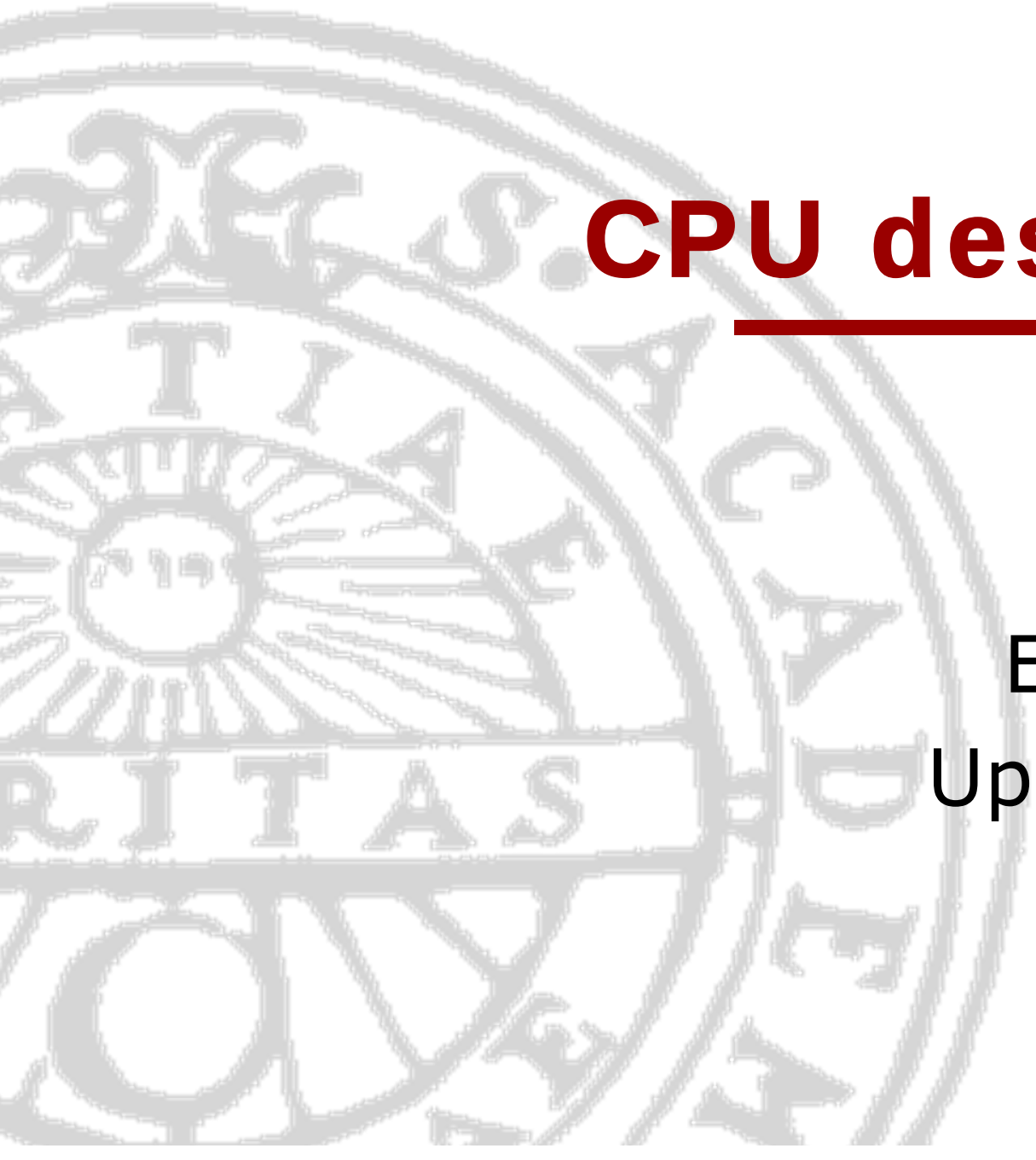


CPU design options

Erik Hagersten
Uppsala University

UPPSALA UNIVERSITET

Goal for this course

- Understand **how and why** modern computer systems are designed the way the are:
  - ⇨ pipelines
  - ✓ memory organization
  - ✓ virtual/physical memory ...

- Understand **how and why** multiprocessors are built
  - ✓ Cache coherence
  - ✓ Memory models
  - ✓ Synchronization…

- Understand **how and why** parallelism is created and
  - ⇨ Instruction-level parallelism
  - ⇨ ✓ Memory-level parallelism
  - ✓ Thread-level parallelism…

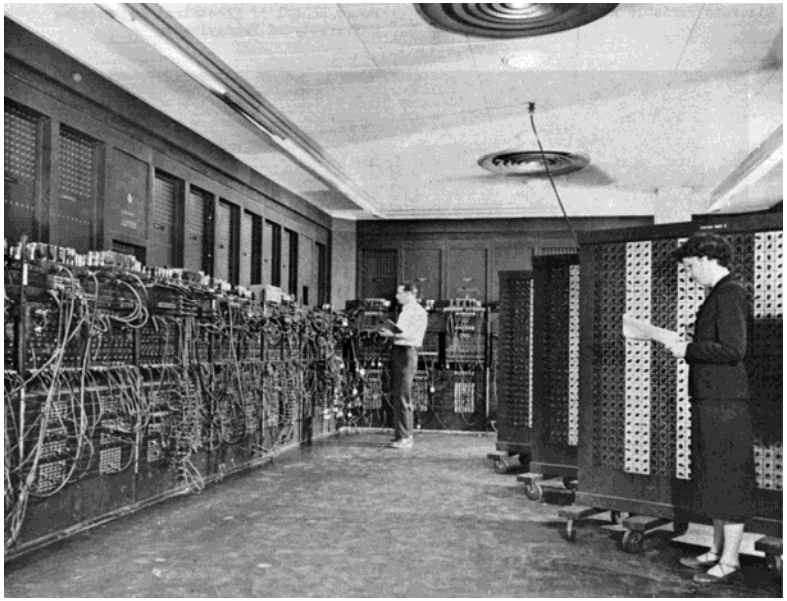
- Understand **how and why** multiprocessors of combined SIMD/MIMD type are built
  - GPU
  - ⇨ Vector processing…

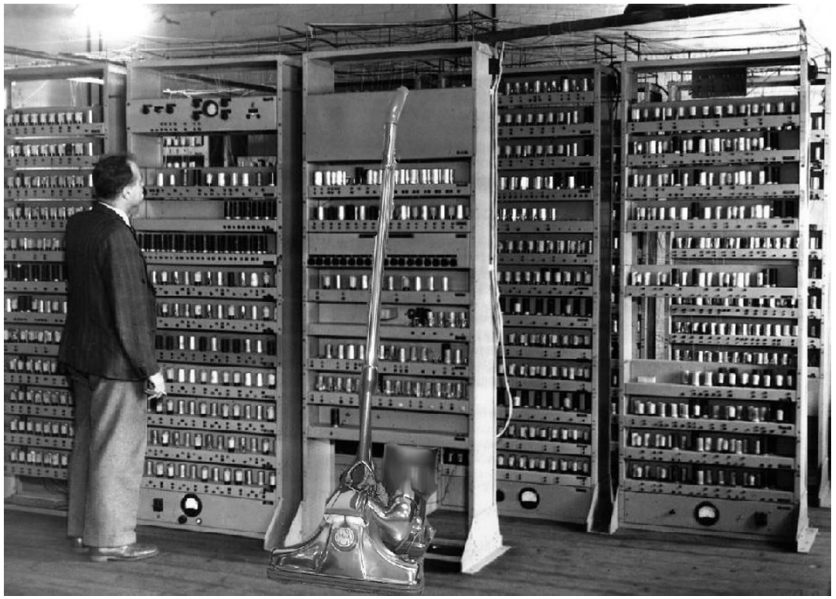
- Understand **how** computer systems are adopted to different usage areas
  - ✓ General-purpose processors
  - Embedded/network processors…

- Understand the physical limitation of modern computers
  - ✓ Bandwidth
  - ✓ Energy
  - ✓ Cooling…

AVDARK 2013

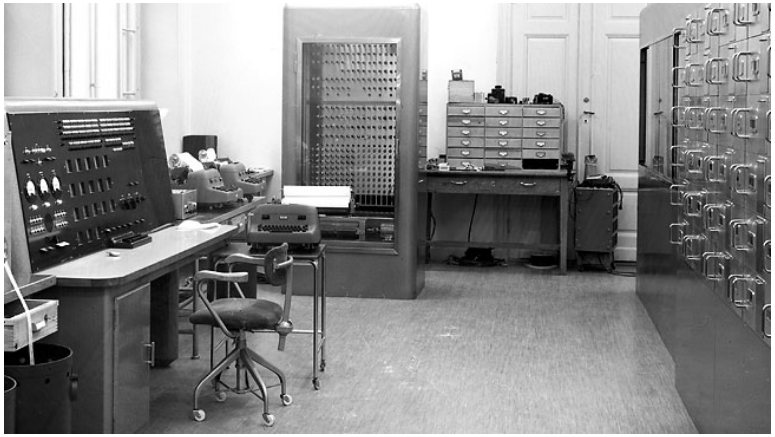
How it all started…the fossils





- ENIAC J.P. Eckert and J. Mauchly, Univ. of Pennsylvania, 1943
  - Electro Numeric Integrator And Calculator, 18.000 vacuum tubes
- Mark-I++, H. Aiken, Harvard, 1944, Electro-mechanic
- EDVAC, J. V Neumann, 1947
  - Electric Discrete Variable Automatic Computer (stored programs)
- EDSAC, M. Wilkes, Cambridge University, 1949
  - Electric Delay Storage Automatic Calculator
- K. Zuse, Germany, electro mech. computer, special purpose, WW2

The Swedish fossils



- BARK, KTH, Gösta Neovius (later at Ericsson), 1950.
  - Electro-mechanic technology from Ericsson
  - Neovius had visited Aiken at Stanford.
  - 32-bit computer. Addition: 150 ms
- BESK, KTH, Erik Stemme (later at Chalmers), 1953.
  - 2400 tubes and 400 Germanium diods
  - Stemme had visited von Neuman at Pennsylvania
  - 40-bit computer. Addition: 45 us.
- SMIL, LTH, 1956. Work on Algol compilers (!!)

UPPSALA UNIVERSITET

Order these machines in the right historical order, starting with the oldest:

AVDARK 2013

UPPSALA UNIVERSITET

The time before 1990

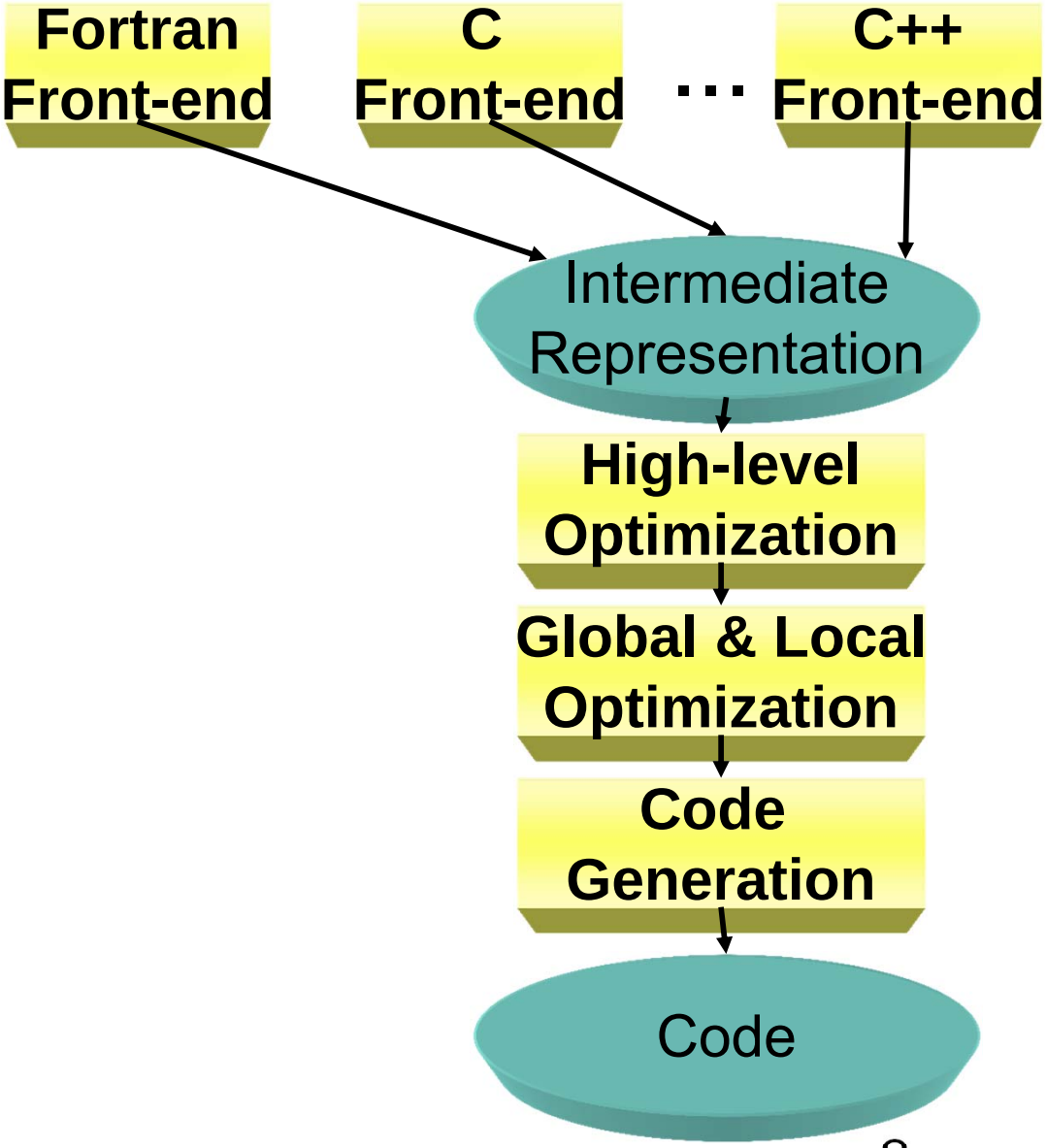
- Assembler was often used directly by programmers
- Complex instructions to bridge the semantic gap
  - e.g., ICL's instructions for arithmetics with pounds, shillings and pence
- Complex Instruction-set Computers (CISC)

UPPSALA UNIVERSITET

ISA trends today

- CPU families built around “Instruction Set Architectures” ISA
- Example: IA32 (~x86)
- ISAs lasting longer (~20 years)
- Many implementations of the same ISA
- Consolidation in the market - fewer ISAs today
- 20 years ago ISAs were driven by academia
- Today ISAs technically do not matter all that much (market-driven)

Compiler Organization



Machine-independent Translation

Procedure in-lining
Loop transformation

Register Allocation
Common sub-expressions

Instruction selection
constant folding

UPPSALA UNIVERSITET

Topics for ISA design

- Operand model
- Instruction format
- Instruction mix
  - Handling of conditional branches
  - Addressing modes
  - Size of immediates
- …
- Implementation independent

AVDARK 2013

Operand model

UPPSALA UNIVERSITET

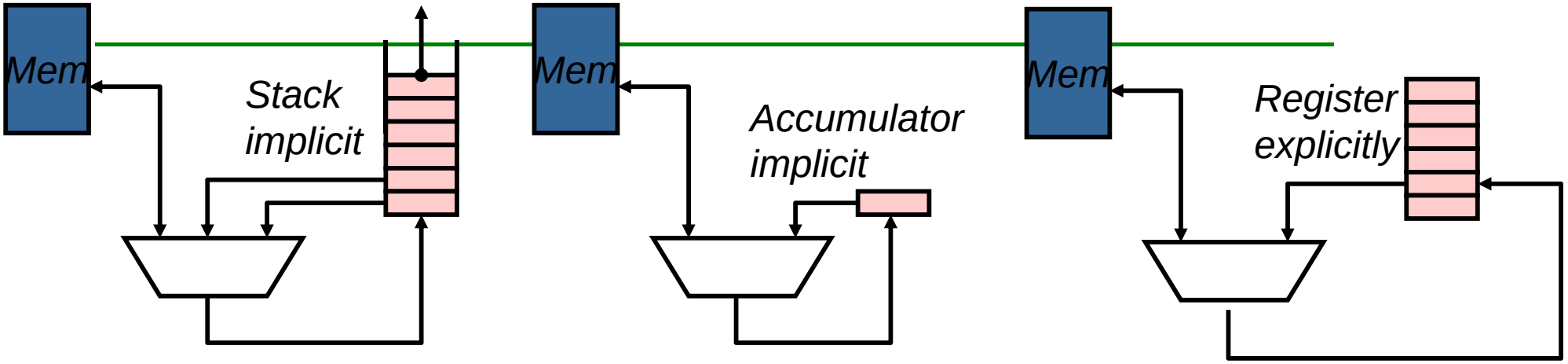
Topics for ISA design

- Operand model
- Instruction format
- Instruction mix
  - Handling of conditional branches
  - Addressing modes
  - Size of immediates
- …
- Implementation independent

Operand models

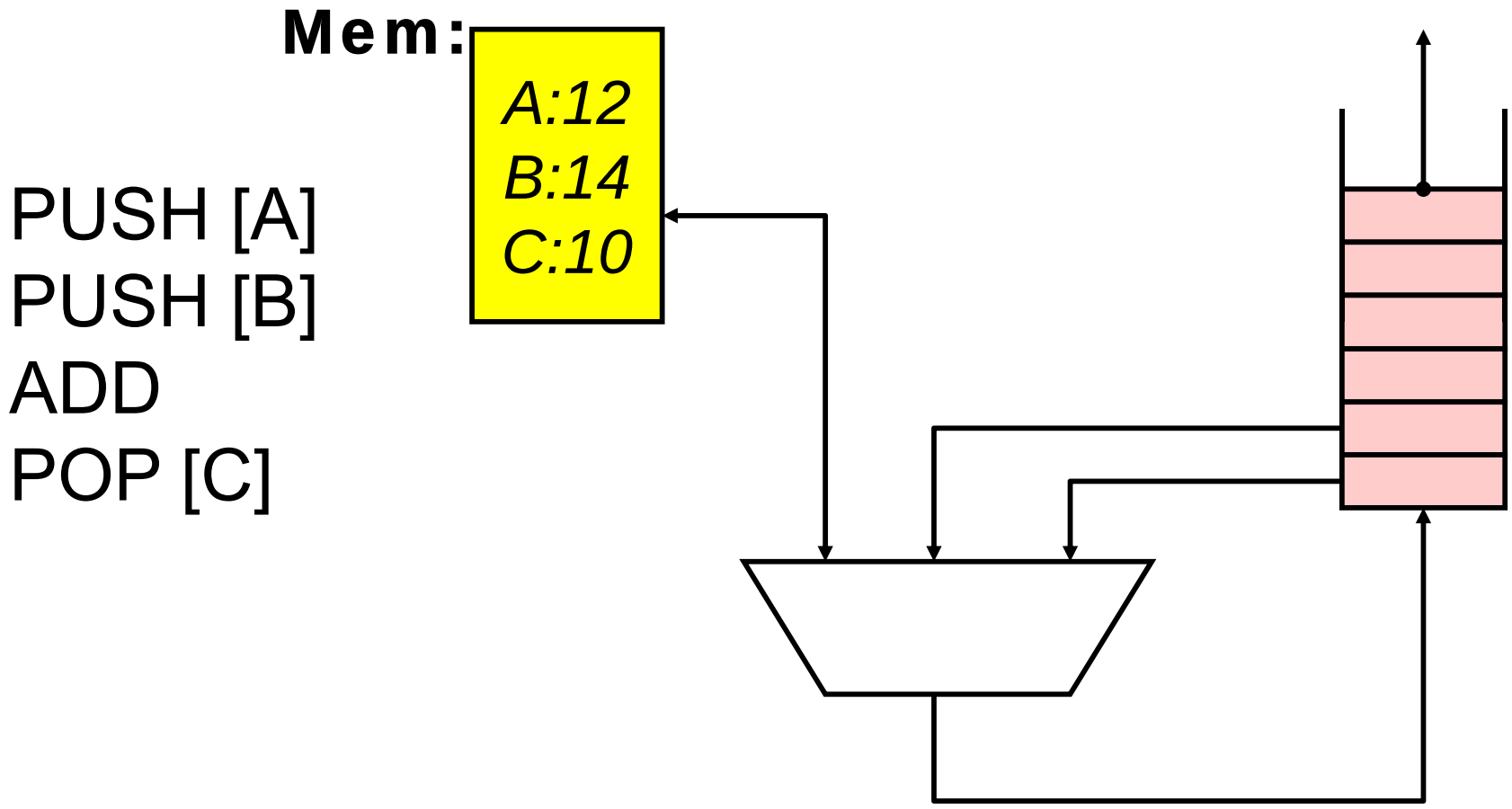
Example: C := A + B

Stack	Accumulator	Register
PUSH [A]	LOAD [A]	LOAD R1,[A]
PUSH [B]	ADD [B]	ADD R1,[B]
ADD	STORE [C]	STORE [C],R1
POP [C]		



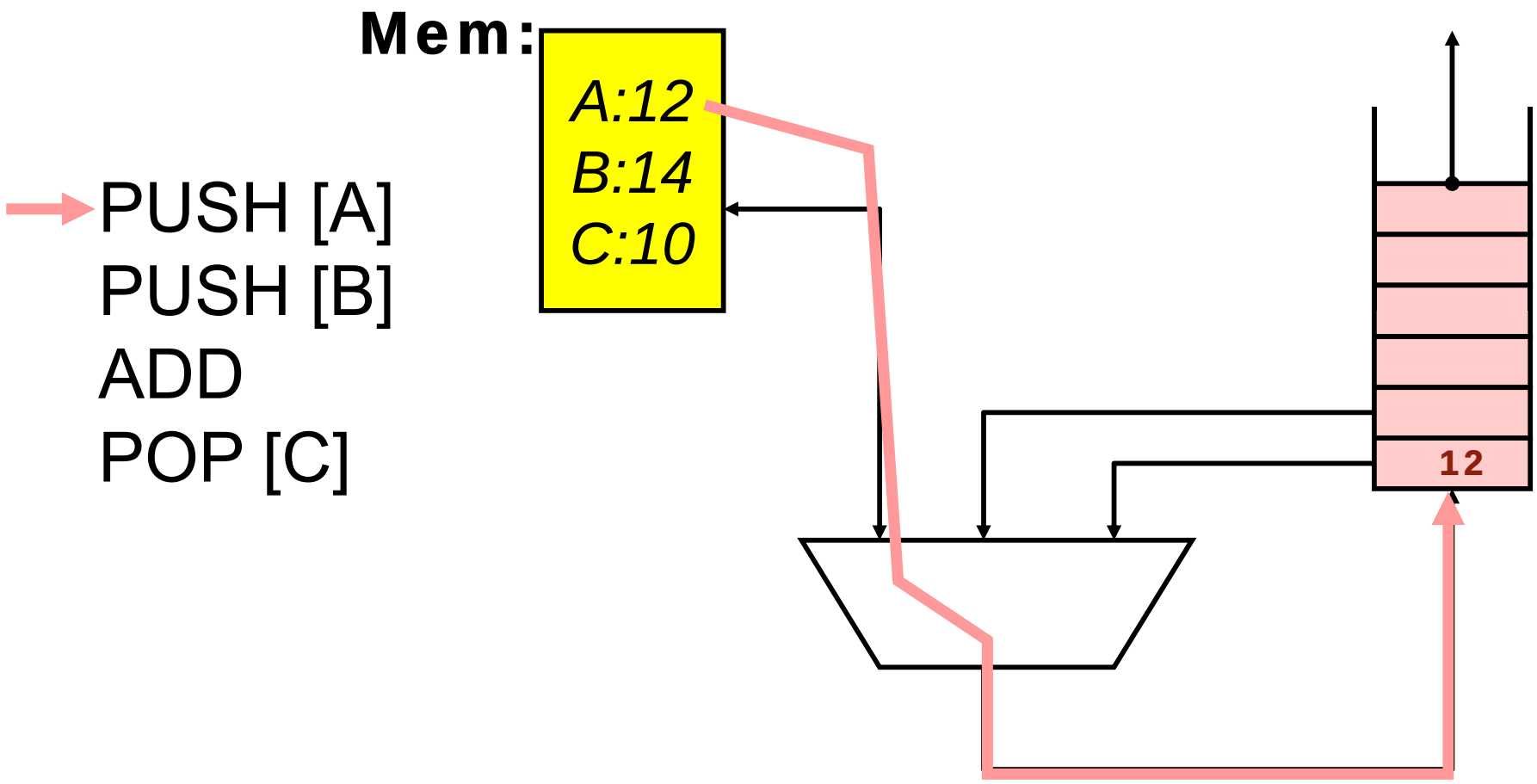
Stack-based machine

Example: C := A + B



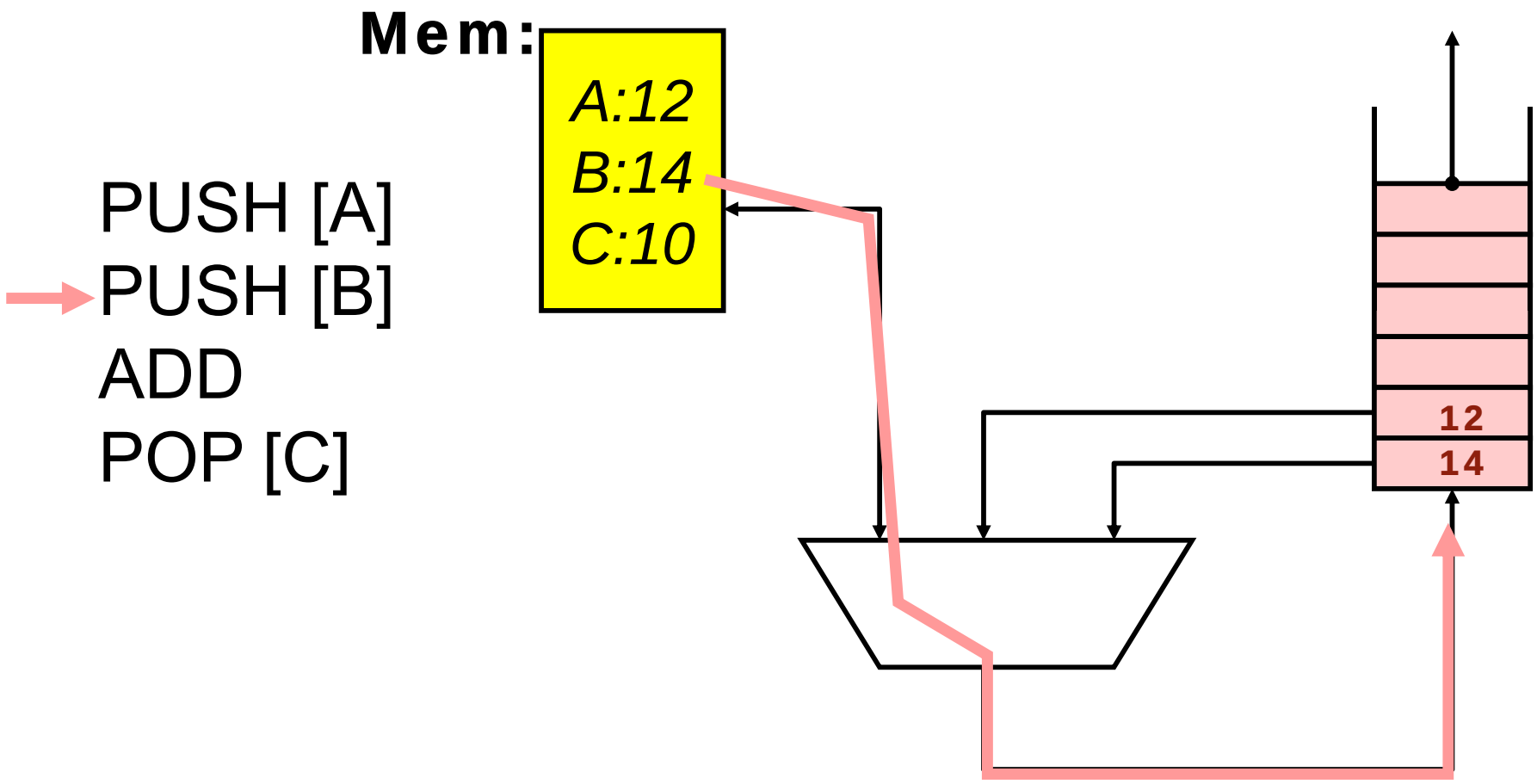
Stack-based machine

Example: C := A + B



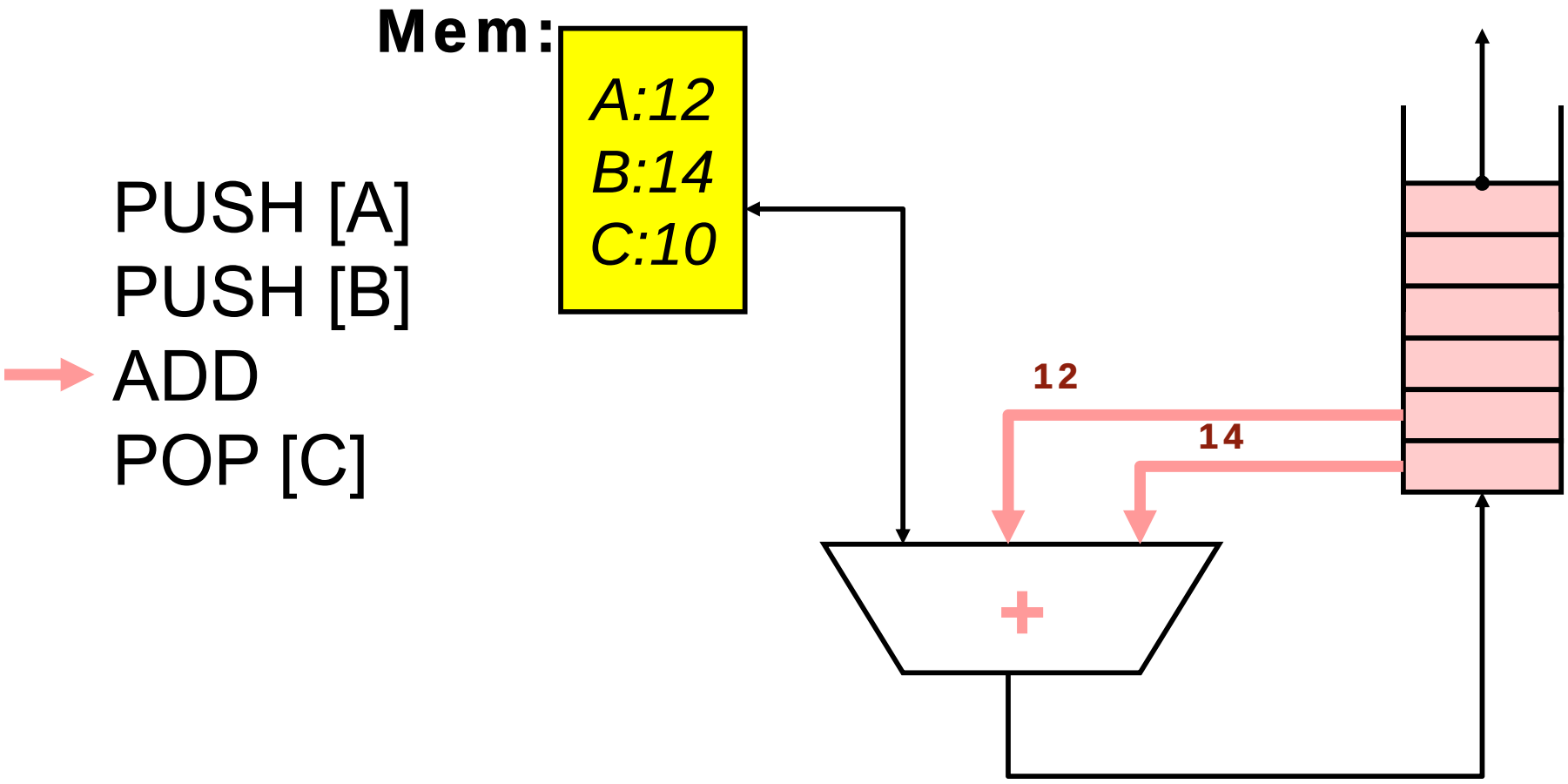
Stack-based machine

Example: C := A + B



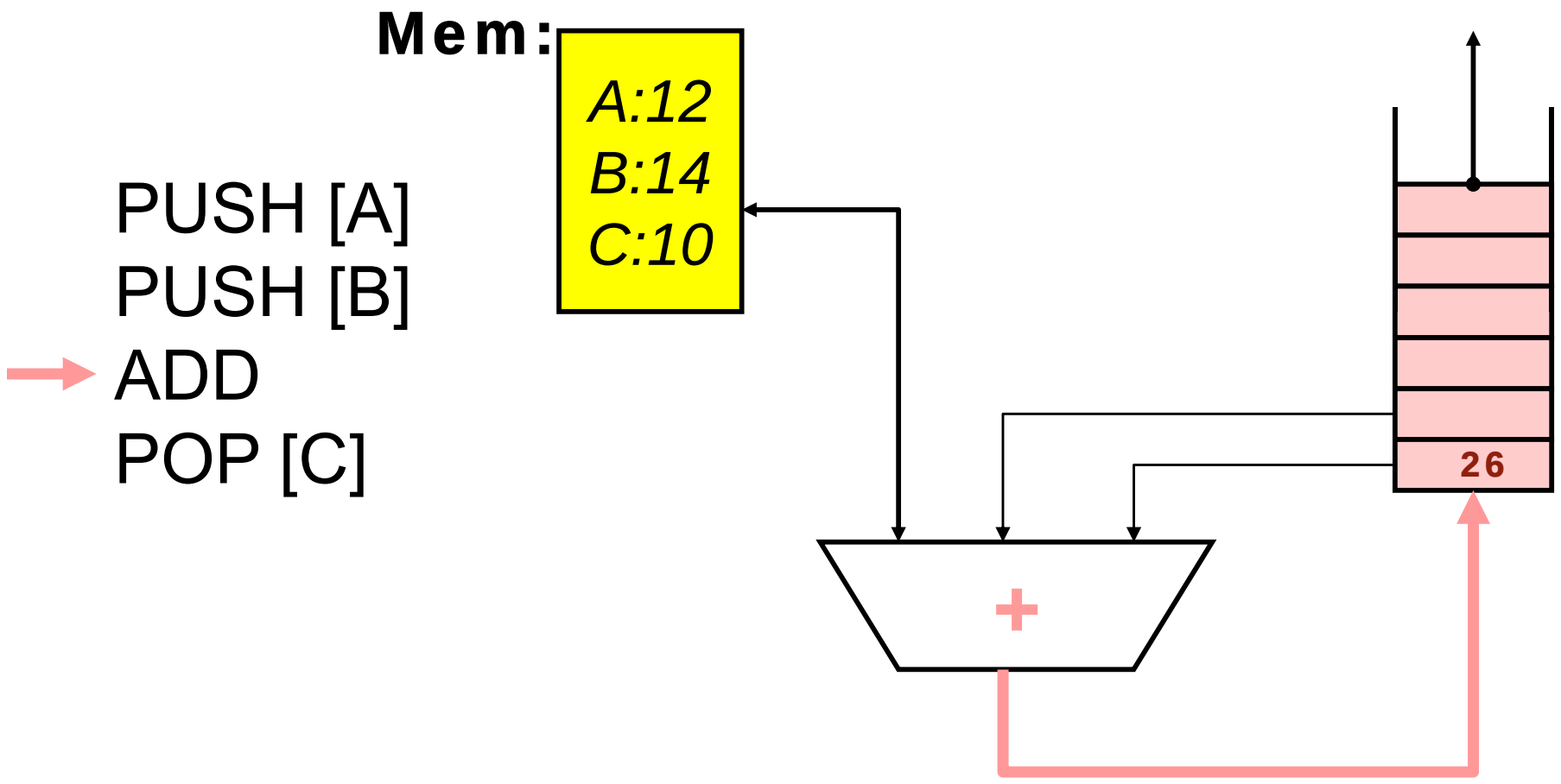
Stack-based machine

Example: C := A + B



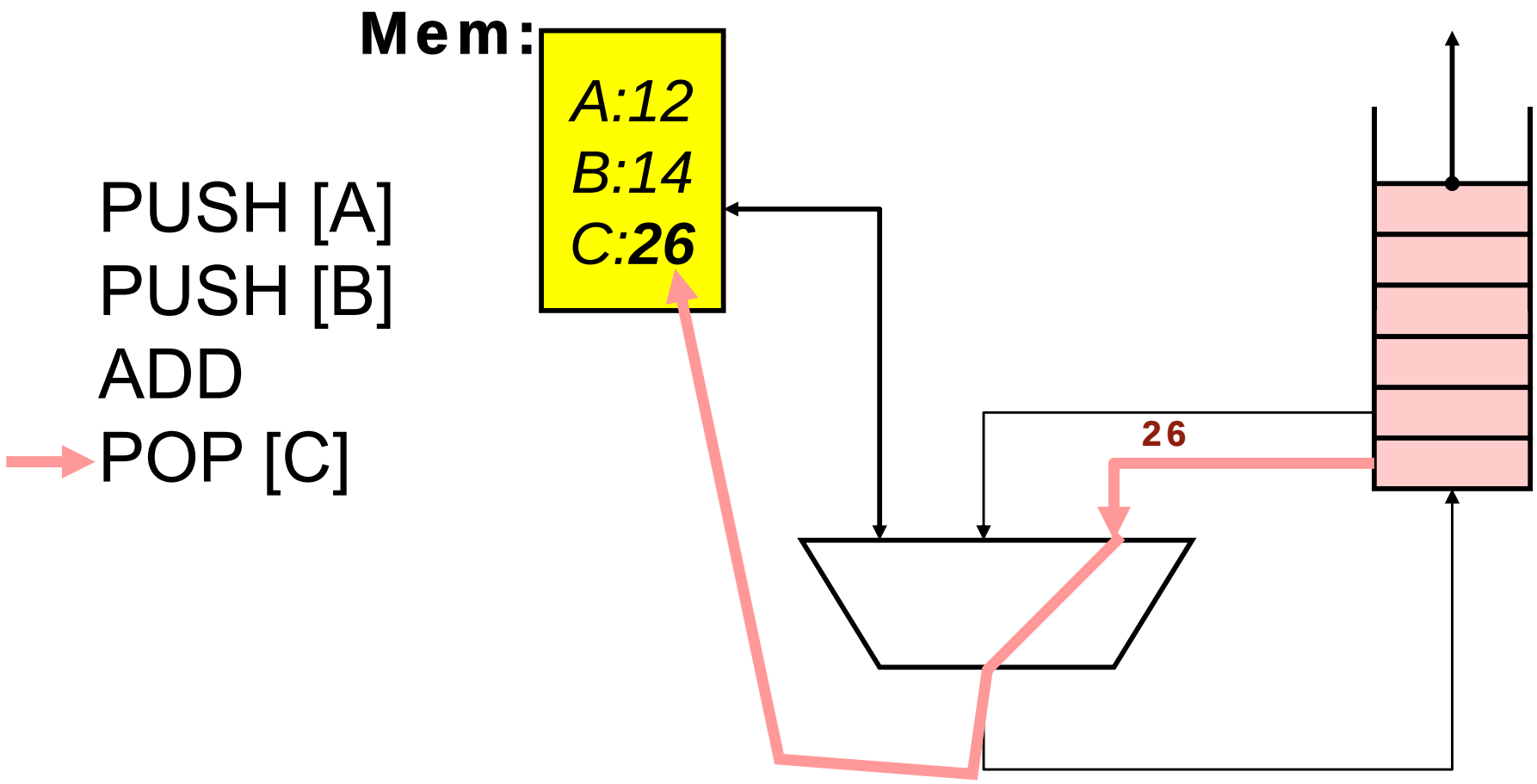
Stack-based machine

Example: C := A + B



Stack-based machine

Example: C := A + B



UPPSALA UNIVERSITET

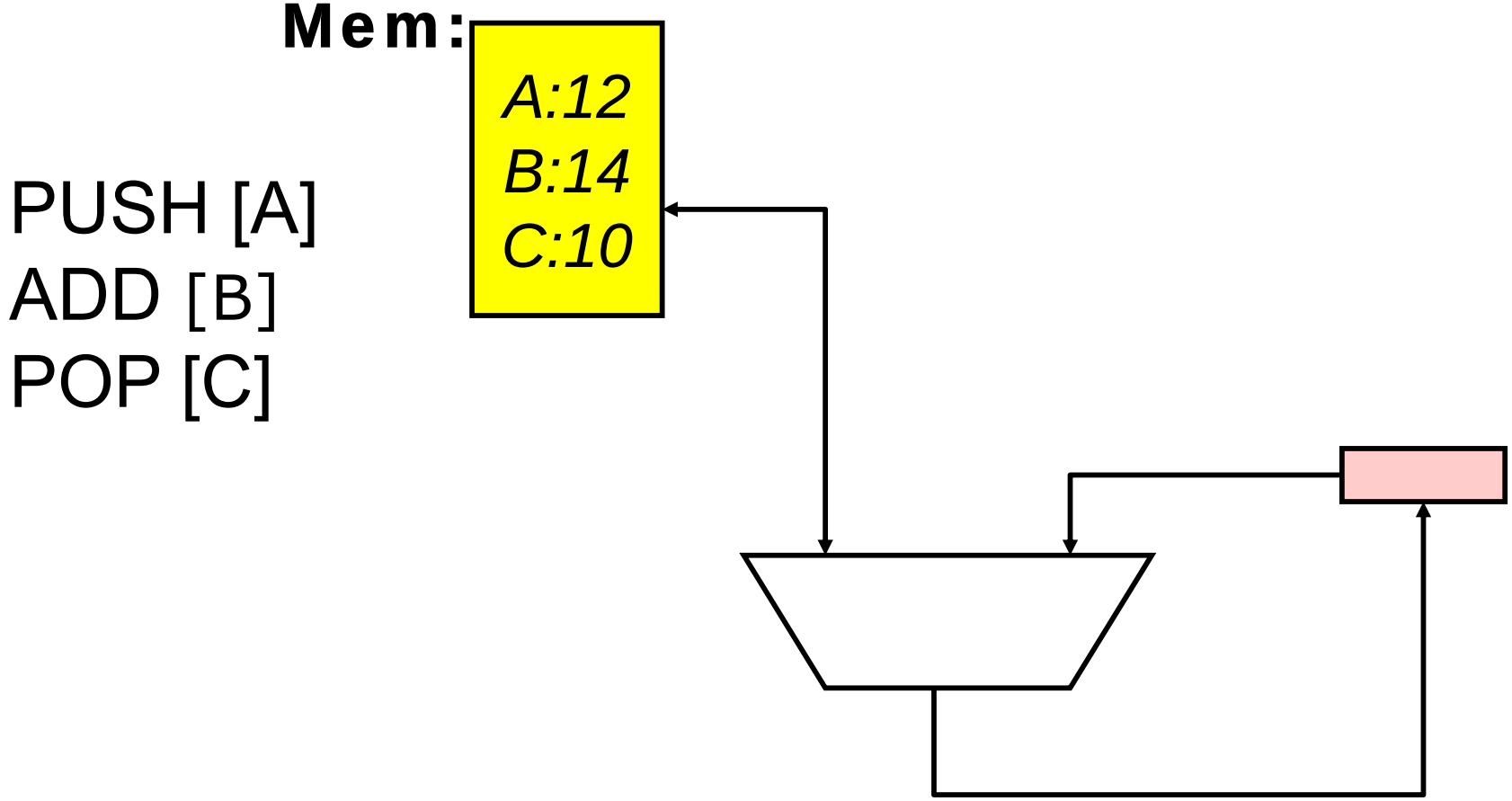
Stack-based

- Implicit operands
- Compact code format (1 instr. = 1byte)
- Simple to implement
- Not optimal for speed!!!

