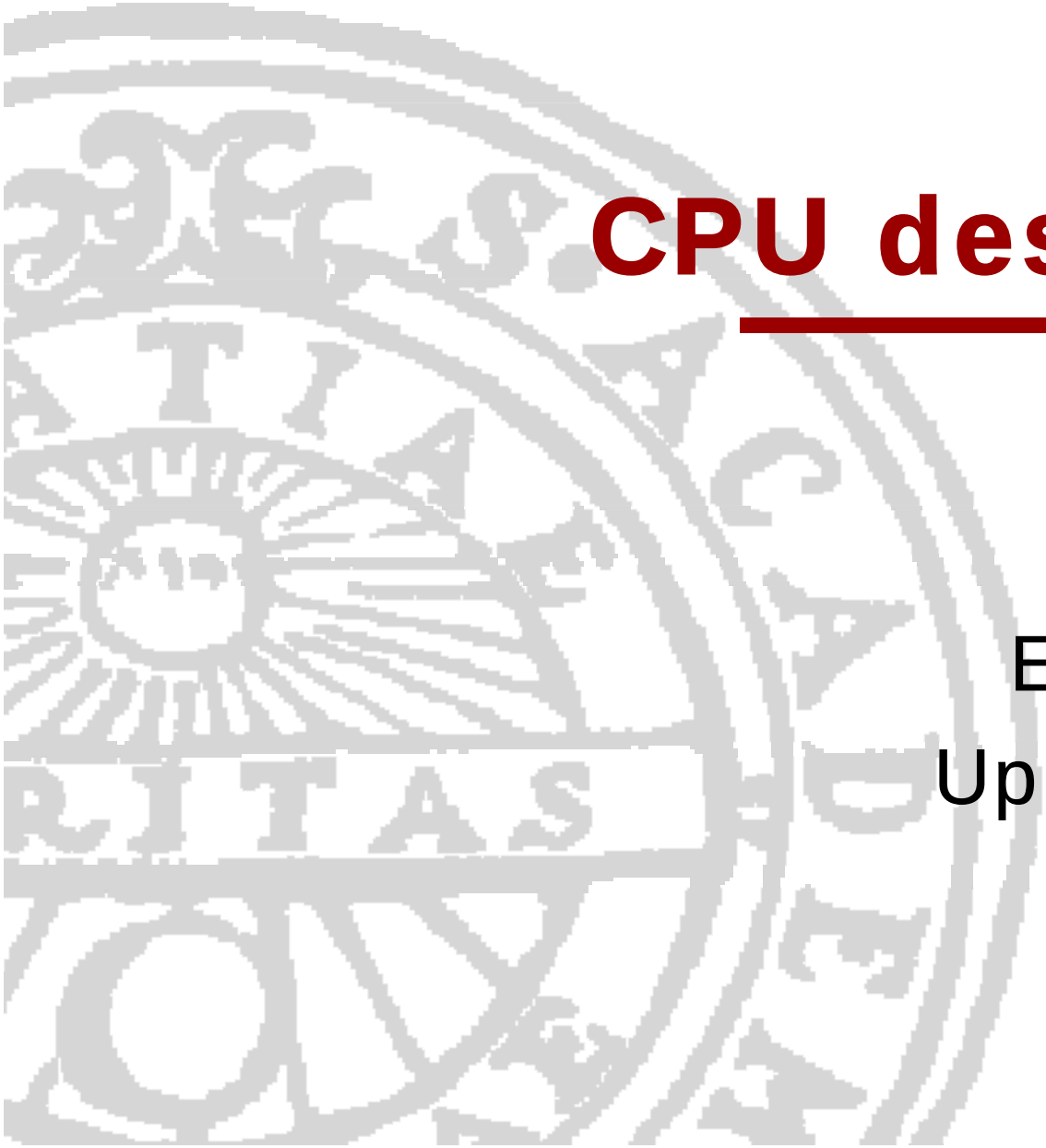


CPU design options

Erik Hagersten
Uppsala University

UPPSALA UNIVERSITET

DARK2 in a nutshell

1. Memory Systems (caches, VM, DRAM, microbenchmarks, ...)
2. Multiprocessors (TLP, coherence, interconnects, scalability, clusters, ...)
3. CPUs (pipelines, ILP, scheduling, Superscalars, VLIWs, embedded, ...)
4. Future: (physical limitations, TLP+ILP in the CPU,...)

UPPSALA UNIVERSITET

How it all started...the fossils

- ENIAC J.P. Eckert and J. Mauchly, Univ. of Pennsylvania, WW2
  - Electro Numeric Integrator And Calculator, 18.000 vacuum tubes
- EDVAC, J. V Neumann, operational 1952
  - Electric Discrete Variable Automatic Computer (stored programs)
- EDSAC, M.Wilkes, Cambridge University, 1949
  - Electric Delay Storage Automatic Calculator
- Mark-I... H. Aiken, Harvard, WW2, Electro-mechanic
- K. Zuse, Germany, electromech. computer, special purpose, WW2
- BARK, KTH, Gösta Neovius, Electro-mechanic early 50s
- BESK, KTH, Erik Stemme (now at Chalmers) early 50s
- SMIL, LTH mid 50s

UPPSALA UNIVERSITET

How do you tell a good idea from a bad

The Book: The performance-centric approach

- CPI = #execution-cycles / #instructions executed (~ISA goodness – lower is better)
- CPI * cycle time ➔ performance
- $CPI = CPI_{CPU} + CPI_{Mem}$

The book rarely covers other design tradeoffs

- The feature centric approach…
- The cost-centric approach…
- Energy-centric approach…
- Verification-centric approach…

DARK2 2008

UPPSALA UNIVERSITET

The Book: Quantitative methodology

Make design decisions based on execution statistics.
Select workloads (programs representative for usage)
Instruction mix measurements: statistics of relative usage of different components in an ISA

Experimental methodologies

- Profiling through tracing
- ISA simulators

UPPSALA UNIVERSITET

Two guiding stars -- the RISC approach:

Make the common case fast

- Simulate and profile anticipated execution
- Make cost-functions for features
- Optimize for overall end result (end performance)

Watch out for Amdahl's law

- $\text{Speedup} = \text{Execution\_time}_{\text{OLD}} / \text{Execution\_time}_{\text{NEW}}$
- $[(1-\text{Fraction}_{\text{ENHANCED}}) + \text{Fraction}_{\text{ENHANCED}} / / \text{Speedup}_{\text{ENHANCED}})]$

UPPSALA UNIVERSITET

Instruction Set Architecture (ISA)

-- the interface between software and hardware.

Tradeoffs between many options:

- functionality for OS and compiler
- wish for many addressing modes
- compact instruction representation
- format compatible with the memory system of choice
- desire to last for many generations
- bridging the semantic gap (old desire...)

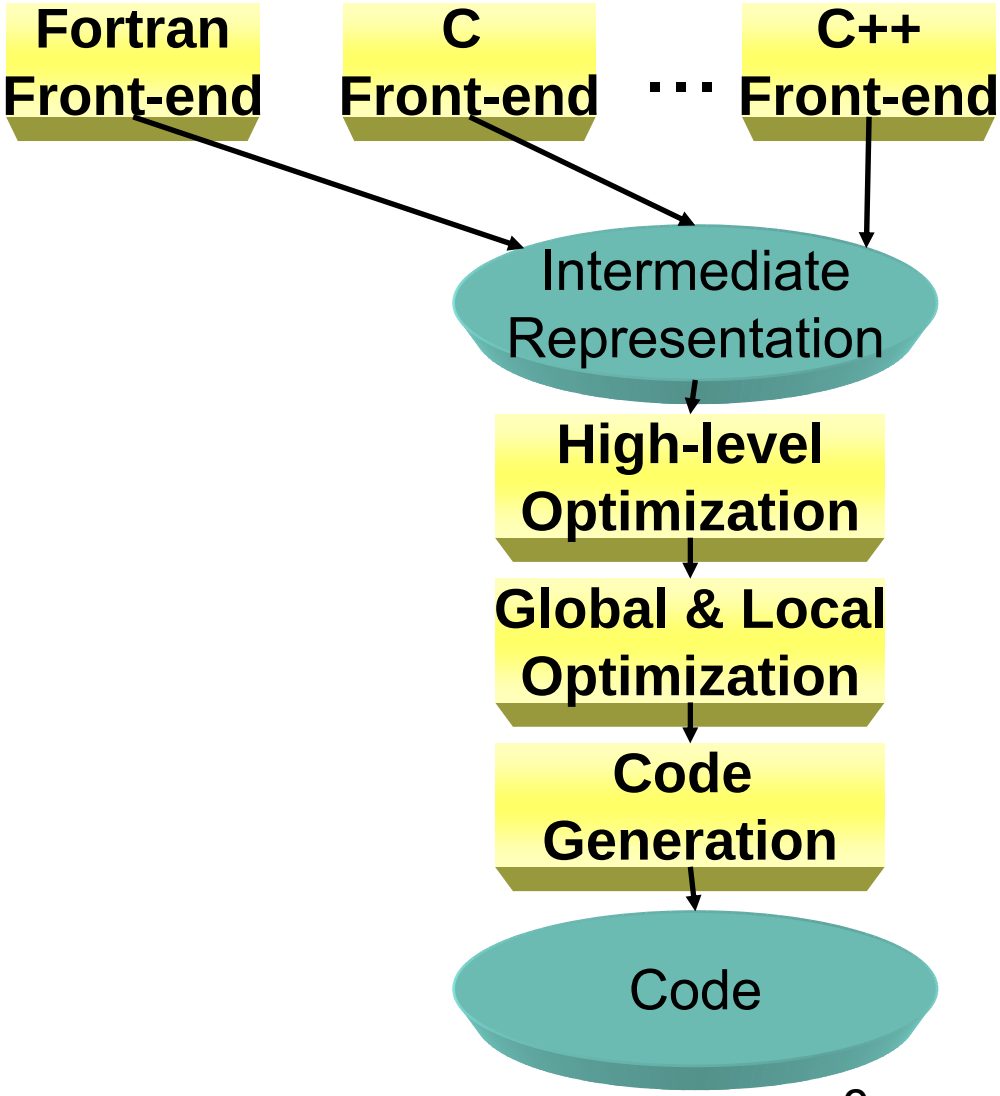
- RISC: the biggest “customer” is the compiler

UPPSALA UNIVERSITET

ISA trends today

- CPU families built around “Instruction Set Architectures” ISA
- Many incarnations of the same ISA
- ISAs lasting longer (~10 years)
- Consolidation in the market - fewer ISAs (not for embedded…)
- 15 years ago ISAs were driven by academia
- Today ISAs technically do not matter all that much (market-driven)
- How many of you will ever design an ISA?
- How many ISAs will be designed in Sweden?

Compiler Organization



Machine-independent Translation

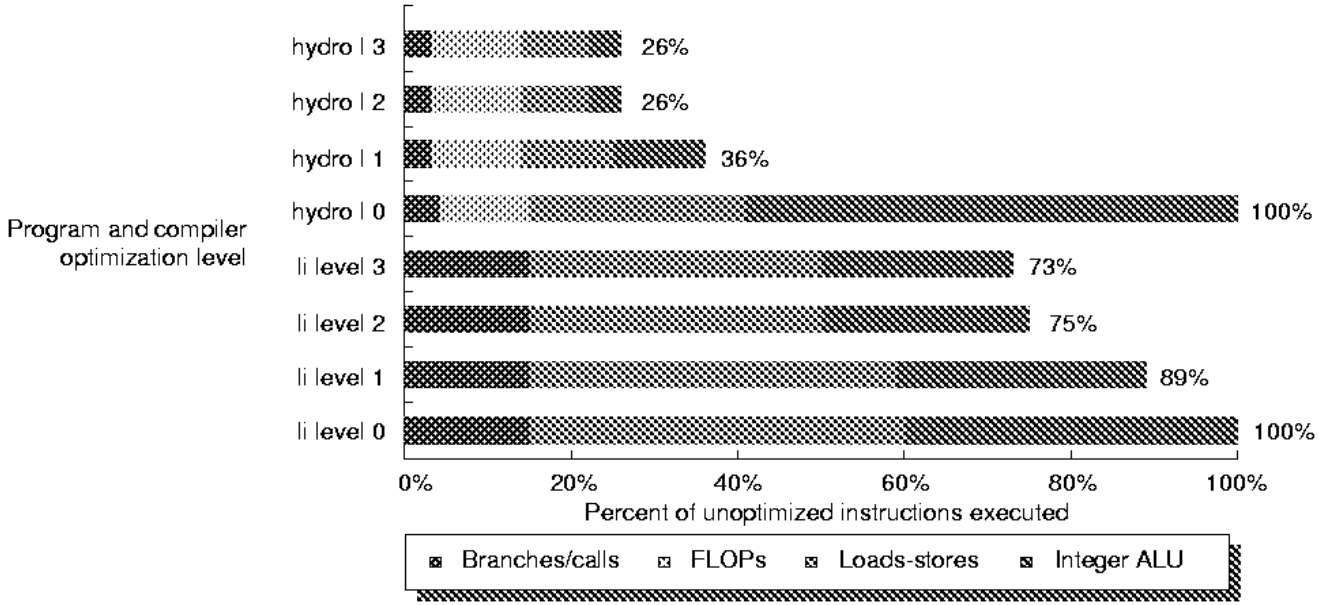
Procedure in-lining
Loop transformation

Register Allocation
Common sub-expressions

Instruction selection
constant folding

UPPSALA UNIVERSITET

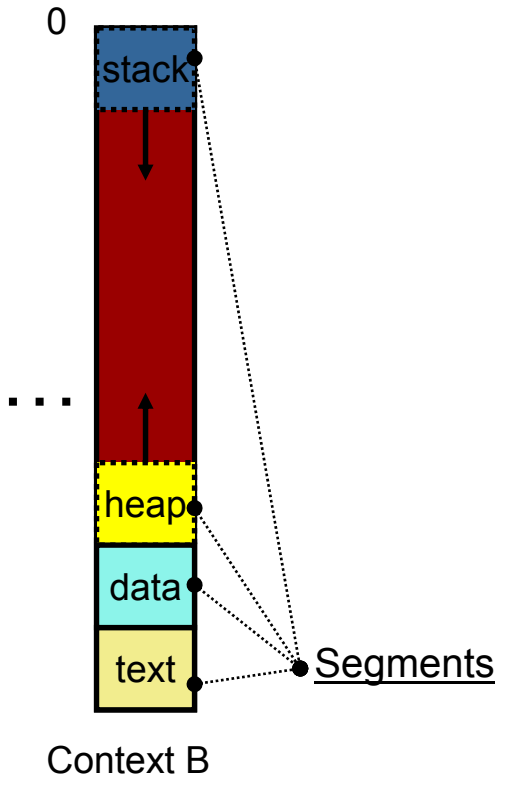
Compilers – a moving target!
The impact of compiler optimizations



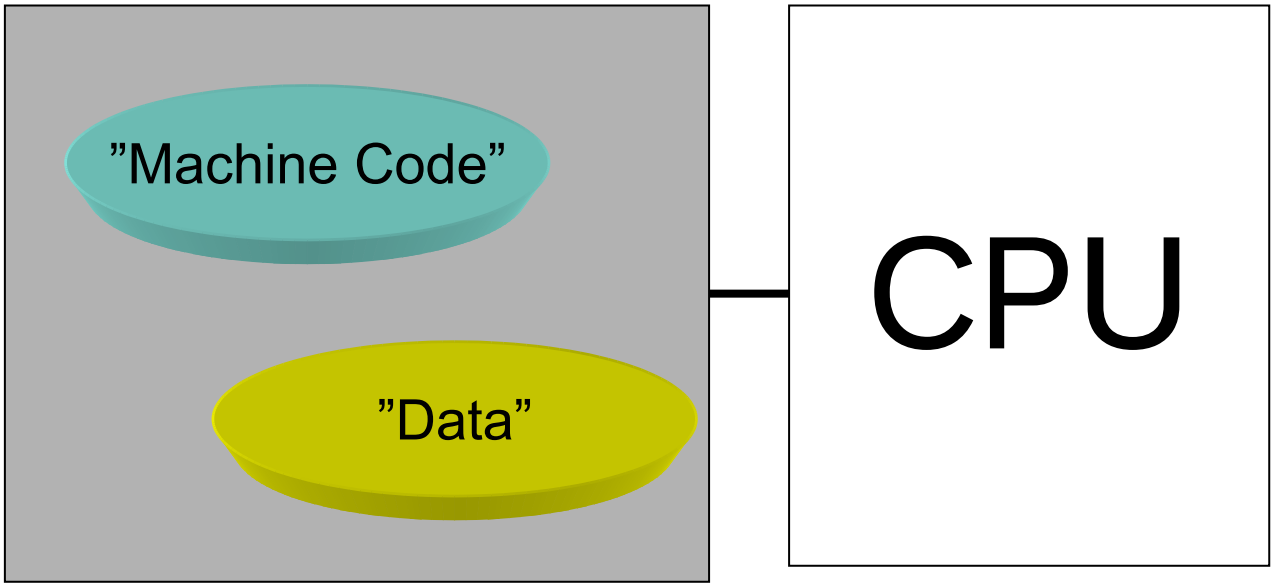
- Compiler optimizations affect the number of instructions as well as the distribution of executed instructions (the instruction mix)

Memory allocation model also has a huge impact

- **Stack**
  - local variables in activation record
  - addressing relative to stack pointer
  - stack pointer modified on call/return
- **Global data area**
  - large constants
  - global static structures
- **Heap**
  - dynamic objects
  - often accessed through pointers



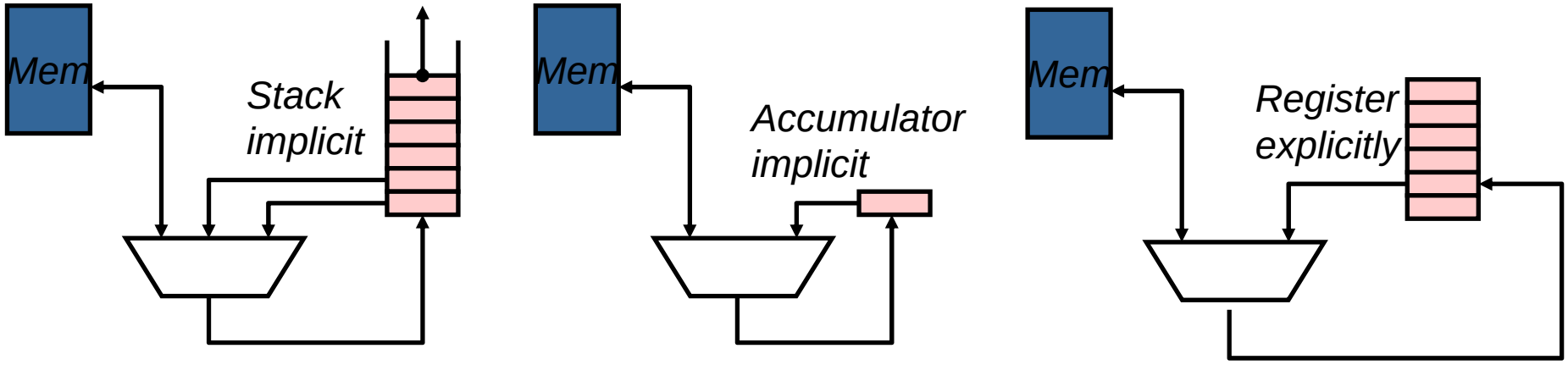
Execution in a CPU



Operand models

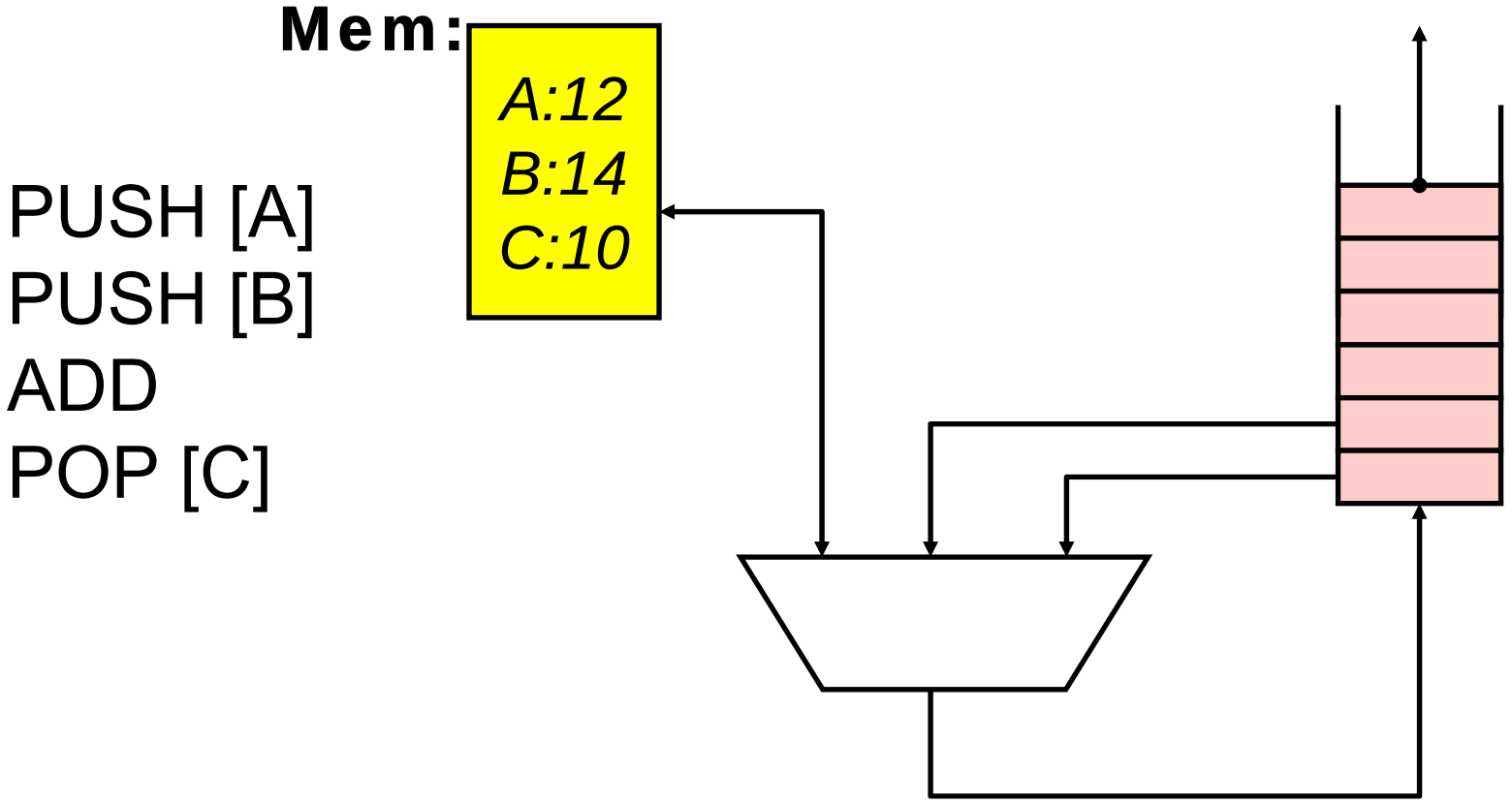
Example: C := A + B

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]		



