

Lecture 15:

Pipelining and Parallelism

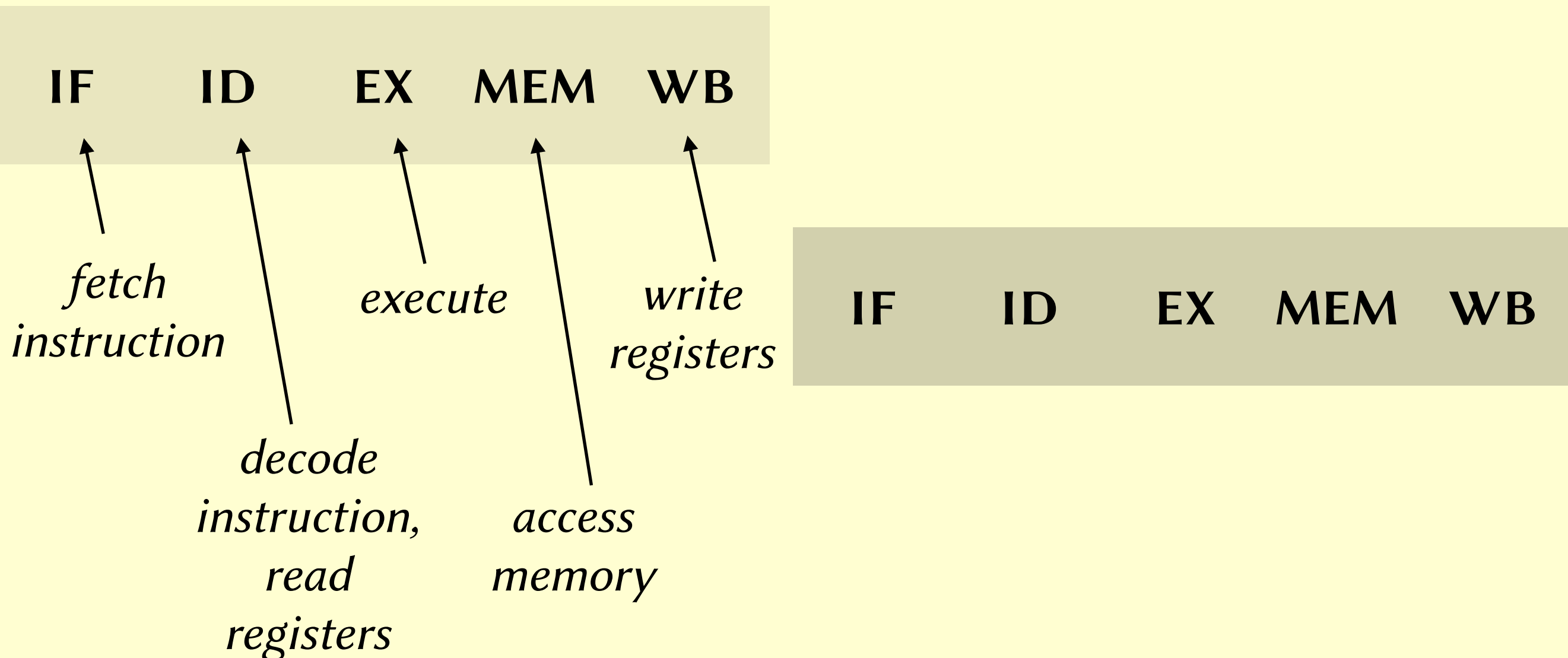
John Wickerson

Compilers

Lecture outline

- Compiler-provided parallelism
 - Exploiting pipelining in hardware
 - Pipelining in software
 - The polyhedral model of loops
- Programmer-provided parallelism

Pipelining



Pipelining

IF

ID

EX

MEM

WB

IF

ID

EX

MEM

Pipelining

`lw t1, 0(t0)`

IF

ID

EX

MEM

WB

`lw t2, 4(t0)`

IF

ID

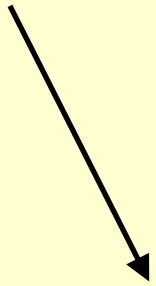
EX

MEM

WB

Pipelining

add t3,t1,t2



add t5,t3,t4

IF

ID

EX

MEM

WB

IF

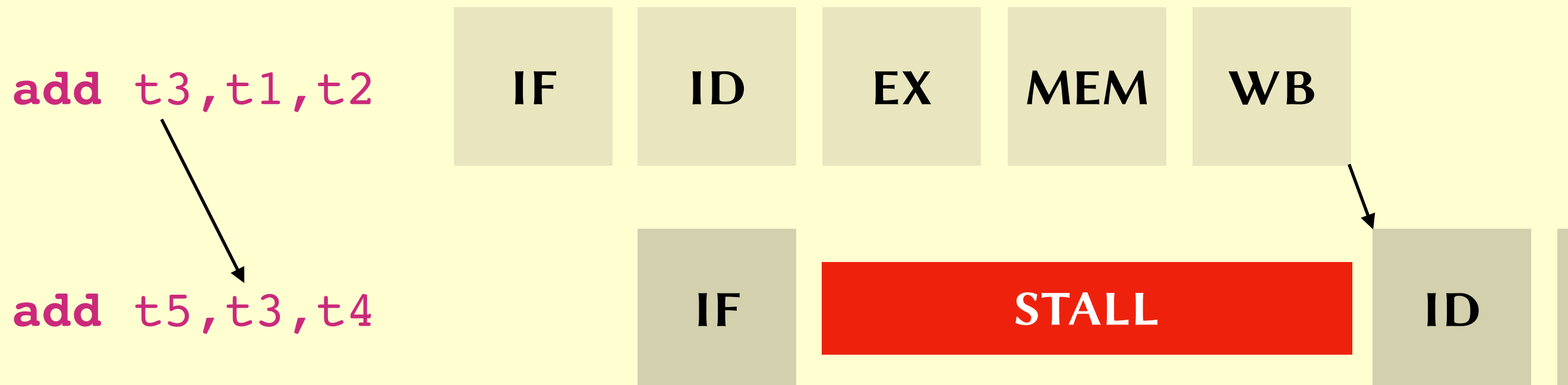
ID

EX

MEM

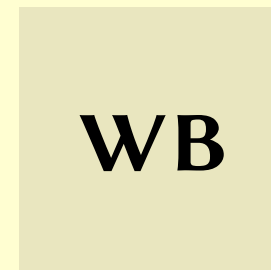
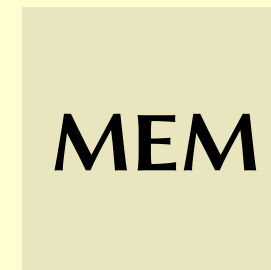
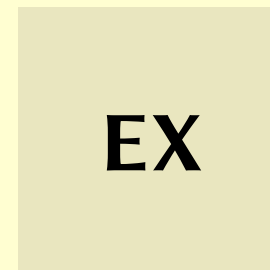
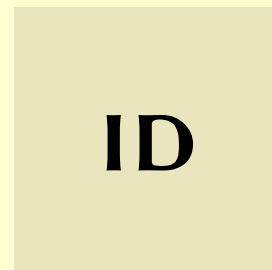
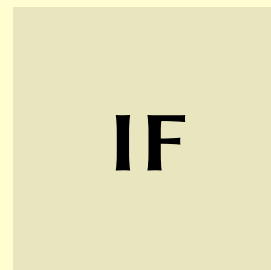
WB

Pipelining

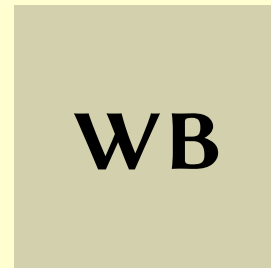
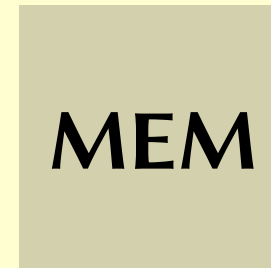
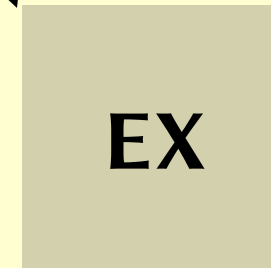
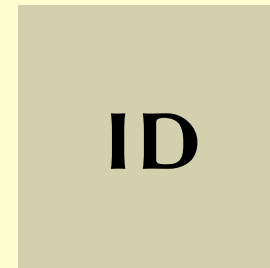
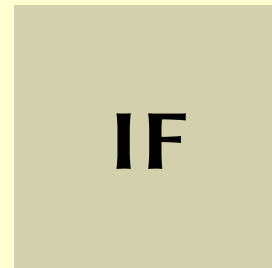


