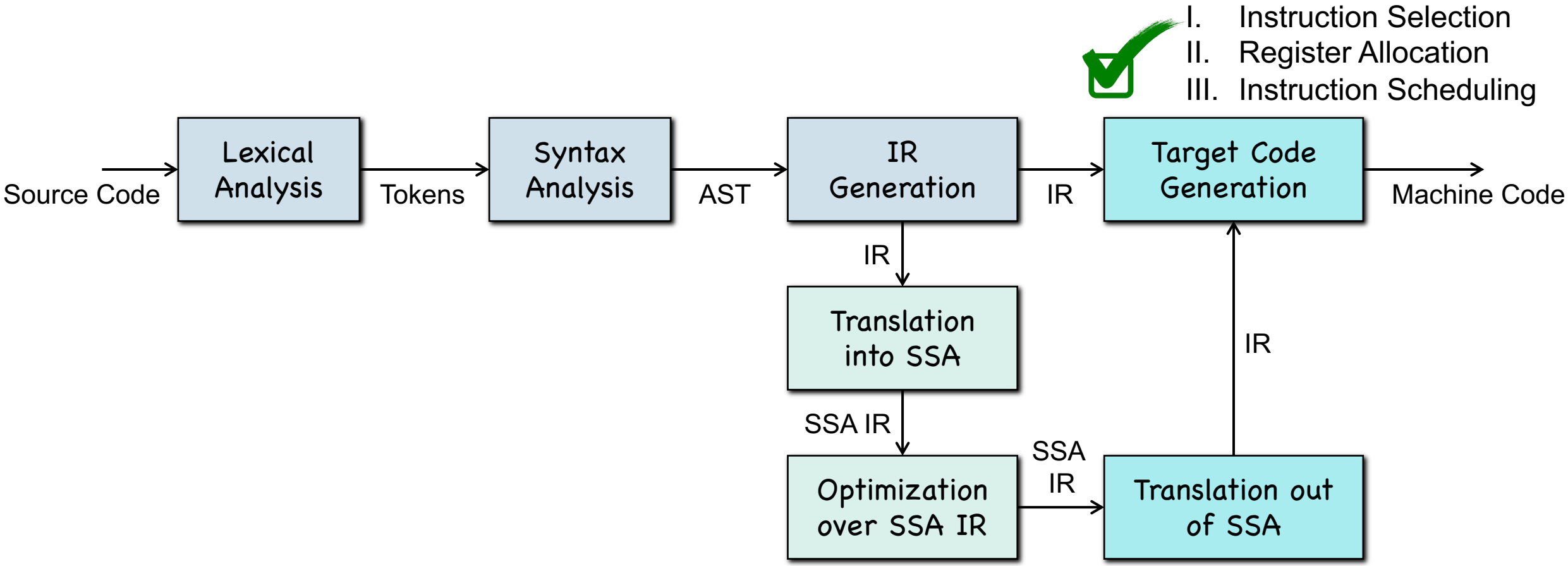
Made by Qingkai Shi
qingkaishi@nju.edu.cn
南京大学编译技术
及软件安全研究组

Recap-3
The Compilers' Back End

Back End of Compilers



PART I: Instruction Selection

Instructions

- Three categories of instructions

- Load/Store
  - LD *R0*, *addr*
  - LD *R0*, *R1*
  - LD *R0*, #500
  - ST *addr*, *R0*
- Calculation
  - OP *dst*, src_1, src_2
  - *e.g.*, SUB *R0*, *R1*, *R2*
- Jump
  - BR *L/addr*
  - Bcond *R*, *L/addr*
  - *e.g.*, BLTZ *R*, *L*

Addressing Modes

- What are addressing modes?

- Addressing Modes
  - LD R1, **a(R2)**
  - LD R1, **100(R2)**
  - LD R1, ***R2**
  - LD R1, ***100(R2)**
  - LD R1, **#100**

Addressing Modes: How we compute the addr

- LD R, ***addr***
- ST ***addr***, R

Note

假设 a 是一个元素为 8 字节值(比如实数)的数组。再假设 a 的元素的下标从 0 开始。我们可以通过下面的指令序列来执行三地址指令 b = a[i]:

```
LD   R1, i          // R1 = i
MUL  R1, R1, 8      // R1 = R1 * 8
LD   R2, a(R1)      // R2 = contents(a + contents(R1))
ST   b, R2          // b = R2
```

这里的第二步计算 $8i$; 而第三步把 a 的第 i 个元素的值放到 R2 中，这个元素位于离数组 a 的基地址 $8i$ 个字节的地方。

类似地，三地址指令 a[j] = c 所代表的对数组 a 的赋值可以实现为:

```
LD   R1, c          // R1 = c
LD   R2, j          // R2 = j
MUL  R2, R2, 8      // R2 = R2 * 8
ST   a(R2), R1      // contents(a +  contents(R2)) = R1
```

PART II: Register Allocation

Big Picture

- Why register allocation?

Big Picture

- Why register allocation?

- Local register allocation
- Global register allocation

Spilling

- What is spilling?

Local Register Allocation

- Steps of local register allocation

Local Register Allocation

- Steps of local register allocation
- 1. Compute MAXLIVE
- 2. if MAXLIVE ≤ k, …
- 3. if MAXLIVE > k, …

MAXLIVE

```
1. LD     R1     #1028          //  R1
2. LD     R2     *R1            //  R1  R2
3. MUL    R3     R1     R2      //  R1  R2  R3
4. LD     R4     x              //  R1  R2  R3  R4
5. SUB    R5     R4     R2      //  R1      R3      R5
6. LD     R6     z              //  R1      R3      R5  R6
7. MUL    R7     R5     R6      //  R1      R3              R7
8. SUB    R8     R7     R3      //  R1                          R8
9. ST     *R1    R8             //
```

MAXLIVE

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1     R2      // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R5     R4     R2      // R1    R3    R5
6. LD     R6     z              // R1    R3    R5 R6
7. MUL    R7     R5     R6      // R1    R3          R7
8. SUB    R8     R7     R3      // R1                   R8
9. ST     *R1    R8             //
```

MAXLIVE

```
1. LD     R1    #1028          // R1
2. LD     R2    *R1            // R1 R2
3. MUL    R3    R1    R2       // R1 R2 R3
4. LD     R4    x              // R1 R2 R3 R4
5. SUB    R5    R4    R2       // R1    R3    R5
6. LD     R6    z              // R1    R3    R5 R6
7. MUL    R7    R5    R6       // R1    R3          R7
8. SUB    R8    R7    R3       // R1                   R8
9. ST     *R1   R8             //
```

MAXLIVE = 4

MAXLIVE

```
1. LD     R1    #1028          // R1
2. LD     R2    *R1            // R1 R2
3. MUL    R3    R1    R2       // R1 R2 R3
4. LD     R4    x              // R1 R2 R3 R4
5. SUB    R5    R4    R2       // R1    R3    R5
6. LD     R6    z              // R1    R3    R5 R6
7. MUL    R7    R5    R6       // R1    R3          R7
8. SUB    R8    R7    R3       // R1                   R8
9. ST     *R1   R8             //
```

MAXLIVE = 4

Enough to have 4 registers

Local Register Allocation

```
1. LD     R1    #1028          // R1
2. LD     R2    *R1            // R1 R2
3. MUL    R3    R1    R2       // R1 R2 R3
4. LD     R4    x              // R1 R2 R3 R4
5. SUB    R5    R4    R2       // R1    R3    R5
6. LD     R6    z              // R1    R3    R5 R6
7. MUL    R7    R5    R6       // R1    R3          R7
8. SUB    R8    R7    R3       // R1                   R8
9. ST     *R1   R8             //
```

MAXLIVE = 4

if k ≥ 4, e.g., k = 4

Local Register Allocation

```
1. LD     R1    #1028          // R1
2. LD     R2    *R1            // R1 R2
3. MUL    R3    R1    R2       // R1 R2 R3
4. LD     R4    x              // R1 R2 R3 R4
5. SUB    R5    R4    R2       // R1    R3    R5
6. LD     R6    z              // R1    R3    R5 R6
7. MUL    R7    R5    R6       // R1    R3          R7
8. SUB    R8    R7    R3       // R1                   R8
9. ST     *R1   R8             //
```

MAXLIVE = 4

if k ≥ 4, e.g., k = 4

Local Register Allocation

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1     R2      // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R2     R4     R2      // R1    R3    R2
6. LD     R6     z              // R1    R3    R2 R6
7. MUL    R7     R2     R6      // R1    R3          R7
8. SUB    R8     R7     R3      // R1                   R8
9. ST     *R1    R8             //
```

MAXLIVE = 4

if k ≥ 4, e.g., k = 4

Local Register Allocation

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1     R2      // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R2     R4     R2      // R1    R3    R2
6. LD     R6     z              // R1    R3    R2 R6
7. MUL    R7     R2     R6      // R1    R3          R7
8. SUB    R8     R7     R3      // R1                   R8
9. ST     *R1    R8             //
```

MAXLIVE = 4

if k ≥ 4, e.g., k = 4

Local Register Allocation

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1     R2      // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R2     R4     R2      // R1    R3    R2
6. LD     R4     z              // R1    R3    R2 R4
7. MUL    R7     R2     R4      // R1    R3          R7
8. SUB    R8     R7     R3      // R1                   R8
9. ST     *R1    R8             //
```

MAXLIVE = 4

if k ≥ 4, e.g., k = 4

Local Register Allocation

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1     R2      // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R2     R4     R2      // R1    R3    R2
6. LD     R4     z              // R1    R3    R2 R4
7. MUL    R7     R2     R4      // R1    R3       R7
8. SUB    R8     R7     R3      // R1                R8
9. ST     *R1    R8             //
```

MAXLIVE = 4

if k ≥ 4, e.g., k = 4

Local Register Allocation

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1     R2      // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R2     R4     R2      // R1    R3    R2
6. LD     R4     z              // R1    R3    R2 R4
7. MUL    R2     R2     R4      // R1    R3          R2
8. SUB    R8     R2     R3      // R1                   R8
9. ST     *R1    R8             //
```

MAXLIVE = 4

if k ≥ 4, e.g., k = 4

Local Register Allocation

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1     R2      // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R2     R4     R2      // R1    R3    R2
6. LD     R4     z              // R1    R3    R2 R4
7. MUL    R2     R2     R4      // R1    R3          R2
8. SUB    R8     R2     R3      // R1                R8
9. ST     *R1    R8             //
```

MAXLIVE = 4

if k ≥ 4, e.g., k = 4

Local Register Allocation

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1     R2      // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R2     R4     R2      // R1    R3    R2
6. LD     R4     z              // R1    R3    R2 R4
7. MUL    R2     R2     R4      // R1    R3          R2
8. SUB    R2     R2     R3      // R1                   R2
9. ST     *R1    R2             //
```

MAXLIVE = 4

if k ≥ 4, e.g., k = 4

Local Register Allocation

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1     R2      // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R5     R4     R2      // R1    R3    R5
6. LD     R6     z              // R1    R3    R5 R6
7. MUL    R7     R5     R6      // R1    R3          R7
8. SUB    R8     R7     R3      // R1                   R8
9. ST     *R1    R8             //
```

MAXLIVE = 4

if k < 4, *e.g.*, 3

Local Register Allocation

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1     R2      // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R5     R4     R2      // R1    R3    R5
6. LD     R6     z              // R1    R3    R5 R6
7. MUL    R7     R5     R6      // R1    R3          R7
8. SUB    R8     R7     R3      // R1                   R8
9. ST     *R1    R8             //
```

MAXLIVE = 4

if k < 4, *e.g.*, 3

Local Register Allocation

```
1. LD     R1    #1028          // R1
2. LD     R2    *R1            // R1 R2
3. MUL    R3    R1    R2       // R1 R2 R3
4. LD     R4    x              // R1 R2 R3 R4
5. SUB    R5    R4    R2       // R1    R3    R5
6. LD     R6    z              // R1    R3    R5 R6
7. MUL    R7    R5    R6       // R1    R3          R7
8. SUB    R8    R7    R3       // R1                   R8
9. ST     *R1   R8             //
```

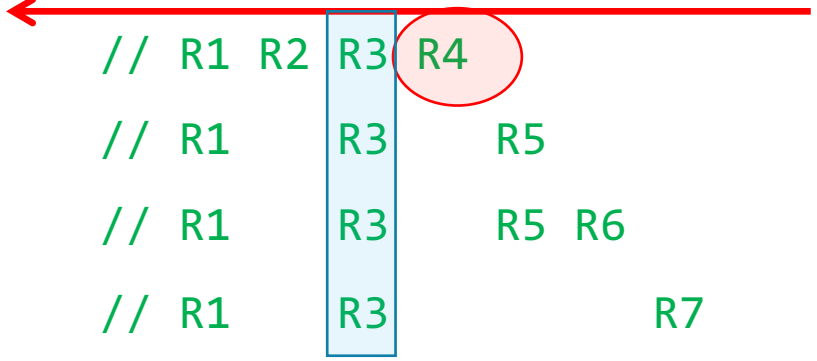
Spill R3 by `ST v R3`

MAXLIVE = 4

if k < 4, *e.g.*, 3

Local Register Allocation

```
1. LD     R1    #1028          // R1
2. LD     R2    *R1            // R1 R2
3. MUL    R3    R1     R2      // R1 R2 R3
4. LD     R4    x              // R1 R2 R3 R4
5. SUB    R5    R4     R2      // R1    R3    R5
6. LD     R6    z              // R1    R3    R5 R6
7. MUL    R7    R5     R6      // R1    R3          R7
8. SUB    R8    R7     R3      // R1                   R8
9. ST     *R1   R8             //
```



Spill R3 by ST v R3

MAXLIVE = 4

if k < 4, *e.g.*, 3

Local Register Allocation

```
1. LD     R1    #1028          // R1
2. LD     R2    *R1            // R1 R2
3. MUL    R3    R1     R2      // R1 R2 R3
4. LD     R4    x              // R1 R2    R4
5. SUB    R5    R4     R2      // R1          R5
6. LD     R6    z              // R1          R5 R6
7. MUL    R7    R5     R6      // R1                R7
8. SUB    R8    R7     R3      // R1                   R8
9. ST     *R1   R8             //
```

Spill R3 by ST v R3

MAXLIVE = 4

if k < 4, *e.g.*, 3

Local Register Allocation

```
1. LD     R1    #1028          // R1
2. LD     R2    *R1            // R1 R2
3. MUL    R3    R1    R2       // R1 R2 R3
4. LD     R4    x              // R1 R2    R4
5. SUB    R5    R4    R2       // R1            R5
6. LD     R6    z              // R1            R5 R6
7. MUL    R7    R5    R6       // R1                  R7
8. SUB    R8    R7    R3       // R1                     R8
9. ST     *R1   R8             //
```

Spill R3 by ST v R3

MAXLIVE = 4

if k < 4, *e.g.*, 3

Reload R3 by LD R3 v

Local Register Allocation

```
1. LD     R1    #1028          // R1
2. LD     R2    *R1            // R1 R2
3. MUL    R3    R1     R2      // R1 R2 R3
4. LD     R3    x              // R1 R2    R3
5. SUB    R5    R3     R2      // R1          R5
6. LD     R6    z              // R1          R5 R6
7. MUL    R7    R5     R6      // R1                R7
8. SUB    R8    R7     R3      // R1                   R8
9. ST     *R1   R8             //
```

Spill R3 by ST v R3

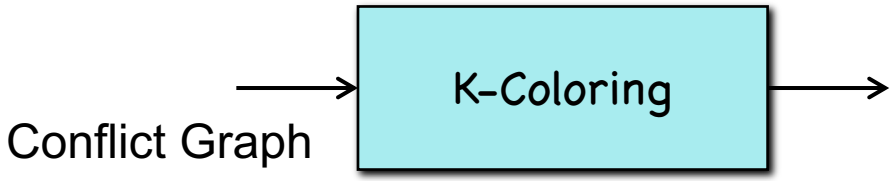
MAXLIVE = 4

if k < 4, *e.g.*, 3

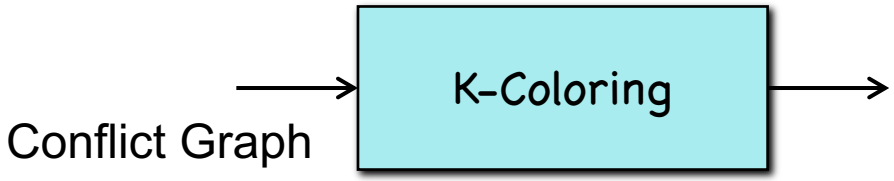
Reload R3 by LD R3 v

Global Register Allocation

Global Register Allocation

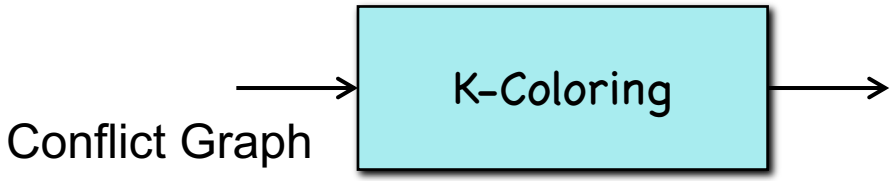


Global Register Allocation



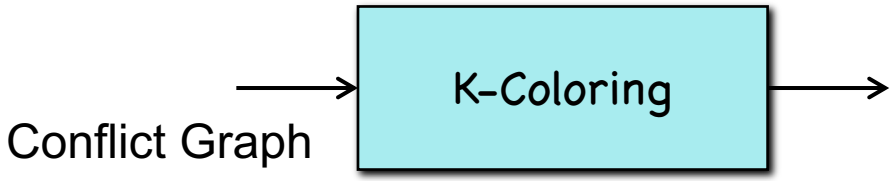

```
1. LD   R1  #1028
2. LD   R2  *R1
3. ST   a   R2
4. MUL  R3  R1    R1
5. ST   x   R3
6. LD   R2  a
7. ST   y   R2
8. ST   z   R1
```