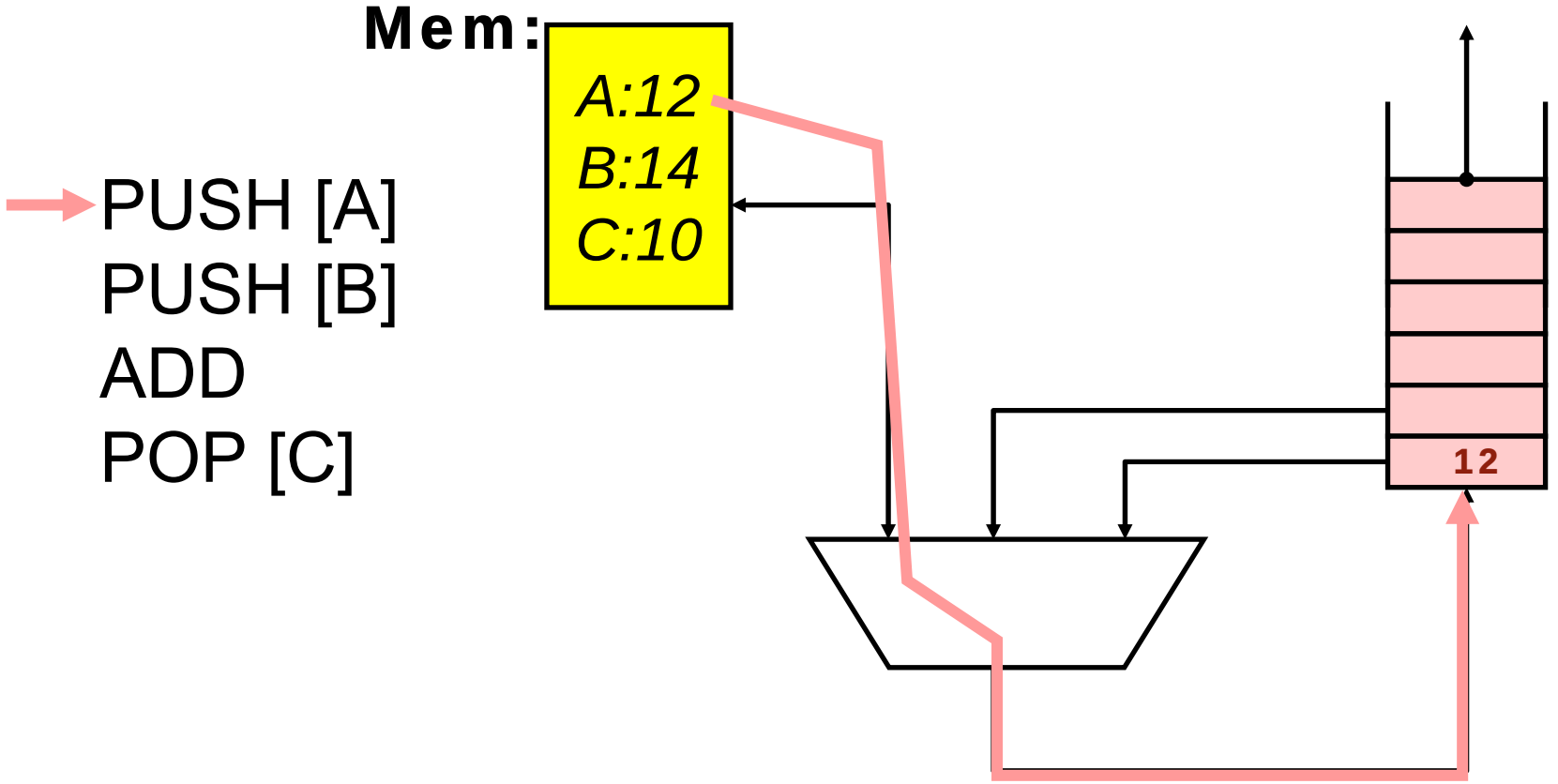
Stack-based machine

Example: C := A + B



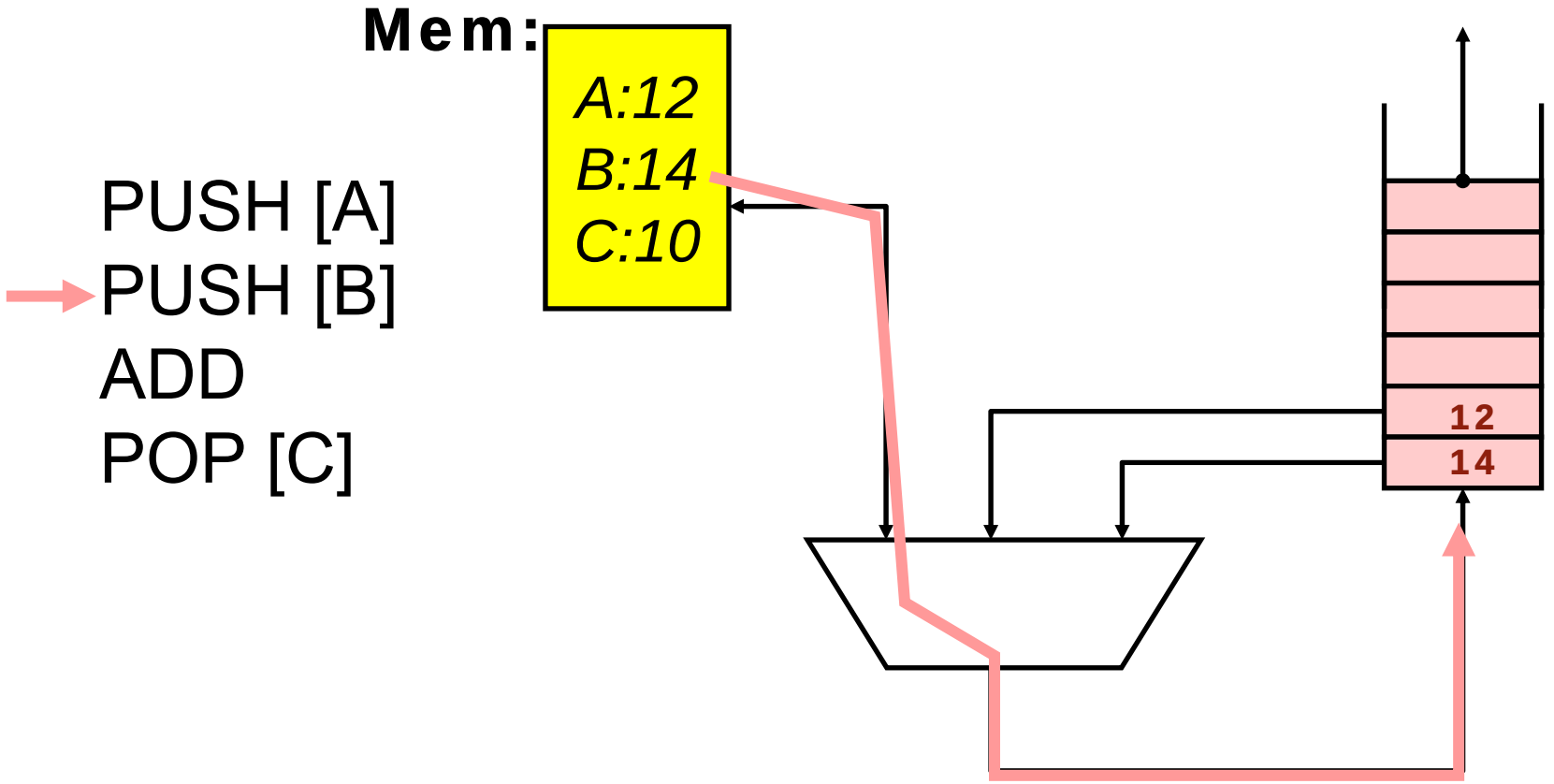
Stack-based machine

Example: C := A + B



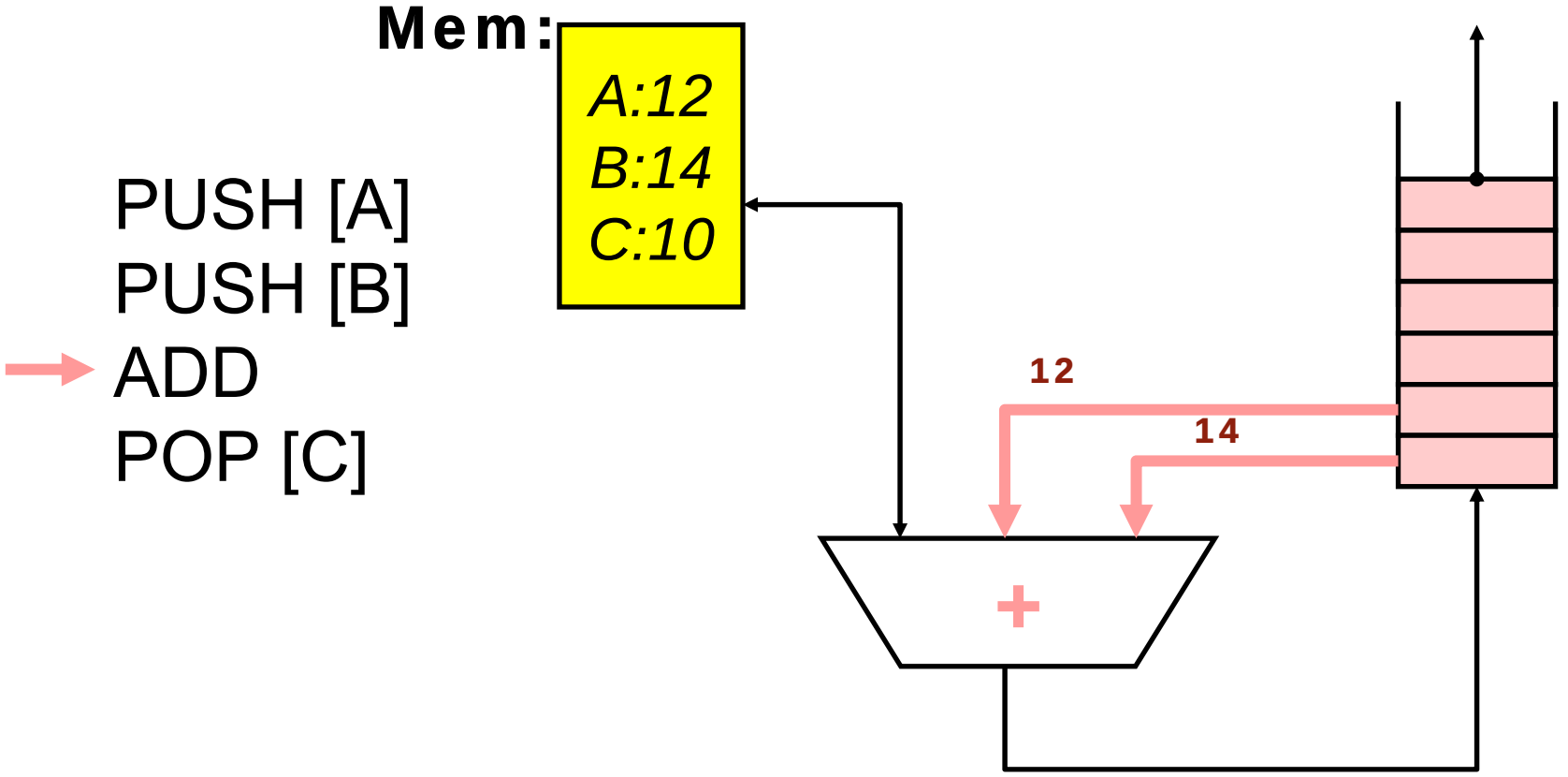
Stack-based machine

Example: C := A + B



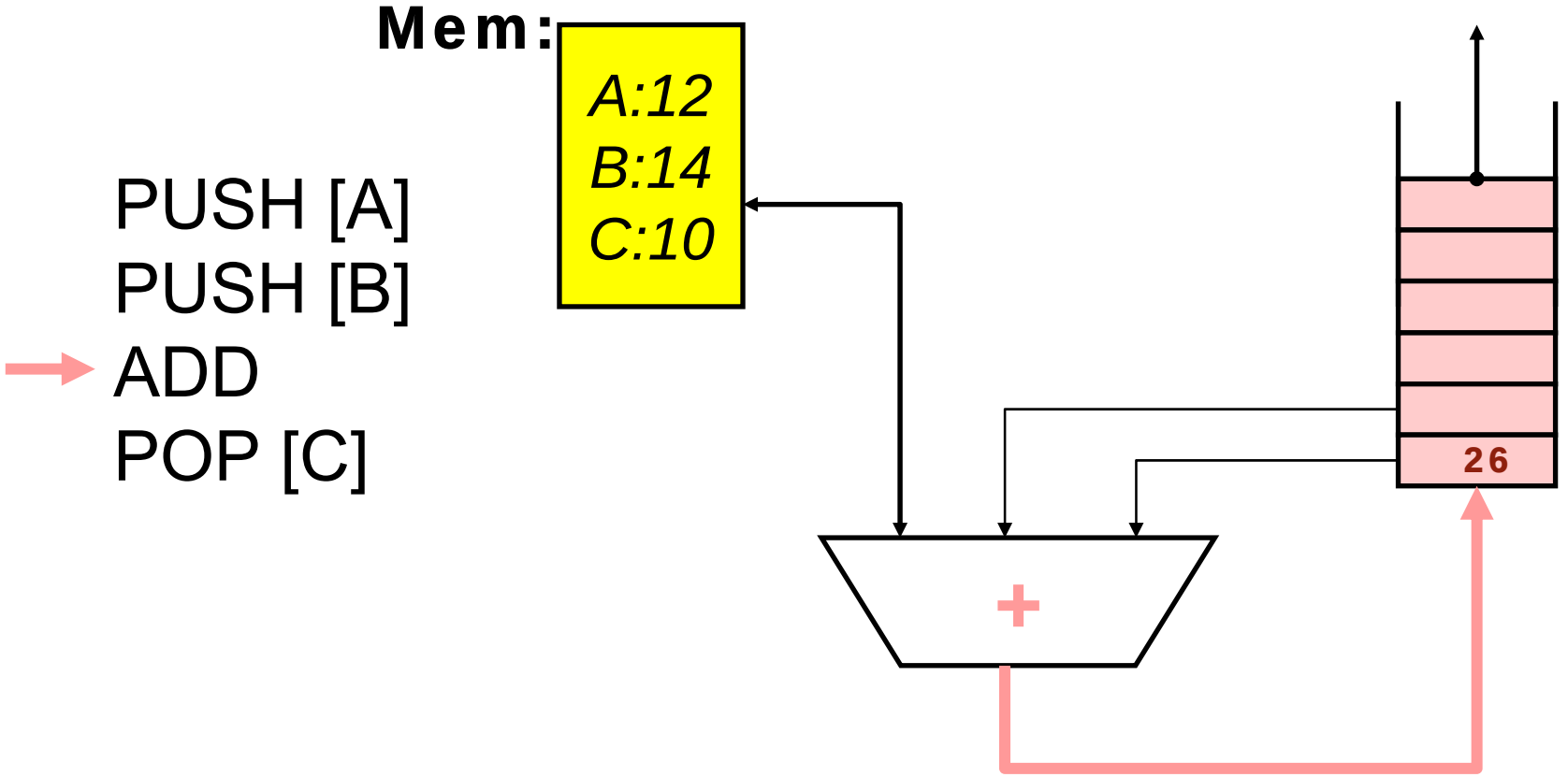
Stack-based machine

Example: C := A + B



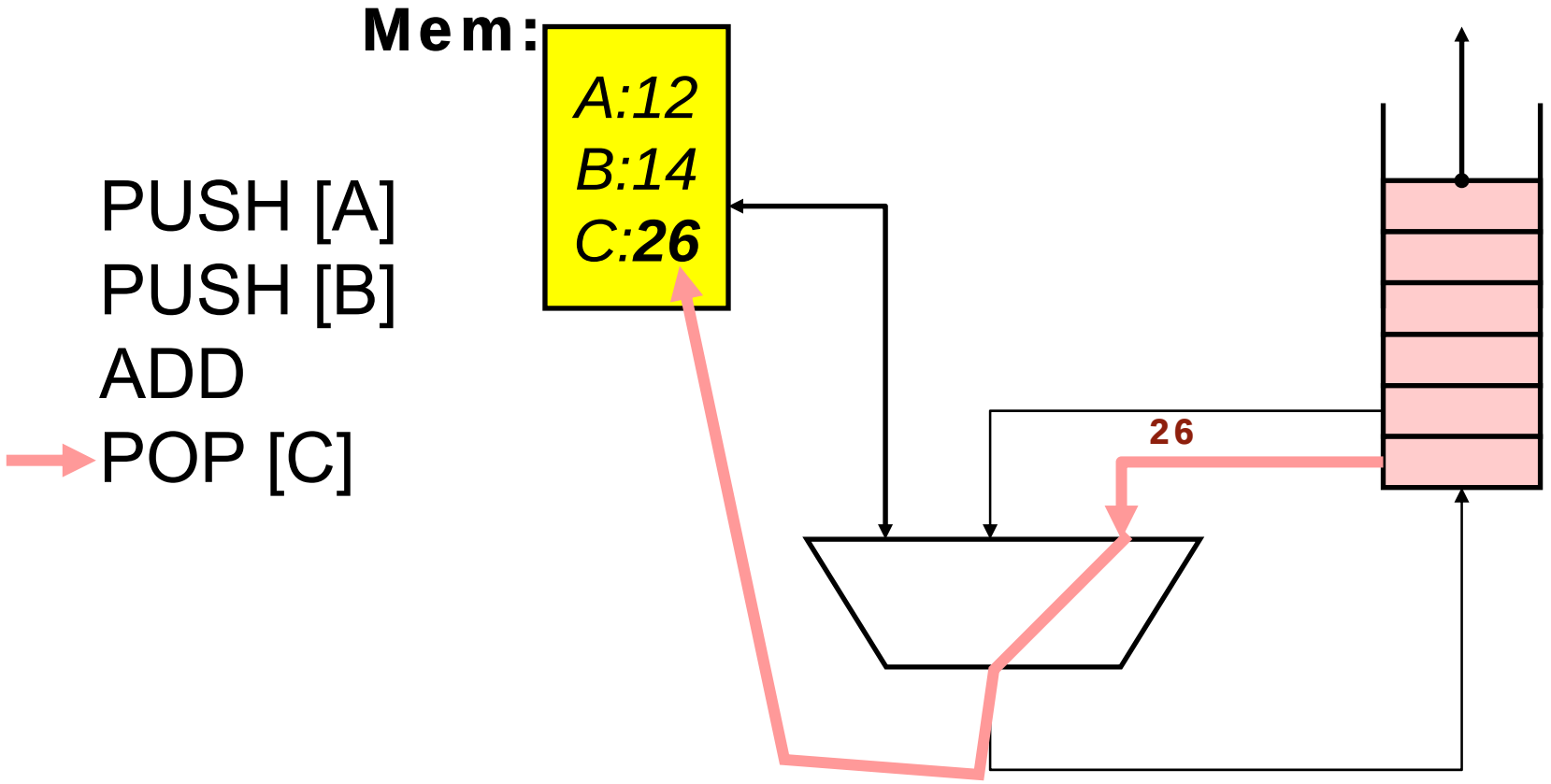
Stack-based machine

Example: C := A + B



Stack-based machine

Example: C := A + B



UPPSALA UNIVERSITET

Stack-based

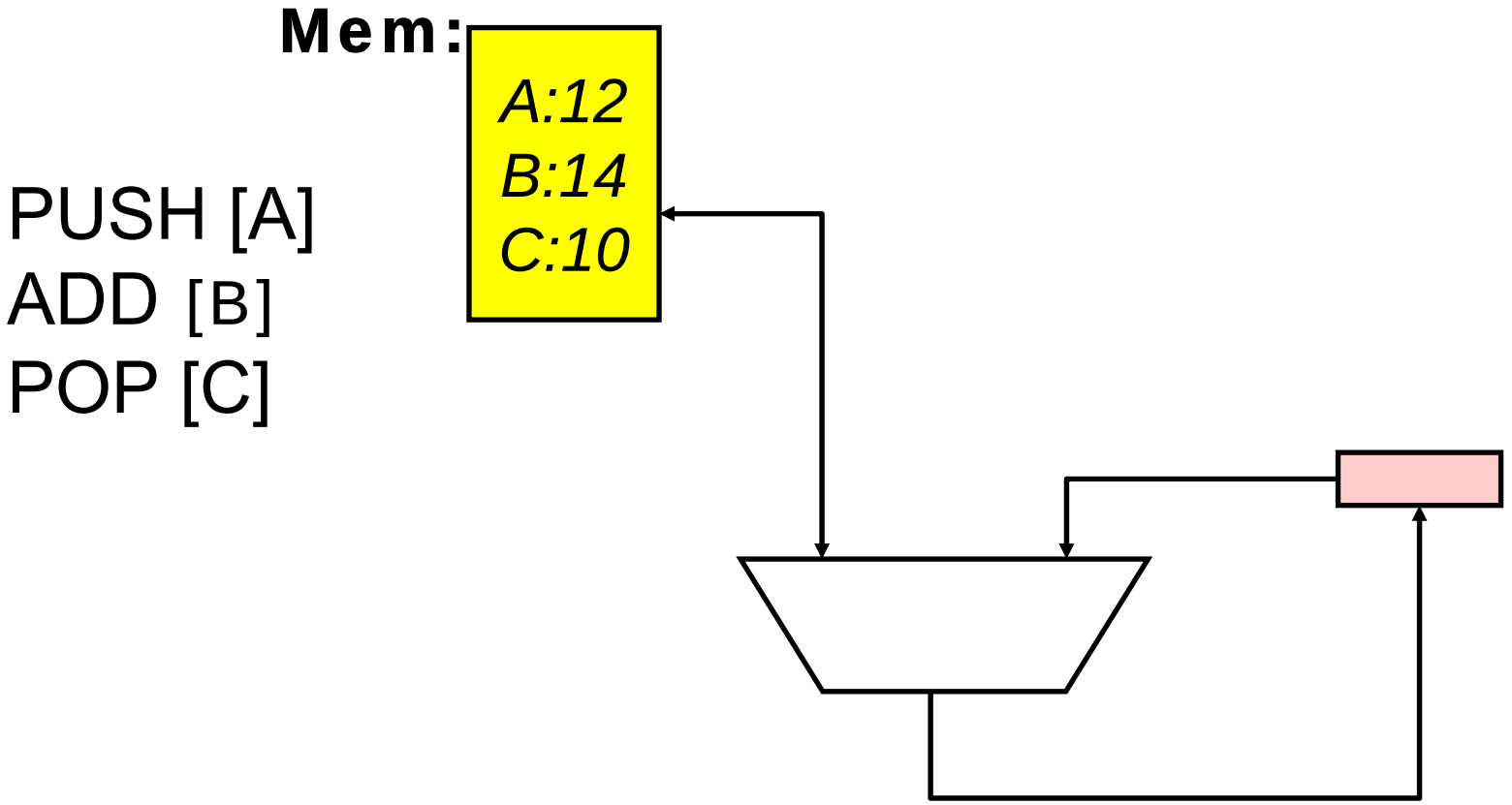
- Implicit operands
- Compact code format (1 instr. = 1byte)
- Simple to implement
- Not optimal for speed!!!

DARK2 2008

Accumulator-based

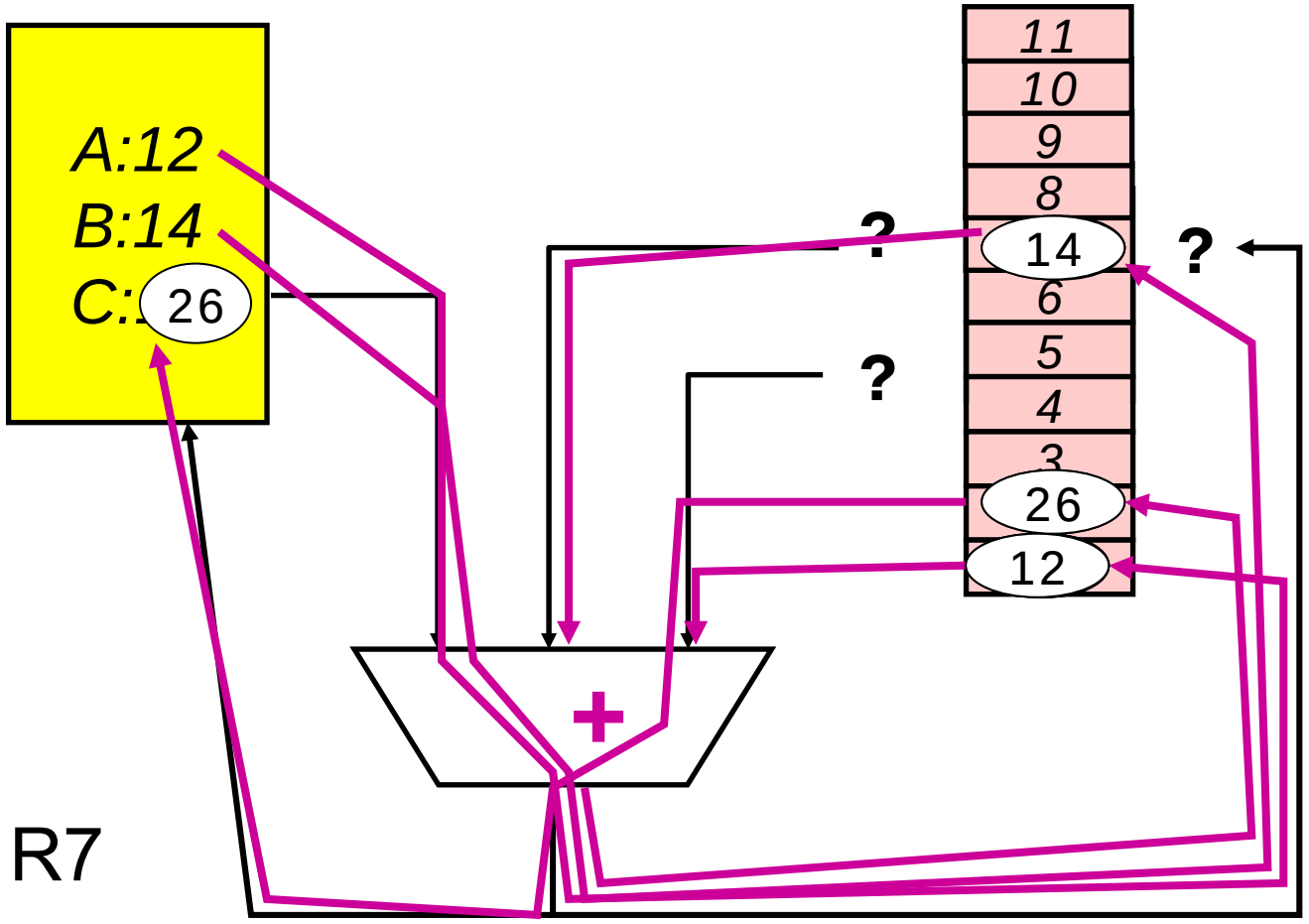
≈ Stack-based with a depth of one

One implicit operand from the accumulator



Register-based machine

Example: C := A + B



"Machine Code"

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

UPPSALA UNIVERSITET

Register-based

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

DARK2 2008

UPPSALA UNIVERSITET

Properties of operand models

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

General-purpose register model dominates today

Reason: general model for compilers and efficient implementation wise

Instruction formats

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

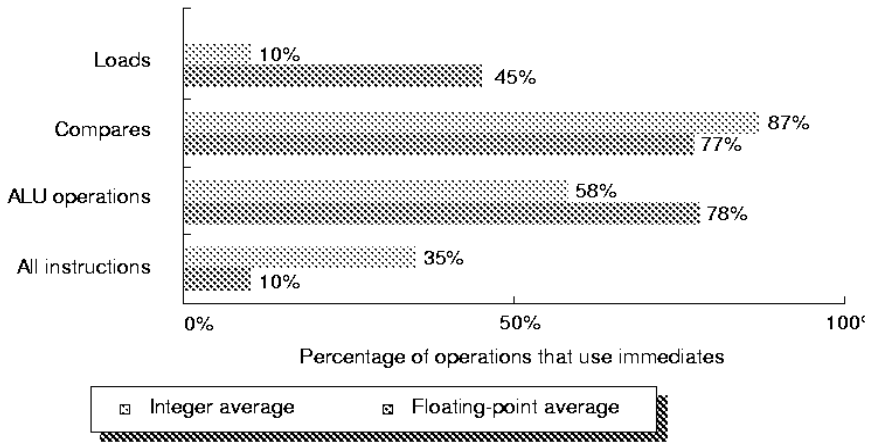
UPPSALA UNIVERSITET

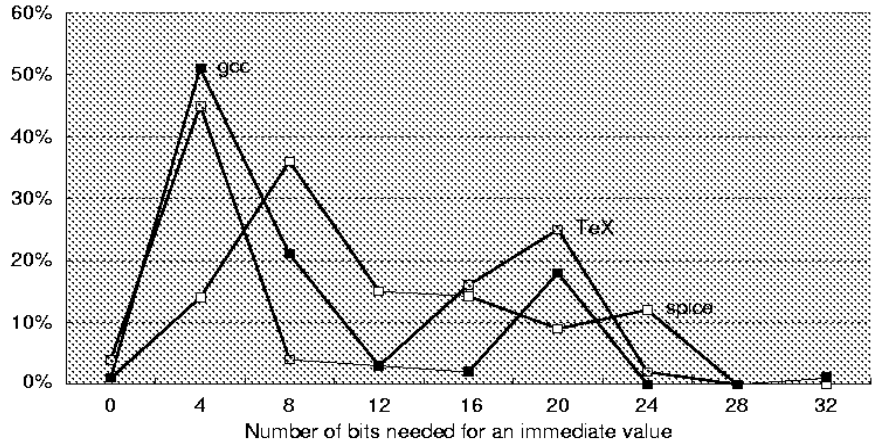
Important Operand Modes

Addressing mode	Example instruction	Meaning	When used
Immediate	Add R3, R4,#3	Regs[R3] ← Regs[R4]+ 3	For constants.
Displacement	Add R3, R4,100(R1)	Regs[R3] ← Regs[R4]+ Mem[100+Regs[R1]]	Accessing local variables.

UPPSALA UNIVERSITET

Size of immediates





- Immediate operands are very important for ALU and compare operations
- 16-bit immediates seem sufficient (75%-80%)

UPPSALA UNIVERSITET

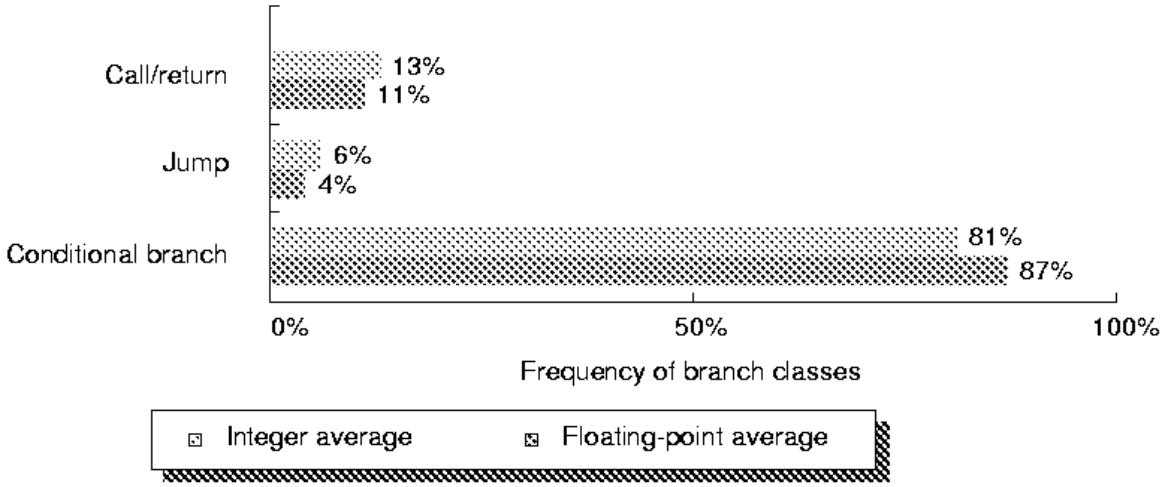
Operation types in the ISA

Operator type	*Examples*
Arithmetical and logical	Integer arithmetic and logical operations: add, and, subtract, or
Data transfer	Loads/stores (move instructions on machines with memory addressing)
Control	Branch, jump, procedure call and return
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply,...
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search

UPPSALA UNIVERSITET

Control instructions

- Conditional branches
- Unconditional branches (jumps)



Conditional branches dominate by far

Intuition: program loops are common!

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

Branch condition evaluation

Name	How?	Advantages	Disadvant-ages
Condition Code (CC)	Special bits are manipulated	CC set for free	Extra state
Condition register	Test general purpose register	simple	Uses up registers
Compare and branch	Compare is part of branch	One instr. Instead of two	Extra work per instr.

UPPSALA UNIVERSITET

Example: DLX- A generic architecture

Load/store architecture (32 bits)

- Many (32) general purpose integer registers (GPR) and single precision floating point registers (GPR0 = 0)
- Fixed instruction width and format
- Addressing modes: immediate and displacement
- Supported data types: bytes, half word (16 bits), word (32 bits), single and double precision IEEE floating points

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 ← KALLE + 4

UPPSALA UNIVERSITET

Generic Move Instructions

- Load and Store
  - LB, LBU, SB -- byte chunks
  - LH, LHU, SH -- half word chunks
  - LW, SW -- word chunks
  - LF, SF -- word chunks to floating point regs
  - LD, SD double precision to FP regs (2 regs per OP)

UPPSALA UNIVERSITET

Generic ALU Instructions

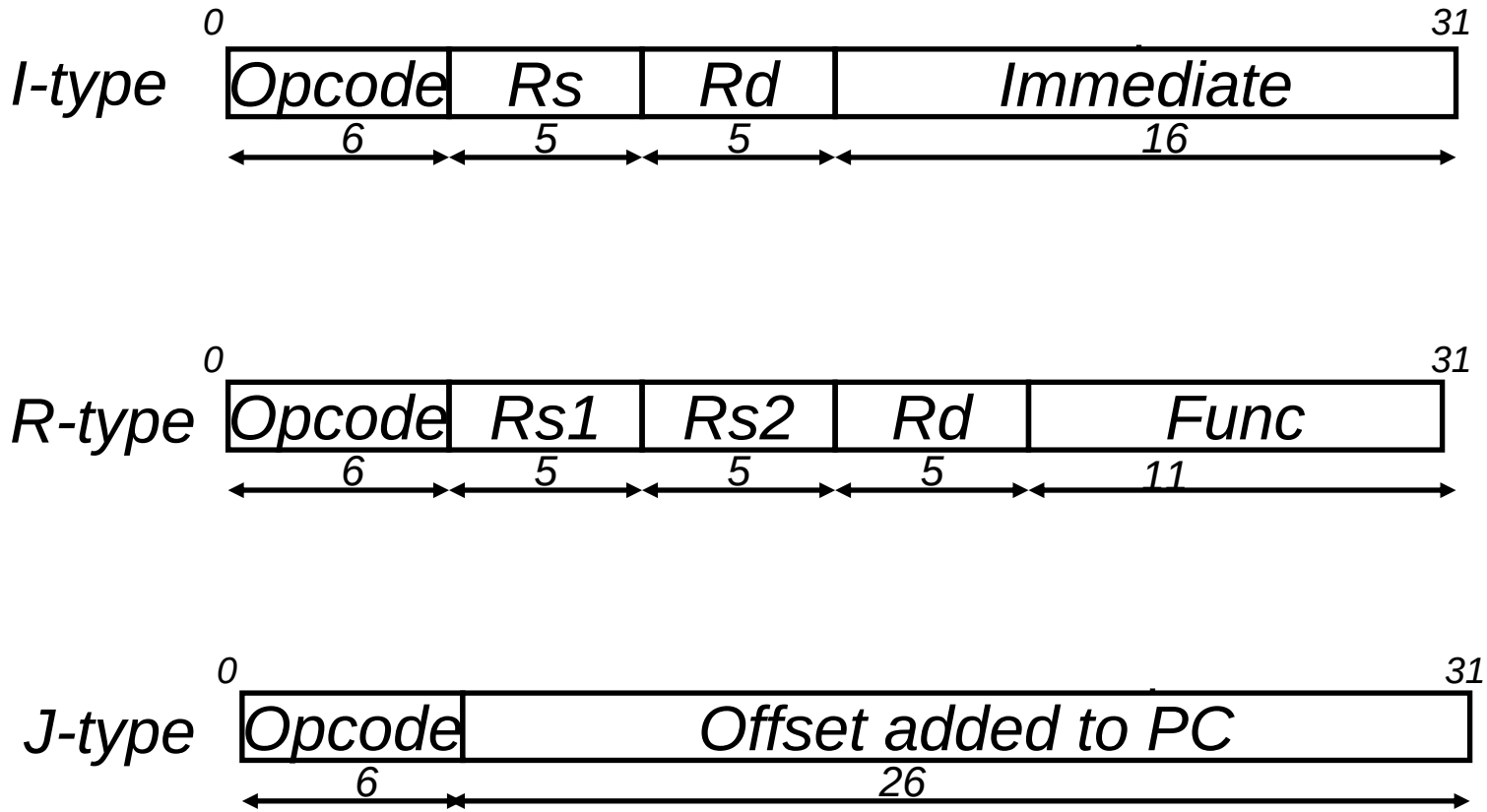
- Integer arithmetic
  - [add, sub] x [signed, unsigned] x [register,immediate]
  - e.g., ADD, ADDI, ADDU, ADDUI, SUB, SUBI, SUBU, SUBUI
- Logical
  - [and, or, xor] x [register, immediate]
  - e.g., AND, ANDI, OR, ORI, XOR, XORI
- Load upper half immediate load
  - It takes two instructions to load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

UPPSALA UNIVERSITET

More Generic ALU Ops

- Shifts
  - [left, right] x [logical, arithmetic] x [immediate, reg]
  - e.g., SLL, SRAI, ...
- Set conditional
  - [lt, gt, le, ge, eq, ne] x [immediate, reg]
  - e.g., SLT, SGEI, ...
  - Puts a 1 or a 0 in the destination register

Generic Instruction Formats



UPPSALA UNIVERSITET

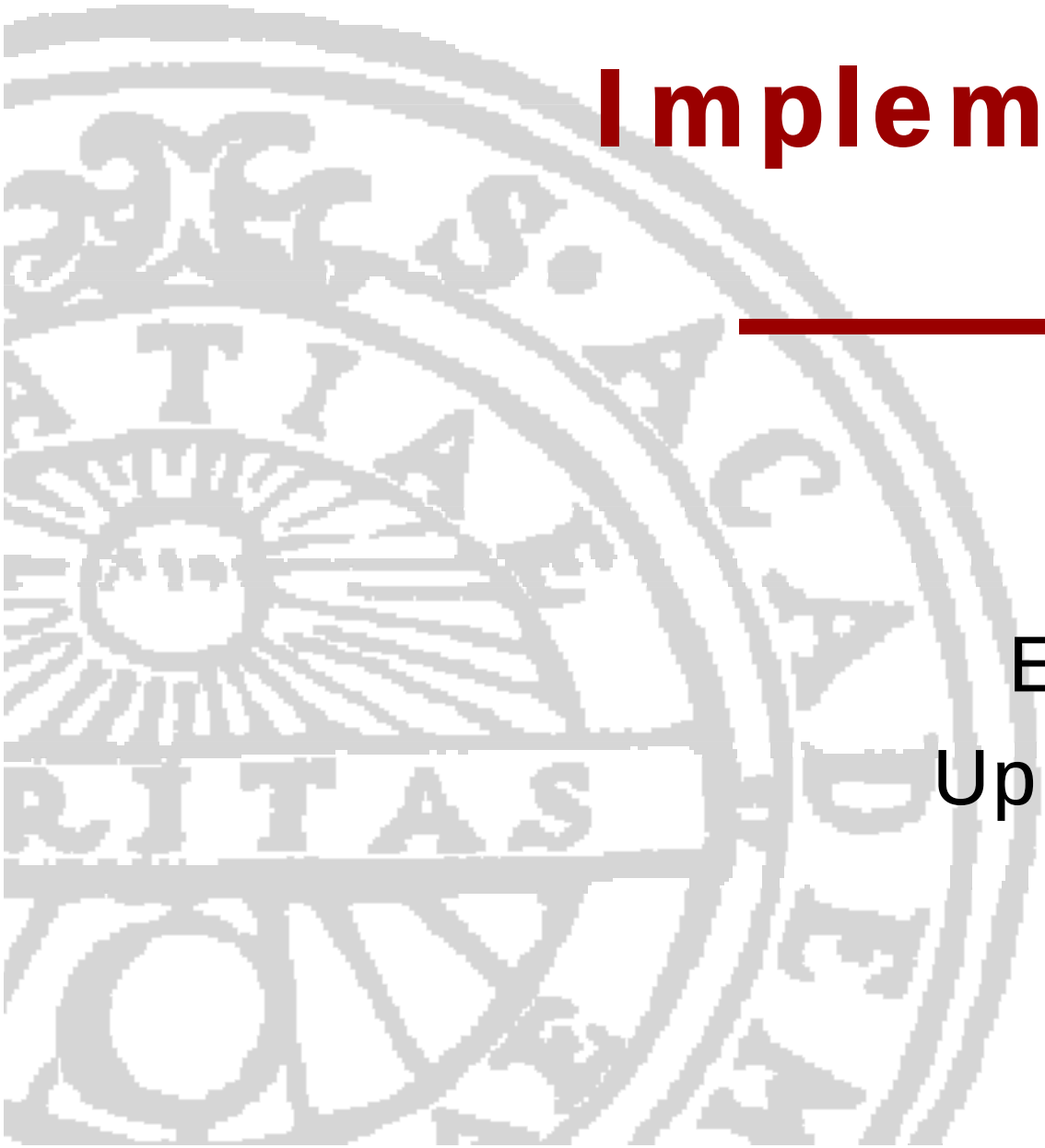
Generic FP Instructions

- Floating Point arithmetic
  - [add, sub, mult, div] x [double, single]
  - e.g., ADDD, ADDF, SUBD, 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

Simple Control

- Branches if equal or if not equal
  - 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”)

