

Exercises for Architectures of Supercomputers

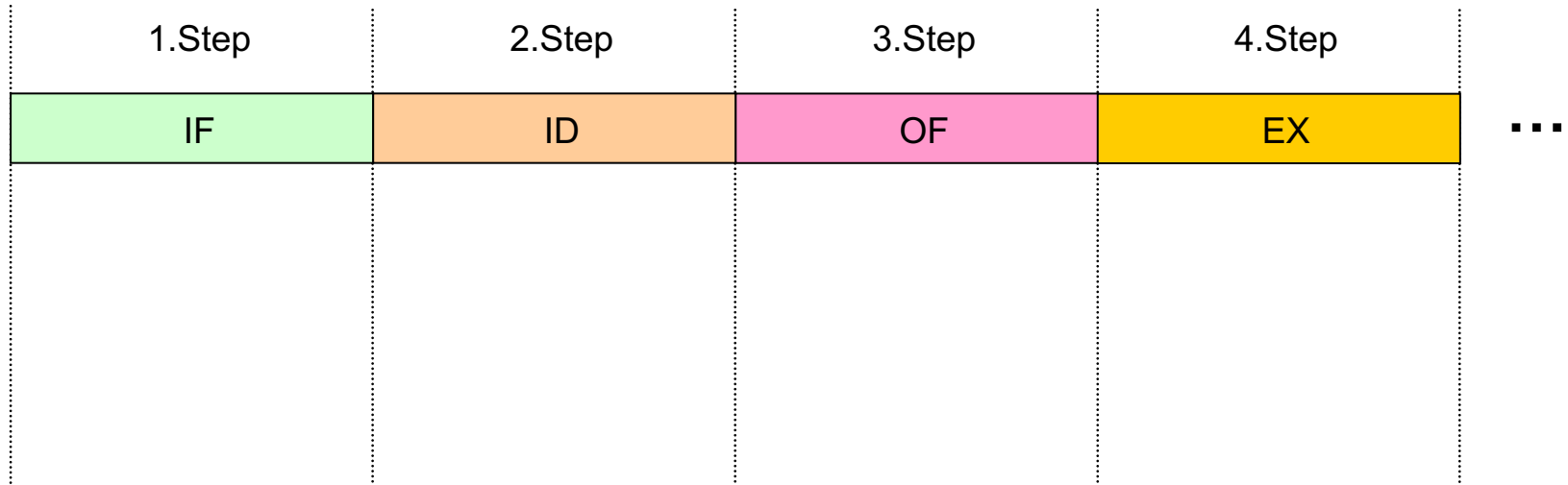
3rd Exercise, 13./14.11.2019



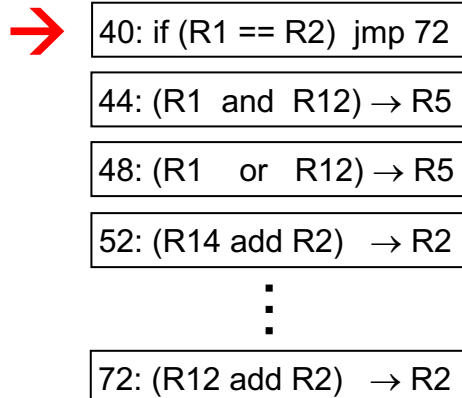
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

