TE2003B
SoC Design: Computer organisation & architecture
Pipeline

Isaac Pérez Andrade

ITESM Guadalajara
School of Engineering & Science
Department of Computer Science
February – June 2021

References

The following material has been adopted and adapted from

Patterson, D. A., Hennessy, J. L., *Computer Organization and design: The hardware/software interface – ARM edition*, Morgan Kaufmann, 2017.

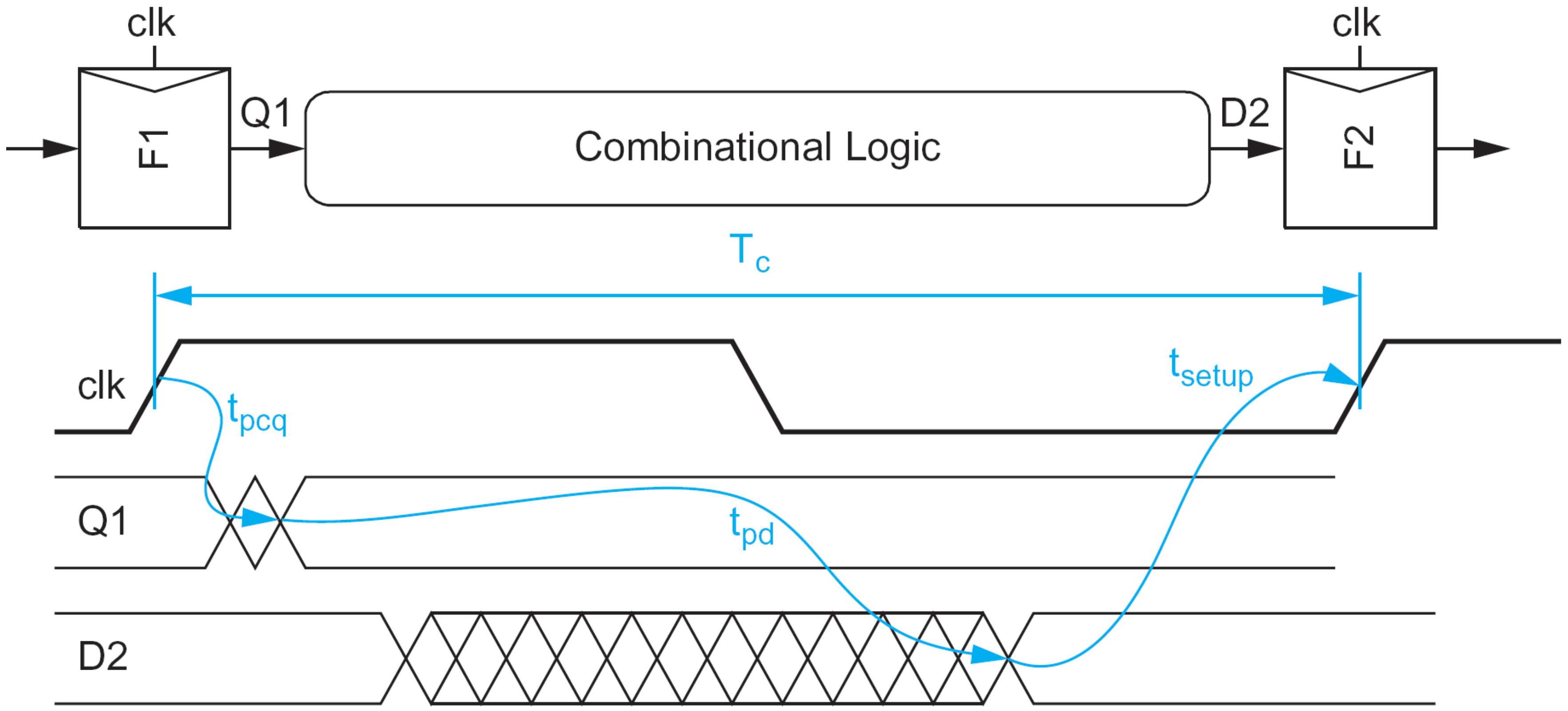
S. L. Harris and D. M. Harris, *Digital design and computer architecture - ARM edition*, Morgan Kaufmann, 2016.

J. Yiu, *The definitive guide to ARM Cortex-M0 and Cortex-M0+ processors*, Second edition, Elsevier, 2015.

Background

- Clock frequency in ICs is determined by the longest propagation delay.
- Propagation delay is the time it takes for a signal to propagate from
  - An input to a flip-flop
  - A flip-flop to an output
  - A flip-flop to another flip-flop

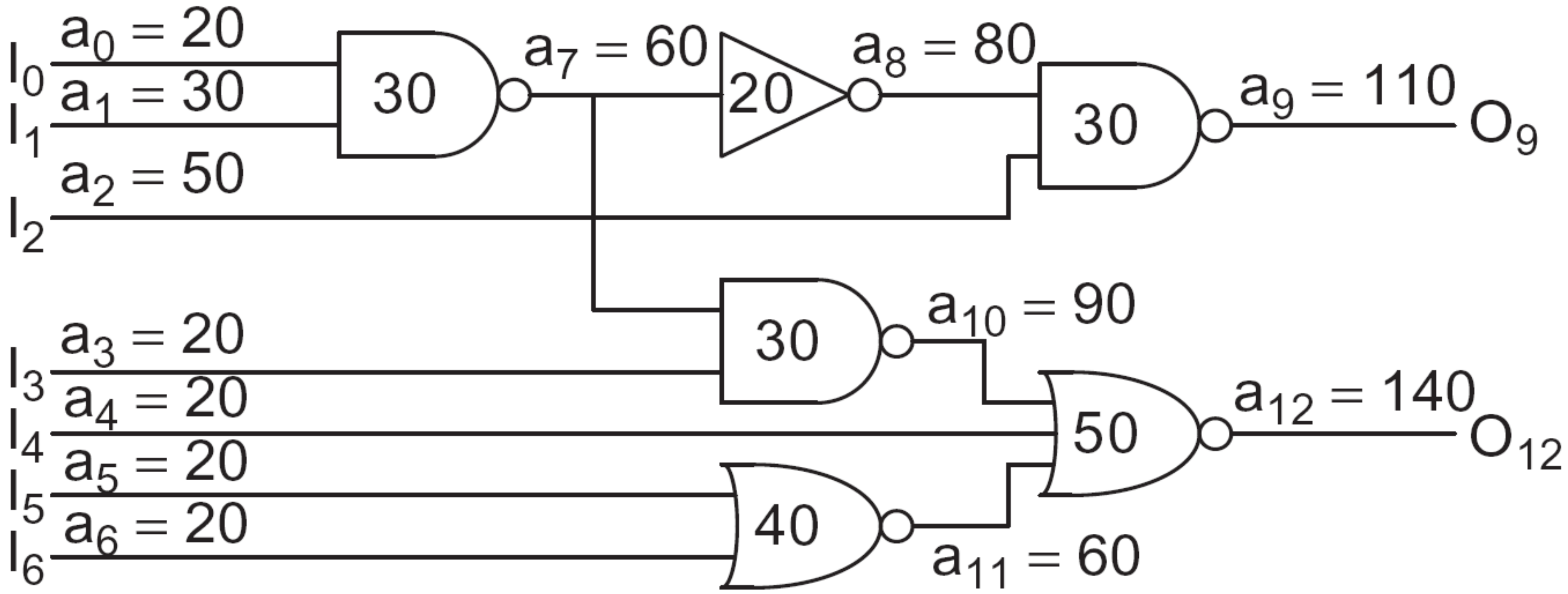
Propagation delay



- Clock period should be larger than largest propagation delay
  - Tc > tpd

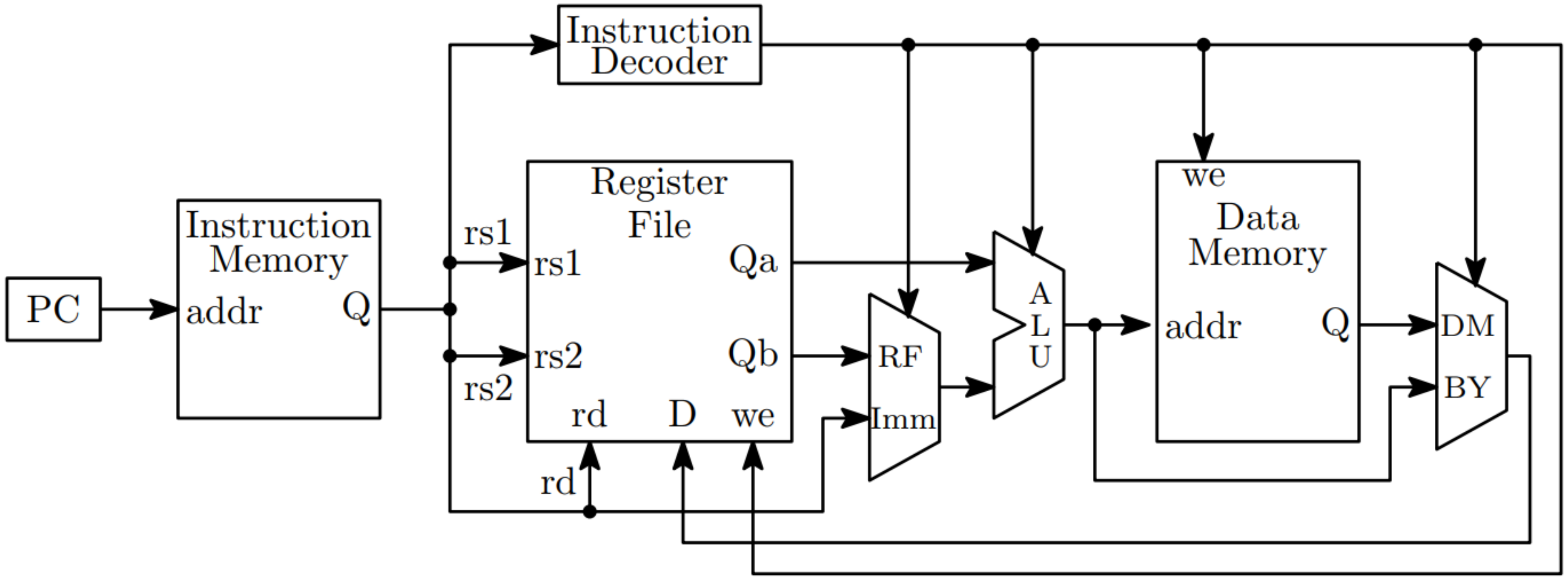
Critical path

- Critical path is the longest propagation delay in a circuit.
    - It determines the minimum clock period of a circuit.
- Which is the critical path in the following



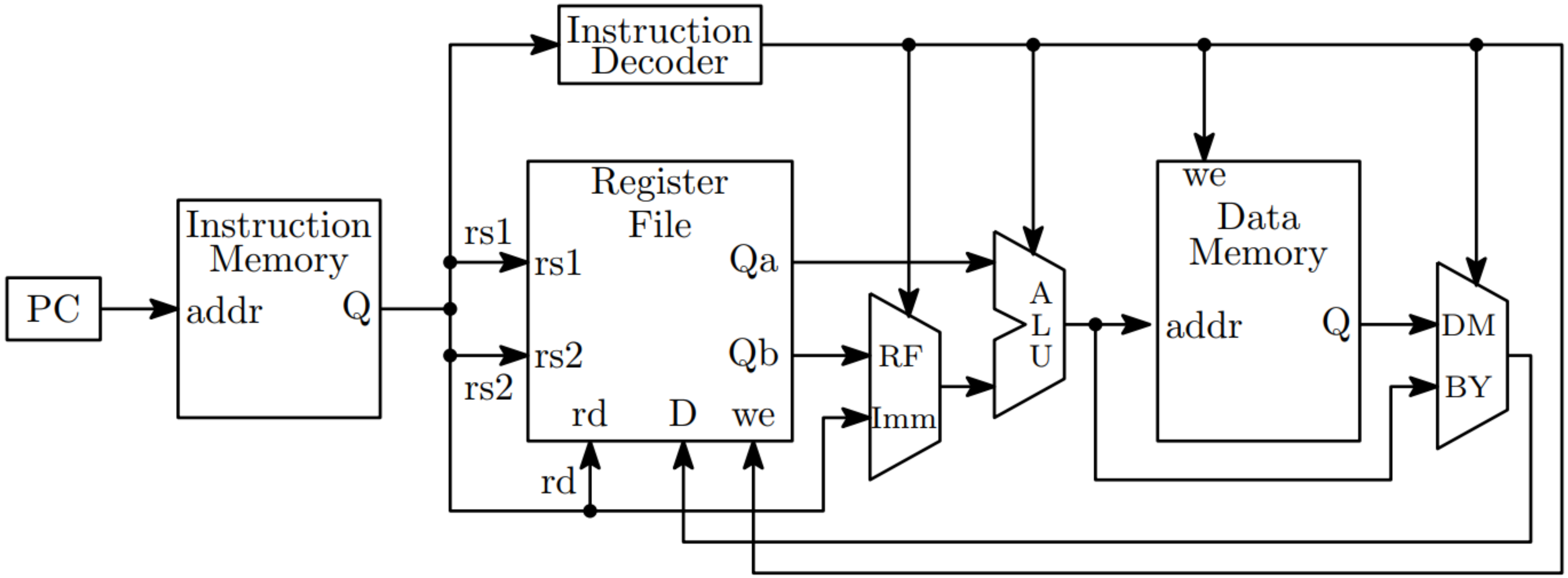
Critical path

- Which is the critical path in this simplified microprocessor block diagram?
  - Which type of instruction takes the longest to execute?