Pipelining

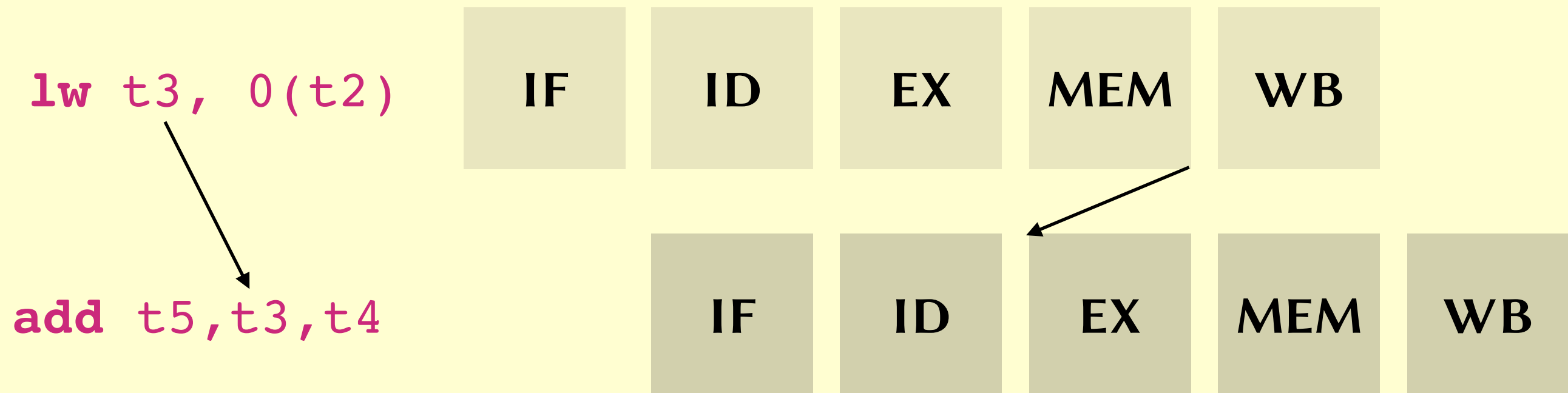
add t3,t1,t2



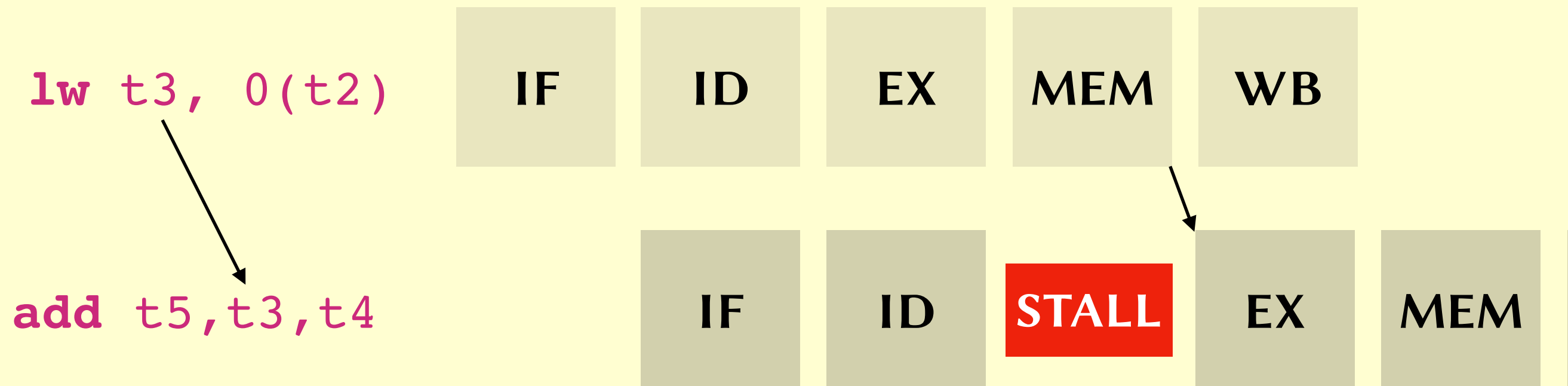
add t5,t3,t4



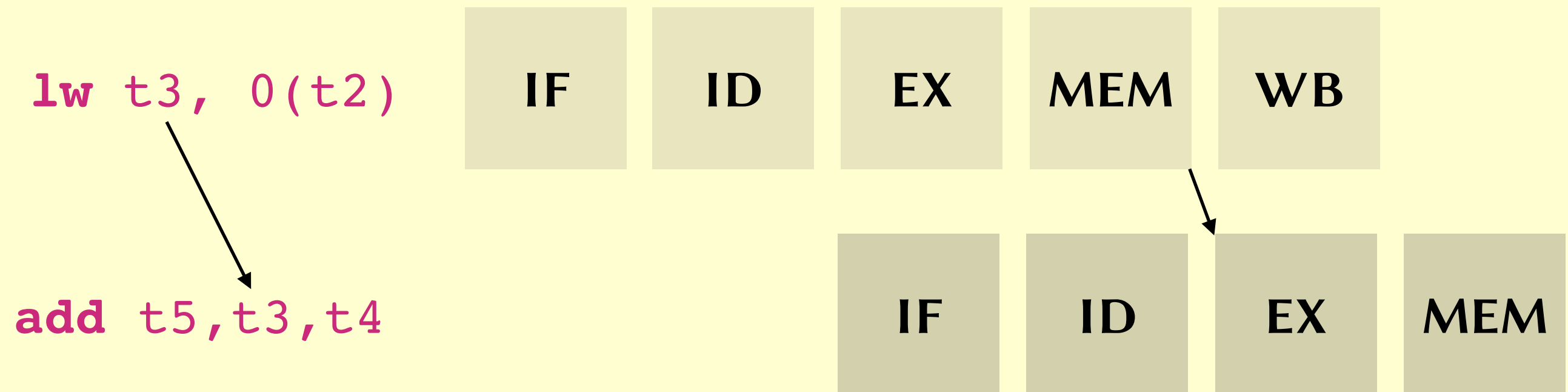
Pipelining



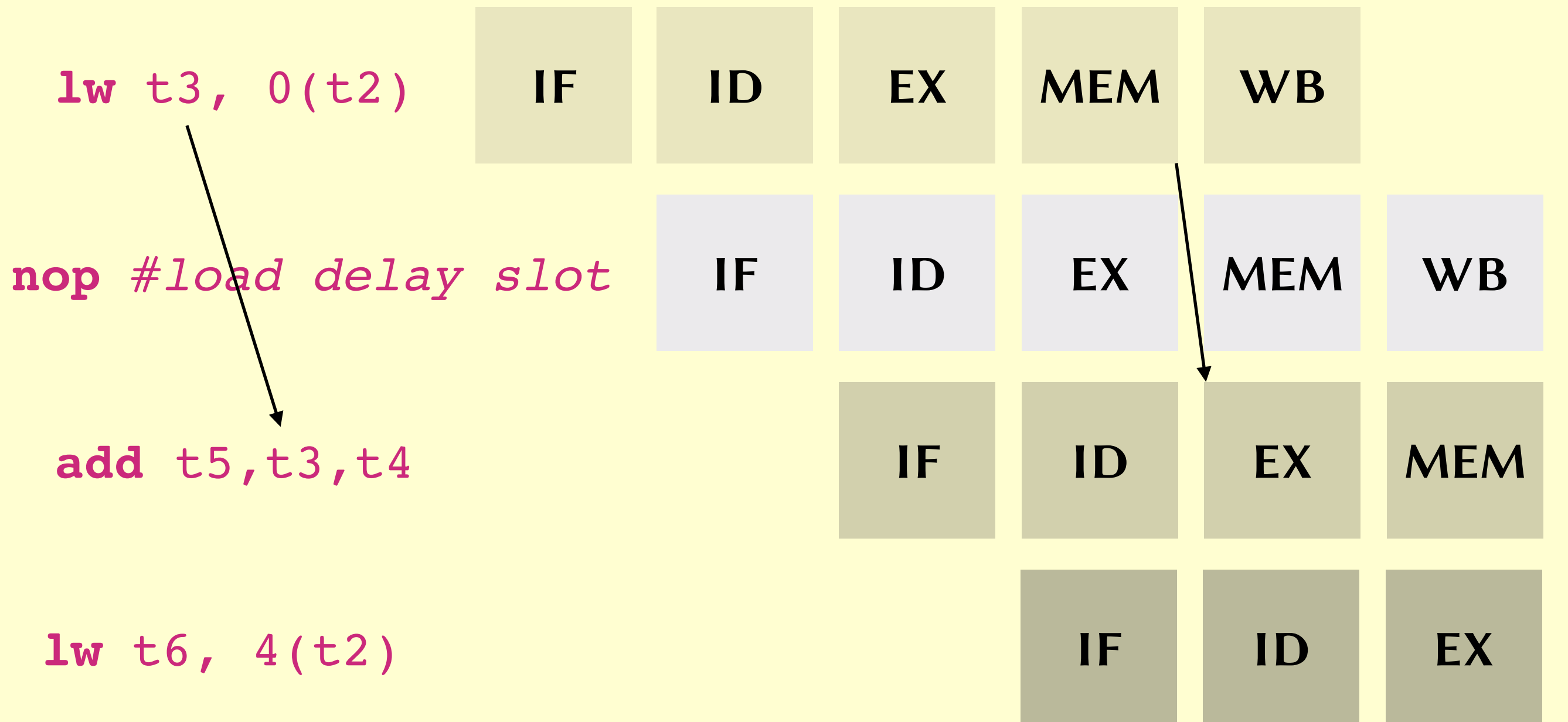
Pipelining



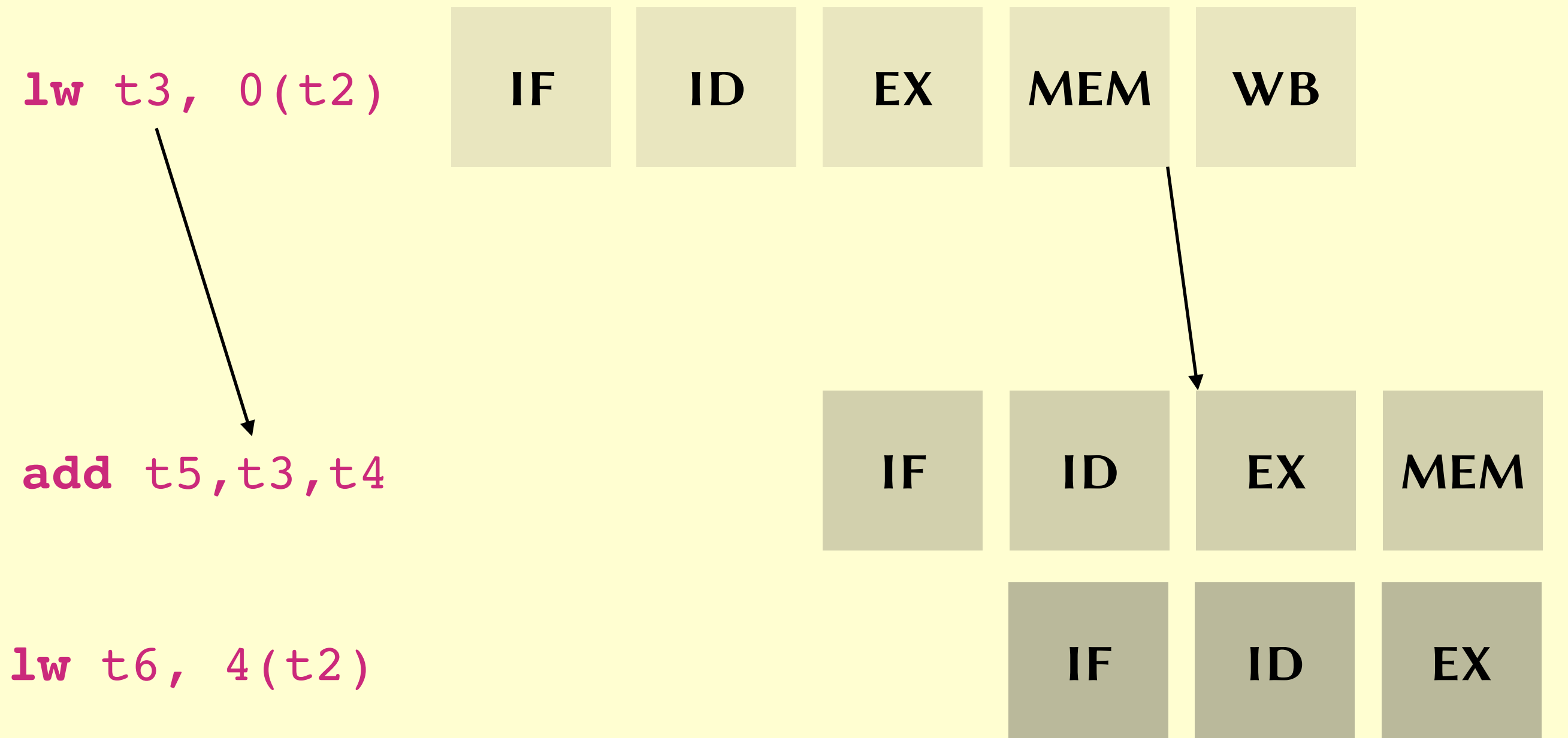
Pipelining



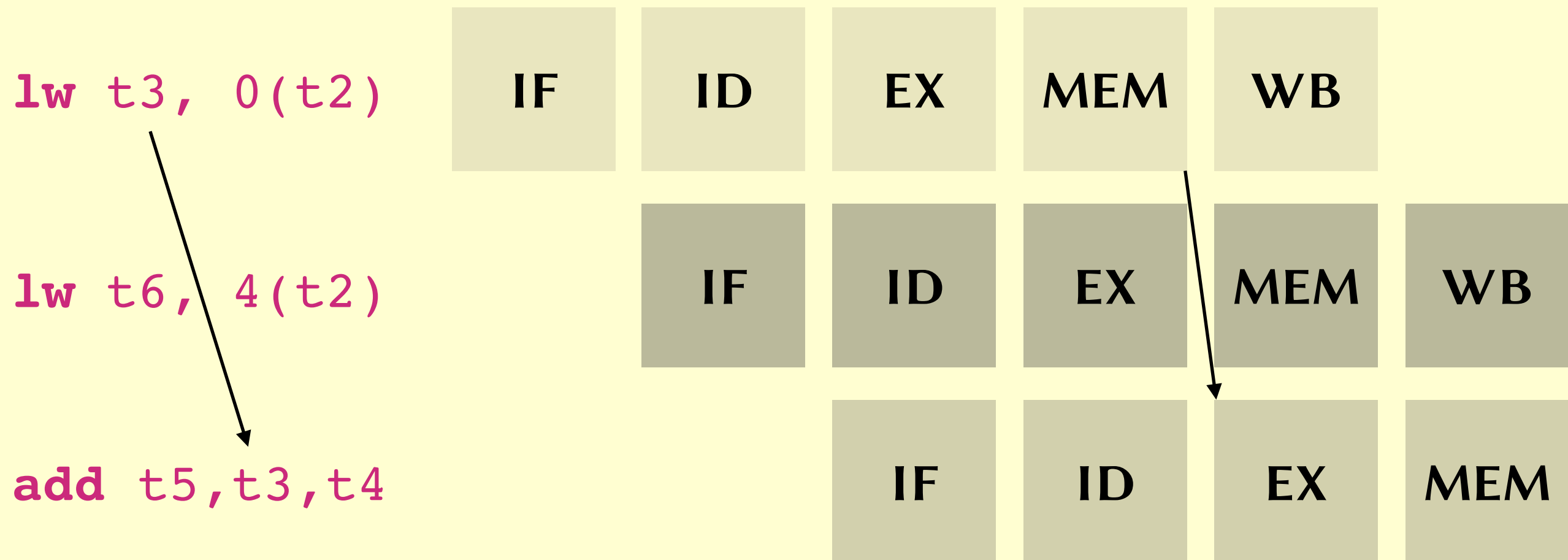
Pipelining



Pipelining



Pipelining



Registers vs. reordering

$(a - b) + c + (d - e)$

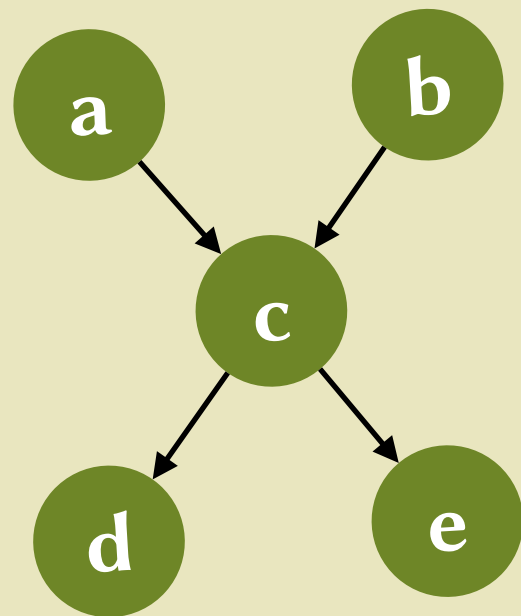
```
lw t0, a
lw t1, b
sub t0, t0, t1
lw t1, c
add t0, t0, t1
lw t1, d
lw t2, e
sub t1, t1, t2
add t0, t0, t1
```

The diagram illustrates the execution of the expression $(a - b) + c + (d - e)$ using MIPS assembly. It shows the flow of data between registers `t0`, `t1`, and `t2`. Arrows indicate the flow of values, with some arrows labeled with the constant `2`, likely representing the word size in bytes. The sequence of instructions is as follows:

- `lw t0, a`: Load word from memory location `a` into register `t0`.
- `lw t1, b`: Load word from memory location `b` into register `t1`.
- `sub t0, t0, t1`: Subtract the value in `t1` from the value in `t0`.
- `lw t1, c`: Load word from memory location `c` into register `t1`.
- `add t0, t0, t1`: Add the value in `t1` to the value in `t0`.
- `lw t1, d`: Load word from memory location `d` into register `t1`.
