# Topic 14: Scheduling

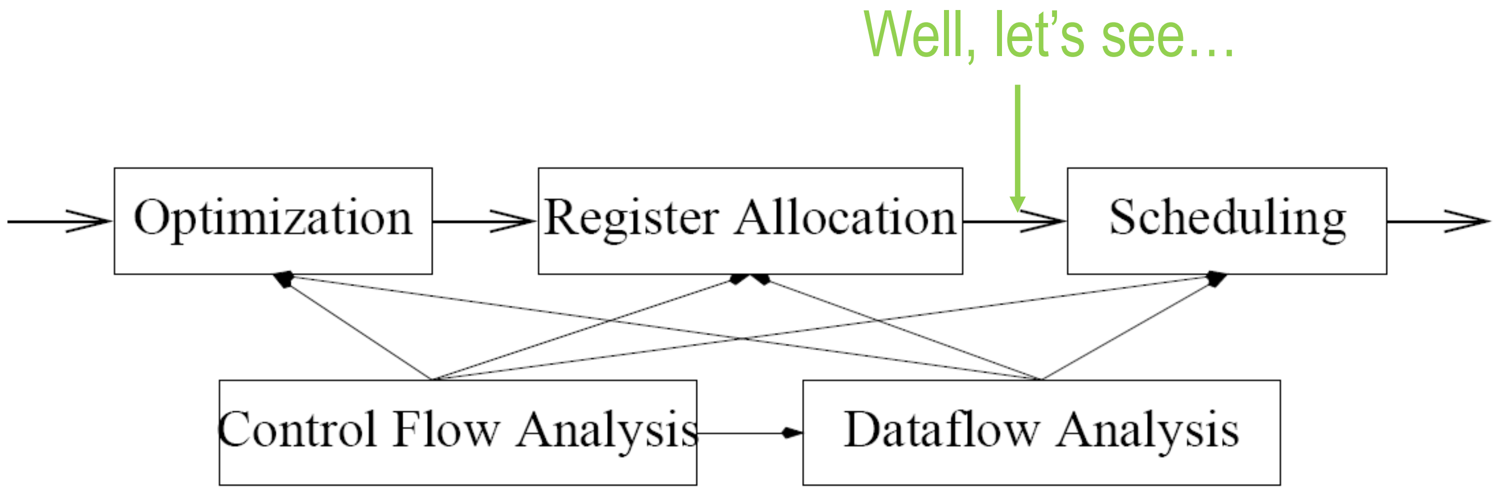
COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

## The Back End



**The Back End:**

1. Maps infinite number of virtual registers to finite number of real registers → *register allocation*
2. Removes inefficiencies introduced by front-end → *optimizer*
3. Removes inefficiencies introduced by programmer → *optimizer*
4. Adjusts pseudo-assembly composition and order to match target machine → *scheduler*

## Motivating example

### Starting point

```
1     r1 = r0 + 0
2     r2 = M[FP + A]
3     r3 = r0 + 4
4     r4 = M[FP + X]

   LOOP:
1     r5 = r3 * r1
2
3     r5 = r2 + r5
4     M[r5]  = r4
5     r1 = r1 + 1
6     BR r1 <= 10, LOOP
```

## Motivating example

Starting point

```
1     r1 = r0 + 0
2     r2 = M[FP + A]
3     r3 = r0 + 4
4     r4 = M[FP + X]

   LOOP:
1     r5 = r3 * r1
2
3     r5 = r2 + r5
4     M[r5]  = r4
5     r1 = r1 + 1
6     BR r1 <= 10, LOOP
```

Multiplication takes 2 cycles

## Motivating example

Instructions take multiple cycles:

fill empty slots with independent instructions!

```
1     r1  =  r0  +  0
2     r2  =  M[FP  +  A]
3     r3  =  r0  +  4
4     r4  =  M[FP  +  X]

   LOOP:
1     r5  =  r3  *  r1
2
3     r5  =  r2  +  r5
4     M[r5]  =  r4
5     r1  =  r1  +  1
6     BR  r1  <=  10,  LOOP
```

```
1     r1  =  r0  +  0
2     r2  =  M[FP  +  A]
3     r3  =  r0  +  4
4     r4  =  M[FP  +  X]

   LOOP:
1     r5  =  r3  *  r1
2     r1  =  r1  +  1
3     r5  =  r2  +  r5
4     M[r5]  =  r4
5     BR  r1  <=  10,  LOOP
```

# Motivating example

Instructions take multiple cycles:

fill empty slots with independent instructions!

what exactly do we mean by “independent”?

```
1     r1 = r0 + 0
2     r2 = M[FP + A]
3     r3 = r0 + 4
4     r4 = M[FP + X]

   LOOP:
1     r5 = r3 * r1
2
3     r5 = r2 + r5
4     M[r5] = r4
5     r1 = r1 + 1
6     BR r1 <= 10, LOOP
```

```
1     r1 = r0 + 0
2     r2 = M[FP + A]
3     r3 = r0 + 4
4     r4 = M[FP + X]

   LOOP:
1     r5 = r3 * r1
2     r1 = r1 + 1
3     r5 = r2 + r5
4     M[r5] = r4
5     BR r1 <= 10, LOOP
```

## Motivating example

When our processor can execute 2 instructions per cycle

```
1    r1 = r0 + 0
2    r2 = M[FP + A]
3    r3 = r0 + 4
4    r4 = M[FP + X]

  LOOP:
1    r5 = r3 * r1
2    r1 = r1 + 1
3    r5 = r2 + r5
4    M[r5] = r4
5    BR r1 <= 10, LOOP
```

## Motivating example

When our processor can execute 2 instructions per cycle:

issue pairs of independent instructions whenever possible

```
1    r1 = r0 + 0
2    r2 = M[FP + A]
3    r3 = r0 + 4
4    r4 = M[FP + X]

 LOOP:
1    r5 = r3 * r1
2    r1 = r1 + 1
3    r5 = r2 + r5
4    M[r5] = r4
5    BR r1 <= 10, LOOP
```

```
1    r1 = r0 + 0    r2 = M[FP + A]
2    r3 = r0 + 4    r4 = M[FP + X]

  LOOP:
1    r5 = r3 * r1   r1 = r1 + 1
2
3    r5 = r2 + r5
4    M[r5] = r4     BR r1 <= 10, LOOP
```

## Motivating example

When our processor can execute 2 instructions per cycle:

issue pairs of independent instructions whenever possible

same notion of "independent"?

```
1    r1 = r0 + 0
2    r2 = M[FP + A]          1    r1 = r0 + 0    r2 = M[FP + A]
3    r3 = r0 + 4             2    r3 = r0 + 4    r4 = M[FP + X]
4    r4 = M[FP + X]

  LOOP:                        LOOP:
1    r5 = r3 * r1            1    r5 = r3 * r1   r1 = r1 + 1
2    r1 = r1 + 1             2
3    r5 = r2 + r5            3    r5 = r2 + r5
4    M[r5] = r4              4    M[r5] = r4     BR r1 <= 10, LOOP
5    BR r1 <= 10, LOOP
```

empty slot reappears – tough luck…

# Instruction Level Parallelism

- Instruction-Level Parallelism (ILP), the concurrent execution of independent assembly instructions. The concurrently executed instructions stem from a single program.
- ILP is a cost effective way to extract performance from programs.
- Exploiting ILP requires global optimization and scheduling.

- Processors can execute several instructions per cycle (Ithanium: up to 6)
- ILP/VLIW: dependencies identified by compiler → instruction bundles
- Super-Scalar: dependencies identified by processor (instruction windows)

Advantages / Disadvantages?

# Instruction Level Parallelism

- Instruction-Level Parallelism (ILP), the concurrent execution of independent assembly instructions. The concurrently executed instructions stem from a single program.
- ILP is a cost effective way to extract performance from programs.
- Exploiting ILP requires global optimization and scheduling.

- Processors can execute several instructions per cycle (Ithanium: up to 6)
- ILP/VLIW: dependencies identified by compiler → instruction bundles
- Super-Scalar: dependencies identified by processor (instruction windows)

Advantages / Disadvantages?

Possible synthesis:

- have compiler take care of register-carried dependencies
- let processor take care of memory-carried dependencies: exploit dynamic resolution of memory aliasing
- use register renaming, register bypassing, out-of-order execution, speculation (branch prediction) to keep all execution units busy

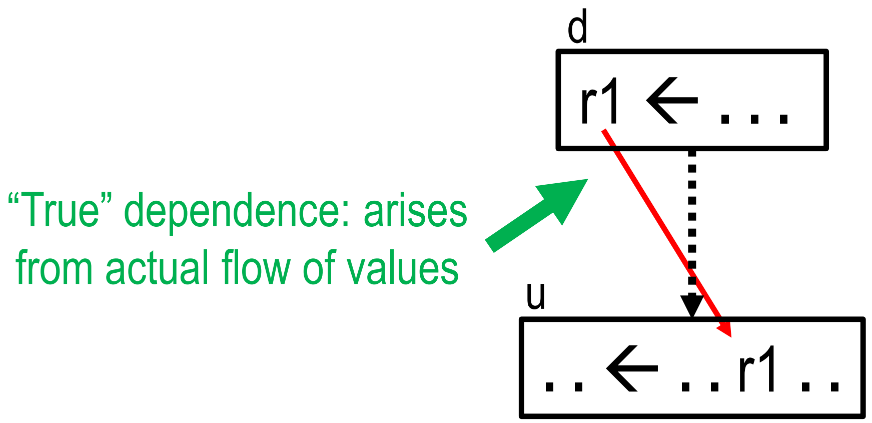
## Scheduling constraints

- **Data dependencies**
  - ordering between instructions that arises from the flow of data
- **Control dependencies**
  - ordering between instructions that arises from flow of control
- **Resource constraints**
  - processors have limited number of functional units
  - not all functional units can execute all instructions (Floating point unit versus Integer-ALU, …)
  - only limited number of instructions can be issued in one cycle
  - only a limited number of register read/writes can be done concurrently

## Data Dependences

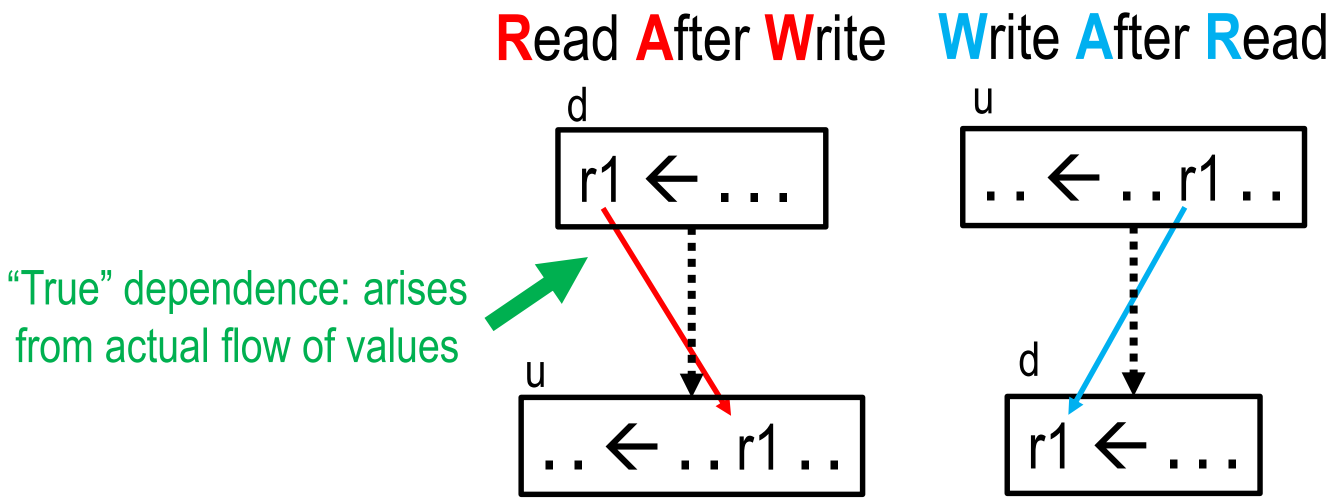
- A *data dependence* is a constraint on scheduling arising from the flow of data between two instructions. Types:
  - RAW: An instruction $u$ is *flow-dependent* on a preceding instruction $d$ if $u$ consumes a value computed by $d$.

Read After Write



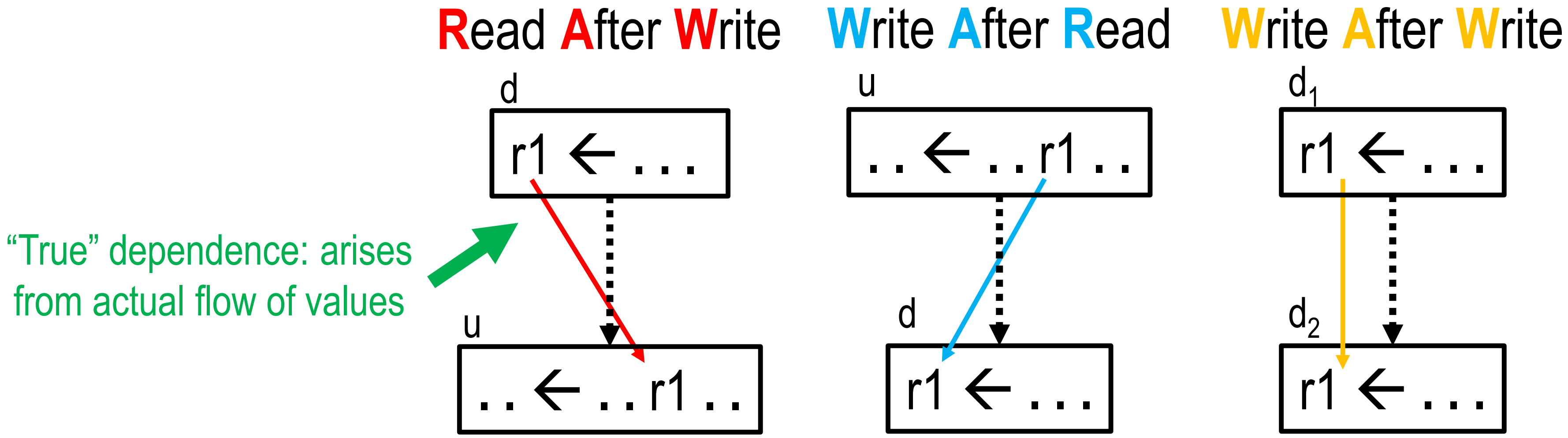
## Data Dependences

- A *data dependence* is a constraint on scheduling arising from the flow of data between two instructions. Types:
  - RAW: An instruction $u$ is *flow-dependent* on a preceding instruction $d$ if $u$ consumes a value computed by $d$.
  - WAR: An instruction $d$ is *anti-dependent* on a preceding instruction $u$ if $d$ writes to a location read by $u$.



## Data Dependences

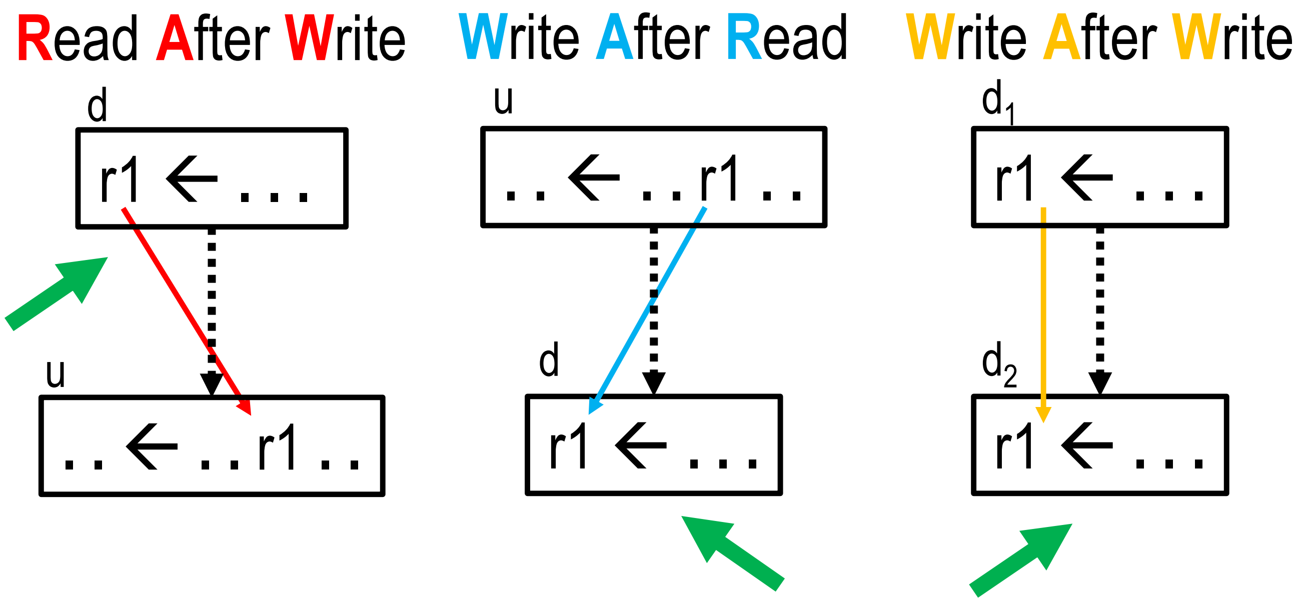
- A *data dependence* is a constraint on scheduling arising from the flow of data between two instructions. Types:
  - RAW: An instruction $u$ is *flow-dependent* on a preceding instruction $d$ if $u$ consumes a value computed by $d$.
  - WAR: An instruction $d$ is *anti-dependent* on a preceding instruction $u$ if $d$ writes to a location read by $u$.
  - WAW: An instruction $d_2$ is *output-dependent* on a preceding instruction $d_1$ if $d_1$ writes to a location also written by $d_2$.



# Data Dependences

- A *data dependence* is a constraint on scheduling arising from the flow of data between two instructions. Types:
  - RAW: An instruction $u$ is *flow-dependent* on a preceding instruction $d$ if $u$ consumes a value computed by $d$.
  - WAR: An instruction $d$ is *anti-dependent* on a preceding instruction $u$ if $d$ writes to a location read by $u$.
  - WAW: An instruction $d_2$ is *output-dependent* on a preceding instruction $d_1$ if $d_1$ writes to a location also written by $d_2$.
- Types of data:
  - Register dependence
  - Memory dependence

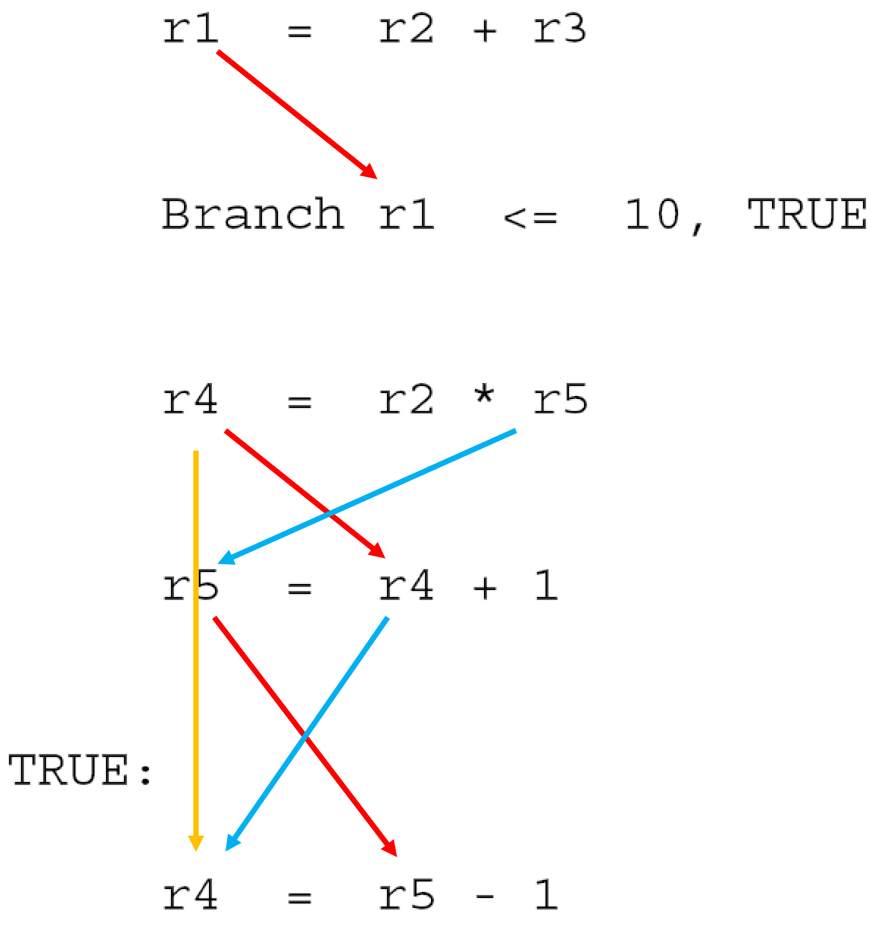
“True” dependence: arises from actual flow of values



“False”/”name” dependences: arise from reuse of location; can often be avoided by (dynamic) renaming

# Eliminating false dependencies

WAW and WAR dependencies can often eliminated by register renaming…

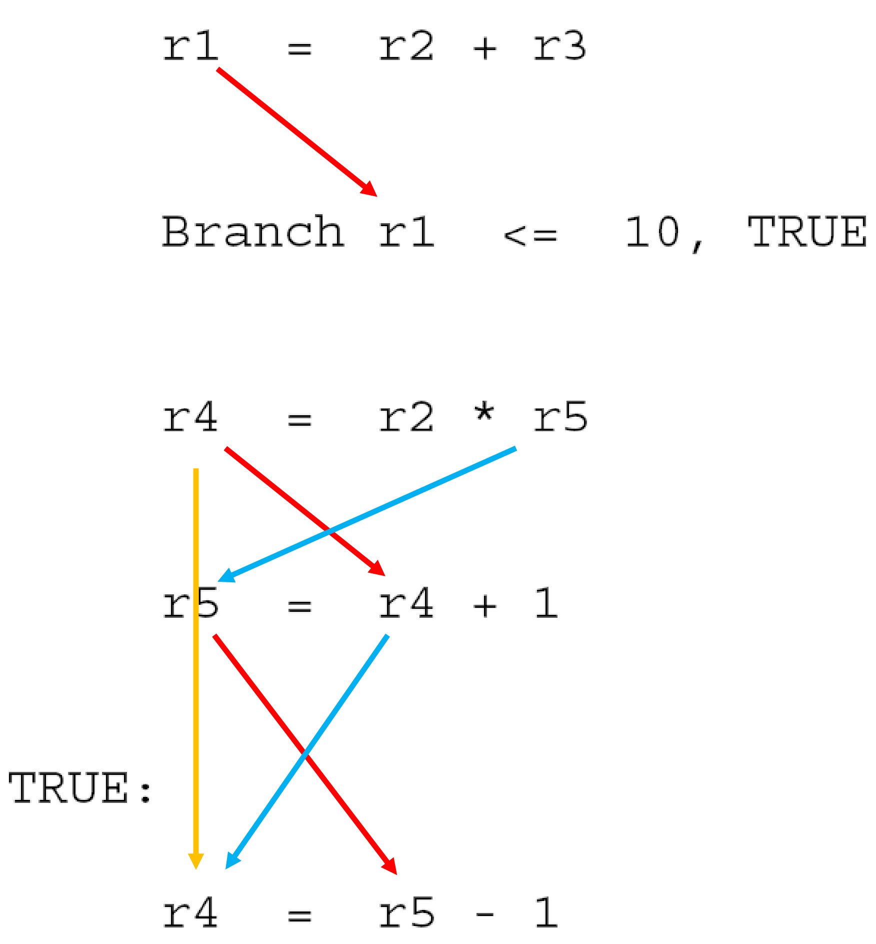


```
      r1     =   r2  +  r3

      Branch r1   <=   10,  TRUE

      r4     =   r2  *  r5

      r5     =   r4  +  1

TRUE:

      r4     =   r5  -  1
```