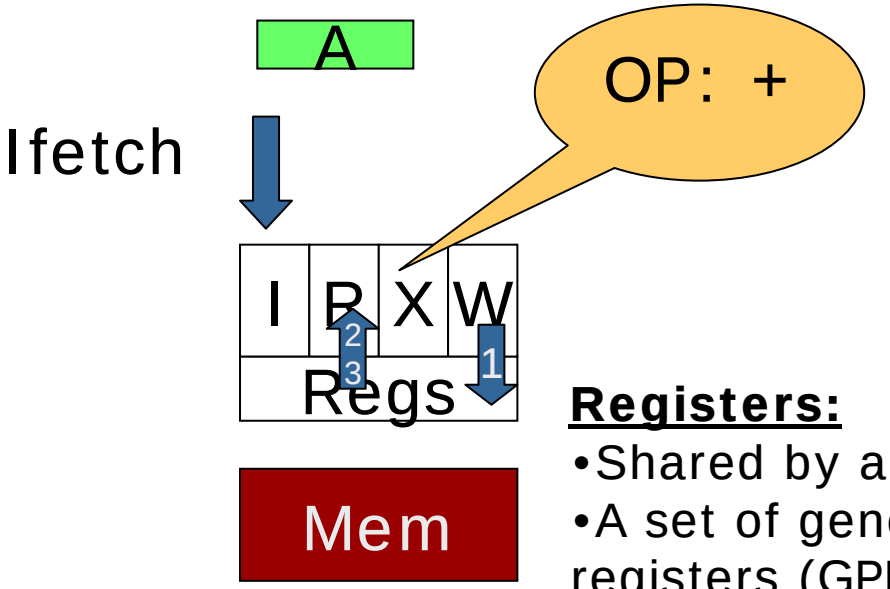


Implementing ISAs --pipelines

Erik Hagersten
Uppsala University

EXAMPLE: pipeline implementation

Add R1, R2, R3

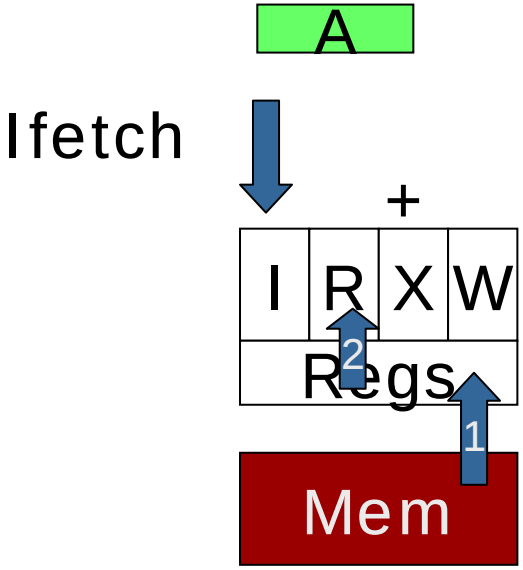


Registers:

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

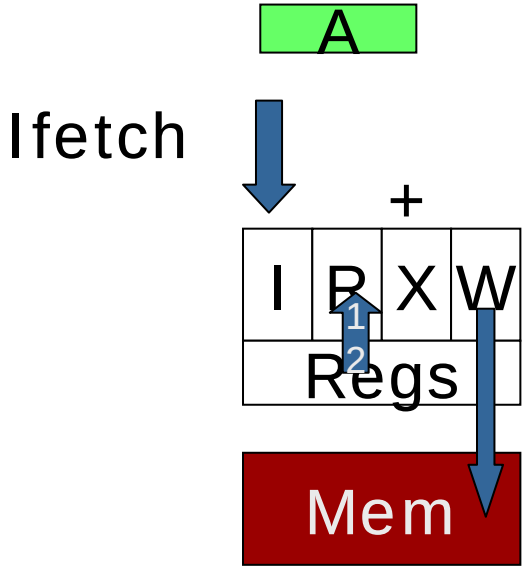
Load Operation:

LD R1, mem[cnst+R2]



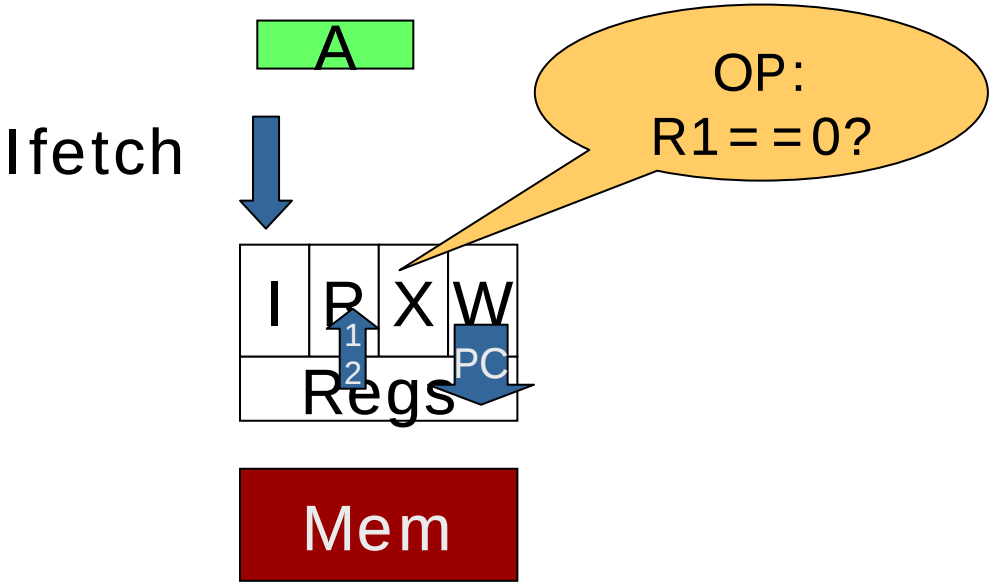
Store Operation:

ST mem[cnst+R1], R2



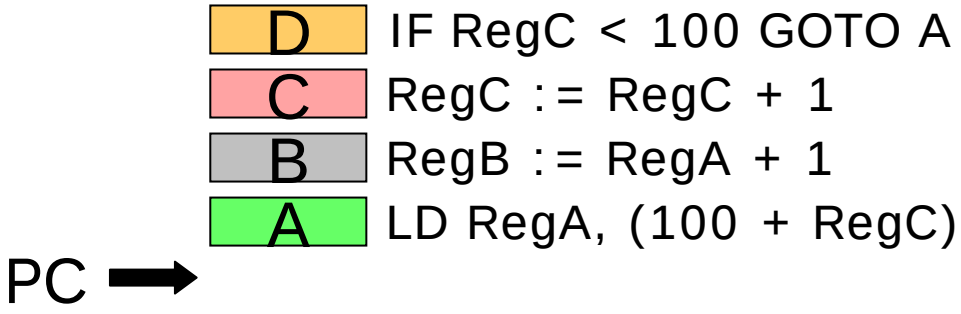
EXAMPLE: Branch to R2 if R1 == 0

BEQZ R1, R2



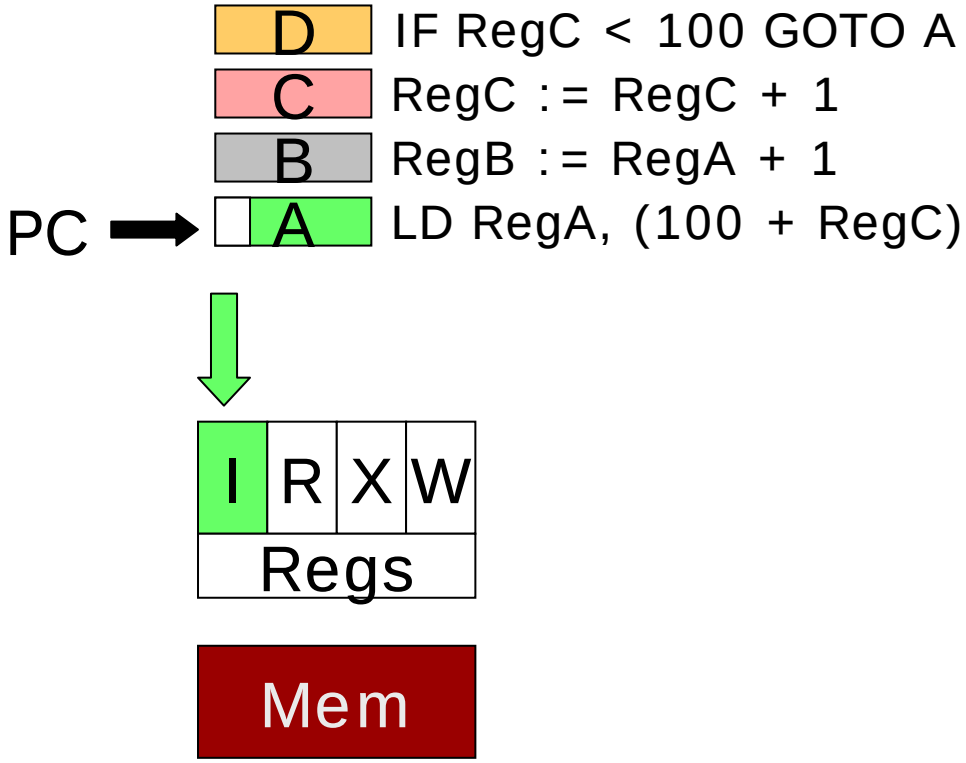
UPPSALA UNIVERSITET

Initially



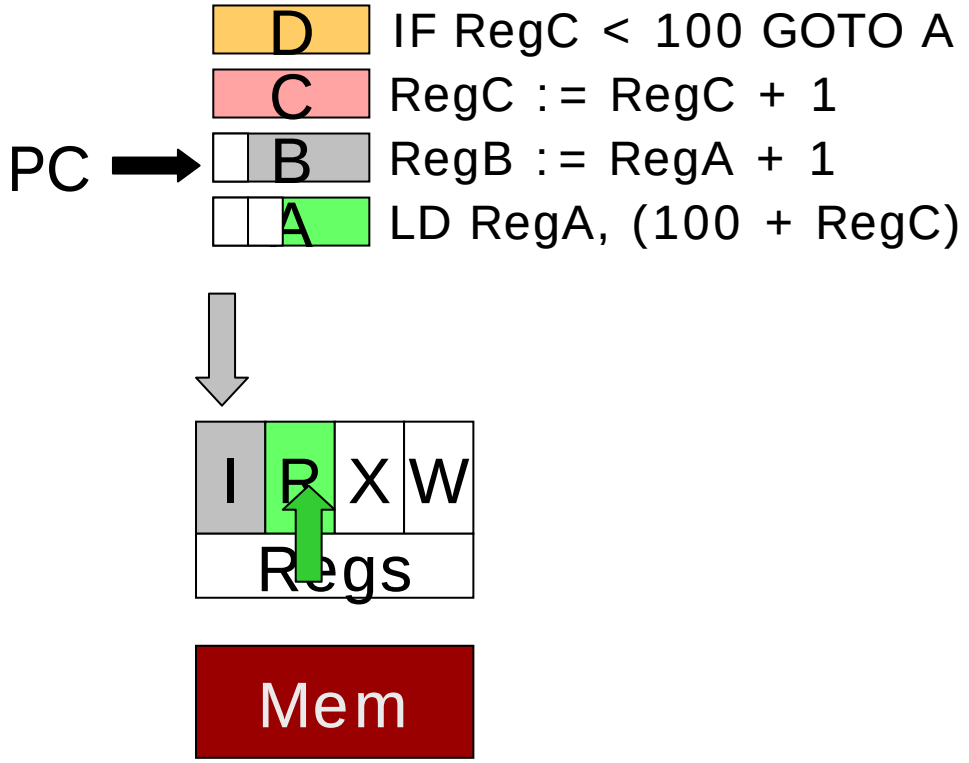

UPPSALA
UNIVERSITET

Cycle 1


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

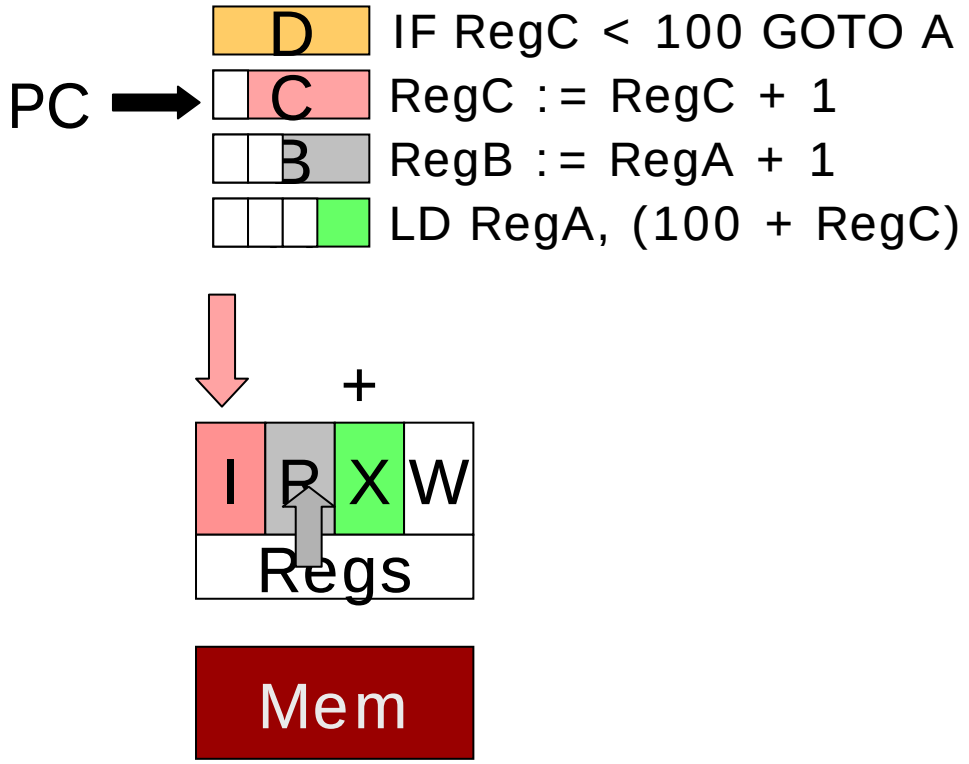
UPPSALA UNIVERSITET

Cycle 2

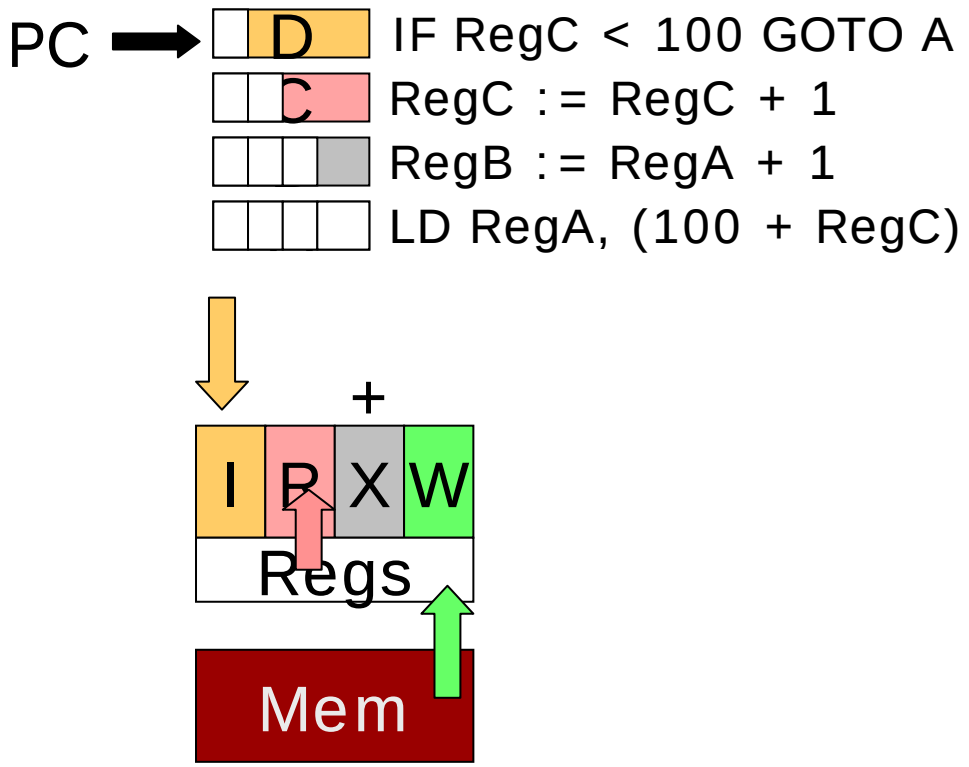



UPPSALA
UNIVERSITET

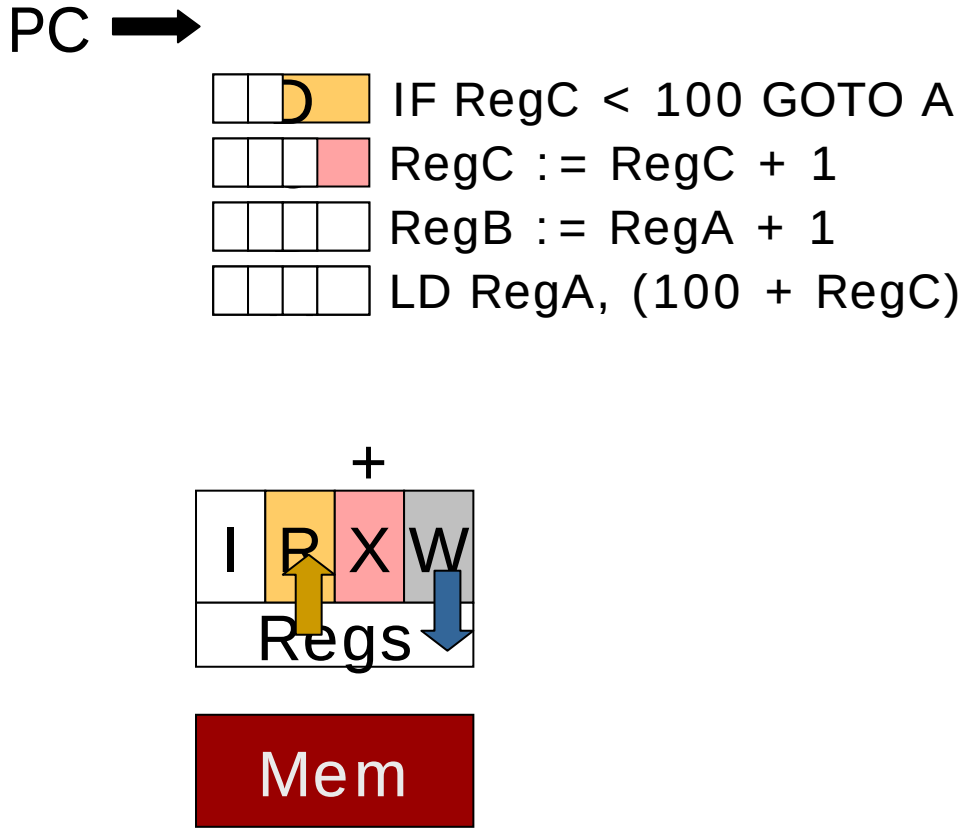
Cycle 3


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 4


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

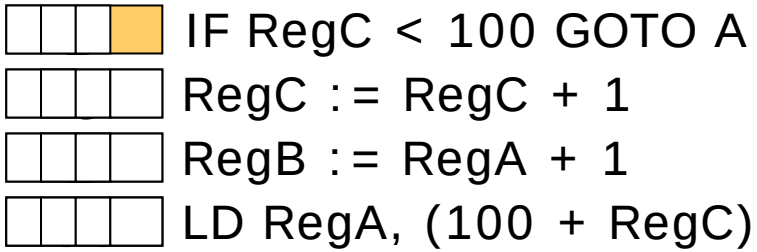
Cycle 5

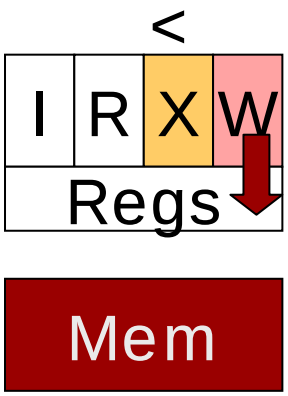

PC
IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
+
I
R
X
W
Regs
Mem


UPPSALA
UNIVERSITET

Cycle 6

PC ➡

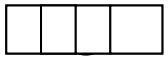

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)


<
I
R
X
W
Regs
Mem

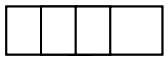
Cycle 7

PC ➡

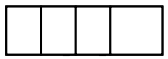
IF RegC < 100 GOTO A



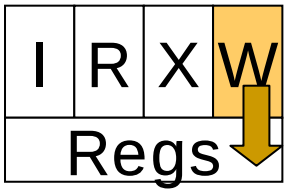
RegC := RegC + 1



RegB := RegA + 1

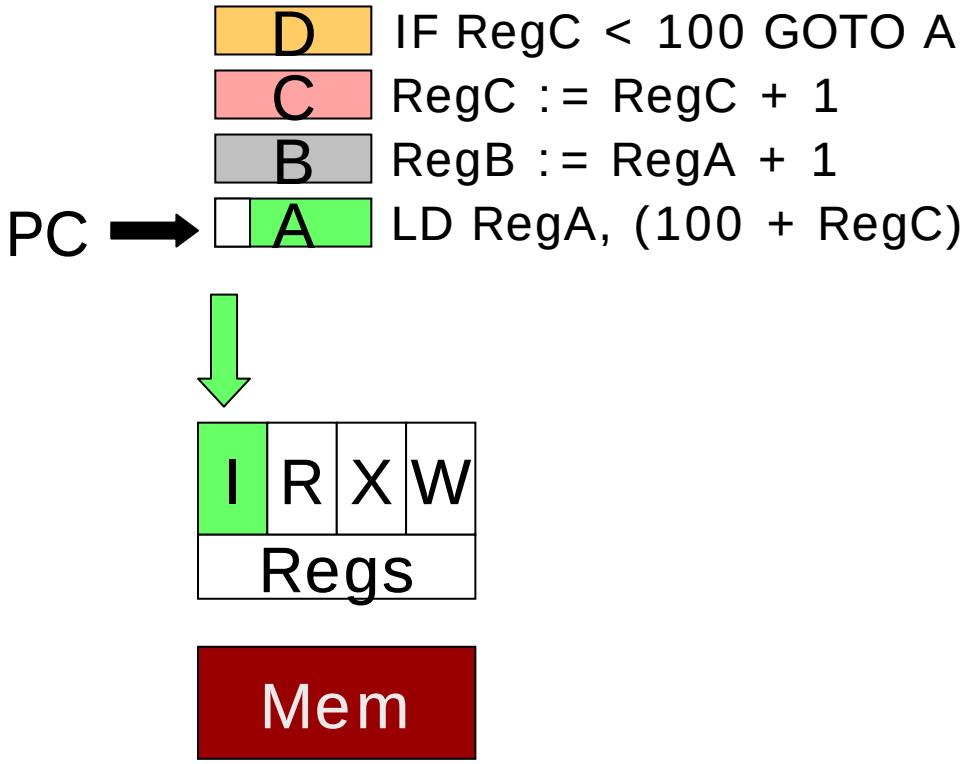


LD RegA, (100 + RegC)