- `lw t2, e`: Load word from memory location `e` into register `t2`.
- `sub t1, t1, t2`: Subtract the value in `t2` from the value in `t1`.
- `add t0, t0, t1`: Add the value in `t1` to the value in `t0`.

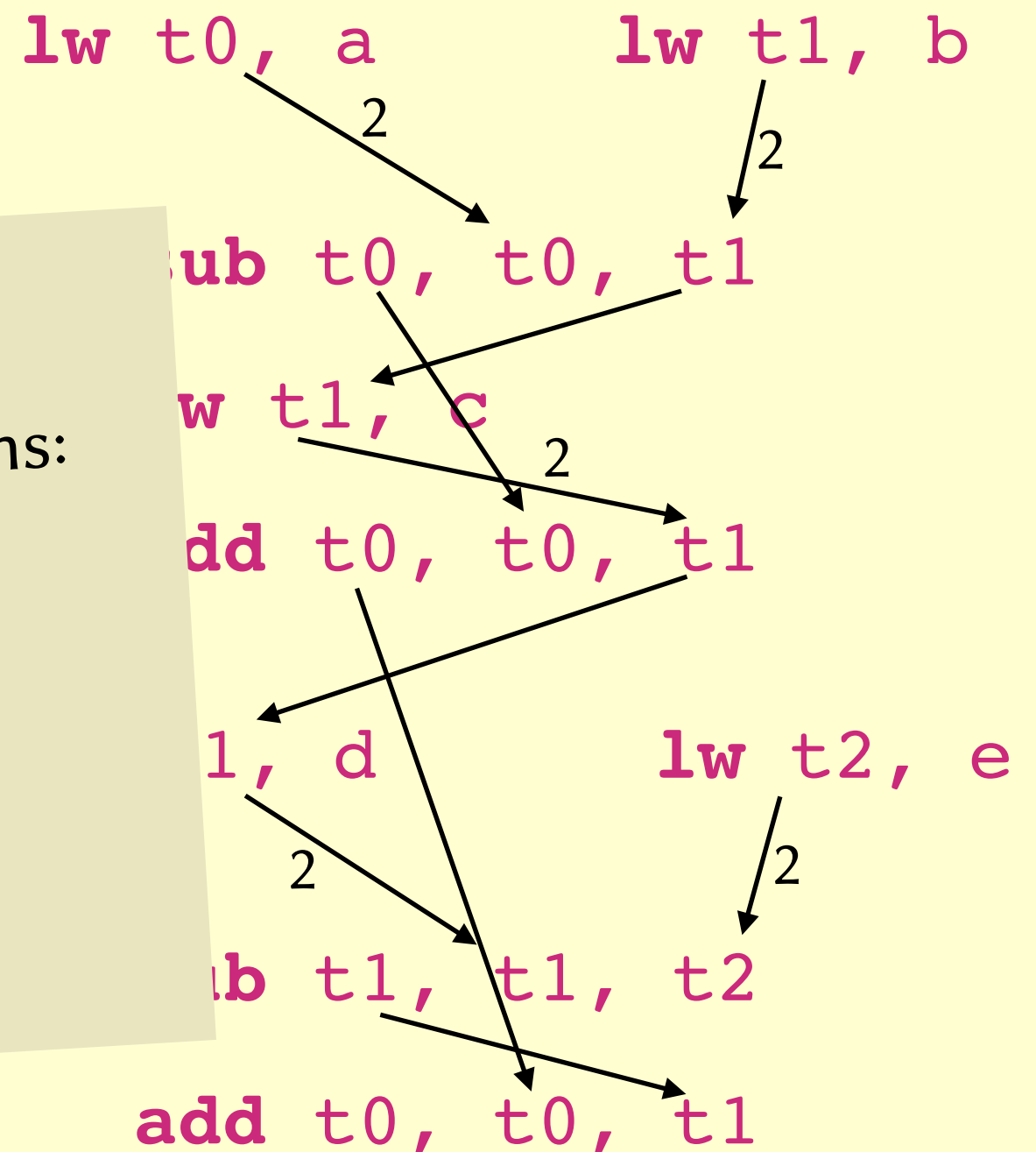
Registers vs. reordering

$(a - b) + c + (d - e)$



Linear extensions:

a, b, c, d, e
 b, a, c, d, e
 a, b, c, e, d
 b, a, c, d, e



Registers vs. reordering

(a -

$r = x + y$
read-after-write
 $z = r + 1$

$r = x + y$
write-after-write
 $r = z + 1$

$x = r + y$
write-after-read
 $r = z + 1$

$\text{lw } t0, a$
 $\text{lw } t1, b$
 $\text{sub } t0, t0, t1$
 $\text{lw } t1, c$
 $\text{add } t0, t0, t1$
 $\text{lw } t1, d$
 $\text{lw } t2, e$
 $\text{sub } t1, t1, t2$
 $\text{add } t0, t0, t1$

Registers vs. reordering

$(a - b) + c + (d - e)$

```
lw t0, a
lw t1, b
sub t0, t0, t1
lw t1, c
add t0, t0, t1
lw t1, d
lw t2, e
sub t1, t1, t2
add t0, t0, t1
```

The diagram illustrates the execution of the expression $(a - b) + c + (d - e)$ using MIPS assembly. It shows the flow of data between registers `t0`, `t1`, and `t2`. Solid arrows represent the flow of values, and dashed arrows represent the flow of values between instructions. Numbers 2 are placed near some arrows, possibly indicating a delay or a specific instruction type.