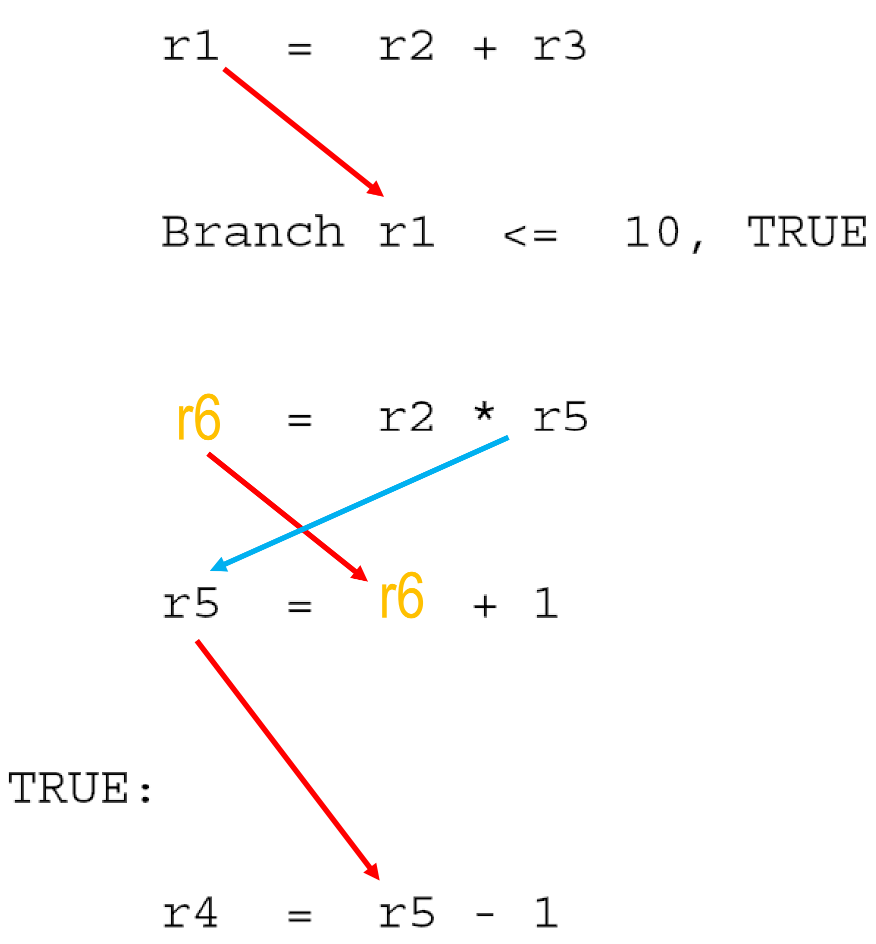
… at the cost of adding registers...

# Eliminating false dependencies

WAW and WAR dependencies can often eliminated by register renaming…



```
      r1   =   r2 + r3

      Branch r1  <=   10, TRUE

      r4   =   r2 * r5

      r5   =   r4 + 1

TRUE:
      r4   =   r5 - 1
```

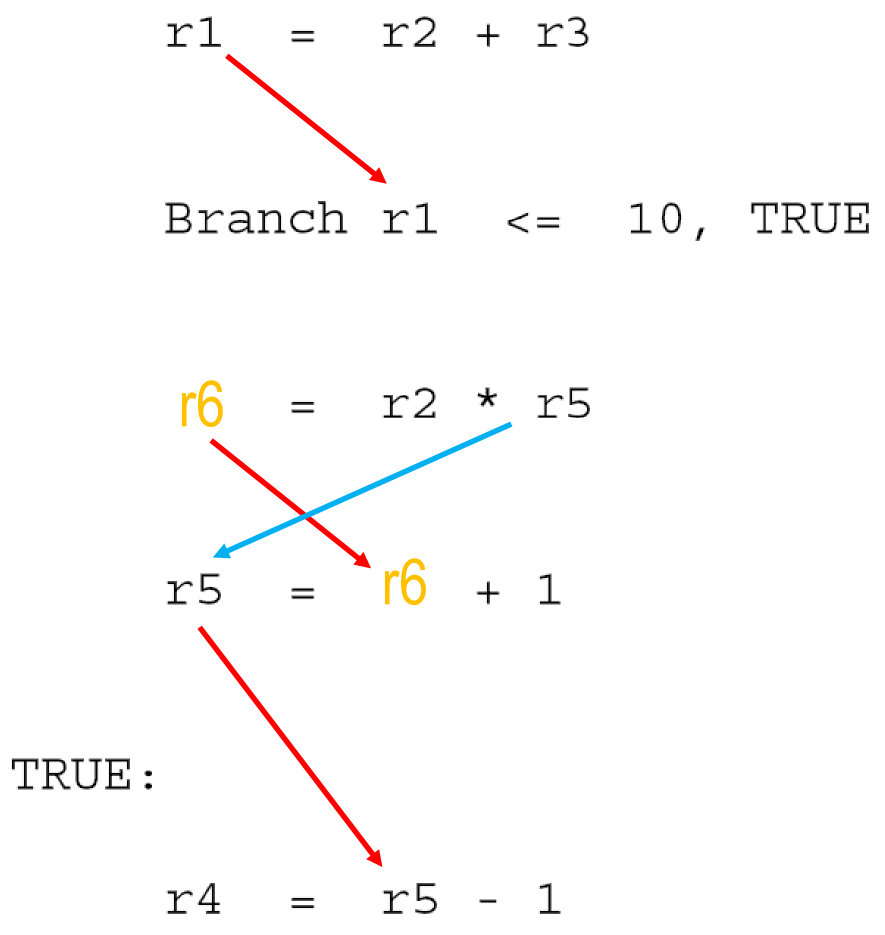


```
      r1   =   r2 + r3

      Branch r1  <=   10, TRUE

      r6   =   r2 * r5

      r5   =   r6 + 1

TRUE:
      r4   =   r5 - 1
```

… at the cost of adding registers...

# Eliminating false dependences

WAR dependencies can often be replaced by RAW dependencies

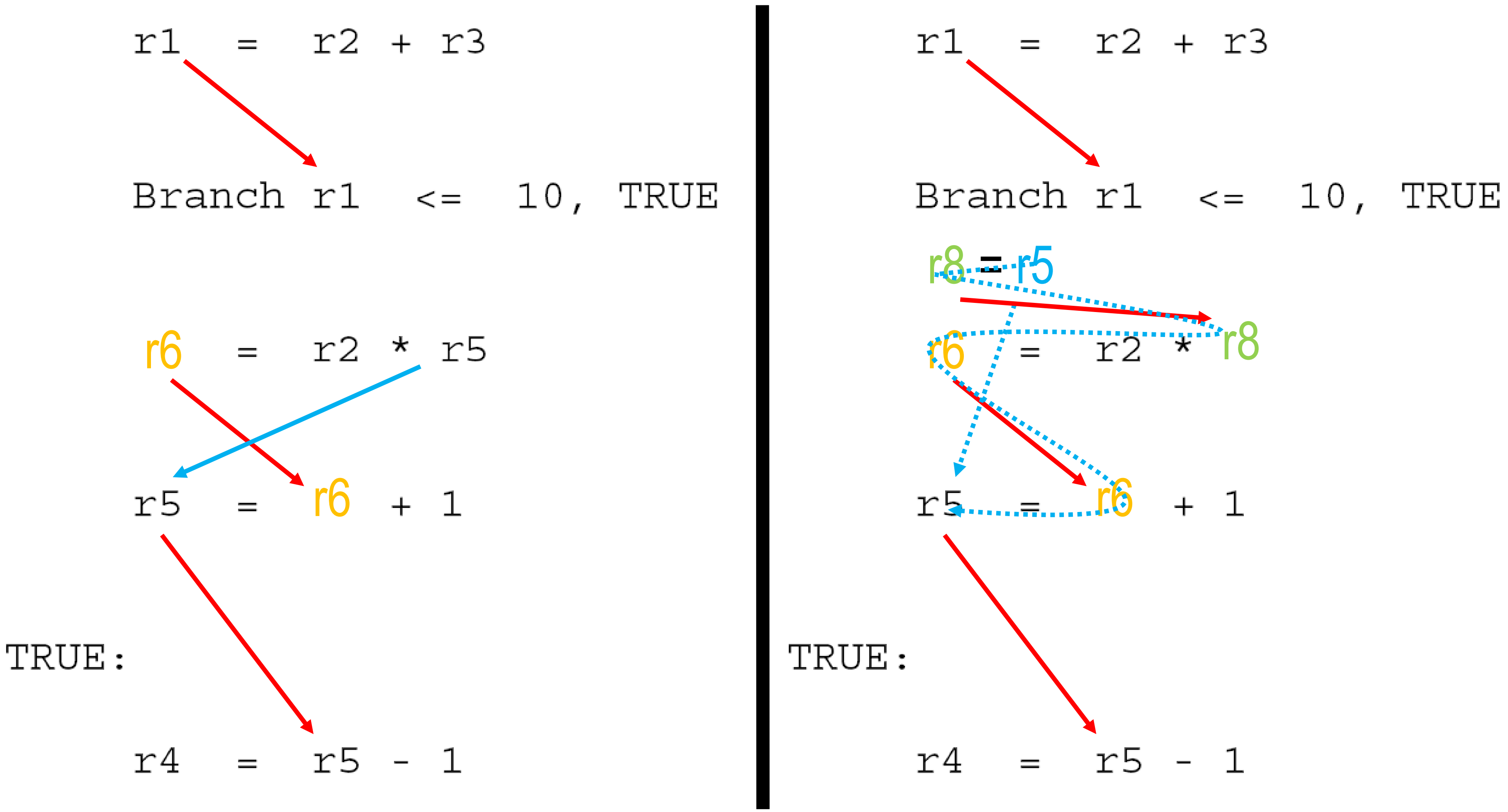


```
      r1     =    r2  +  r3

      Branch r1    <=    10,  TRUE

      r6     =    r2  *  r5

      r5     =    r6  +  1

TRUE:

      r4     =    r5  -  1
```

... at the price of using yet another register, and a (move) instruction ….

# Eliminating false dependences

WAR dependencies can often be replaced by RAW dependencies



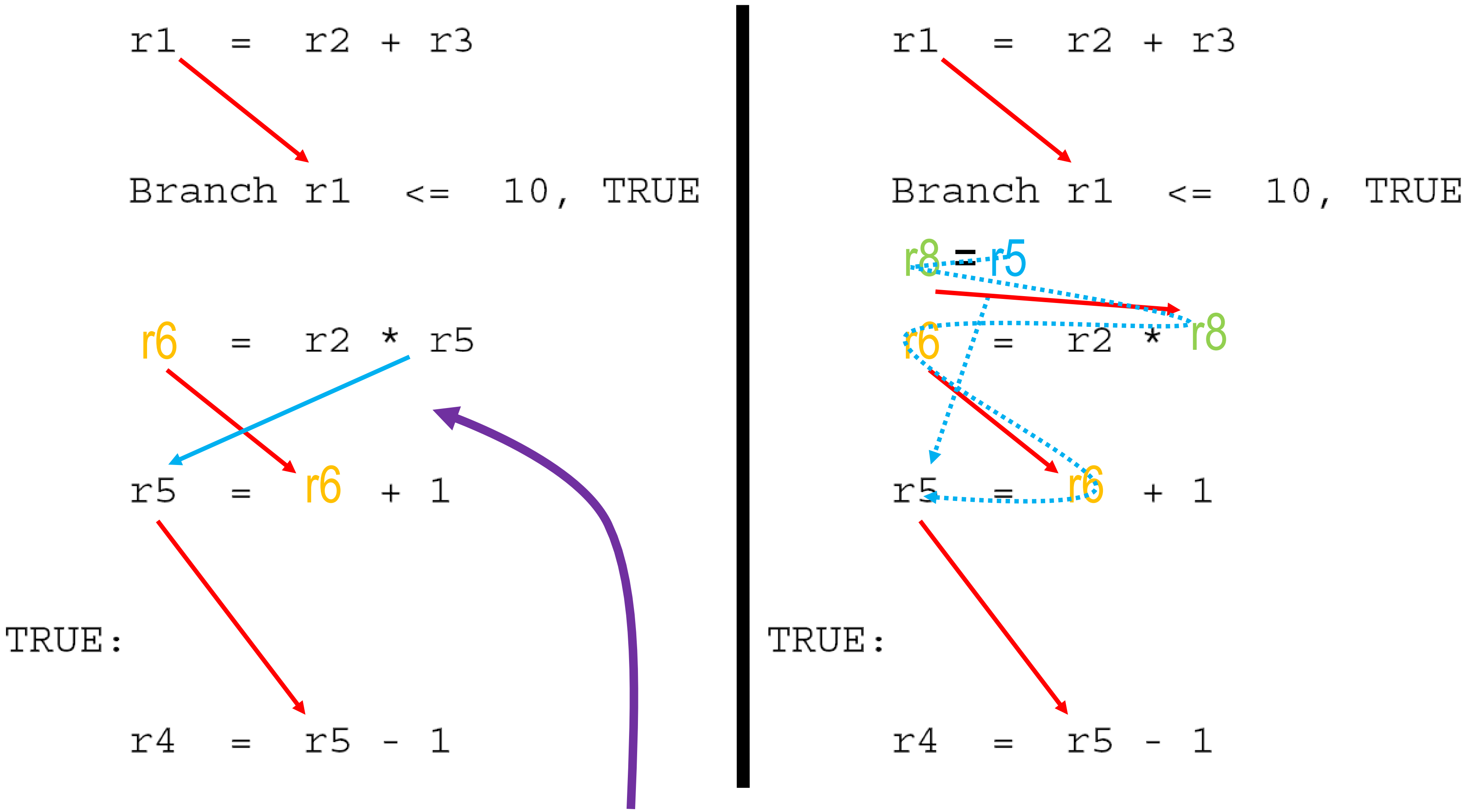
```
   r1   =   r2 + r3

   Branch r1   <=   10, TRUE

   r6   =   r2 * r5

   r5   =   r6  + 1

TRUE:

   r4   =   r5 - 1
```

```
   r1   =   r2 + r3

   Branch r1   <=   10, TRUE
   r8 = r5

   r6   =   r2 * r8

   r5   =   r6  + 1

TRUE:

   r4   =   r5 - 1
```

... at the price of using yet another register, and a (move) instruction ….

# Eliminating false dependences

WAR dependencies can often be replaced by RAW dependencies



```
     r1   =   r2 + r3

     Branch r1  <=   10, TRUE

     r6   =   r2 * r5

     r5   =   r6  + 1

TRUE:

     r4   =   r5 - 1
```

```
     r1   =   r2 + r3

     Branch r1  <=   10, TRUE
     r8 = r5

     r6   =   r2 * r8

     r5   =   r6  + 1

TRUE:

     r4   =   r5 - 1
```

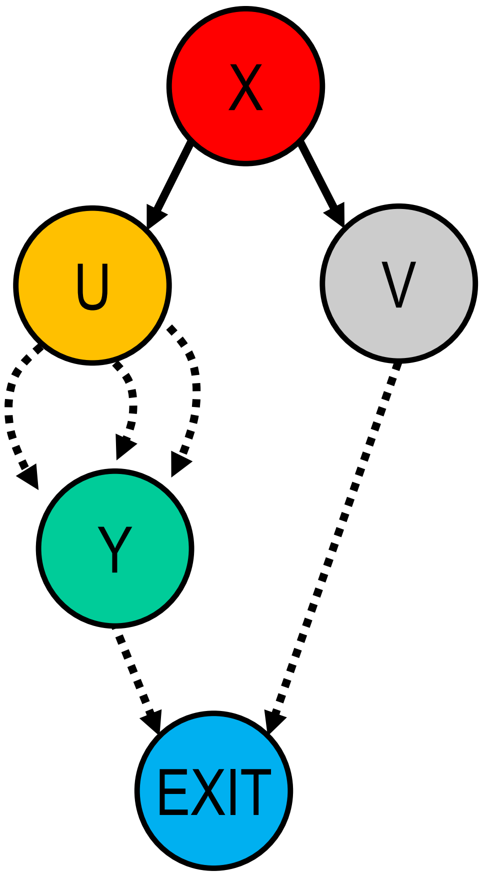
In fact, the WAR dependence on r5 already is respected/implied by the RAW dependence on r6 here!

... at the price of using yet another register, and a (move) instruction ....

# Control Dependence

Node **y** is control dependent on **x** if

- **x** is a branch, with successors **u**, v
- **y** post-dominates **u** in the CFG: each path from **u** to **EXIT** includes **y**
- **y** does not post-dominate v in the CFG: there is a path from v to **EXIT** that avoids **y**



Schedule must respect control dependences: don't move instructions past their control dependence ancestors!

## Dependences

**Latency**

- Amount of time after the execution of an instruction that its result is ready.
- An instruction can have more than one latency! eg load, depending on cache-hit/miss

**Data Dependence Graph**

- A *data dependence graph* consists of instructions and a set of directed data dependence edges among them in which each edge is labeled with its latency and type of dependence.
- Scheduling (code motion) must respect dependence graph.

**Program dependence graph**: overlay of data dependence graph with control dependencies (two kinds of edges)

## Hardware Scheduling

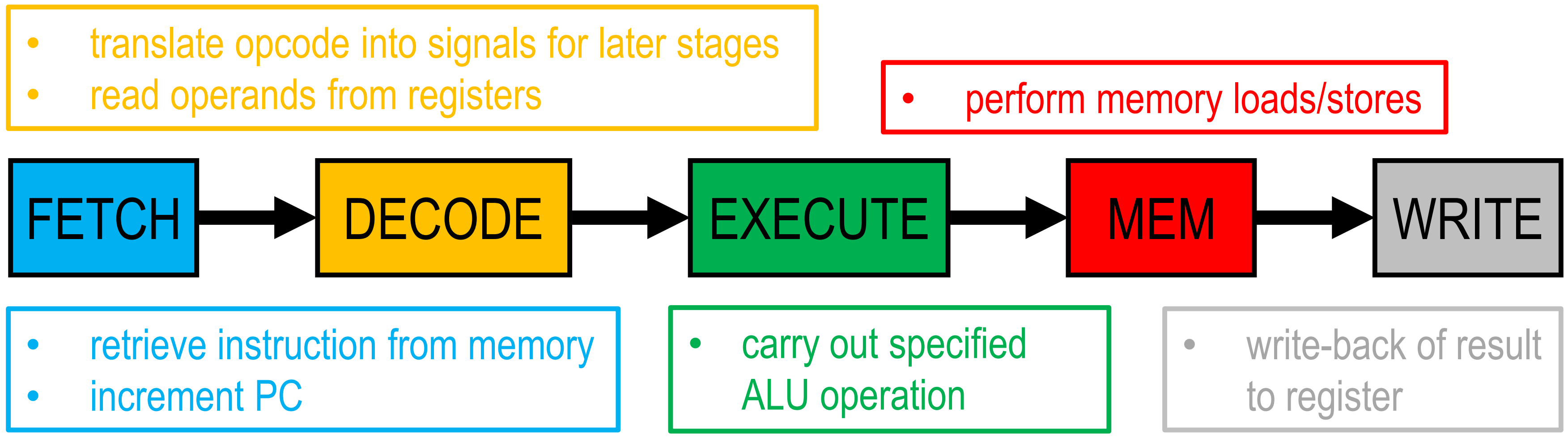
**Machines can also do scheduling...**

- hardware schedulers process code after it has been fetched
- hardware finds independent instructions
- works with legacy architectures (found in x86 / Pentium)
- program knowledge more precise at run-time - memory dependence
  - control flow resolved

**But compiler still important.**

- Hardware schedulers have a small window.
- Hardware complexity increases.
- Hardware does not benefit directly from compiler optimization.

