LINKÖPINGS UNIVERSITET

# Code Generation for RISC and Instruction-Level Parallel Processors

**RISC/ILP Processor Architecture Issues**

**Instruction Scheduling**

**Register Allocation**

**Phase Ordering Problems**

**Integrated Code Generation**

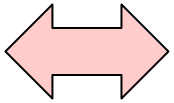
LINKÖPINGS UNIVERSITET

# 1. RISC and Instruction-Level Parallel Target Architectures

# CISC vs. RISC

**CISC**

- Complex Instruction Set Computer
- Memory operands for arithmetic and logical operations possible
- M(r1+r2) ← M(r1+r2) * M(r3+disp)



- Many instructions
- Complex instructions
- Few registers, not symmetric
- Variable instruction size
- Instruction decoding (often done in microcode) takes much silicon overhead
- Example: 80x86, 680x0

**RISC**

- Reduced Instruction Set Computer
- Arithmetic/logical operations only on registers
- add r1, r2, r1
  load (r1), r4
  load r3+disp, r5
  mul r4, r5
  store r5, (r1)
- Fewer, simple instructions
- Many registers, all general-purpose typically, 32 ... 256 registers
- Fixed instruction size and format
- Instruction decoding hardwired
- Example: POWER, HP-PA RISC, MIPS, ARM, SPARC, Apple M1-M3

# Instruction-Level Parallel (ILP) architectures

**Single-Issue:** (can start at most one instruction per clock cycle)

- Simple, pipelined RISC processors with one or multiple functional units
  - e.g. ARM9E, DLX

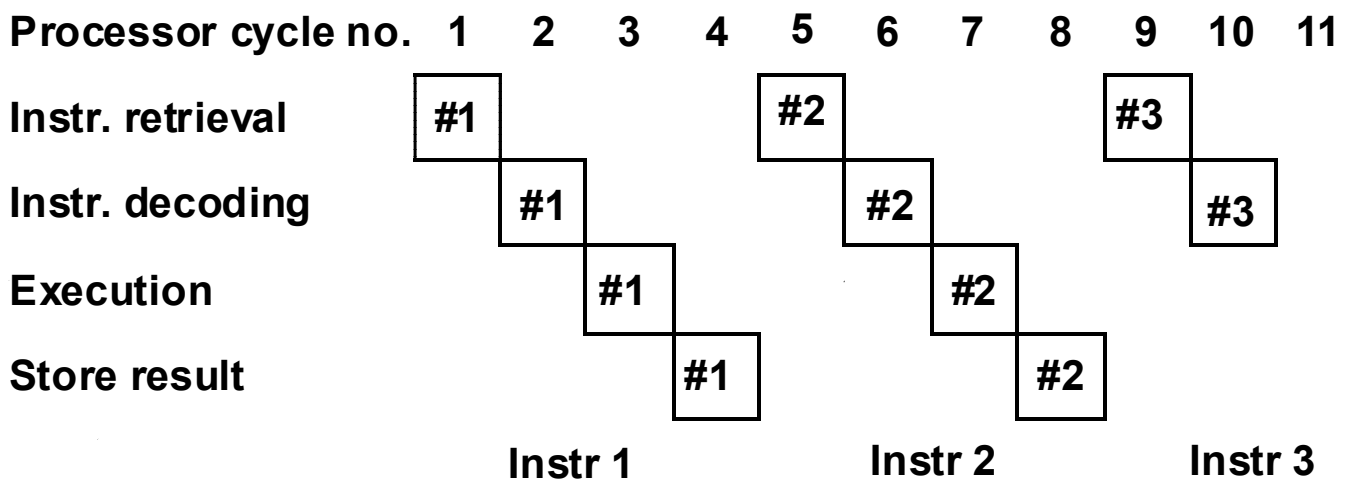
**Multiple-Issue:** (can start several instructions per clock cycle)

- Superscalar processors
  - e.g. Sun SPARC, MIPS R10K, Alpha 21264, IBM Power2, Pentium
- VLIW processors (Very Long Instruction Word)
  - e.g. Multiflow Trace, Cydrome Cydra-5, Intel i860, HP Lx, Transmeta Crusoe; most DSPs, e.g. Philips Trimedia TM32, TI 'C6x
- EPIC processors (Explicitly Parallel Instruction Computing)
  - e.g. Intel Itanium family (IA-64)

# Processors with/without Pipelining

Traditional processor without pipelining

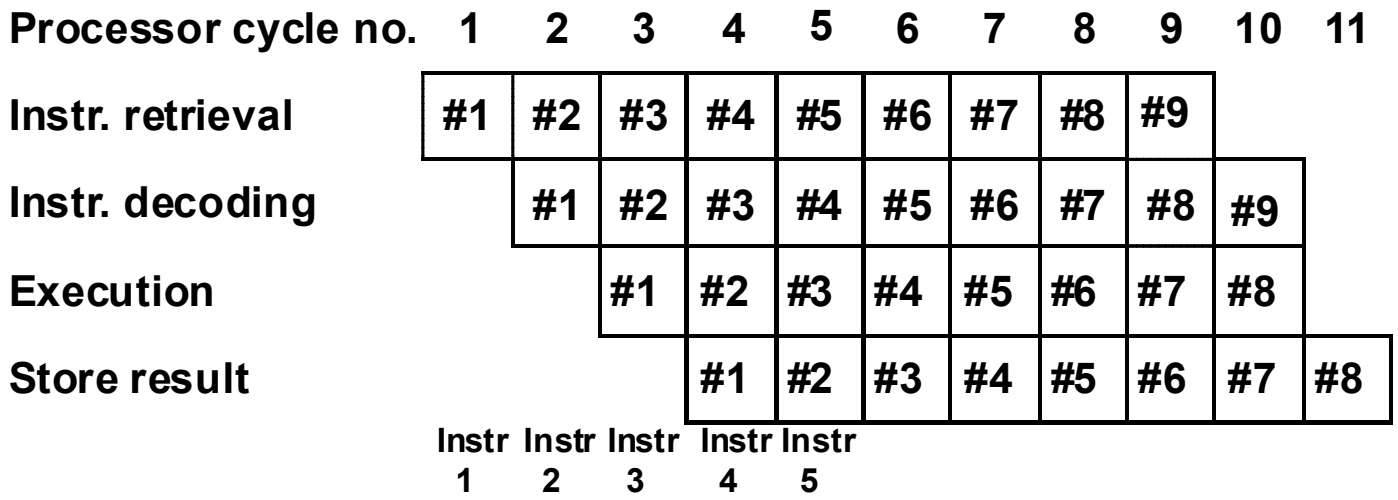
One instruction takes 4 processor cycles, i.e. 0.25 instructions/cycle



# Processor with Simple Pipelining

An instruction takes 1 cycle on average with pipeline
i.e. 1 instruction/cycle

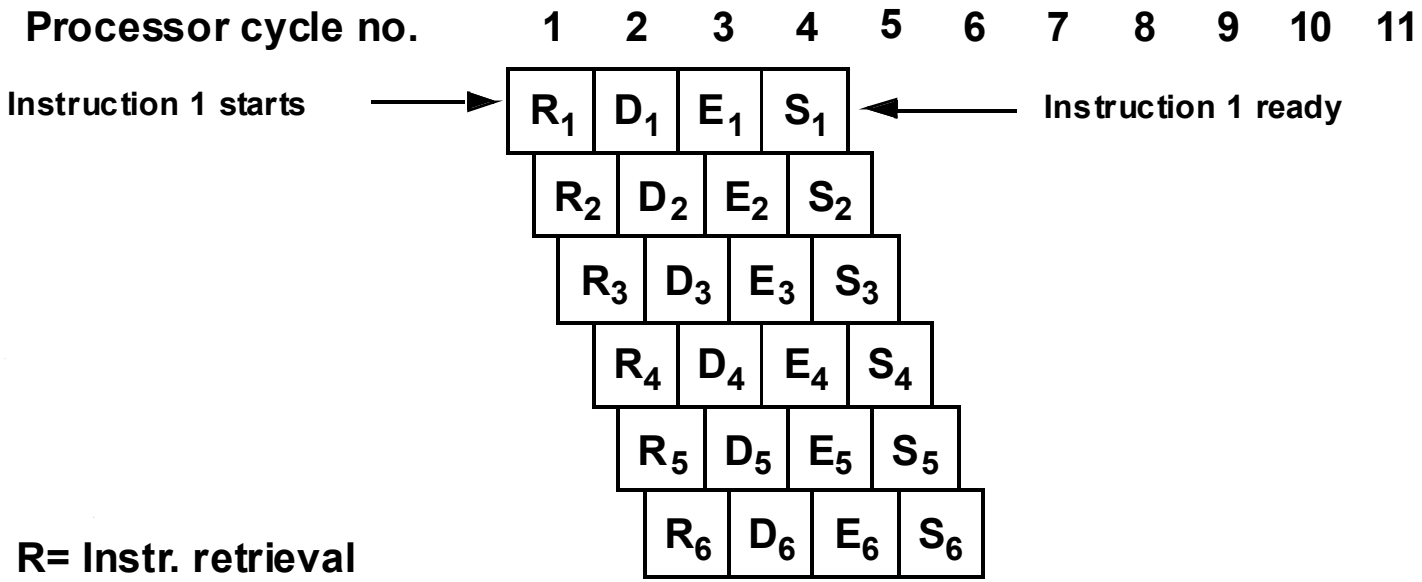
This pipeline achieves 4-way parallelism



# Processor with Super-Pipelining

A new instruction can begin before the previous one is finished.

Thus, you manage on average 3 instr/cycle when the pipeline is full.



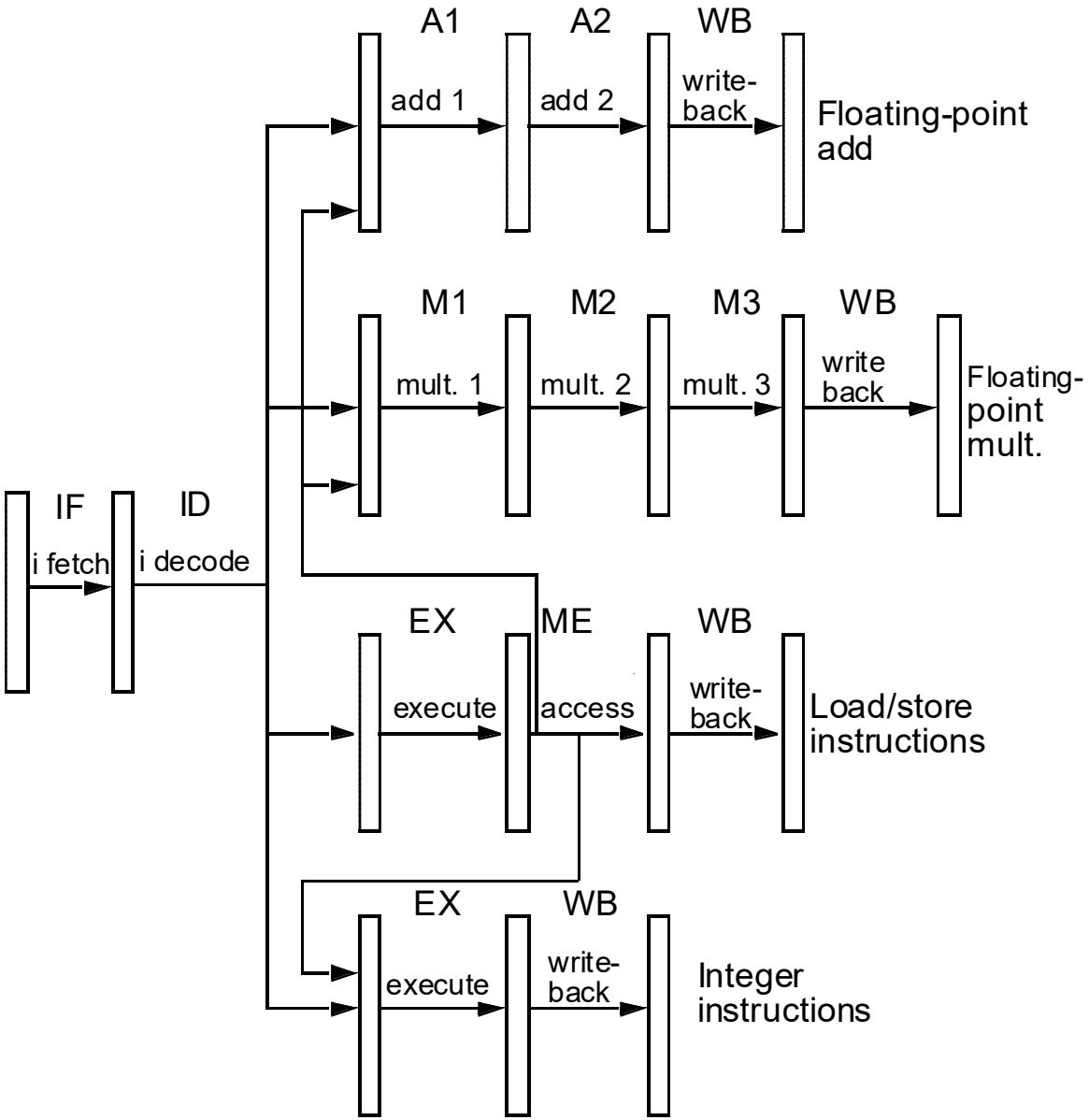

**R= Instr. retrieval**

**D= Instr. decoding**

**E= Execution**

**S= Store result**

# A Processor with Parallel Pipelines

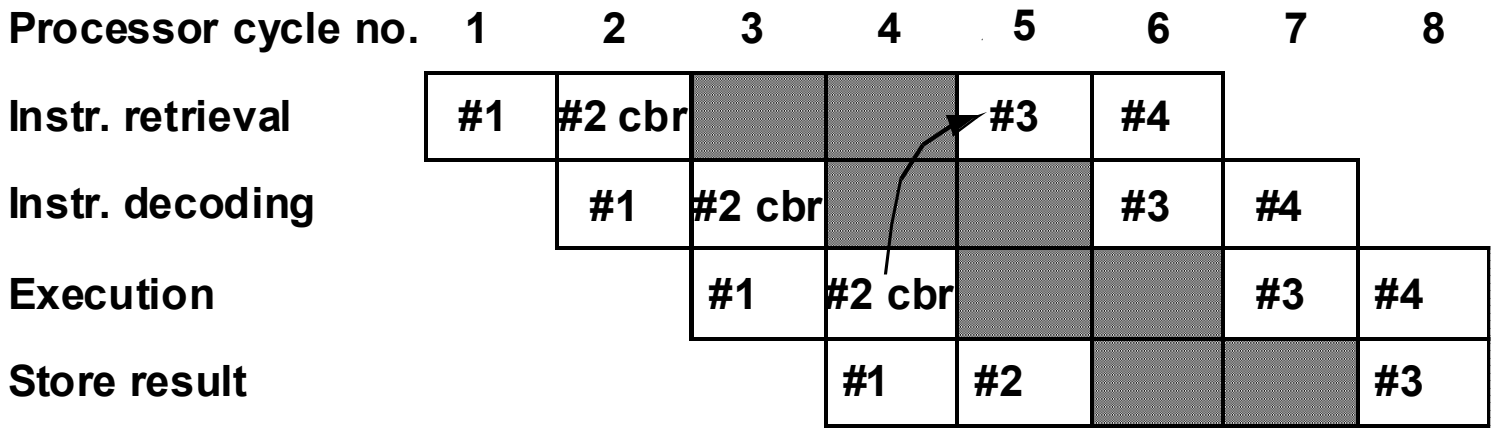


# Problems using Branch Instructions on Simple Pipelined Processors

Branch instructions force the pipeline to *restart* and thus reduce performance.

