面向目标机器的代码优化
——指令并行与调度

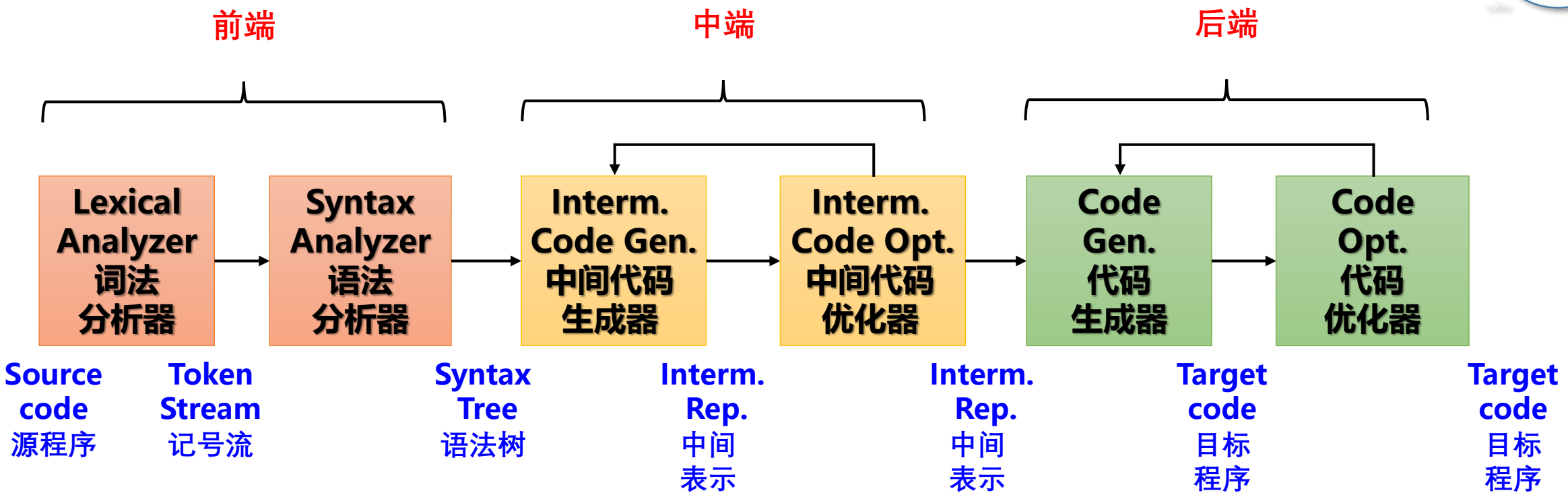
李 诚

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

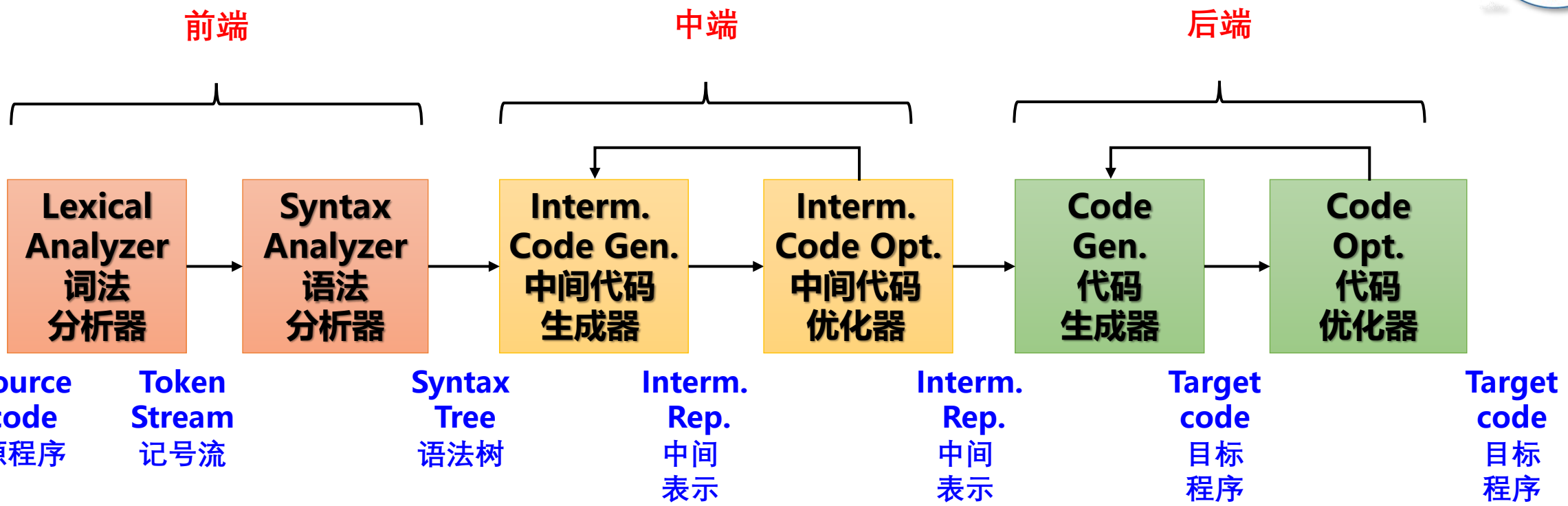
2023年11月29日

前情回顾



- 编译器的基本步骤

前情回顾

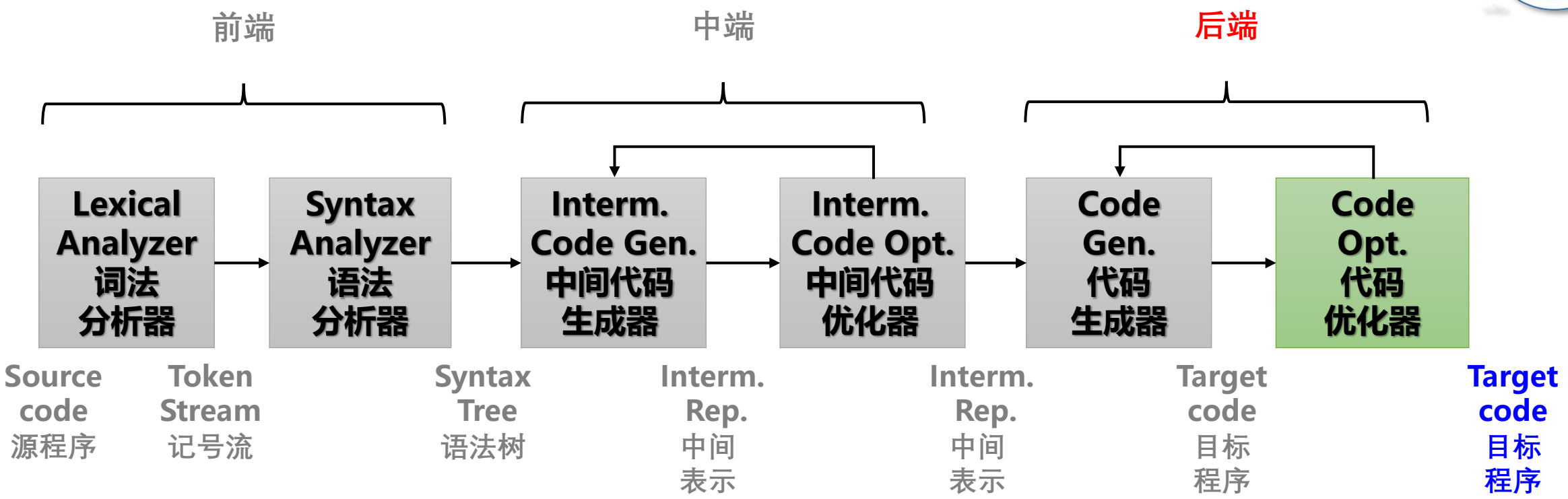


技术方案：Flex → Bison → LLVM/LightIR → LoongArch64

实验：Lab0 → Lab1 → Lab2 → Lab3 → Lab4

- 编译器的基本步骤与实验安排

前情回顾



技术方案：Flex → Bison → LLVM/LightIR → LoongArch64

实验：Lab0 → Lab1 → Lab2 → Lab3 → Lab4

- 编译器的基本步骤与实验安排

面向目标机器的代码优化 – almost final

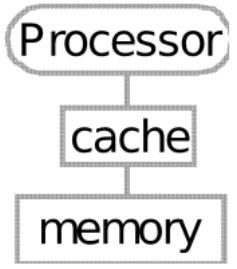
- 目标：优化生成的机器代码，与机器无关的优化不同，这一层级的信息是IR层无法获取的。
- 面向目标机器的代码优化十分重要，但往往很难实现:
  - 难以跨机器架构复用
  - 难以跨语言复用

面向目标机器的代码优化 – 种类

- 减少操作数量
  - 算术操作、内存访问等
- 用代价小的操作替换代价高的操作
  - 例如：replace 4-cycle multiplication with 1-cycle shift
- 降低缓存缺失（Cache miss）
  - 覆盖数据和指令的访问
- 并行计算
  - 单线程内部的指令调度
  - 跨线程的并行执行

