Assembly Language Programming

Ľudmila Jánošíková

Department of Mathematical Methods and Operations Research
Faculty of Management Science and Informatics
University of Žilina

tel.: +421 41 513 4200
Ludmila.Janosikova@fri.uniza.sk
http://frdsa.fri.uniza.sk/~janosik/

Recommended texts

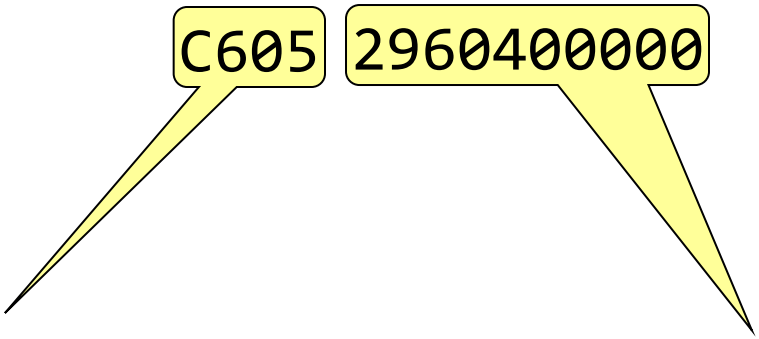
- Hyde, R.: The Art of Assembly Language Programming http://www.plantation-productions.com/Webster/www.artofasm.com/Windows/HTML/AoATOC.html
- Irvine, K.R.: Assembly Language for x86 Processors. 7th edition. Pearson, 2017.


He uses Java
rouming.cz
ROFLBOT

What is assembly language?

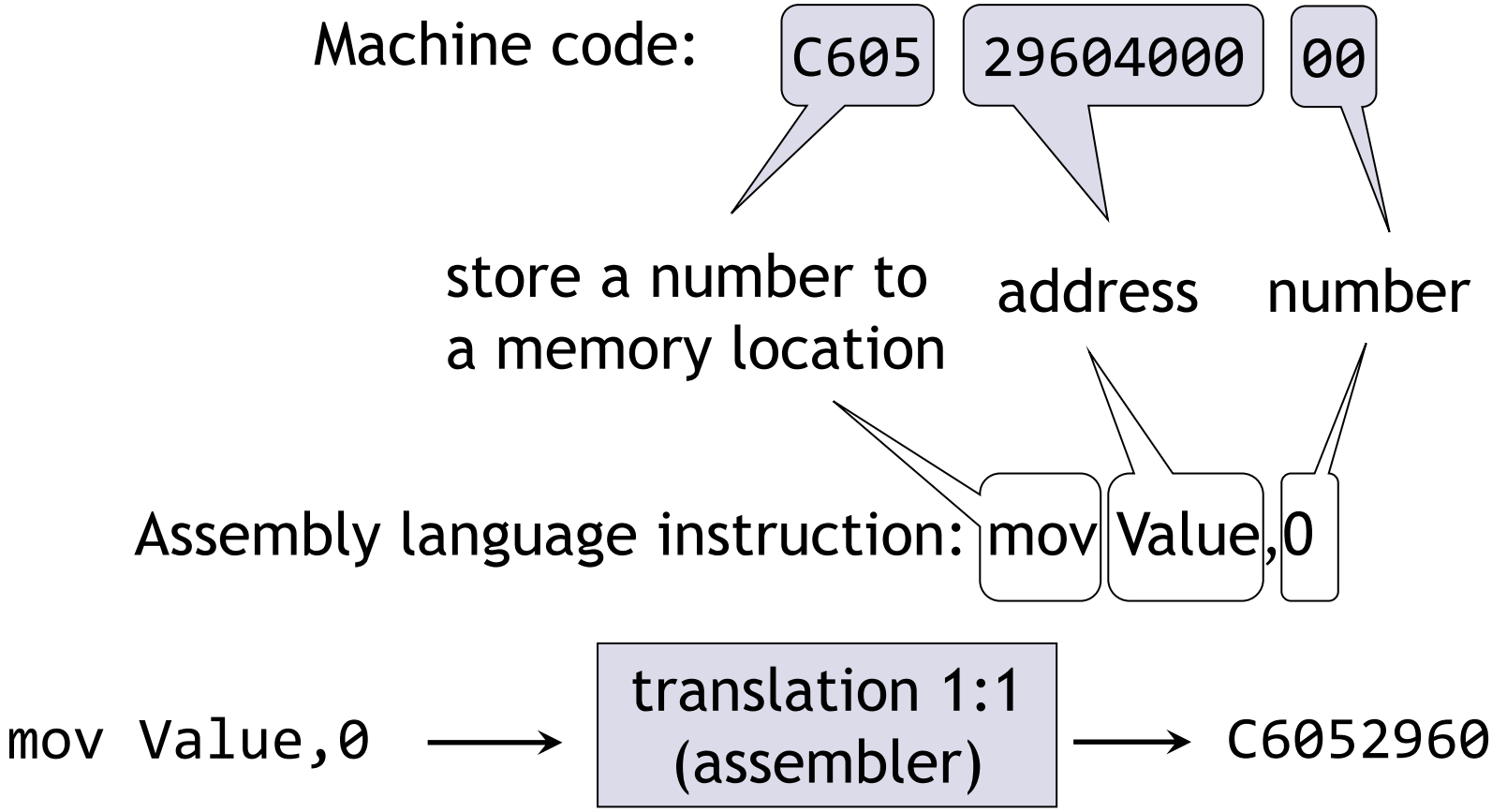
- low level programming language (like machine code)

Machine code: `C6052960400000`



operation code (opcode) –
what to do

operands –
data to be processed



Every processor has its own instruction set,

thus its own machine code,

thus its own assembly language.

Advantages of the assembly language

- The program written in the assembly language runs faster than the corresponding program written in a high level language
- and occupies less memory.
- You can exploit all capabilities of the processor (all operations the processor can perform).
- Knowledge of the assembly language enables you to understand, how a computer works, and to write better programs in a high level language.

When to use the assembly language?

- when you have not got another compiler
- when your program is supposed to run fast
- for programming a part of an operating system (e.g. a driver of an input or output device)
- for real-time computing when a control computer must response the event „immediately“
- for debugging a program written in a high level language

Disadvantages of the assembly language

- The program is hardware dependent (your program will not run on a different processor).
- You have to know the details on hardware and addressing modes.
- The source code is long and readable with difficulty.

➢ Example:
Add two integer variables `i` a `j` and store the result into variable `k`.

Java: `k = i + j;` `for (int p = 0; p < k; p++)`

Assembly language:
```
mov eax,i
add eax,j
mov k,eax
mov ecx,k
```

Optimized assembly language code?

Goal of the course

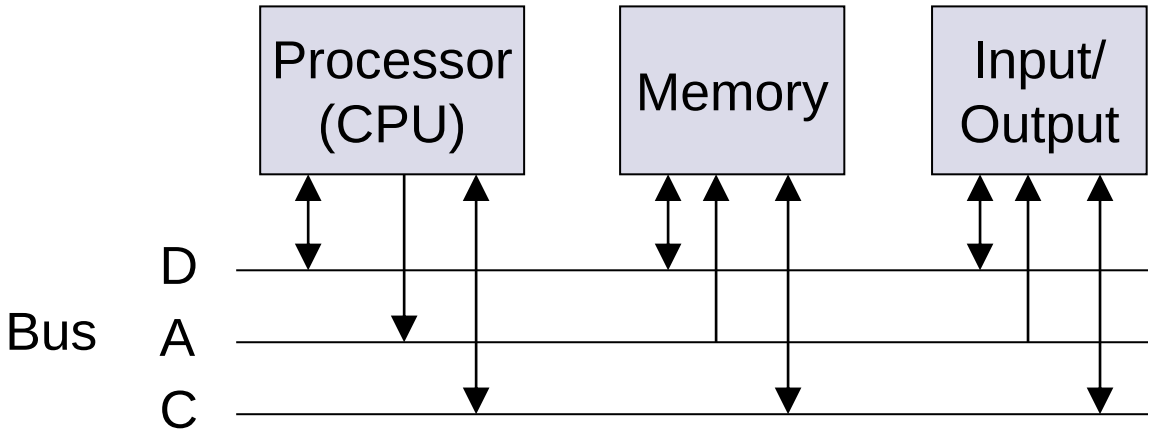
To learn how to

think

in assembly language.