Global Register Allocation

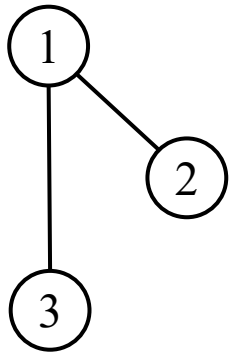



```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1      R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```

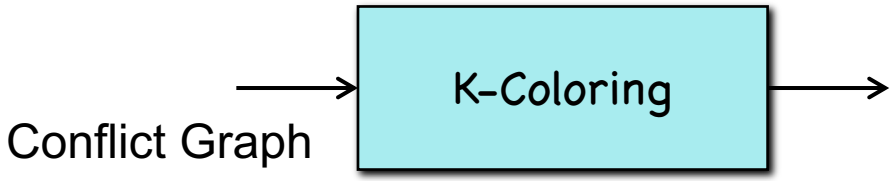
Global Register Allocation



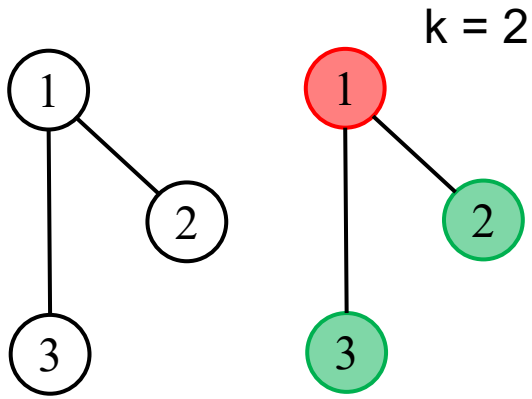

```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1      R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```



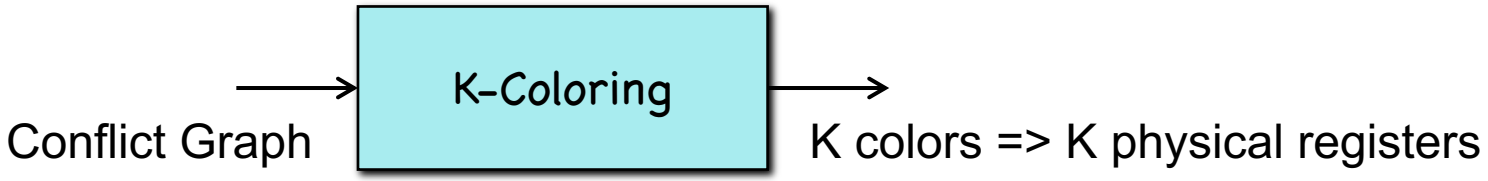
Global Register Allocation



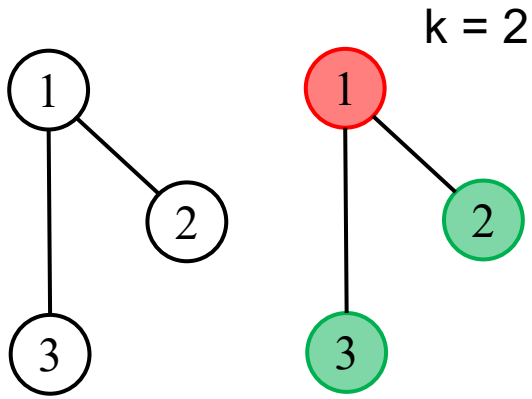

```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1   R1     // R1      R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```



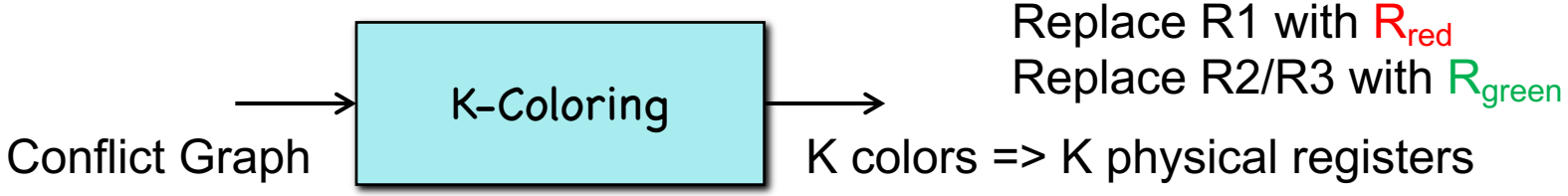
Global Register Allocation



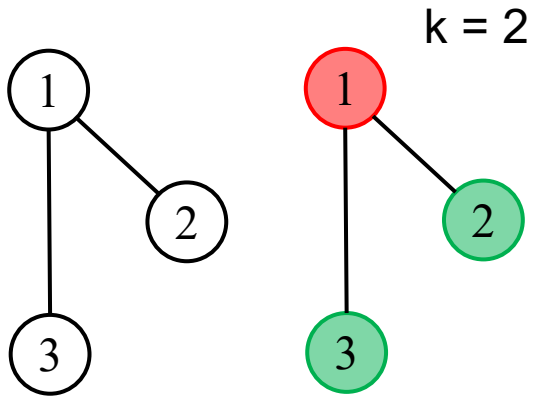

```
1. LD    R1  #1028        // R1
2. LD    R2  *R1          // R1 R2
3. ST    a   R2           // R1
4. MUL   R3  R1    R1     // R1      R3
5. ST    x   R3           // R1
6. LD    R2  a            // R1 R2
7. ST    y   R2           // R1
8. ST    z   R1           //
```



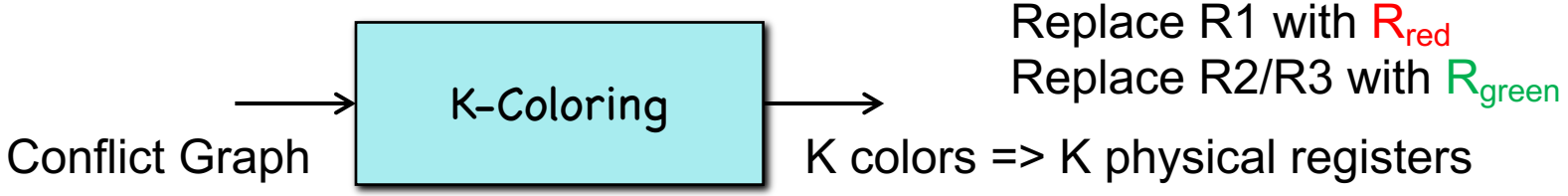
Global Register Allocation



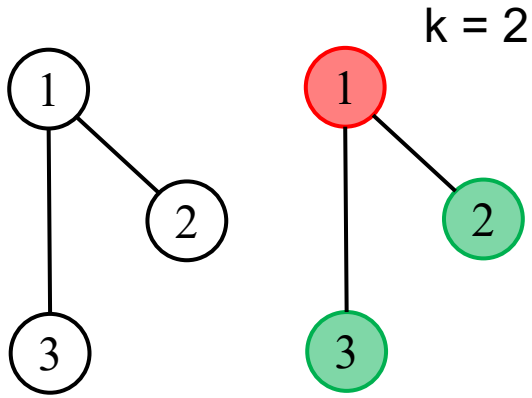

```
1. LD    R1  #1028        // R1
2. LD    R2  *R1          // R1 R2
3. ST    a   R2           // R1
4. MUL   R3  R1    R1     // R1      R3
5. ST    x   R3           // R1
6. LD    R2  a            // R1 R2
7. ST    y   R2           // R1
8. ST    z   R1           //
```



Global Register Allocation



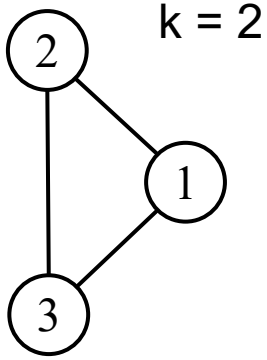

```
1. LD   R1  #1028          // R1
2. LD   R2  *R1            // R1 R2
3. ST   a   R2             // R1
4. MUL  R3  R1    R1       // R1      R3
5. ST   x   R3             // R1
6. LD   R2  a              // R1 R2
7. ST   y   R2             // R1
8. ST   z   R1             //
```



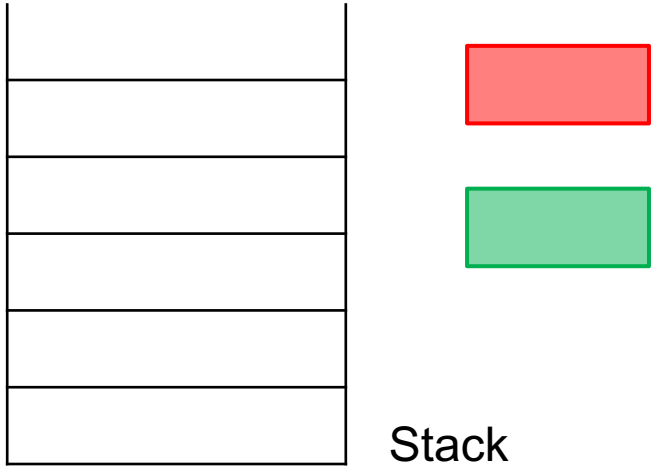

```
1. LD   Rred    #1028
2. LD   Rgreen  *Rred
3. ST   a       Rgreen
4. MUL  Rgreen  Rred      Rred
5. ST   x       Rgreen
6. LD   Rgreen  a
7. ST   y       Rgreen
8. ST   z       Rred
```

Chaitin's Algorithm

```
1. LD    R1  #1028        // R1
2. LD    R2  *R1          // R1 R2
3. MUL   R3  R1    R2    // R1 R2 R3
4. ST    x   R3           // R1 R2
5. ST    y   R2           // R1
6. ST    z   R1           //
```



k = 2

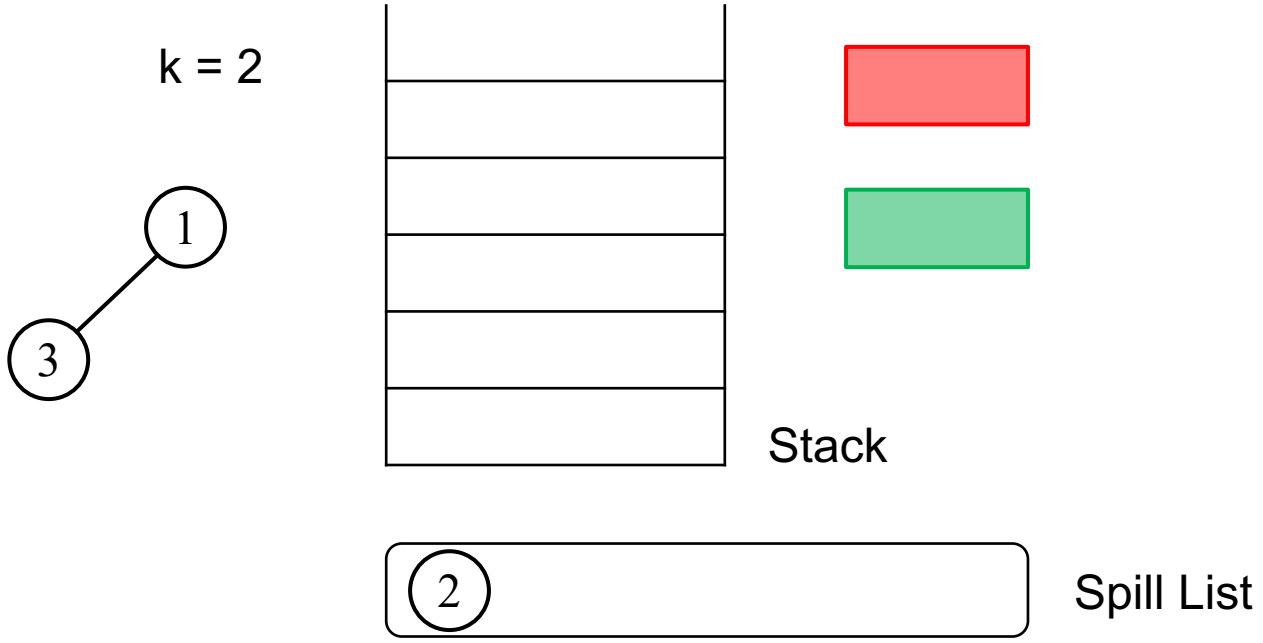


Stack

Spill List

Chaitin's Algorithm

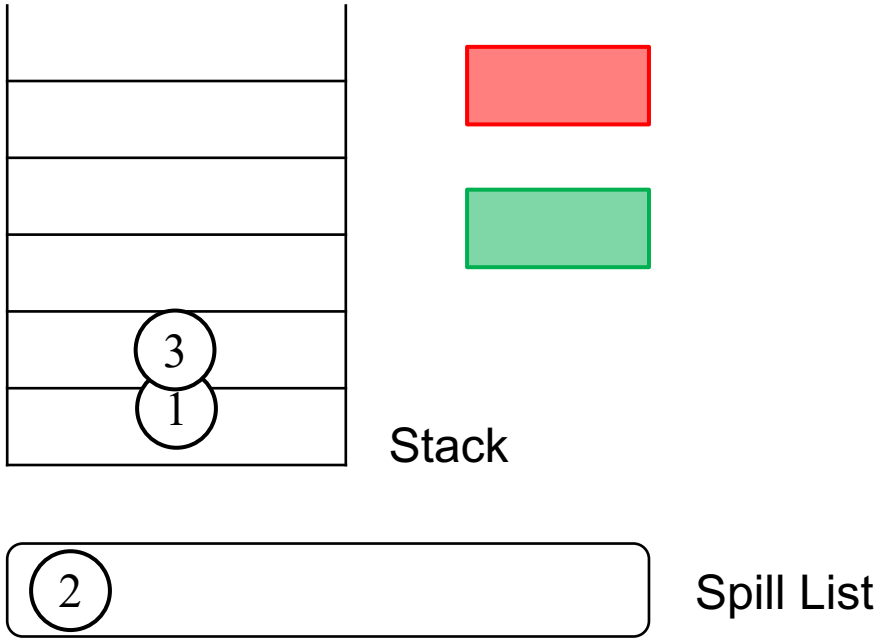
```
1. LD    R1  #1028        // R1
2. LD    R2  *R1          // R1 R2
3. MUL   R3  R1    R2    // R1 R2 R3
4. ST    x   R3           // R1 R2
5. ST    y   R2           // R1
6. ST    z   R1           //
```



Chaitin's Algorithm

```
1. LD    R1  #1028        // R1
2. LD    R2  *R1          // R1 R2
3. MUL   R3  R1    R2    // R1 R2 R3
4. ST    x   R3           // R1 R2
5. ST    y   R2           // R1
6. ST    z   R1           //
```

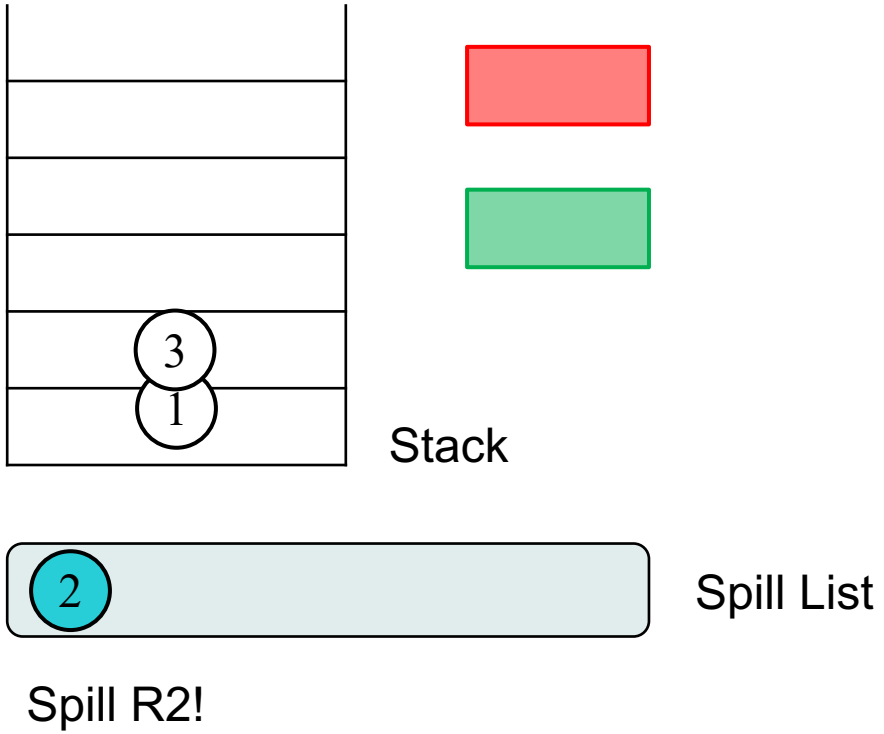
k = 2



Chaitin's Algorithm

```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. MUL  R3  R1    R2    // R1 R2 R3
4. ST   x   R3          // R1 R2
5. ST   y   R2          // R1
6. ST   z   R1          //
```

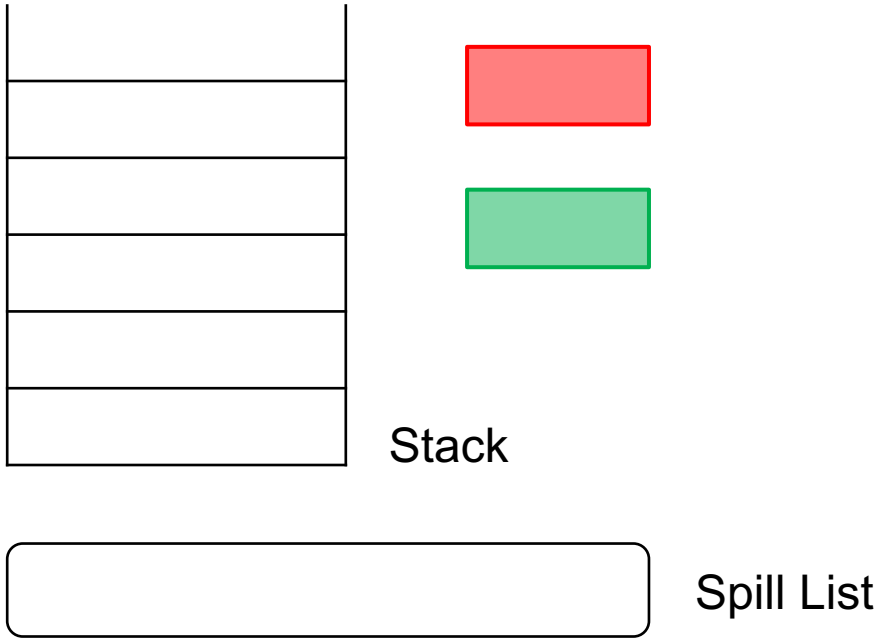
k = 2



Chaitin's Algorithm

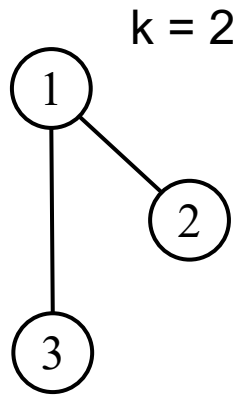
```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1     R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```