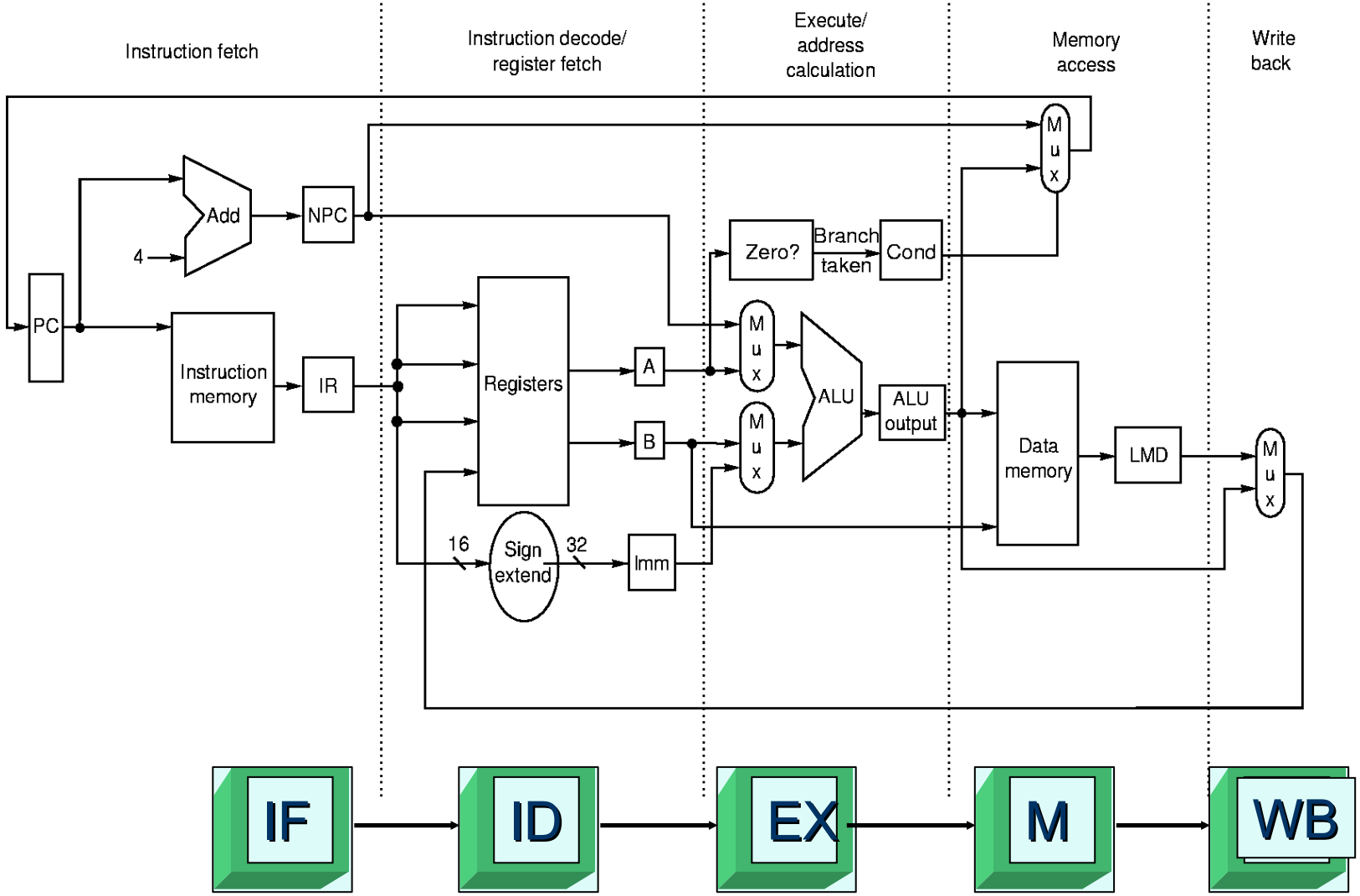


Branch ➔ Next PC

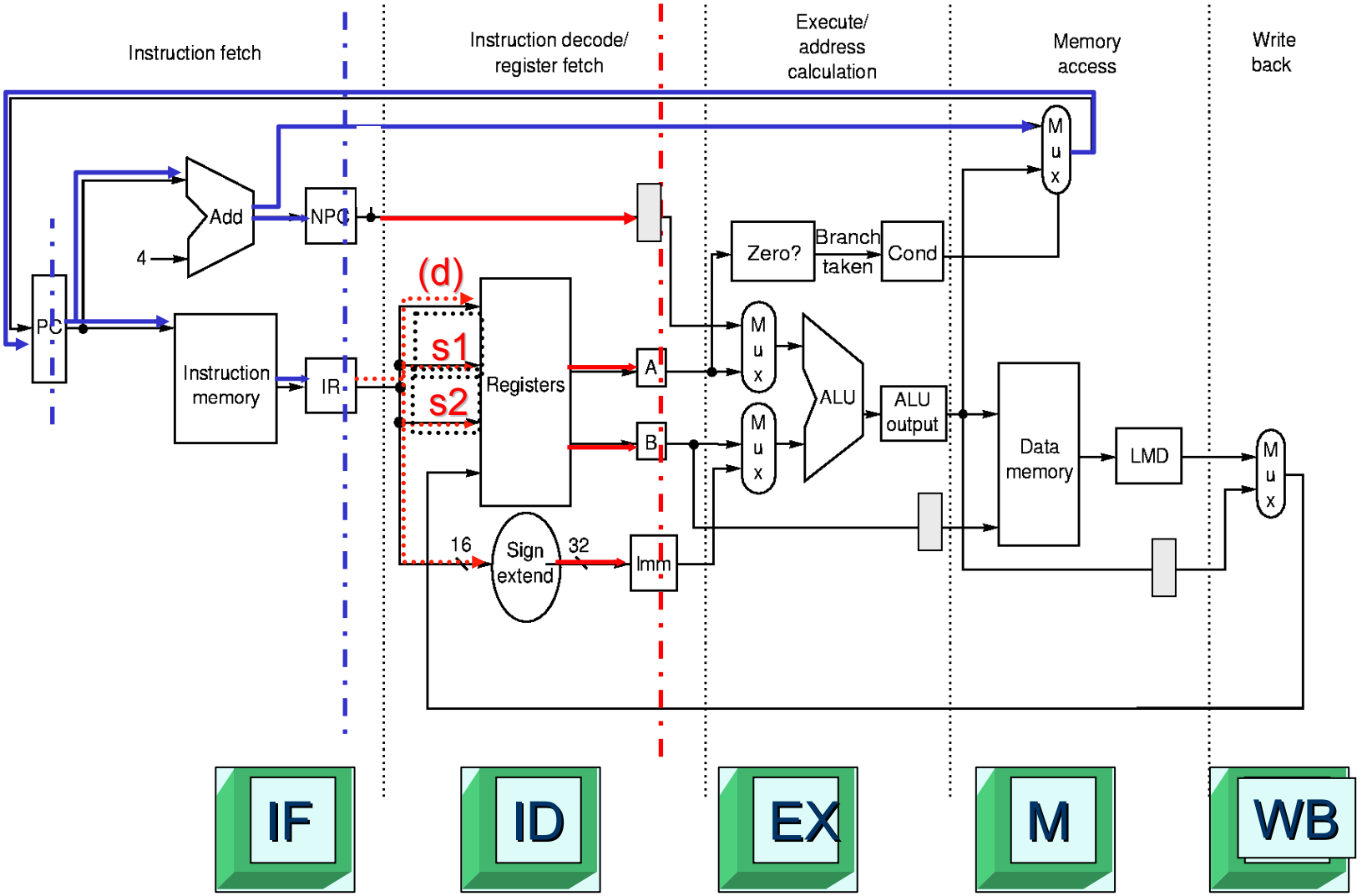
Cycle 8


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

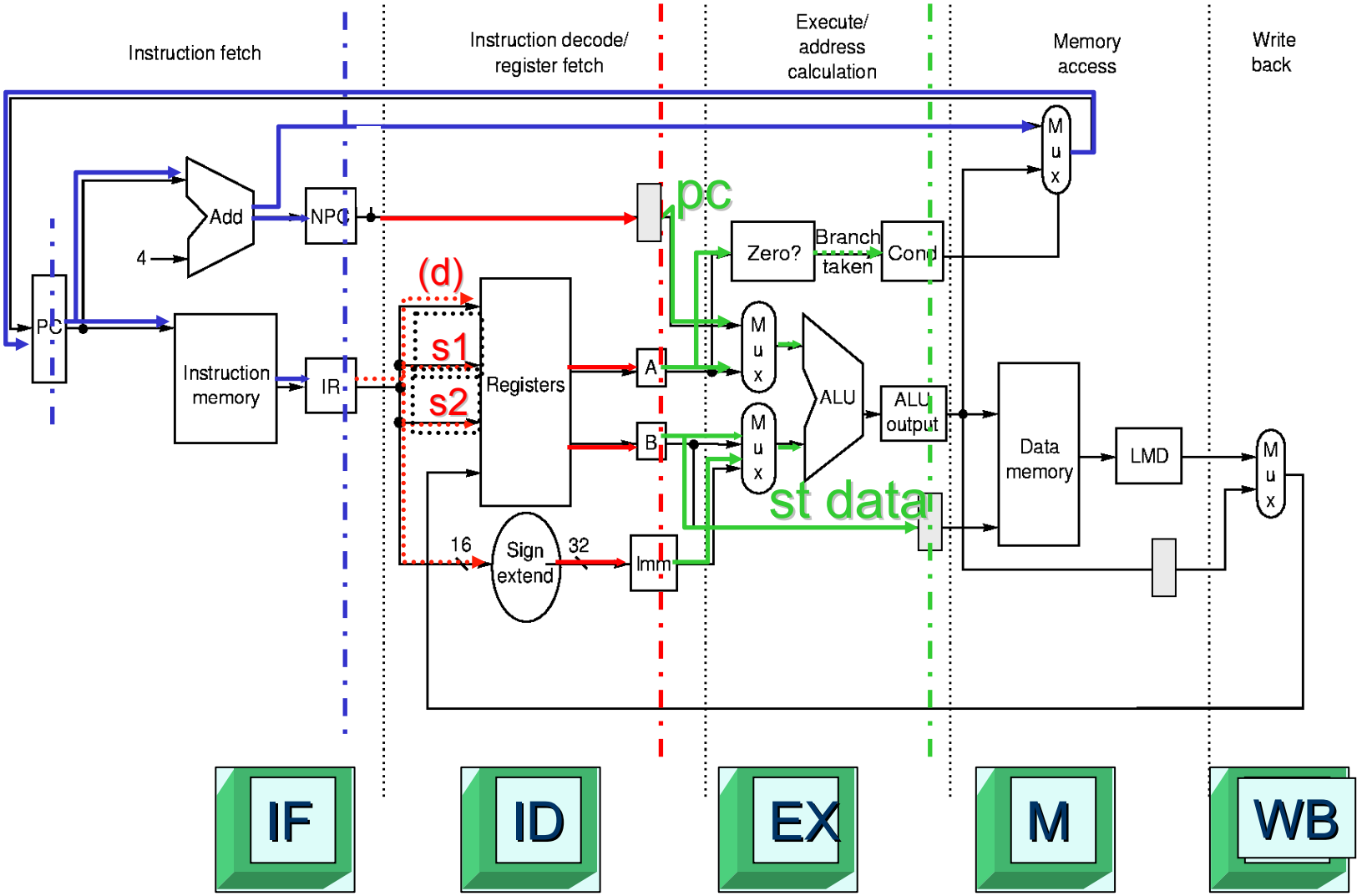
Example: 5-stage pipeline



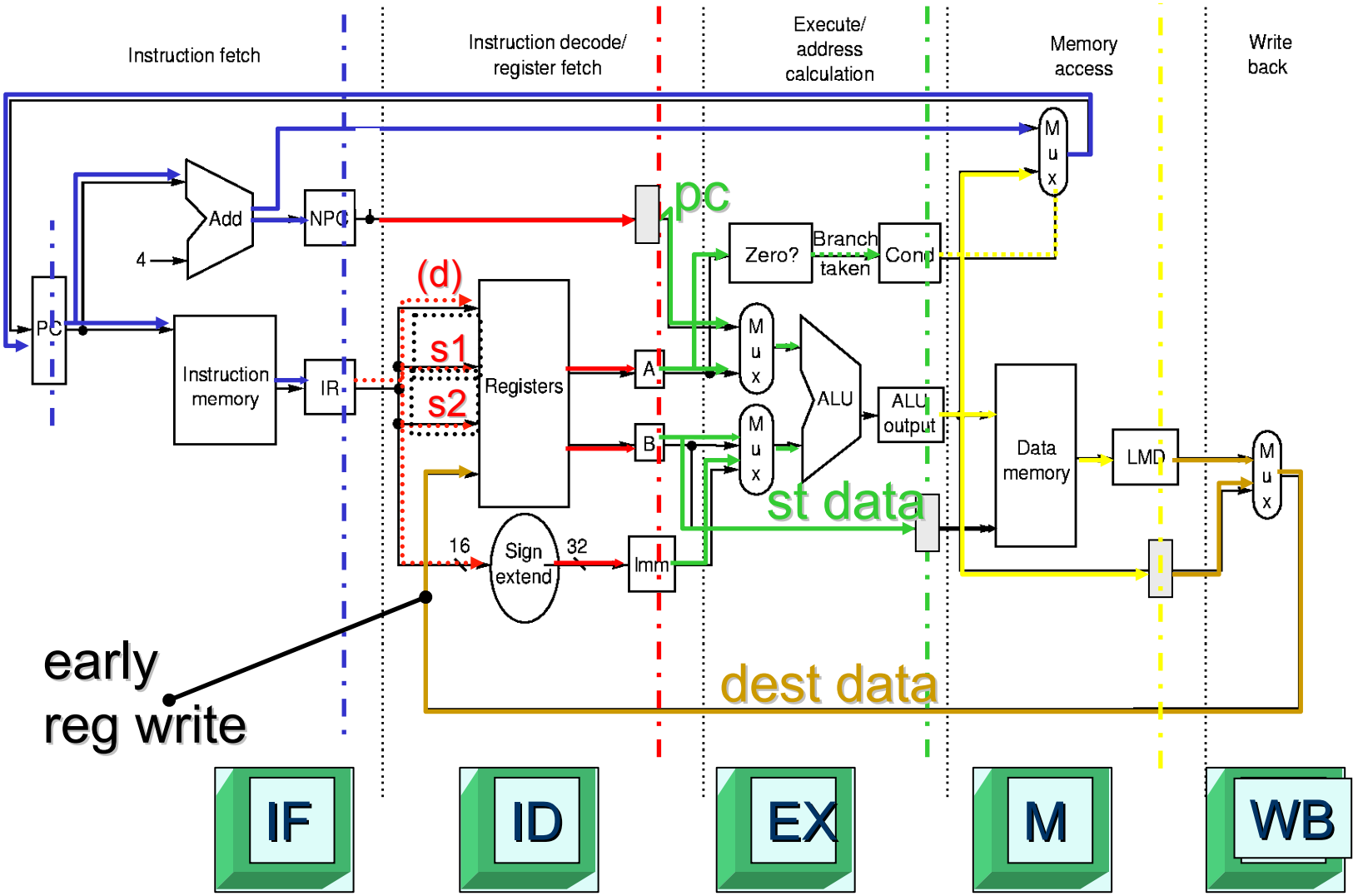
Example: 5-stage pipeline



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 - 100 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) Op i+1 modifies A before Opi. The value in A is the one written by Opi, i.e., an old A.

UPPSALA UNIVERSITET

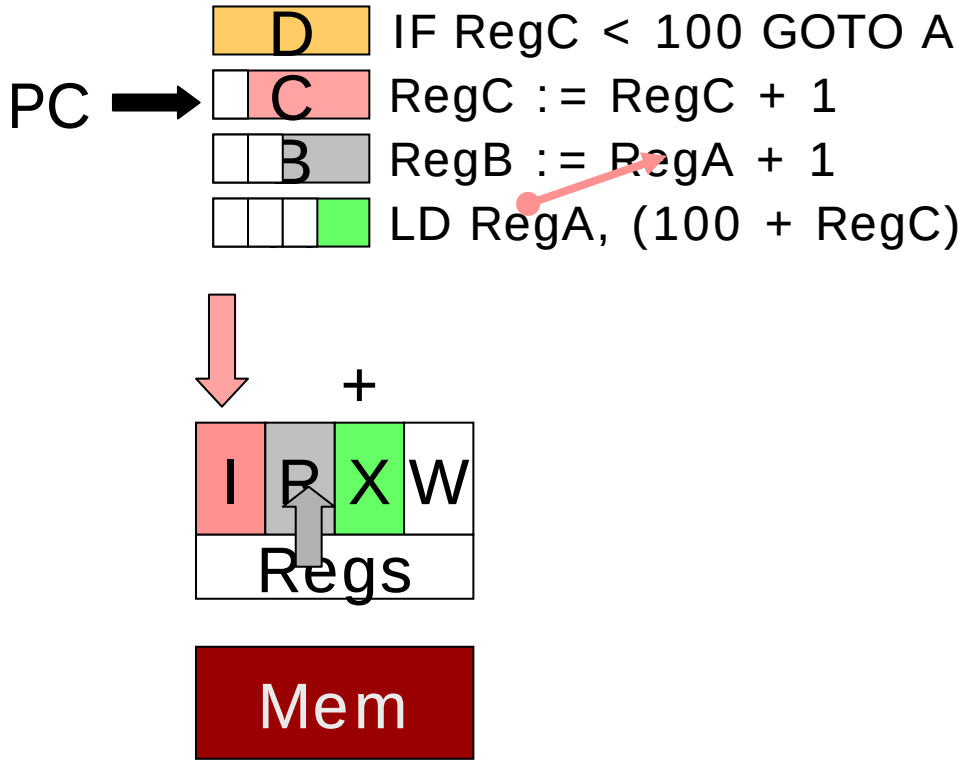
Hazard avoidance techniques

Static techniques (compiler): code scheduling to avoid hazards

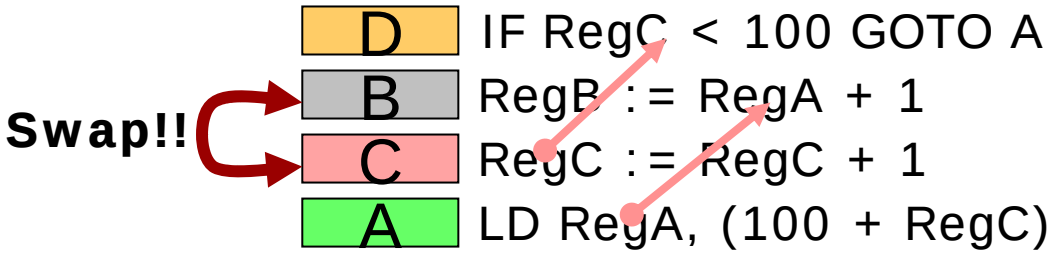
Dynamic techniques: hardware mechanisms to eliminate or reduce impact of hazards (e.g., out-of-order stuff)

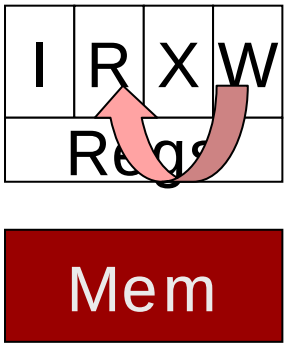
Hybrid techniques: rely on compiler as well as hardware techniques to resolve hazards (e.g. VLIW support – later)

Cycle 3

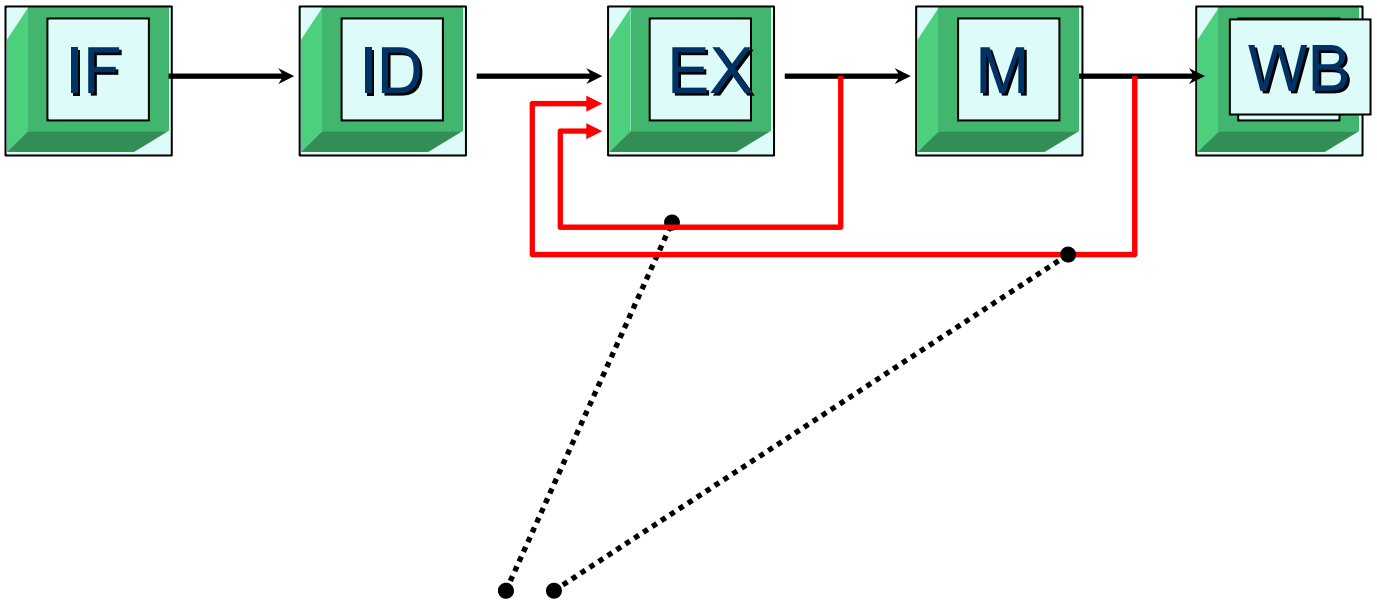

D
C
B
PC
IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
+
I
P
X
W
Regs
Mem

Fix alt1: code scheduling

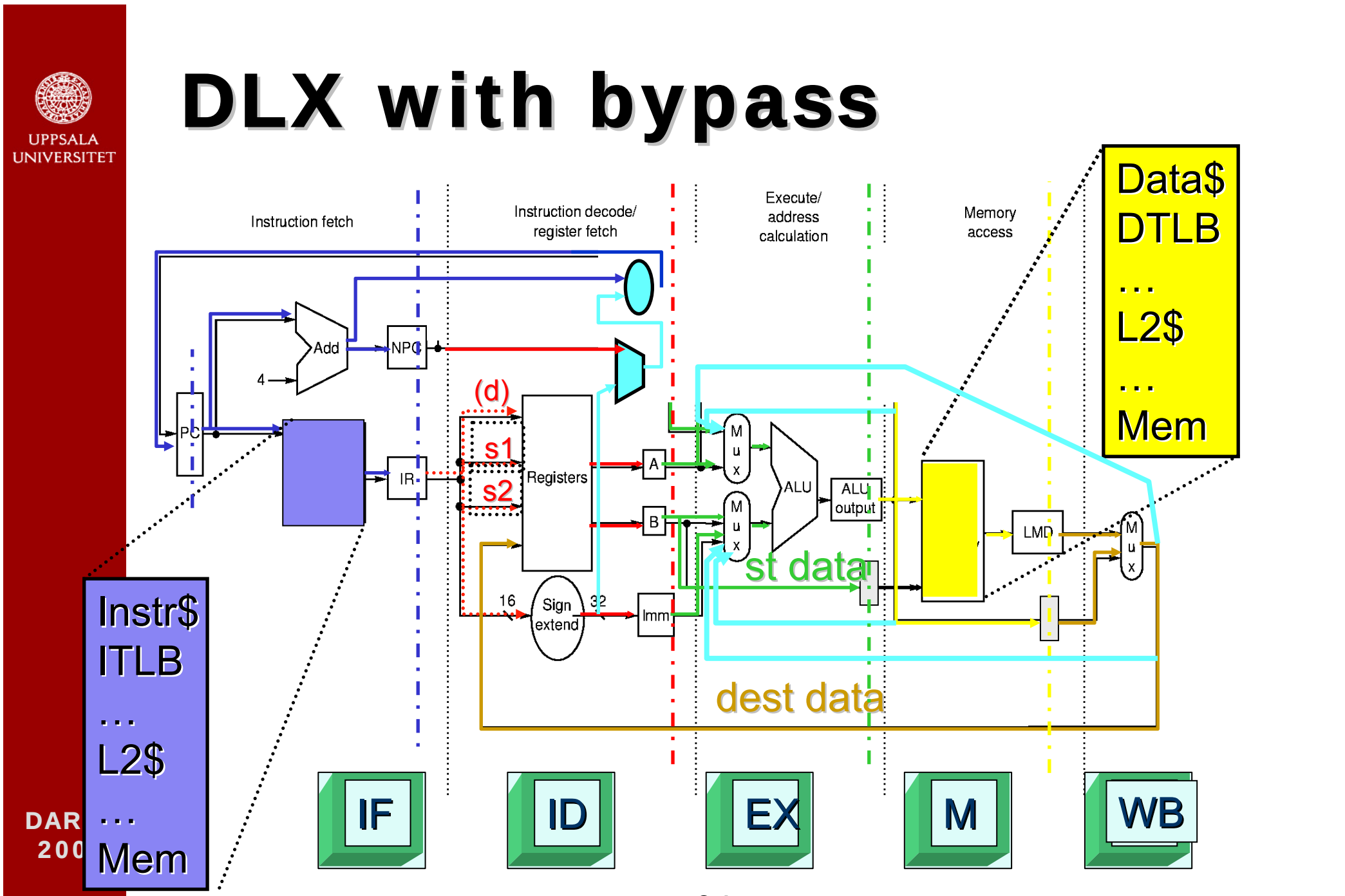




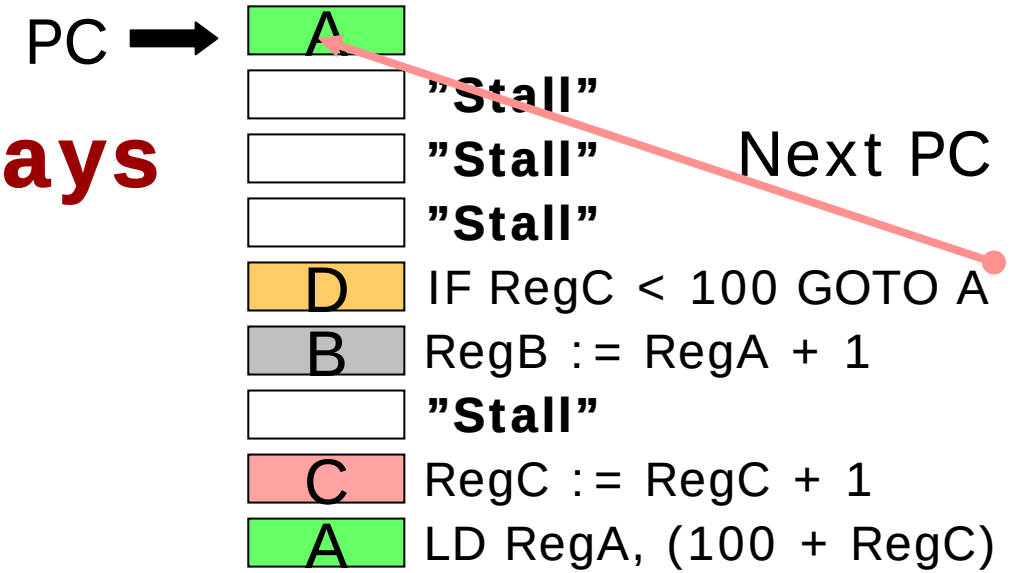
Fix alt2: Bypass hardware

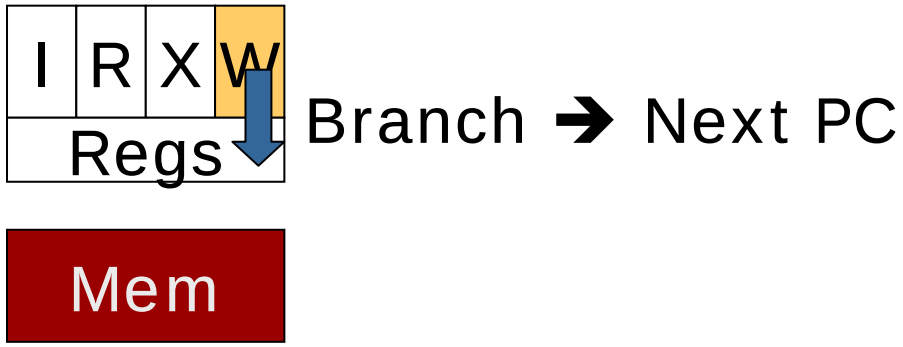


- Forwarding (or bypassing): provides a direct path from M and WB to EX
- Only helps for ALU ops. What about load operations?


DLX with bypass
Instruction fetch
Instruction decode/ register fetch
Execute/ address calculation
Memory access
Data$
DTLB
…
L2$
…
Mem
Add
4
NPC
PC
IR
(d)
s1
s2
Registers
A
B
Imm
16
Sign extend
32
M u x
ALU
ALU output
LMD
st data
dest data
Instr$
ITLB
…
L2$
…
Mem
IF
ID
EX
M
WB

Branch delays

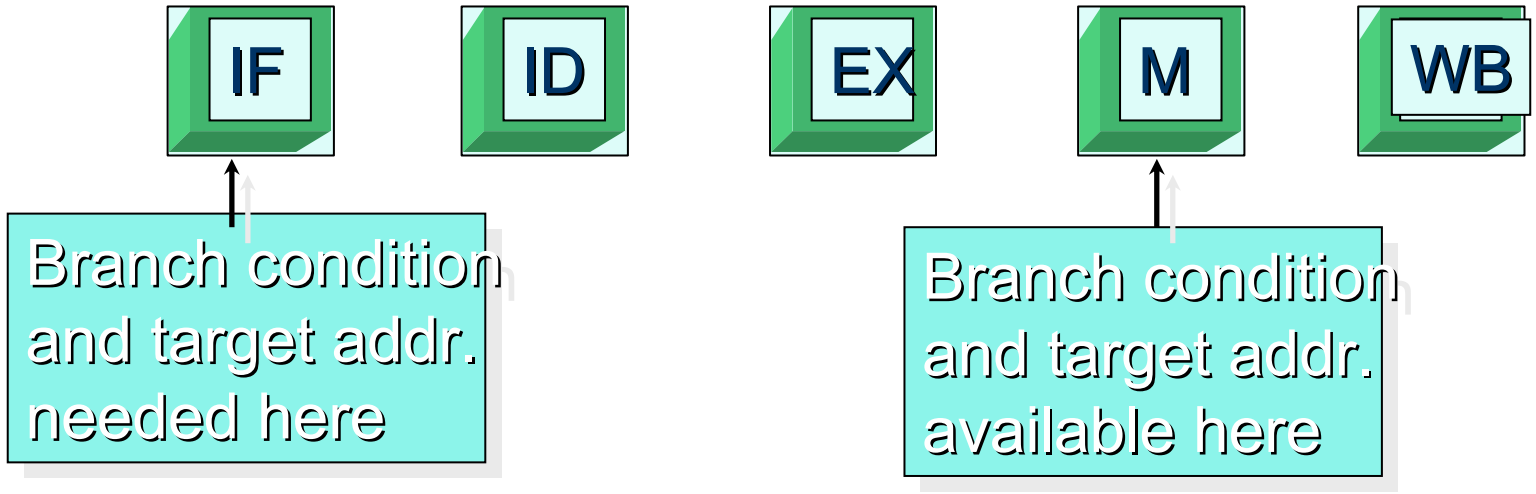




8 cycles per iteration of 4 instructions ☹

Need longer basic blocks with independent instr.

Avoiding control hazards



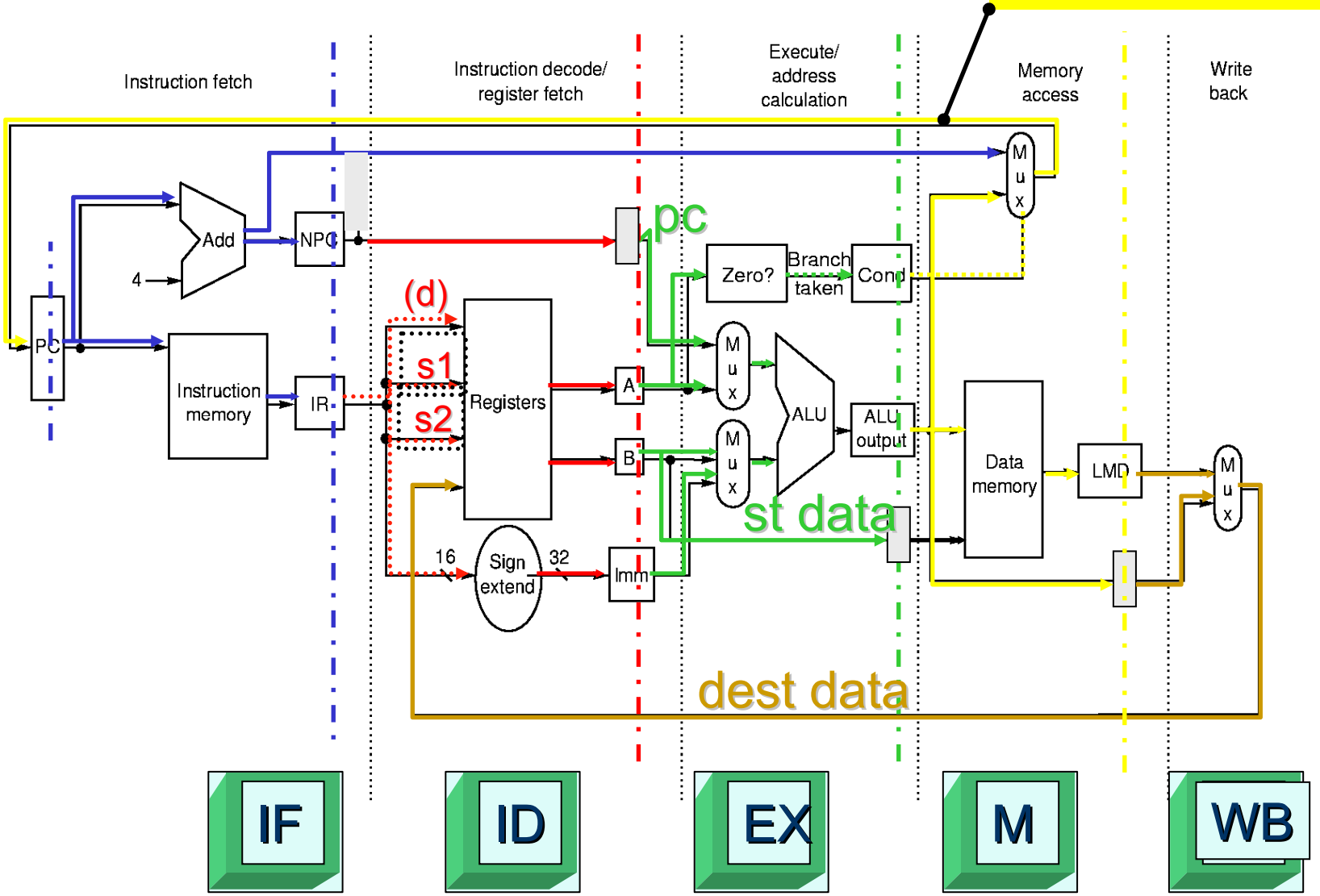
Duplicate resources in ALU to compute branch condition and branch target address earlier

Branch delay cannot be completely eliminated

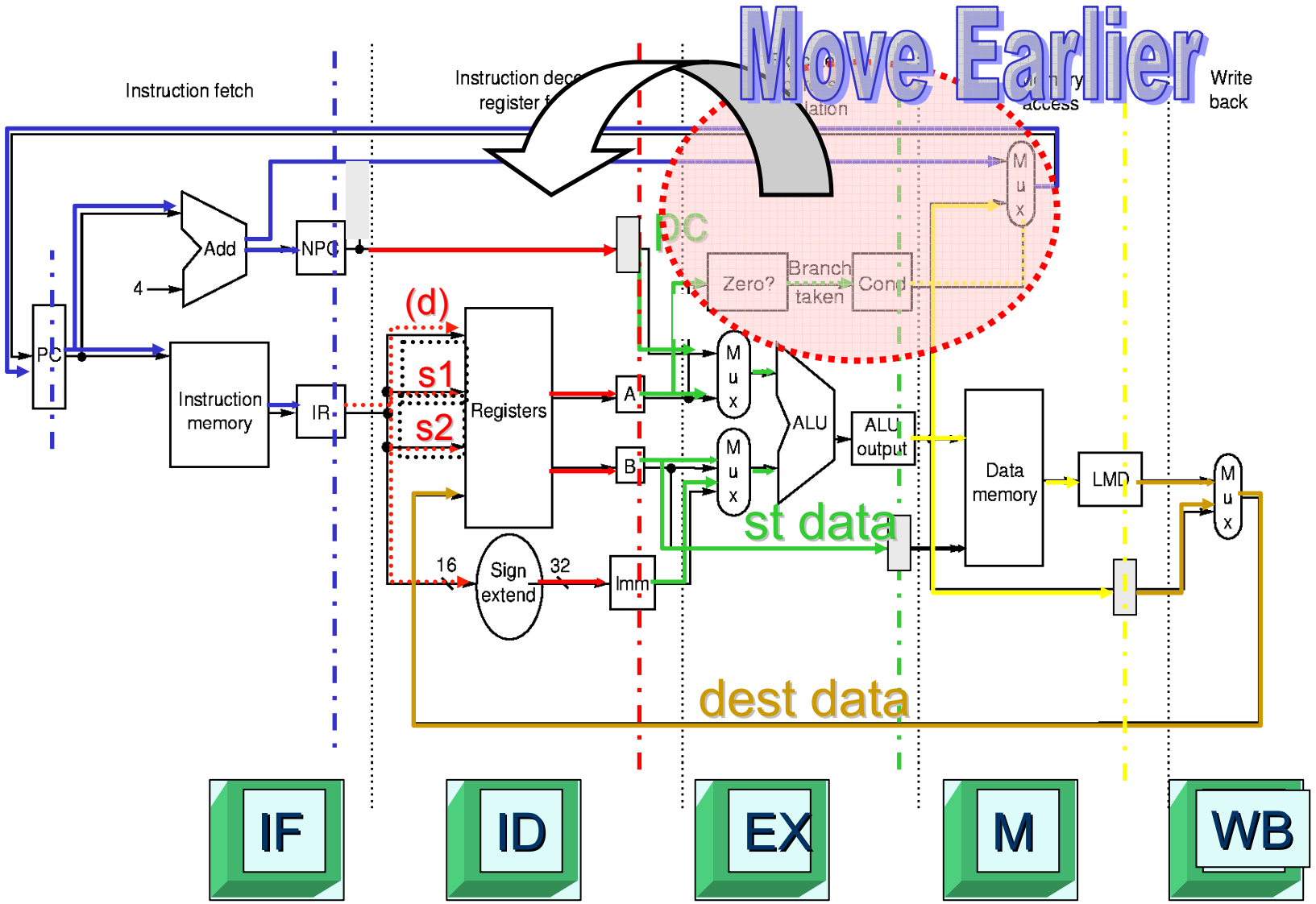
Branch prediction and code scheduling can reduce the branch penalty

Taking a Branch

PC := PC + Imm



Fix1: Minimizing Branch Delay Effects



UPPSALA UNIVERSITET

Fix2: Static tricks

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

Predict Branch not taken (a fairly rare case)

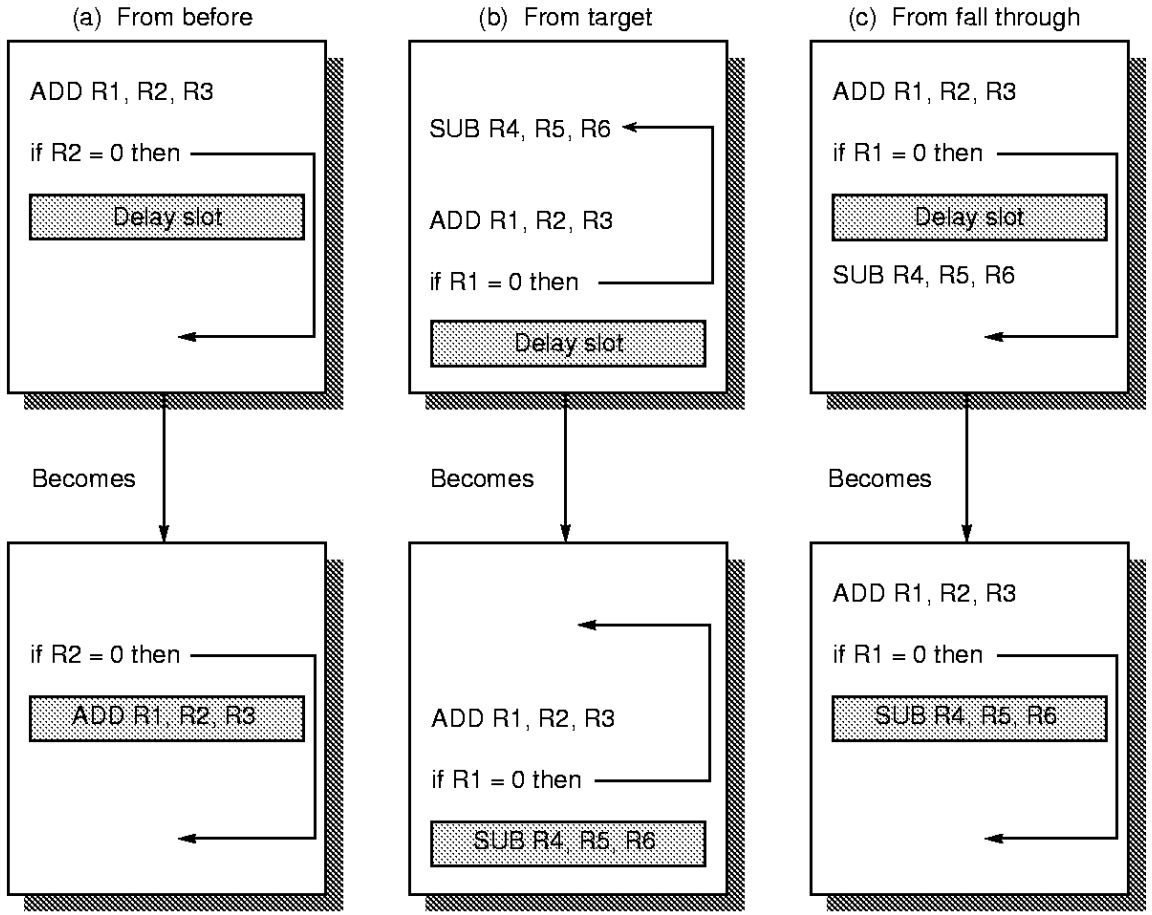
- Execute successor instructions in sequence
- "Squash" instructions in pipeline if the branch is actually taken
- Works well if state is updated late in the pipeline
- 30%-38% of conditional branches are not taken on average

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

UPPSALA UNIVERSITET

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

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 cycle delay slot

DARK2 2008

UPPSALA UNIVERSITET

Scheduling example

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

Iterations are independent => parallel execution

```
loop:   LD     F0, 0(R1)      ; F0 = array element
        ADDD   F4, F0, F2     ; Add scalar constant
        SD     0(R1), F4      ; Save result
        SUBI   R1, R1, #8     ; decrement array ptr.
        BNEZ   R1, loop       ; reiterate if R1 != 0
```

Can we eliminate all penalties in each iteration?

