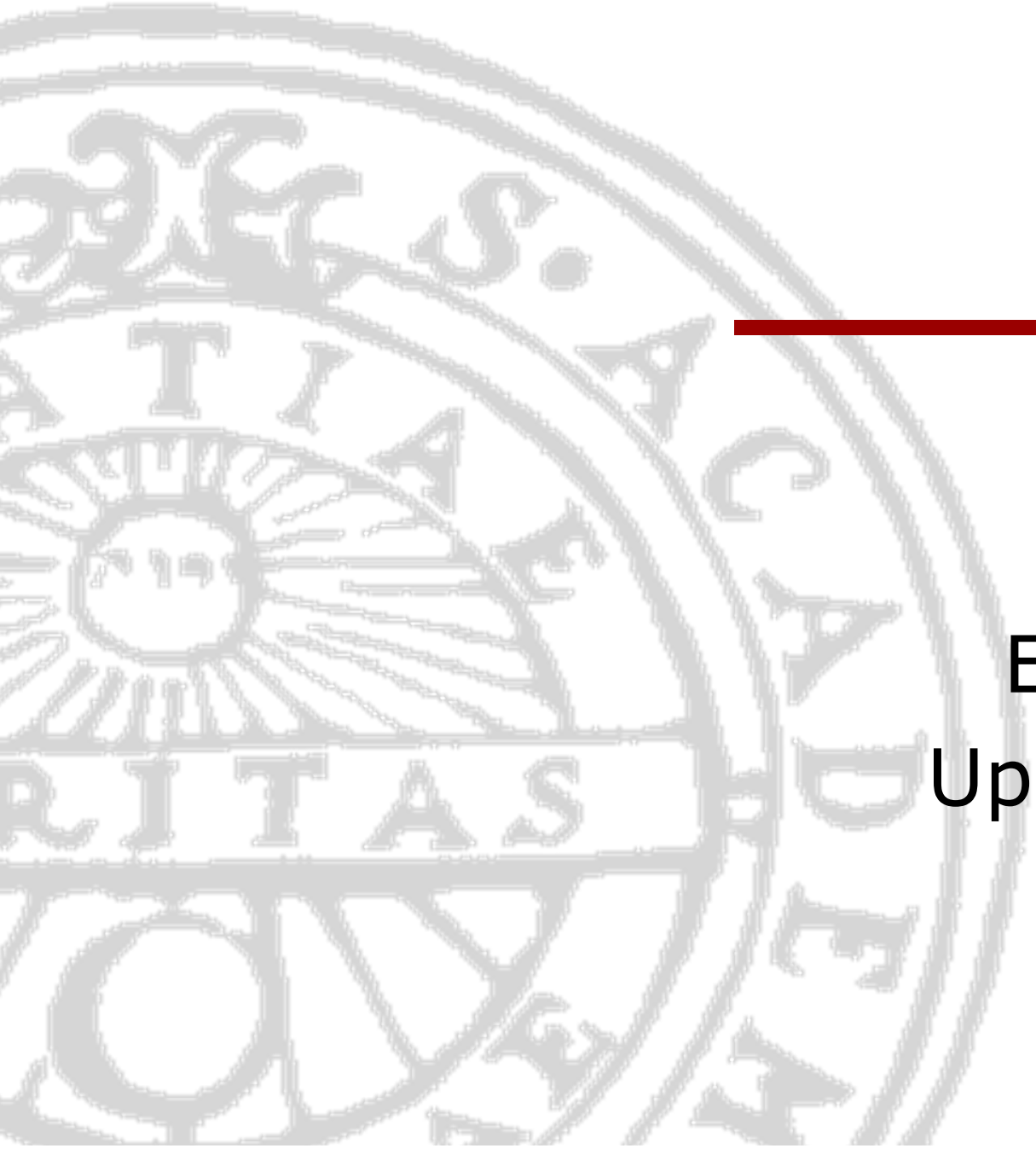
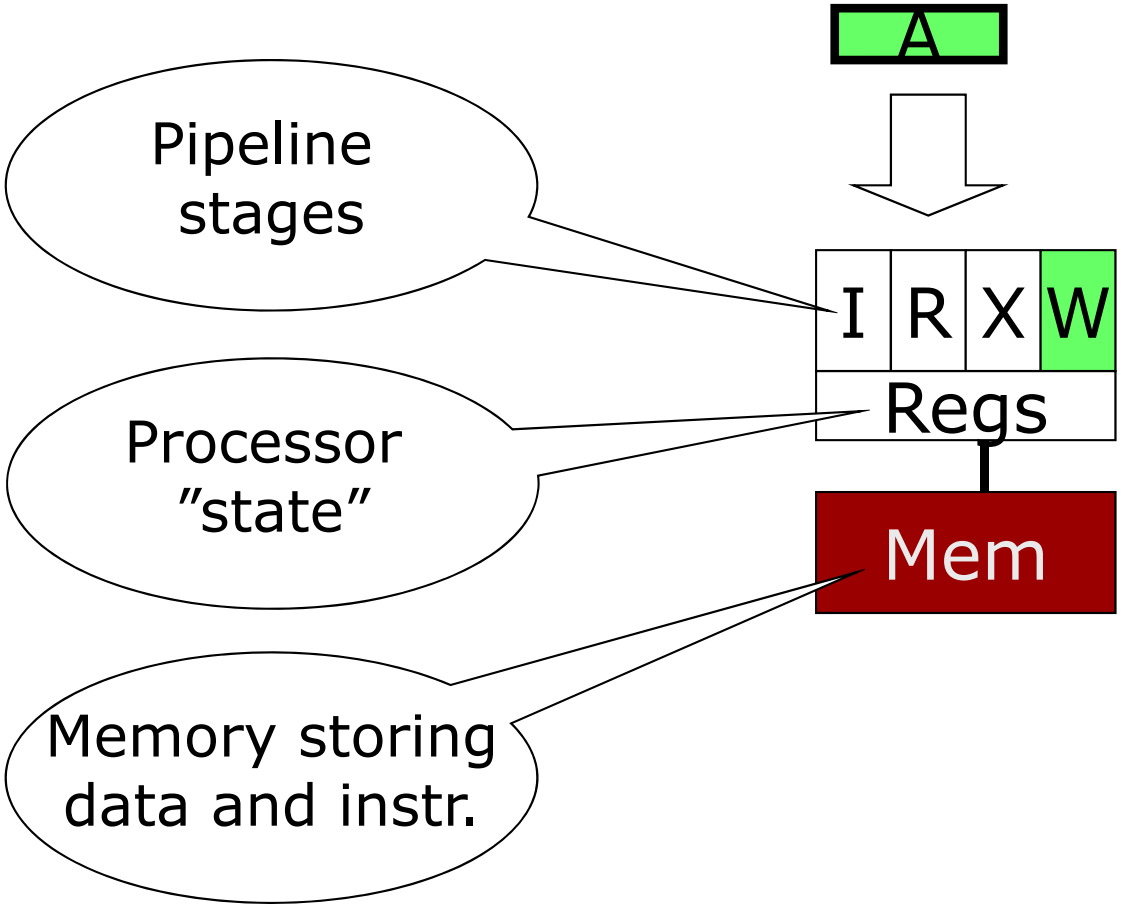


# Pipelines

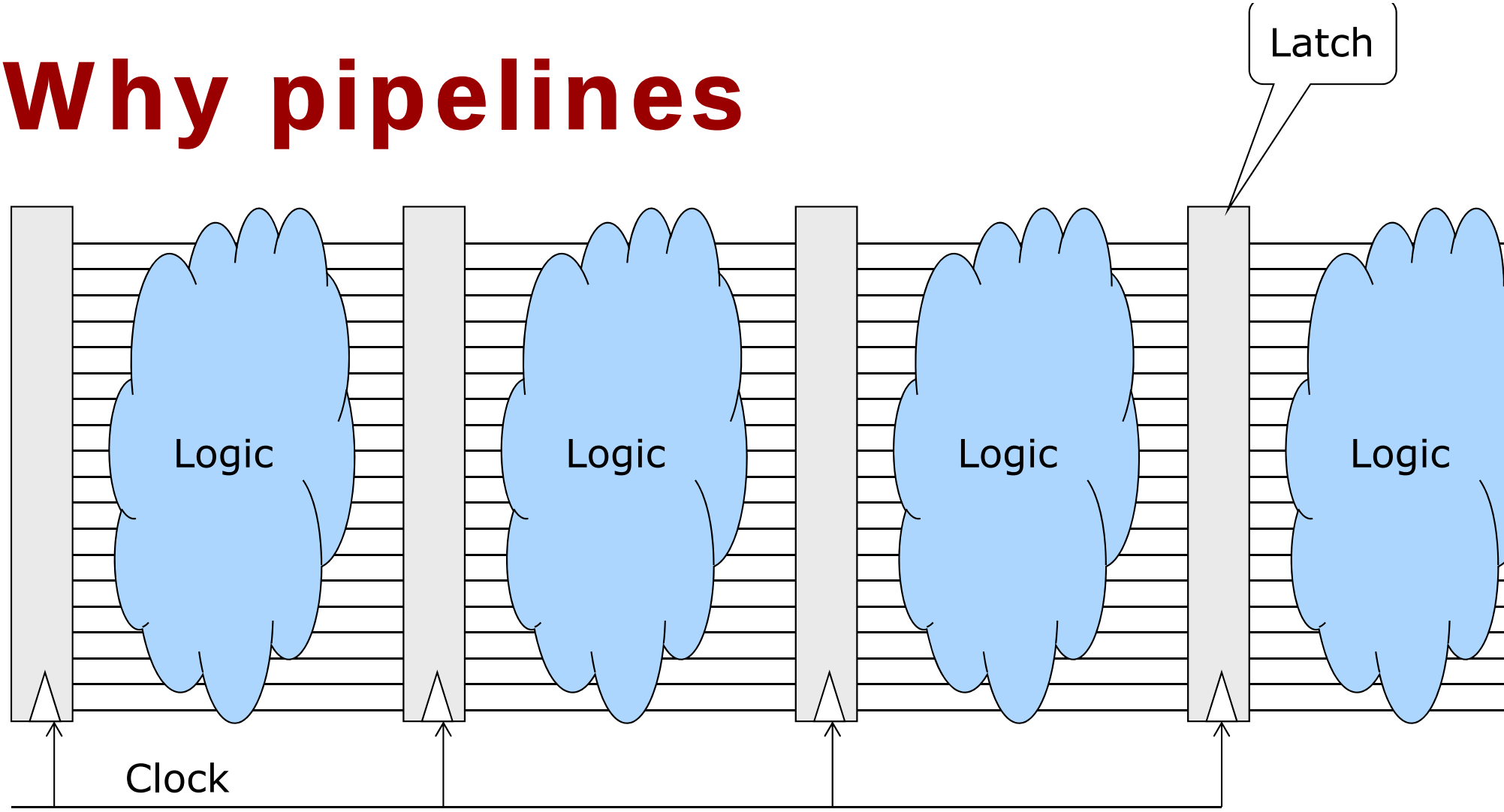
Erik Hagersten
Uppsala University

# Pipeline:




I = Instruction fetch
R = Read register
X = Execute
W= Write register/memory

# Why pipelines




**Example of pipeline limitations:**

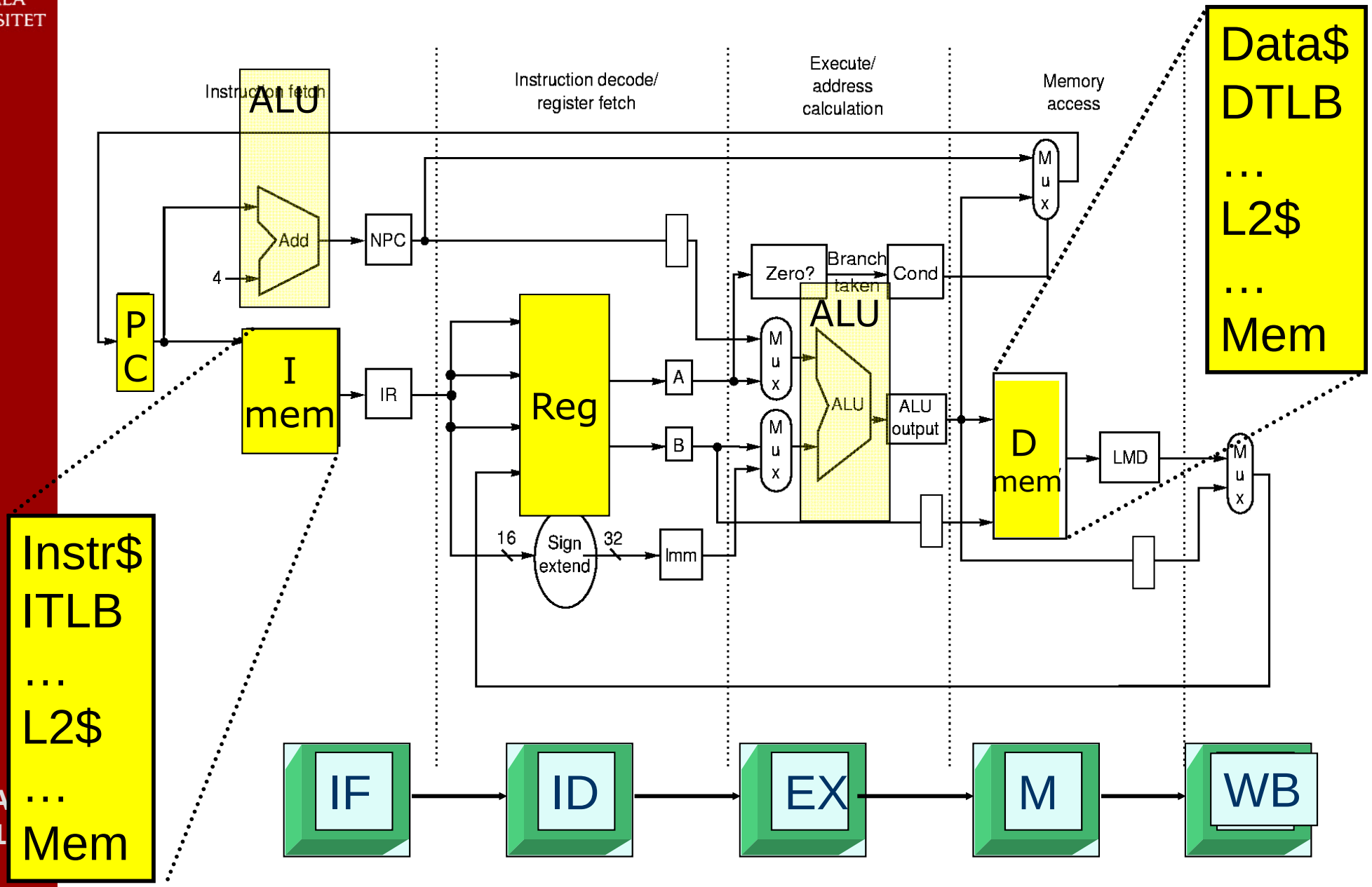
Set-up time: Minimum time between stable latch input signal and clock