- Timestep n



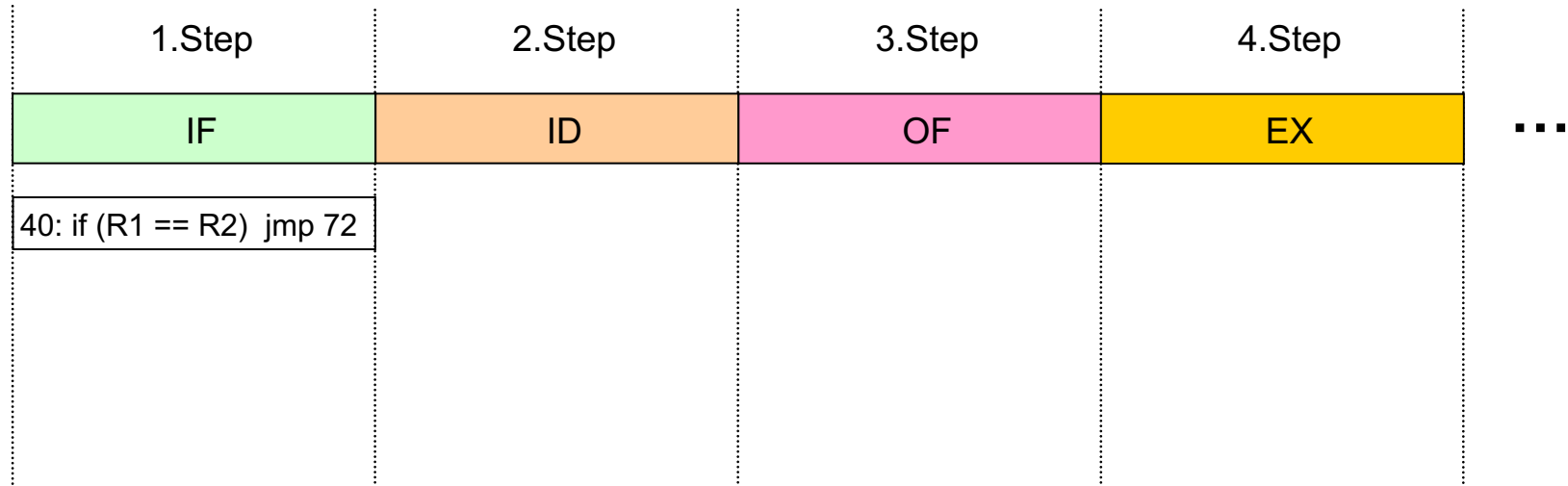
Instruction stream



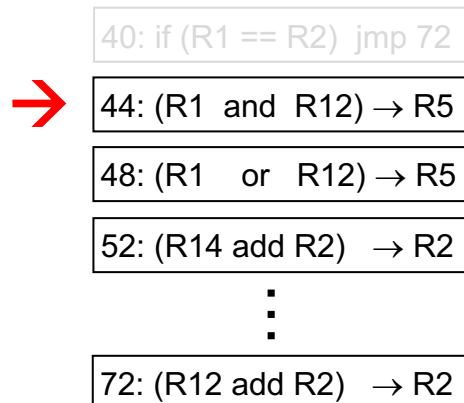
Register contents

R1:	0	R7:	9
R2:	0	R8:	10
R3:	2	R9:	23
R4:	1	R10:	34
R5:	23	R11:	15
R6:	4	R12:	8

- Timestep $n+1$: Fetching the compare-and-branch instruction



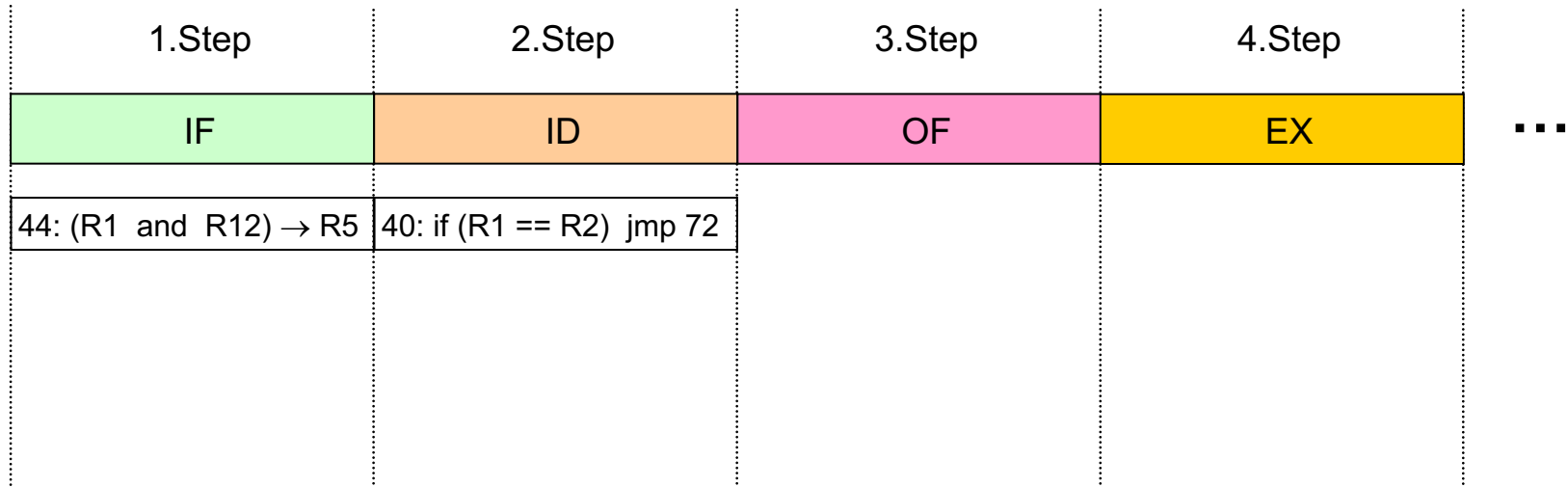
Instruction stream



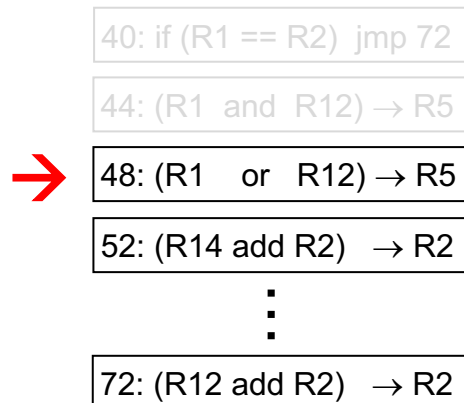
Register contents

R1:	0	R7:	9
R2:	0	R8:	10
R3:	2	R9:	23
R4:	1	R10:	34
R5:	23	R11:	15
R6:	4	R12:	8