How about moving SD down?

DARK2 2008

UPPSALA UNIVERSITET

Scheduling in each loop iteration

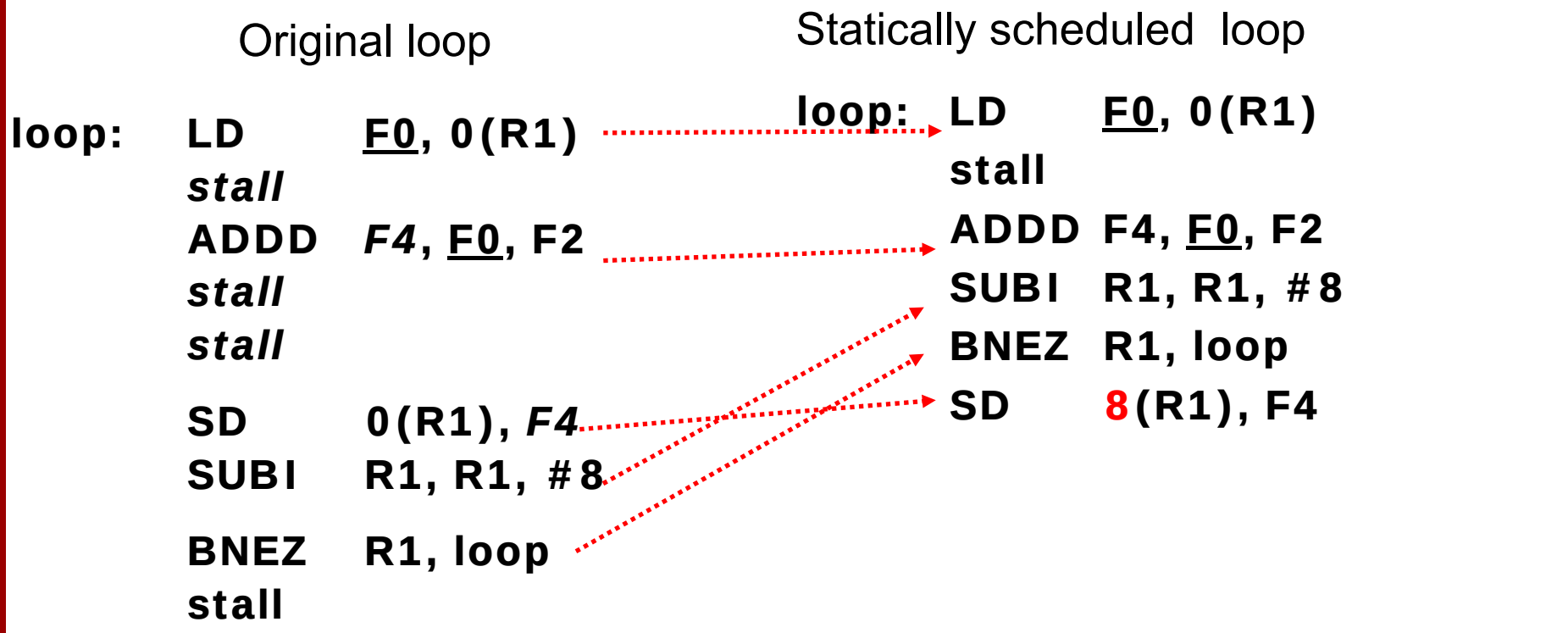
Original loop

```
loop:   LD      F0, 0(R1)
        stall
        ADDD    F4, F0, F2
        stall
        stall

        SD      0(R1), F4
        SUBI    R1, R1, #8

        BNEZ    R1, loop
        stall
```

5 instructions + 4 bubbles = 9 cycles / iteration
(~one cycle per iteration on a vector architecture)

Can we do better by scheduling across iterations?

Scheduling in each loop iteration



5 instruction + 4 bubbles = 9c / iteration

5 instruction + 1 bubble = 6c / iteration

Can we do even better by scheduling across iterations?

UPPSALA UNIVERSITET

Unoptimized loop unrolling 4x

```
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
```

24c/ 4 iterations = 6 c / iteration

DARK2 2008

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 steps:

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

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

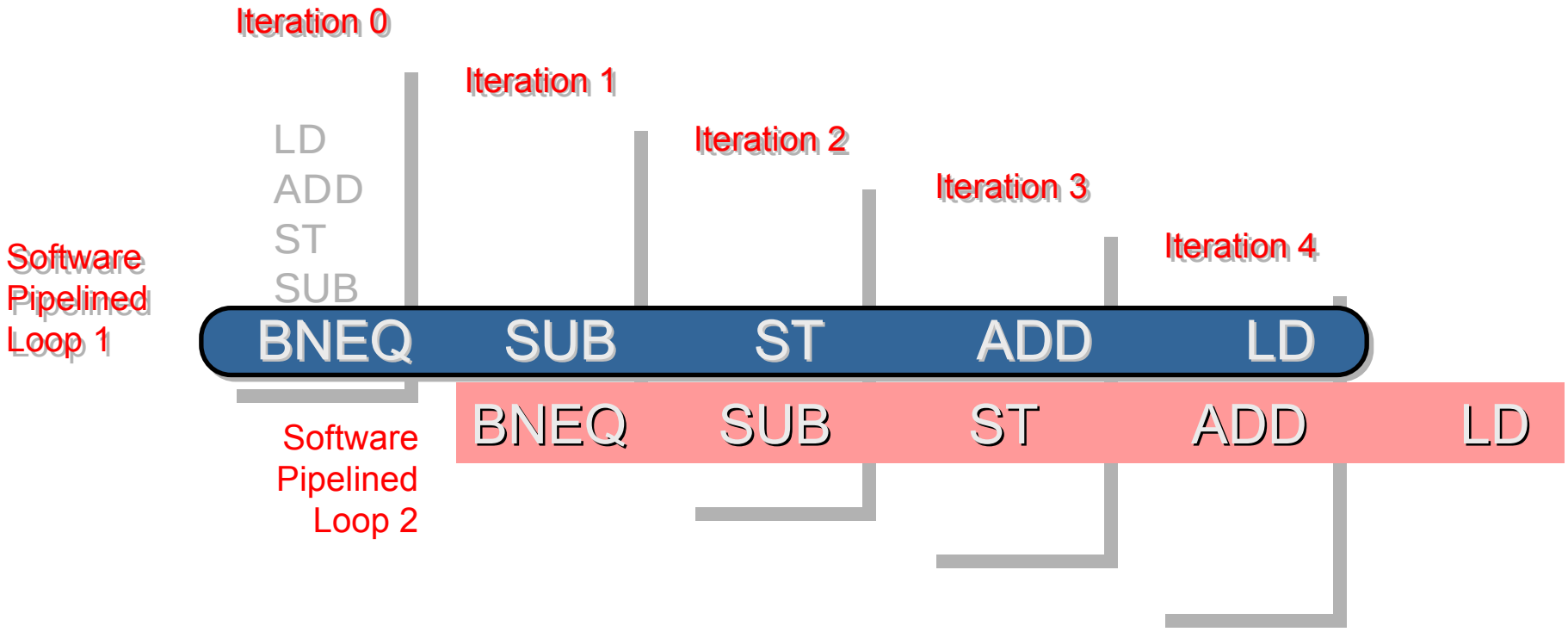
From 9c to 3.5c per iteration ==> speedup 2.6

DARK2 2008

Software pipelining 1(3)

Symbolic loop unrolling

- The instructions in a loop are taken from different iterations in the original loop

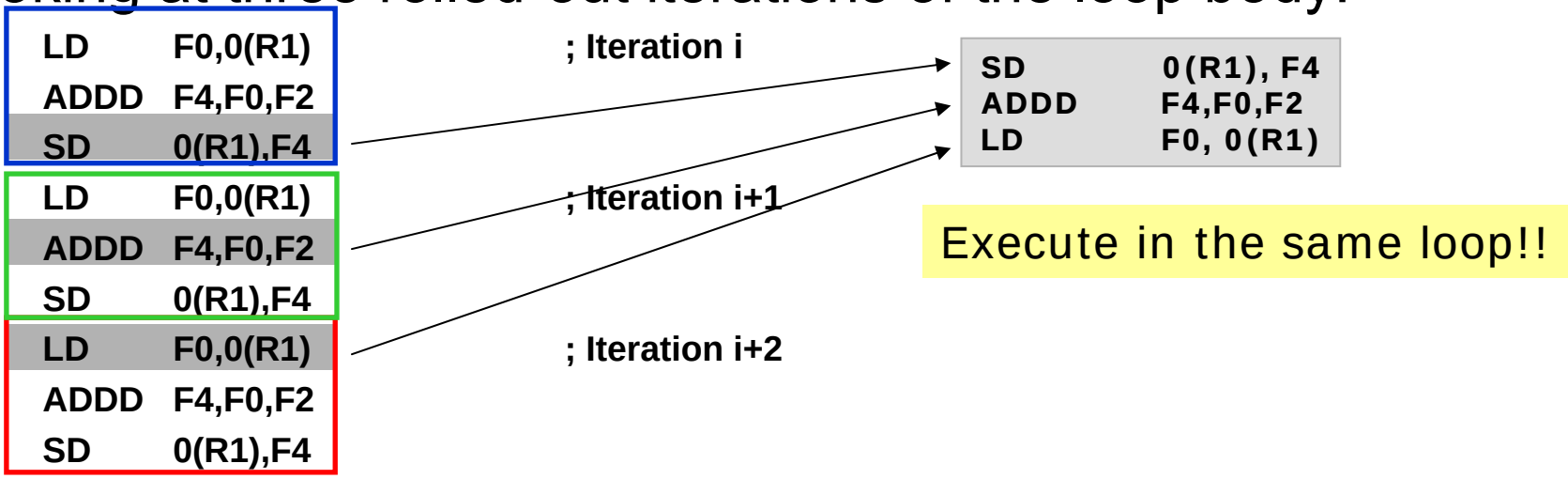


Software pipelining 2(3)

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:

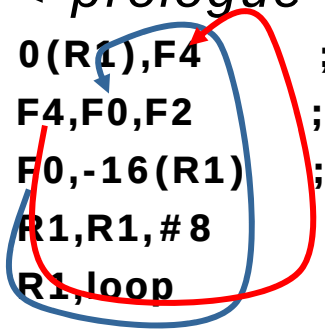


Execute in the same loop!!

Software pipelining 3(3)

Instructions from three consecutive iterations form the loop body:

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



- No data dependencies *within* a loop iteration
- The dependence distance is 1 iterations
- WAR hazard elimination is needed (register renaming)
- 5c / iteration, but only uses 2 FP regs (instead of 8)

UPPSALA UNIVERSITET

Software pipelining

- ”Symbolic Loop Unrolling”
- Very tricky for complicated loops
- Less code expansion than outlining
- Register-poor if ”rotating” is used
- Needed to hide large latencies (see IA-64)

DARK2 2008

UPPSALA UNIVERSITET

Dependencies: Revisited

Two instructions must be ***independent*** in order to execute in parallel

- Three classes of dependencies that limit parallelism:
- Data dependencies

  X := …

  …. := … X ….
- Name dependencies

  … := … X

  X := …
- Control dependencies

  If (X > 0) then

  Y := …

Getting desperate for ILP

Erik Hagersten
Uppsala University
Sweden

UPPSALA UNIVERSITET

Multiple instruction issue per clock

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

Superscalar:

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

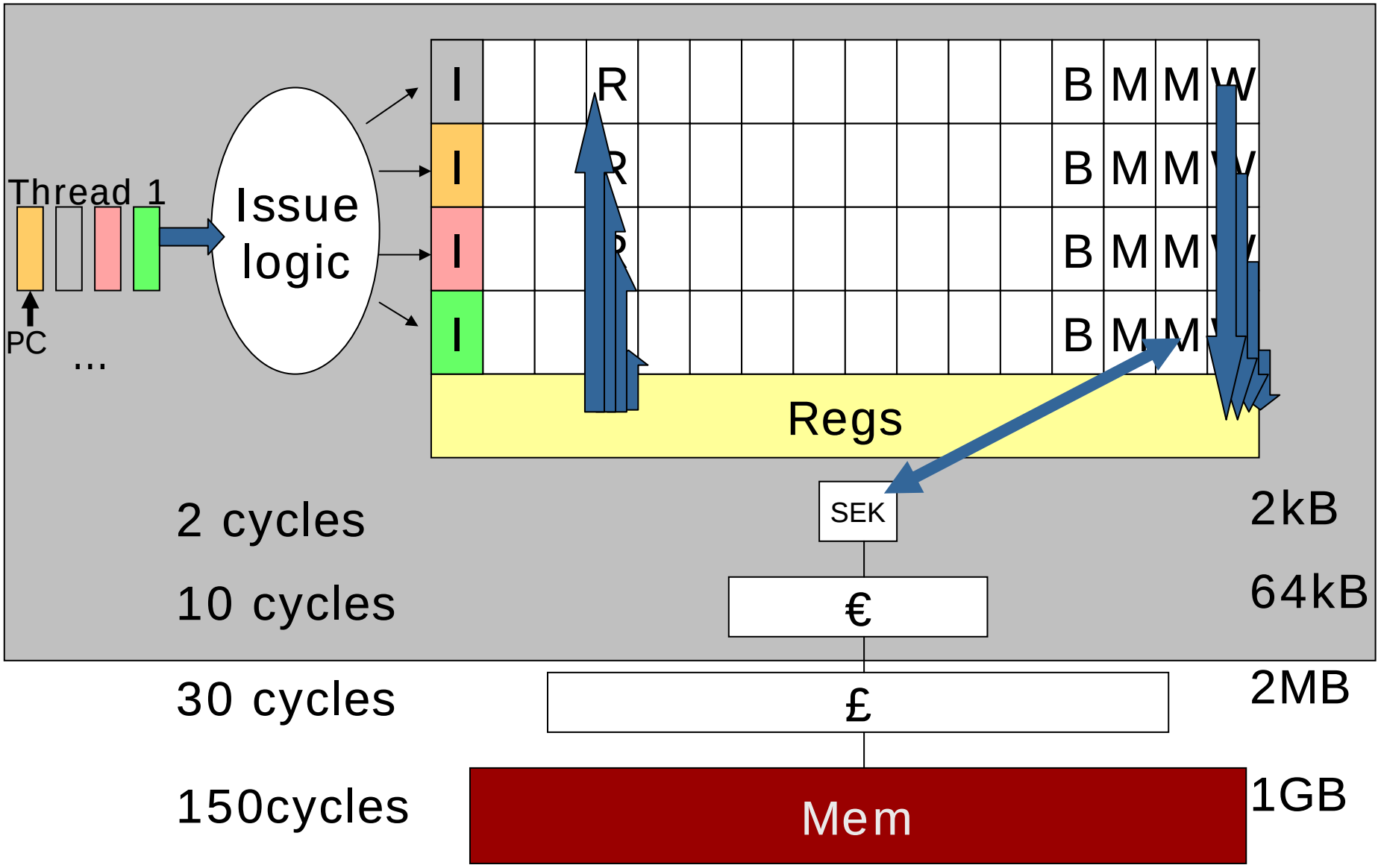
- Predominant: (PowerPC, SPARC, Alpha, HP-PA)

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)

Superscalars



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***						
Int.	IF	ID	EX	MEM	WB		
FP	**IF**	**ID**	**EX**	**MEM**	**WB**		
Int.		IF	ID	EX	MEM	WB	
FP		**IF**	**ID**	**EX**	**MEM**	**WB**	
Int.			IF	ID	EX	MEM	WB
FP			**IF**	**ID**	**EX**	**MEM**	**WB**

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

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

Can be scheduled dynamically with Tomasulo's alg.

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

UPPSALA UNIVERSITET

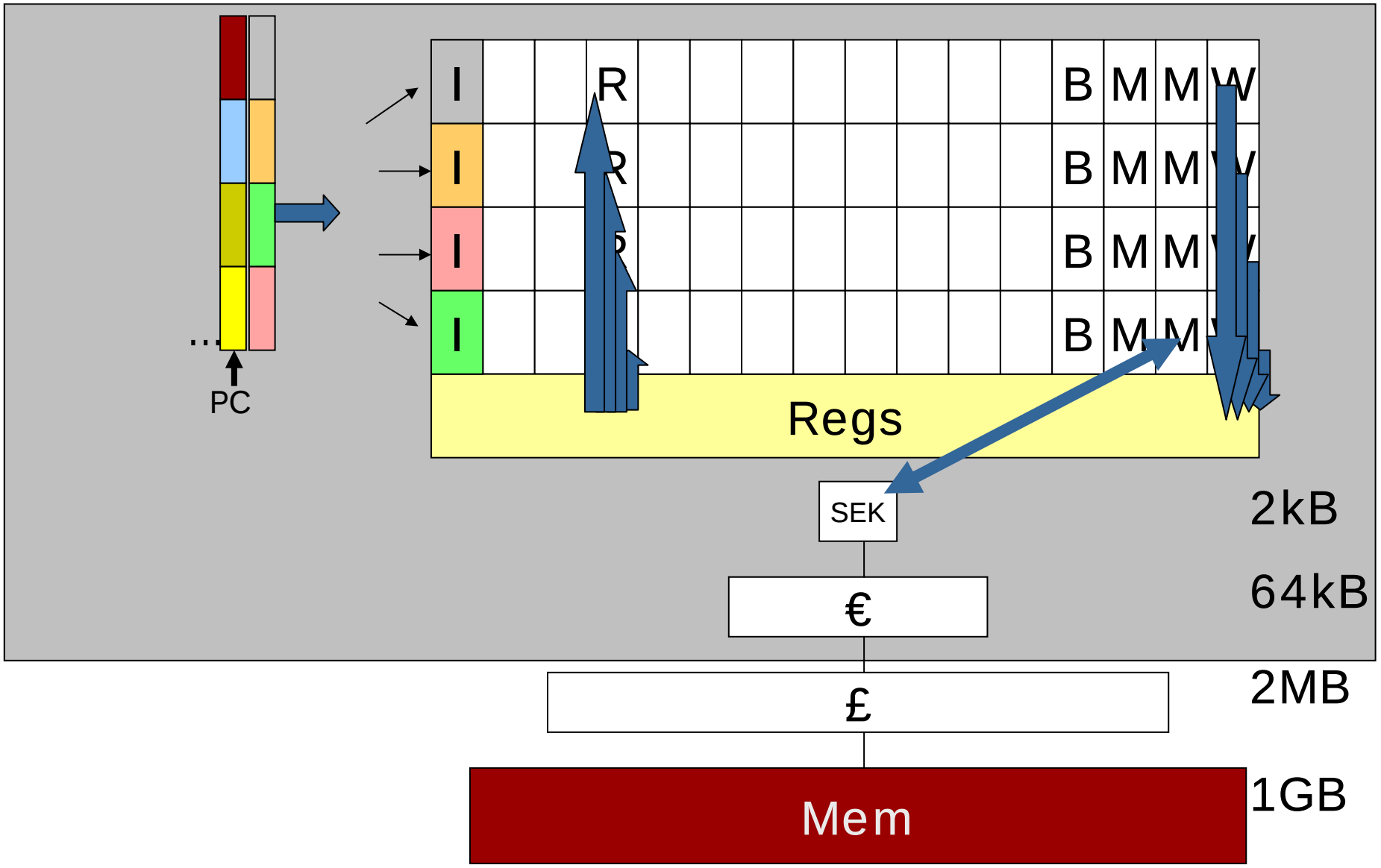
Limits to superscalar execution

- 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

Techniques to enlarge the instruction window to extract more ILP are important

Simple superscalars relying on compiler instead of HW complexity ➔ VLIW

VLIW: Very Long Instruction Word



UPPSALA UNIVERSITET

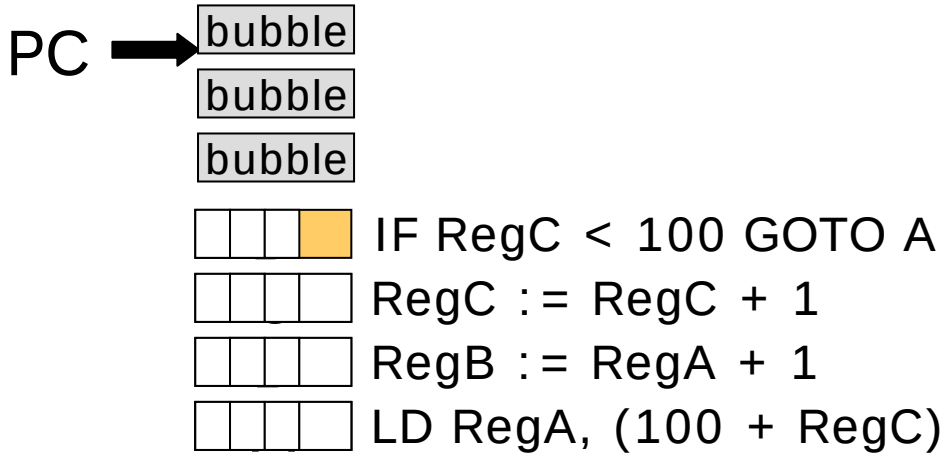
Very Long Instruction Word (VLIW)

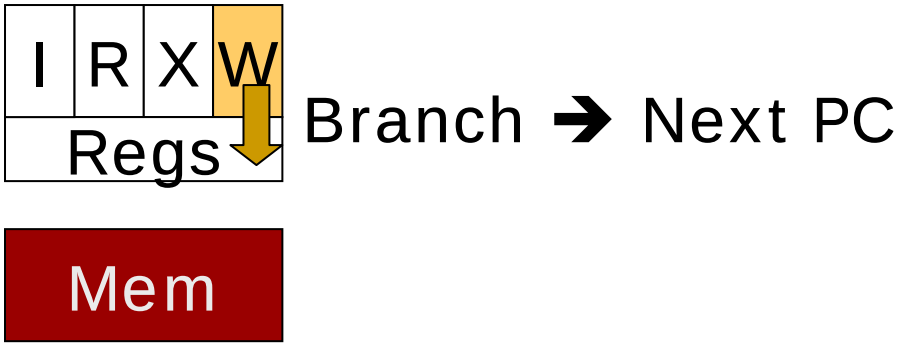
Compiler is responsible for instruction scheduling

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

VLIW will be revisited later on….

Predict next PC

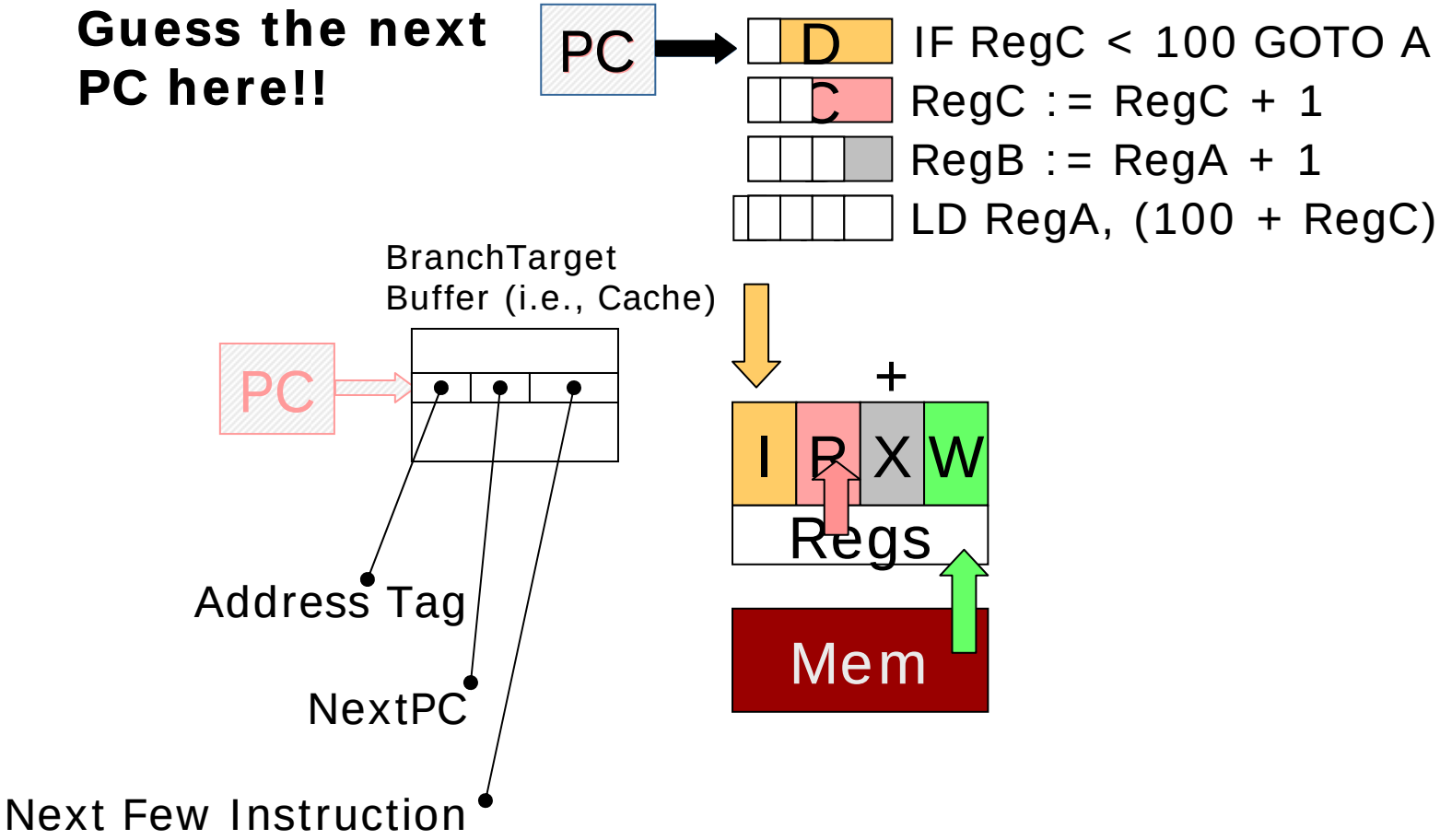




DARK2
2008

Cycle 4

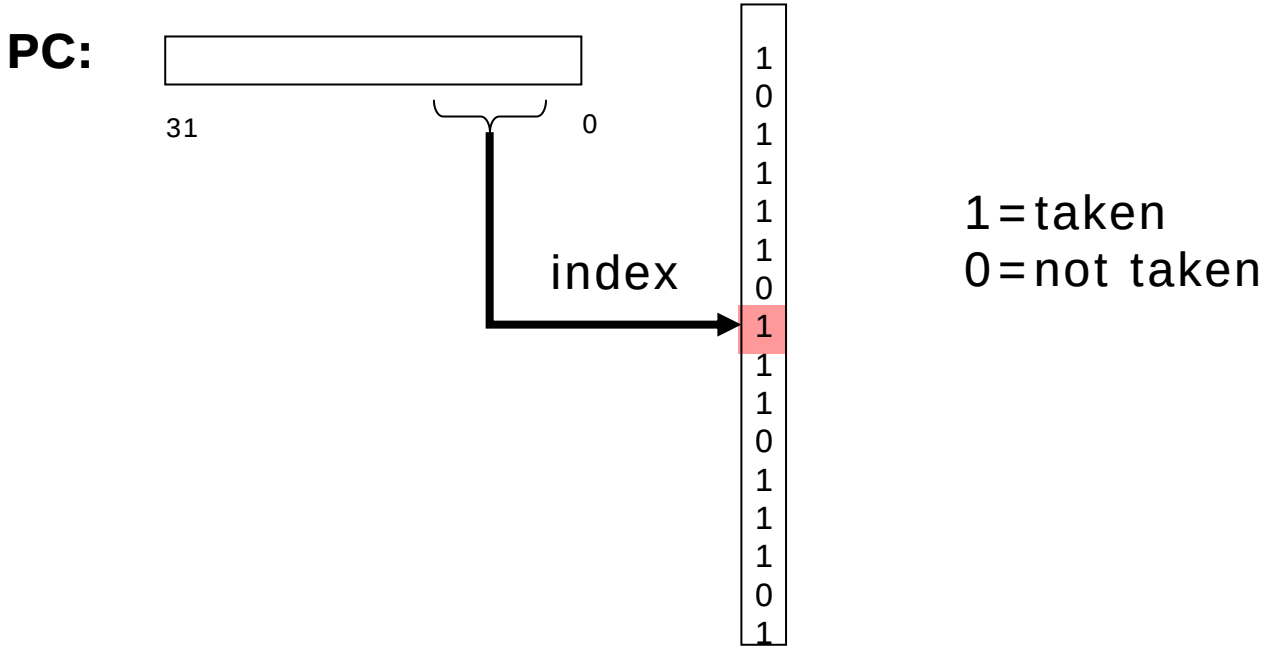
Guess the next PC here!!