Pileline delay: Latency from clock to stable output signal

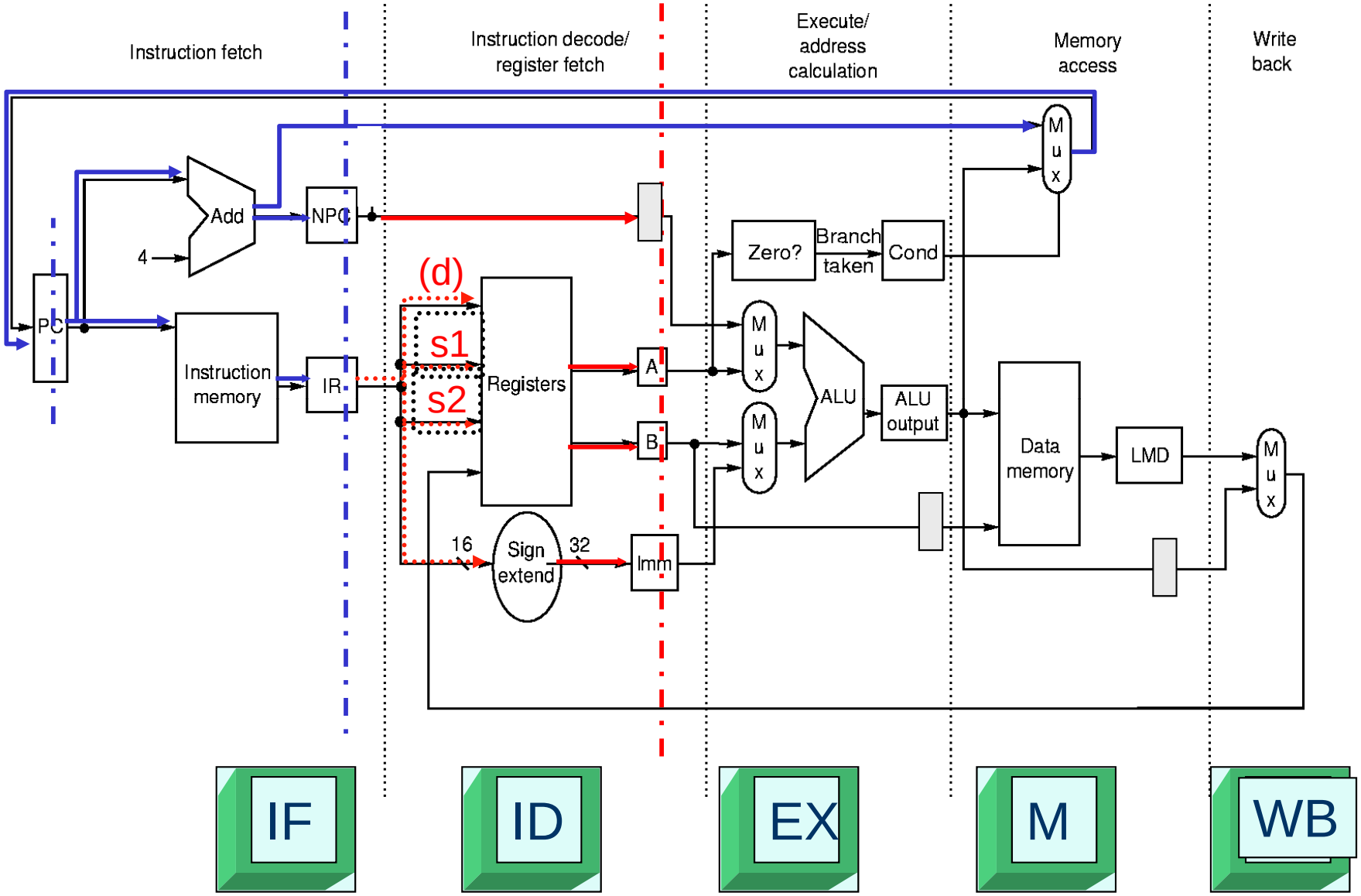
Clock-skew: The jitter between the clock signals arriving at the latches

Imbalance between the pipeline stages

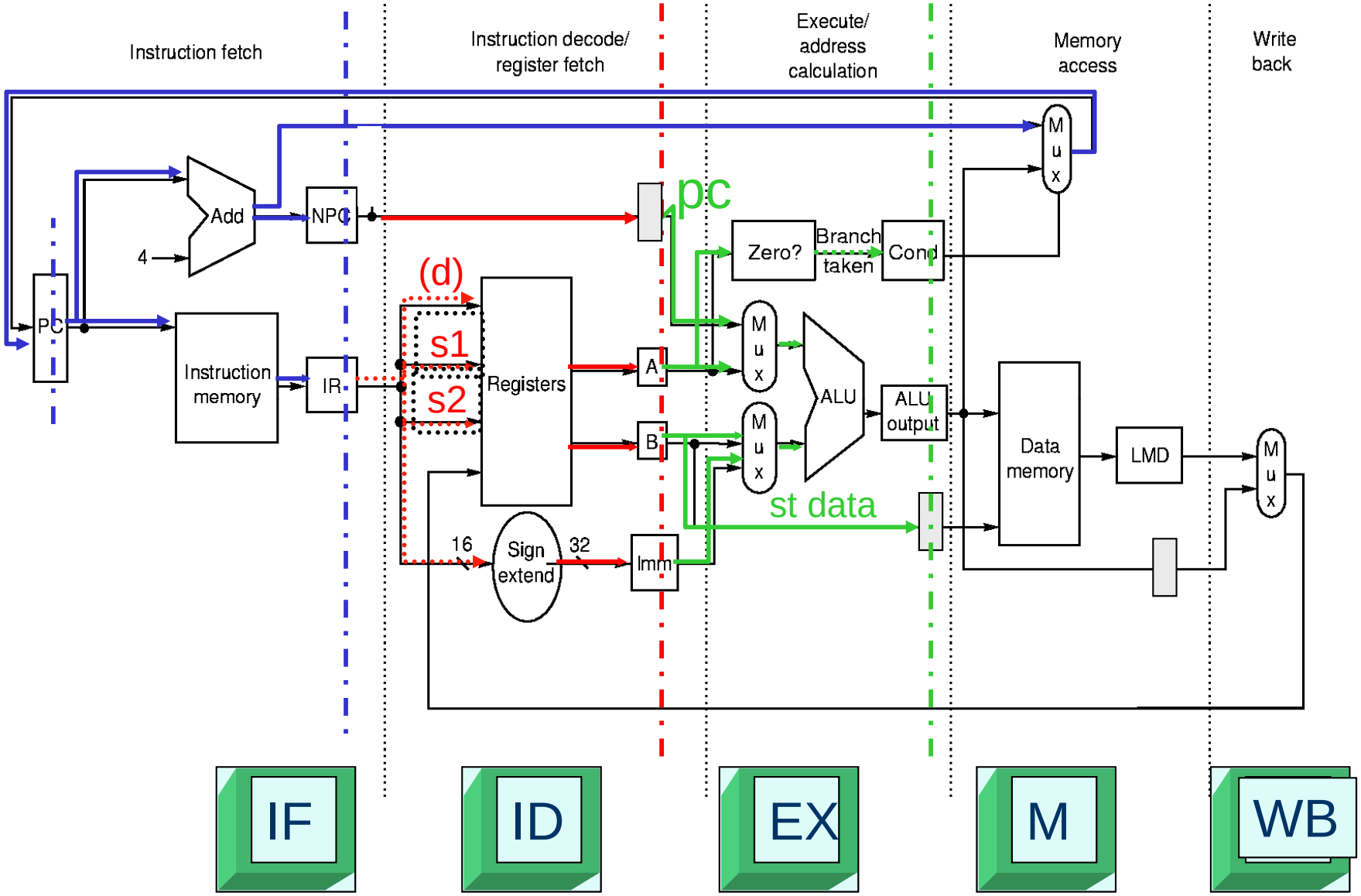
# Example: 5-stage pipeline



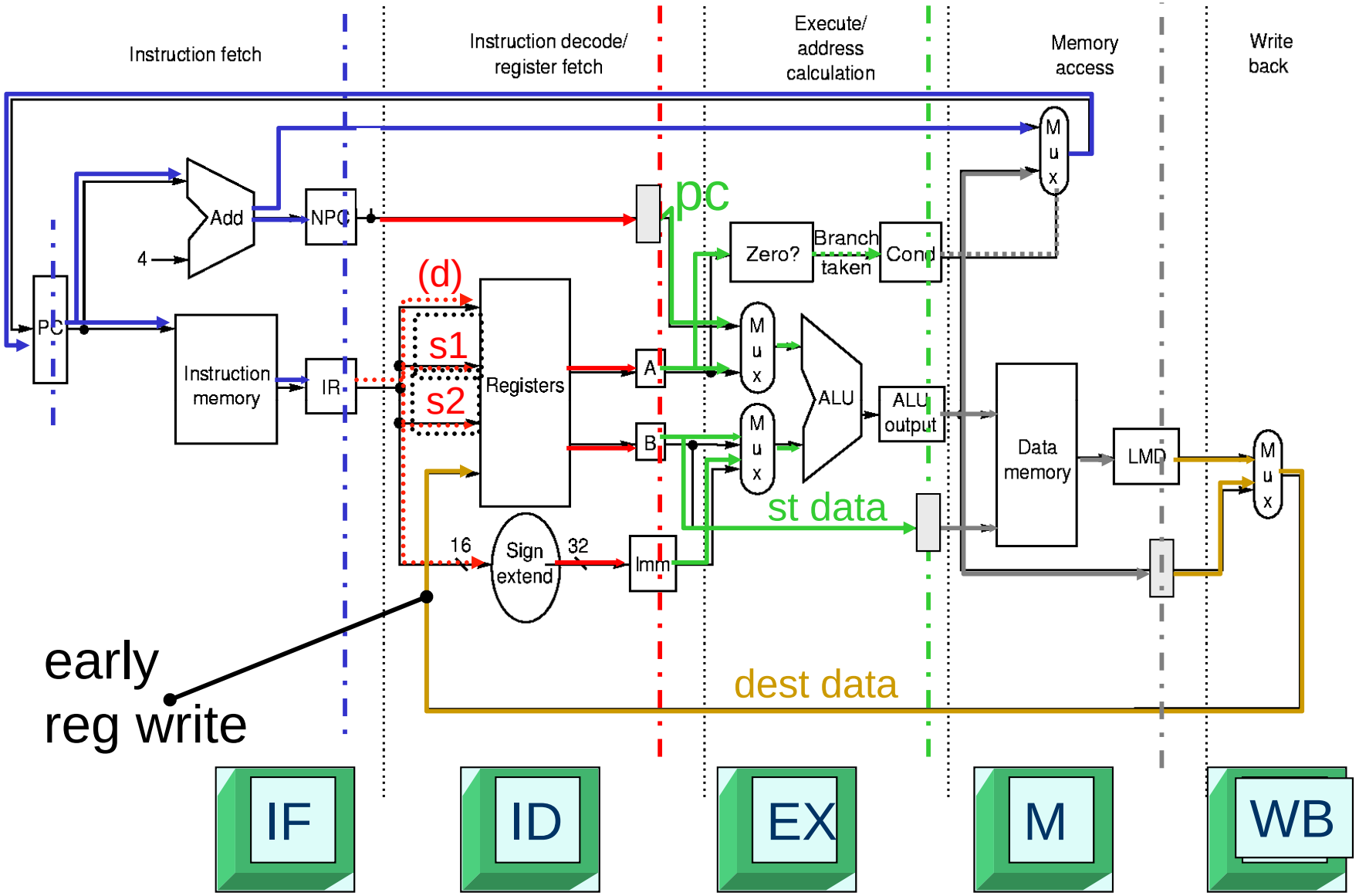
# Example: 5-stage pipeline



# Example: 5-stage pipeline



# Example: 5-stage pipeline



UPPSALA UNIVERSITET

# Fundamental limitations

Hazards prevent instructions from executing in parallel:

**Structural hazards: Simultaneous use of same resource**

**If unified I+D$: LW will conflict with later I-fetch**

**Data hazards: Data dependencies between instructions**

```
LW R1, 100(R2)   /* result avail in 2-150 cycles */
ADD R5, R1, R7
```

**Control hazards: Change in program flow**

```
BNEQ R1, #OFFSET
ADD R5, R2, R3
```

**Serialization of the execution by stalling the pipeline is one, although inefficient, way to avoid hazards**

UPPSALA UNIVERSITET

# Fundamental types of data hazards

Code sequence: $Op_i$ A

$Op_{i+1}$A

**RAW (Read-After-Write)**

Opi+1 reads A before Opi modifies A. Opi+1 reads old A!

