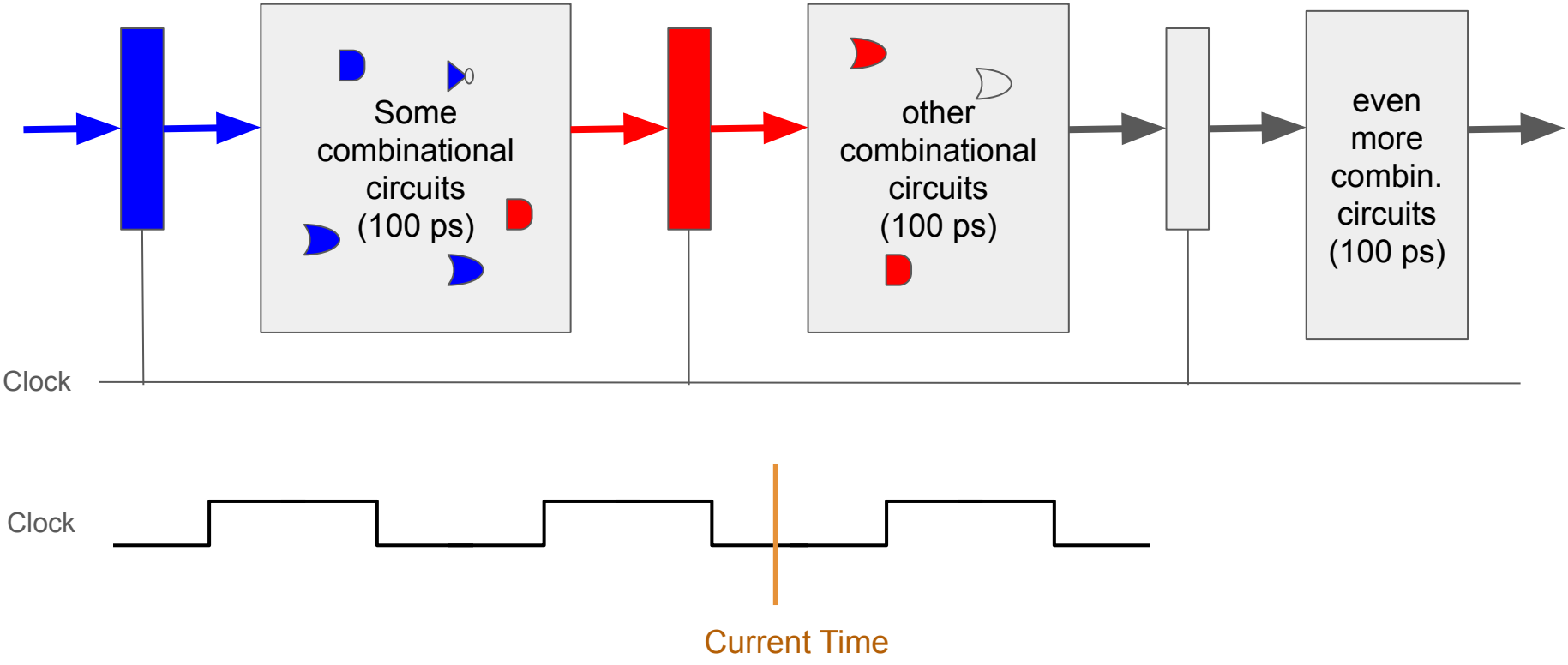
Pipelining: What if we add clocked registers?



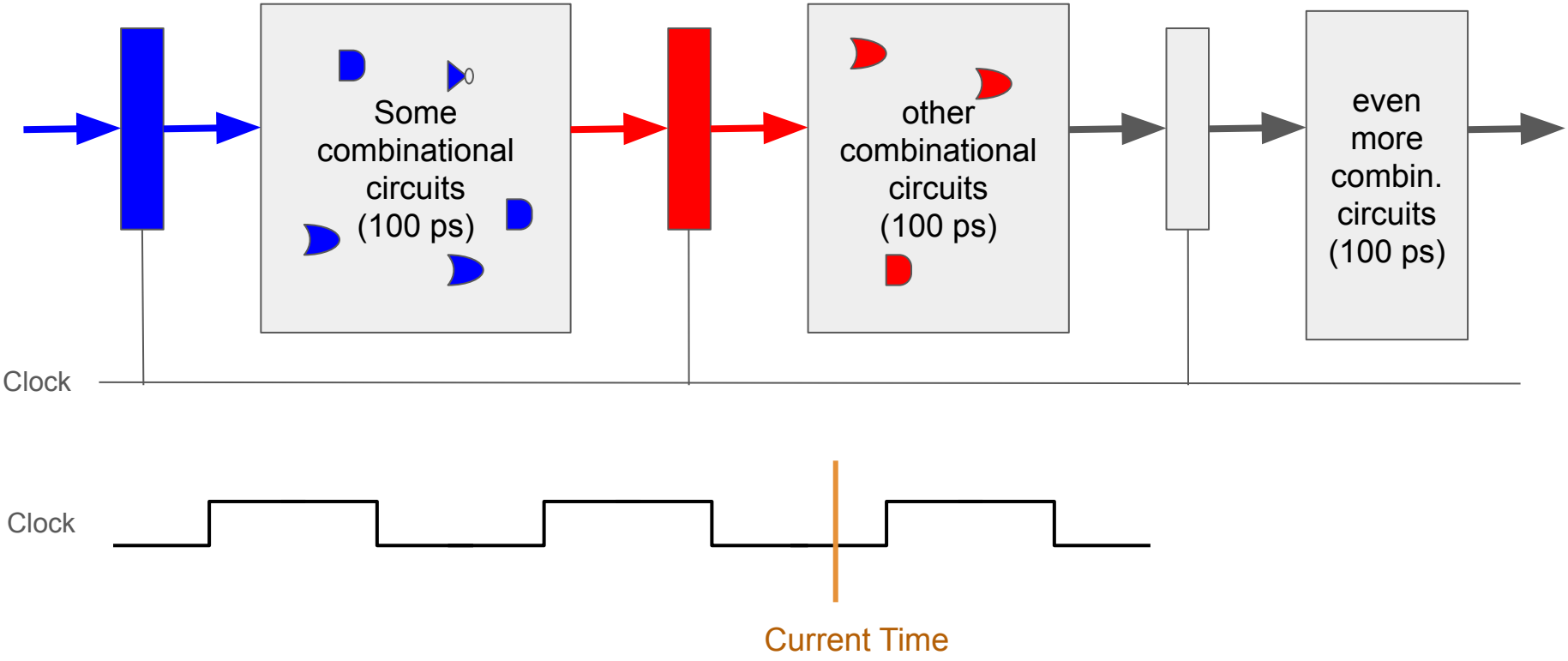
Pipelining: What if we add clocked registers?



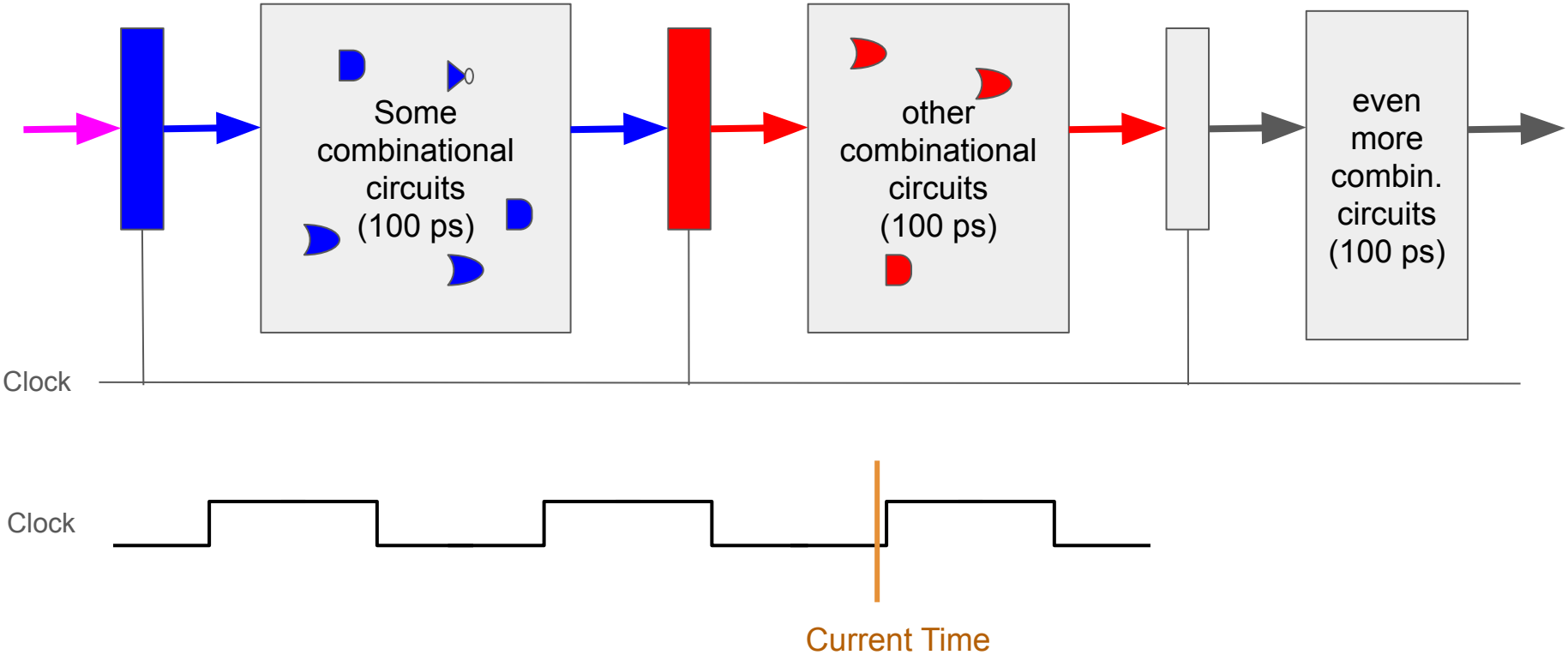
Pipelining: What if we add clocked registers?