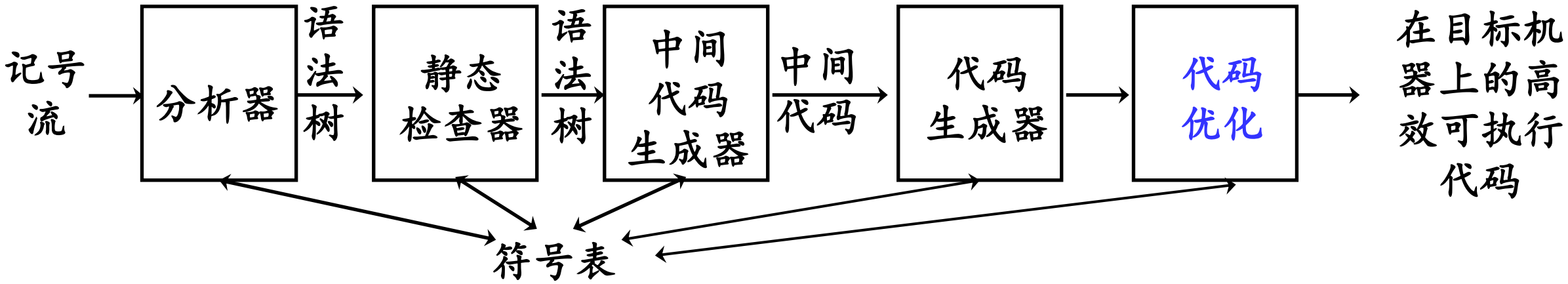
执行时间的计算公式：

Execution time = Operation count * Machine cycles per operation



冯诺依曼体系结构

本节提纲



- 现代处理器架构
- 流水线并行的例子
- 指令调度与数据依赖分析
- 数据依赖指导下的指令调度
- 科技前沿——大模型的流水并行训练

现代处理器架构

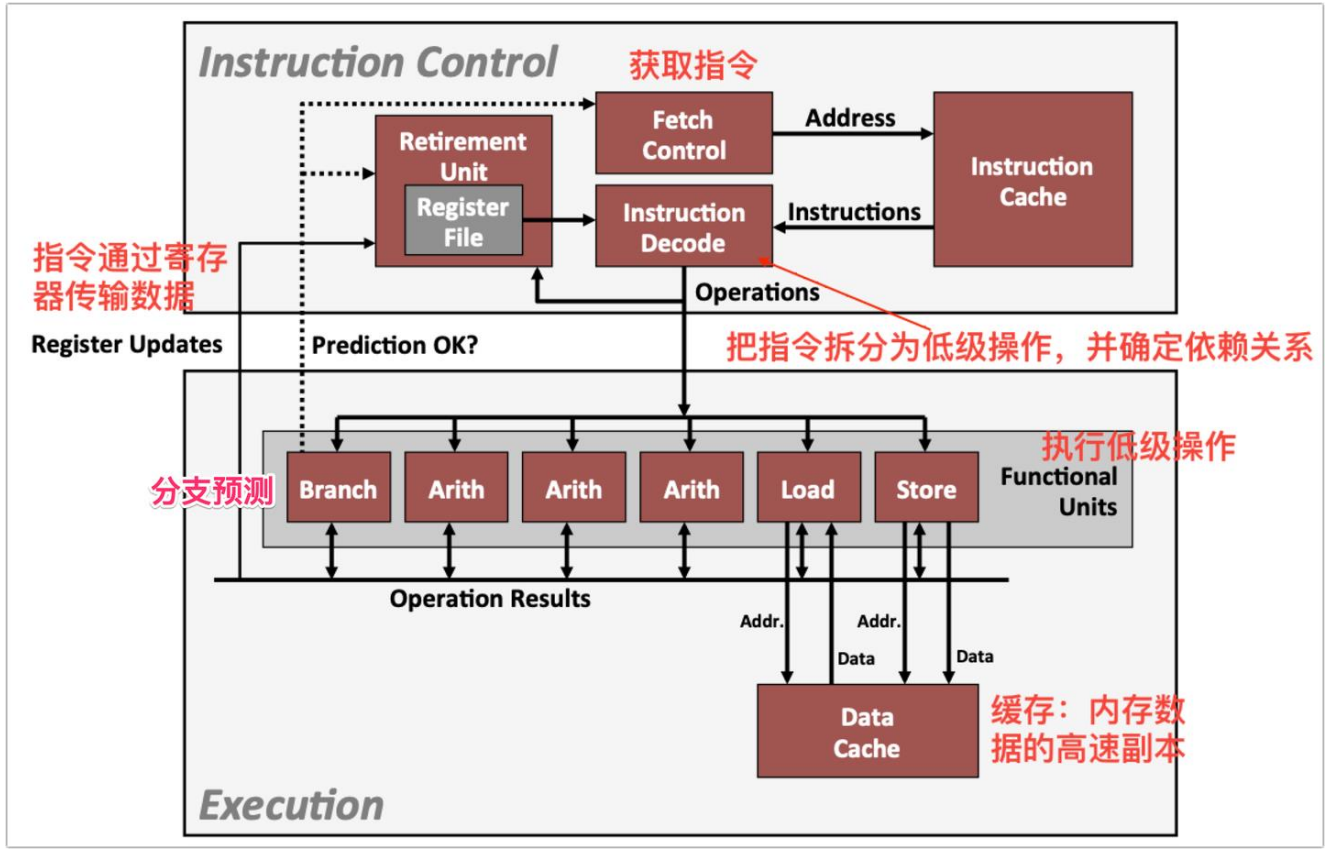
指令控制单元(ICU)