Architecture (of a computer or processor) – organization of basic components with regard to their interconnection and operations.

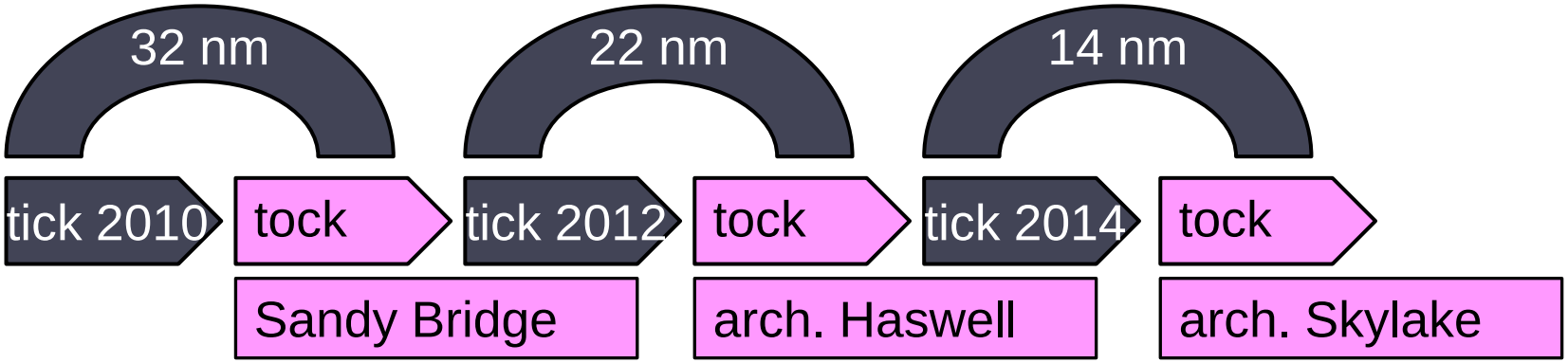
Bus architecture of a computer



- Address bus: unidirectional; group of wires which carries address information bits from processor to memory and peripherals (16, 20, 24 or more parallel signal lines)
- Data bus: bidirectional; group of wires which carries data from processor to memory and peripherals and vice-versa
- Control bus: bidirectional; group of wires which carries control signals from processor to memory and peripherals and vice-versa

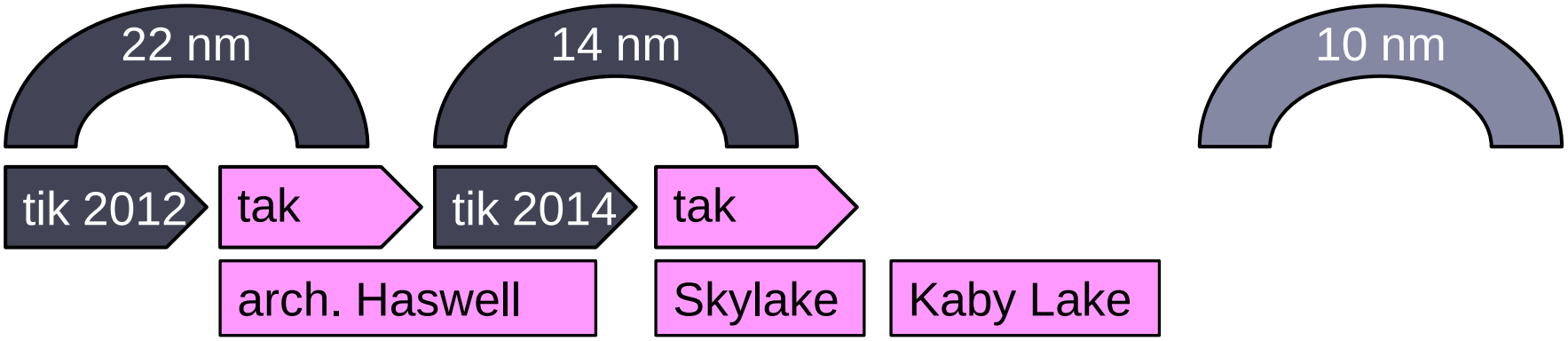
Architecture Intel Skylake

- launched in August 2015
- Skylake is the last commercial step „tock“ in Intel’s “tick-tock” design model:
  - „tick“ – improvement of manufacturing process technology (higher density of transistors on a chip)
  - „tock“ – better architecture with the aim to improve performance, reduce power consumption, and to bring new functions
- for desktops, servers and mobile devices

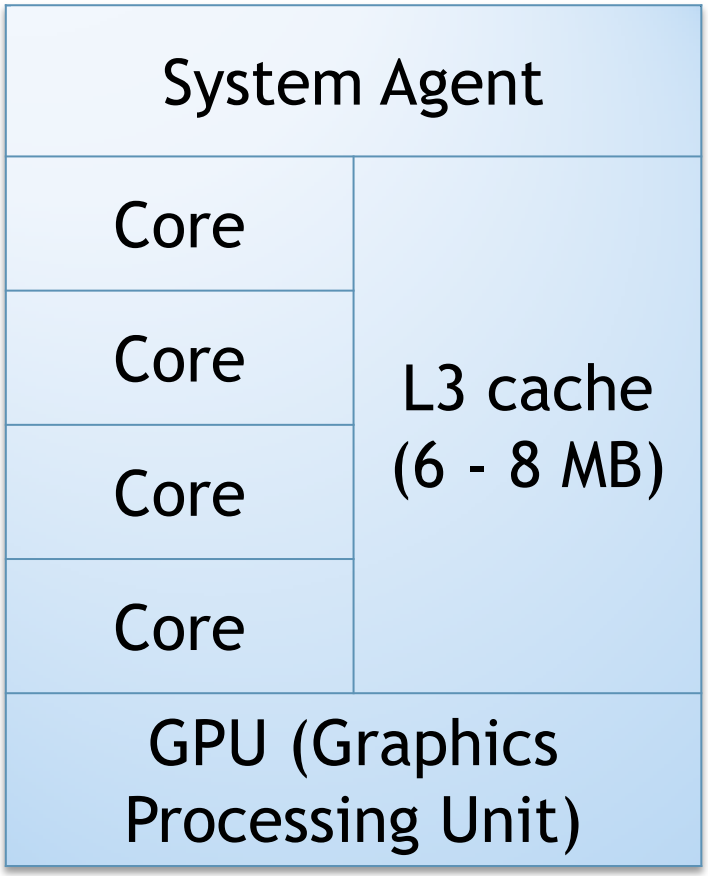


Architecture Intel Kaby Lake

- announced in August 2016
- Kaby Lake is produced using the same 14 nanometer manufacturing process technology as Skylake
- with Kaby Lake Intel broke “tick-tock” design model (technology – architecture) and turned to a new model technology – architecture – optimization.