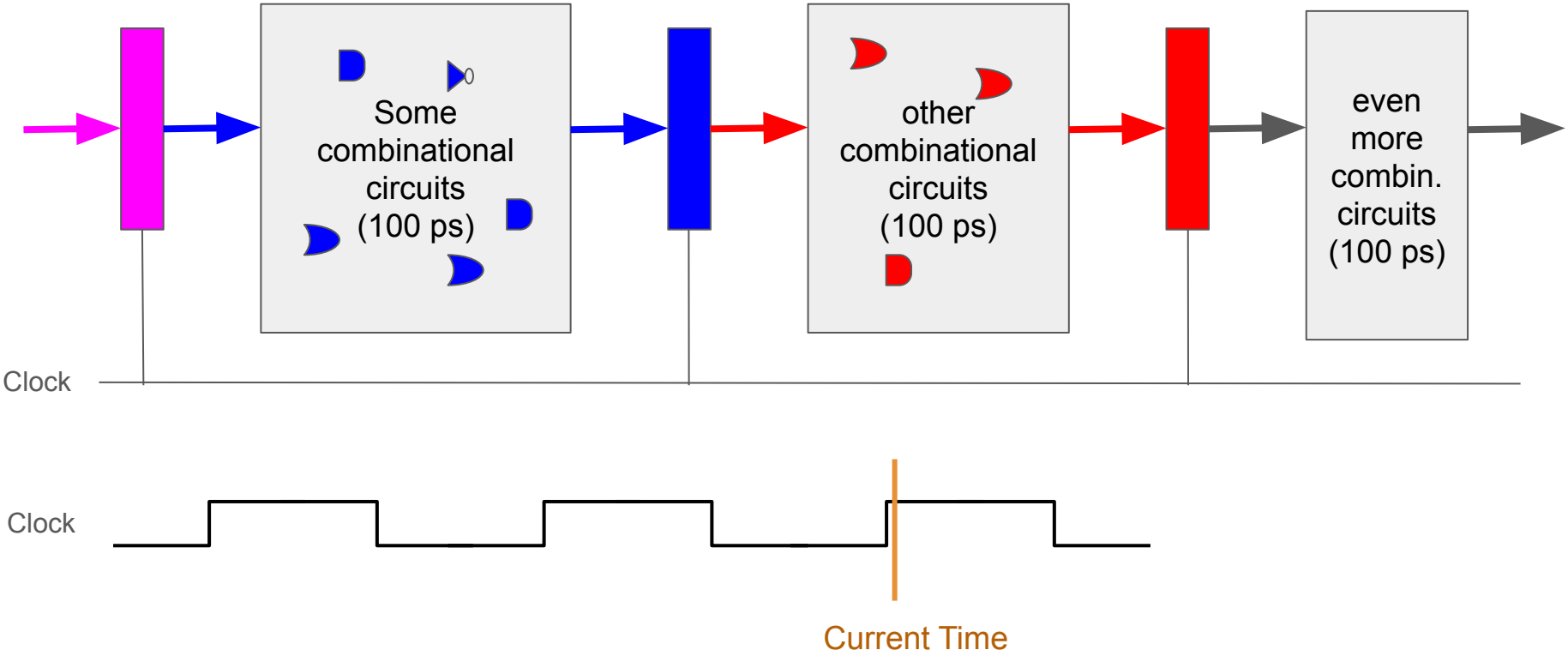
Pipelining: What if we add clocked registers?



Pipelining: What if we add clocked registers?



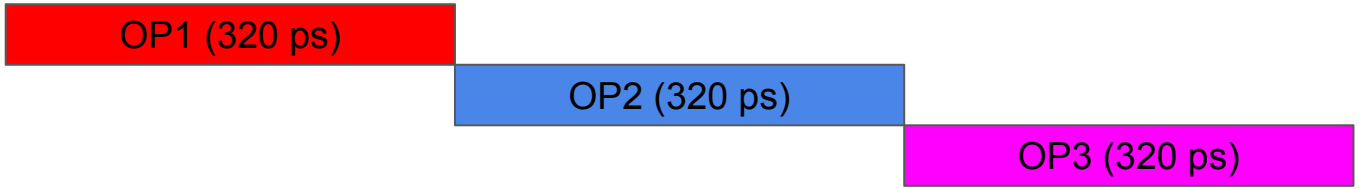
Pipelining: What if we add clocked registers?



Pipeline Timing: Executing 3 instructions

To execute three instructions OP1, OP2 and OP3 on our two different systems:

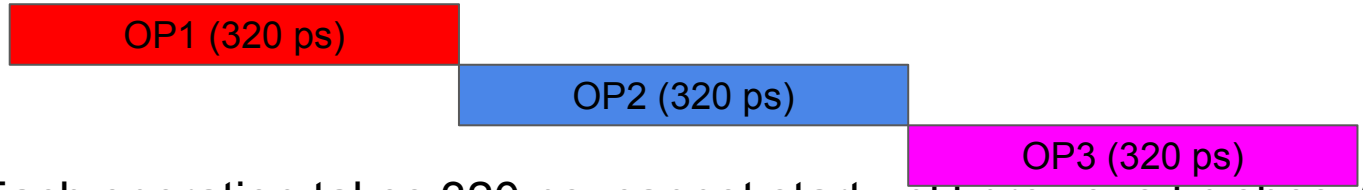
Unpipelined:



Pipeline Timing: Executing 3 instructions

To execute three instructions OP1, OP2 and OP3 on our two different systems:

Unpipelined:

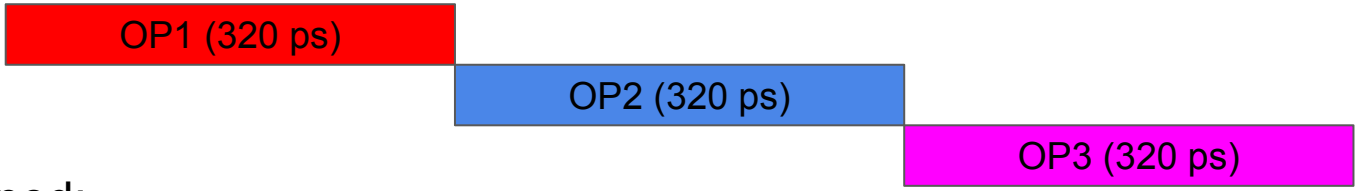


- Each operation takes 320 ps, cannot start until previous finishes. 960 ps total

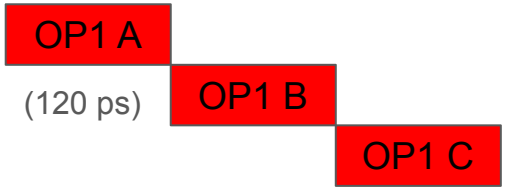
Pipeline Timing: Executing 3 instructions

To execute three instructions OP1, OP2 and OP3 on our two different systems:

Unpipelined:



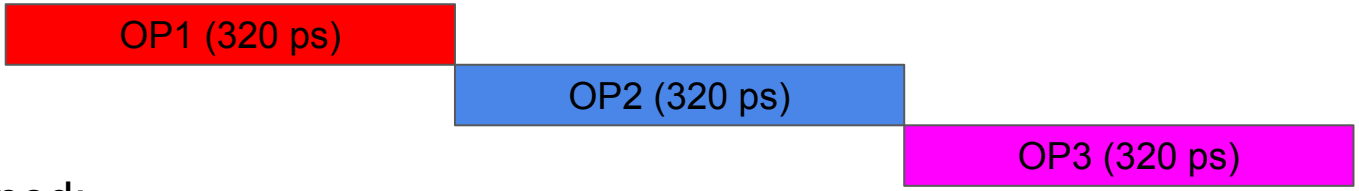
Pipelined:



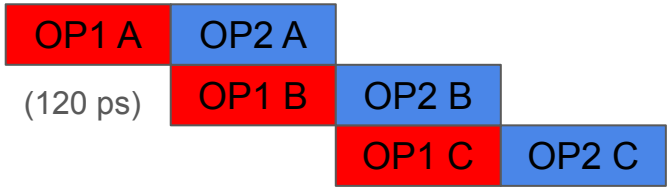
Pipeline Timing: Executing 3 instructions

To execute three instructions OP1, OP2 and OP3 on our two different systems:

Unpipelined:



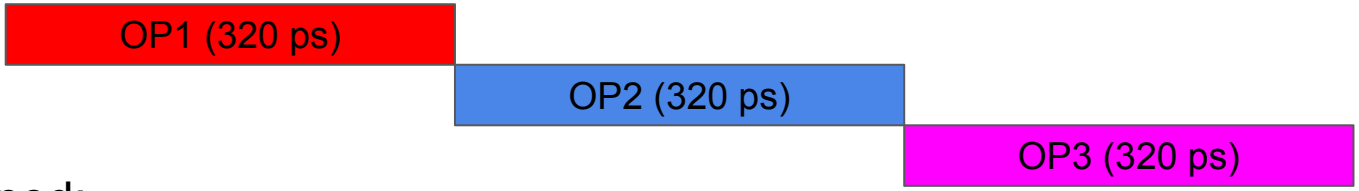
Pipelined:



