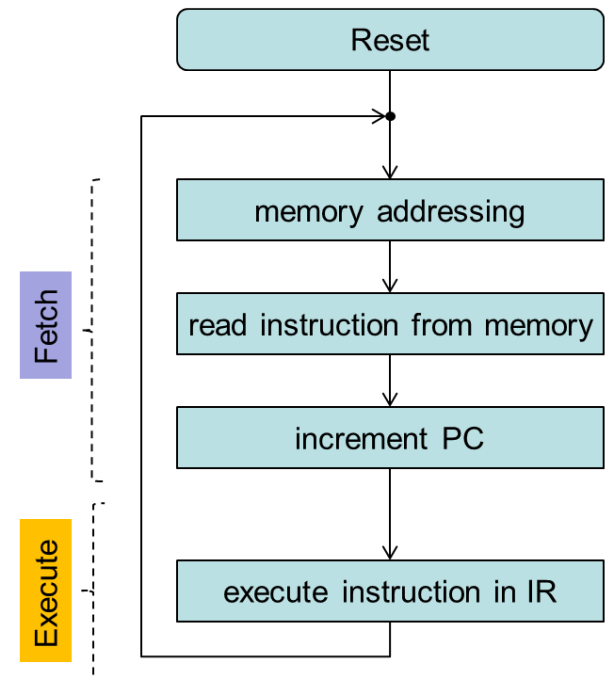


Pipelining

Computer Engineering 1

**CT Team: A. Gieriet, J. Gruber, R. Gübeli, M. Meli, M. Rosenthal,
A. Rüst, J. Scheier, M. Thaler**

- **Instruction executed as a sequence**
 - Fetch / decode / execute / write back, etc.
- **Parallelization of stages**
- **Increase processing speed**



*Picture from lecture
“Cortex-M Architecture”*

Agenda

- Motivation
- Repetition
Instruction execution
- Pipelining principle
- Latency
- Pipeline hazards
- Solutions for pipeline hazards