UPPSALA UNIVERSITET

Branch history table

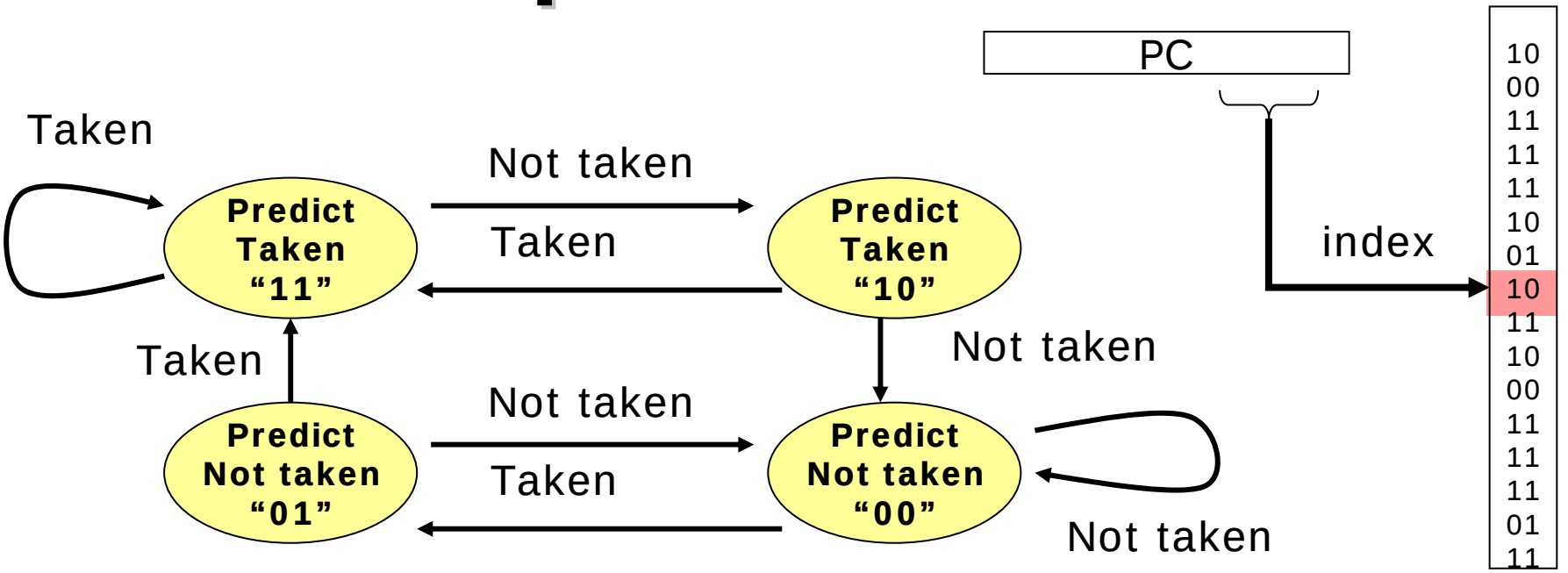
A simple branch prediction scheme



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

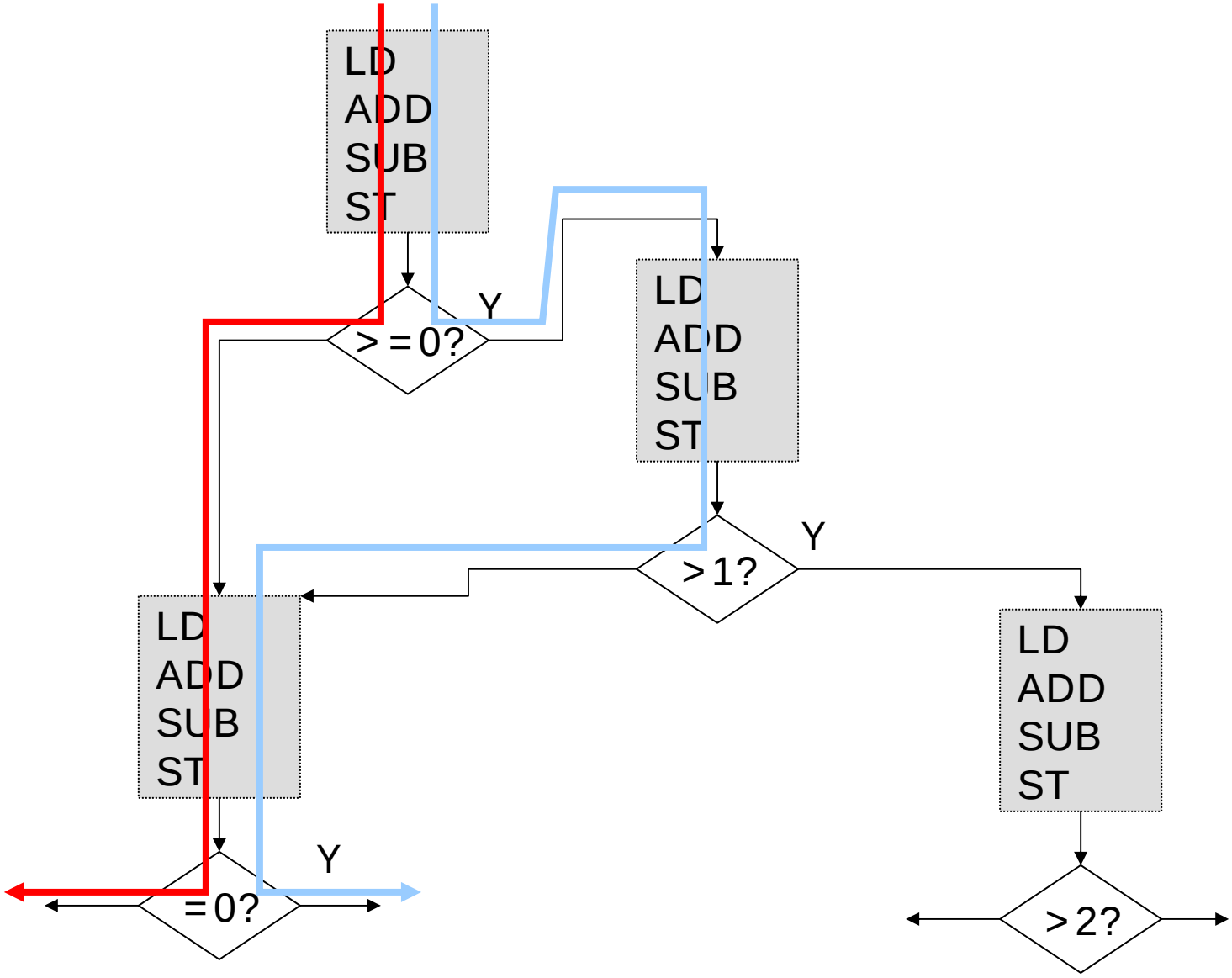
Problem: can cause two mispredictions in a row

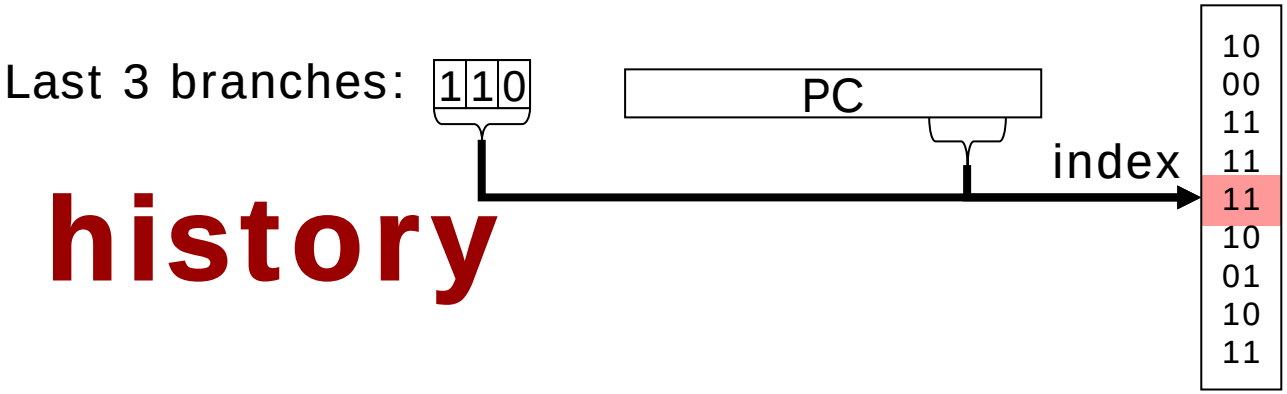
A two-bit prediction scheme



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

Dynamic Scheduling Of Branches





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

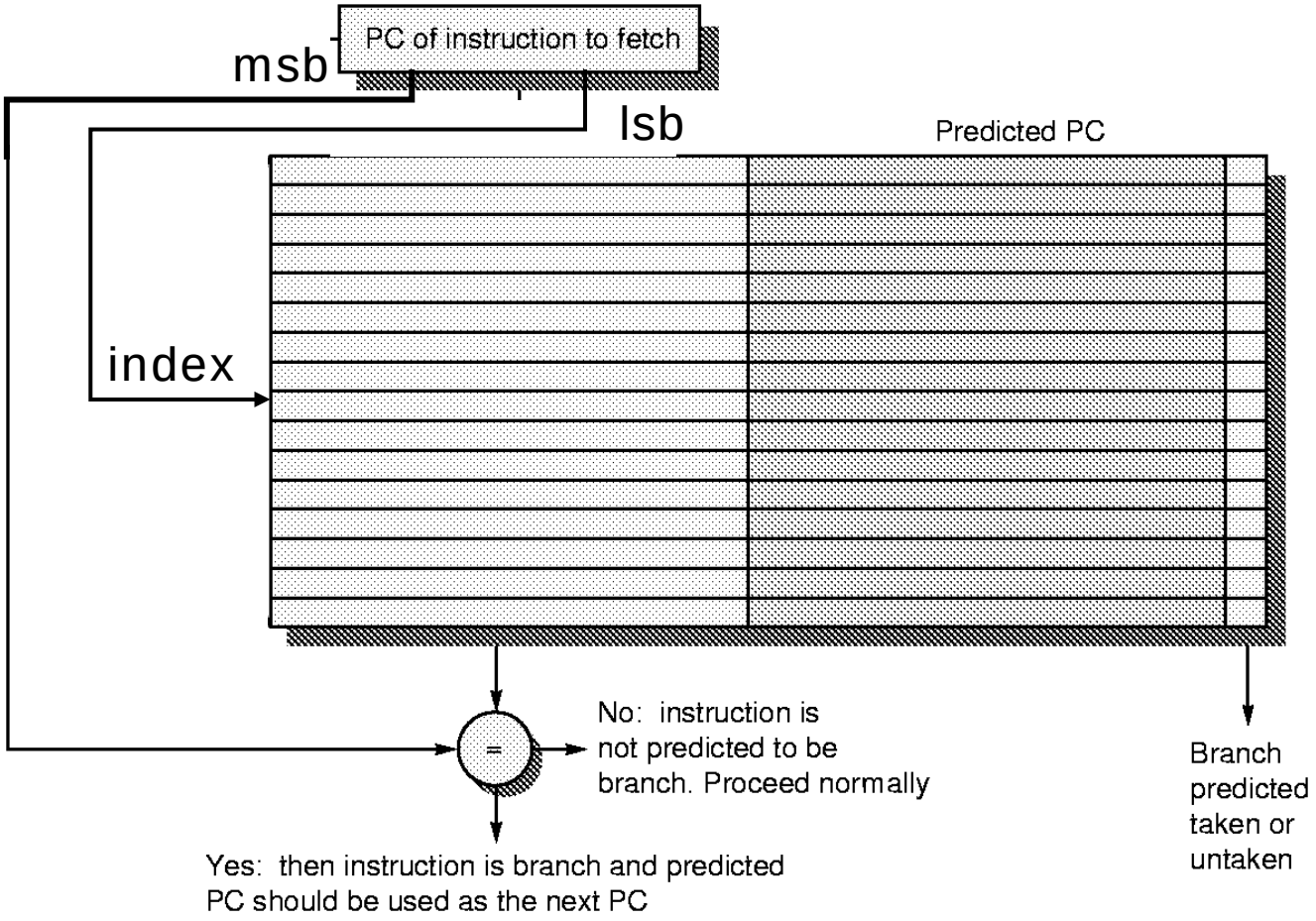
(N,M) prediction = N levels of M-bit prediction

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



- 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 stage highly integrated for branch optimizations

DARK2 2008

UPPSALA UNIVERSITET

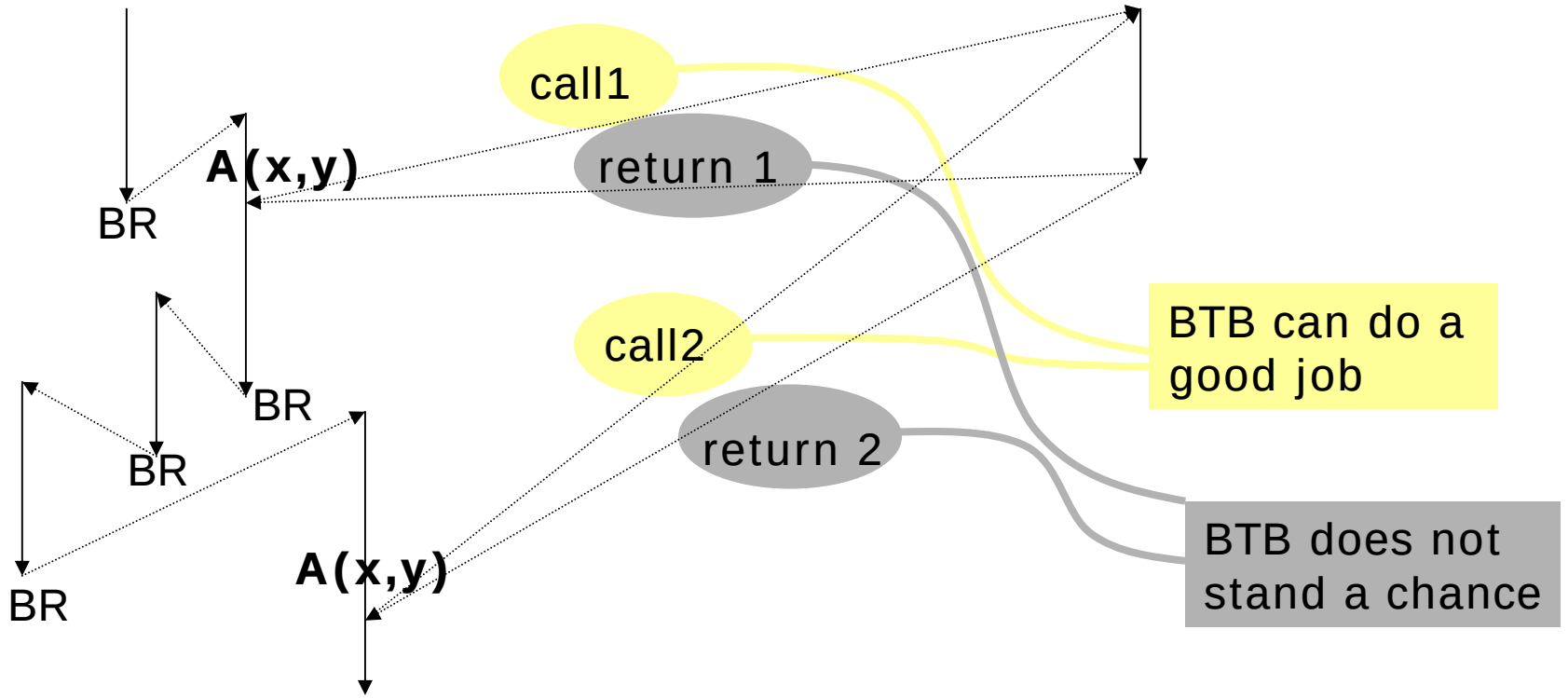
Folding branches

- BTB often 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)*
  - ”Branch folding”

DARK2 2008

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

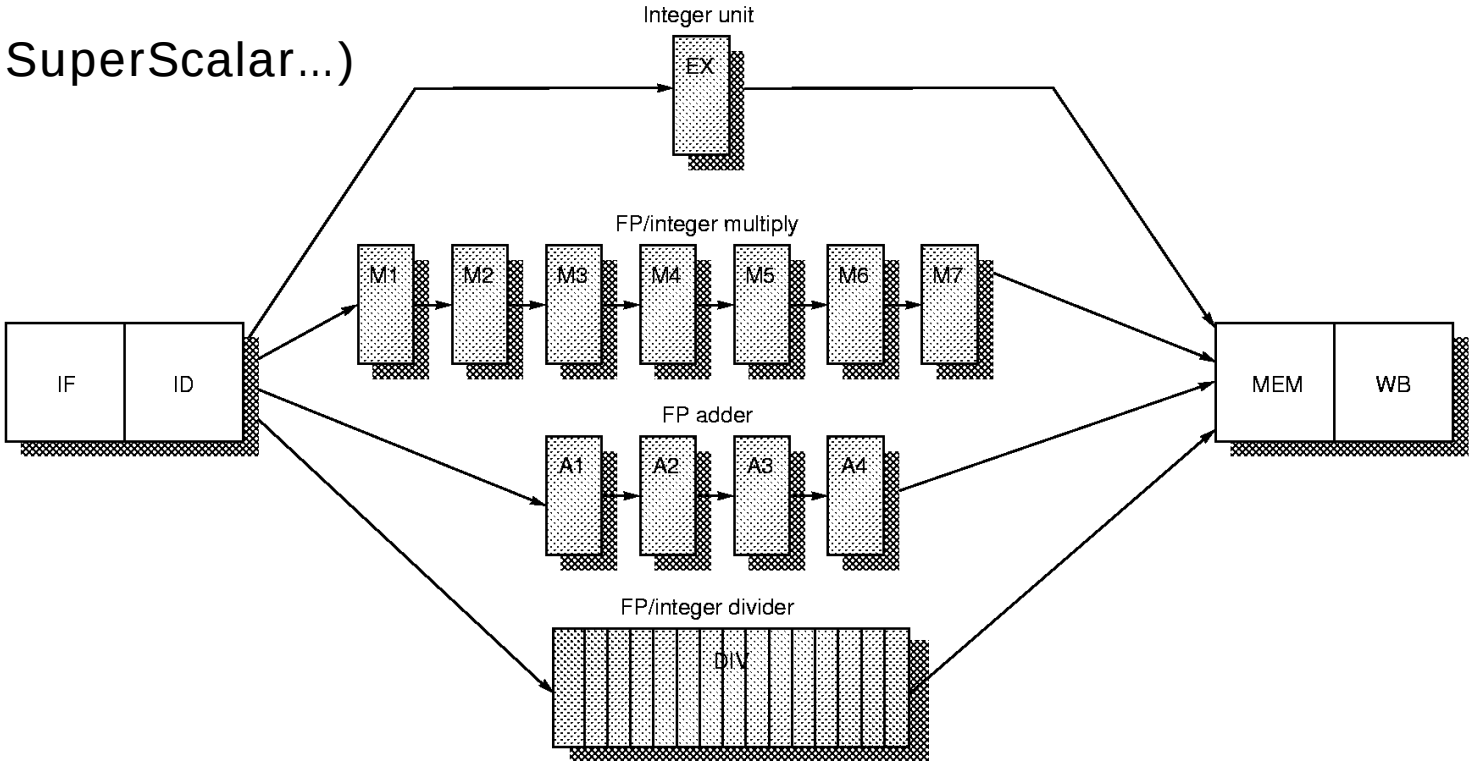
Overlapping Execution

Erik Hagersten
Uppsala University
Sweden

UPPSALA UNIVERSITET

Multicycle operations in the pipeline (floating point)

(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)

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:

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

UPPSALA UNIVERSITET

WAR and WAW hazards for multicycle operations

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

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

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

UPPSALA UNIVERSITET

Dynamic Instruction Scheduling

Key idea: allow subsequent independent instructions to proceed

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

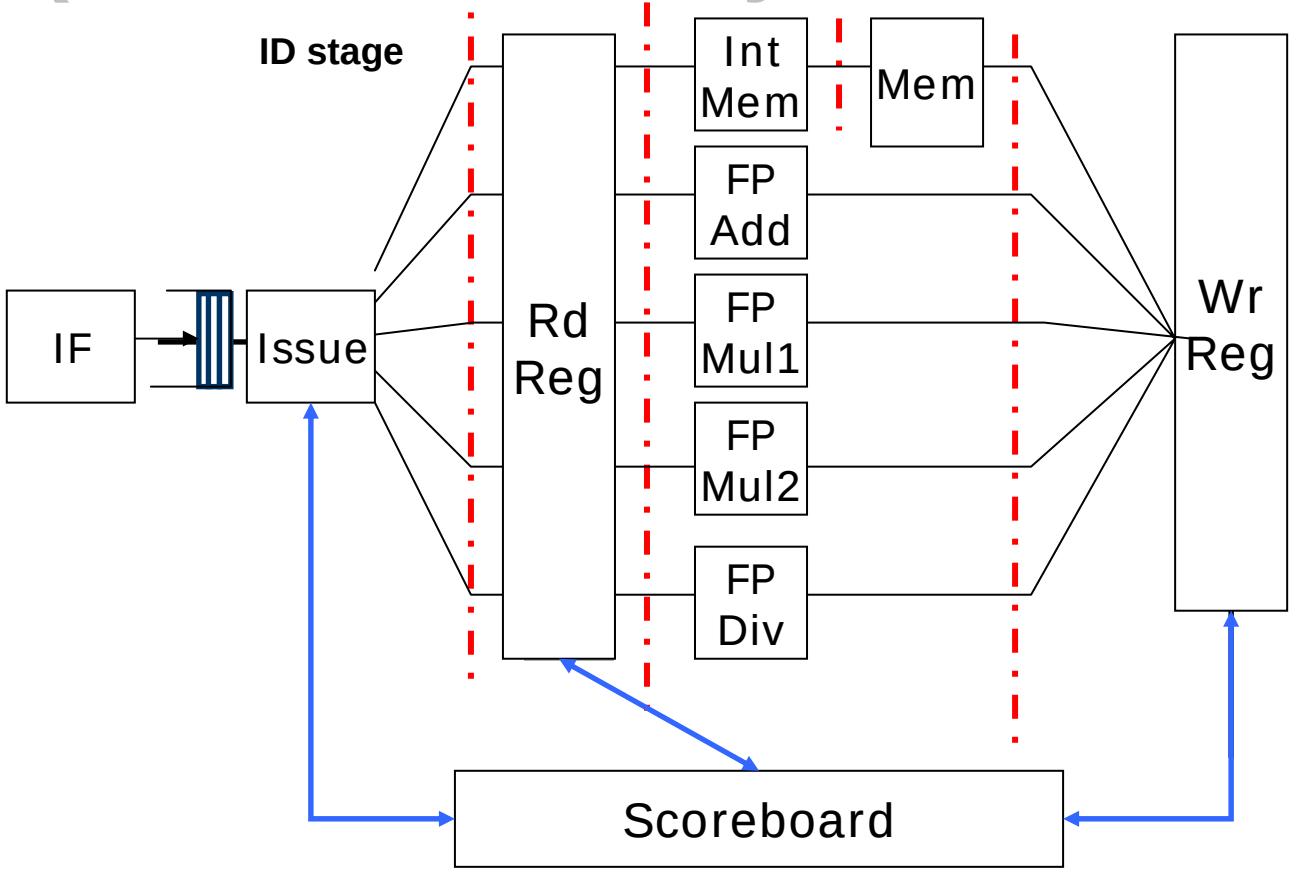
- Enables out-of-order execution (& out-of-order completion)

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 (covered briefly in this course)



- **Issue**: Decode and check for structural hazards
- **Read operands**: wait until no RAW hazard, then read operands (RAW)
- All data hazards are handled by the scoreboard mechanism

UPPSALA UNIVERSITET

Extended Scoreboard

Issue: Instruction is issued when:

No structural hazard for a functional unit

No WAW with an instruction in execution

Read: Instruction reads operands when they become available (RAW)

EX: Normal execution

Write: Instruction writes when all previous instructions have read or written this operand (WAW, WAR)

The scoreboard is updated when an instruction proceeds to a new stage

UPPSALA UNIVERSITET

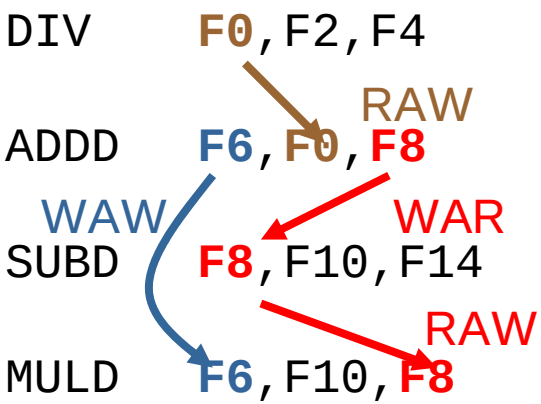
Limitations with scoreboards

The scoreboard technique is limited by:

- Number of scoreboard entries (*window size*)
- Number and types of functional units
- Number of ports to the register bank
- Hazards caused by name dependencies

Tomasulo's algorithm addresses the last two limitations

A more complicated example



WAR and WAW avoided through ”register renaming”

Register Renaming:

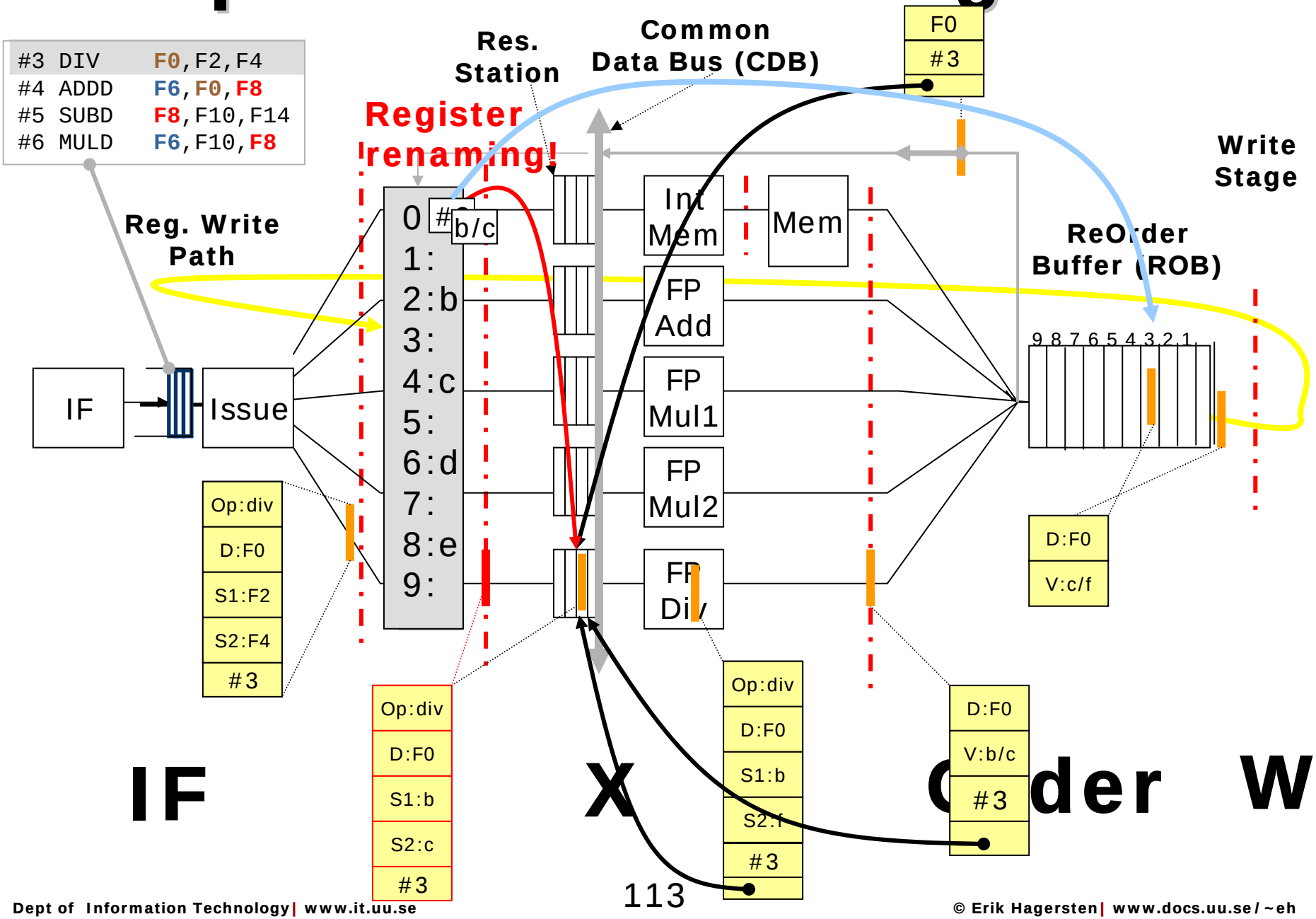
```
DIV    F0,F2,F4

ADDD   F6,F0,F8

SUBD   tmp1,F10,F14   ;can be executed right away

MULD   tmp2,F10,tmp1  ;delayed a few cycles
```

UPPSALA UNIVERSITET

Tomasulo's Algorithm

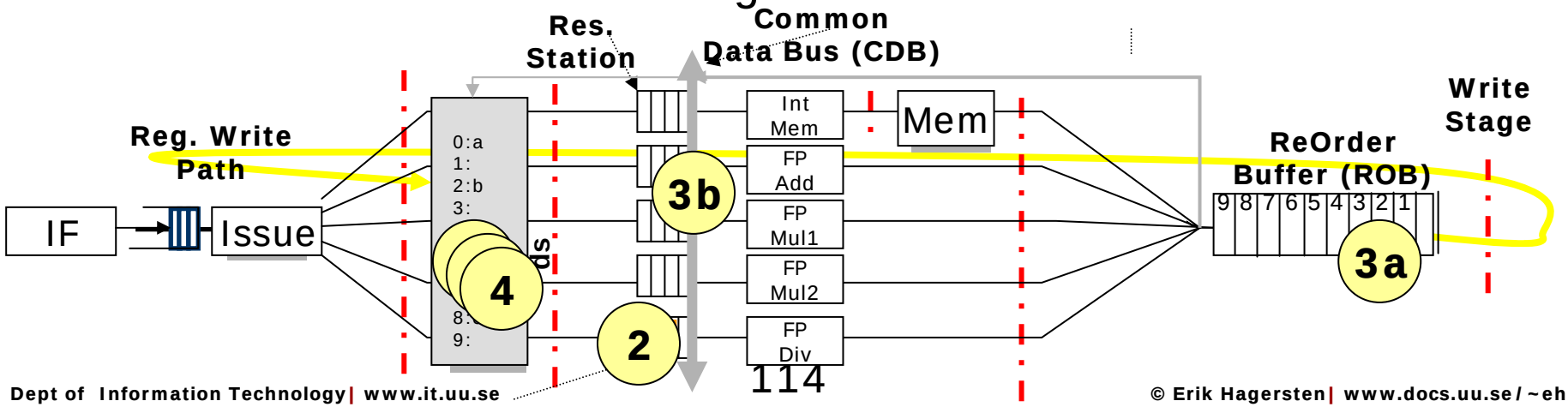
- IBM 360/91 mid 60's
- High performance without compiler support
- Extended for modern architectures
- Many implementations (PowerPC, Pentium…)

Simple Tomasulo's Algorithm

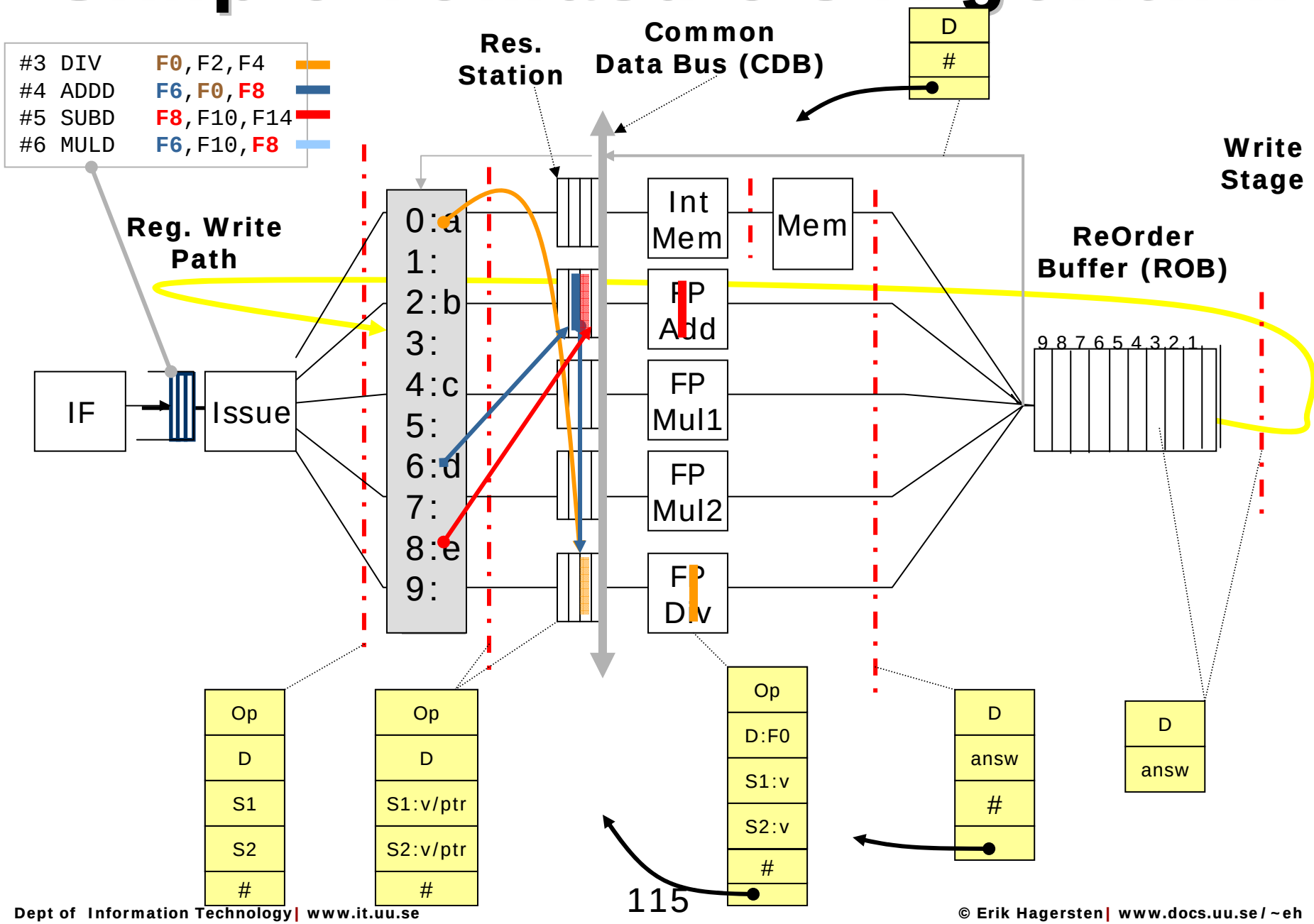


Tomasulo's: What is going on?

