

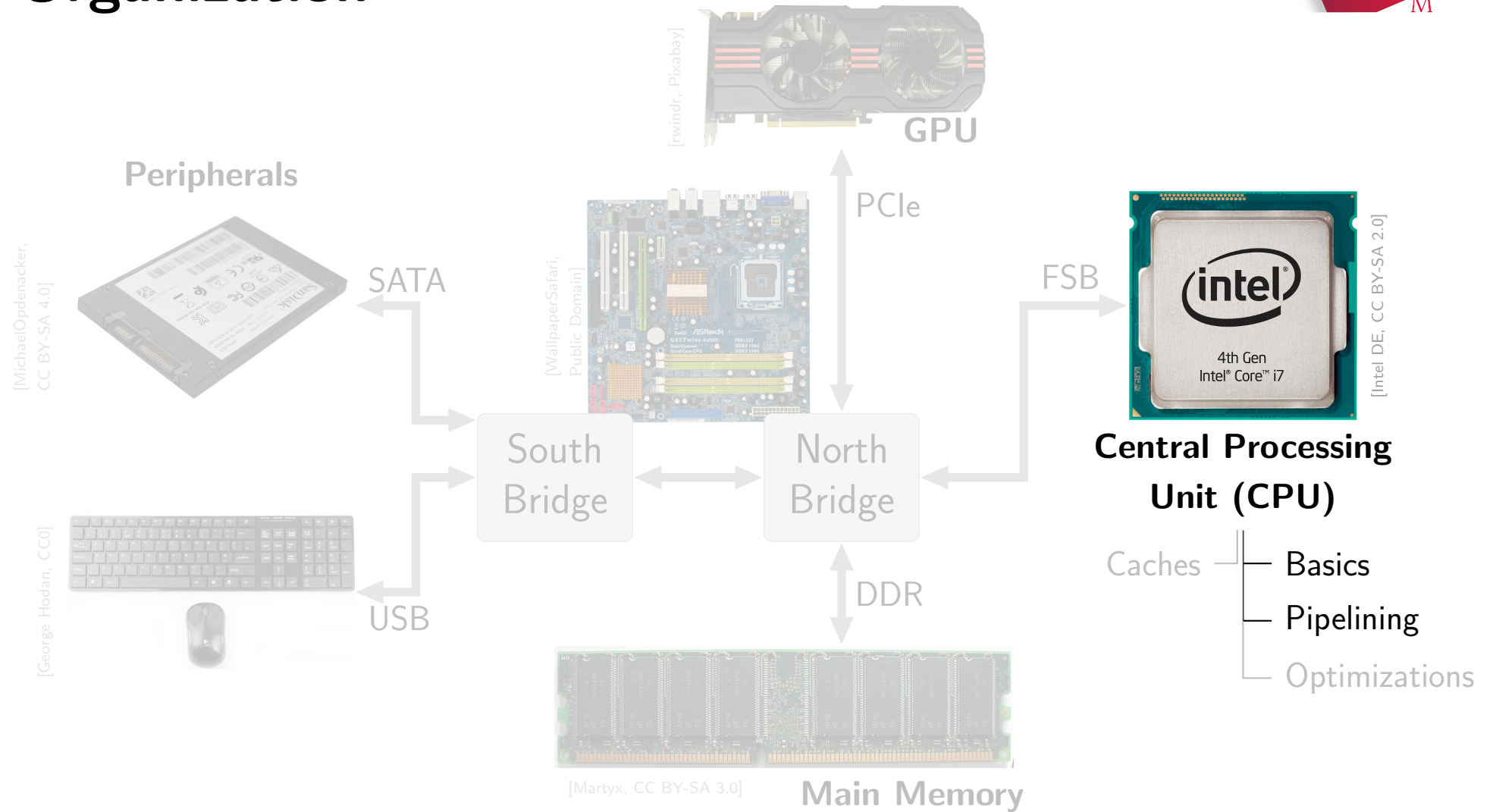
COMPUTER ARCHITECTURE

Chapter 3 – CPU Pipelining

Prof. Dr.-Ing. Stefan Wallentowitz

Department 07 – Munich University of Applied Sciences

Course Organization





Recap: Instruction Processing

Fetch instruction (short: FE)

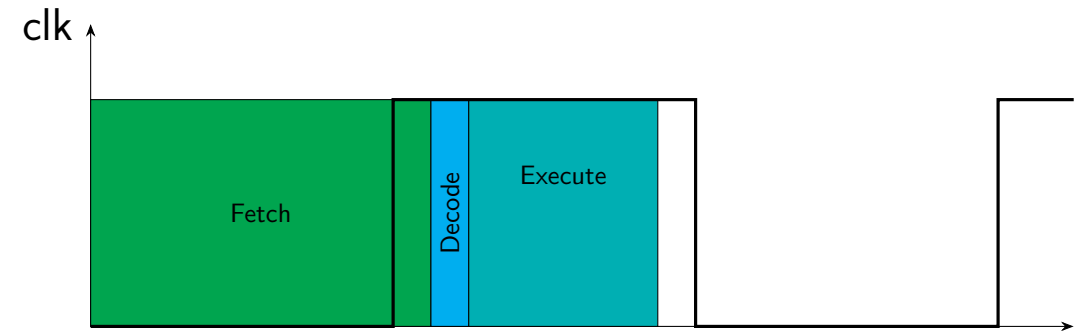
- Pointer to next instruction from current program counter
- Load the instruction from memory

Decode instruction (DE)

- Get operands from register file
- Extend sign if needed

Execute (EX)

- ALU-Operation from instruction
- Calculate effective address
- Control flow: check condition





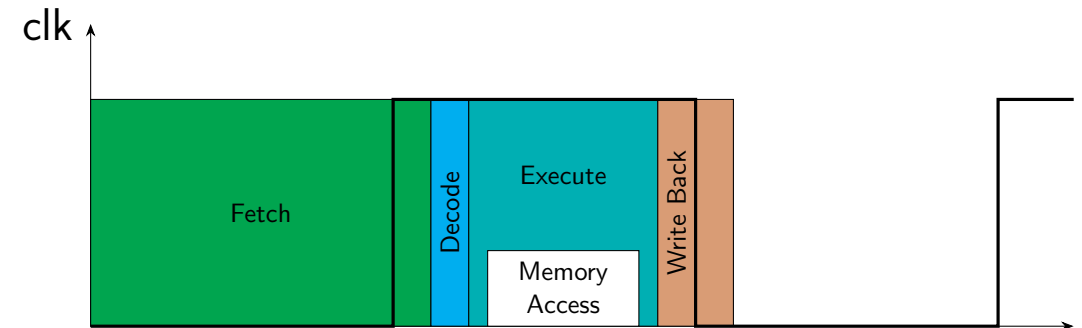
Recap: Instruction Processing

Memory Access (MA)

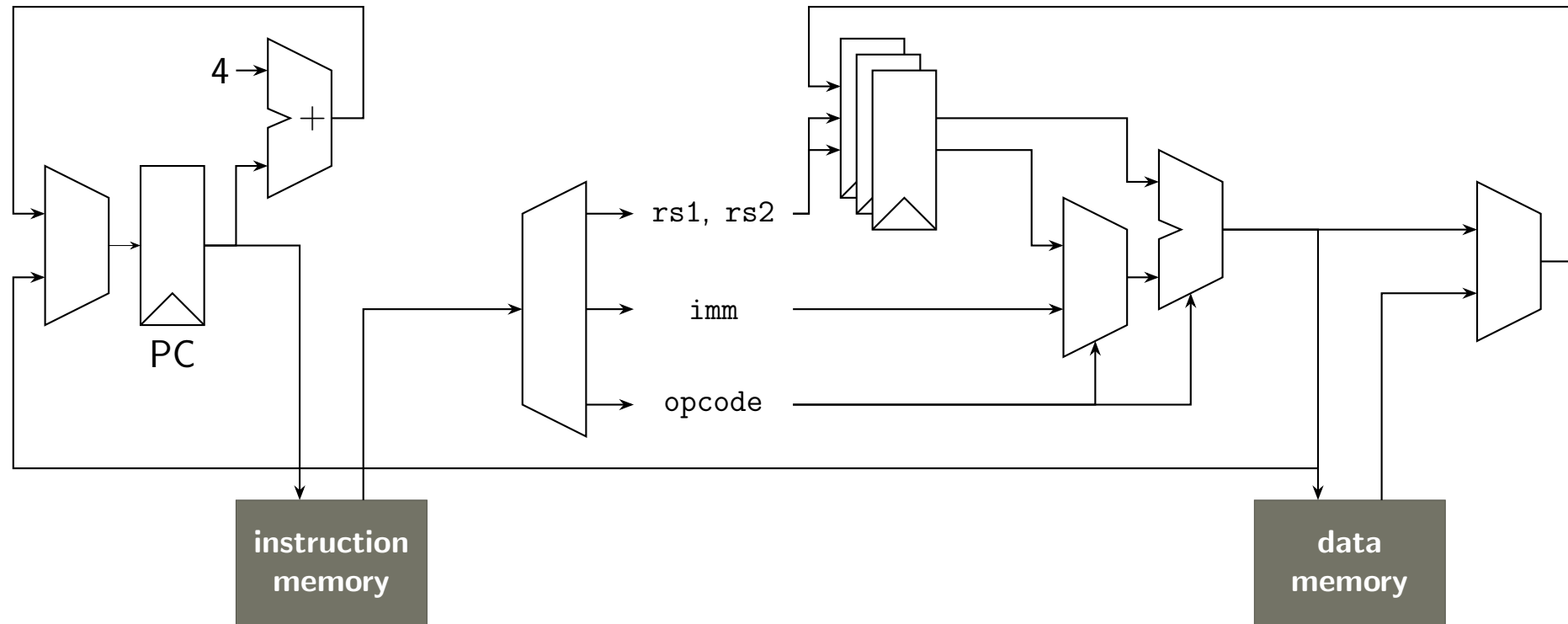
- Reading or writing to memory
- optional