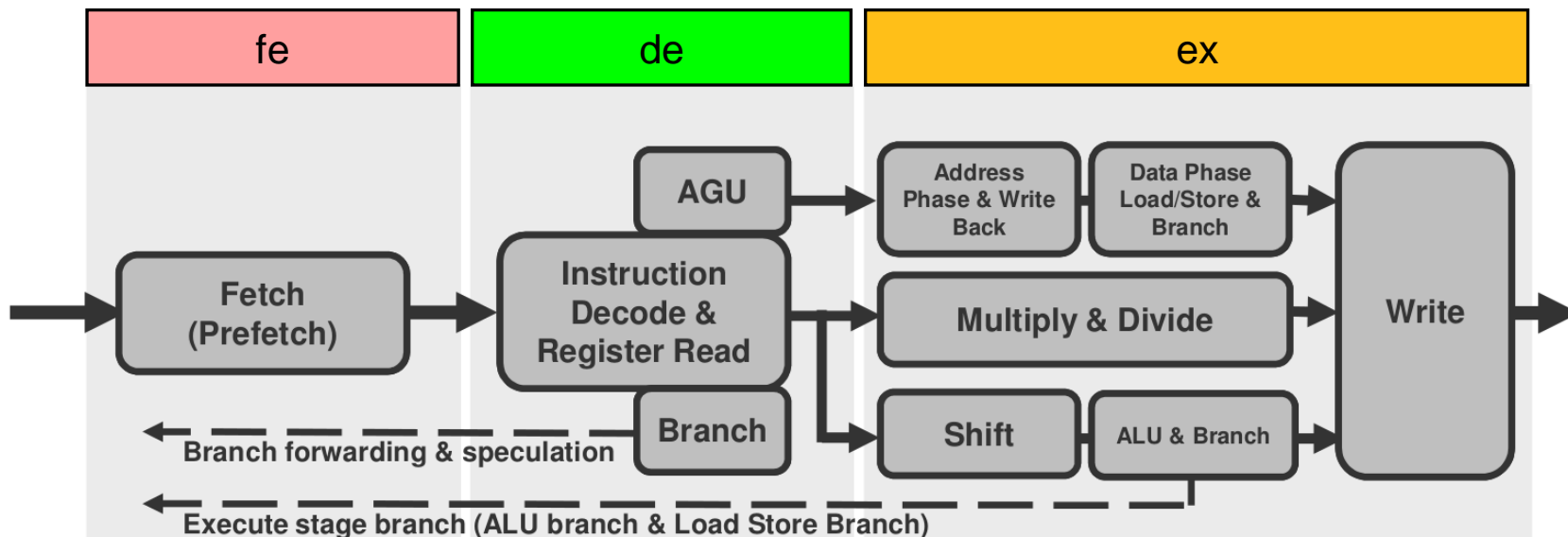


- **At the end of this lesson you will be able**
 - to describe the idea of pipelining
 - to explain the pipeline stages of Cortex-M3
 - to calculate processing performance improvement through pipelining
 - to enumerate and describe pipeline hazards
 - to explain reasons for pipeline hazards
 - to solve pipeline hazards

Instruction Execution

■ Instruction sequence of ARM Cortex-M3

- **fe: fetch** Read instruction
- **de: decode** Decode instruction, read register or memory
- **ex: execute** Execute instruction, write back result



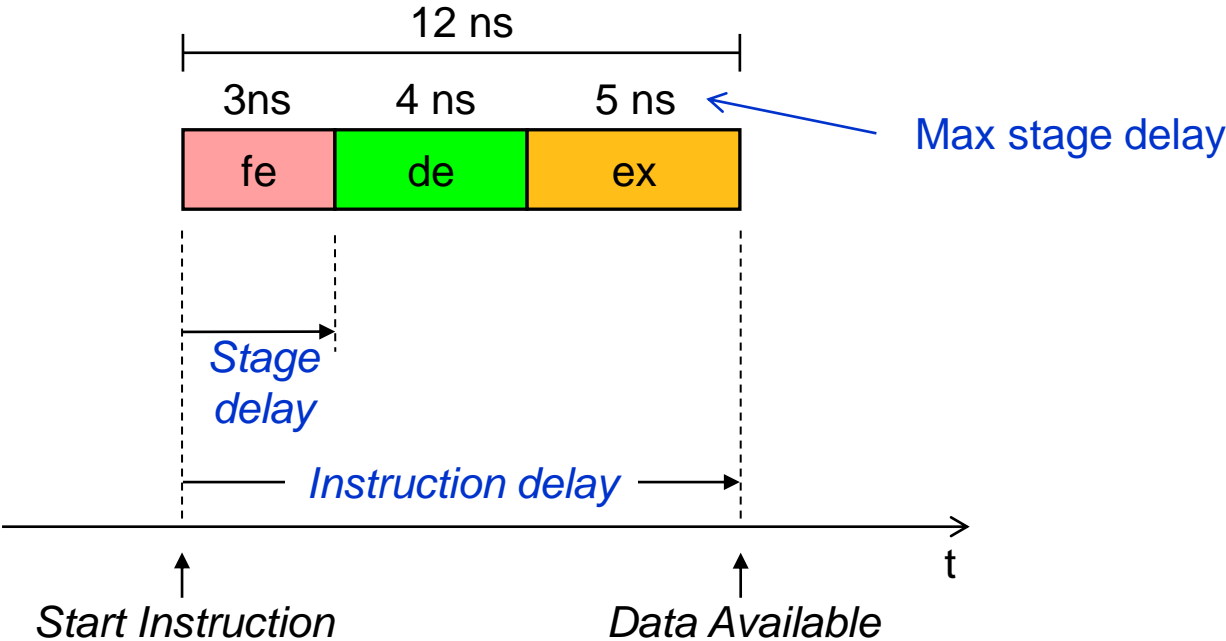
Source: ARM

Instruction Execution

■ Timings and definitions

- Example: Cortex-M3

- **fe: fetch** Read instruction 3 ns¹
- **de: decode** Decode instruction, read register or memory 4 ns¹
- **ex: execute** Execute instruction, write back result 5 ns¹



¹timings estimated

Pipelining Principle

■ Example: Conventional manufacturing

One worker can build a car in 20 days

- Overall production time for one car: 20 days
- Always after 20 days, a single car is finished

■ How to increase output?

Increase number of workers

- 20 workers finish 20 cars in 20 days
- Always after 20 days, 20 cars are finished

■ What are the disadvantages of this procedure?

- Every worker must know all operations
- Every worker (every station) needs the same tools at the same time

Pipelining Principle

■ Example: Conventional manufacturing

One worker can build a car in 20 days

- Overall production time for one car: 20 days
- Always after 20 days, a single car is finished



■ New procedure!