Structure of a quad-core processor

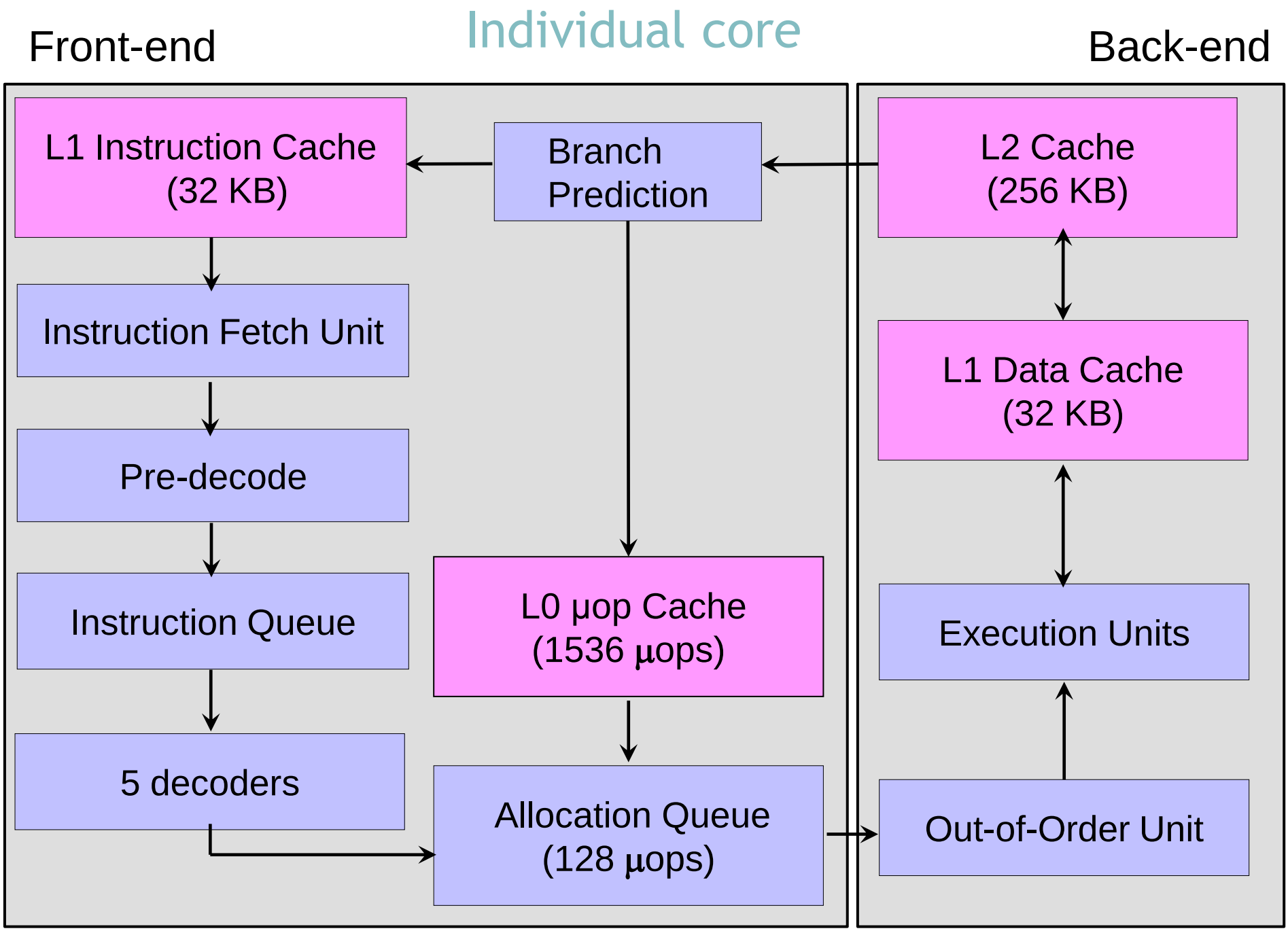


System Agent contains:

- memory controller – manages the flow of data going to and from the computer's main memory;
- bus interface;
- Image Processing Unit (only for mobile devices) – supports video and graphics processing functions (e.g. rotation, vertical/horizontal inversion) => access to main memory is minimized.

GPU

- hardware support for graphical operations


Front-end
Individual core
Back-end
L1 Instruction Cache
(32 KB)
Branch
Prediction
L2 Cache
(256 KB)
Instruction Fetch Unit
L1 Data Cache
(32 KB)
Pre-decode
Instruction Queue
L0 µop Cache
(1536 µops)
Execution Units
5 decoders
Allocation Queue
(128 µops)
Out-of-Order Unit

Important features of modern processors

- cache memory
- pipelining
- dynamic execution
- SIMD technology (Single Instruction Multiple Data)

Goal:

Faster execution through parallelism.

Cache memory

- CPU cache is used to reduce the access time to data and instructions in memory. The cache is a smaller, faster memory which stores copies of the data from main memory locations.
- Instructions in your program cannot address it; the hardware decides on its contents.
- Motivation: in a short time period the program repeatedly accesses the same or adjacent memory locations

Temporal locality of reference

- In a short time period (e.g. during a loop) the program repeatedly accesses the same memory locations.

Spatial locality of reference

- The program frequently accesses adjacent memory locations.

```
for (int i = 0; i < 10; i++) A[i] = 0;
```

Temporal locality: variable `i`; bytes containing the coded instructions of the loop

Spatial locality: adjacent entries of array `A`; bytes containing the coded instructions of the loop

That is why a block of data rather than a single variable or instruction is fetched from the main memory to the cache.

Write policy from the data cache to the higher level memory

- Write-through – data are written to the cache and at the same time to the higher level memory.
- Write-back – more efficient; the cache control unit tracks, whether data in the cache were changed; if they did, they are written to the higher level memory when they are released from the cache due to the lack of space.

Architecture Skylake:

Write through: data L1 $\rightarrow$ L2

Write back: L2 $\rightarrow$ L3, L3 $\rightarrow$ main memory

Pipelining

- More instructions are executed at the same time.

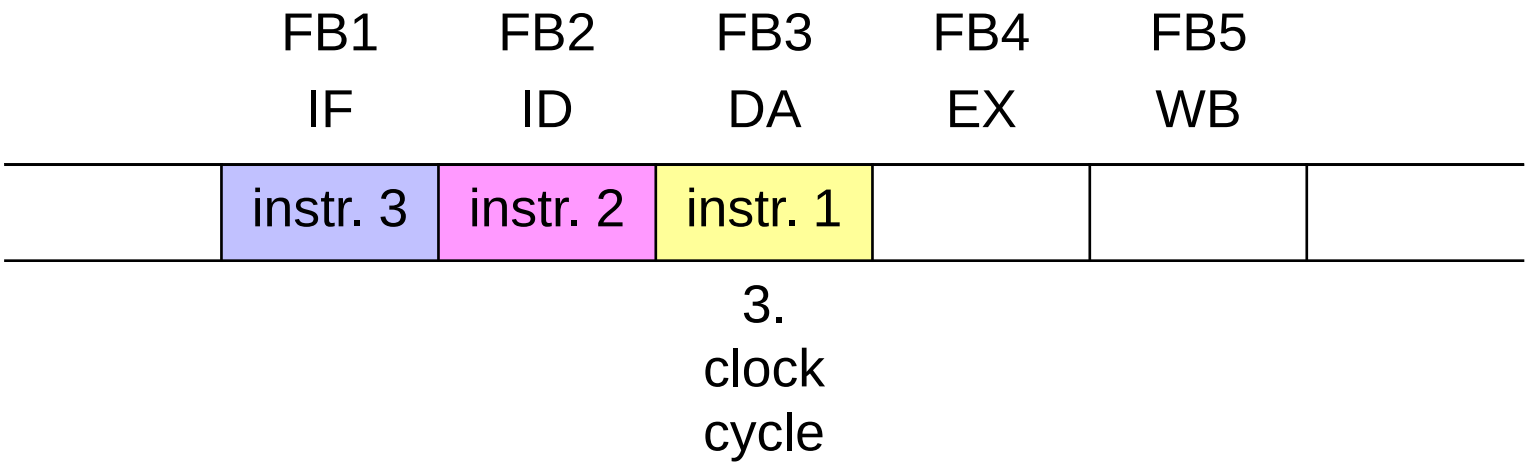
Basic computer operation cycle

1. Instruction Fetch (IF) – a coded instruction is read from memory
2. Instruction Decode (ID) – processor divides the operation into basic steps
3. Data Access (DA) - instruction that needs data from memory presents the address to the memory subsystem and receives back the data
4. Execution (EX)
5. Write Back (WB) – store the results in the proper place

Every stage is performed in a separate functional block, or is decomposed to simpler operations and performed in several functional blocks.

Functional block (FB) – a group of the logic circuits performing a common task (e.g. Arithmetic and Logic Unit – ALU – arithmetic operations with integers and logical operations with bits).

Pipelining: functional blocks are arranged logically one after another and form a pipe:



Clock cycles	1	2	3	4	5	6	7	8	9
Instruction 1	IF	ID	DA	EX	WB				
Instruction 2		IF	ID	DA	EX	WB			
Instruction 3			IF	ID	DA	EX	WB		
Instruction 4				IF	ID	DA	EX	WB	
Instruction 5					IF	ID	DA	EX	WB

5 instructions are being executed in this cycle.