Write Back (WB)

- Write result into destination register
- Commit new program counter

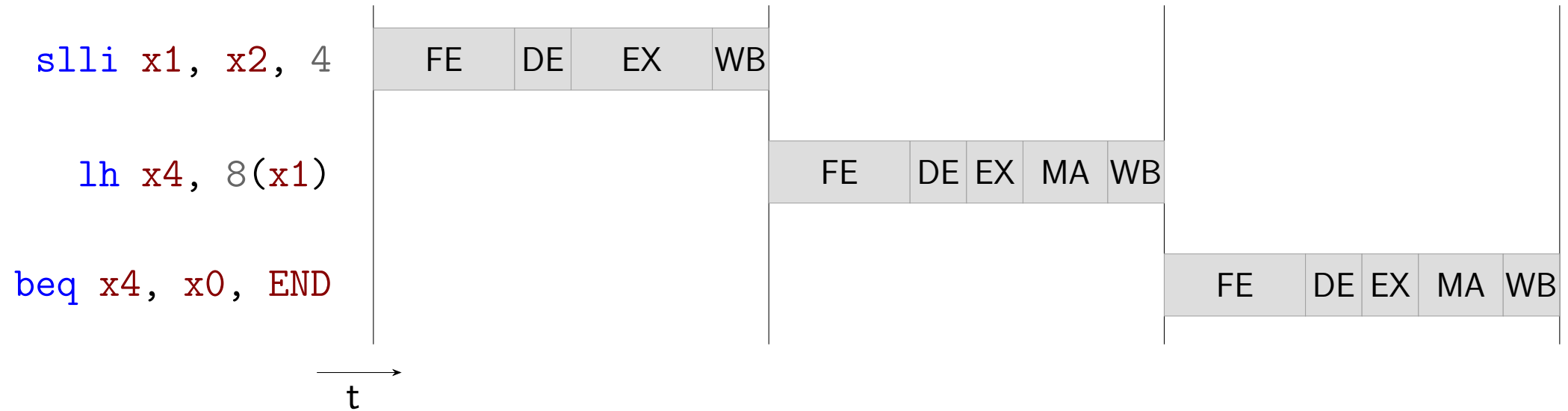


Hardware Implementation (simplified)





Serial Execution



Can this be executed more efficiently?



Inspiration: Assembly Line



(c) BlueSpringsFordParts, CC BY 2.0





Instruction Pipelining

Overlapping execution of instructions

- Start phase for next instruction once current completes phase
- Parallelization of execution: Multiple concurrent instructions

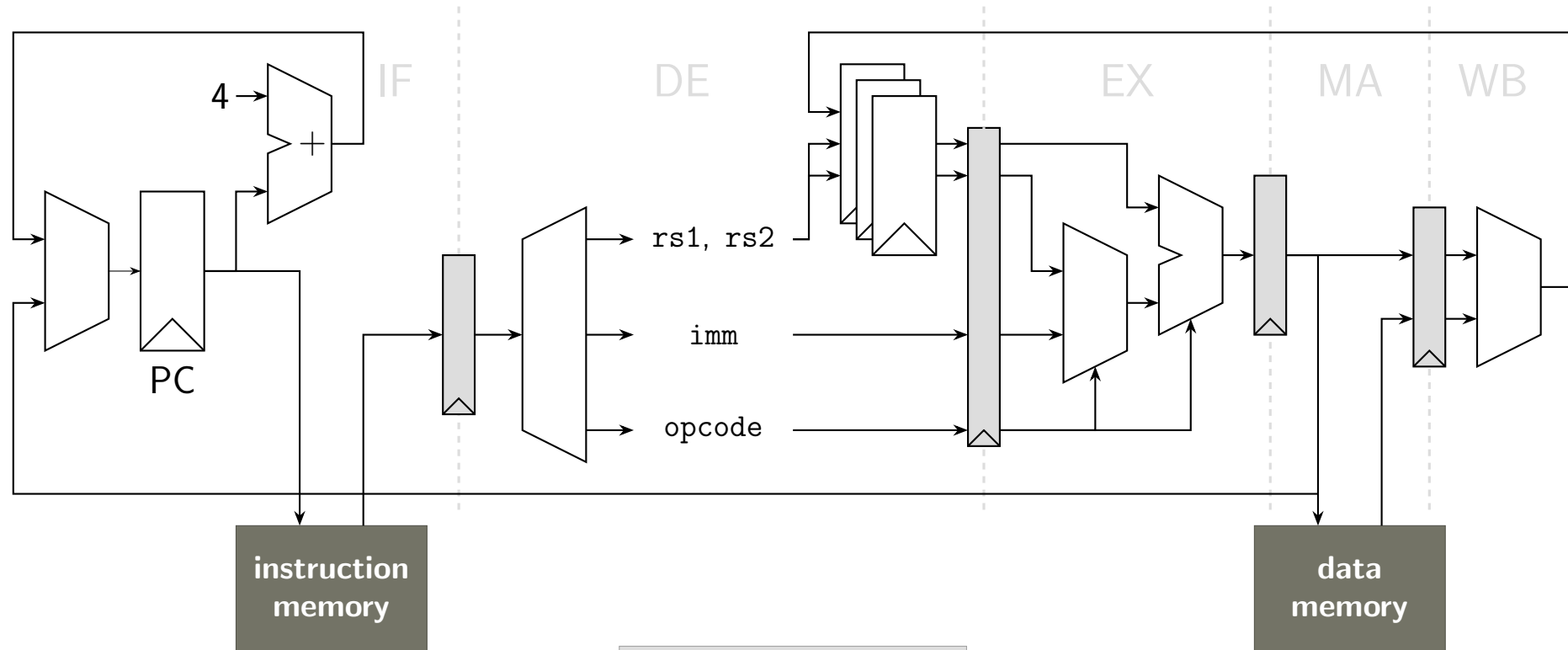
Pipeline stages are synchronized, handover at same time