Accumulator-based

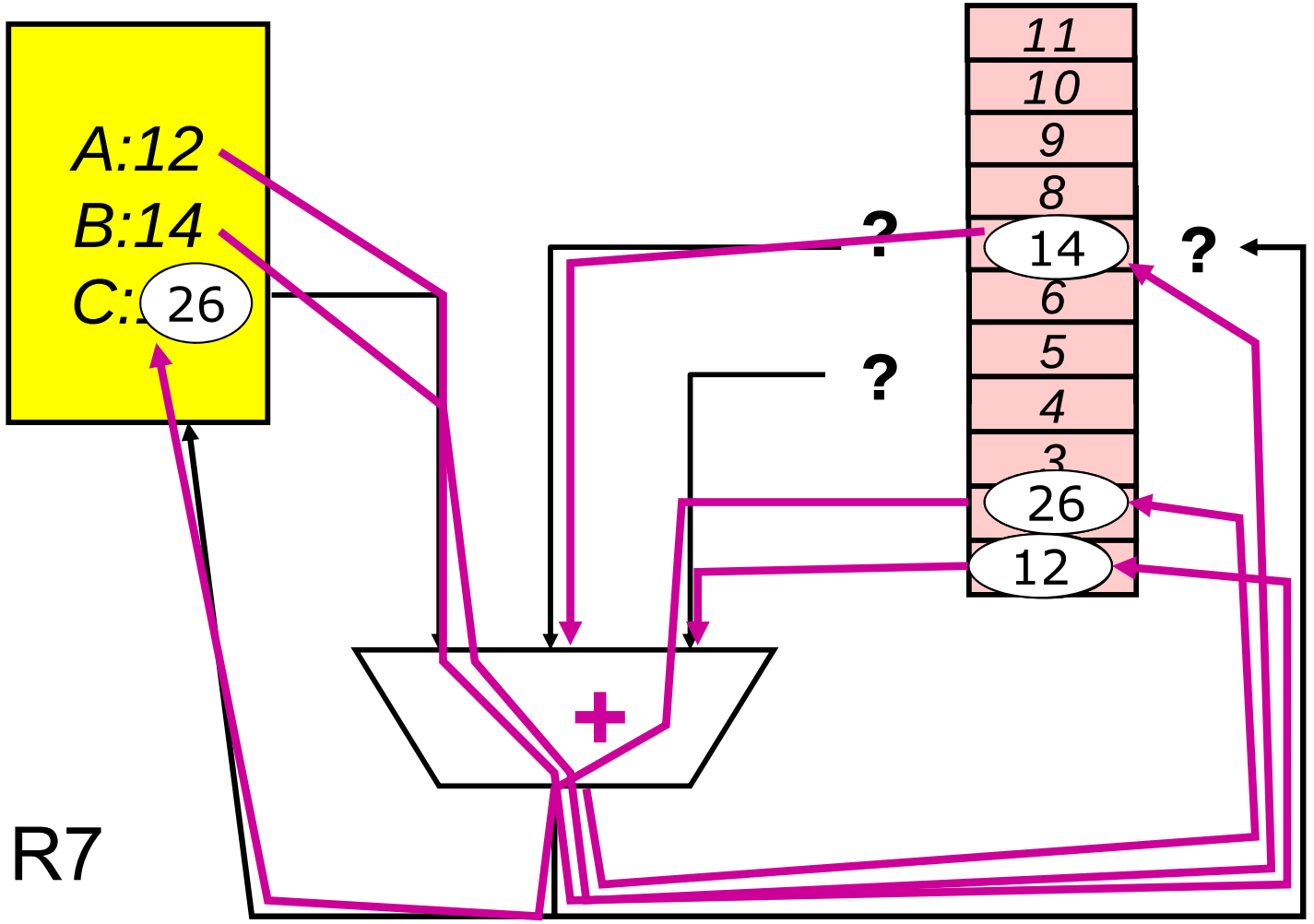
≈ Stack-based with a depth of one

One implicit operand from the accumulator



Register-based machine

Example: C := A + B



→ LD R1, [A]
→ LD R7, [B]
→ ADD R2, R1, R7
→ ST R2, [C]

UPPSALA UNIVERSITET

Register-based

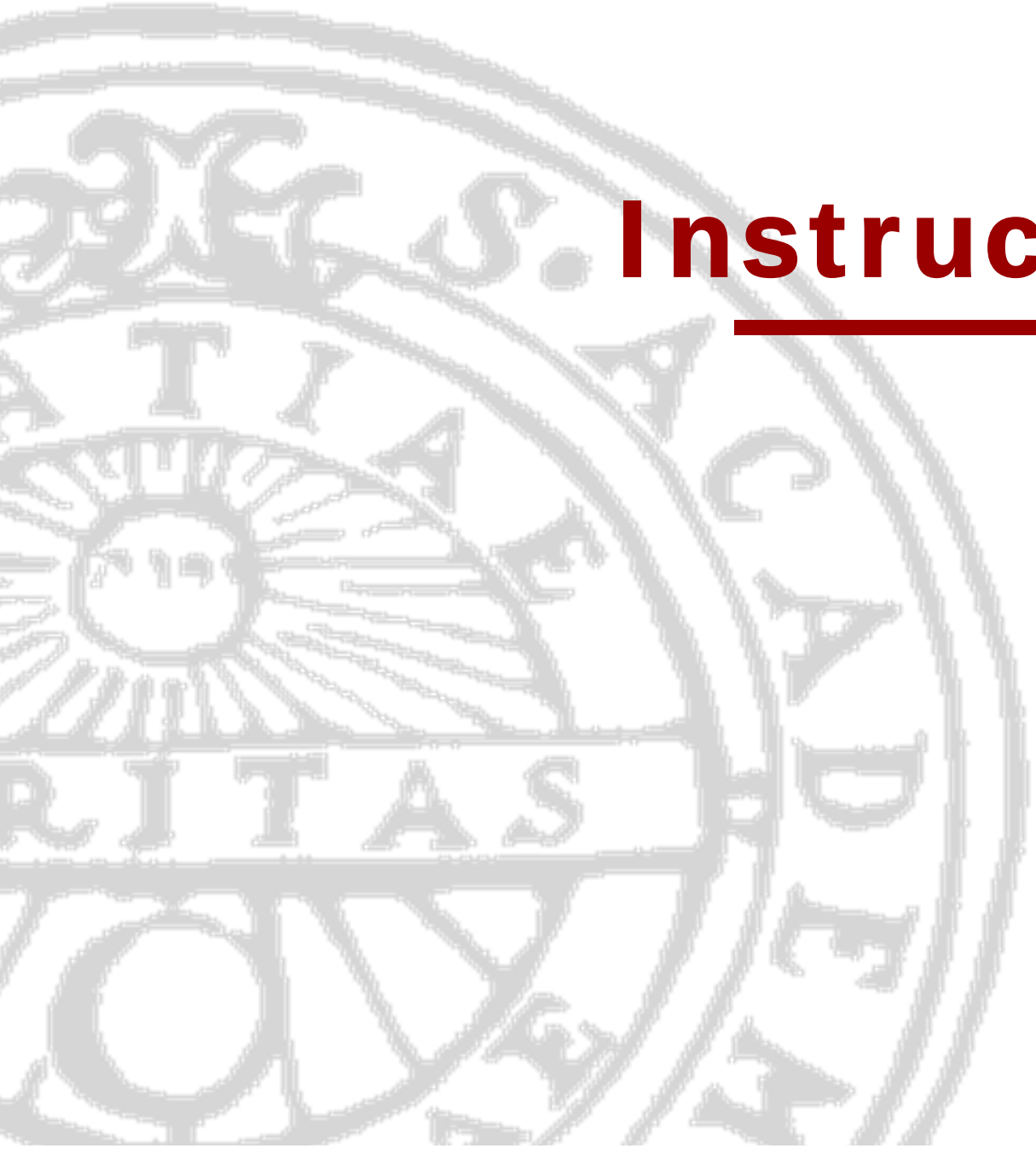
- Explicit operands (i.e., ”registers”)
- Wasteful instr. format (1instr.≈ 4bytes)
- Suits optimizing compilers
- Optimal for speed!!!
- Commercial success:
  - CISC: X86, x86-64
  - RISC: SPARC, Power, MIPS, ARM, (Alpha, HP-PA)
  - VLIW: IA64

UPPSALA UNIVERSITET

Properties of operand models

	Compiler Construction	Implementation Efficiency	Code Size
Stack	+	--	++
Accumulator	--	-	+
Register	++	++	--

AVDARK 2013



Instructions in ISA

UPPSALA UNIVERSITET

Topics for ISA design

- ✓ Operand model
- Instruction format
- Instruction mix
  - Handling of conditional branches
  - Addressing modes
  - Size of immediates
- …
- Implementation independent

AVDARK 2013

Instruction formats

Bit representation in memory:

Operation & no. of operands	Address specifier 1	Address field 1	...	Address specifier n	Address field n

(a) Variable (e.g., VAX)

Operation	Address field 1	Address field 2	Address field 3

(b) Fixed (e.g., DLX, MIPS, Power PC, Precision Architecture, SPARC)

Operation	Address specifier	Address field

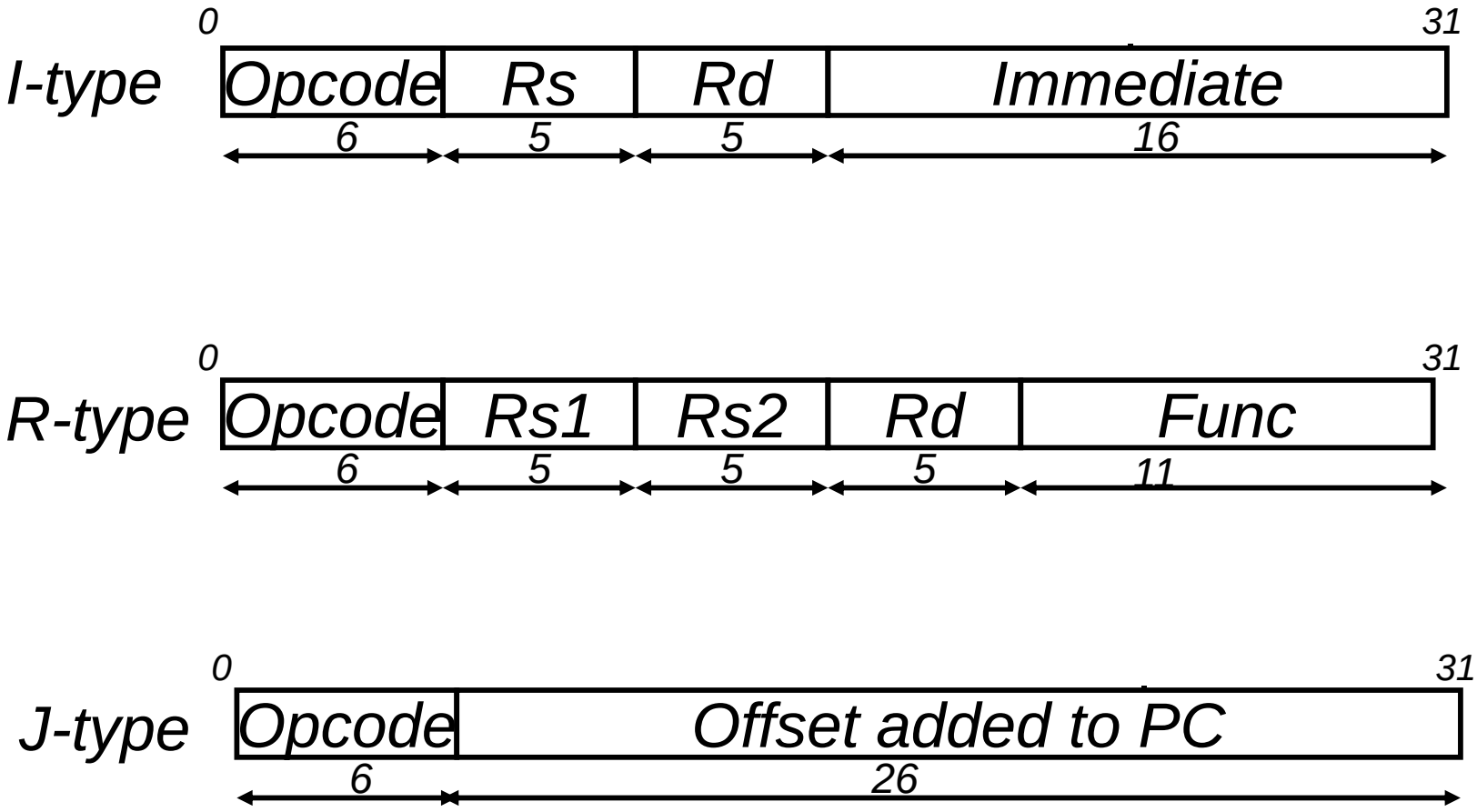
Operation	Address specifier 1	Address specifier 2	Address field

Operation	Address specifier	Address field 1	Address field 2

(c) Hybrid (e.g., IBM 360/70, Intel 80x86)

- A variable instruction format yields compact code but instruction decoding is more complex

Generic Instruction Formats in Book



UPPSALA UNIVERSITET

Generic instructions (Load/Store Architecture)

Instruction type	*Example*	*Meaning*
Load	LW R1,30(R2)	Regs[R1] ← Mem[30+Regs[R2]]
Store	SW 30(R2),R1	Mem[30+Regs[R2]] ← Regs[R1]
ALU	ADD R1,R2,R3	Regs[R1] ← Regs[R2] + Regs[R3]
Control	BEQZ R1,KALLE	if (Regs[R1]==0) PC ← PC + KALLE + 4

UPPSALA UNIVERSITET

Generic ALU Instructions

- Integer arithmetic
  - [add, sub, …] x [signed, unsigned] x [register,immediate]
  - e.g., ADD, ADDI, ADDU, ADDUI, SUB, SUBI, SUBU, SUBUI
- Logical
  - [and, or, xor] x [register, immediate]
  - e.g., AND, ANDI, OR, ORI, XOR, XORI
- Load upper half immediate load
  - It takes two instructions to load a 32 bit immediate

UPPSALA UNIVERSITET

Conditional Branches

Three options:

- Condition Code: Most operations have ”side effects” on set of CC-bits. A branch depends on some CC-bit
- Condition Register. A named register is used to hold the result from a compare instruction. A following branch instruction names the same register.
- Compare and Branch. The compare and the branch is performed in the same instruction.

UPPSALA UNIVERSITET

Conditional Regs: Simple Control

- **Branches if Rx equal to (or if not equal to)**
  - **BEQZ, BNEZ, cmp to register,**
    PC := PC+4+immediate$_{16}$
  - **BFPT, BFPF, cmp to "FP compare bit",**
    PC := PC+4+immediate$_{16}$
- **Jumps**
  - **J: Jump --**
    PC := PC + immediate$_{26}$
  - **JAL: Jump And Link --**
    R31 := PC+4; PC := PC + immediate$_{26}$
  - **JALR: Jump And Link Register --**
    R31 := PC+4; PC := PC + Reg
  - **JR: Jump Register –**
    PC := PC + Reg ("return from JAL or JALR")

UPPSALA UNIVERSITET

Generic FP Instructions

- Floating Point arithmetic
  - [add, sub, mult, div] x [double, single]
  - e.g., ADDD, SUBD, …
- Compares (sets "compare bit")
  - [lt, gt, le, ge, eq, ne] x [double, immediate]
  - e.g., LTD, GEF, …
- Convert from/to integer, Fpregs
  - CVTF2I, CVTF2D, CVTI2D, …

UPPSALA UNIVERSITET

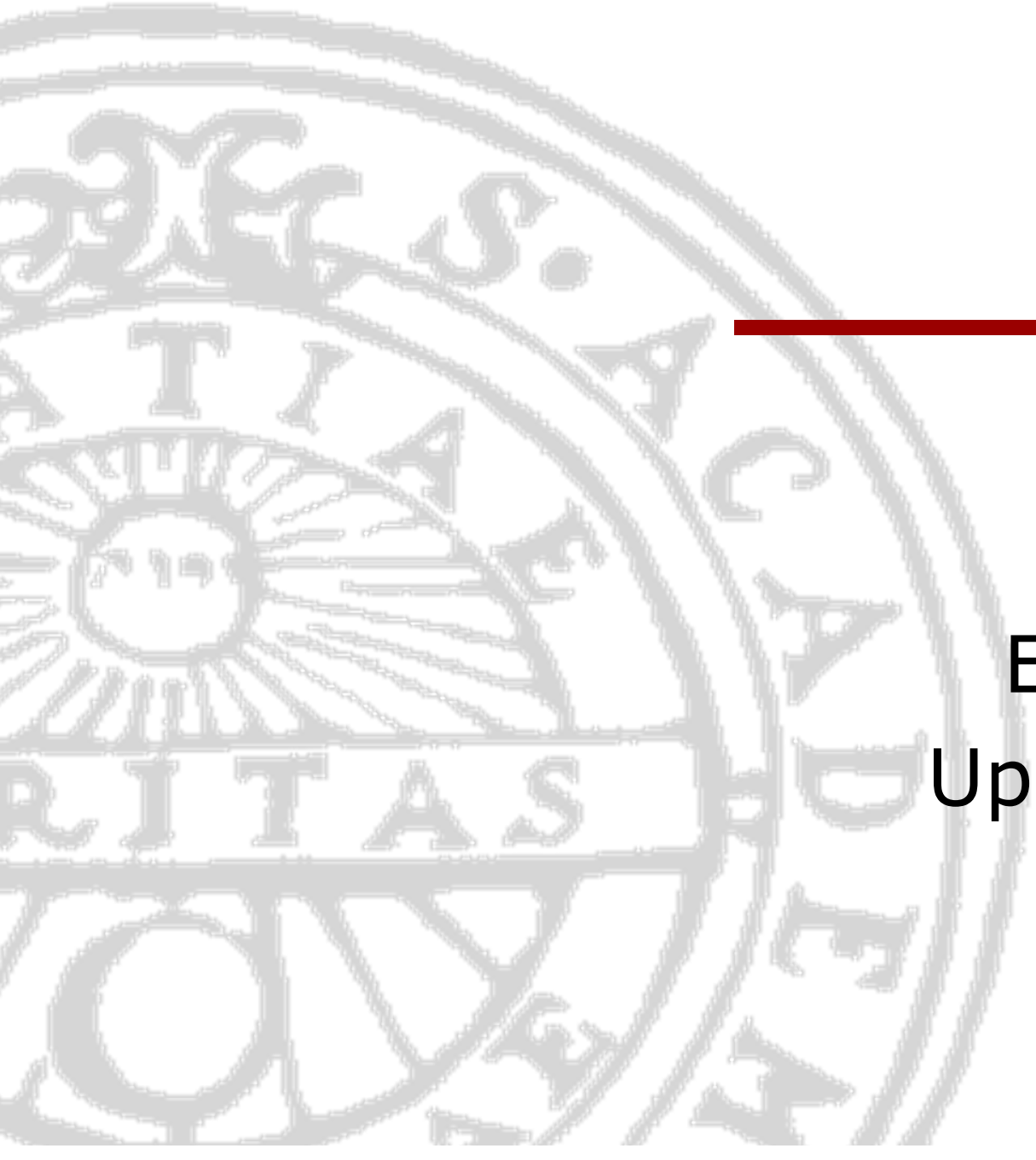
Generic instructions (Load/Store Architecture)

Instruction type	*Example*	*Meaning*
Load	LW R1,30(R2)	Regs[R1] ← Mem[30+Regs[R2]]
Store	SW 30(R2),R1	Mem[30+Regs[R2]] ← Regs[R1]
ALU	ADD R1,R2,R3	Regs[R1] ← Regs[R2] + Regs[R3]
Control	BEQZ R1,KALLE	if (Regs[R1]==0) PC ← PC + KALLE + 4

UPPSALA UNIVERSITET

RISC vs. CISC

- Reduced Instruction Set Computer (RISC)
  - LD/ST architecture
  - Fixed-length instruction format
  - Few addressing modes
  - General-purpose registers
  - Also added various suggestions
    - No interlocking pipeline stages (MIPS, Stanford)
    - Register window, delayed branches (RISC, UCB)

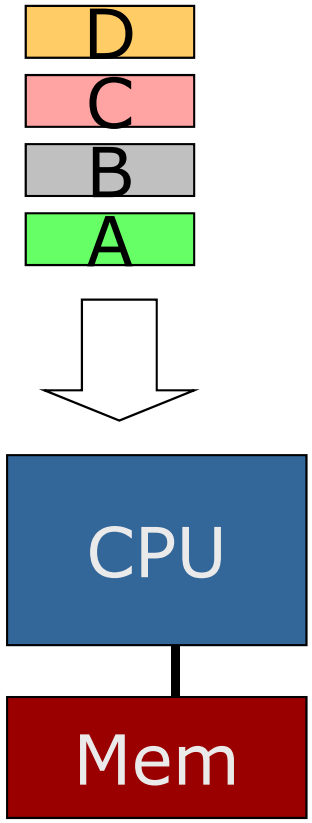


Pipelines

Erik Hagersten
Uppsala University

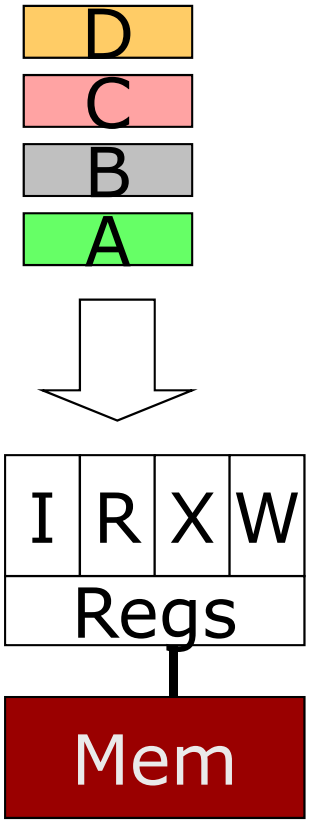
Lifting the CPU hood (simplified…)

Instructions:

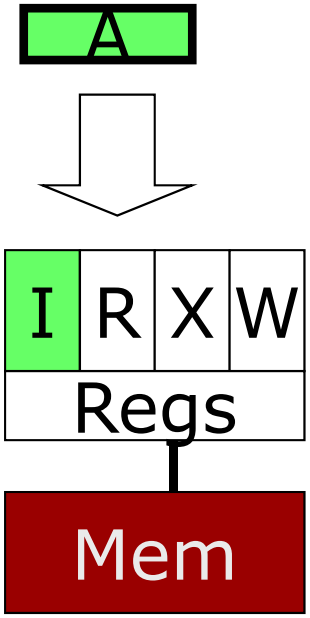


Pipeline

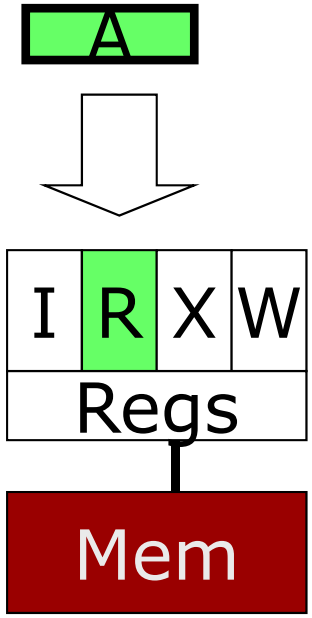
Instructions:



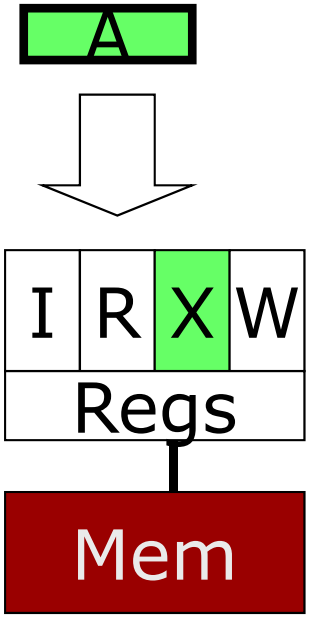
Pipeline



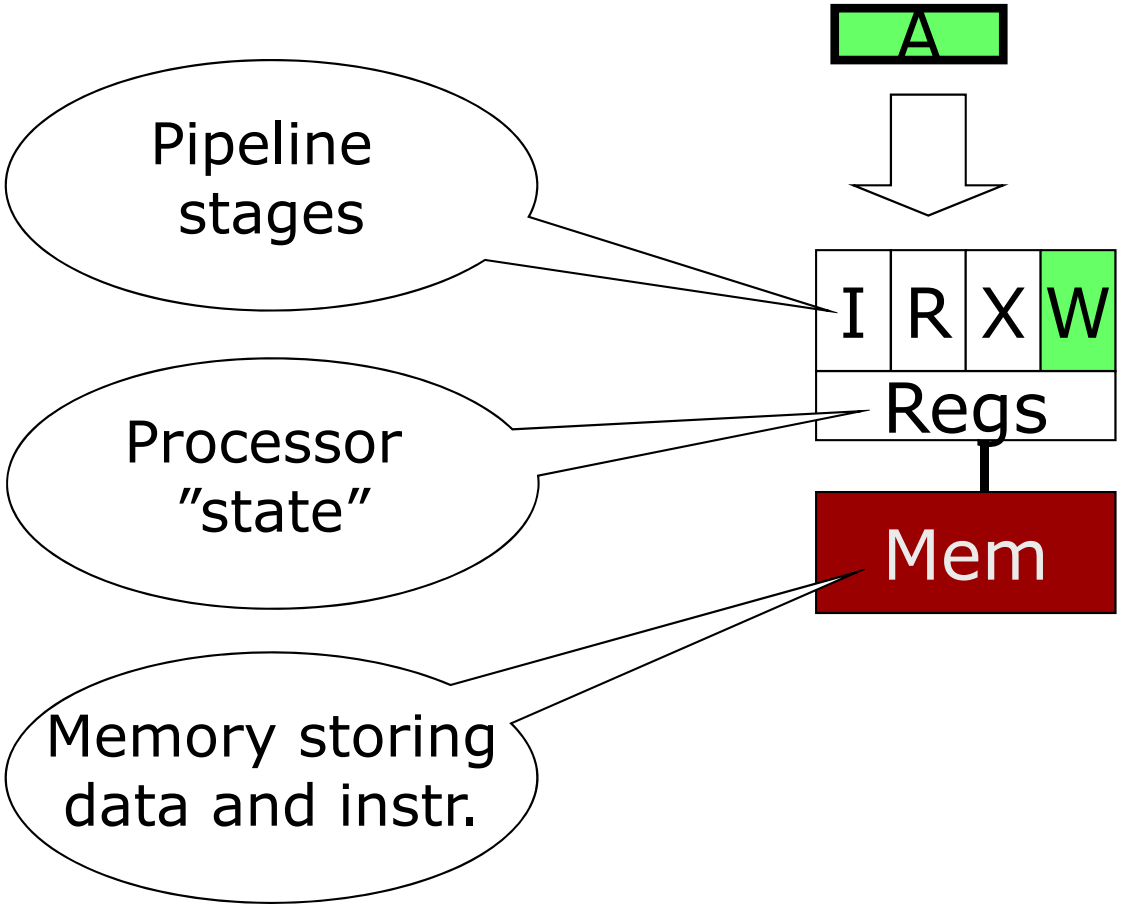
Pipeline



Pipeline

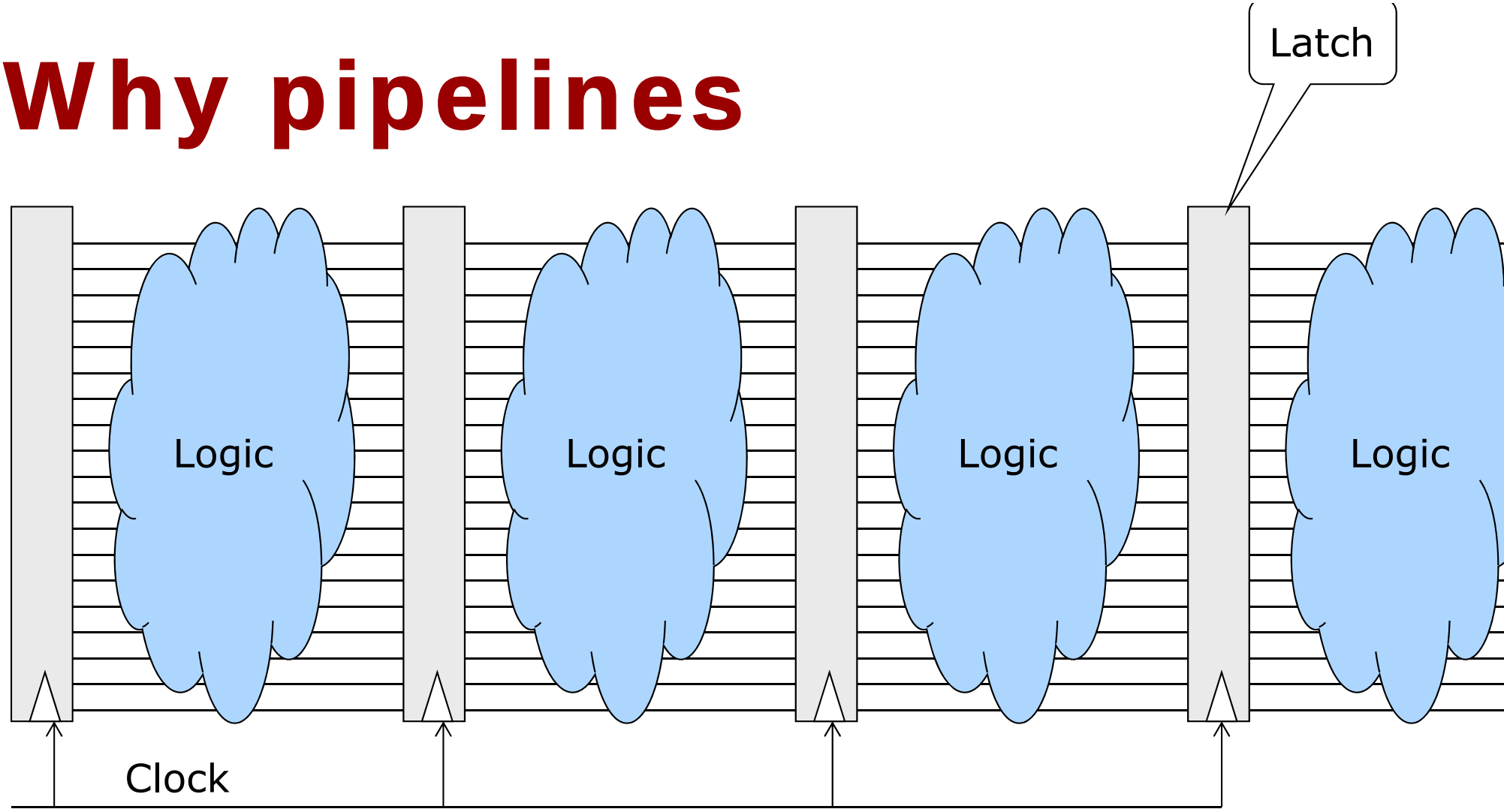


Pipeline:



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

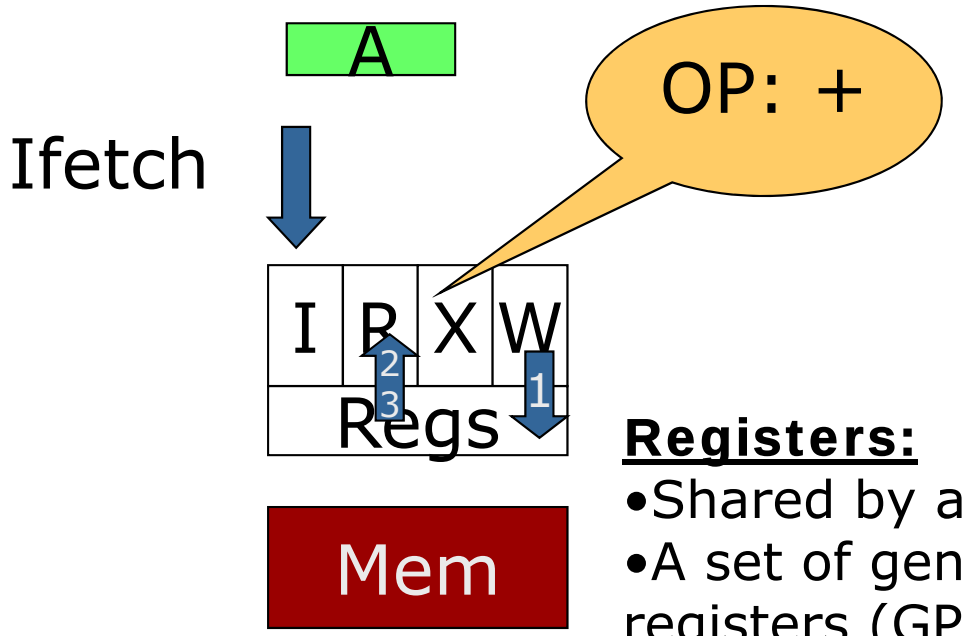
Pileline delay: Latency from clock to stable output signal

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

Imbalance between the pipeline stages

EXAMPLE: pipeline implementation

Add R1, R2, R3

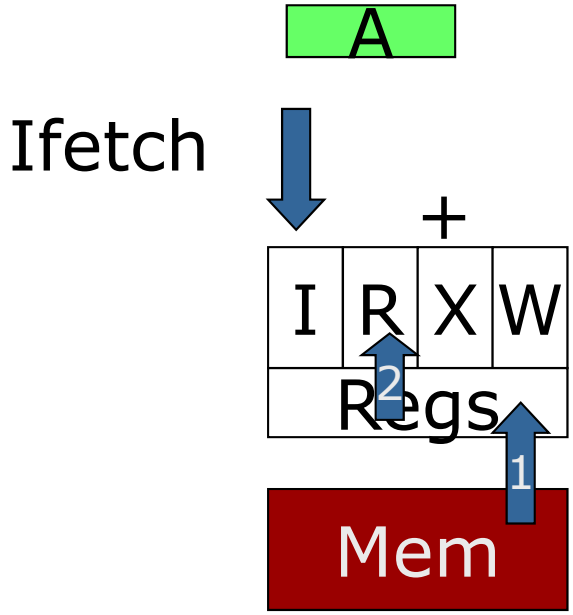


Registers:

- Shared by all pipeline stages
- A set of general purpose registers (GPRs)
- Some specialized registers (e.g., PC)

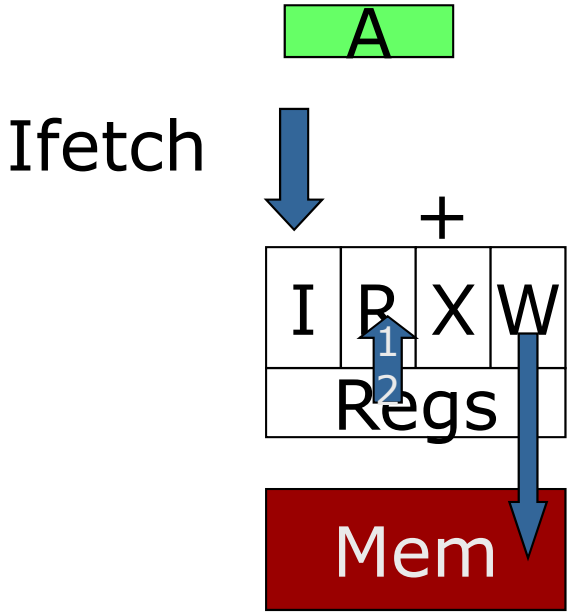
Load Operation:

LD R1, mem[cnst+R2]



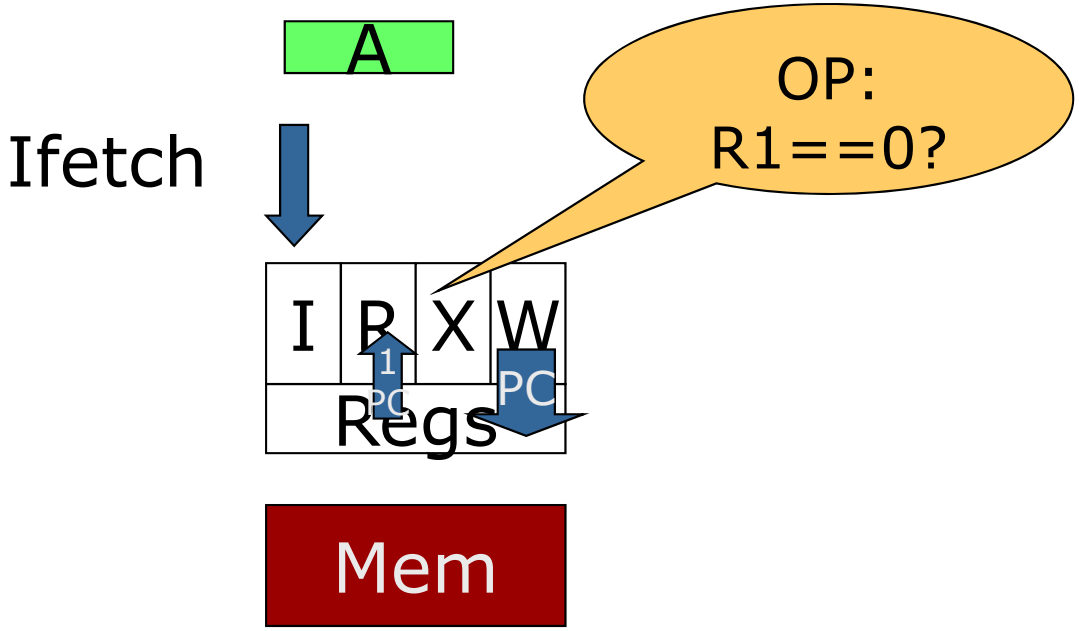
Store Operation:

ST mem[cnst+R1], R2

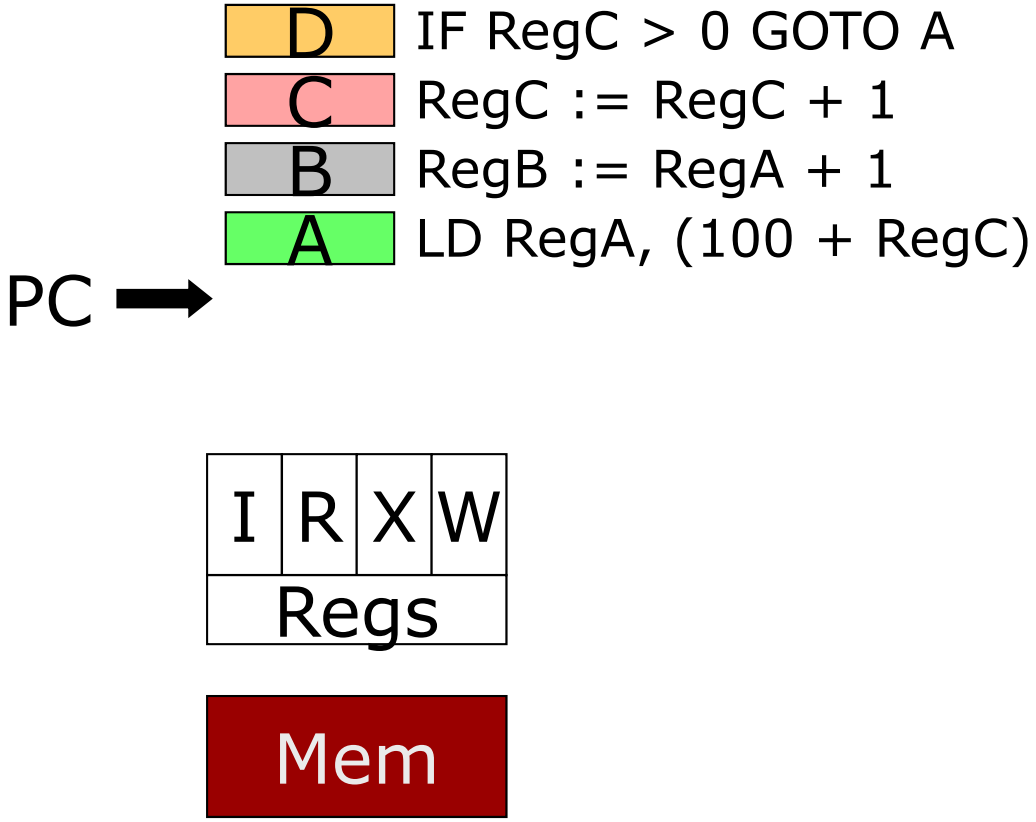


EXAMPLE: Branch to R2 if R1 == 0

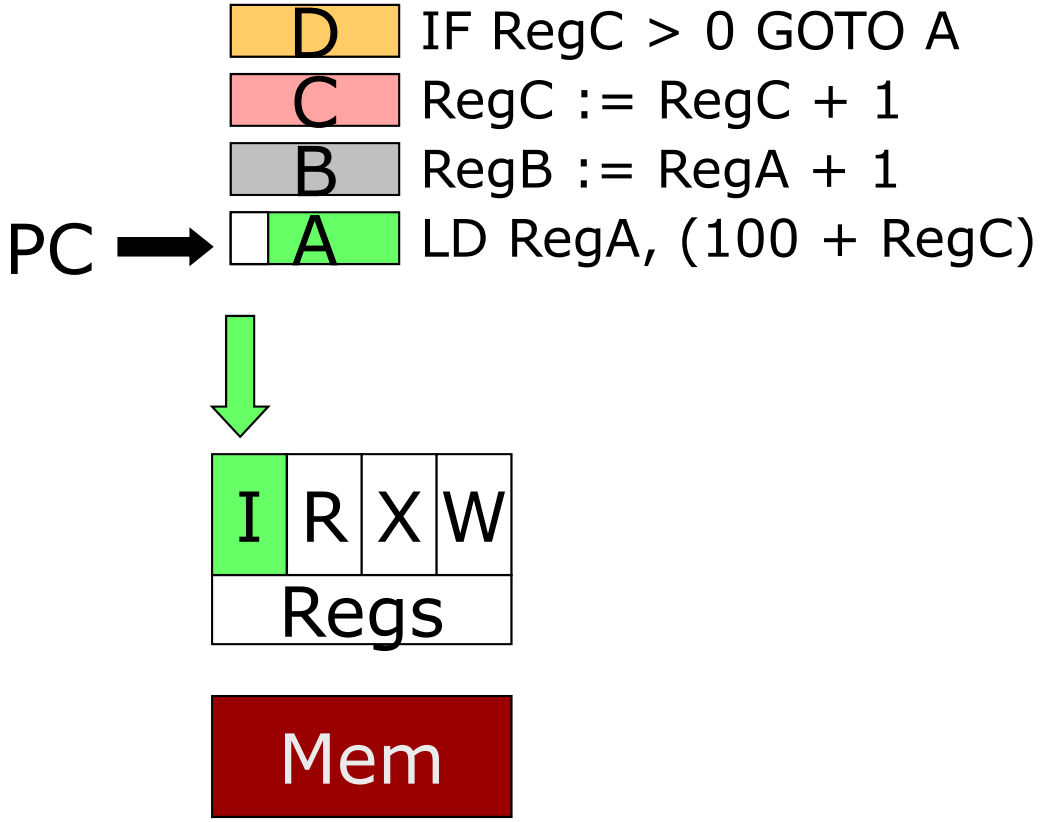
BEQZ R1, #OFFSET



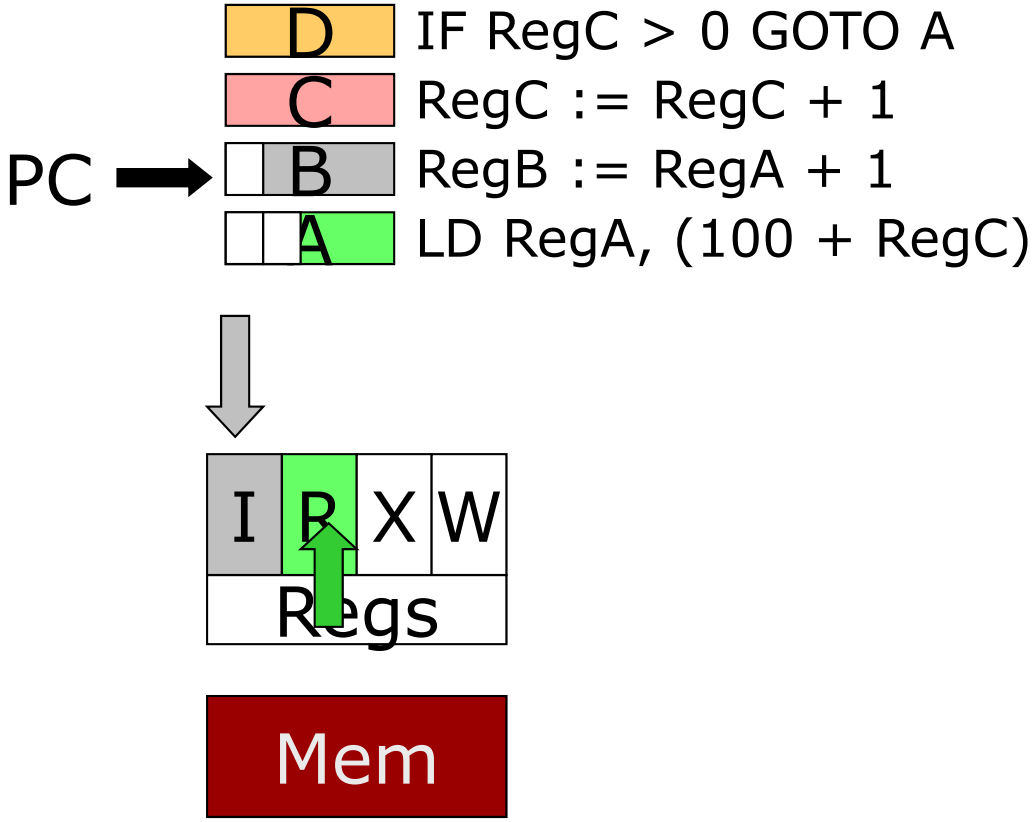
Initially


D
IF RegC > 0 GOTO A
C
RegC := RegC + 1
B
RegB := RegA + 1
A
LD RegA, (100 + RegC)
PC
I
R
X
W
Regs
Mem

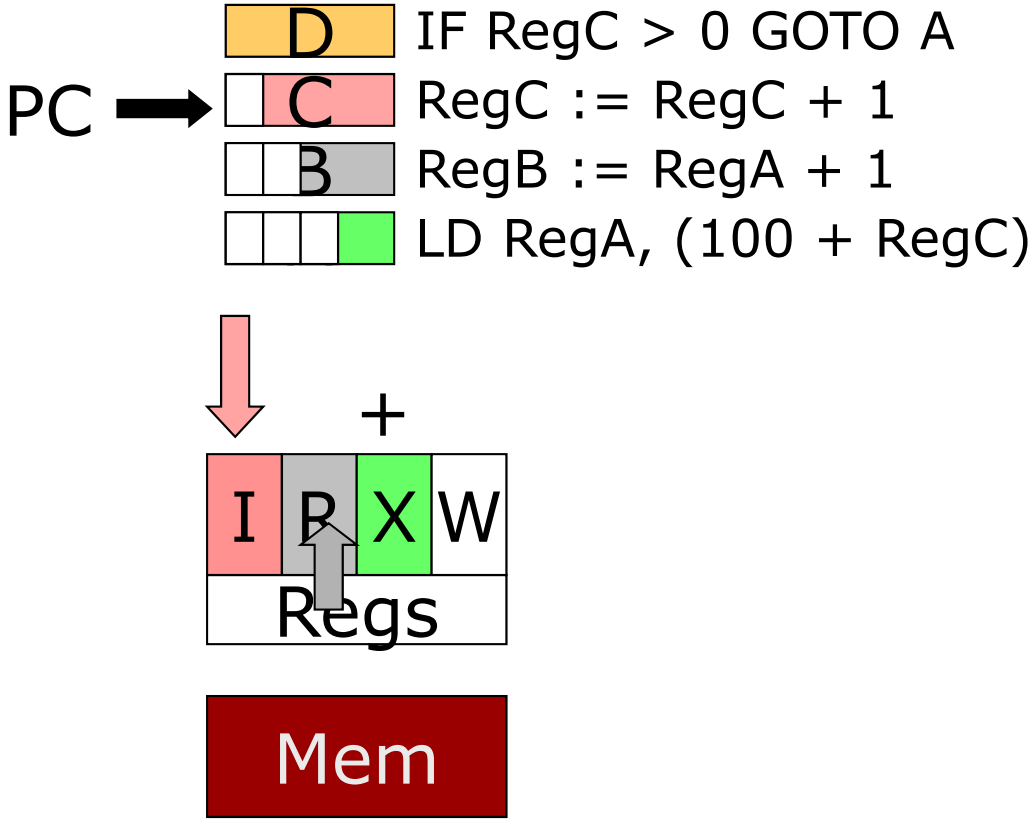
Cycle 1


D
IF RegC > 0 GOTO A
C
RegC := RegC + 1
B
RegB := RegA + 1
PC
A
LD RegA, (100 + RegC)
I
R
X
W
Regs
Mem