Registers vs. reordering

$(a - b) + c + (d - e)$

The diagram illustrates the execution of the expression $(a - b) + c + (d - e)$ using registers. It shows a sequence of assembly-like instructions with arrows indicating data flow and a constant value of 2.

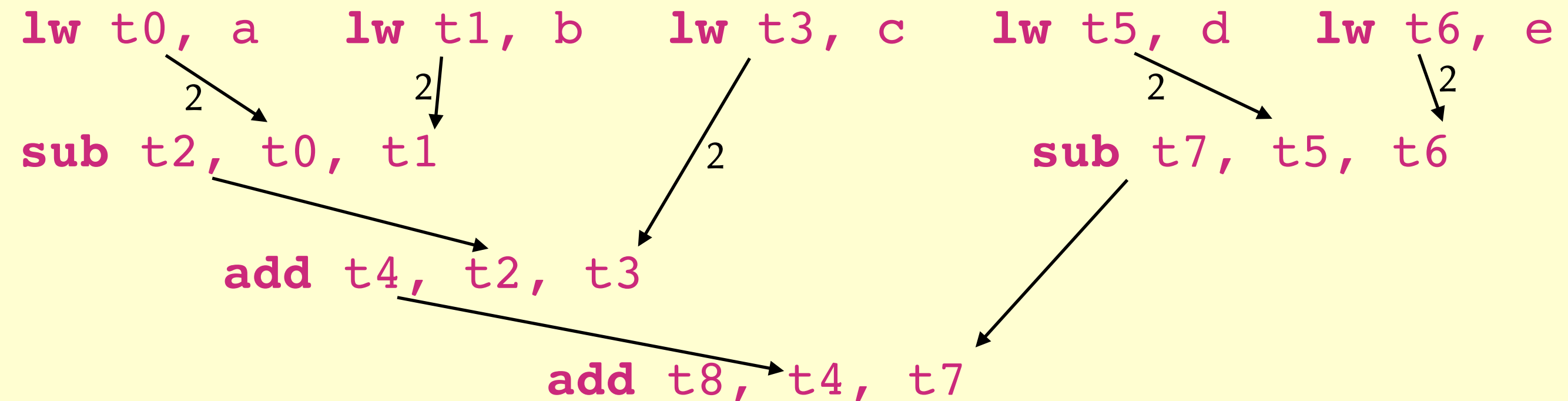
```
lw t0, a
lw t1, b
sub t2, t0, t1
lw t3, c
add t4, t2, t3
lw t5, d
lw t6, e
sub t7, t5, t6
add t8, t4, t7
```

Arrows and annotations:

- From `lw t0, a` to `sub t2, t0, t1` with a '2' above the arrow.
- From `lw t1, b` to `sub t2, t0, t1` with a '2' above the arrow.
- From `sub t2, t0, t1` to `add t4, t2, t3` with a '2' above the arrow.
- From `lw t3, c` to `add t4, t2, t3` with a '2' above the arrow.
- From `add t4, t2, t3` to `add t8, t4, t7` with a '2' above the arrow.
- From `lw t5, d` to `sub t7, t5, t6` with a '2' above the arrow.
- From `lw t6, e` to `sub t7, t5, t6` with a '2' above the arrow.
- From `sub t7, t5, t6` to `add t8, t4, t7` with a '2' above the arrow.

Registers vs. reordering

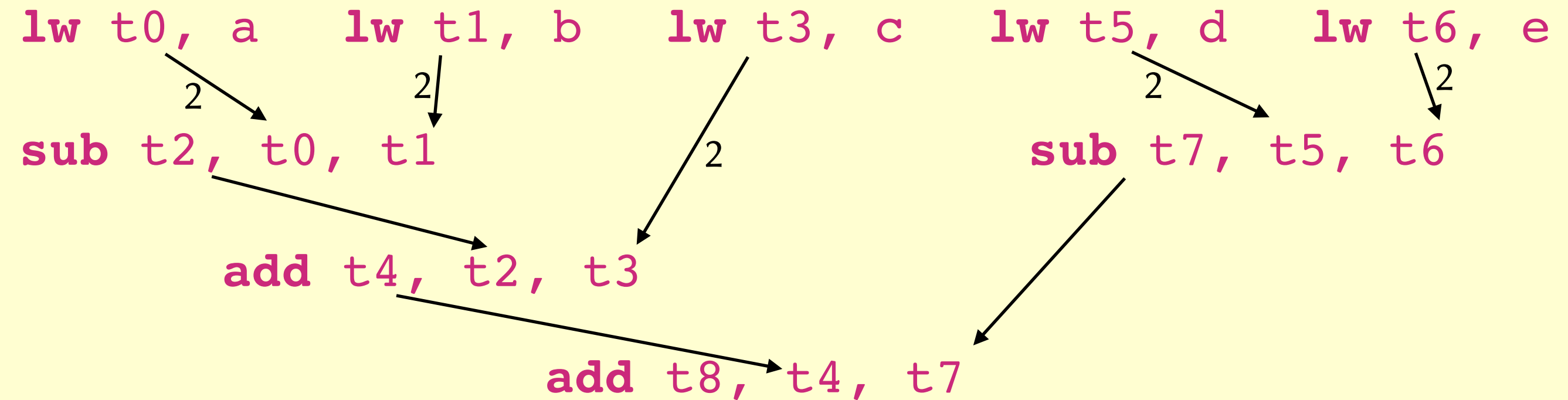
$(a - b) + c + (d - e)$



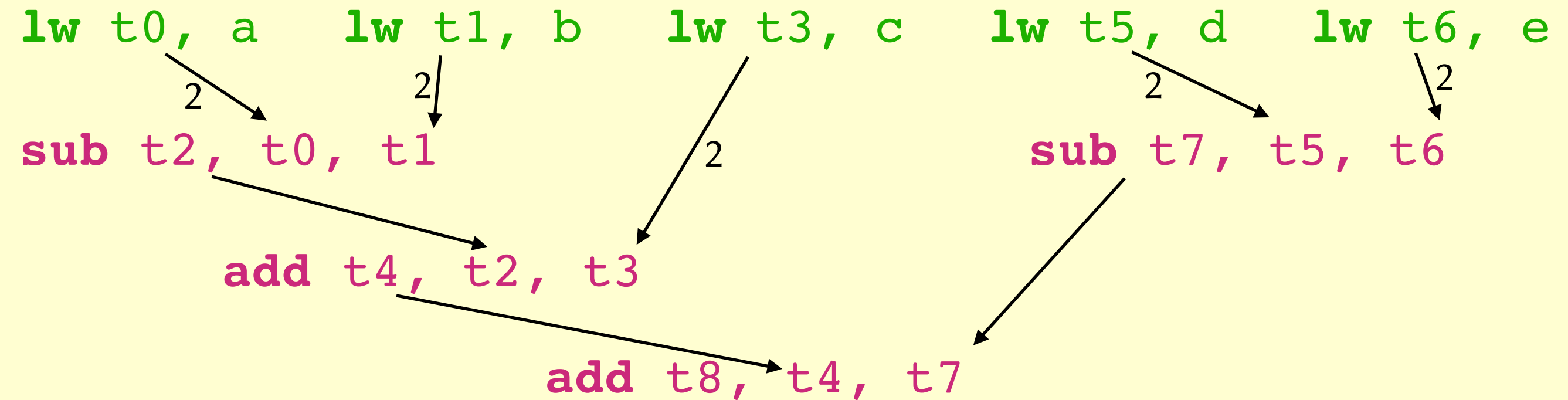
Scheduling

- Hard constraints:
 - if I' uses value calculated by I , then $pos(I) < pos(I')$
 - if I' uses value loaded by I , then $pos(I) + 1 < pos(I')$
- Soft constraints (heuristics):
 - pick instruction likely to conflict with a future instruction
 - pick instruction with furthest path to final instruction
 - keep instructions in source-code order