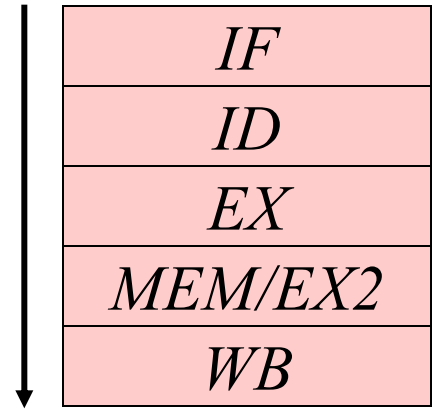
The diagram below shows execution of a branch
(cbr = conditional branch) to instruction #3, which makes the pipeline restart.

The grey area indicates lost performance.
Only 4 instructions start in 6 cycles instead of the maximum of 6.



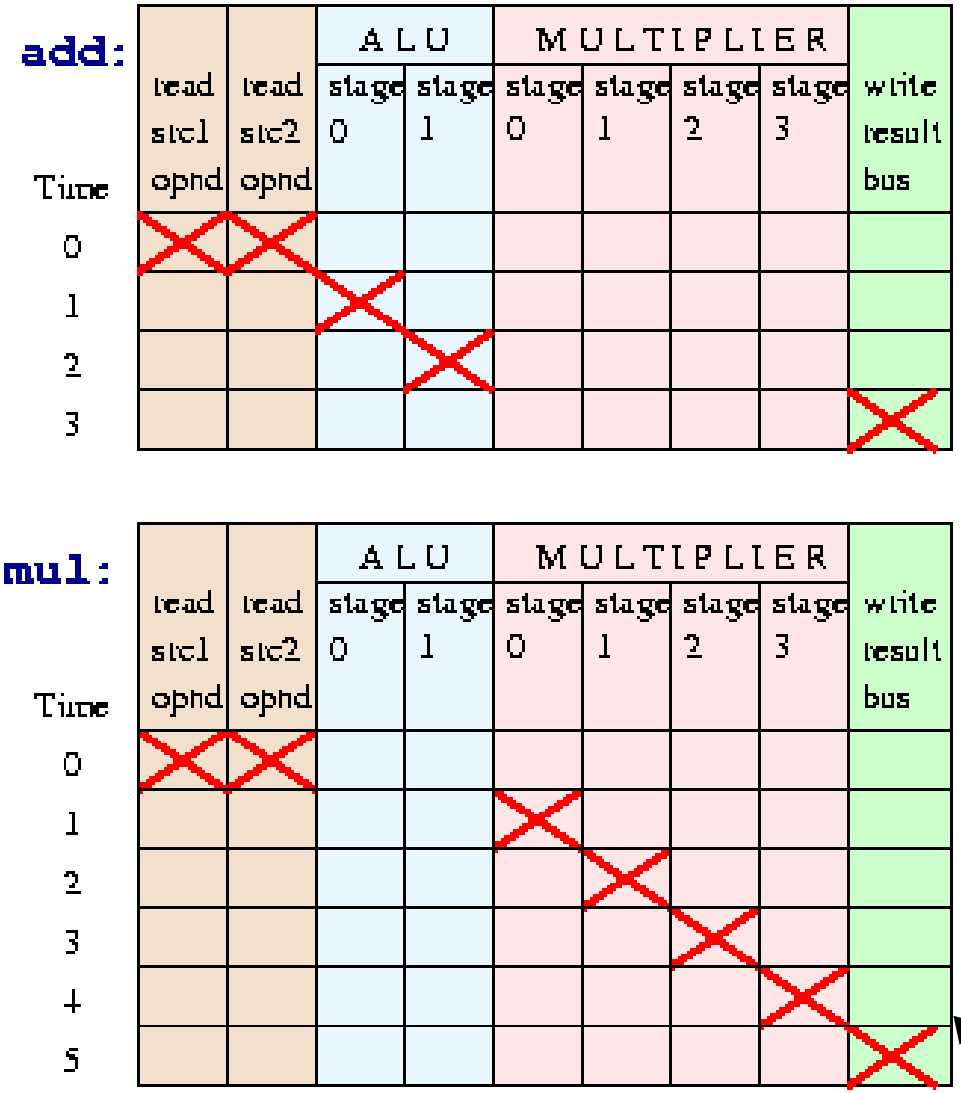
# Summary Pipelined RISC Architectures

- A single instruction is issued per clock cycle
- Possibly several parallel functional units / resources
- Execution of different phases of subsequent instructions overlaps in time. This makes them prone to:
  - data hazards (may have to delay op until operands ready),
  - control hazards (may need to flush pipeline after wrongly predicted branch),
  - structural hazards (required resource(s)/ e.g. functional units, bus, register, must not be occupied)
- Static scheduling (insert NOPs to avoid hazards) vs. Run-time treatment by pipeline stalling




| issue | cycle | $PM$ | Decoder | $ALU_1$ | $DM/ALU_2$ | Regs |
|---|---|---|---|---|---|---|
| $I_1$ | 1 | $IF_1$ | | | | |
| $I_2$ | 2 | $IF_2$ | $ID_1$ | | | |
| $I_3$ | 3 | $IF_3$ | $ID_2$ | $EX_1$ | | |
| $I_4$ | 4 | $IF_4$ | $ID_3$ | $EX_2$ | $MEM_1$ | |
| $I_5$ | 5 | $IF_5$ | $ID_4$ | $EX_3$ | $MEM_2$ | $WB_1$ |
| $I_6$ | 6 | $IF_6$ | $ID_5$ | $EX_4$ | $MEM_3$ | $WB_2$ |
| ... | | | | | | |

# Reservation Table, Scheduling Hazards
## (avoid hazards = resource collisions)




**Reservation table** specifies required resource occupations

[Davidson 1975]

If we start **add** at t+2, the bus write will appear at cycle t+5

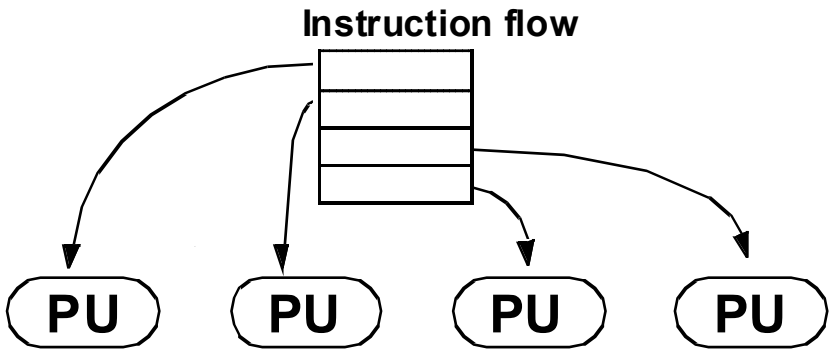
```
t:    mul ...
t+1:  ...
t+2:  add ...
...
```

*structural hazard at t=5*

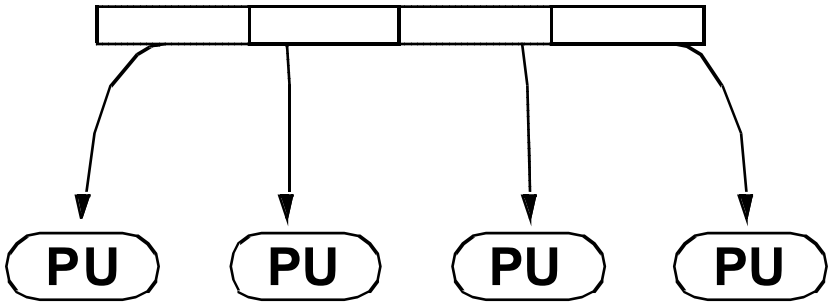
# Comparison between Superscalar Processors and VLIW processors

## Superscalar Processors

with multiple loading of instructions
(multi-issue)



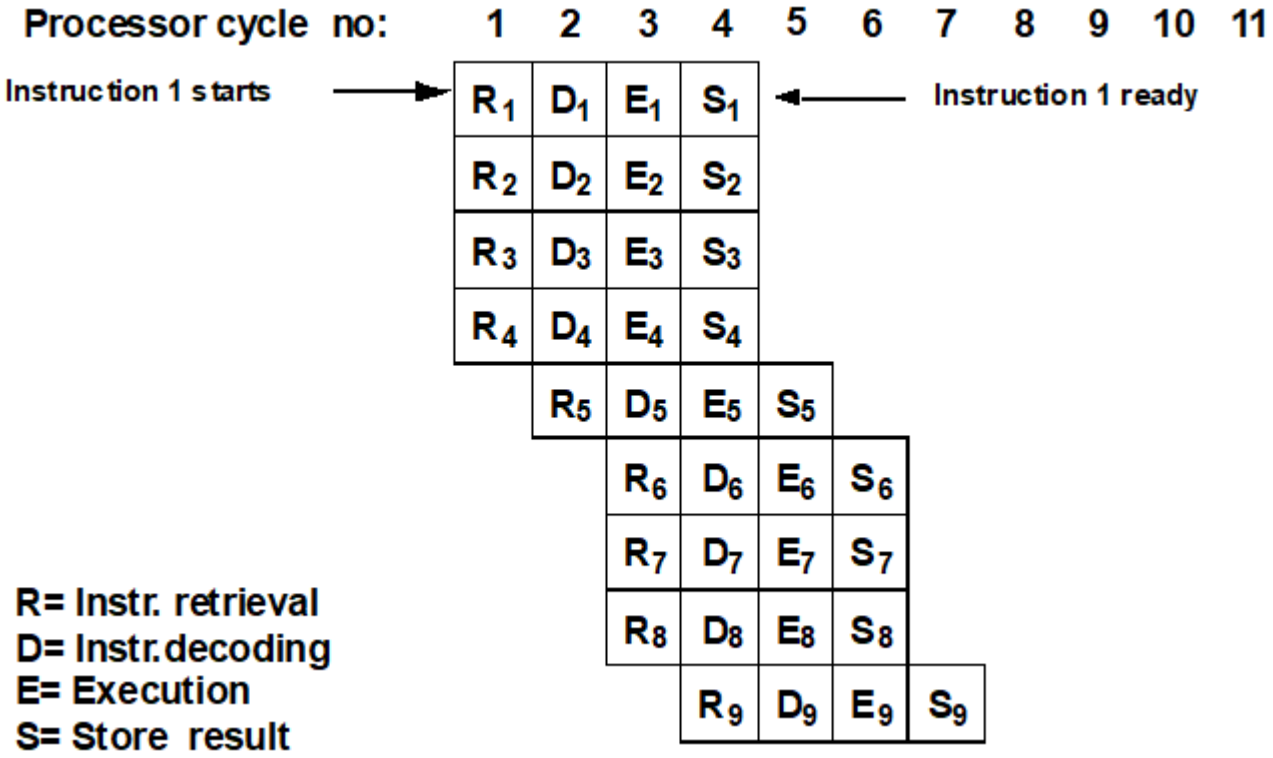

## VLIW Processors
(Very Long Instruction Word)




Several processor units are loaded simultaneously be different operations in the same instructions.
E.g. the multiflow machine,
1024 bits, 28 operations,
or specialized graphics processors

# Superscalar Processors

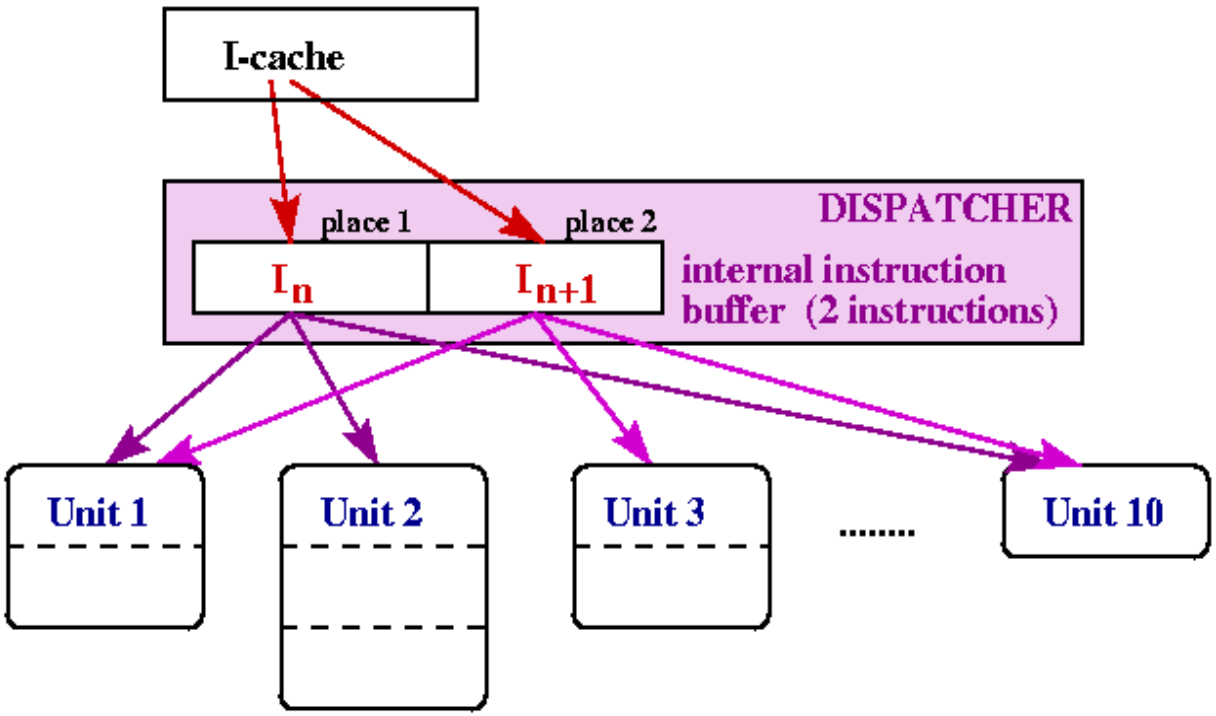
- A superscalar processor has several function units that can work in parallel, and which can load more than 1 instruction per cycle.
- The word **superscalar** comes from the fact that the processor executes more than 1 instruction per cycle.
- The diagram below shows how a maximum of 4 units can work in parallel, which in theory means they work 4 times faster.
- The type of parallelism used depends on the type of instruction and dependencies between instructions.