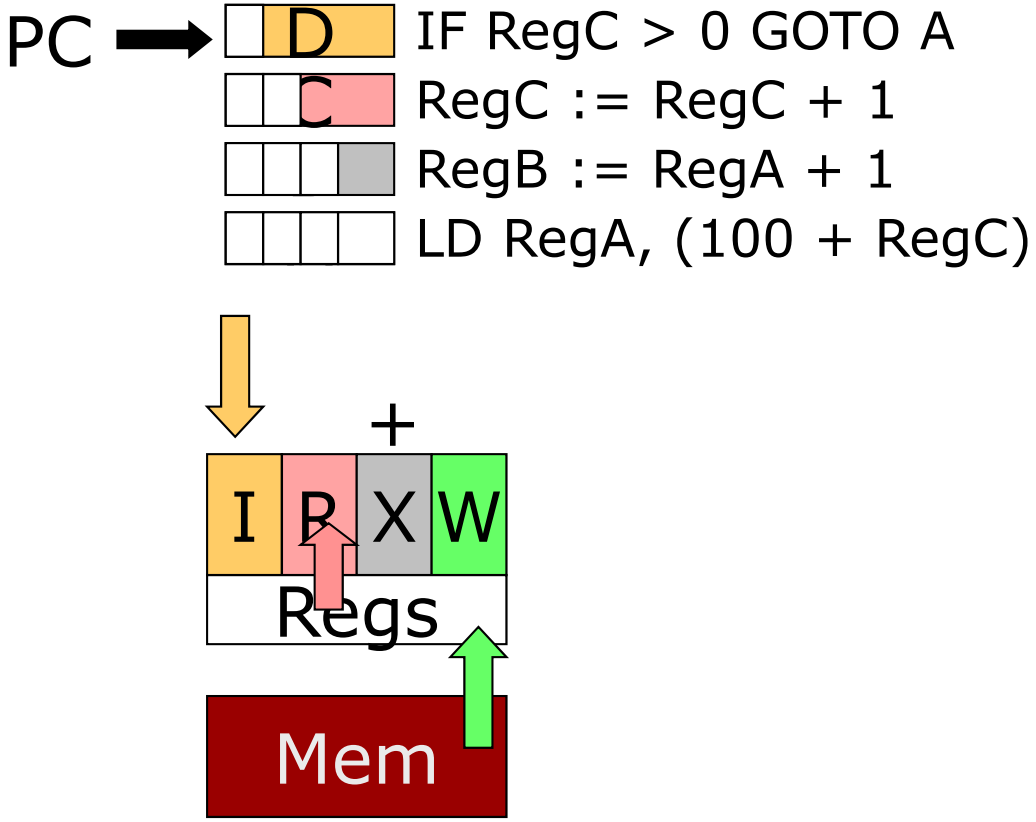
Cycle 2


D
C
B
A
IF RegC > 0 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
PC
I
R
X
W
Regs
Mem

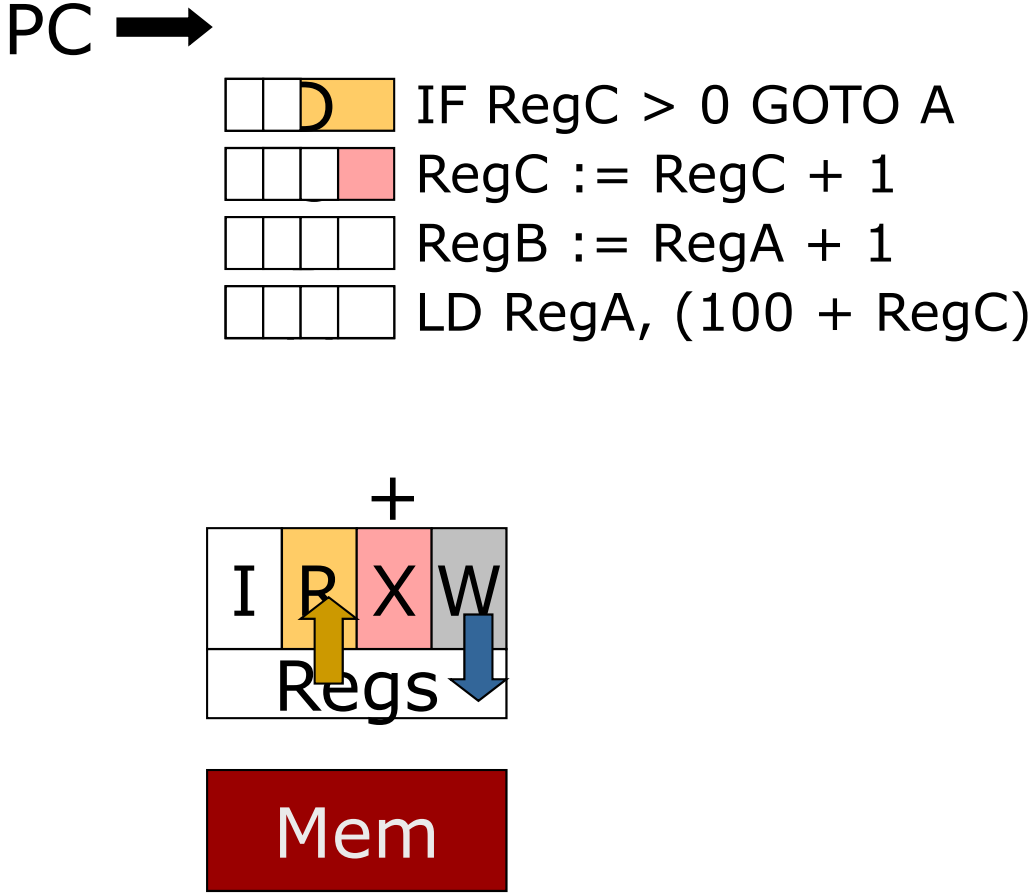
Cycle 3


D
C
B
PC
IF RegC > 0 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
+
I
R
X
W
Regs
Mem

Cycle 4

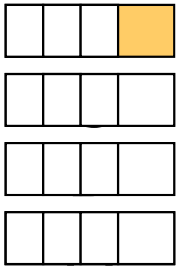

PC
D
C
IF RegC > 0 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
+
I
R
X
W
Regs
Mem

Cycle 5

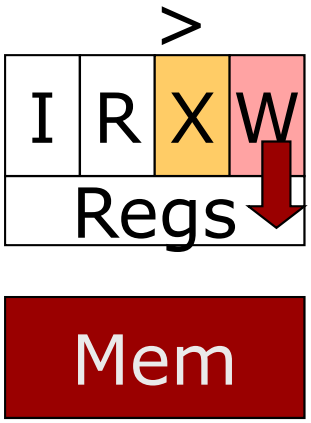

PC
D
IF RegC > 0 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
+
I
R
X
W
Regs
Mem

Cycle 6

PC ➡



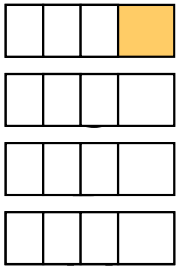
IF RegC > 0 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)

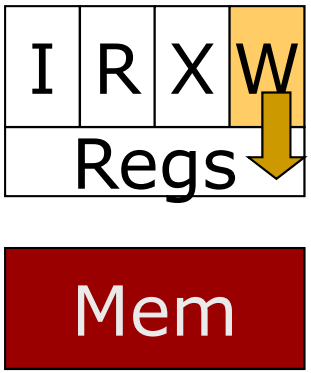


UPPSALA UNIVERSITET

Cycle 7

PC ➞

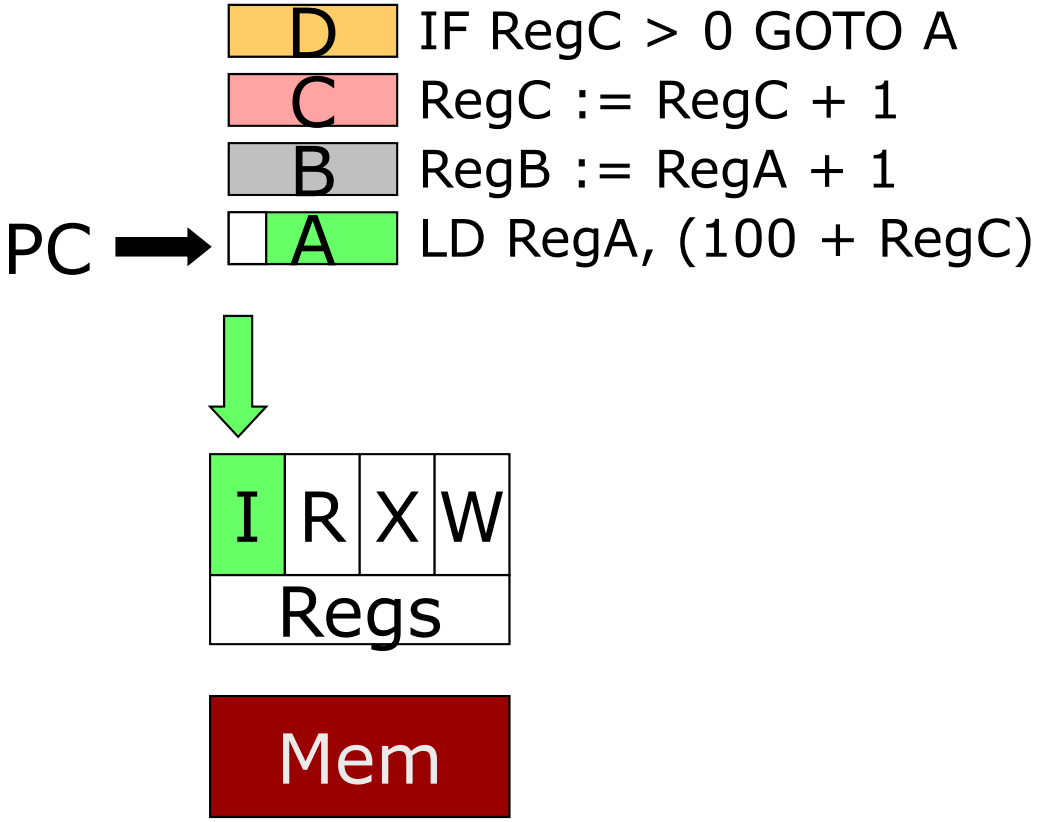


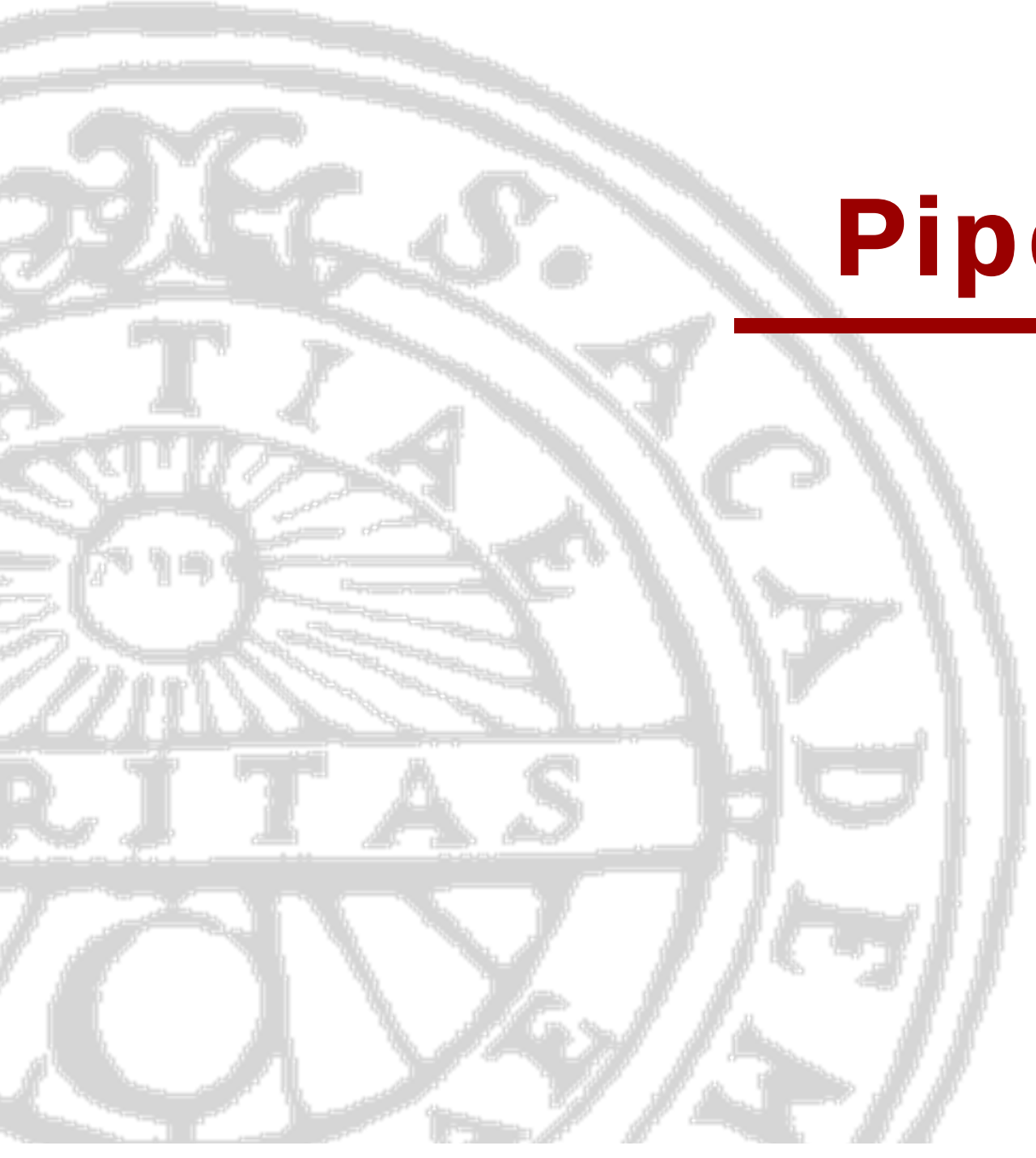


PC ← (PC + 4 + ”A”)

AVDARK 2013

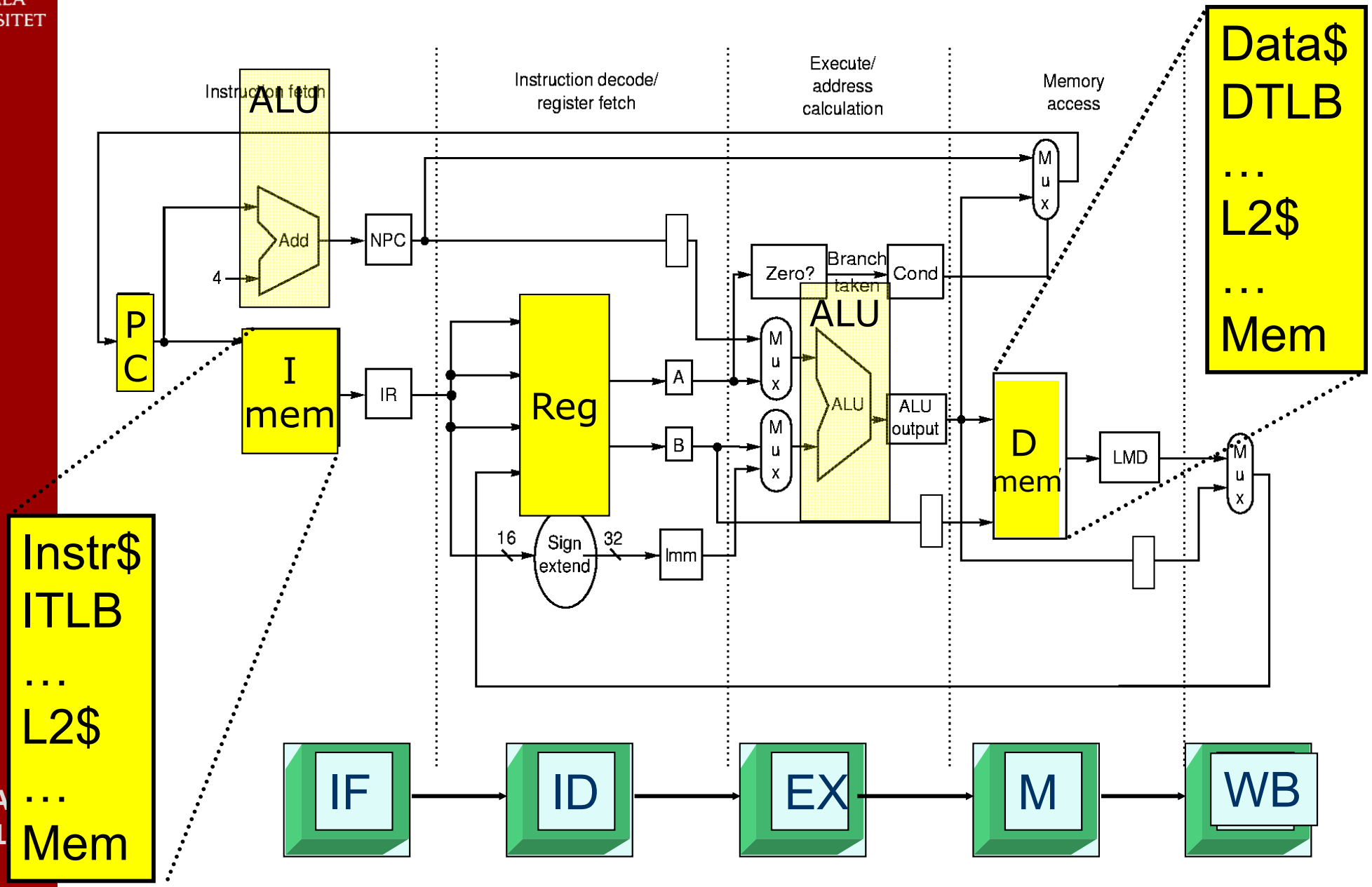
Cycle 8


D
IF RegC > 0 GOTO A
C
RegC := RegC + 1
B
RegB := RegA + 1
PC
A
LD RegA, (100 + RegC)
I
R
X
W
Regs
Mem

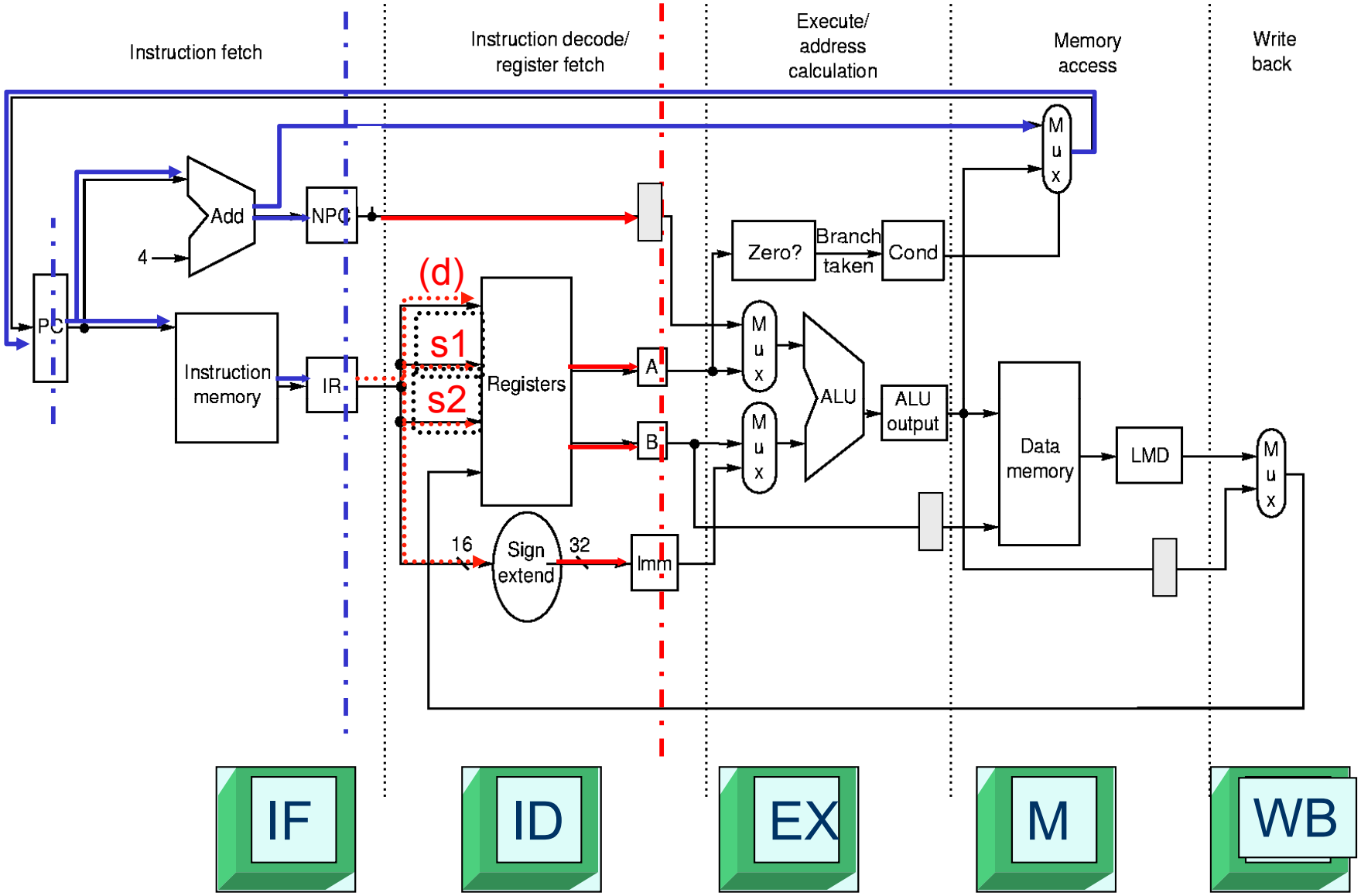


Pipeline design

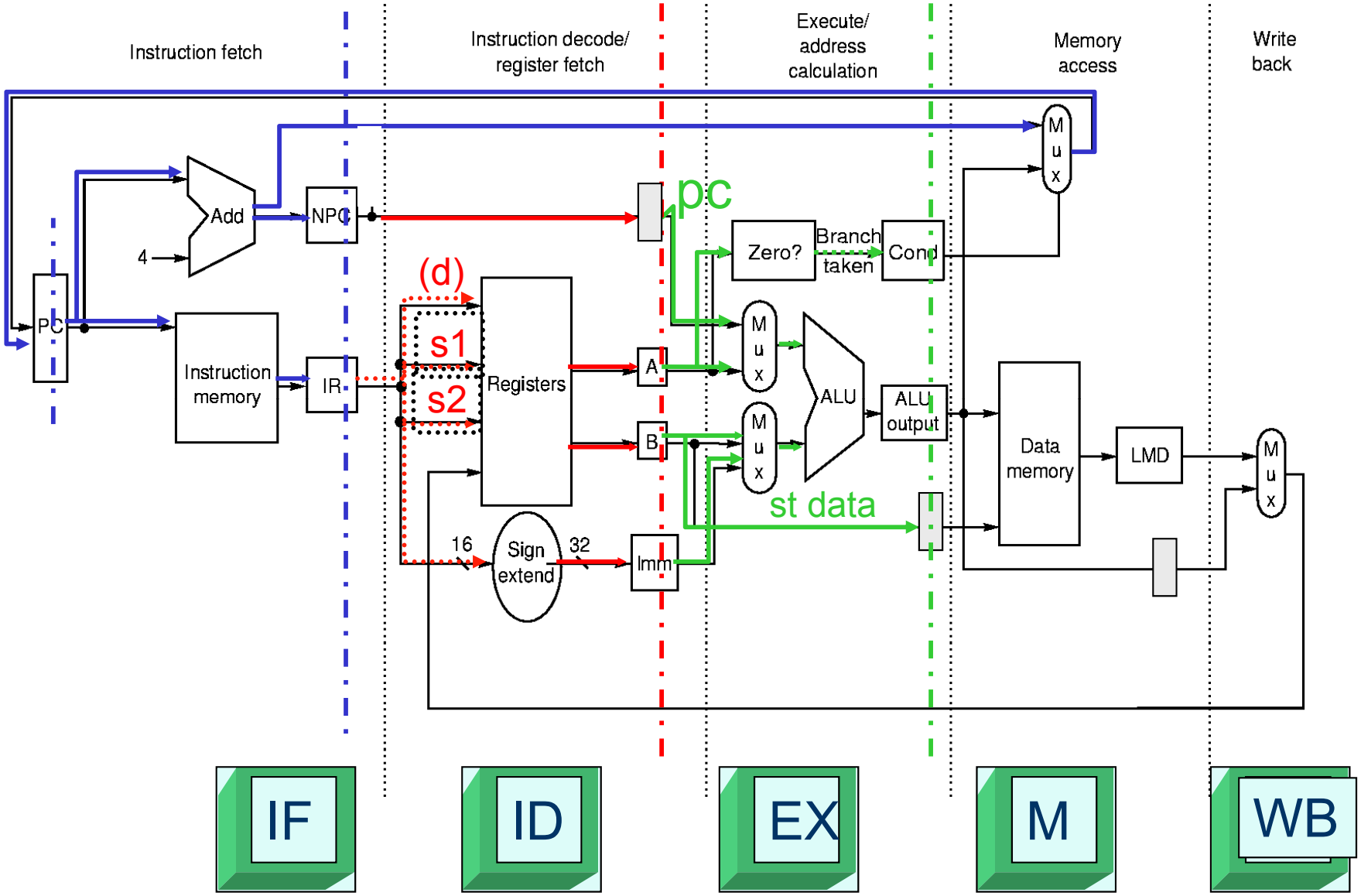
Example: 5-stage pipeline



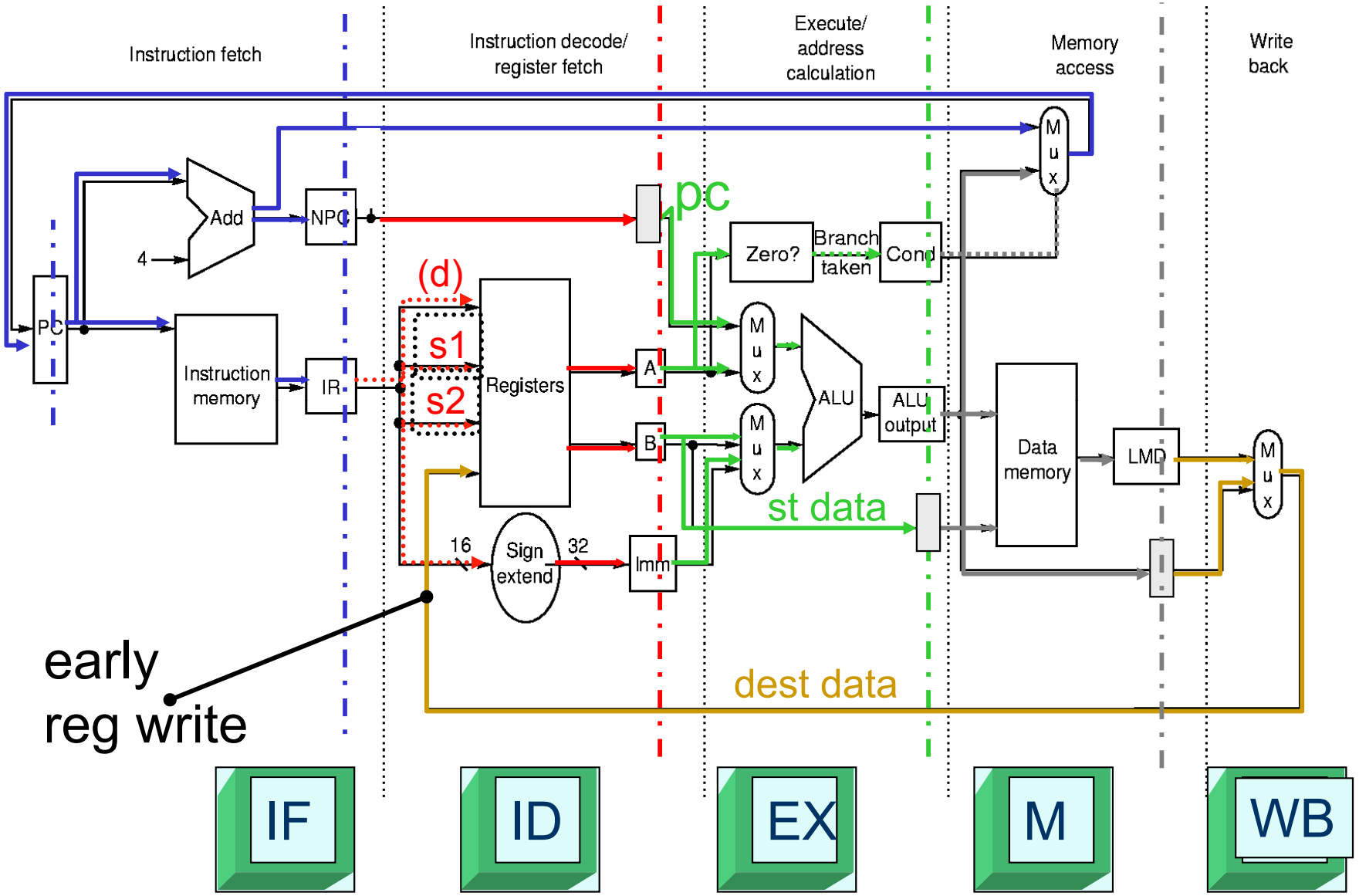
Example: 5-stage pipeline



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.

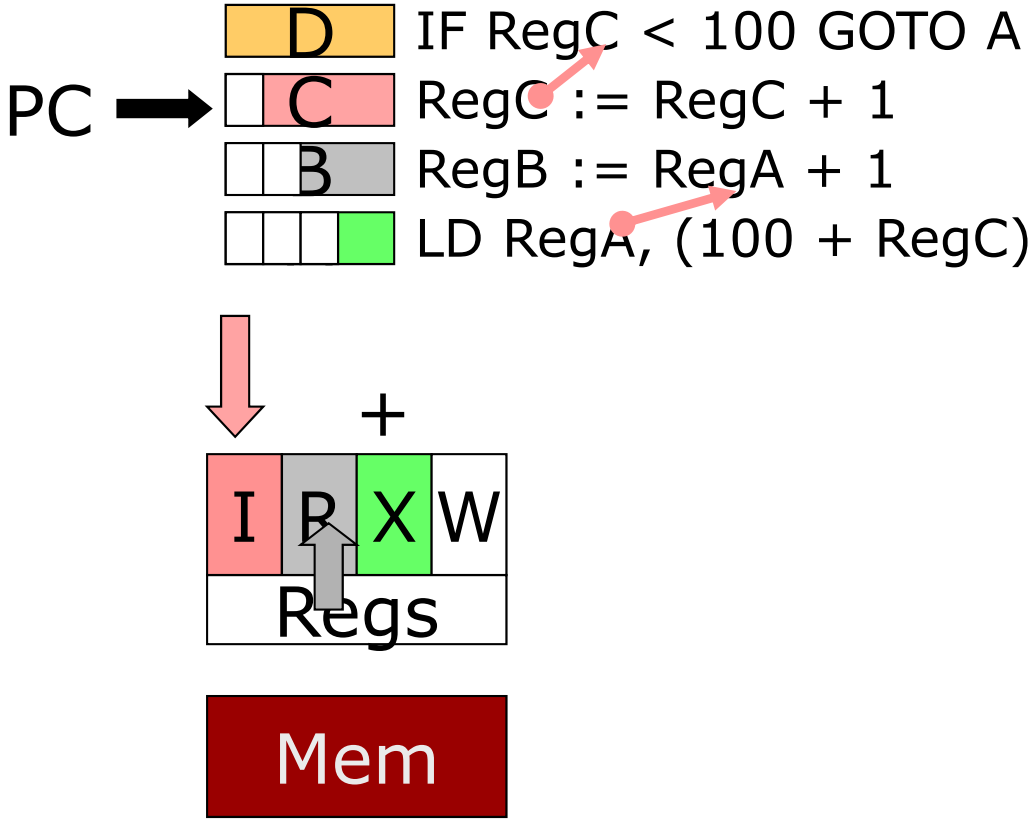
Opi reads new A

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.

Cycle 3


D
C
B
PC
IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
+
I
R
X
W
Regs
Mem

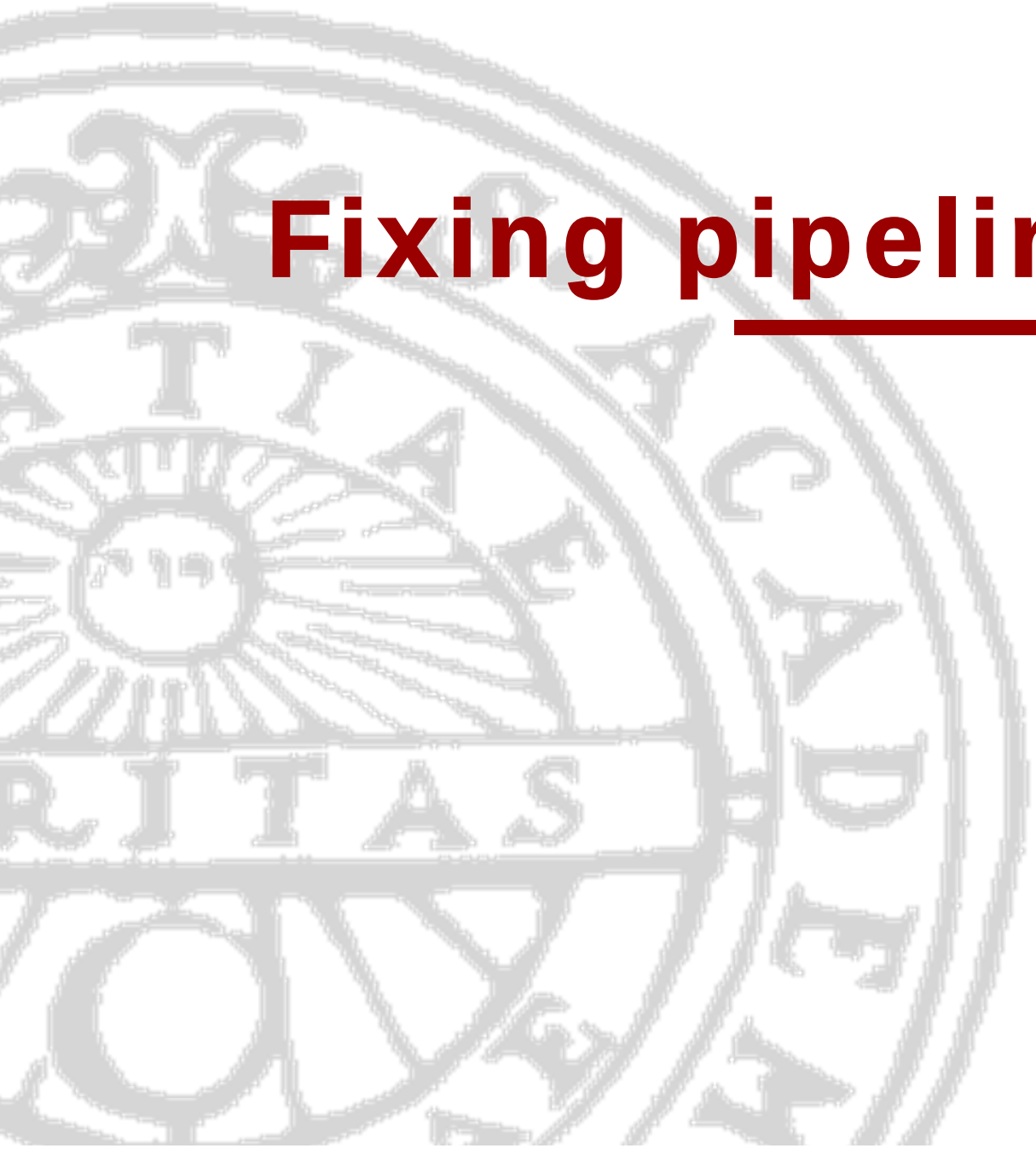
UPPSALA UNIVERSITET

Hazard avoidance techniques

Static techniques (compiler): code scheduling to avoid hazards

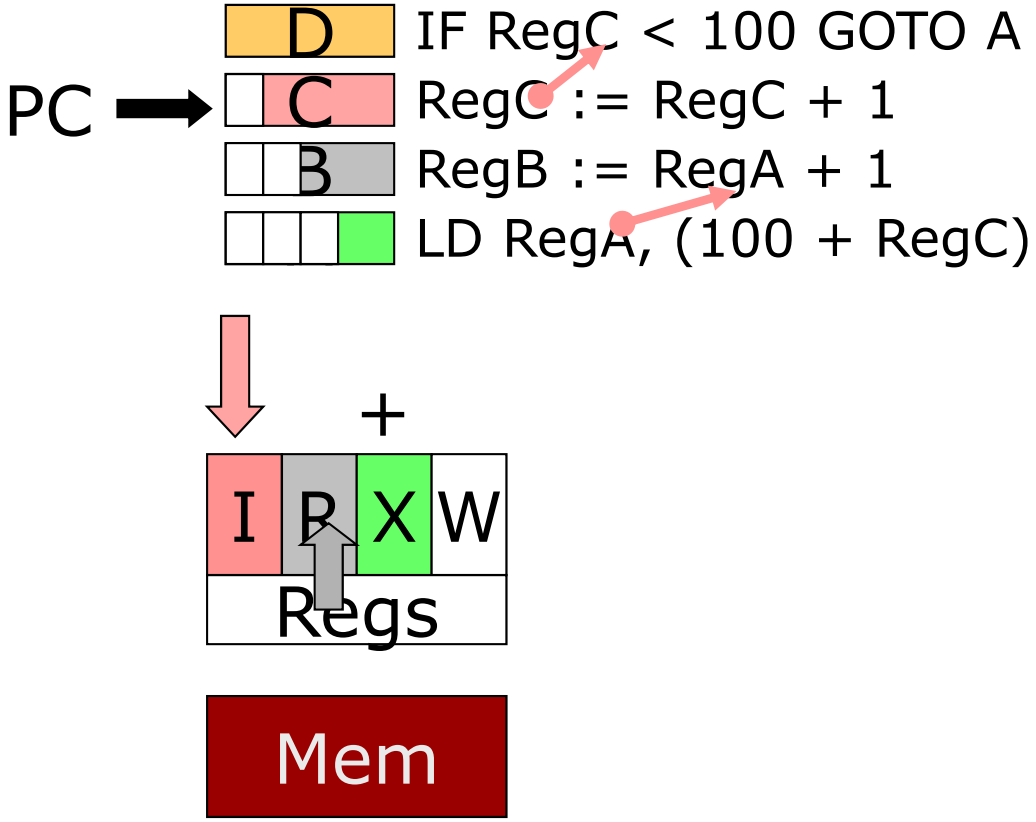
Dynamic techniques: hardware mechanisms to eliminate or reduce impact of hazards (e.g., out-of-order stuff)

Hybrid techniques: rely on compiler as well as hardware techniques to resolve hazards (e.g. VLIW support – later)



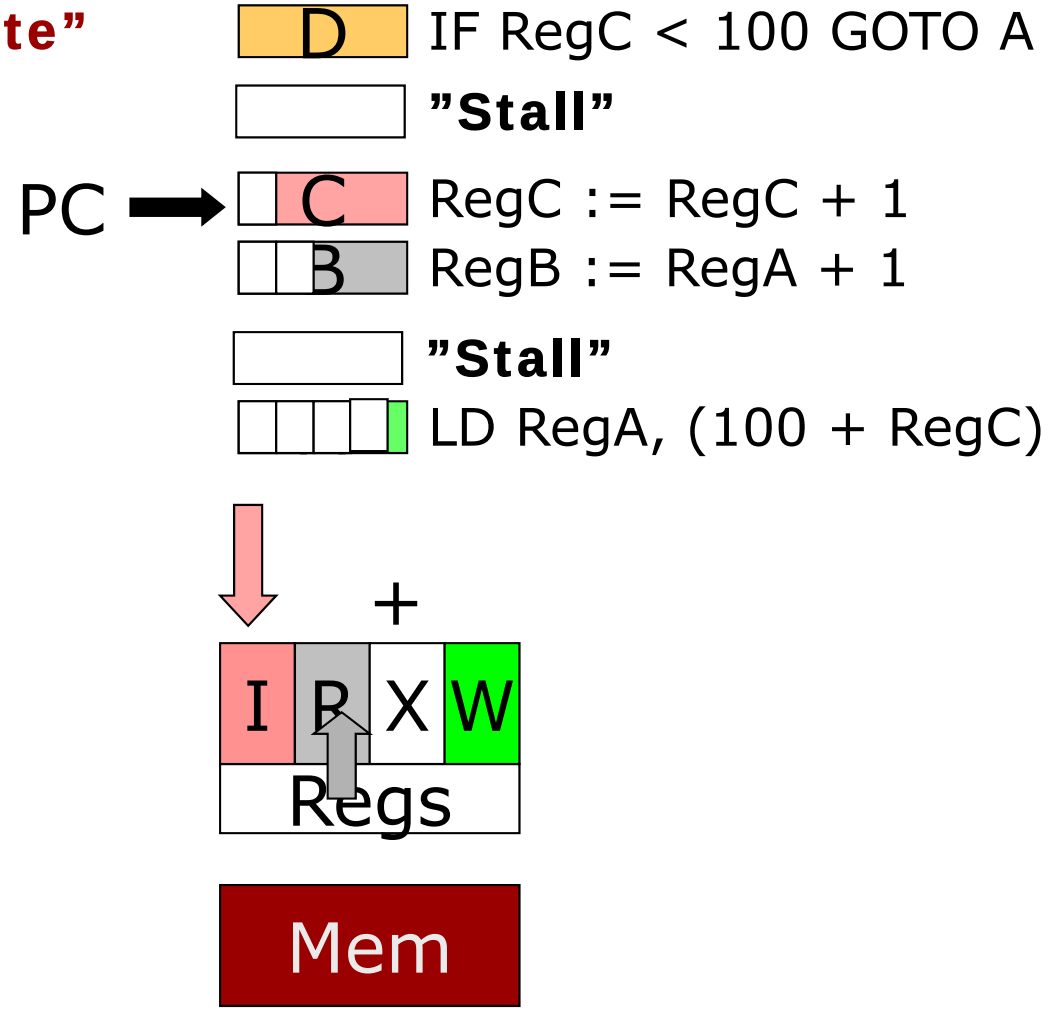
Fixing pipeline problems

Cycle 3

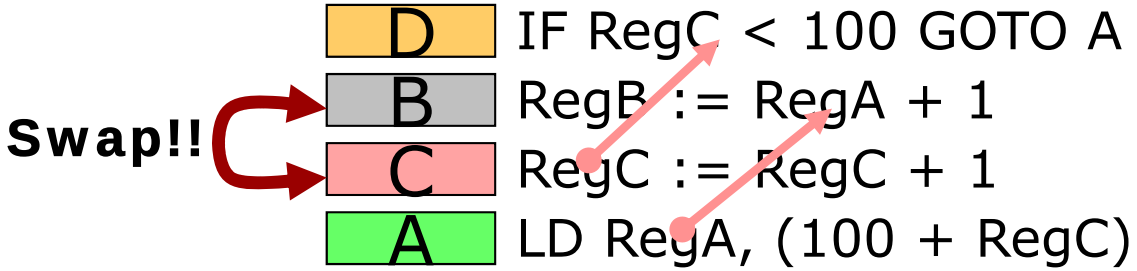

D
C
B
PC
IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
+
I
R
X
W
Regs
Mem

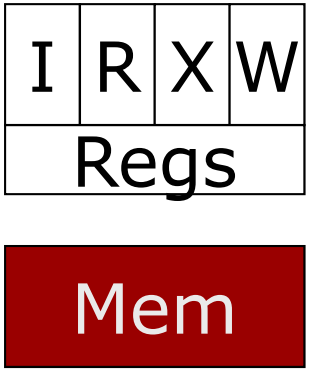
Cycle 3

assuming “early write”

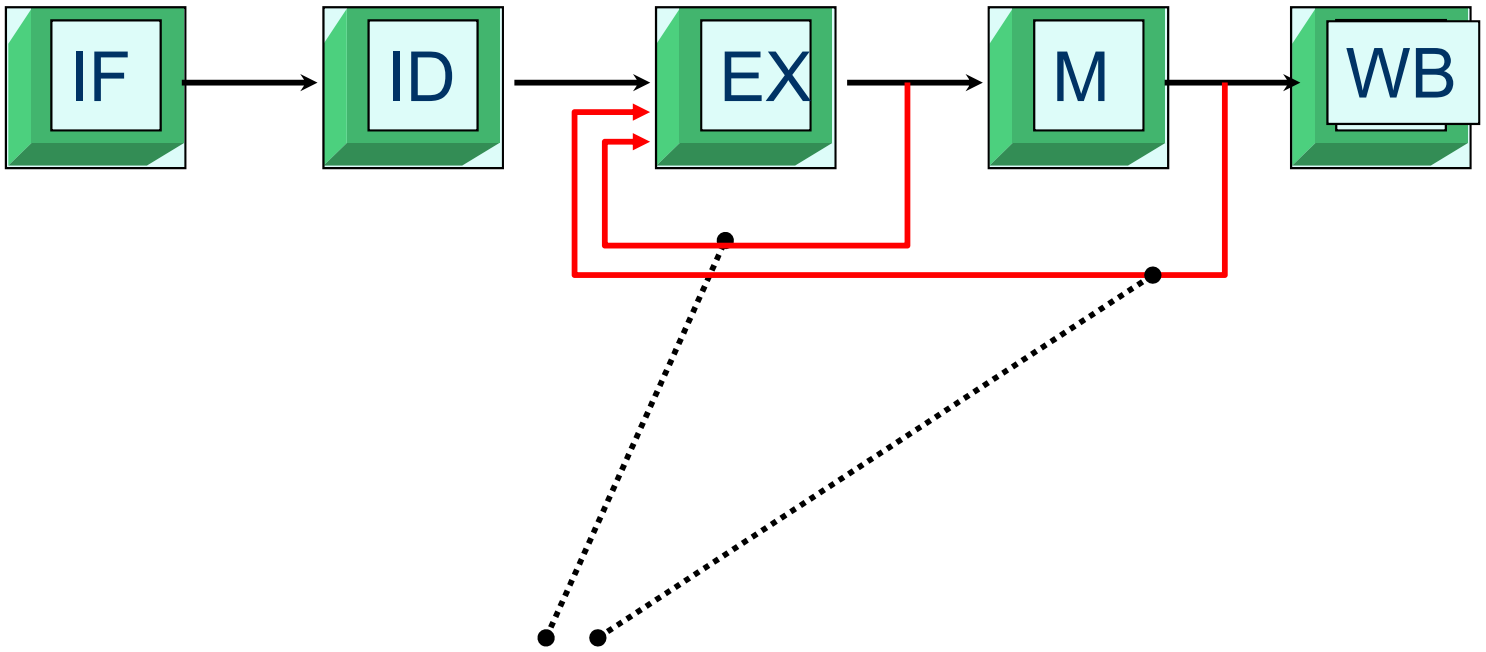


Fix alt1: code scheduling



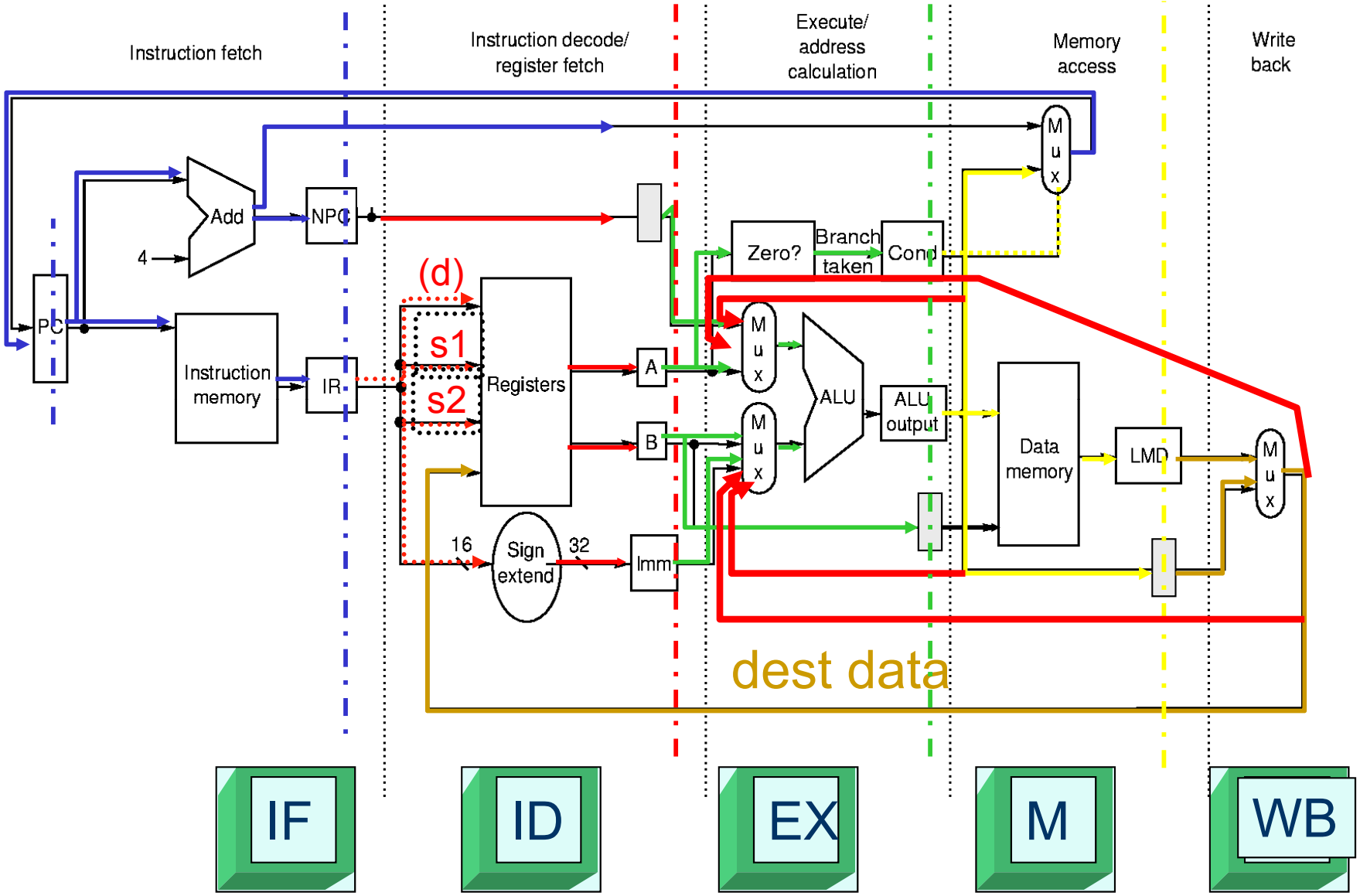


Fix alt2: Bypass hardware

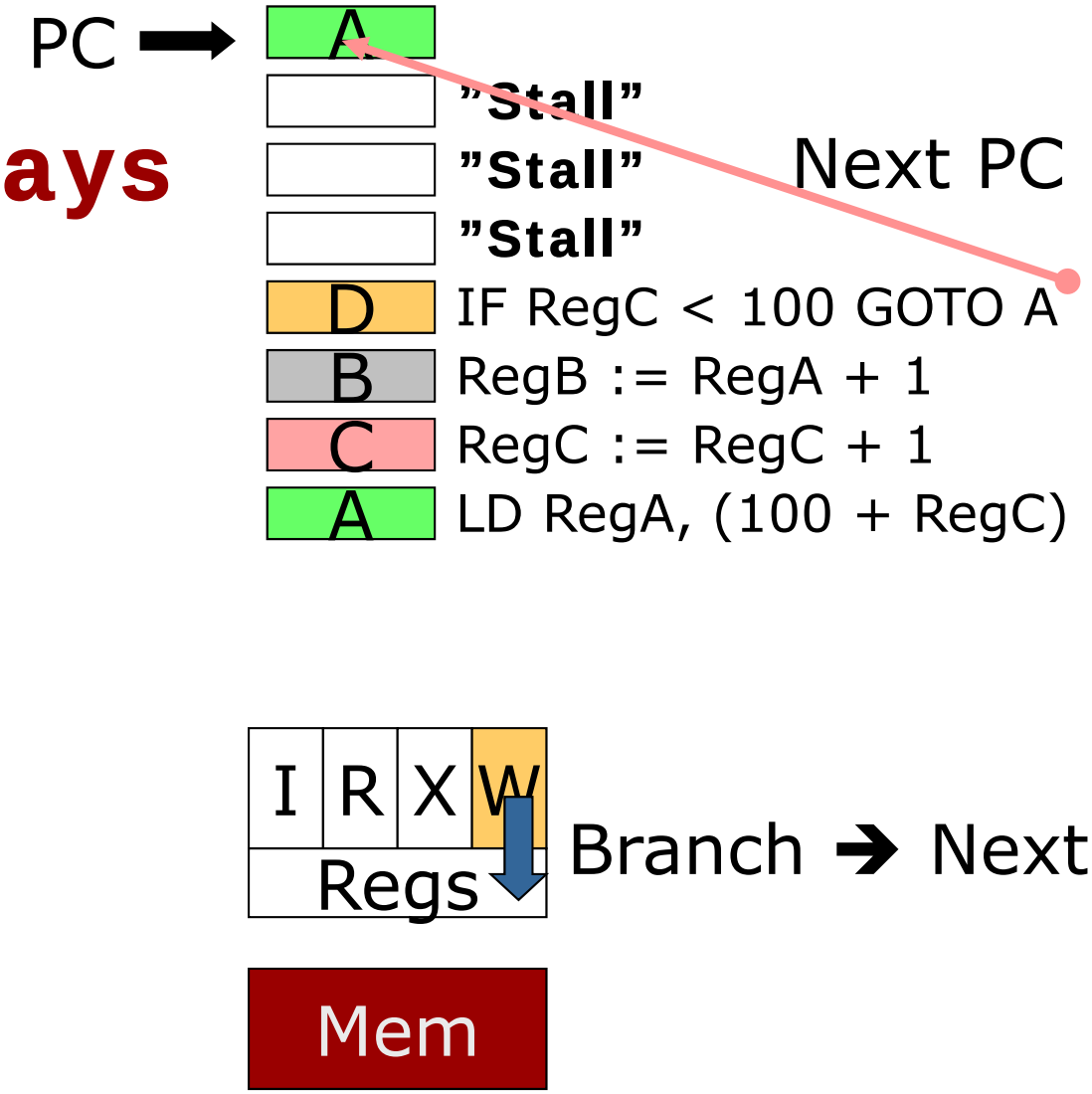


- Forwarding (or bypassing): provides a direct path from M and WB to EX
- Only helps for ALU ops. What about load operations?