Fetch control

- 包含分支预测的功能

Instruction decode

- 从icache中读取指令，然后翻译为一组微操作
- 例如，addq %rax, %rdx转换为单个微操作
- 例如，addq%rax, 8(%rdx)转换为内存读取、加法和内存写入三个微操作。

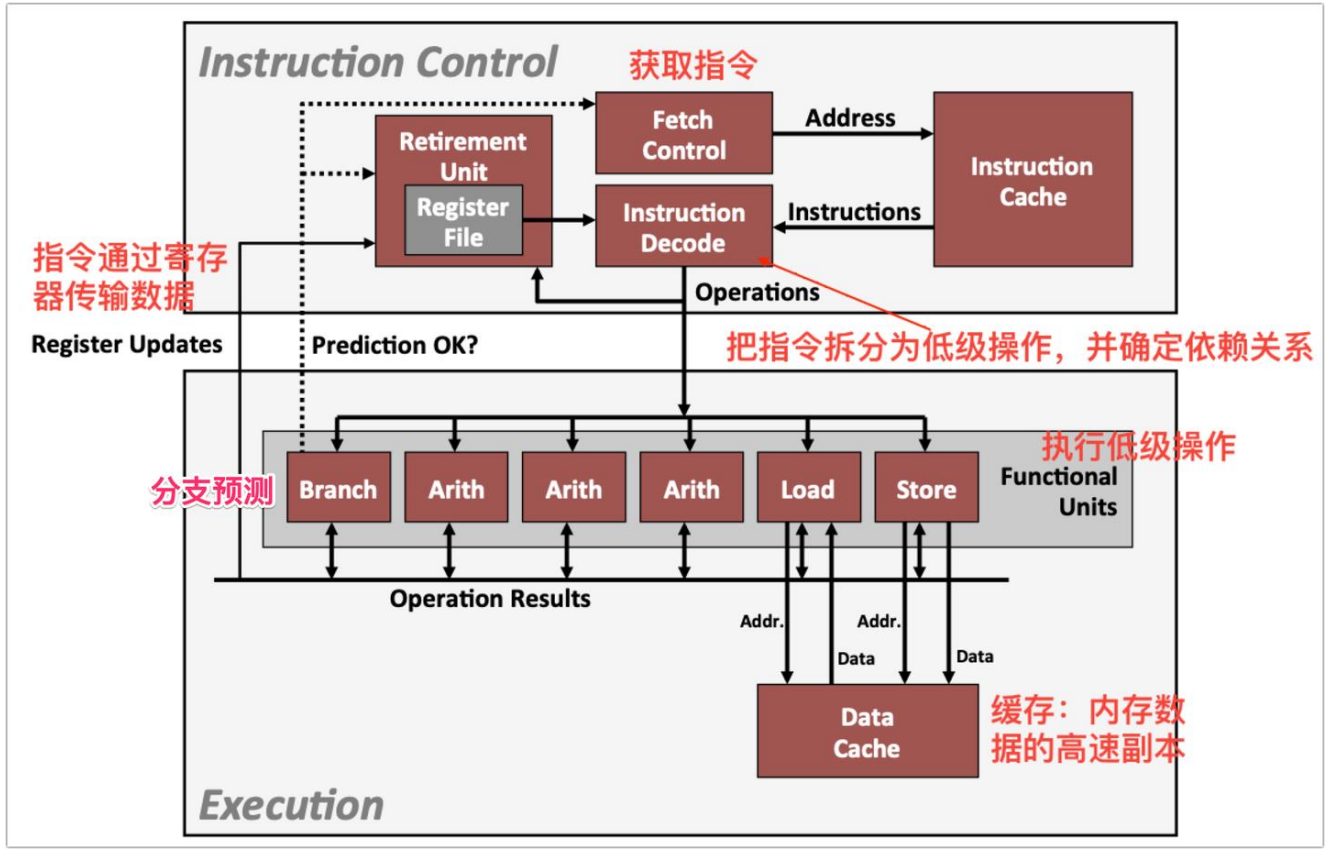


现代处理器架构

现代处理器架构

执行单元(EU, Execution Unit)