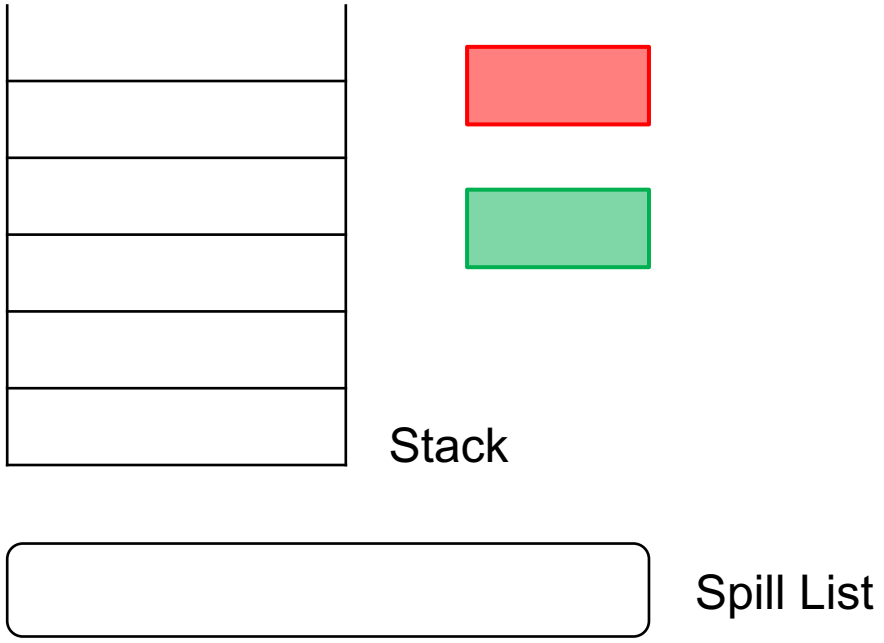
k = 2



Chaitin's Algorithm

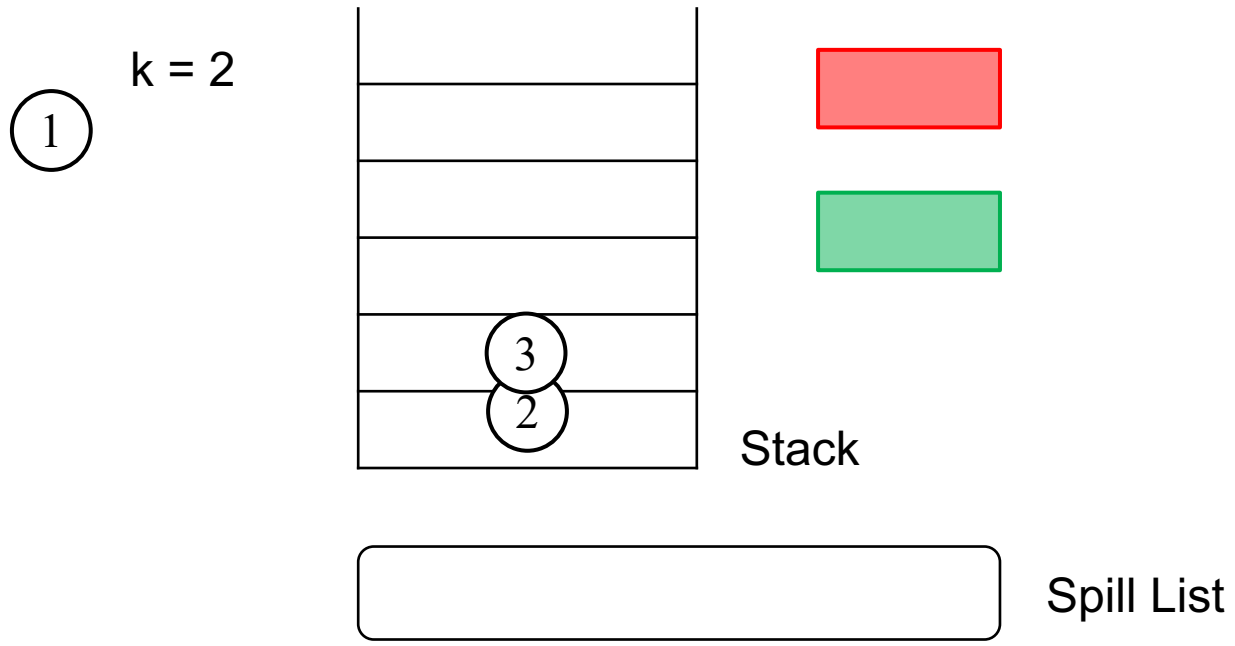
```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1      R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```





Chaitin's Algorithm

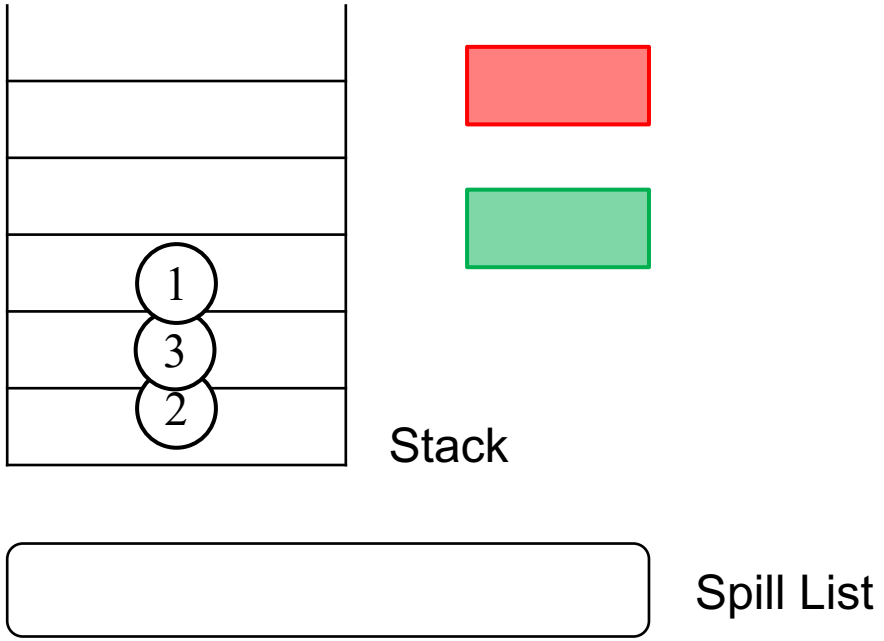
```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1      R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```



Chaitin's Algorithm

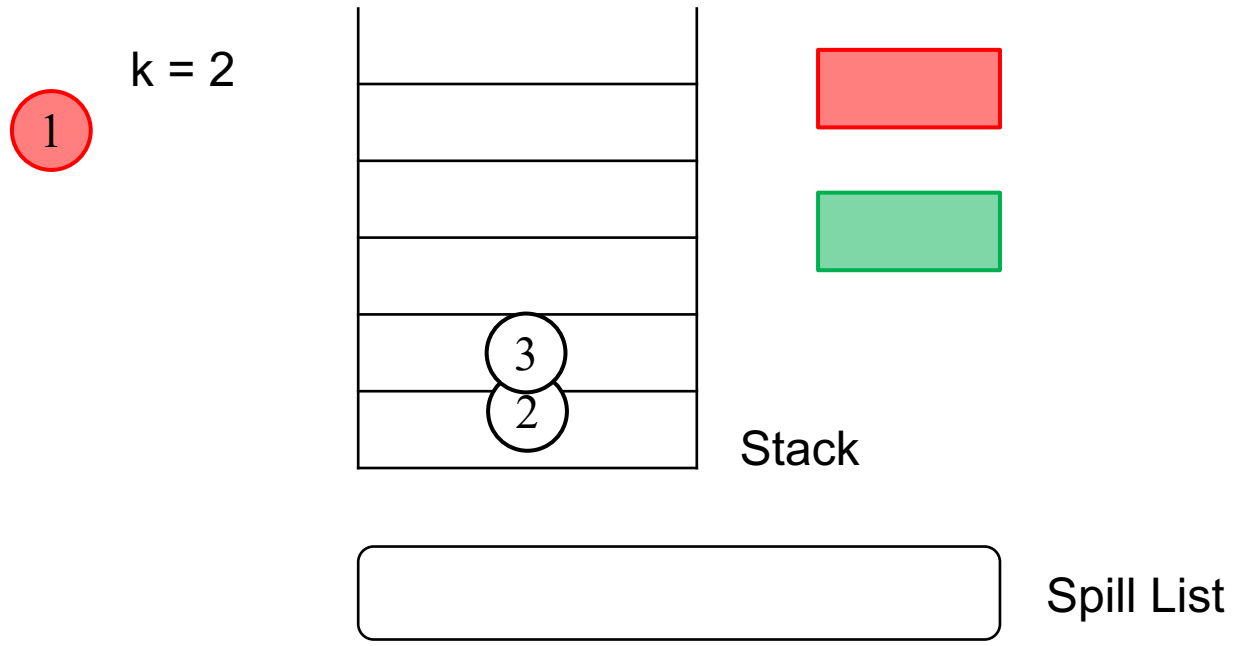
```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1      R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```

k = 2



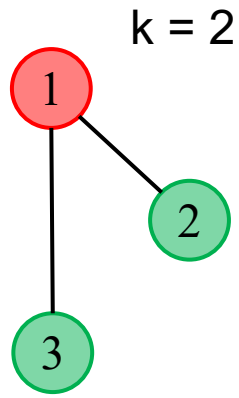
Chaitin's Algorithm

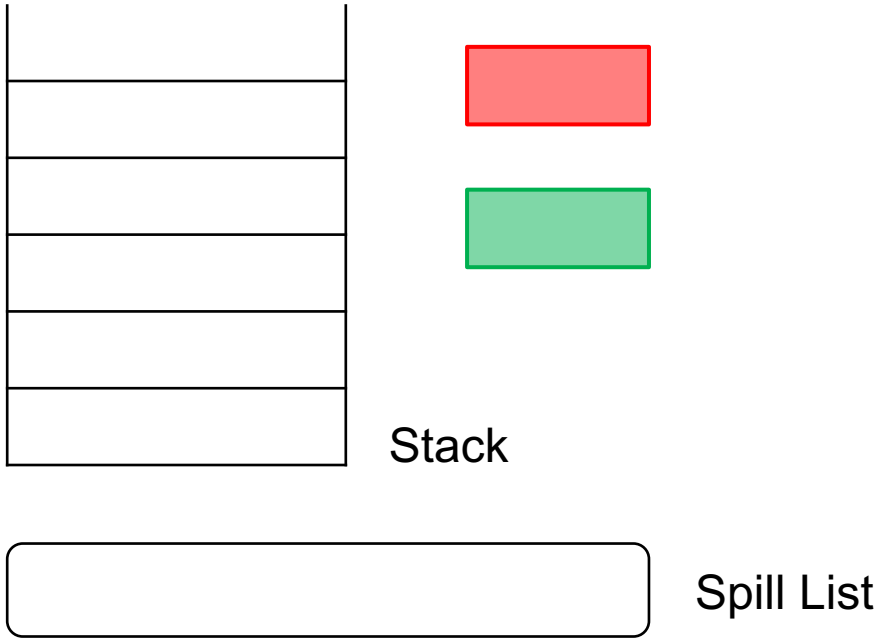
```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1      R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```



Chaitin's Algorithm

```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1      R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```

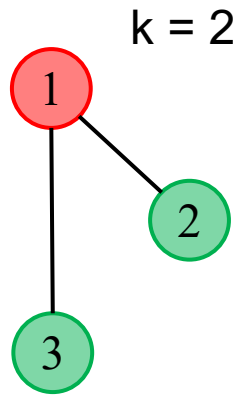


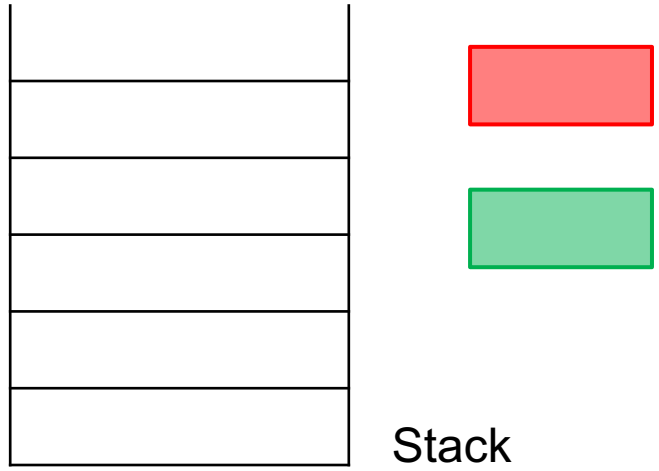


Chaitin's Algorithm

Replace R1 with R_{red}
Replace R2/R3 with R_{green}

```
1. LD    R1  #1028        // R1
2. LD    R2  *R1          // R1 R2
3. ST    a   R2           // R1
4. MUL   R3  R1    R1     // R1      R3
5. ST    x   R3           // R1
6. LD    R2  a            // R1 R2
7. ST    y   R2           // R1
8. ST    z   R1           //
```

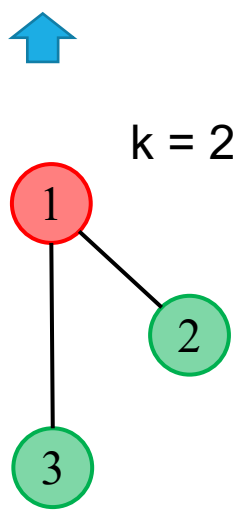


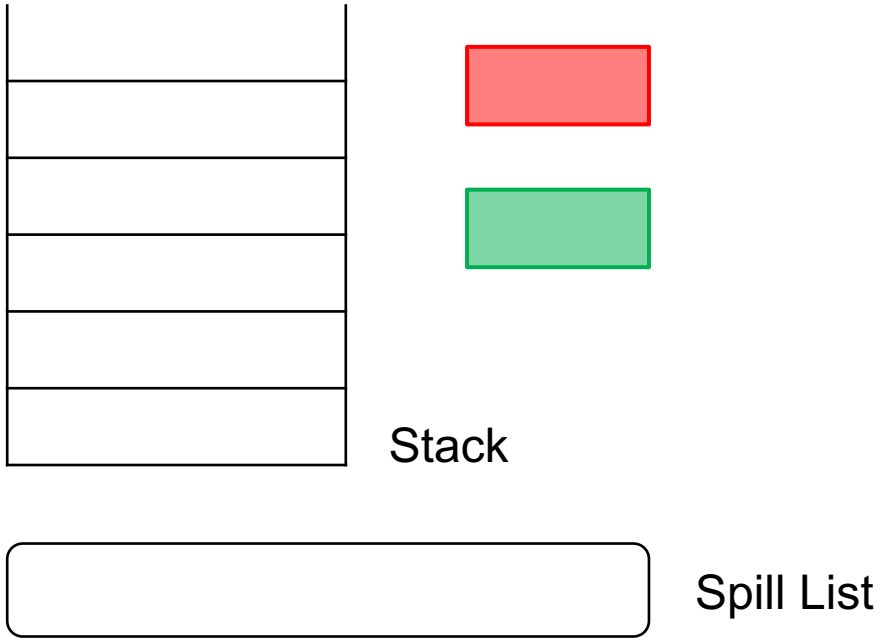


Chaitin's Algorithm

Replace R1 with R_{red}
Replace R2/R3 with R_{green}