# RISC-style processor pipeline



• translate opcode into signals for later stages
• read operands from registers
• perform memory loads/stores
FETCH
DECODE
EXECUTE
MEM
WRITE
• retrieve instruction from memory
• increment PC
• carry out specified ALU operation
• write-back of result to register

<u>Modern processors:</u>

- many more stages (up to 20-30)
- different stages take different number of cycles per instruction
- some (components of) stages duplicated, eg super-scalar

<u>Common characteristics: resource constraints</u>

- each stage can only hold a fixed number of instruction per cycle
- but: instructions can be in-flight concurrently (pipeline – more later)
- register bank can only serve small number of reads/writes per cycle

## Goal of scheduling

Construct a sorted version of the dependence graph that

- produces the same result as the sequential program: respect dependencies, latencies
- obeys the resource constrains
- minimizes execution time (other metrics possible)

## Goal of scheduling

Construct a sorted version of the dependence graph that

- produces the same result as the sequential program: respect dependencies, latencies
- obeys the resource constrains
- minimizes execution time (other metrics possible)

Solution formulated as a table that indicates the issue cycle of each instruction:

| Cycle | **Resoure 1** | **Resource 2** | **…** | **Resource n** |
|---|---|---|---|---|
| 1 | 1 | | | 2 |
| 2 | | 3 | | 4 |
| 3 | | | | |
| : | | | | |

Even simplified version of the scheduling problem are typically NP-hard

→ heuristics

## A classification of scheduling heuristics

Schedule within a basic block (local)

- instructions cannot move past basic block boundaries
- schedule covers only one basic block

Example technique: (priority) **list scheduling**

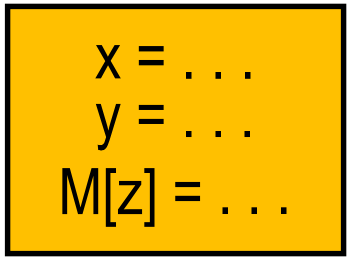
```
x = . . .
y = . . .
M[z] = . . .
```

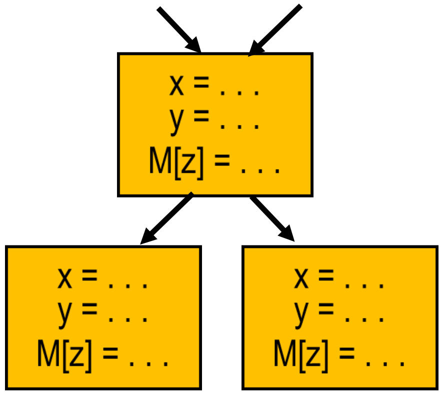
# A classification of scheduling heuristics

Schedule within a basic block (local)

- instructions cannot move past basic block boundaries
- schedule covers only one basic block

Example technique: (priority) **list scheduling**



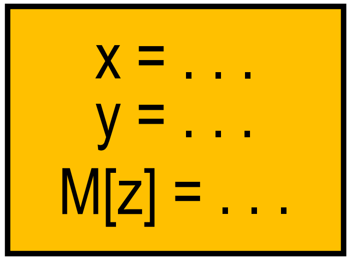


Scheduling across basic blocks (global)

- instructions move past basic block boundaries
- schedule typically covers a (frequently executed ) trace

Example technique: **trace scheduling**

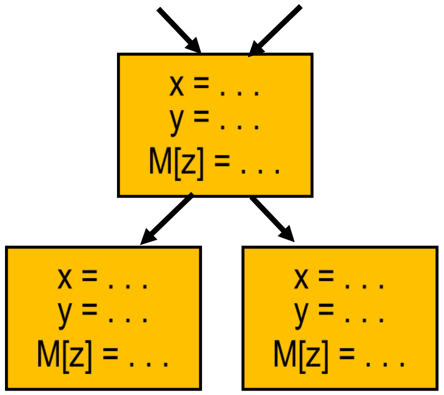
# A classification of scheduling heuristics



Schedule within a basic block (local)

- instructions cannot move past basic block boundaries
- schedule covers only one basic block

Example technique: (priority) **list scheduling**



Scheduling across basic blocks (global)

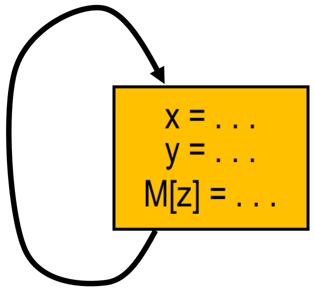
- instructions move past basic block boundaries
- schedule typically covers a (frequently executed ) trace

Example technique: **trace scheduling**

Loop scheduling

- instructions cannot move past basic block boundaries
- each schedule covers body of a loop
- exploits/reflects pipeline structure of modern processors

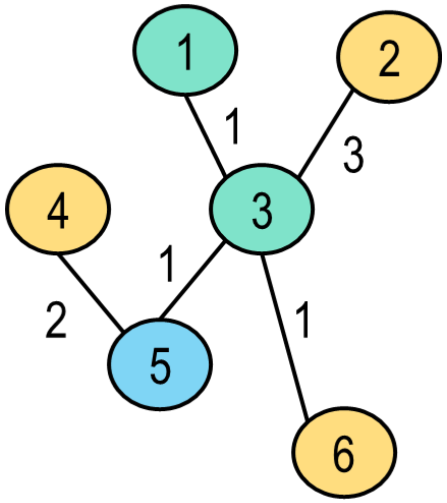
Example technique: **SW pipelining, modulo scheduling**



## Local scheduling: list scheduling

Advantage: can disregard control dependencies

**Input**:
- data dependence graph of straight-line code, annotated with (conservative) latencies
- instruction forms annotated with suitable type of Functional Units
- #available Functional Units of each type

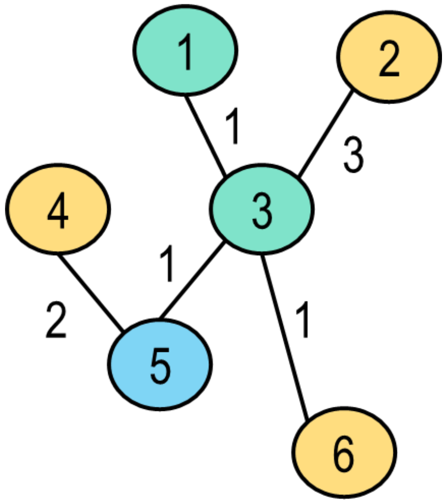


| Integer-ALU | FP | MEM |
|---|---|---|
| 2 | 1 | 1 |

## Local scheduling: list scheduling

Advantage: can disregard control dependencies

**Input**:
- data dependence graph of straight-line code, annotated with (conservative) latencies
- instruction forms annotated with suitable type of Functional Units
- #available Functional Units of each type



| Integer-ALU | FP | MEM |
|---|---|---|
| 2 | 1 | 1 |

**Output**: cycle-accurate assignment of instructions to functional units

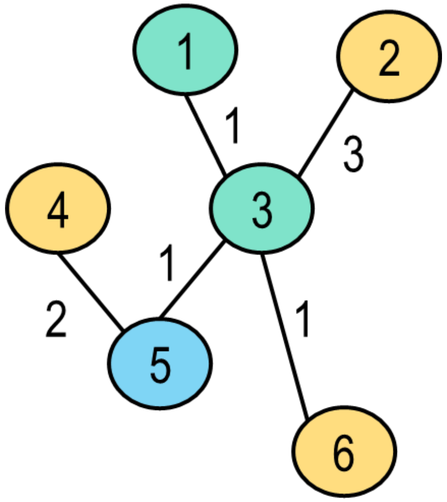
| Cycle | ALU1 | ALU2 | FP | MEM |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |

Can be refined for pipelined architectures, where latency != reservation period for FU

## Local scheduling: list scheduling

Advantage: can disregard control dependencies

**Input**:
- data dependence graph of straight-line code, annotated with (conservative) latencies
- instruction forms annotated with suitable type of Functional Units
- #available Functional Units of each type



| Integer-ALU | FP | MEM |
|---|---|---|
| 2 | 1 | 1 |

**Output**: cycle-accurate assignment of instructions to functional units

| Cycle | ALU1 | ALU2 | FP | MEM |
|---|---|---|---|---|
| 1 | 1 | | | 2 |
| 2 | | | | |
| 3 | | | | |
| 4 | | 3 | | 4 |
| 5 | | | | |
| 6 | | | 5 | 6 |

Can be refined for pipelined architectures, where latency != reservation period for FU

## List scheduling: algorithm (sketch)

1. Insert nodes that have no predecessors into **queue**
2. Start with cycle count **c**=1

## List scheduling: algorithm (sketch)

1. Insert nodes that have no predecessors into **queue**
2. Start with cycle count **c**=1
3. While queue not empty:
   - select an instruction i from the **queue** such that all predecessors were scheduled "sufficiently long ago" (latency information)

priority: e.g. length of path to EXIT, maybe weighted by latency of RAW (+WAW/WAR?) deps

## List scheduling: algorithm (sketch)

1. Insert nodes that have no predecessors into **queue**
2. Start with cycle count **c**=1
3. While queue not empty:
   - select an instruction **i** from the **queue** such that all predecessors were scheduled "sufficiently long ago" (latency information)
   - if a functional unit **u** for **i** is available:
     - insert **i** in (**c**, **u**), and remove it from the **queue**
     - insert any successor of **i** into **queue** for which all predecessors have now been scheduled

priority: e.g. length of path to EXIT, maybe weighted by latency of RAW (+WAW/WAR?) deps

# List scheduling: algorithm (sketch)

1. Insert nodes that have no predecessors into **queue**
2. Start with cycle count **c**=1
3. While queue not empty:
   - select an instruction **i** from the **queue** such that all predecessors were scheduled "sufficiently long ago" (latency information)
   - if a functional unit **u** for **i** is available:
     - insert **i** in (**c**, **u**), and remove it from the **queue**
     - insert any successor of **i** into **queue** for which all predecessors have now been scheduled
   - if no functional unit is available for **i**, select another instruction

priority: e.g. length of path to EXIT, maybe weighted by latency of RAW (+WAW/WAR?) deps

## List scheduling: algorithm (sketch)

1. Insert nodes that have no predecessors into **queue**
2. Start with cycle count c=1
3. While queue not empty:
   - select an instruction i from the **queue** such that all predecessors were scheduled "sufficiently long ago" (latency information)
   - if a functional unit u for i is available:
     - insert i in (c, u), and remove it from the **queue**
     - insert any successor of i into **queue** for which all predecessors have now been scheduled
   - if no functional unit is available for i, select another instruction
   - if no instruction from the queue was scheduled, increment c

priority: e.g. length of path to EXIT, maybe weighted by latency of RAW (+WAW/WAR?) deps

## List scheduling: algorithm

1. Insert nodes that have no predecessors into **queue**
2. Start with cycle count c=1
3. While queue not empty:
   - select an instruction i from the **queue** such that all predecessors were scheduled "sufficiently long ago" (latency information)
   - if a functional unit u for i is available:
     - insert i in (c, u), and remove it from the **queue**
     - insert any successor of i into **queue** for which all predecessors have now been scheduled
   - if no functional unit is available for i, select another instruction
   - if no instruction from the queue was scheduled, increment c

priority: e.g. length of path to EXIT, maybe weighted by latency of RAW (+WAW/WAR?) deps

Variation:
- start at nodes without successors and cycle count LAST
- work upwards, entering finish times of instructions in table
- availability of FU's still governed by start times

## Trace scheduling

Observation: individual basic blocks often don’t have much ILP

- speed-up limited
- many slots in list schedule remain empty: poor resource utilization
- problem is accentuated by deep pipelines, where many instructions could be concurrently in-flight

Q: How can we extend scheduling to many basic blocks?

## Trace scheduling

Observation: individual basic blocks often don't have much ILP

- speed-up limited
- many slots in list schedule remain empty: poor resource utilization
- problem is accentuated by deep pipelines, where many instructions could be concurrently in-flight

Q: How can we extend scheduling to many basic blocks?

A: By considering sets of basic blocks that are often executed together

- select instructions along **frequently executed traces**

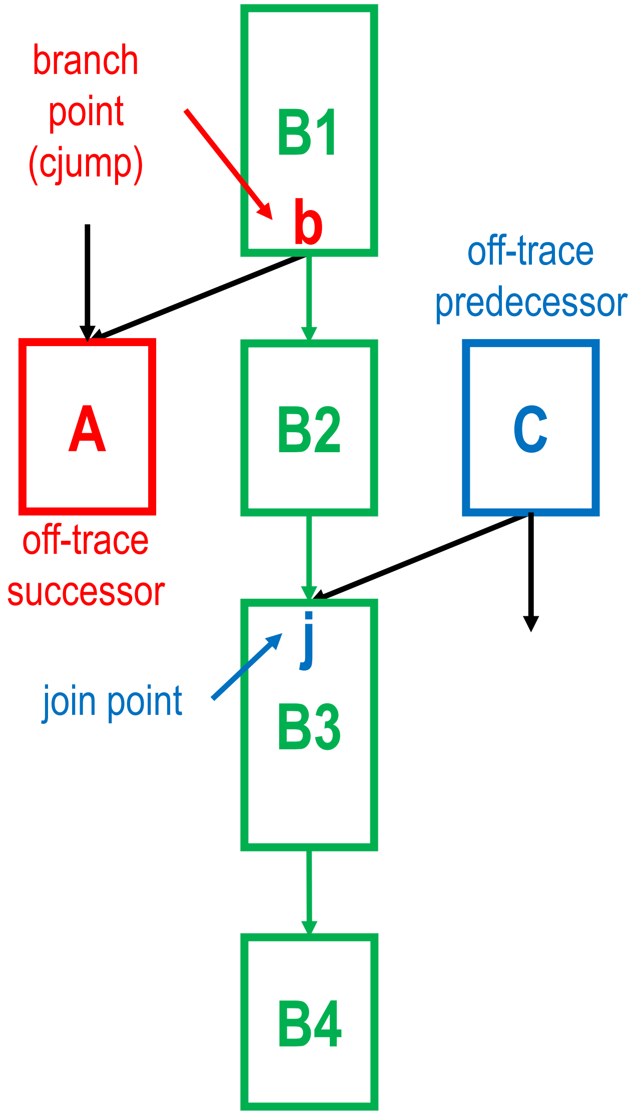
e.g. by profiling, counting the traversals of each CFG edge

acyclic path through CFG

- schedule trace members using list scheduling
- adjust off-trace code to deal with executions that only traverse parts of the trace

Details: Joseph A. Fisher: **Trace Scheduling: A Technique for Global Microcode Compaction.** IEEE Trans. Computers 30(7): 478-490 (1981)

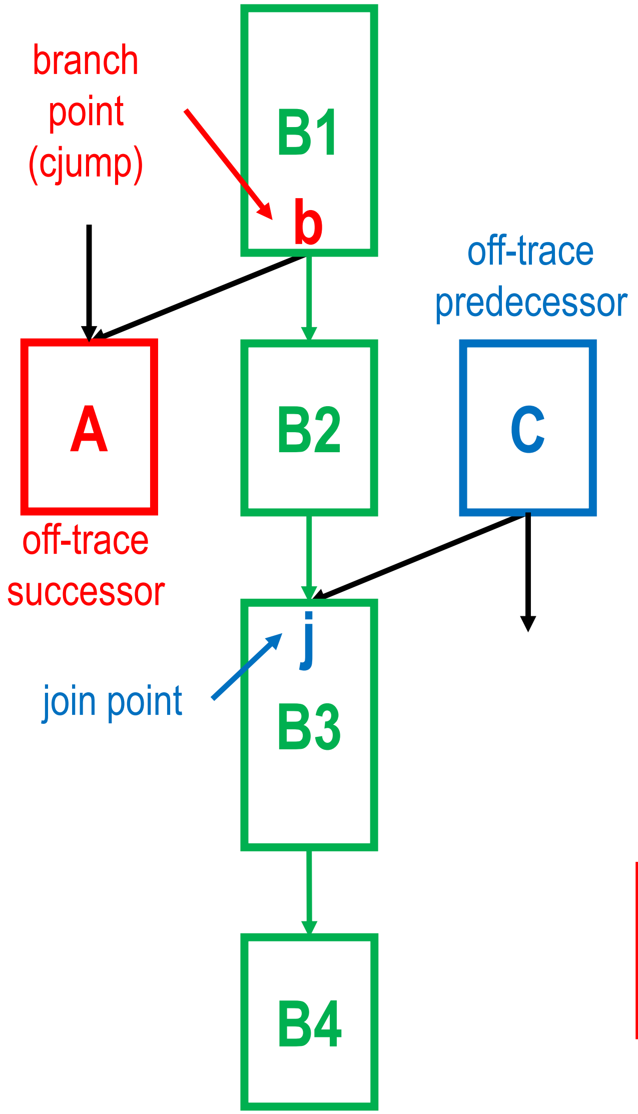
# Trace scheduling



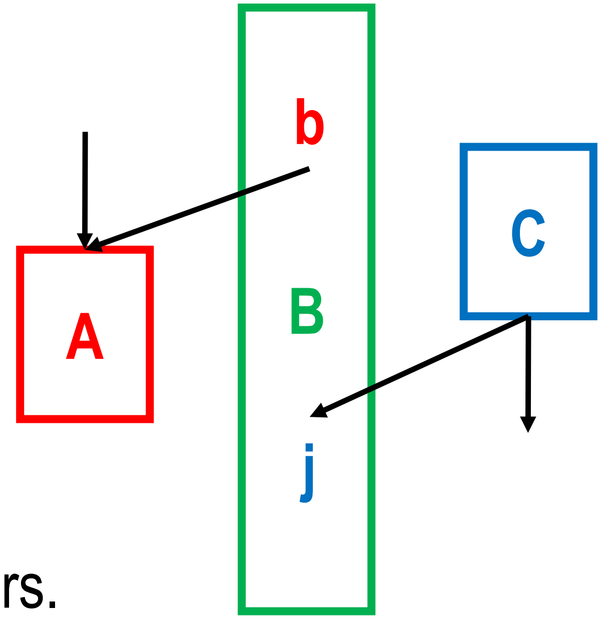
1. construct data dependence graph of instructions on trace, but consider liveIn's of **A** to be read by b.

A trace **t**, and its neighbors.

## Trace scheduling

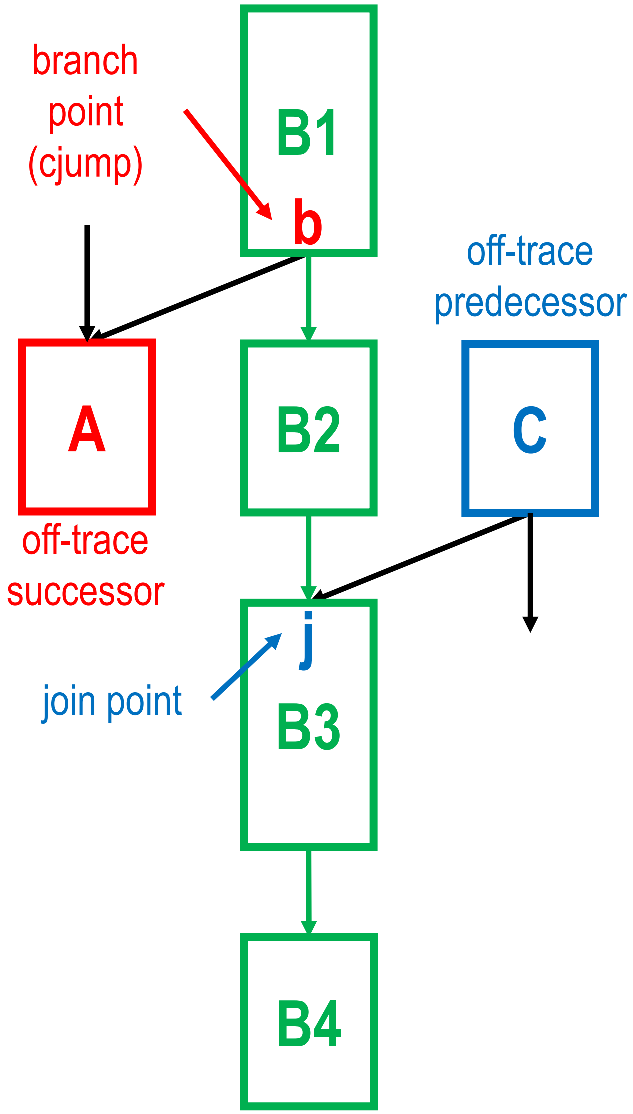


1. construct data dependence graph of instructions on trace, but consider liveIn's of A to be read by b.
2. (list-)schedule instructions in t



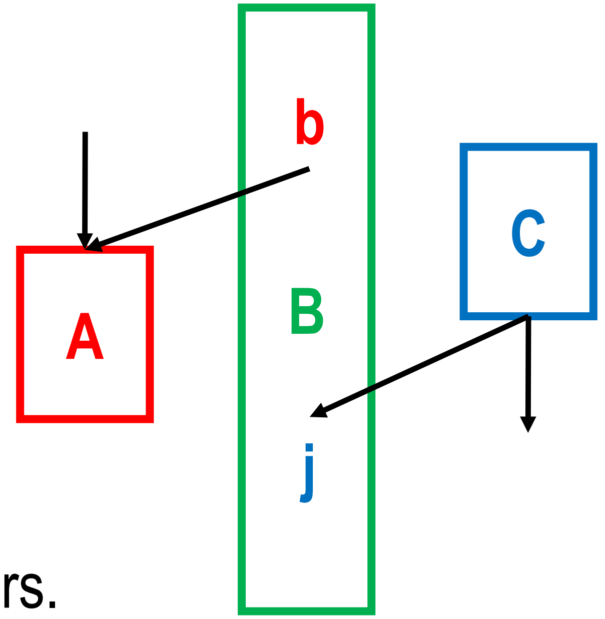
A trace t, and its neighbors.

## Trace scheduling



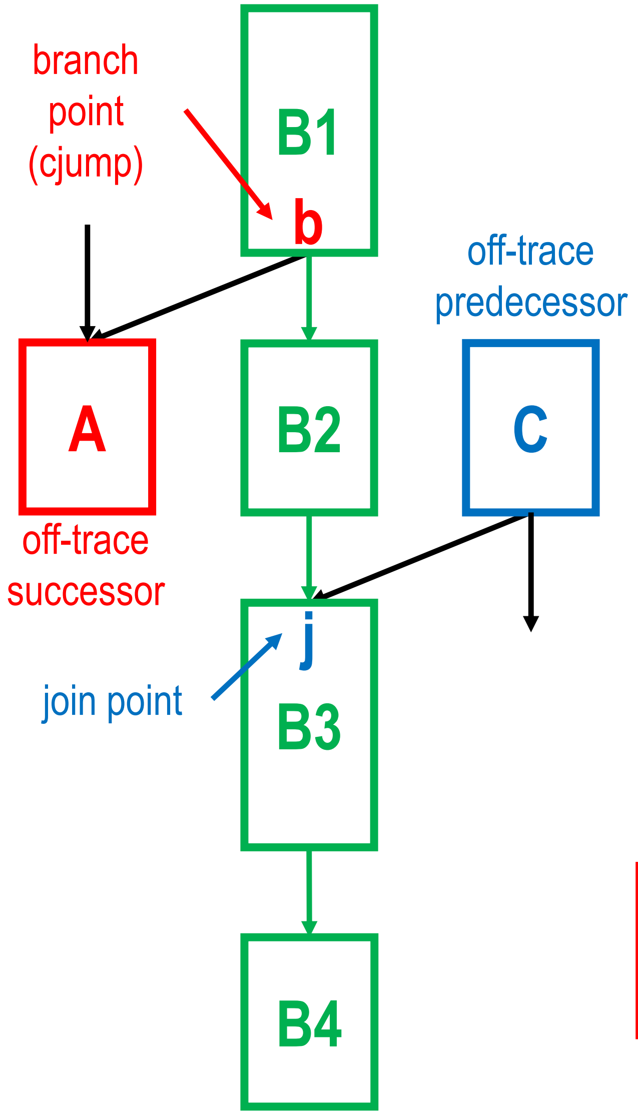
A trace t, and its neighbors.