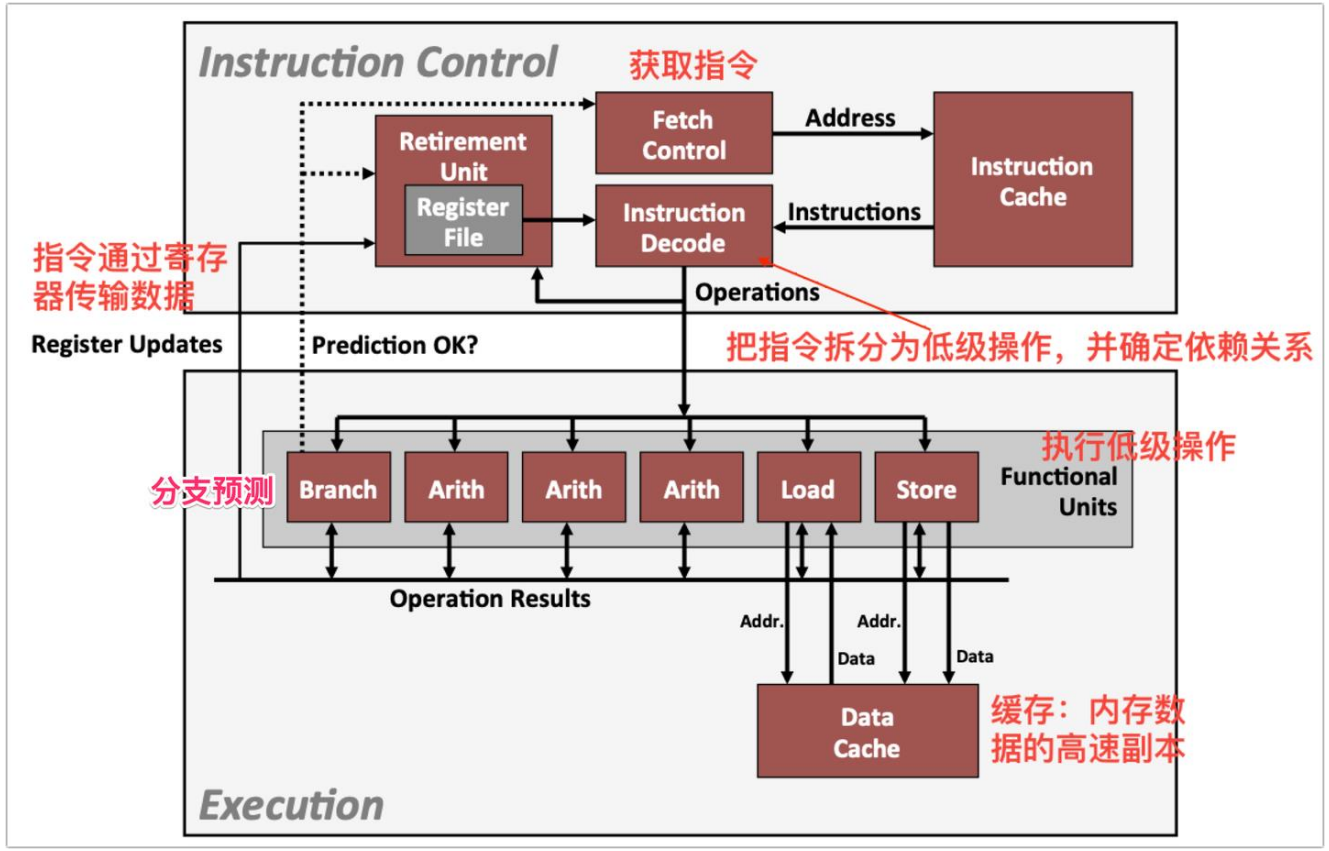
- 接收来自ICU的微操作，分发到各个功能单元执行。
- Load和Store单元
  - 包含一个加法器计算地址，和dcache交互
- Branch单元
  - 预测结果会保存在EU内的队列中，若预测错误，则会丢弃保存的执行结果，并通知Fetch Control单元，之后才能获取正确的指令
- 其它各种功能单元
  - 整数运算、浮点乘、整数乘、分支等等