Slowest stage determines clock frequency

Key technology for fast CPU implementations

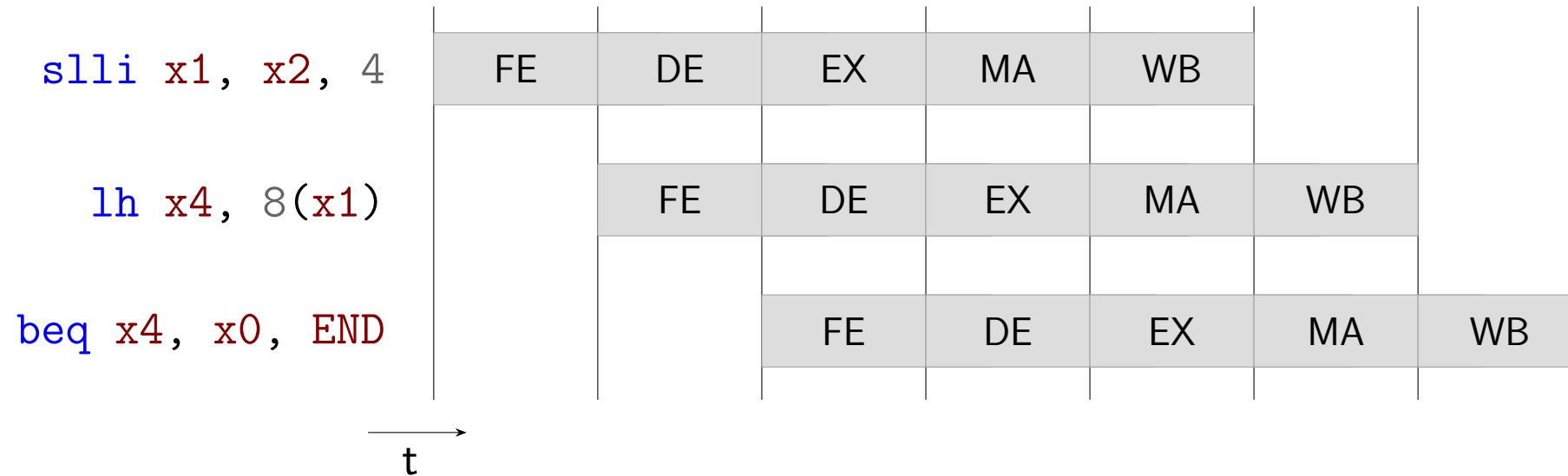
Hardware Implementation (simplified)



Where are the pipeline stages?



Pipeline Speedup



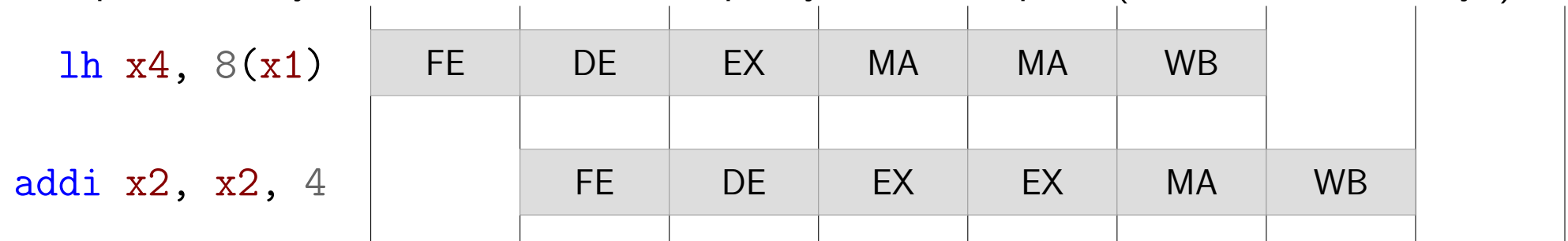


Structural Hazards

Problem: Instruction may need multiple cycles to complete stage

Next instruction is blocked → The IPC decreases

Example: Memory access that needs multiple cycles to complete (see lecture "memory")



Structural hazards can generally not be avoided!





Pipeline Hazards

In general, pipeline hazards are situations that block an instructions from entering the next pipeline stage

- **Structural hazards** are resource conflicts due to hardware availability
- **Data hazards** occur when a result of a previous command is not available
- **Control hazards** are a result of changes in the control flow

Hazards lead to a *pipeline stall* → IPC decreases



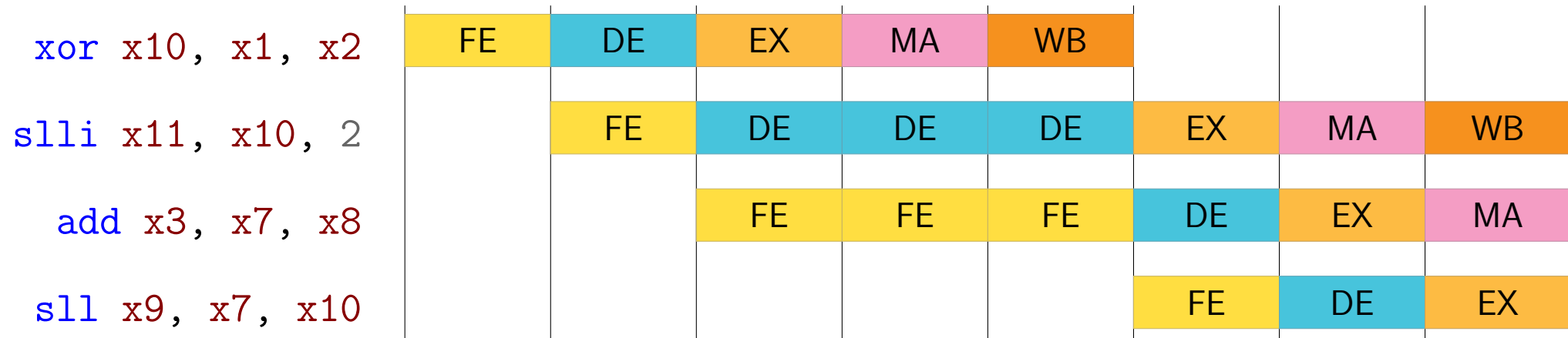


Data Hazards

Problem: Conflict of operands between instructions

Example: Need result of previous instruction

- Result written in Writeback stage → Execution blocks until result becomes visible



How can we avoid the problem?





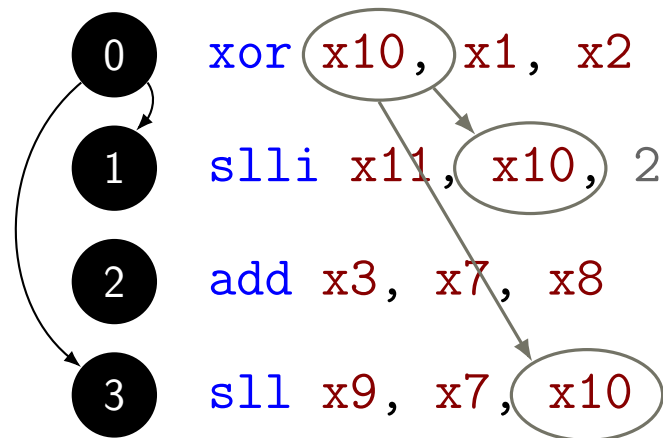
Instruction Level Parallelism

Instruction stream is sequentially stored in memory

But:

- Can we reorder instructions?
- Can instructions be executed in parallel?

Foundation of a large number of optimizations in computer architecture



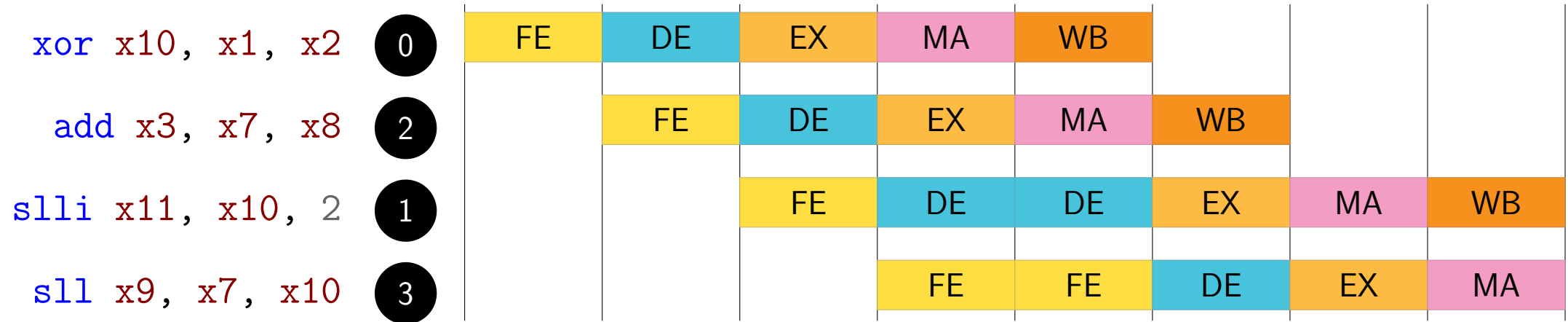
Why not execute in any order?