```
1. LD    R_red     #1028
2. LD    R_green   *R_red
3. ST    a         R_green
4. MUL   R_green   R_red     R_red
5. ST    x         R_green
6. LD    R_green   a
7. ST    y         R_green
8. ST    z         R_red
```





PART III: Instruction Scheduling

Why Scheduling?

- Every modern high-performance processor can execute several operations in a single clock cycle.

Why Compilers Matter?

- Hardware only can execute instructions that have been fetched
- Hardware has limited space to buffer stalled operations

- Compilers can place independent operations close together to allow better hardware utilization

In-Block Scheduling

- True/Fake Dependence
- Dependence Graph
- List Scheduling

In-Block Scheduling

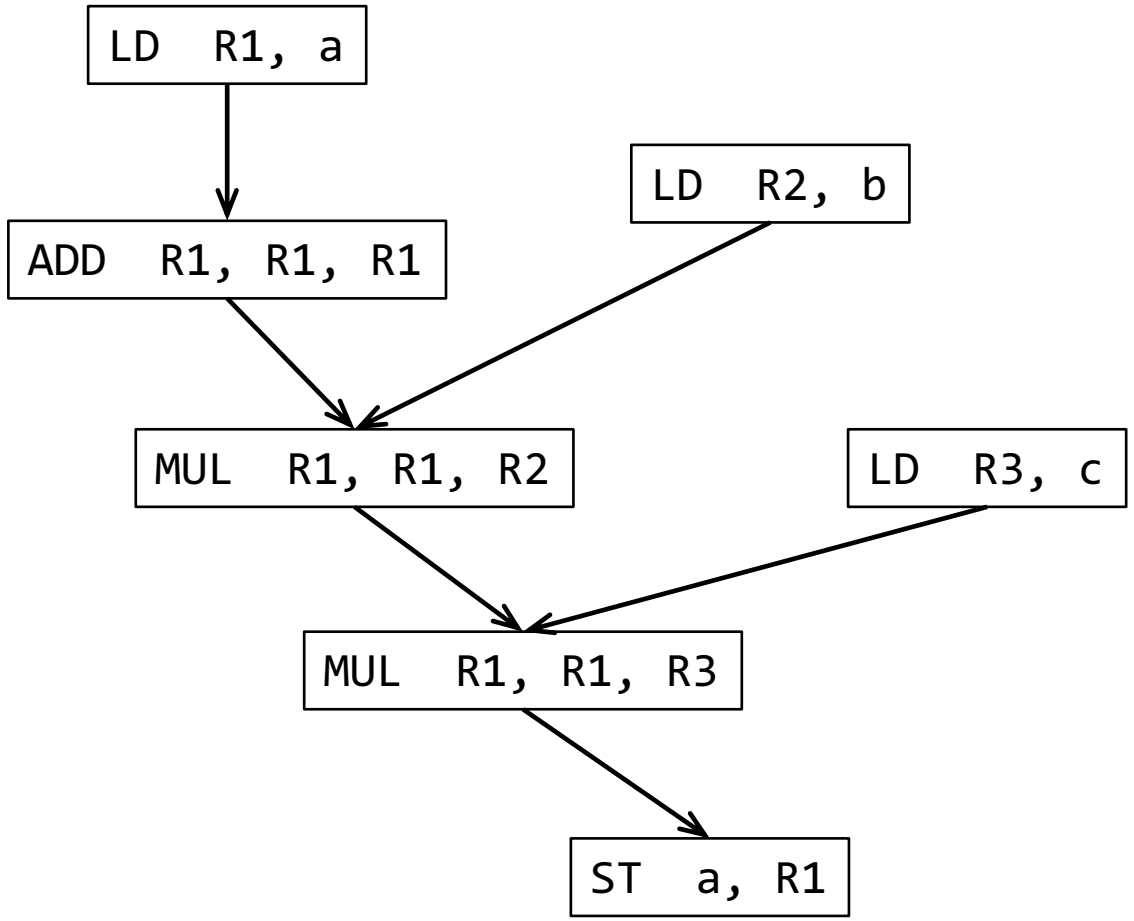
- True/Fake Dependence
- Dependence Graph
- List Scheduling

```
LD   R1,  a
ADD  R1,  R1,  R1
LD   R2,  b
MUL  R1,  R1,  R2
LD   R2,  c
MUL  R1,  R1,  R2
ST   a,   R1
```

In-Block Scheduling

- True/Fake Dependence
- Dependence Graph
- List Scheduling

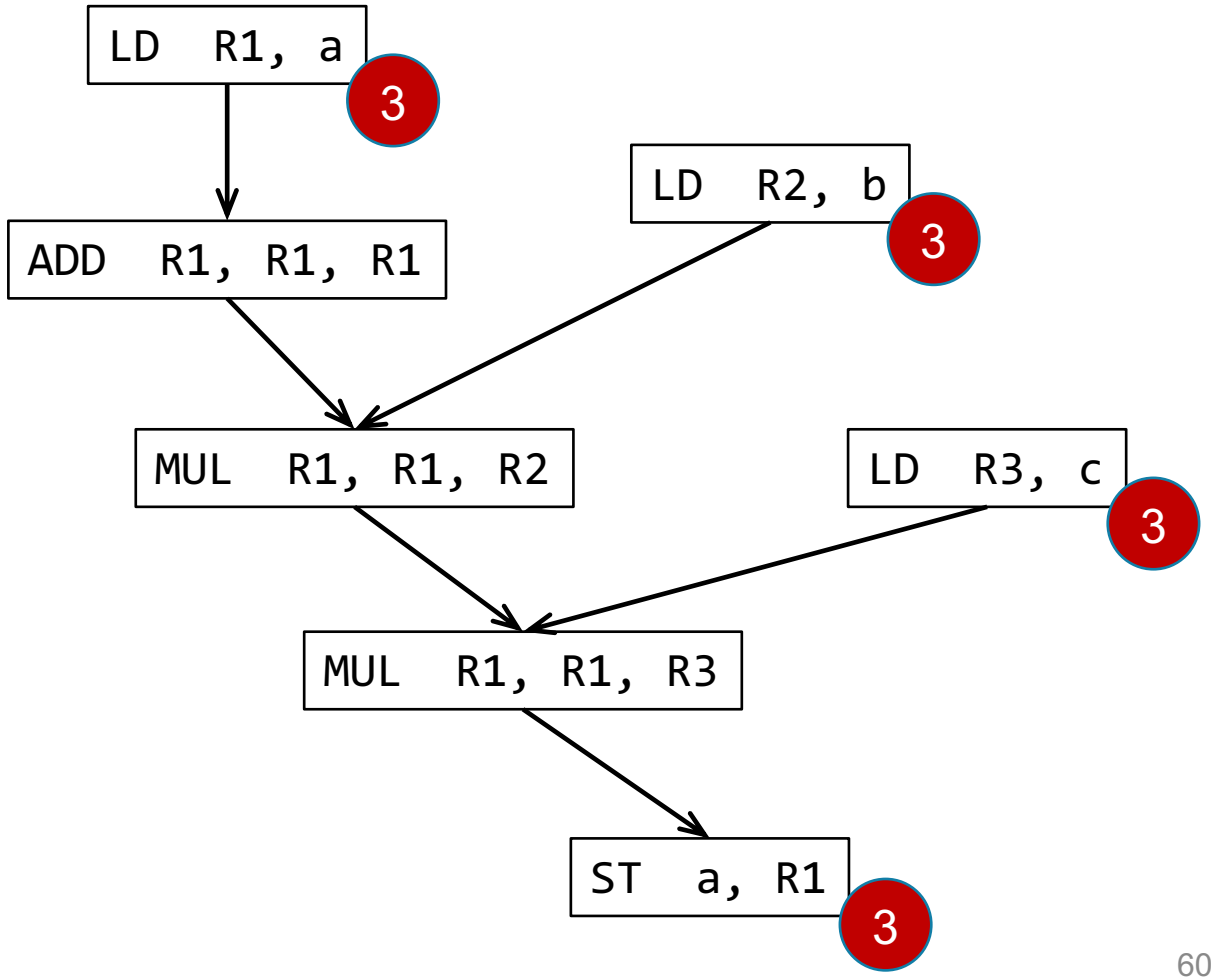
LD	R1,	a	
ADD	R1,	R1,	R1
LD	R2,	b	
MUL	R1,	R1,	R2
LD	R2,	c	
MUL	R1,	R1,	R2
ST	a,	R1	



In-Block Scheduling

- True/Fake Dependence
- Dependence Graph
- List Scheduling

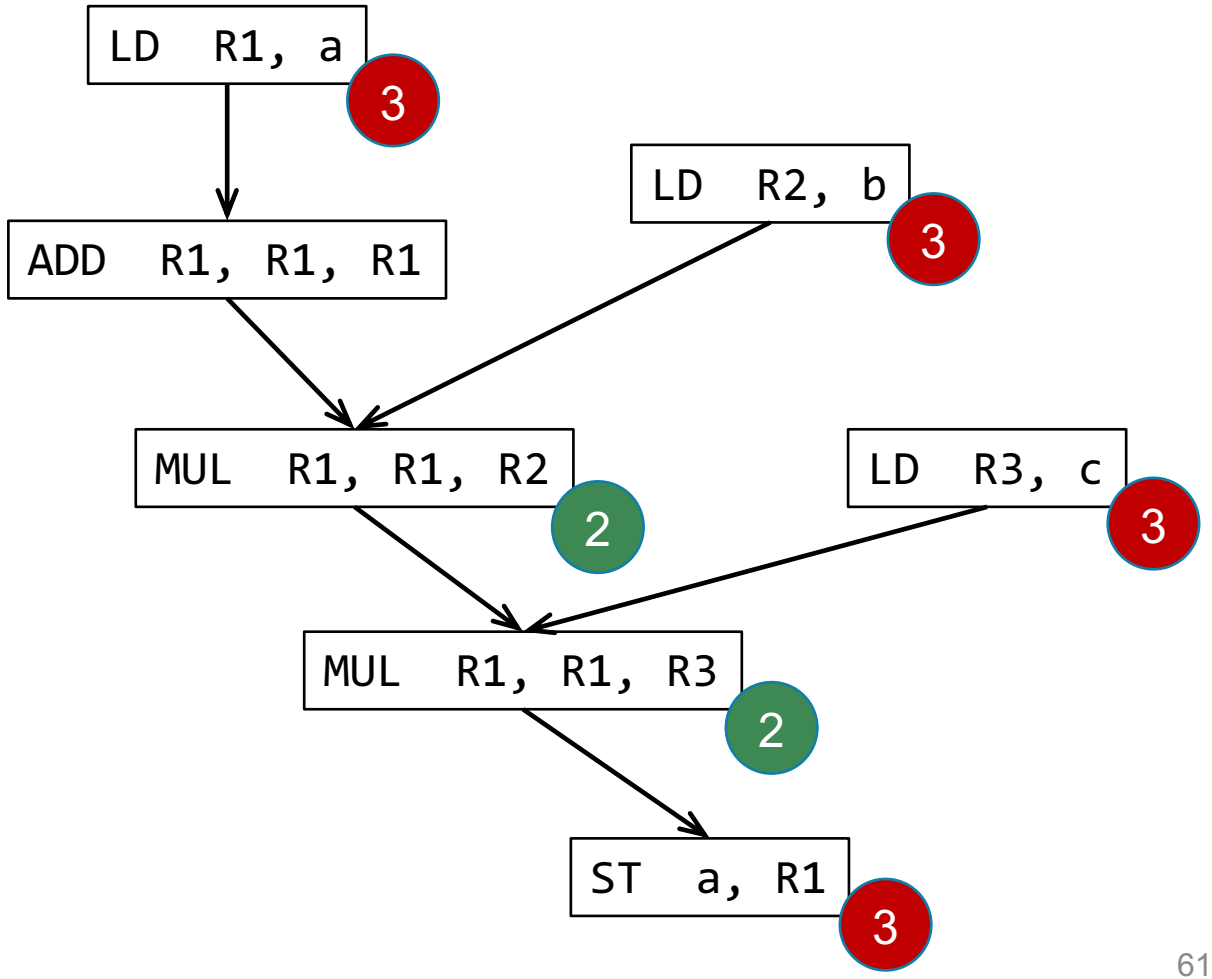
LD	R1,	a	
ADD	R1,	R1,	R1
LD	R2,	b	
MUL	R1,	R1,	R2
LD	R2,	c	
MUL	R1,	R1,	R2
ST	a,	R1	



In-Block Scheduling

- True/Fake Dependence
- Dependence Graph
- List Scheduling

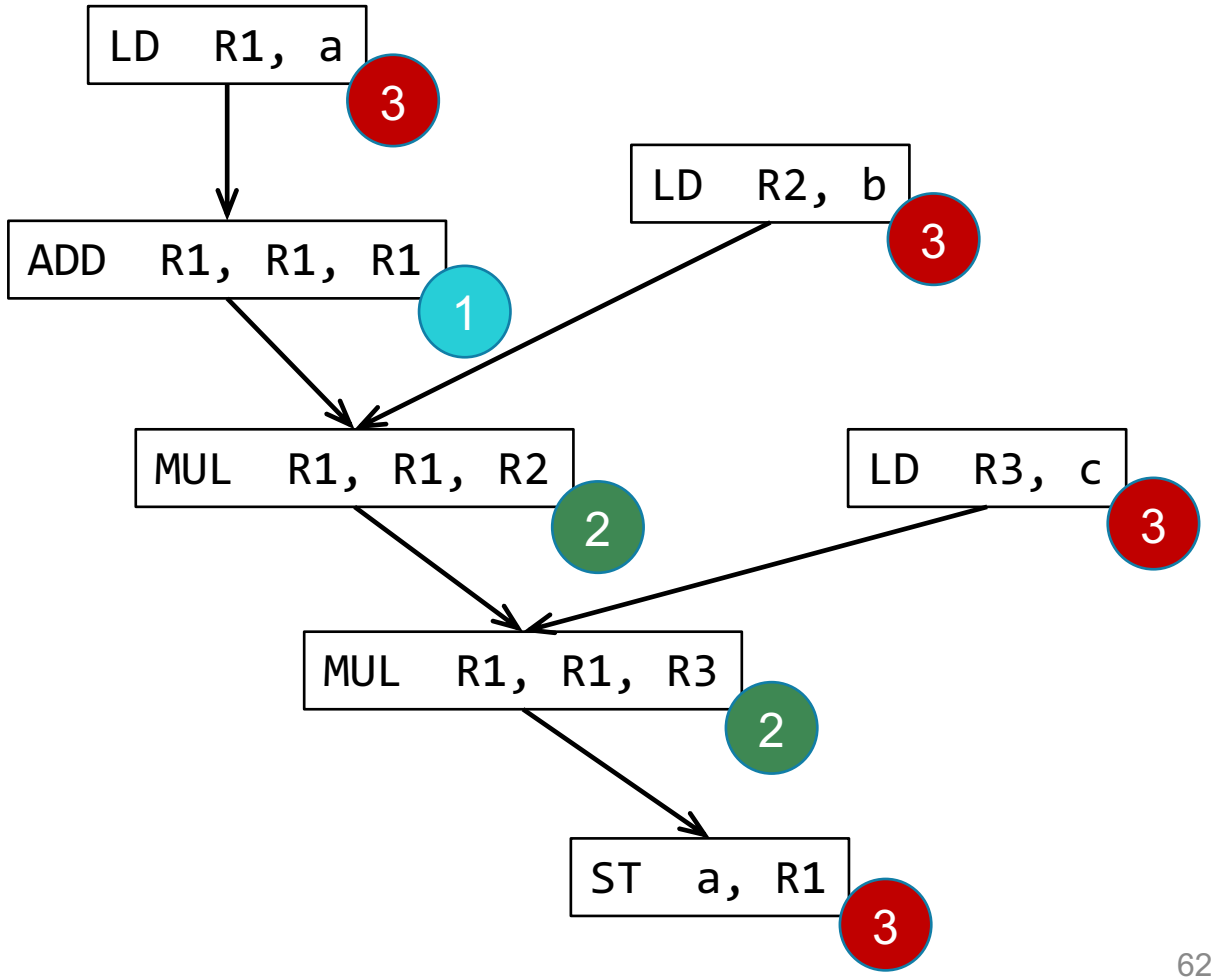
```
LD   R1,  a
ADD  R1,  R1,  R1
LD   R2,  b
MUL  R1,  R1,  R2
LD   R2,  c
MUL  R1,  R1,  R2
ST   a,   R1
```



In-Block Scheduling

- True/Fake Dependence
- Dependence Graph
- List Scheduling

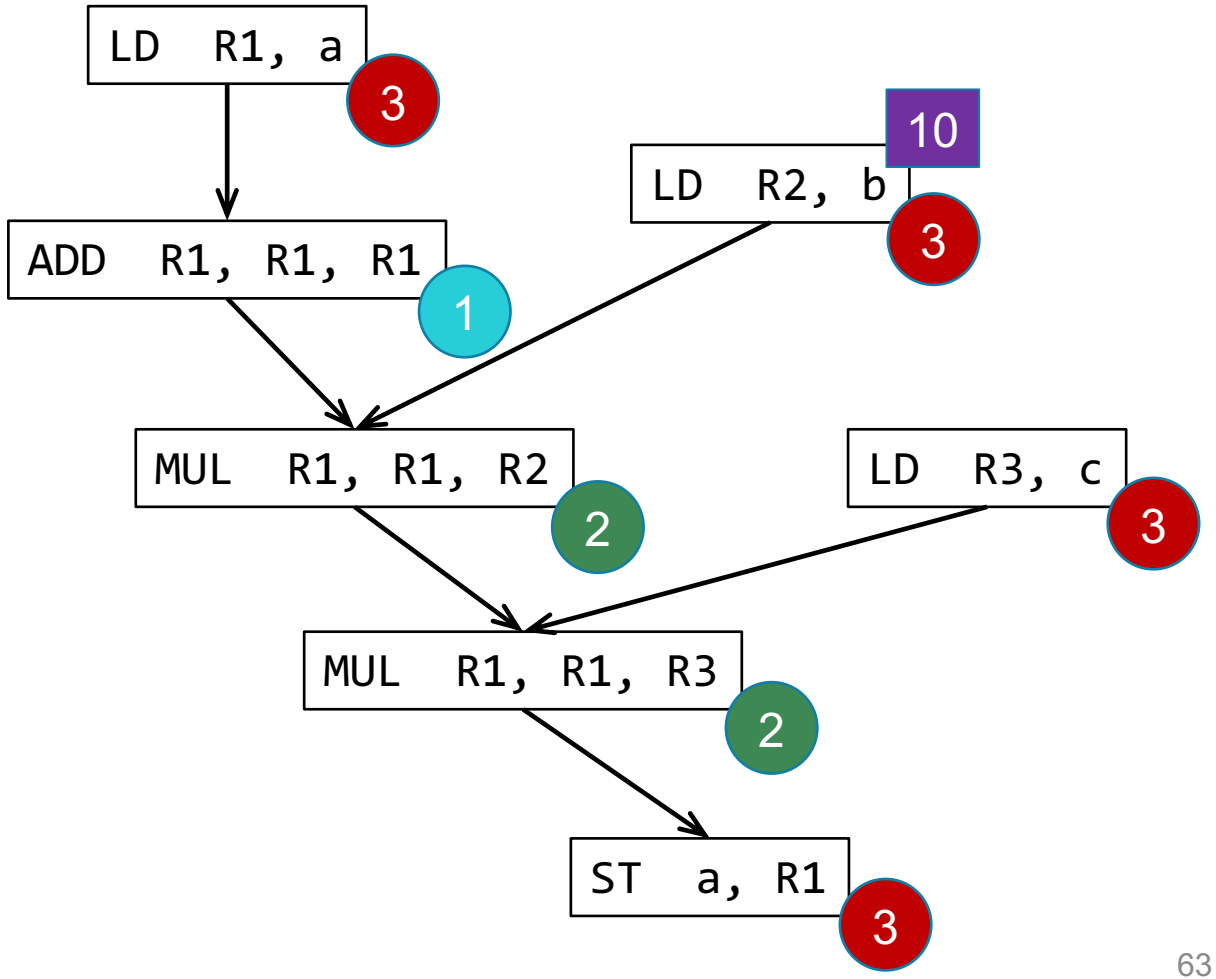
```
LD   R1, a
ADD  R1, R1, R1
LD   R2, b
MUL  R1, R1, R2
LD   R2, c
MUL  R1, R1, R2
ST   a, R1
```



In-Block Scheduling

- True/Fake Dependence
- Dependence Graph
- List Scheduling

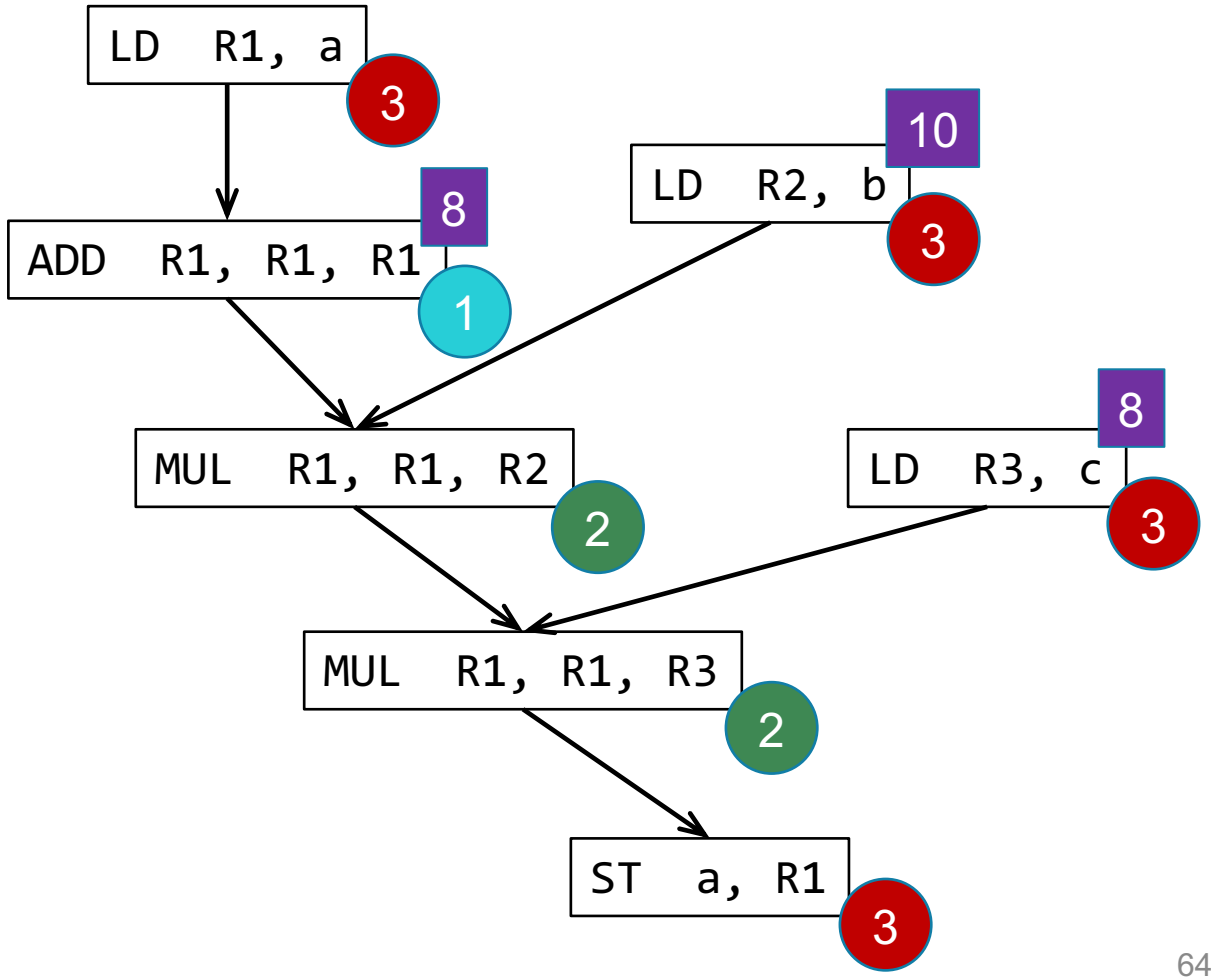
```
LD   R1,  a
ADD  R1,  R1,  R1
LD   R2,  b
MUL  R1,  R1,  R2
LD   R2,  c
MUL  R1,  R1,  R2
ST   a,   R1
```



In-Block Scheduling

- True/Fake Dependence
- Dependence Graph
- List Scheduling

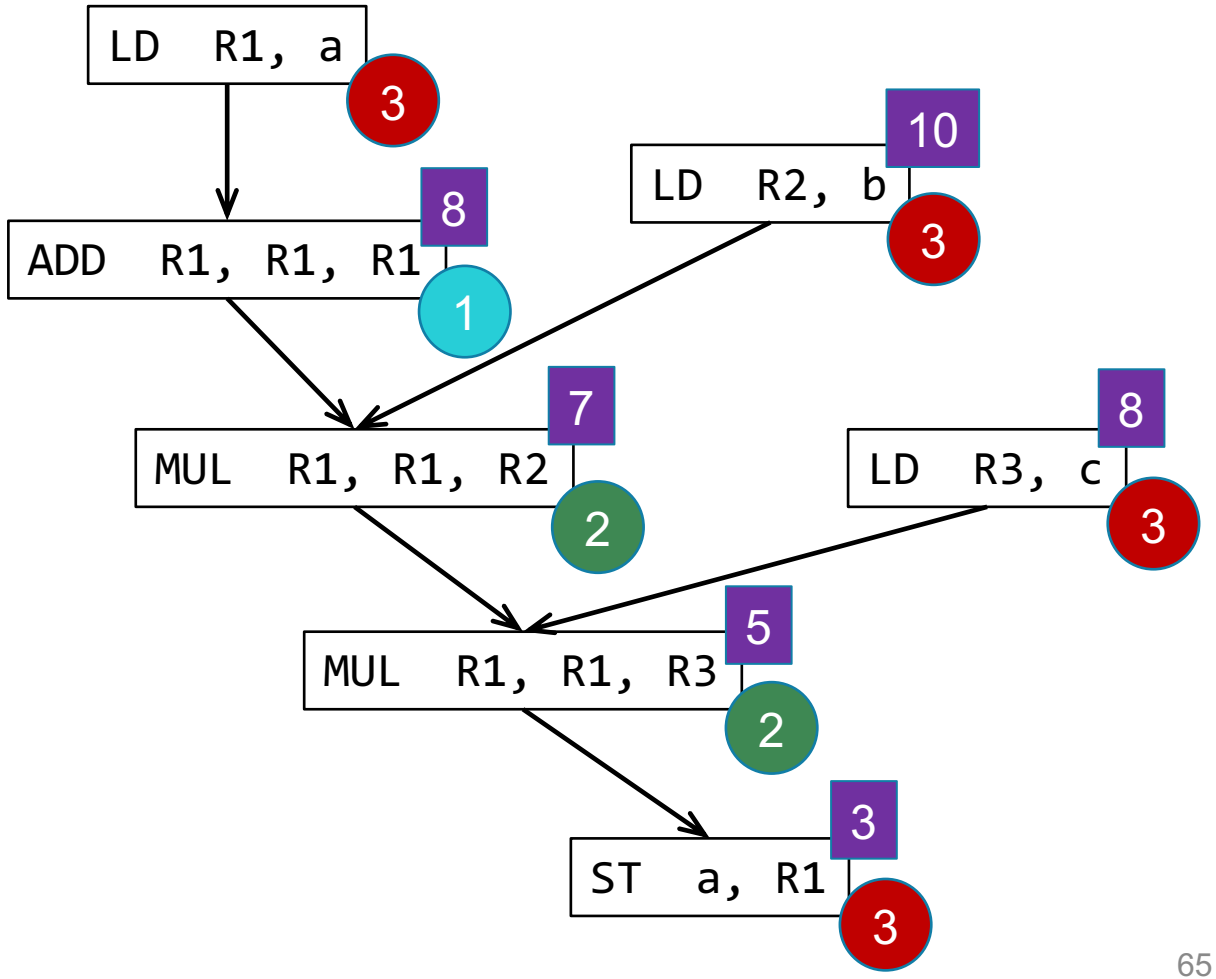
```
LD   R1,  a
ADD  R1,  R1,  R1
LD   R2,  b
MUL  R1,  R1,  R2
LD   R2,  c
MUL  R1,  R1,  R2
ST   a,   R1
```



In-Block Scheduling

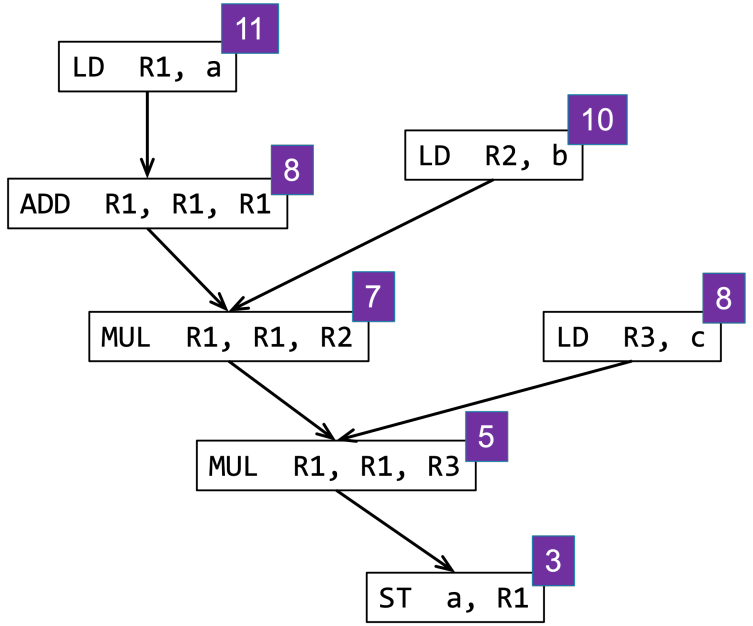
- True/Fake Dependence
- Dependence Graph
- List Scheduling