- Timestep $n+2$: Decoding the compare-and-branch instruction



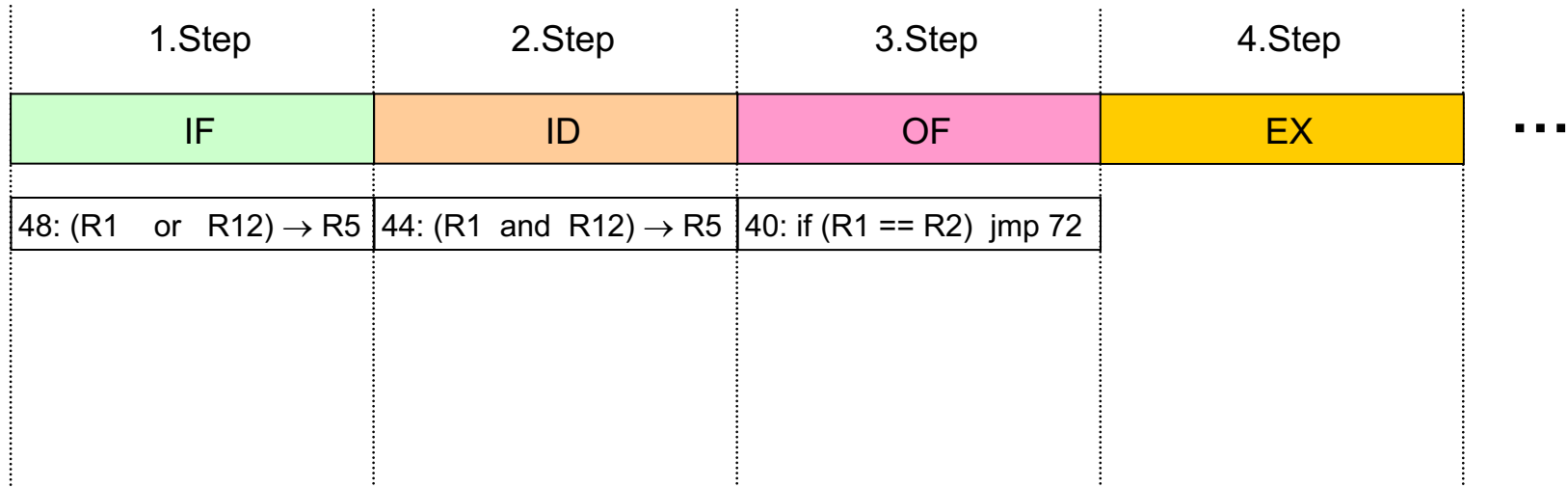
Instruction stream



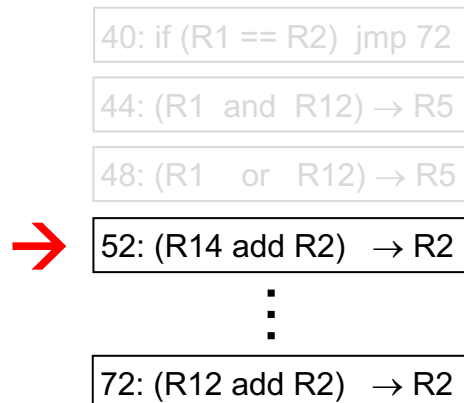
Register contents

R1:	0	R7:	9
R2:	0	R8:	10
R3:	2	R9:	23
R4:	1	R10:	34
R5:	23	R11:	15
R6:	4	R12:	8

- Timestep $n+3$: Fetching the instruction's operands



Instruction stream

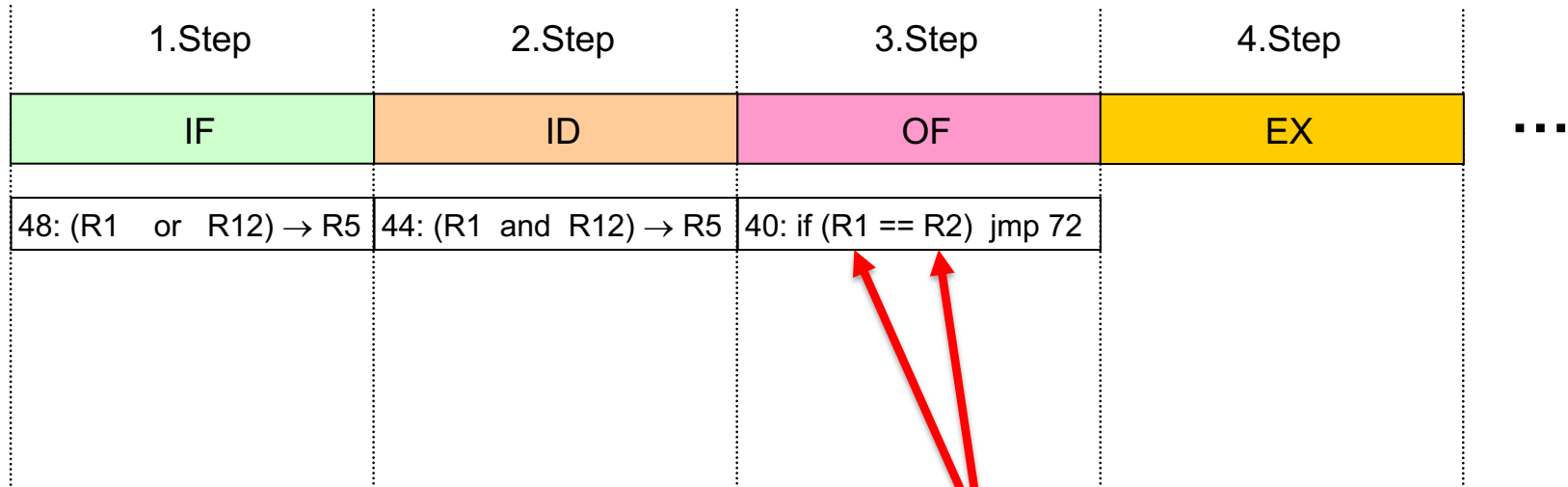


Register contents

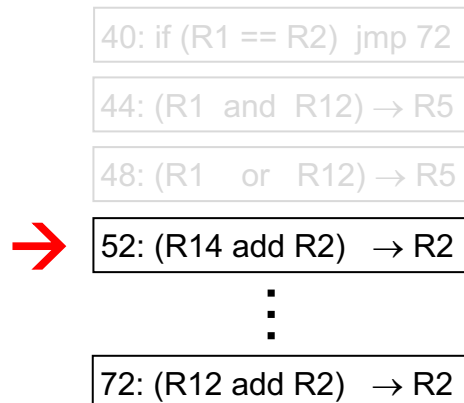
R1:	0	R7:	9
R2:	0	R8:	10
R3:	2	R9:	23
R4:	1	R10:	34
R5:	23	R11:	15
R6:	4	R12:	8