Pipeline hazards

- data dependency – the problem of trying to use data before they are available (1st instr. WB, 2nd instr. DA))

```
mov ebx,Address; store address to register ebx
mov eax,[ebx]; copy the doubleword addressed by
ebx to register eax
```

Consequence: bubbles in pipeline

Clock cycles	1	2	3	4	5	6	7	8	9
Instruction 1	IF	ID	DA	EX	WB				
Instruction 2		IF	ID		DA	EX	WB		
Instruction 3			IF		ID	DA	EX	WB	
Instruction 4					IF	ID	DA	EX	WB
Instruction 5						IF	ID	DA	EX

- structural hazards – multiple instructions require the same processor resource during a given clock cycle, e.g. address bus (3rd cycle) or cache memory (5th cycle) => do calculations with registers rather than variables

Clock cycles	1	2	3	4	5	6	7	8	9
Instruction 1	IF	ID	DA	EX	WB				
Instruction 2		IF	ID	DA	EX	WB			
Instruction 3			IF	ID	DA	EX	WB		
Instruction 4				IF	ID	DA	EX	WB	
Instruction 5					IF	ID	DA	EX	WB

- control hazards – they are caused by branch instructions that contain a logic condition which must be evaluated first to decide if the branch should be taken or not. If the pipeline contains instructions from the wrong branch, they need to be killed.

```
cmp k,0
je Finish
mov eax,i
add eax,j
mov k,eax
...
Finish:
```

Instruction `je` in step WB modifies instruction pointer (points to the next instruction that is to be fetched from memory), so the subsequent instructions in pipeline have to be cancelled.

5. clock	FB1 IF	FB2 ID	FB3 DA	FB4 EX	FB5 WB	
	instr. 5	instr. 4	instr. 3	instr. 2	instr. 1	

Branch prediction

Jumps (15 to 25 % of instructions):

- unconditional – like “goto”.
  Unconditional jumps have no specific condition (they do not depend on the result of the previous operation). Immediately after decoding the processor knows where the program execution will continue and it can start fetching instructions from the new address that is an operand of the jump instruction.
- conditional – if, case, for, repeat, while.
  A conditional jump contains a logic condition that must be evaluated first. If the condition is met, the branch is taken and the execution continues at the target location.

The goal of branch prediction is to predict:

- if the conditional jump will be taken or not;
- the target address.

Prediction:

- dynamic – used for jumps that have already occurred in the program (e.g. a jump at the end of the loop)
- static – if the jump instruction has not executed yet

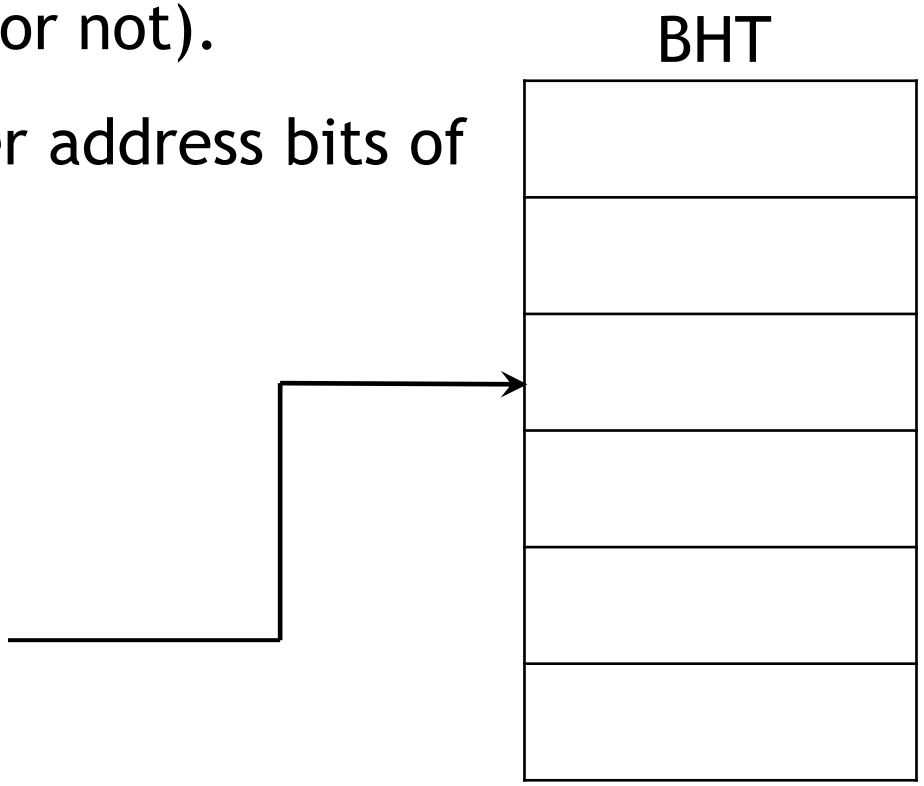
Two-level dynamic prediction

From the behaviour of the jump instruction in the past the processor predicts its behaviour in the future.

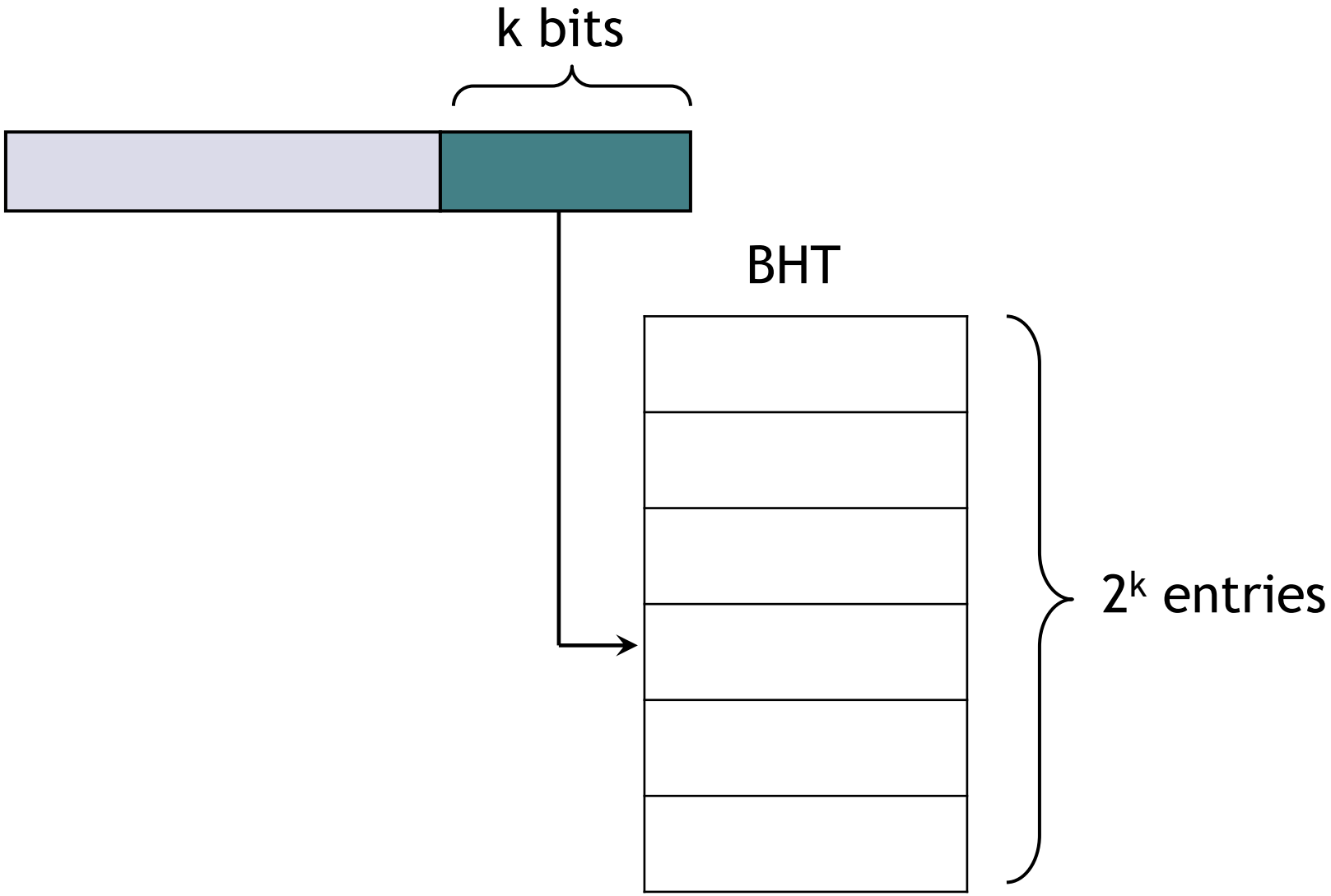
Branch History Table (BHT) stores the history of several hundreds of latest jumps (whether the branch was recently taken or not).