现代处理器架构

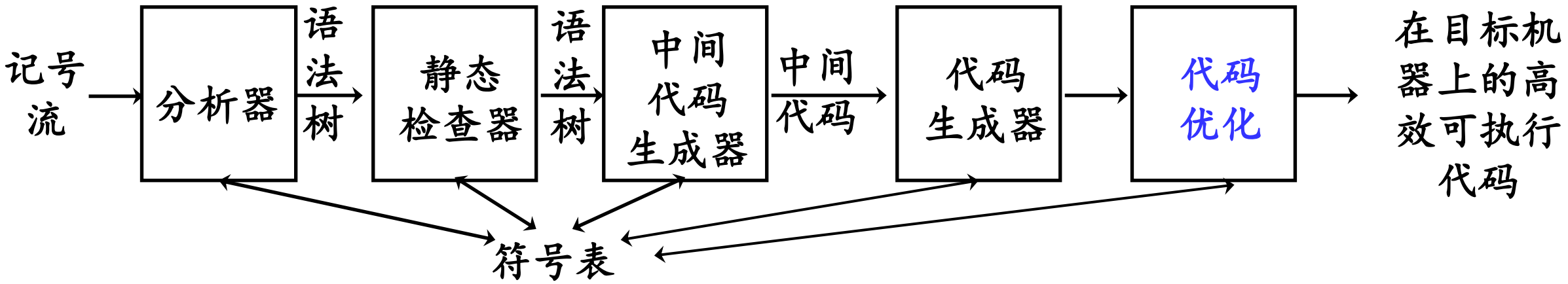
现代处理器架构：乱序 + 超标量

- 现代处理器一般是乱序且是超标量的。
  - 超标量：通过实现多个硬件单元，可以在每个时钟周期执行多个操作
  - 乱序：指令执行的顺序和二进制代码中的顺序不一定相同