Bypass (pipeline interlocking)

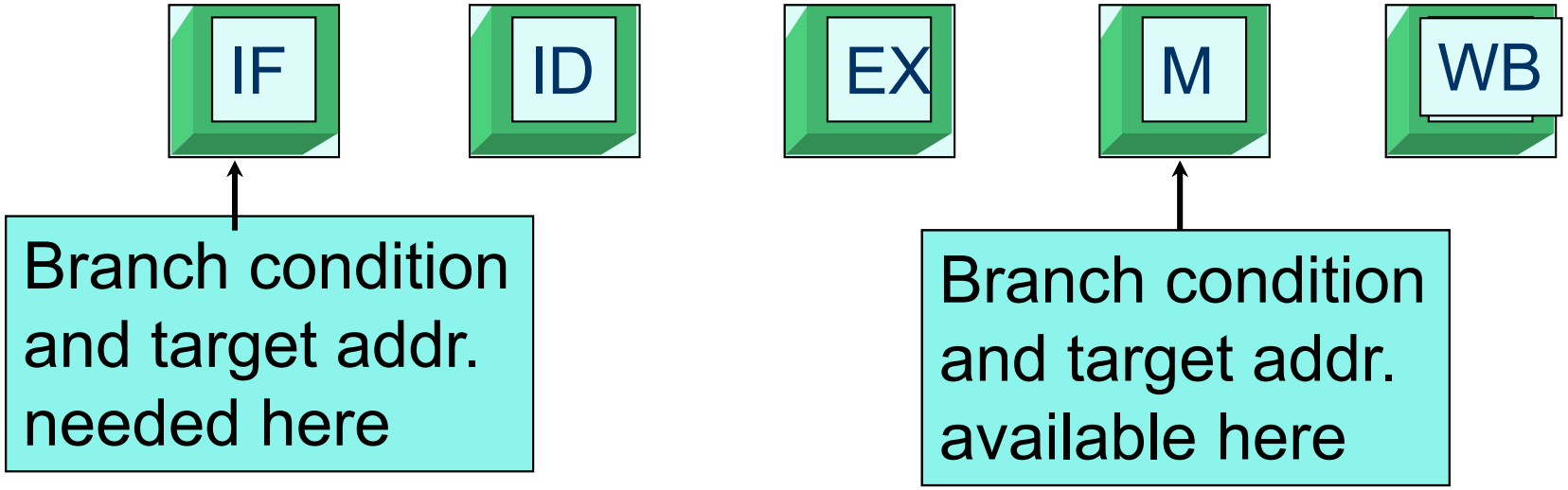


Branch delays



8 cycles per iteration of 4 instructions ☹
Need longer basic blocks with independent instr.

Avoiding control hazards

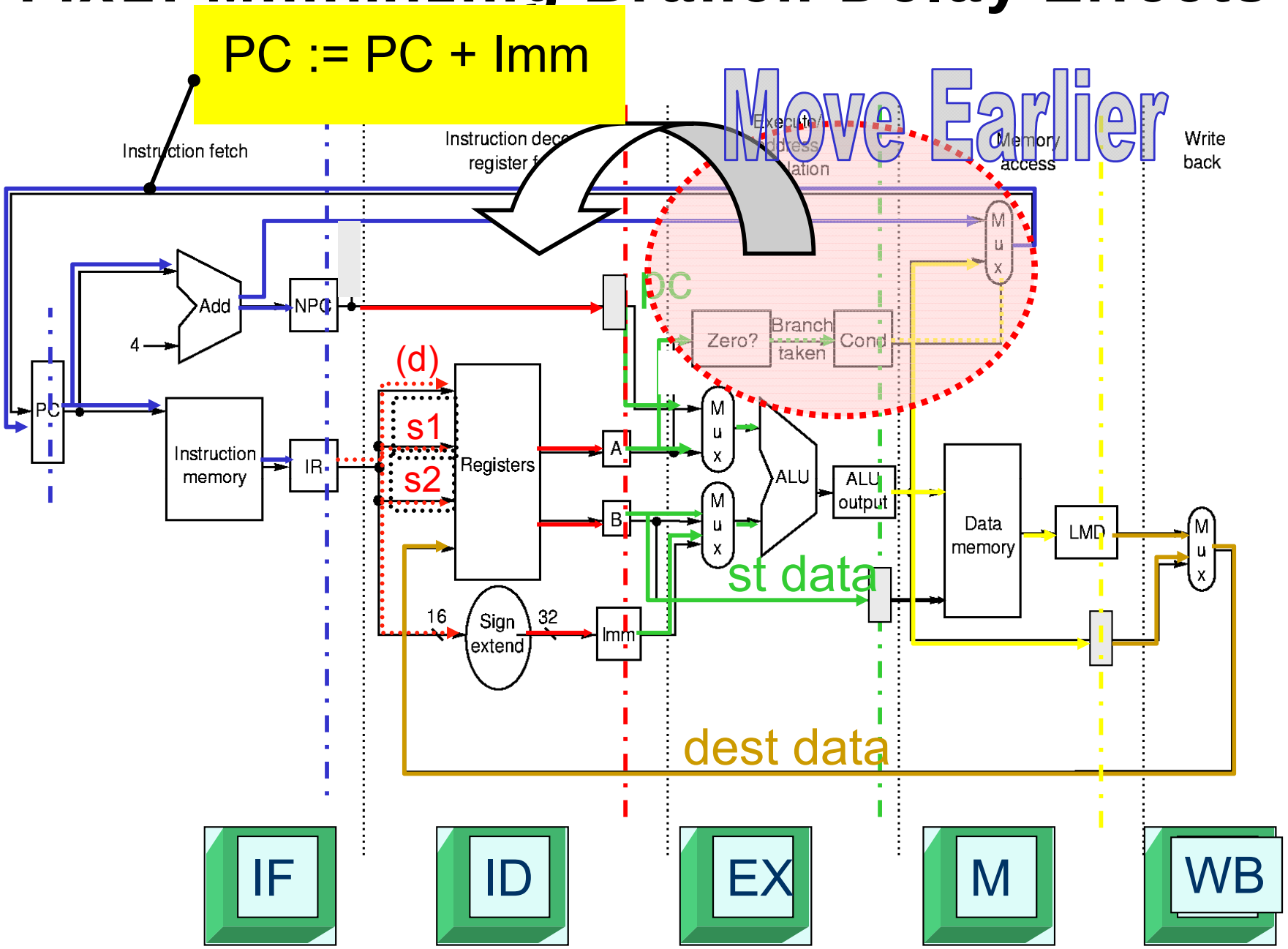


Duplicate resources in ALU to compute branch condition and branch target address earlier

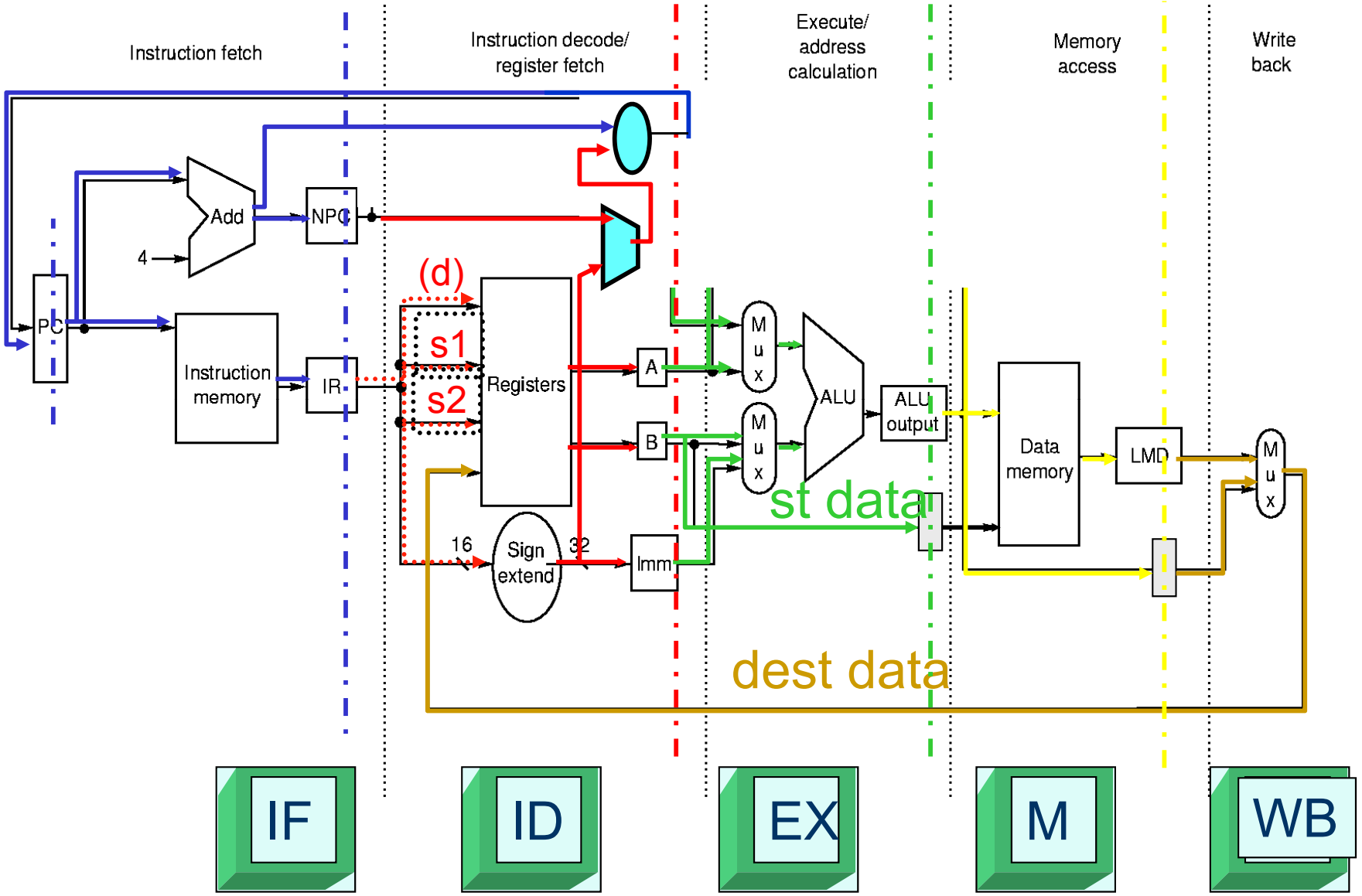
Branch delay cannot be completely eliminated

Branch prediction and code scheduling can reduce the branch penalty

Fix1: Minimizing Branch Delay Effects



Fix1: Minimizing Branch Delay Effects



UPPSALA UNIVERSITET

Fix2: Static tricks

Predict Branch not taken (a fairly rare case)

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if the branch is actually taken
- Works well if state is updated late in the pipeline
- 30%-38% of conditional branches are not taken on average

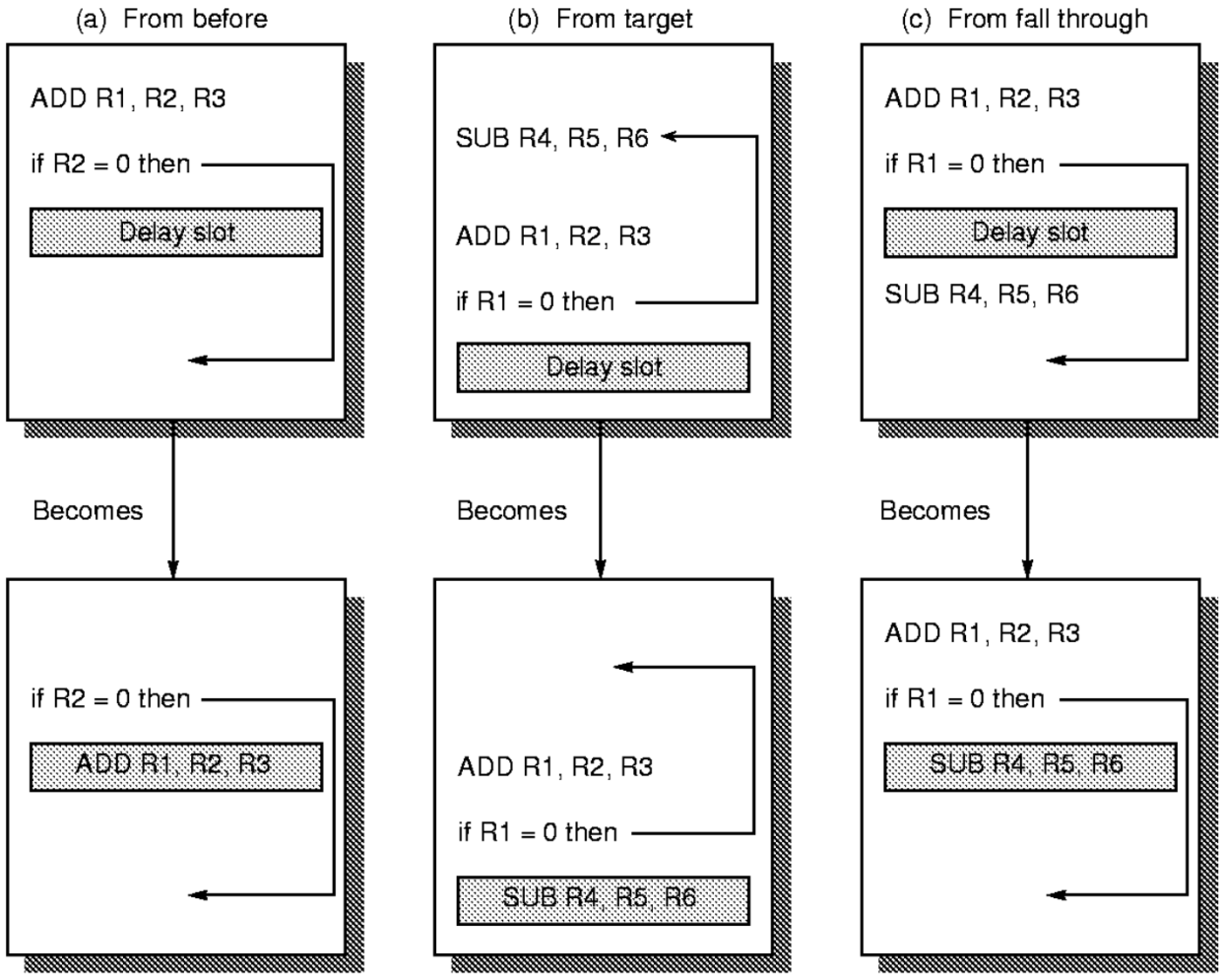
Predict Branch taken (a fairly common case)

- 62%-70% of conditional branches are taken on average
- Does not make sense for the generic arch. but may do for other pipeline organizations

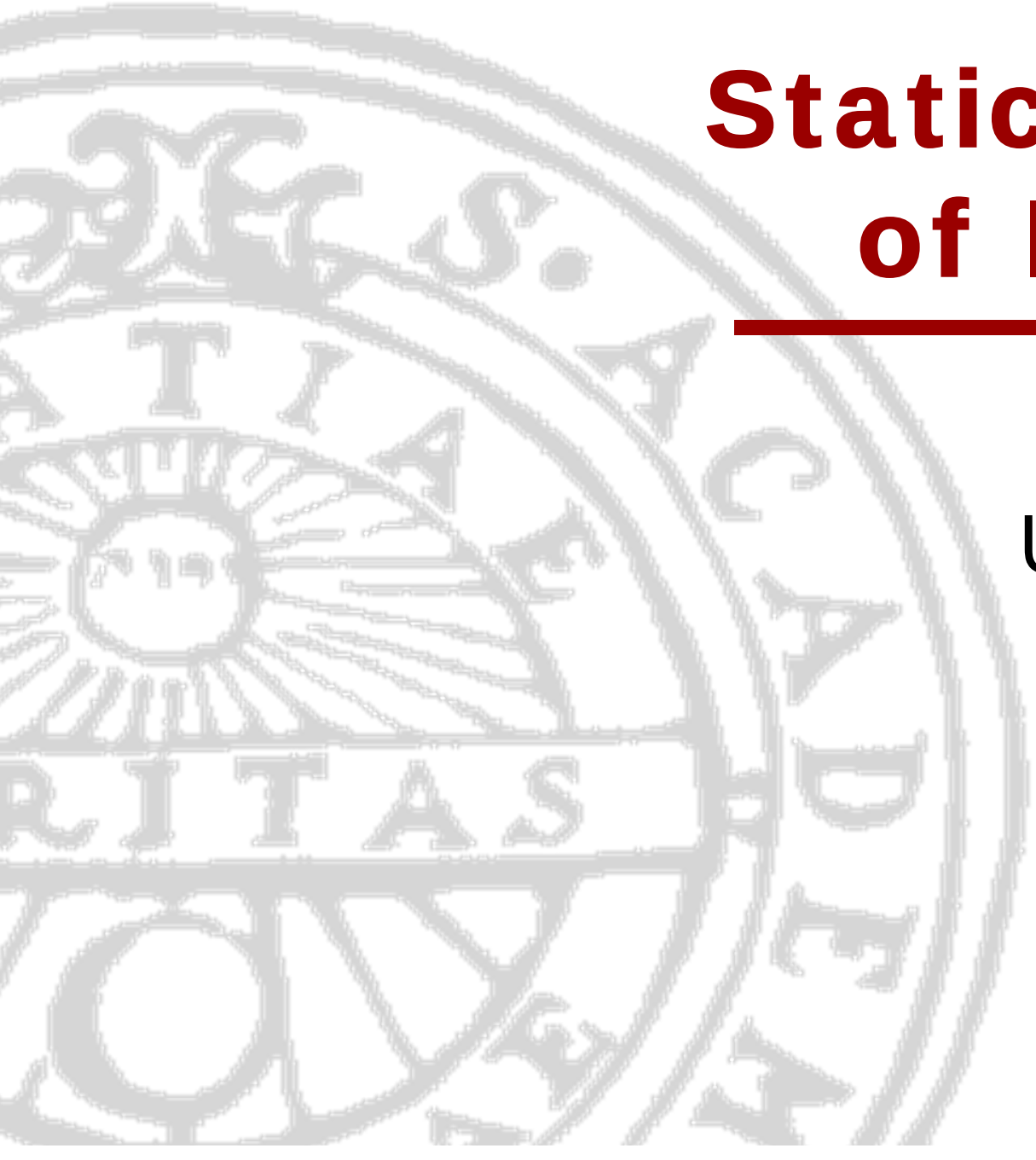
Delayed branch (schedule useful instr. in delay slot)

- Define branch to take place after a following instruction
- CONS: this is visible to SW, i.e., forces compatibility between generations

Static scheduling to avoid stalls



- **Scheduling an instruction from before is always safe**
- **Scheduling from target or from the not-taken path is not always safe; must be guaranteed that speculative instr. do no harm.**



Static Scheduling of Instructions

Erik Hagersten
Uppsala University
Sweden

UPPSALA UNIVERSITET

Architectural assumptions

INT ALU produces the results the next cycle
FPU ALU produces the results 3 cycles later

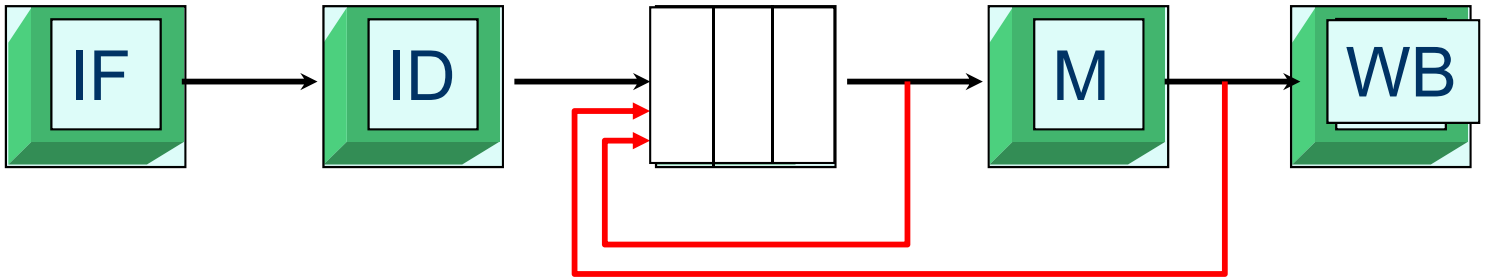
From	To	Latency
FP ALU	FP ALU	3
FP ALU	SD	2
LD	FP ALU	1

Latency=number of cycles between the two adjacent instructions

Delayed branch: one branch delay slot

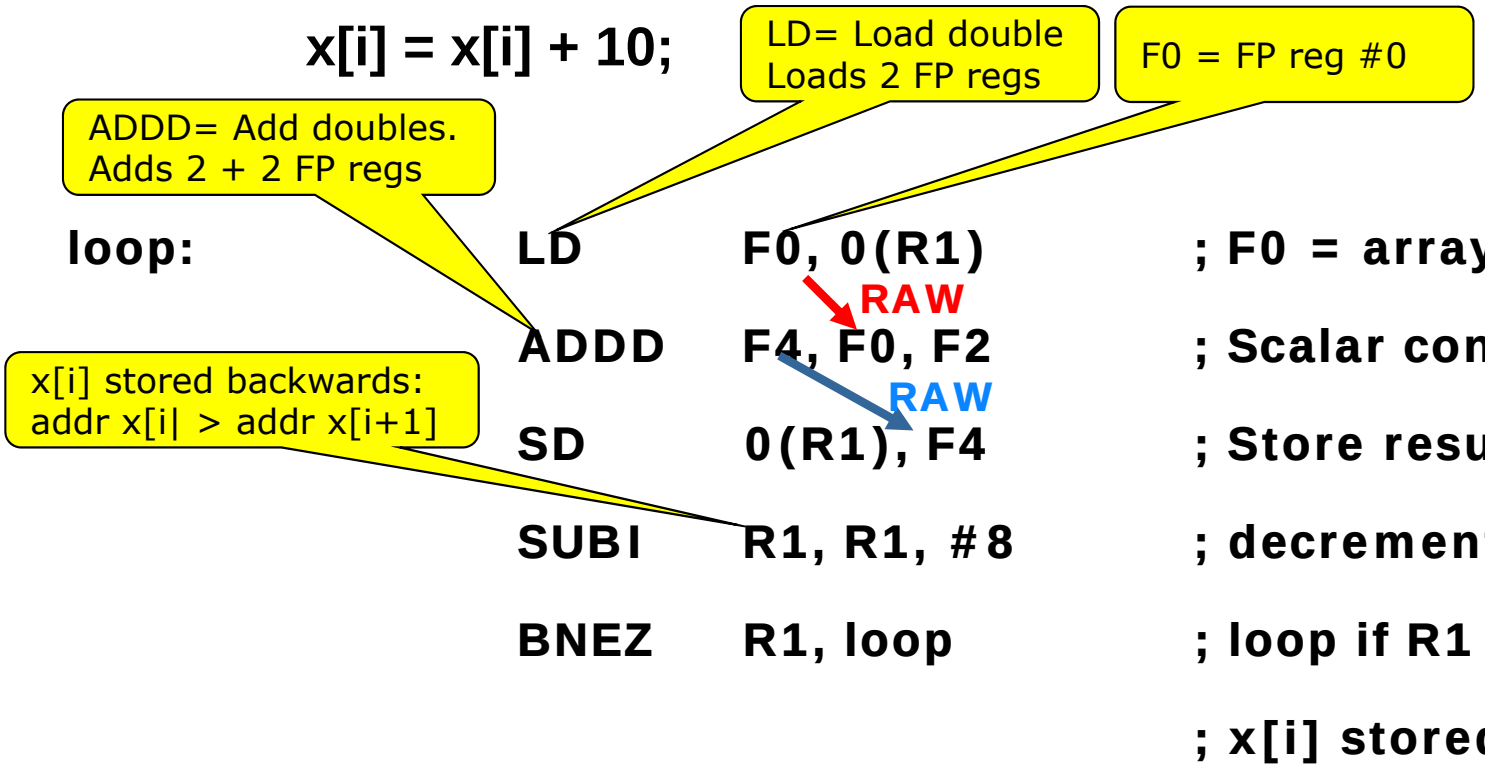
Why does not forwarding / bypass trick work for floating points?

FP takes 3 cycles to produce results ☹



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]
        ADDD    F4, F0, F2      ; Scalar constant “10” in F2
        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

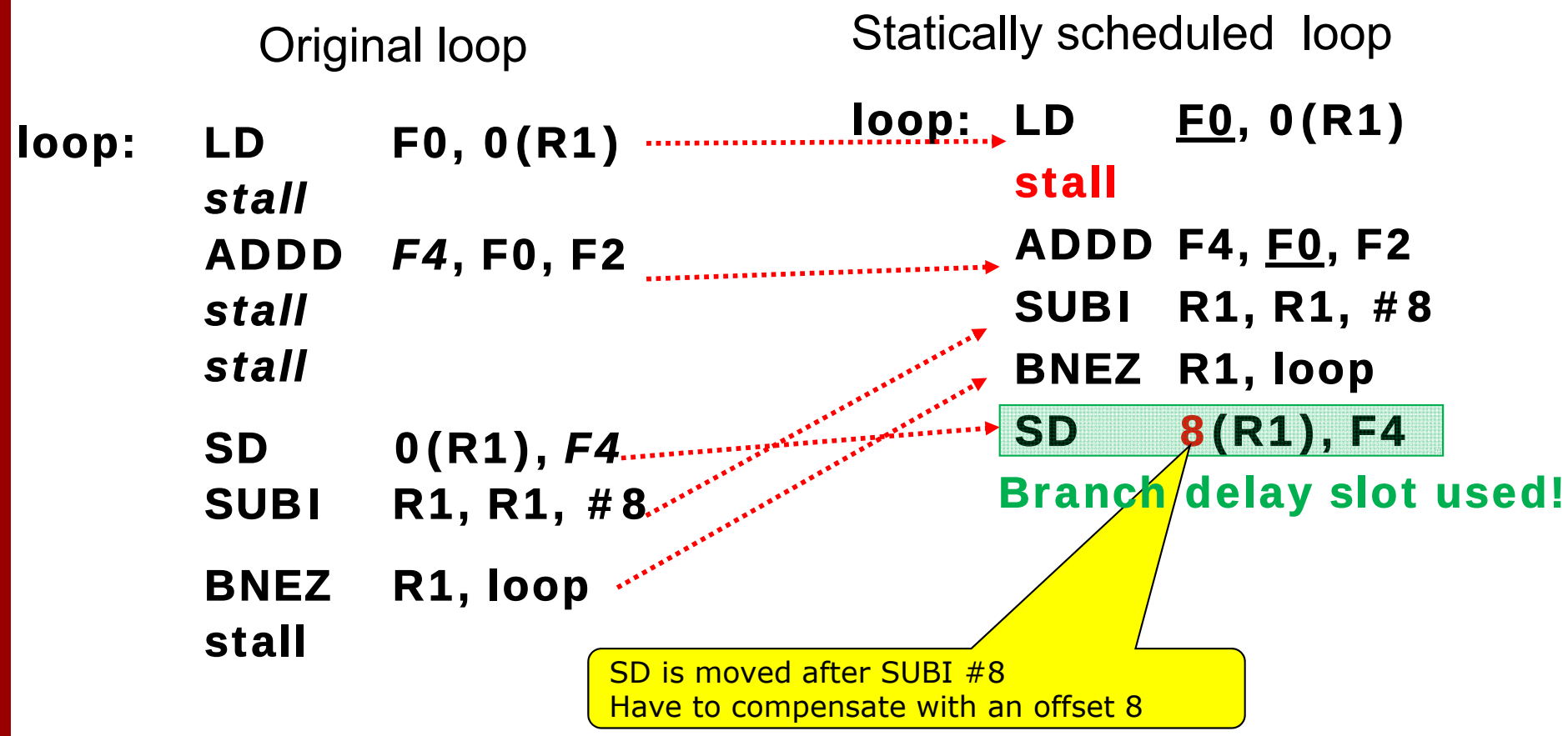
RAW

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.

AVDARK 2013

Scheduling in each loop iteration



5 instruction + 4 bubbles = 9c / iteration

5 instruction + 1 bubble = 6c / iteration

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

loop:   1.LD    F0, 0(R1)       ; F0 = array element x[i]
        2.LD    F4, 0(R2)       ; F4 = array element y[i]
        3.ADDD  F2, F4, F0      ; RAW
        4.SD    0(R1), F2       ; RAW
        5.SUBI  R1, R1, #8      ; Store result
        6.SUBI  R2, R2, #8      ; decrement array ptr x
        7.BNEZ  R1, loop        ; decrement array ptr y
                                ; loop if R1 != 0
                                ; x[i] stored "backwards"
```


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

loop:   1.LD    F0, 0(R1)       ; F0 = array element y[i]
        2.LD    F4, 0(R2)       ; F4 = array element x[i]
        stall                   RAW
        3.ADDD  F2, F4, F0
        stall                   RAW
        stall
        4.SD    0(R1), F2       ; Store result
        5.SUBI  R1, R1, #8      ; decrement array ptr x
        6.SUBI  R2, R2, #8      ; decrement array ptr y
        7.BNEZ  R1, loop        ; loop if R1 != 0
        stall
```

UPPSALA UNIVERSITET

Improving Instruction-Level Parallelism (ILP)

- Not enough independent instructions
- Compiler optimizations can improve the ILP!
- The pipeline can work on more independent instructions at the same time
- At the same time = in parallel

Can we do even better by scheduling across iterations?

Loop unrolling 4x

UPPSALA UNIVERSITET

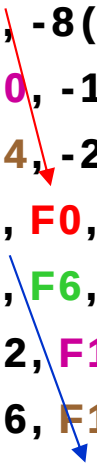
```
<prologue>
loop:   LD      F0, 0(R1)
        stall
        ADDD    F4, F0, F2
        stall                       ; drop SUBI & BNEZ
        stall
        SD      0(R1), F4
        LD      F6, -8(R1)
        stall
        ADDD    F8, F6, F2
        stall                       ; drop SUBI & BNEZ
        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

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

UPPSALA UNIVERSITET

Scheduling example

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

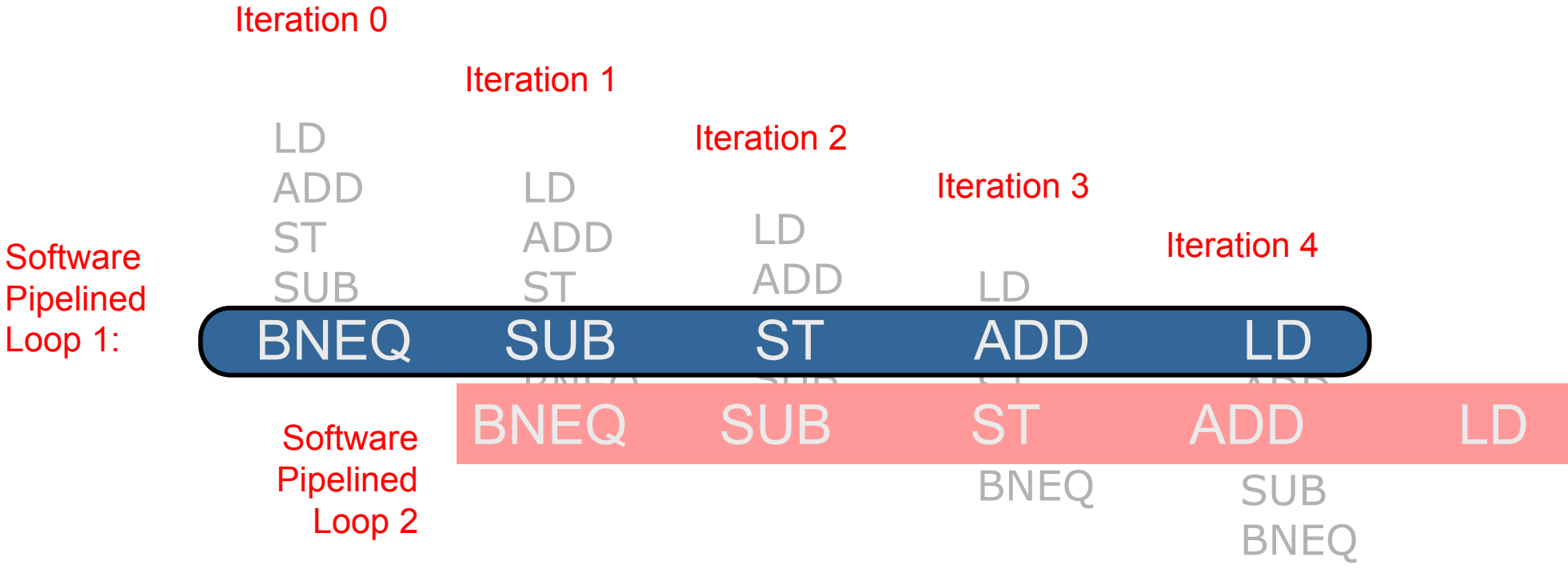
```
loop:   LD     F0, 0(R1)      ; F0 = array element x[i]
                  RAW
        ADDD   F4, F0, F2     ; Scalar constant "10" in F2
                       RAW
        SD     0(R1), F4      ; Save result
        SUBI   R1, R1, #8     ; decrement array ptr.
        BNEZ   R1, loop       ; loop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

Here: Each iteration in the source code corresponds to one loop in the assembler. Is that a requirement?

Software pipelining 1(3)

Symbolic loop unrolling

- Stagger the start of each iteration in the instruction scheduling

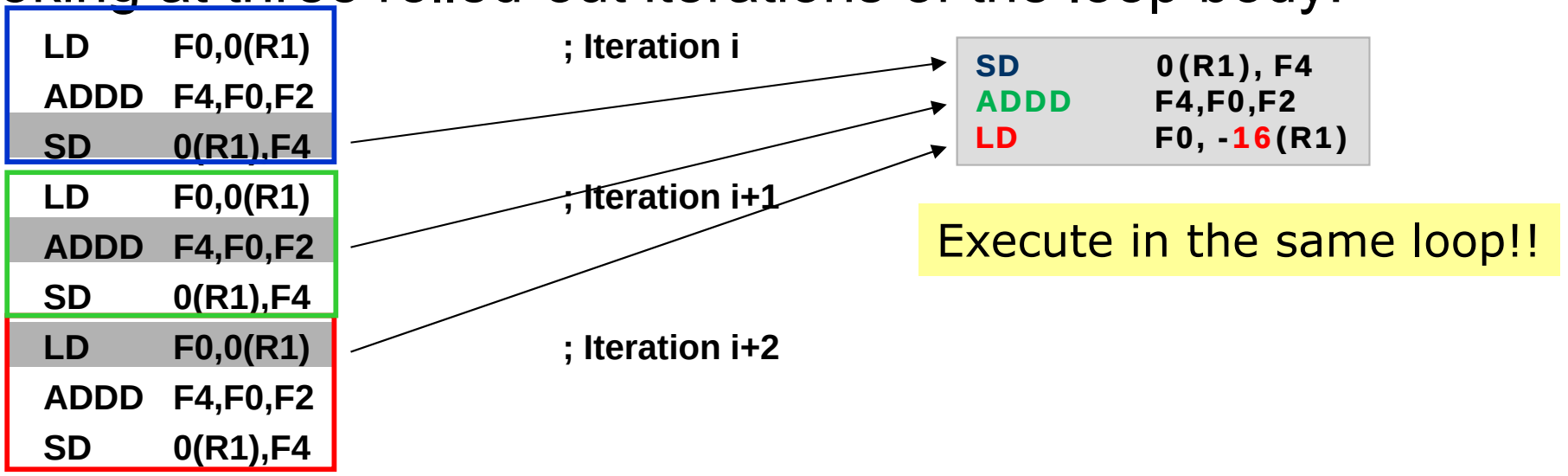


Software pipelining 2(3)

Example:

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

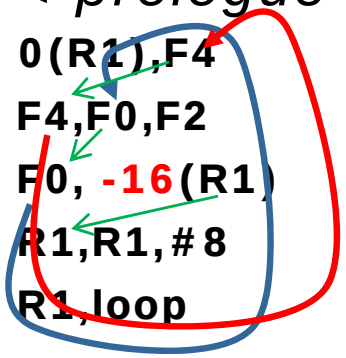
Looking at three rolled-out iterations of the loop body:



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)

UPPSALA UNIVERSITET

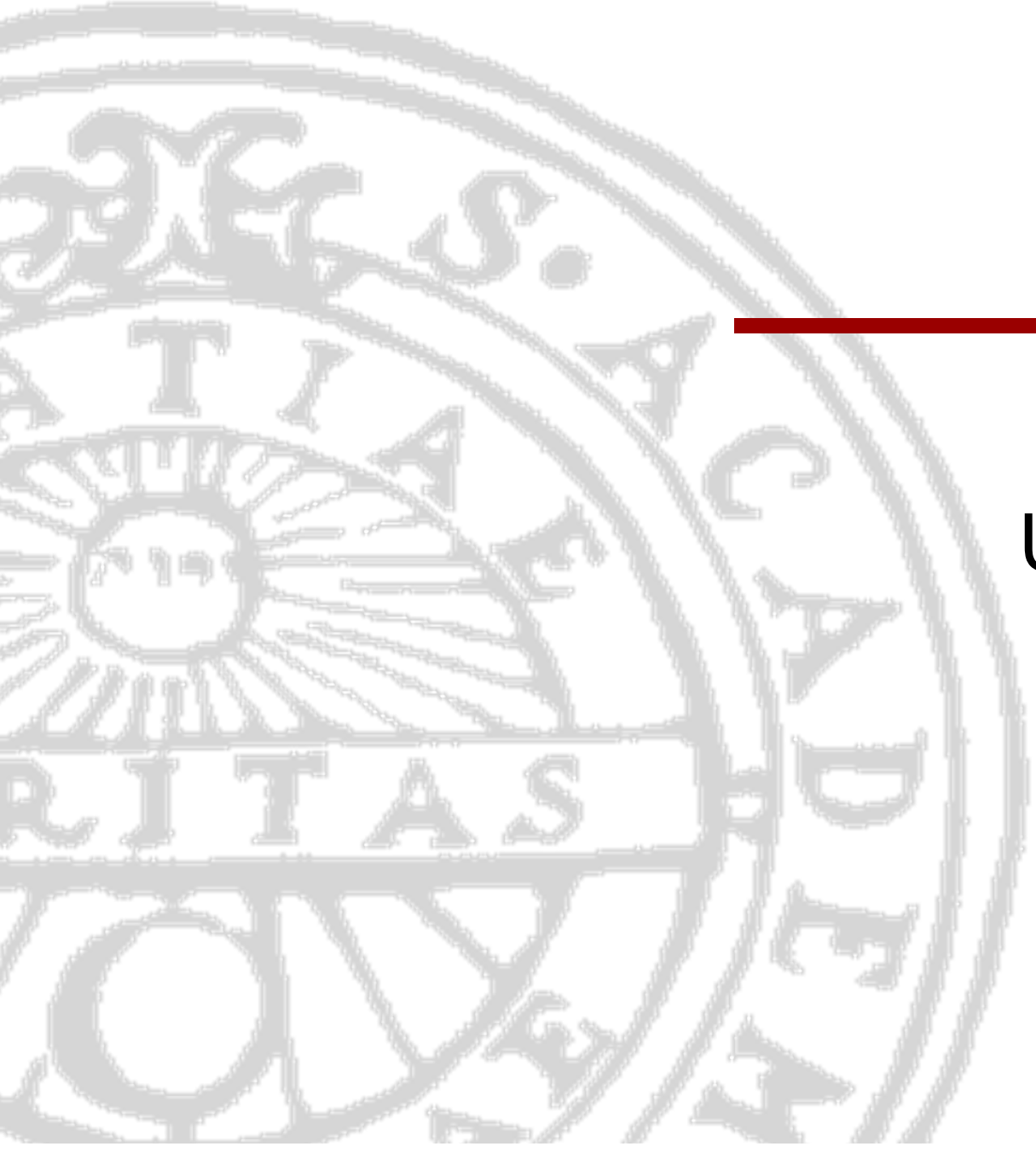
Software pipelining

- ”Symbolic Loop Unrolling”
- Very tricky for complicated loops
- Less code expansion than unrolling
- Register-poor if ”rotating” is used

UPPSALA UNIVERSITET

Improving Instruction-Level Parallelism (ILP)

- Not enough independent instructions
- Compiler optimizations can improve the ILP!
- The pipeline can work on more independent instructions at the same time
- At the same time = in parallel