Assembly line

- 20 workers at 20 individual stations
- First car finished after 20 days,
Overall production time for one car: 20 days
- Always after one day, one car is finished
- Requirements
 - Same time needed at all stations
 - No dependencies between stations (e.g. tools)

Pipelining Principle

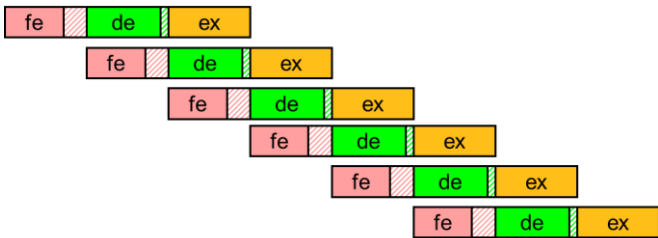
■ Assembly line

- Same working time at all stations
- No resource dependencies between different stages
- Stopping the process results in refilling the assembly



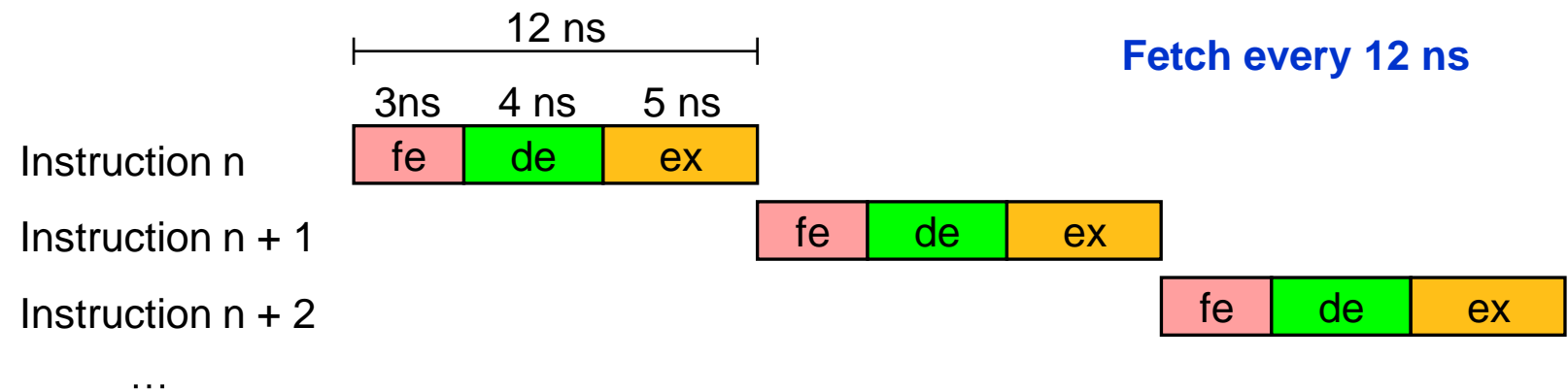
■ Instruction pipeline

- • Even process time for all stages
- • No dependencies between different instruction stages
- • Branch instructions may clear the pipeline