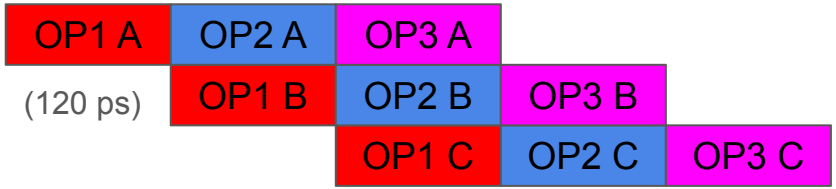
Pipeline Timing: Executing 3 instructions

To execute three instructions OP1, OP2 and OP3 on our two different systems:

Unpipelined:



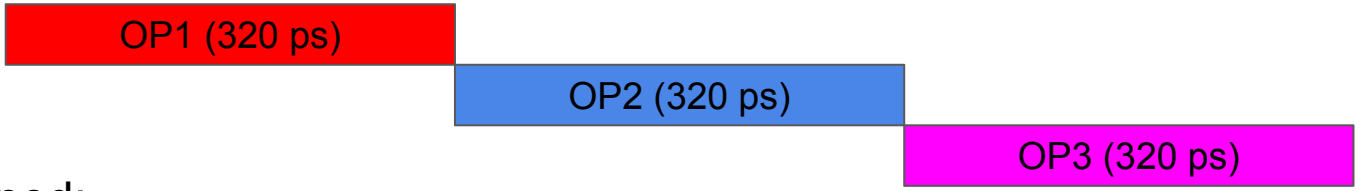
Pipelined:



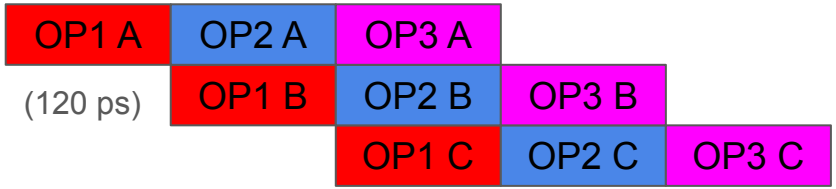
Pipeline Timing: Executing 3 instructions

To execute three instructions OP1, OP2 and OP3 on our two different systems:

Unpipelined:



Pipelined:

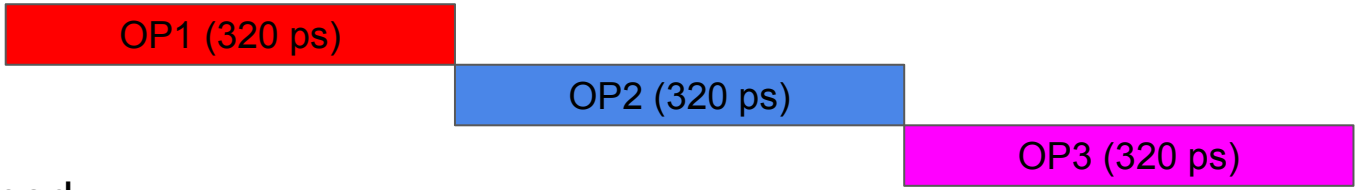


- Begin new operation every 120 ps

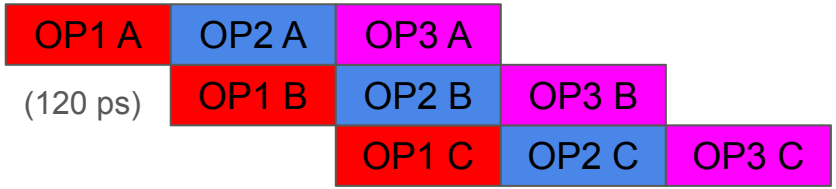
Pipeline Timing: Executing 3 instructions

To execute three instructions OP1, OP2 and OP3 on our two different systems:

Unpipelined:

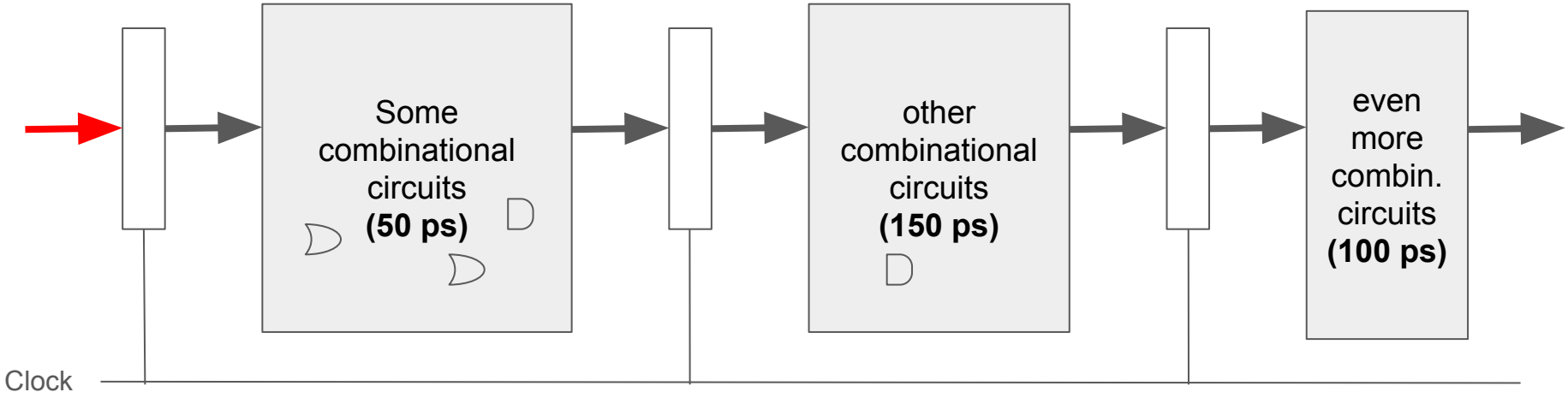


Pipelined:



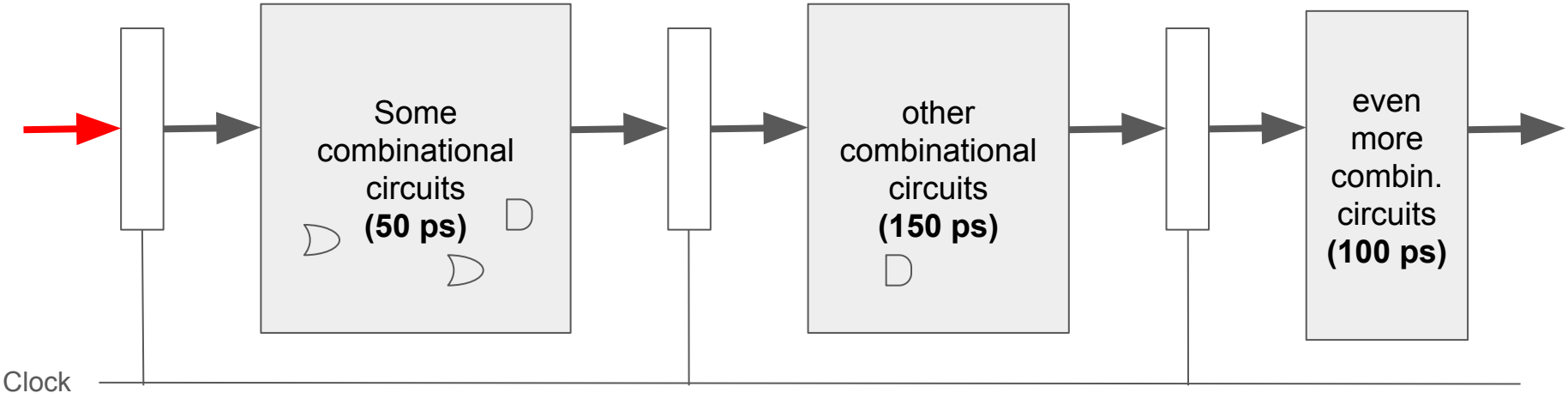
- Begin new operation every 120 ps. Throughput 1 instr / 120 ps ~ 8.3 inst/ns

Pipeline Challenges: Nonuniform delays



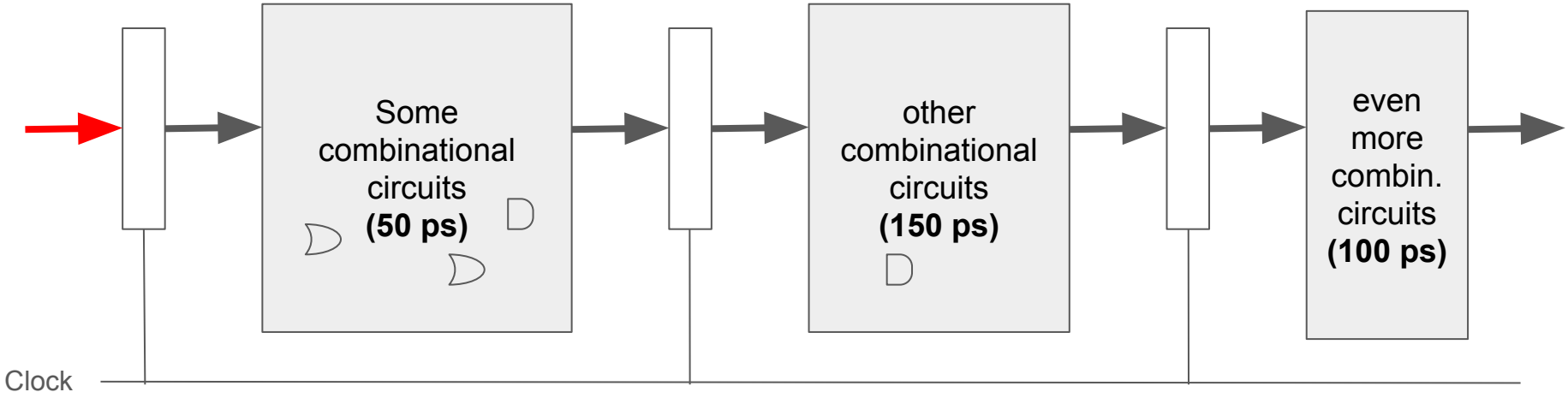
- It is not easy to divide combinational logic into 3 exactly equal length stages