Critical path

- LDR (load) instruction.



- PC → Instruction memory → register file → mux → ALU → data memory → mux → register file

Critical path

- This critical path limits the clock frequency and the number of instructions per second that we could theoretically perform.

- For example, if LDR instruction takes 10 ns to execute, what's the maximum clock frequency we can operate?

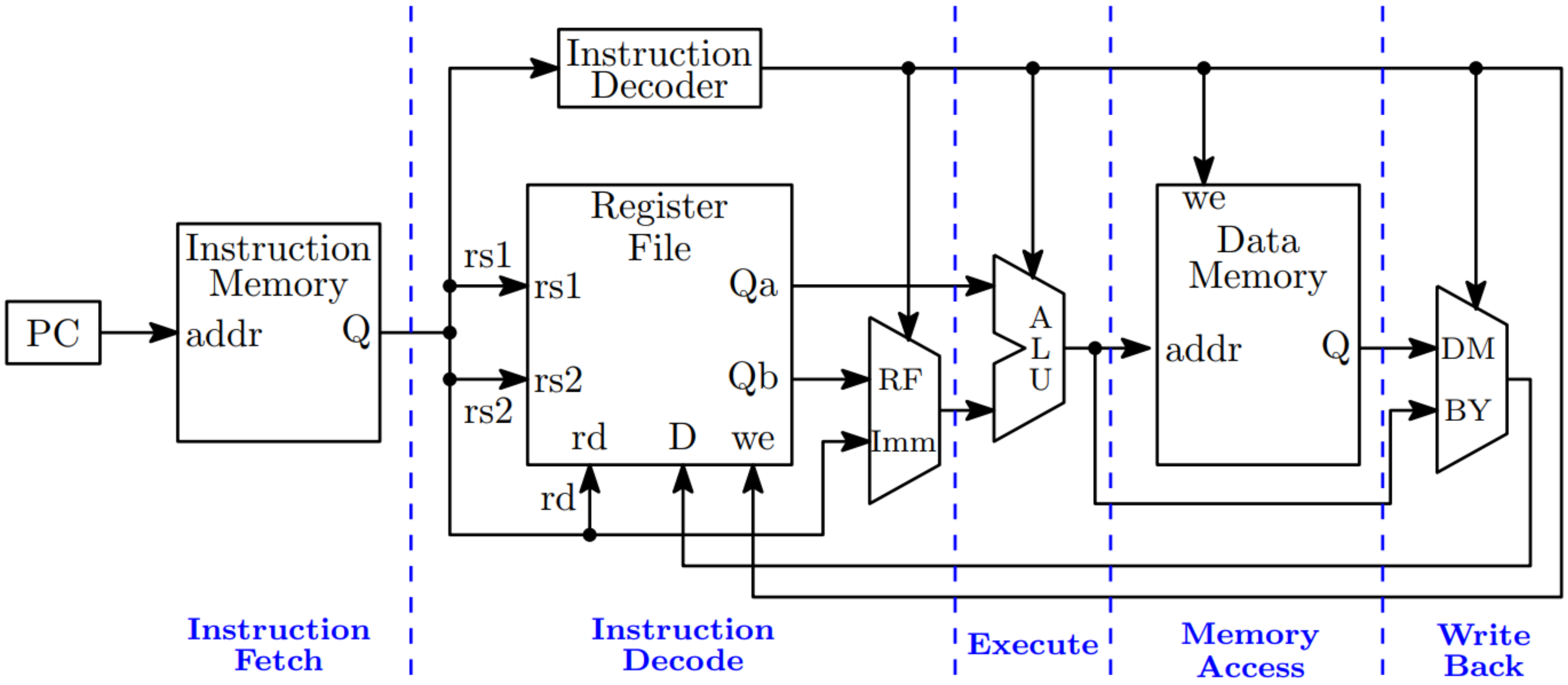
$$f_{max} = \frac{1}{10\,ns} = 100MHz$$

Critical path

- What if we are required to run at a faster clock frequency?
    - For example, our processor must operate at 500 MHz or nobody would ever buy it!
- We could try to reduce the length of the critical path.
    - Split the critical path into smaller logic chunks.
    - Include flip-flops in the boundary of each new logic chunk.

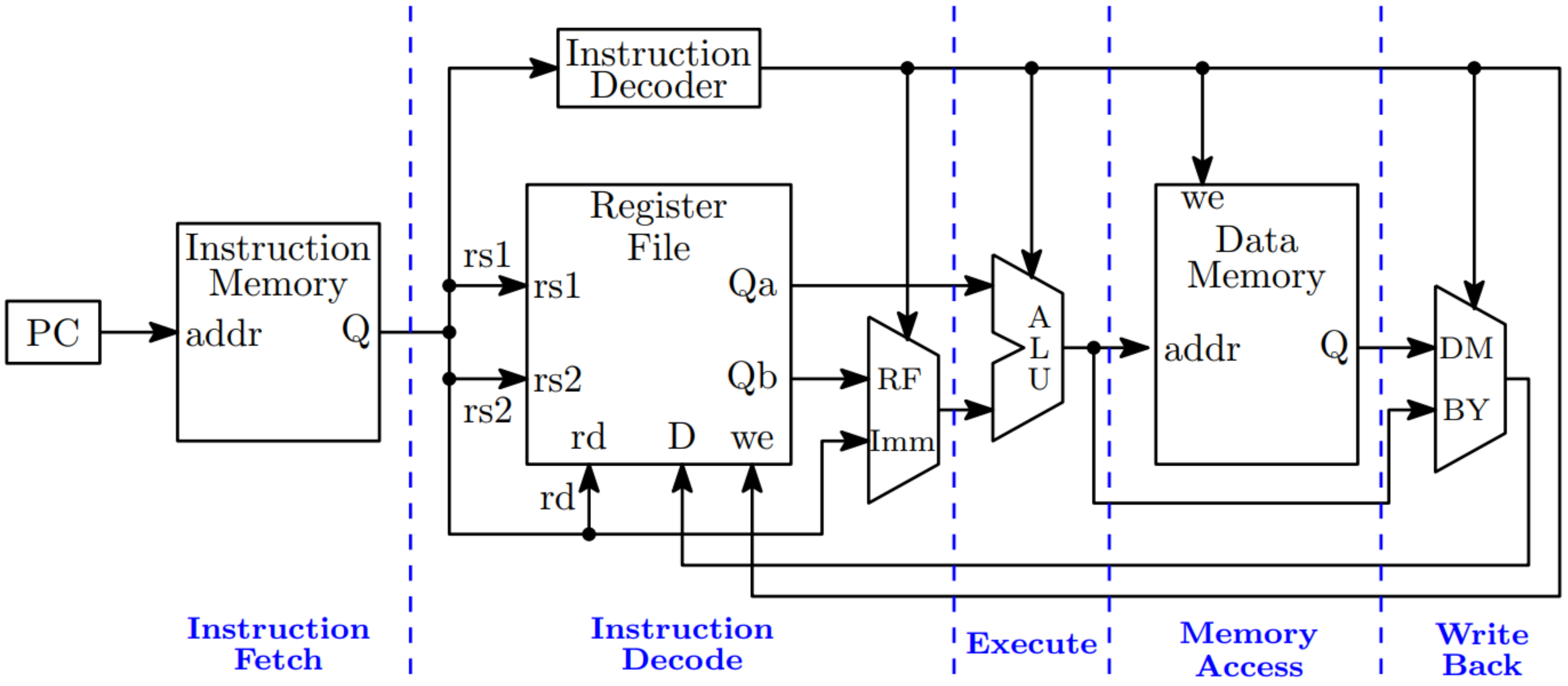
Splitting the critical path

- We could split our single-cycle microprocessor execution into different stages.