Scheduling

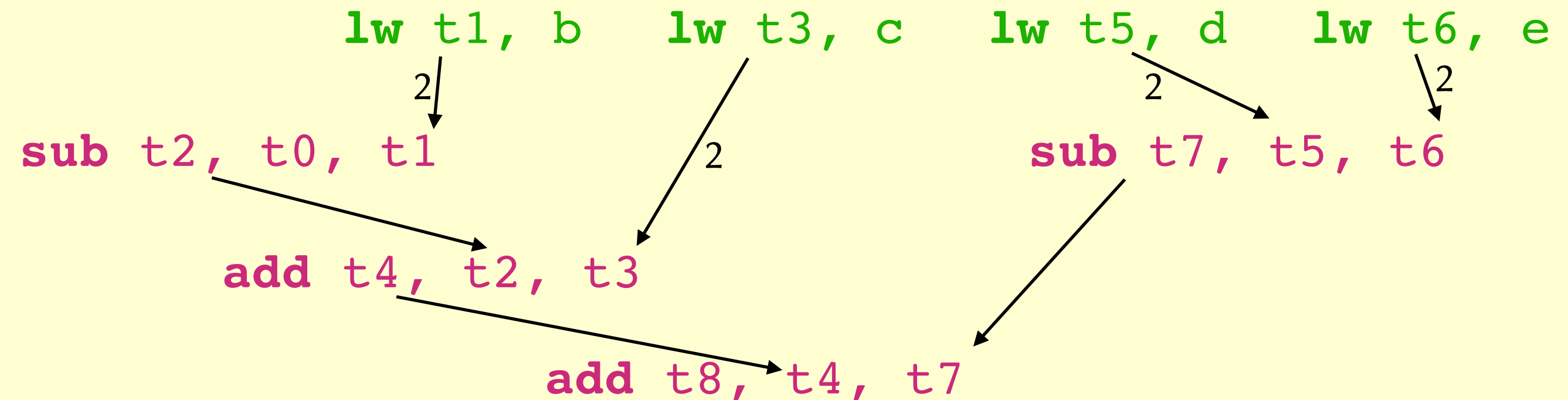


Scheduling



lw t0, a

Scheduling



Scheduling

lw t0, a

lw t1, b

lw t3, c

lw t5, d

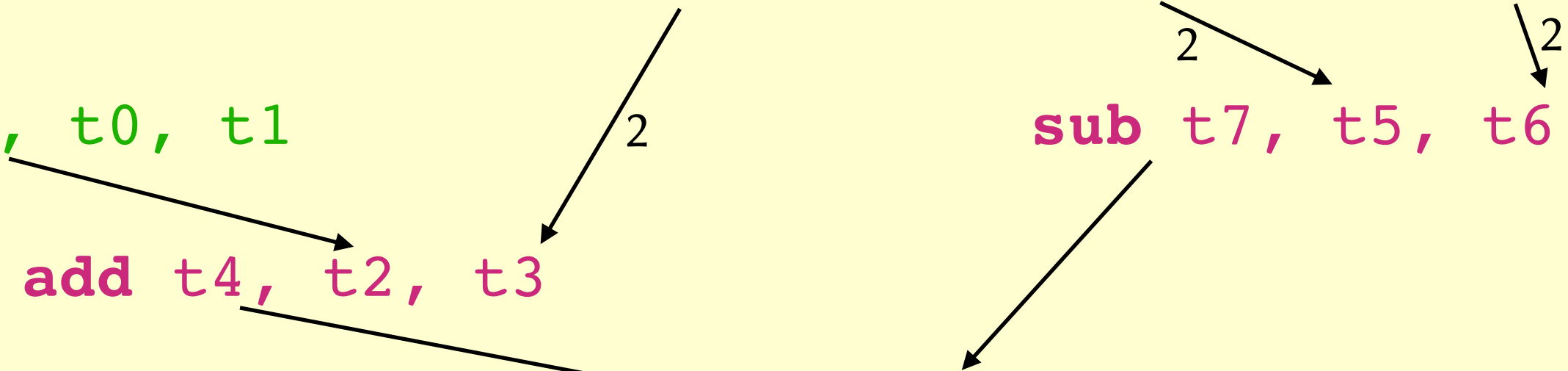
lw t6, e

sub t2, t0, t1

sub t7, t5, t6

add t4, t2, t3

add t8, t4, t7



Scheduling

lw t0, a

lw t1, b

lw t3, c

sub t2, t0, t1

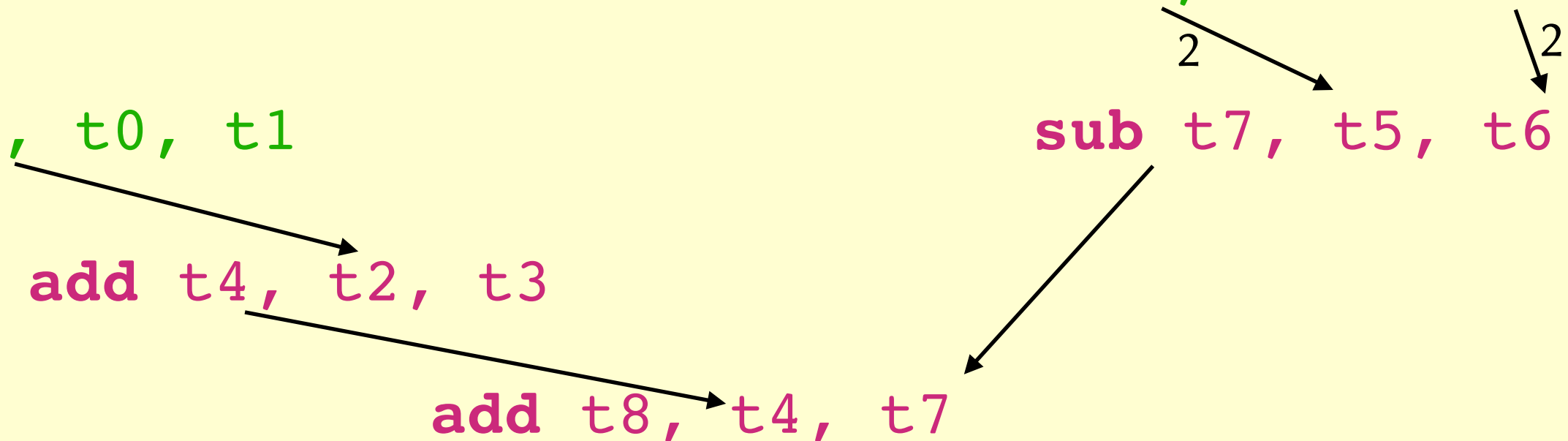
add t4, t2, t3

add t8, t4, t7

lw t5, d

lw t6, e

sub t7, t5, t6



Scheduling

lw t0, a

lw t1, b

lw t3, c

lw t5, d

lw t6, e

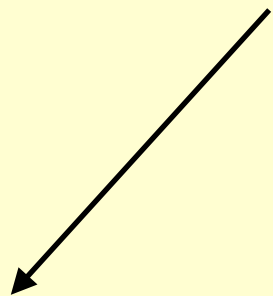
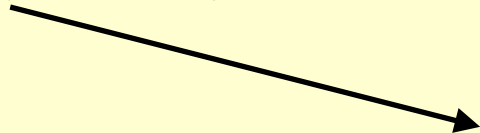
2

sub t2, t0, t1

sub t7, t5, t6

add t4, t2, t3

add t8, t4, t7



Scheduling

lw t0, a

lw t1, b

lw t3, c

lw t5, d

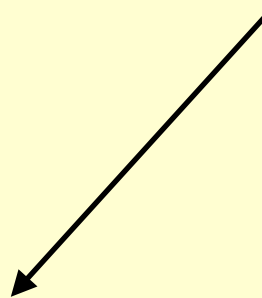
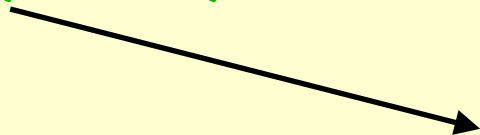
lw t6, e

sub t2, t0, t1

sub t7, t5, t6

add t4, t2, t3

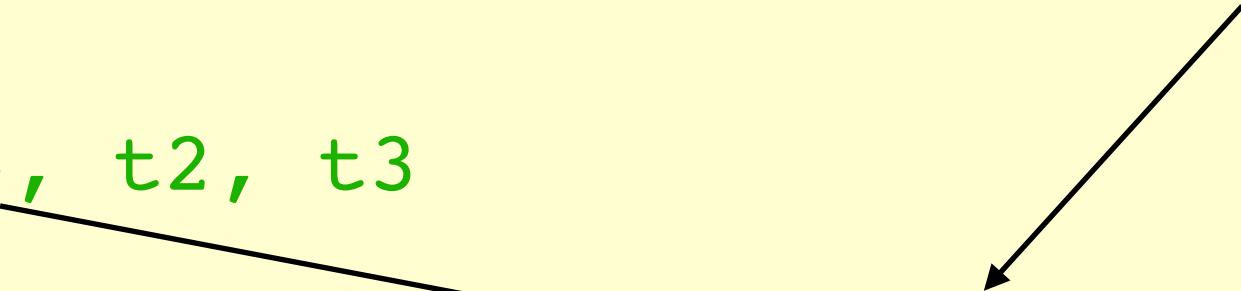
add t8, t4, t7



Scheduling

```
lw t0, a
lw t1, b
lw t3, c
lw t5, d
lw t6, e
sub t2, t0, t1
```

```
add t4, t2, t3
sub t7, t5, t6
add t8, t4, t7
```



Scheduling

lw t0, a

lw t1, b

lw t3, c

lw t5, d

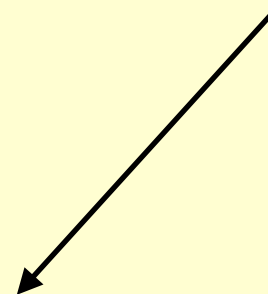
lw t6, e

sub t2, t0, t1

add t4, t2, t3

sub t7, t5, t6

add t8, t4, t7



Scheduling

lw t0, a

lw t1, b

lw t3, c

lw t5, d

lw t6, e

sub t2, t0, t1

add t4, t2, t3

sub t7, t5, t6

add t8, t4, t7

Scheduling

lw t0, a

lw t1, b

lw t3, c

lw t5, d

lw t6, e

sub t2, t0, t1

add t4, t2, t3

sub t7, t5, t6

add t8, t4, t7


```
lw t0, a
lw t1, b
lw t3, c
lw t5, d
lw t6, e
sub t0, t0, t1
add t1, t0, t3
sub t0, t5, t6
add t0, t1, t0
```

Scheduling

Static vs. dynamic scheduling

- **Question.** Why not just leave the instruction scheduling problem for the processor to handle at runtime?