BHT is indexed by the low-order address bits of the jump instruction.

```
mov eax,i
add eax,j
mov k,eax
cmp k,0
je Finish
```

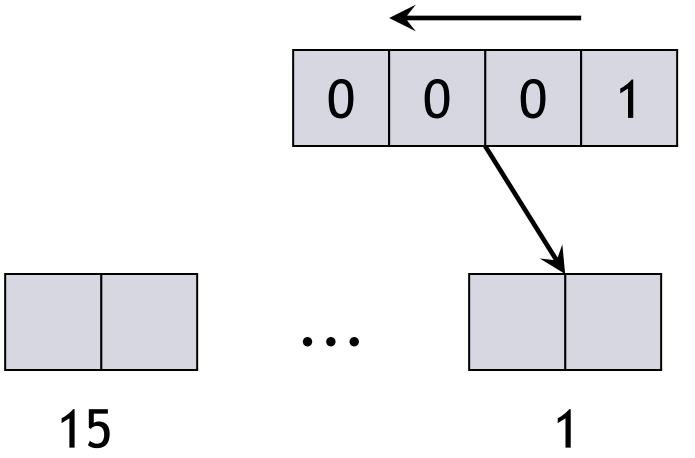


address of the jump instruction

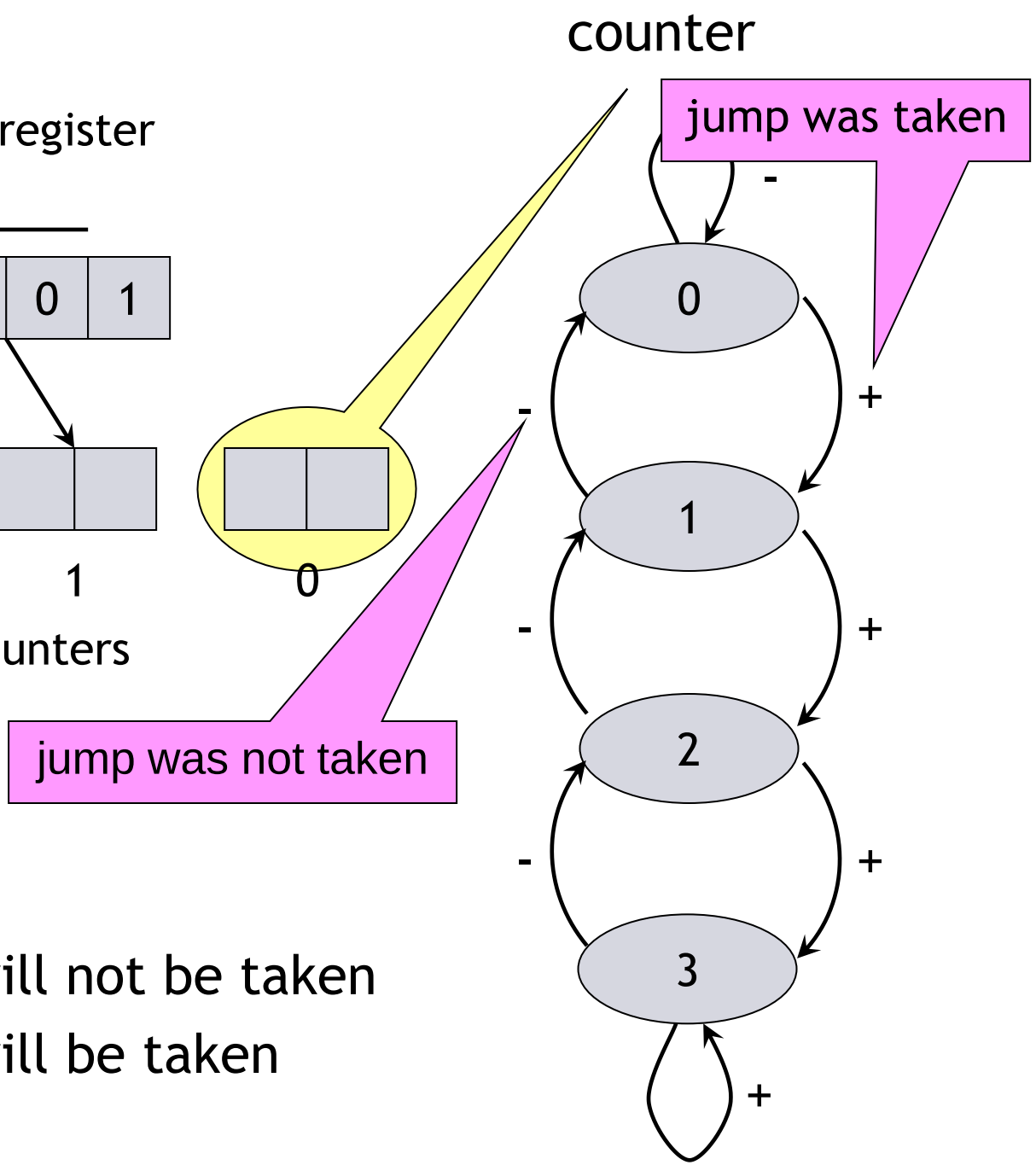


BHT entry

1. level: 4-bit shift register



2. level: 16 2-bit counters



Prediction:

- in state 0 a 1: will not be taken
- in state 2 a 3: will be taken

When the jump is taken for the first time, the target address is added to the list of target addresses (Branch Target Buffer – BTB).

If the prediction is „the branch will be taken“, then the target address is retrieved from BTB and the instruction need not be decoded again.

Static prediction

– is used if the jump has not occurred yet (it has no BHT entry).

Static prediction is based on the statistical analysis of behaviour of typical branching instructions like the loop instruction. The loop will more probably continue (through a jump to its beginning if the termination condition is at the end of the loop) than terminate. That is why backward branches are predicted to be taken, while forward branches are predicted to be not taken.

If the termination condition is at the end of the loop, backward jump to the beginning repeats the loop.

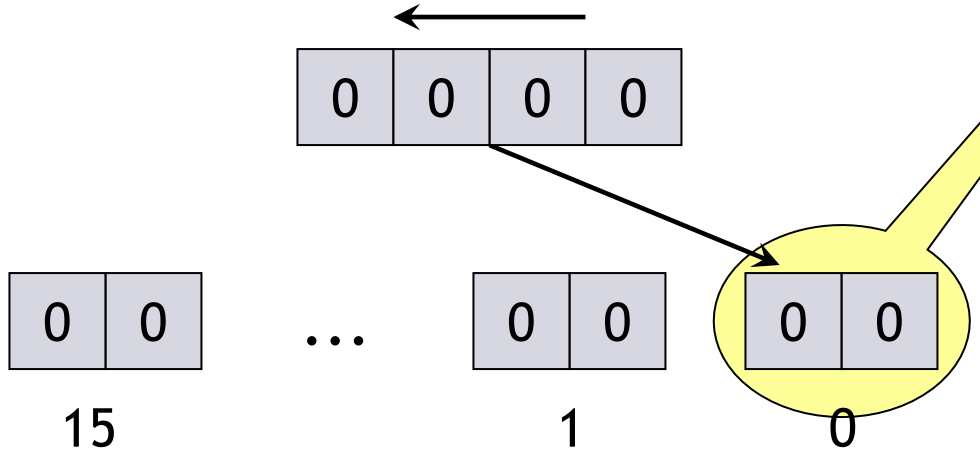
```
    mov edi,0; edi points to the first character of the string
WriteLoop:
    mov al,[edx+edi]; copy character on the offset edx+edi into
                      register al
    call WriteChar; display character, whose ASCII code is in al
    inc edi; increment index by 1
    cmp edi,n; compare index with variable n
    jl WriteLoop; if less, jump to WriteLoop
Finish:
```