1. construct data dependence graph of instructions on trace, but consider liveIn's of A to be read by b.
2. (list-)schedule instructions in t

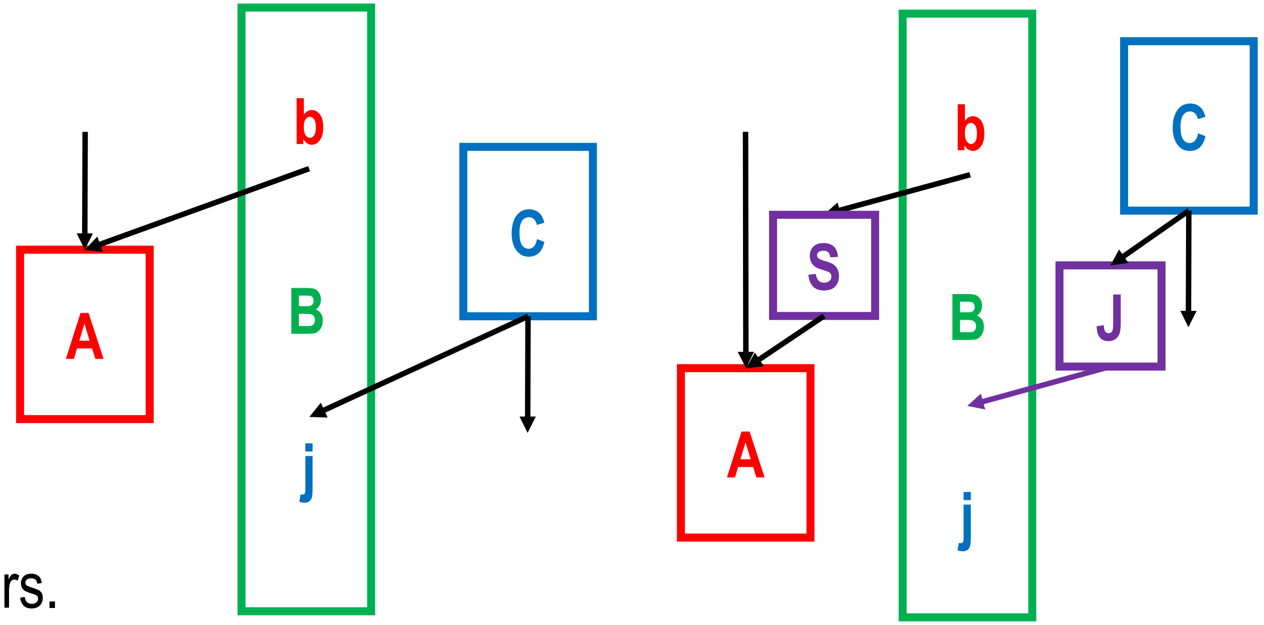


prevents those instructions in B2, B3, B4 that **define** variables that are **used** in A from being moved **up** past b, by creating a WAR dependence.

# Trace scheduling

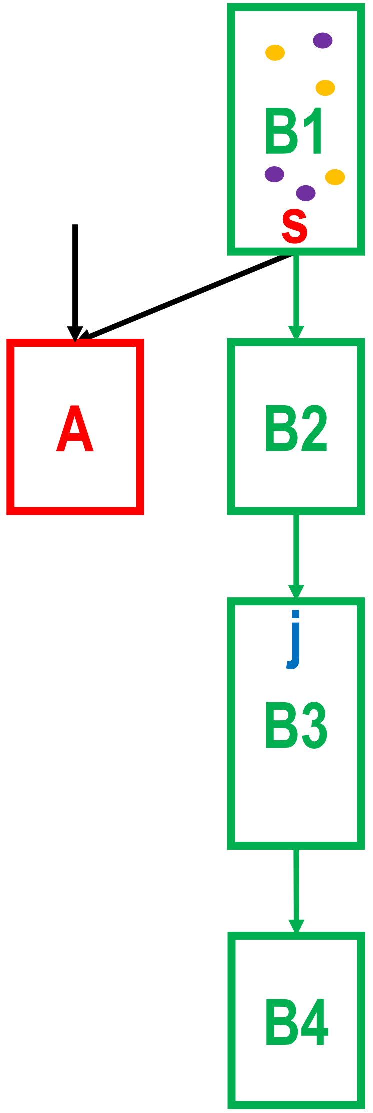


1. construct data dependence graph of instructions on trace, but consider liveIn's of A liveIn of b.
2. (list-)schedule instructions in t
3. adjust code outside of t

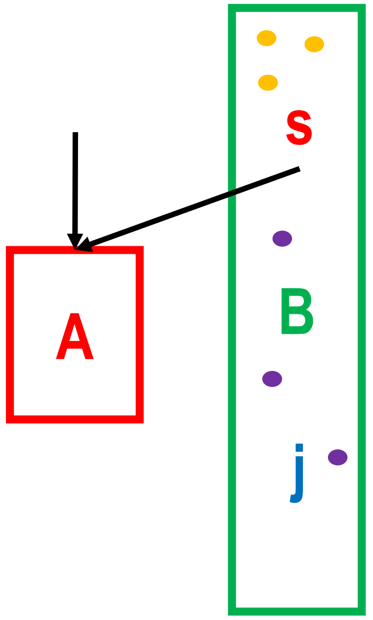


A trace t, and its neighbors.

# Trace scheduling: compensation code S



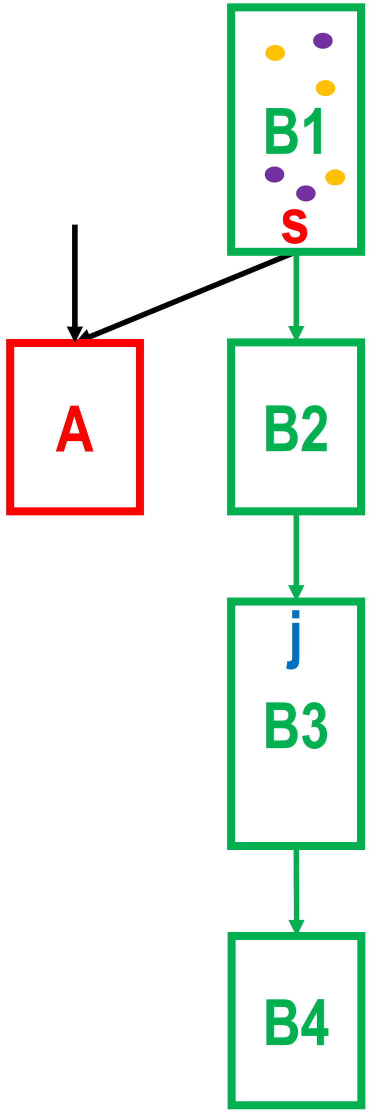
In step 2, some instructions in **B1** end up above s in **B**, others **below**.

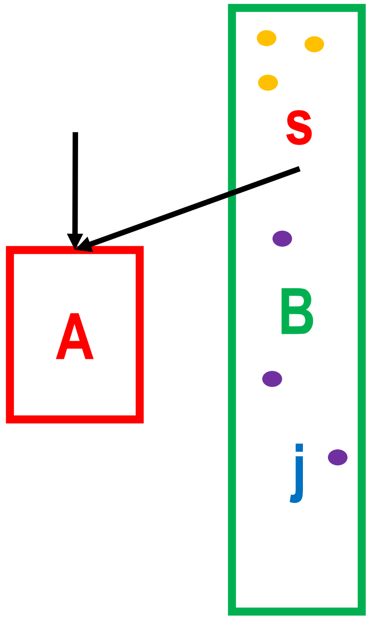


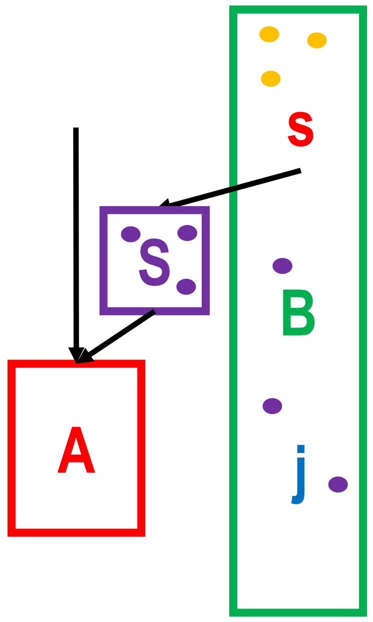
# Trace scheduling: compensation code S

In step 2, some instructions in **B1** end up **above** s in **B**, others **below**.

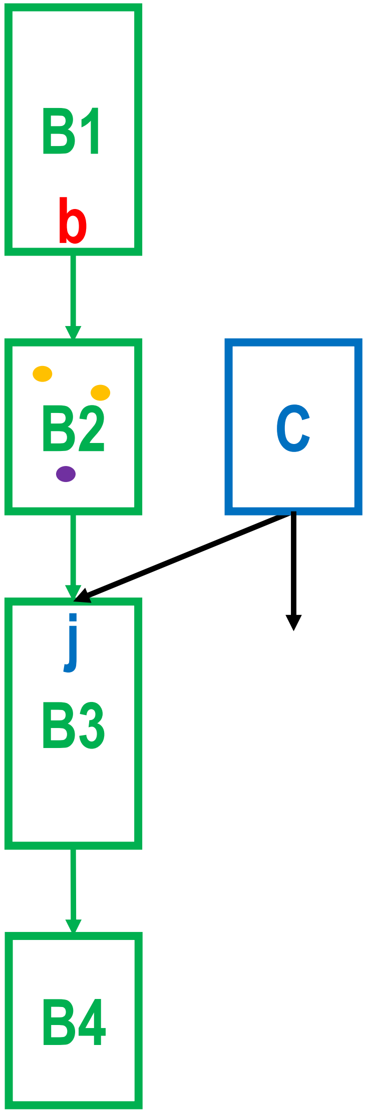
**Copy** the latter ones into the edge **s** → **A**, into a new block **S** so that they're executed when control flow follows **B1** → **s** → **A**, but not when **A** is entered through a different edge.



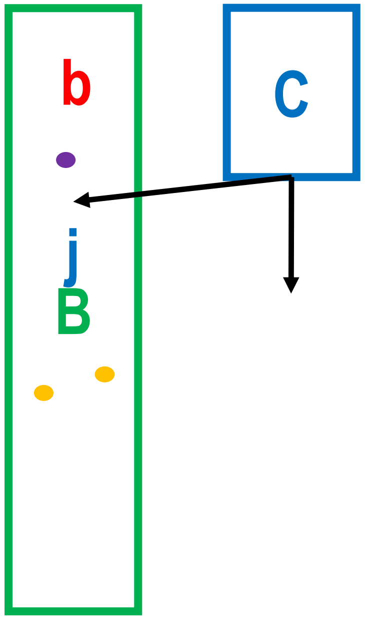




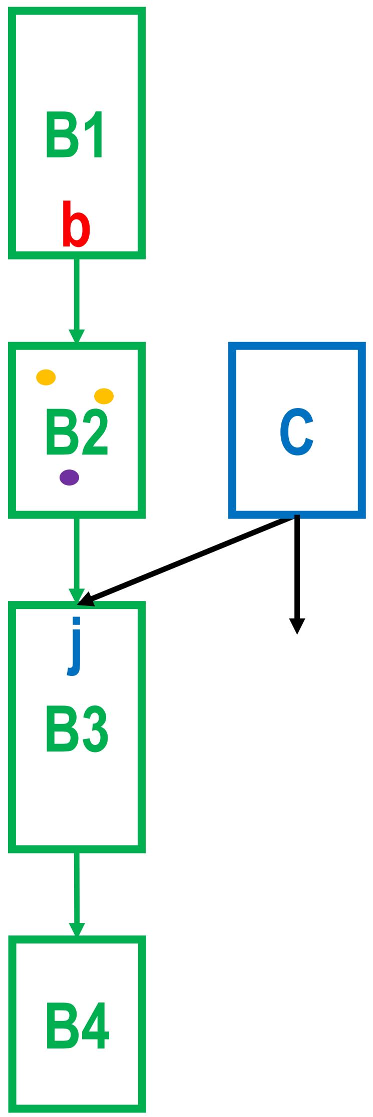
# Trace scheduling: adjust code jumping to j



In step 2, some instructions in B2 end up above j in B, others below.

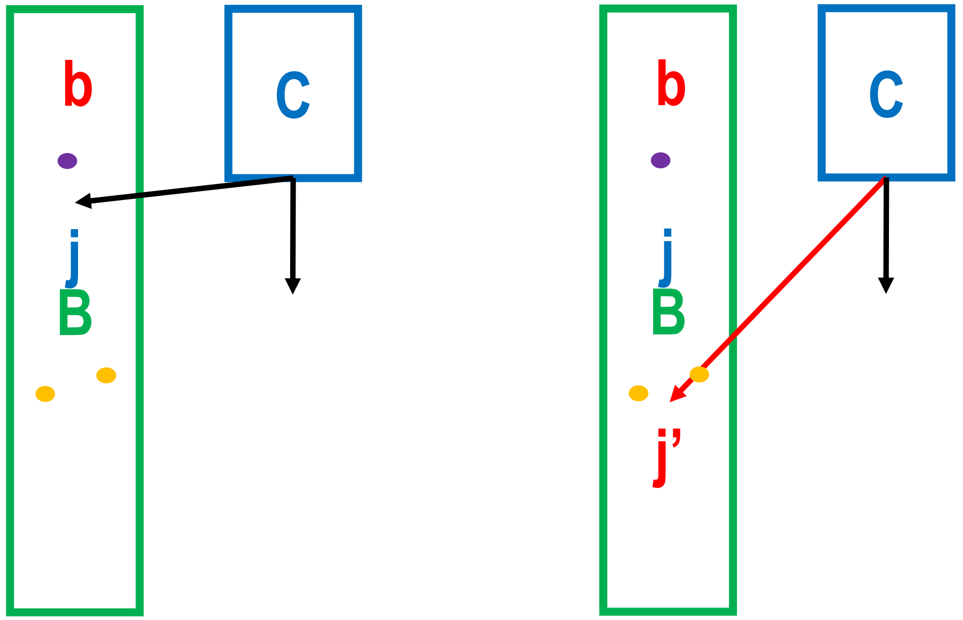


# Trace scheduling: adjust code jumping to j



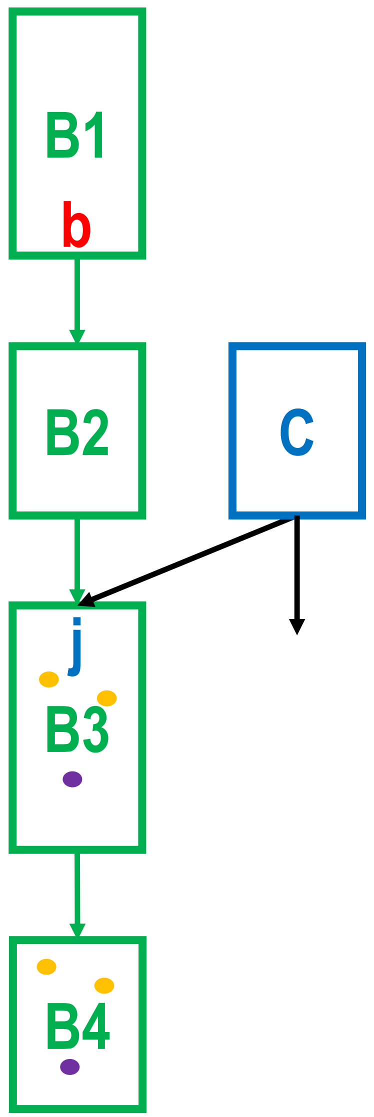
In step 2, some instructions in **B2** end up **above** j in **B**, others **below**.

**Adjust** the jump in **C** to point to the first instruction (bundle) following the **last instruction in B** that stems from **B2** – call the new jump target **j'**. Thus **yellow** instructions remain non-executed if control enters **B** from **C**: all instructions from **B2** are above **j'**.

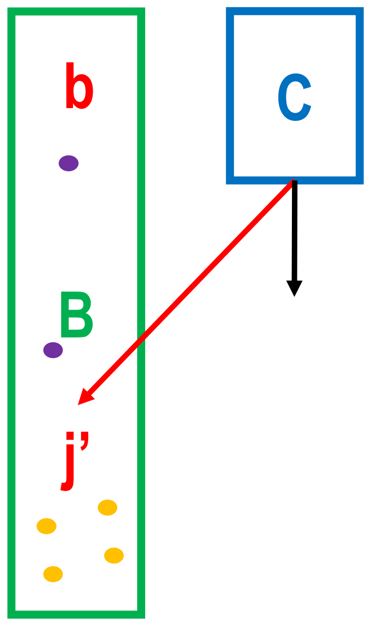


Note: if there's no yellow instruction, we're in fact adjusting j upwards: j' follows the last **purple** instruction.

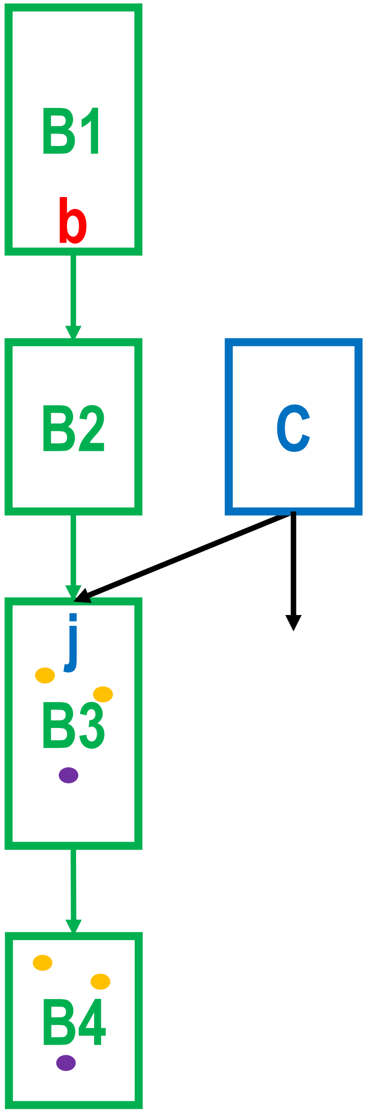
# Trace scheduling: adjusting code jumping into B



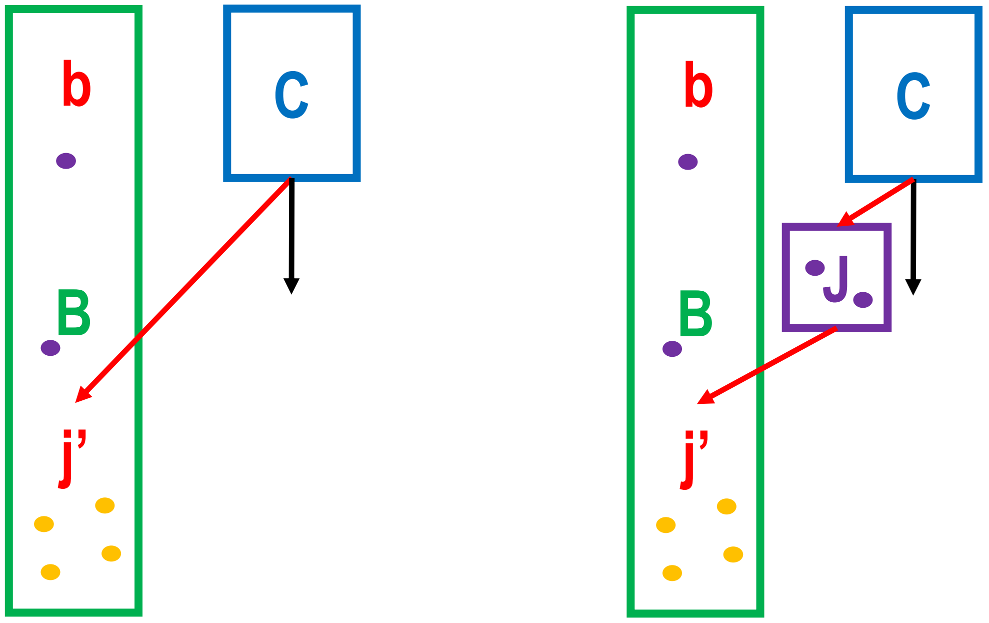
Next, some instructions from B3/B4 end up above j' in B, others below.



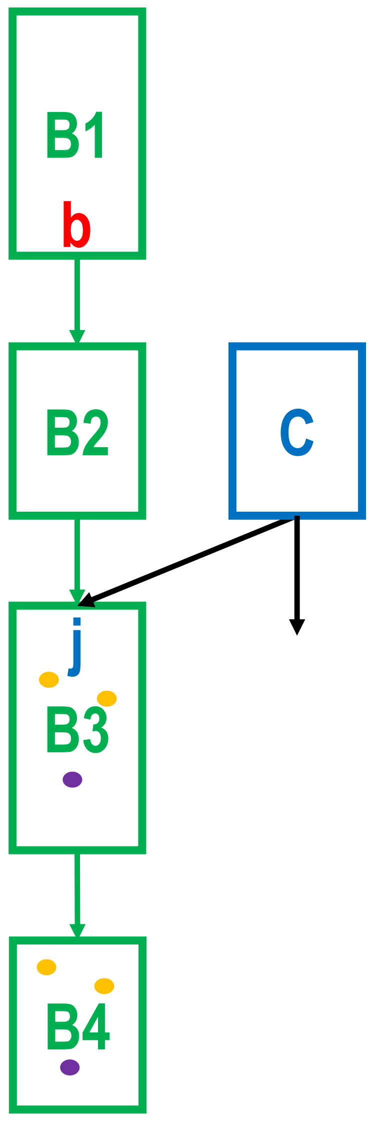
# Trace scheduling: adjust code jumping to j



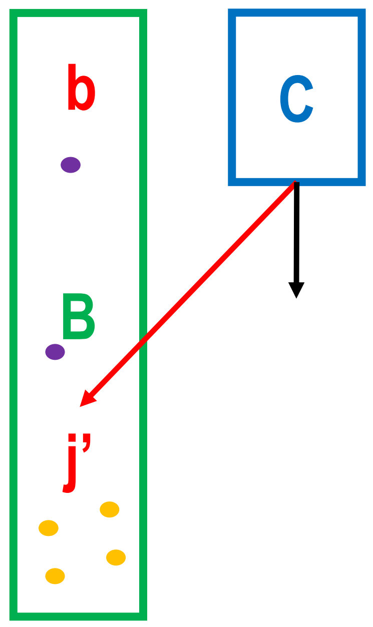
Next, some instructions from **B3/B4** end up **above** **j'** in **B**, others **below**. **Copy** the former ones into the edge **C** → **j'**, into a new block **J**, ensuring that instructions following **j'** receive correct data when flow enters **B** via **C**.

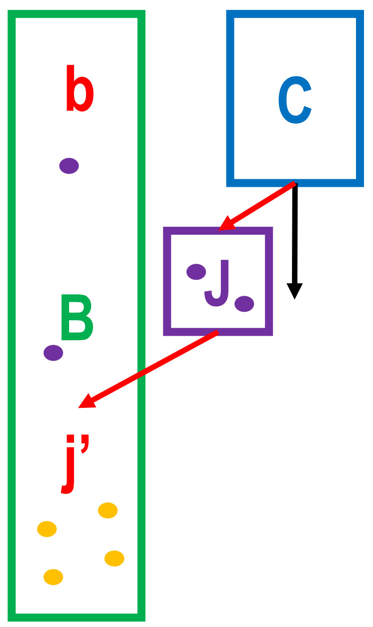


# Trace scheduling: cleaning up S and J



Next, some instructions from B3/B4 end up above j’ in B, others below. Copy the former ones into the edge C → j’, into a new block J, ensuring that instructions following j’ receive correct data when flow enters B via C.