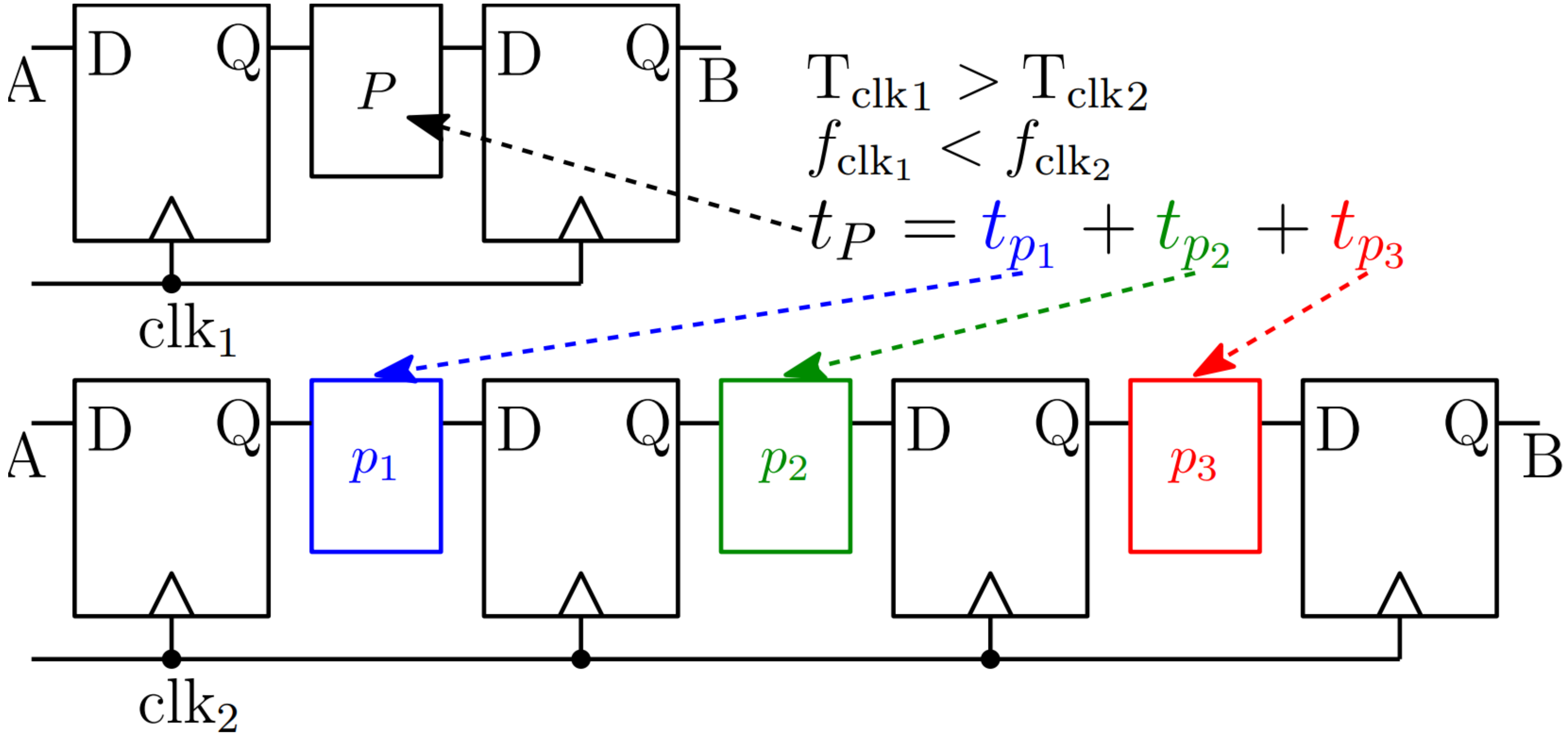
Splitting the critical path

- This is the principle behind ***pipelining.***



Pipelining basics

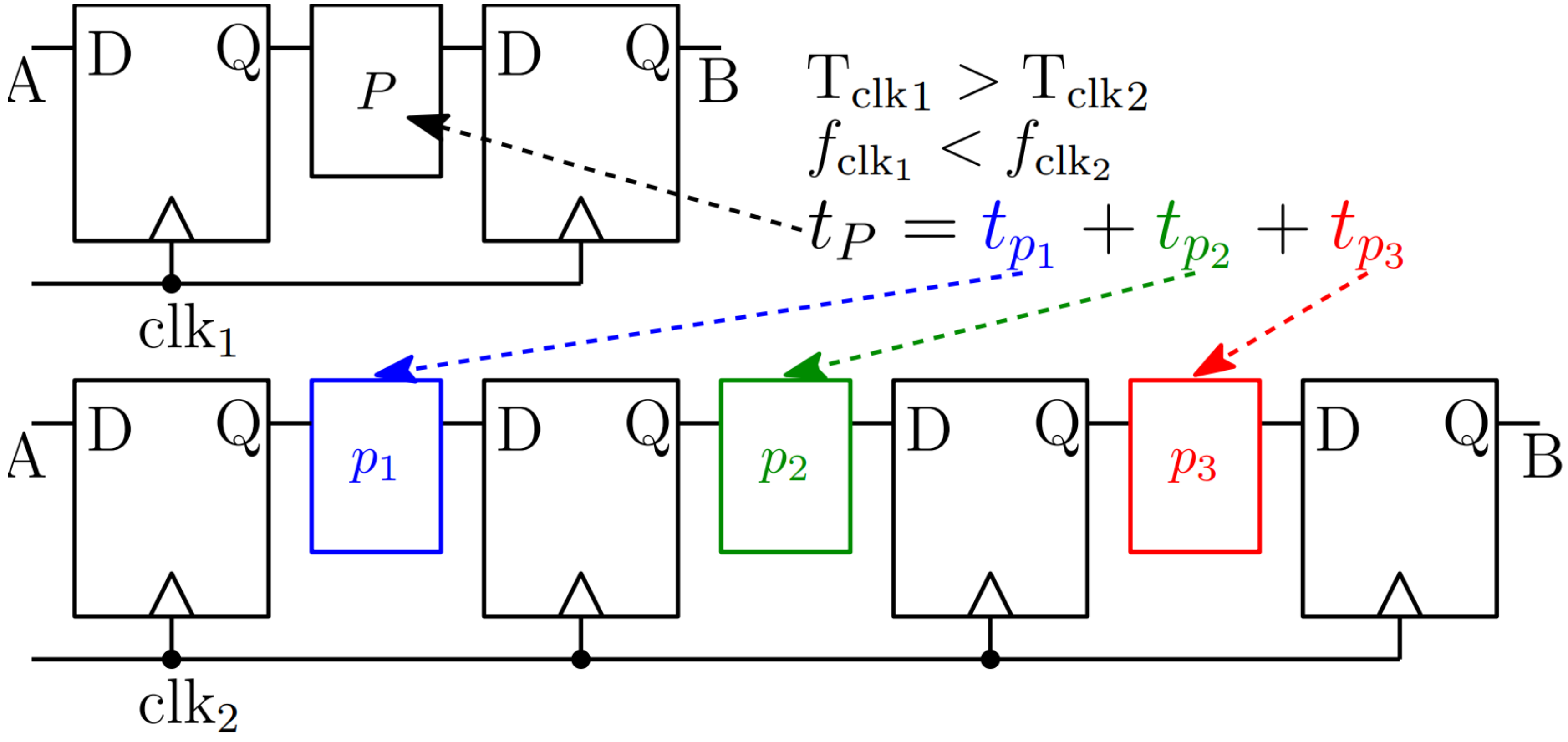
- Pipelining consist in breaking down a single task into several stages.



A
D
Q
P
D
Q
B
$\mathrm{T}_{\mathrm{clk}1} > \mathrm{T}_{\mathrm{clk}2}$
$f_{\mathrm{clk}_1} < f_{\mathrm{clk}_2}$
$t_P = t_{p_1} + t_{p_2} + t_{p_3}$
clk_1
A
D
Q
p_1
D
Q
p_2
D
Q
p_3
D
Q
B
clk_2

Pipelining basics

- Task from A to B takes the same time in both cases. What's the advantage?



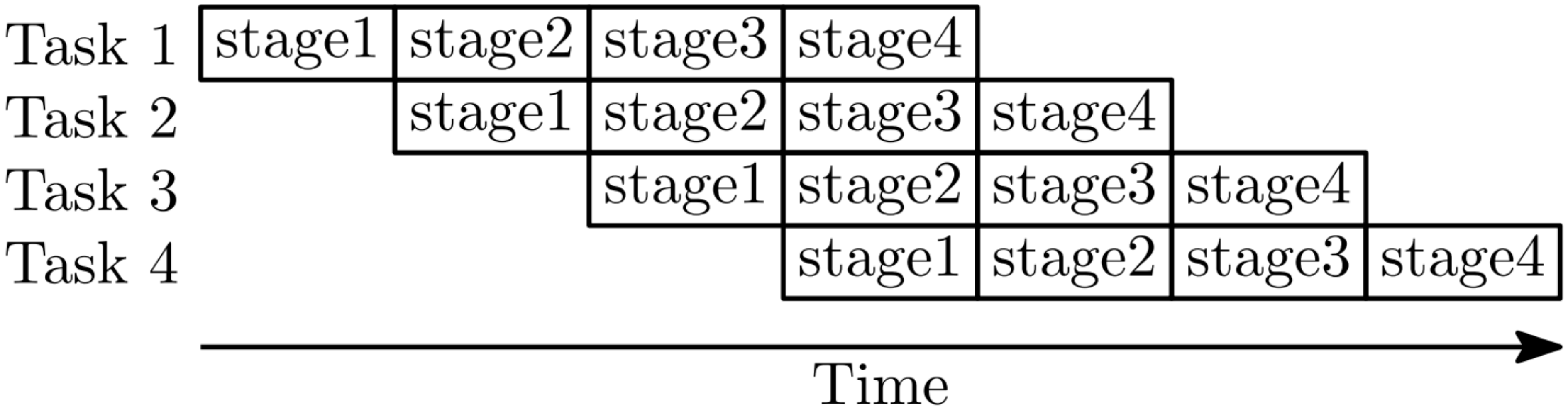
$\mathrm{T}_{\mathrm{clk}1} > \mathrm{T}_{\mathrm{clk}2}$

$f_{\mathrm{clk}_1} < f_{\mathrm{clk}_2}$

$t_P = t_{p_1} + t_{p_2} + t_{p_3}$

Pipeline basics

- We can now perform tasks in parallel!

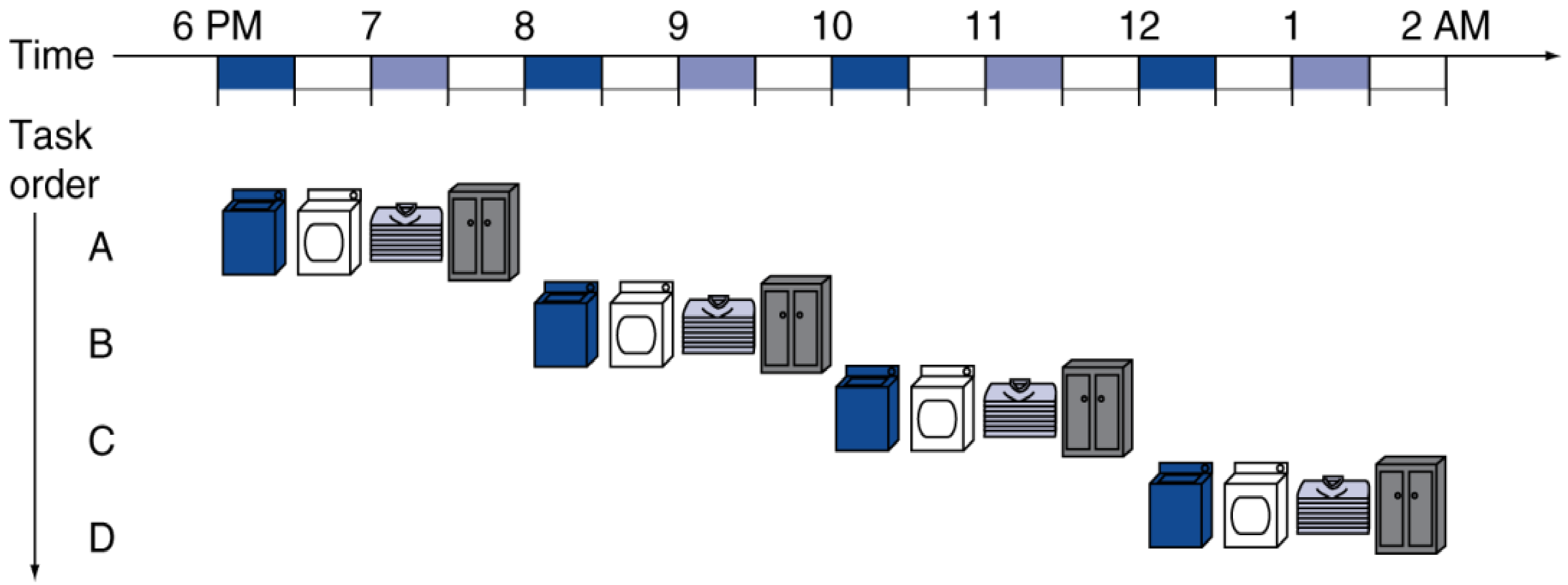


Pipeline analogy

- Assume you work in a laundry shop.
- You divide your job into the following stages.
    1. Wash (stage1)
    2. Dry (stage2)
    3. Fold (stage3)
    4. Store (stage4)
- Assume each stage takes 30 minutes to complete.
- How low would it take you to complete a single laundry task?