Pipeline Challenges: Nonuniform delays



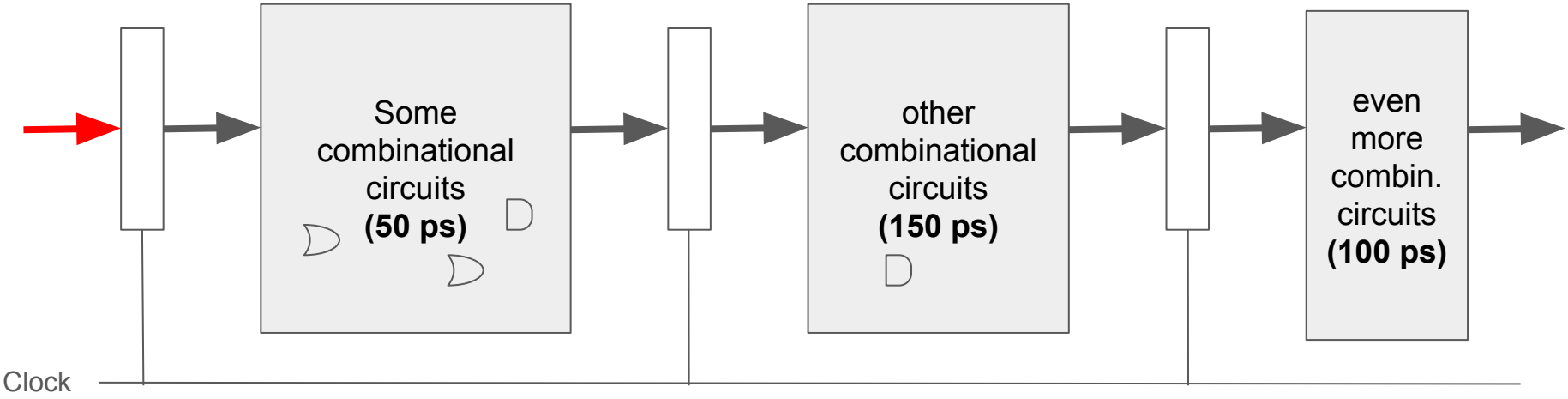
- It is not easy to divide combinational logic into 3 exactly equal length stages
- Throughput limited by slowest stage

Pipeline Challenges: Nonuniform delays



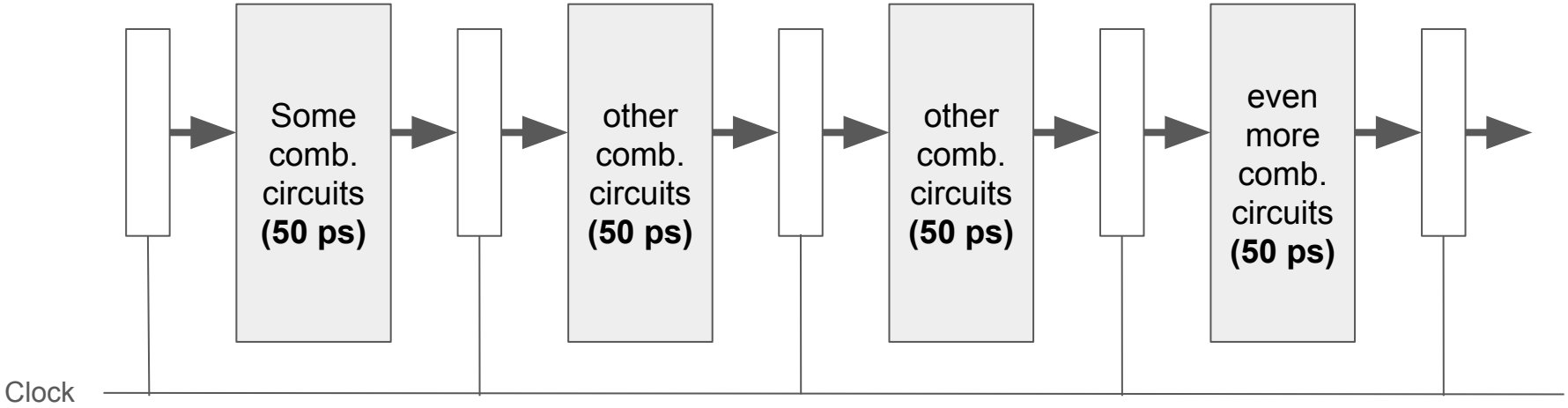
- It is not easy to divide combinational logic into 3 exactly equal length stages
- Throughput limited by slowest stage
- Other stages will sit idle while waiting for slowest to complete

Pipeline Challenges: Nonuniform delays



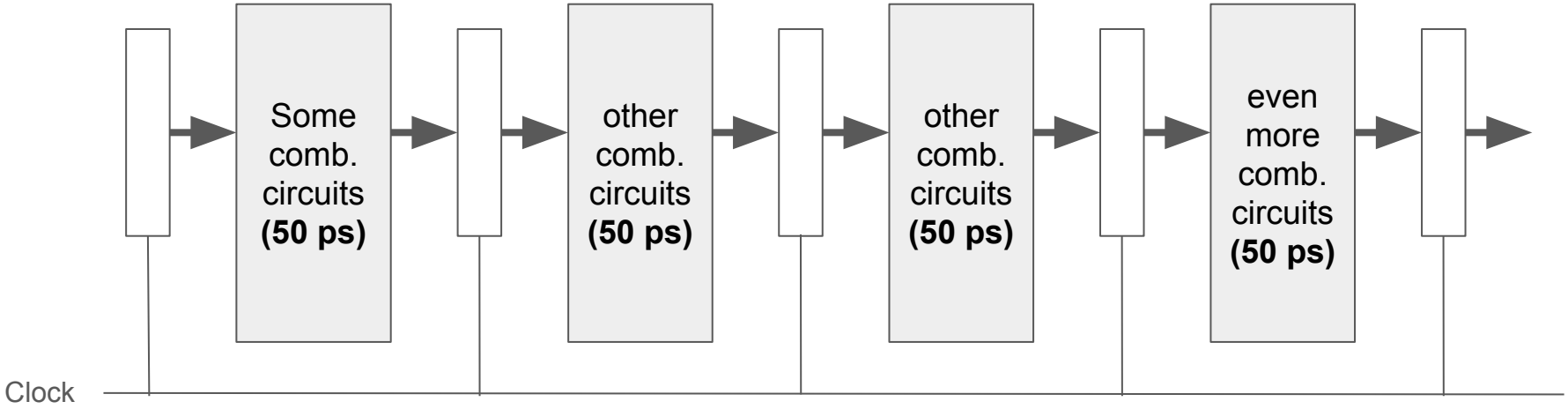
- It is not easy to divide combinational logic into 3 exactly equal length stages
- Throughput limited by slowest stage
- Other stages will sit idle while waiting for slowest to complete
- Maybe just add more registers?

Pipeline Challenges: Register Overhead



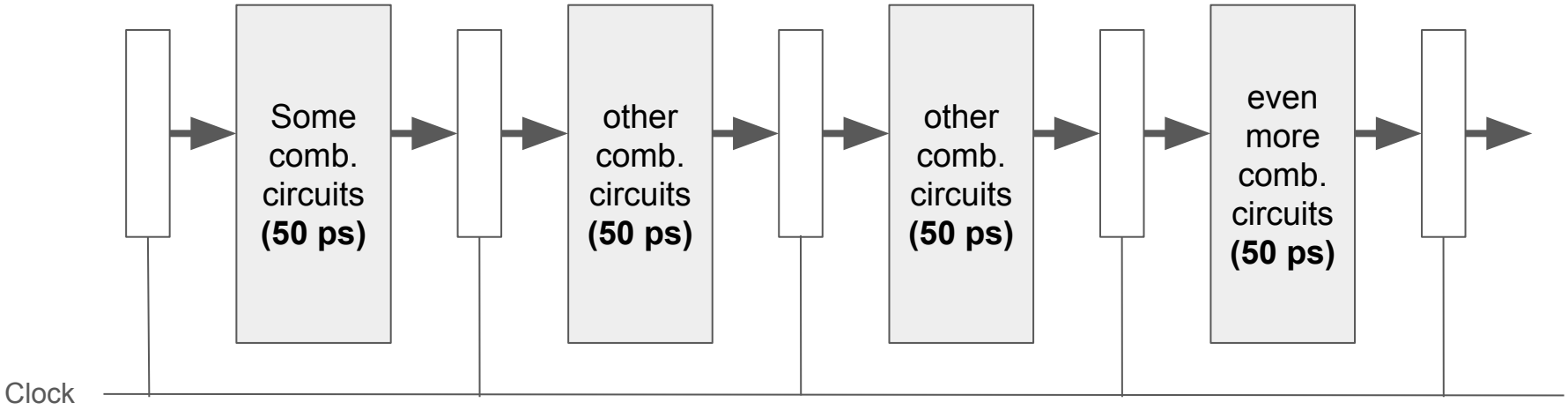
- As number of stages increases, so does number of clocked registers, each of which adds to total computation time

Pipeline Challenges: Register Overhead



- As number of stages increases, so does number of clocked registers, each of which adds to total computation time
- Increasing number of stages gets diminishing returns