Superscalars

Erik Hagersten
Uppsala University
Sweden

UPPSALA UNIVERSITET

Multiple instruction issue per clock

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

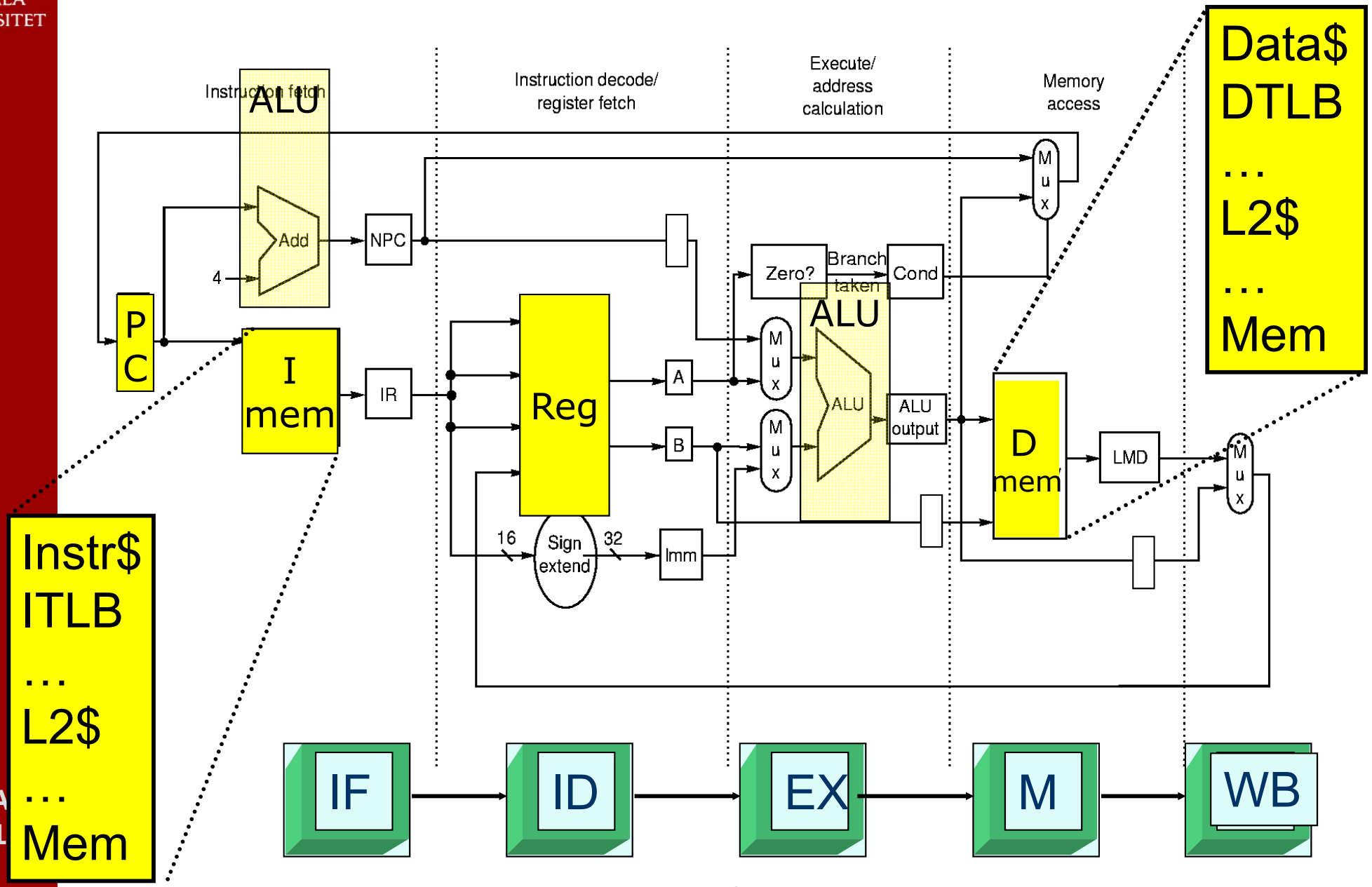
NOW: Superscalar

- 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):

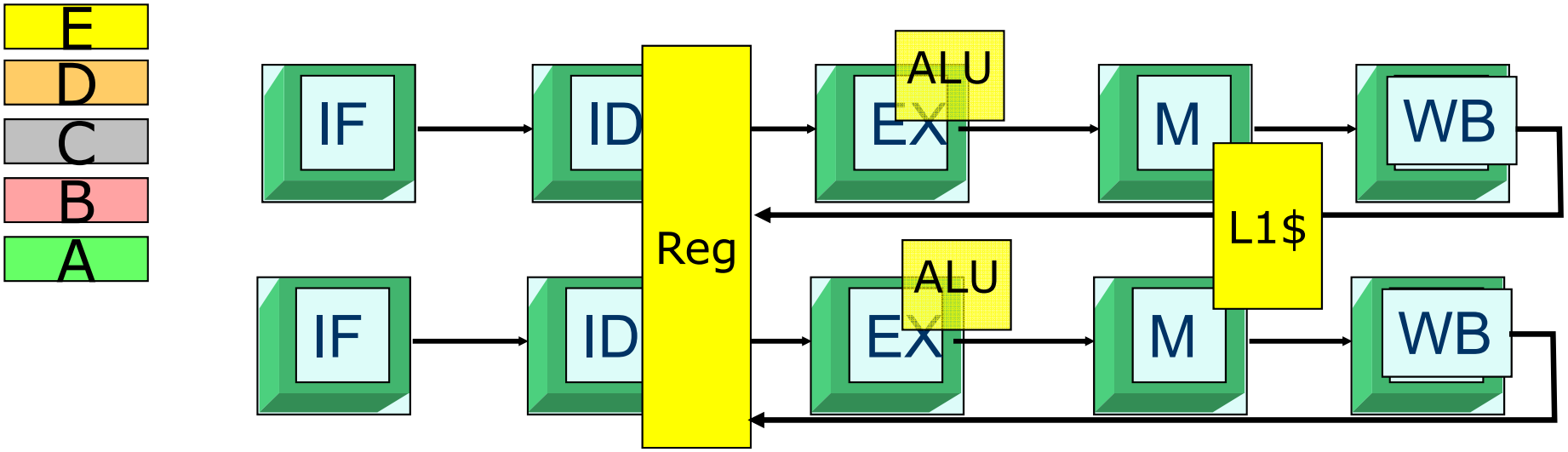
- Static scheduling used to form packages of independent instructions that can be issued together
- Relies on compiler to find independent instructions (IA-64 “Itanium”, Transmeta)

Example: 5-stage pipeline

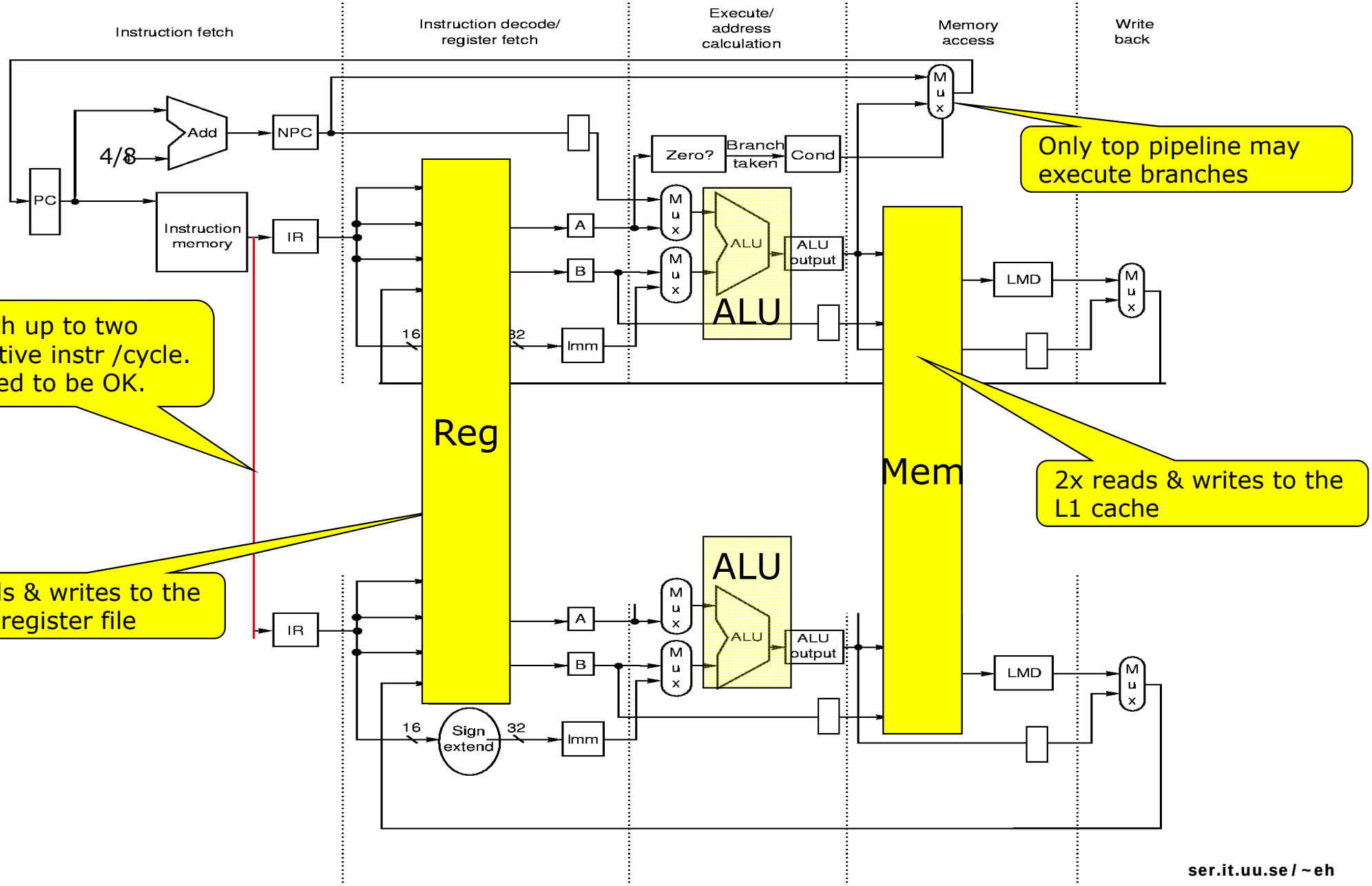


A 5-stage superscalar pipeline

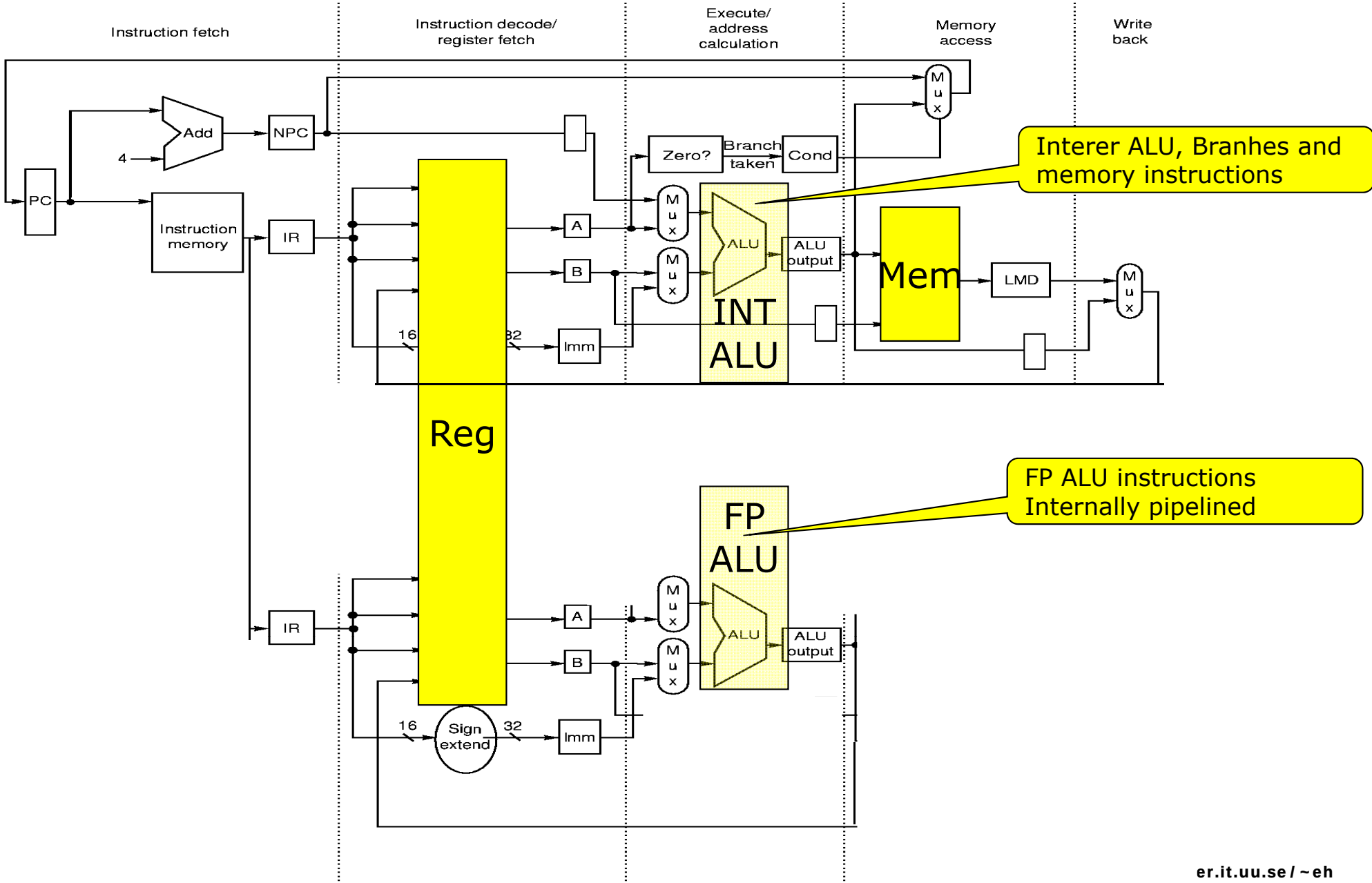
One sequential program:



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



The 5-stage pipeline Superscalar DLX



UPPSALA UNIVERSITET

Example: A Superscalar DLX

- **Issue 2 instructions simultaneously: 1 FP & 1 integer**
  - **Fetch 64-bits/clock cycle; Integer instr. on left, FP on right**
  - **Can only issue 2nd instruction if 1st instruction issues**
  - **Need more ports to the register file**

Type	*Pipe stages*						
INSTR	***CYCLE: 1***	***2***	***3***	***4***	***5***	***6***	***7***
1. Int.	IF	ID	EX	MEM	WB		
2. FP	**IF**	**ID**	**EX**	**MEM**	**WB**		
3. Int.		IF	ID	EX	MEM	WB	
4. FP		**IF**	**ID**	**EX**	**MEM**	**WB**	
5. Int.			IF	ID	EX	MEM	WB
6. FP			**IF**	**ID**	**EX**	**MEM**	**WB**

- EX stage should be fully pipelined
- 1 load delay slot corresponds to three instructions!

AVDARK 2013

UPPSALA UNIVERSITET

Statically Scheduled Superscalar DLX

	Integer instruction		FP instruction	Clock cycle
Loop:	LD	F0,0(R1)		1
	LD	F6,-8(R1)		2
	LD	F10,-16(R1)	ADDD F4,F0,F2	3
	LD	F14,-24(R1)	ADDD F8,F6,F2	4
	LD	F18,-32(R1)	ADDD F12,F10,F2	5
	SD	0(R1),F4	ADDD F16,F14,F2	6
	SD	-8(R1),F8	ADDD F20,F18,F2	7
	SD	-16(R1),F12		8
	SD	-24(R1),F16		9
	SUBI	R1,R1,#40		10
	BNEZ	R1,LOOP		11
	SD	-32(R1),F20		12

Issue: Difficult to find a sufficient number of instr. to issue

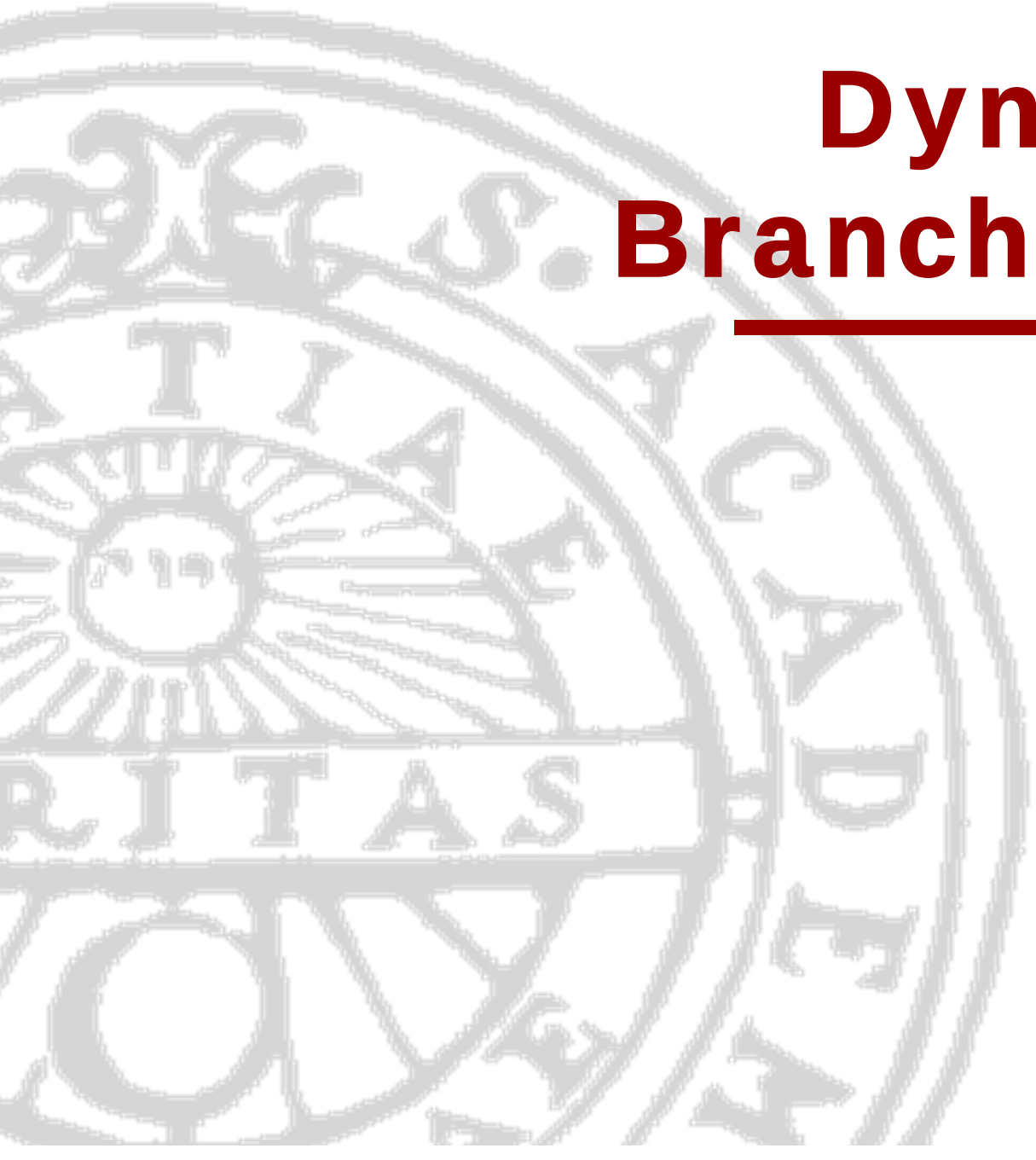
(5 loops in 12 cycles)

UPPSALA UNIVERSITET

Limits to superscalar execution: Instruction-Level Parallelism (ILP)

- Difficulties in scheduling within the constraints on number of functional units and the ILP in the code chunk
- Instruction decode complexity increases with the number of issued instructions
- Data and control dependencies are in general more costly in a superscalar processor than in a single-issue processor

Techniques to enlarge the instruction window to extract more ILP are important (Coming soon: Out-of-order pipelines)



Dynamic tricks: Branch Predictions

Erik Hagersten
Uppsala University

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

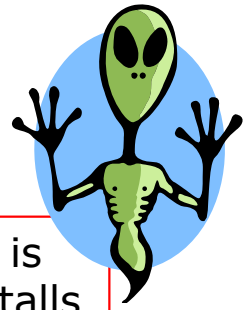
Data hazards: Data dependencies between instructions

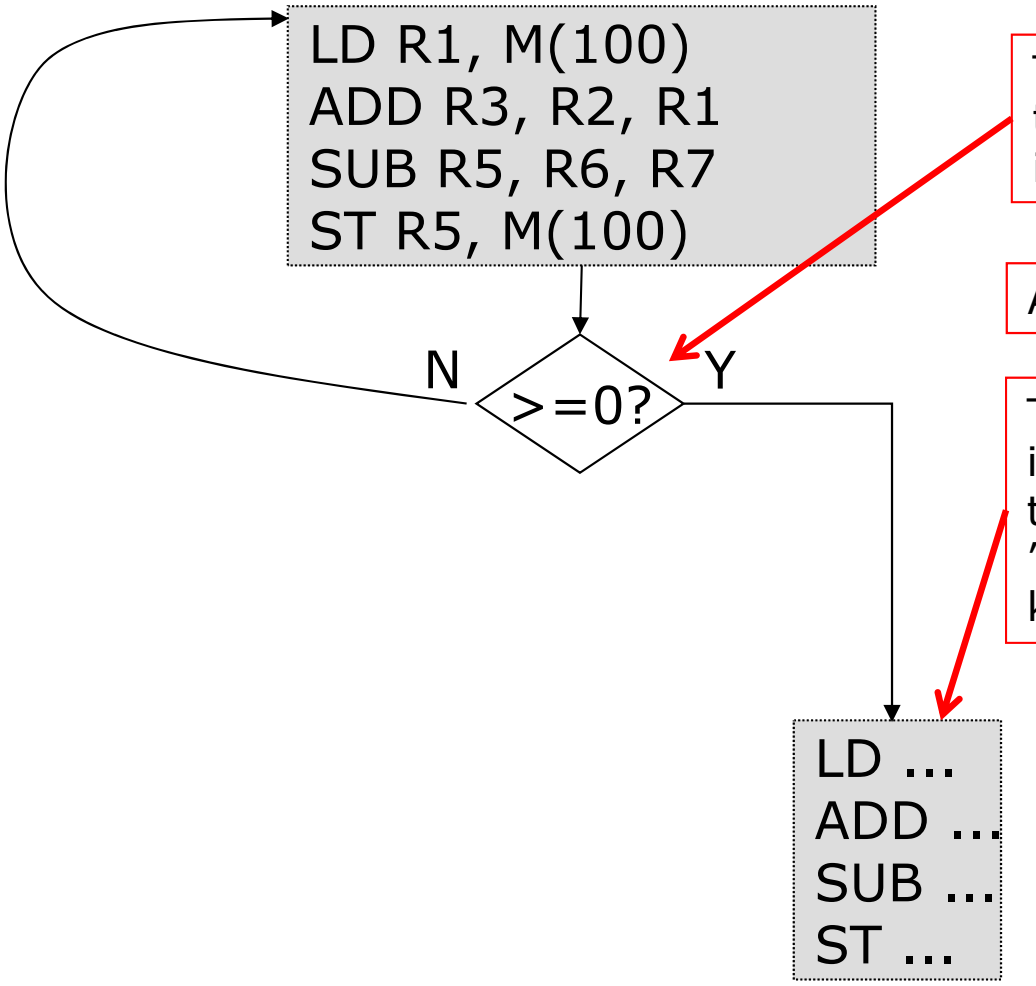
```
LW R1, 100(R2)   /* result avail in 2-150 cycles */
ADD R5, R1, R7
```

Control hazards: Change in program flow

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

From the Crashcourse



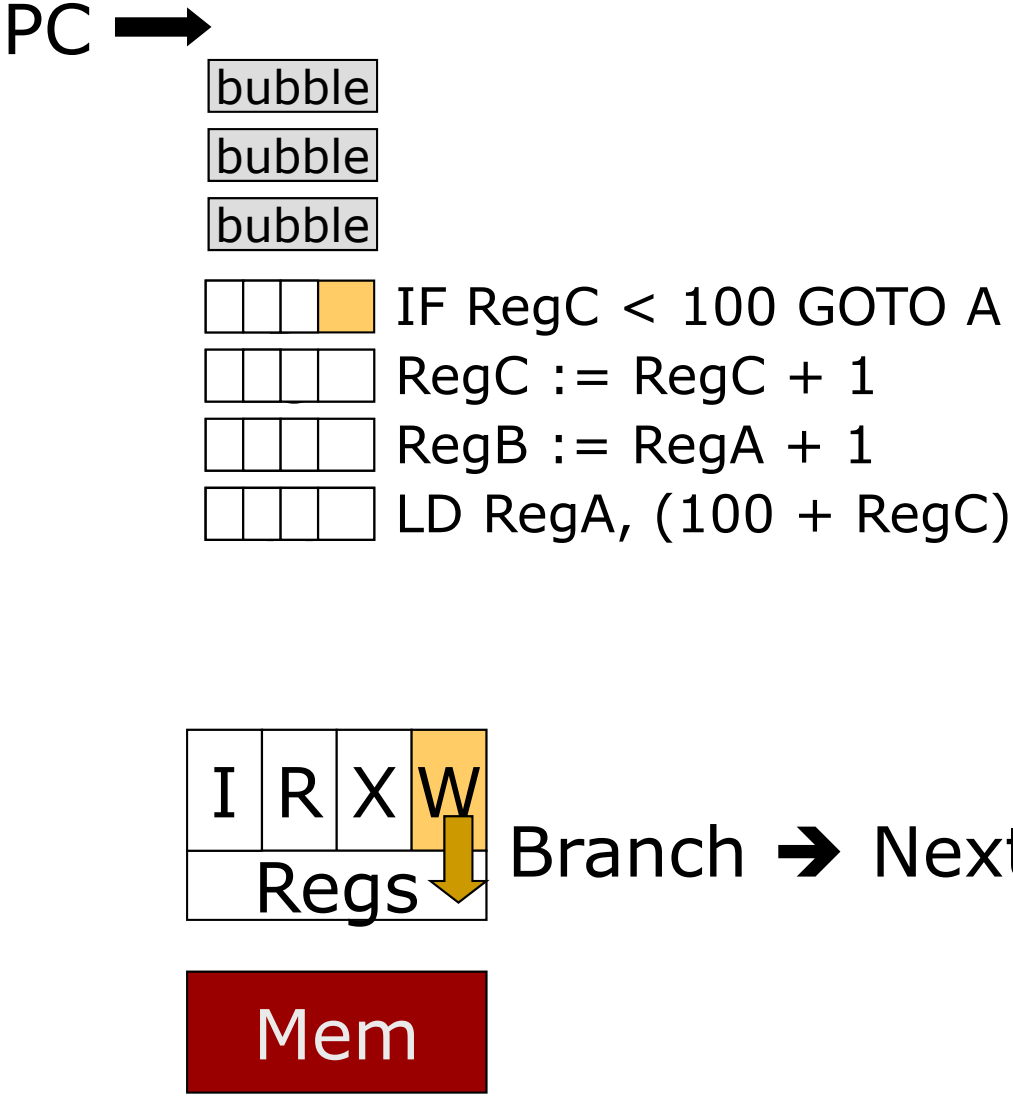


The HW can guess if the branch is taken or not and avoid branch stalls if the guess is correct.

Assume the guess is ”Y”.

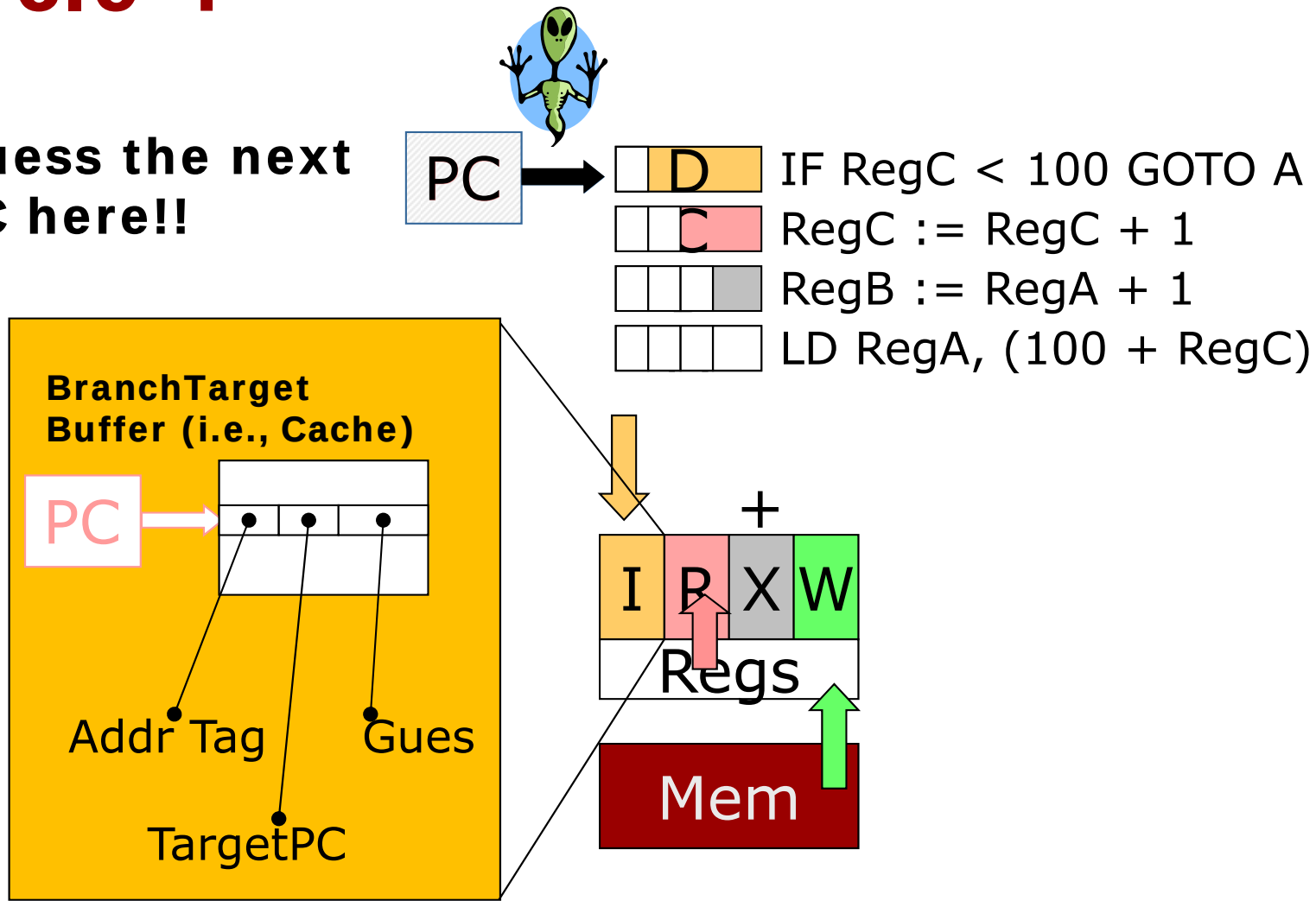
The HW can start to execute these instruction **before** the outcome the the branch is known, but cannot allow any ”side-effect” to take place until the outcome is known

Bubble-trouble:

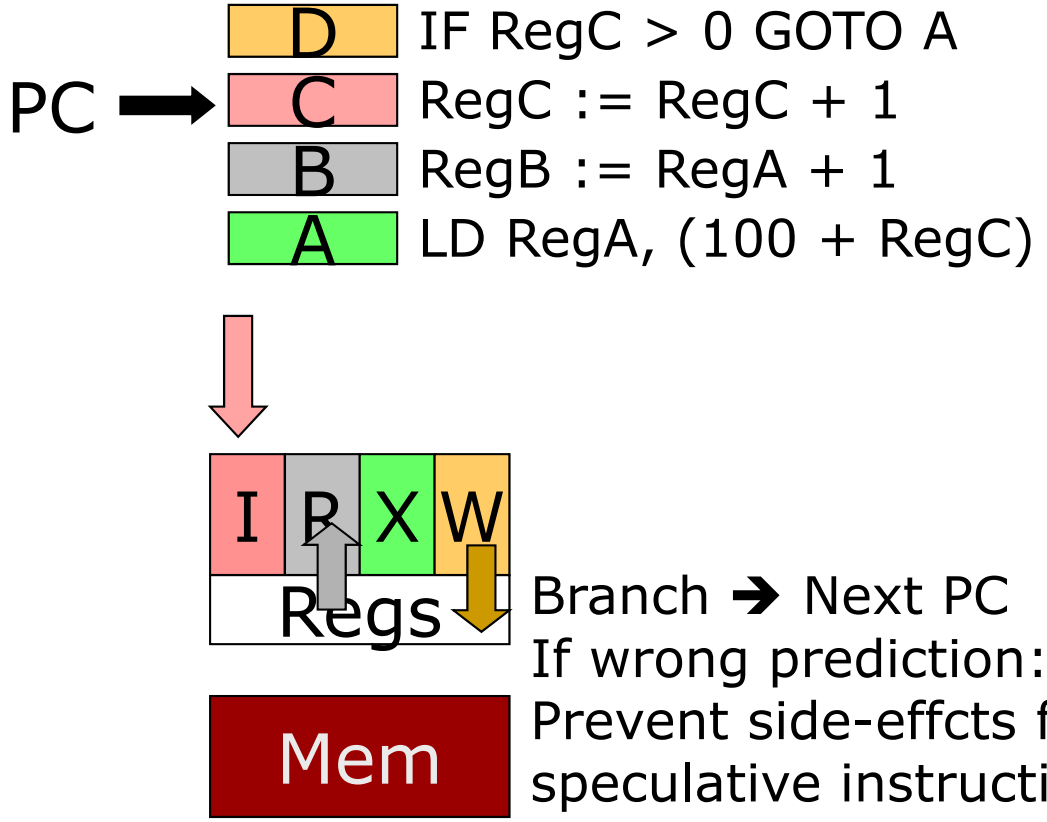


Cycle 4

Guess the next PC here!!

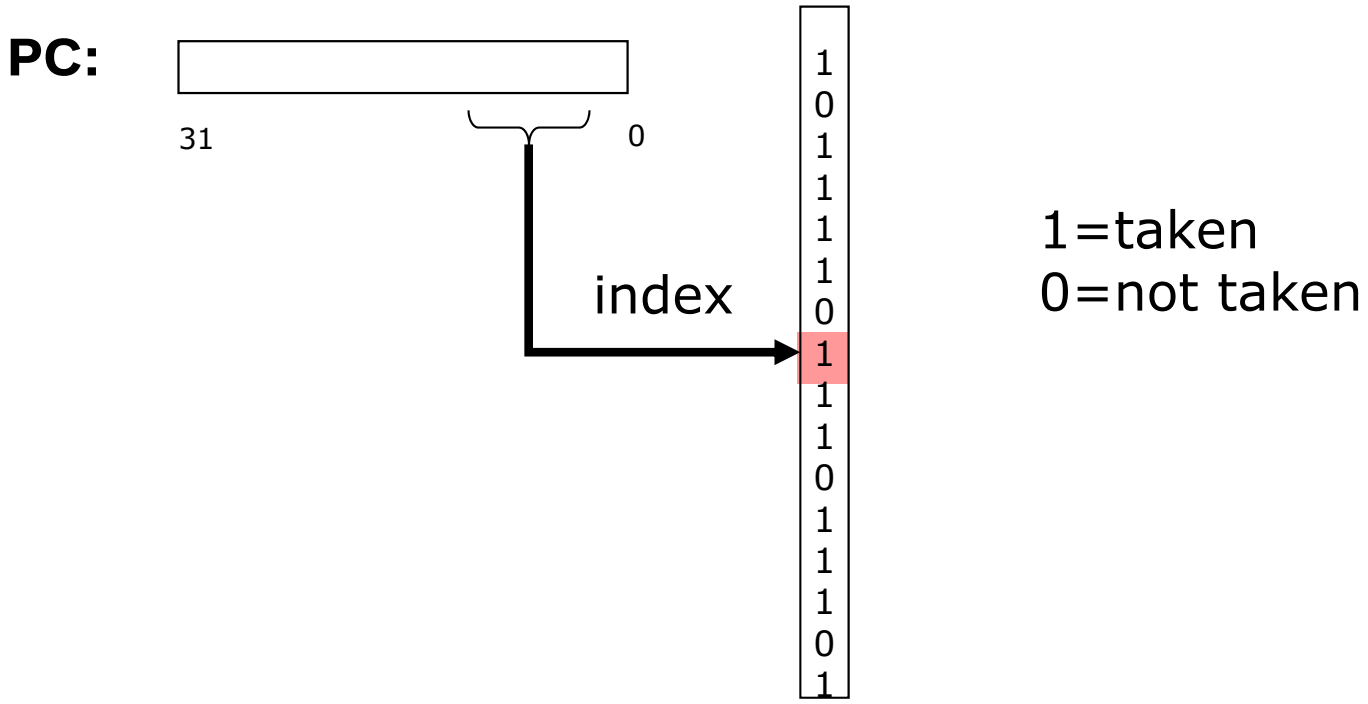


Squashing speculative work



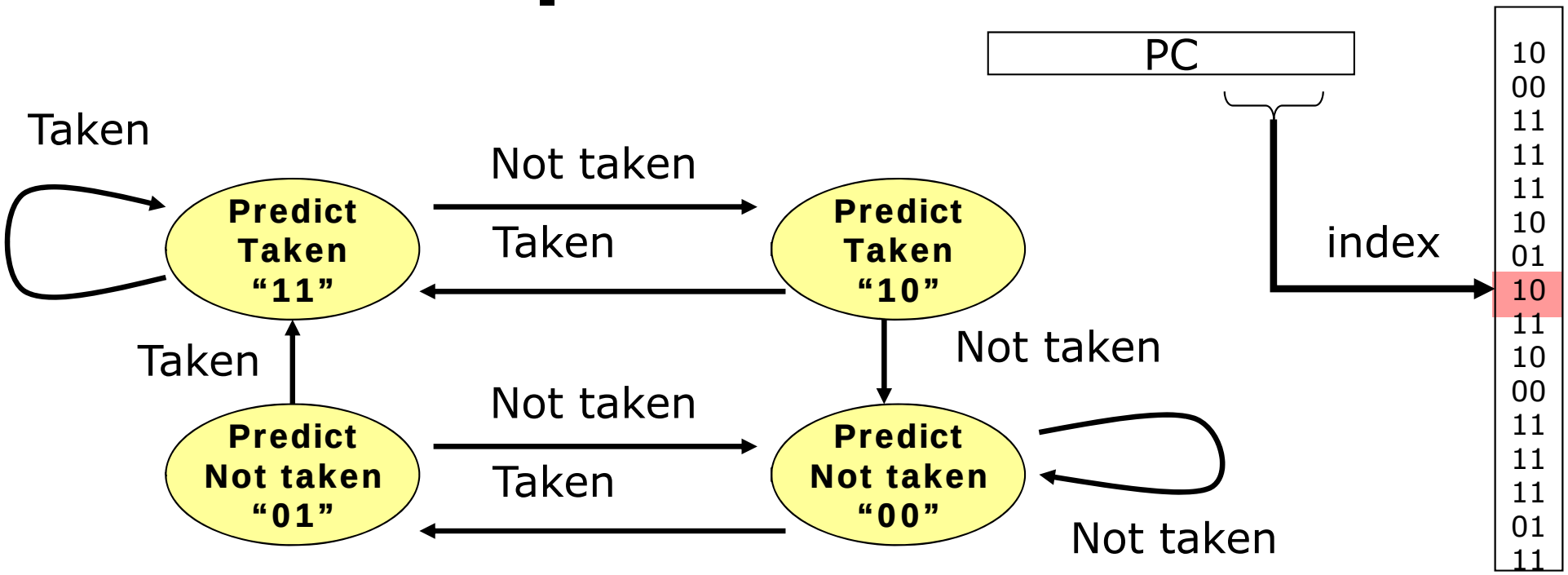
Branch history table

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

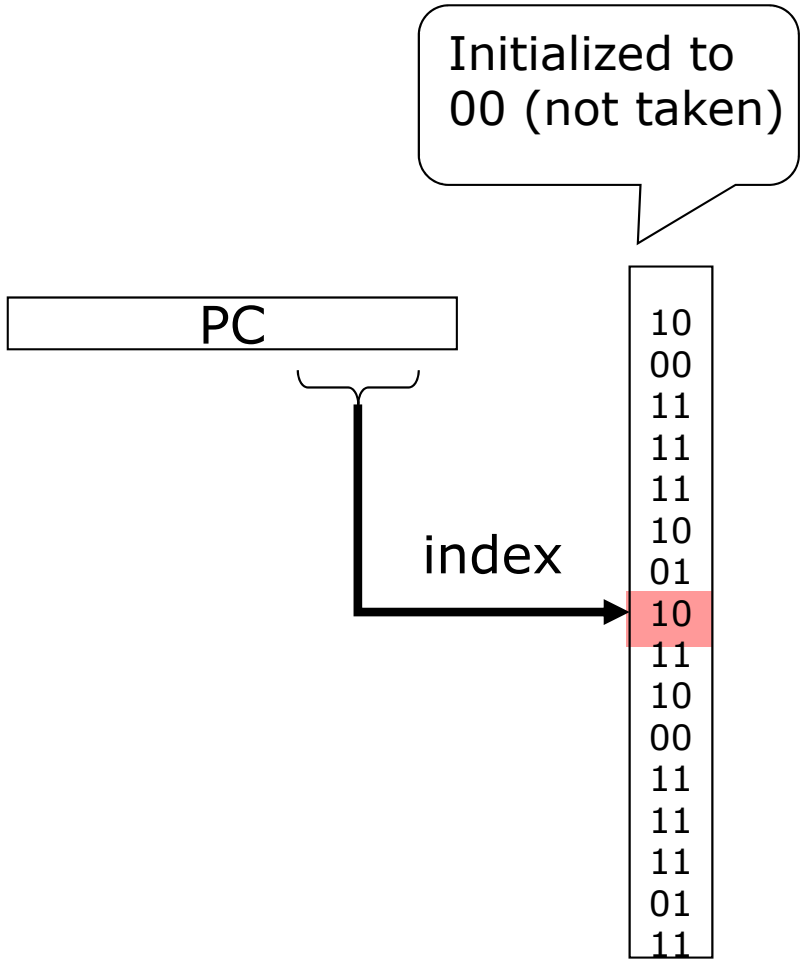
A two-bit prediction scheme



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

UPPSALA UNIVERSITET

```
This code write ones to a
matrix (N is large)
for (i = 0; i < N)
    for (j = 0; j < N) {
        x[i,j] = 1
}
```

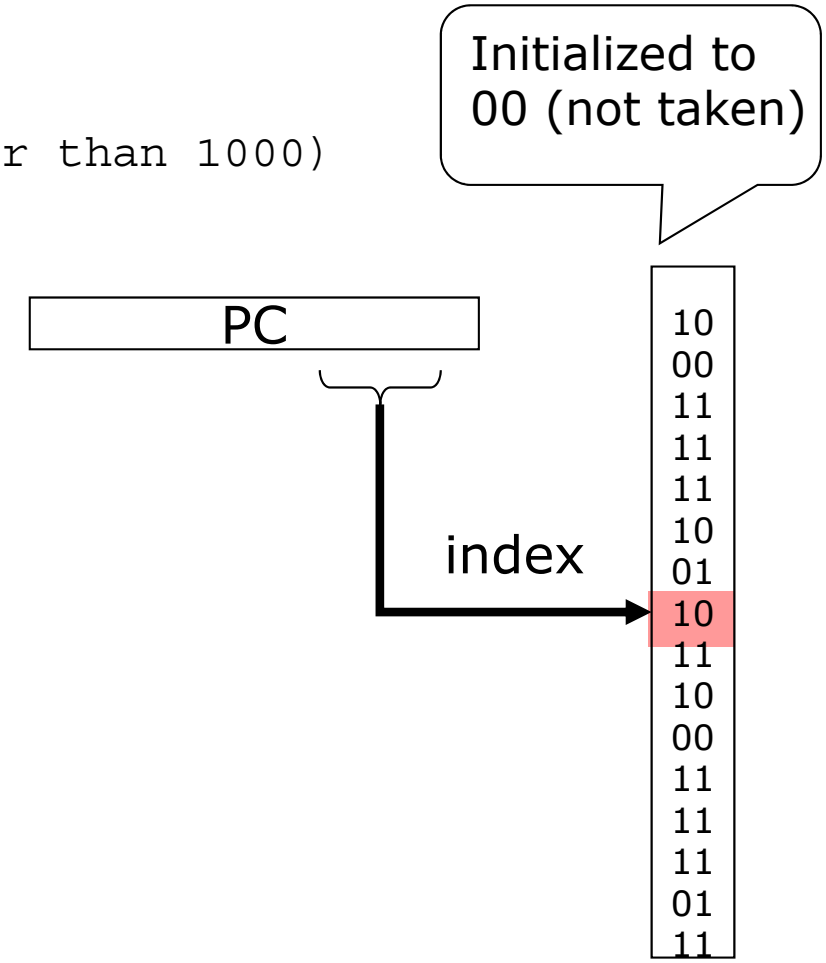


AVDARK 2013

This code write zeroes to the rightmost diagonal of a matrix (N is larger than 1000)

```
for (i = 0; i < N)
   for (j = 0; j < N) {
       if (j > i)
          x[i,j] = 1
       else
          x[i,j] = 0
   }
```