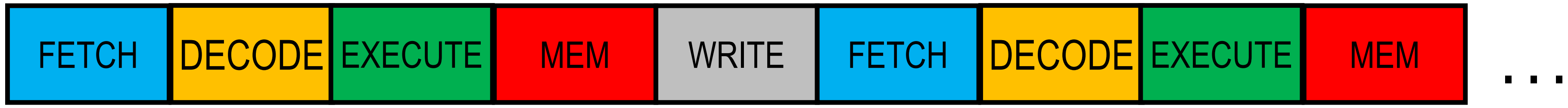




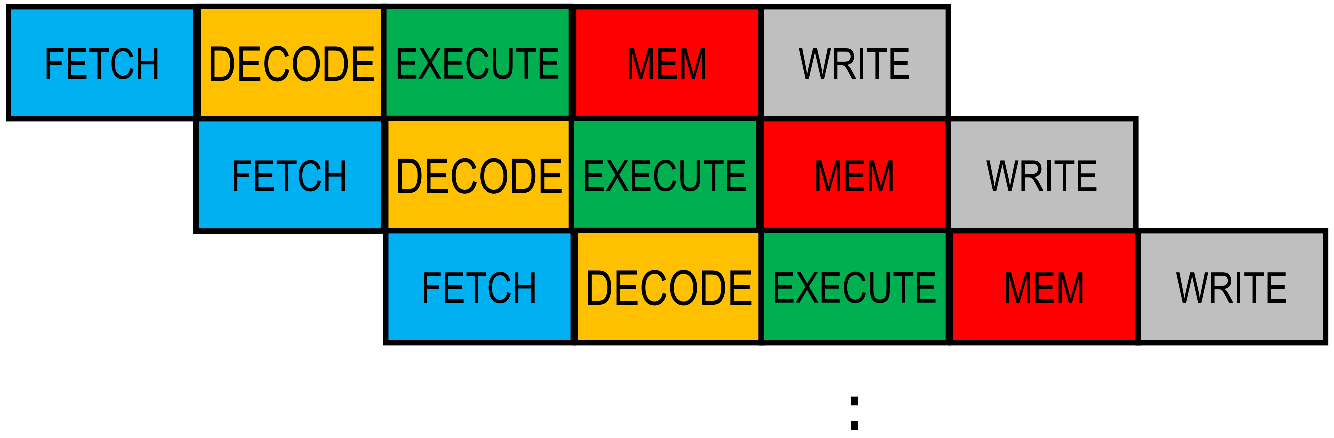
Final cleanup: some instructions in S and J may be dead – eliminate them. Then, S and J can be (list-)scheduled or be part of the next trace.

## Pipelining

Purely sequential execution:



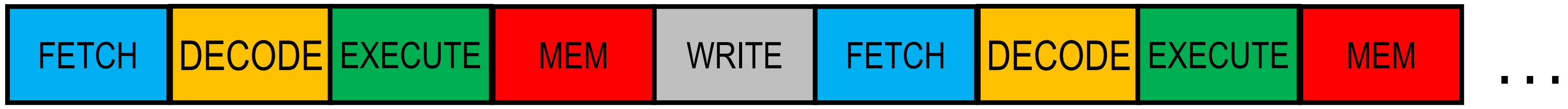
Pipelining - can partially overlap instructions:



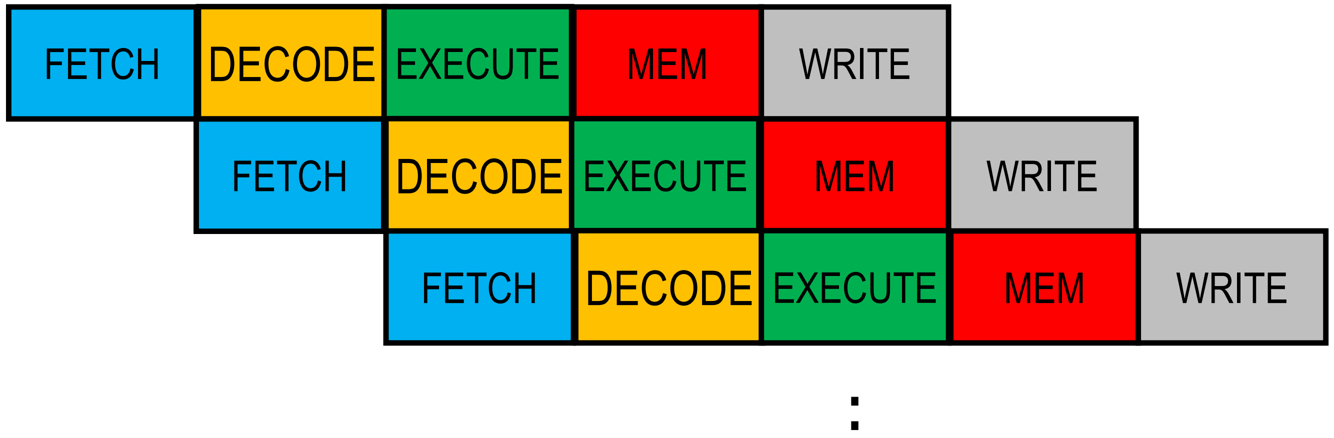
One instruction issued (and retired) each cycle – speedup ≈ pipeline depth

## Pipelining

Purely sequential execution:



Pipelining - can partially overlap instructions:



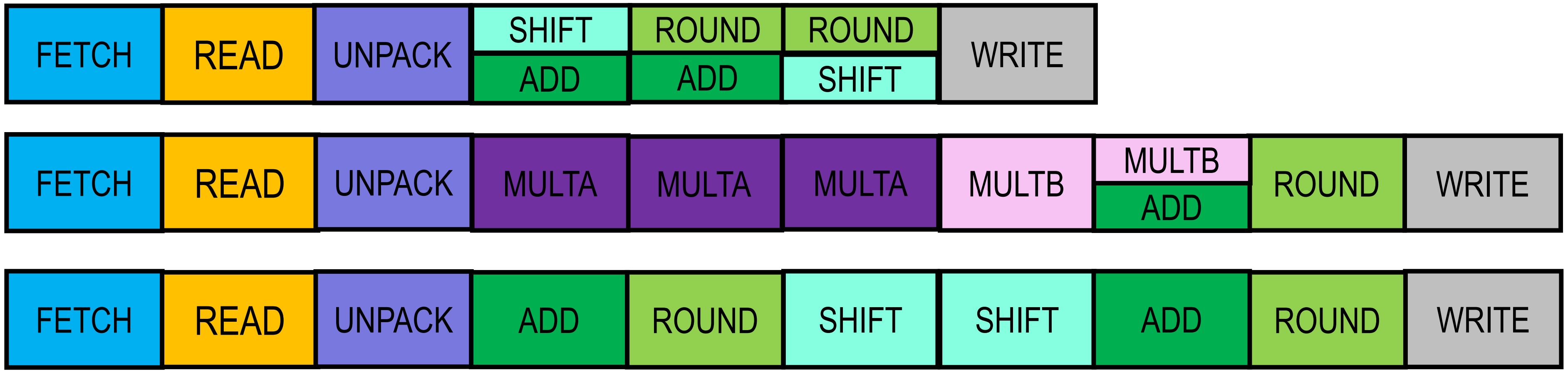
One instruction issued (and retired) each cycle – speedup ≈ pipeline depth

… assuming that
- each instruction spends one cycle in each stage
- all instruction forms visit same (sequence of) FU’s
- there are no (data) dependencies

## Pipelining for realistic processors

Different instructions visit different sets/sequences of functional units, and occasionally multiple types of functional units in the same cycle:

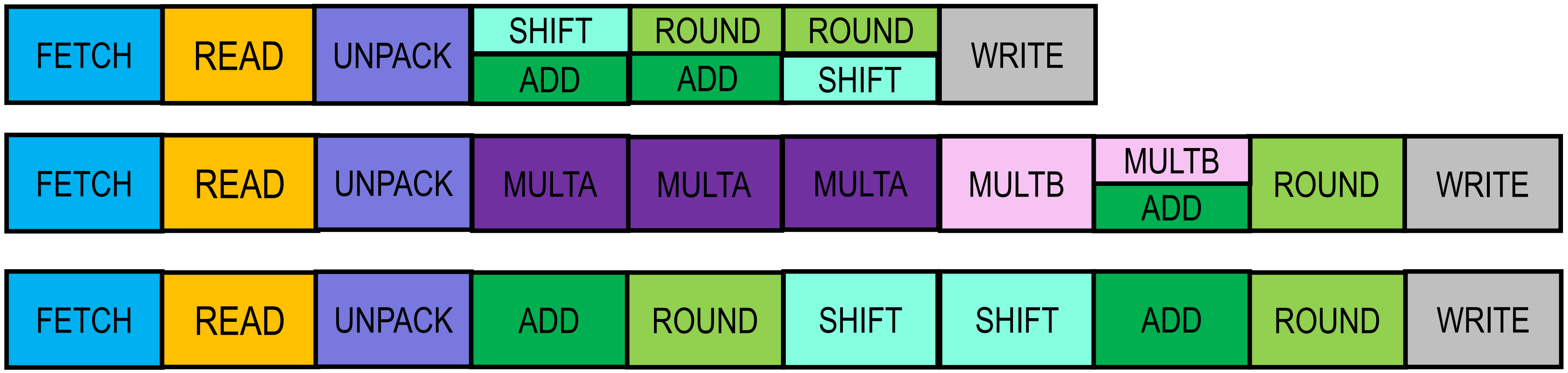
Example: floating point instructions on MIPS R4000 (ADD, MUL, CONV)



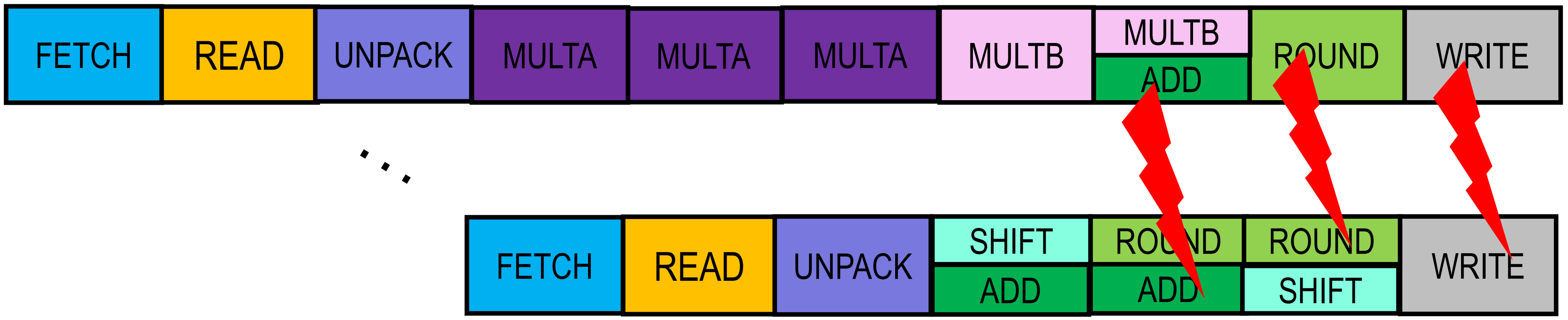
# Pipelining for realistic processors

Different instructions visit different sets/sequences of functional units, and occasionally multiple types of functional units in the same cycle:

Example: floating point instructions on MIPS R4000 (ADD, MUL, CONV)

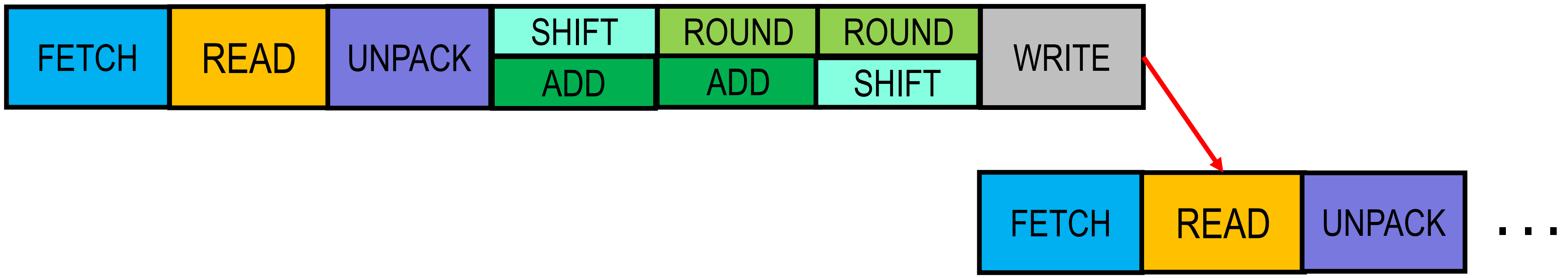


Contention for FU's means some pipelinings must be avoided:



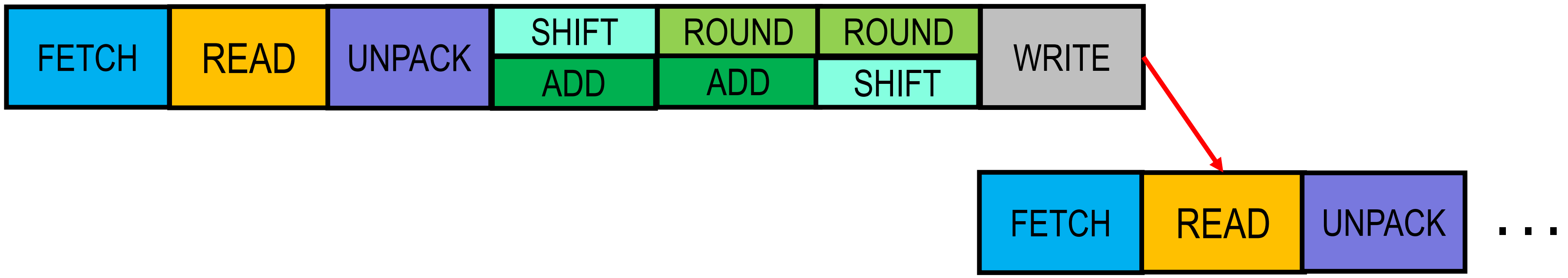
## Pipelining constraints: data dependencies

RAW dependency:

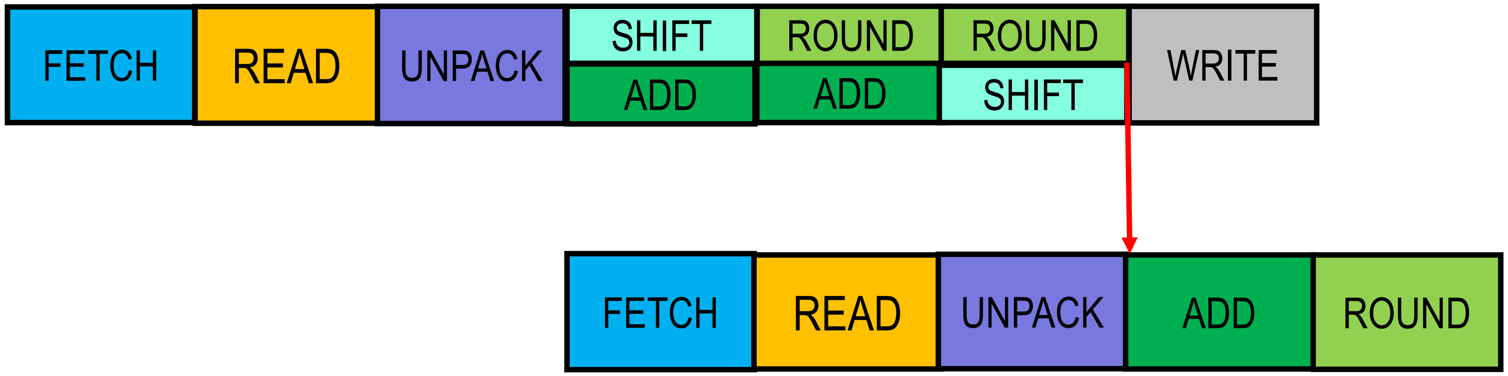


## Pipelining constraints: data dependencies

RAW dependency:



Register bypassing / operand forwarding: extra HW to communicate data directly between FU's



Result of one stage is available at another stage in the next cycle.

# Loop scheduling without resource bounds

- illustrates use of loop unrolling and introduces terminology for full SW pipelining
  - but not useful in practice

```
for i ← 1 to N
    a ← j ◊ V [ i – 1]
    b ← a ◊ f
    c ← e ◊ j
    d ← f ◊ c
    e ← b ◊ d
    f ← U [ i ]
    g: V [ i ] ← b
    h: W [ i ] ← d
    j ← X [ i ]
```

$$\xrightarrow[\text{make “iteration index” explicit}]{\text{scalar replacement}}$$

◊ some binary op(s)

# Loop scheduling without resource bounds

- illustrates use of loop unrolling and introduces terminology for full SW pipelining
  - but not useful in practice

```
for i ← 1 to N
    a ← j ◊ V [ i – 1]
    b ← a ◊ f
    c ← e ◊ j
    d ← f ◊ c
    e ← b ◊ d
    f ← U [ i ]
    g: V [ i ] ← b
    h: W [ i ] ← d
    j ← X [ i ]
```

**scalar replacement** $\longrightarrow$ **make "iteration index" explicit**

```
for i ← 1 to N
    $a_i$ ← $j_{i-1}$ ◊ $b_{i-1}$
    $b_i$ ← $a_i$ ◊ $f_{i-1}$
    $c_i$ ← $e_{i-1}$ ◊ $j_{i-1}$
    $d_i$ ← $f_{i-1}$ ◊ $c_i$
    $e_i$ ← $b_i$ ◊ $d_i$
    $f_i$ ← U [ i ]
    g: V [ i ] ← $b_i$
    h: W [ i ] ← $d_i$
    $j_i$ ← X [ i ]
```

◊ some binary op(s)

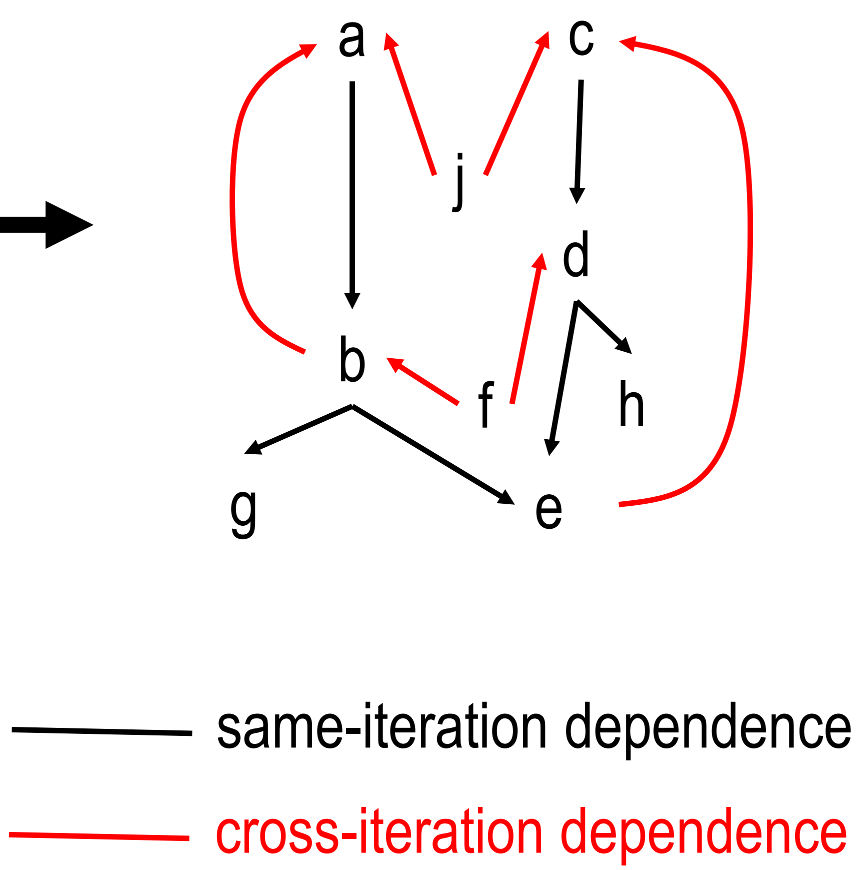
Scalar replacement: poor-man's alternative to alias analysis (again) but often helpful

# Loop scheduling without resource bounds

```
for i ← 1 to N
    $a_i$ ← $j_{i-1}$ ◊ $b_{i-1}$
    $b_i$ ← $a_i$ ◊ $f_{i-1}$
    $c_i$ ← $e_{i-1}$ ◊ $j_{i-1}$
    $d_i$ ← $f_{i-1}$ ◊ $c_i$
    $e_i$ ← $b_i$ ◊ $d_i$
    $f_i$ ← U [ i ]
    g: V [ i ] ← $b_i$
    h: W [ i ] ← $d_i$
    $j_i$ ← X [ i ]
```