Pipeline analogy

- 30min x 4 stages = 2 hours



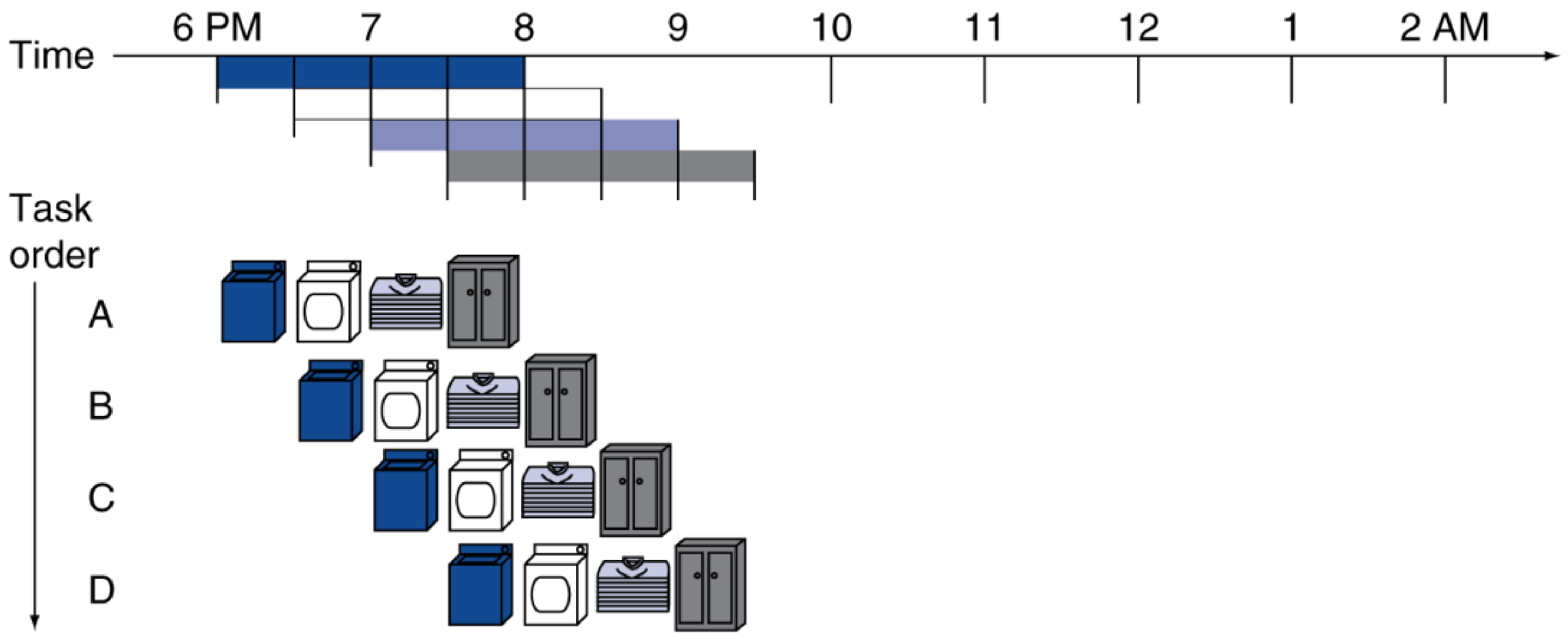
- What about 4 laundry tasks?
  - $4 \times (30 \times 4) = 480$ minutes $= 8$ hours!
- What about 1000 laundry tasks?

Pipeline analogy

- Improving our laundry efficiency.
  1. Start a new load right after the washing machine becomes available.
  2. Dry first load and wash second load at the same time.
  3. Fold first load, dry second load and wash third load at the same time
  4. Continue with this principle until you complete all four loads

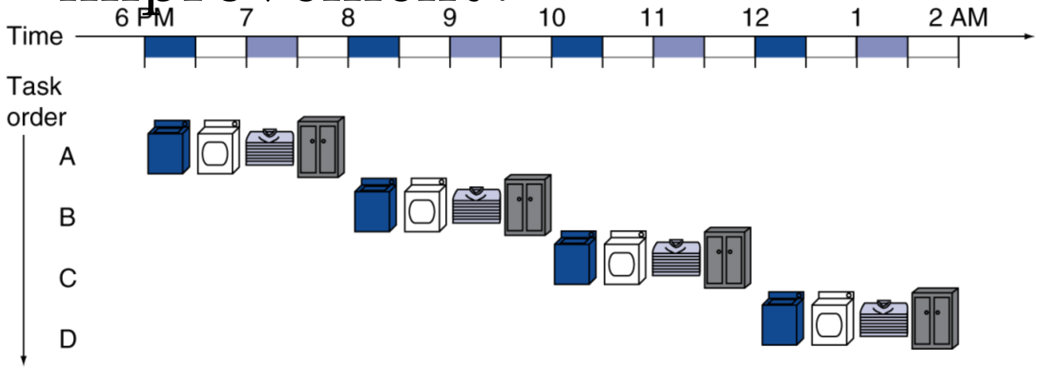
Pipeline analogy

- 4 loads now take 3.5 hours, compared to 8 hours in a sequential scheme.



Pipeline speedup

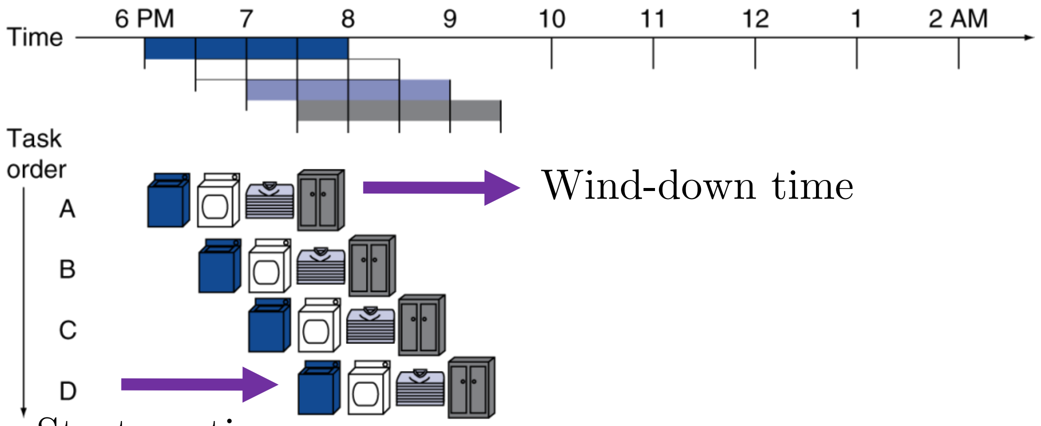
- How can we measure the performance improvement?



Four loads:

Speedup = 8/3.5 = 2.28

We are doing our job 2.3 times faster!



Wind-down time

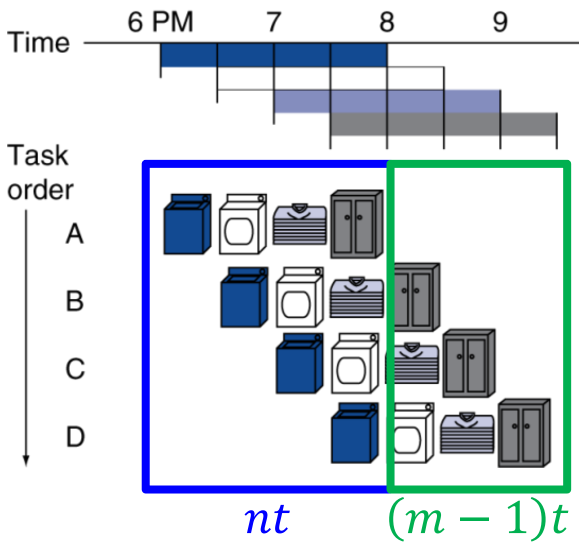
Start-up time

Full pipeline: All resources are used at the same time

What about infinite number of loads?

Pipeline speedup

- Speedup with an infinite number of laundry loads.



m is the number of tasks
n is the number of stages per task
t is the time required to complete a stage

$$T_{seq} = m \cdot \sum_{i=1}^{n} t_i$$

If all n stages take the same time t, then

$$T_{seq} = m \cdot n \cdot t$$

$$T_{pipe} = (m-1) \cdot t + n \cdot t$$

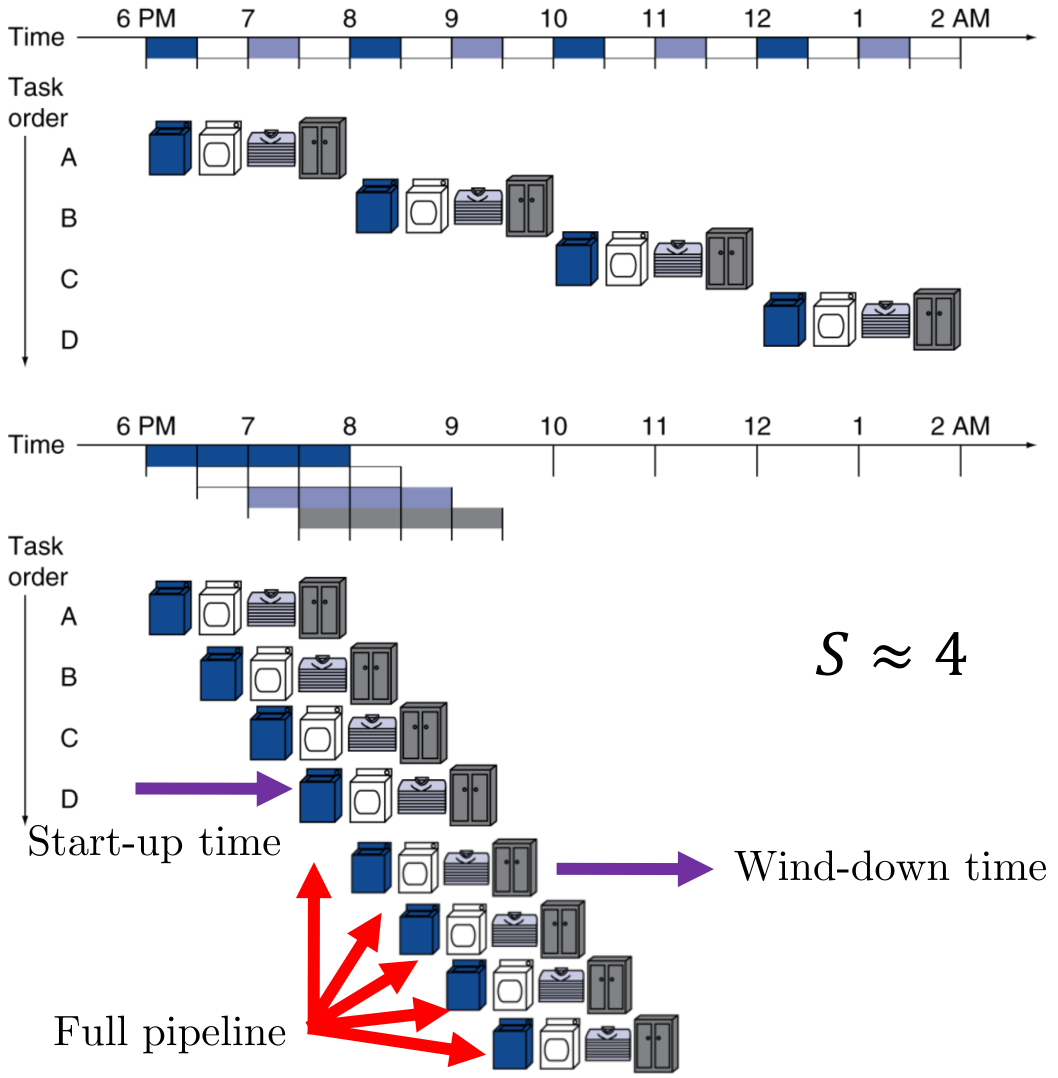
$$S = \frac{T_{seq}}{T_{pipe}} = \frac{m \cdot n \cdot t}{(m-1) \cdot t + n \cdot t} = \frac{m \cdot n}{(m-1)+n}$$

$$\lim_{m \to \infty} \frac{m \cdot n}{(m-1)+n} = n$$

Speedup is directly determined by the number of pipeline stages

Pipeline speedup

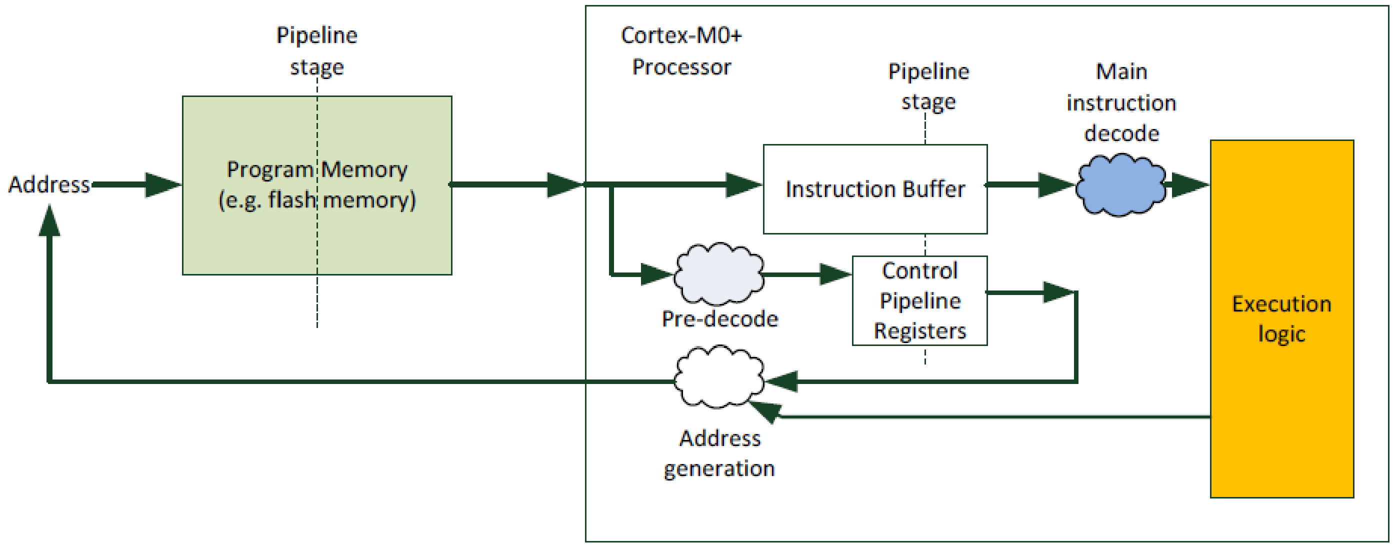
- Speedup for an infinite number of laundry loads