Reorder instructions

Reorder instruction so that the data hazard is resolved



Reordering limited, especially with "deeper" (more stages) pipelines

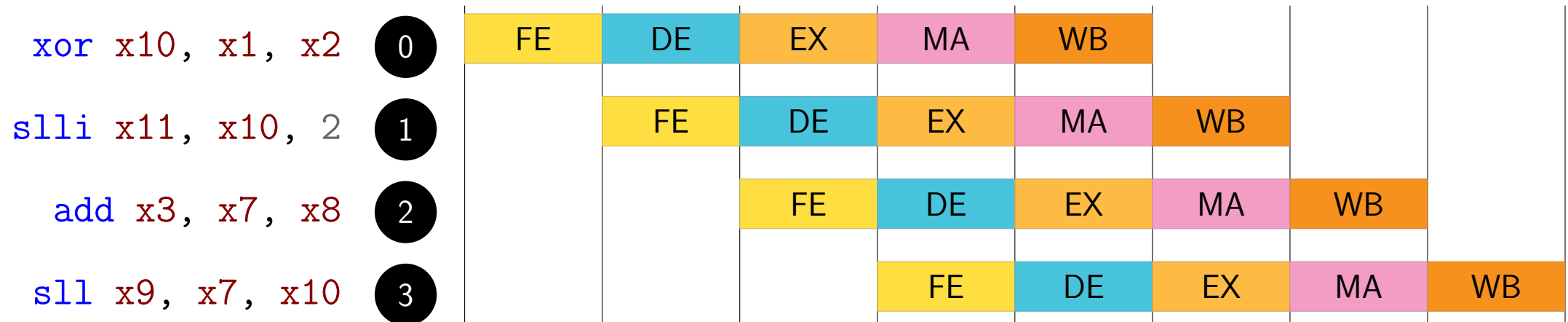


Pipeline Forwarding

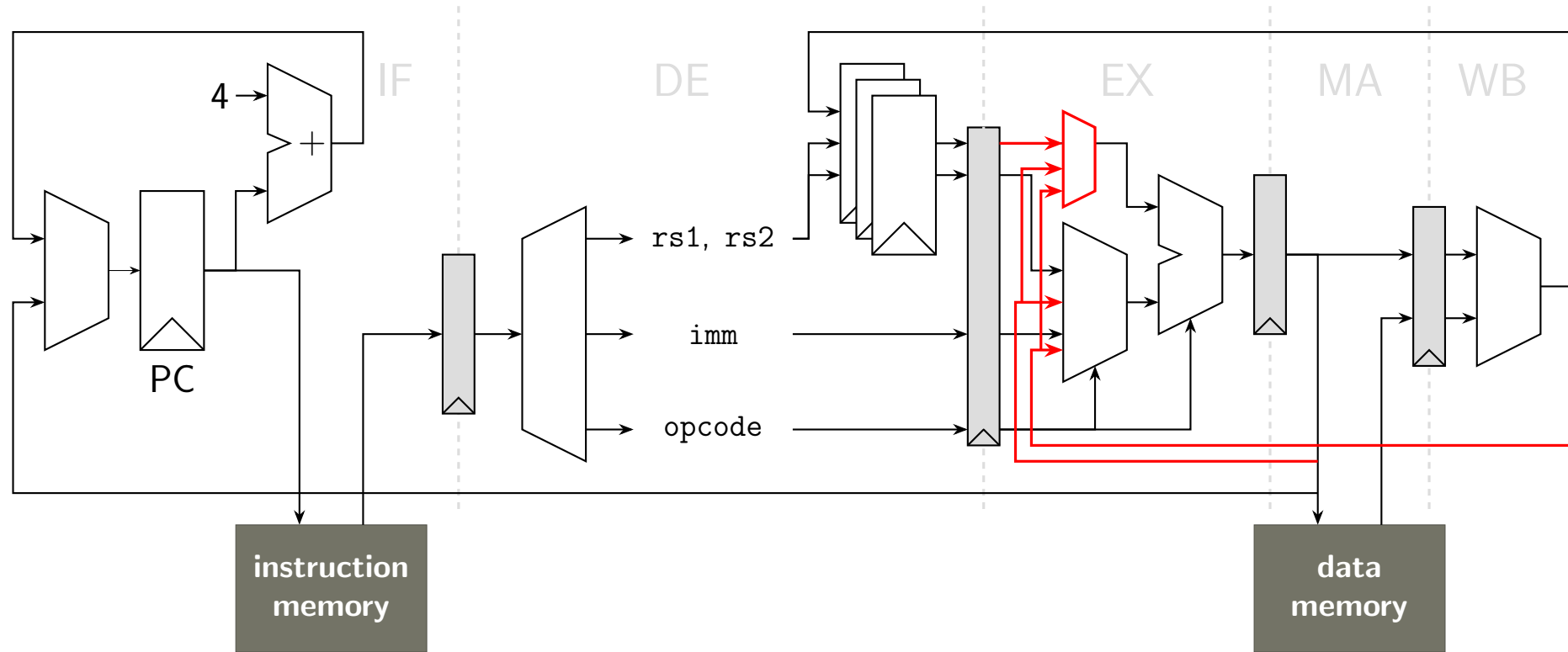


Reordering instructions has limitations

Pipeline Forwarding: Make result available to earlier stages before writeback



Pipeline Forwarding (simplified)





Data Hazard (continued)

Are we fine now?

```
lw x4, 8(x2)
```

```
addi x5, x4, 3
```

```
sll x6, x4, x5
```

```
andi x5, x6, 3
```

```
sw x5, 4(x2)
```

There are other data dependencies



Background: Variables and Registers

Compiler generates intermediate representation: instructions from code, variables as symbols

Example: Static Single Assignment Form (*versions* of variables)

```
int f(int *a, int b) {  
    return a[0]+a[1]+a[2]-b;  
}
```

```
define i32 @f(i32* %0, i32 %1) #0 {  
    %3 = load i32, i32* %0, align 4  
    %4 = getelementptr inbounds i32, i32* %0, i64 1  
    %5 = load i32, i32* %4, align 4  
    %6 = getelementptr inbounds i32, i32* %0, i64 2  
    %7 = load i32, i32* %6, align 4  
    %8 = sub i32 %3, %1  
    %9 = add i32 %8, %5  
    %10 = add i32 %9, %7  
    ret i32 %10  
}
```

At this point we only have the data dependencies from above



Background: Register Allocation

Compiler generates machine code, unbound number of symbols must be mapped to registers

```
define i32 @f(i32* %0, i32 %1) #0 {  
    %3 = load i32, i32* %0, align 4  
    %4 = getelementptr inbounds i32, i32* %0, i64 1  
    %5 = load i32, i32* %4, align 4  
    %6 = getelementptr inbounds i32, i32* %0, i64 2  
    %7 = load i32, i32* %6, align 4  
    %8 = sub i32 %3, %1  
    %9 = add i32 %8, %5  
    %10 = add i32 %9, %7  
    ret i32 %10  
}  
  
f:  
    lw    a2, 0(a0)  
    lw    a3, 4(a0)  
    lw    a0, 8(a0)  
    sub   a1, a2, a1  
    add   a1, a1, a3  
    addw  a0, a0, a1  
    ret
```

Problem: Registers are scarce (RISC-V 31, minus ABI registers)



Data Hazards

Limited availability of registers forces compiler to reuse registers, limits CPU optimizations

Read-After-Write dependency, also *True Dependency*

- Dependency from before
- Mostly eliminated by forwarding

```
addi x3, x1, 3  
slli x4, x2, x3
```

Write-After-Read, also *Anti-Dependency*

- Register is used for another symbol versions
- Reordering would eliminate the required value

```
addi x3, x4, 3  
lw x4, 0(x2)
```

Write-After-Write, also *Output Dependency*

- Register is used for another symbol versions
- Reordering would switch values

```
slli x3, x2, 8  
addi x3, x5, 7
```

Read-after-Read is not a hazard

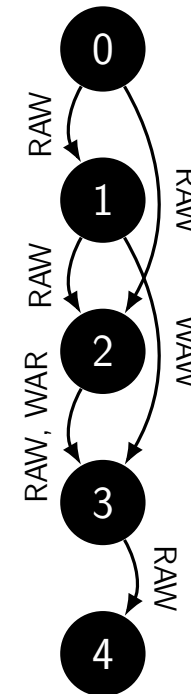
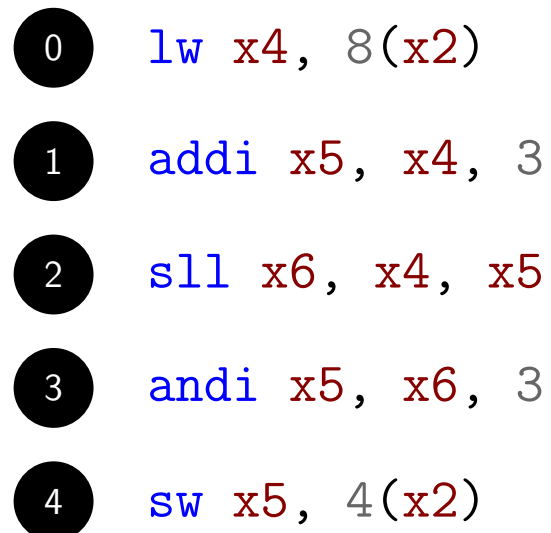