Pipelining and control hazards

- Timestep $n+3$: Fetching the instruction's operands



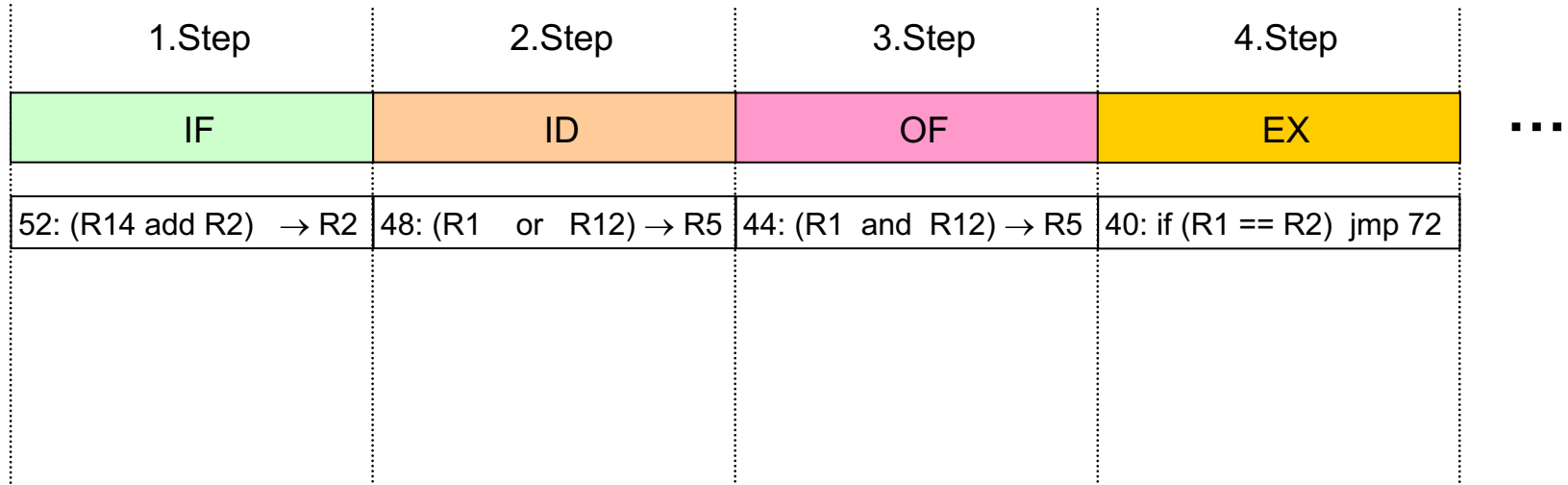
Instruction stream



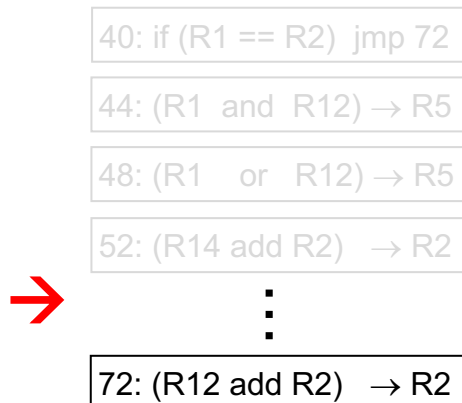
Register contents

R1:	0	R7:	9
R2:	0	R8:	10
R3:	2	R9:	23
R4:	1	R10:	34
R5:	23	R11:	15
R6:	4	R12:	8

- Timestep $n+4$



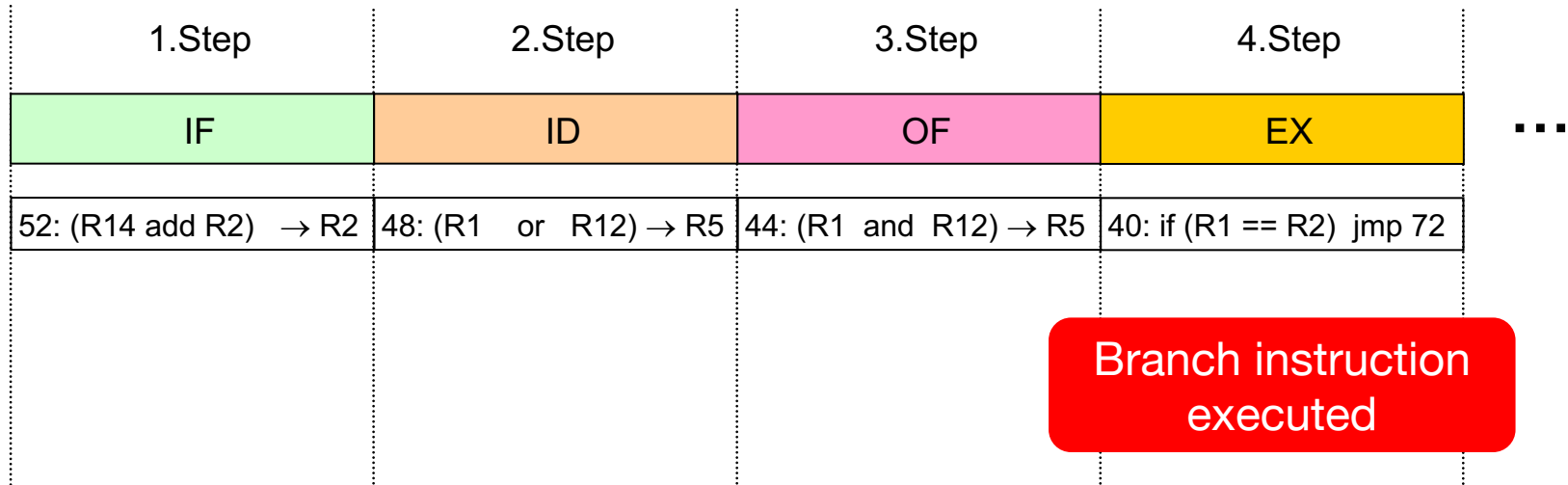
Instruction stream



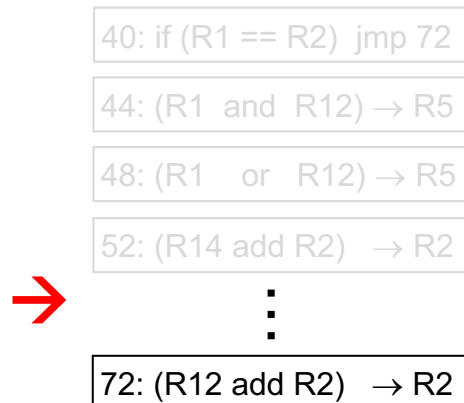
Register contents

R1:	0	R7:	9
R2:	0	R8:	10
R3:	2	R9:	23
R4:	1	R10:	34
R5:	23	R11:	15
R6:	4	R12:	8

- Timestep $n+4$



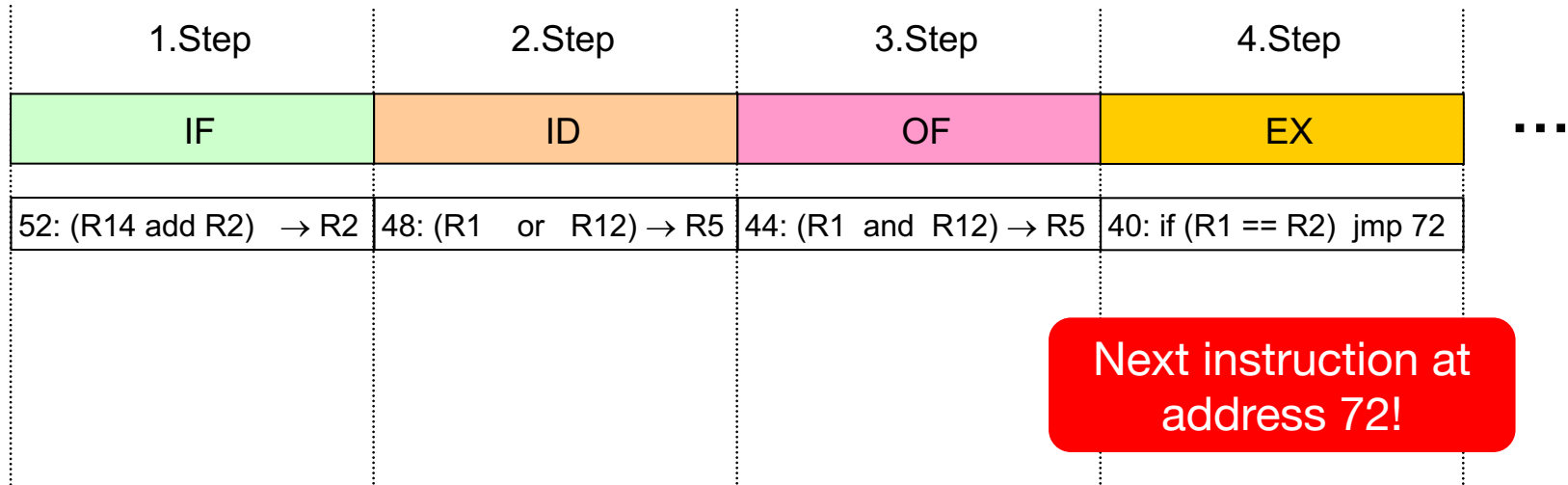
Instruction stream



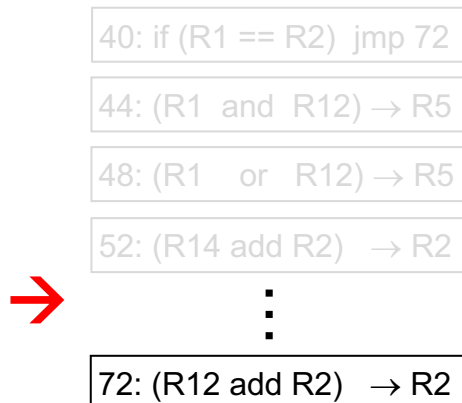
Register contents

R1:	0	R7:	9
R2:	0	R8:	10
R3:	2	R9:	23
R4:	1	R10:	34
R5:	23	R11:	15
R6:	4	R12:	8

- Timestep $n+4$



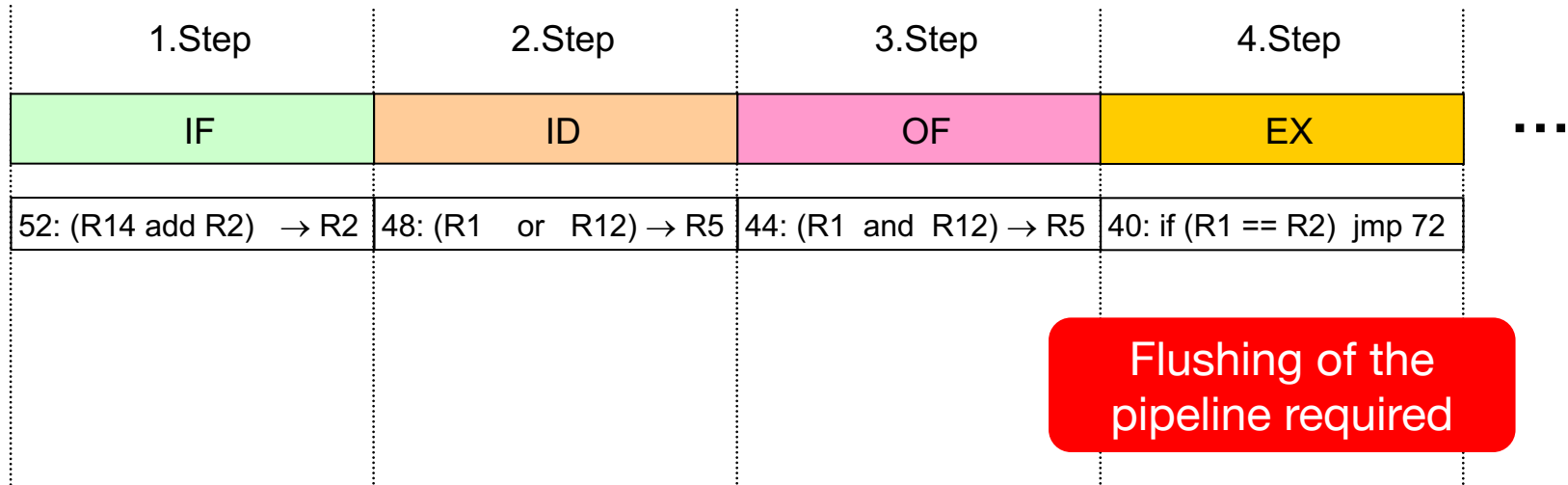
Instruction stream



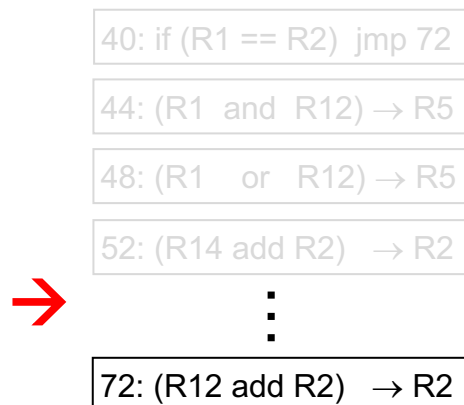
Register contents

R1:	0	R7:	9
R2:	0	R8:	10
R3:	2	R9:	23
R4:	1	R10:	34
R5:	23	R11:	15
R6:	4	R12:	8

- Timestep $n+4$



Instruction stream

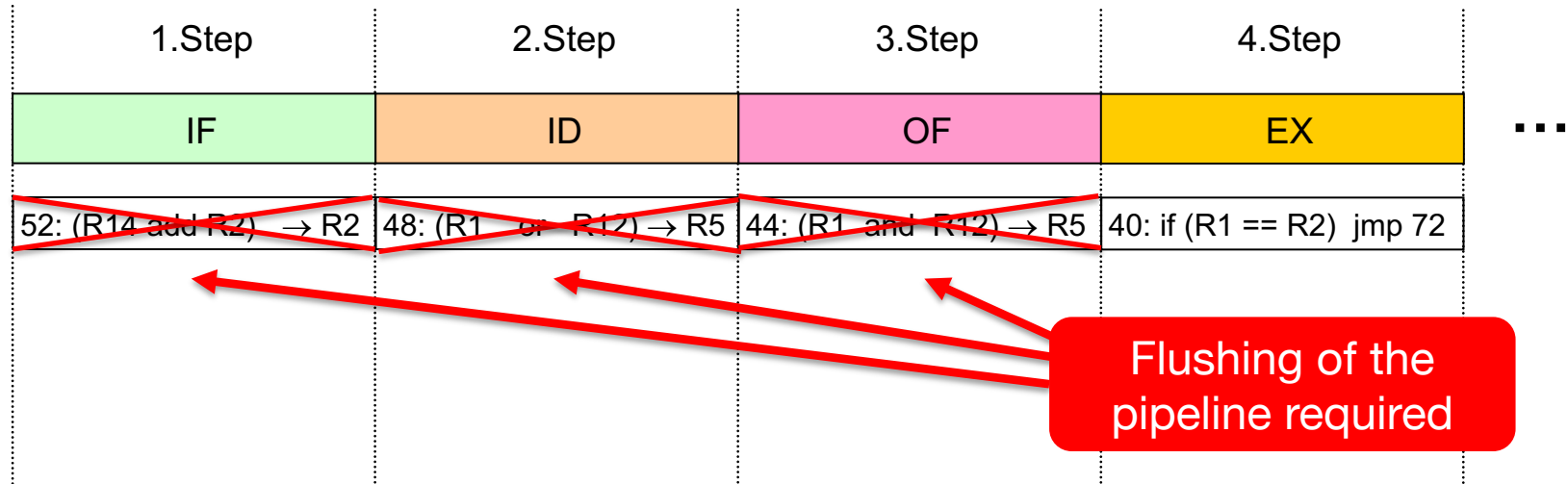


Register contents

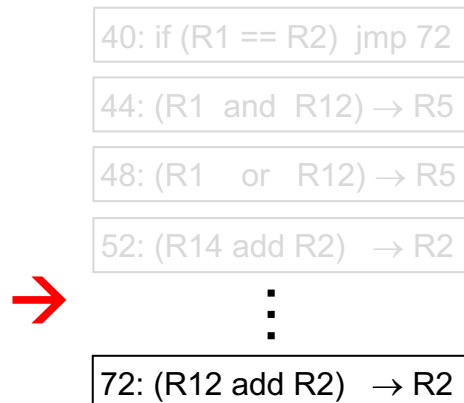
R1:	0	R7:	9
R2:	0	R8:	10
R3:	2	R9:	23
R4:	1	R10:	34
R5:	23	R11:	15
R6:	4	R12:	8

Pipelining and control hazards

- Timestep $n+4$



Instruction stream



Register contents

R1:	0	R7:	9
R2:	0	R8:	10
R3:	2	R9:	23
R4:	1	R10:	34
R5:	23	R11:	15