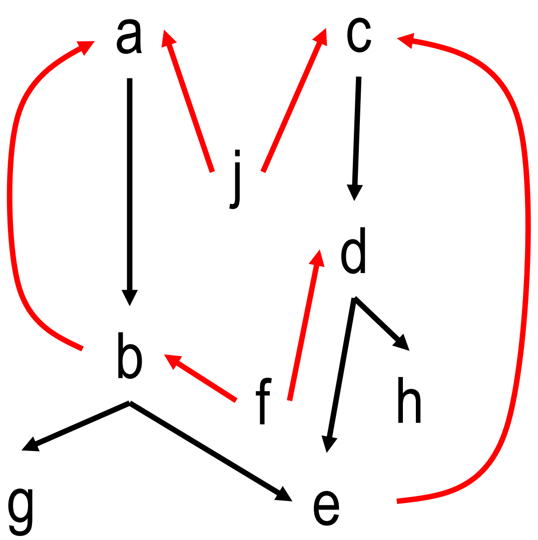
$\longrightarrow$

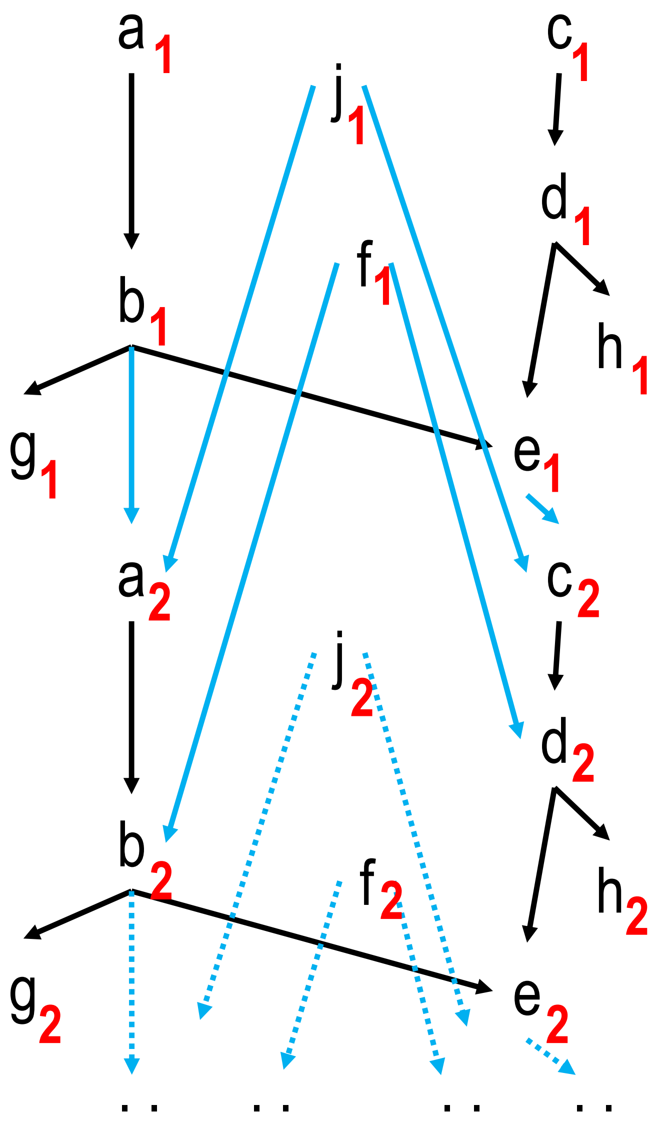
**Data dependence graph of body**



# Loop scheduling without resource bounds

**Data dependence graph of unrolled body – acyclic!**





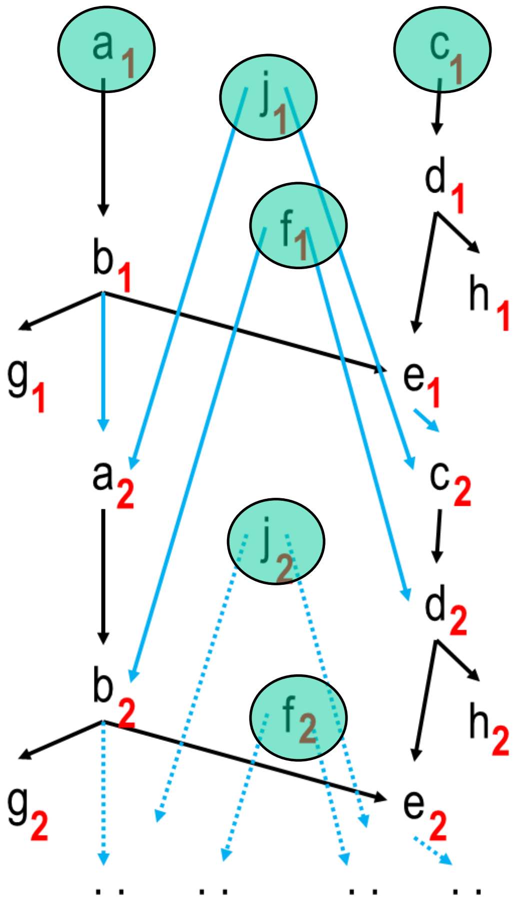
same-iteration dependence

cross-iteration dependence

# Loop scheduling without resource bounds

## Arrange in tableau

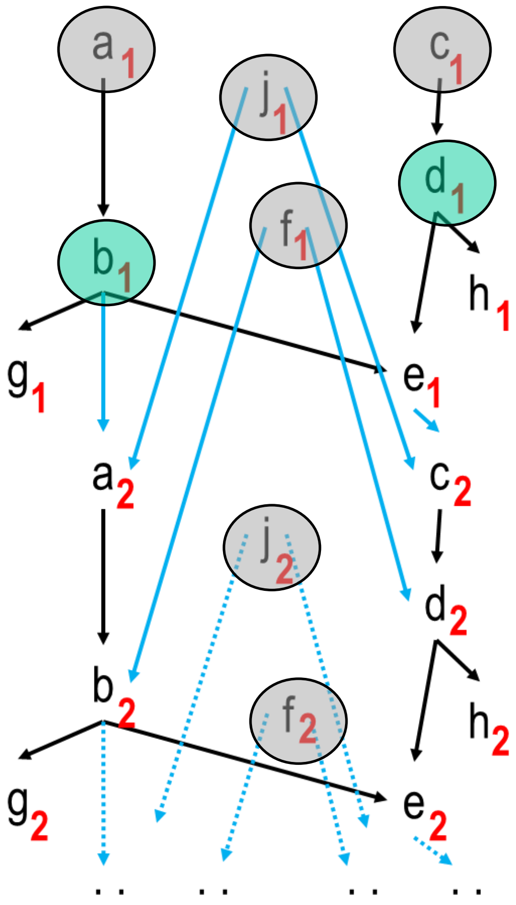
- rows: cycles
- columns: iterations
- unlimited resources



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | a c f j | f j | f j | f j | f j | f j |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |
| 7 | | | | | | |
| 8 | | | | | | |
| 9 | | | | | | |
| 10 | | | | | | |
| 11 | | | | | | |
| 12 | | | | | | |
| 13 | | | | | | |
| 14 | | | | | | |
| 15 | | | | | | |

# Loop scheduling without resource bounds

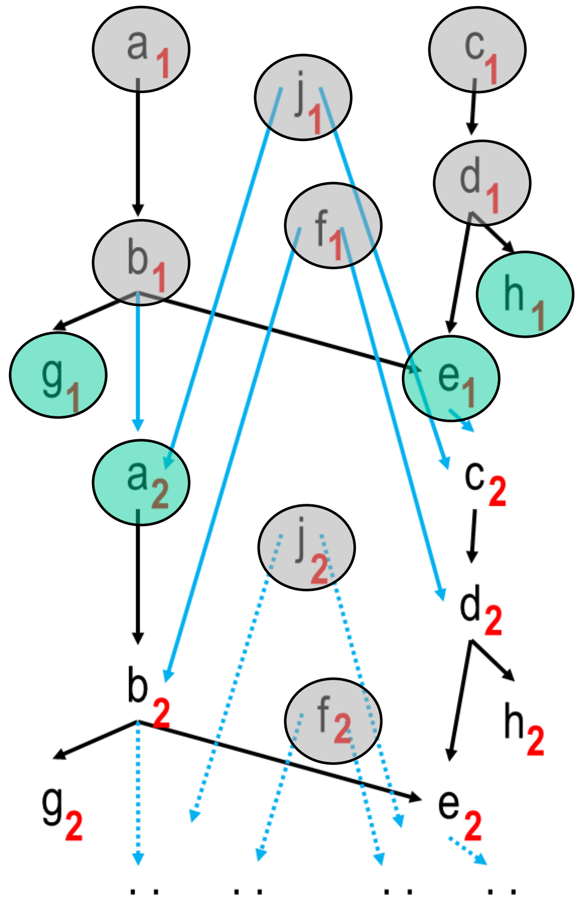
**Arrange in tableau**



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | a c f j | f j | f j | f j | f j | f j |
| 2 | b d | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |
| 7 | | | | | | |
| 8 | | | | | | |
| 9 | | | | | | |
| 10 | | | | | | |
| 11 | | | | | | |
| 12 | | | | | | |
| 13 | | | | | | |
| 14 | | | | | | |
| 15 | | | | | | |

# Loop scheduling without resource bounds

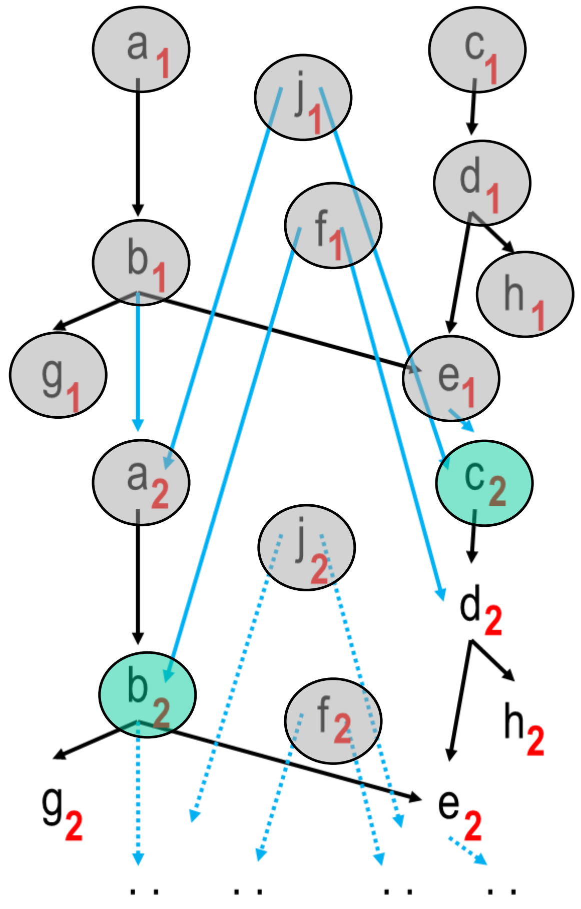
**Arrange in tableau**



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | a c f j | f j | f j | f j | f j | f j |
| 2 | b d | | | | | |
| 3 | e g h | a | | | | |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |
| 7 | | | | | | |
| 8 | | | | | | |
| 9 | | | | | | |
| 10 | | | | | | |
| 11 | | | | | | |
| 12 | | | | | | |
| 13 | | | | | | |
| 14 | | | | | | |
| 15 | | | | | | |

# Loop scheduling without resource bounds

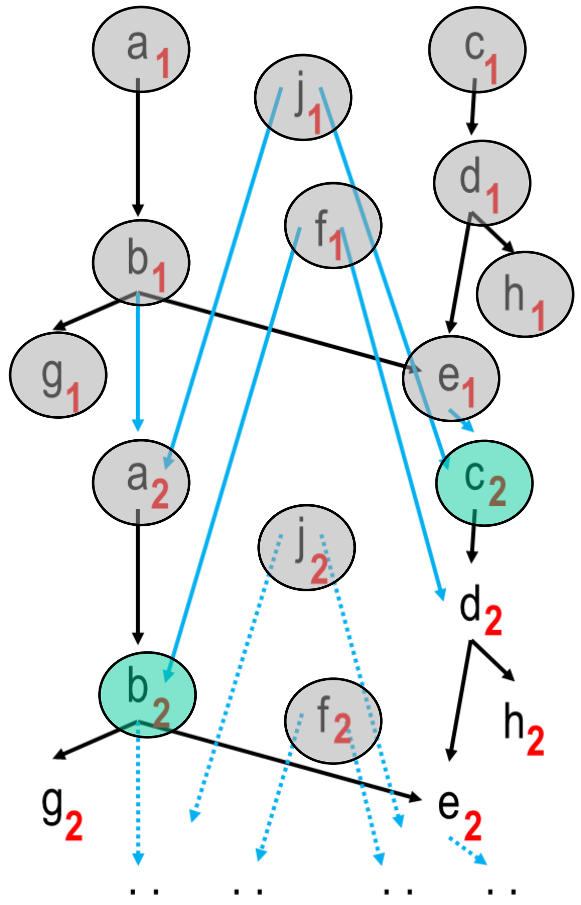
**Arrange in tableau**



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | a c f j | f j | f j | f j | f j | f j |
| 2 | b d | | | | | |
| 3 | e g h | a | | | | |
| 4 | | b c | | | | |
| 5 | | | | | | |
| 6 | | | | | | |
| 7 | | | | | | |
| 8 | | | | | | |
| 9 | | | | | | |
| 10 | | | | | | |
| 11 | | | | | | |
| 12 | | | | | | |
| 13 | | | | | | |
| 14 | | | | | | |
| 15 | | | | | | |

# Loop scheduling without resource bounds

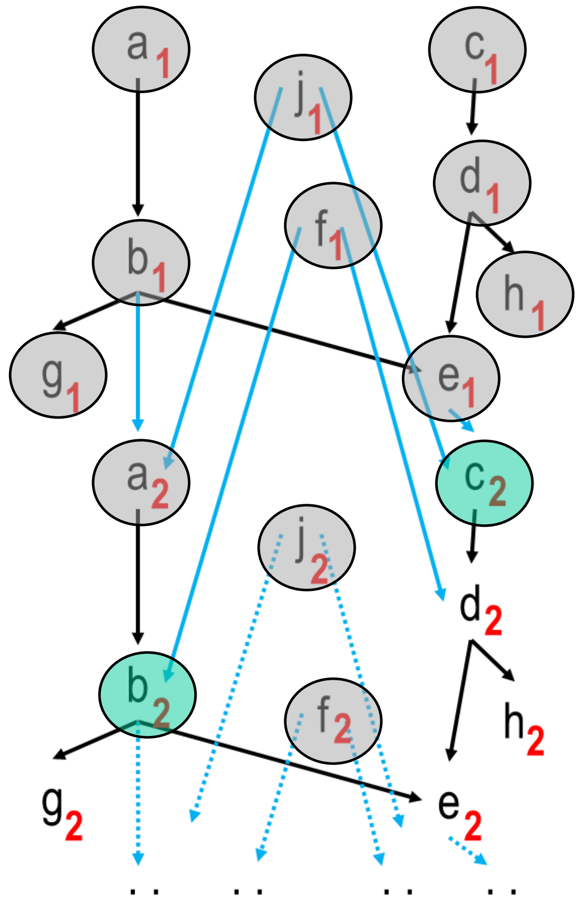
**… some more iterations.**

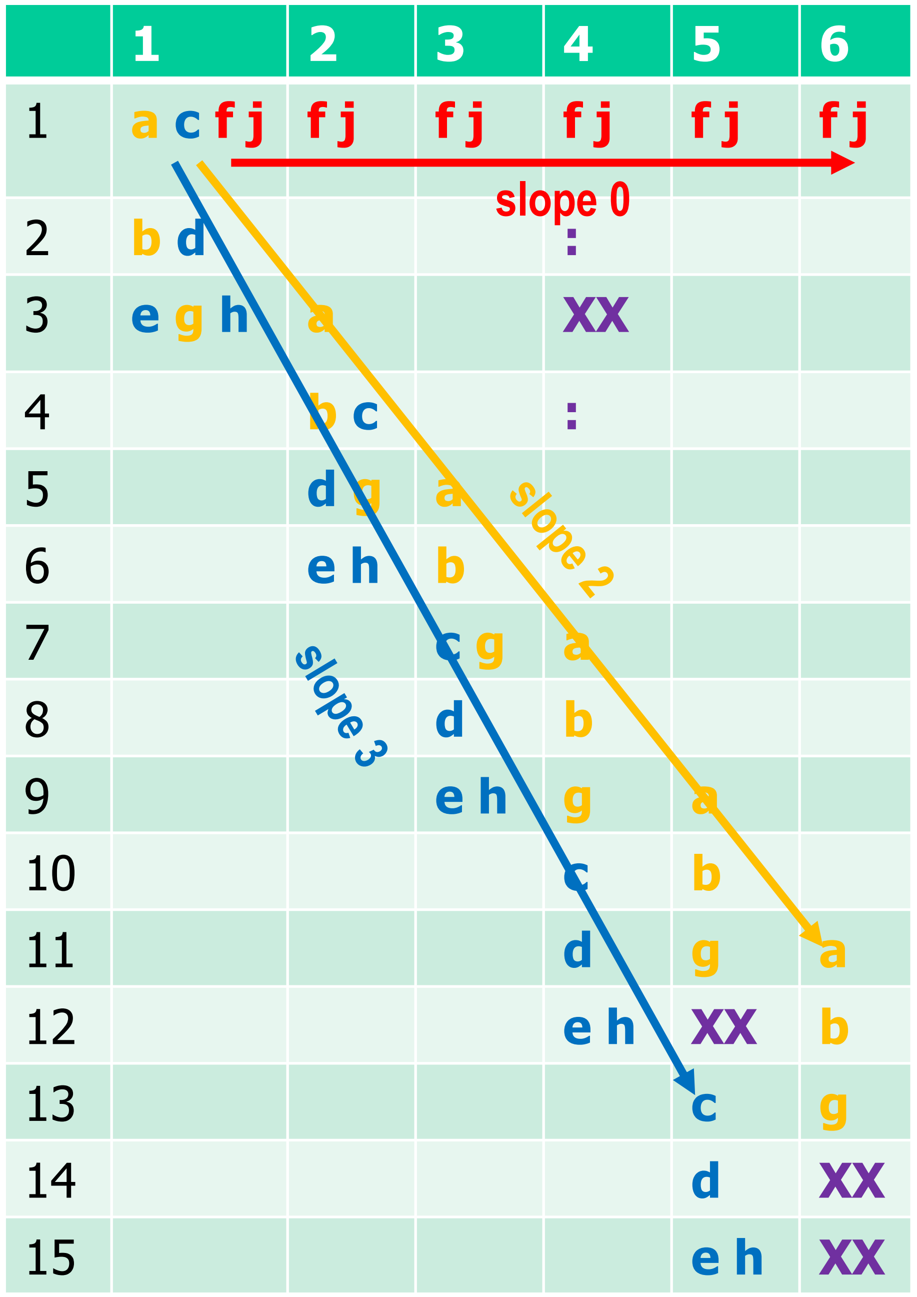


| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | a c f j | f j | f j | f j | f j | f j |
| 2 | b d | | | | | |
| 3 | e g h | a | | | | |
| 4 | | b c | | | | |
| 5 | | d g | a | | | |
| 6 | | e h | b | | | |
| 7 | | | c g | a | | |
| 8 | | | d | b | | |
| 9 | | | e h | g | a | |
| 10 | | | | c | b | |
| 11 | | | | d | g | a |
| 12 | | | | e h | | b |
| 13 | | | | | c | g |
| 14 | | | | | d | |
| 15 | | | | | e h | |

# Loop scheduling without resource bounds

**Identify groups of instructions; note gaps**



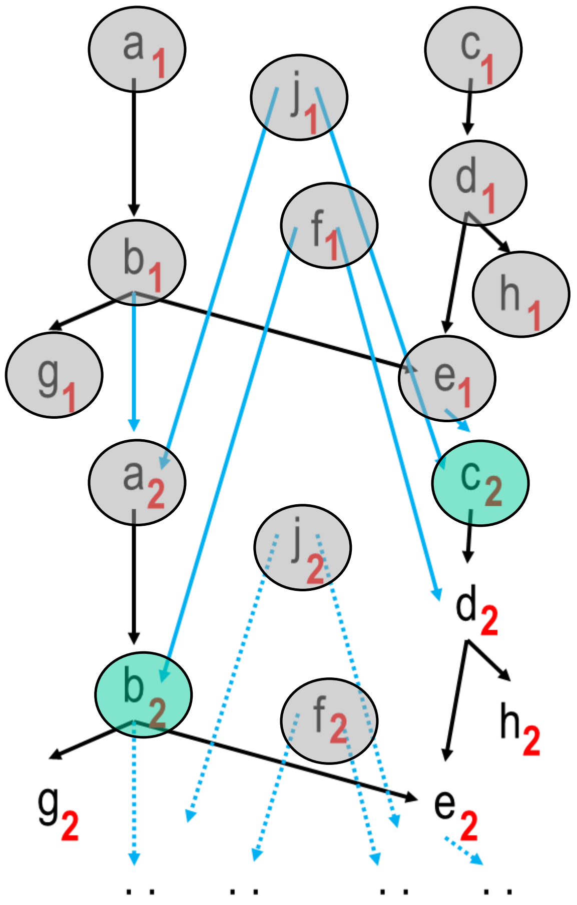


slope 0
slope 2
slope 3

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | a c f j | f j | f j | f j | f j | f j |
| 2 | b d | | | : | | |
| 3 | e g h | a | | XX | | |
| 4 | | b c | | : | | |
| 5 | | d g | a | | | |
| 6 | | e h | b | | | |
| 7 | | | c g | a | | |
| 8 | | | d | b | | |
| 9 | | | e h | g | a | |
| 10 | | | | c | b | |
| 11 | | | | d | g | a |
| 12 | | | | e h | XX | b |
| 13 | | | | | c | g |
| 14 | | | | | d | XX |
| 15 | | | | | e h | XX |

# Loop scheduling without resource bounds

**Close gaps by delaying fast instruction groups**

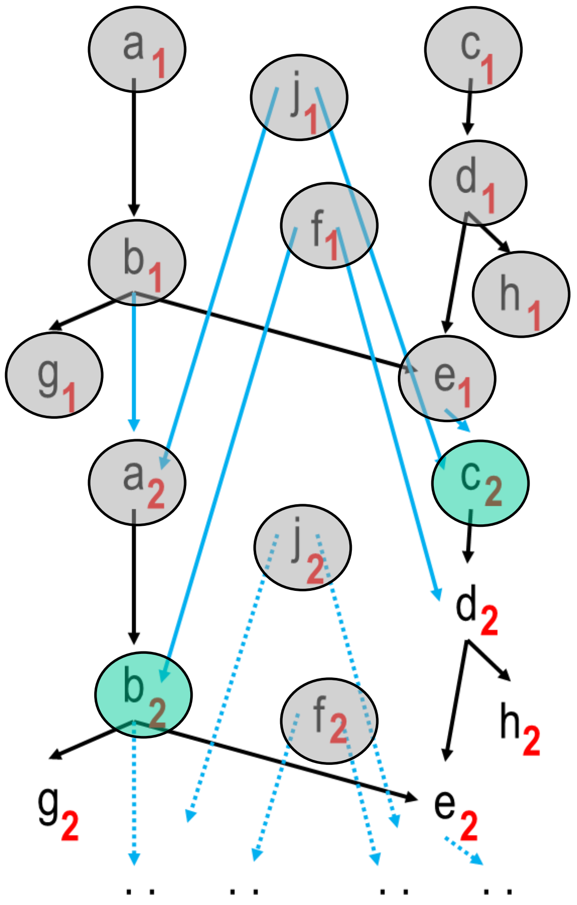


| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | a c f j | | | | | |
| 2 | b d | f j | | | | |
| 3 | e g h | a | | | | |
| 4 | | b c | f j | | | |
| 5 | | d g | a | | | |
| 6 | | e h | b | f j | | |
| 7 | | | c g | a | | |
| 8 | | | d | b | | |
| 9 | | | e h | g | f j | |
| 10 | | | | c | a | |
| 11 | | | | d | b | |
| 12 | | | | e h | g | f j |
| 13 | | | | | c | a |
| 14 | | | | | d | b |
| 15 | | | | | e h | g |

slope 3

# Loop scheduling without resource bounds

**Identify "steady state" – of slope 3**



prologue

epilogue

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | a c f j | | | | | |
| 2 | b d | f j | | | | |
| 3 | e g h | a | | | | |
| 4 | | b c | f j | | | |
| 5 | | d g | a | | | |
| 6 | | e h | b | f j | | |
| 7 | | | c g | a | | |
| 8 | | | d | b | | |
| 9 | | | e h | g | f j | |
| 10 | | | | c | a | |
| 11 | | | | d | b | |
| 12 | | | | e h | g | f j |
| 13 | | | | | c | a |
| 14 | | | | | d | b |
| 15 | | | | | e h | g |

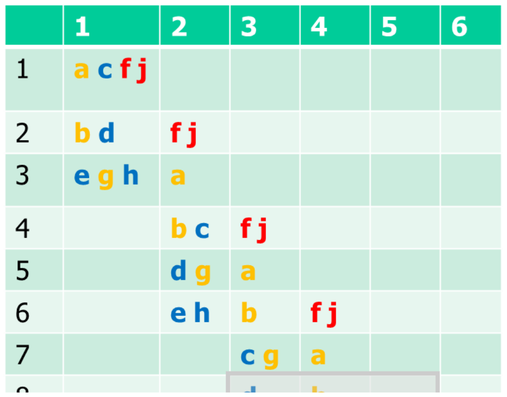
# Loop scheduling without resource bounds

**Expand instructions**

- No cycle has > 5 instructions
- Instructions in a row execute in parallel; reads in RHS happen before writes in LHS