**WAR (Write-After-Read)**

Opi+1 modifies A before Opi reads A.

Opi reads new A

**WAW (Write-After-Write)**

Opi+1 modifies A before Opi.

The value in A is the one written by Opi, i.e., an old A.

UPPSALA UNIVERSITET

# Scheduling example

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + 10;
```

```
loop:   LD      F0, 0(R1)       ; F0 = array element x[i]
                    RAW
        ADDD    F4, F0, F2      ; Scalar constant “10” in F2
                        RAW
        SD      0(R1), F4       ; Store result x[i]
        SUBI    R1, R1, #8      ; decrement array ptr.
        BNEZ    R1, loop        ; loop if R1 != 0
                                ; x[i] stored ”backwards”
```

*How many cycles to execute one loop?*

UPPSALA UNIVERSITET

# Scheduling in each loop iteration

Original loop

```
loop:   LD      F0, 0(R1)
        stall                   ; LD to FP ALU delay
        ADDD    F4, F0, F2
        stall                   ; FP ALU to SD delay
        stall                   ; FP ALU to SD delay
        SD      0(R1), F4
        SUBI    R1, R1, #8
        BNEZ    R1, loop
        stall                   ; delay slot
```

RAW

RAW

5 instructions + 4 bubbles = 9 cycles / iteration

*Can we reschedule instructions to avoid stalls?*
*Tip: Move loads up and stores down in the basic block.*

# Scheduling in each loop iteration

Original loop

```
loop:  LD     F0, 0(R1)
       stall
       ADDD   F4, F0, F2
       stall
       stall
       SD     0(R1), F4
       SUBI   R1, R1, #8
       BNEZ   R1, loop
       stall
```

Statically scheduled loop

```
loop:  LD     F0, 0(R1)
       stall
       ADDD   F4, F0, F2
       SUBI   R1, R1, #8
       BNEZ   R1, loop
       SD     8(R1), F4
```

Branch delay slot used!

SD is moved after SUBI #8
Have to compensate with an offset 8

5 instruction + 4 bubbles = 9c / iteration

5 instruction + 1 bubble = 6c / iteration

# Loop unrolling 4x

UPPSALA UNIVERSITET

```
<prologue>
loop:   LD      F0, 0(R1)
        stall
        ADDD    F4, F0, F2
        stall                       ; drop SUBI & BNEZ
        stall
        SD      0(R1), F4
        LD      F6, -8(R1)
        stall
        ADDD    F8, F6, F2
        stall                       ; drop SUBI & BNEZ
        stall
        SD      -8(R1), F8
        LD      F10, -16(R1)
        stall
        ADDD    F12, F10, F2
        stall                       ; drop SUBI & BNEZ
        stall
        SD      -16(R1), F12
        LD      F14, -24(R1)
        stall
        ADDD    F16, F14, F2
        SUBI    R1, R1, #32         ; alter to 4*8
        BNEZ    R1, loop
        SD      -24(R1), F16
<epilogoue>
```

24c/ 4 iterations = 6 c / iteration

AVDARK 2013

UPPSALA UNIVERSITET

# Optimized scheduled unrolled loop

```
loop:   LD      F0, 0(R1)
        LD      F6, -8(R1)
        LD      F10, -16(R1)
        LD      F14, -24(R1)
        ADDD    F4, F0, F2
        ADDD    F8, F6, F2
        ADDD    F12, F10, F2
        ADDD    F16, F14, F2
        SD      0(R1), F4
        SD      -8(R1), F8
        SD      -16(R1), F12
        SUBI    R1, R1, #32
        BNEZ    R1, loop
        SD      8(R1), F16
```

**Important tricks:**

Push loads up

Push stores down

Note: the displacement of the last store must be changed

**Benefits of loop unrolling:**

Provides a larger seq. instr. window (larger basic block)

Simplifies for static and dynamic methods to extract ILP

All penalties are eliminated! CPI=1,00

14 cycles / 4 iterations ==> 3.5 cycles / iteration

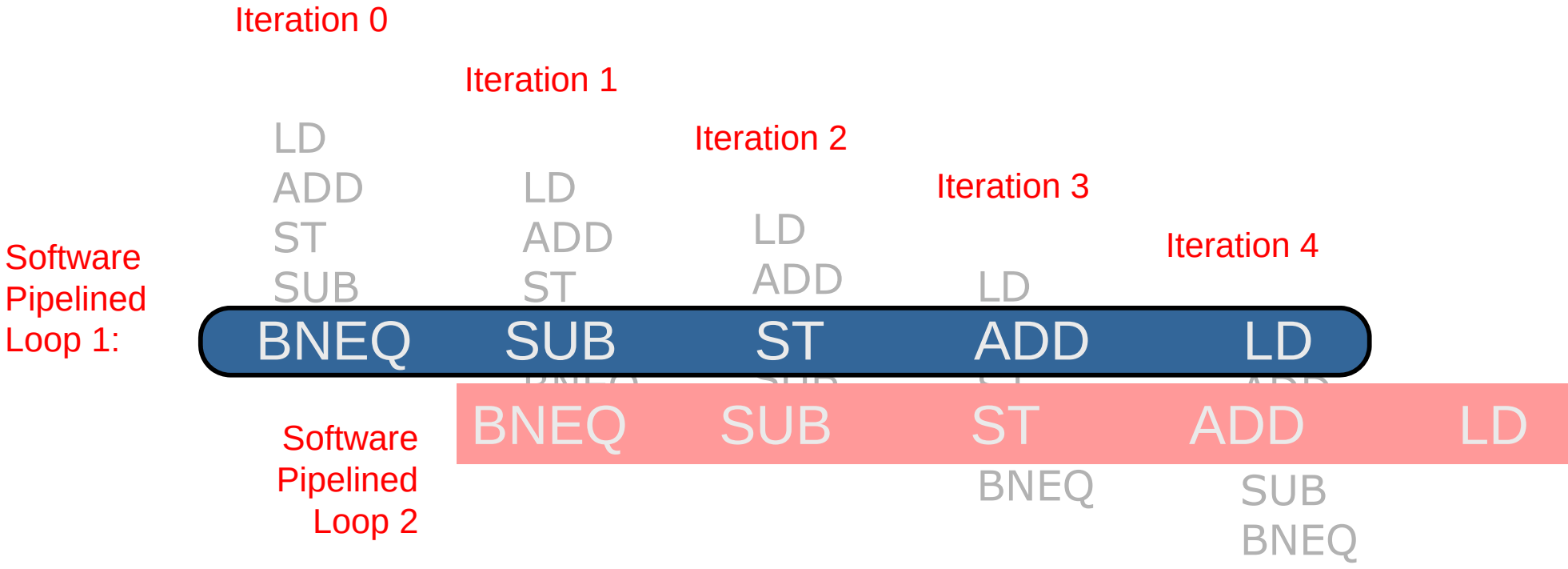
From 9c to 3.5c per iteration ==> speedup 2.6

# Software Pipelining

# Software pipelining 1(3)

## Symbolic loop unrolling

- Stagger the start of each iteration in the instruction scheduling

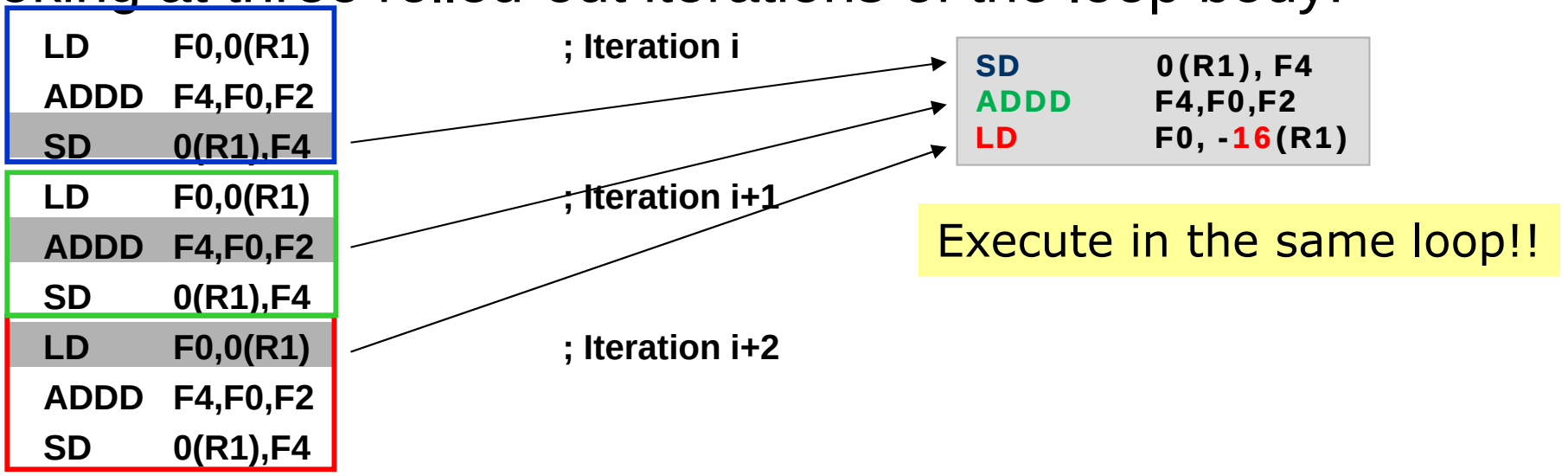


# Software pipelining 2(3)

Example:

```
loop:   LD    F0,0(R1)
        ADDD  F4,F0,F2
        SD    0(R1),F4
        SUBI  R1,R1,#8
        BNEZ  R1,loop
```

Looking at three rolled-out iterations of the loop body:

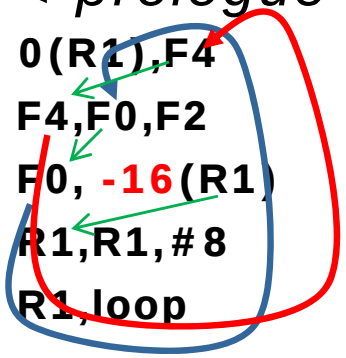



Execute in the same loop!!

# Software pipelining 3(3)

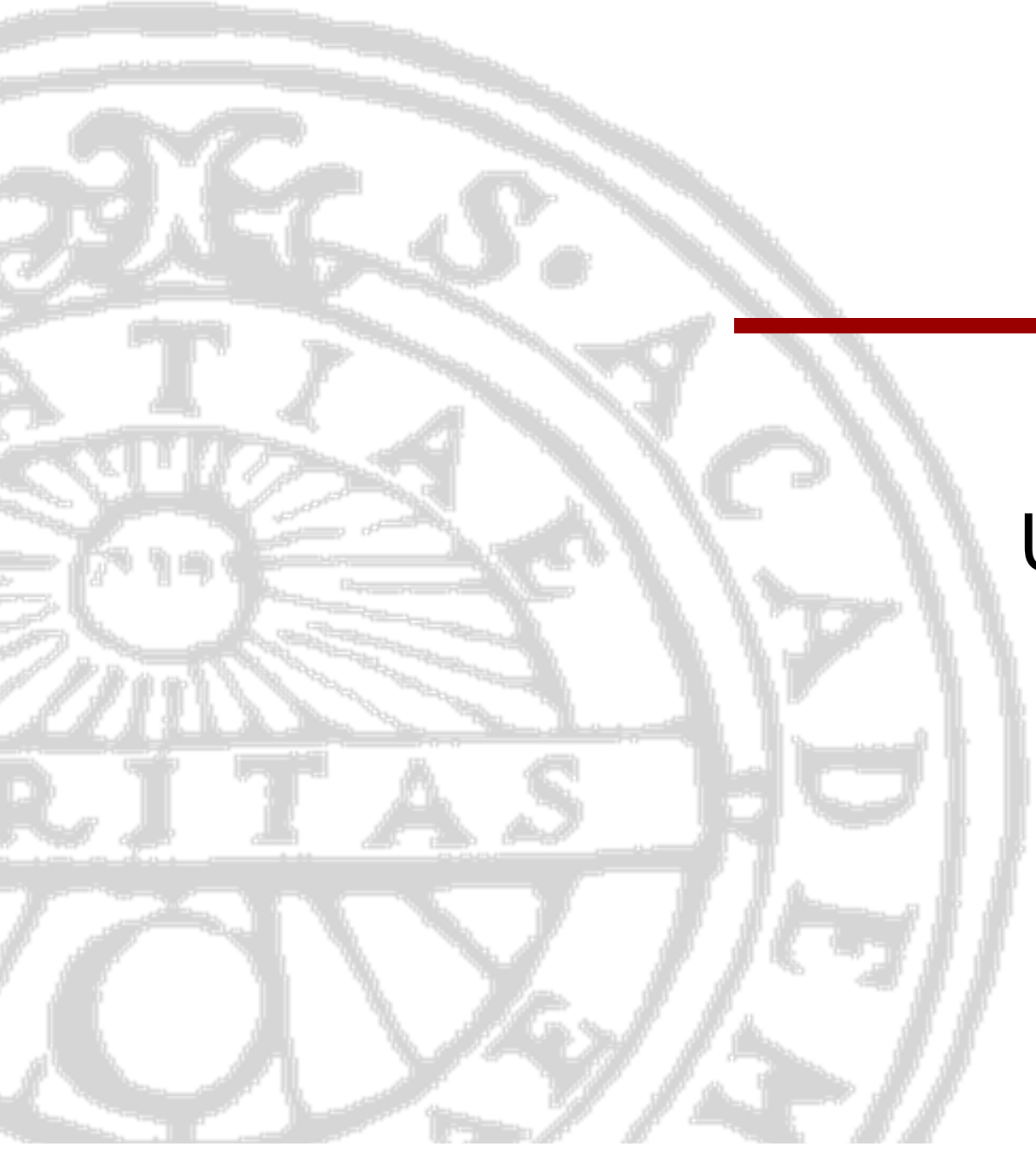
Instructions from three consecutive iterations form the loop body:

```
                < prologue code >
loop: 1.SD      0(R1),F4        ; from iteration i
      2.ADDD    F4,F0,F2        ; from iteration i+1
      3.LD      F0,-16(R1)      ; from  iteration i+2
      4.SUBI    R1,R1,#8
      5.BNEZ    R1,loop
                < prologue code >
```



RAW dependence to next loop

- No RAW data dependencies *within* a loop
- WAR (↙) hazard elimination may be needed to increase ILP
- 6c / iteration, but only uses 2 FP regs (instead of 8 for unrolled)



# Superscalars

Erik Hagersten
Uppsala University
Sweden

UPPSALA UNIVERSITET

# Multiple instruction issue per clock

Goal: Extracting ILP so that CPI < 1 , i.e., IPC > 1

## NOW: Superscalar

- Combine static and dynamic scheduling to issue multiple instructions per clock
- HW finds independent instructions in “sequential” code

- Predominant: (PowerPC, SPARC, Alpha, HP-PA, x86, x86-64)

## Later: Very Long Instruction Words (VLIW):

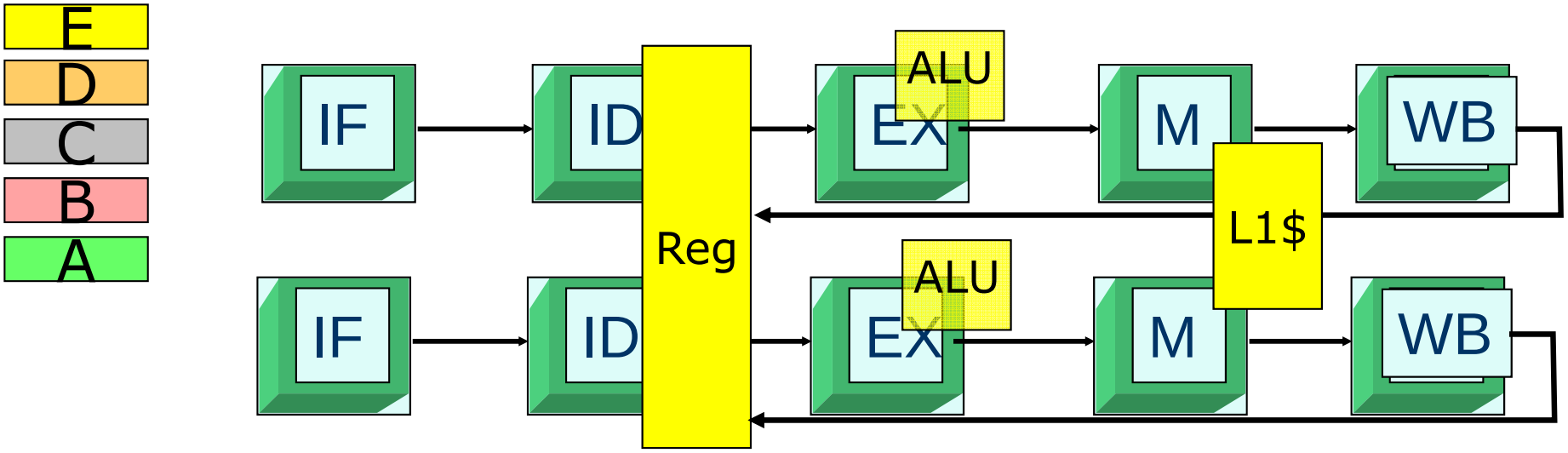
- Static scheduling used to form packages of independent instructions that can be issued together