现代处理器架构

本节提纲

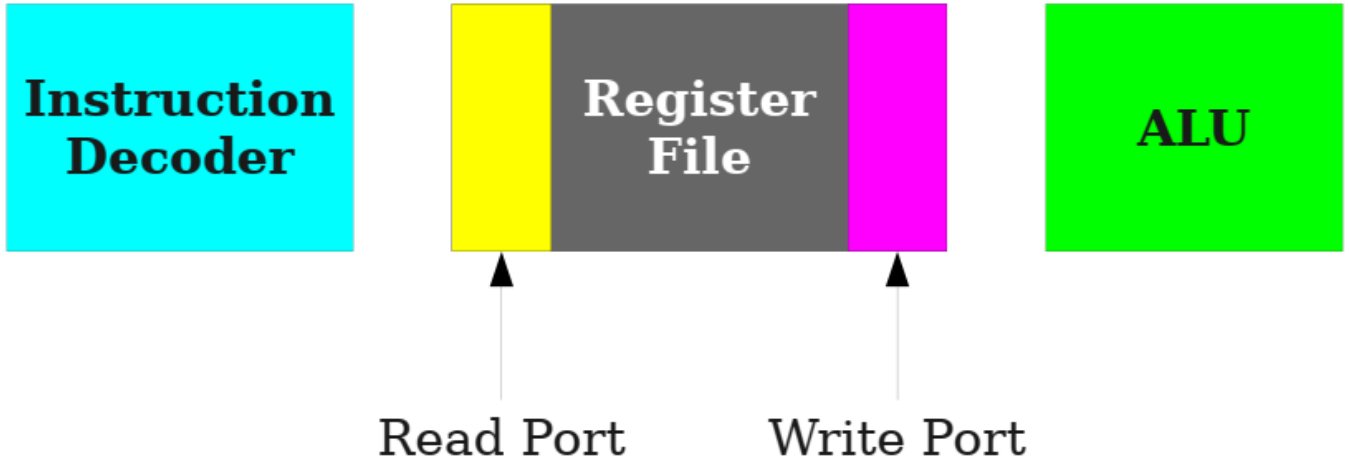


- 现代处理器架构
- 流水线并行的例子
- 指令调度与数据依赖分析
- 数据依赖指导下的指令调度
- 科技前沿——大模型的流水并行训练

处理器流水线

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

处理器流水线

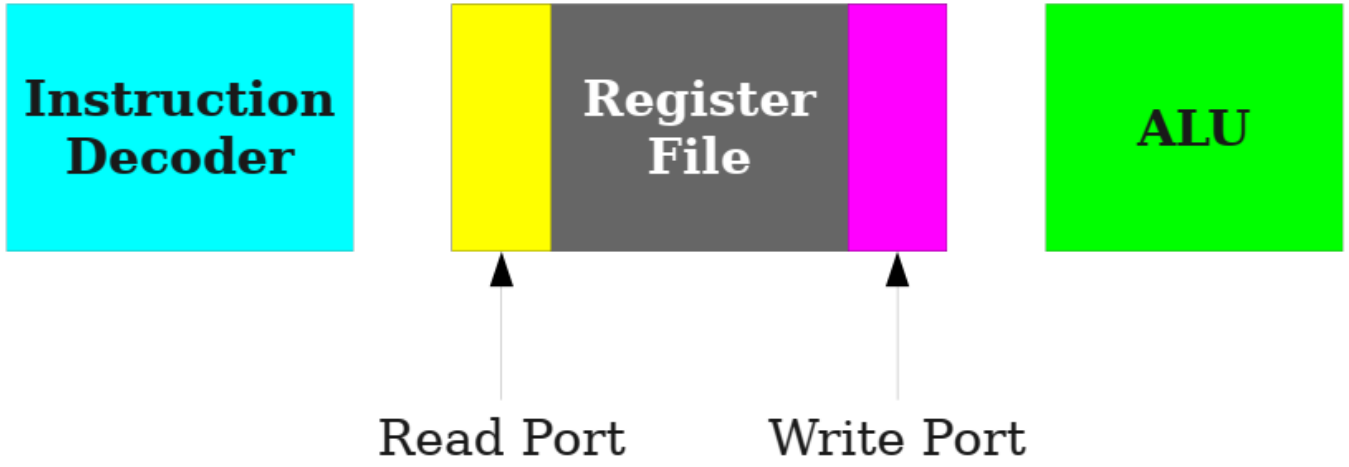



```
add $t2, $t0, $t1      # $t2 = $t0 + $t1

add $t5, $t3, $t4      # $t5 = $t3 + $t4

add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

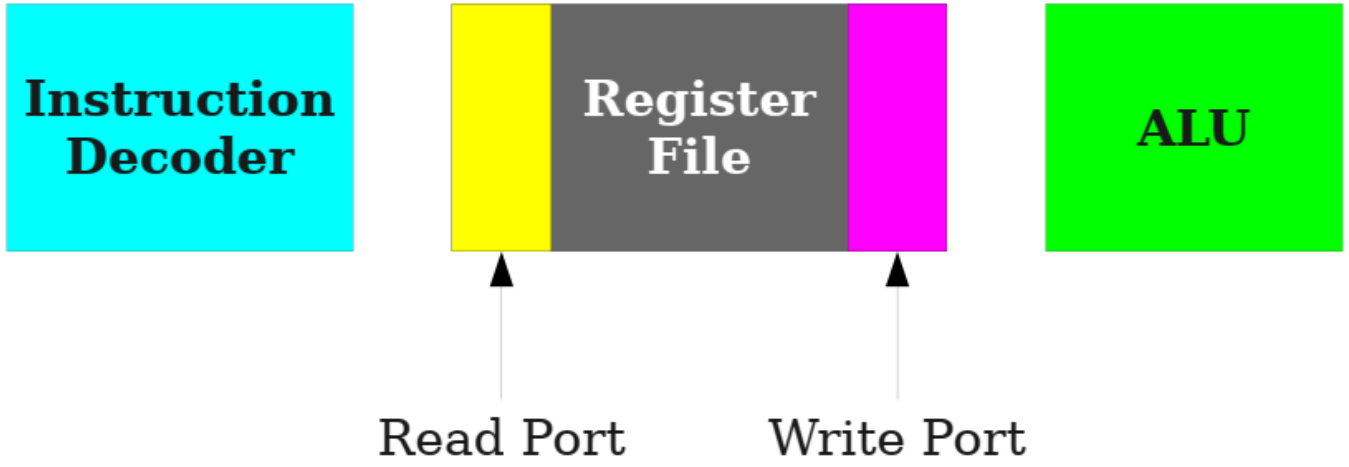