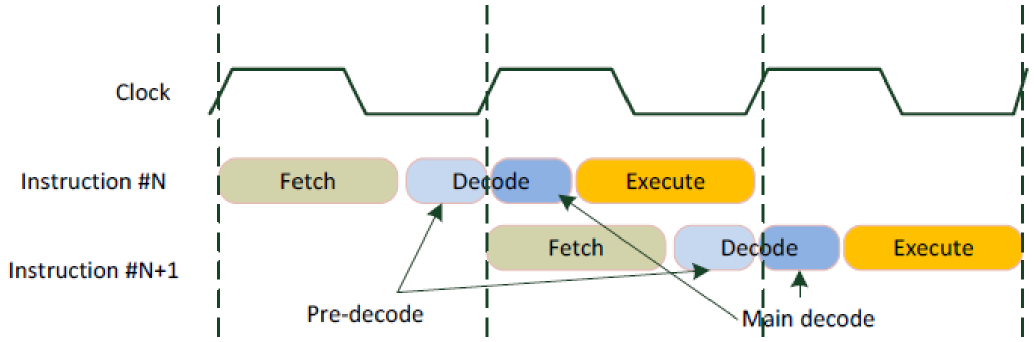


$S \approx 4$

Cortex-M0+ pipeline

- Cortex-M0+ implements a two-stage pipeline.





- We will focus on a more general five-stage pipeline

ARM pipeline

- ARM pipeline. Five instruction cycle stages, one step per stage.
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

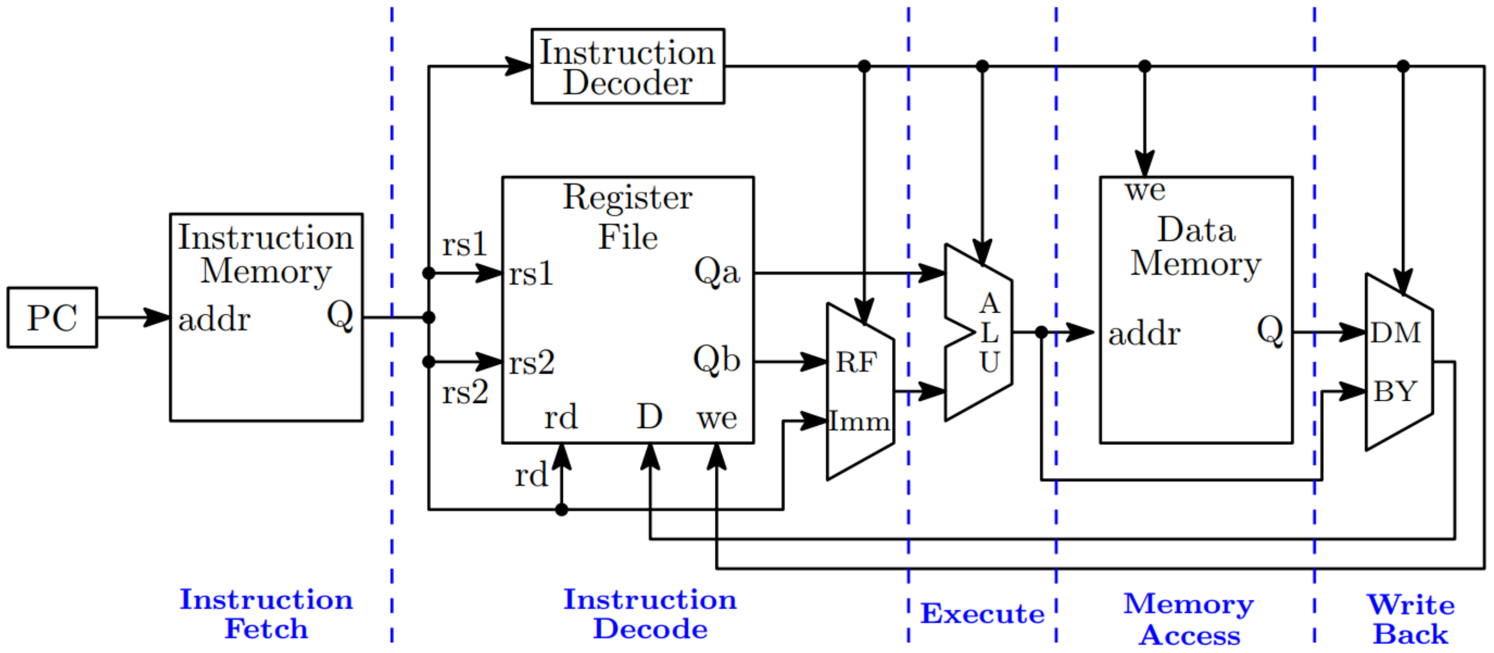
Pipeline performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Fetch	Decode	Execute	Memory access	Write back	Total
LDR	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
STR	200 ps	100 ps	200 ps	200 ps		700 ps
R-Type	200 ps	100 ps	200 ps		100 ps	600 ps
CBZ	200 ps	100 ps	200 ps			500 ps

Pipeline performance

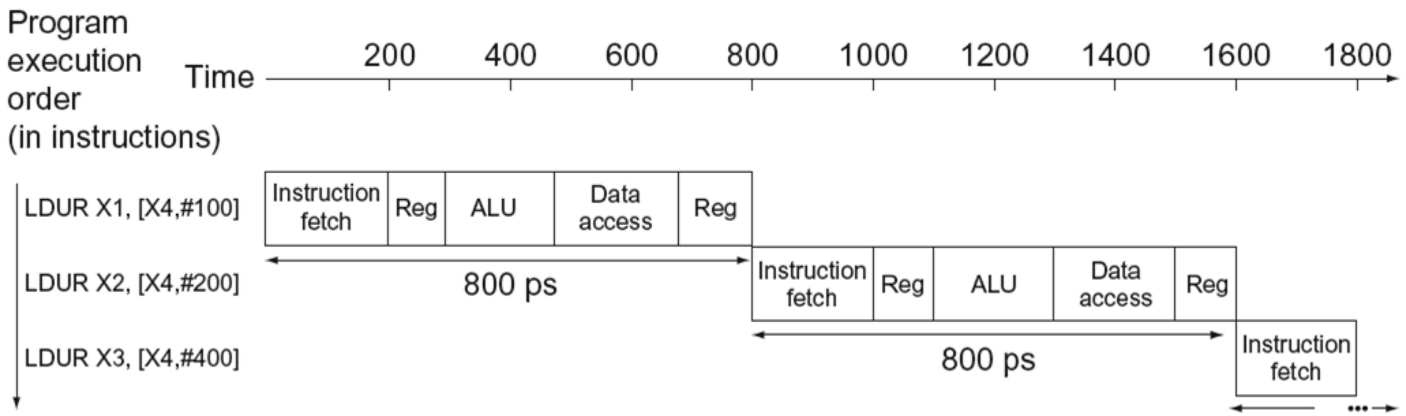
Instr	Fetch	Decode	Execute	Memory access	Write back	Total
LDR	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
STR	200 ps	100 ps	200 ps	200 ps		700 ps
R-Type	200 ps	100 ps	200 ps		100 ps	600 ps
CBZ	200 ps	100 ps	200 ps			500 ps



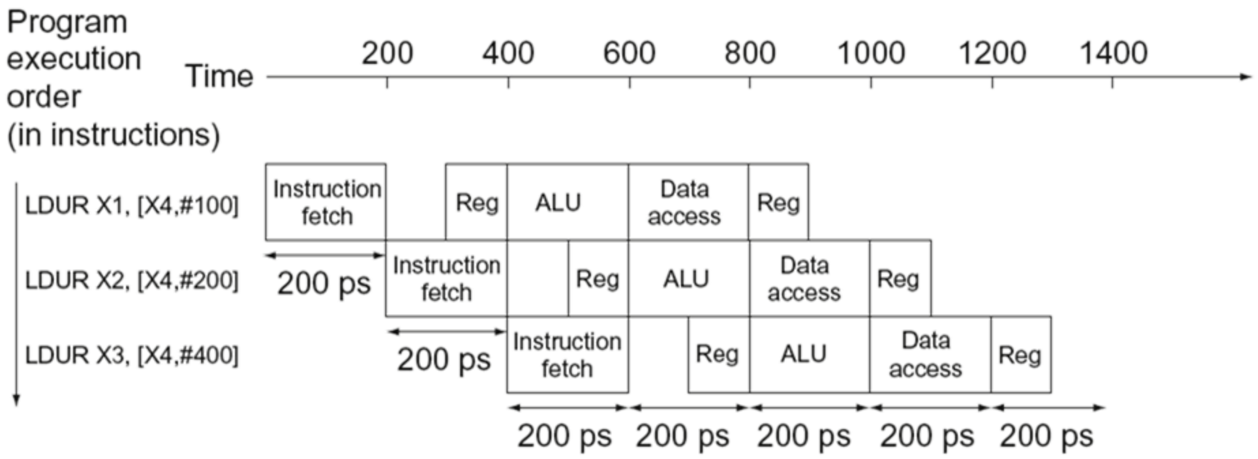
Pipeline performance

- What's the critical path?

Single-cycle (T_c= 800ps) → 1 LDR takes 800 ps



Pipelined (T_c= 200ps) → 1 LDR takes 1000 ps



Pipeline speedup

- Pipeline increases throughput.
  - Throughput. Number of tasks performed in a given time.
  - Instruction Level Parallelism (ILP).
- Pipeline does not improve (reduce) instruction latency. In fact, instruction latency is usually increased.
  - Latency. Required time for completing a task.