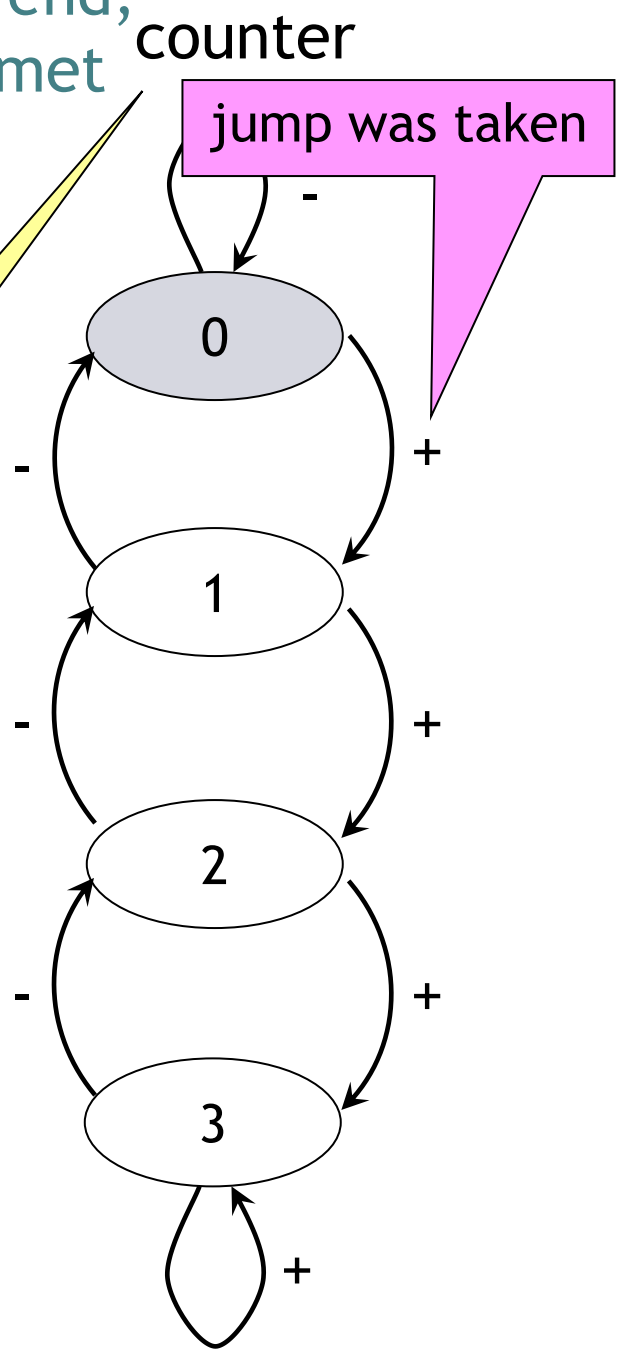
If the termination condition is at the beginning of the loop, forward jump behind the loop terminates the loop.

```
    mov edi,0; edi points to the first character of the string
WriteLoop:
    mov al,[edx+edi]; copy character on the offset edx+edi into
                      register al
    cmp al,0; compare al with zero
    je Finish; if equal, jump to Finish
    call WriteChar; display character, whose ASCII code is in al
    inc edi; increment index by 1
    jmp WriteLoop; unconditional jump to the beginning
Finish:
```

Loop with the termination condition at the end; return to the beginning, if the condition is met

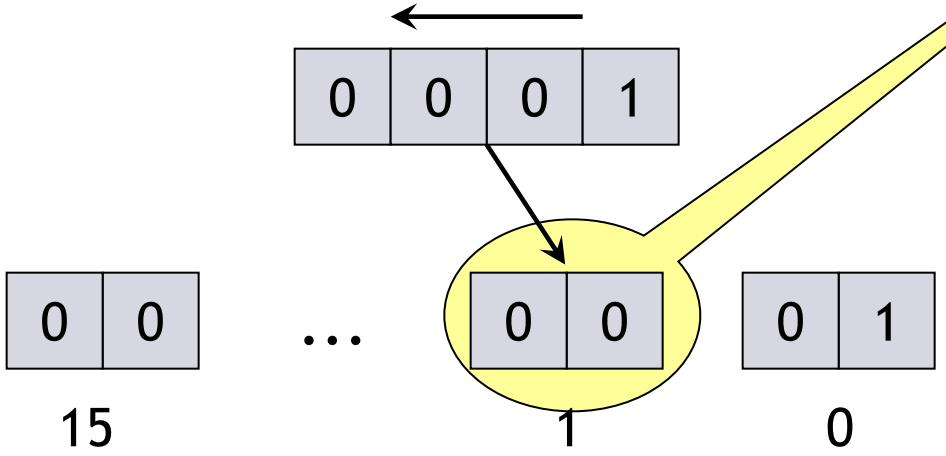
1. run: static prediction – branch should be taken

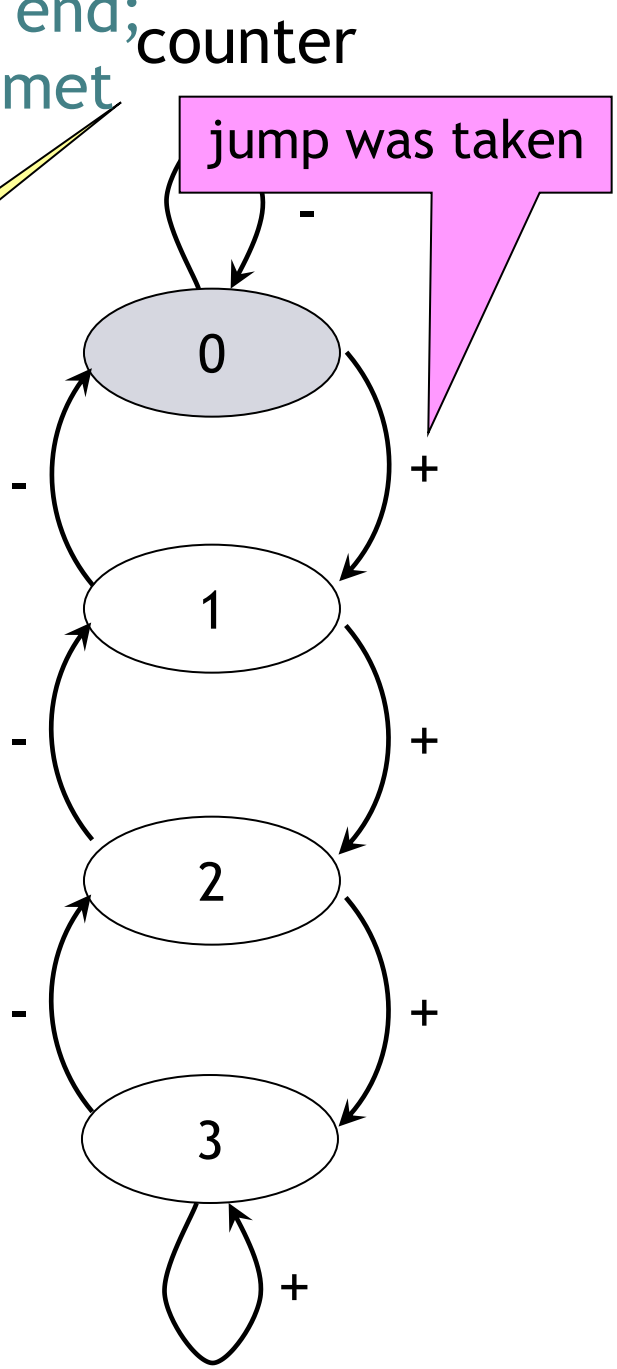




Loop with the termination condition at the end; return to the beginning, if the condition is met

2. run:

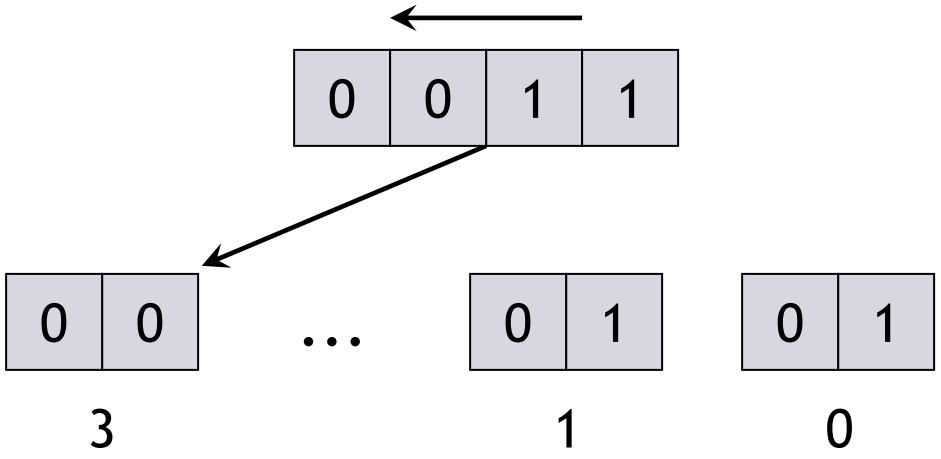




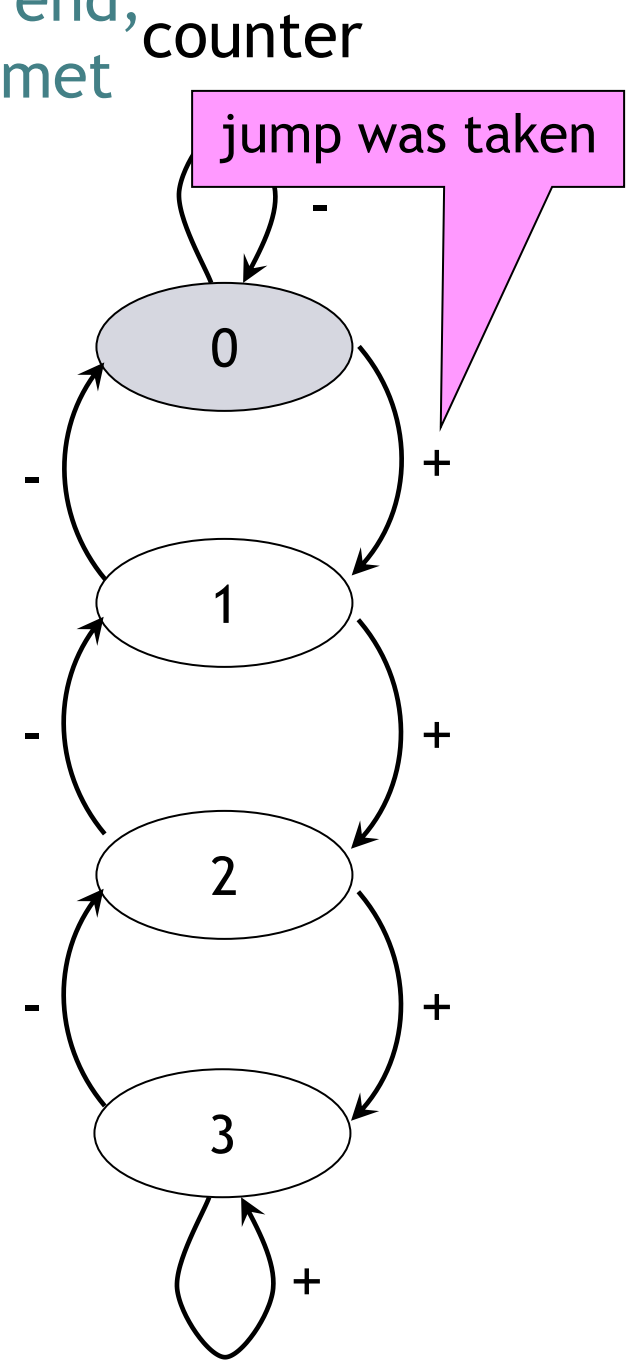
Bad prediction!

Loop with the termination condition at the end; return to the beginning, if the condition is met

3. run:

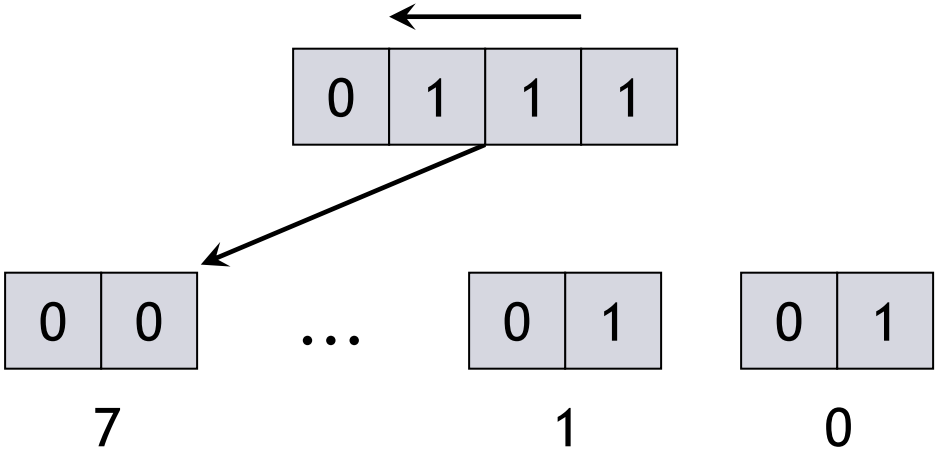


Bad prediction!

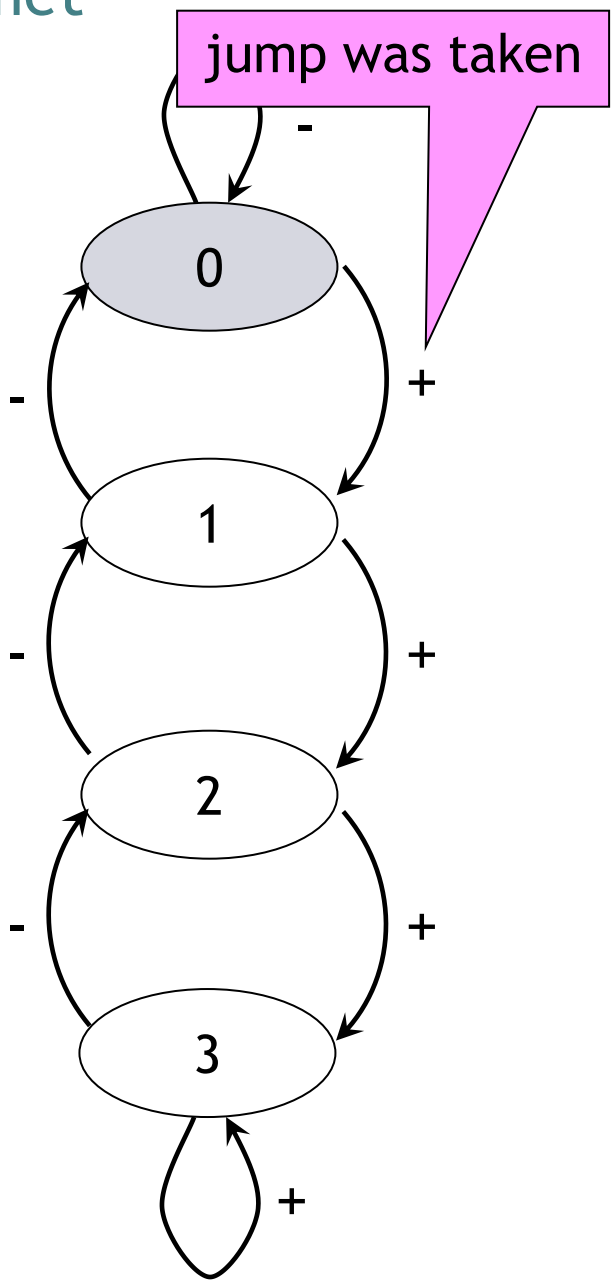


Loop with the termination condition at the end; return to the beginning, if the condition is met

4. run:

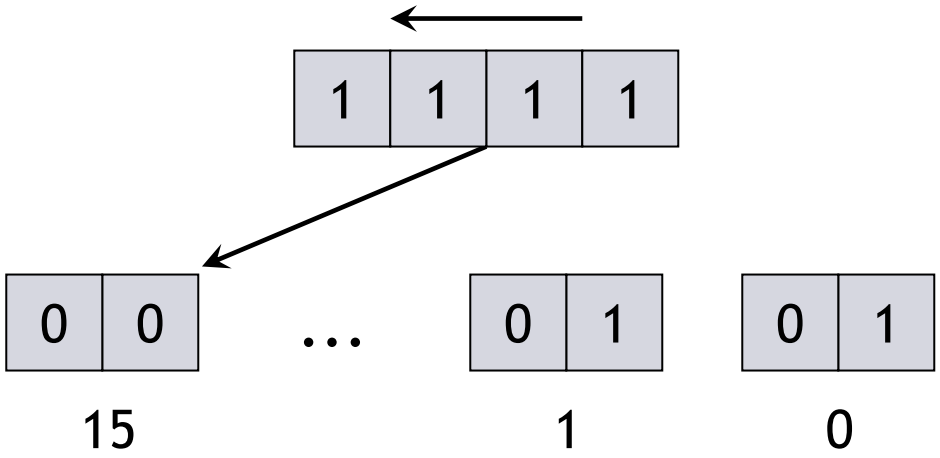


Bad prediction!