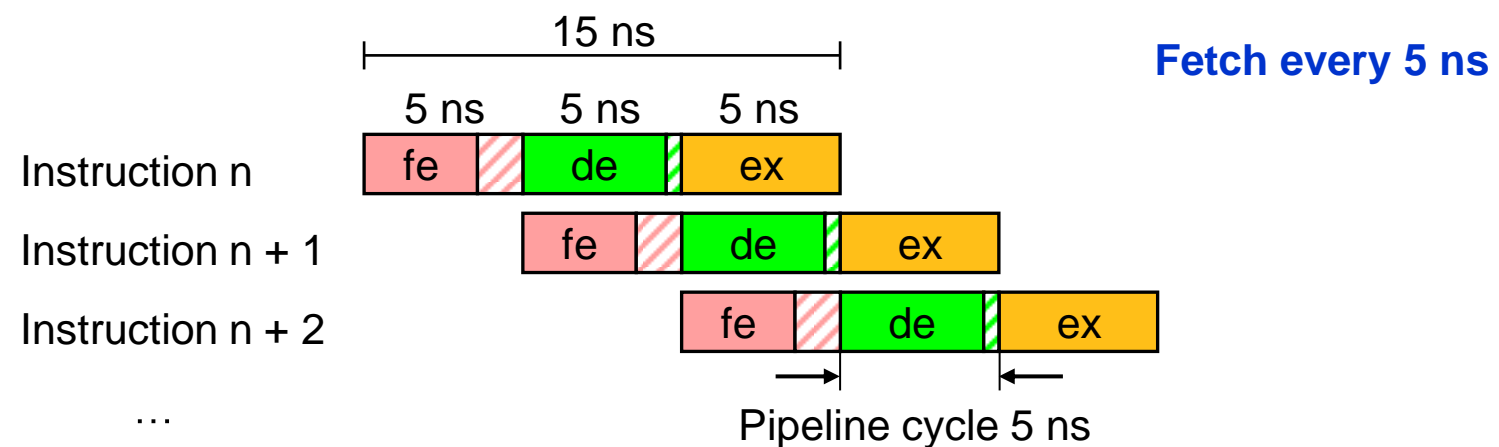


Pipelining Principle

■ Sequential execution



■ Pipelined execution



■ Advantages

- All stages are set to the same execution time
- Massive performance gain
- Simpler hardware at each stage allows for a higher clock rate

■ General

- Number of execution stages is design decision
- Typically 2 – 12 Stages

■ Instructions per second

- Without pipelining

$$\text{Instructions per second} = \frac{1}{\text{Instruction delay}}$$

- With pipelining
 - Pipeline needs to be filled first
 - After filling, instructions are executed after every stage

$$\text{Instructions per second} = \frac{1}{\text{Max stage delay}}$$

■ Example Cortex-M3

- Without pipelining: $1/12\text{ns} = 83,3$ million instructions per second
- With pipelining: $1/5\text{ns} = 200$ million instructions per second

■ Optimal Pipelining

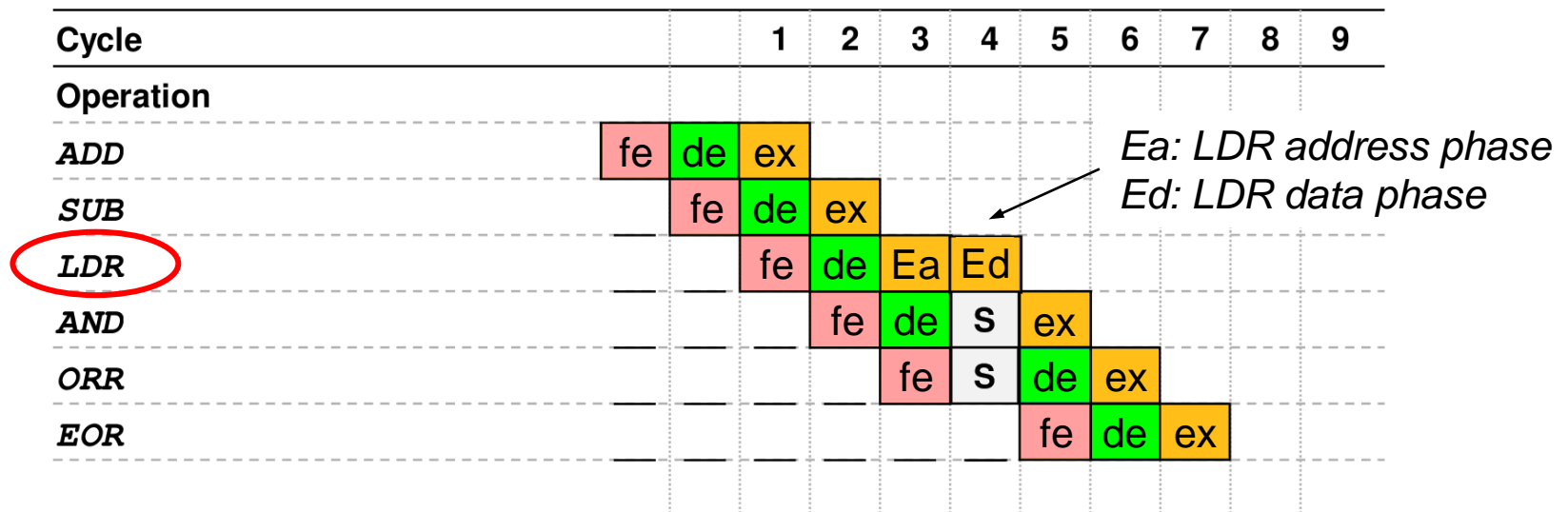
- All operations here are on registers (single cycle execution)
- In this example it takes 6 clock cycles to execute 6 instructions
- Clock cycles per instruction (CPI) = 1

| Cycle | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------------|----|----|----|----|----|----|----|----|---|---|
| Operation | | | | | | | | | | |
| <i>ADD</i> | fe | de | ex | | | | | | | |
| <i>SUB</i> | | fe | de | ex | | | | | | |
| <i>ORR</i> | | | fe | de | ex | | | | | |
| <i>AND</i> | | | | fe | de | ex | | | | |
| <i>ORR</i> | | | | | fe | de | ex | | | |
| <i>EOR</i> | | | | | | fe | de | ex | | |