Pipelining and ISA design

- ARM ISA designed for pipelining
    - All instructions are 32-bits
        - Easier to fetch and decode in one cycle
        - Compared to Intel's x86 ISA using between 1- to 17-byte instructions.
    - Few and regular instruction formats
        - Can decode and read registers in one step
    - Load/store addressing
        - Can calculate address in 3rd stage, access memory in 4th stage
    - Alignment of memory operands
        - Memory access takes only one cycle

Pipeline hazards

Hazards

- Situations that prevent starting the next instruction in the next cycle.
- Structure hazards
    - A required resource is busy
- Data hazard
    - Need to wait for previous instruction to complete its data read/write
- Control hazard
    - Deciding on control action depends on previous instruction

Structure hazards

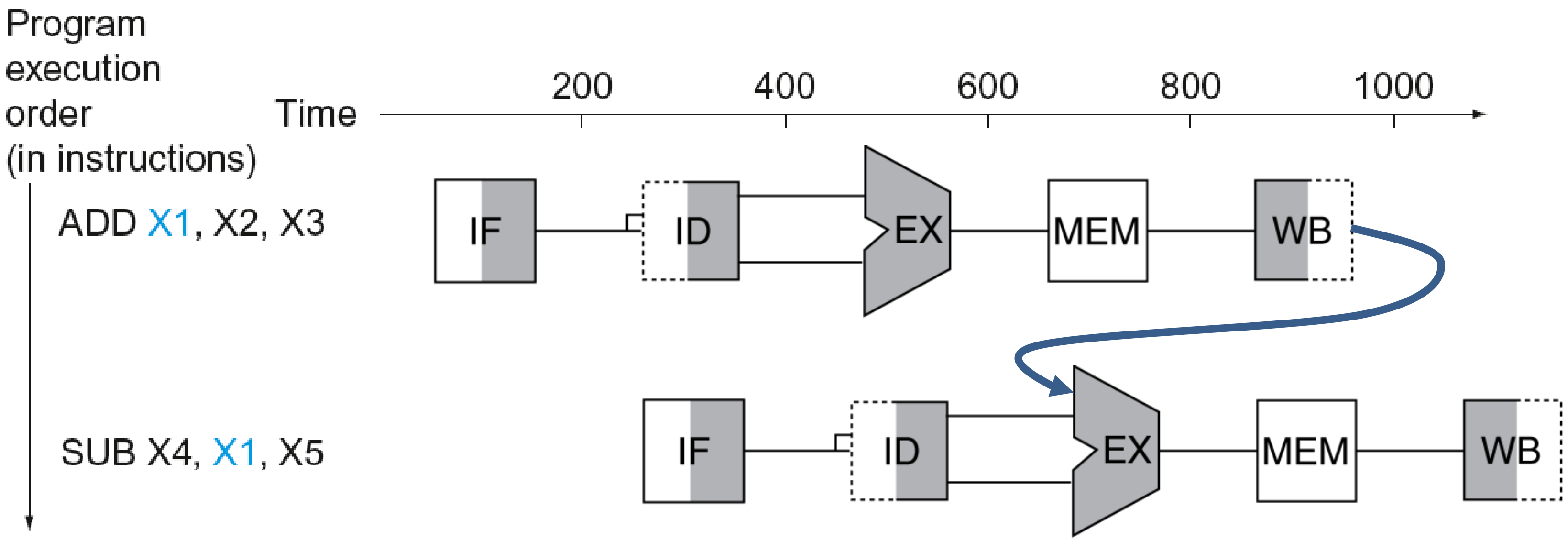
- Hardware cannot support the combination of instructions in the same clock cycle.
- Conflict for use of a resource.
  - We require to use the dryer for two different sets of clothes exactly at the same time.
- ARM pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Data hazards

- An instruction depends on completion of data access by a previous instruction

```
ADD x1, x2, x3
SUB x4, x1, x5
```

We can't go back in time!

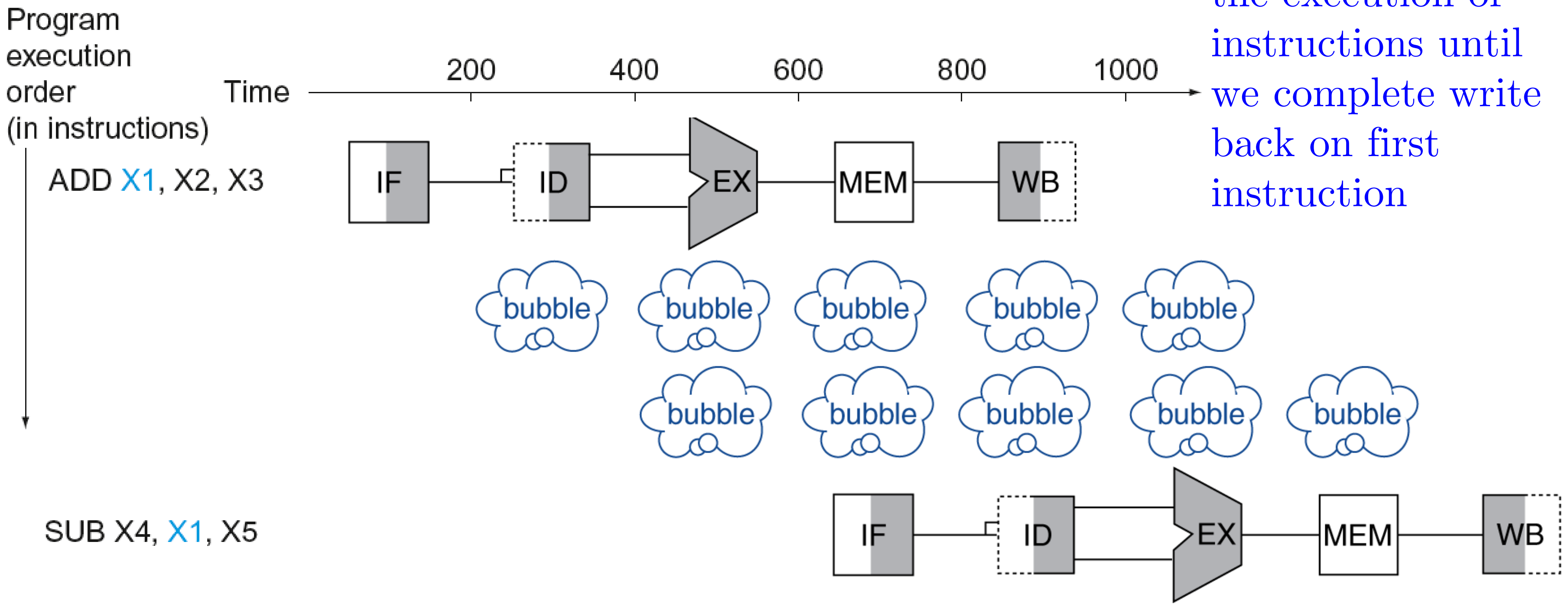


Data hazards

- An instruction depends on completion of data access by a previous instruction

```
ADD x1, x2, x3
SUB x4, x1, x5
```

Instead of going back in time, we must stall (stop) the execution of instructions until we complete write back on first instruction



Data hazards

- Compiler might detect this data dependency and insert bubbles (NOP instructions).

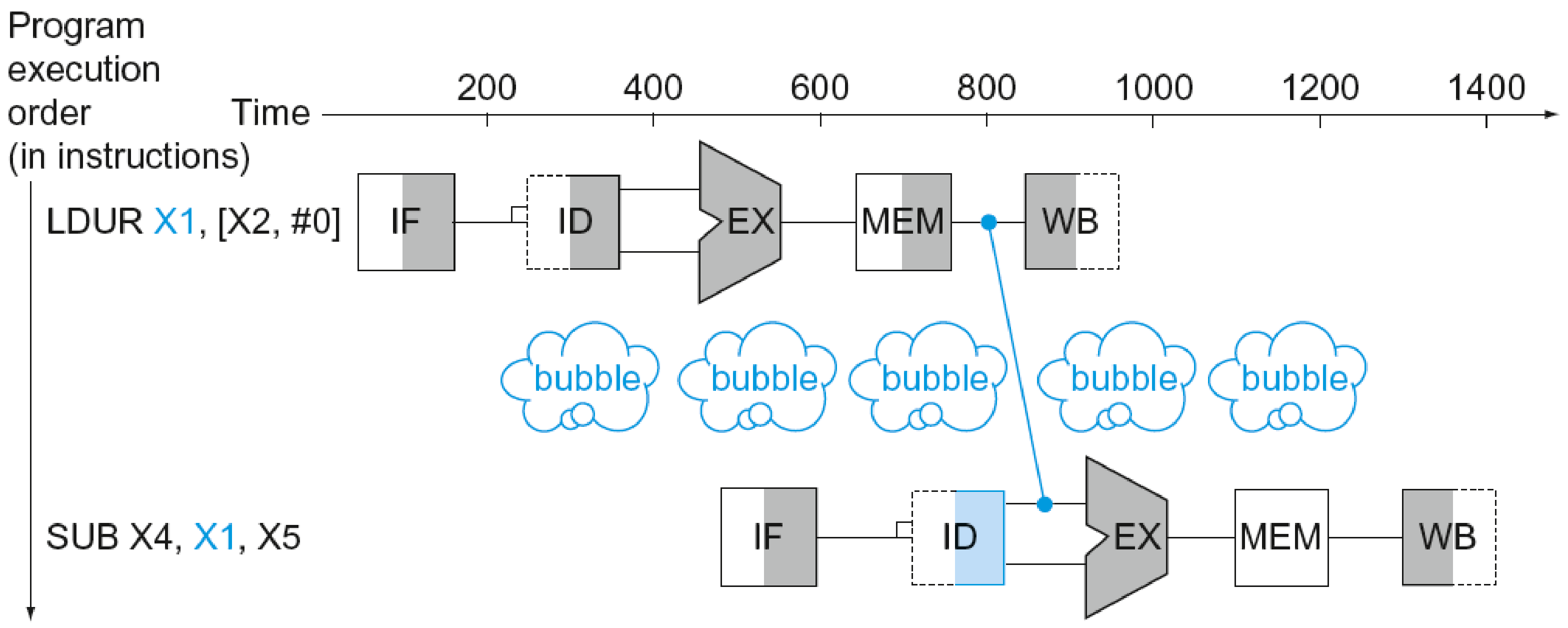
```
ADD x1, x2, x3
NOP
NOP
SUB x4, x1, x5
```

- NOP instructions do not do anything.
  - Used for creating pipeline bubbles.

Data hazards

- Load instruction

```
LDUR X1, [x2, #0]
SUB X4, X1, X5
```

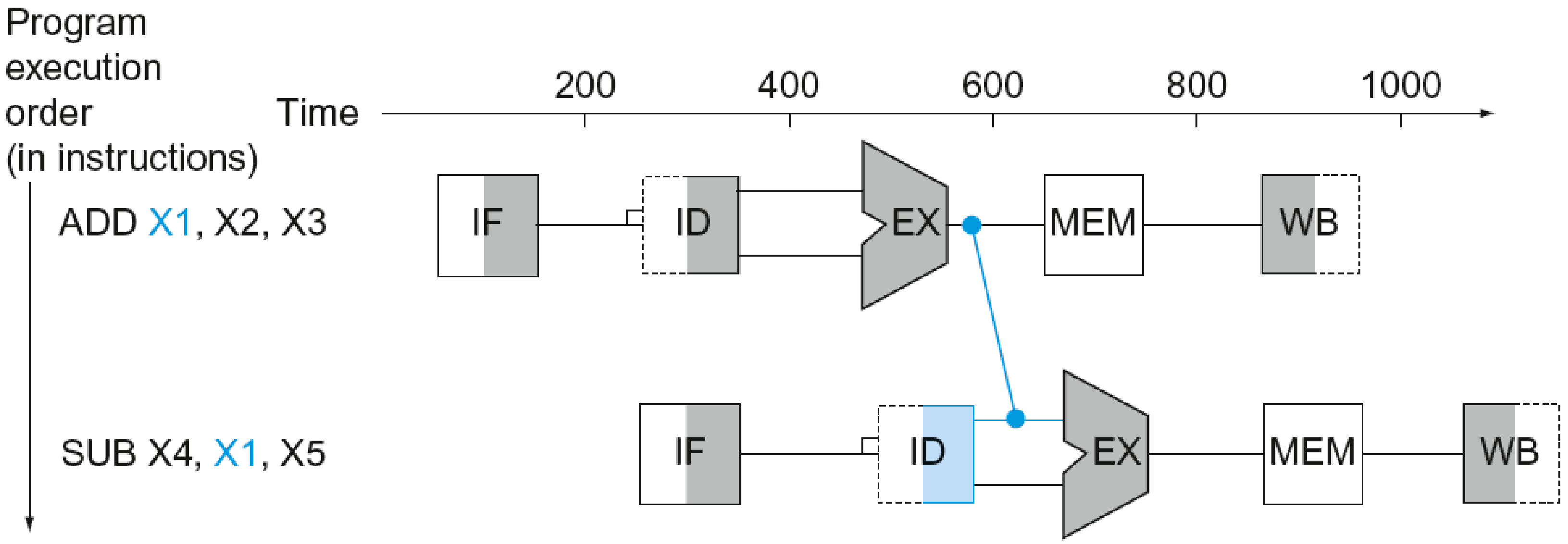


Data hazards – forwarding (aka bypassing)