```
LD   R1,  a
ADD  R1,  R1,  R1
LD   R2,  b
MUL  R1,  R1,  R2
LD   R2,  c
MUL  R1,  R1,  R2
ST   a,   R1
```



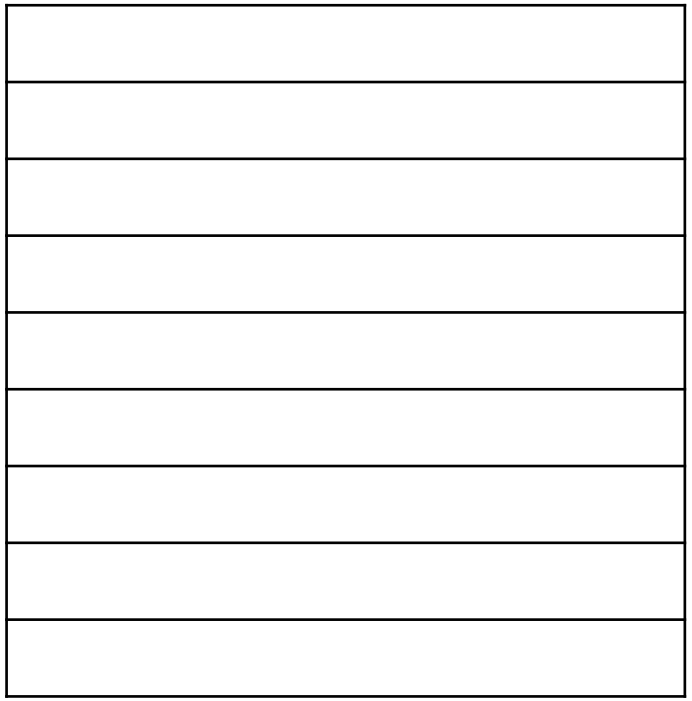
List Scheduling

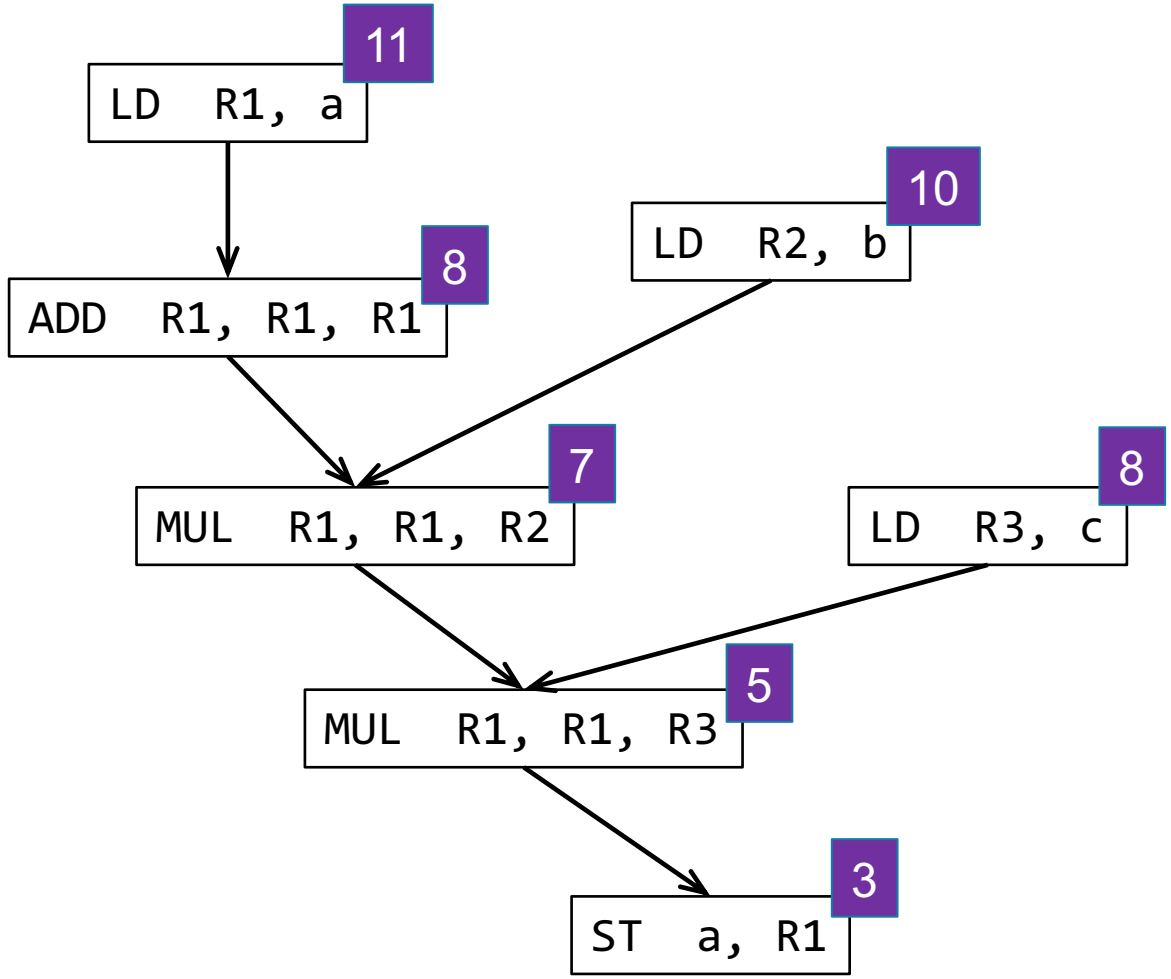
- Check each clock cycle
- Schedule instructions when they are ready
- When multi instructions can be scheduled, check their **priority**
  - The longest latency path (max depth)
  - Using the most resources
  - …



List Scheduling

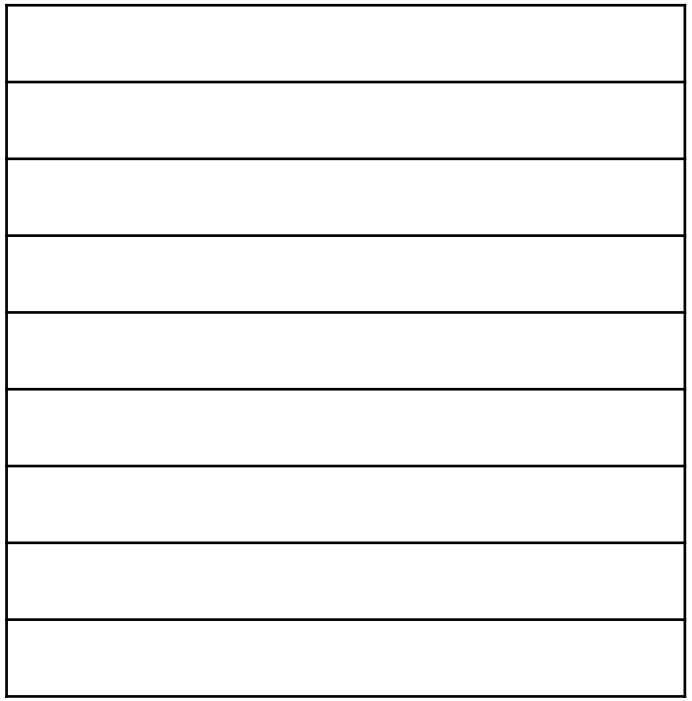
- Cycle = 1

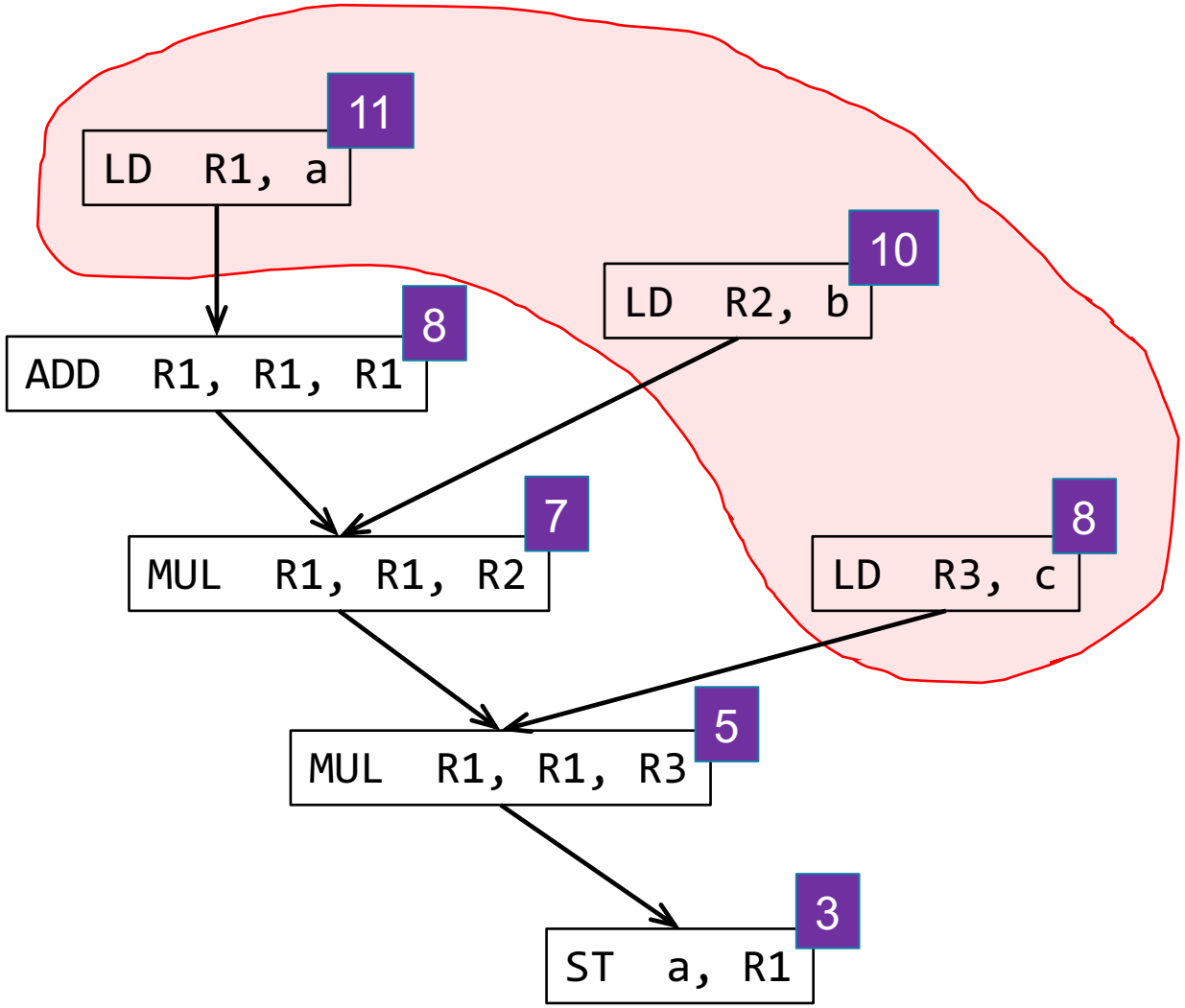




List Scheduling

- Cycle = 1

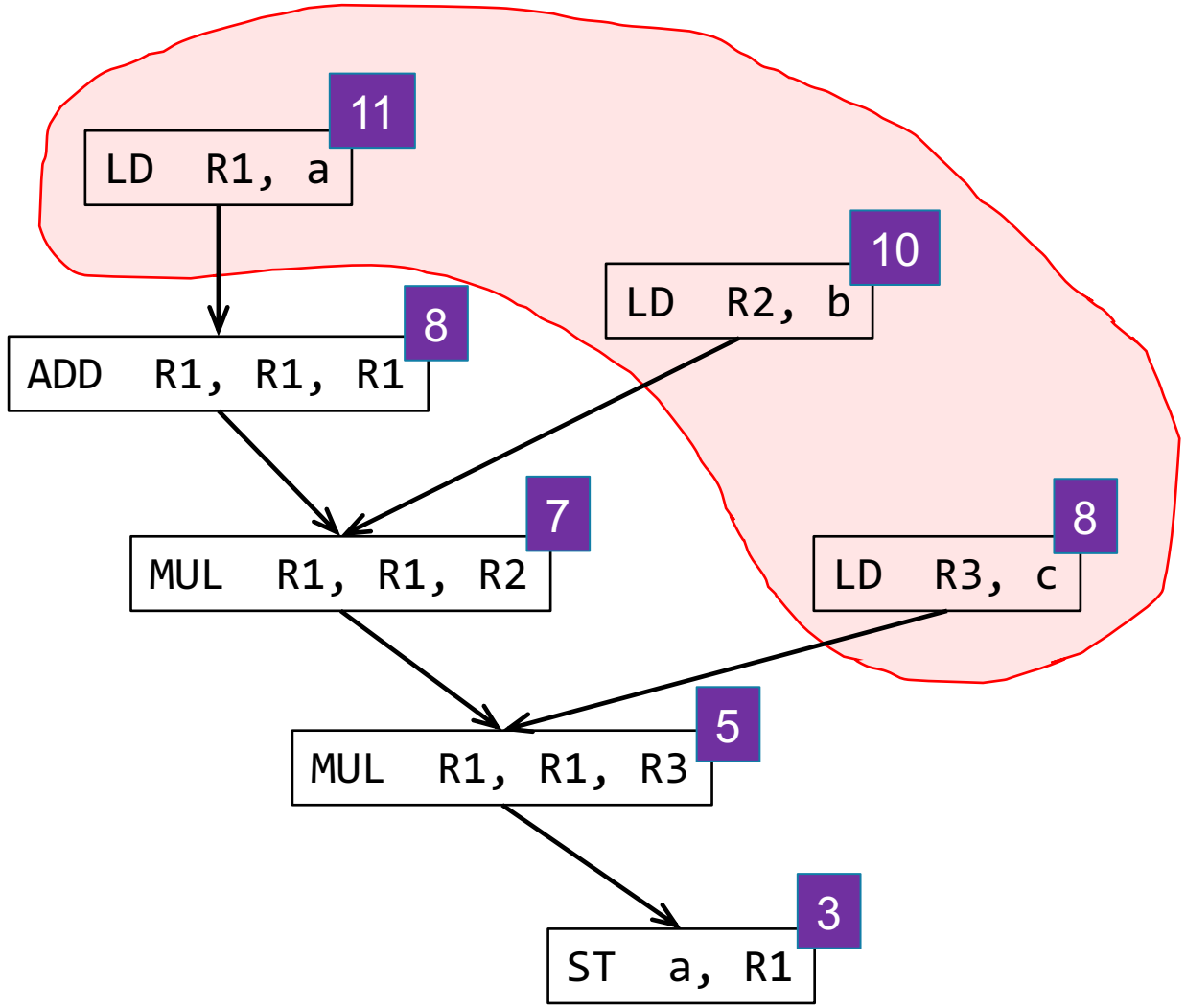




List Scheduling

- Cycle = 1

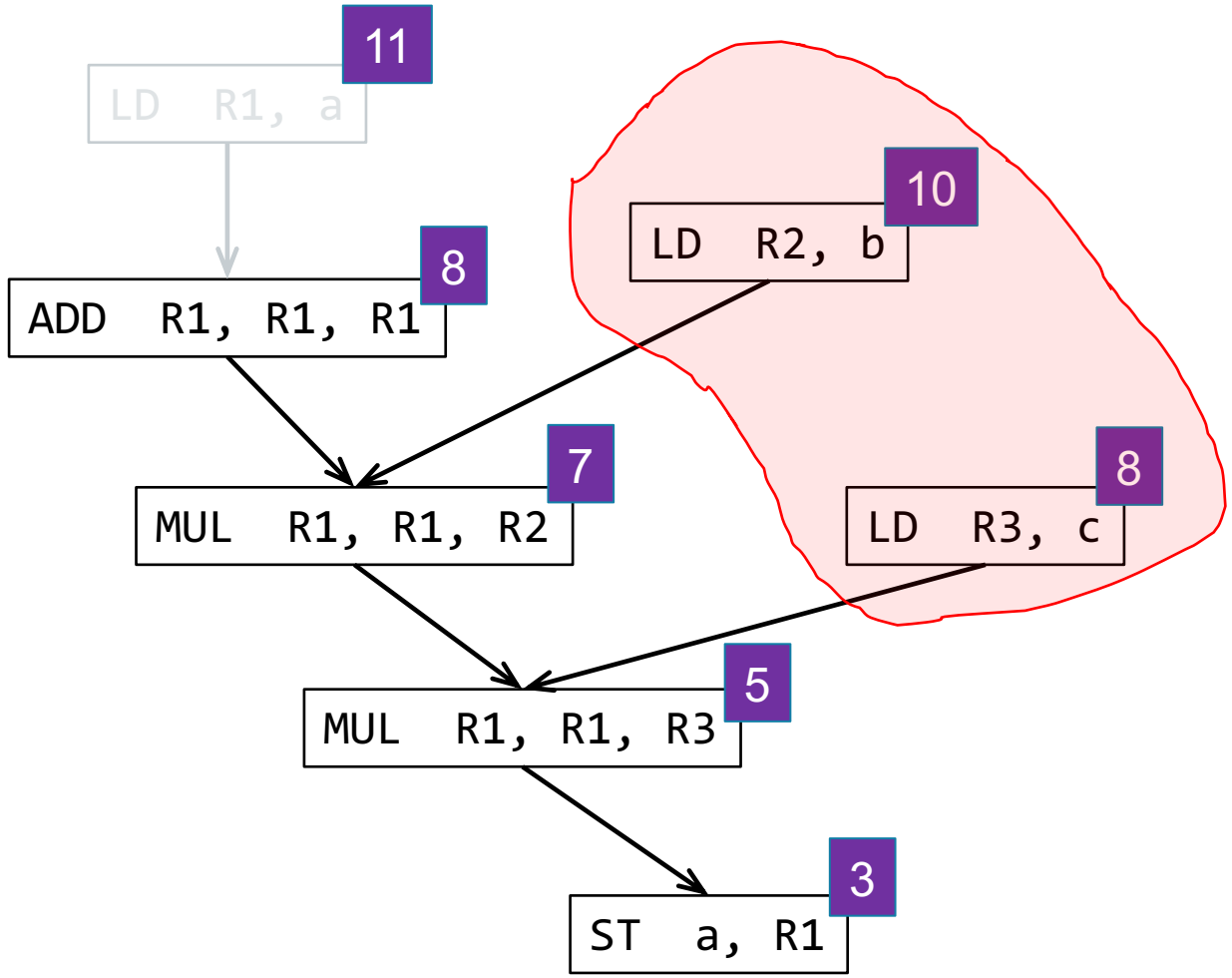
LD R1, a



List Scheduling

- Cycle = 2

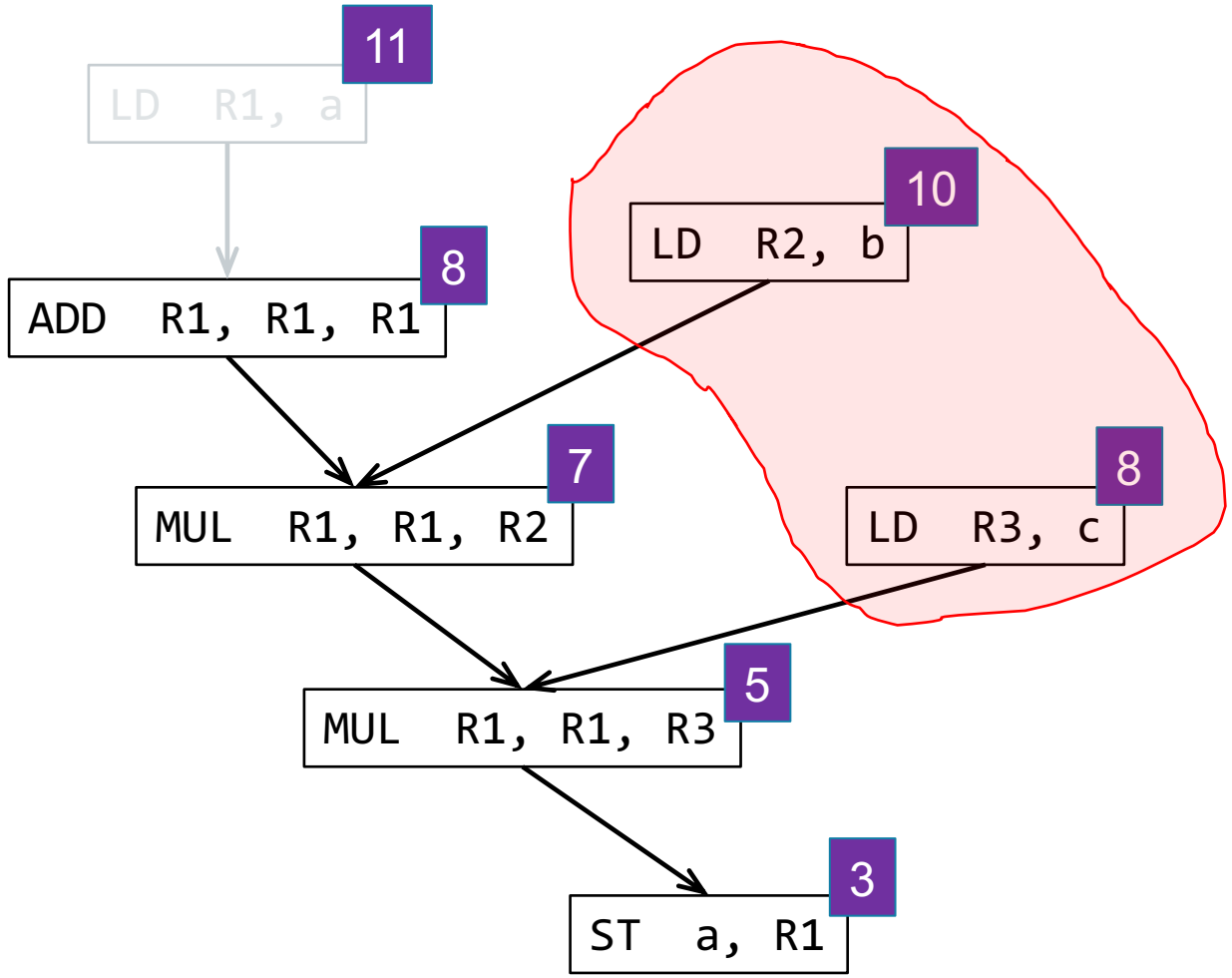
LD R1, a



List Scheduling

- Cycle = 2

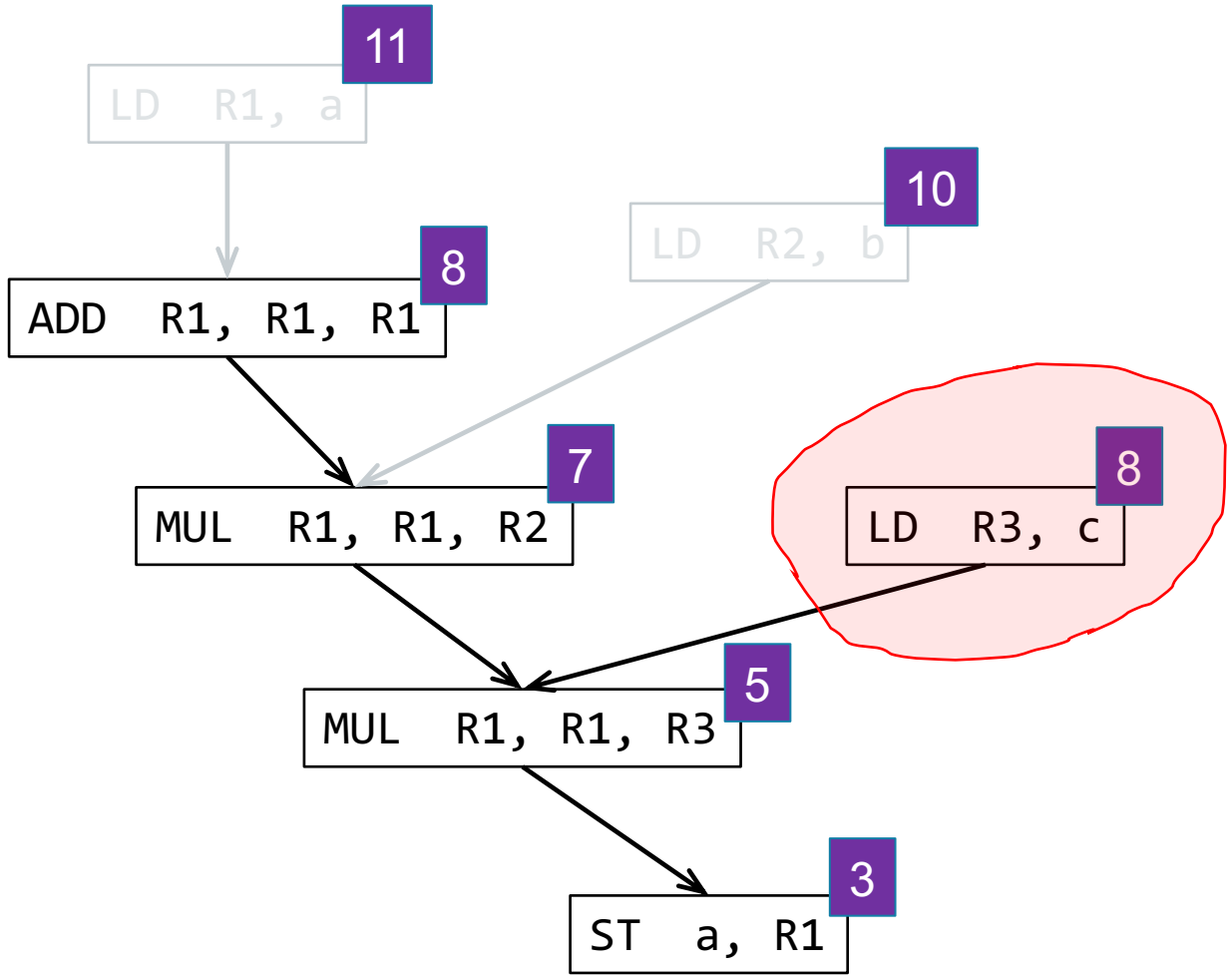
LD R1, a
LD R2, b



List Scheduling

- Cycle = 3

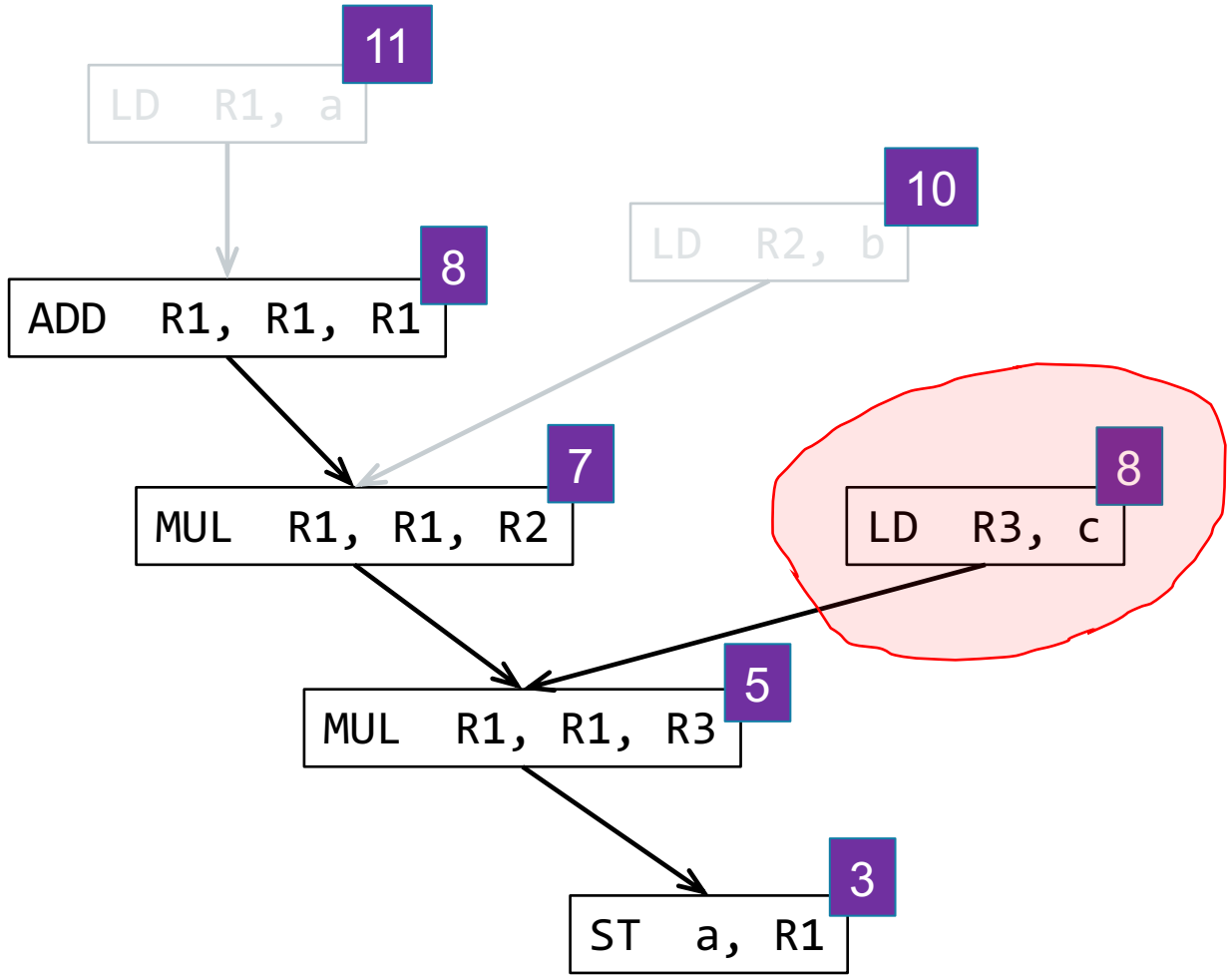
LD R1, a
LD R2, b



List Scheduling

- Cycle = 3

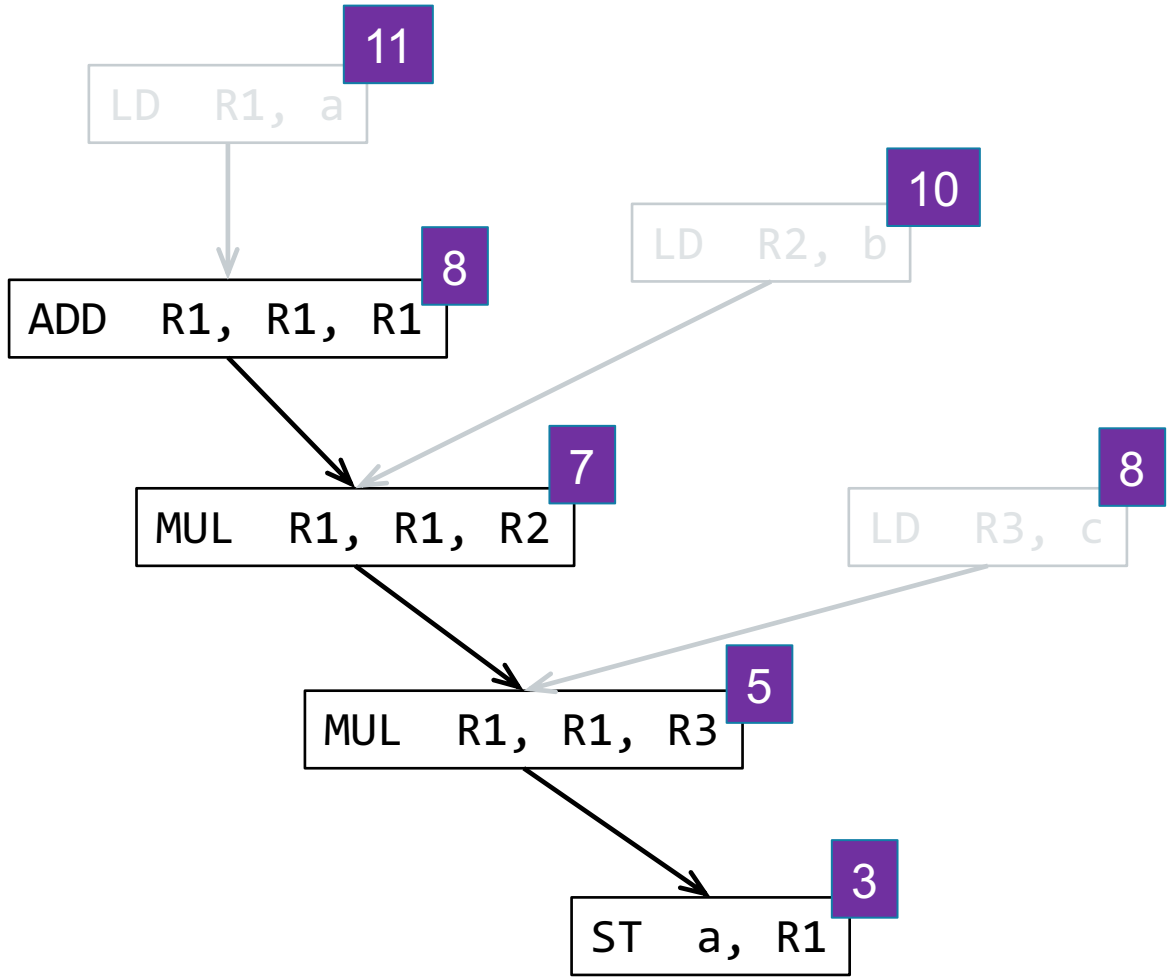
LD R1, a
LD R2, b
LD R3, c



List Scheduling

- Cycle = 4

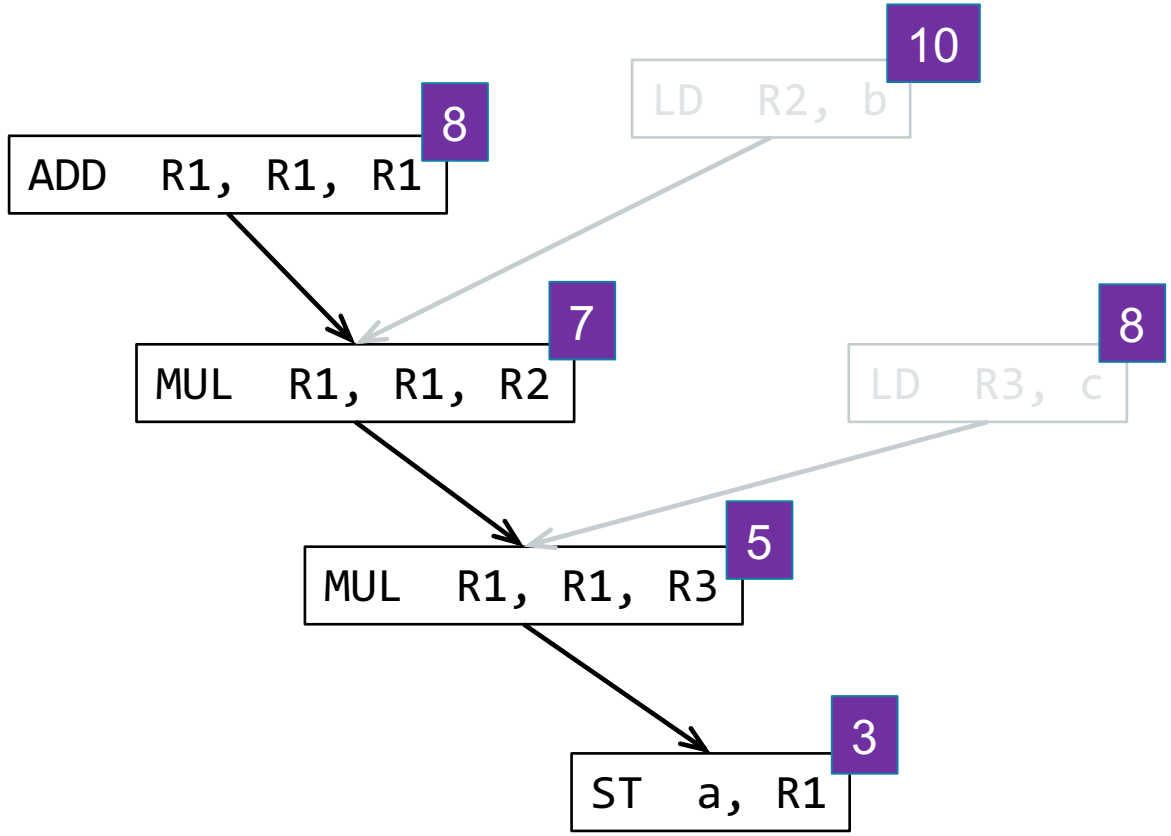
LD R1, a
LD R2, b
LD R3, c



List Scheduling