- Relies on compiler to find independent instructions (IA-64 “Itanium”, Transmeta)

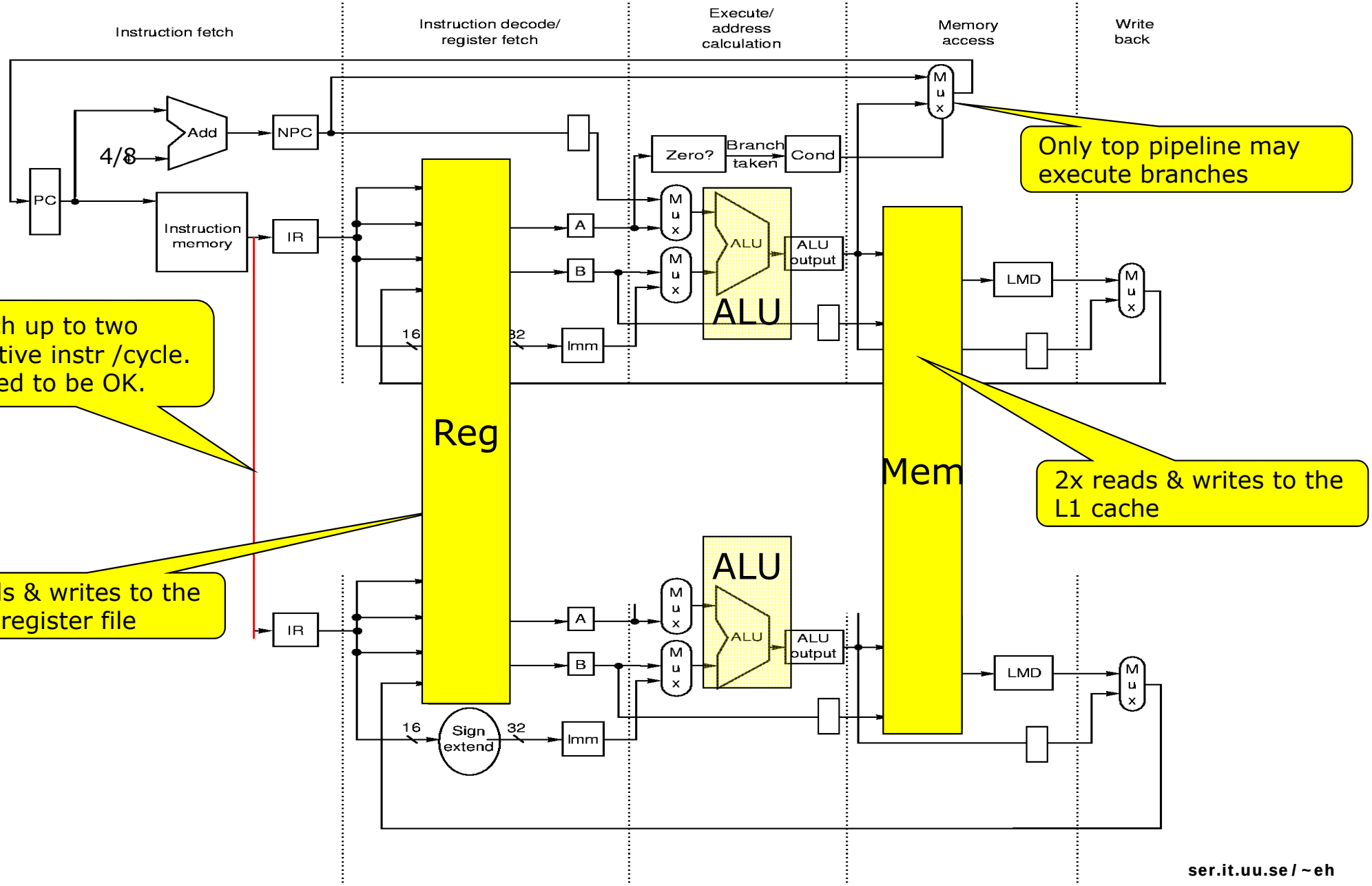
UPPSALA UNIVERSITET

# A 5-stage superscalar pipeline

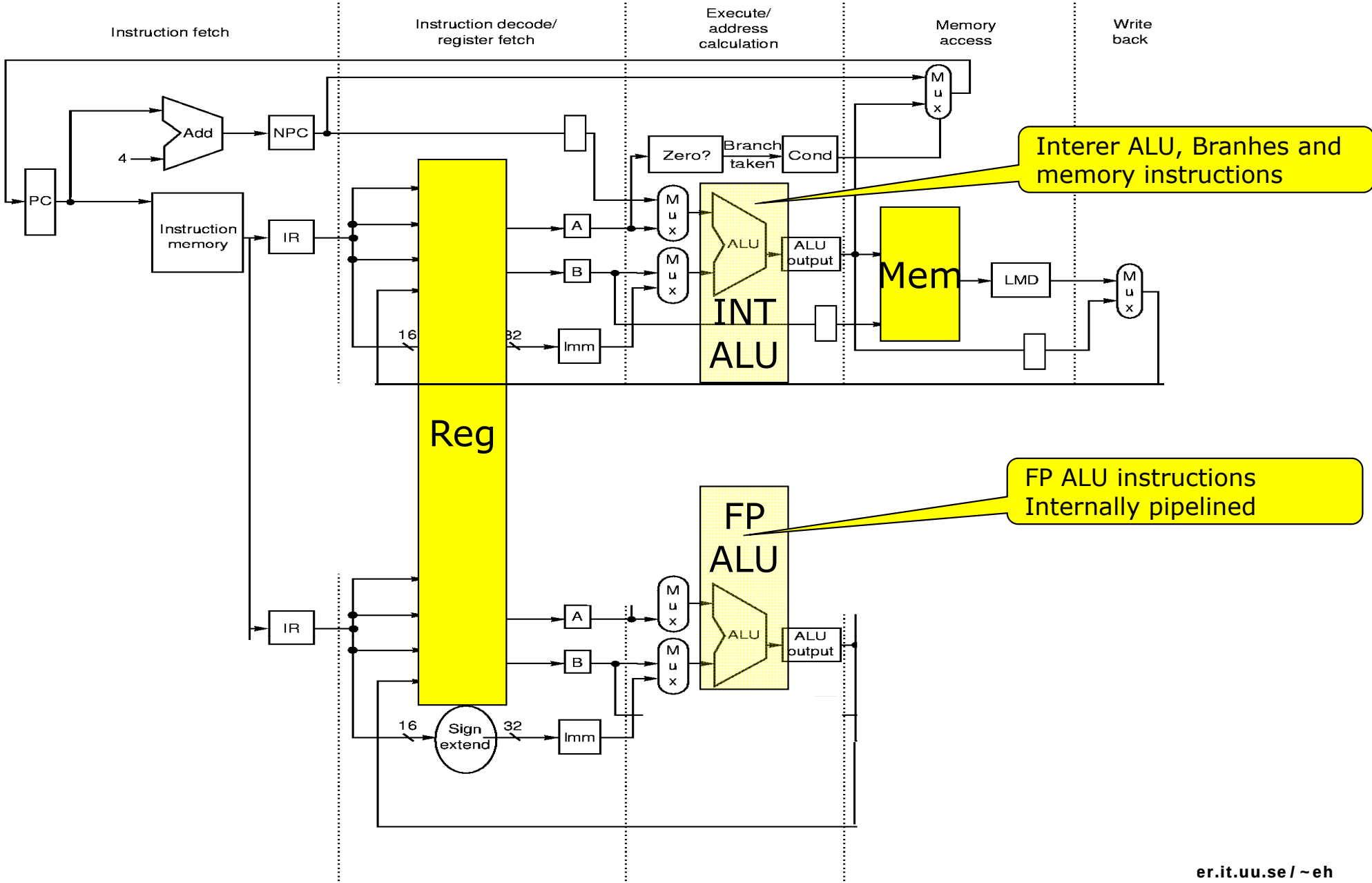
**One sequential program:**

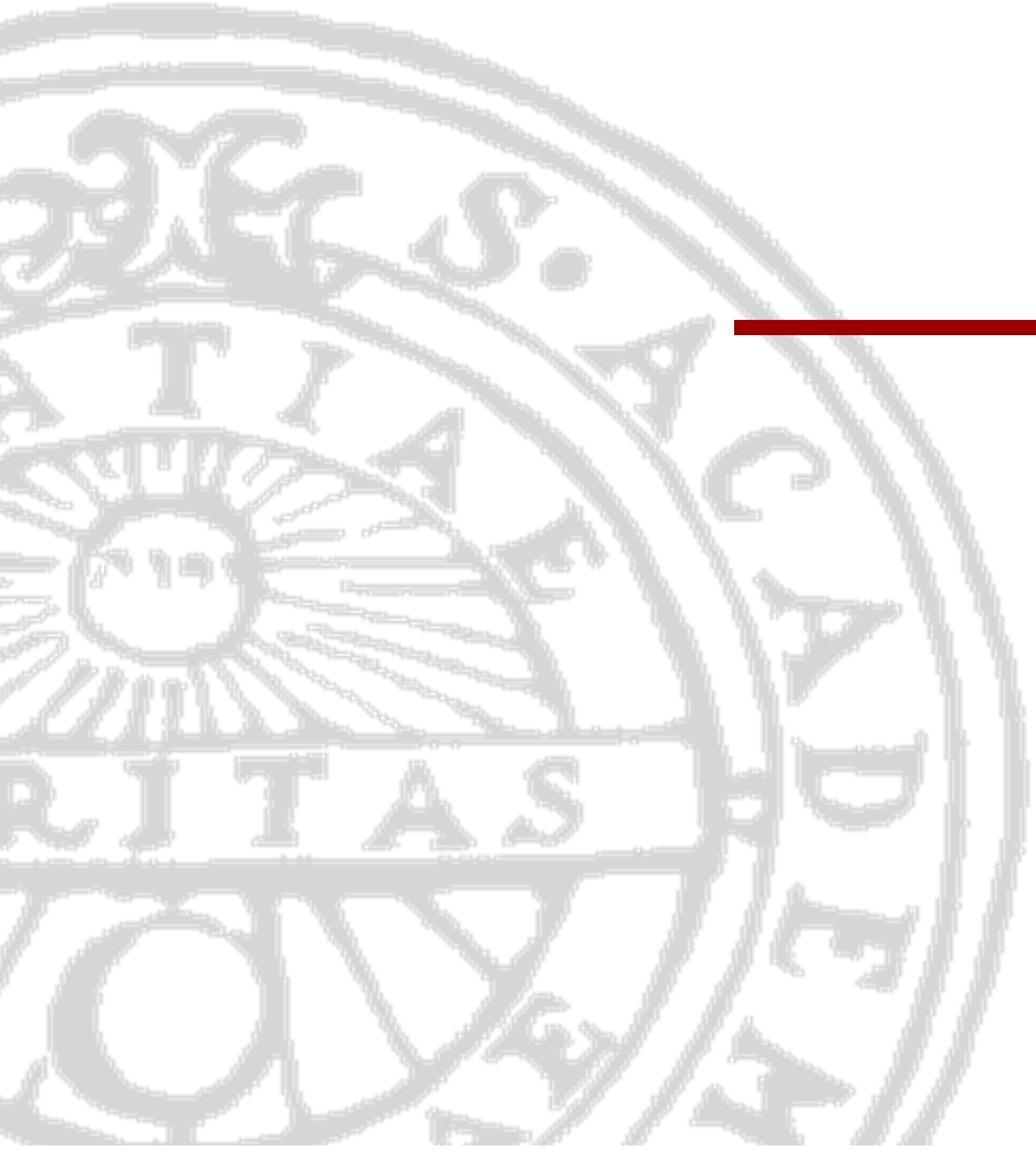


# A 2-way superscalar 5-stage pipeline Need 2x ILP!



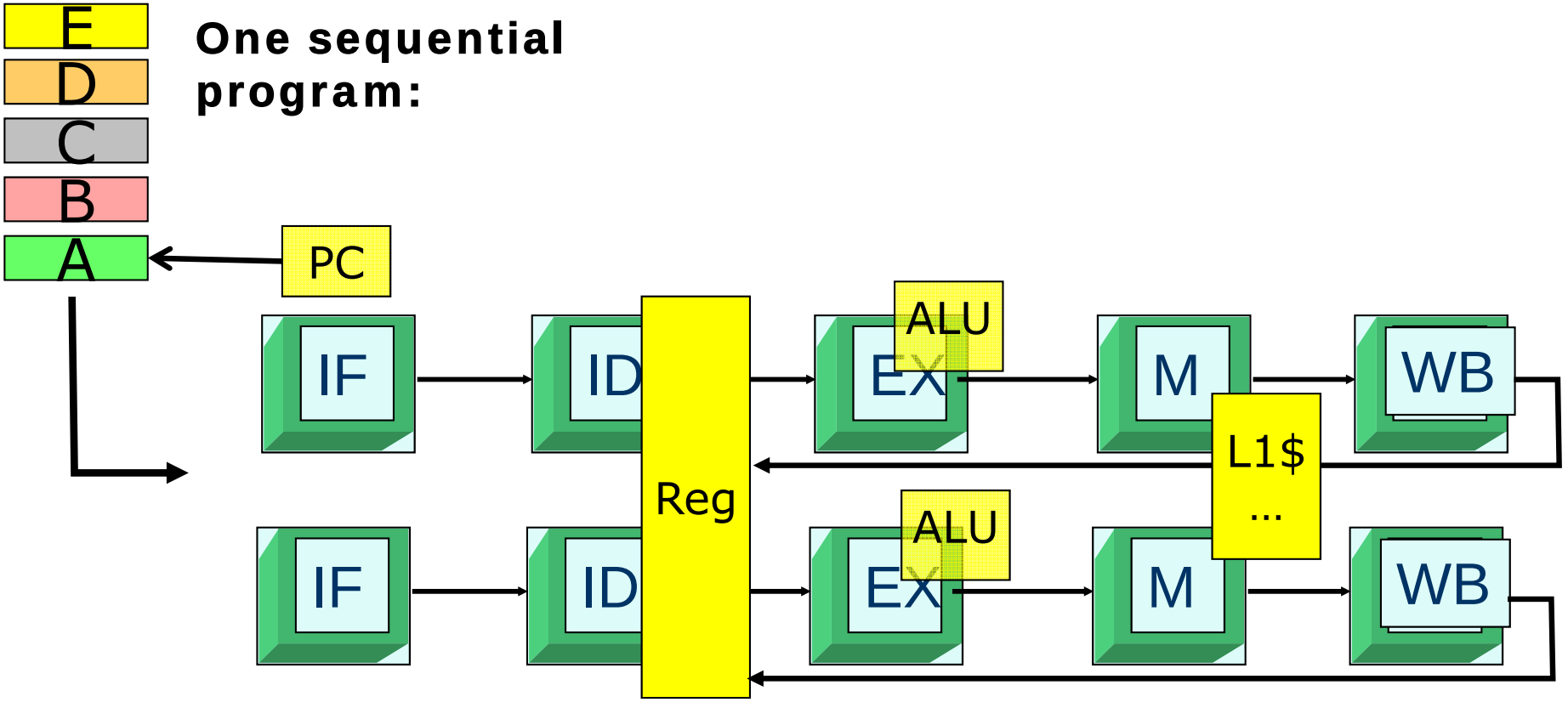
# The Superscalar DLX



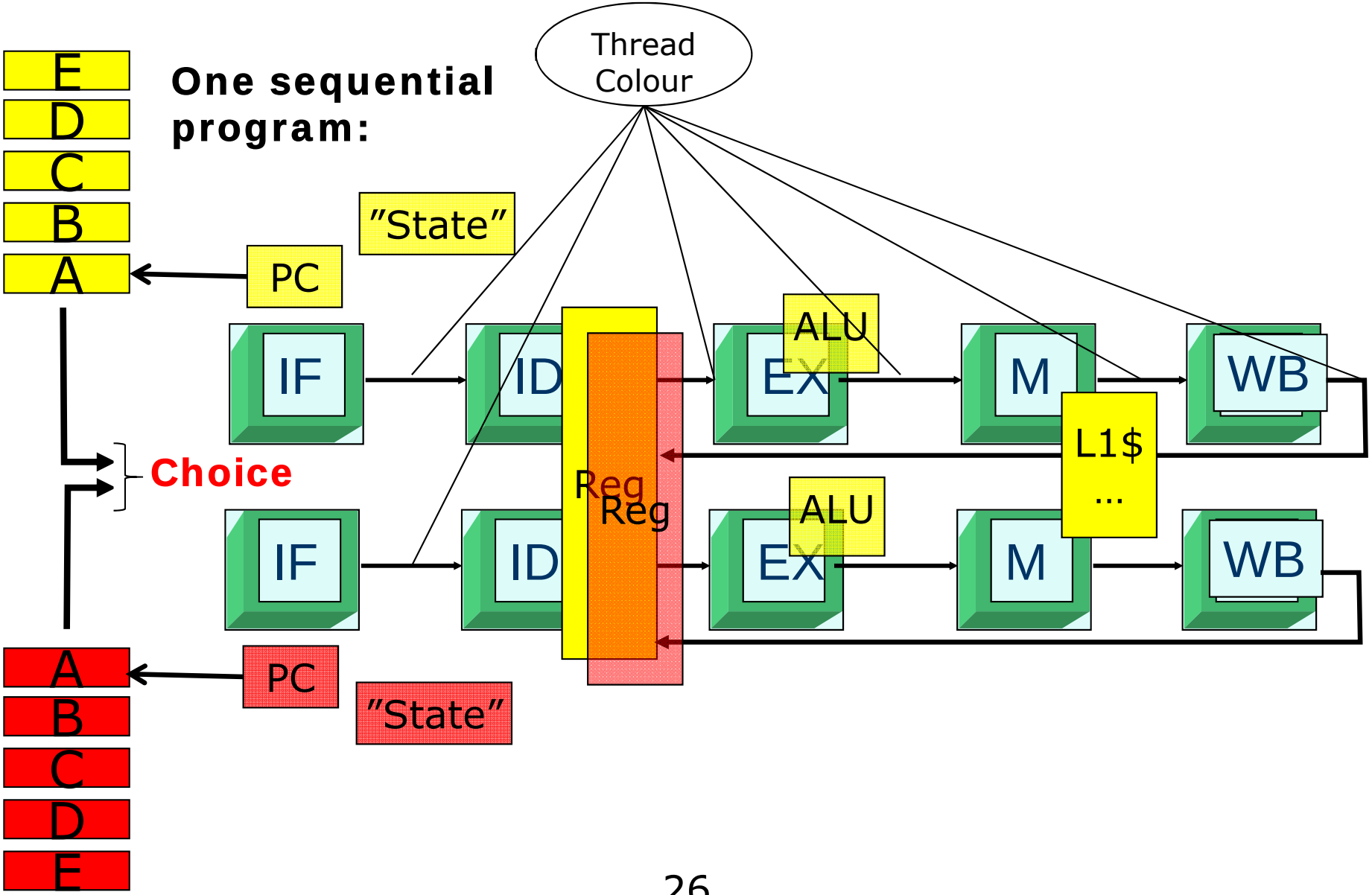


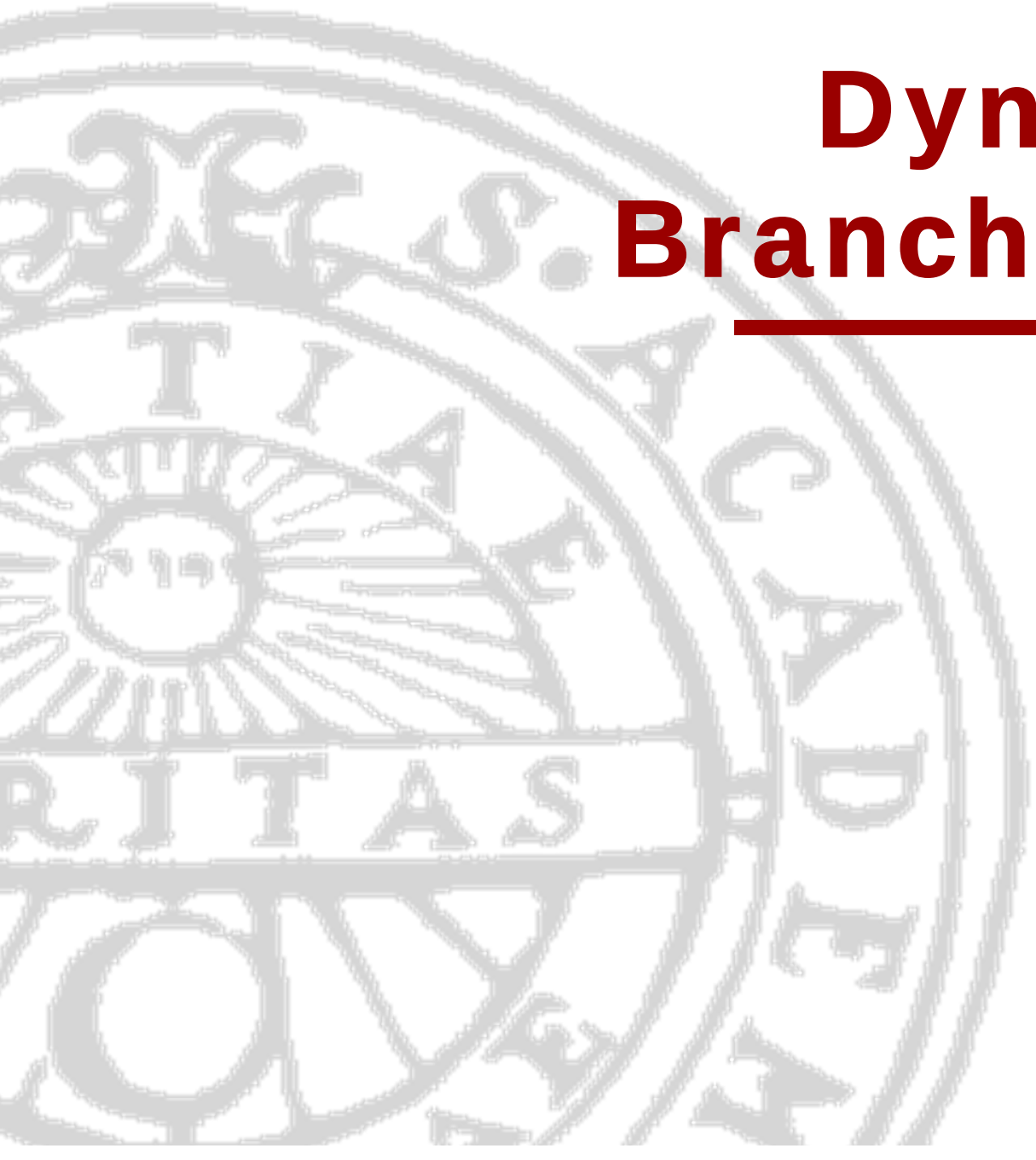
# SMTs

# A 5-stage 2-way superscalar pipeline



# A 5-stage 2-way superscalar pipeline, Multithreaded 2-ways



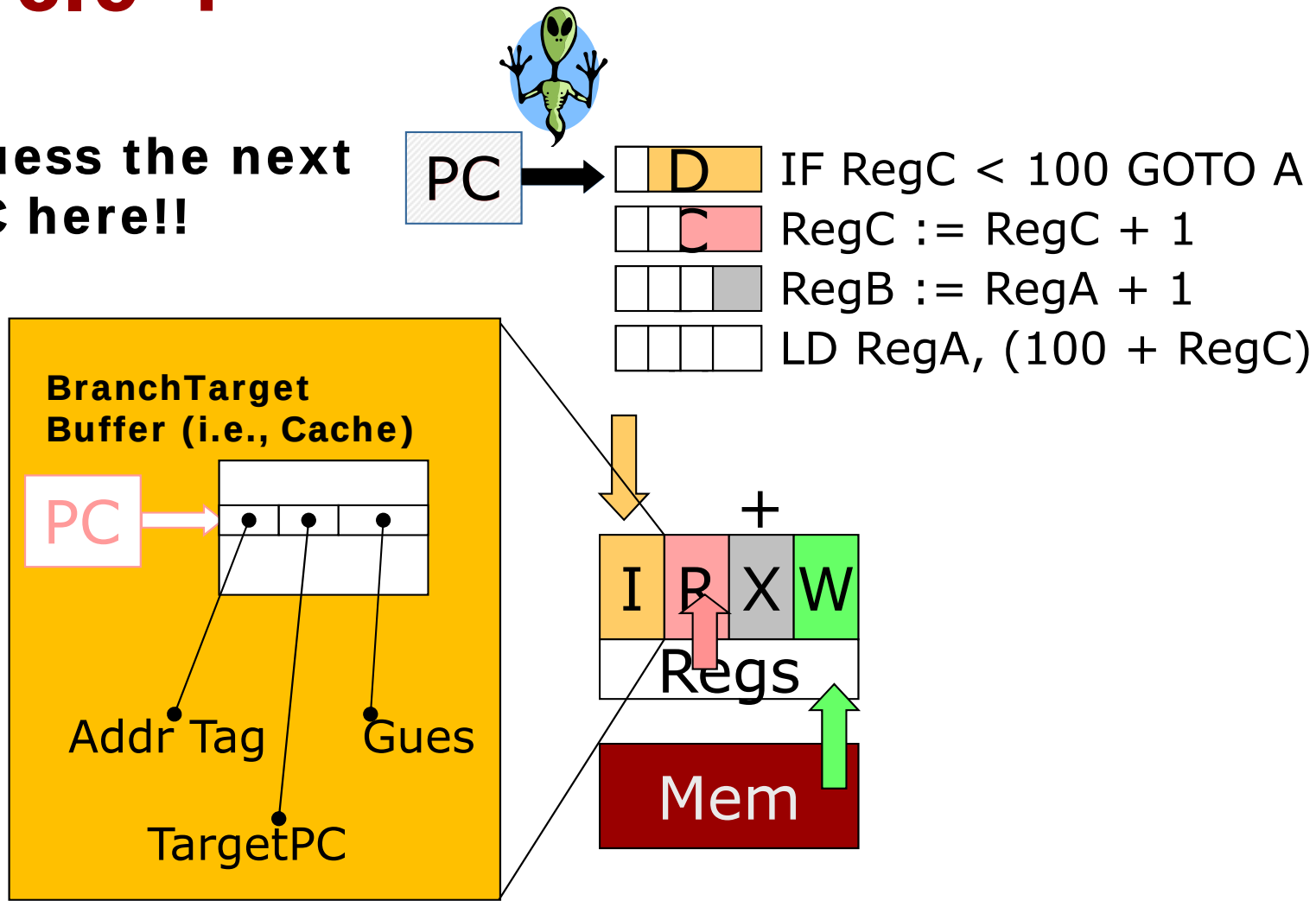


# Dynamic tricks: Branch Predictions

Erik Hagersten
Uppsala University

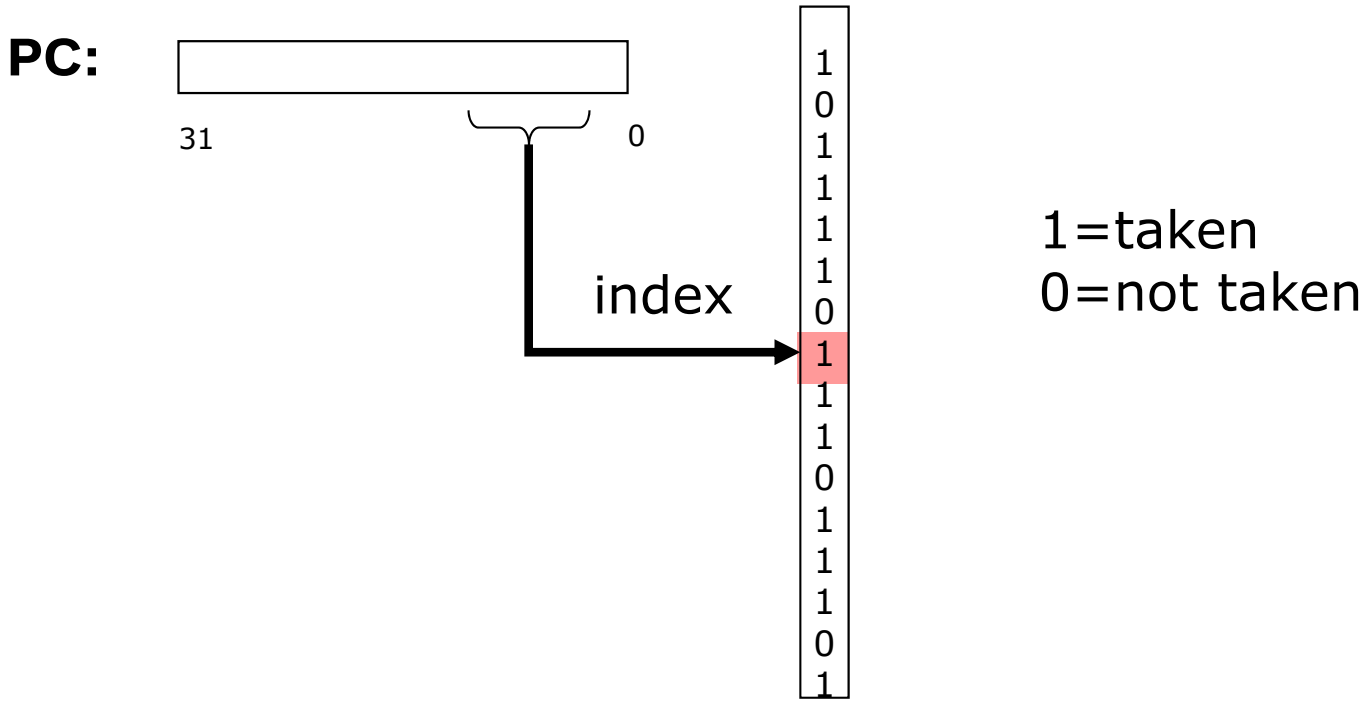
# Cycle 4

**Guess the next PC here!!**



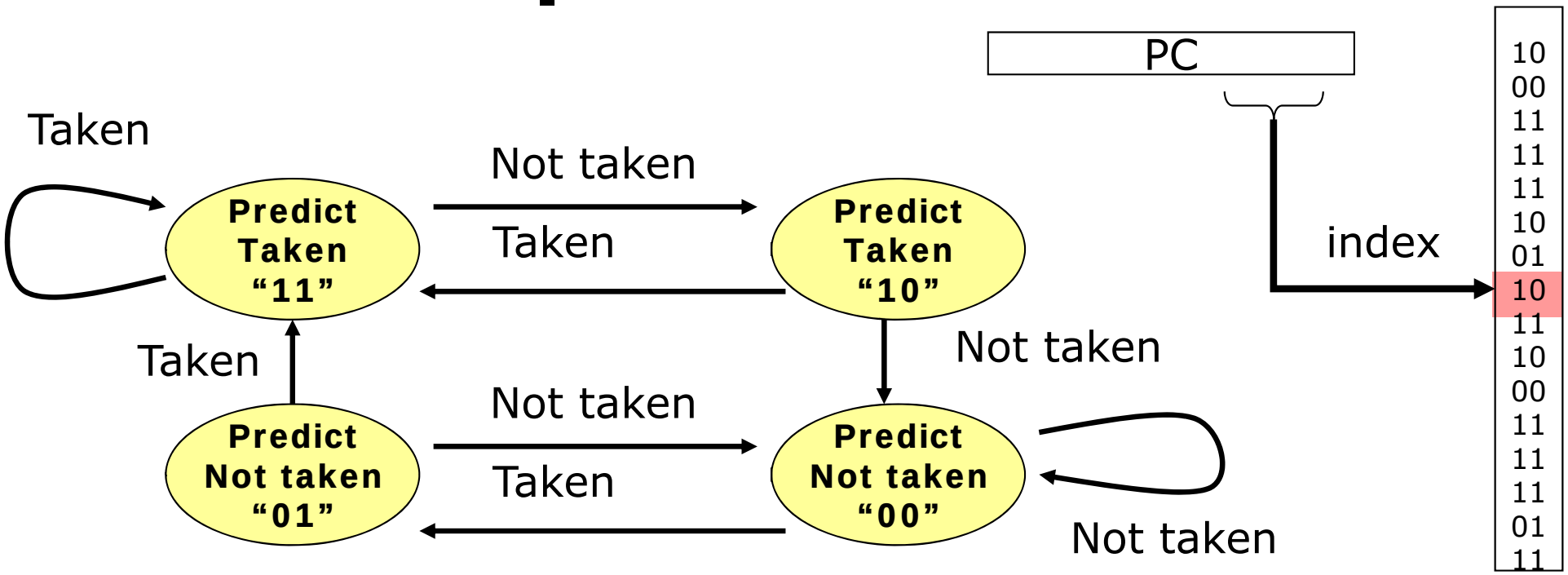

# Branch history table

## A simple branch prediction scheme



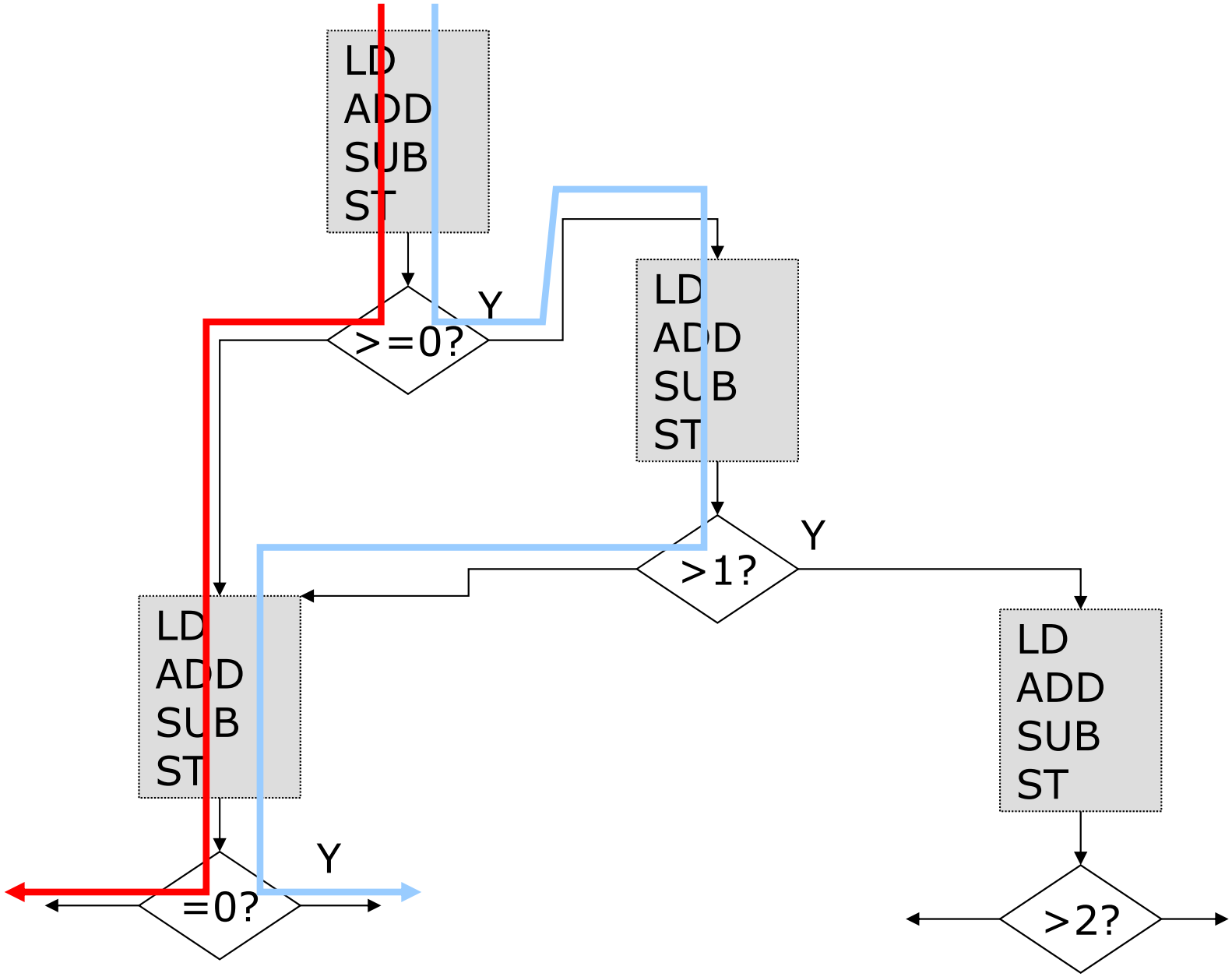