- Forwarding tries to mitigate bubbles due to data hazards

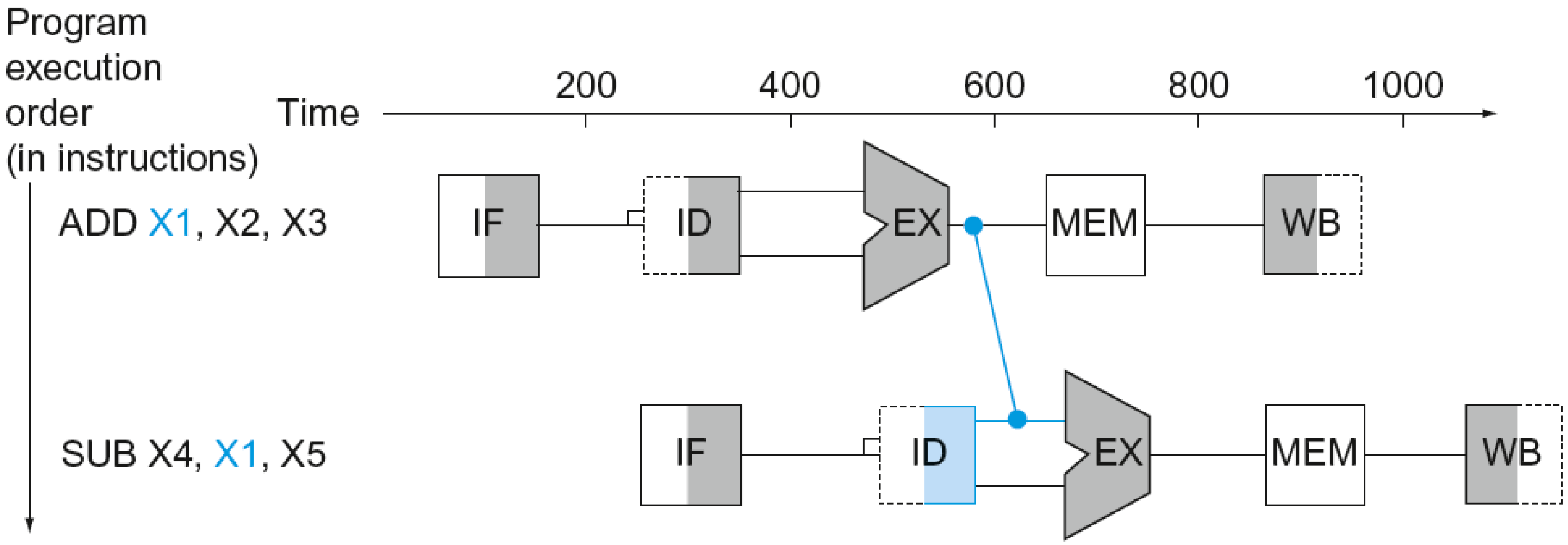
```
ADD x1, x2, x3
SUB x4, x1, x5
```

Instead of waiting to a write back to occur, we take the required data directly from the execute stage.



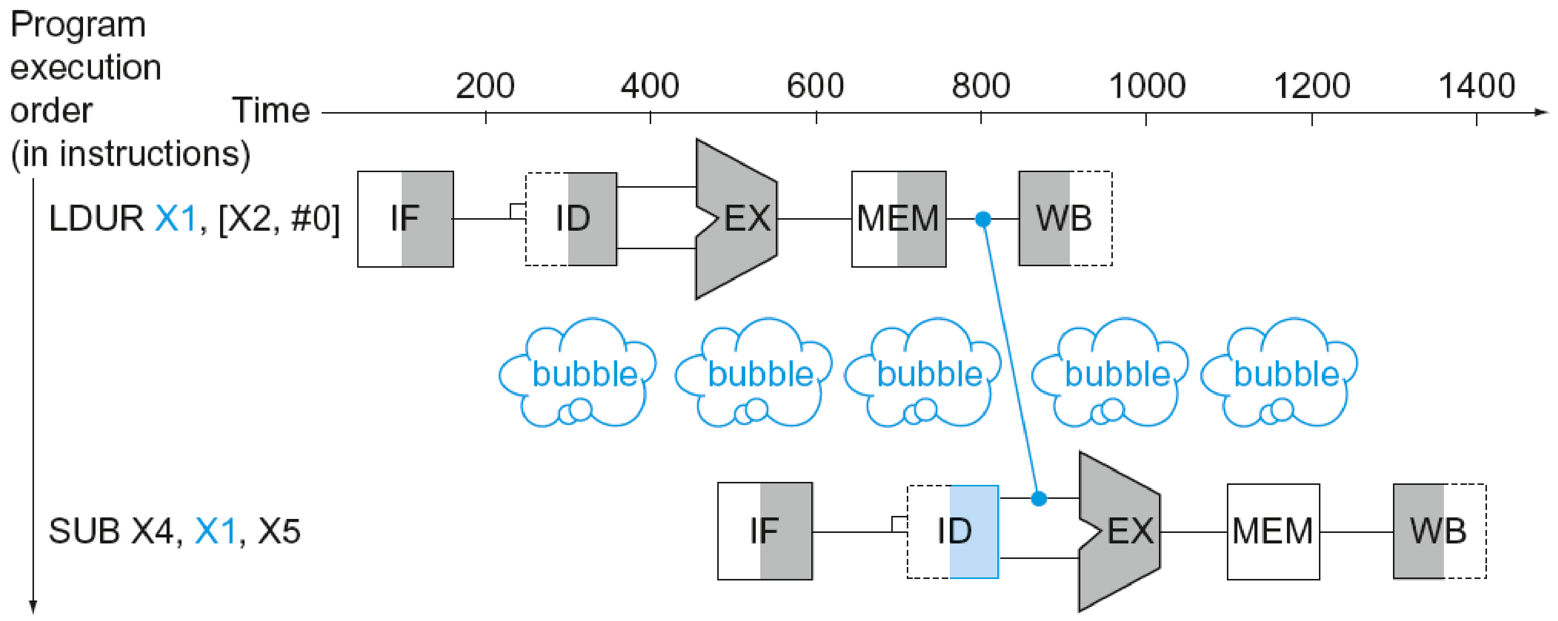
Data hazards - forwarding (aka bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



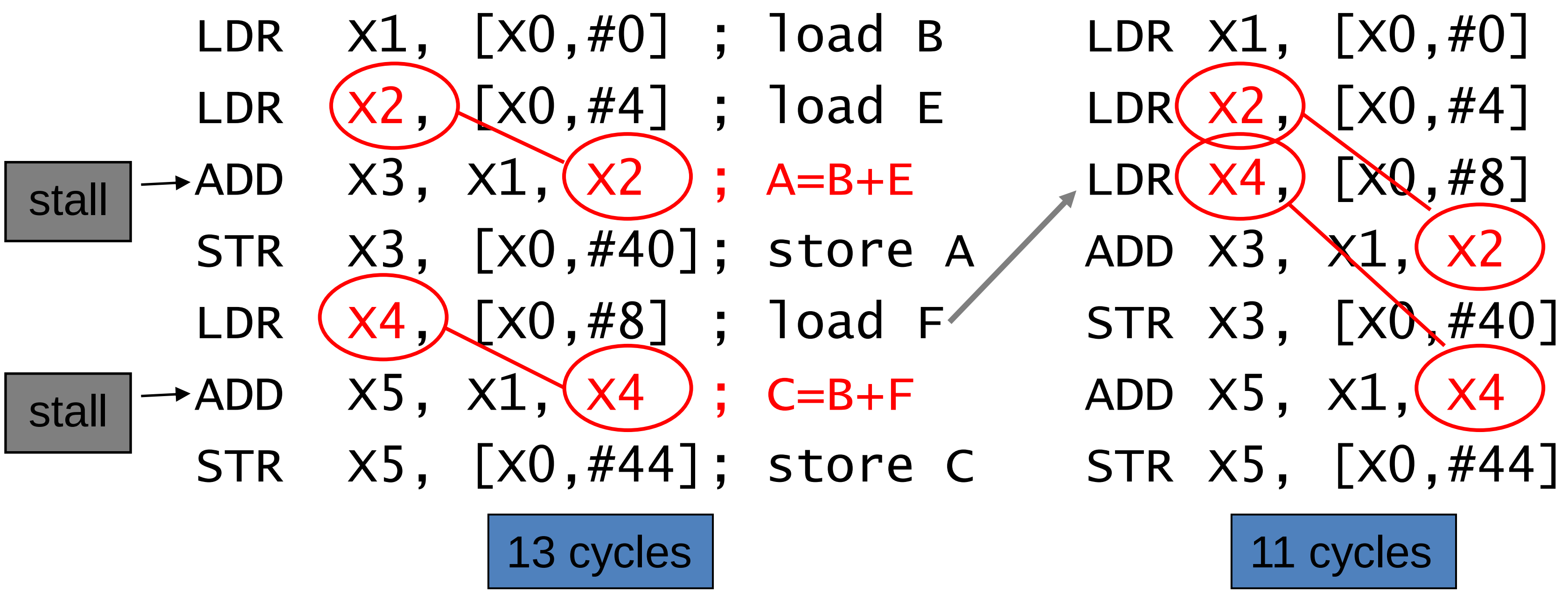
Data hazards

- Can't always avoid stalls by forwarding
    - If value not computed when needed
    - Can't forward backward in time!



Code scheduling to avoid stalls

- Reorder code to avoid use of load result in the next instruction
- `A = B + E;`
- `C = B + F;`
- `B = Mem[0]; E = Mem[4]; F = Mem[8]`



```
LDR   X1, [X0,#0] ; load B
LDR   X2, [X0,#4] ; load E
ADD   X3, X1, X2  ; A=B+E
STR   X3, [X0,#40]; store A
LDR   X4, [X0,#8] ; load F
ADD   X5, X1, X4  ; C=B+F
STR   X5, [X0,#44]; store C
```

```
LDR X1, [X0,#0]
LDR X2, [X0,#4]
LDR X4, [X0,#8]
ADD X3, X1, X2
STR X3, [X0,#40]
ADD X5, X1, X4
STR X5, [X0,#44]
```

Control hazards

- Branch determines flow of control.
  - Fetching next instruction depends on branch outcome.
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch.
  - Need to compare registers and compute target early in the pipeline.
  - Add hardware to do it in ID stage

Control hazards

-
  ```
  if(a==0)
      b = c + d
  else
      b = c - d
  ```
- Example 1. Assume a != 0
  ```
      CMP a, #0x00 ; Compare a==0
      BEQ if_label ; Branch if a==0
  else_label
      SUB b, c, d  ; else = Branch not taken
                   ; b = c - d
      :
  if_label         ; if = Branch taken
      ADD b, c, d  ; b = c + d
  ```

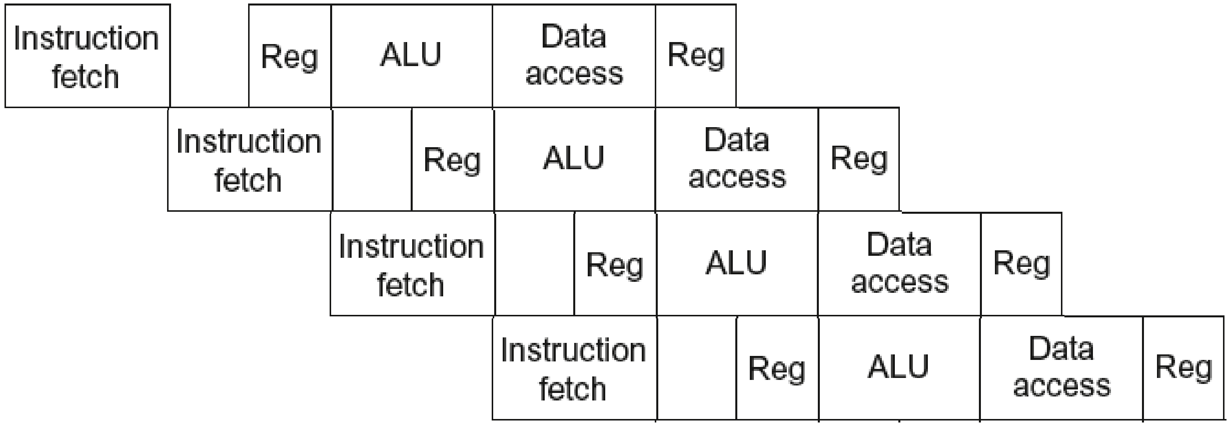
Control hazards

-
  ```
  if(a==0)
      b = c + d
  else
      b = c - d
  ```
- Example 2. Assume a == 0
  ```
      CMP a, #0x00 ; Compare a==0
      BEQ if_label ; Branch if a==0
  else_label
      SUB b, c, d  ; else = Branch not taken
                   ; b = c - d
      :
  if_label         ; if = Branch taken
      ADD b, c, d  ; b = c + d
  ```

Control hazards

- Example 1. a != 0: Branch not taken
  - Order of instructions is not modified
  - Instructions already in pipeline might continue to execute.