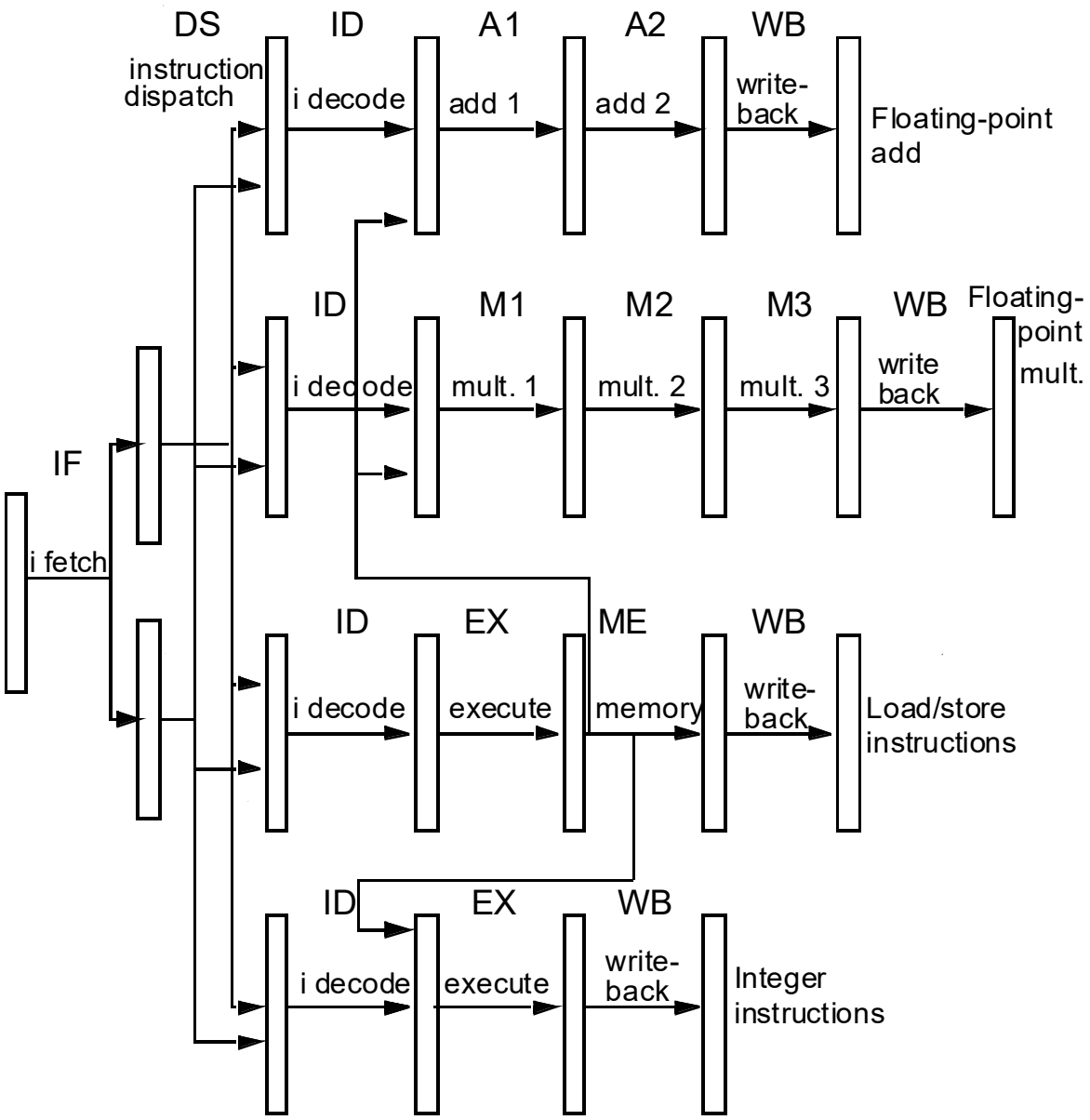
# Superscalar Processor

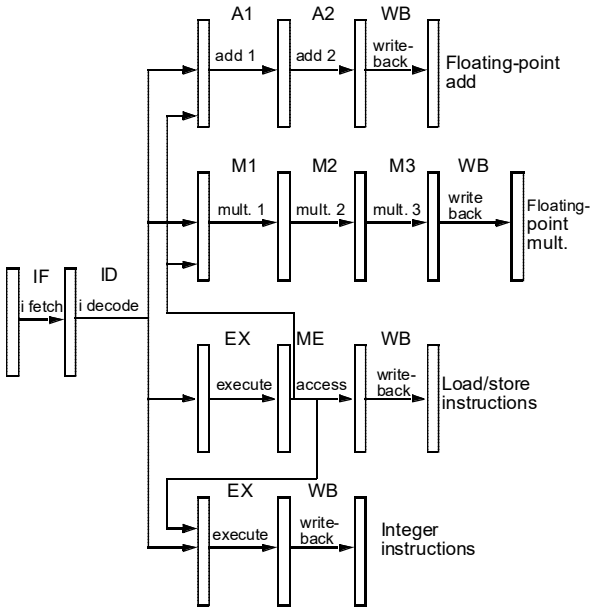
- Run-time scheduling by instruction dispatcher
  - convenient (sequential instruction stream – as usual)
  - limited look-ahead buffer to analyze dependences, reorder instr.
  - high silicon overhead, high energy consumption
- Example: Motorola MC 88110

  2-way, in-order issue superscalar



# A Parallel Superscalar Pipeline



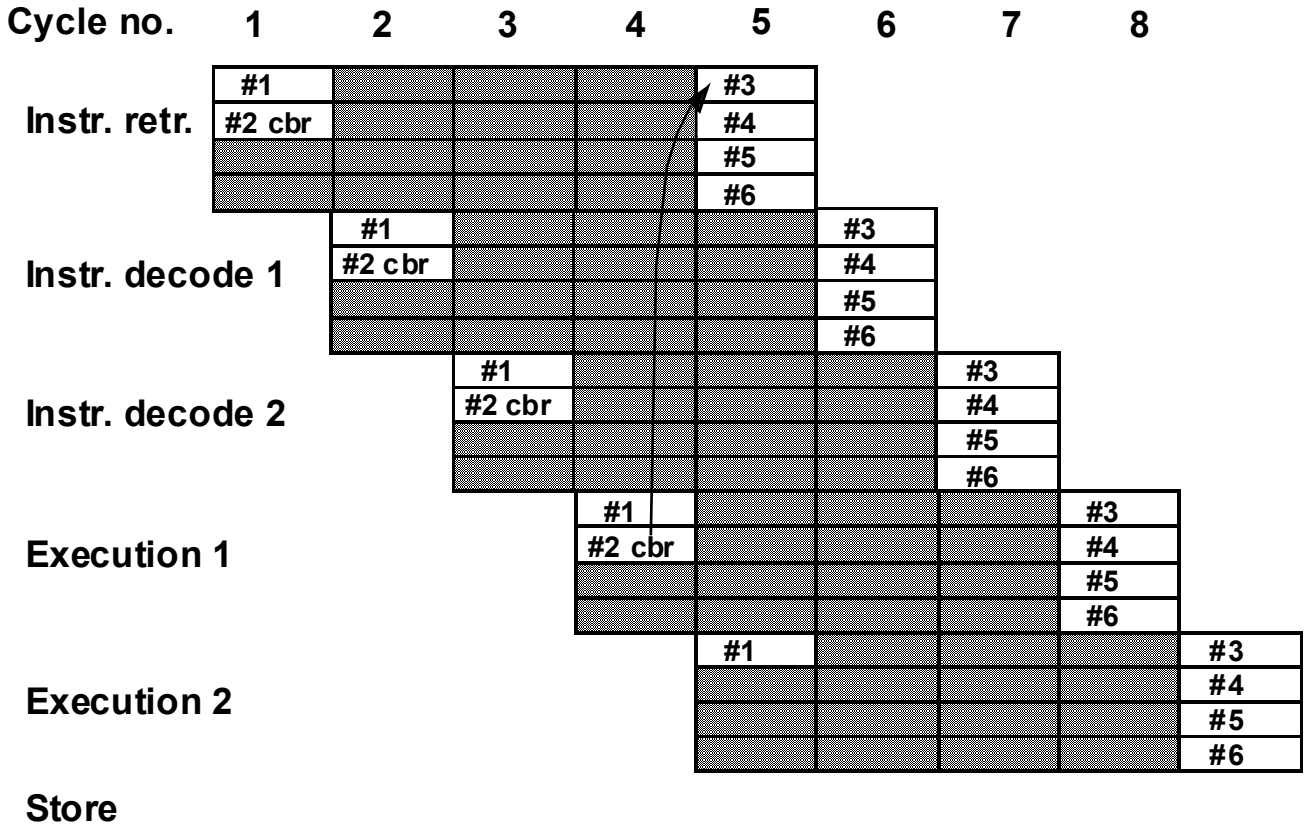



# Branch Effects on Performance for Deeply Pipelined Superscalar Processors

Branch-instructions force the pipeline to restart and thus reduce performance. Worse on deeply pipelined superscalar processors.

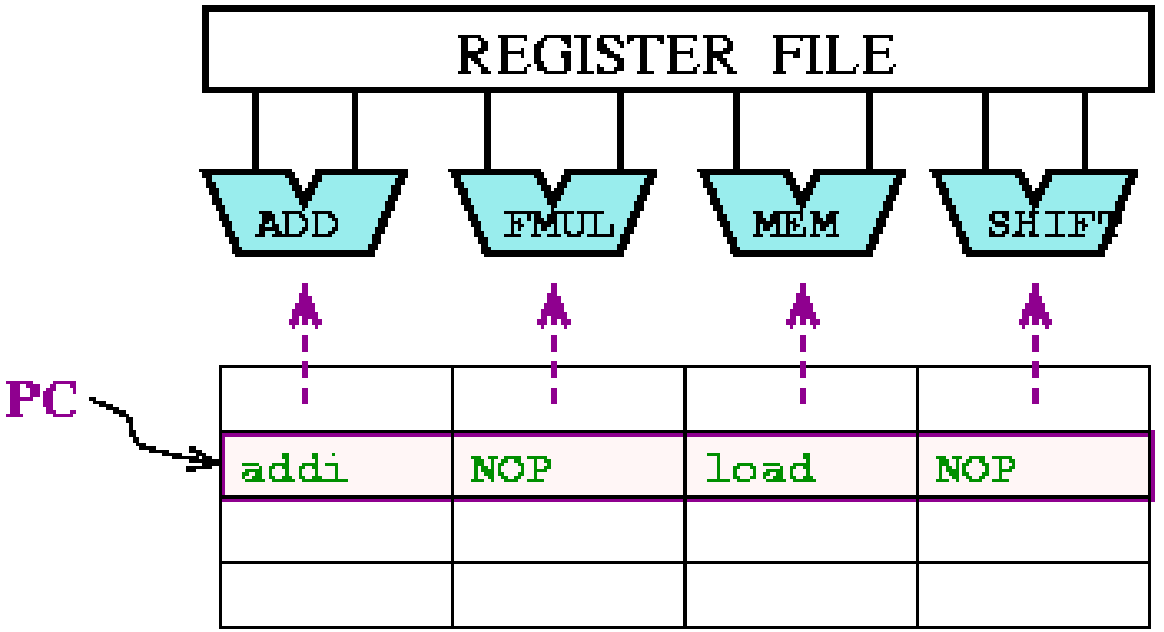
The diagram shows execution of a branch (cbr = conditional branch) to instruction #3, which makes the pipeline restart.

The grey area indicates lost performance.
Only 6 instructions start during 5 cycles instead of a maximum of 20.



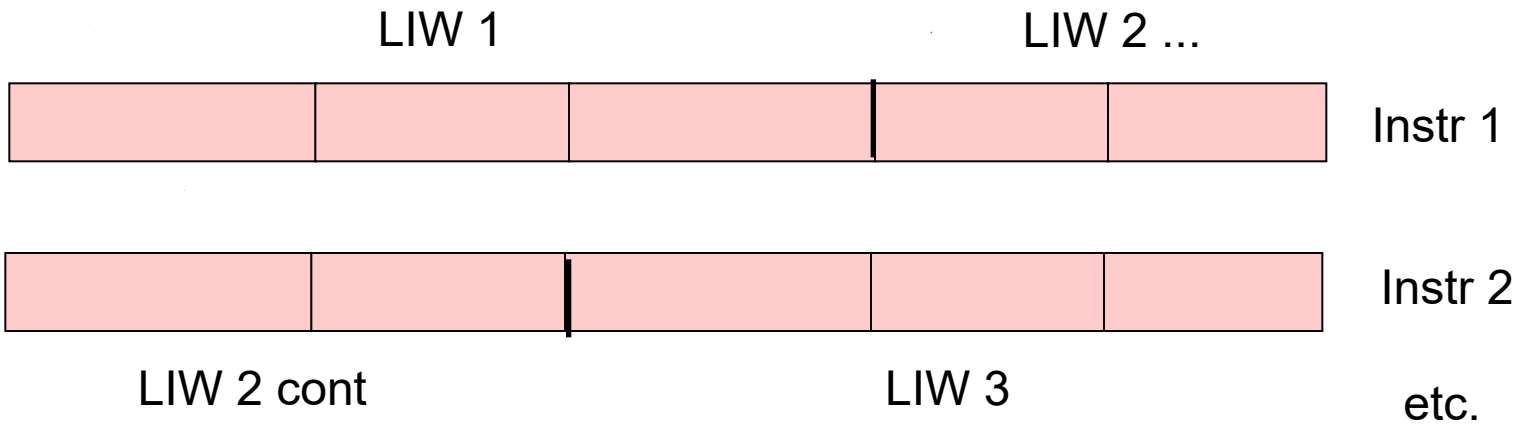
# VLIW (Very Long Instruction Word) architectures

- Multiple slots for instructions in long instruction-word
  - Direct control of functional units and resources – low decoding OH
- Compiler (or assembler-level programmer) must determine the schedule statically
  - independence, unit availability, packing into long instruction words
  - Challenging! But the compiler has more information on the program than an on-line scheduler with a limited lookahead window.
  - Silicon- and energy-efficient



# EPIC Architectures
## (Explicitly Parallel Instruction Computing)

- Based on VLIW
- Compiler groups instructions to LIW's (bundles)
- Compiler takes care of resource and latency constraints
- Compiler marks sequences of independent instructions
- Dynamic scheduler assigns resources and reloads new bundles as required



LINKÖPINGS UNIVERSITET

# 2. Instruction Scheduling

# The Instruction Scheduling Problem

- Schedule the instructions in such an order that parallel function units are used to the greatest possible degree.