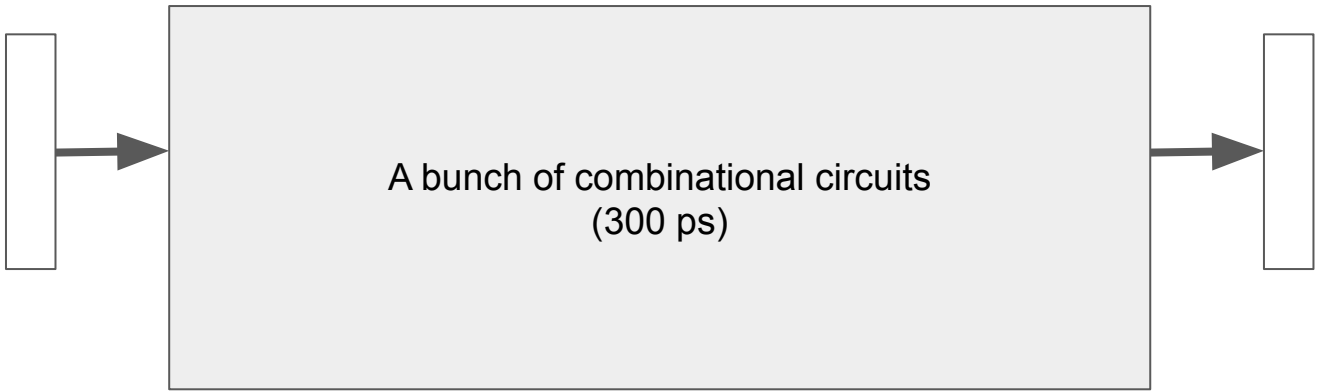
Pipeline Challenges: Register Overhead



- As number of stages increases, so does number of clocked registers, each of which adds to total computation time
- Increasing number of stages gets diminishing returns
- Question: What is the best possible throughput?

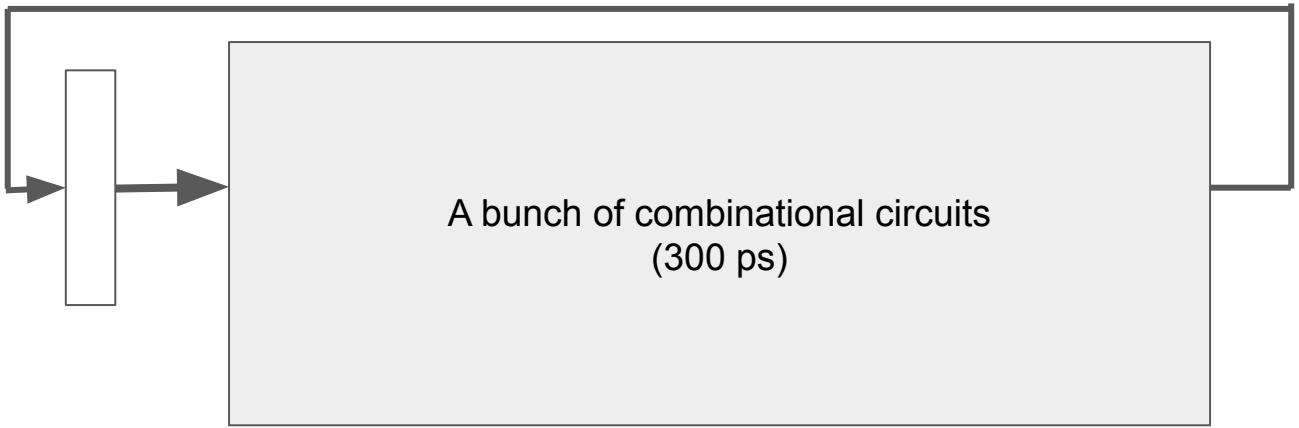
Pipeline Challenges: Feedback

One more thing to remember: Our original system had a loop



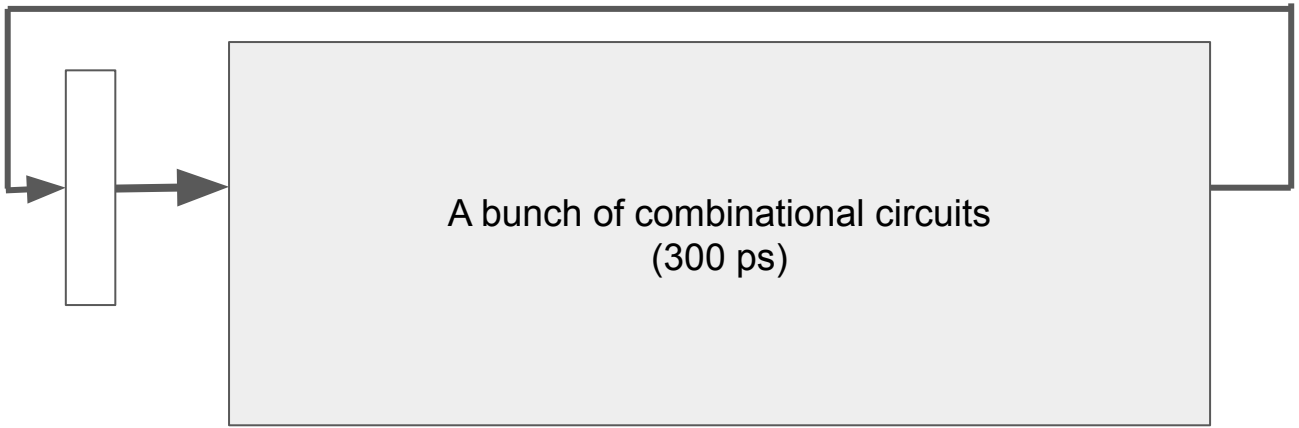
Pipeline Challenges: Feedback

One more thing to remember: Our original system had a loop



Pipeline Challenges: Feedback

One more thing to remember: Our original system had a loop



Sometimes, later instructions depended on results of earlier instructions

Pipeline Challenges: Feedback

Consider instruction sequence:

```
irmovq $22, %rcx
addq %rcx, %rdi
mrmovq 100(%rdi), %r9
```

Pipeline Challenges: Feedback

Consider instruction sequence:

```
irmovq $22, %rcx
addq %rcx, %rdi
mrmovq 100(%rdi), %r9
```

Pipeline Challenges: Feedback

Consider instruction sequence:

```
irmovq $22, %rcx
addq %rcx, %rdi
mrmovq 100(%rdi), %r9
```

Pipeline Challenges: Feedback

Consider instruction sequence:

```
irmovq $22, %rcx
addq %rcx, %rdi
mrmovq 100(%rdi), %r9
```

We will introduce pipelining to Y86, but we'll need to handle these situations…

Our registers are supposed to be updated, but what if they haven't been updated yet?

Pipeline Challenges: Feedback

Consider instruction sequence:

```
.LOOP:
 subq %rdx, %rcx
 jne .END
 addq %rax, %rdx
 jmp .LOOP
```

Pipeline Challenges: Feedback

Consider instruction sequence:

```
.LOOP:
 subq %rdx, %rcx
 jne .END
 addq %rax, %rdx
 jmp .LOOP
```

Conditional jump depends on result of previous operation

Pipeline Challenges: Feedback

Consider instruction sequence:

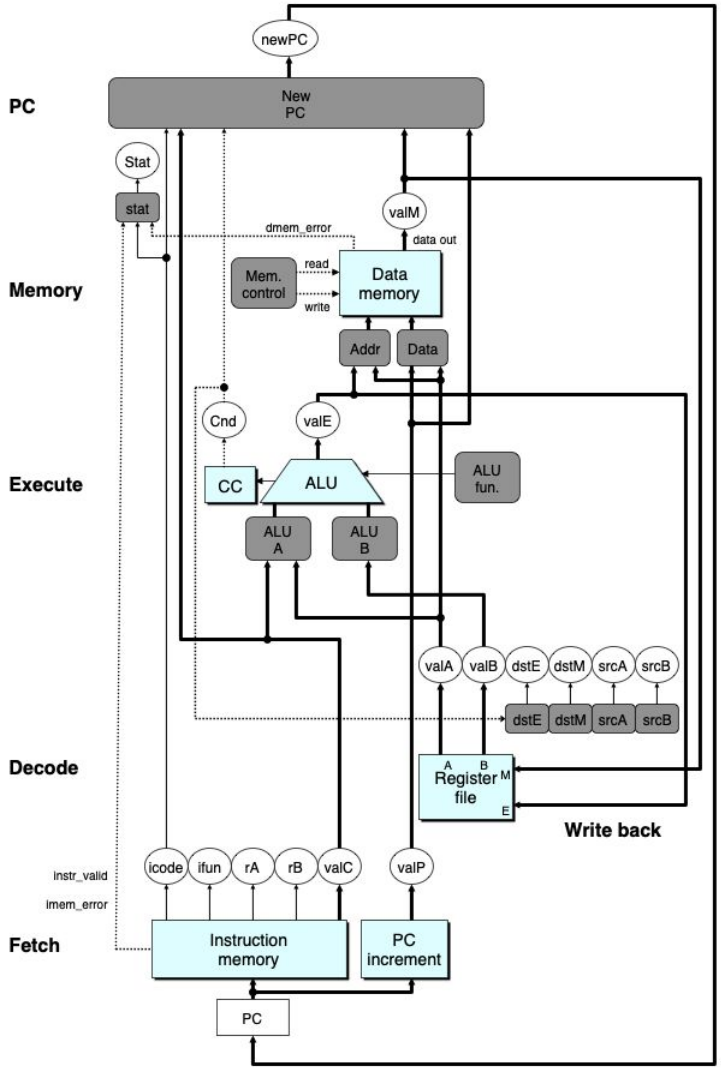
```
.LOOP:
 subq %rdx, %rcx
 jne .END
 addq %rax, %rdx
 jmp .LOOP
```

Conditional jump depends on result of previous operation

But I want to start executing my jne instruction! How far can I get?