Data Flow Graph

Data flow graph can be used to identify instruction level parallelism

Graph of data dependencies

- Each instruction is a *vertex*
- Each dependency is an *edge*



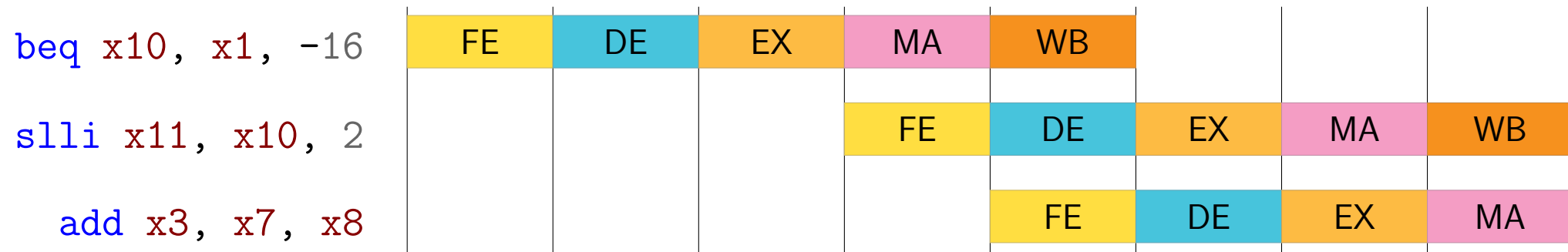
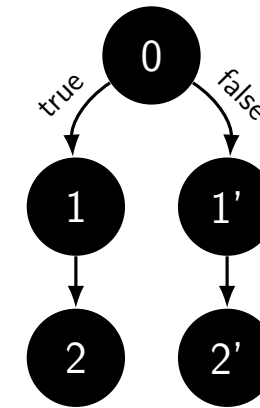


Control hazards

So far: data dependencies, instructions are sequential

But they are not sequential:

- Branch: Two possible "paths" in control flow
- Which instruction to fetch next?
- Decision depends on EX stage





Influence of Hazards on IPC

Recap: IPC of 1 is ideal

RAW: Fundamental impact

- Example: **Every fourth** instruction depends on result, penalty: **2 cycles**

$$\text{CPI} = 1 + \frac{1}{4} \cdot 2 \Rightarrow \text{IPC} = \frac{1}{\text{CPI}} = \frac{1}{1 + \frac{1}{4} \cdot 2} = \frac{2}{3}$$

- Nearly entirely solved by forwarding

WAW and WAR: Result from register allocation

- For simple pipeline they are not important, but limit optimizations (see part 4)

Control Hazards: Many branches, impact considerably height

- Waiting for decision is penalty, can we guess it? (remain of this part)





Pipeline Optimizations

Goal: Bring IPC up (near to one or even above)

Speculative Execution

- Branch prediction: Reduce the impact of branch decisions
- Other kinds of speculation: Address, data, ...

Parallelism

- Instruction Level Parallelism (ILP)
 - ▶ Pipelining
 - ▶ Superscalar execution, out-of-order execution (lecture part 4)
- Data parallelism
 - ▶ Data vectors, single instruction multiple data
- Thread parallelism
 - ▶ Execution of multiple different instruction streams





Branch Prediction: Motivation

Branches are problematic for pipelining

- Decision delayed until EX stage
- Stall pipeline until decision made → IPC goes down

Branch prediction:

- Execute one of the paths *speculatively*
- Withdraw execution if decision is different to speculation

Impact on IPC:

- $IPC=1$ if we always select the right path
- $IPC<1$ if we select wrong path, misprediction penalty

Problem: *Which path to predict?*

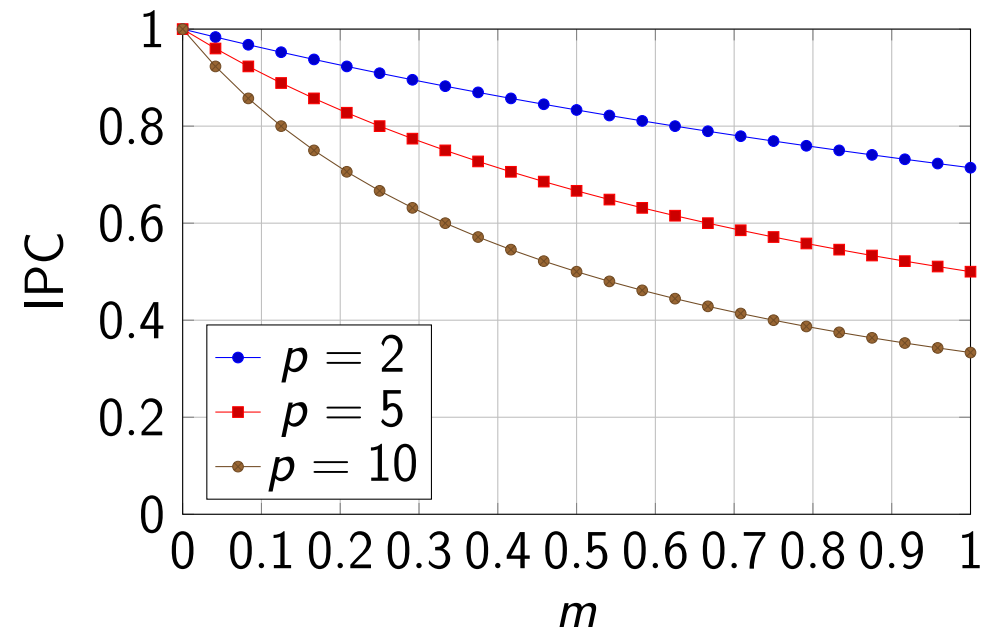


Branch Prediction and the IPC

Parameters

- b : Branch rate (relative number of branch instructions)
- m : Misprediction rate (how many of branches are wrongly predicted)
- p : Penalty for mispredicts (extra cycles to flush pipeline)

$$\text{IPC} = \frac{1}{1 + b \cdot m \cdot p}$$





Reduce the Impact of Branches

Avoid Branches

Replace branch instructions with other instructions

Predict Branches

- The deeper the pipeline, the more expensive (lower IPC) mispredicts become
- Increasing the rate of correct predictions has significant impact
- Two types of branch prediction
 - ▶ *Static branch prediction*: Only use information at hand
 - ▶ *Dynamich branch prediction*: Keep book about previous decisions





Avoid Branches

Conditional instructions

- Instructions that are only executed based on flag, see part 2
- Not in RISC-V, example ARM:

```
cmp    r1, r2
subgt  r1, r1, r2
```

Modify code, example **loop unrolling**

- Rewrite loops in repeating code sequences
- Reduces number of branches, but increases code size
- This is most often done by the compiler (inner loops, few iterations)



Loop Unrolling

Example: 3 nested loops

```
for (int i=0; i<3; i++)  
    for (int j=0; j<3; j++)  
        for (int k=0; k<3; k++)  
            Z[i][j] += X[i][k] * Y[k][j];
```

Loop unrolling of most inner loop:

```
for (int i=0; i<3; i++)  
    for (int j=0; j<3; j++)  
        Z[i][j] += X[i][0] * X[0][j] + X[i][1] * X[0][1]  
                + X[i][2] * X[2][j];
```



Static Branch Prediction

Predict if branch is taken or not solely based on the instruction

User-controlled

- Use a bit in opcode to indicate if branch is probable
- Generally useful for loop counters
- Examples in: PowerPC, Alpha, MMIX
- Not in RISC-V: Coding space is too precious and the result is not better than what hardware-based branch prediction (remain of this part) can achieve

Machine decision

- Predict based on inspection of the instruction





Static Branch Prediction: Always

Observation: Probability of branch taken is 60-70%

Idea: Always predict the jump

Assumptions:

- 20% of instructions are branches
- 70% of branches are taken
- Misprediction penalty: 5 cycles (realistic for 11-15 stage pipeline)

What is the IPC?

$$\text{IPC} = \frac{1}{1 + 0.2 \cdot (1 - 0.7) \cdot 5} = \frac{1}{1.3} = 0.77$$