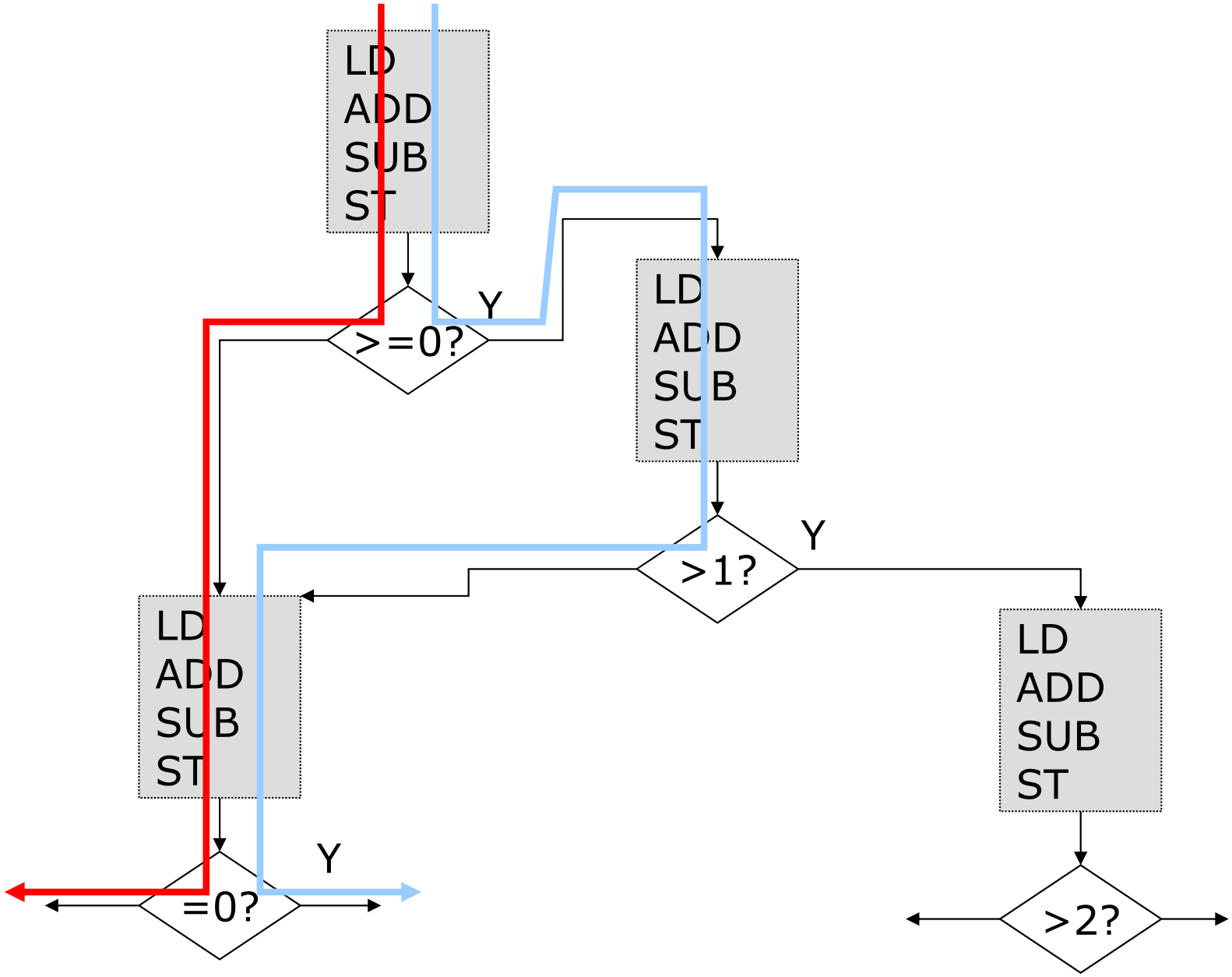
0	1	1	1	1	1	1
0	0	1	1	1	1	1
0	0	0	1	1	1	1
0	0	0	0	1	1	1
0	0	0	0	0	1	1
0	0	0	0	0	0	1
0	0	0	0	0	0	0



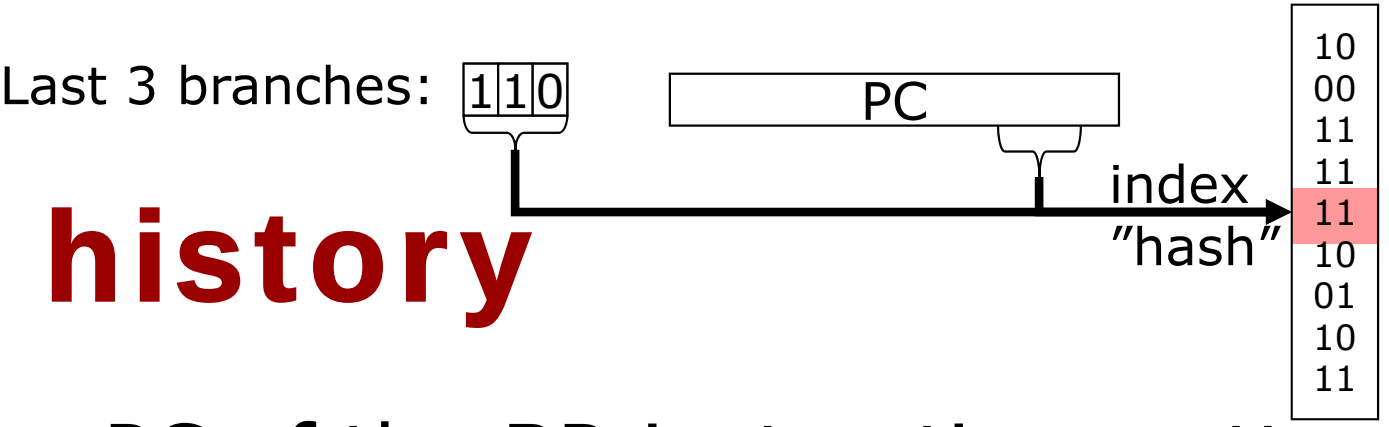
Improving Branch Predictions

Erik Hagersten
Uppsala University

Adding Global History



N-level history



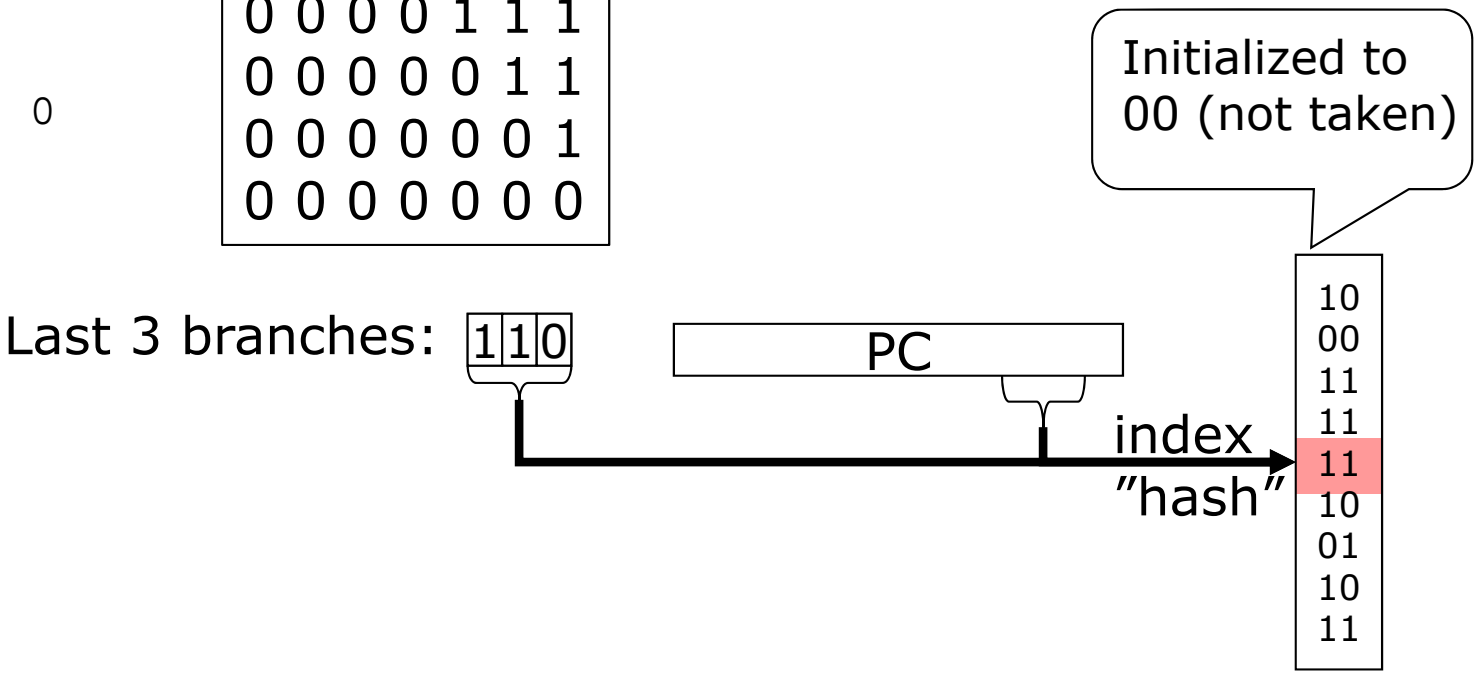
- Not only the PC of the BR instruction matters, also how you've got there is important
- Approach:
  - Record the outcome of the last N branches in a vector of N bits
  - Include the bits in the indexing of the branch table
- Pros/Cons: Same BR instruction may have multiple entries in the branch table

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

This code write zeroes to the rightmost diagonal of a matrix (N is larger than 1000)

```
for (i = 0; i < N)
   for (j = 0; j < N) {
       if (j > i)
          x[i,j] = 1
       else
          x[i,j] = 0
   }
```

0	1	1	1	1	1	1
0	0	1	1	1	1	1
0	0	0	1	1	1	1
0	0	0	0	1	1	1
0	0	0	0	0	1	1
0	0	0	0	0	0	1
0	0	0	0	0	0	0

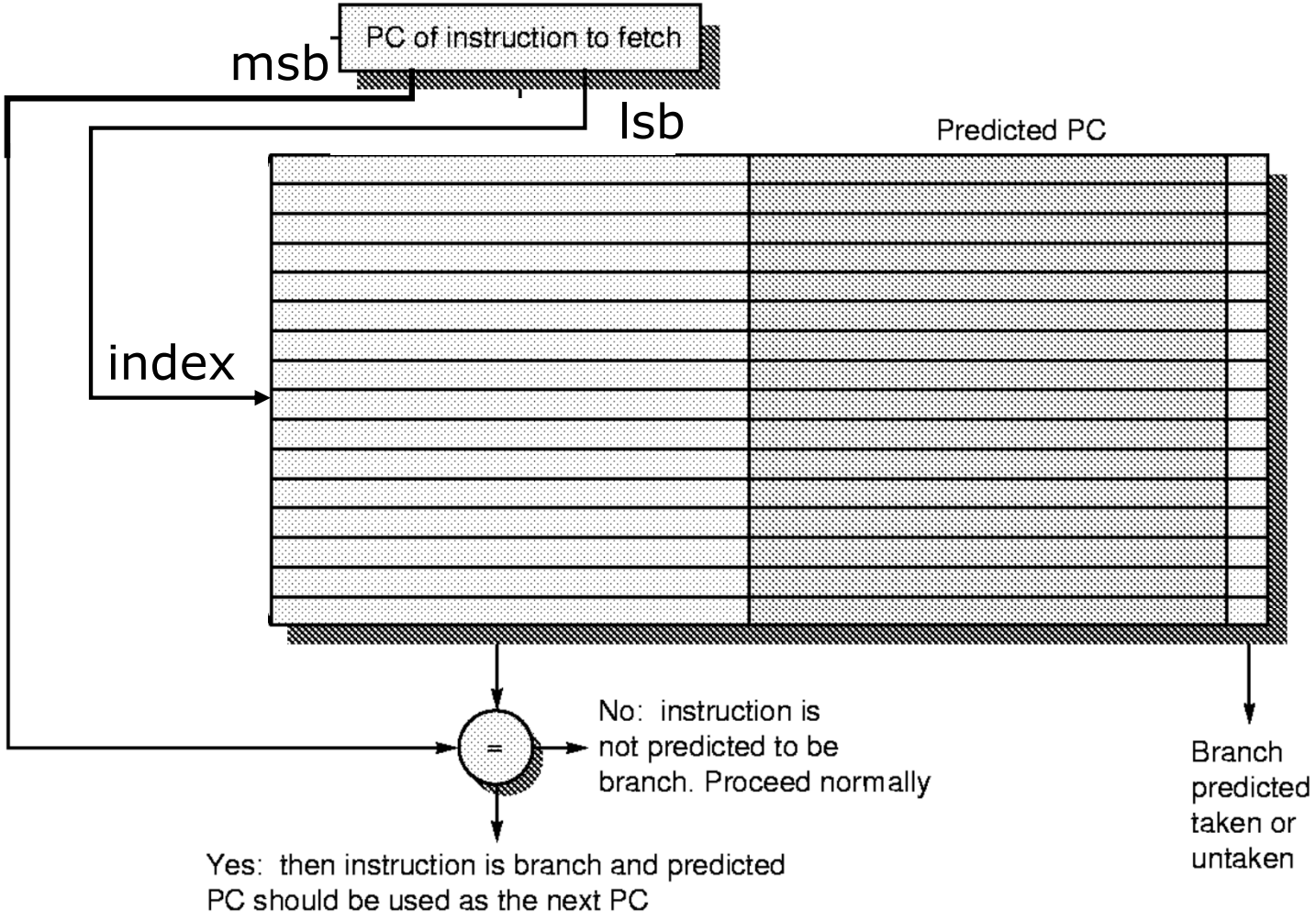


UPPSALA UNIVERSITET

Tournament prediction

- Issues:
  - No one predictor suits all applications
- Approach:
  - Implement several predictors and dynamically select the most appropriate one
- Performance example SPEC98:
  - 2-bit prediction: 7% miss prediction
  - (2,2) 2-level, 2-bit: 4% miss prediction
  - Tournaments: 3% miss prediction

Branch target buffer (BTB)



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

UPPSALA UNIVERSITET

Putting it together

- BTB stores info about taken instructions
- Combined with a separate branch history table
- Instruction fetch pipeline stage highly integrated for branch optimizations
- Speculative execution prohibited from side-effects until the outcome of the branch is known.

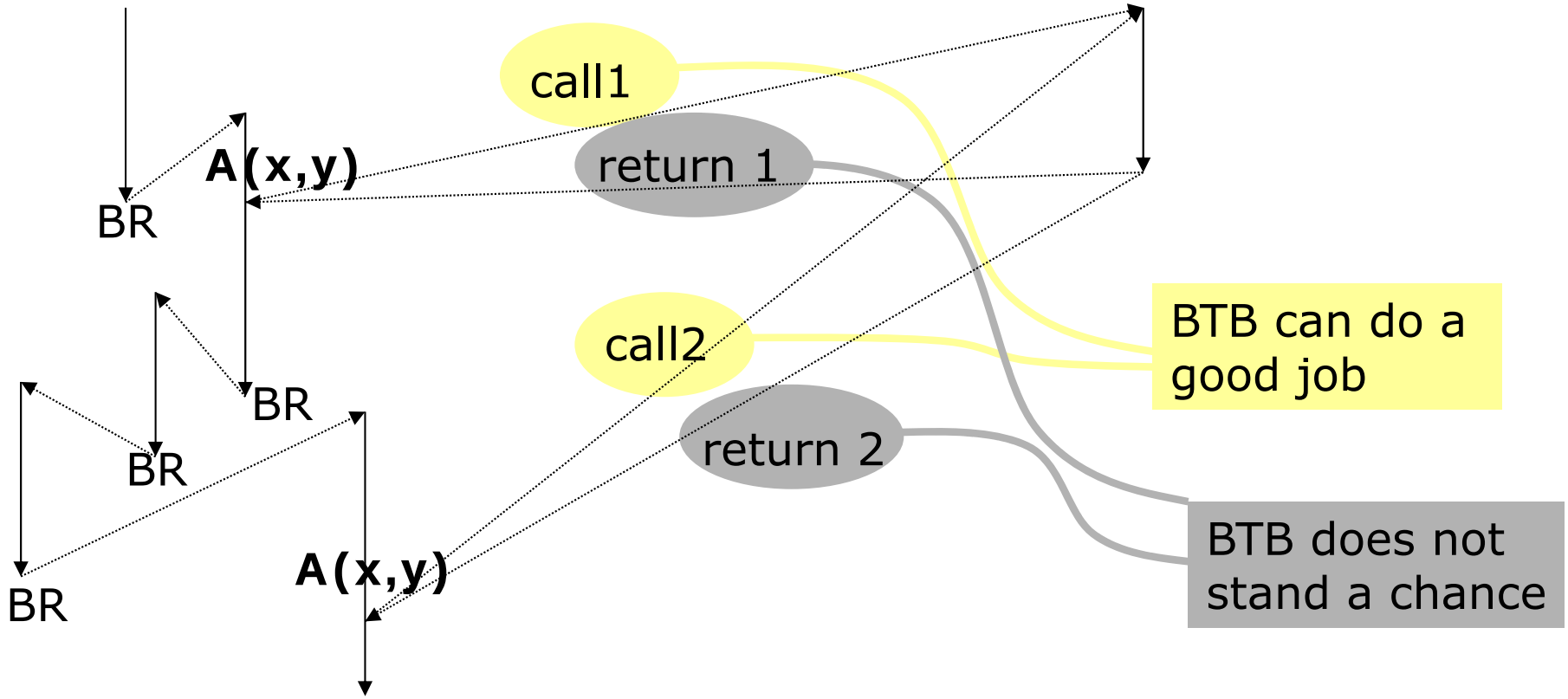
UPPSALA UNIVERSITET

Folding branches

- BTB may also contains the next few instructions at the destination address
- Unconditional branches (and some cond as well) branches execute in zero cycles
  - Execute the dest instruction instead of the branch *(if there is a hit in the BTB at the IF stage)*
  - Called "Branch folding"

Procedure calls & BTB

BTB can predict "normal" branches



UPPSALA UNIVERSITET

Return address stack

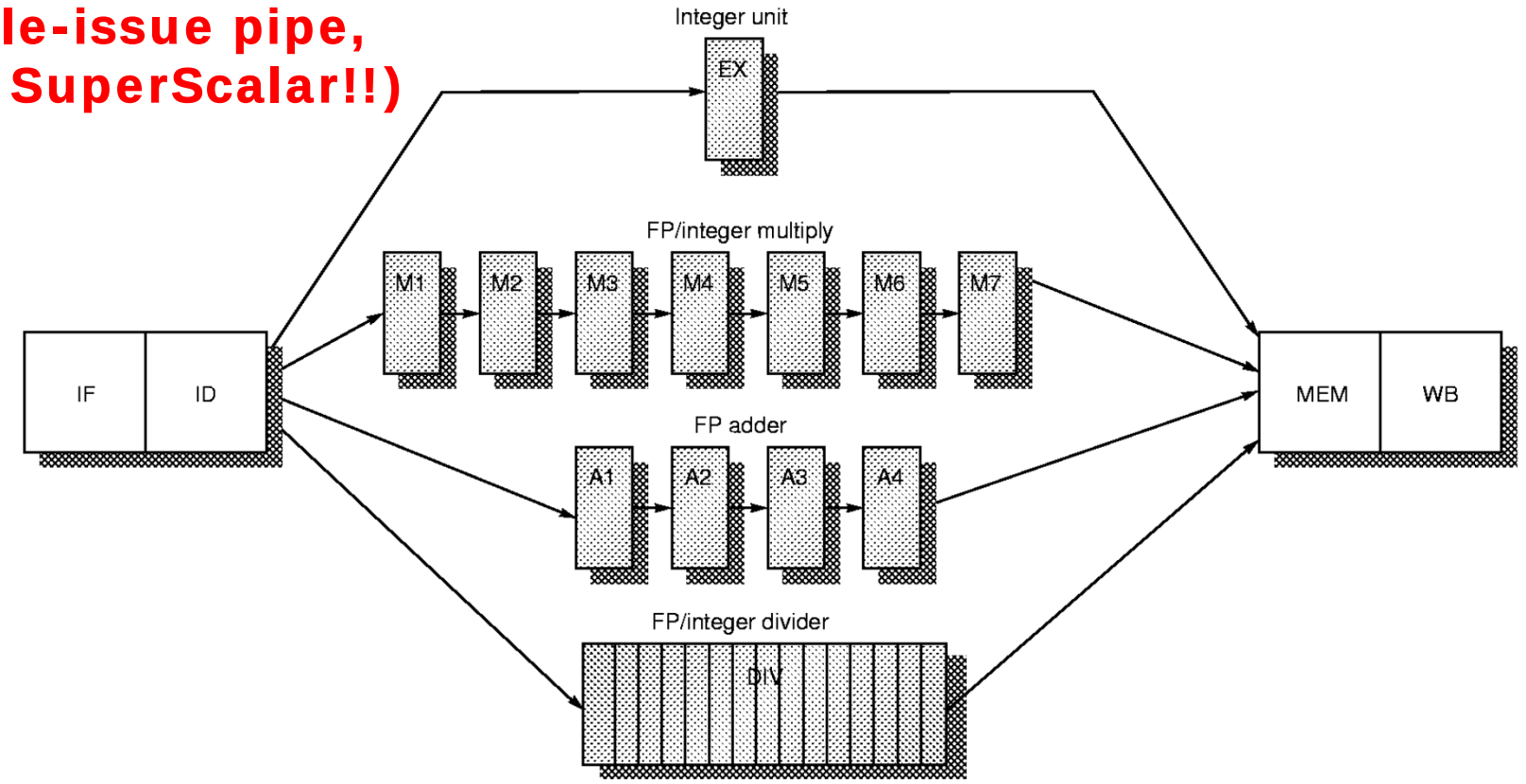
- Popular subroutines are called from many places in the code.
- Branch prediction may be confused!!
- May hurt other predictions
- New approach:
  - Push the return address on a [small] stack at the time of the call
  - Pop addresses on return

Overlapping Execution & Out-of-order Execution

Erik Hagersten
Uppsala University
Sweden

Multicycle operations in the pipeline (e.g., floating point)

(Single-issue pipe, not a SuperScalar!!)



- Integer unit: Handles integer instructions, branches, and loads/stores
- Other units: May take several cycles each. Some units are pipelined (mult,add) others are not (div)

UPPSALA UNIVERSITET

Parallelism between integer and FP instructions

MULTD	F2,F4,F6	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADDD	F8,F10,F12		IF	ID	*A1*	A2	A3	**A4**	MEM	WB		
SUBI	R2,R3,#8			IF	ID	***EX***	MEM	WB				
LD	F14,0(R2)				IF	ID	*EX*	**MEM**	WB			

How to avoid structural and RAW hazards:

Hazard: Stall in ID stage when:

- The functional unit can be occupied
- Many instructions can reach the WB stage at the same time

RAW hazards:

- Normal bypassing from MEM and WB stages
- Stall in ID stage if any of the source operands is a destination operand of an instruction in any of the FP functional units

AVDARK 2013

UPPSALA UNIVERSITET

WAR and WAW hazards for multicycle operations

WAW Example:

DIVF F0,F2,F4 FP divide 24 cycles

...

SUBF F0,F8,F10 FP sub 3 cycles

SUB finishes before DIV ; out-of-order completion

WAW hazards are avoided by:

- stalling the SUBF until DIVF reaches the MEM stage, or
- disabling the write to register F0 for the DIVF instruction

WAR hazards are (still) a non-issue because operands are read in program order (in-order)

Dynamic Instruction Scheduling

Key idea: allow subsequent independent instructions to proceed

```
DIVD    F0,F2,F4    RAW  ; takes long time
ADDD    F10,F0,F8        ; stalls waiting for F0
SUBD    F12,F8,F13       ; Let this instr. bypass the ADDD
```

Can we do out-of-order execution?

(& out-of-order completion)

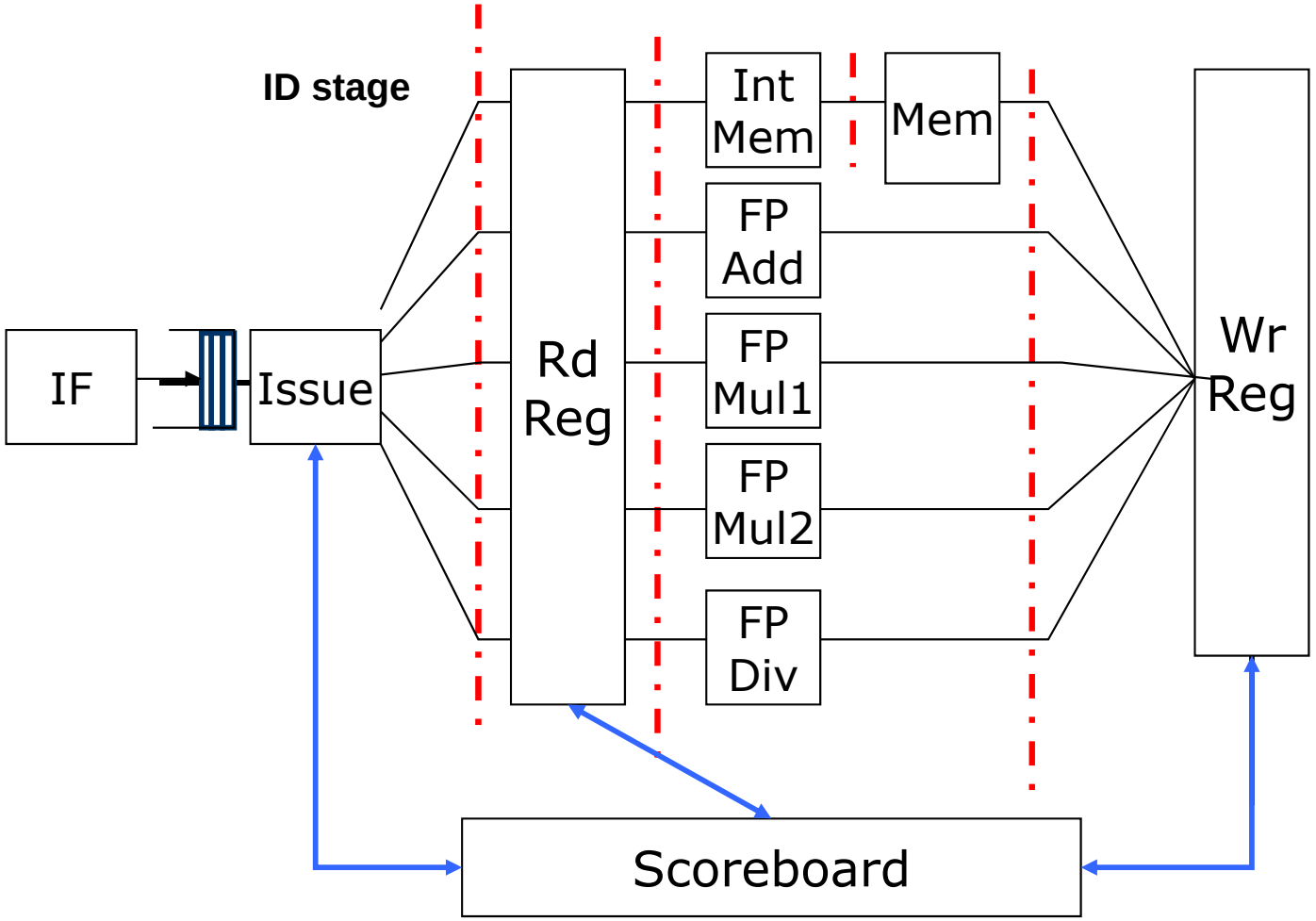
➔ Opportunities and problems…

Two historical schemes used in “recent” machines:

Tomasulo in IBM 360/91 in 1967 (also in Power-2)

Scoreboard dates back to CDC 6600 in 1963

Simple Scoreboard Pipeline



Issue: Decode and check for structural hazards and WAW & WAR.

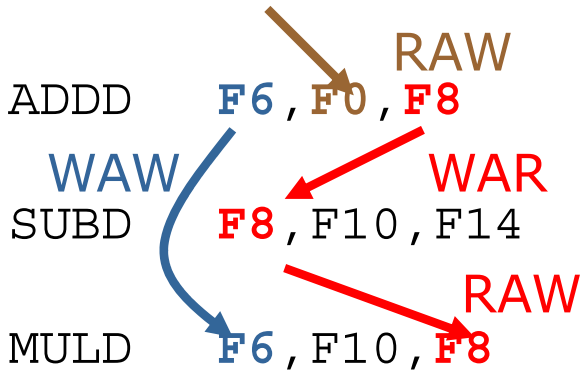
Rd Reg: Each Reg has a scoreboard bit: Set for W instr.

R instr: Wait until scoreboard bit is cleared.

Wr Reg: Clear corresponding scoreboard bit.

A more complicated example

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



UPPSALA UNIVERSITET

Tomasulo's Algorithm

- IBM 360/91 mid 60's
- High performance without compiler support
- Extended for modern architectures
- In most modern high-performance implementations

A more complicated example

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

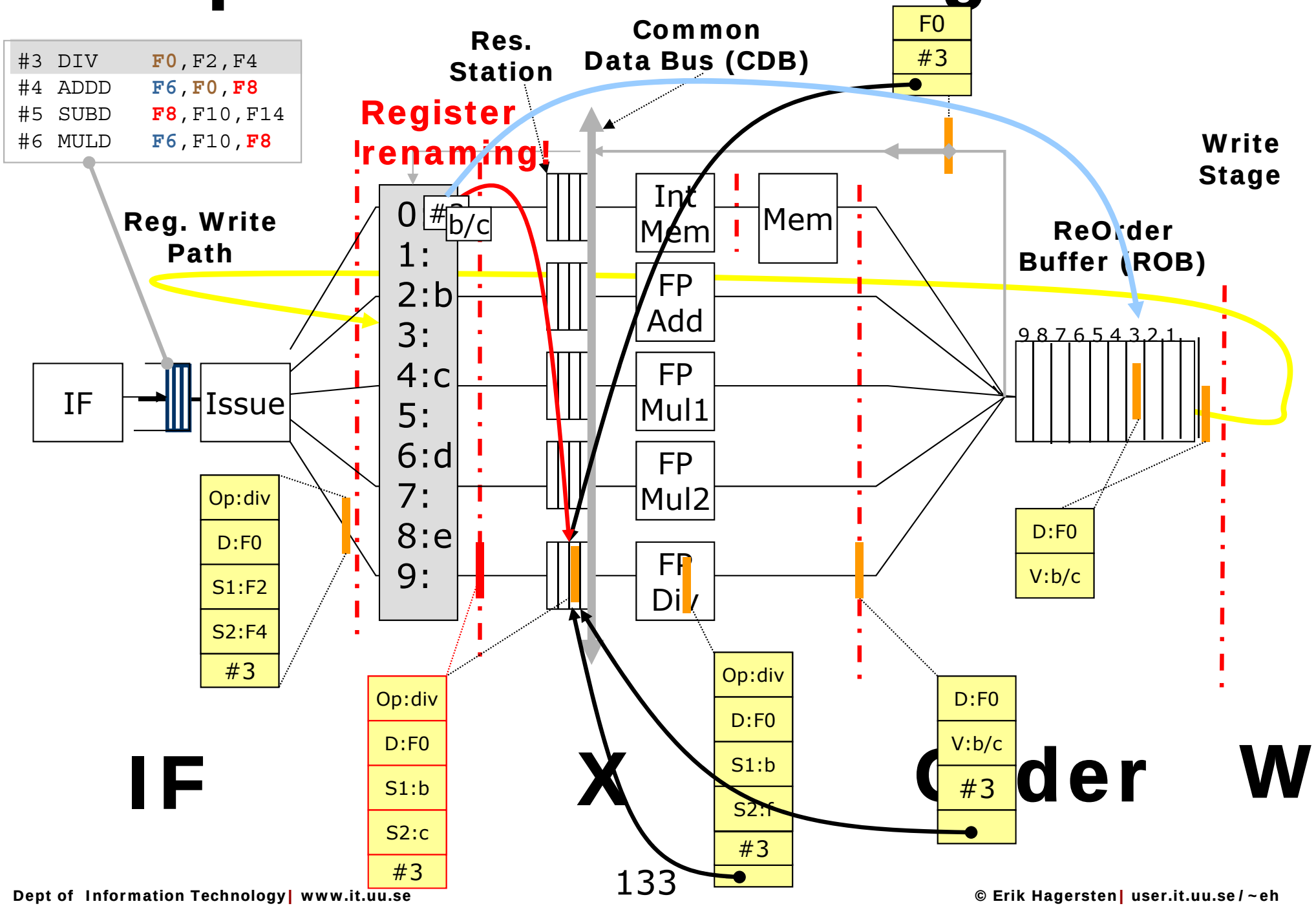
RAW (DIV F0 → ADDD F0)
WAW (ADDD F6 → MULD F6)
WAR (ADDD F8 → SUBD F8)
RAW (SUBD F8 → MULD F8)

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

Register Renaming:

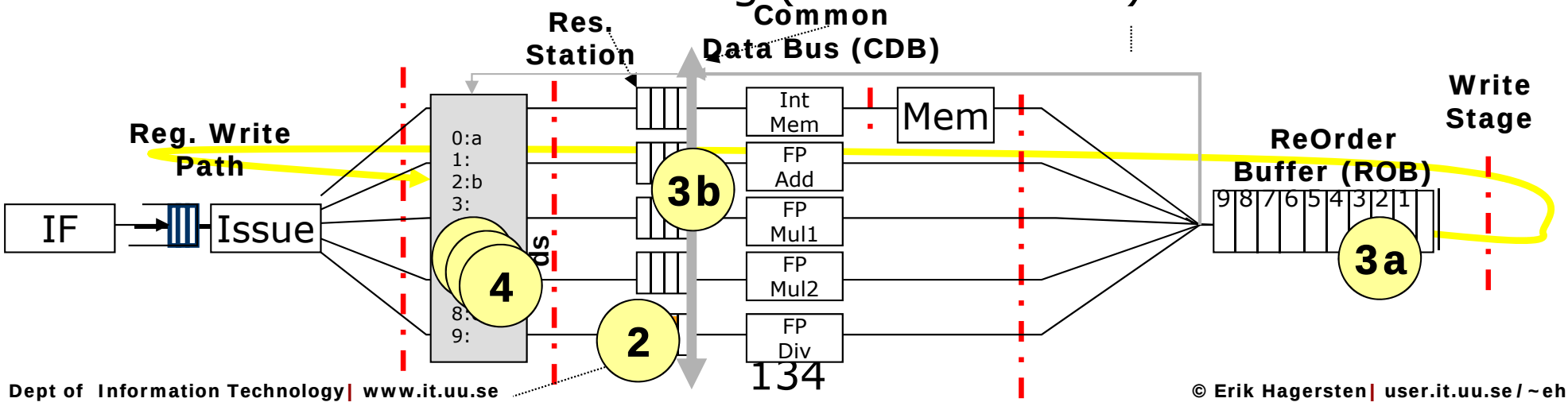
```
DIV    F0,F2,F4
ADDD   F6,F0,F8
SUBD   tmp1,F10,F14   ;can be executed right away
MULD   tmp2,F10,tmp1  ;delayed a few cycles
```

Simple Tomasulo's Algorithm

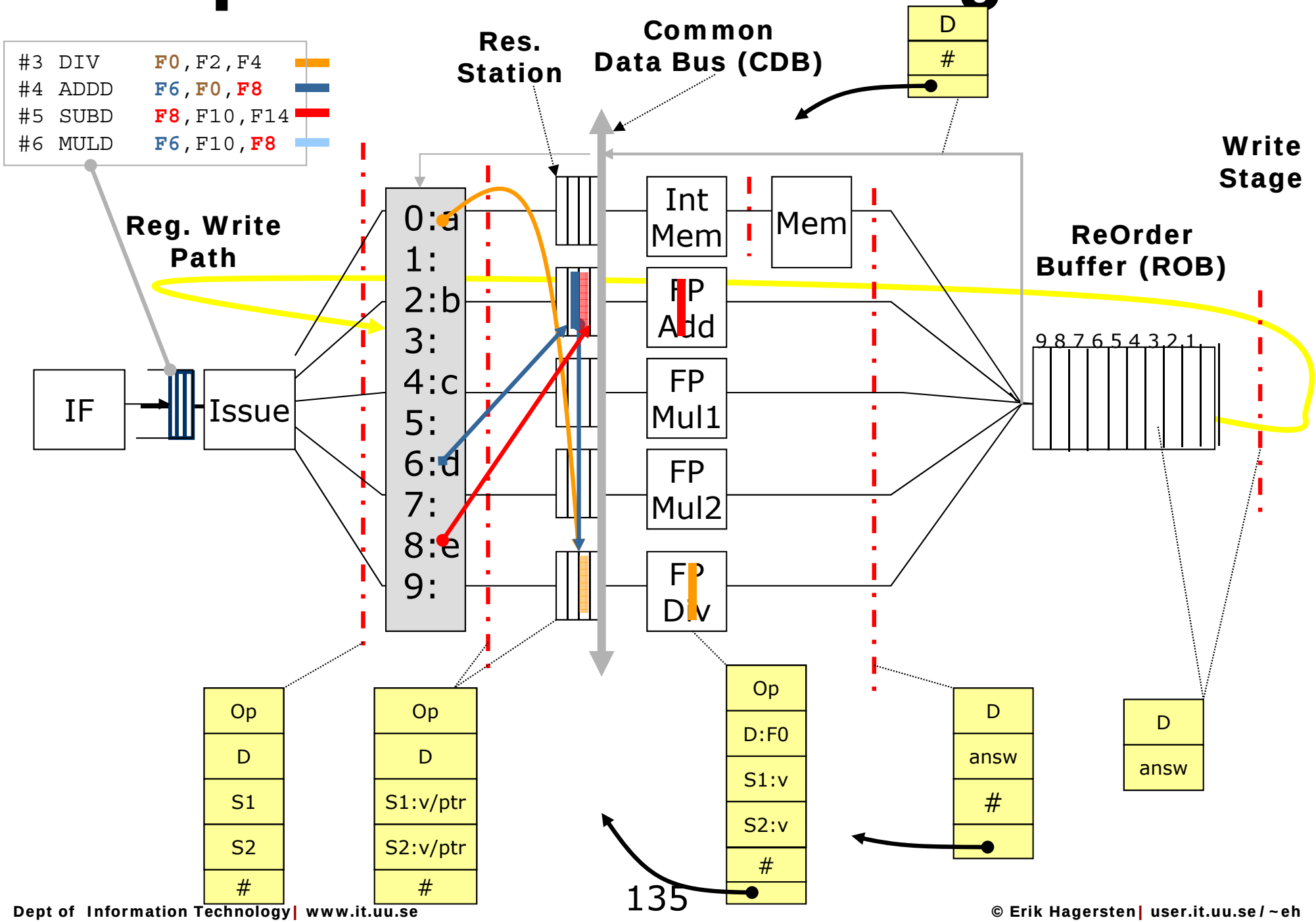


Tomasulo's: What is going on?

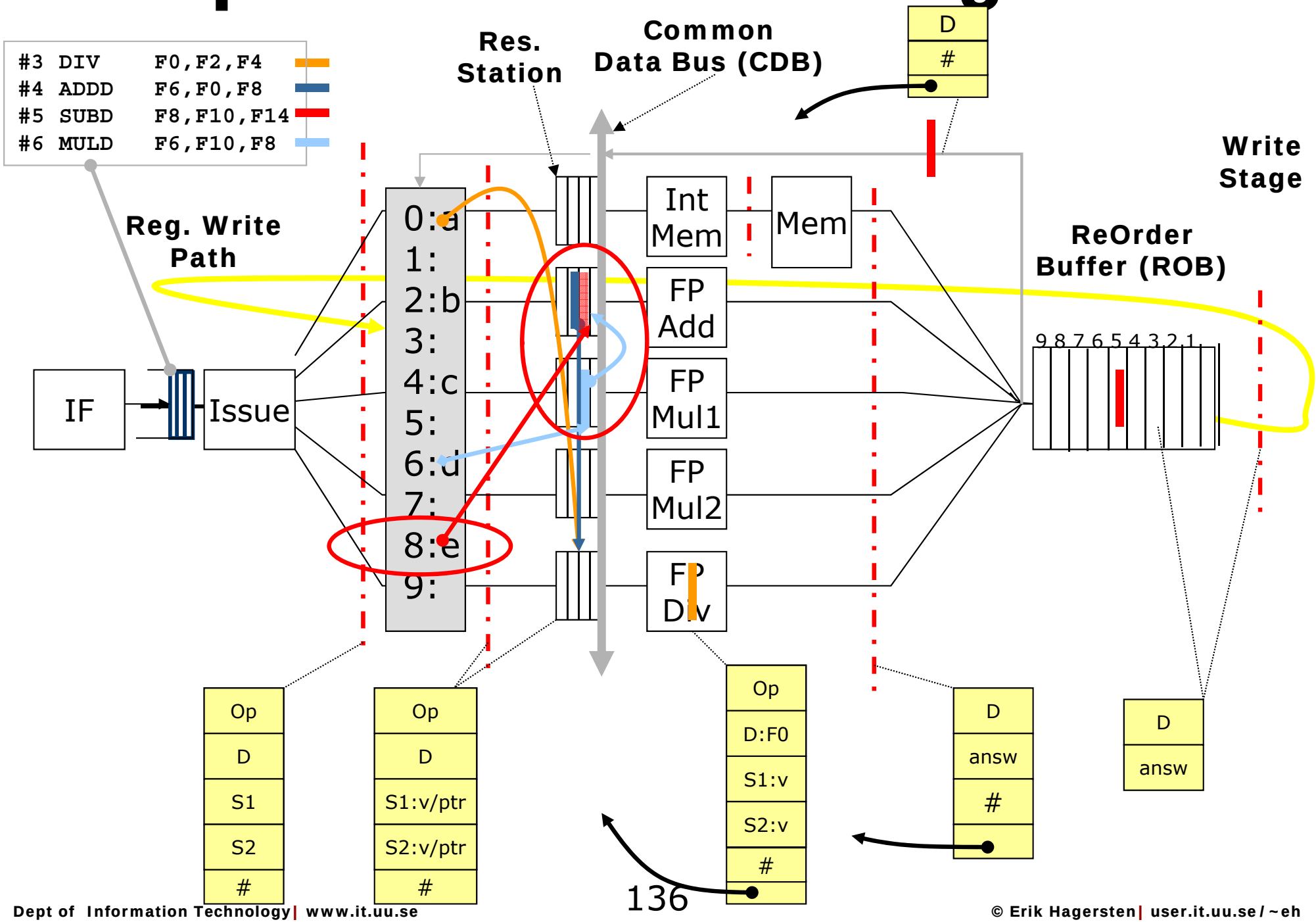
1. Read Register:
   - Rename DestReg to the Res. Station location
2. Wait for all RAW dependencies at Res. Station
3. After Execution
   a) Put result in Reorder Buffer (ROB)
   b) Broadcast result on CDB to all waiting instructions
   c) Rename DestReg to the ROB location
4. When all preceeding instr.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