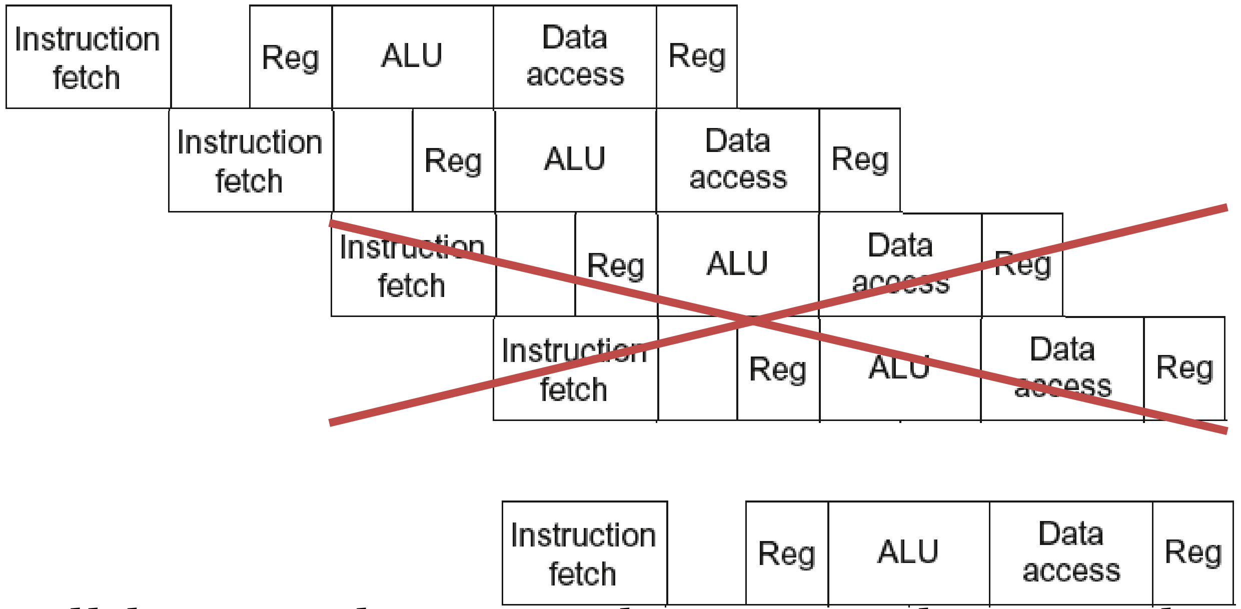
```
CMP a, #0x00
BEQ if_label
SUB b, c, d
AND e, f, g
```



Control hazards

- Example 2. a == 0: Branch taken
  - Order of instructions is modified
  - Instructions already in pipeline will have to be discarded.

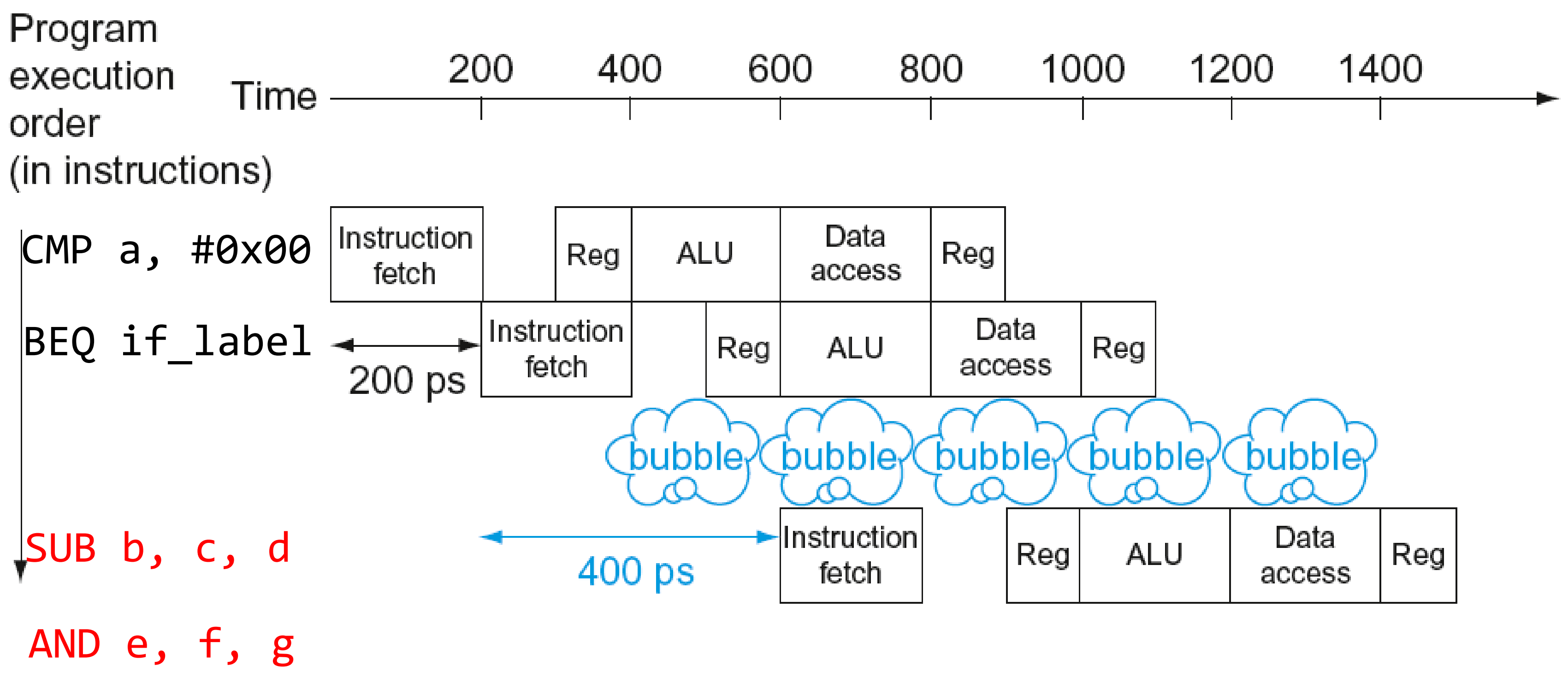
```
CMP a, #0x00

BEQ if_label

SUB b, c, d

AND e, f, g


ADD b, c, d
```



  - Pipeline will have to be started again with a penalty due to wasted instructions

Stall on branch

- Wait until branch outcome determined before fetching next instruction



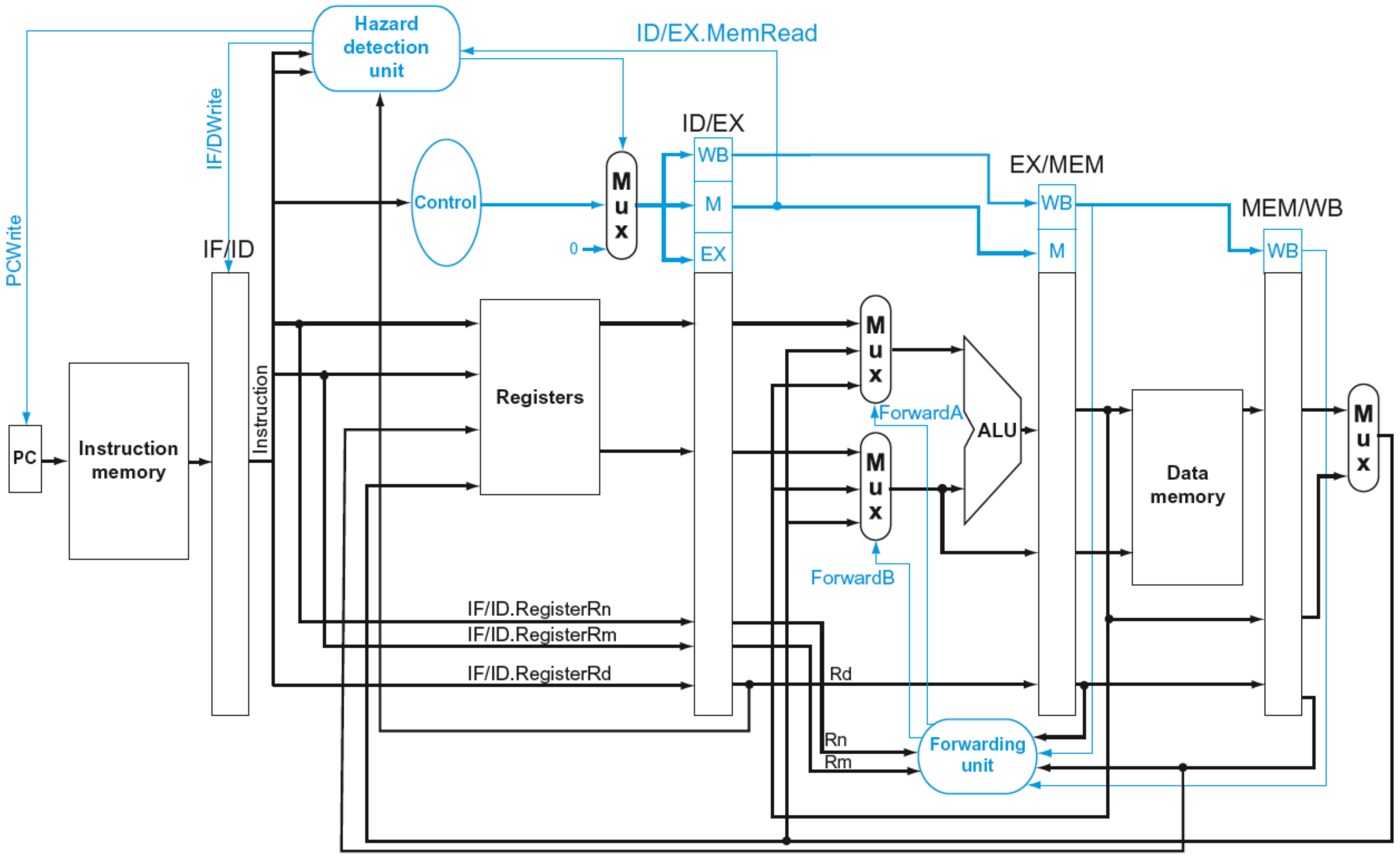
Branch prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In ARM pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

More-realistic branch prediction

- Static branch prediction
    - Based on typical branch behavior
    - Example: loop and if-statement branches
        - Predict backward branches taken
        - Predict forward branches not taken
- Dynamic branch prediction
    - Hardware measures actual branch behavior
        - e.g., record recent history of each branch
    - Assume future behavior will continue the trend
        - When wrong, stall while re-fetching, and update history

Pipeline uA



Hazard detection unit
ID/EX.MemRead
IF/DWrite
PCWrite
Control
0
M u x
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
PC
Instruction memory
Instruction
Registers
M u x
ForwardA
ALU
M u x
ForwardB
Data memory
M u x
IF/ID.RegisterRn
IF/ID.RegisterRm
IF/ID.RegisterRd
Rd
Rn
Rm
Forwarding unit

Pipeline summary

- Pipelining improves performance by increasing instruction throughput.
  - Executes multiple instructions in parallel.
  - Each instruction has the same (or worse) latency.
- Subject to hazards.
  - Structure, data, control.
- Instruction set design affects complexity of pipeline implementation.