Lecture outline

- Compiler-provided parallelism
 - ✓ Exploiting pipelining in hardware
 - Pipelining in software
 - The polyhedral model of loops
- Programmer-provided parallelism

Software pipelining

```
for (i=0; i<100; i++) {  
    A[i] = A[i] + 42;  
}
```

Software pipelining

```
for (i=0; i<100; i++) {  
    r1 = A[i];  
    r2 = r1 + 42;  
    A[i] = r2;  
}
```

Software pipelining

```
for (p=A; p<A+100; p++) {  
    r1 = *p;  
    r2 = r1 + 42;  
    *p = r2;  
}
```

```
for (p=A; p<A+100; p++) {
```

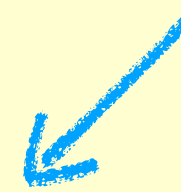
```
    r1 = *p;
```

```
    r2 = r1 + 42;
```

```
    *p = r2;
```

```
}
```

base address of A



```
addi t3, t0, 400
```

```
L1:
```

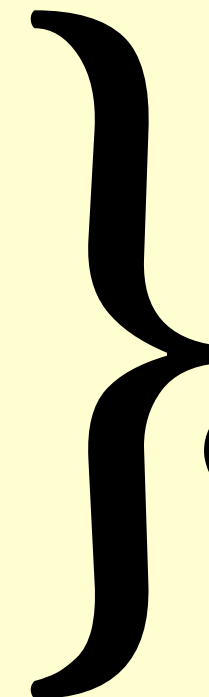
```
lw t1, 0(t0)
```

```
addi t2, t1, 42
```

```
sw t2, 0(t0)
```

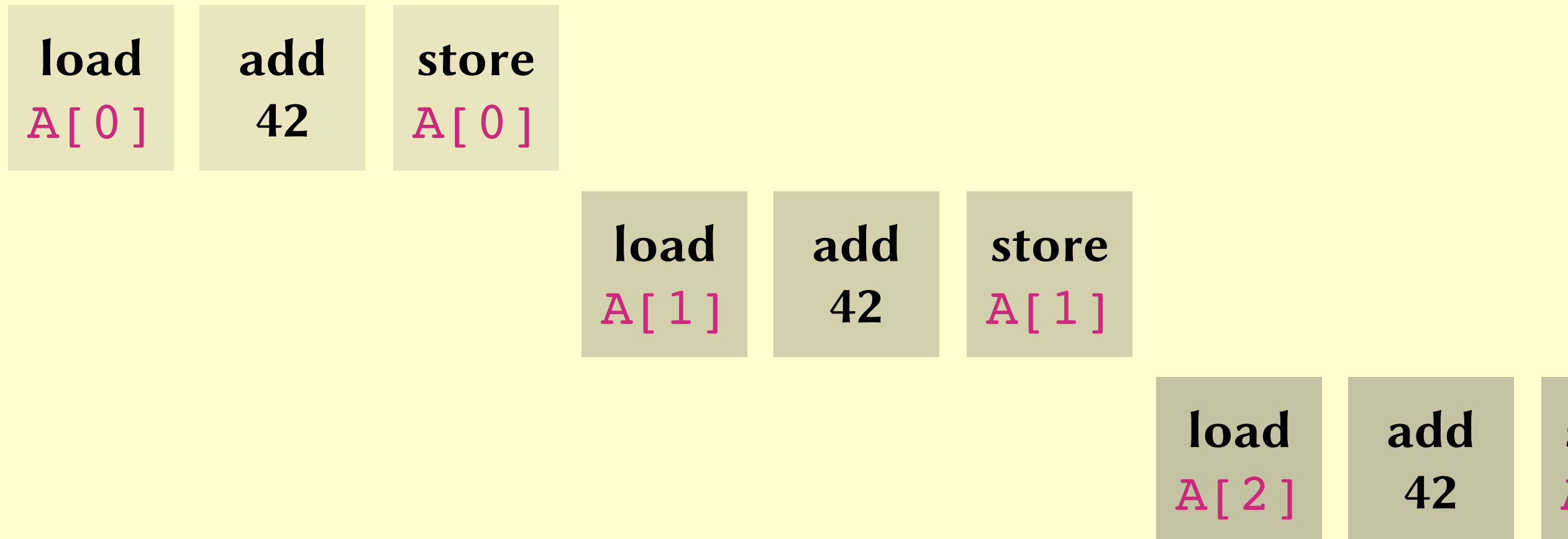
```
addi t0, t0, 4
```

```
bne t0, t3, L1
```

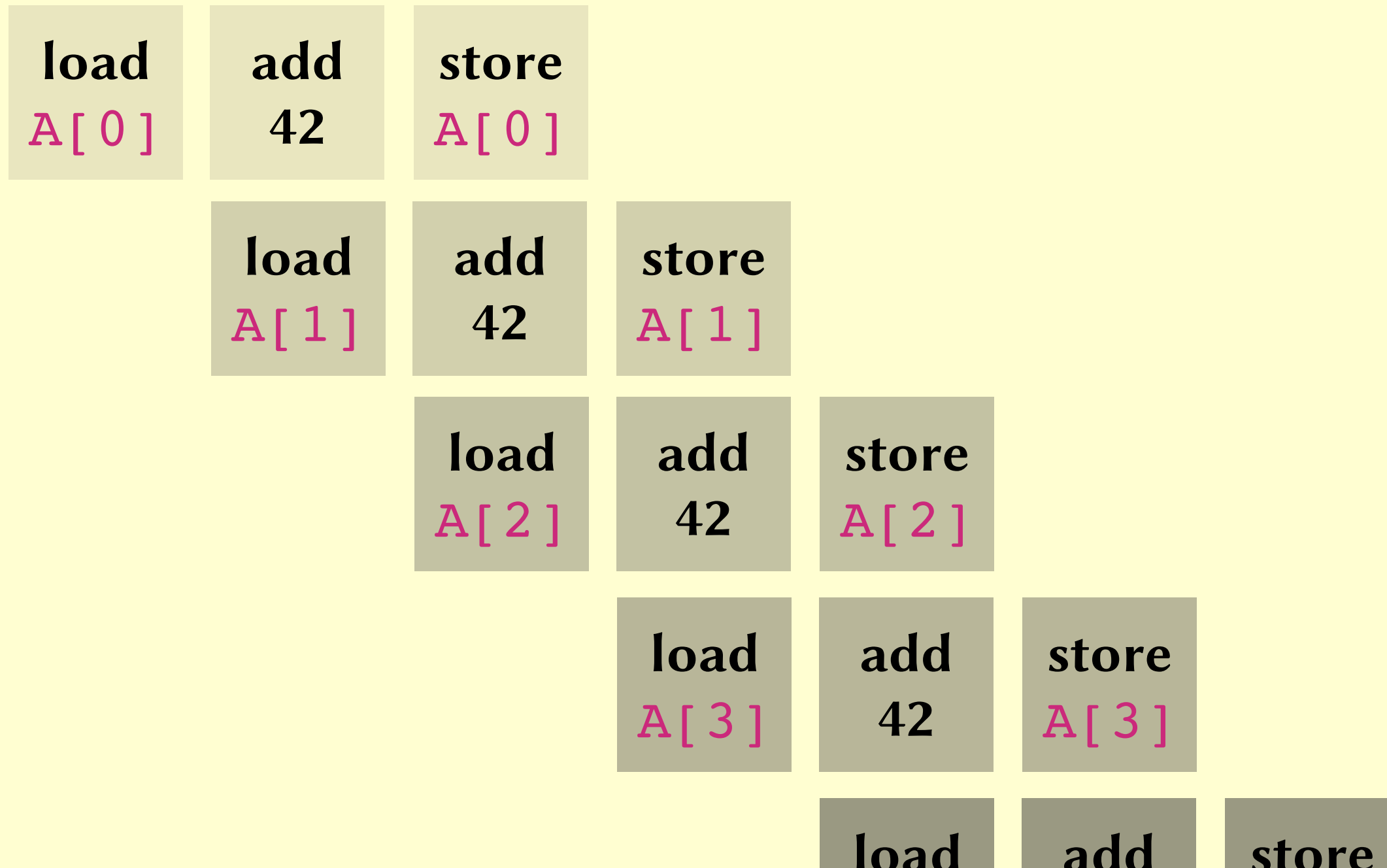


6 timeslots
per iteration
(incl. load delay slot)

Software pipelining



Software pipelining



Software pipelining

preamble `r1 = A[0];`

`r2 = r1 + 42;`

`r1 = A[1];`

for (`i=2; i<100; i++`) {

`A[i-2] = r2;`

`r2 = r1 + 42;`

`r1 = A[i];`

`}`

coda

`A[98] = r2;`

`r2 = r1 + 42;`

`A[99] = r2;`

load
`A[0]`

add
42

store
`A[0]`

load
`A[1]`

add
42

store
`A[1]`

load
`A[2]`

add
42

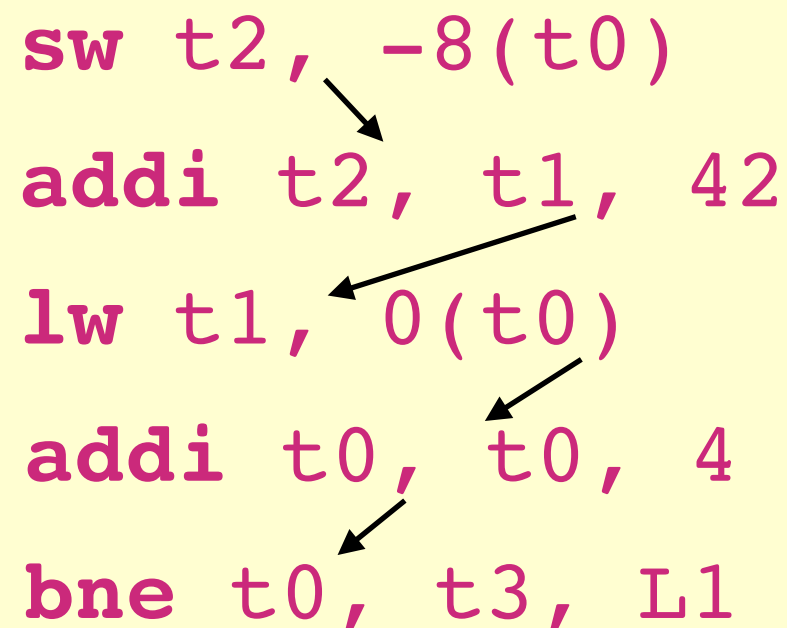
store
`A[2]`

preamble

```
addi t0, t0, 8
addi t3, t0, 400
lw t1, -8(t0)
addi t2, t1, 42
lw t1, -4(t0)
```