- The branch-prediction buffer is indexed by some bits from branch-instruction's PC values
- If prediction is wrong, then invert prediction

# A two-bit prediction scheme

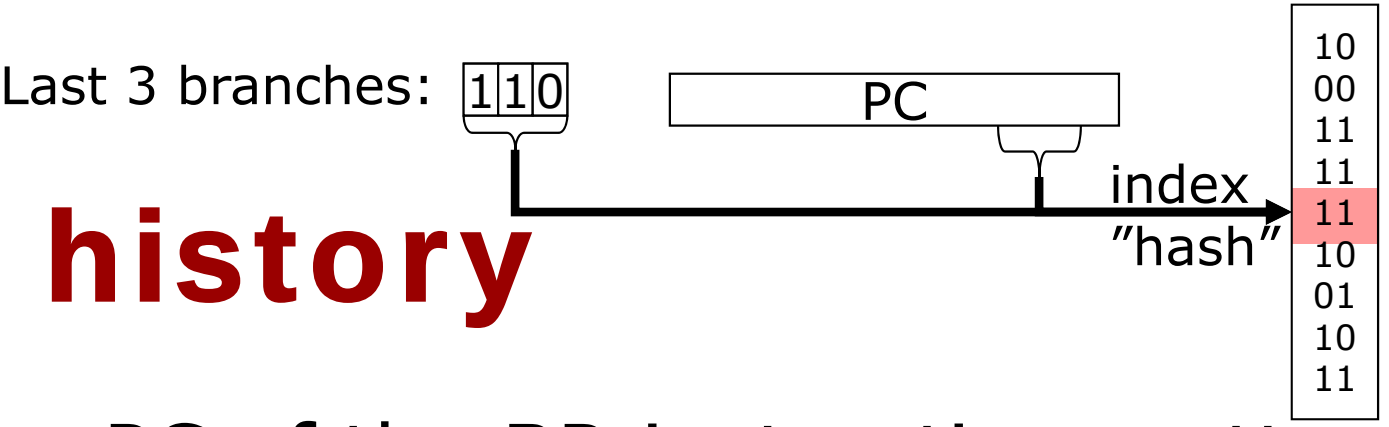



- Requires prediction to miss twice in order to change prediction => better performance

# Adding Global History



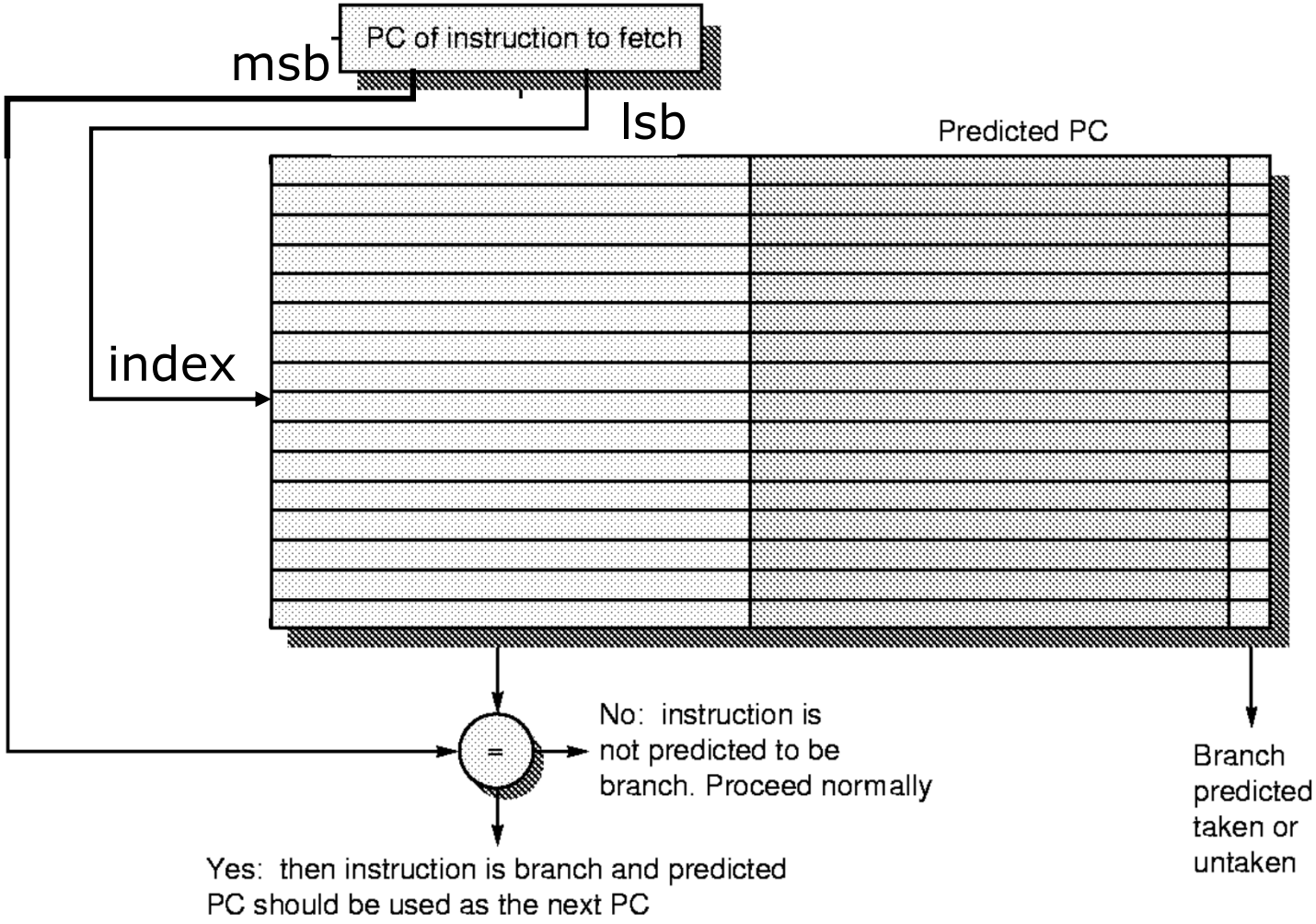
# N-level history




- Not only the PC of the BR instruction matters, also how you've got there is important
- Approach:
  - Record the outcome of the last N branches in a vector of N bits
  - Include the bits in the indexing of the branch table
- Pros/Cons: Same BR instruction may have multiple entries in the branch table

(N,M) prediction = N levels of M-bit prediction

# Branch target buffer (BTB)



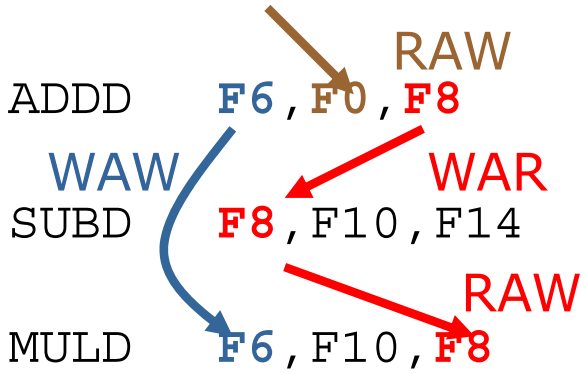

- Predicts *branch target address* in the ***IF*** stage
- Can be combined with 2-bit branch prediction

# Overlapping Execution & Out-of-order Execution

Erik Hagersten
Uppsala University
Sweden

# A more complicated example