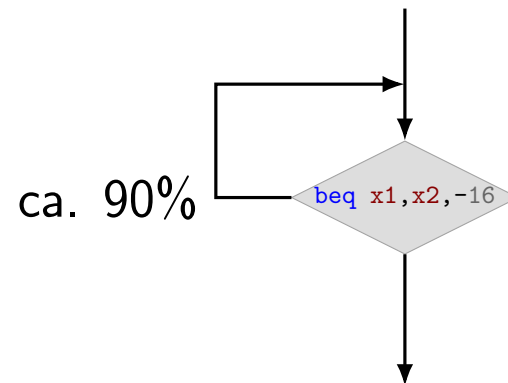
Static Branch Prediction: Direction



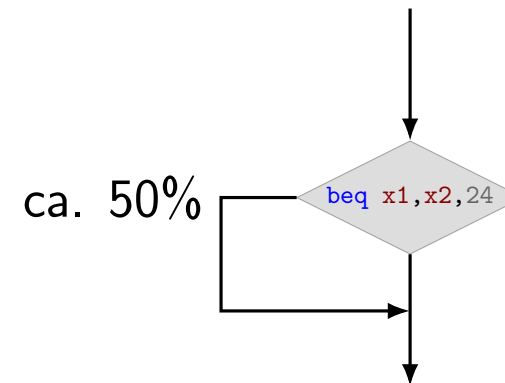
Look closer at branches

Observation: Differences by branch direction

Backward branch



Forward branch





Static Branch Prediction with Direction: IPC

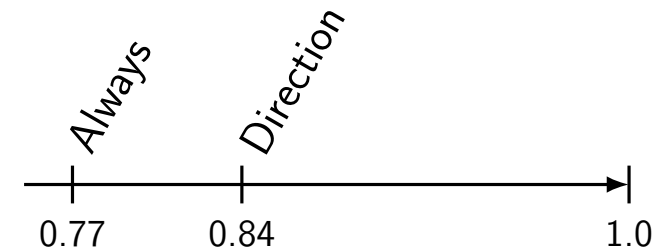
Assumptions:

- 80% of branches are backwards branches
- Branch taken 90%/50% for backward/forward branch

Impact on IPC:

$$\text{CPI} = 1 + 0.2 \cdot (0.8 \cdot (1 - 0.9) + 0.2 \cdot (1 - 0.5)) \cdot 5 = 1,18$$

$$\text{IPC} = \frac{1}{\text{CPI}} = 0,84$$





Dynamic Branch Prediction

Observation: Static branch prediction works well, but not for forward branch

Approach: Branch prediction depends on *history*

Based on **correlations**

- *Temporal correlation*

If a branch was taken recently, it will probably be taken again (loops, etc.)

- *Spatial correlation*

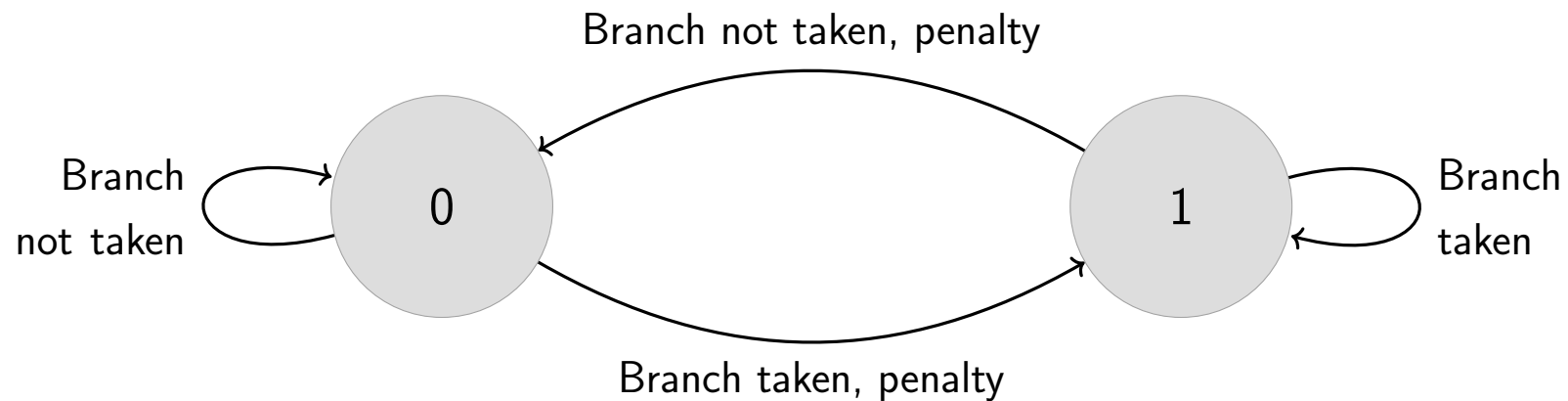
Branches on an execution path will probably behave similarly with each execution of the path



Dynamische Branch Prediction: 1-bit Predictor

Idea: Consider last branch decision

1 bit counter/state machine





1-bit Predictor: Example

PC							Predictor
[0x120]	<code>beqz x10, 24</code>	FE	DE	EX	MA	WB	0
[0x124]	<code>slli x11, x10, 2</code>		FE	DE	EX	MA	0
[0x128]	<code>add x11, x11, x8</code>			FE	DE	EX	0
[0x12c]	<code>lw x12, 0(x11)</code>				FE	DE	0
	...						
[0x120]	<code>beqz x10, 24</code>	FE	DE	EX	MA	WB	0
[0x124]	<code>slli x11, x10, 2</code>		FE	DE			0
[0x128]	<code>add x11, x11, x8</code>			FE			0
[0x148]	<code>sll x9, x7, x10</code>				FE	DE	1



1-bit Predictor: Multiple Branches

Single bit to track all branches

Problem: Multiple branches

- Nested loop, control structures, function calls
- Share the same predictor and mispredicts

Ideal solution: One predictor per branch

- No interference, exclusive resource
- **but:** Need as many predictors as potential branches

Real solution: Use multiple predictors



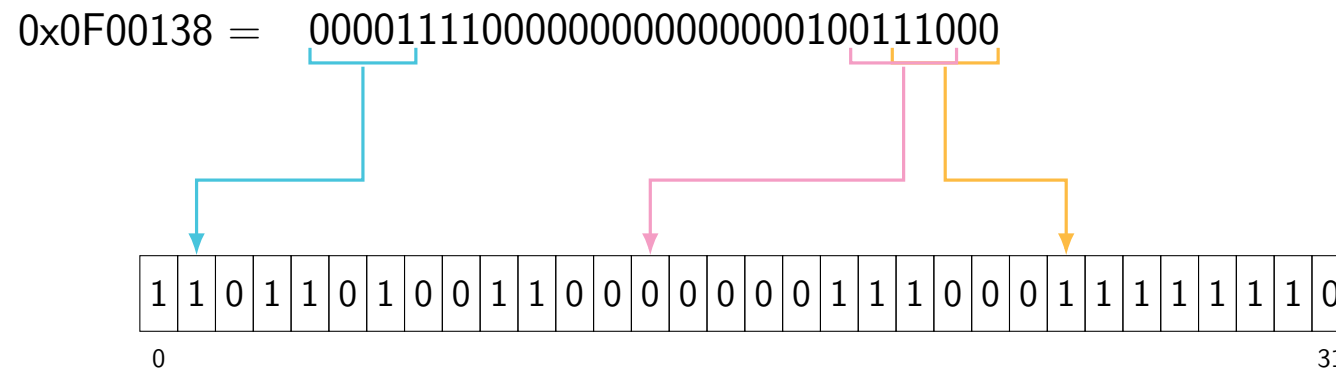


Multiple 1-bit Predictors

Selection of multiple predictors based on program counter

Which portion of program counter?

- **Most significant bits** are problematic: aliasing of adjacent branches
- **Least significant bits** are problematic: 3 out of 4 predictors never addressed
- **Least significant, non-static bits**: adjacent branches map to different predictors



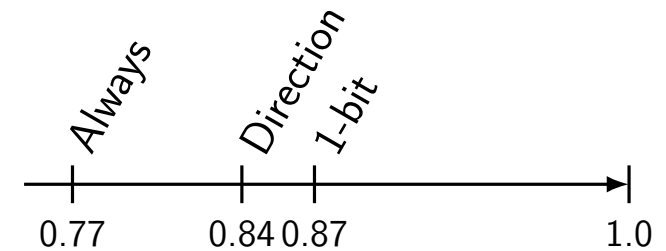


1-bit Predictor: IPC

Observation: 85% accuracy

$$\text{CPI} = 1 + 0.2 \cdot (1 - 0.85) \cdot 5 = 1.15$$

$$\text{IPC} = \frac{1}{\text{CPI}} = \frac{1}{1.15} = 0.87$$





1-bit Predictor: Limitations

Consider branch taken over time for a particular branch (1/0 branch taken/not taken)

Example: Inner loop

- Branch decisions of *y* loop (take loop again):

111011101110...