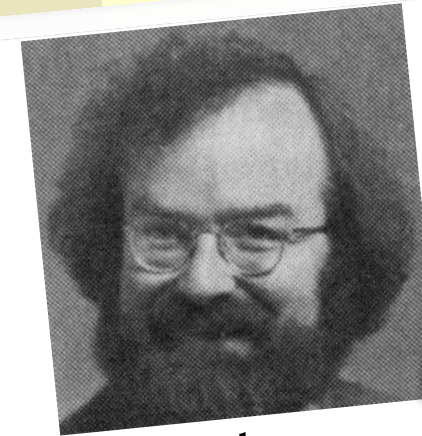
L1:

```
sw t2, -8(t0)
addi t2, t1, 42
lw t1, 0(t0)
addi t0, t0, 4
bne t0, t3, L1
```



coda

```
sw t2, -8(t0)
addi t2, t1, 42
sw t2, -4(t0)
```



Alan
Charlesworth



Monica Lam

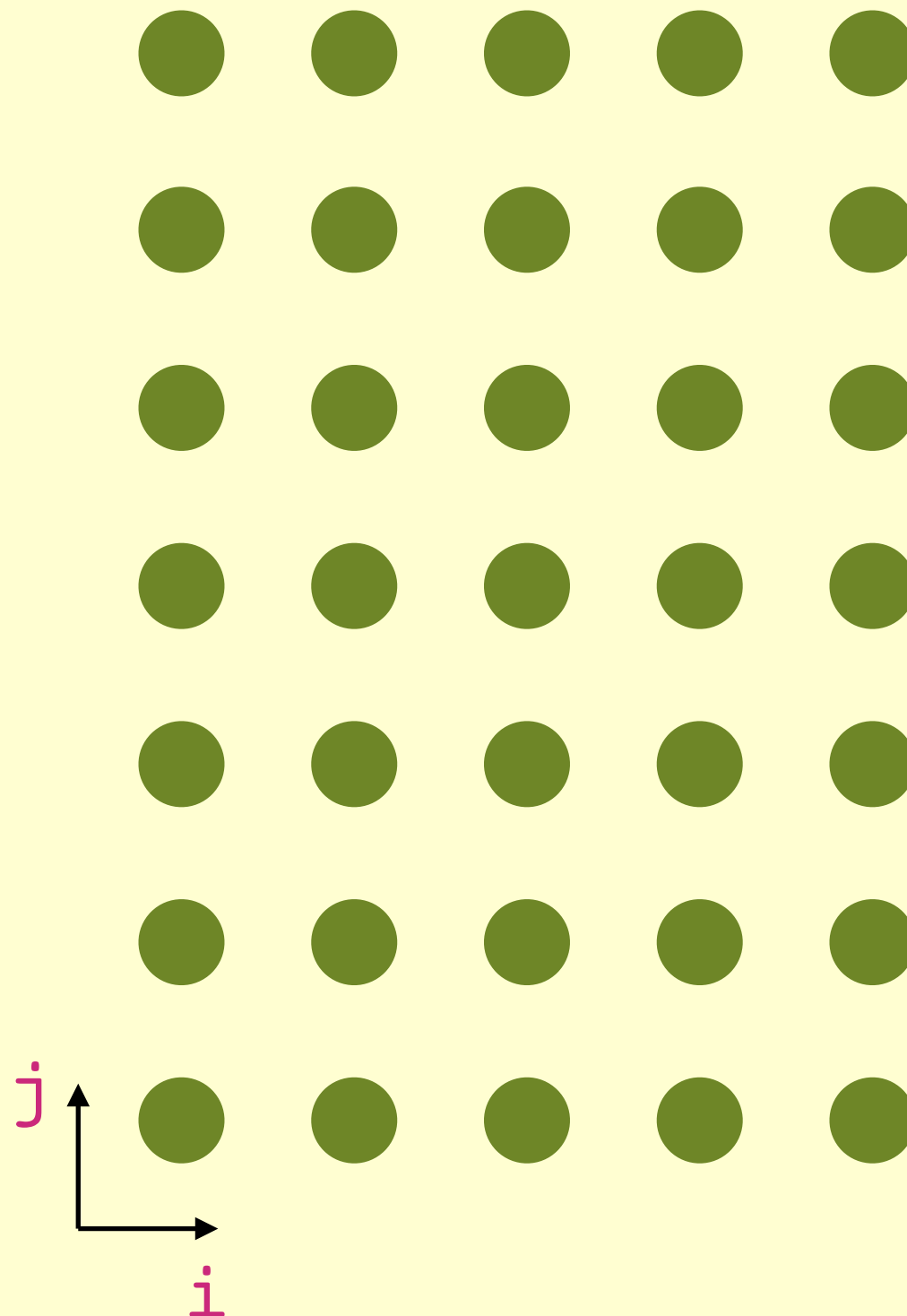
} 5 timeslots
per iteration

Lecture outline

- Compiler-provided parallelism
 - ✓ Exploiting pipelining in hardware
 - ✓ Pipelining in software
 - The polyhedral model of loops
- Programmer-provided parallelism

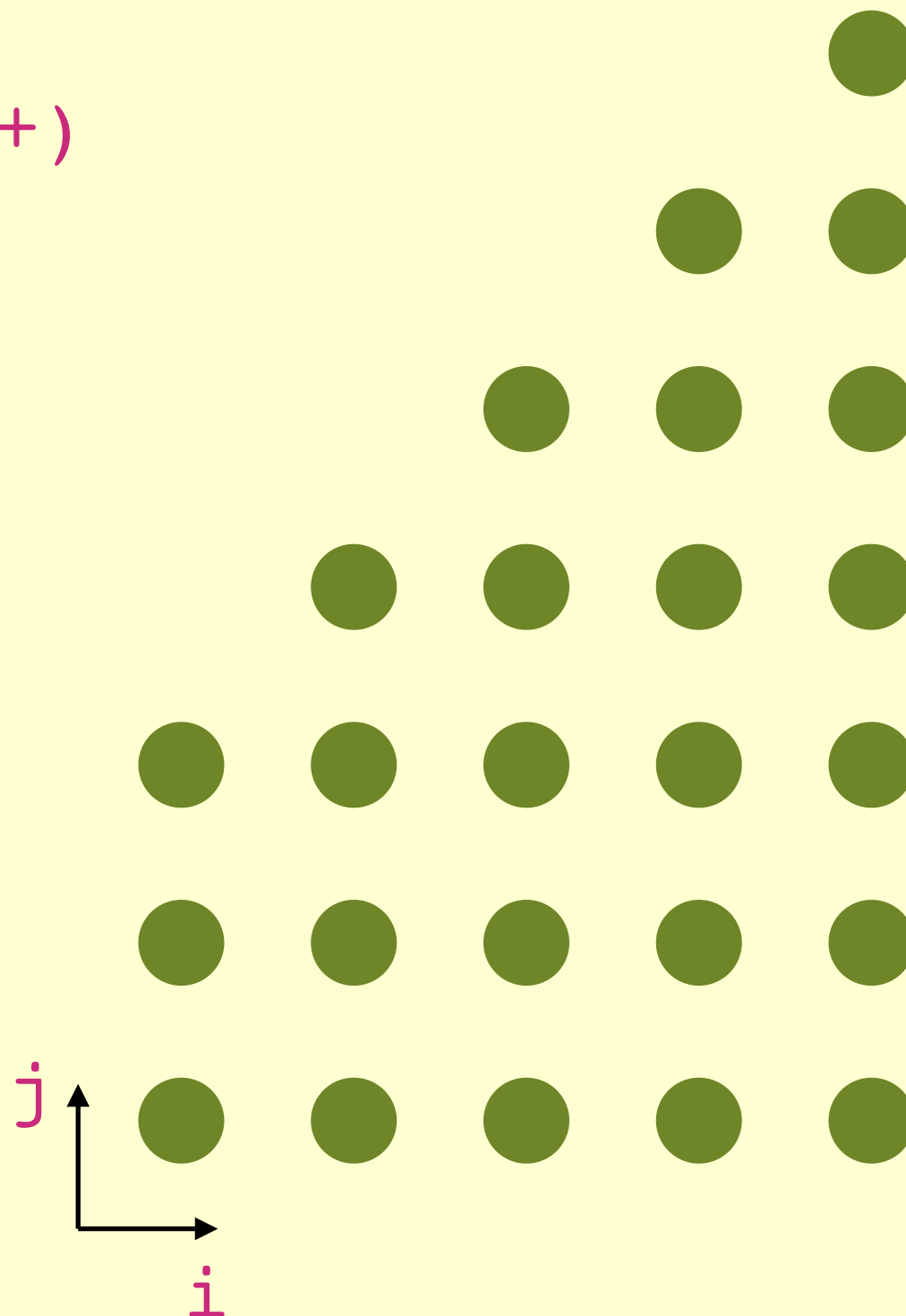
The polyhedral model

```
for (i=1; i<6; i++)  
  for (j=1; j<=7; j++)  
    A[i][j] = 42;
```