- Cycle = 4

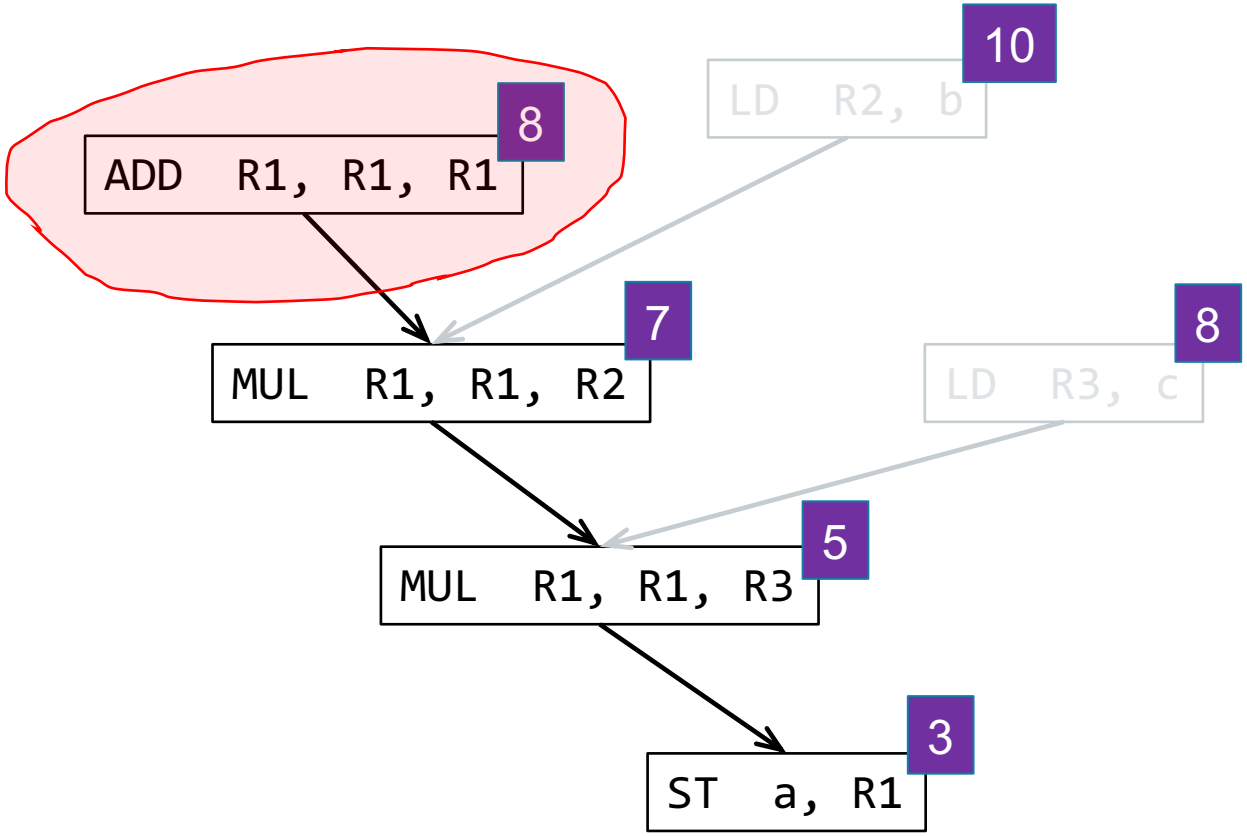
LD R1, a
LD R2, b
LD R3, c



List Scheduling

- Cycle = 4

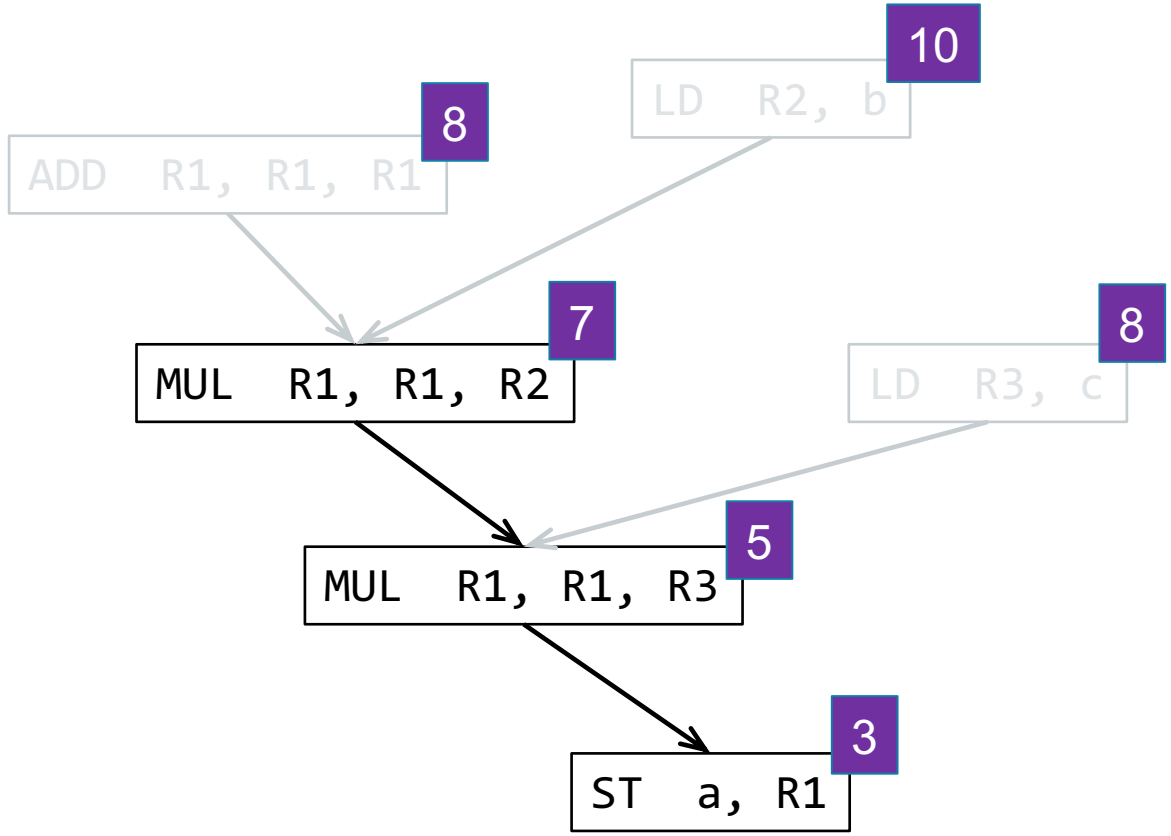
LD R1, a
LD R2, b
LD R3, c



List Scheduling

- Cycle = 4

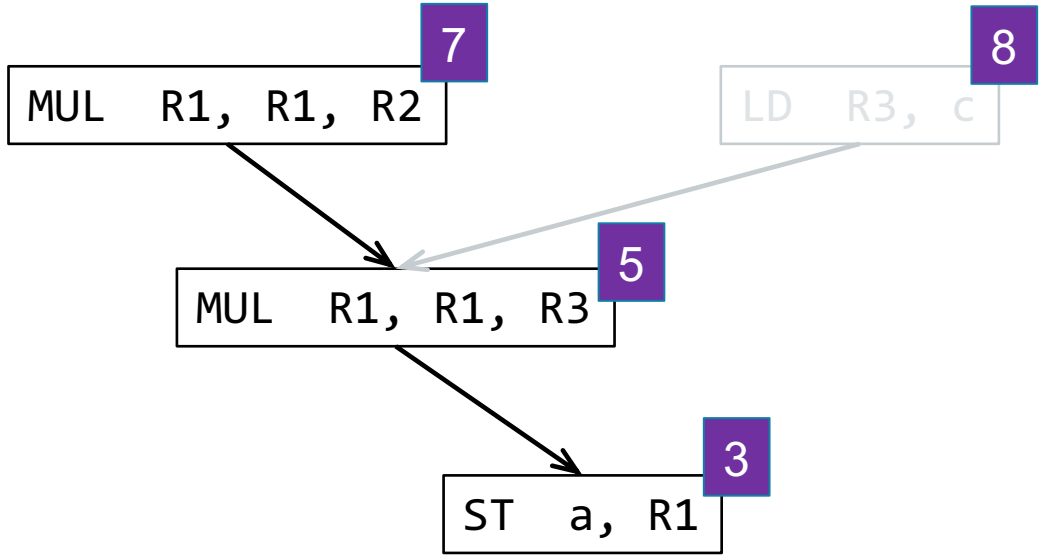
LD	R1,	a	
LD	R2,	b	
LD	R3,	c	
ADD	R1,	R1,	R1



List Scheduling

- Cycle = 5

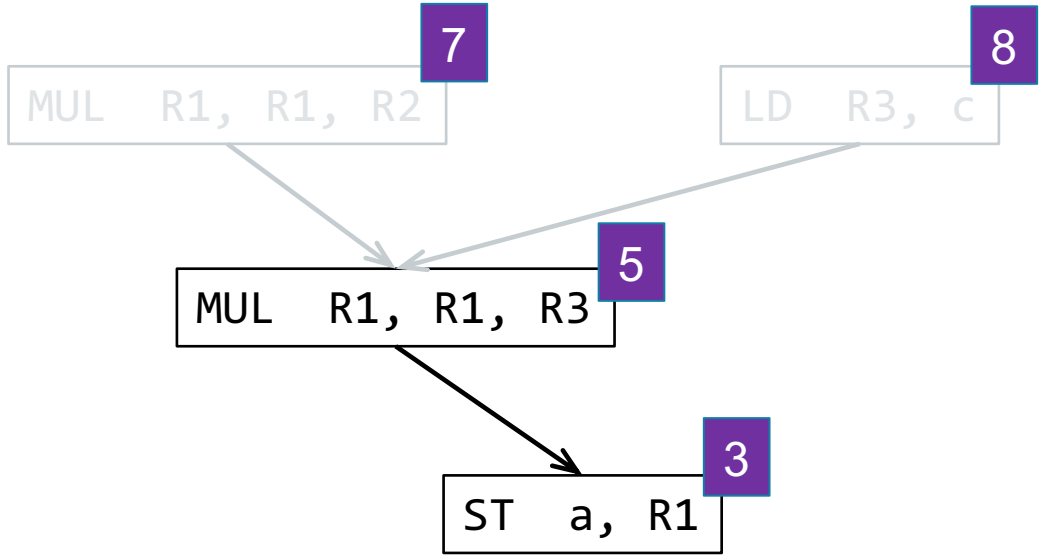
LD	R1,	a	
LD	R2,	b	
LD	R3,	c	
ADD	R1,	R1,	R1



List Scheduling

- Cycle = 5

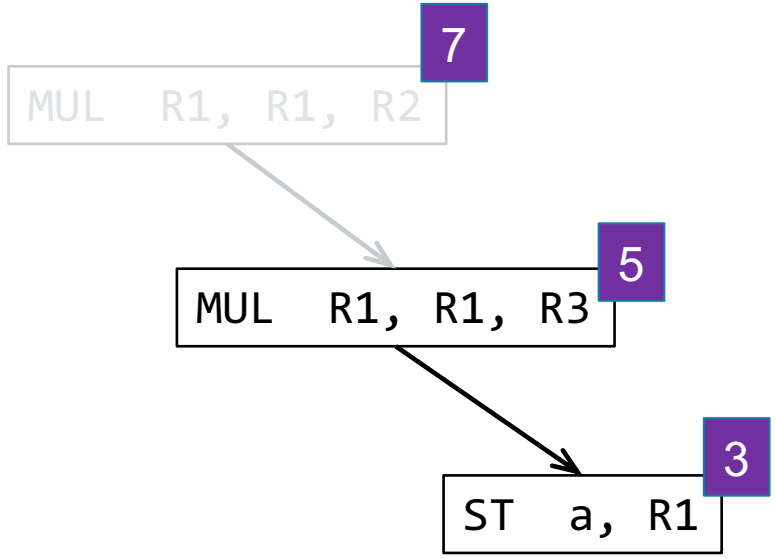
LD	R1,	a	
LD	R2,	b	
LD	R3,	c	
ADD	R1,	R1,	R1
MUL	R1,	R1,	R2



List Scheduling

- Cycle = 6

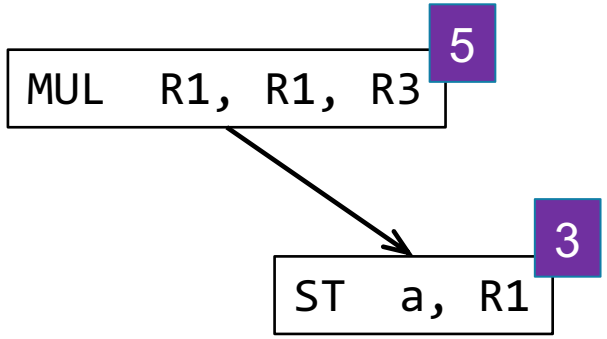
LD R1, a
LD R2, b
LD R3, c
ADD R1, R1, R1
MUL R1, R1, R2
NOP (no operation)



List Scheduling

- Cycle = 7

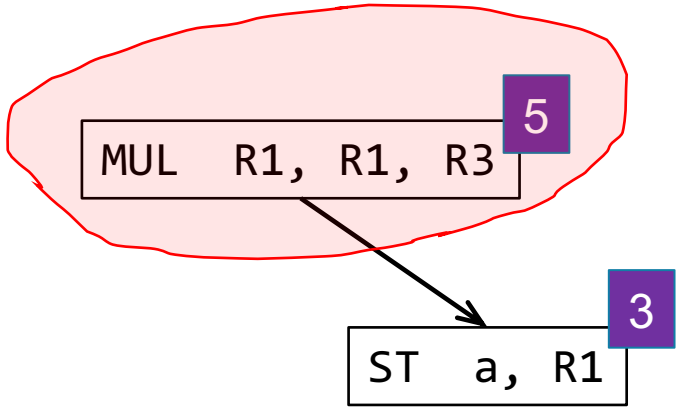
LD R1, a
LD R2, b
LD R3, c
ADD R1, R1, R1
MUL R1, R1, R2
NOP (no operation)



List Scheduling

- Cycle = 7

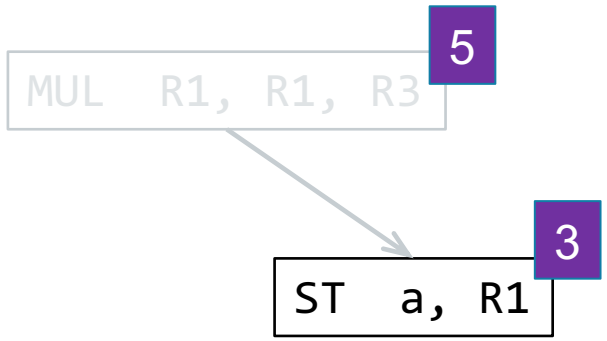
LD R1, a
LD R2, b
LD R3, c
ADD R1, R1, R1
MUL R1, R1, R2
NOP (no operation)



List Scheduling

- Cycle = 7, 8

LD R1, a
LD R2, b
LD R3, c
ADD R1, R1, R1
MUL R1, R1, R2
NOP (no operation)
MUL R1, R1, R3
NOP (no operation)



List Scheduling

- Cycle = 9, 10, 11