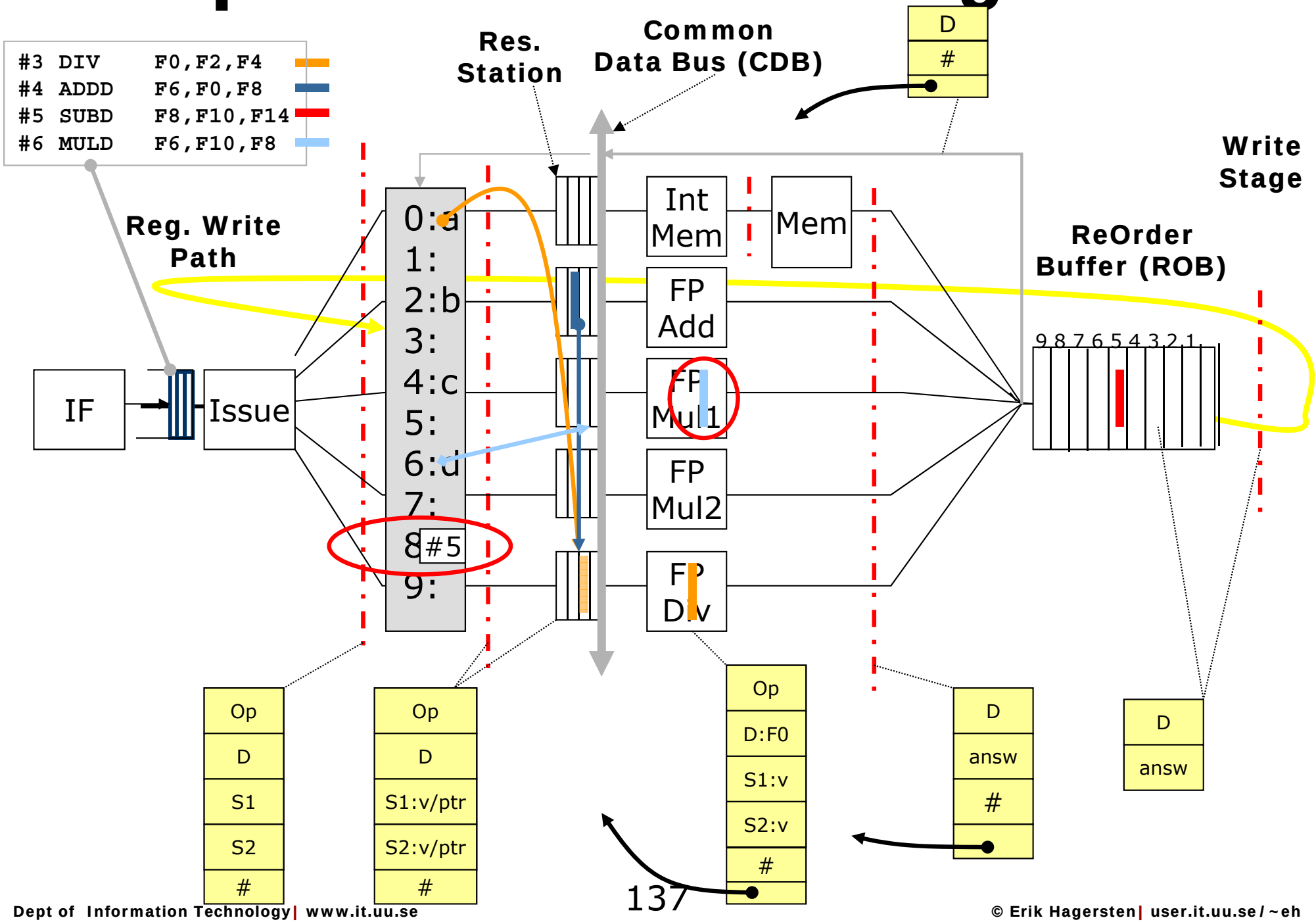
Simple Tomasulo's Algorithm



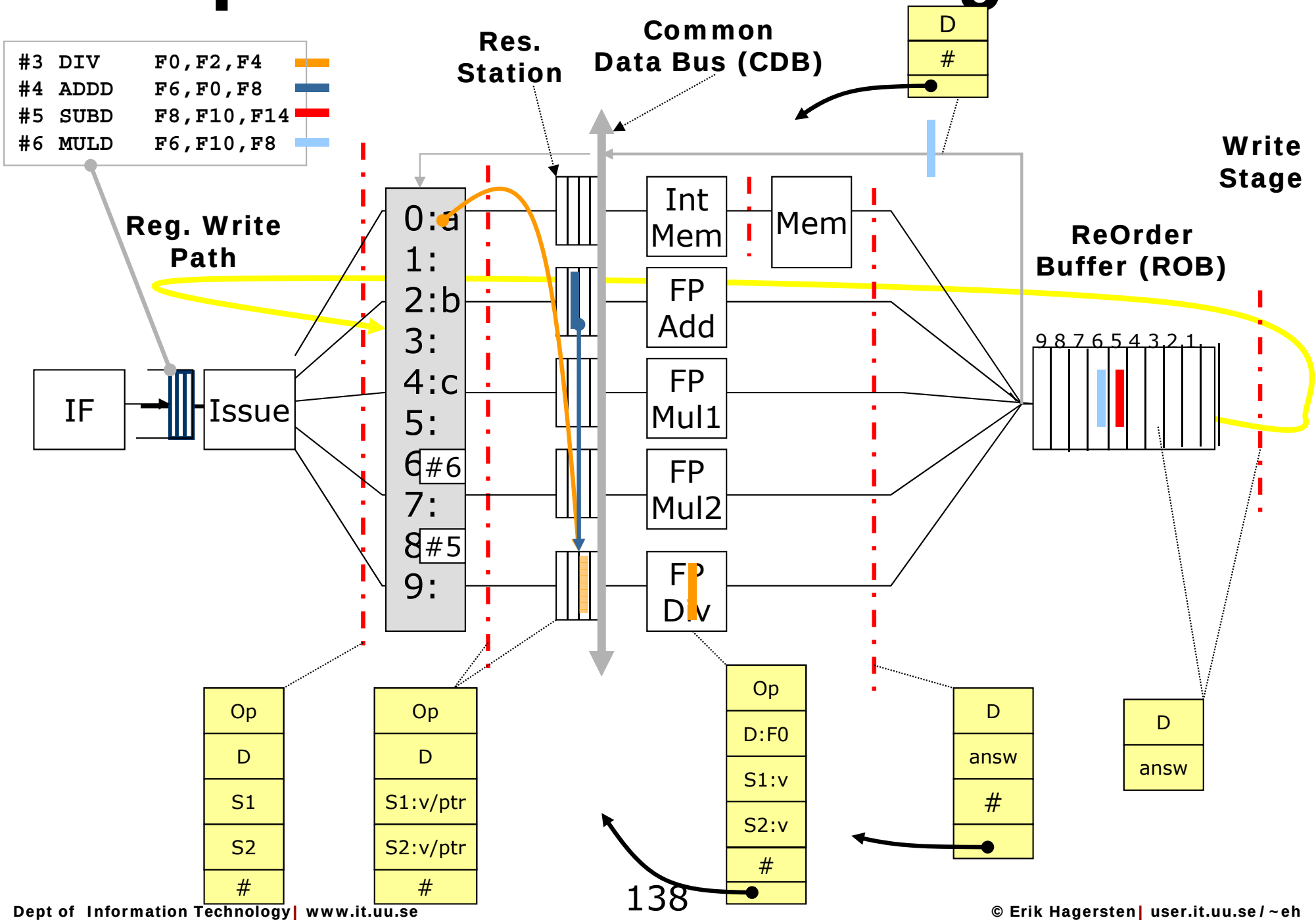
Simple Tomasulo's Algorithm



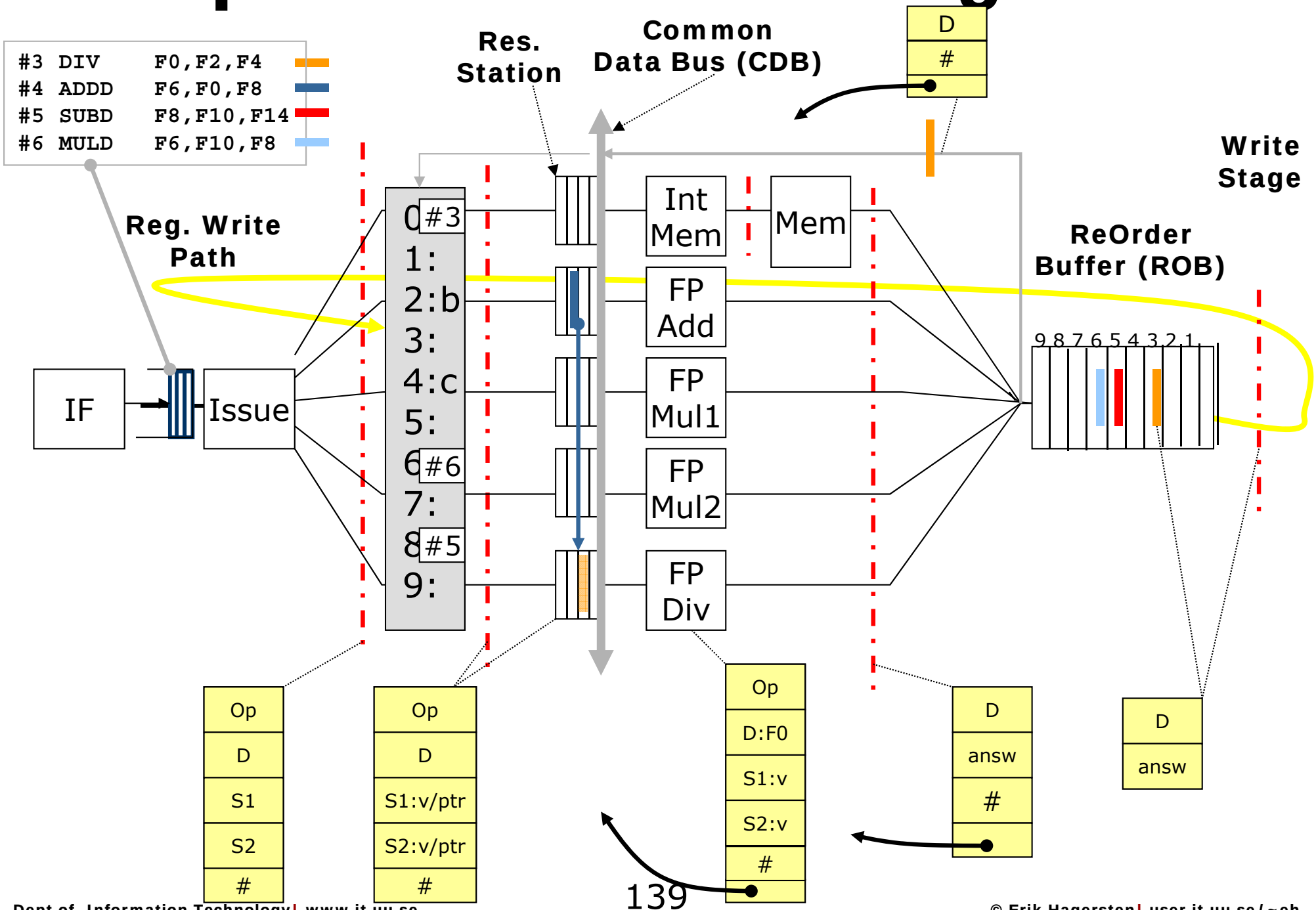
Simple Tomasulo's Algorithm



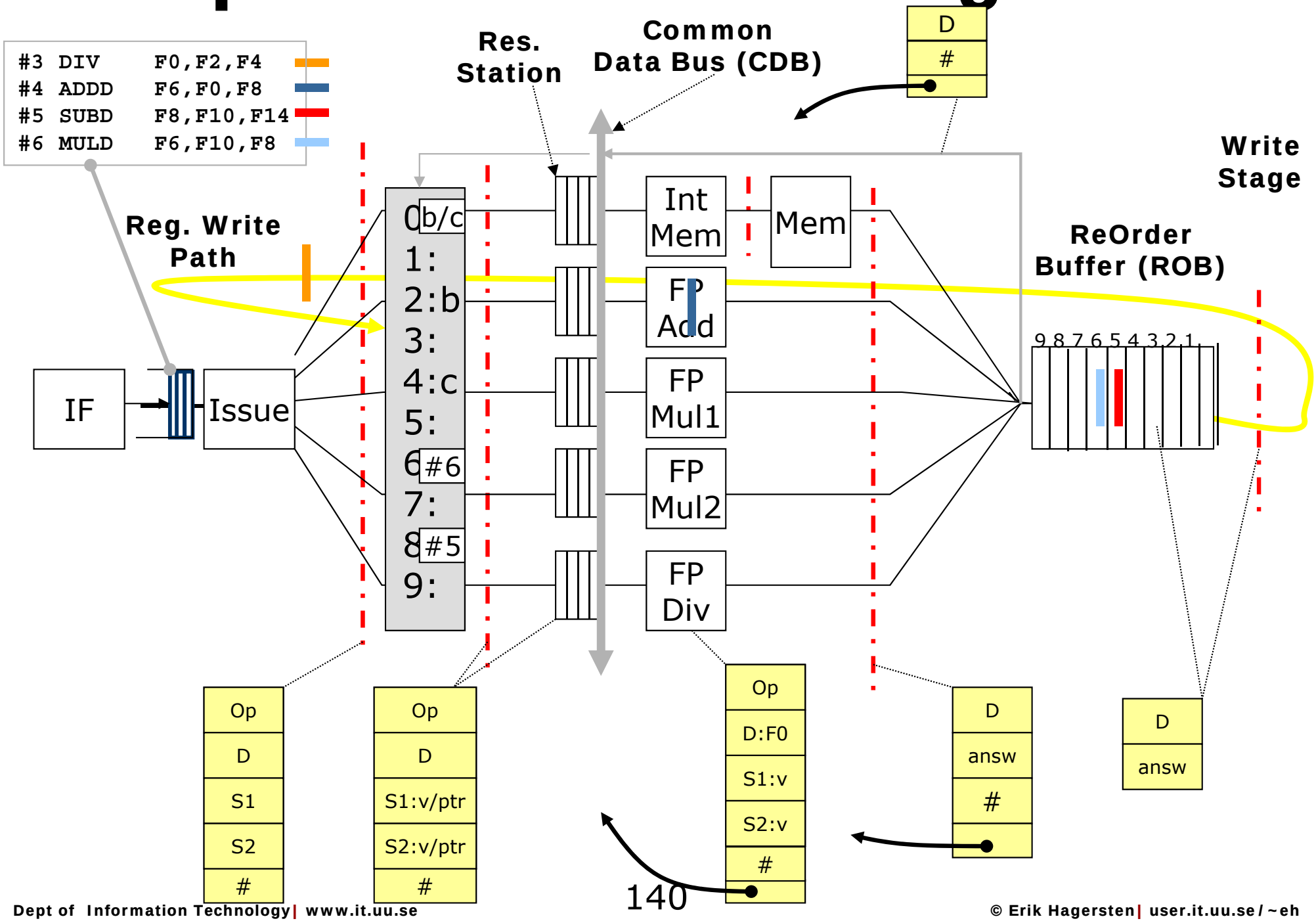
Simple Tomasulo's Algorithm



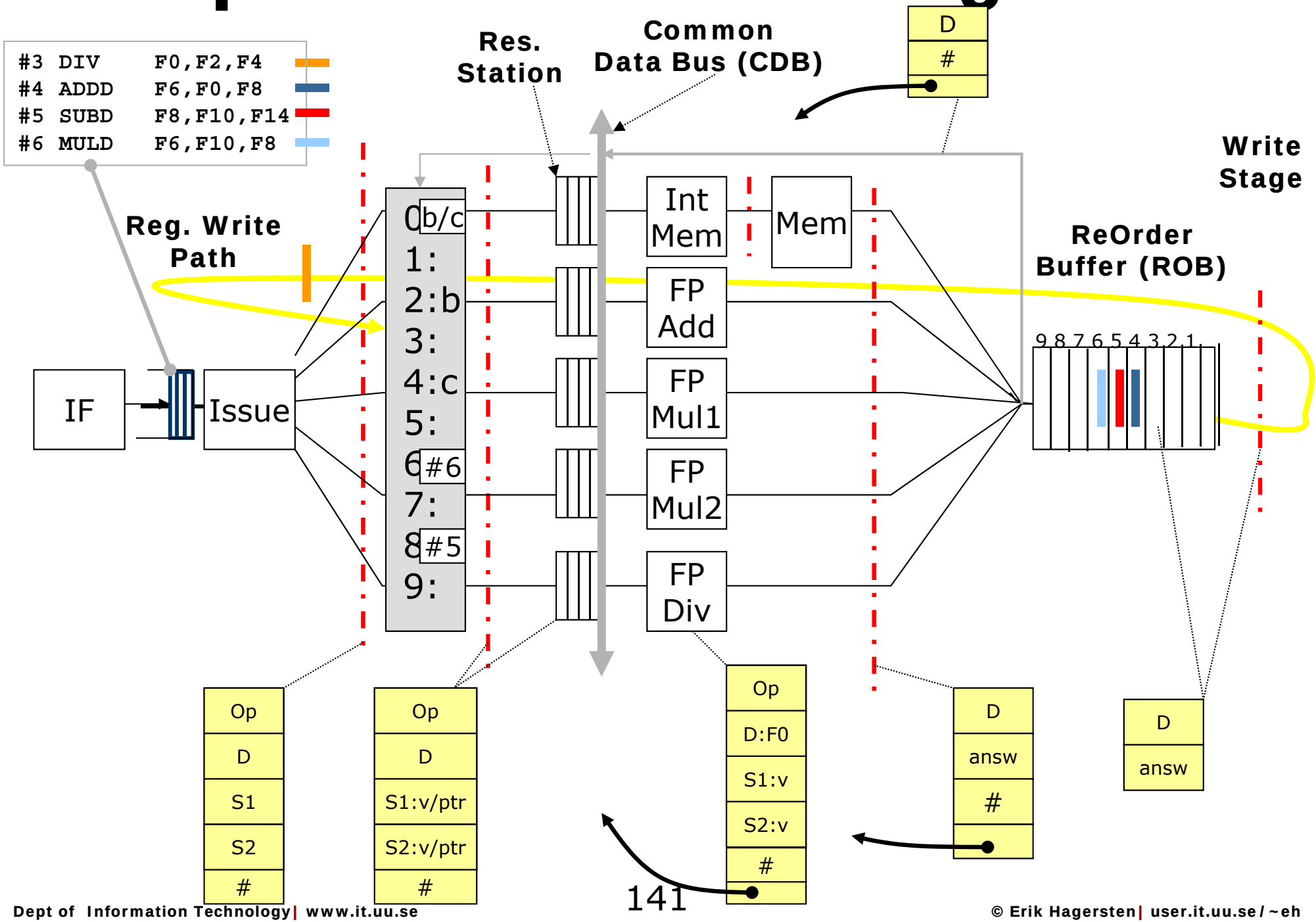
Simple Tomasulo's Algorithm



Simple Tomasulo's Algorithm

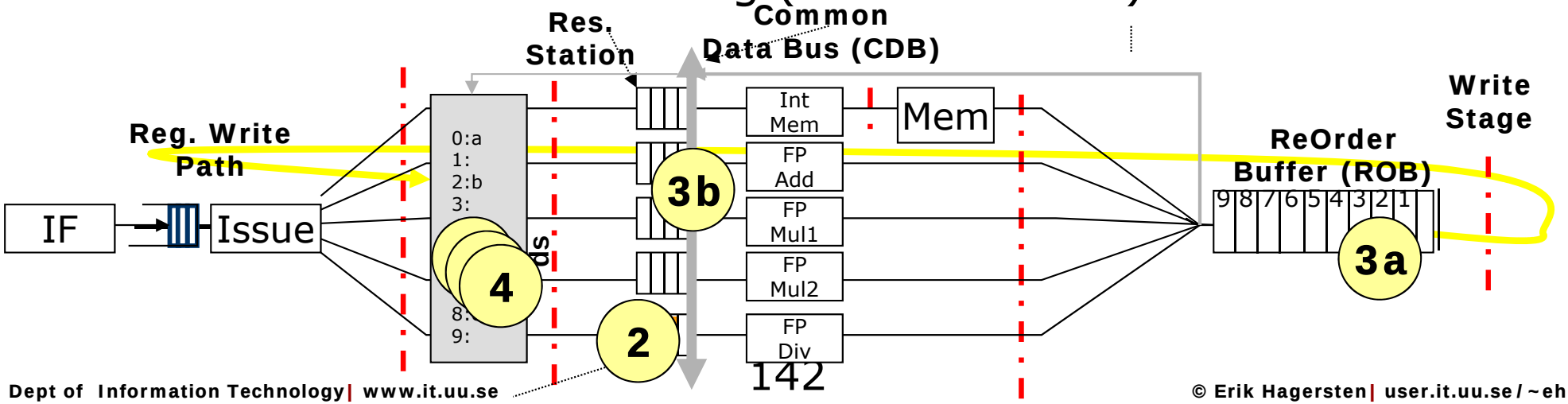


Simple Tomasulo's Algorithm



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

Load/Stores (??)

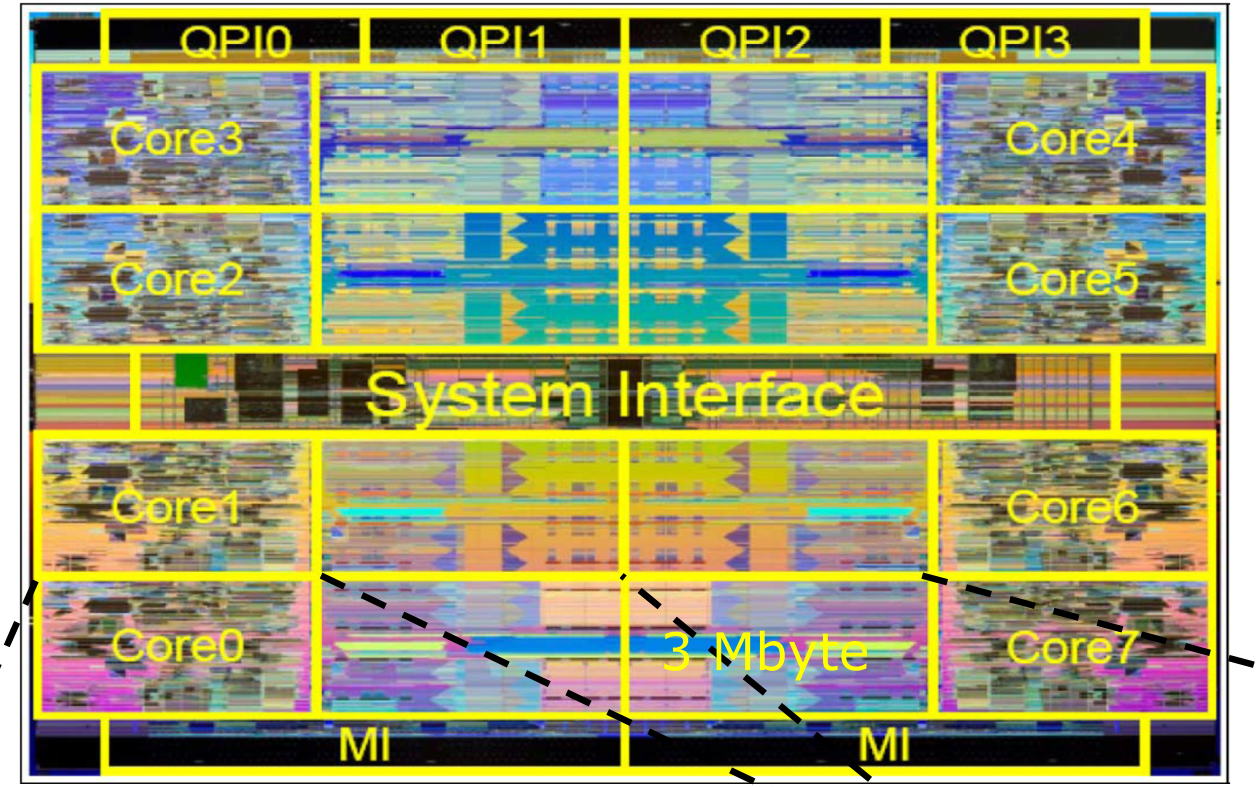
- Loads to memory locations that were changed between register-read and commit an issue.
- Load-Store queue: a very associative and power-hungry beast handling such reads (and coherence invalidations)
- Energy grows more than linear with the ROB window size

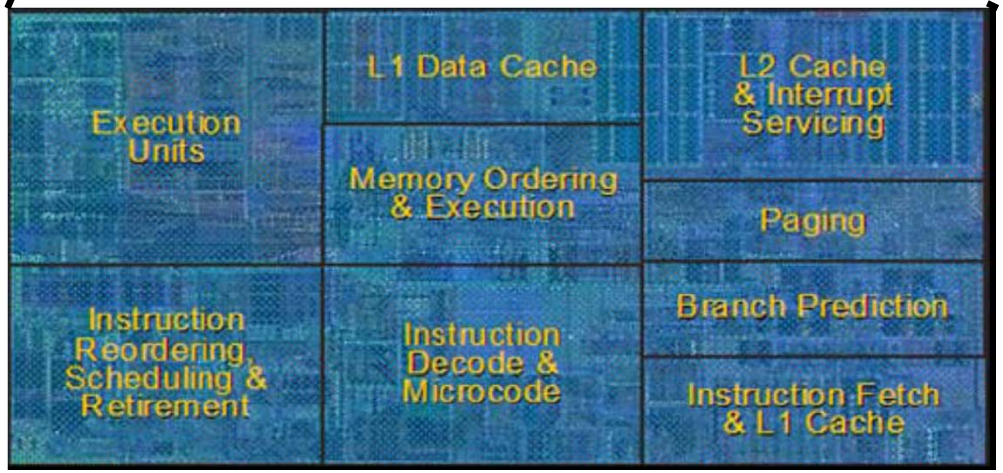
UPPSALA UNIVERSITET

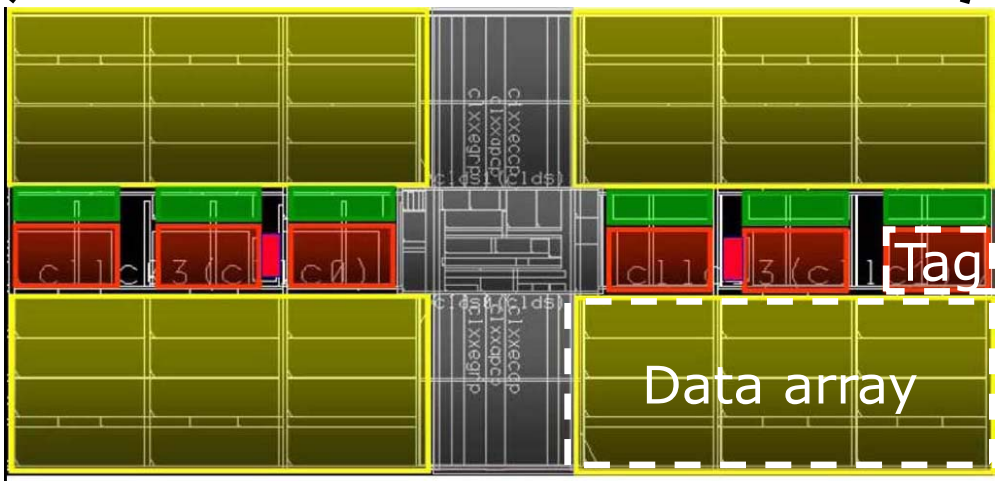
In a nutshell

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

How is the silicon used (i7-Ex)?

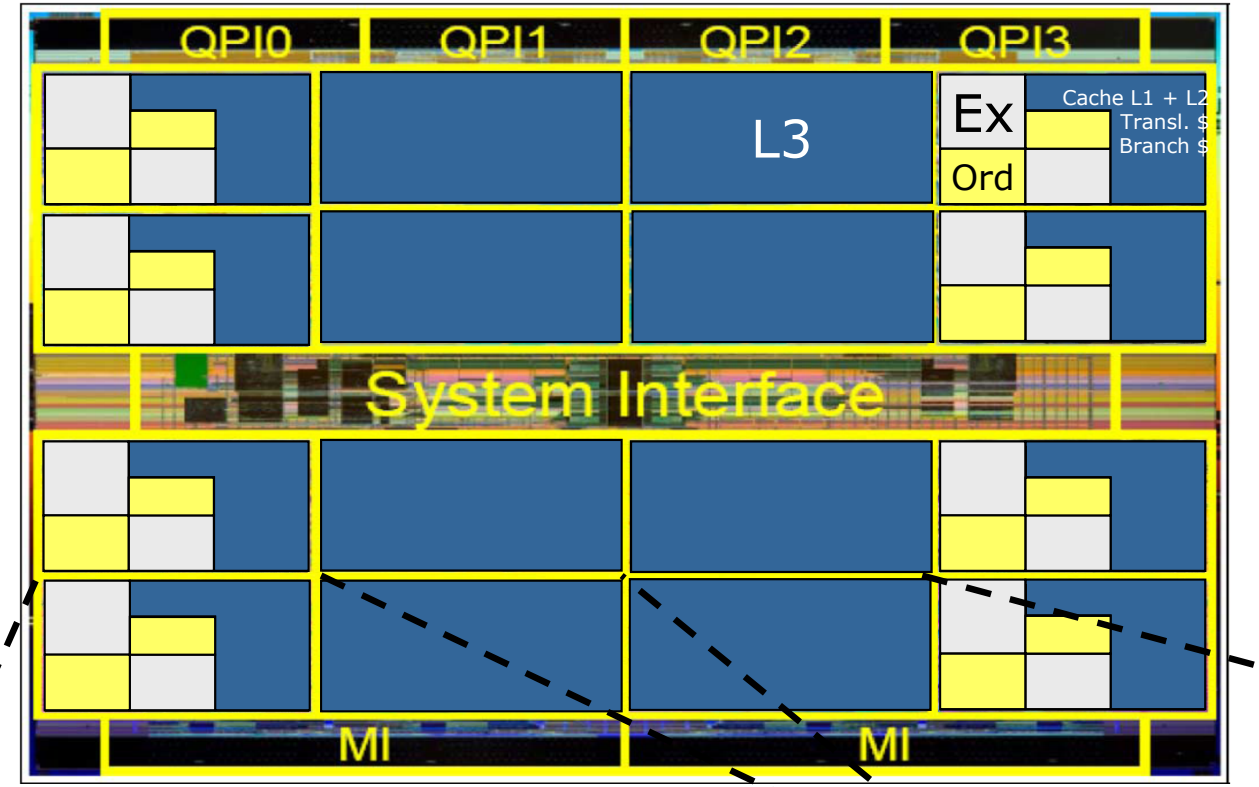


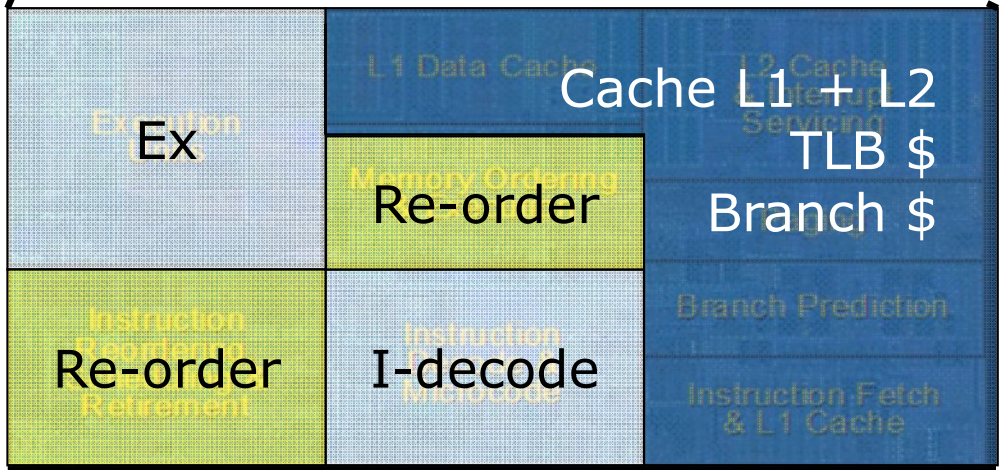


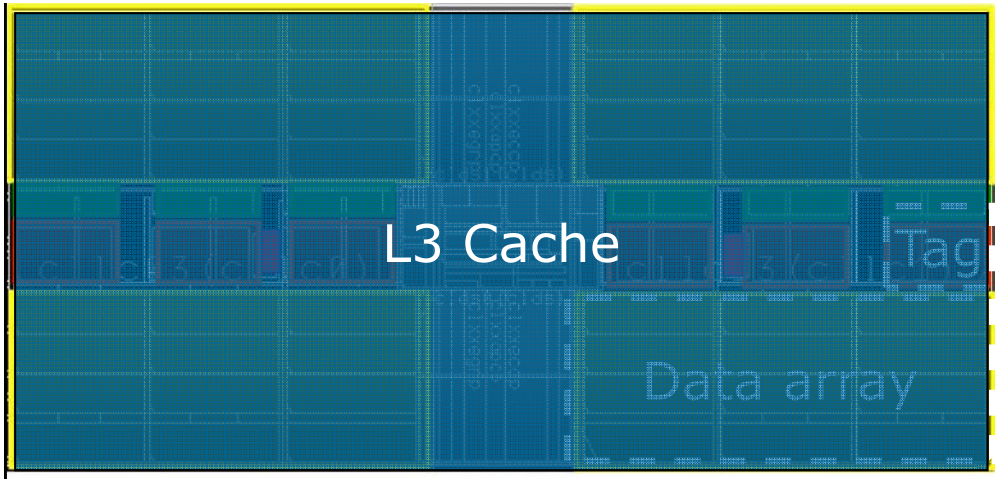


Source: JSSC Jan 2010, Rusu et. al

How is the silicon used?





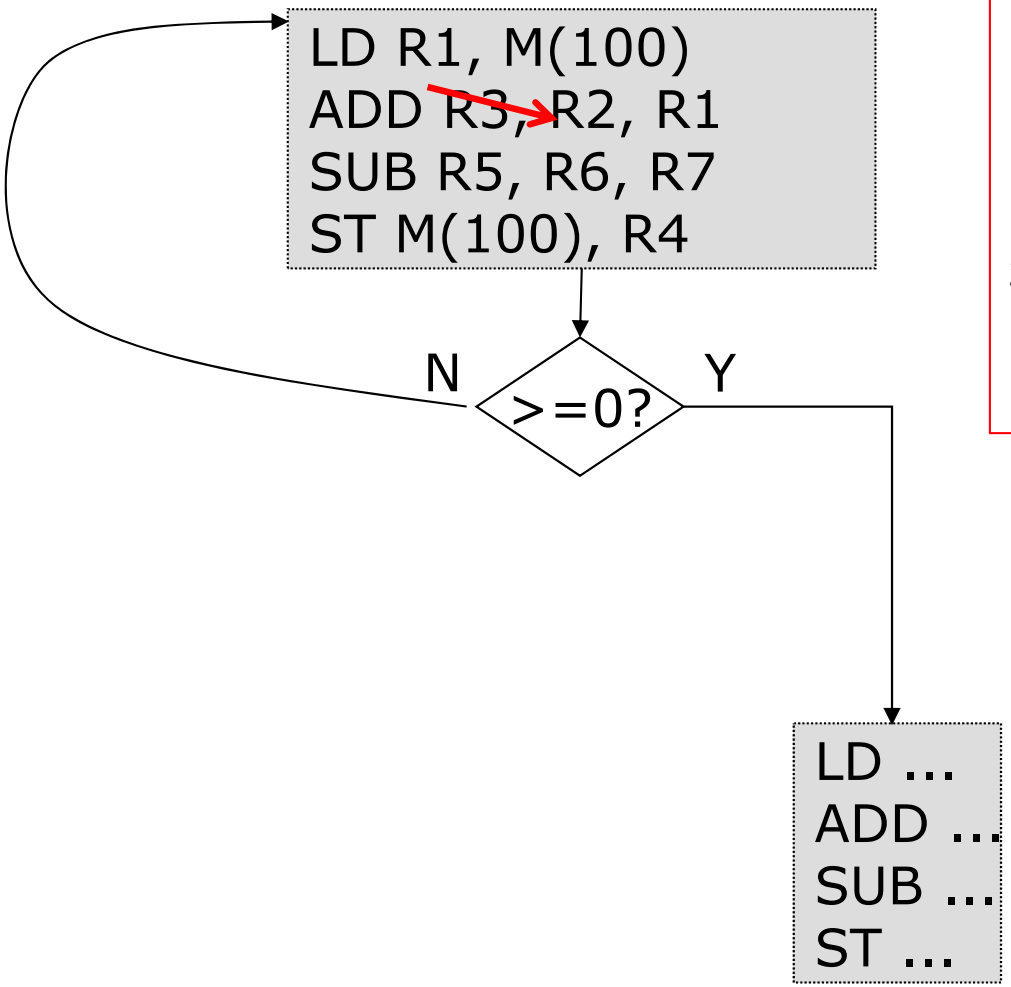


Source: JSSC Jan 2010, Rusu et. al

UPPSALA
UNIVERSITET

Why is this such a big deal?

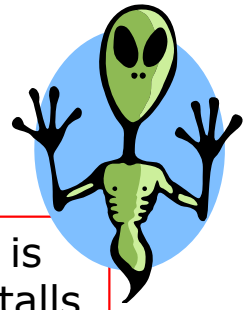
Fix 1: Out-of order execution:
Improving ILP

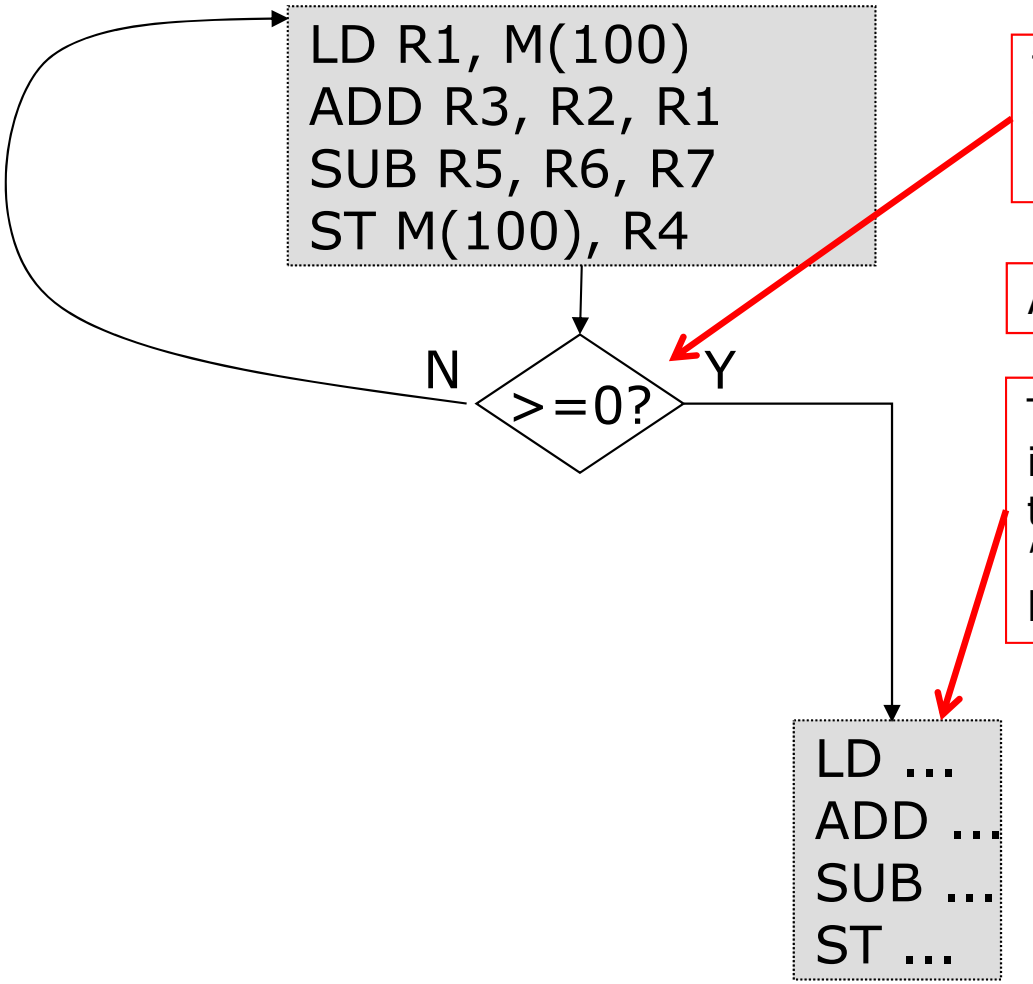


The HW may execute the instructions of a ”basis block” in a different order, but will make the ”side-effects” of the instructions appear in order.

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





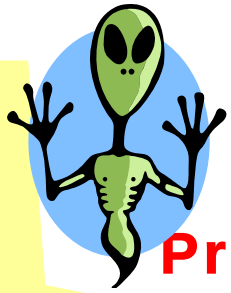
The HW can guess if the branch is taken or not and avoid branch stalls if the guess is correct.

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

"Predict taken"

LD
ADD
SUB
ST

>=0?

Y

LD
ADD
SUB
ST

"Predict taken"

Y

>1?

LD
ADD
SUB
ST

=0?

Y

LD
ADD
SUB
ST

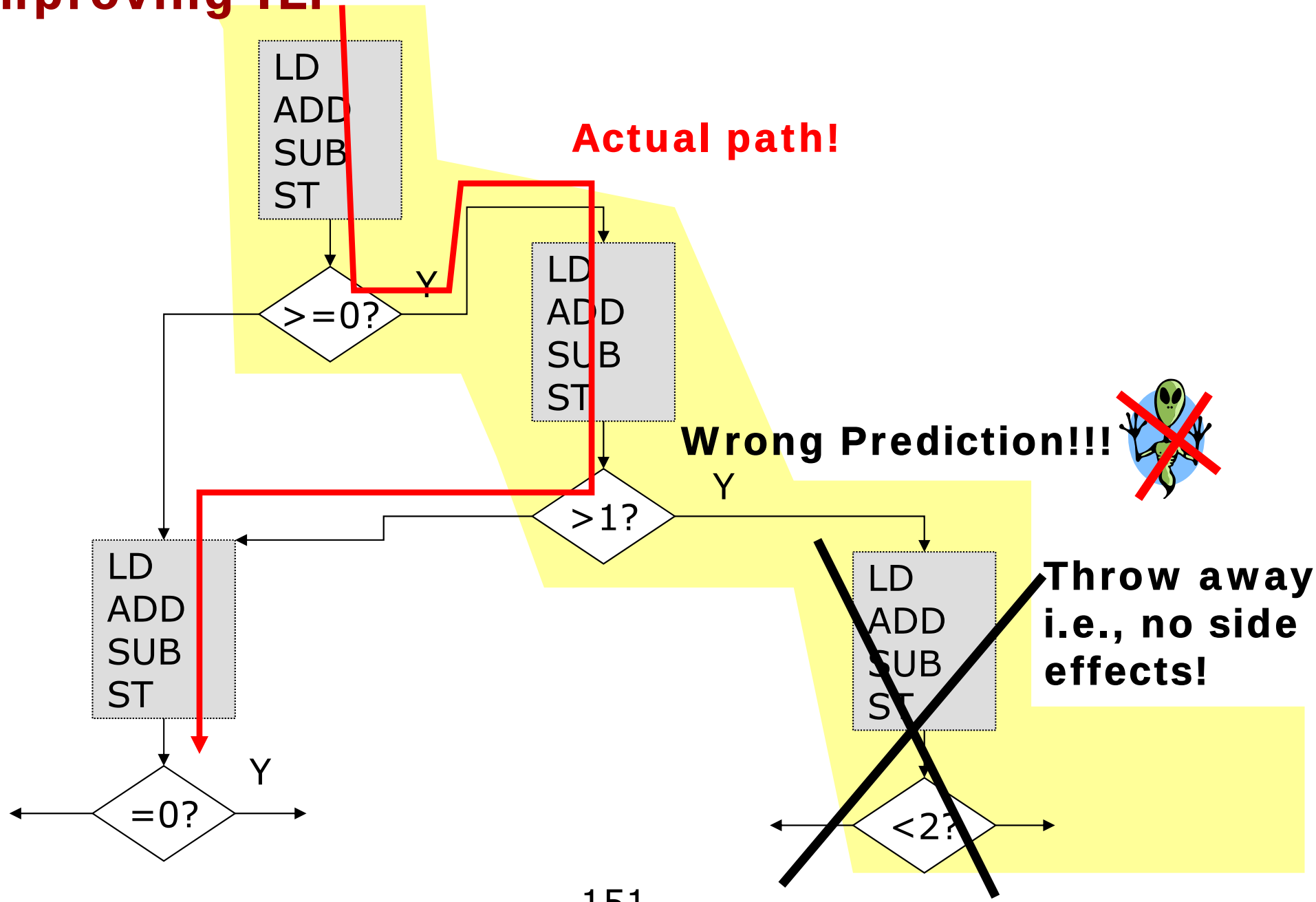
"Predict taken"

<2?

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

Fix 3: Scheduling Past Branches

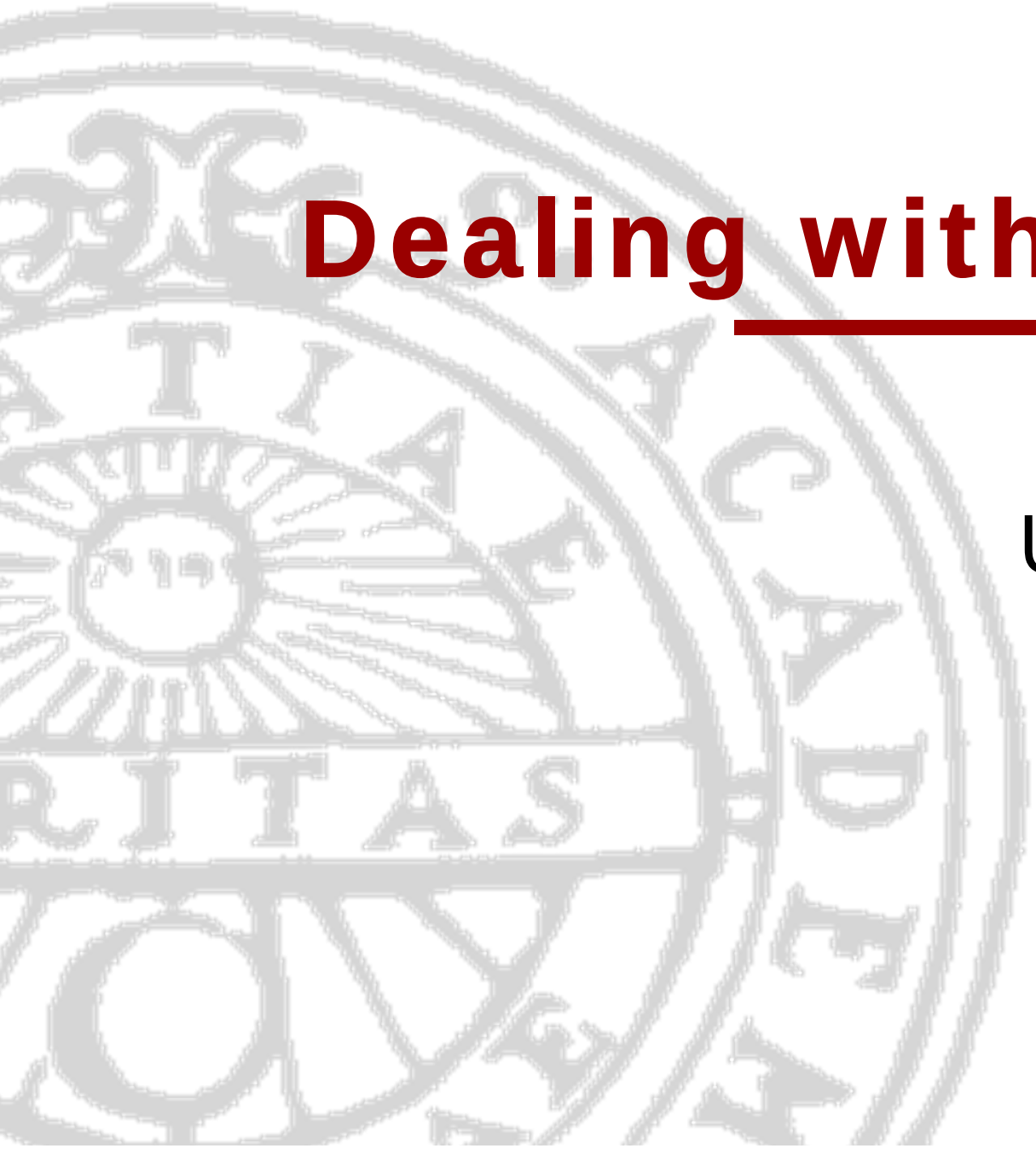
Improving ILP



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)



Dealing with Exceptions

Erik Hagersten
Uppsala University
Sweden

UPPSALA UNIVERSITET

Exception handling in pipelines

Example: Page fault from TLB may be handled by the kernel.

Must restart the instruction that causes an exception (interrupt, trap, fault) "precise interrupts" …

…as well as all instructions following it.