- Input:
  - Instructions to be scheduled
  - A data dependency graph
  - A processor architecture
  - Register allocation has (typically) been performed

- Output:
  - A scheduling of instructions which minimizes execution time or energy

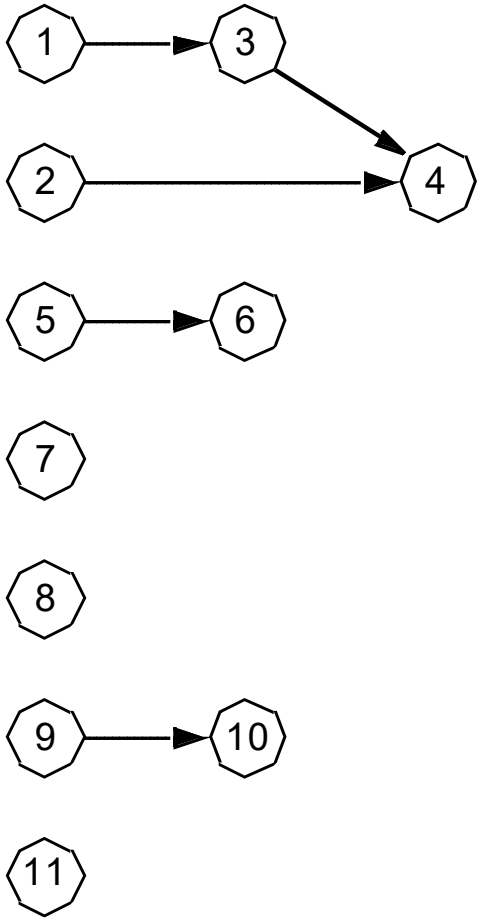
# Example Instructions to be Scheduled

LINKÖPINGS UNIVERSITET

**Instructions**

```
(01)  mov    rax, 5
(02)  mov    rcx, [rbp-16]
(03)  mul    rax, 8
(04)  mov    [rcx-64], rax
(05)  push   4
(06)  call   L6
(07)  inc    [rbp-8]
(08)  dec    [rbp+8]
(09)  mov    rdx, [rsp-32]
(10)  mov    [rsp-40], rdx
(11)  ret
```

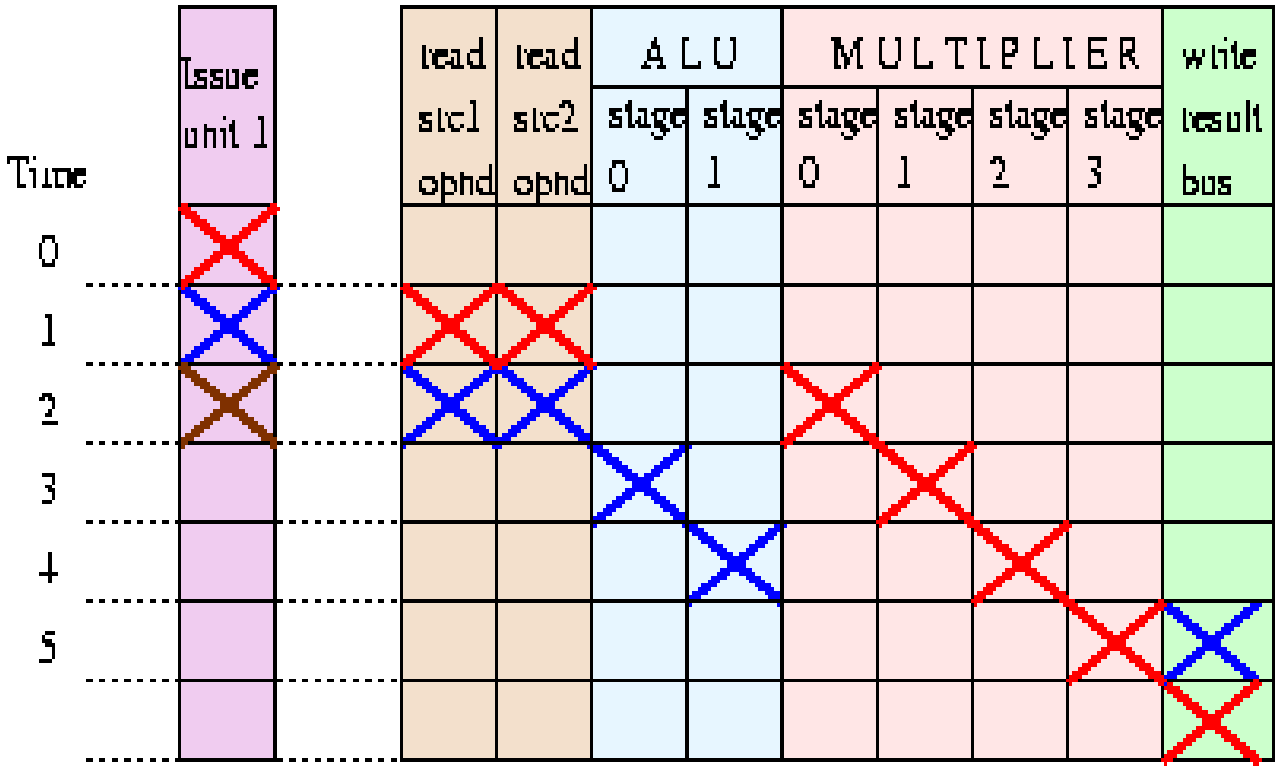
**Dependency graph**



# Instruction Scheduling (1)

- Map instructions to time slots on issue units (and resources), such that no hazards occur
  → ***Global reservation table, resource usage map***
- Example without data dependences:

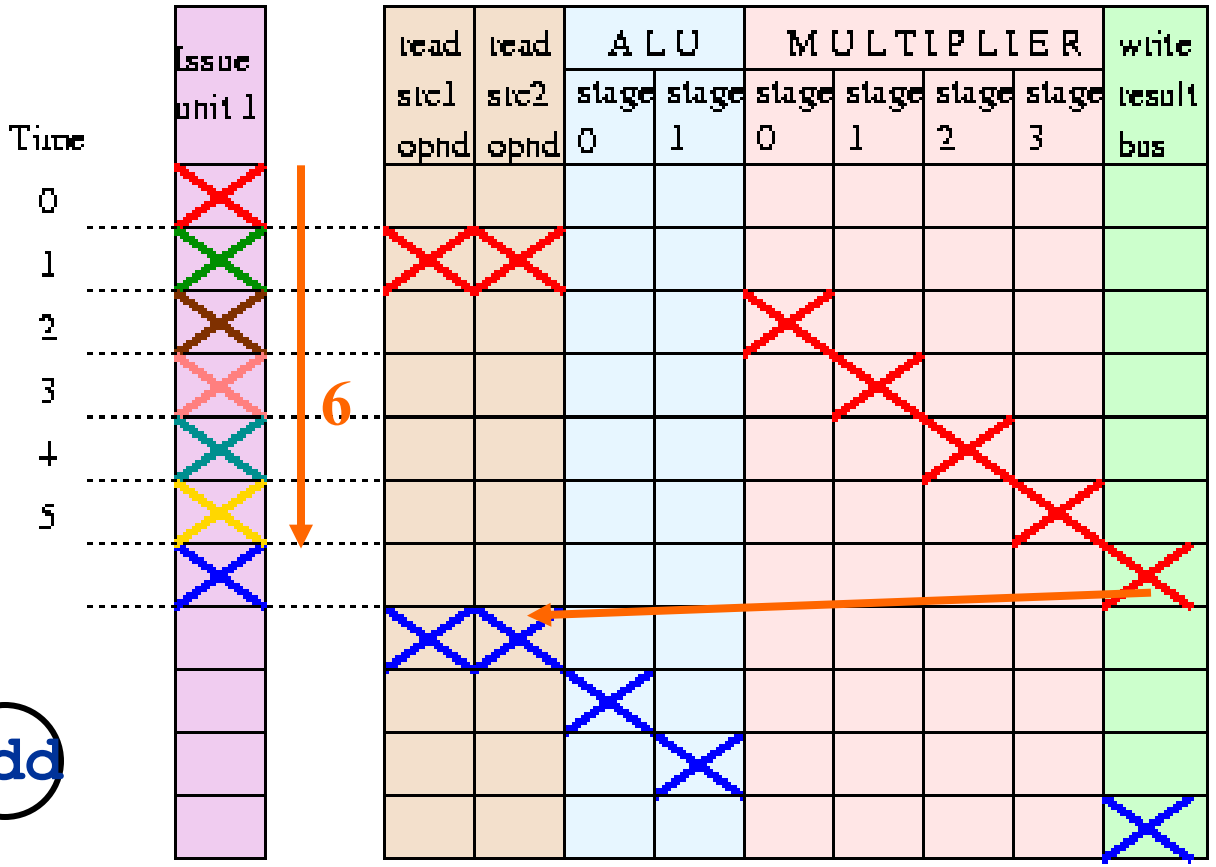
```
t:    mul ...
t+1:  add ...
t+2:  nop ...
      ...
```



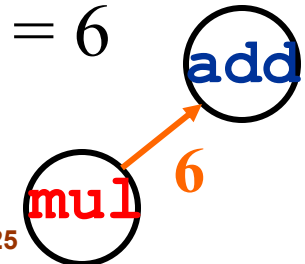
# Instruction Scheduling (2)

- Data dependences imply **latency constraints**
  → target-level data flow graph / data dependence graph

```
t:    mul R1,...
t+1:  nop ...
t+2:  nop ...
t+3:  nop ...
t+4:  nop ...
t+5:  nop ...
t+6:  add ...,R1
```




*latency*(mul) = 6

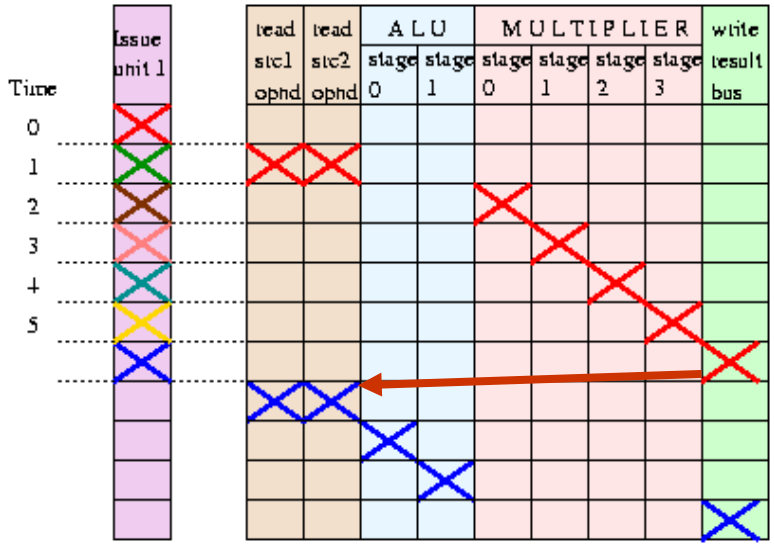


# Instruction Scheduling

## Generic Resource model

- Reservation table

```
t:   mul R1,...
t+1: nop ...
t+2: nop ...
t+3: nop ...
t+4: nop ...
t+5: nop ...
t+6: add ...,R1
```




## Local Scheduling

(f. Basic blocks / DAGs)

- Data dependences
  → Topological sorting
  - List Scheduling
    (diverse heuristics)

## Global Scheduling

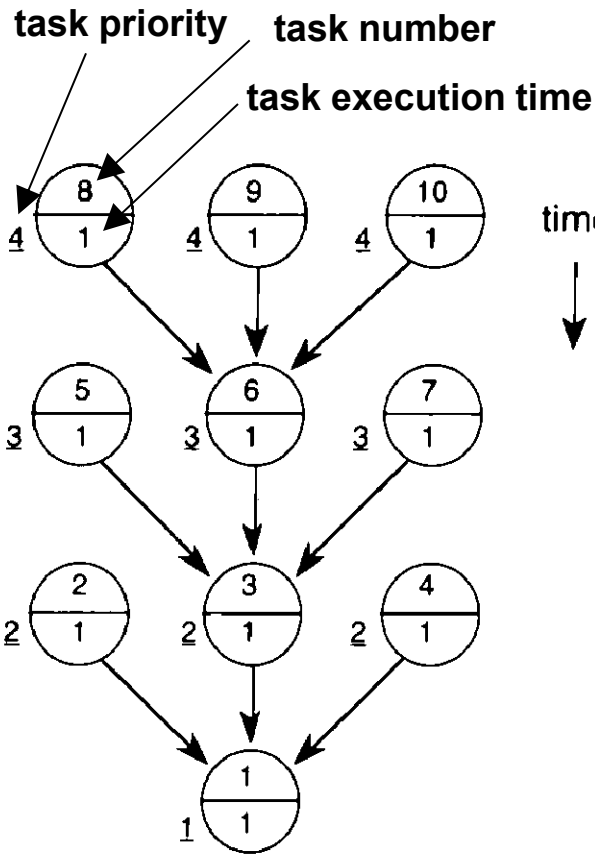
- Trace scheduling, Region scheduling, ...
- Cyclic scheduling (Software pipelining)

There exist **retargetable schedulers**
and **scheduler generators**, e.g. for GCC since 2003

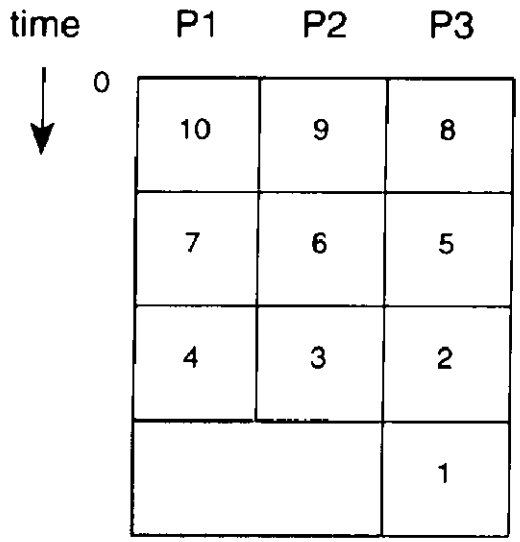
# Example of List Scheduling Algorithm

- The **level** of a task (i.e., instruction) node is the maximal number of nodes that are passed on the way to the final node, itself included.
- The algorithm:
  - The **level** of each node is used as **priority**.
  - When a processor/function unit is free, assign the unexecuted task which has **highest priority,** and which is ready to be executed.

Example of Highest Level First algorithm on a tree structured task graph, 3 processor units




Task Graph

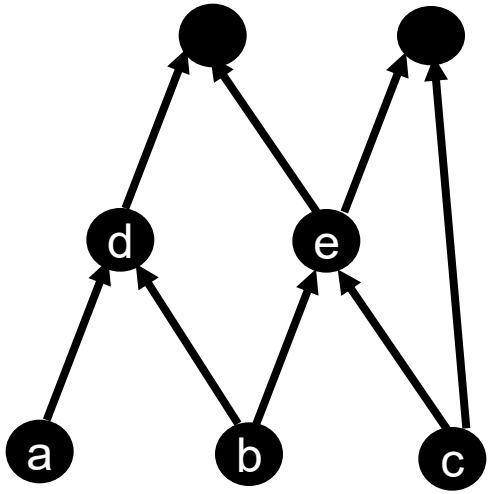



Gantt Chart

# Example: Topological Sorting (0) According to Data Dependencies

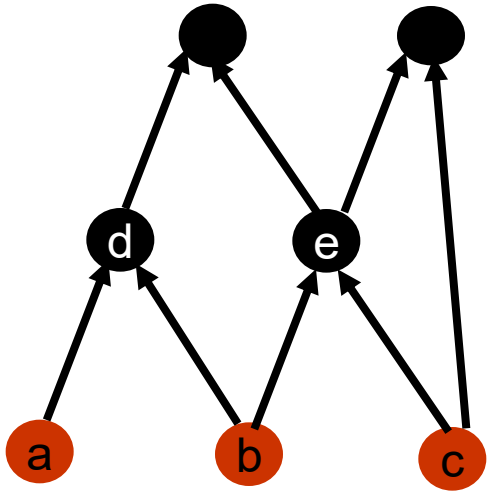
LINKÖPINGS UNIVERSITET

- Not yet considered
- Data ready (zero indegree set)
- Already scheduled, still live
- Already scheduled, no longer referenced