Special Pipeline Situations

■ LDR Pipeline Example

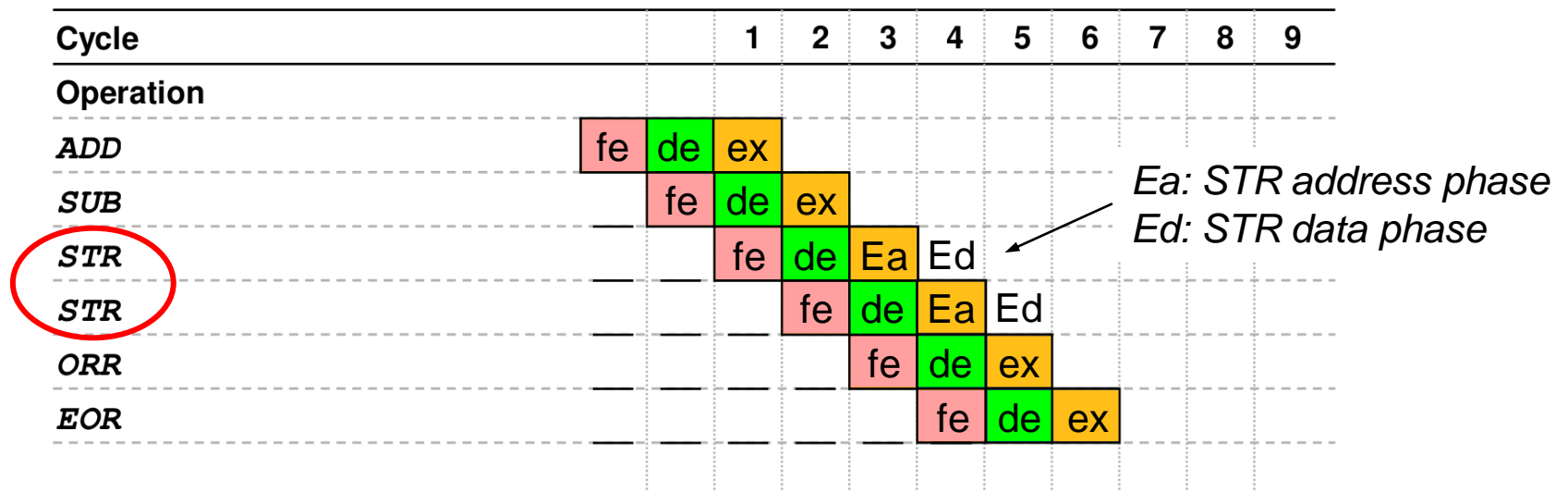
- In this example it takes 7 clock cycles to execute 6 instructions
- Read cycle must complete on the bus before LDR instruction can complete (reason only one writeback port in the register file)
- Clock cycles per instruction (CPI) = 1.2



Special Pipeline Situations

■ STR-STR Pipeline Example

- Store buffer allows STR instruction to finish before store cycle completes on the bus
- Back-to-back STR instructions pipeline on the AHB-Lite bus
 - Also works for LDR-STR-LDR-STR



■ Control Hazards

- Occur with branches. The processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline at “fetch” stage

■ Data Hazards

- Occur when instructions use data, that is modified in different stages of a pipeline
 - read after write (RAW), a true dependency
 - write after read (WAR), an anti-dependency
 - write after write (WAW), an output dependency

■ Structural Hazards

- Occur when a part of the processor's hardware is needed by two or more instructions at the same time

■ Control Hazards

- Occur with branches. The processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline at “fetch” stage

■ Data Hazards

- Occur when instructions use data, that is modified in different stages of a pipeline
 - read after write (RAW), a true dependency
 - write after read (WAR), an anti-dependency
 - write after write (WAW), an output dependency


■ Structural Hazards

- Occur when a part of the processor's hardware is needed by two or more instructions at the same time