Pipelining: Converting from SEQ

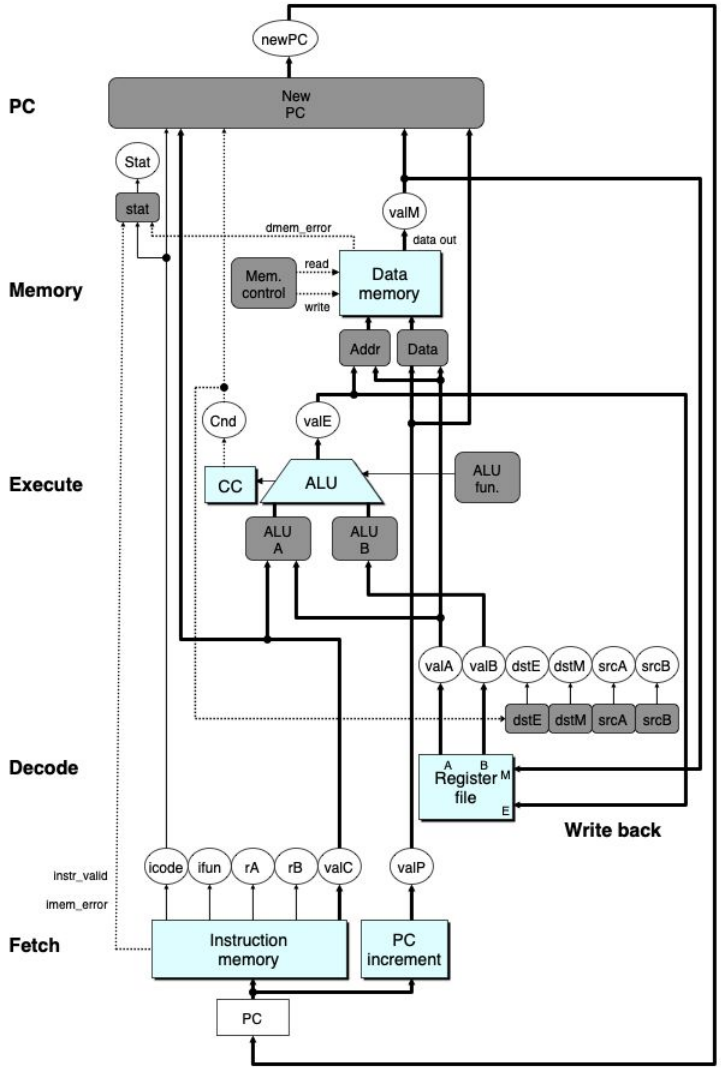
SEQ Architecture



Pipelining: Converting from SEQ

SEQ Architecture

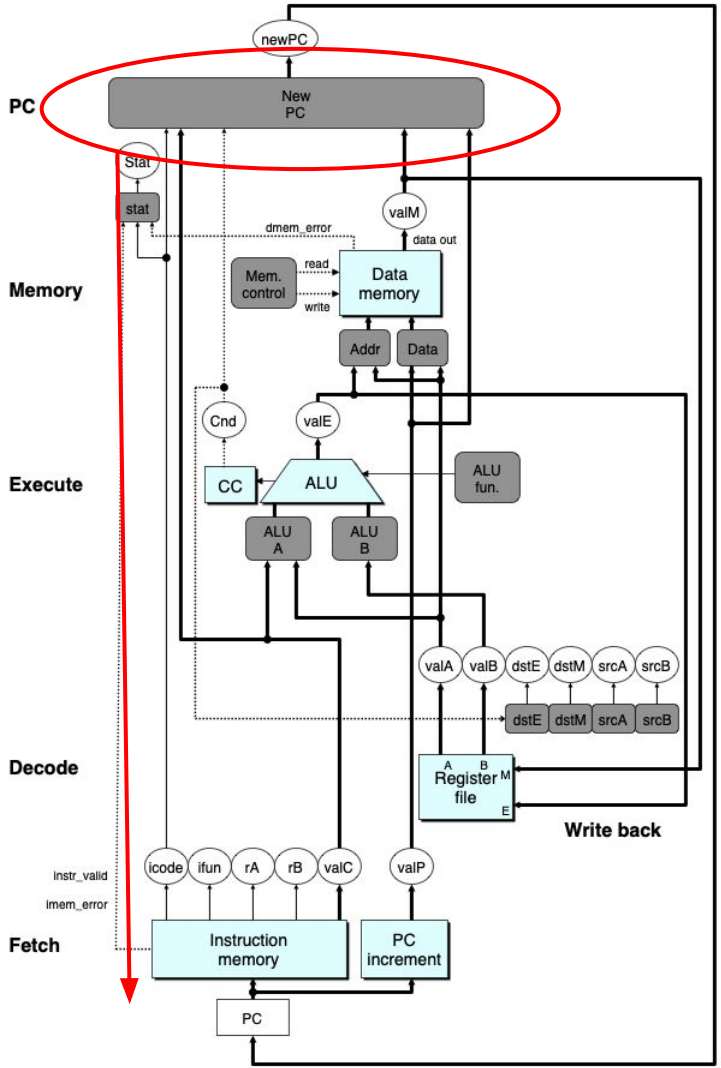
Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.



Pipelining: Converting from SEQ

SEQ Architecture

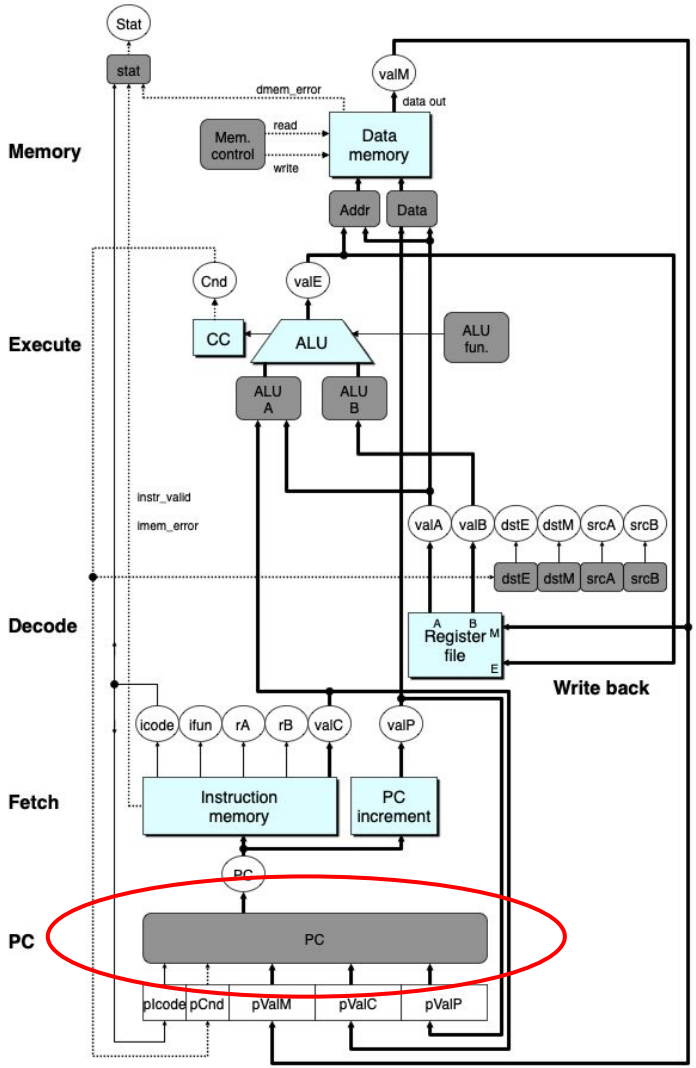
Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.



Pipelining: Converting from SEQ

SEQ Architecture

Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.



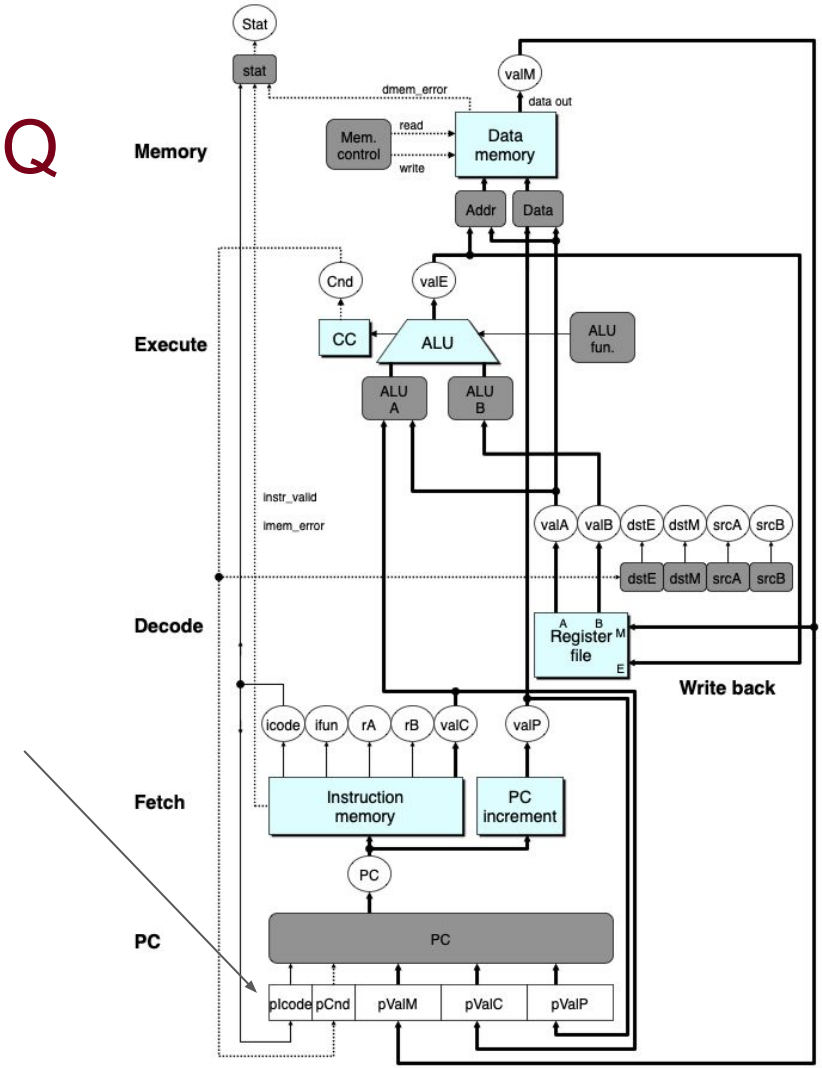
Pipelining: Converting from SEQ

SEQ Architecture

Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.