# Example: Topological Sorting (1) According to Data Dependencies

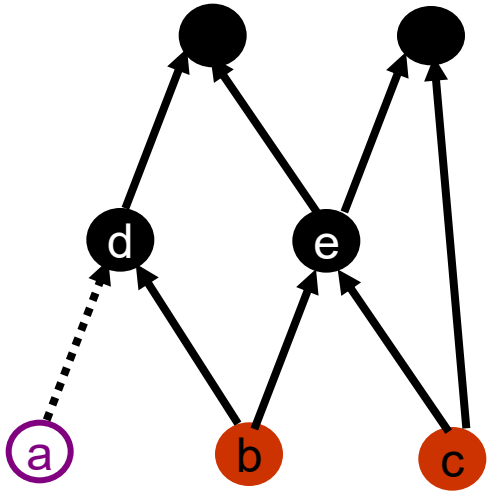
- Not yet considered
- Data ready (zero indegree set)
- Already scheduled, still live
- Already scheduled, no longer referenced




a

# Example: Topological Sorting (2) According to Data Dependencies

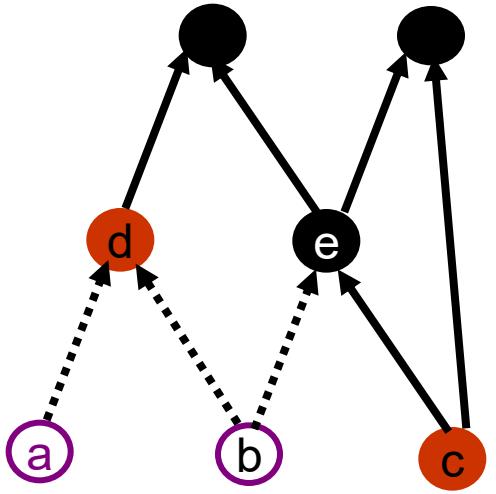
- Not yet considered
- Data ready (zero indegree set)
- Already scheduled, still live
- Already scheduled, no longer referenced




a b

# Example: Topological Sorting (3) According to Data Dependencies

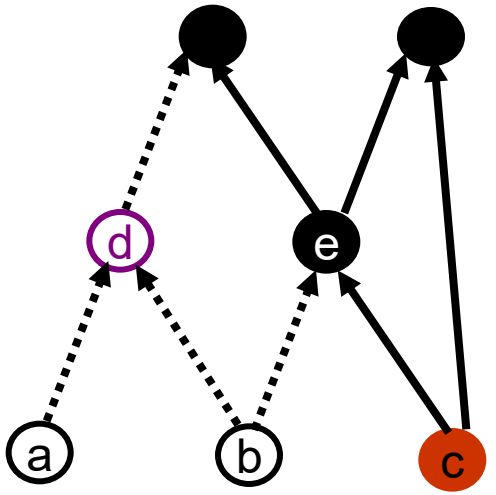
- Not yet considered
- Data ready (zero indegree set)
- Already scheduled, still live
- Already scheduled, no longer referenced




a b d

# Example: Topological Sorting (4) According to Data Dependencies

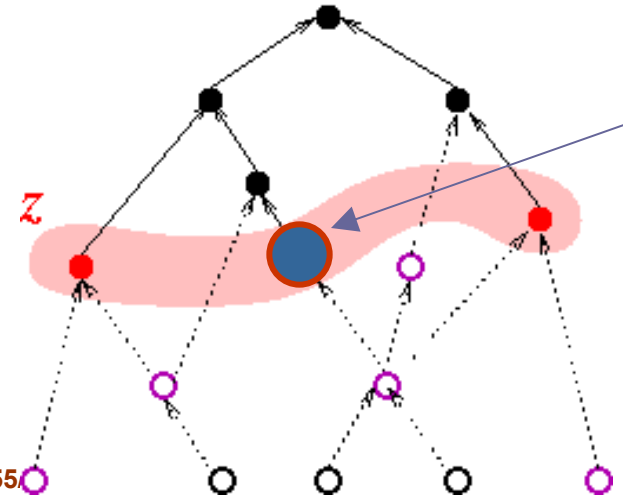
- Not yet considered
- Data ready (zero indegree set)
- Already scheduled, still live
- Already scheduled, no longer referenced



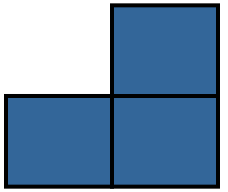

a b d and so on...

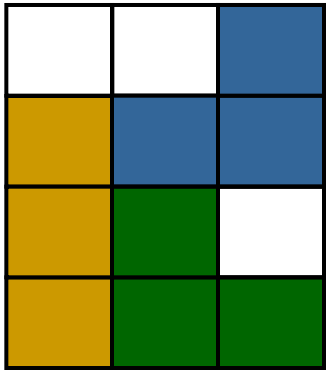
# Topological Sorting and Scheduling

- Construct schedule incrementally in topological (= causal) order
  - ”Appending” instructions to partial code sequence: close up in target schedule reservation table (as in ”Tetris”)
  - Idea: Find optimal target-schedule by enumerating all topological sorting options ...
    - Beware of scheduling anomalies with complex reservation tables!



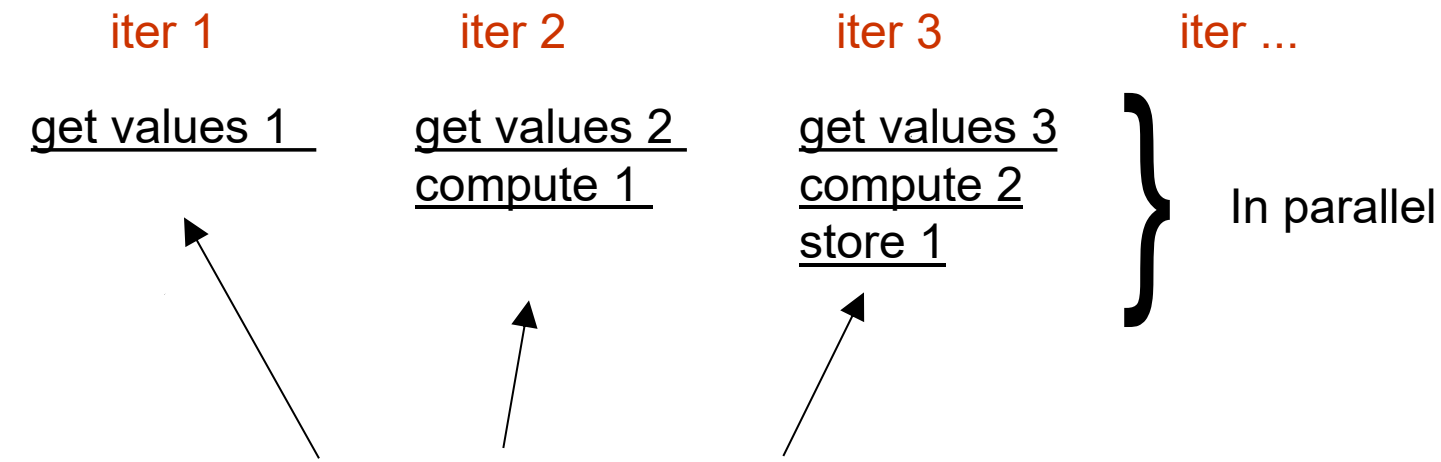

Instruction needing
3 functional units



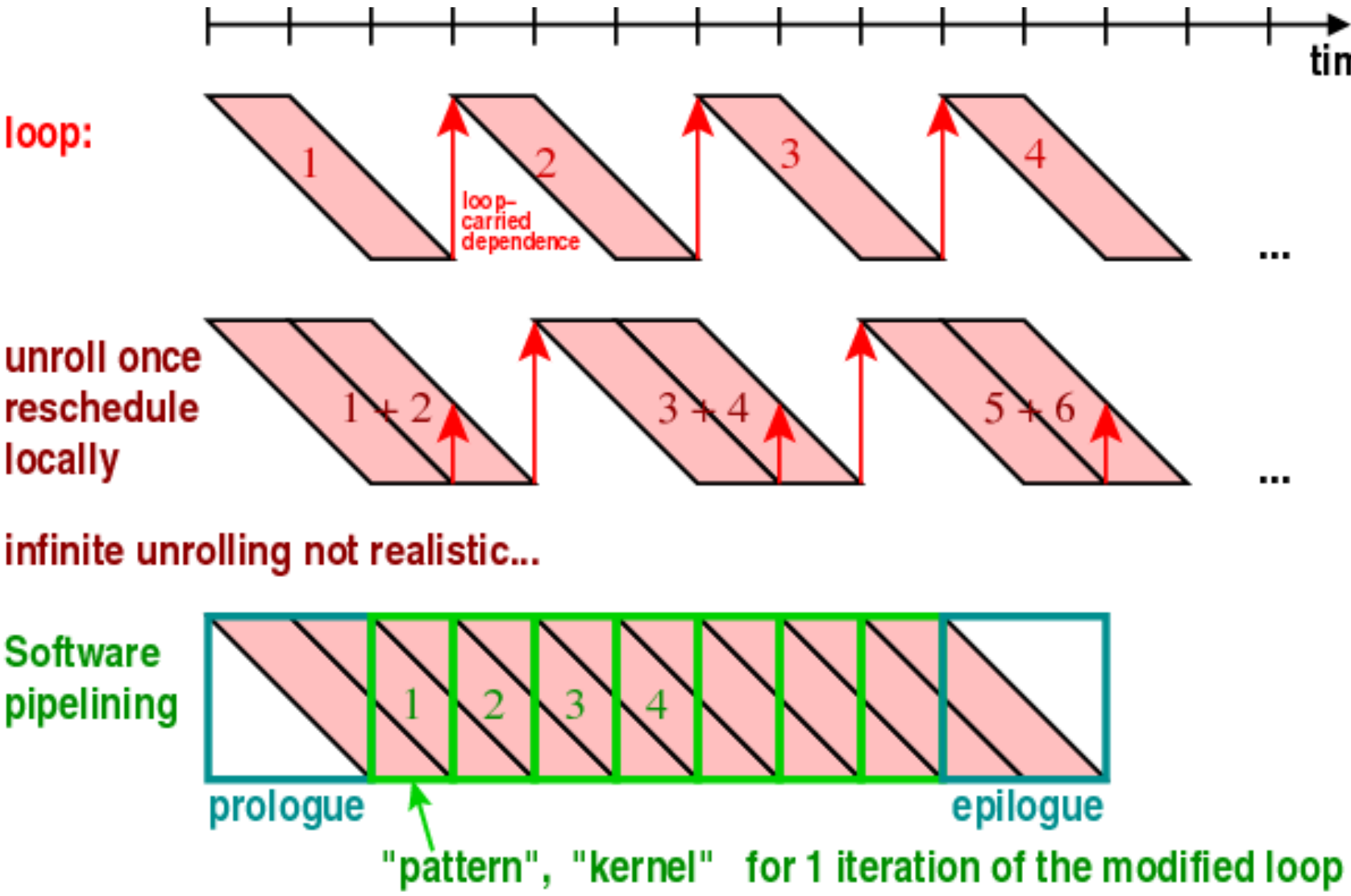


# Software Pipelining

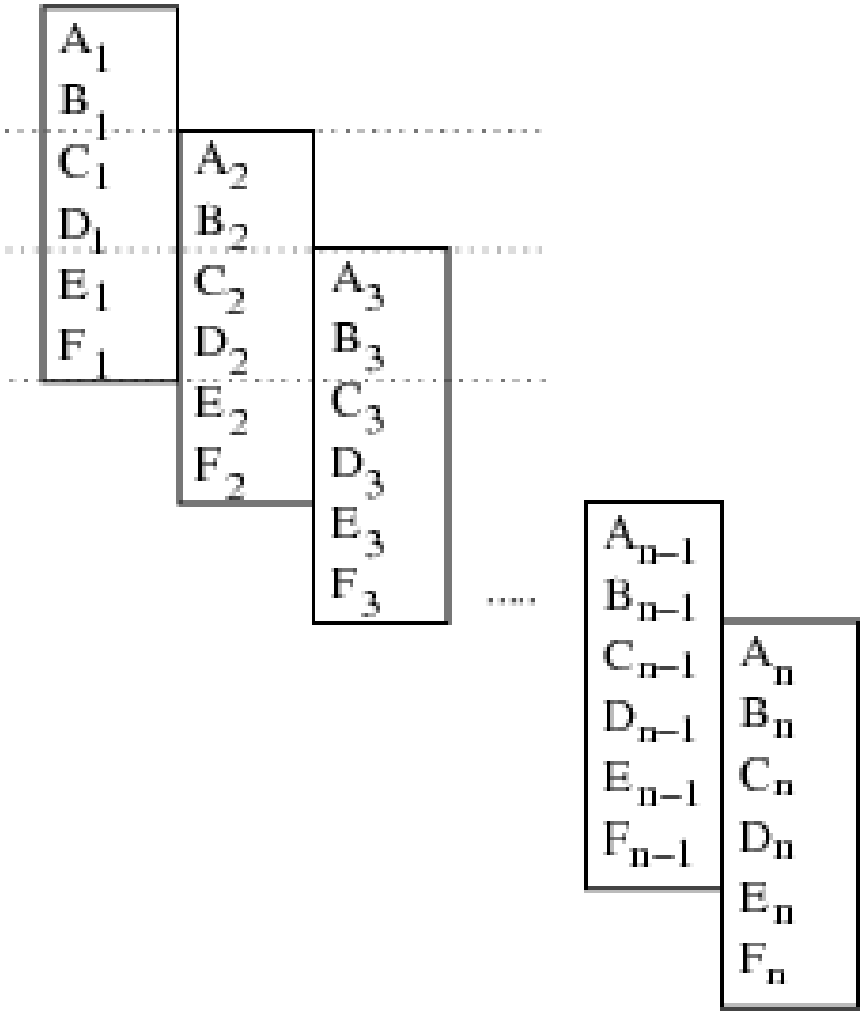
```
for i := 1  to  n
              get values;
              compute;
              store;
end for
```



# Software Pipelining of Loops (1)



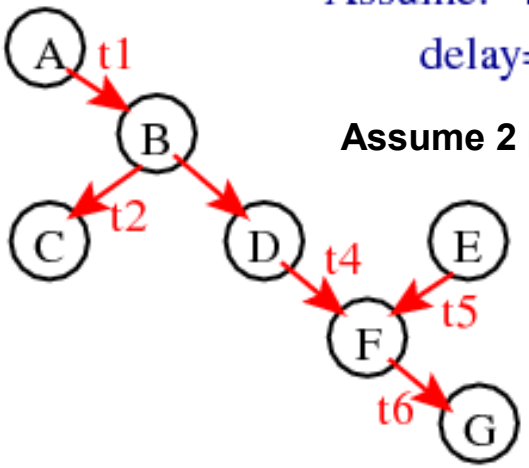
# Software Pipelining of Loops (2)




| | Unit1 | Unit2 | Unit3 |
|---|---|---|---|
| Prologue: | $A_1$ | | |
| | $B_1$ | | |
| | $C_1$ | $A_2$ | |
| | $D_1$ | $B_2$ | |
| | DO i=1, n–2 | | |
| Pattern: | $E_i$ | $C_{i+1}$ | $A_{i+2}$ |
| | $F_i$ | $D_{i+1}$ | $B_{i+2}$ |
| Epilogue: | | $E_{n-1}$ | $C_n$ |
| | | $F_{n-1}$ | $D_n$ |
| | | | $E_n$ |
| | | | $F_n$ |

→ More about Software Pipelining in TDDC86
*Compiler Optimizations and Code Generation*