```
LD   R1,  a
LD   R2,  b
LD   R3,  c
ADD  R1,  R1,  R1
MUL  R1,  R1,  R2
NOP (no operation)
MUL  R1,  R1,  R3
NOP (no operation)
ST   a,   R1
```

3

```
ST  a, R1
```

PART IV: Exercises

Exercise - I

- Consider how we compile the following assignment to machine code: $e = (a - 1) * \left(\frac{(b+c)}{3} + d\right)$
  - LD R, var; ST var, R
  - OP R1, R2, R3, where OP can be ADD, SUB, MUL, DIV, …, R2 and R3 can be immediate number
- **Q1: What's the purpose of register allocation? (4')**
- **Q2: What's the minimum registers required if spilling is not allowed (2')**
- **Q3: Write the machine code (4')**

Exercise - II

- Consider the following code, construct an interference graph

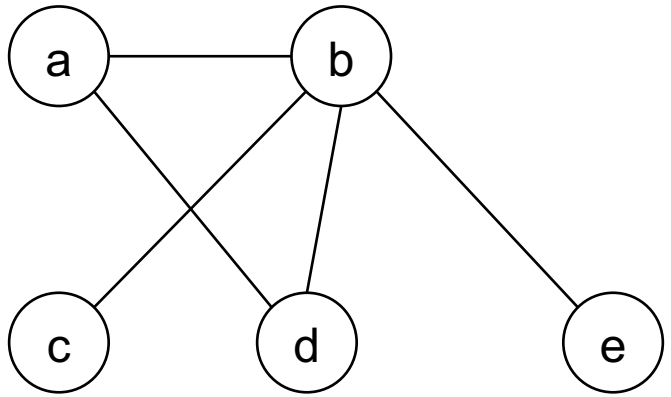
```
1   a = read();
2   b = read();
3   if (a > b) {
4      c = read();
5      e = c - b;
6   } else {
7      d = b;
8      e = a + d;
9   }
10  print(b);
11  print(e);
```

- How many colors are required to avoid spilling registers

Exercise - II

- Consider the following code, construct an interference graph