- Predicted: 111101110111...

- Mispredicts: 000110011001...

Outliers lead to double mispredict

How can we suppress this behavior?

```
for (x = 1024; x > 0; x--)  
    for (y = 4; y > 0; y--)  
        do_something(x,y);
```

is compiled to:

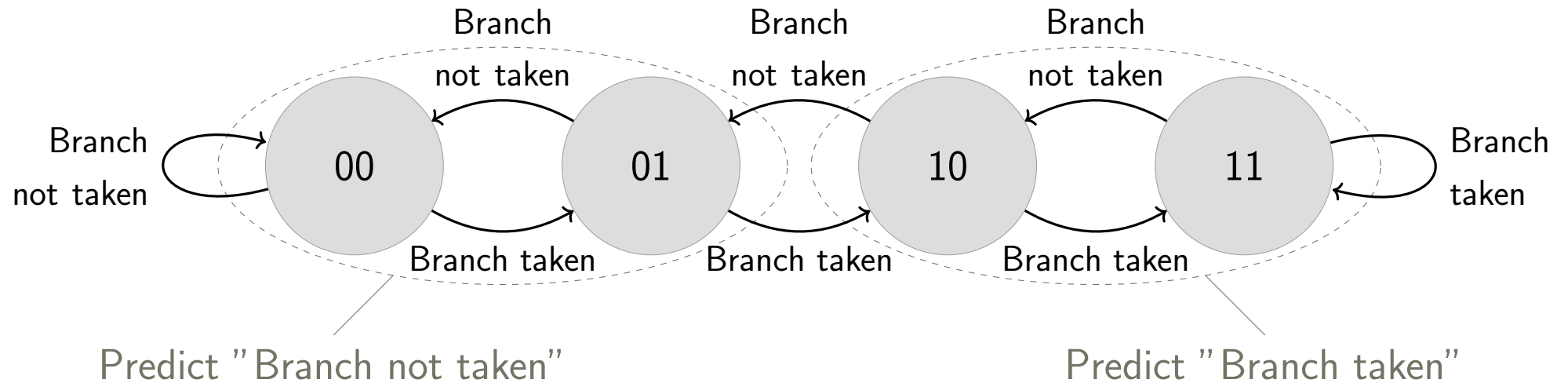
```
        li s0, 1024  
xloop: li s1, 4  
yloop: mv a0, s0  
        mv a1, s1  
        jal ra, do_something  
        addi s1, s1, -1  
        bnez s1, yloop  
        addi s0, s0, -1  
        bnez s0, xloop
```



2-bit Predictor

Approach: Make robust against "outliers" (filter)

- *Saturating 2-bit counter*



When do we get a penalty?

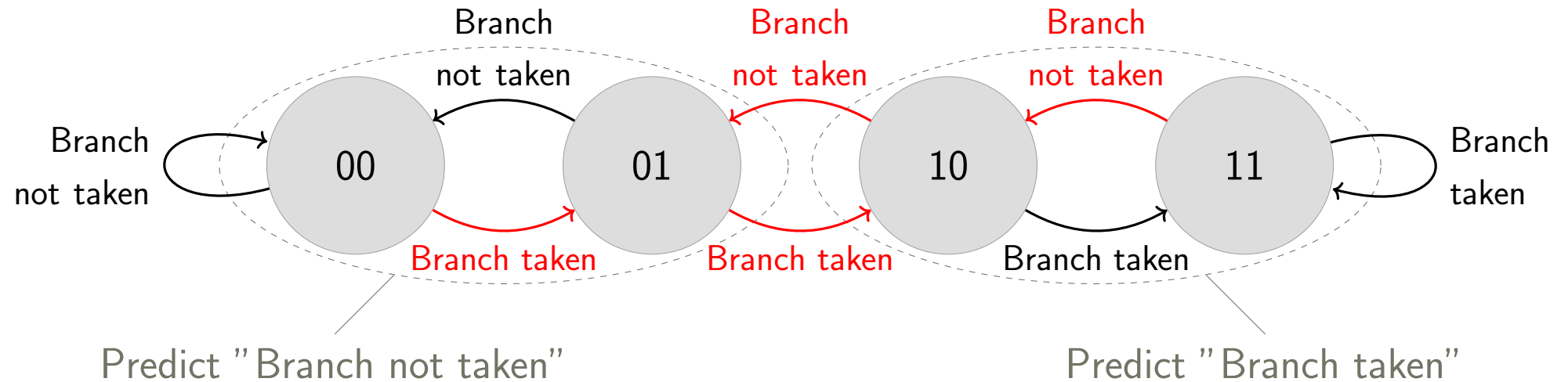




2-bit Predictor

Approach: Make robust against "outliers" (filter)

- *Saturating 2-bit counter*



When do we get a penalty?



2-bit Predictor: Improvement



Comparison to 1-bit predictor, example with nested loop

Branch taken? 1 1 1 0 1 1 1 0 1 ...

1-bit
state 0 1 1 1 0 1 1 1 0 1
predict 0⚡ 1 1 1⚡ 0⚡ 1 1 1⚡ 0⚡

2-bit
state 01 10 11 11 10 11 11 11 10 11
predict 0⚡ 1 1 1⚡ 1 1 1 1⚡ 1

```
for (x = 1024; x > 0; x--)  
  for (y = 4; y > 0; y--)  
    do_something(x,y);
```

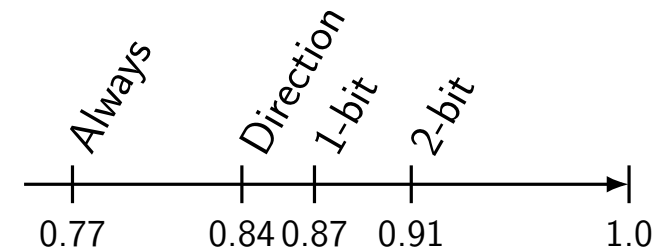
2-bit Predictor: IPC



Observation: 90% accuracy

$$\text{CPI} = 1 + 0.2 \cdot (1 - 0.9) \cdot 5 = 1.1$$

$$\text{IPC} = \frac{1}{\text{CPI}} = \frac{1}{1.1} = 0.91$$





2-bit Predictor: Limits

Still penalty on regular patterns:

- Recap: Nested loop iterations: 111011101110...
- Branches often show such regular patterns

Can we incorporate this regularity?





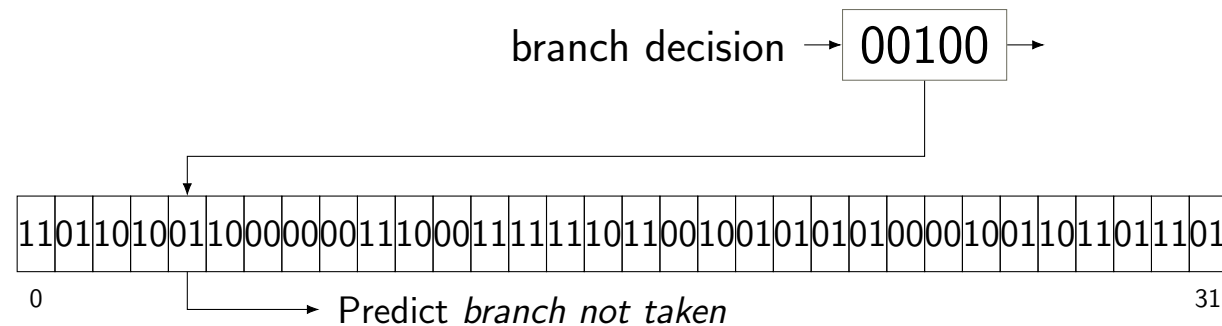
2-Way Adaptive Predictor

Save the last branch decisions

Select predictor based on history

- Before: Selection based on PC
- Now: Use history of most recent branch decisions

The *actual predictor* ("2nd way") stays the same (for example 2-bit predictor)





Adaptive Predictor: Example

5 bit of history (init: 00000), 32 Predictors

2-bit predictors (init: 01)

```

        li s0, 1024
xloop: li s1, 4
yloop: mv a0, s0
        mv a1, s1
        jal ra, do_something
        addi s1, s1, -1
branch y → bnez s1, yloop
        addi s0, s0, -1
branch x → bnez s0, xloop
    
```