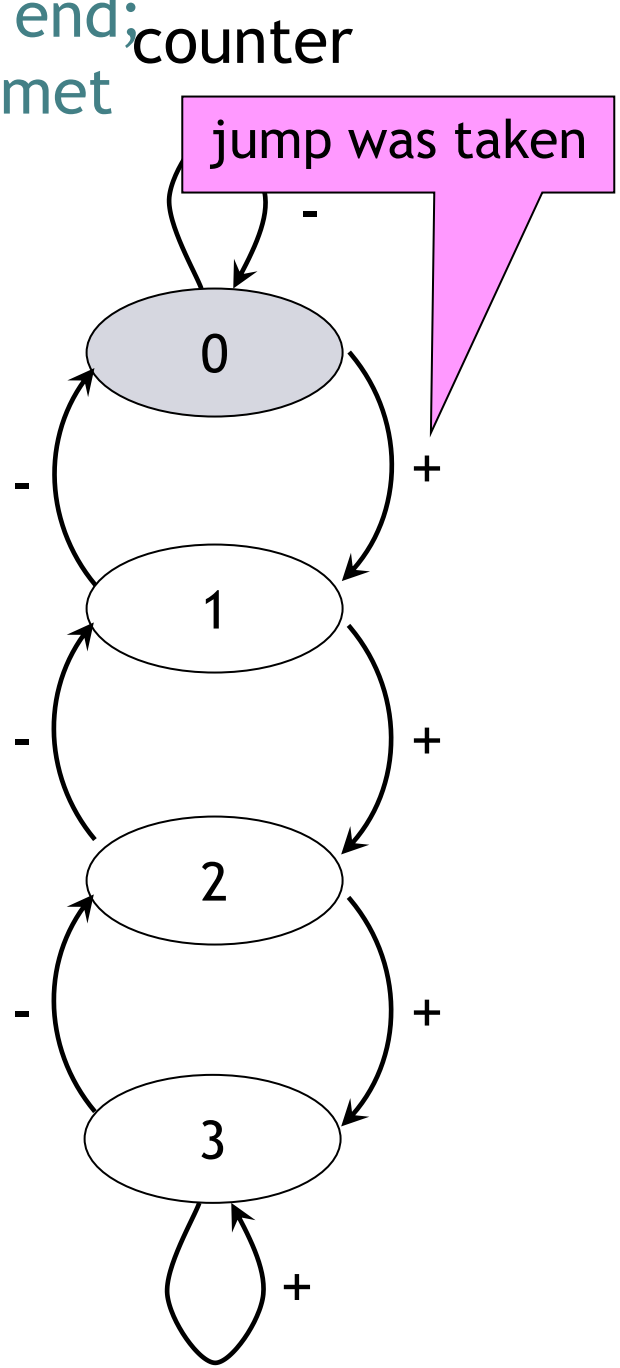


Loop with the termination condition at the end; return to the beginning, if the condition is met

5. run:

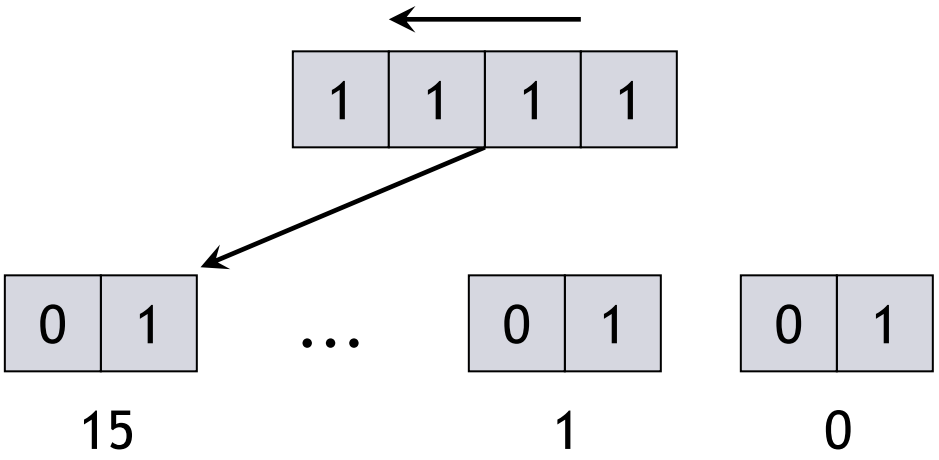


Bad prediction!

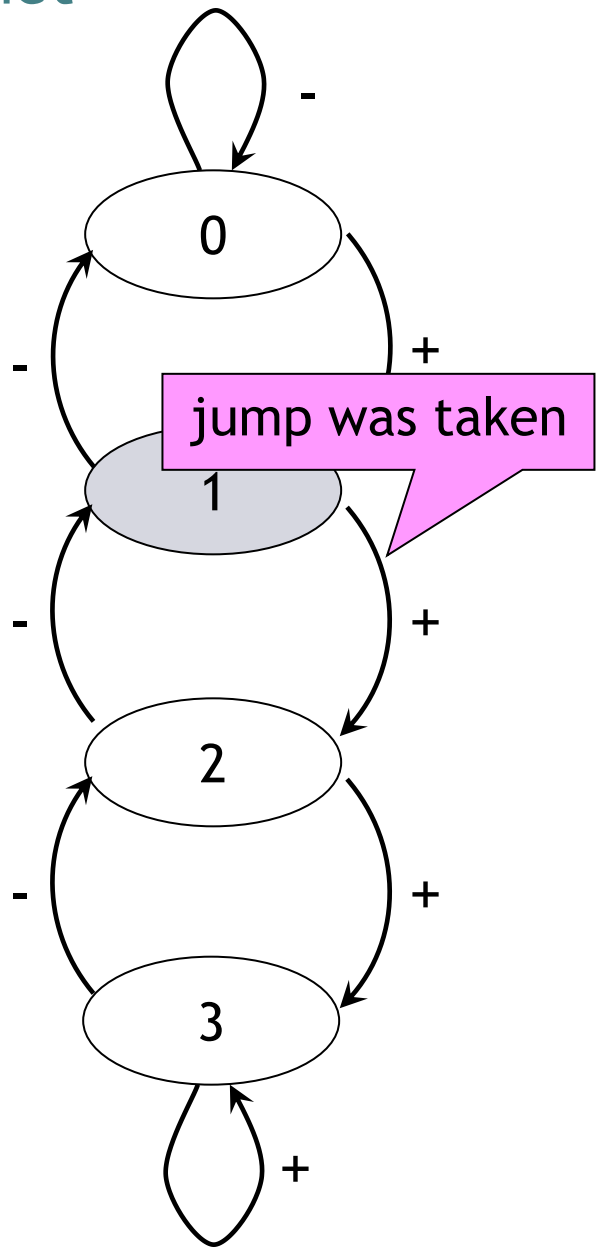


Loop with the termination condition at the end; return to the beginning, if the condition is met

6. run:

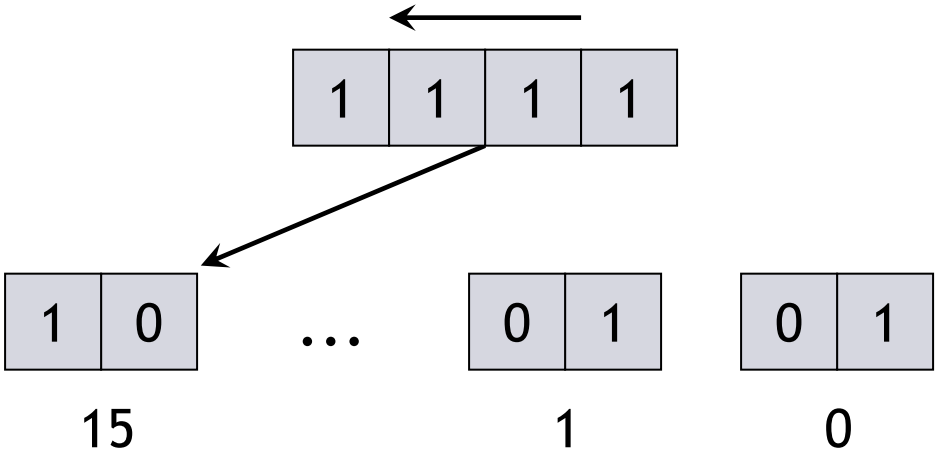


Bad prediction!



Loop with the termination condition at the end; return to the beginning, if the condition is met

7. run:



Good prediction!