The book calls this “SEQ+”

Note that our single clocked register now stores multiple values, which are used to calculate PC



Pipelining: Converting from SEQ

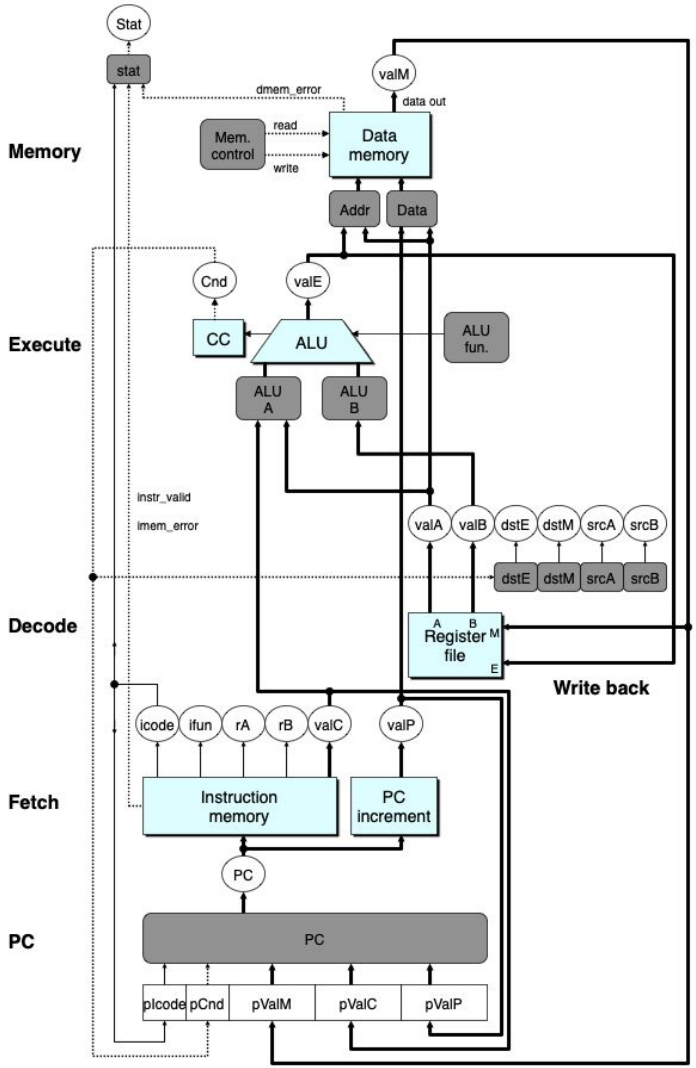
SEQ Architecture

Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.

The book calls this “SEQ+”

Note that our single clocked register now stores multiple values, which are used to calculate PC

Step 2: Add 5 pipeline registers, F, D, E, M, W



Pipelining: Converting from SEQ

SEQ Architecture

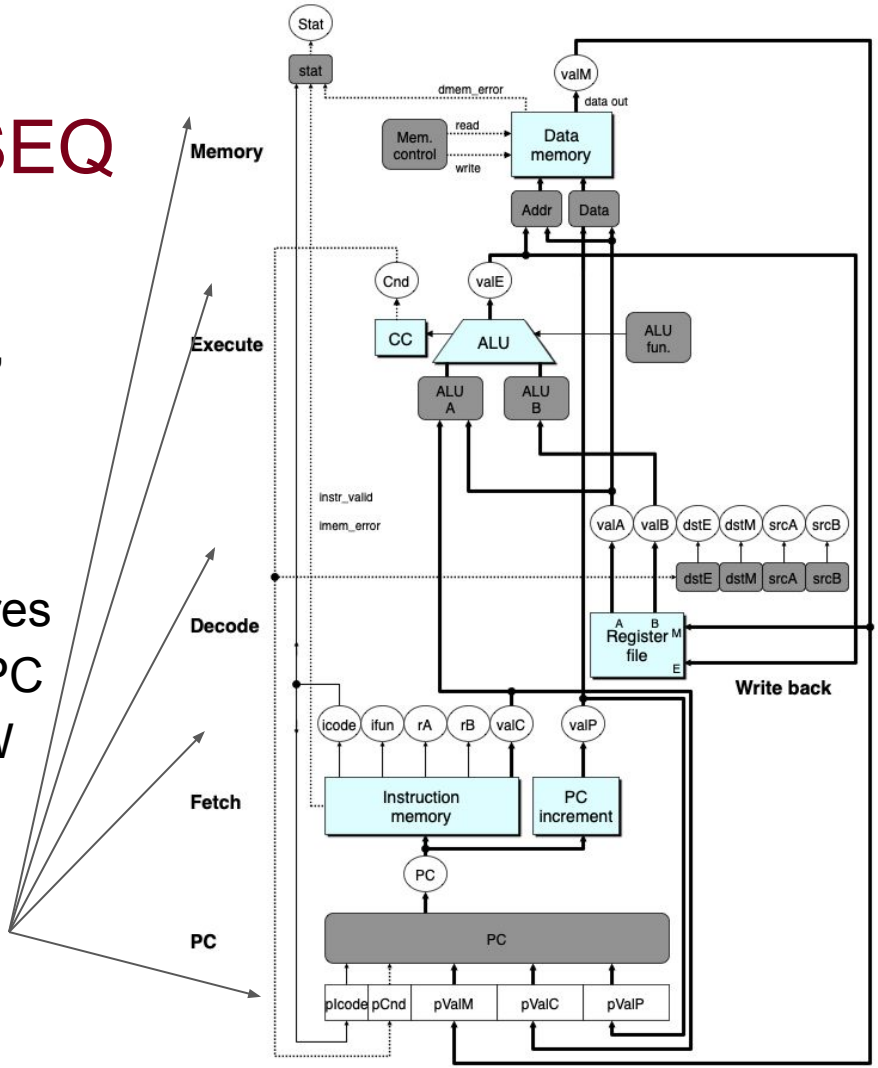
Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.

The book calls this “SEQ+”

Note that our single clocked register now stores multiple values, which are used to calculate PC

Step 2: Add 5 pipeline registers, F, D, E, M, W

The registers will go *before* each stage



Pipelining: Converting from SEQ

SEQ Architecture

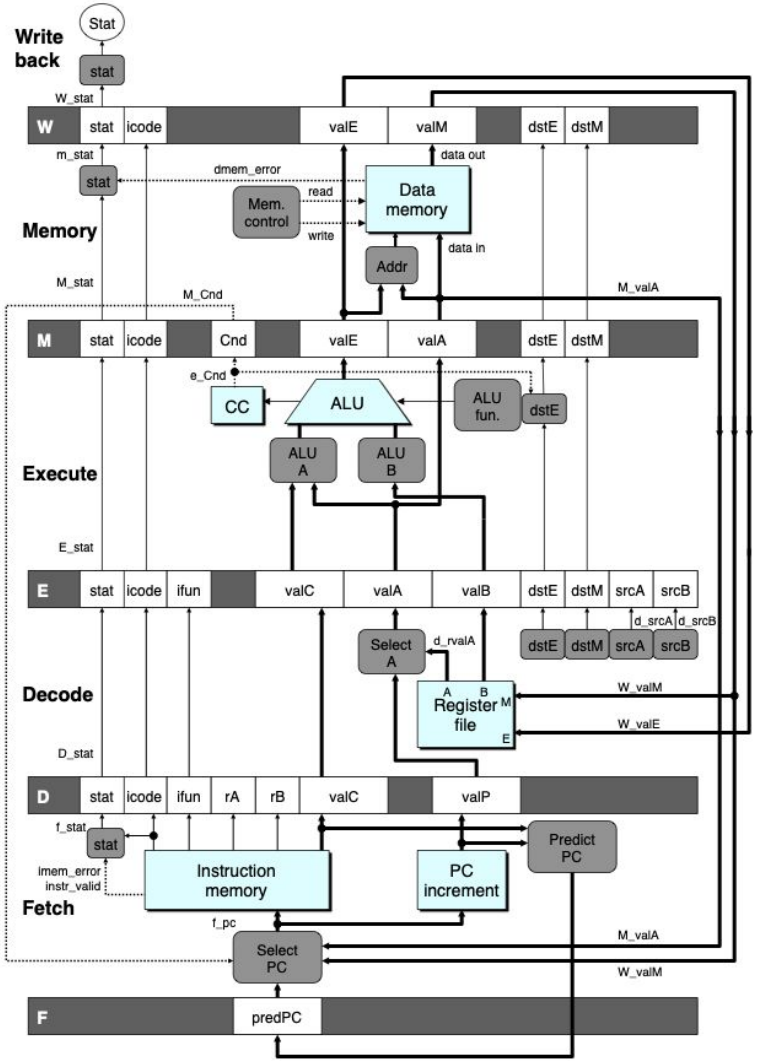
Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.