1. Prologue – also set up i

prologue



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | a c f j | | | | | |
| 2 | b d | f j | | | | |
| 3 | e g h | a | | | | |
| 4 | | b c | f j | | | |
| 5 | | d g | a | | | |
| 6 | | e h | b | f j | | |
| 7 | | | c g | a | | |

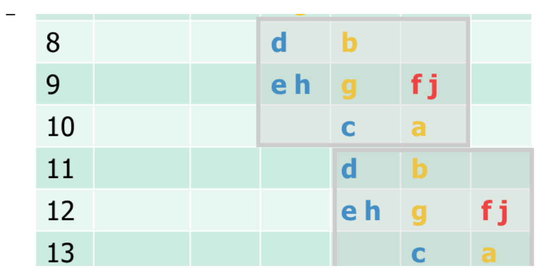
| | | | | |
|---|---|---|---|---|
| $a_1 \leftarrow j_0 \lozenge b_0$ | $c_1 \leftarrow e_0 \lozenge j_0$ | $f_1 \leftarrow U[1]$ | $j_1 \leftarrow X[1]$ | |
| $b_1 \leftarrow a_1 \lozenge f_0$ | $d_1 \leftarrow f_0 \lozenge c_1$ | $f_2 \leftarrow U[2]$ | $j_2 \leftarrow X[2]$ | |
| $e_1 \leftarrow b_1 \lozenge d_1$ | $V[1] \leftarrow b_1$ | $W[1] \leftarrow d_1$ | $a_2 \leftarrow j_1 \lozenge b_1$ | |
| $b_2 \leftarrow a_2 \lozenge f_1$ | $c_2 \leftarrow e_1 \lozenge j_1$ | $f_3 \leftarrow U[3]$ | $j_3 \leftarrow X[3]$ | |
| $d_2 \leftarrow f_1 \lozenge c_2$ | $V[2] \leftarrow b_2$ | $a_3 \leftarrow j_2 \lozenge b_2$ | | |
| $e_2 \leftarrow b_2 \lozenge d_2$ | $W[2] \leftarrow d_2$ | $b_3 \leftarrow a_3 \lozenge f_2$ | $f_4 \leftarrow U[4]$ | $j_4 \leftarrow X[4]$ |
| $c_3 \leftarrow e_2 \lozenge j_2$ | $V[3] \leftarrow b_3$ | $a_4 \leftarrow j_3 \lozenge b_3$ | | $i \leftarrow 3$ |

```
for i ← 1 to N
    $a_i \leftarrow j_{i-1} \lozenge b_{i-1}$
    $b_i \leftarrow a_i \lozenge f_{i-1}$
    $c_i \leftarrow e_{i-1} \lozenge j_{i-1}$
    $d_i \leftarrow f_{i-1} \lozenge c_i$
    $e_i \leftarrow b_i \lozenge d_i$
    $f_i \leftarrow U[i]$
    g: $V[i] \leftarrow b_i$
    h: $W[i] \leftarrow d_i$
    $j_i \leftarrow X[i]$
```

# Loop scheduling without resource bounds

<u>**Expand instructions**</u>

- no cycle has > 5 instructions
- Instructions in a row execute in parallel; reads in RHS happen before writes in LHS



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 8 | | | d | b | | | |
| 9 | | | e h | g | f j | | |
| 10 | | | | c | a | | |
| 11 | | | | d | b | | |
| 12 | | | | e h | g | f j | |
| 13 | | | | | c | a | |

2. Loop body – also increment counter and insert (modified) exit condition

incorrect index $a_i$ in MCIML book

**L:**

| | | | | |
|---|---|---|---|---|
| $d_i \leftarrow f_{i-1} \Diamond c_i$ | $b_{i+1} \leftarrow a_{i+1} \Diamond f_i$ | | | |
| $e_i \leftarrow b_i \Diamond d_i$ | $W[i] \leftarrow d_i$ | $V[i+1] \leftarrow b_{i+1}$ | $f_{i+2} \leftarrow U[i+2]$ | $j_{i+2} \leftarrow X[i+2]$ |
| $c_{i+1} \leftarrow e_i \Diamond j_i$ | $a_{i+2} \leftarrow j_{i+1} \Diamond b_{i+1}$ | $i \leftarrow i + 1$ | **if i < N-2 goto L** | |

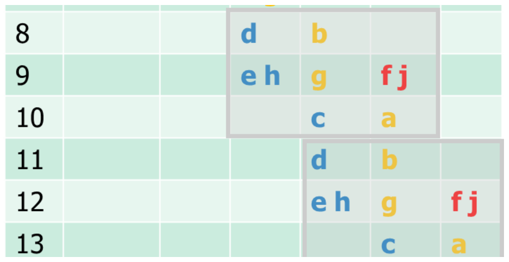
```
for i ← 1 to N
    $a_i \leftarrow j_{i-1} \Diamond b_{i-1}$
    $b_i \leftarrow a_i \Diamond f_{i-1}$
    $c_i \leftarrow e_{i-1} \Diamond j_{i-1}$
    $d_i \leftarrow f_{i-1} \Diamond c_i$
    $e_i \leftarrow b_i \Diamond d_i$
    $f_i \leftarrow U[i]$
    g: $V[i] \leftarrow b_i$
    h: $W[i] \leftarrow d_i$
    $j_i \leftarrow X[i]$
```

As expected, the loop body has one copy of each instruction a-j, plus induction variable update + test

# Loop scheduling without resource bounds

**Expand instructions**

- no cycle has > 5 instructions
- Instructions in a row execute in parallel; reads in RHS happen before writes in LHS



| | | | | | | |
|---|---|---|---|---|---|---|
| 8 | | | d | b | | |
| 9 | | | e h | g | f j | |
| 10 | | | | c | a | |
| 11 | | | | d | b | |
| 12 | | | | e h | g | f j |
| 13 | | | | | c | a |

3. Loop epilogue – finish all N iterations

| | | | | |
|---|---|---|---|---|
| $d_{N-1} \leftarrow f_{N-1} \Diamond c_{N-2}$ | $b_N \leftarrow a_N \Diamond f_{N-1}$ | | | |
| $e_{N-1} \leftarrow b_{N-1} \Diamond d_{N-1}$ | $W[N-1] \leftarrow d_{N-1}$ | $V[N] \leftarrow b_N$ | | |
| $c_N \leftarrow e_{N-1} \Diamond j_{N-1}$ | | | | |
| $d_N \leftarrow f_N \Diamond c_{N-1}$ | | | | |
| $e_N \leftarrow b_N \Diamond d_N$ | $W[N] \leftarrow d_N$ | | | |

```
for i ← 1 to N
    $a_i \leftarrow j_{i-1} \Diamond b_{i-1}$
    $b_i \leftarrow a_i \Diamond f_{i-1}$
    $c_i \leftarrow e_{i-1} \Diamond j_{i-1}$
    $d_i \leftarrow f_{i-1} \Diamond c_i$
    $e_i \leftarrow b_i \Diamond d_i$
    $f_i \leftarrow U[i]$
    g: $V[i] \leftarrow b_i$
    h: $W[i] \leftarrow d_i$
    $j_i \leftarrow X[i]$
```

# Loop scheduling without resource bounds

| | | | | |
|---|---|---|---|---|
| $a_1 \leftarrow j_0 \Diamond b_0$ | $c_1 \leftarrow e_0 \Diamond j_0$ | $f_1 \leftarrow U[1]$ | $j_1 \leftarrow X[1]$ | |
| $b_1 \leftarrow a_1 \Diamond f_0$ | $d_1 \leftarrow f_0 \Diamond c_1$ | $f_2 \leftarrow U[2]$ | $j_2 \leftarrow X[2]$ | |
| $e_1 \leftarrow b_1 \Diamond d_1$ | $V[1] \leftarrow b_1$ | $W[1] \leftarrow d_1$ | $a_2 \leftarrow j_1 \Diamond b_1$ | |
| $b_2 \leftarrow a_2 \Diamond f_1$ | $c_2 \leftarrow e_1 \Diamond j_1$ | $f_3 \leftarrow U[3]$ | $j_3 \leftarrow X[3]$ | |
| $d_2 \leftarrow f_1 \Diamond c_2$ | $V[2] \leftarrow b_2$ | $a_3 \leftarrow j_2 \Diamond b_2$ | | |
| $e_2 \leftarrow b_2 \Diamond d_2$ | $W[2] \leftarrow d_2$ | $b_3 \leftarrow a_3 \Diamond f_2$ | $f_4 \leftarrow U[4]$ | $j_4 \leftarrow X[4]$ |
| $c_3 \leftarrow e_2 \Diamond j_2$ | $V[3] \leftarrow b_3$ | $a_4 \leftarrow j_3 \Diamond b_3$ | | $i \leftarrow 3$ |

| | | | | |
|---|---|---|---|---|
| $d_i \leftarrow f_{i-1} \Diamond c_i$ | $b_{i+1} \leftarrow a_{i+1} \Diamond f_i$ | | | |
| $e_i \leftarrow b_i \Diamond d_i$ | $W[i] \leftarrow d_i$ | $V[i+1] \leftarrow b_{i+1}$ | $f_{i+2} \leftarrow U[i+2]$ | $j_{i+2} \leftarrow X[i+2]$ |
| $c_{i+1} \leftarrow e_i \Diamond j_i$ | $a_{i+2} \leftarrow j_{i+1} \Diamond b_{i+1}$ | $i \leftarrow i + 1$ | if $i < N-2$ goto L | |

| | | | | |
|---|---|---|---|---|
| $d_{N-1} \leftarrow f_{N-1} \Diamond c_{N-2}$ | $b_N \leftarrow a_N \Diamond f_{N-1}$ | | | |
| $e_{N-1} \leftarrow b_{N-1} \Diamond d_{N-1}$ | $W[N-1] \leftarrow d_{N-1}$ | $V[N] \leftarrow b_N$ | | |
| $c_N \leftarrow e_{N-1} \Diamond j_{N-1}$ | | | | |
| $d_N \leftarrow f_N \Diamond c_{N-1}$ | | | | |
| $e_N \leftarrow b_N \Diamond d_N$ | $W[N] \leftarrow d_N$ | | | |

## Loop scheduling without resource bounds

Final step: eliminate indices i from variables – want "constant" variables/registers in body!

| | | | | |
|---|---|---|---|---|
| $d_i \leftarrow f_{i-1} \Diamond c_i$ | $b_{i+1} \leftarrow a_{i+1} \Diamond f_i$ | | | |
| $e_i \leftarrow b_i \Diamond d_i$ | $W[i] \leftarrow d_i$ | $V[i+1] \leftarrow b_{i+1}$ | $f_{i+2} \leftarrow U[i+2]$ | $j_{i+2} \leftarrow X[i+2]$ |
| $c_{i+1} \leftarrow e_i \Diamond j_i$ | $a_{i+2} \leftarrow j_{i+1} \Diamond b_{i+1}$ | $i \leftarrow i + 1$ | if i < N-2 goto L | |

need 3 copies of j since up to 3 copies are live: $j_{i+2} \rightarrow j$, $j_{i+1} \rightarrow j'$, $j_i \rightarrow j''$

# Loop scheduling without resource bounds

Final step: eliminate indices i from variables – want "constant" variables/registers in body!

| | | | | |
|---|---|---|---|---|
| $d_i \leftarrow f_{i-1} \lozenge c_i$ | $b_{i+1} \leftarrow a_{i+1} \lozenge f_i$ | | | |
| $e_i \leftarrow b_i \lozenge d_i$ | $W[i] \leftarrow d_i$ | $V[i+1] \leftarrow b_{i+1}$ | $f_{i+2} \leftarrow U[i+2]$ | $j_{i+2} \leftarrow X[i+2]$ |
| $c_{i+1} \leftarrow e_i \lozenge j_i$ | $a_{i+2} \leftarrow j_{i+1} \lozenge b_{i+1}$ | $i \leftarrow i + 1$ | if i < N-2 goto L | |

need 3 copies of j since up to 3 copies are live: $j_{i+2} \rightarrow j$, $j_{i+1} \rightarrow j'$, $j_i \rightarrow j''$

| | | | | |
|---|---|---|---|---|
| $d_i \leftarrow f_{i-1} \lozenge c_i$ | $b_{i+1} \leftarrow a_{i+1} \lozenge f_i$ | | | |
| $e_i \leftarrow b_i \lozenge d_i$ | $W[i] \leftarrow d_i$ | $V[i+1] \leftarrow b_{i+1}$ | $f_{i+2} \leftarrow U[i+2]$ | $j \leftarrow X[i+2]$ |
| $c_{i+1} \leftarrow e_i \lozenge j''$ | $a_{i+2} \leftarrow j' \lozenge b_{i+1}$ | $i \leftarrow i + 1$ | if i < N-2 goto L | |

# Loop scheduling without resource bounds

Final step: eliminate indices i from variables – want "constant" variables/registers in body!

| | | | | |
|---|---|---|---|---|
| $d_i \leftarrow f_{i-1} \lozenge c_i$ | $b_{i+1} \leftarrow a_{i+1} \lozenge f_i$ | | | |
| $e_i \leftarrow b_i \lozenge d_i$ | $W[i] \leftarrow d_i$ | $V[i+1] \leftarrow b_{i+1}$ | $f_{i+2} \leftarrow U[i+2]$ | $j_{i+2} \leftarrow X[i+2]$ |
| $c_{i+1} \leftarrow e_i \lozenge j_i$ | $a_{i+2} \leftarrow j_{i+1} \lozenge b_{i+1}$ | $i \leftarrow i + 1$ | if i < N-2 goto L | |

need 3 copies of j since up to 3 copies are live: $j_{i+2} \rightarrow j$, $j_{i+1} \rightarrow j'$, $j_i \rightarrow j''$

| | | | | |
|---|---|---|---|---|
| $d_i \leftarrow f_{i-1} \lozenge c_i$ | $b_{i+1} \leftarrow a_{i+1} \lozenge f_i$ | $j'' \leftarrow j'$ | $j' \leftarrow j$ | |
| $e_i \leftarrow b_i \lozenge d_i$ | $W[i] \leftarrow d_i$ | $V[i+1] \leftarrow b_{i+1}$ | $f_{i+2} \leftarrow U[i+2]$ | $j \leftarrow X[i+2]$ |
| $c_{i+1} \leftarrow e_i \lozenge j''$ | $a_{i+2} \leftarrow j' \lozenge b_{i+1}$ | $i \leftarrow i + 1$ | if i < N-2 goto L | |

- the copies live across an iteration need to be updated in each iteration.
- also, need to initialize the live-in copies of the loop at the end of prologue (j, j')
- also, can replace the indexed live-in copies of the epilogue with primed versions
- all this for all variables a, ..j (see book – modulo typo regarding a, a')

# Loop scheduling without resource bounds

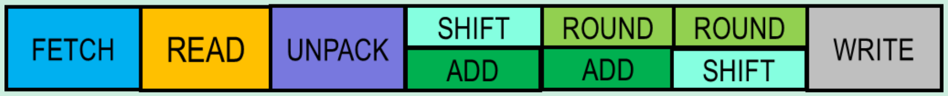
## Summary of main steps

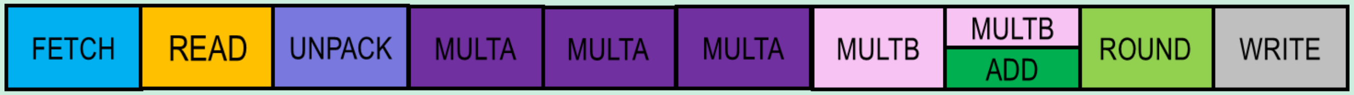
1. calculate data dependence graph of unrolled loop
2. schedule each instruction from each loop as early as possible
3. plot the tableau of iterations versus cycles
4. identify groups of instructions, and their slopes
5. coalesce the slopes by slowing down fast instruction groups
6. identify steady state, and loop prologue and epilogue
7. reroll the loop, removing the iteration-indexed variable names

## Loop scheduling **with** resource bounds

**Input:**

- data dependences of loop, with latency annotations
- resource requirements of all instruction forms:

ADD


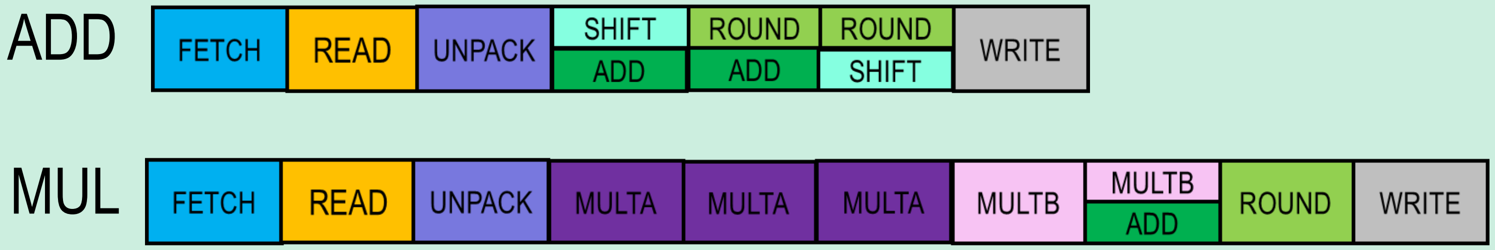
MUL


- #available Functional Units of each type, and descriptions of FU types:
  - # of instructions that can be issued in one cycle,
  - restrictions which instruction forms can be issued simultaneously etc

# Loop scheduling **with** resource bounds

**Input:**

- data dependences of loop, with latency annotations
- resource requirements of all instruction forms:



- #available Functional Units of each type, and descriptions of FU types:
  - # of instructions that can be issued in one cycle,
  - restrictions which instruction forms can be issued simultaneously etc

Modulo scheduling:

- find schedule that satisfies resource and (data) dependency requirements; **then** do register allocation
- try to schedule loop body using Δ cycles, for $\Delta = \Delta_{min}, \Delta_{min} + 1, \Delta_{min} + 2 \ldots$
- body surrounded by prologue and epilogue as before

# Modulo scheduling: where's the mod?

```
for i ← 1 to N
    $a_i ← j_{i-1} \Diamond b_{i-1}$
    $b_i ← a_i \Diamond f_{i-1}$
    $c_i ← e_{i-1} \Diamond j_{i-1}$
    $d_i ← f_{i-1} \Diamond c_i$
    $e_i ← b_i \Diamond d_i$
    $f_i ← U[i]$
    g: $V[i] ← b_i$
    h: $W[i] ← d_i$
    $j_i ← X[i]$
```

**Observation:** if resource constraints prevent an instruction from being scheduled at time **t**, they also prevent t from being scheduled at times **t** + Δ, **t** + 2Δ, . . . or indeed any **t'** with **t** = **t' mod Δ**.

**Example:** Δ=3, machine can only execute 1 load instruction at a time, loop body from previous example

| | | |
|---|---|---|
| 0 | | |
| 1 | $f_i ← U[i]$ | $j_i ← X[i]$ |
| 2 | | |

🚫

# Modulo scheduling: where's the mod?

```
for i ← 1 to N
    $a_i \leftarrow j_{i-1} \diamond b_{i-1}$
    $b_i \leftarrow a_i \diamond f_{i-1}$
    $c_i \leftarrow e_{i-1} \diamond j_{i-1}$
    $d_i \leftarrow f_{i-1} \diamond c_i$
    $e_i \leftarrow b_i \diamond d_i$
    $f_i \leftarrow U[i]$
    g: $V[i] \leftarrow b_i$
    h: $W[i] \leftarrow d_i$
    $j_i \leftarrow X[i]$
```

**Observation:** if resource constraints prevent an instruction from being scheduled at time **t**, they also prevent t from being scheduled at times **t** + Δ, **t** + 2Δ, . . . or indeed any **t'** with **t** = **t' mod Δ**.

**Example:** Δ=3, machine can only execute 1 load instruction at a time, loop body from previous example

| | | |
|---|---|---|
| 0 | | |
| 1 | $f_i \leftarrow U[i]$ | $j_i \leftarrow X[i]$ |
| 2 | | |

🚫

| | | |
|---|---|---|
| 0 | $f_i \leftarrow U[i]$ | |
| 1 | | $j_i \leftarrow X[i]$ |
| 2 | | |

| | | |
|---|---|---|
| 0 | | |
| 1 | | $j_i \leftarrow X[i]$ |
| 2 | $f_i \leftarrow U[i]$ | |

🙂

# Modulo scheduling: where's the mod?

```
for i ← 1 to N
    $a_i ← j_{i-1} ◊ b_{i-1}$
    $b_i ← a_i ◊ f_{i-1}$
    $c_i ← e_{i-1} ◊ j_{i-1}$
    $d_i ← f_{i-1} ◊ c_i$
    $e_i ← b_i ◊ d_i$
    $f_i ← U[i]$
    g: $V[i] ← b_i$
    h: $W[i] ← d_i$
    $j_i ← X[i]$
```

**Observation:** if resource constraints prevent an instruction from being scheduled at time t, they also prevent t from being scheduled at times t + Δ, t + 2Δ, . . . or indeed any t' with t = t' mod Δ.

**Example:** Δ=3, machine can only execute 1 load instruction at a time, loop body from previous example

| | | |
|---|---|---|
| 0 | | |
| 1 | $f_i ← U[i]$ | $j_i ← X[i]$ |
| 2 | | |

| | | |
|---|---|---|
| 0 | $f_i ← U[i]$ | |
| 1 | | $j_i ← X[i]$ |
| 2 | | |

| | | |
|---|---|---|
| 0=3 | $f_{i-1} ← U[i]$ | |
| 1 | | $j_i ← X[i]$ |
| 2 | | |

| | | |
|---|---|---|
| 0 | | |
| 1 | | $j_i ← X[i]$ |
| 2 | $f_i ← U[i]$ | |

| | | |
|---|---|---|
| 0 | | |
| 1 | | $j_i ← X[i]$ |
| 2=-1 | $f_{i+1} ← U[i]$ | |

# Modulo scheduling: where's the mod?

```
for i ← 1 to N
    $a_i \leftarrow j_{i-1} \diamond b_{i-1}$
    $b_i \leftarrow a_i \diamond f_{i-1}$
    $c_i \leftarrow e_{i-1} \diamond j_{i-1}$
    $d_i \leftarrow f_{i-1} \diamond c_i$
    $e_i \leftarrow b_i \diamond d_i$
    $f_i \leftarrow U[i]$
    g: $V[i] \leftarrow b_i$
    h: $W[i] \leftarrow d_i$
    $j_i \leftarrow X[i]$
```

**Observation:** if resource constraints prevent an instruction from being scheduled at time t, they also prevent t from being scheduled at times t + Δ, t + 2Δ, . . . or indeed any t' with t = t' mod Δ.