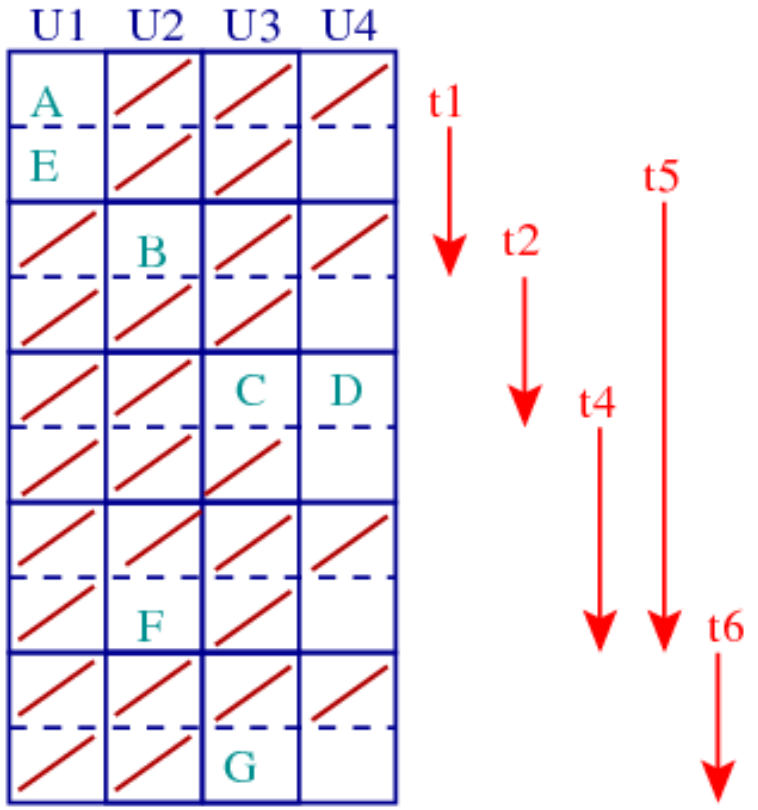
# Software Pipelining of Loops (3)
# Modulo Scheduling




Assume: 4 units, fully pipelined
delay=2 for all instructions

**Assume 2 processor cycles**

**7 instructions**
**A, B, C, D,...G**




No dependence cycles

ResMII = ceil( 7/4 ) = 2

**ResMII = Resource Constrained Minimum Initiation Interval**

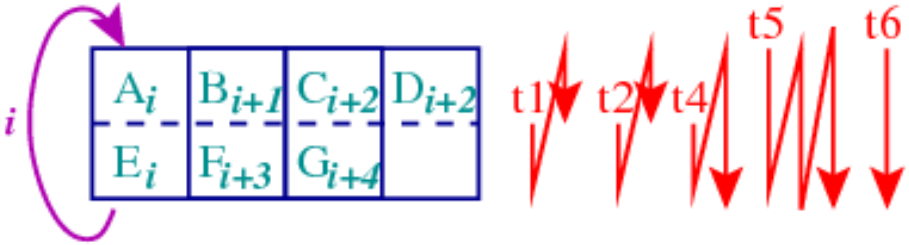
Begin with II = ResMII = 2

Apply some local scheduling heuristic
e.g.: list scheduling ( A B C D E F G )

Apply some placement heuristic
e.g.: as early as possible

Mark occupied slots in all iterations...

If not possible, increase II and try again...



LINKÖPINGS UNIVERSITET

# 3. Register Allocation

# Global Register Allocation

- **Register Allocation**: Determines values (variables, temporaries, constants) to be kept when in registers.
- **Register Assignment**: Determine in which physical register such a value should reside.

- Essential for Load-Store Architectures
- Reduce memory traffic (→ memory / cache latency, energy)
- Limited resource
- Values that are live simultaneously cannot be kept in the same register
- Strong interdependence with instruction scheduling
  - scheduling determines live ranges
  - spill code needs to be scheduled
- **Local register allocation** (for a single basic block) can be done in linear time (see previous lecture)
- **Global register allocation** on whole procedure body (with minimal spill code) is NP-complete.
  Can be modeled as a graph coloring problem [Ershov'62] [Cocke'71].

# When do Register Allocation

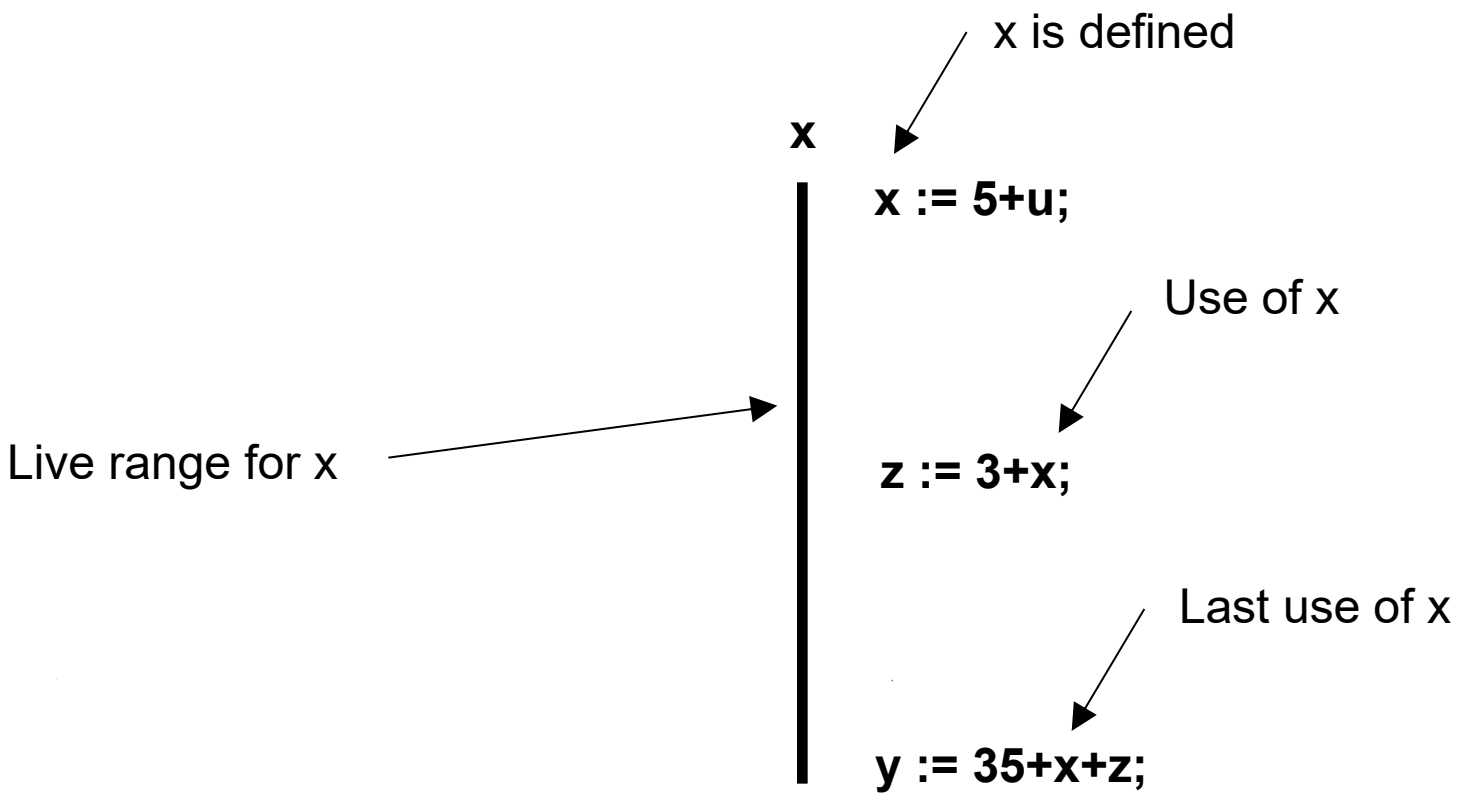
- Register allocation is normally performed **at the end of global optimization**, when the final structure of the code and all potential use of registers is known.
- It is **performed on abstract machine code** where you have access to an unlimited number of registers or some other intermediary form of program.
- The code is divided into sequential blocks (basic blocks) with accompanying control flow graph.

# Live Range

(Here, variable = program variable or temporary)

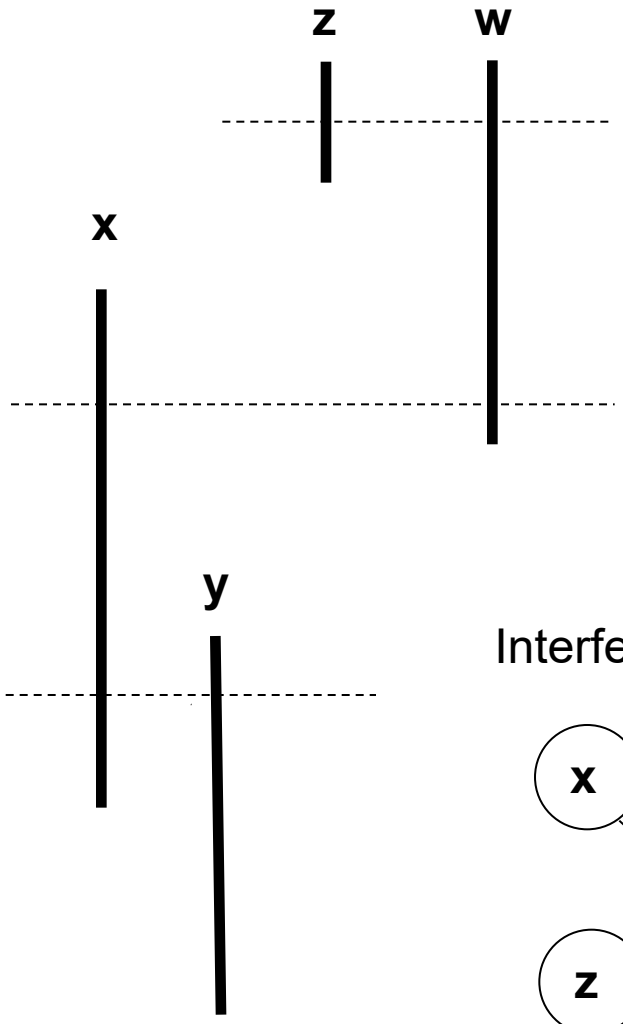
- A variable is being **defined** at a program point if it is written (given a value) there.
- A variable is **used** at a program point if it is read (referenced in an expression) there.
- A variable is **live** at a point if it is referenced there or at some following point that has not (may not have) been preceded by any definition.
- A variable is **reaching** a point if an (arbitrary) definition of it, or usage (because a variable can be used before it is defined) reaches the point.
- A variable's **live range** is the area of code (set of instructions) where the variable is both live and reaching.
  - does not need to be consecutive in program text.

# Live Range Example

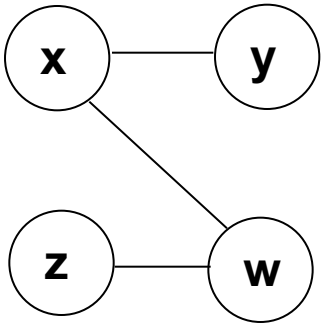


# Interference Graphs

- The live ranges of two variables *interfere* if their intersection is not empty.
- Each live range builds a node in the *interference graph* (or *conflict graph)*
- If two live ranges interfere, an edge is drawn between the nodes.
- Two adjacent nodes (connected by a vertex) in the graph can not be assigned the same register.




Interference graph:



# Register Allocation vs Graph Coloring

- Register allocation can be compared with the classic coloring problem.
  - That is, to find a way of coloring - with a maximum of *k* colors - the interference graph which does not assign the same color to two adjacent nodes.
- *k* = the number of registers.
  - On a RISC-machine there are, for example, 16 or 32 general registers. Certain methods use some registers for other tasks. e.g., for spill code.
- Determining whether a graph is colorable using *k* colors is NP-complete for k>3
  - In other words, it is unmanageable always to find an optimal solution.

# Register Allocation by Graph Coloring

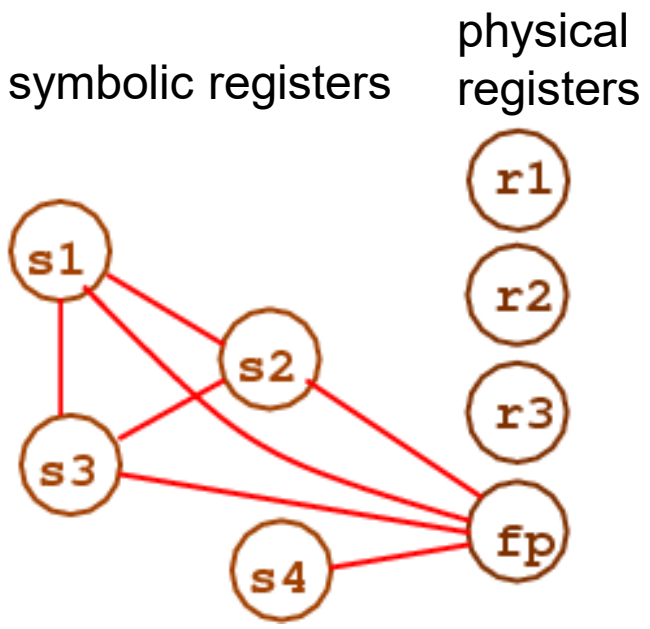
- **Step 1:** Given a program with symbolic registers s1, s2, ...
  - Determine live ranges of all variables

```
i = c+4;  load 8(fp),s1    ! c
          nop
          addi s1,#4,s2
          store s2,4(fp)   ! i
d = c-2;  subi s1,#2,s3
          store s3,12(fp)  ! d
c = c*i;  muli s1,s2,s4
          store s4,8(fp)   ! c
```

# Register Allocation by Graph Coloring

- **Step 2:** Build the **Register Interference Graph**
  - Undirected edge connects two symbolic registers (s*i*, s*j*) if live ranges of s*i* and s*j* overlap in time
  - Reserved registers (e.g. fp) interfere with all s*i*

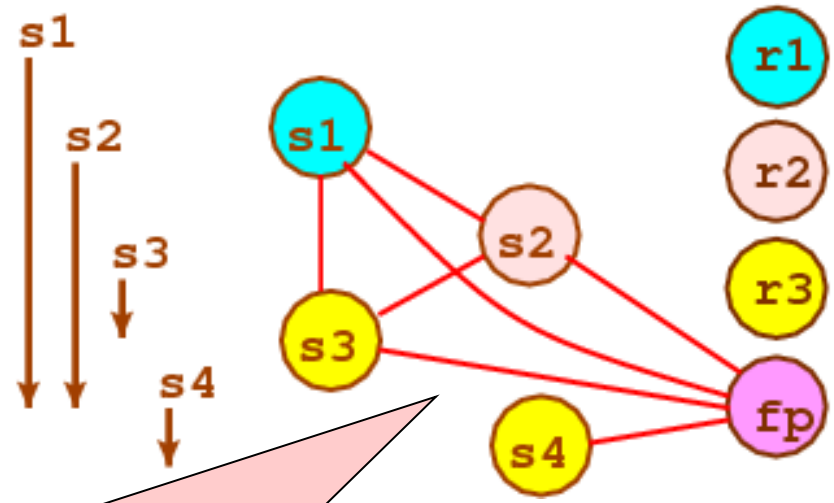
```
i = c+4;   load 8(fp),s1    ! c
           nop
           addi s1,#4,s2
           store s2,4(fp)   ! i
d = c-2;   subi s1,#2,s3
           store s3,12(fp)  ! d
c = c*i;   muli s1,s2,s4
           store s4,8(fp)   ! c
```

# Reg. Alloc. by Graph Coloring Cont.

- **Step 3**: Color the register interference graph with *k* colors, where *k* = #available registers.
  - If not possible: pick a victim s*i* to spill, generate spill code (store after def., reload before use)
    - This may remove some interferences.
      Rebuild the register interference graph + repeat Step 3...

```
i = c+4;  load 8(fp),s1    ! c
          nop
          addi s1,#4,s2
          store s2,4(fp)   ! i
d = c-2;  subi s1,#2,s3
          store s3,12(fp)  ! d
c = c*i;  muli s1,s2,s4
          store s4,8(fp)   ! c
```