```
DIV    F0,F2,F4     //delayed a long time
ADDD   F6,F0,F8
SUBD   F8,F10,F14
MULD   F6,F10,F8
```



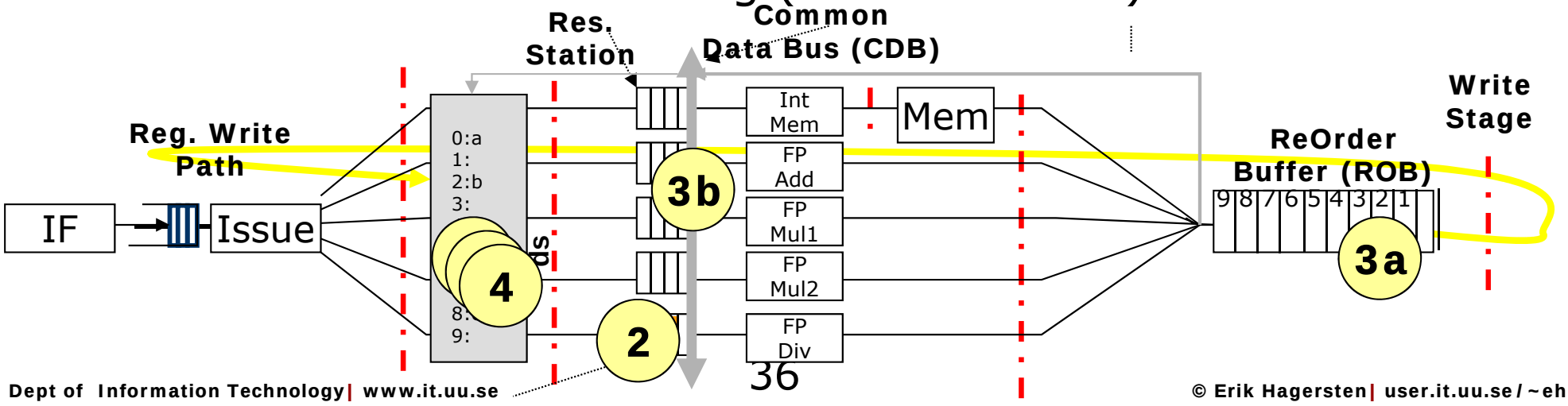

Note: WAR and WAW ("name dependencies") can be avoided through "register renaming"

**Register Renaming:**