**Example:** Δ=3, machine can only execute 1 load instruction at a time, loop body from previous example

| | | |
|---|---|---|
| 0 | | |
| 1 | $f_i \leftarrow U[i]$ | $j_i \leftarrow X[i]$ |
| 2 | | |

| | | |
|---|---|---|
| 0 | | |
| 1=4 | $f_{i-1} \leftarrow U[i-1]$ | $j_i \leftarrow X[i]$ |
| 2 | | |

| | | |
|---|---|---|
| 0 | $f_i \leftarrow U[i]$ | |
| 1 | | $j_i \leftarrow X[i]$ |
| 2 | | |

| | | |
|---|---|---|
| 0=3 | $f_{i-1} \leftarrow U[i]$ | |
| 1 | | $j_i \leftarrow X[i]$ |
| 2 | | |

| | | |
|---|---|---|
| 0 | | |
| 1 | | $j_i \leftarrow X[i]$ |
| 2 | $f_i \leftarrow U[i]$ | |

| | | |
|---|---|---|
| 0 | | |
| 1 | | $j_i \leftarrow X[i]$ |
| 2=-1 | $f_{i+1} \leftarrow U[i]$ | |

## Modulo scheduling

**<u>Interaction with register allocation:</u>**

- **delaying** an instruction d: z $\leftarrow$ x op y
  - **extends** the liveness-range of d's **uses**, namely x and y;
    may overlap with other (iteration count-indexed) versions of z, so may need to maintain multiple copies, as in previous example
  - **shortens** liveness range of the **def(s)** of d, namely, z, to its **uses**;
    range < 1 illegal; ie need to postpone uses, too
- similarly, **schedudling an instruction earlier** **shortens** the liveness ranges of its **uses** and **extends** the liveness range of its **defs**
- hence, scheduling affects liveness/register allocation

## Modulo scheduling: estimating $\Delta_{min}$

**Identification of $\Delta_{min}$ as the maximum of the following:**

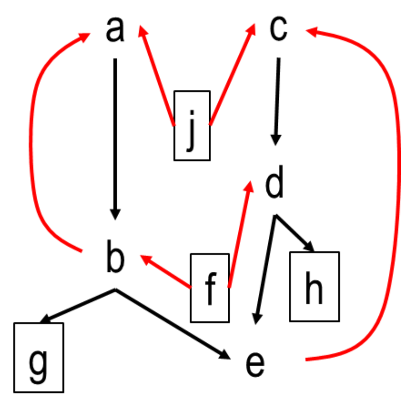
- resource estimator: for each FU
  - calculate requested cycles: add cycle requests of all instructions mapped to that FU
  - divide request by number of instances of the FU type
  - max over all FU types is lower bound on $\Delta_{max}$
- data-dependence estimator: sum of latencies along a simple cycle through the data dependence graph

# Modulo scheduling: estimating $\Delta_{min}$

**Identification of $\Delta_{min}$ as the maximum of the following:**

- resource estimator: for each FU
  - calculate requested cycles: add cycle requests of all instructions mapped to that FU
  - divide request by number of instances of the FU type
  - max over all FU types is lower bound on $\Delta_{max}$
- data-dependence estimator: sum of latencies along a simple cycle through the data dependence graph

Example: 1 ALU, 1 MEM; both issue 1 instruction/cycle; instr. latency 1 cycle



(MEM instructions in box)

Data dependence estimator: 3 ($c \rightarrow d \rightarrow e \rightarrow c$)

ALU-estimator: 5 instrs, 1 cycle each, 1 ALU $\Rightarrow$ 5

MEM-estimator: 4 instrs, 1 cycle each, 1 MEM $\Rightarrow$ 4

$\boxed{\text{Hence } \Delta_{min} = 5}$

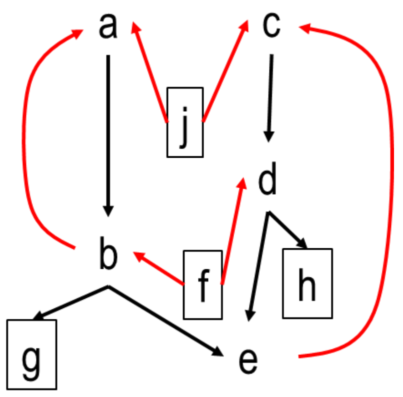
## Modulo scheduling: priority of instructions

**Algorithm schedules instructions according to priorities**

Possible metrics:

- membership in data dependence cycle of max latency
- execution on FU type that's most heavily used (resource estimate)

Example: [c, d, e, a, b, f, j, g, h]



(MEM instructions in box)

# Modulo scheduling: sketch of algorithm

Main data structures:

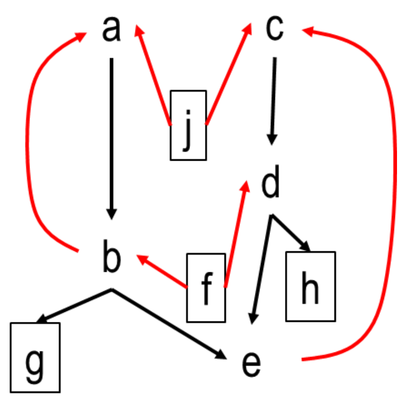
- array SchedTime, assigning to each instruction a cycle time
- table ResourceMap, assigning to each FU and cycle time < Δ an instruction

| | |
|---|---|
| Instr 1 | 8 |
| Instr 2 | 4 |
| Instr 3 | 0 |
| : | : |

| | FU1 | FU2 |
|---|---|---|
| 0 | Instr 1 | Instr 4 |
| 1 | Instr 2 | |
| 2 | | Instr 3 |
| : | : | : |

- pick highest-priority instruction that's not yet scheduled: i
- schedule i at earliest cycle that
  - respects the data dependencies w.r.t. the **already scheduled instructions**
  - has the right FU for i available
  - if i can't be scheduled for current Δ, place i without respecting resource constraint: evict current inhabitant and/or data-dependence successors of i hat are now scheduled too early. Evictees need to scheduled again.
- in principle evictions could go on forever
  - define a cut-off (heuristics) at which point Δ is increased

## Modulo scheduling: example



[c, d, e, a, b, f, j, g, h]

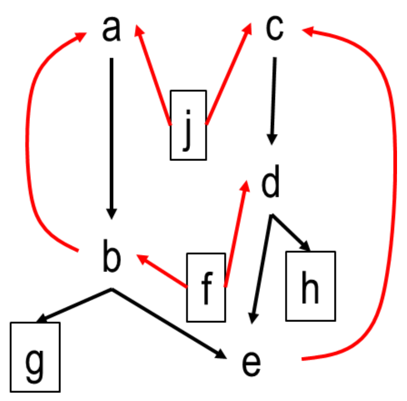
$\Delta_{min} = 5$

| | ALU | MEM |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |

| | |
|---|---|
| a | |
| b | |
| c | |
| d | |
| e | |
| f | |
| g | |
| h | |
| j | |

- highest-priority, unscheduled instruction: c
- earliest cycle with free ALU s.t. data-deps w.r.t. scheduled instructions are respected: 0

## Modulo scheduling: example



[~~c~~, d, e, a, b, f, j, g, h]

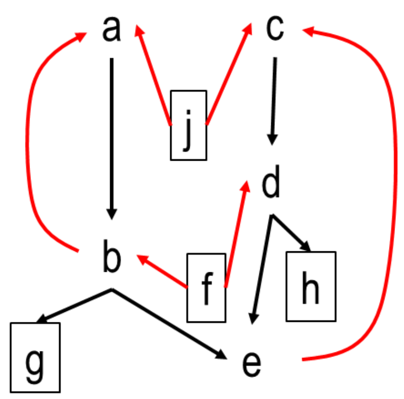
$\Delta_{min} = 5$

| | ALU | MEM |
|---|---|---|
| 0 | c | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |

| | |
|---|---|
| a | |
| b | |
| c | 0 |
| d | |
| e | |
| f | |
| g | |
| h | |
| j | |

- highest-priority, unscheduled instruction: c
- earliest cycle with free ALU s.t. data-deps w.r.t. scheduled instructions are respected: 0
- so schedule c in cycle 0

# Modulo scheduling: example



[~~c~~, ~~d~~, ~~e~~, ~~a~~, b, f, j, g, h]

$\Delta_{min} = 5$

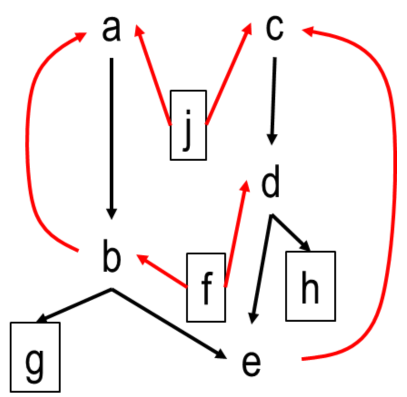
| | ALU | MEM |
|---|---|---|
| 0 | c | |
| 1 | d | |
| 2 | e | |
| 3 | a | |
| 4 | | |

| | |
|---|---|
| a | 3 |
| b | |
| c | 0 |
| d | 1 |
| e | 2 |
| f | |
| g | |
| h | |
| j | |

- highest-priority, unscheduled instruction: d
- earliest cycle with free ALU s.t. data-deps w.r.t. scheduled instructions are respected: 1
- so schedule d in cycle 1

Similarly: e → 2, a → 3. Next instruction: b

## Modulo scheduling: example



[~~c~~, ~~d~~, ~~e~~, ~~a~~, b, f, j, g, h]

$\Delta_{min} = 5$

| | ALU | MEM |
|---|---|---|
| 0 | c | |
| 1 | d | |
| 2 | e | |
| 3 | a | |
| 4 | | |

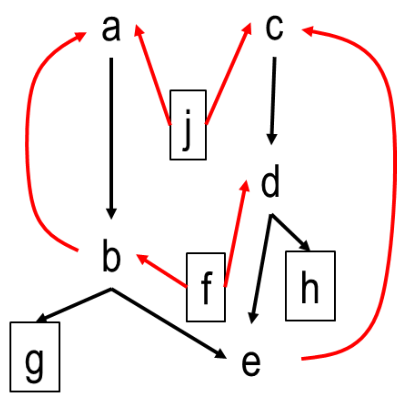
| | |
|---|---|
| a | 3 |
| b | |
| c | 0 |
| d | 1 |
| e | 2 |
| f | |
| g | |
| h | |
| j | |

- highest-priority, unscheduled instruction: d
- earliest cycle with free ALU s.t. data-deps w.r.t. scheduled instructions are respected: 1
- so schedule d in cycle 1

Similarly: e → 2, a → 3. Next instruction: b

Earliest cycle in which ALU is available: 4. But: b's successor e is scheduled in (earlier) cycle 2! Hence: place b in cycle 4, but evict e.

# Modulo scheduling: example



[~~c~~, ~~d~~, e, ~~a~~, ~~b~~, f, j, g, h]

$\Delta_{min} = 5$

| | ALU | MEM |
|---|---|---|
| 0 | c | |
| 1 | d | |
| 2 | ~~e~~ | |
| 3 | a | |
| 4 | b | |

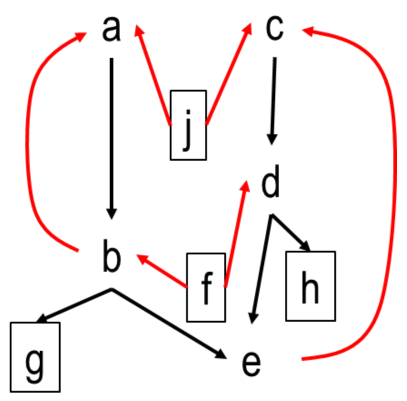
| | |
|---|---|
| a | 3 |
| b | 4 |
| c | 0 |
| d | 1 |
| e | ~~2~~ |
| f | |
| g | |
| h | |
| j | |

- highest-priority, unscheduled instruction: d
- earliest cycle with free ALU s.t. data-deps w.r.t. scheduled instructions are respected: 1
- so schedule d in cycle 1

Similarly: e → 2, a → 3. Next instruction: b

Earliest cycle in which ALU is available: 4. But: b's successor e is scheduled in (earlier) cycle 2! Hence: place b in cycle 4, but evict e.

# Modulo scheduling: example



[~~c~~, ~~d~~, e, ~~a~~, ~~b~~, f, j, g, h]

$\Delta_{min} = 5$

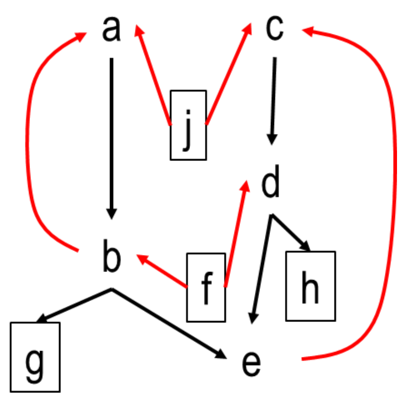
| | ALU | MEM |
|---|---|---|
| 0 | c | |
| 1 | d | |
| 2 | ~~e~~ | |
| 3 | a | |
| 4 | b | |

| | |
|---|---|
| a | 3 |
| b | 4 |
| c | 0 |
| d | 1 |
| e | ~~2~~ |
| f | |
| g | |
| h | |
| j | |

- highest-priority, unscheduled instruction: e
- ALU-slot for e: 2 (again)
- But: data dependence e → c violated – yes, cross iteration deps count!

So: schedule e in cycle 7 (= 2 mod Δ), but evict c - see next slide…

# Modulo scheduling: example



[c, ~~d~~, ~~e~~, ~~a~~, ~~b~~, f, j, g, h]

$\Delta_{min} = 5$

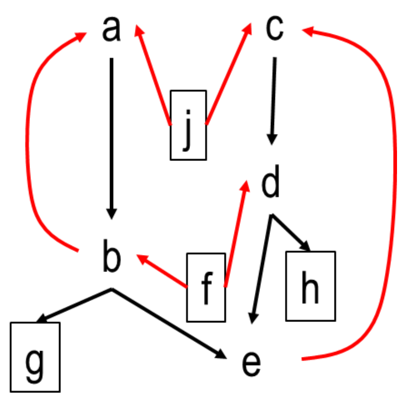
|   | ALU | MEM |
|---|---|---|
| 0 | ~~c~~ |  |
| 1 | d |  |
| 2 | ~~e~~ e |  |
| 3 | a |  |
| 4 | b |  |

|   |   |
|---|---|
| a | 3 |
| b | 4 |
| c | ~~0~~ |
| d | 1 |
| e | ~~2~~ 7 |
| f |  |
| g |  |
| h |  |
| j |  |

- highest-priority, unscheduled instruction: c
- ALU-slot for c: 0 (again)
- But: data dependence c → d violated

So, schedule c in cycle 5 (= 0 mod Δ), but evict d – see next slide…

# Modulo scheduling: example



[~~c~~, d, ~~e~~, ~~a~~, ~~b~~, f, j, g, h]

$\Delta_{min} = 5$

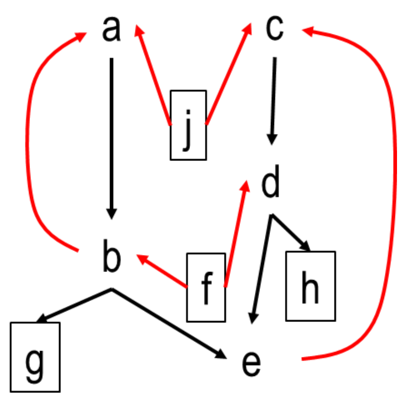
| | ALU | MEM |
|---|---|---|
| 0 | ~~c~~ c | |
| 1 | ~~d~~ | |
| 2 | ~~e~~ e | |
| 3 | a | |
| 4 | b | |

| | |
|---|---|
| a | 3 |
| b | 4 |
| c | ~~0~~ 5 |
| d | ~~1~~ |
| e | ~~2~~ 7 |
| f | |
| g | |
| h | |
| j | |

- highest-priority, unscheduled instruction: d
- ALU-slot for d: 1 (again)
- Hooray - data dependence d → e respected

So, schedule d in cycle 6 (= 1 mod Δ). No eviction – see next slide…

# Modulo scheduling: example



[~~c~~, ~~d~~, ~~e~~, ~~a~~, ~~b~~, f, j, g, h]

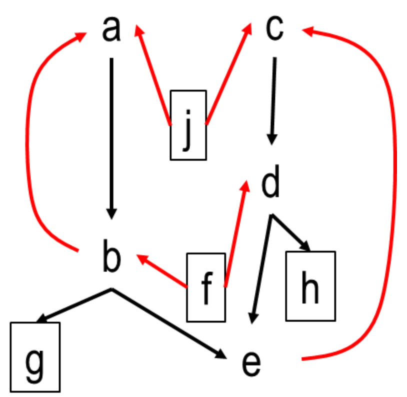
$\Delta_{min} = 5$

| | ALU | MEM |
|---|---|---|
| 0 | ~~c~~ c | |
| 1 | ~~d~~ d | |
| 2 | ~~e~~ e | |
| 3 | a | |
| 4 | b | |

| | |
|---|---|
| a | 3 |
| b | 4 |
| c | ~~0~~ 5 |
| d | ~~1~~ 6 |
| e | ~~2~~ 7 |
| f | |
| g | |
| h | |
| j | |

- highest-priority, unscheduled instruction: f
- MEM-slot for f: 0; no data-deps, so schedule f:0

# Modulo scheduling: example



[~~c~~, ~~d~~, ~~e~~, ~~a~~, ~~b~~, ~~f~~, j, g, h]

$\Delta_{min} = 5$

| | ALU | MEM |
|---|---|---|
| 0 | ~~c~~ c | f |
| 1 | ~~d~~ d | |
| 2 | ~~e~~ e | |
| 3 | a | |
| 4 | b | |

| | |
|---|---|
| a | 3 |
| b | 4 |
| c | ~~0~~ 5 |
| d | ~~1~~ 6 |
| e | ~~2~~ 7 |
| f | 0 |
| g | |
| h | |
| j | |

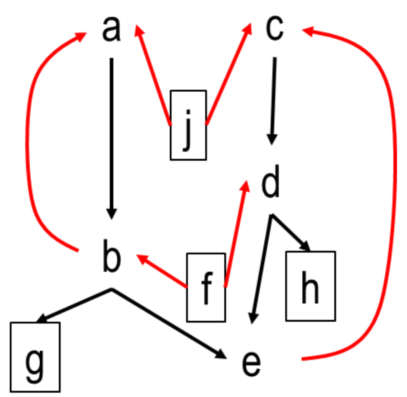
- highest-priority, unscheduled instruction: f
- MEM-slot for f: 0; no data-deps, so schedule f:0

- highest-priority, unscheduled instruction: j
- MEM-slot for j: 1; no data-deps, so schedule j:1

- highest-priority, unscheduled instruction: g
- MEM-slot for g: 2; and earliest cycle c = 2+ k*Δ where data-dep b→g is respected is 7.

So schedule g:7 – see next slide…

# Modulo scheduling: example



[~~c~~, ~~d~~, ~~e~~, ~~a~~, ~~b~~, ~~f~~, ~~j~~, ~~g~~, h]

$\Delta_{min} = 5$

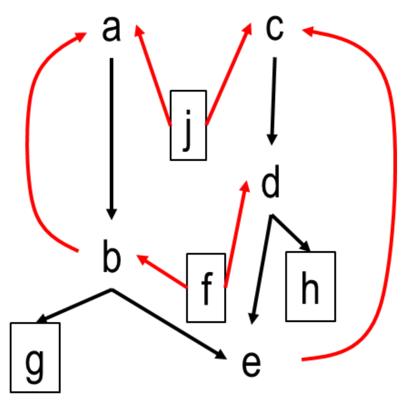
|   | ALU | MEM |
|---|-----|-----|
| 0 | ~~c~~ c | f |
| 1 | ~~d~~ d | j |
| 2 | ~~e~~ e | g |
| 3 | a |  |
| 4 | b |  |

|   |   |
|---|---|
| a | 3 |
| b | 4 |
| c | ~~0~~ 5 |
| d | ~~1~~ 6 |
| e | ~~2~~ 7 |
| f | 0 |
| g | 7 |
| h |  |
| j | 1 |

- highest-priority, unscheduled instruction: h
- MEM-slot for h: 3; earliest cycle c = 3 + k*Δ where data-dep d → h is respected is 8.

So schedule h:8 – final schedule on next slide.

# Modulo scheduling: example



[~~c~~, ~~d~~, ~~e~~, ~~a~~, ~~b~~, ~~f~~, ~~j~~, ~~g~~, ~~h~~]

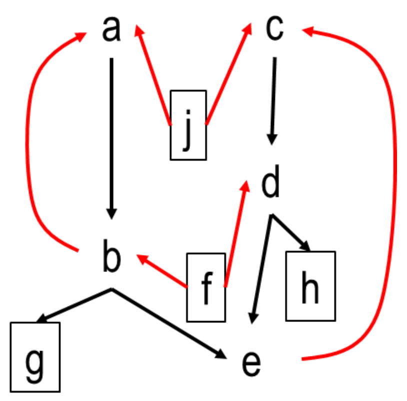
$\Delta_{min} = 5$

|  | ALU | MEM |
|---|---|---|
| 0 | c | f |
| 1 | d | j |
| 2 | e | g |
| 3 | a | h |
| 4 | b |  |

|  |  |
|---|---|
| a | 3 |
| b | 4 |
| c | 5 |
| d | 6 |
| e | 7 |
| f | 0 |
| g | 7 |
| h | 8 |
| j | 1 |

Instructions c, d, e, g, h are scheduled 1 iteration off.

# Modulo scheduling: example



[~~c~~, d, ~~e~~, ~~a~~, ~~b~~, f, j, g, h]

$\Delta_{min} = 5$

| | ALU | MEM |
|---|---|---|
| 0 | ~~c~~ c | |
| 1 | ~~d~~ | |
| 2 | ~~e~~ e | |
| 3 | a | |
| 4 | b | |

| | |
|---|---|
| a | 3 |
| b | 4 |
| c | ~~0~~ 5 |
| d | ~~1~~ |
| e | ~~2~~ 7 |
| f | |
| g | |
| h | |
| j | |

- highest-priority, unscheduled instruction: c
- ALU-slot for c: 0 (again)
- But: data dependence c → d violated

So, schedule c in cycle 5 (= 0 mod Δ), but evict d…

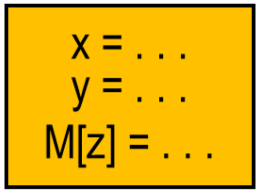
# Summary of scheduling

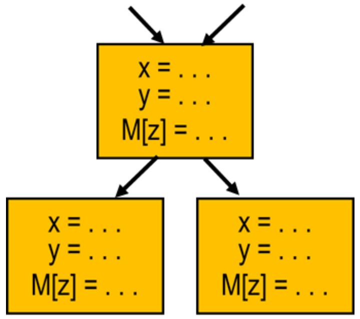
Challenges arise from interaction between

- program properties: data dependencies (RAW, WAR, WAW) and control dependencies
- hardware constraints (FU availability, latencies, …)

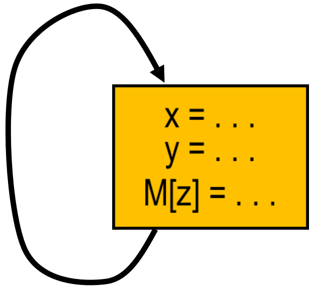
Optimal solutions typically infeasible → heuristics

Scheduling within a basic block (local): **list scheduling**





Scheduling across basic blocks (global): **trace scheduling**



Loop scheduling: **SW pipelining, modulo scheduling**