**This register interference graph cannot be colored with less than 4 colors, as it contains a 4-clique**

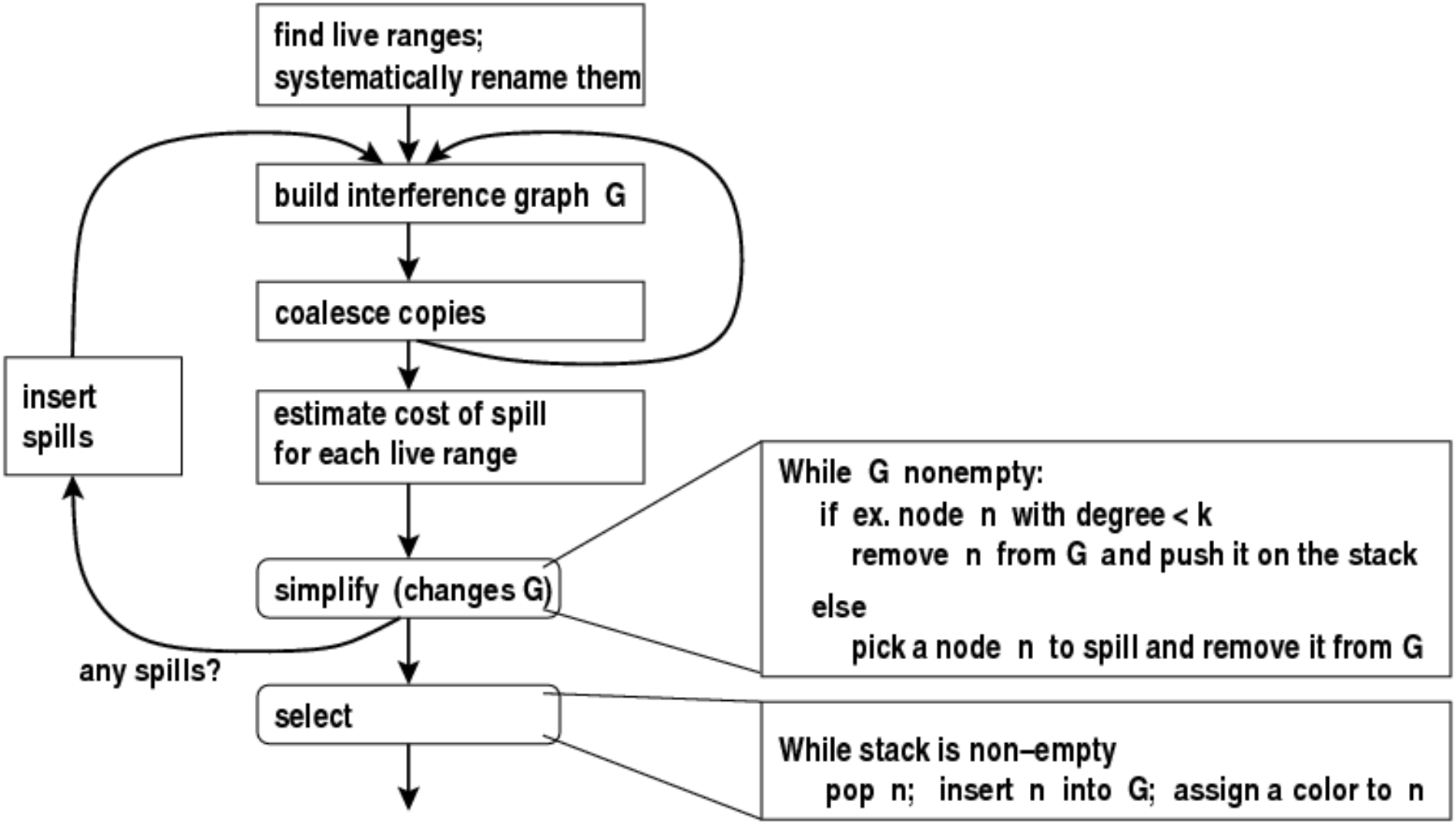
# Coloring a Graph with *k* Colors

- NP-complete for k > 3
- **Chromatic number** $\gamma(G)$ = minimum number of colors to color a graph G
- $\gamma(G) >= c$ if the graph contains a c-clique
  - A *c*-**clique** is a completely connected subgraph of *c* nodes

- **Chaitin's heuristic** (1981):

```
S ← { s1, s2, ... }    // set of spill candidates
while ( S not empty )
    choose some s in S.
    if s has less than k neighbors in the graph
        then // there will be some color left for s:
             delete s (and incident edges) from the graph
        else modify the graph (spill, split, coalesce ... nodes)
             and restart.
// once we arrive here, the graph is empty:
color the nodes greedily in reverse order of removal.
```

# Chaitin's Register Allocator (1981)

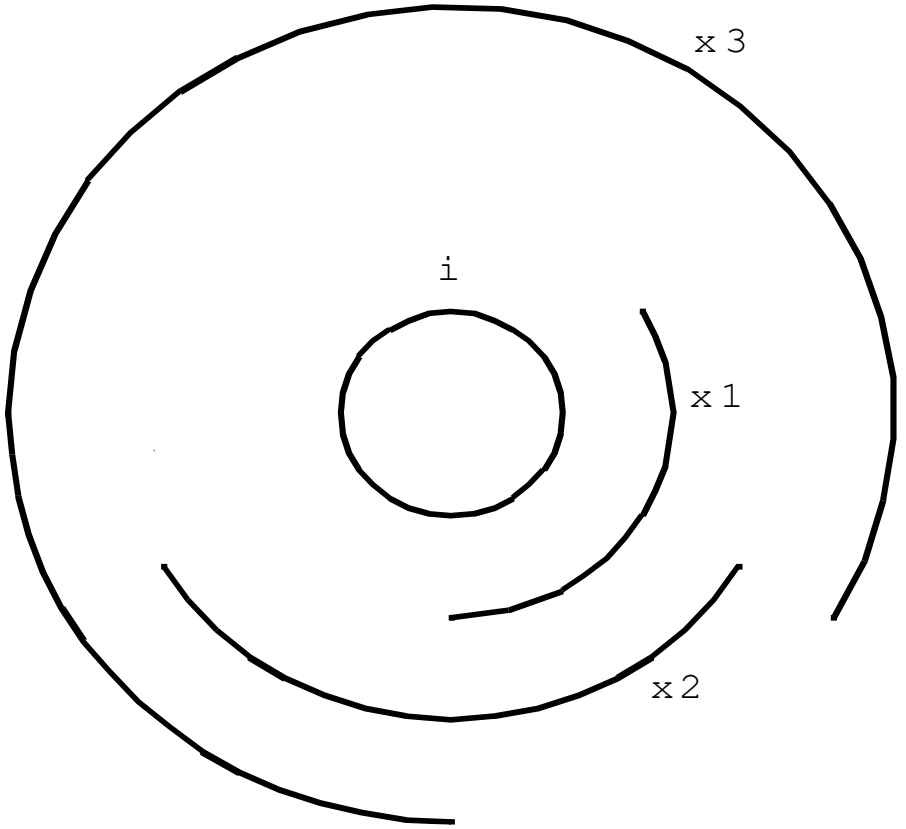


# Register Allocation for Loops (1)

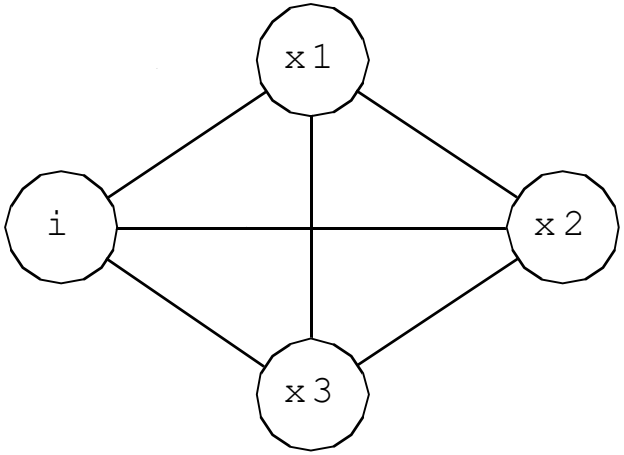
- Interference graphs have some weaknesses:
  - Imprecise information on how and when live ranges interfere.
  - No special consideration is taken of loop variables' live ranges (except when calculating priority).
- Instead, in a cyclic interval graph:
  - The time relationships between the live ranges are explicit.
  - Live ranges are represented for a variable whose live range crosses iteration limits by cyclic intervals.
- Notation for cyclic live intervals for loops:
  - Intervals for loop variables which do not cross the iteration limit are included precisely once.
  - Intervals which cross the iteration limit are represented as an interval pair, *cyclic interval*:
    $([0, t'), [t, t_{end}])$

# Register Allocation for Loops (2)

Circular edge graph
Only 3 interferences at the same time




Traditional interference graph,
all variables interfere, 4 registers needed

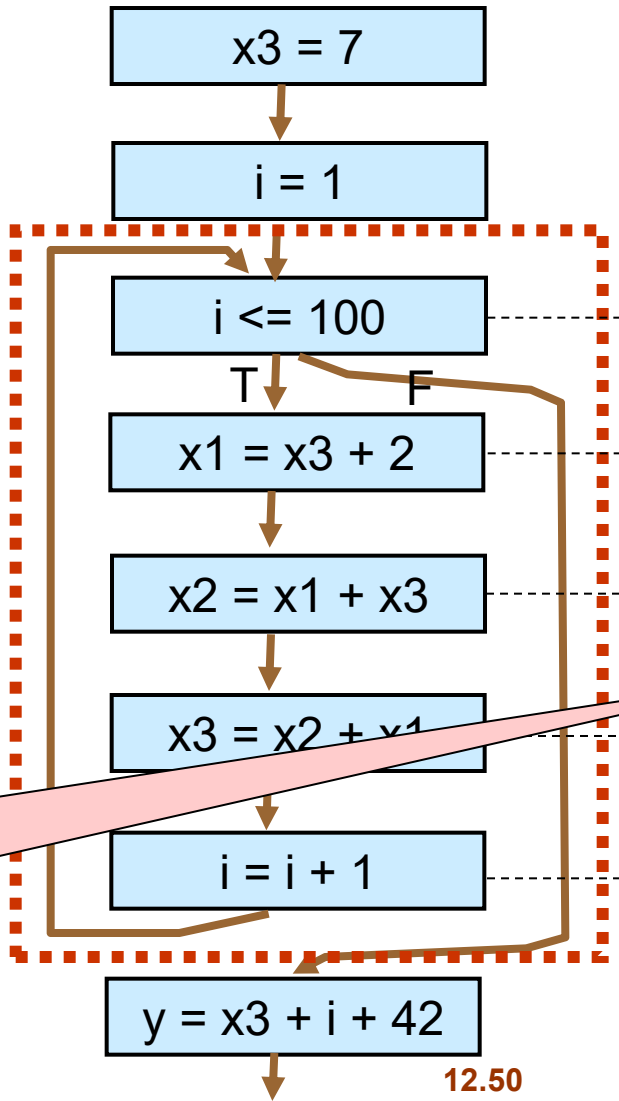


# Register Allocation for Loops (3)

**Example**:

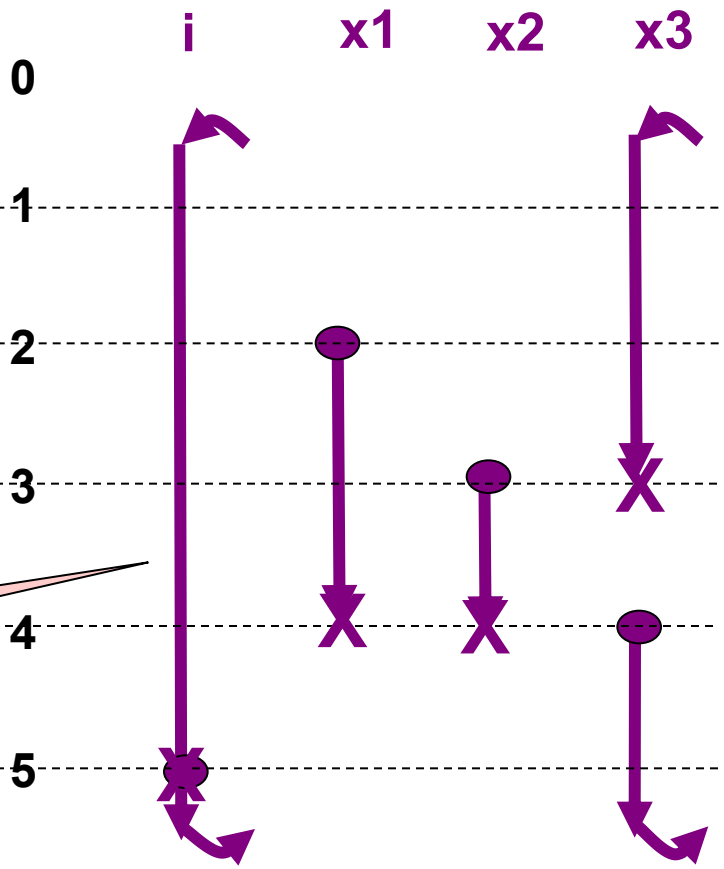
```
x3 = 7
for i = 1 to 100 {
    x1 = x3 + 2
    x2 = x1 + x3
    x3 = x2 + x1
}
y = x3 + 42
```

**Control flow graph**




All variables interfere with each other – need 4 regs?

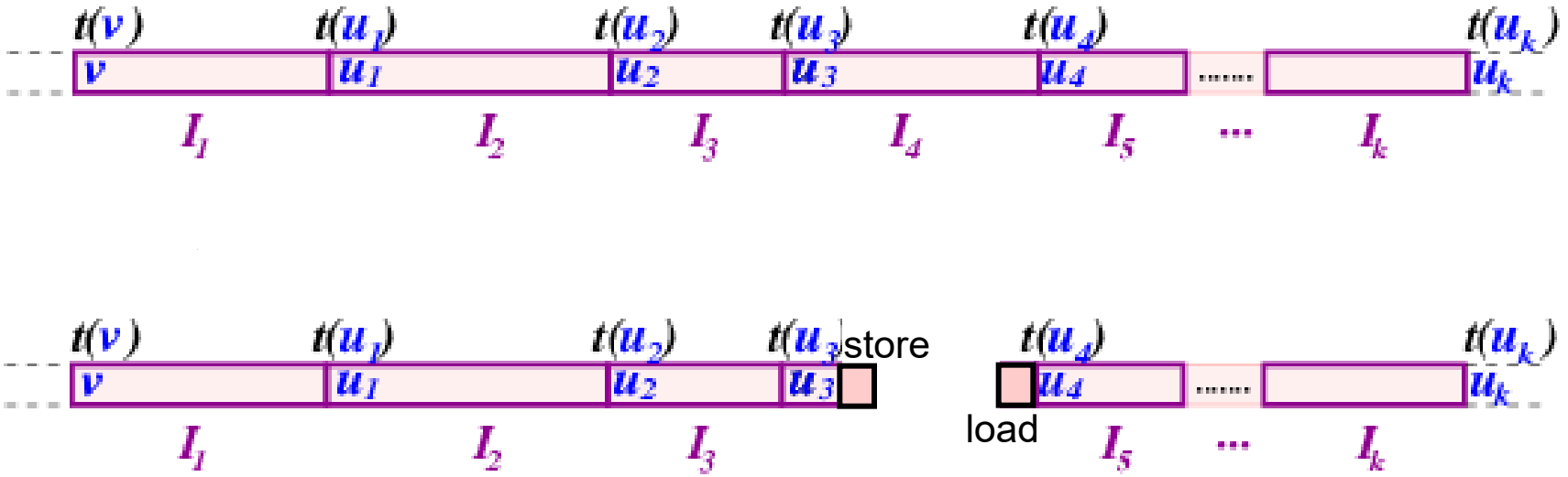
**Live ranges** (loop only):
cyclic intervals
e.g. for i: [0, 5), [5, 6]
x1: [2, 4) x2: [3, 5)
x3: ([0, 3), [4, 6])




At most 3 values live at a time
→ 3 registers sufficient

# Live Range Splitting

- Instead of spilling completely (reload before each use), it may be sufficient to split a live range at one position where register pressure is highest
  - save, and reload once



# Live Range Coalescing/Combining
## (Reduces Register Needs)

- For a copy instruction $s_j \leftarrow s_i$
  - where $s_i$ and $s_j$ do not interfere
  - and $s_i$ and $s_j$ are not rewritten after the copy operation
- Merge $s_i$ and $s_j$:
  - patch (rename) all occurrences of $s_i$ to $s_j$
  - update the register interference graph
- and remove the copy operation.