The book calls this “SEQ+”

Note that our single clocked register now stores multiple values, which are used to calculate PC

Step 2: Add 5 pipeline registers, F, D, E, M, W

The registers will go *before* each stage



Pipelining: Converting from SEQ

SEQ Architecture

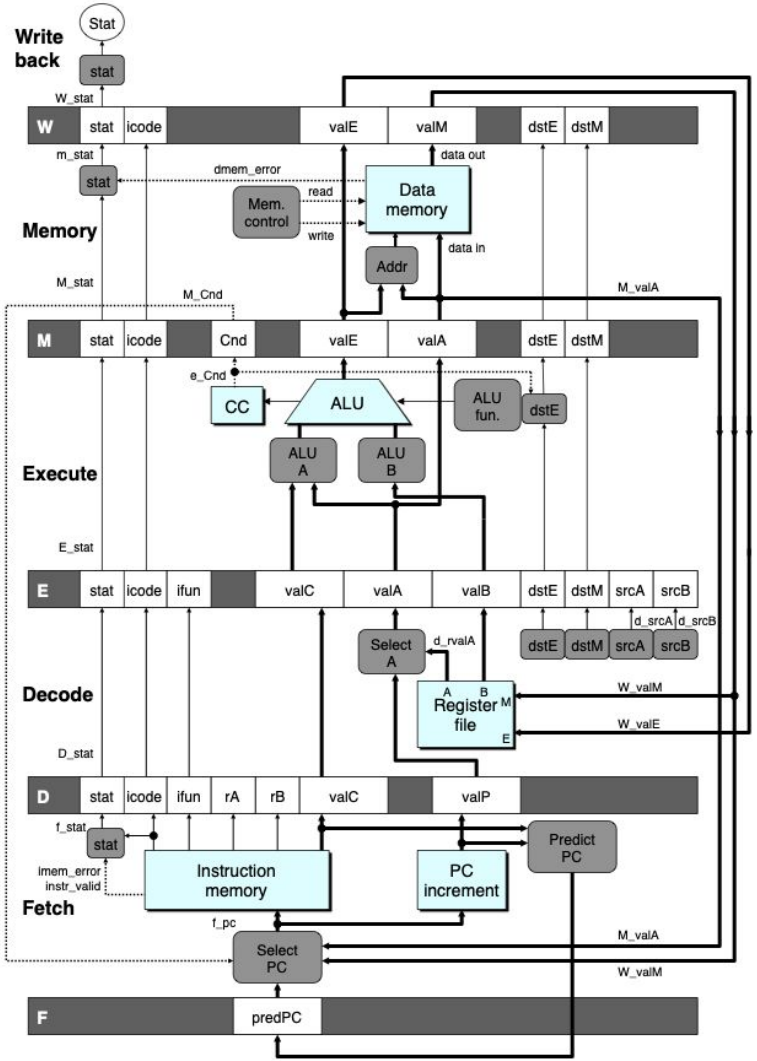
Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.

The book calls this “SEQ+”

Note that our single clocked register now stores multiple values, which are used to calculate PC

Step 2: Add 5 pipeline registers, F, D, E, M, W

Most of these pipeline registers hold multiple values



Pipelining: Converting from SEQ

SEQ Architecture

Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.

The book calls this “SEQ+”

Note that our single clocked register now stores multiple values, which are used to calculate PC

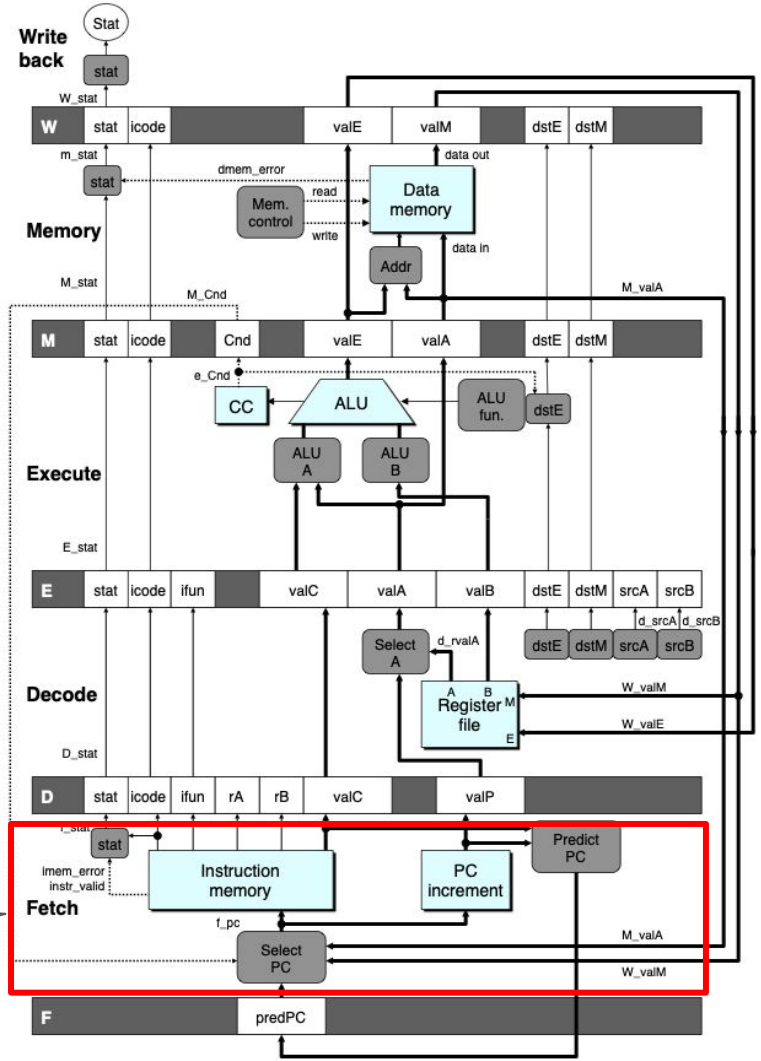
E, M, W

ultiple

```
irmovq $22, %rcx

addq %rcx, %rdi

mrmovq 100(%rdi), %r9
```



Pipelining: Converting from SEQ

SEQ Architecture

Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.

The book calls this “SEQ+”

Note that our single clocked register now stores multiple values, which are used to calculate PC

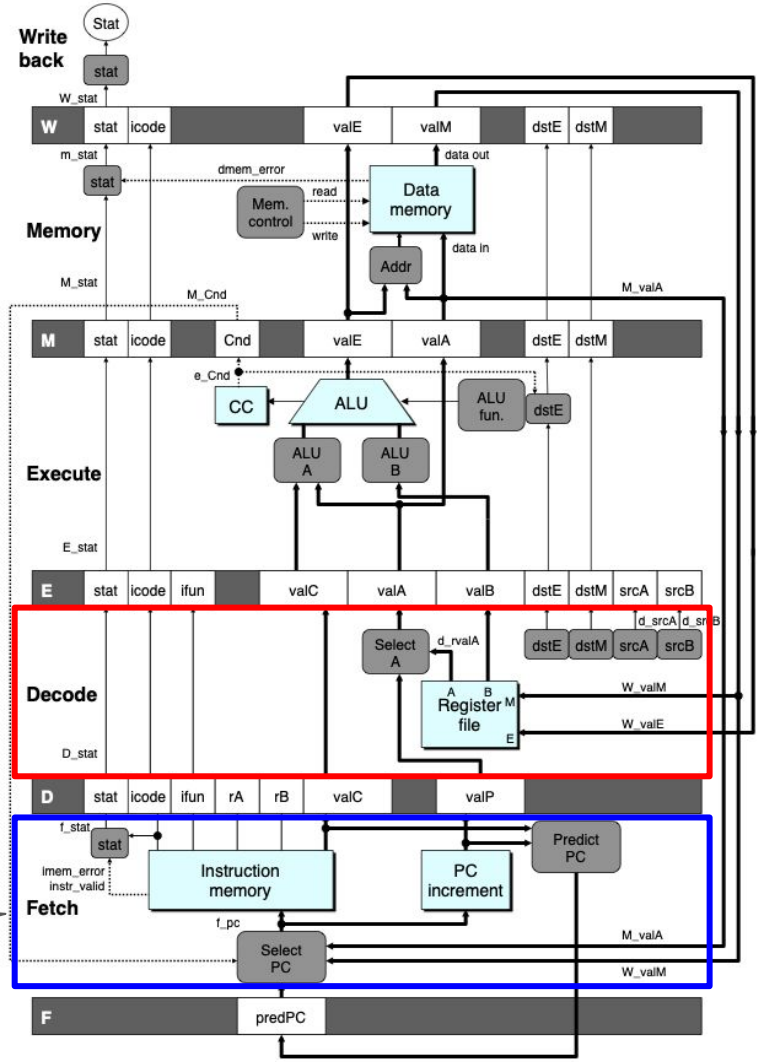
E, M, W

ultiple

```
irmovq $22, %rcx

addq %rcx, %rdi

mrmovq 100(%rdi), %r9
```



Pipelining: Converting from SEQ

SEQ Architecture

Step 1: We will make a small change to SEQ, moving PC calculation logic from “end” to “beginning” of the combinational logic.

The book calls this “SEQ+”

Note that our single clocked register now stores multiple values, which are used to calculate PC

E, M, W

ultiple

```
irmovq $22, %rcx

addq %rcx, %rdi

mrmovq 100(%rdi), %r9
```