■ Reason

- Instructions are fetched in stage 1
- Branch and jump decisions occur in stage 3
- Next instruction not fetched until 2 cycles after branch/jump
- Branch and jump instructions are a substantial part of a program (this is a qualitative statement)

| Type | Distribution |
|----------------|--------------|
| Data transfers | 43% |
| Control flow | 23% |
| Arithmetic | 15% |
| Compare | 13% |
| Logic | 5% |
| Others | 1% |

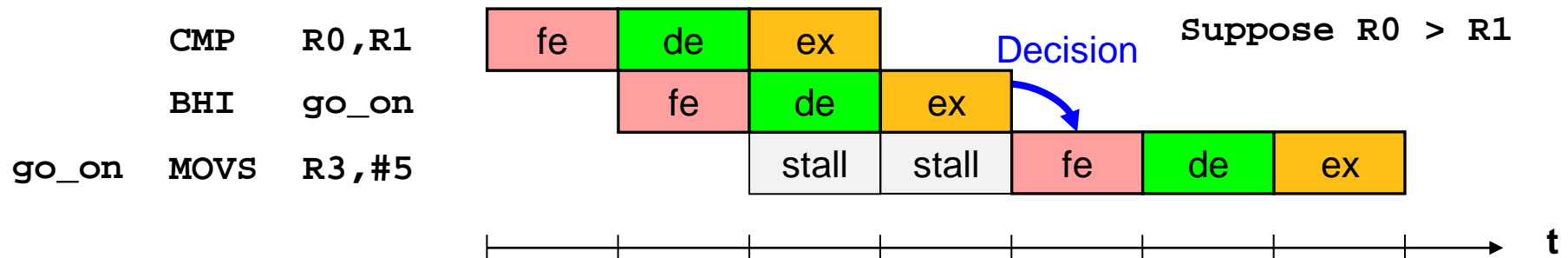
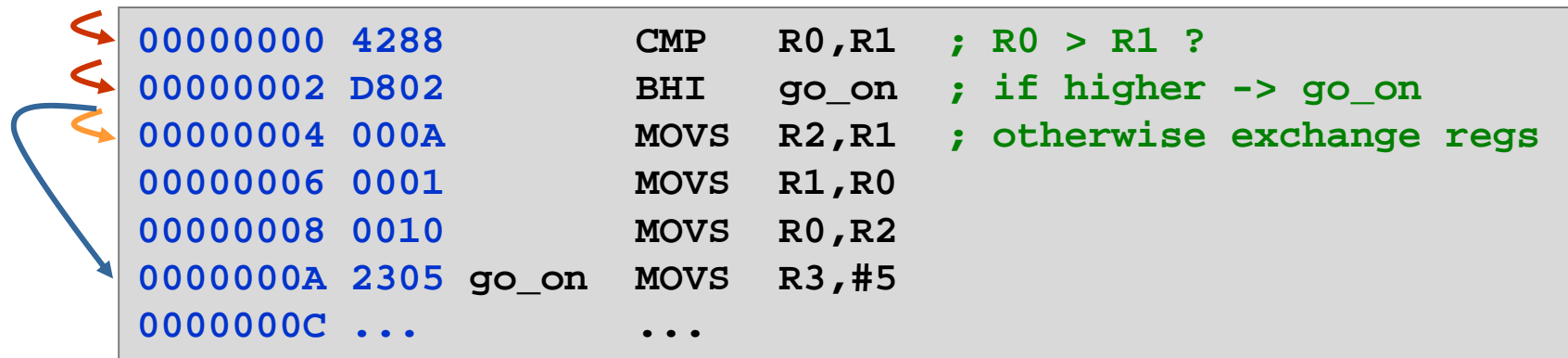


→ Stall and delay slots

Control Hazards

■ Example

- 3 cycles to complete branch
- Worst case scenario – conditional branch taken



■ Common solution: 2-Bit branch prediction scheme¹

- Idea: use two bits to remember if the branch was taken or not the last time.
- Only change prediction on two successive mispredictions