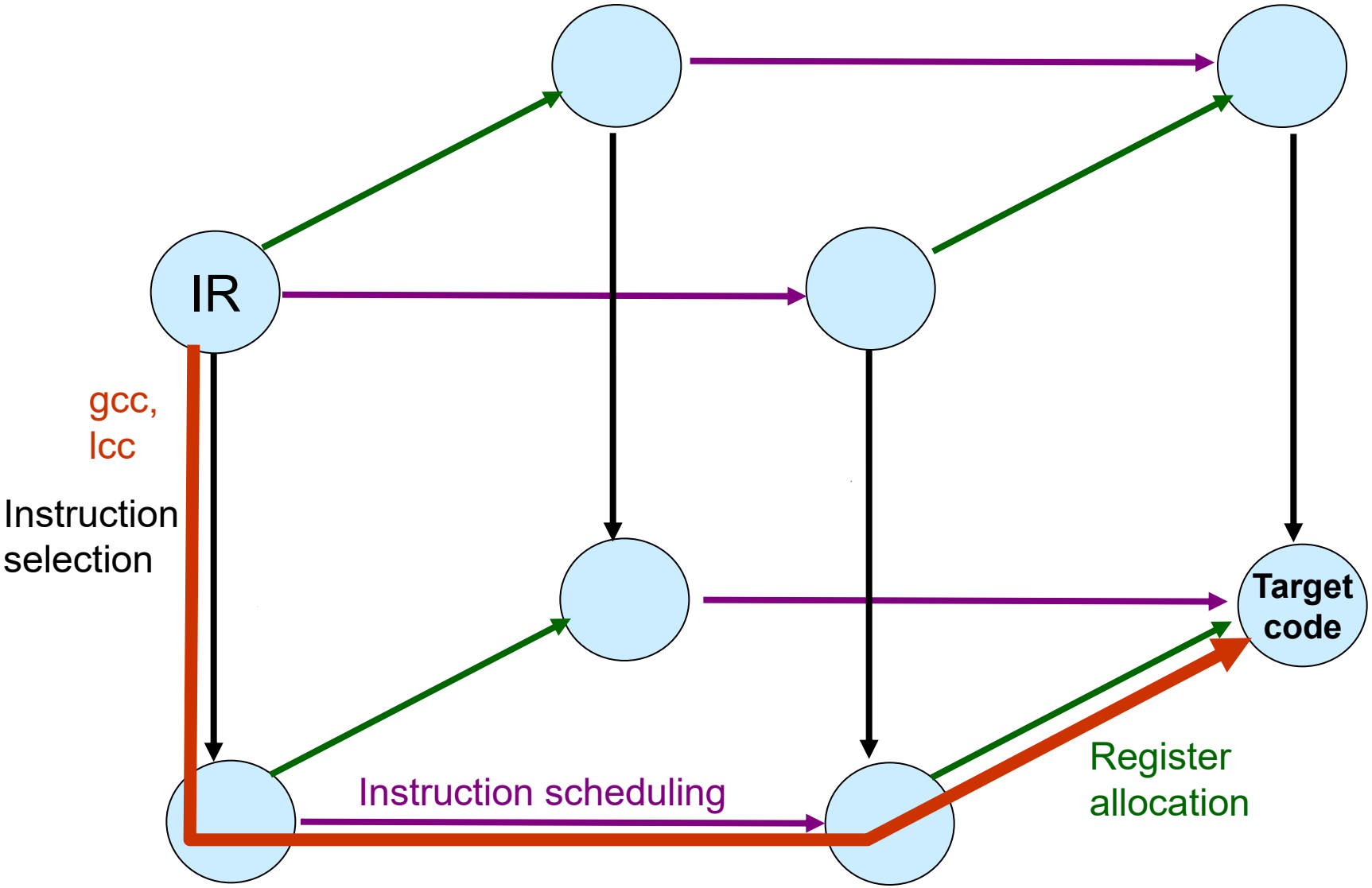
```
s2 ← ...
...
s3 ← s2
...
... s3 ...
```

```
s3 ← ...
...
s3 ← s3
...
... s3 ...
```

LINKÖPINGS UNIVERSITET

# 4. Phase Ordering Problems and Integrated Code Generation

# Phase Ordering Problems

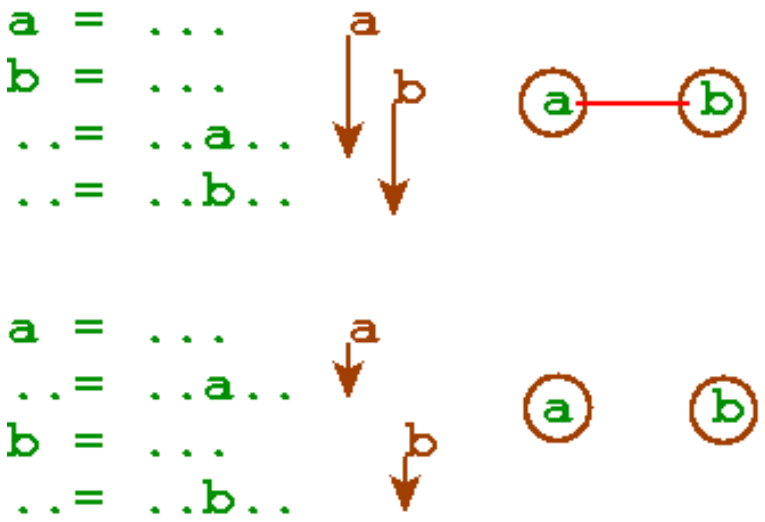


# Phase Ordering Problems (1)

## Instruction scheduling vs. register allocation

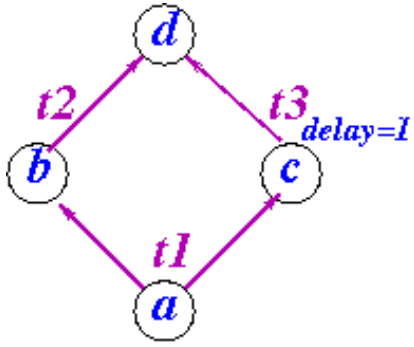
**(a) Scheduling first:**

determines Live-Ranges
→ Register need,
possibly spill-code to be
inserted afterwards




**(b) Register allocation first:**

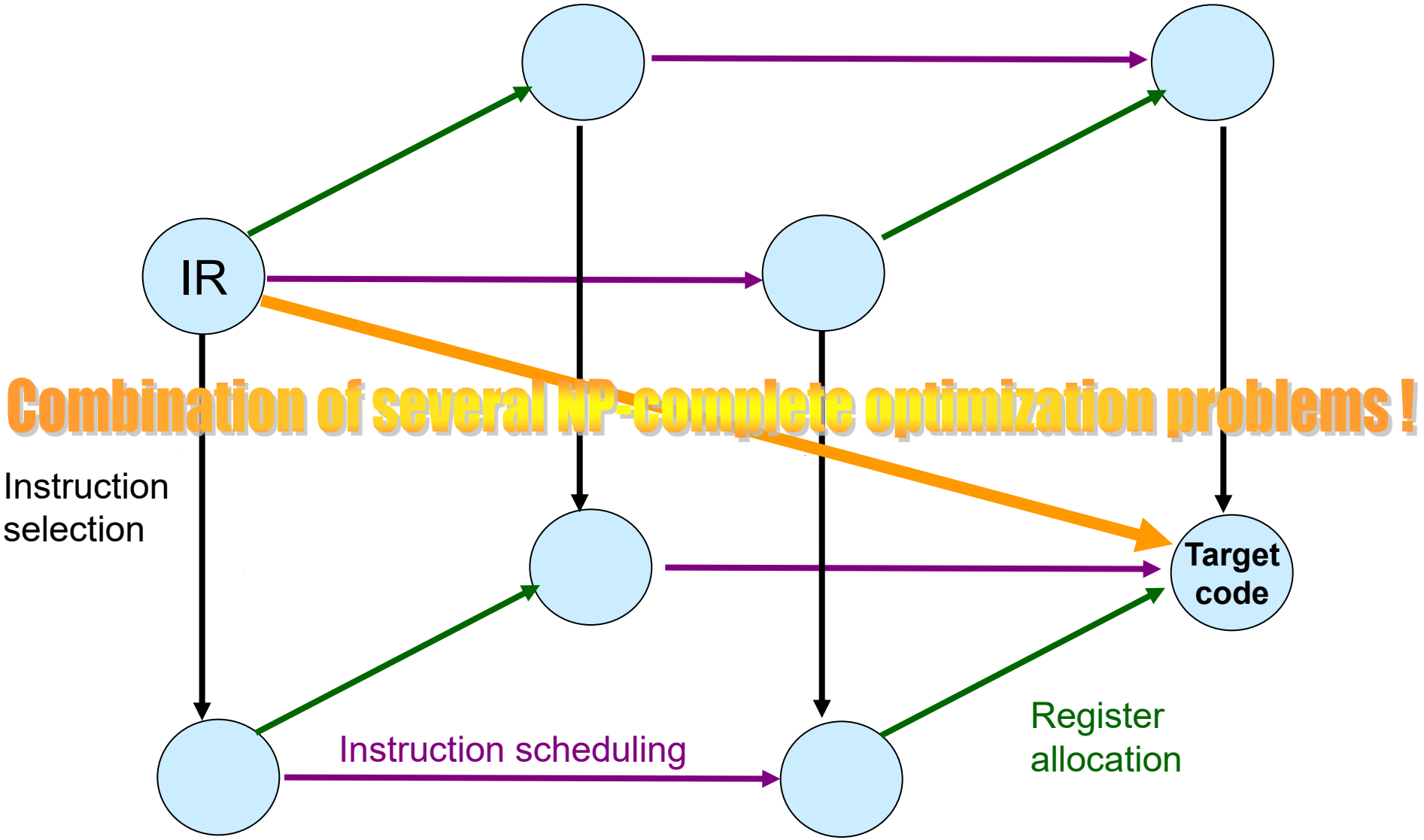
Reuse of same register by different values introduces ”artificial” data dependences
→ constrains scheduler




$S_1$: a b c → d
1 2 3 4 5

$S_2$: a c b d
1 2 3 4

# 5. Integrated Code Generation

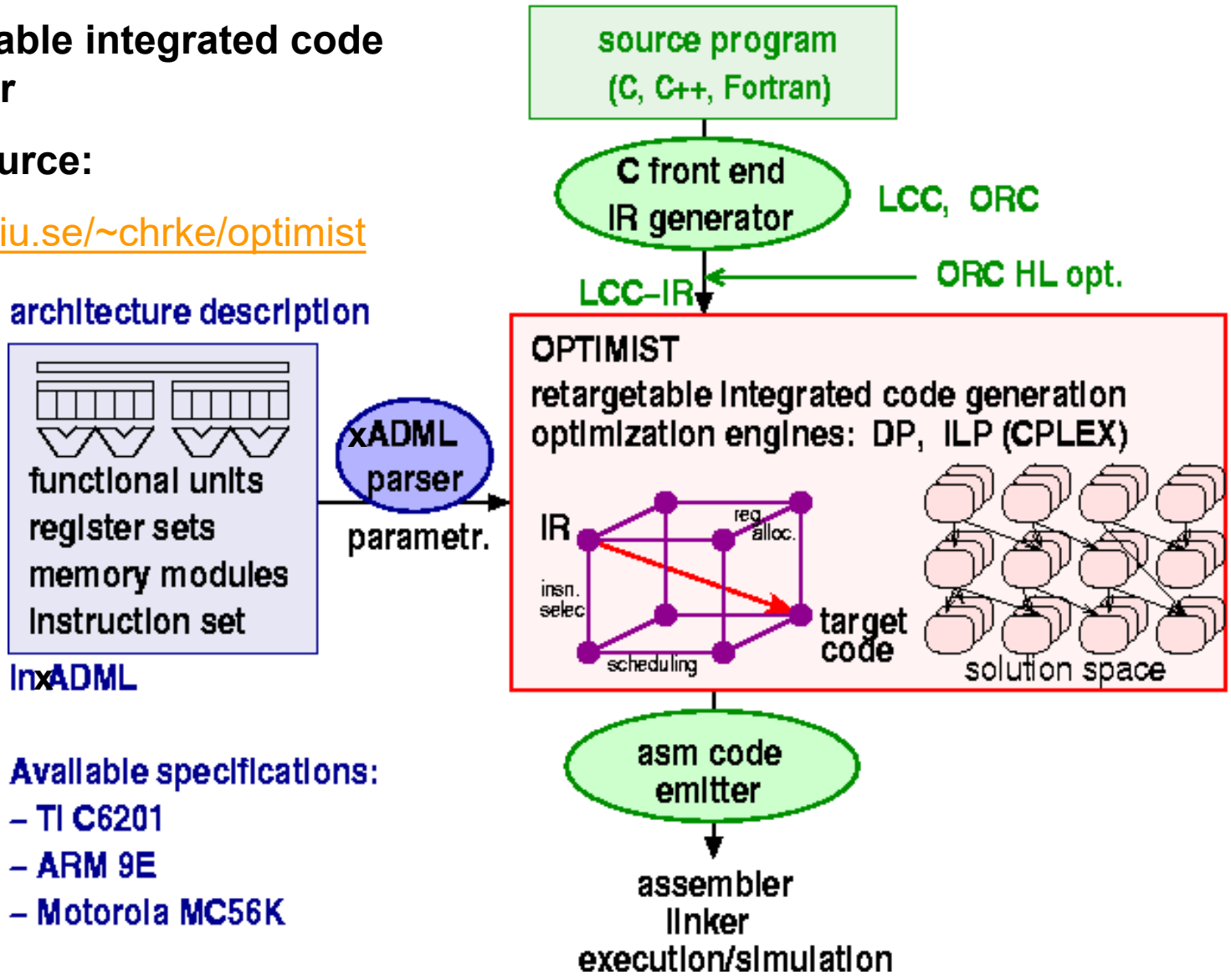


# Project at PELAB (Kessler): OPTIMIST

**Retargetable integrated code generator**

**Open Source:**

www.ida.liu.se/~chrke/optimist




**Available specifications:**

- TI C6201
- ARM 9E
- Motorola MC56K

# Thank you!

- Any questions?
- L13 – Error Management in Compilers and Run-time Systems