R6:	4	R12:	8

Exercise 3: Impact of control hazards

- How can the impact of control-hazard induced pipeline flush be quantified?

Exercise 3: Impact of control hazards

- How can the impact of control-hazard induced pipeline flush be quantified?
 1. Measure performance of some code that **does** contains branch instructions

Exercise 3: Impact of control hazards

- How can the impact of control-hazard induced pipeline flush be quantified?
 1. Measure performance of some code that **does** contains branch instructions
 2. Measure performance **the same some** code that contain **no** branch instructions
- The cost of rewinding the pipeline is the difference in runtime of both versions

Exercise 3: Impact of control hazards

- How can the impact of control-hazard induced pipeline flush be quantified?
 1. Measure performance of some code that **does** contains branch instructions
 2. Measure performance **the same some** code that contain **no** branch instructions
 - The cost of rewinding the pipeline is the difference in runtime of both versions
- Problem: branch-prediction unit(s)
 - Modern processors contain hardware that tries to predict the target of a branch instruction

Exercise 3: Impact of control hazards

- How can the impact of control-hazard induced pipeline flush be quantified?
 1. Measure performance of some code that **does** contains branch instructions
 2. Measure performance **the same some** code that contain **no** branch instructions
 - The cost of rewinding the pipeline is the difference in runtime of both versions
- Problem: branch-prediction unit(s)
 - Modern processors contain hardware that tries to predict the target of a branch instruction
 - In case the prediction was correct → no penalty, because correct instructions are in pipeline

Exercise 3: Impact of control hazards

- How can the impact of control-hazard induced pipeline flush be quantified?
 1. Measure performance of some code that **does** contains branch instructions
 2. Measure performance **the same some** code that contain **no** branch instructions
- The cost of rewinding the pipeline is the difference in runtime of both versions
- Problem: branch-prediction unit(s)
 - Modern processors contain hardware that tries to predict the target of a branch instruction
 - In case the prediction was correct → no penalty, because correct instructions are in pipeline
 - In case the prediction was wrong → penalty

Exercise 3: Impact of control hazards

- How can the impact of control-hazard induced pipeline flush be quantified?
 1. Measure performance of some code that **does** contains branch instructions
 2. Measure performance **the same some** code that contain **no** branch instructions
 - The cost of rewinding the pipeline is the difference in runtime of both versions
- Problem: branch-prediction unit(s)
 - Modern processors contain hardware that tries to predict the target of a branch instruction
 - In case the prediction was correct → no penalty, because correct instructions are in pipeline
 - In case the prediction was wrong → penalty
 - Branch-prediction units do a very good job detecting regular branch patterns

Exercise 3: Impact of control hazards

- How can the impact of control-hazard induced pipeline flush be quantified?
 1. Measure performance of some code that **does** contains branch instructions
 2. Measure performance **the same some** code that contain **no** branch instructions
 - The cost of rewinding the pipeline is the difference in runtime of both versions
- Problem: branch-prediction unit(s)
 - Modern processors contain hardware that tries to predict the target of a branch instruction
 - In case the prediction was correct → no penalty, because correct instructions are in pipeline
 - In case the prediction was wrong → penalty
 - Branch-prediction units do a very good job detecting regular branch patterns
 - To prevent the branch-prediction unit from interfering with measurements, we need to “confuse” the unit on purpose by using irregular branches