A solution (in-order…):

1. Force a "trap" instruction into the pipeline
2. Turn off all writes for the faulting instruction and following instructions
3. Save the PC for the faulting instruction
   - to be used in return from exception

UPPSALA UNIVERSITET

Guaranteeing the execution order

Exceptions may be generated in another order than the instruction execution order

Pipeline stage	*Problem causing exception*
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data access; misaligned memory access; memory protection violation
WB	none

AVDARK 2013

UPPSALA UNIVERSITET

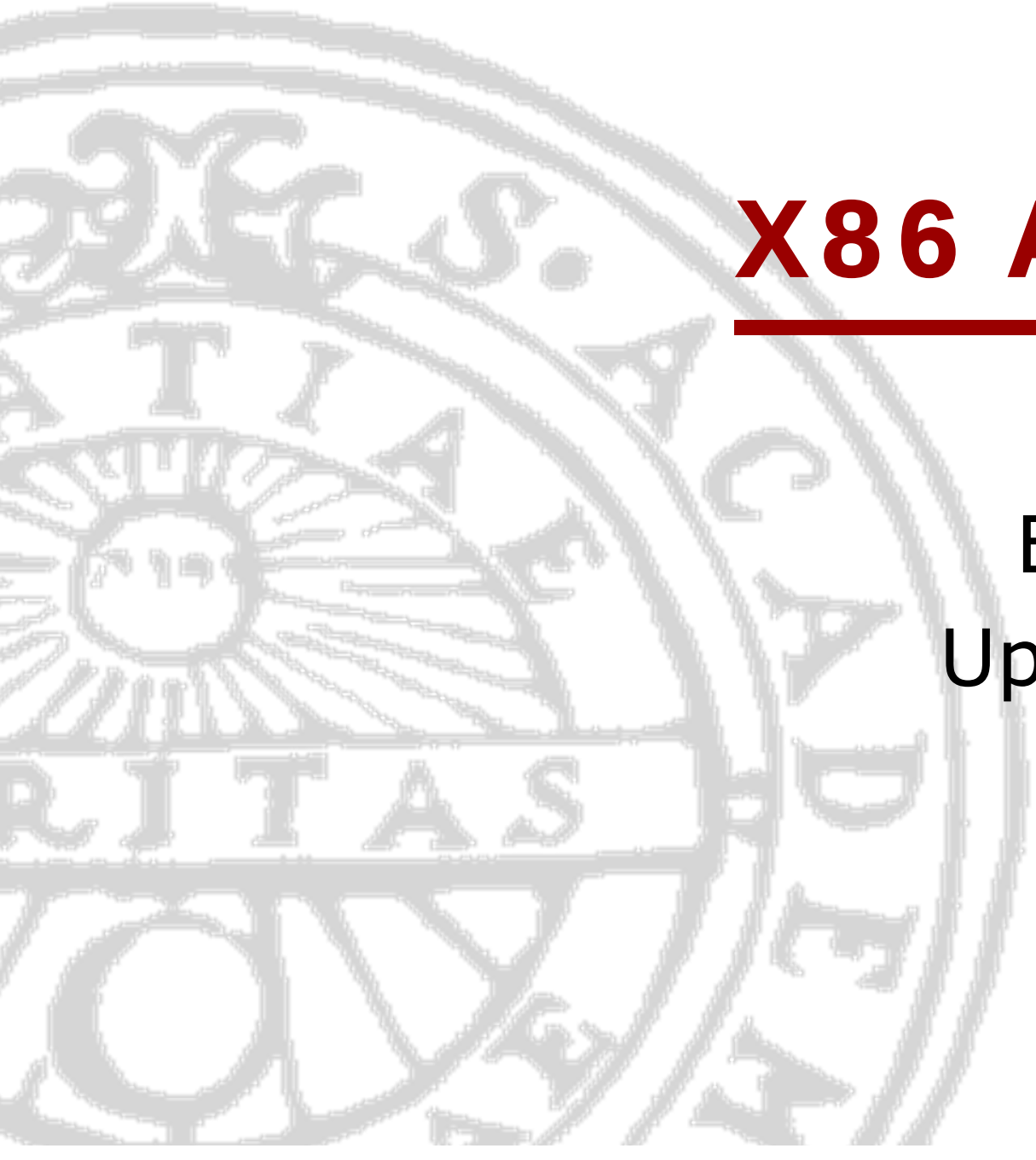
Revisiting Exceptions:

A pipeline implements precise interrupts iff:

All instructions before the faulting instruction can complete

All instructions after (and including) the faulting instruction must not change the system state and must be restartable

ROB helps the implementation in OoO execution: Do not commit instructions after a trap/interrupt



X86 Architecture

Erik Hagersten
Uppsala University
Sweden

UPPSALA UNIVERSITET

x86 Archeology

- (8080: 1974, 6.0 kTransistors, 2MHz, 8bit)
- 8086: 1978, 29 kT, 5-10MHz, 16bit (PC!)
- (80186:1982 ? kT, 4-40MHz, integration!)
- 80286: 1982, 0.1MT, 6-25MHz, chipset (PC-AT)
- 80386: 1985, 0.3MT, 16-33MHz, 32 bits
- 80486: 1989, 1.2MT, 25-50MHz, I&D$, FPU
- Pentium: 1993, 3.1 MT, 66 MHz, superscalar
- Pentium Pro: 1997, 5.5 MT, 200 MHz, O-O-O, 3-way superscalar
- Intel Pentium4:2001, 42 MT, 1.5 GHz, Super-pipe, L2$ on-chip
- …

UPPSALA UNIVERSITET

8086 registers

- AX (Accumulator)
- BX (Base)
- CX (Count)
- DX (Data)
- SP (Stack ptr)
- BP (Base ptr)
- SI (Source index)
- DI (Destination index)
- CS (Code segment)
- DS (Data segment)
- SS (Stack segment)

”General purpose” registers

”Addressing registers”

Segmented addressing (extending the address range)

UPPSALA UNIVERSITET

Complex instructions of x86

- RISC (Reduced Instruction Set Computer)
  - LD/ST with a limited set of address modes
  - ALU instructions (a minimum)
  - Many general purpose registers
  - Simplifications (e.g., read R0 returns the value 0)
  - Simpler ISA ➔ more efficient implementations

- x86 CISC (Complex Instruction Set Computer)
  - ALU/Memory in the same instruction
  - Complicated instructions
  - Few specialized registers (actually accumulator architecture)
  - Variable instruction length
  - x86 was lagging in performance to RISC in the 90s

UPPSALA UNIVERSITET

x86 Micro-ops

- Newer x86 pipelines implements RISC-ish µ-ops.
- Some complex x86 instructions are expanded to several µ-ops at runtime.
- The translated µ-ops may be cached in a trace-cache [in their predicted order] (first: Pentium4)
- Expanded to "loop cache" in Core-2

UPPSALA UNIVERSITET

x86-64

- ISA extension to x86 (by AMD 2001)
- 64-bit virtual address space
- 64-bit GP registers x16

  x86's regs extended: rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi

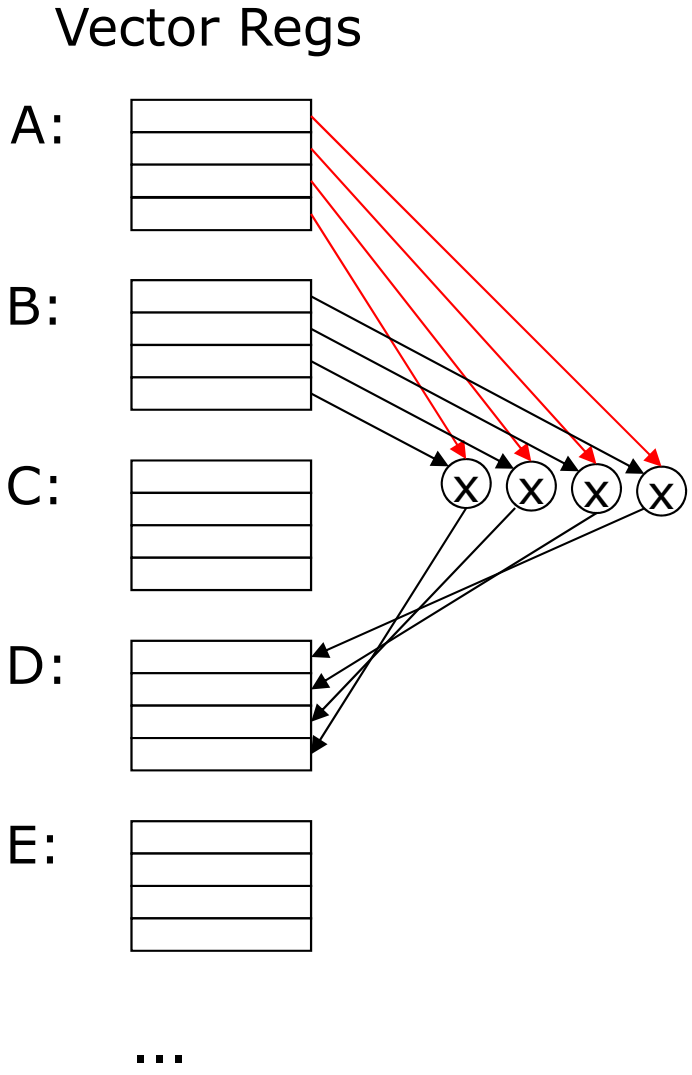
  x86-64 also has: r8, r9, ... r15 (i.e., a total of 16 regs)

  NOTE: dynamic register renaming makes the effective number of regs higher
- SSE*n*: 16 128-bit SSE "vector" registers
- Backwards compatible with x86

Intel adoptions: IA-32e, EM64T, Intel64

NOTE: IA-64 is Itanium

Examples of vector instructions




```
VEC_LD    A, (0)R1
VEC_LD    B, (0)R2
VEC_MUL   D, B, A
VEC_ST    D, (0)R3
```

UPPSALA UNIVERSITET

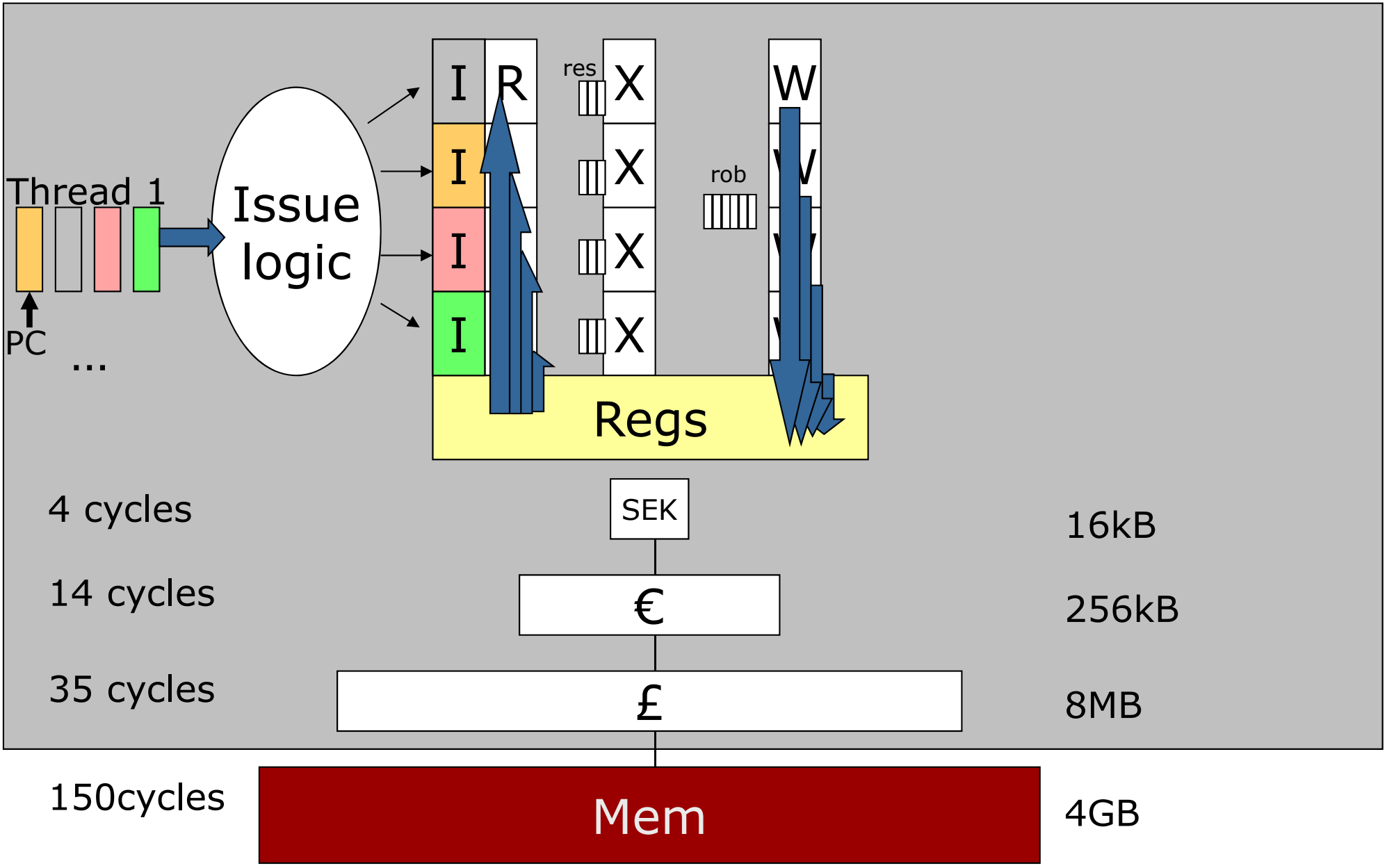
x86 Vector instructions

- MMX: 64 bit vectors (e.g., two 32bit ops)
- SSE*n*: 128 bit vectors(e.g., four 32 bit ops)
- AVX: 256 bit vectors(e.g., eight 32 bit ops)
  (in Sandy Bridge, ~Q1 2011)
- MIC/Xeon Phi: ”512-bit vectors” (e.g. 16x 32 bit ops)
  (introduced Q4 2012)

VLIW vs. Superscalar

Erik Hagersten
Uppsala University
Sweden

Superscalars



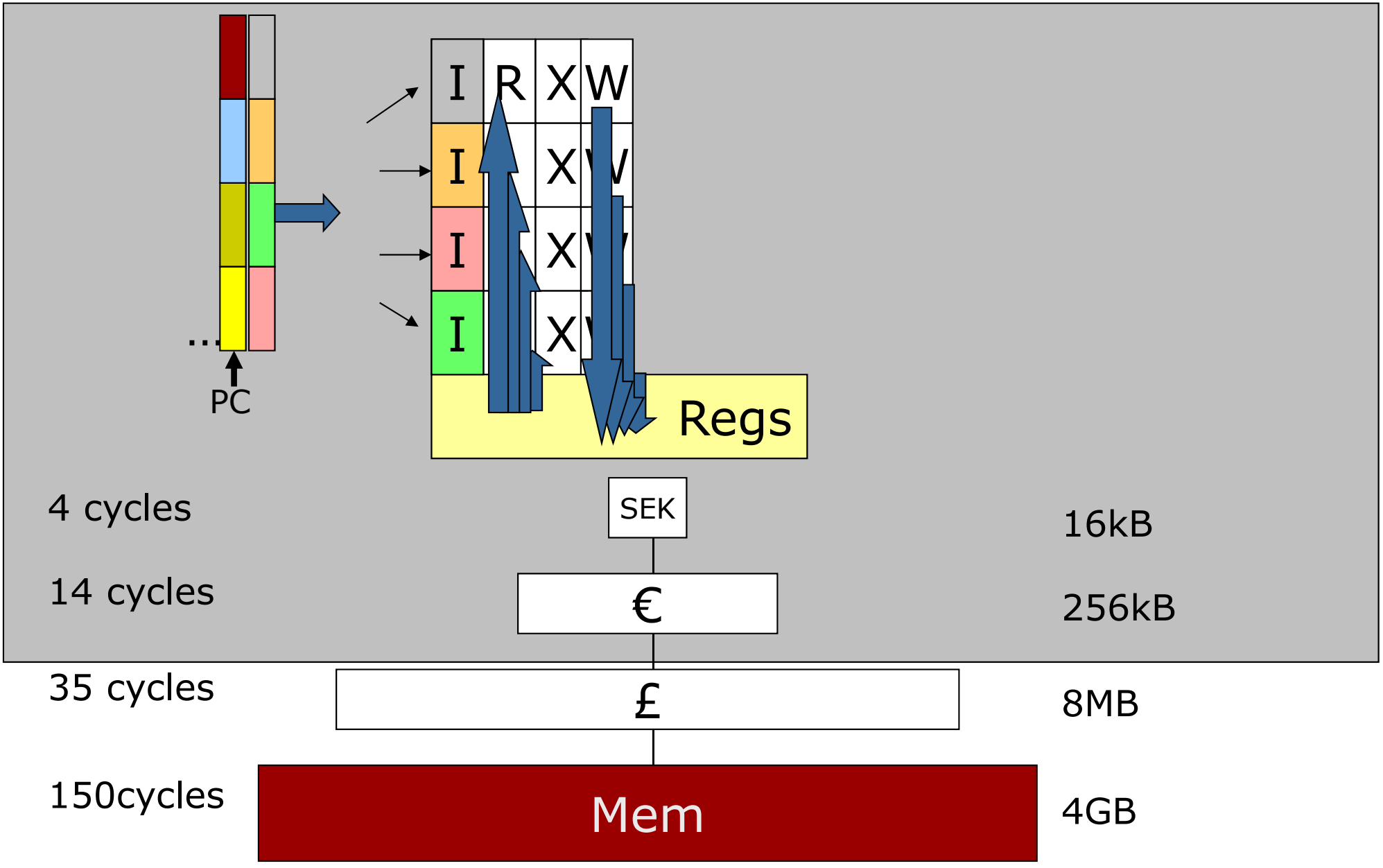
UPPSALA UNIVERSITET

Limits to superscalar execution

- Difficulties in scheduling within the constraints on number of functional units and the ILP in the code chunk
- Instruction decode complexity increases with the number of issued instructions
- Data and control dependencies are in general more costly in a superscalar processor than in a single-issue processor

VLIW: Simple ”superscalar” relying on compiler, instead of HW complexity, to find ILP

VLIW: Very Long Instruction Word



UPPSALA UNIVERSITET

Very Long Instruction Word (VLIW)

Compiler is responsible for instruction scheduling

Mem ref 1	*Mem ref 2*	*FP op 1*	*FP op 2*	*Int op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)	NOP	NOP	NOP	1
LD F10,-16(R1)	LD F14,-24(R1)	NOP	NOP	NOP	2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2	NOP	3
LD F26,-48(R1)	NOP	ADDD F12,F10,F2	ADDD F16,F14,F2	NOP	4
NOP	NOP	ADDD F20,F18,F2	ADDD F24,F22,F2	NOP	5
SD 0(R1), F4	SD -8(R1), F8	ADDD F28,F26,F2	NOP	NOP	6
SD -16(R1), F12	SD -24(R1), F8	NOP	NOP	NOP	7
SD -32(R1),F20	SD -40(R1),F24	NOP	NOP	SUBI R1,R1,#48	8
SD 0(R1),F28	NOP	NOP	NOP	BNEZ R1,LOOP	9

(7 loops in 9 cycles)

UPPSALA UNIVERSITET

Limits to VLIW

Difficult to find enough parallelism (ILP)

- *N* functional units and *K* ”dependent” pipeline stages implies *N*x *K* independent instructions to avoid stalls

No dynamic optimization (branch prediction & OoO)

How much ILP can a compiler find?

No binary code compatibility

Code size

But, …. simpler hardware

- short schedule
- high frequency

UPPSALA UNIVERSITET

HW support for static speculation

- Move LD up and ST down. But, how far?
  - Normally not outside of the basic block!
  - Often stopped by disambiguation problems
- Basic blocks are too small
  - 6 parallel pipelines can not help if BB is 4 instructions

HW support for [static] speculation and improved ILP

Erik Hagersten
Uppsala University
Sweden

UPPSALA UNIVERSITET

HW support for static speculation

- Move LD up and ST down. But, how far?
  - Normally not outside of the basic block!
  - Often stopped by disambiguationLD/ST problems
- Basic blocks are too small

➔ Need techniques to allow larger moves and increase the effective size of a basic block

  - Move LD above ST: disambiguation detection
  - Move LD above branch: avoid false exceptions
  - Removing branches: predicate execution

Moving LD above a branch?

```
....
BRNZ R7, #200

...
LD R1, 100(R2)
```

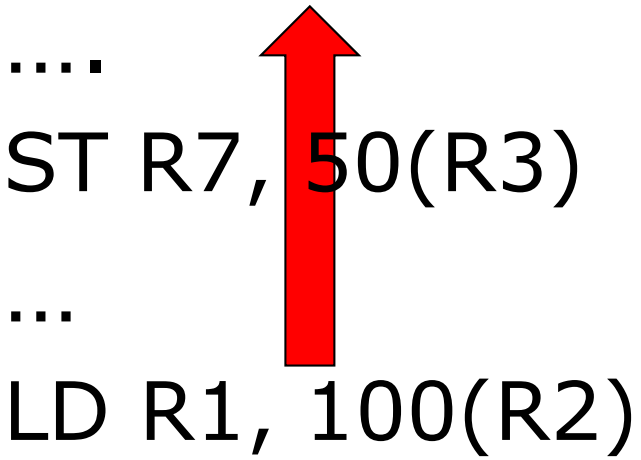
UPPSALA UNIVERSITET

Example: Moving LD above a branch

```
LD.s R1, 100(R2)      ;"Speculative  LD" to R1
....                  ;set "poison bit" in R1 if exception
BRNZ R7, #200
...
LD.chk R1             ;Get exception if poison bit of R1 is set
```

Good performance if the branch is not taken

Moving LD above a ST ?



UPPSALA UNIVERSITET

Moving LD above a ST

```
LD.a R1, 100(R2)      ; "advanced LD"
                      ; create entry in the ALAT <addr,reg>

....
ST  R7, 50(R3)        ;invalidate entry if ALAT addr match

...
LD.c R1               ; Redo LD if entry in ALAT invalid
                      ; remove entry in ALAT
```

ALAT (advanced load address table) is an associative data structure storing tuples of: <addr, dest-reg>

UPPSALA UNIVERSITET

Conditional execution

- Make basic blocks larger and avoid branch overhead
- Conditional Instructions
  - Conditional register move

```
CMOVZ R1, R2, R3          ;move R2 to R1 if (R3 == 0)
```

  - Compare-and-swap (atomics memory operations later)

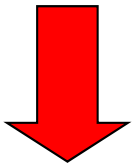
```
CAS R1, R2, R3            ;swap R2 and mem(R1) if (mem(R1)== R3)
```

- Predicate execution
  - A more generalized technique
  - Each instruction executed if the associated 1-bit predicate REG is 1.

UPPSALA UNIVERSITET

Predicate example

```
IF R1 > R2 then
        LD R7, 100(R1)
        ADD R1, R1, #1
else
        LD R7, 100(R2)
        ADD R2, R2, #1
end
```



Standard Technique

```
        CGT R3,R1,R2
        BRNZ R3, else
        LD R7, 100(R1)
        ADD R1, R1, #1
        BR end
else:   LD R7, 100(R2)
        ADD R2, R2, #1
end:
```

5 instr executed in ”then path”
2 branches

AVDARK 2013

UPPSALA UNIVERSITET

Predicate example

```
IF R1 > R2 then
        LD R7, 100(R1)
        ADD R1, R1, #1
else
        LD R7, 100(R2)
        ADD R2, R2, #1
end
```

Using Predicates

```
…
{IF R1 > R2 then P6=1;P7=0
            else P6=0;P7=1} ; //one instr!
P6: LD R7, 100(R1)
P6: ADD R1, R1, #1
P7: LD R7, 100(R2)
P7: ADD R2, R2, #1
```

One instruction sets the two predicate Regs
Each instr. in the ”then” guarded by P6
Each instr. in the ”else” guarded by P7
➔ One basic block
➔ Fewer total instr
5 instr executed in ”then path”
0 branch

Standard Technique

```
        CGT R3,R1,R2
        BRNZ R3, else
        LD R7, 100(R1)
        ADD R1, R1, #1
        BR end
else:   LD R7, 100(R2)
        ADD R2, R2, #1
end:
```

5 instr executed in ”then path”
2 branches

AVDARK 2013

UPPSALA UNIVERSITET

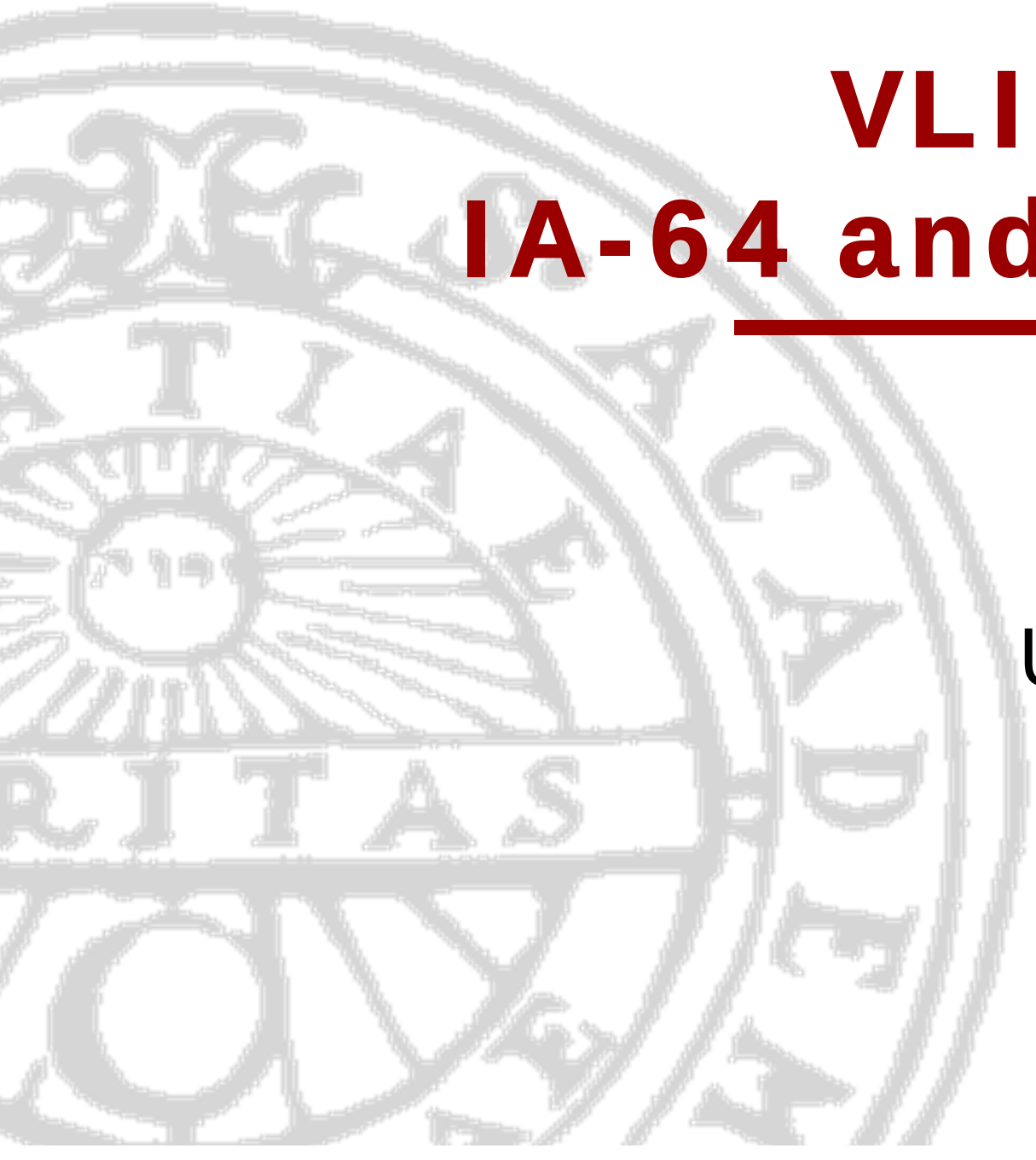
HW vs. SW speculation

Advantages:

- Dynamic runtime disambiguation of memory addresses
- Dynamic branch prediction is often better than static which limits the performance of SW speculation.
- HW speculation can maintain a precise exception model

Main disadvantage:

- Complex implementation and extensive need of hardware resources (conforms with technology trends)



VLIW Example: IA-64 and Itanium(I)

Erik Hagersten
Uppsala University
Sweden

UPPSALA UNIVERSITET

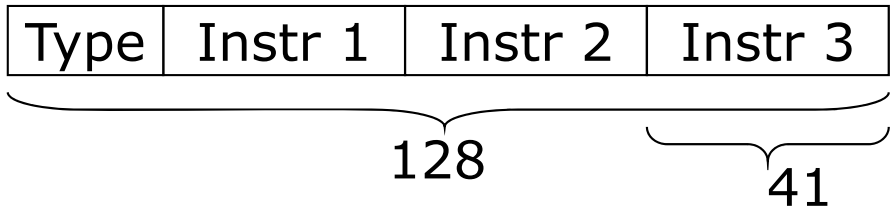
Little of everything

- VLIW
- Advanced loads supported by ALAT
- Load speculation supported by predication
- Dynamic branch prediction
- Multicore
- Multithreading
- ”All the tricks in the book”

AVDARK 2013

UPPSALA UNIVERSITET

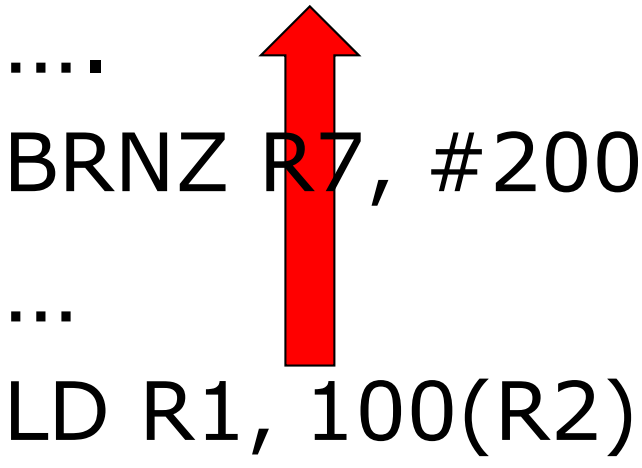
Itanium (I) specifics



- Instruction bundle (128 bits)
  - (5bits) template (identifies I types and dependencies)
  - 3 x (41bits) instruction
- Can issue up to two bundles per cycle (6 instr)
- The "Type" specifies if the instr. are independent
- Latencies:

Instruction	Latency
INT-LD	1
FP-LD	9
Predicted branch	0-3
Misspred branch	0-9
INT-ALU	0
FP-ALU	4

Moving LD above a branch?



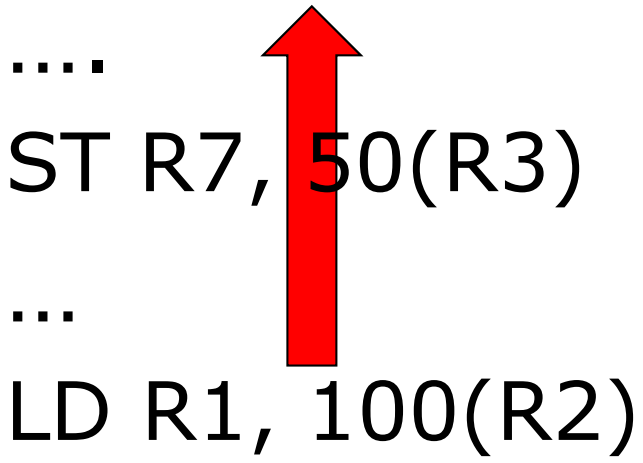
UPPSALA UNIVERSITET

Example: Moving LD above a branch

```
LD.s R1, 100(R2)      ;"Speculative LD" to R1
....                  ;set "poison bit" in R1 if exception
BRNZ R7, #200
...
LD.chk R1             ;Get exception if poison bit of R1 is set
```

Good performance if the branch is not taken

Moving LD above a ST ?



UPPSALA UNIVERSITET

Moving LD above a ST

```
LD.a R1, 100(R2)      ; "advanced LD"
                      ; create entry in the ALAT <addr,reg>

....
ST  R7, 50(R3)        ;invalidate entry if ALAT addr match

...
LD.c R1               ; Redo LD if entry in ALAT invalid
                      ; remove entry in ALAT
```

ALAT (advanced load address table) is an associative data structure storing tuples of: <addr, dest-reg>

UPPSALA UNIVERSITET

Conditional execution

- Make basic blocks larger and avoid branch overhead
- Conditional Instructions
  - Conditional register move

```
CMOVZ R1, R2, R3          ;move R2 to R1 if (R3 == 0)
```

  - Compare-and-swap (atomics memory operations later)

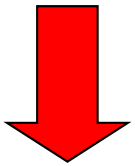
```
CAS R1, R2, R3            ;swap R2 and mem(R1) if (mem(R1)== R3)
```

- Predicate execution
  - A more generalized technique
  - Each instruction executed if the associated 1-bit predicate REG is 1.

UPPSALA UNIVERSITET

Predicate example

```
IF R1 > R2 then
        LD R7, 100(R1)
        ADD R1, R1, #1
else
        LD R7, 100(R2)
        ADD R2, R2, #1
end
```



Standard Technique

```
        CGT R3,R1,R2
        BRNZ R3, else
        LD R7, 100(R1)
        ADD R1, R1, #1
        BR end
else:   LD R7, 100(R2)
        ADD R2, R2, #1
end:
```

5 instr executed in ”then path”
2 branches

AVDARK 2013

Predicate example

```
IF R1 > R2 then
        LD R7, 100(R1)
        ADD R1, R1, #1
else
        LD R7, 100(R2)
        ADD R2, R2, #1
end
```

Using Predicates

```
…
{IF R1 > R2 then P6=1;P7=0
            else P6=0;P7=1} ; //one instr!
P6: LD R7, 100(R1)
P6: ADD R1, R1, #1
P7: LD R7, 100(R2)
P7: ADD R2, R2, #1
```

One instruction sets the two predicate Regs
Each instr. in the ”then” guarded by P6
Each instr. in the ”else” guarded by P7
➔ One basic block
➔ Fewer total instr
5 instr executed in ”then path”
0 branch

Standard Technique

```
        CGT R3,R1,R2
        BRNZ R3, else
        LD R7, 100(R1)
        ADD R1, R1, #1
        BR end
else:   LD R7, 100(R2)
        ADD R2, R2, #1
end:
```

5 instr executed in ”then path”
2 branches

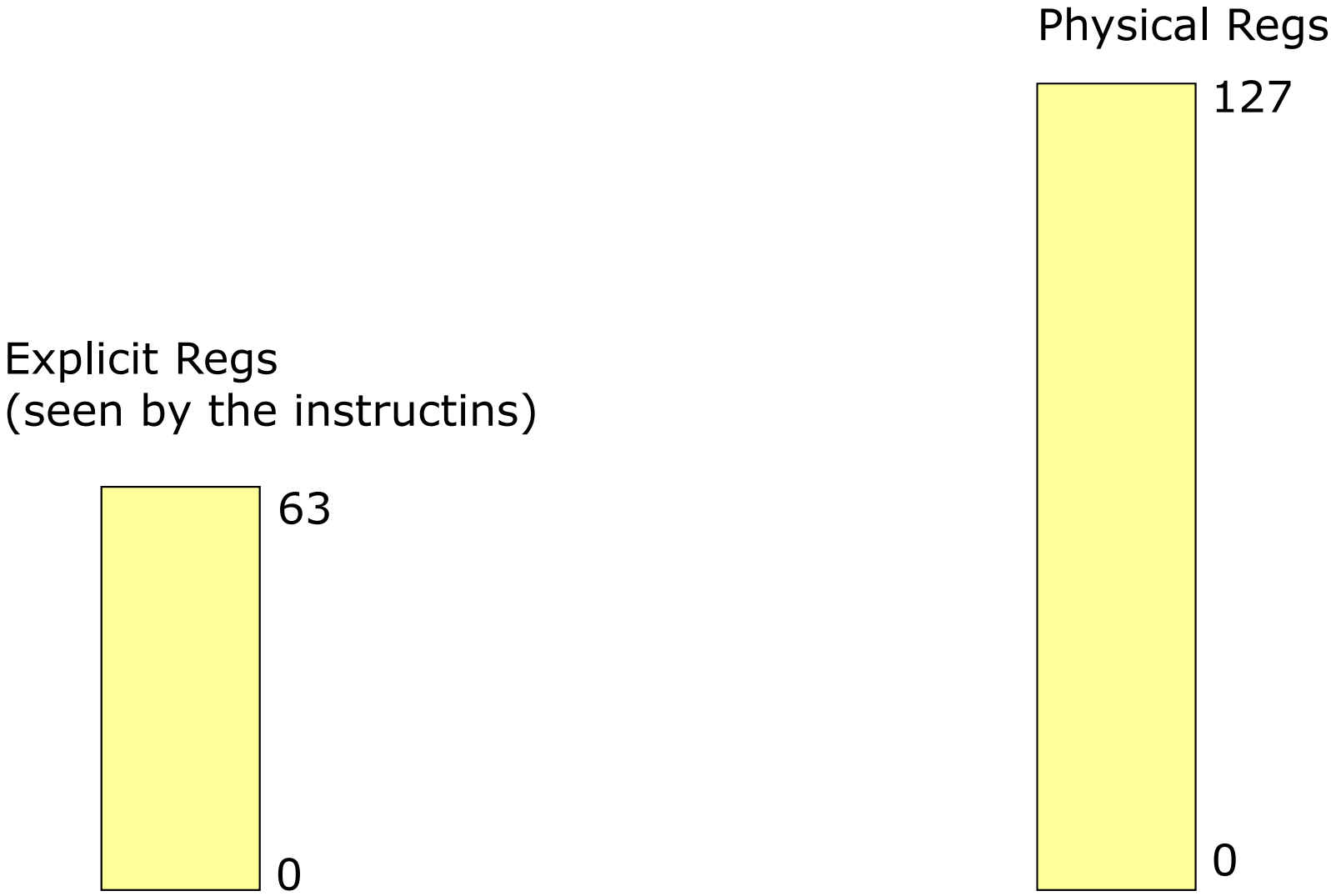
UPPSALA UNIVERSITET

Itanium Registers

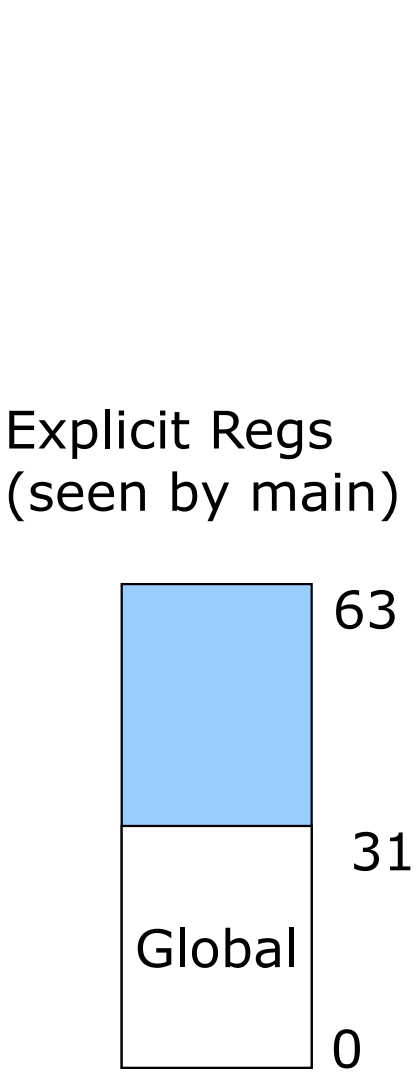
- 128 65-bit GPR (w/ poison bit)
- 128 82-bit FP REGS
- 64 1-bit predicate REGS
- A bunch of CSRs (control/status registers)

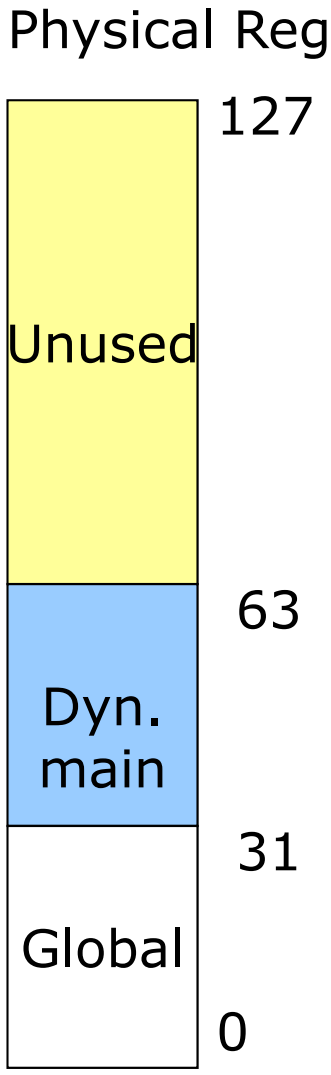
AVDARK 2013

Dynamic register window



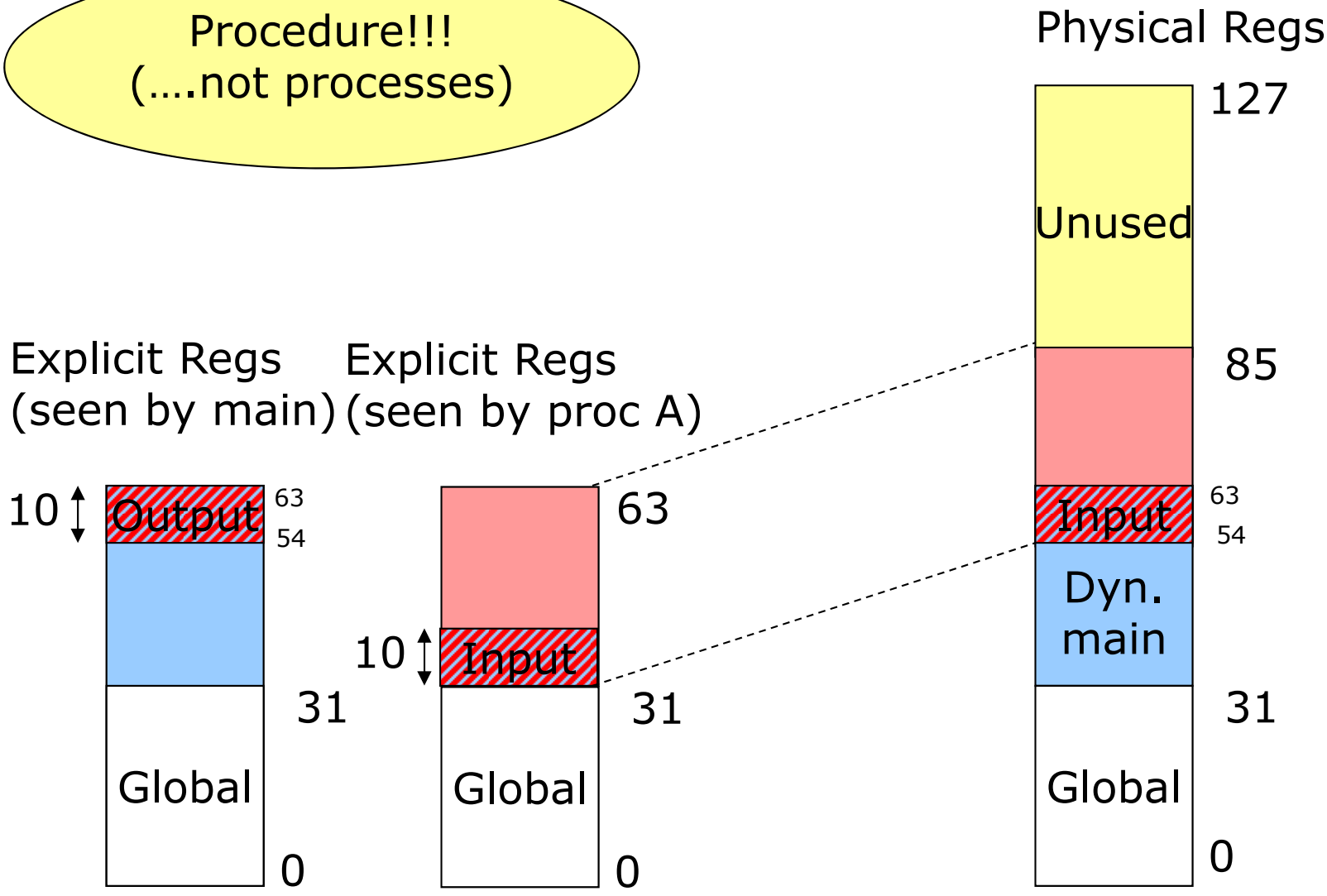
Dynamic register window for GPRs



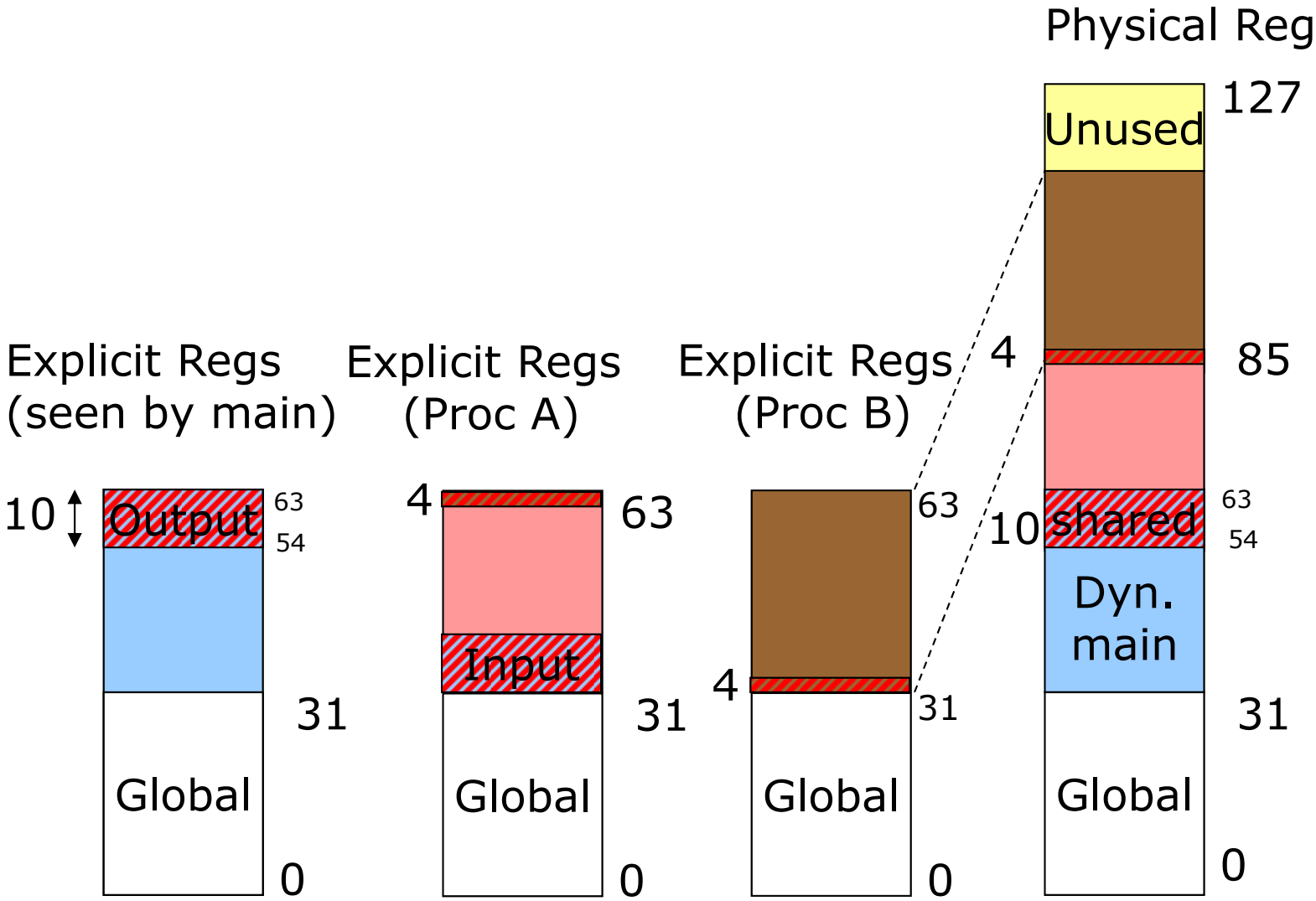


Calling Procedure A

Procedure!!!
(….not processes)



Calling Procedure B (automatic passing of parameters)



UPPSALA UNIVERSITET

Register Stack Engine (RSE)

- Saves and restores registers to memory on register spills
- Implemented in hardware
- Works in the background
- Gives the illusion of an unlimited register stack

- Register stacks also in SPARC and UCB RISC

UPPSALA UNIVERSITET

More recent Itanium (”Titanium”)

- 2012 “Poulson” is the latest Itanium
- 32 nm process technology
- 12-wide VLIW architecture
- Up to eight cores per chip
- “Multithreading enhancements”
- New instructions to take advantage of parallelism, especially in virtualization
- L3 cache: 32 MB (shared)
- L2 cache: 6 MB (shared)
- L1I: 512 kB
- L1D: 256 kB

UPPSALA UNIVERSITET

What is the alternative?

- VLIW was meant to simplify HW
- Itanium is a half-way solution.
- Will its ILP scale with technology?
- Other alternatives:
  - More "traditional" OoO cores
  - Many skinny IO cores (MIC)
  - Many power-lean cores (Atom)
  - Mixing core sizes (ARM: bigLITTLE)
  - Leverage graphics technology (GPUs)

UPPSALA UNIVERSITET

Transmeta

- Dave Ditzel (PhD: RISC)
- 256-bit VLIW running x86 applications
- First chip released y2000
- ”Code morphing” binary rewriting in software: x86 → VLIW
- Low power focus (and low performance)
- This is how much fun you can have with $1B

- Dave Ditzel is now with Intel
- Intel Labs are exploring similar ideas…