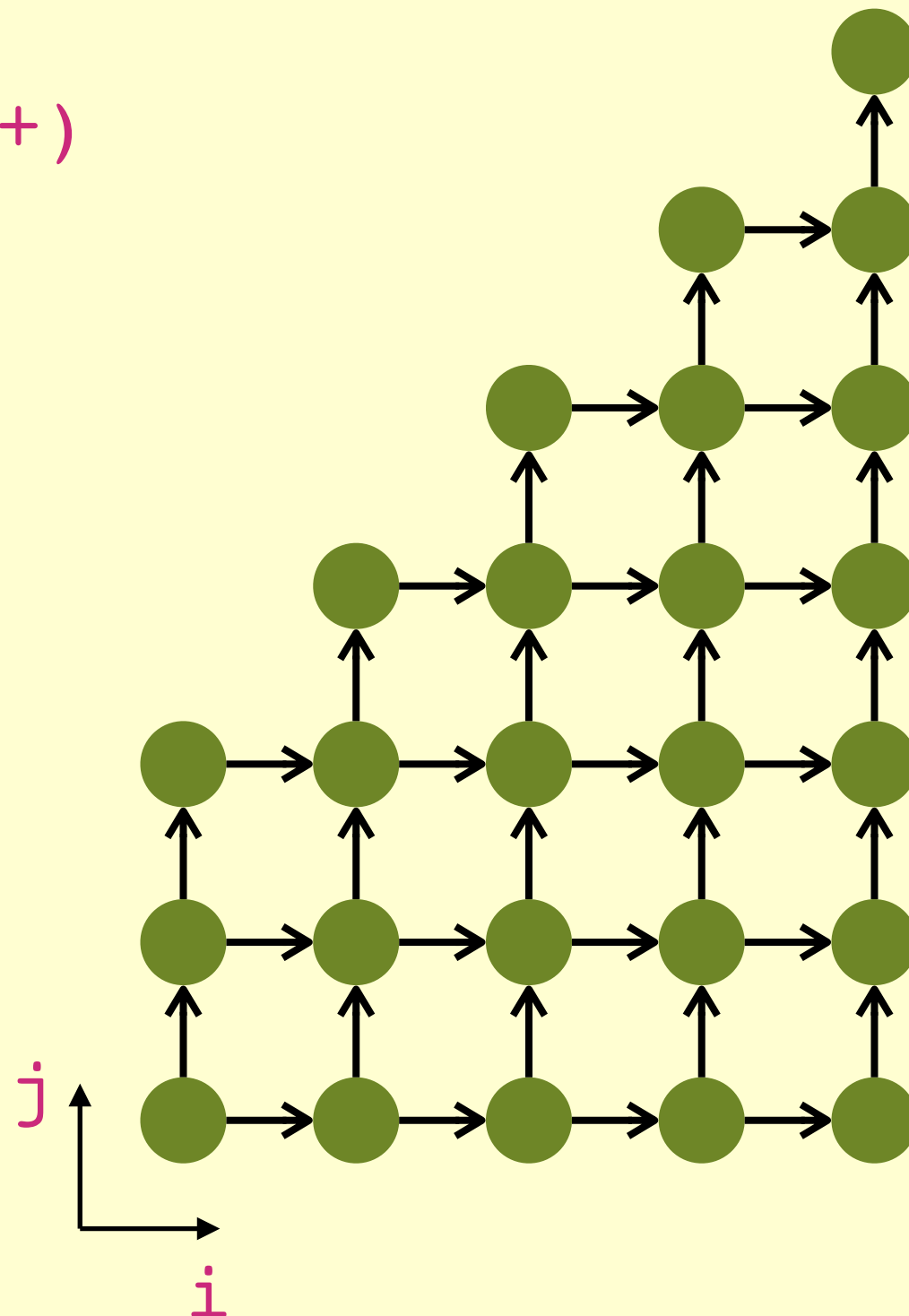
The polyhedral model

```
for (i=1; i<6; i++)  
  for (j=1; j<=i+2; j++)  
    A[i][j] = 42;
```

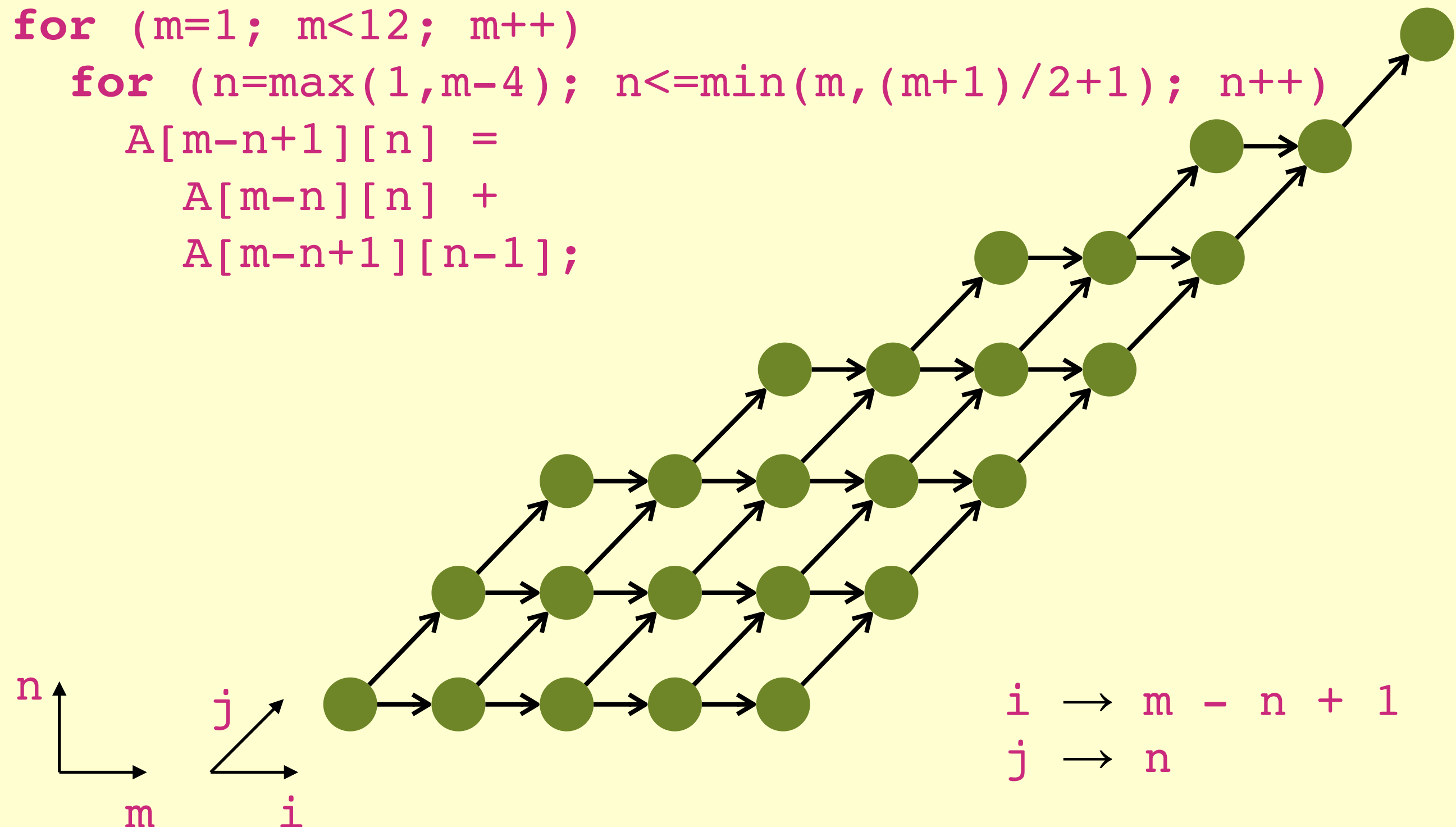


The polyhedral model

```
for (i=1; i<6; i++)  
  for (j=1; j<=i+2; j++)  
    A[i][j] =  
      A[i-1][j] +  
      A[i][j-1];
```



The polyhedral model



Lecture outline

- ✓ Compiler-provided parallelism
 - ✓ Exploiting pipelining in hardware
 - ✓ Pipelining in software
 - ✓ The polyhedral model of loops
- Programmer-provided parallelism

Concurrency vs. reordering

```
MOV [x], $1
```

```
MOV EAX, [y]
```



```
MOV [y], $1
```

```
MOV EBX, [x]
```

Concurrency vs. reordering

```
MOV [x], $1
```

```
MOV EAX, [y]
```



```
MOV [y], $1
```

```
MOV EBX, [x]
```

```
EAX = 1
```

```
EBX = 1
```

Concurrency vs. reordering

```
MOV [x], $1  
MOV EAX, [y]
```



```
MOV [y], $1  
MOV EBX, [x]
```

EAX = 1	EAX = 0
EBX = 1	EBX = 1

Concurrency vs. reordering

MOV [x], \$1		MOV [y], \$1
MOV EAX, [y]		MOV EBX, [x]

EAX = 1

EBX = 1

EAX = 0

EBX = 1

EAX = 1

EBX = 0

Concurrency vs. reordering

```
MOV [x], $1  
MOV EAX, [y]
```

```
MOV [y], $1  
MOV EBX, [x]
```

```
EAX = 1  
EBX = 1
```

```
EAX = 0  
EBX = 1
```

```
EAX = 1  
EBX = 0
```

Concurrency vs. reordering

```
MOV EAX, [y]
```

```
MOV [y], $1
```

```
MOV EBX, [x]
```

```
MOV [x], $1
```

```
EAX = 1
```

```
EBX = 1
```

```
EAX = 0
```

```
EBX = 1
```

```
EAX = 1
```

```
EBX = 0
```

```
EAX = 0
```

```
EBX = 0
```



Concurrency vs. reordering

```
int x = 0; int y = 0;
```

```
x = 1;
```

```
r0 = y;
```

```
||
```

```
y = 1;
```

```
r1 = x;
```


Concurrency vs. reordering

```
atomic_int x = 0; atomic_int y = 0;
```

```
atomic_store(&x, 1);
```

```
r0 = atomic_load(&y);
```

```
||
```

```
atomic_store(&y, 1);
```

```
r1 = atomic_load(&x);
```

Concurrency vs. reordering

```
MOV [x], $1  
MOV EAX, [y]
```

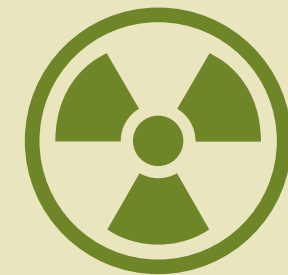
```
MOV [y], $1  
MOV EBX, [x]
```

```
EAX = 1  
EBX = 1
```

```
EAX = 0  
EBX = 1
```

```
EAX = 1  
EBX = 0
```

```
EAX = 0  
EBX = 0
```



Concurrency vs. reordering

```
MOV [x], $1
```

```
MFENCE
```

```
MOV EAX, [y]
```



```
MOV [y], $1
```

```
MFENCE
```

```
MOV EBX, [x]
```

```
EAX = 1
```

```
EBX = 1
```

```
EAX = 0
```

```
EBX = 1
```

```
EAX = 1
```

```
EBX = 0
```

```
EAX = 0
```

```
EBX = 0
```

Key message. Be aware of what the target architecture guarantees when compiling concurrency.



Summary

- Compilers can reorder instructions to exploit pipelined CPUs.
- Minimising registers conflicts with maximising reorderability.
- Optimal reorderings are hard to find, so heuristics are used.
- Can apply pipelining ideas to loops in software.
- Can view loops as polyhedra to expose further optimisations.
- Architectures may [appear to] reorder instructions even if your compiler doesn't, so be careful when compiling concurrency.