```
1   a = read();
2   b = read();
3   if (a > b) {
4      c = read();
5      e = c - b;
6   } else {
7      d = b;
8      e = a + d;
9   }
10  print(b);
11  print(e);
```

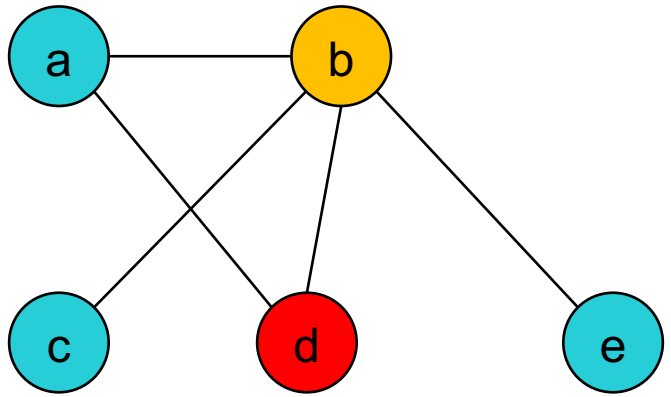


- How many colors are required to avoid spilling registers

Exercise - II

- Consider the following code, construct an interference graph

```
1   a = read();
2   b = read();
3   if (a > b) {
4      c = read();
5      e = c - b;
6   } else {
7      d = b;
8      e = a + d;
9   }
10  print(b);
11  print(e);
```



- How many colors are required to avoid spilling registers

Exercise - III

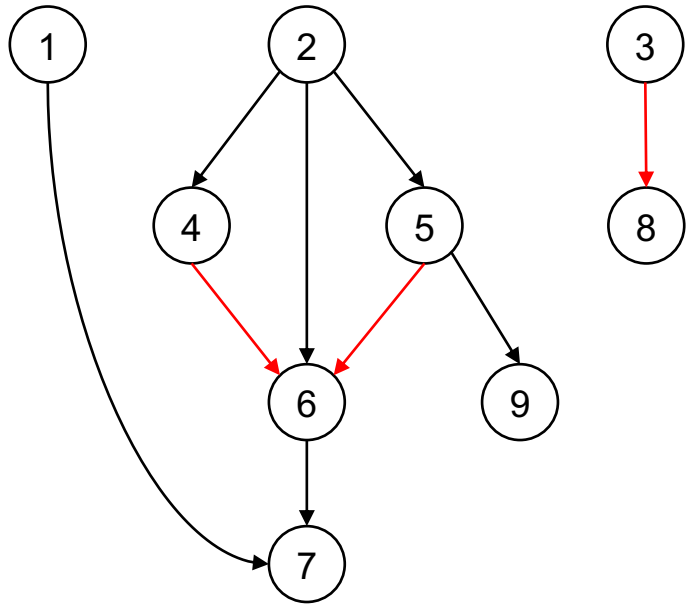
- Schedule the following code, assuming LD/ST needs 3 cycles and add needs 1 cycle (**SP is a special register and cannot be renamed**)

```
1.  ADD   R2,    R1,     #1
2.  ADD   SP,    12,     SP
3.  ST    a,     R0
4.  LD    R3,    -4(SP)
5.  LD    R4,    -8(SP)
6.  ADD   SP,    SP,     #8
7.  ST    0(SP), R2
8.  LD    R5,    a
9.  ADD   R4,    R4,     #1
```

Exercise - III

- Schedule the following code, assuming LD/ST needs 3 cycles and add needs 1 cycle (**SP is a special register and cannot be renamed**)

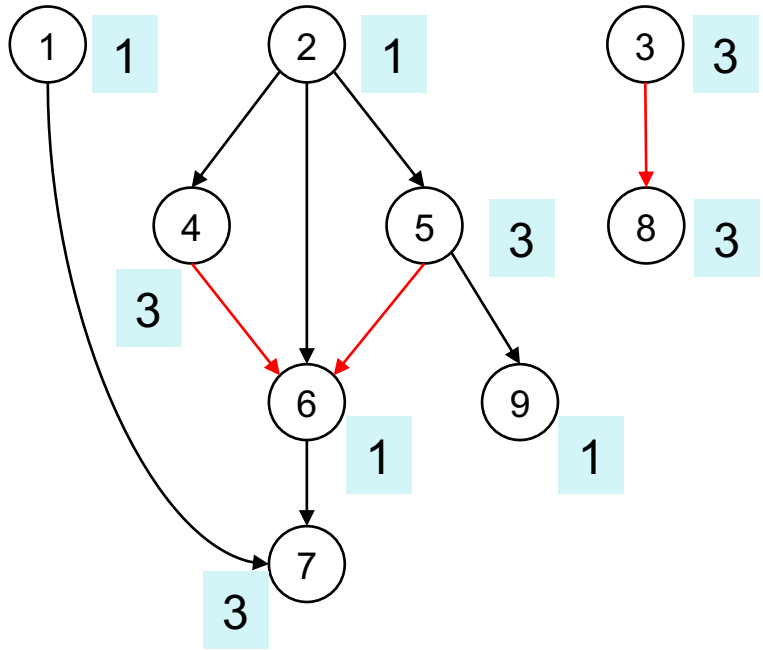
```
1.  ADD   R2,    R1,     #1
2.  ADD   SP,    12,     SP
3.  ST    a,     R0
4.  LD    R3,    -4(SP)
5.  LD    R4,    -8(SP)
6.  ADD   SP,    SP,     #8
7.  ST    0(SP), R2
8.  LD    R5,    a
9.  ADD   R4,    R4,     #1
```



Exercise - III

- Schedule the following code, assuming LD/ST needs 3 cycles and add needs 1 cycle (**SP is a special register and cannot be renamed**)

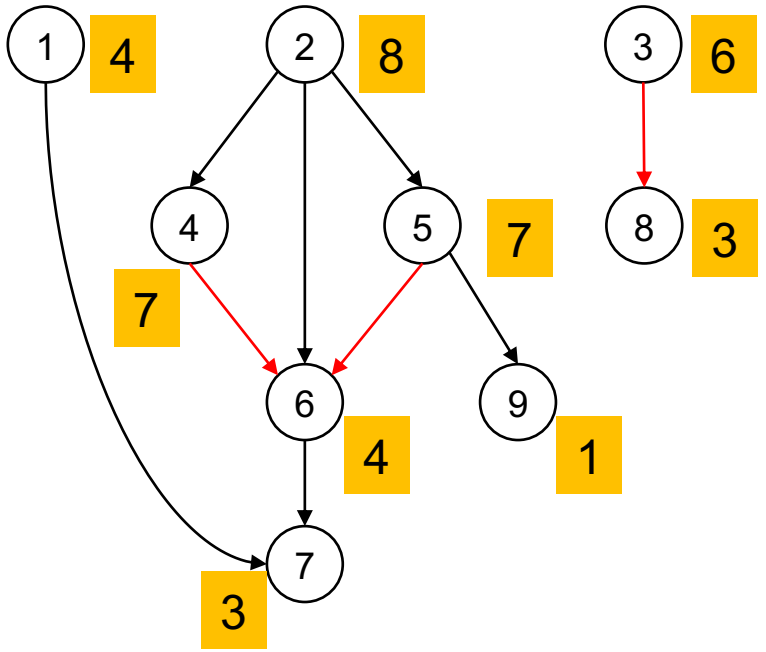
```
1.  ADD   R2,    R1,     #1
2.  ADD   SP,    12,     SP
3.  ST    a,     R0
4.  LD    R3,    -4(SP)
5.  LD    R4,    -8(SP)
6.  ADD   SP,    SP,     #8
7.  ST    0(SP), R2
8.  LD    R5,    a
9.  ADD   R4,    R4,     #1
```



Exercise - III

- Schedule the following code, assuming LD/ST needs 3 cycles and add needs 1 cycle (**SP is a special register and cannot be renamed**)

```
1.  ADD   R2,    R1,     #1
2.  ADD   SP,    12,     SP
3.  ST    a,     R0
4.  LD    R3,    -4(SP)
5.  LD    R4,    -8(SP)
6.  ADD   SP,    SP,     #8
7.  ST    0(SP), R2
8.  LD    R5,    a
9.  ADD   R4,    R4,     #1
```



THANKS!