ST: Strongly Taken

WT: Weakly Taken

WN: Weakly Not Taken

SN: Strongly Not Taken

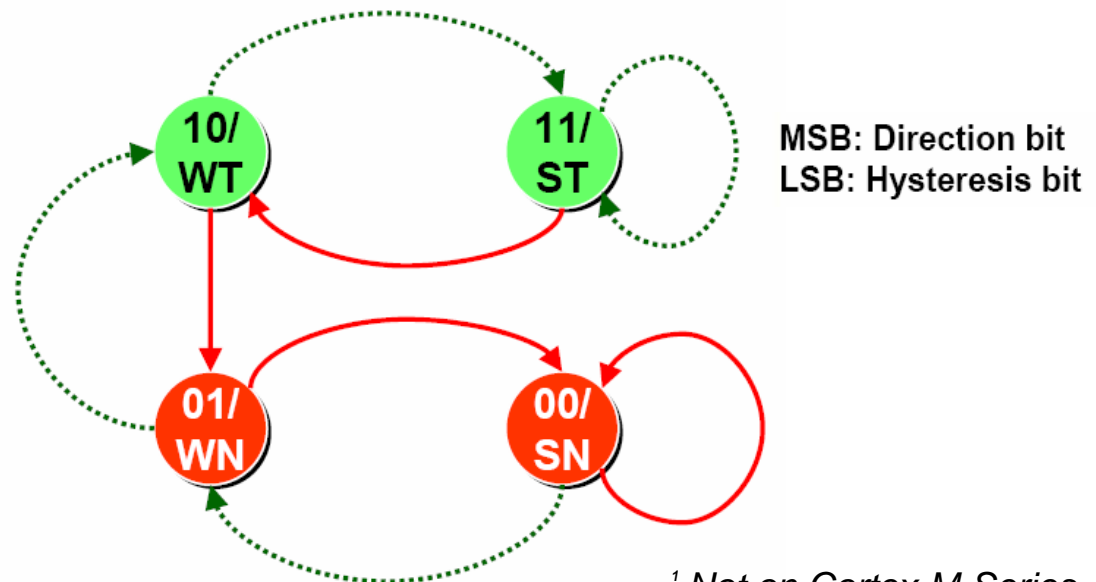
.....➡ Taken

➡ Not Taken

● Predict Not taken

● Predict taken

2-bit branch prediction
State diagram



¹ Not on Cortex-M Series

■ Reduce control hazards

- Loop fusion reduces control hazards
- Simple example for illustration

```
/* Two loops → double number of control hazards*/  
for (i = 0; i < N; i++) {  
    a[i] = 1/b[i] * c[i];  
}  
for (i = 0; i < N; i++) {  
    d[i] = a[i] + c[i];  
}  
  
/* Better performance: Fusion into one loop */  
for (i = 0; i < N; i++) {  
    a[i] = 1/b[i] * c[i];  
    d[i] = a[i] + c[i];  
}
```

2*N Conditional
Branches

N Conditional
Branches

■ Control Hazards

- Occur with branches. The processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline at “fetch” stage

■ Data Hazards

- Occur when instructions use data, that is modified in different stages of a pipeline
 - read after write (RAW), a true dependency
 - write after read (WAR), an anti-dependency
 - write after write (WAW), an output dependency

■ Structural Hazards

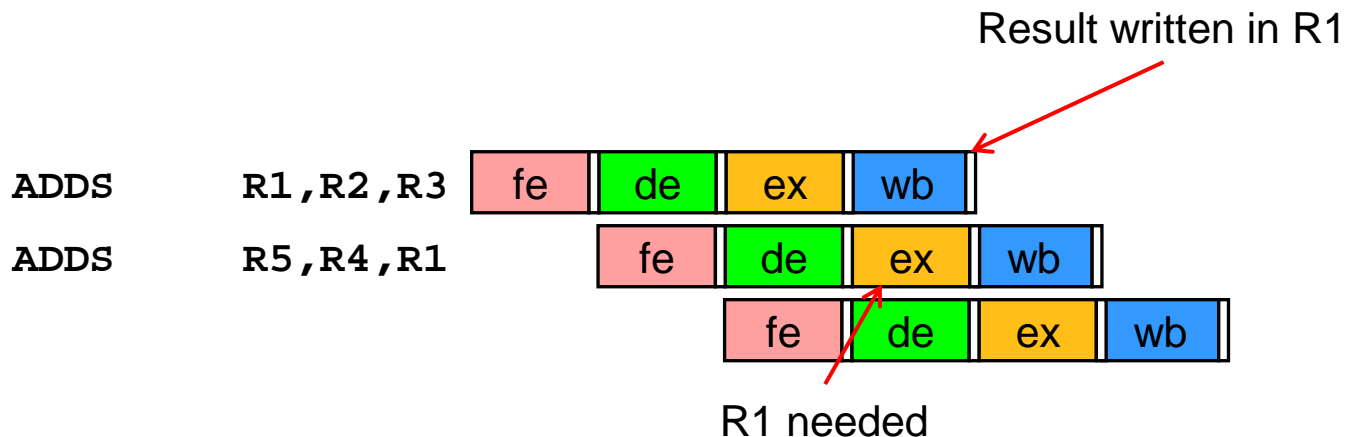
- Occur when a part of the processor's hardware is needed by two or more instructions at the same time