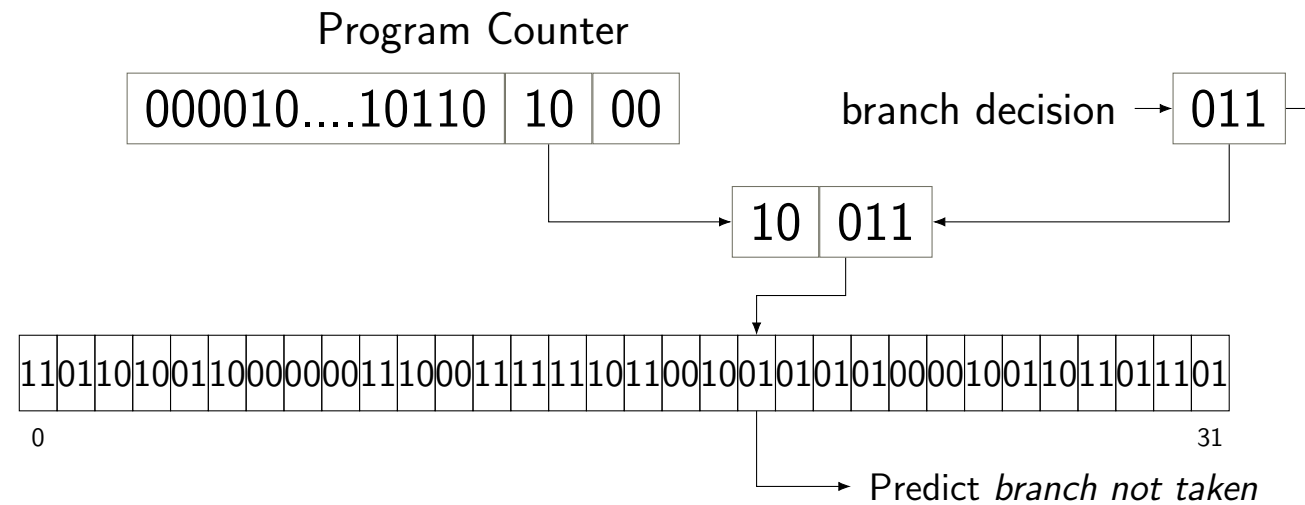
		History	0	0	0	0	0	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	
			0	0	0	0	1	1	1	0	1	1	1	1	0	1	1	1	1	1	0	1	1
Branch		line	y	y	y	y	x	y	y	y	y	x	y	y	y	y	x	y	y	y	y	x	
		taken	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	1	1	0	1
Predictor		id	0	1	3	7	14	29	27	23	15	30	29	27	23	15	30	29	27	23	15	30	
		state	01	01	01	01	01	01	01	01	01	10	10	10	10	00	10	11	11	11	11	00	11
		predict	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	1	1	0	1
		new state	10	10	10	00	10	10	10	10	10	00	10	11	11	11	00	11	11	11	11	11	00



2-Way Adaptive Global Predictor



Avoid aliasing with adding part of program counter to selection



Are we good now?



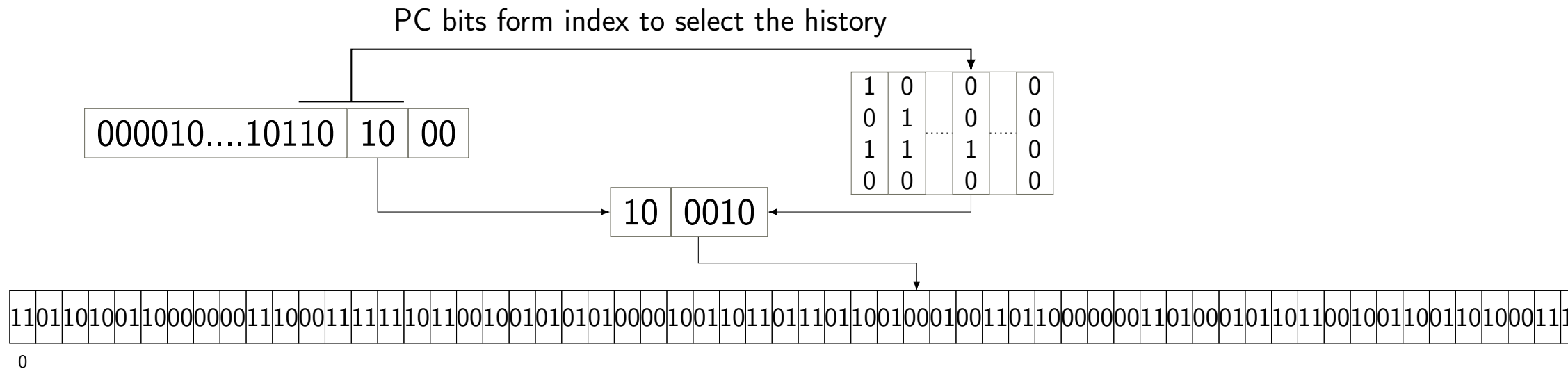


2-Way Adaptive Local Predictor

Problem with adaptive global predictor: Branch interference

- Branches influence each other, for example: deeply nested loops, function calls

Branch History Table: Keep multiple histories for diversion based on PC

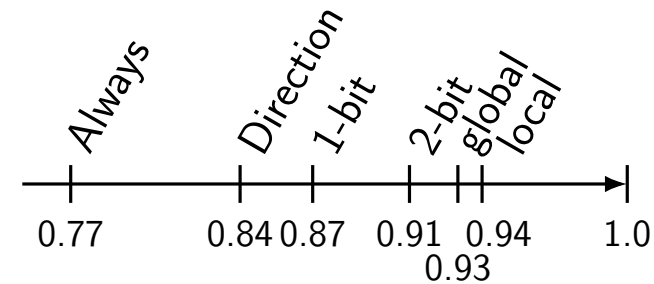




Adaptive Predictors: IPC

Observation:

- 93% accuracy for global predictor
- 94% accuracy for local predictor



This is 1990s technology, since then accuracy is up to about 99%

Modern CPUs incorporate neural network (perceptron-based) branch predictors



Branch Target Prediction

So far "branch taken" prediction, but also "branch target" needed

RISC-V: `jalr rd, imm(rs1)` instruction, content of `rs1` unknown

Branch Target Buffer

- Content addressable memory for lookups
- Store recent jump targets into table
- Replacement strategy to update table, evicts entries

Branch program counter	Last branch target
0x0400ab40	0x0400ab8c
0x04000804	0x0400aaf0
...	...
0x04000800	0x0400440c



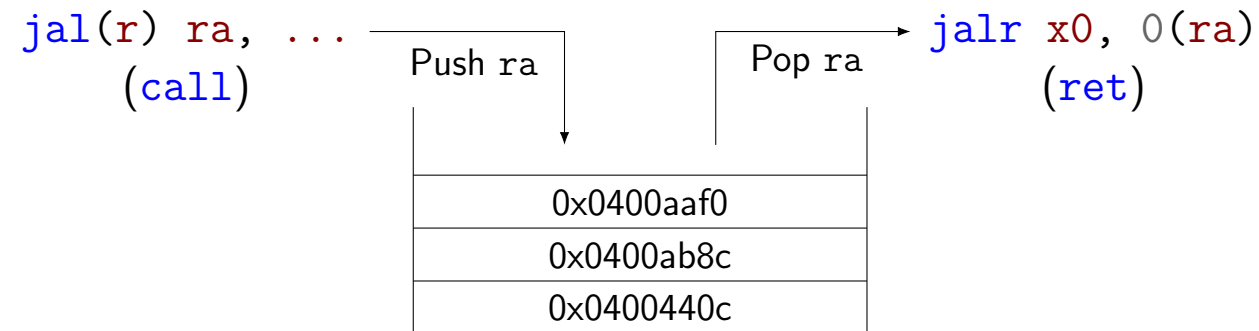
Return Address Stack

Problems with BTB:

- Expensive hardware (content addressable memory), limits entries
- Reduced gain for common function call patterns

Adding semantics: **Return Address Stack**

- `jalr` as part of function calls (see conventions)
- Idea: Store return address on separate hardware stack



Summary and Learnings



Pipelining is key to CPU performance

Hazards reduce the IPC

Pipeline optimizations based on speculative execution and parallelism

Speculative execution: Branch taken prediction and branch target prediction

Difference between predictors and predictor selection