- We can create an irregular branch pattern using random numbers

- We can create an irregular branch pattern using random numbers
- Use an `init()` function to fill an array (e.g., an integer array) **randomly** with either **one** or **zero**
 - To this end, you can use the `srand(3)` and `rand(3)` functions on Linux
 - To access the manual, type “`man 3 rand`” on the command line

Random branches

- We can create an irregular branch pattern using random numbers
- Use an `init()` function to fill an array (e.g., an integer array) **randomly** with either **one** or **zero**
 - To this end, you can use the `srand(3)` and `rand(3)` functions on Linux
 - To access the manual, type “`man 3 rand`” on the command line
- You can then use the data in the array to decide whether to jump or not, e.g.:

```
for (i=0; i<N; ++i) {  
    if (A[i] == 0)  
        some_code;  
    else  
        other_code;  
}
```

- In your `main()` function, dynamically allocate memory for an (integer)-array that can hold one million elements

- In your `main()` function, dynamically allocate memory for an (integer)-array that can hold one million elements
- Next, call the `init()` function to initialize the array with random values of zero and one

- In your `main()` function, dynamically allocate memory for an (integer)-array that can hold one million elements
- Next, call the `init()` function to initialize the array with random values of zero and one
- Implement a `benchmark()` function that is passed the pointer to the array and its length via parameters

- In your `main()` function, dynamically allocate memory for an (integer)-array that can hold one million elements
- Next, call the `init()` function to initialize the array with random values of zero and one
- Implement a `benchmark()` function that is passed the pointer to the array and its length via parameters
 - The function should initialize a variable called `result` to zero and iterate over the array with the random numbers
 - If the array contains a value of **zero**, the value one should be **added** to the `result` variable
 - If the array contains a value of **one**, the value one should be **subtracted** from the `result` variable

- In your `main()` function, dynamically allocate memory for an (integer)-array that can hold one million elements
- Next, call the `init()` function to initialize the array with random values of zero and one
- Implement a `benchmark()` function that is passed the pointer to the array and its length via parameters
 - The function should initialize a variable called `result` to zero and iterate over the array with the random numbers
 - If the array contains a value of **zero**, the value one should be **added** to the `result` variable
 - If the array contains a value of **one**, the value one should be **subtracted** from the `result` variable
- Make sure the `benchmark()` function is executed for at least 100ms
 - To this end, apply the same method as in the previous exercise

- In your `main()` function, dynamically allocate memory for an (integer)-array that can hold one million elements
- Next, call the `init()` function to initialize the array with random values of zero and one
- Implement a `benchmark()` function that is passed the pointer to the array and its length via parameters
 - The function should initialize a variable called `result` to zero and iterate over the array with the random numbers
 - If the array contains a value of **zero**, the value one should be **added** to the `result` variable
 - If the array contains a value of **one**, the value one should be **subtracted** from the `result` variable
- Make sure the `benchmark()` function is executed for at least 100ms
 - To this end, apply the same method as in the previous exercise
- Afterwards, calculate the execution time (in cycles) of one loop iteration of the `benchmark()` function

- In your `main()` function, dynamically allocate memory for an (integer)-array that can hold one million elements
- Next, call the `init()` function to initialize the array with random values of zero and one
- Implement a `benchmark()` function that is passed the pointer to the array and its length via parameters
 - The function should initialize a variable called `result` to zero and iterate over the array with the random numbers
 - If the array contains a value of **zero**, the value one should be **added** to the `result` variable
 - If the array contains a value of **one**, the value one should be **subtracted** from the `result` variable
- Make sure the `benchmark()` function is executed for at least 100ms
 - To this end, apply the same method as in the previous exercise
- Afterwards, calculate the execution time (in cycles) of one loop iteration of the `benchmark()` function
- Hint: Make sure to only use the `-O3` compiler option when compiling

- Next, we will execute the same code without branches

Exercise: Overview (II)

- Next, we will execute the same code without branches
- Write a `benchmark_nobbranch()` function that does **not** check the array's values but **always** adds a value of one to the `result` variable

Exercise: Overview (II)

- Next, we will execute the same code without branches
- Write a `benchmark_nobbranch()` function that does **not** check the array's values but **always** adds a value of one to the `result` variable
- Then, calculate the runtime of one loop iteration of this code

Exercise: Overview (II)

- Next, we will execute the same code without branches
- Write a `benchmark_nobbranch()` function that does **not** check the array's values but **always** adds a value of one to the `result` variable
- Then, calculate the runtime of one loop iteration of this code
- Question 1: What penalty do you observe for the code that contains branches?

- Next, we will execute the same code without branches
- Write a `benchmark_nobbranch()` function that does **not** check the array's values but **always** adds a value of one to the `result` variable
- Then, calculate the runtime of one loop iteration of this code
- Question 1: What penalty do you observe for the code that contains branches?
- Question 2: Assuming that the branch-prediction unit correctly predicts the branch-target address 50% of the time (a reasonable assumption, when using evenly distributed random numbers), how many pipeline stages do you think there before the execution stage in the Ivy Bridge microarchitecture?