```
DIV    F0,F2,F4
ADDD   F6,F0,F8
SUBD   tmp1,F10,F14   ;can be executed right away
MULD   tmp2,F10,tmp1  ;delayed a few cycles
```

# Tomasulo's: What is going on?

1. Read Register:
   - Rename DestReg to the Res. Station location
2. Wait for all RAW dependencies at Res. Station
3. After Execution
   a) Put result in Reorder Buffer (ROB)
   b) Broadcast result on CDB to all waiting instructions
   c) Rename DestReg to the ROB location
4. When all preceeding instr. have arrived at ROB:
   - Write value to DestReg (called Commit)



UPPSALA UNIVERSITET

# In a nutshell

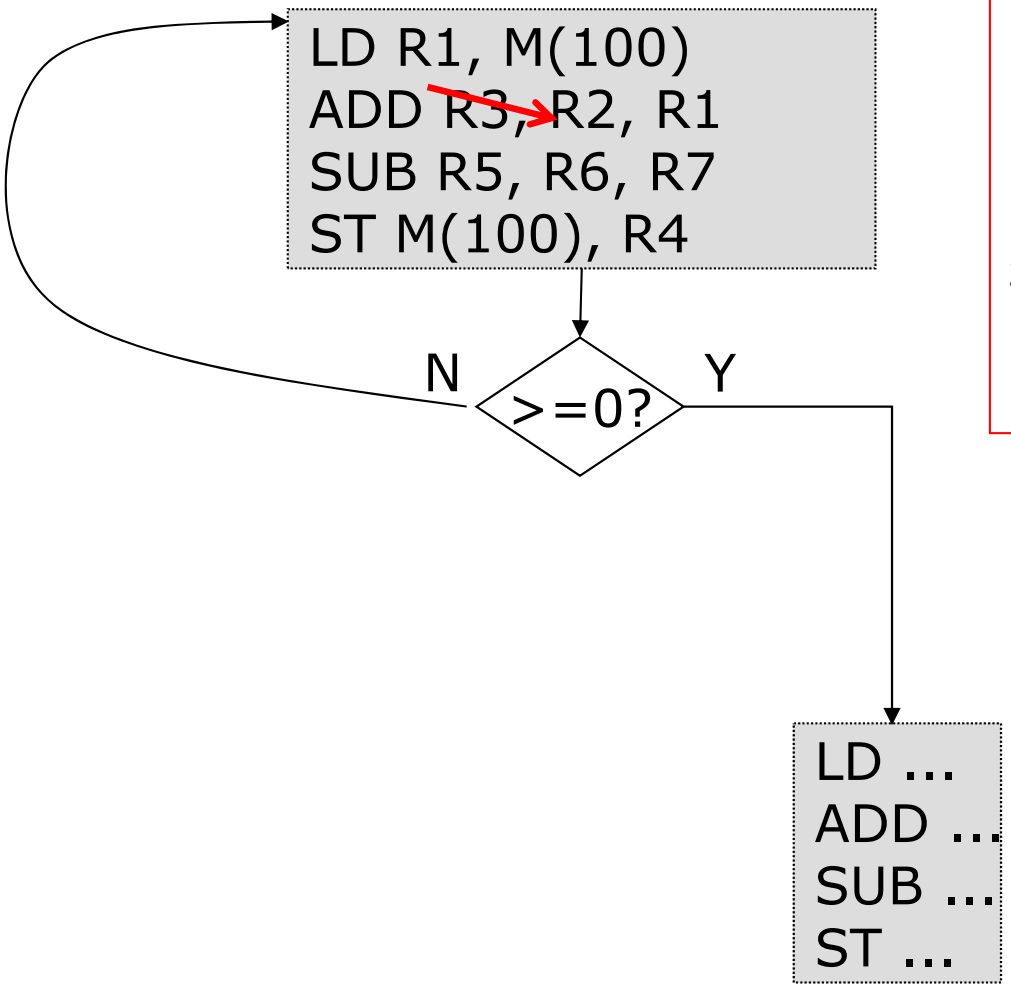
- Register renaming handles WAW&WAR
- If there is no RAW, re-ordering is OK
- Commit (apply side-effects) in order.
- For Load/store violations: Mem ops may need to be ”re-done”.

AVDARK 2013

UPPSALA UNIVERSITET

# Why is this such a big deal?

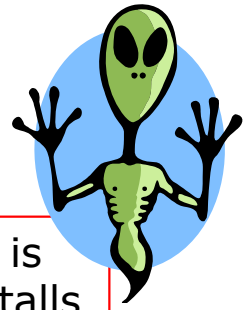

# Fix 1: Out-of order execution:
## Improving ILP

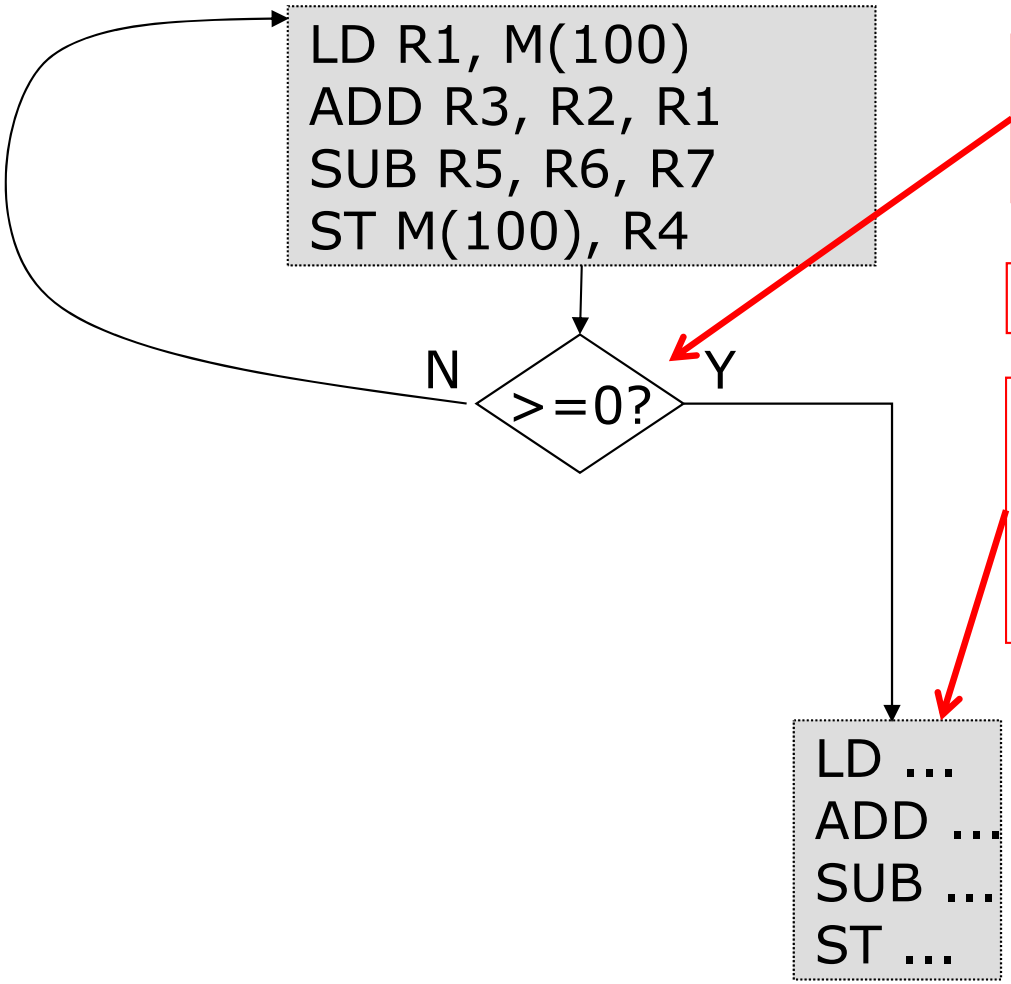



The HW may execute the instructions of a ”basis block” in a different order, but will make the ”side-effects” of the instructions appear in order.

Assume that LD takes a long time.
The ADD is dependent on the LD ☹
Start the SUB and ST before the ADD
Update R5 and M(100) **after** R3

# Fix 2: Branch prediction





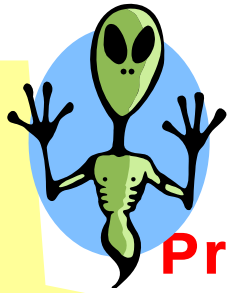

The HW can guess if the branch is taken or not and avoid branch stalls if the guess is correct.

Assume the guess is ”Y”.

The HW can start to execute these instruction **before** the outcome the the branch is known, but cannot allow any ”side-effect” to take place until the outcome is known

# Fix 3: Scheduling Past Branches

## Improving ILP

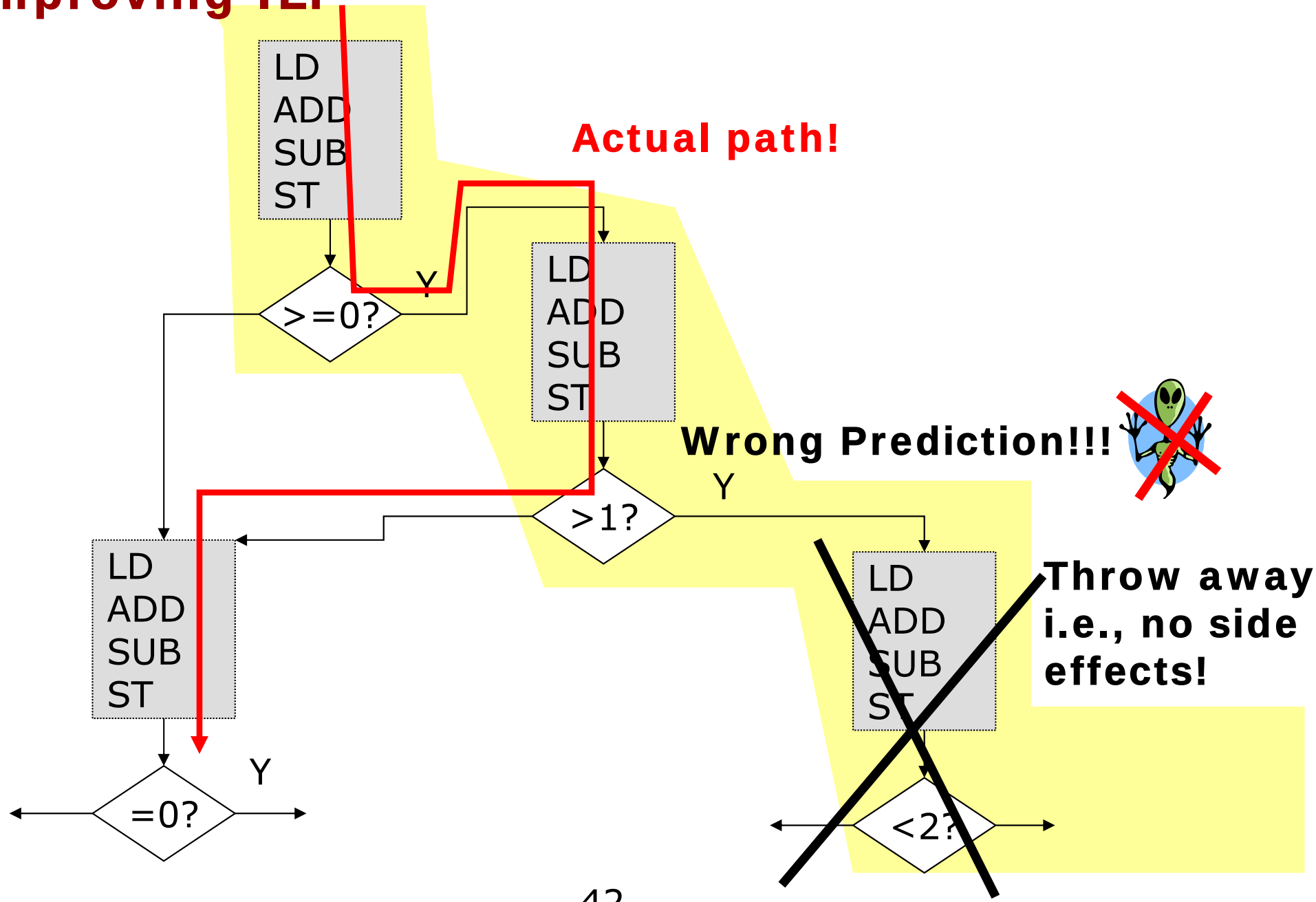


Predicted path

All instructions along the predicted path can be executed out-of-order (side-effects in-order)

# Fix 3: Scheduling Past Branches

## Improving ILP



UPPSALA UNIVERSITET

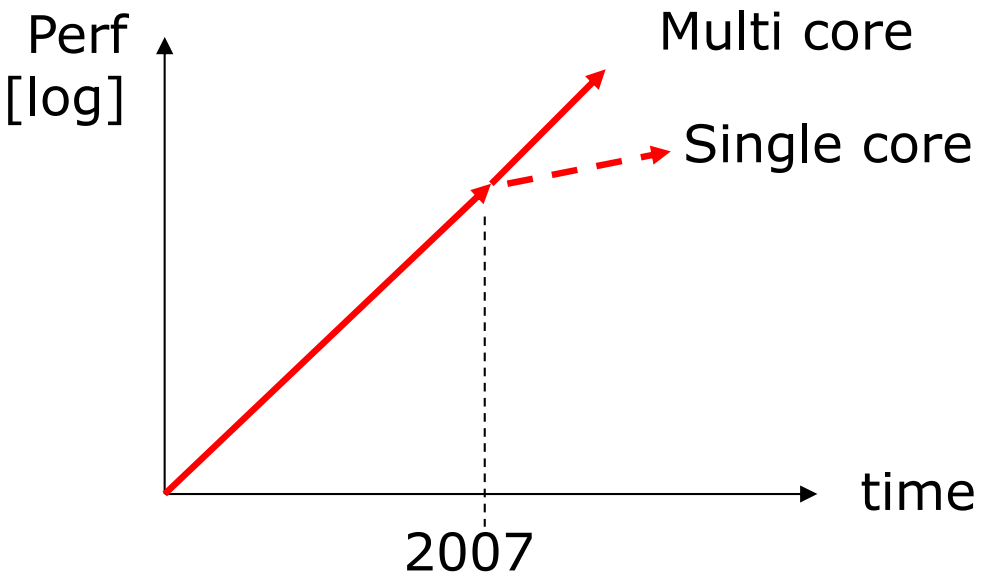
# Summing up Tomasulo's

- "Register renaming" avoids WAW, WAR
- Out-of-order (O-O-O) execution
- In order commit
  - Allows for speculative execution (beyond branches)
  - Allows for precise exceptions
- Distributed implementation
  - Reservation stations – wait for RAW resolution
  - Reorder Buffer (ROB)
  - Common Data Bus "snoops" (CDB)
- Costly to implement (complexity and power)

# Multicore:
## Why is it happening now?
### eller
### Hur Mår Moore's Lag?

Erik Hagersten
Uppsala Universitet


# Everybody is doing it! But, why now?




1. Not enough ILP to get payoff from using more transistors
2. Signal propagation delay » transistor delay
3. Power consumption $P_{dyn} \sim C \bullet f \bullet V^2$

UPPSALA UNIVERSITET

# Dennard Scaling

$$P_{dyn} = C * f * V^2 \approx \text{area} * \text{freq.} * \text{voltage}^2$$

Roughly constant between generations

Can potentially increase with smaller transistors

Robert Dennard (IBM): Lower voltage when you shrink the transistor. This trend has stopped ☹

**➔ Can not increase the frequency! Other options? TLP!**