■ Example: read after write, RAW

- Not occurring on 3-stages pipeline → Not on Cortex M3
- On 4-stages pipeline: Fetch, Decode, Execute, Write Back

- Example:

| | | |
|------|----------|----------------|
| ADDS | R1,R2,R3 | ; R2 + R3 → R1 |
| ADDS | R5,R4,R1 | ; R4 + R1 → R5 |



■ Solution: Forwarding

- Result can be forwarded to the ex phase of the next instruction

ADDS **R1**,R2,R3



ADDS R5,R4,**R1**



AND R4,R4,**R1**



SUB R5,R5,R1



■ Control Hazards

- Occur with branches. The processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline at “fetch” stage

■ Data Hazards

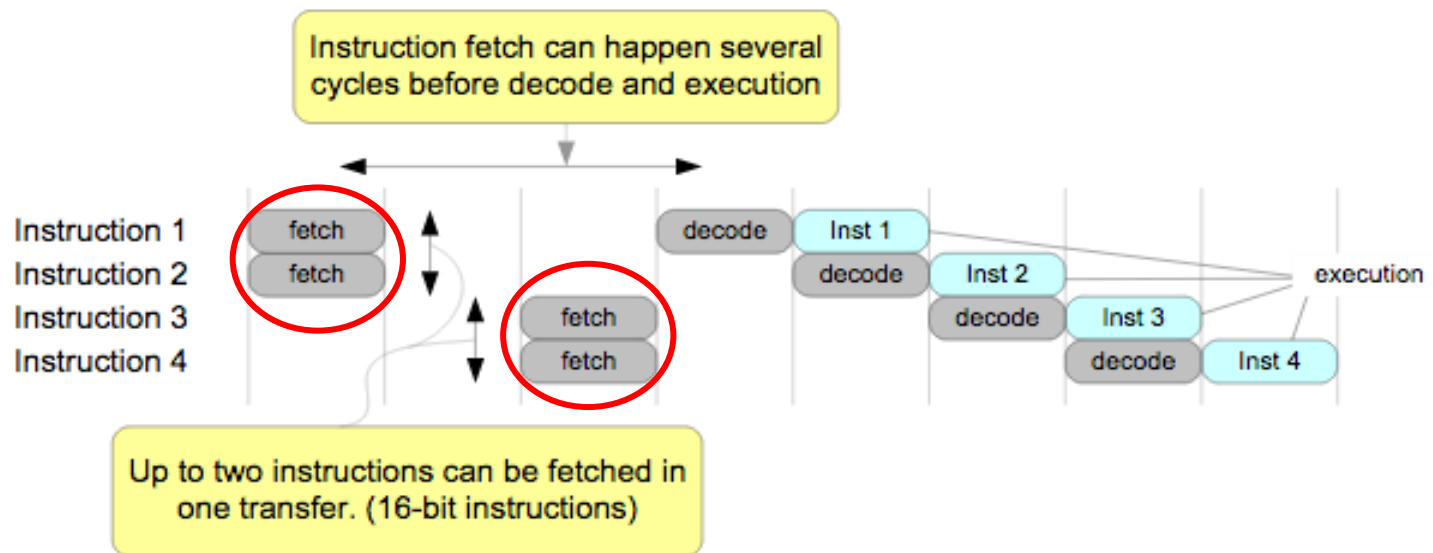
- Occur when instructions use data, that is modified in different stages of a pipeline
 - read after write (RAW), a true dependency
 - write after read (WAR), an anti-dependency
 - write after write (WAW), an output dependency

■ Structural Hazards

- Occur when a part of the processor's hardware is needed by two or more instructions at the same time

■ Avoid structural hazards

- Fetch multiple instructions at once
 - Fetch instructions earlier than needed
- Data bus is free for other data in next cycle



Example: Cortex-M3

■ Pipeline

- Instruction execution
- Principle of pipelining
- Latency
- Hazards
- Solutions for pipeline hazards