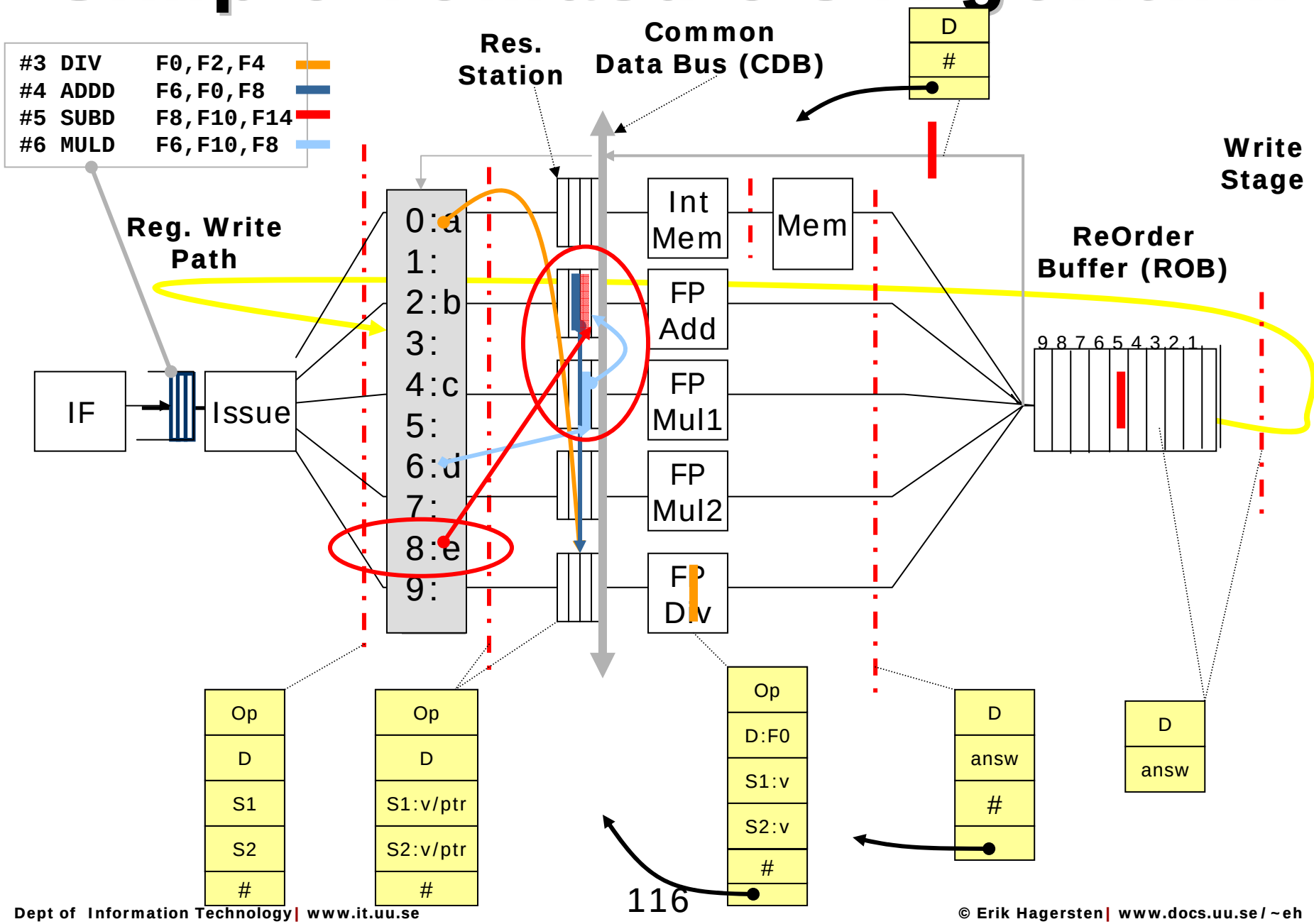
1. Read Register:
   - Rename DestReg to the Res. Station location
2. Wait for all dependencies at Res. Station
3. After Execution
   a) Put result in Reorder Buffer (ROB)
   b) Broadcast result on CDB to all waiting instructions
   c) Rename DestReg to the ROB location
4. When all preceeding instr.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



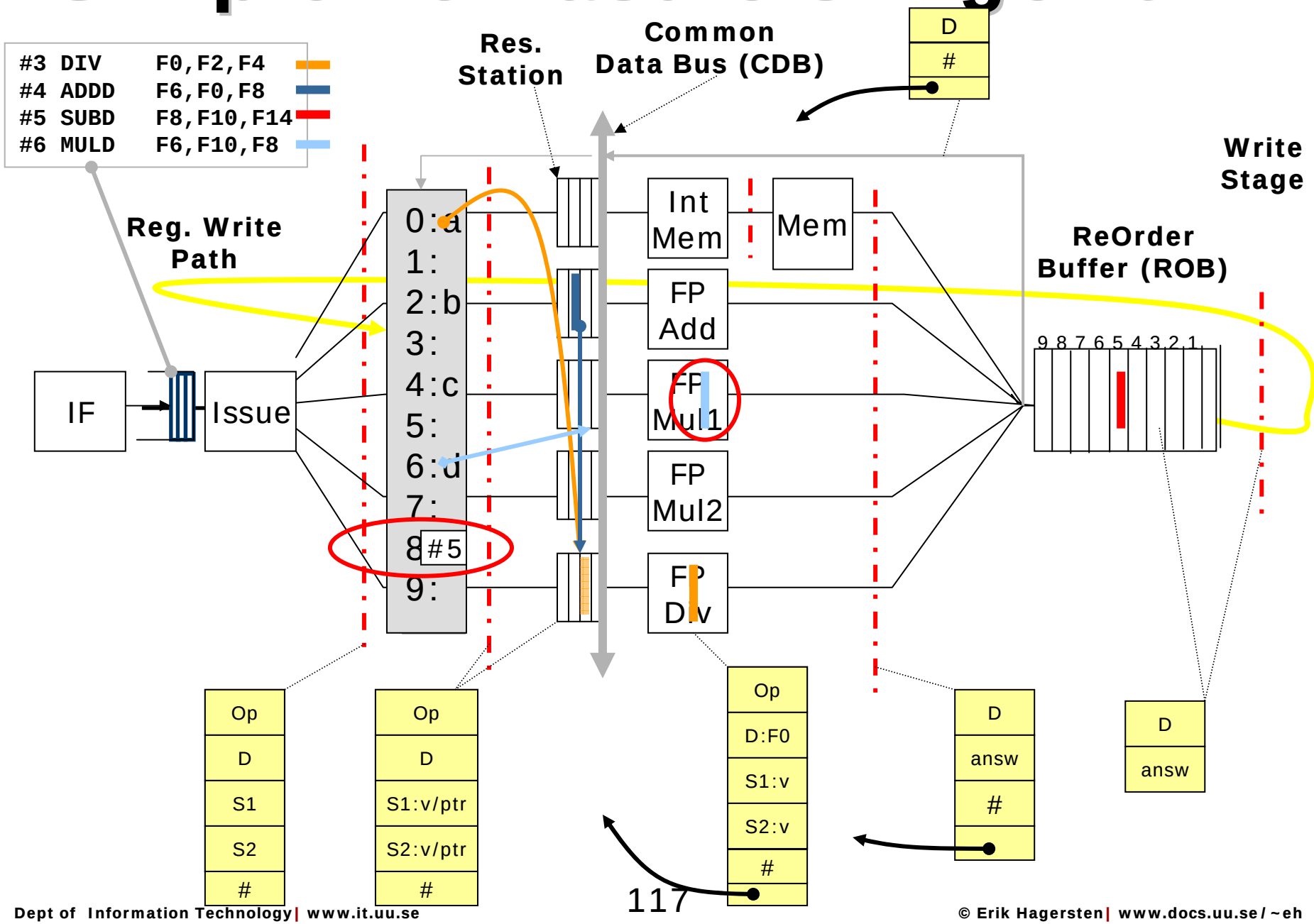
Simple Tomasulo's Algorithm



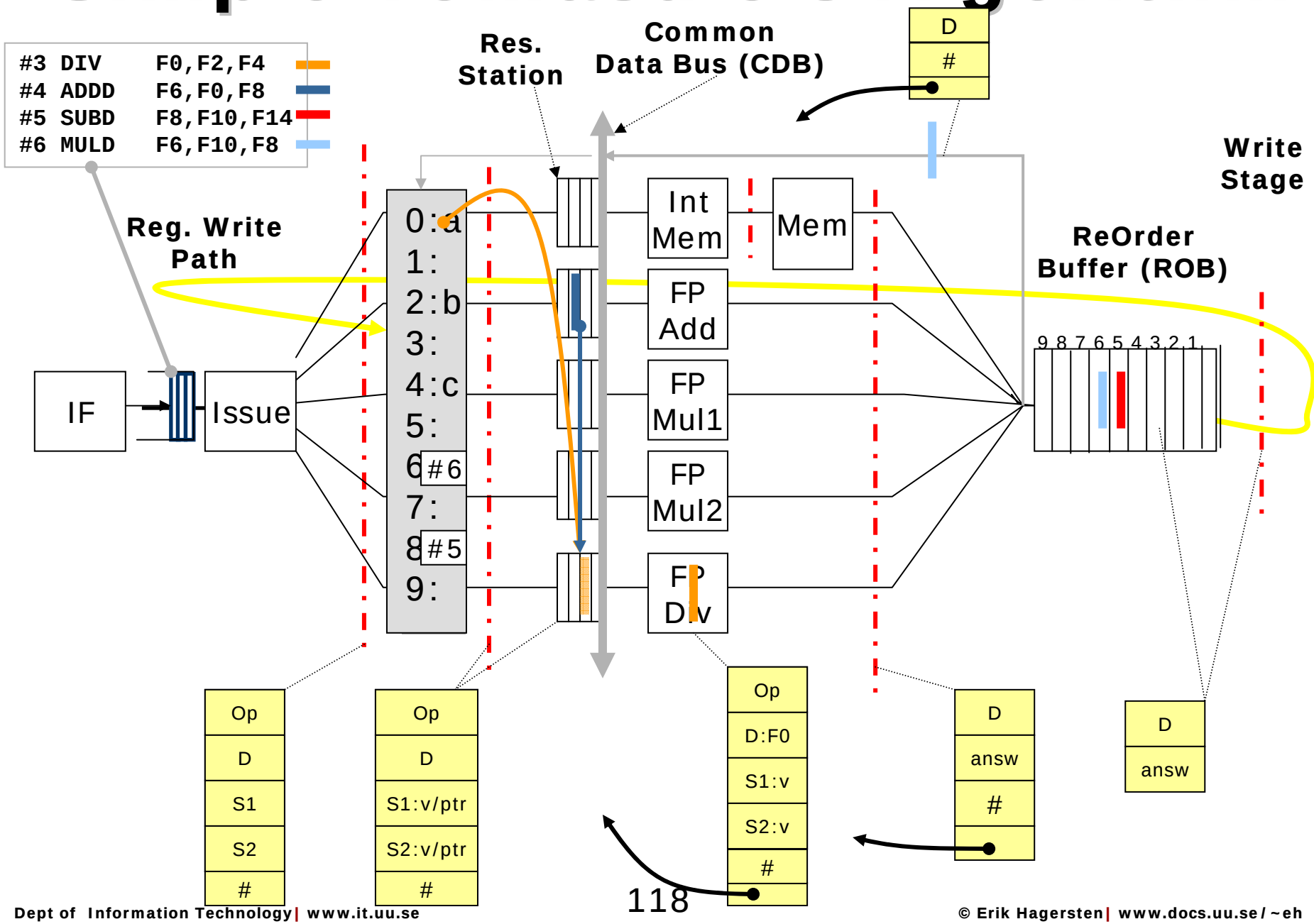
Simple Tomasulo's Algorithm



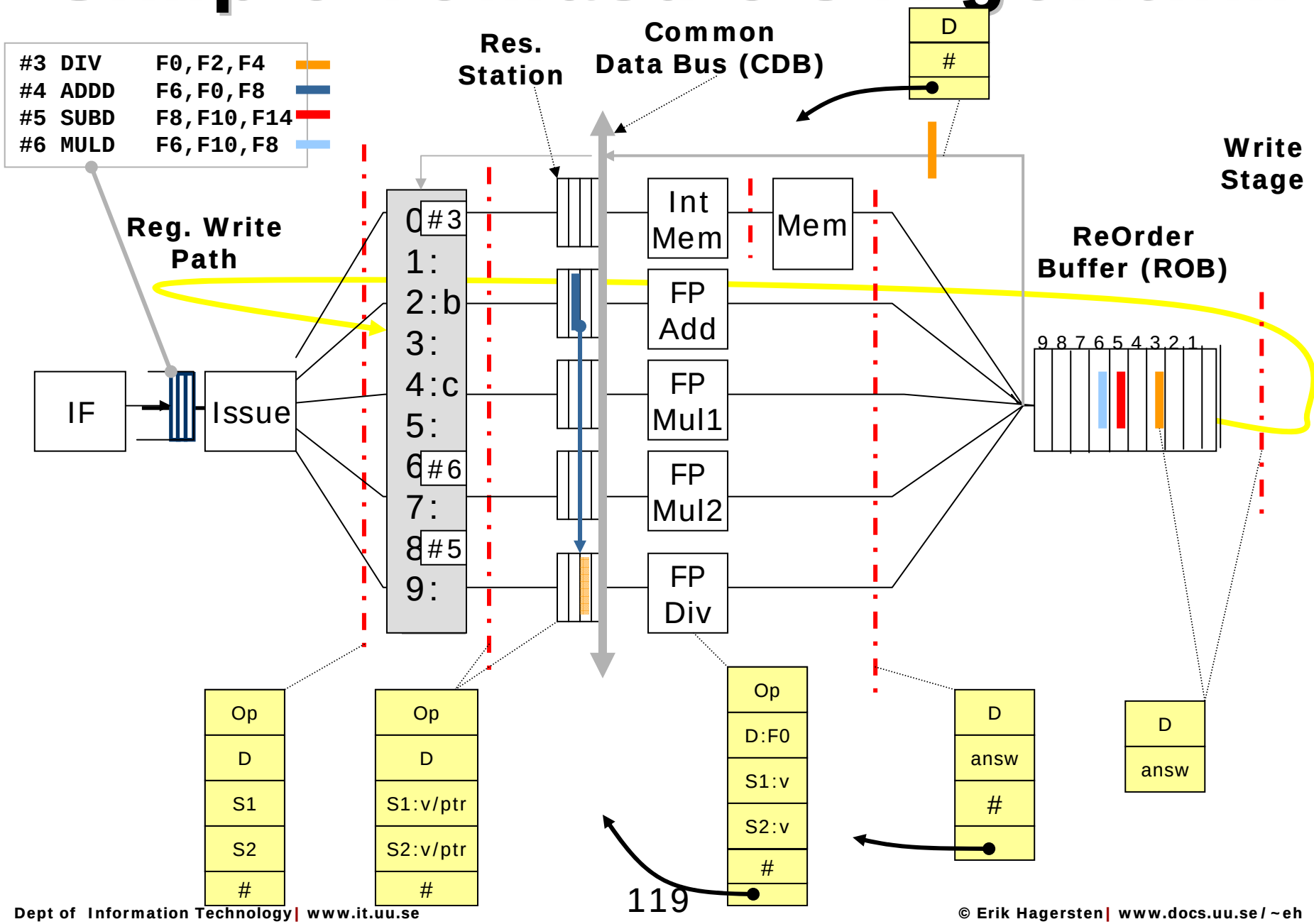
Simple Tomasulo's Algorithm



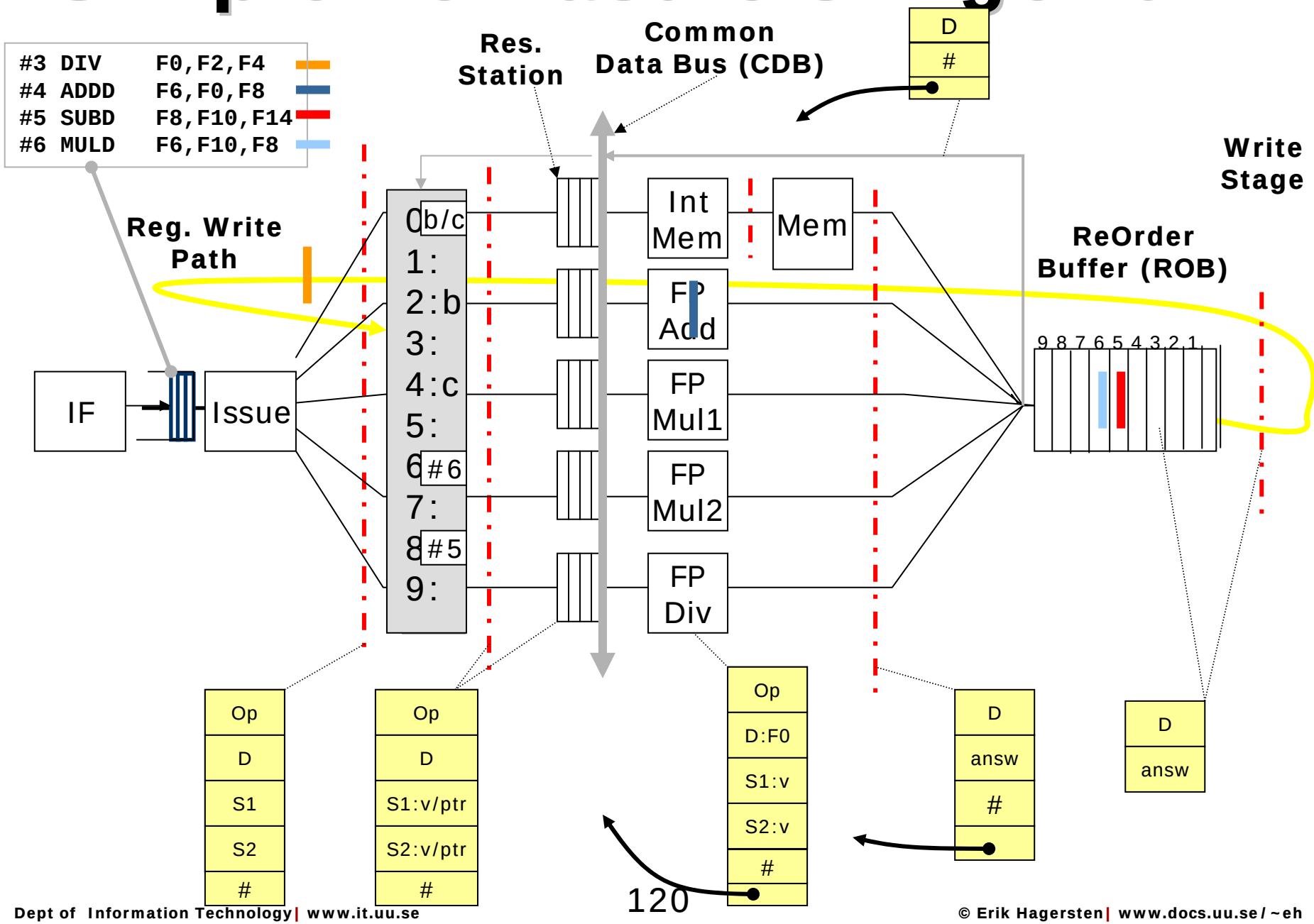
Simple Tomasulo's Algorithm



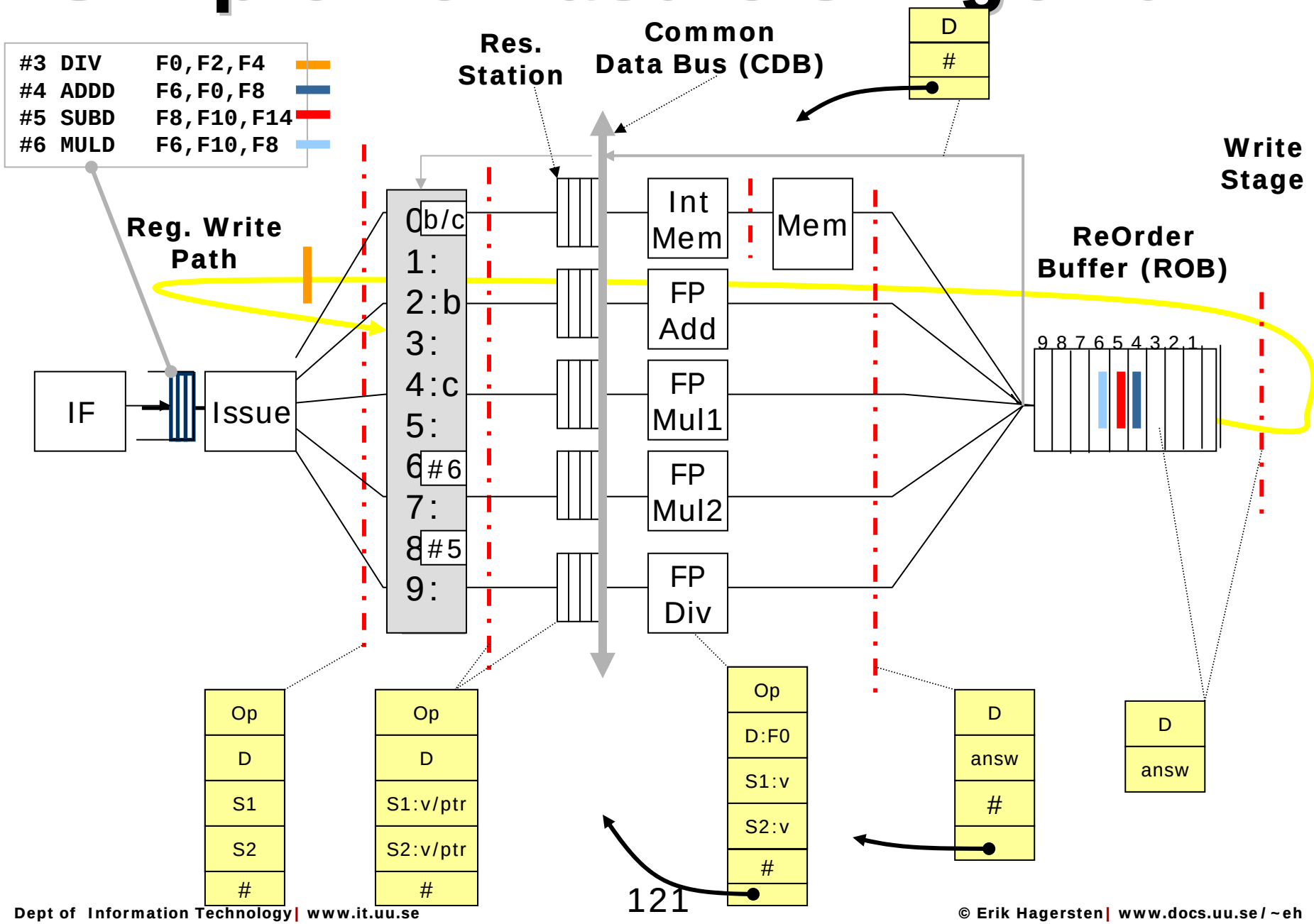
Simple Tomasulo's Algorithm



Simple Tomasulo's Algorithm

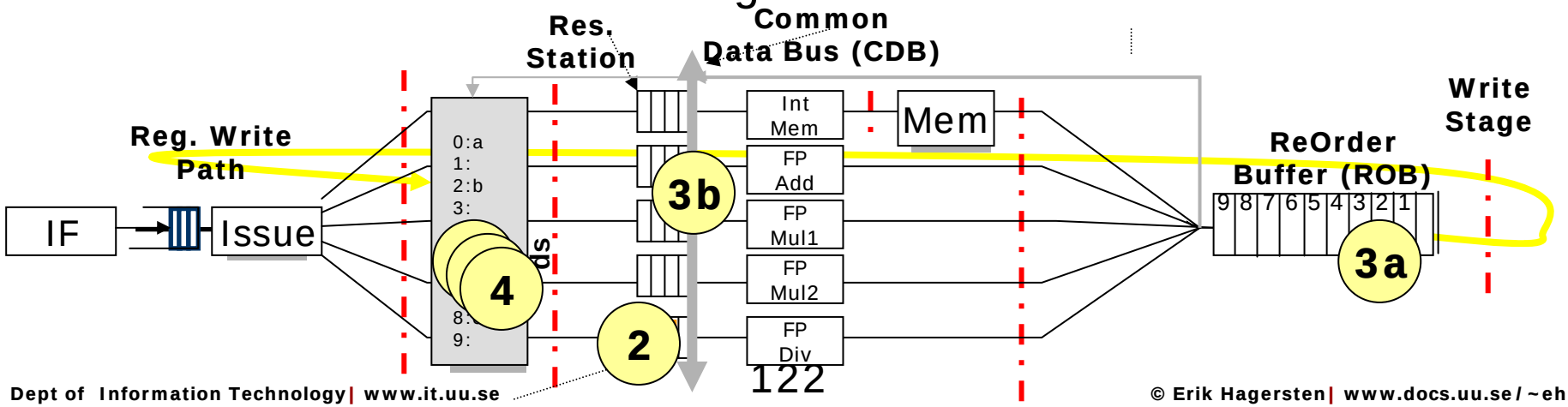


Simple Tomasulo's Algorithm

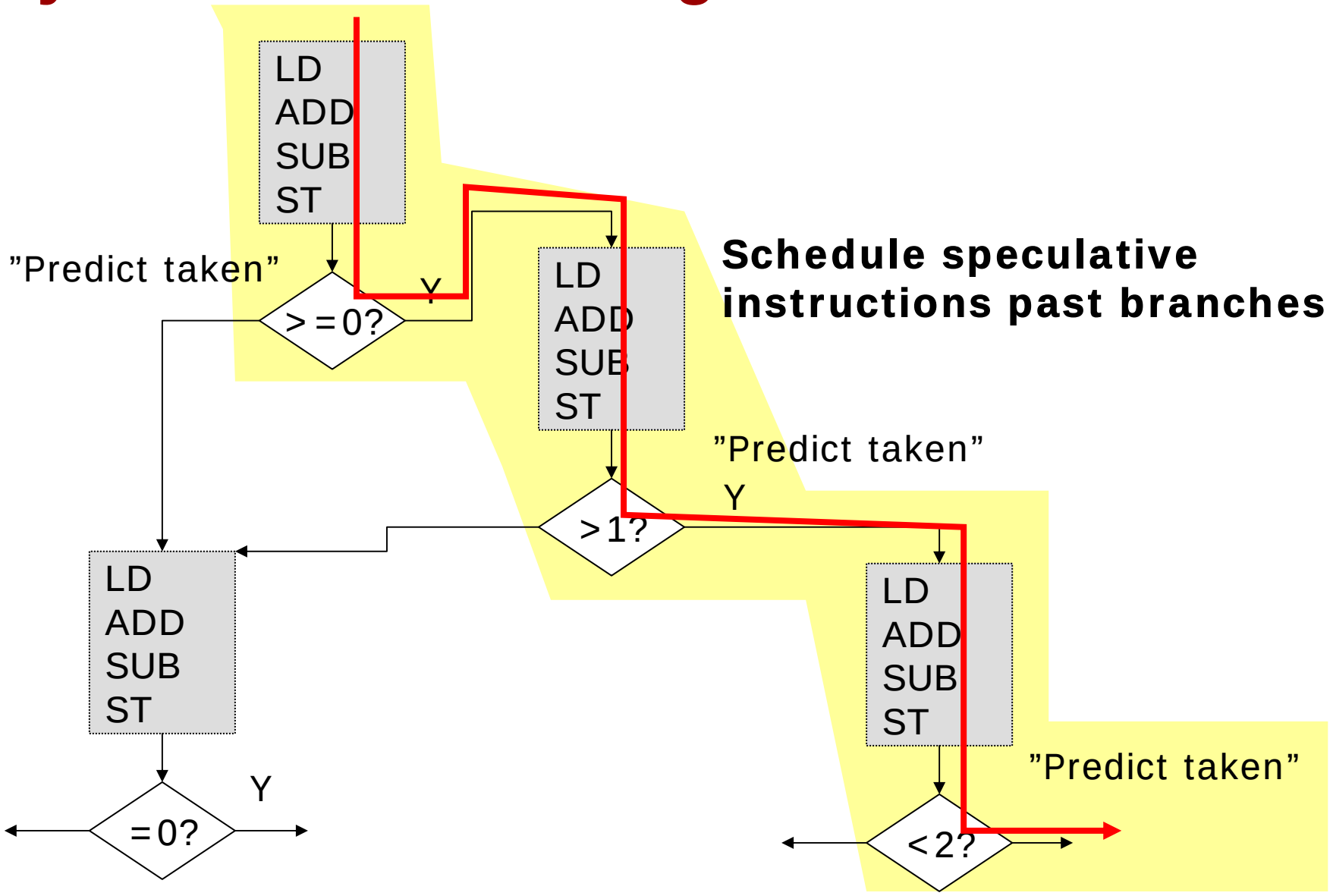


Tomasulo's: What is going on?

1. Read Register:
   - Rename DestReg to the Res. Station location
2. Wait for all 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

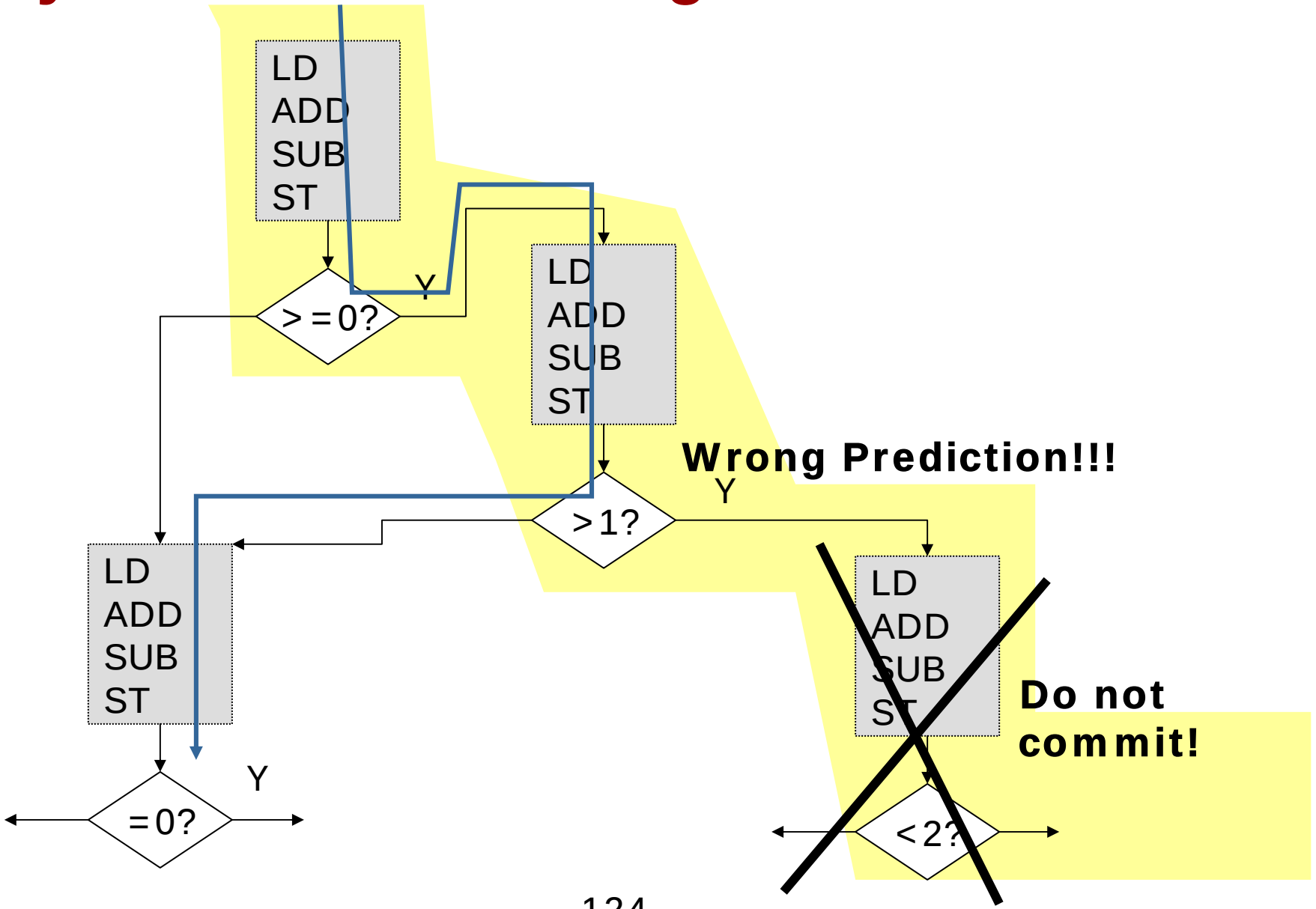


Dynamic Scheduling Past Branches



Schedule speculative instructions past branches

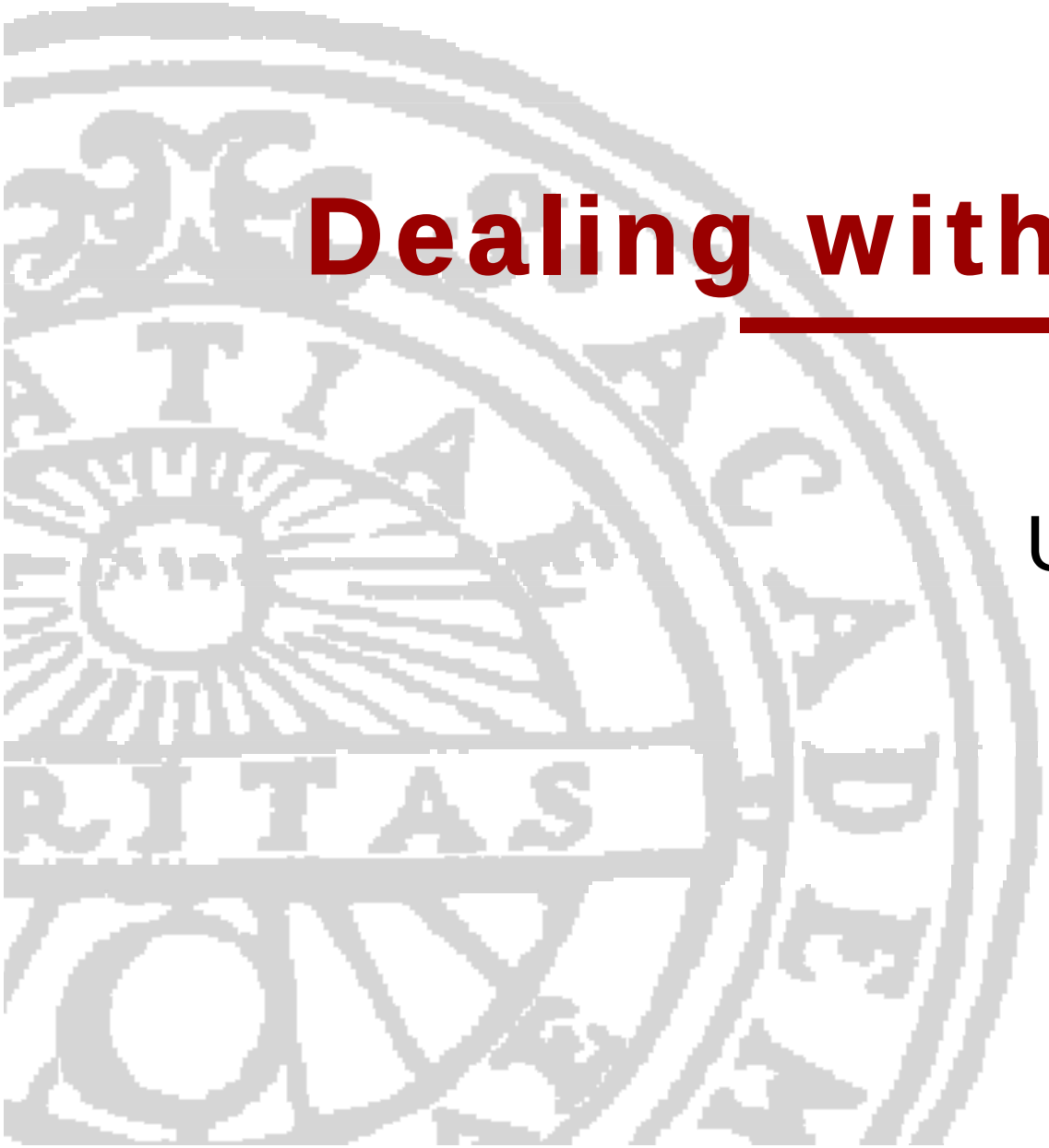
Dynamic Scheduling Past Branches



UPPSALA UNIVERSITET

Summing up Tomasulo's

- 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)
- "Register renaming" avoids WAW, WAR
- 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

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
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

Example sequence:

lw (e.g., page fault in MEM)

add (e.g., page fault in IF)

UPPSALA UNIVERSITET

FP Exceptions

Example:

```
DIVF  F0,F2,F4      24 cycles
ADDF  F10,F10,F8    3 cycles
SUBF  F12,F12,F14   3 cycles
```

SUBF may generate a trap before DIVF has completed!!

UPPSALA UNIVERSITET

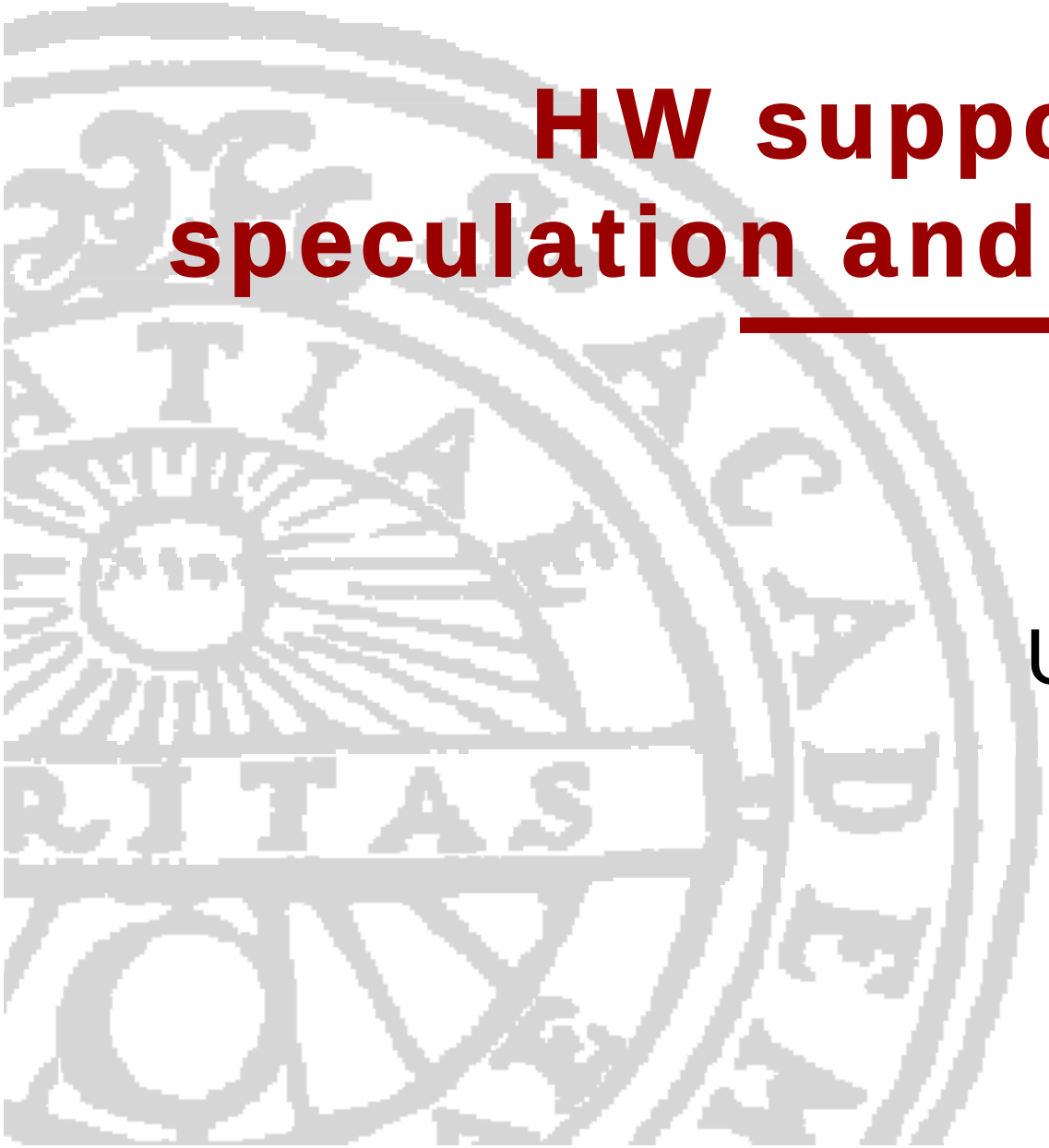
Revisiting Exceptions:

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

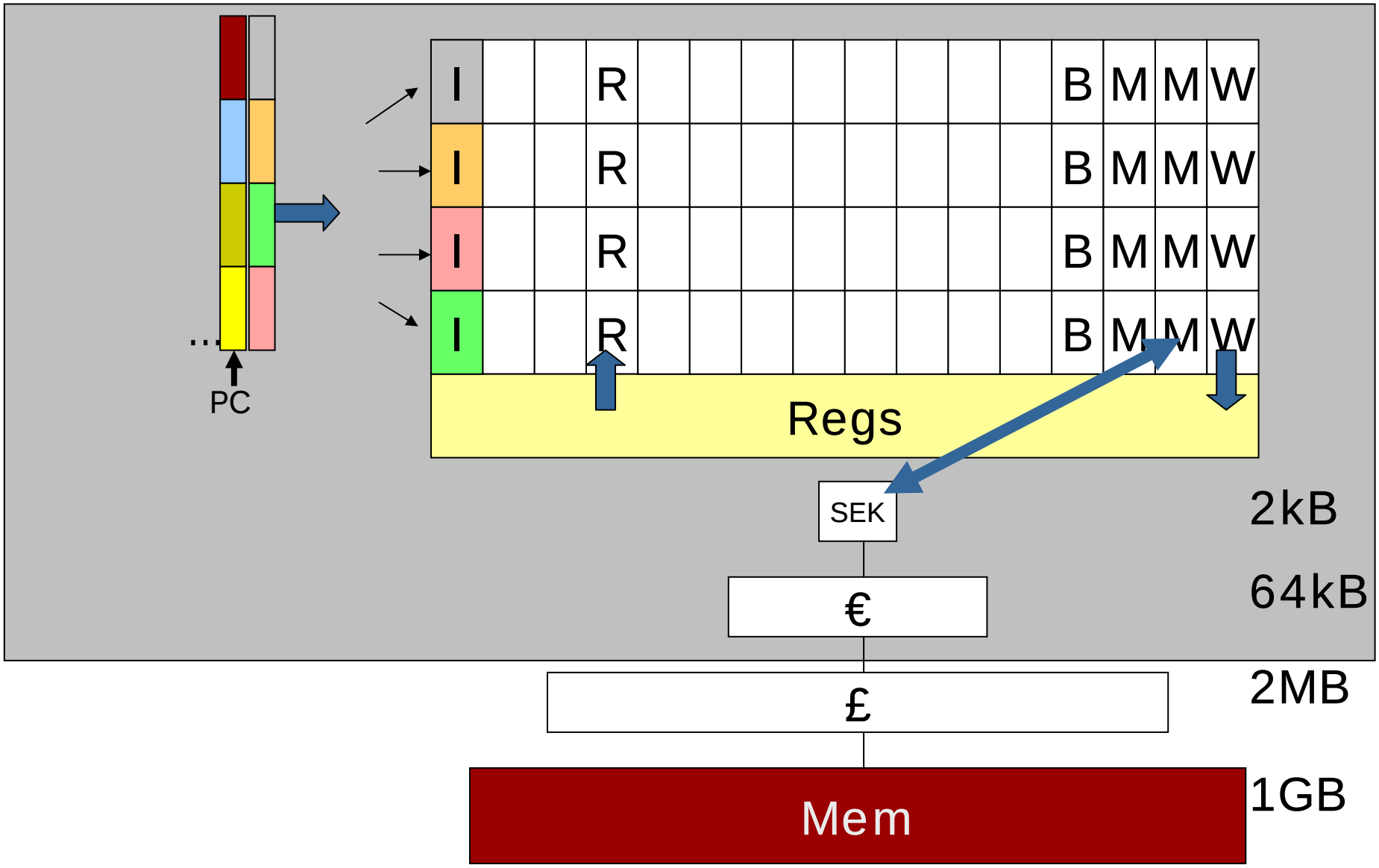
ROB helps the implementation in O-O-O execution



HW support for [static] speculation and improved ILP

Erik Hagersten
Uppsala University
Sweden

VLIW: Very Long Instruction Word



UPPSALA UNIVERSITET

Very Long Instruction Word (VLIW)

- Independent functional units with no hazard detection

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

UPPSALA UNIVERSITET

Limits to VLIW

Difficult to exploit parallelism

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

Memory and register bandwidth

Code size

No binary code compatibility

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!
- These techniques will allow larger moves and increase the effective size of a basic block
  - Removing branches: predicate execution
  - Move LD above ST: hazard detection
  - Move LD above branch: avoid false exceptions

Compiler speculation

The compiler moves instructions before a branch so that they can be executed before the branch condition is known

Advantage: creates longer schedulable code sequences => more ILP can exploited

Example: if (A == 0) then A = B; else A = A+4;

	Non speculative code				***Speculative code***	
	LW	**R1,0(R3)**	***Move past BR + reg rename***		**LW**	**R1,0(R3)**
	BNEZ	**R1,L1**			**LW**	**R14,0(R2)**
	LW	**R1,0(R2)**			**BEQZ**	**R1,L3**
	J	**L2**			**ADD**	**R14,R14,4**
L1:	**ADD**	**R1,R1,4**		**L3:**	**SW**	**0(R3),R14**
L2:	**SW**	**0(R3),R1**				

- What about exceptions?

UPPSALA UNIVERSITET

Speculative instructions

Moving a LD up, may make it *speculative*

- Moving past a branch
- Moving past a ST (that may be to the same address)

Issues:

- Non-intrusive
- Correct exception handling (again)
- Low overhead
- Good prediction

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

UPPSALA UNIVERSITET

Example: 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

- Removes the need for some branches ☺
- 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)
    ```

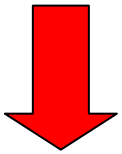
  - Avoiding a branch makes the basic block larger!!!

    ➔ More instructions for the code scheduler to play with
- 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

DARK2 2008

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

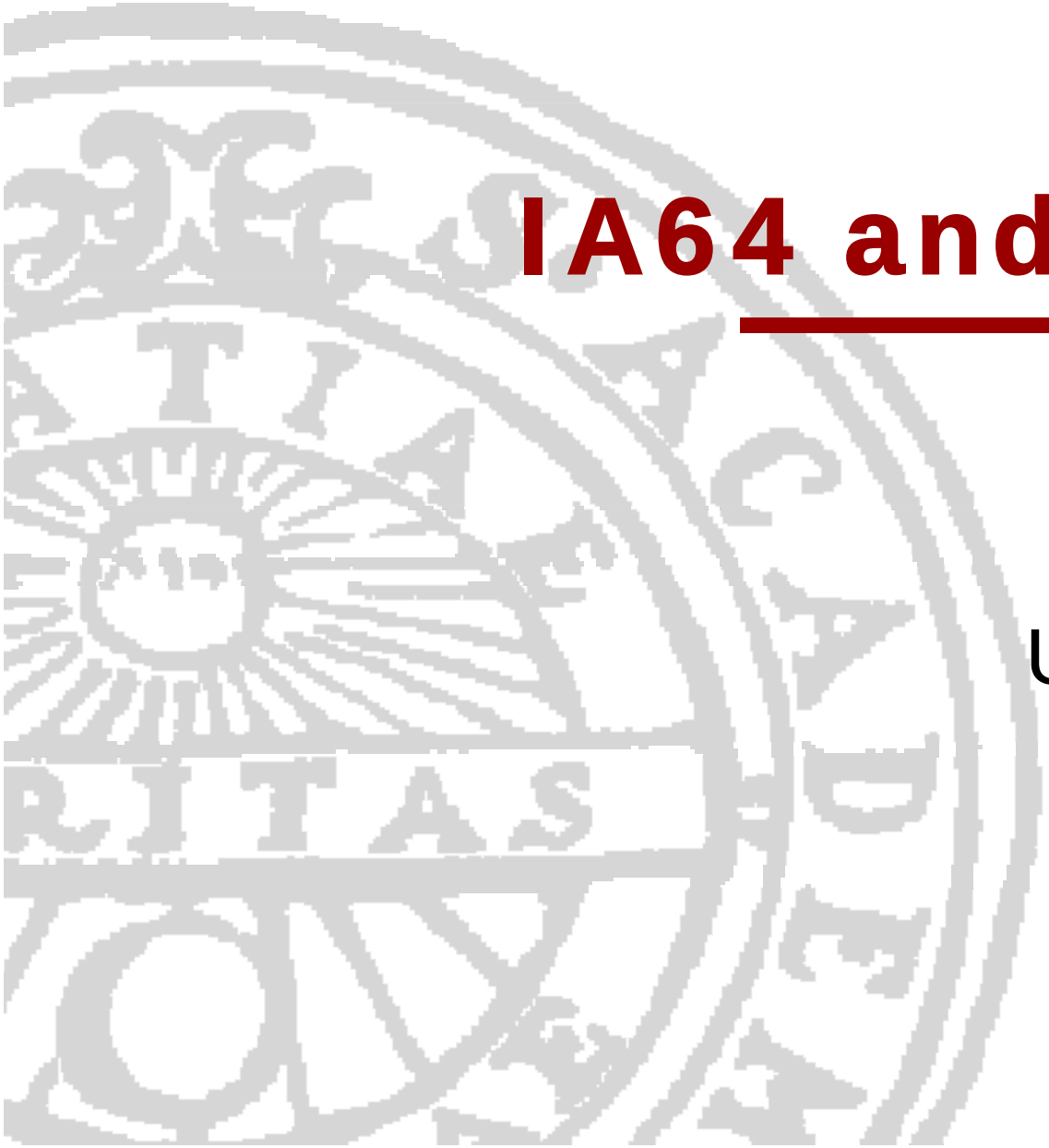
HW vs. SW speculation

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)



Example: IA64 and Itanium(I)

Erik Hagersten
Uppsala University
Sweden

UPPSALA UNIVERSITET

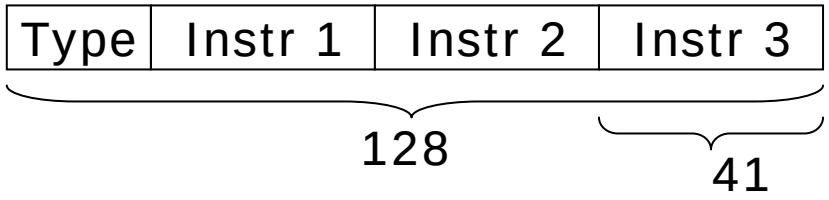
Little of everything

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

DARK2 2008

UPPSALA UNIVERSITET

Itanium instructions



- 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
I-LD	1
FP-LD	9
Pred branch	0-3
Misspred branch	0-9
I-ALU	0
FP-ALU	4

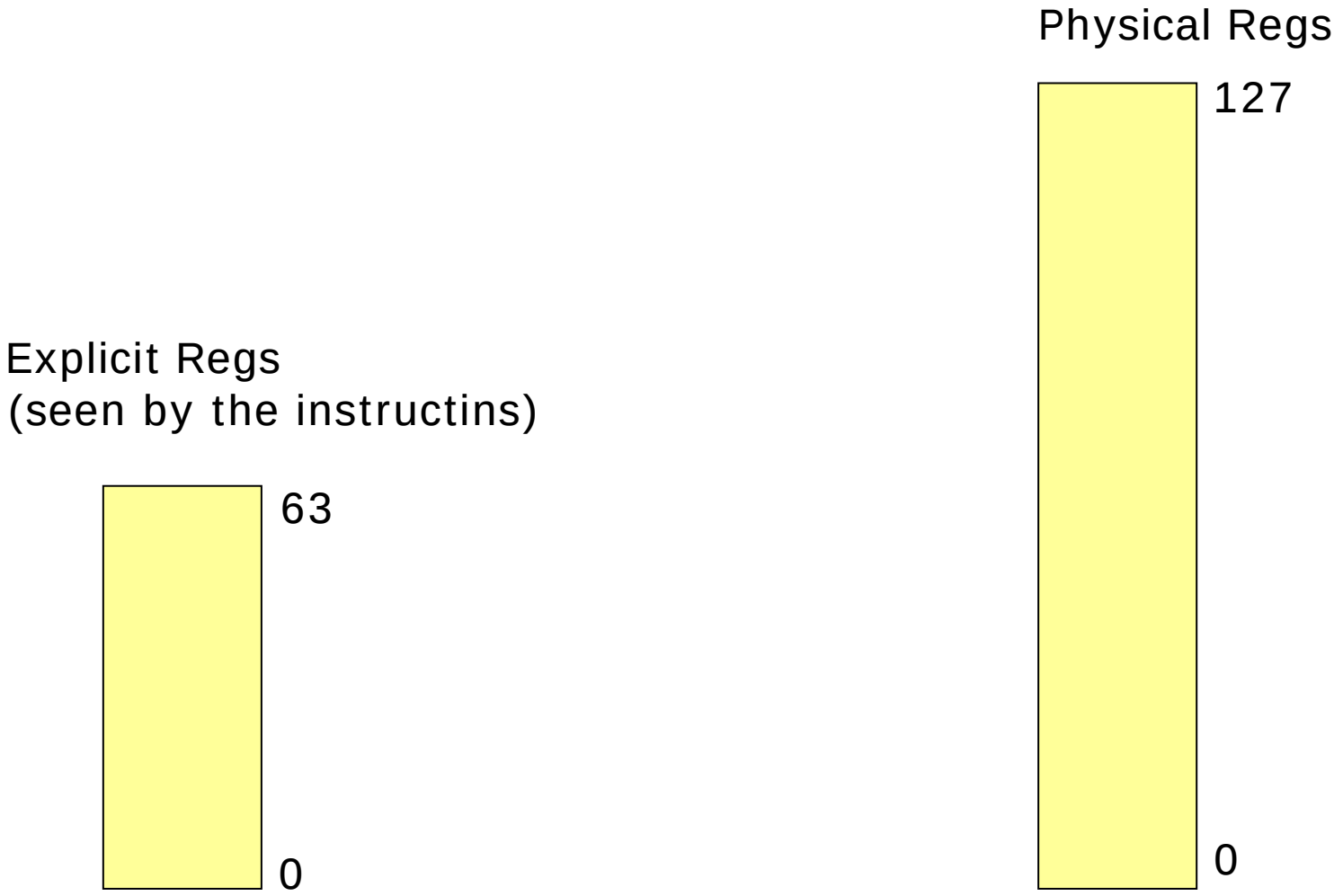
UPPSALA UNIVERSITET

Itanium Registers

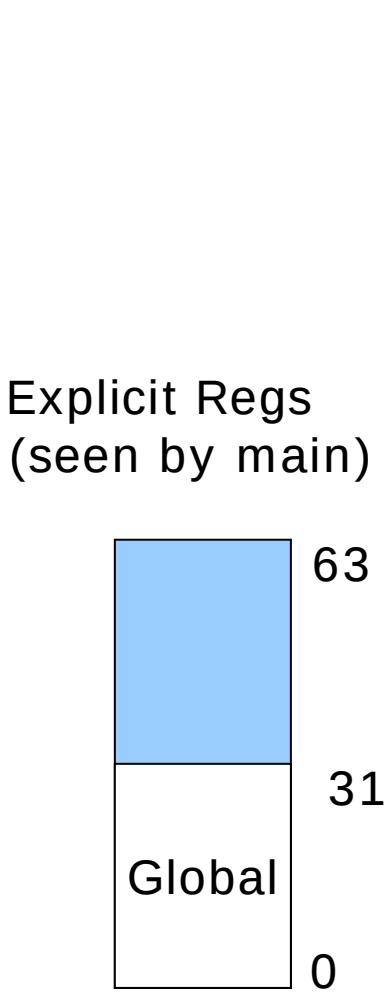
- 128 65-bit GPR (w/ poison bit)
- 128 82-bit FP REGS
- 64 1-bit predicate REGS
- A bunch of CSRs (control/status registers)

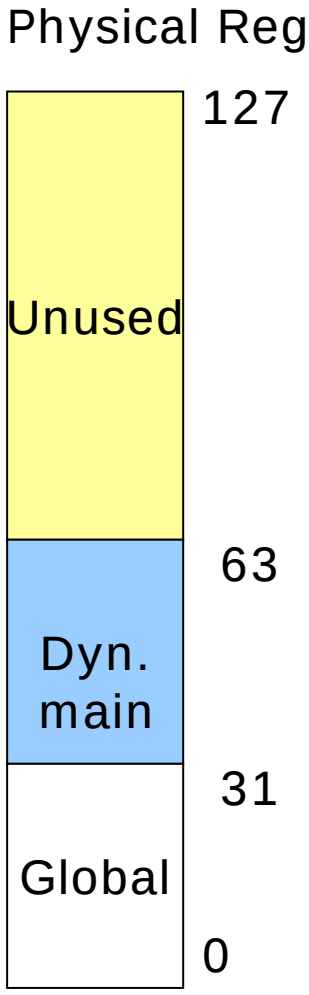
UPPSALA
UNIVERSITET

Dynamic register window

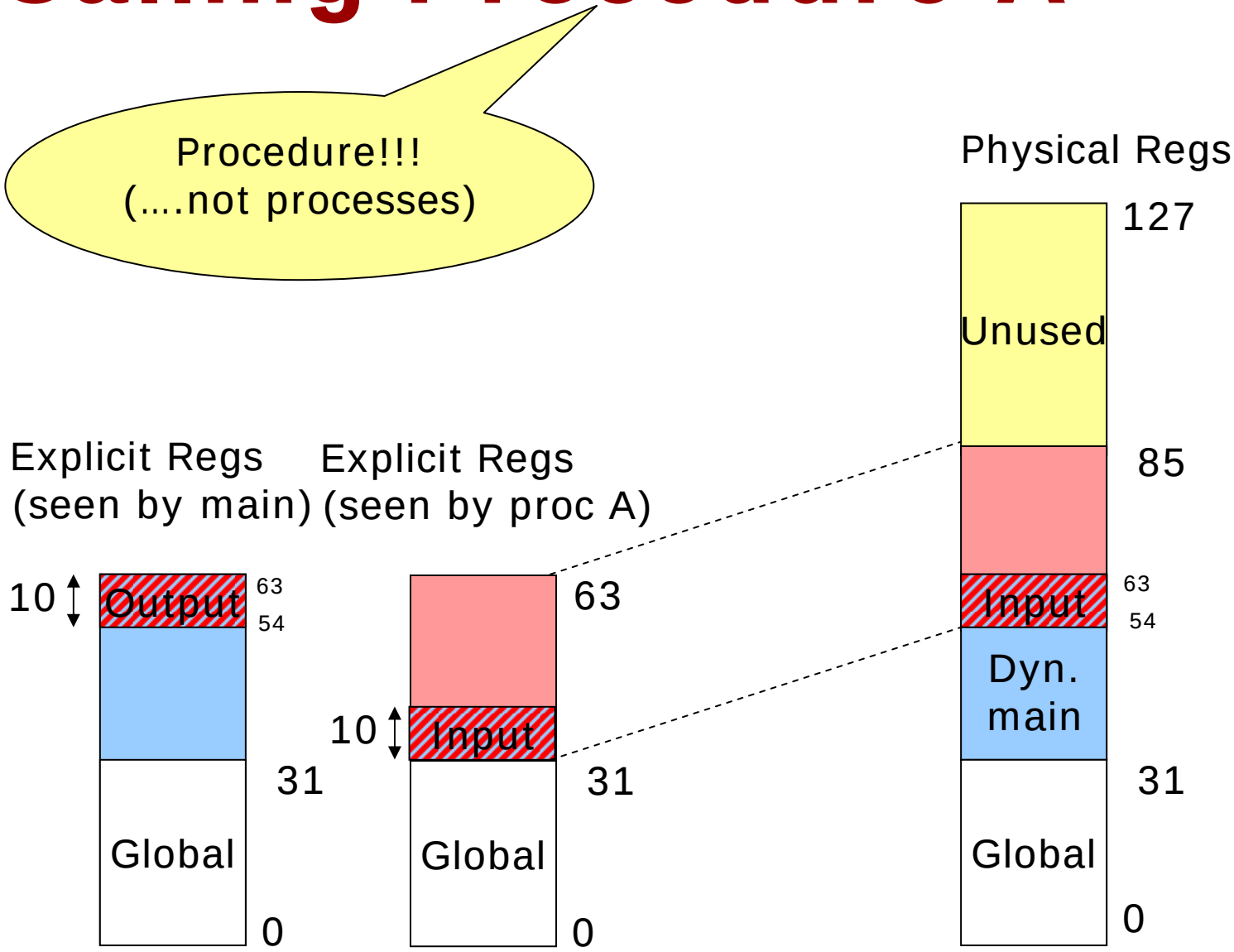


Dynamic register window for GPRs

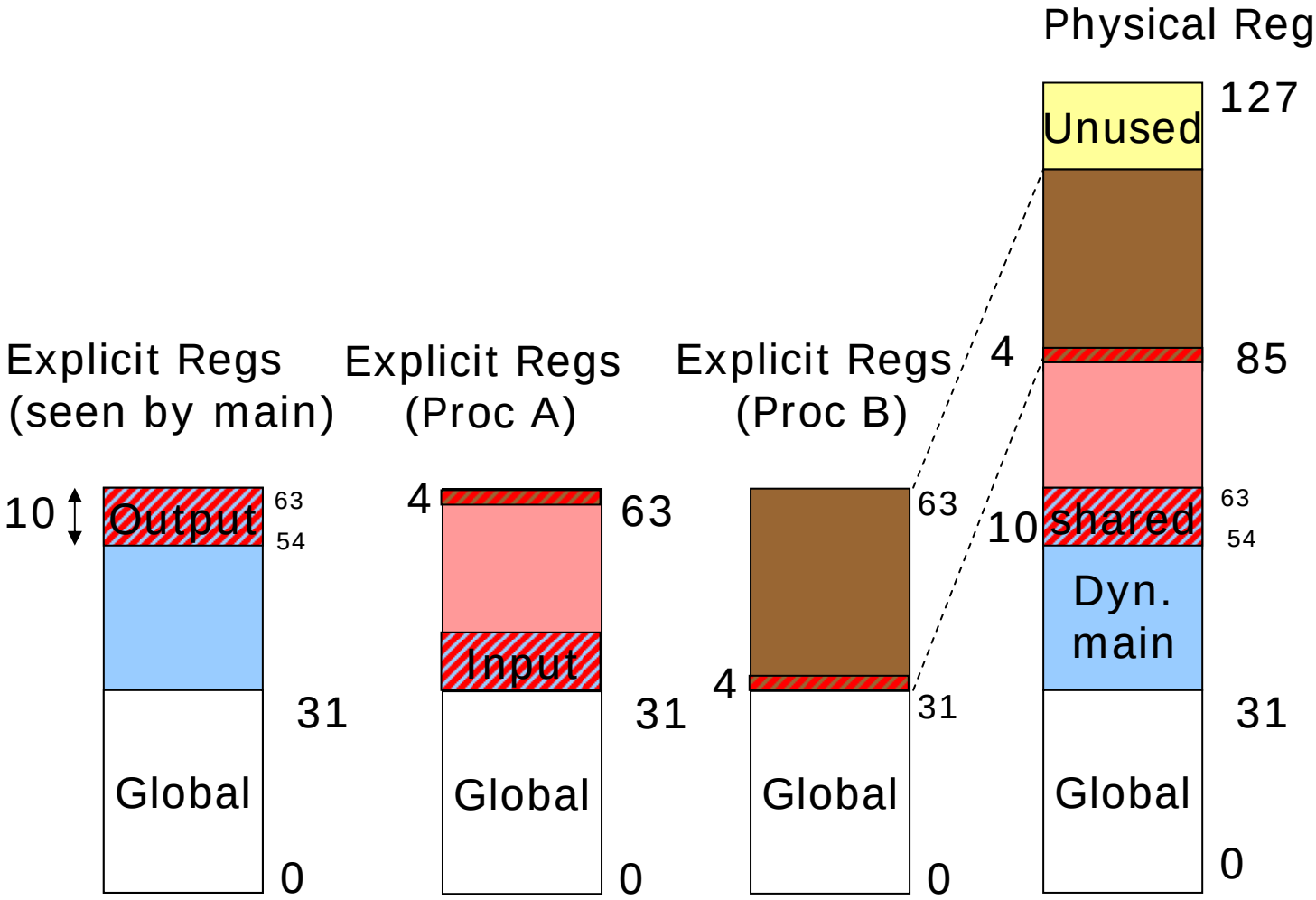




Calling Procedure A



Calling Procedure B (automatic passing of parameters)



151

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

- This is similar to SPARC and UCB's RISC

UPPSALA UNIVERSITET

Register rotation: FP and GPRs

- Used in software pipelining
- Register renaming for each iteration
- Removes the need for prologue/epilogue
- RSE (register stack engine)

UPPSALA UNIVERSITET

What is the alternative?

- VLIW was meant to simplify HW
- Itanium has 230 M transistors and consumes 130W?
- Will it scale with technology?
- Other alternatives:
  - Increase cache size,
  - Increase the frequency, or,
  - Run more than one thread/chip (More about this during “Future Technologies”)