处理器流水线




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

ID	RR	ALU	RW

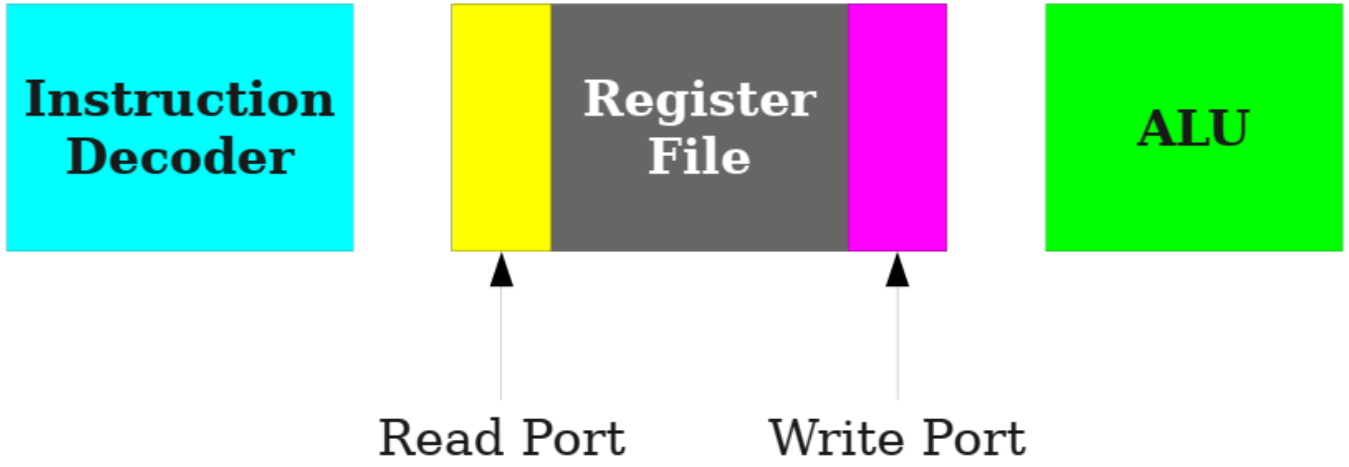
处理器流水线




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

ID	RR	ALU	RW

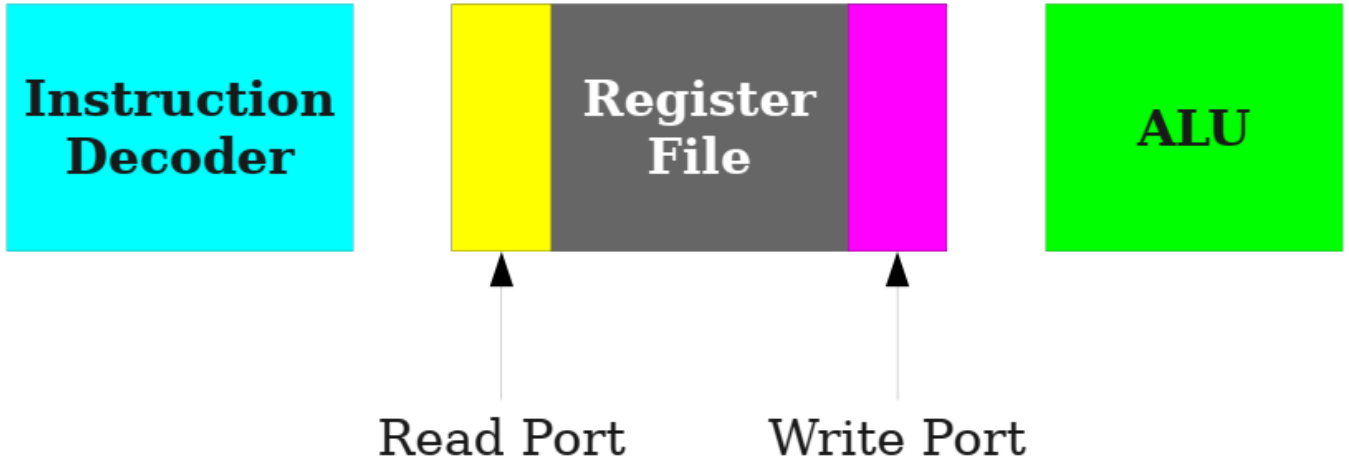
处理器流水线



ID	RR	ALU	RW

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

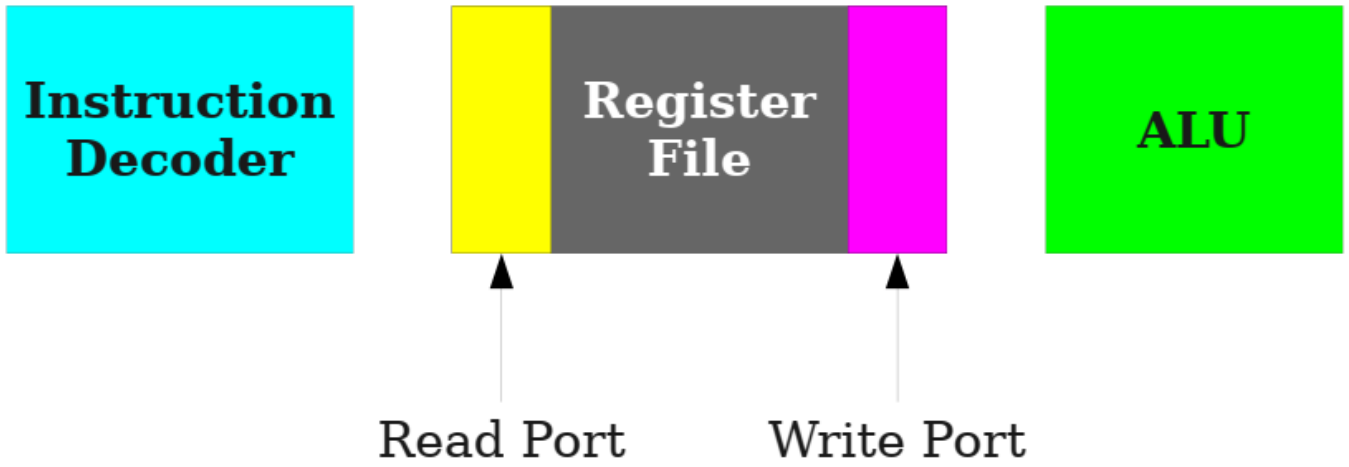
处理器流水线



ID	RR	ALU	RW

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

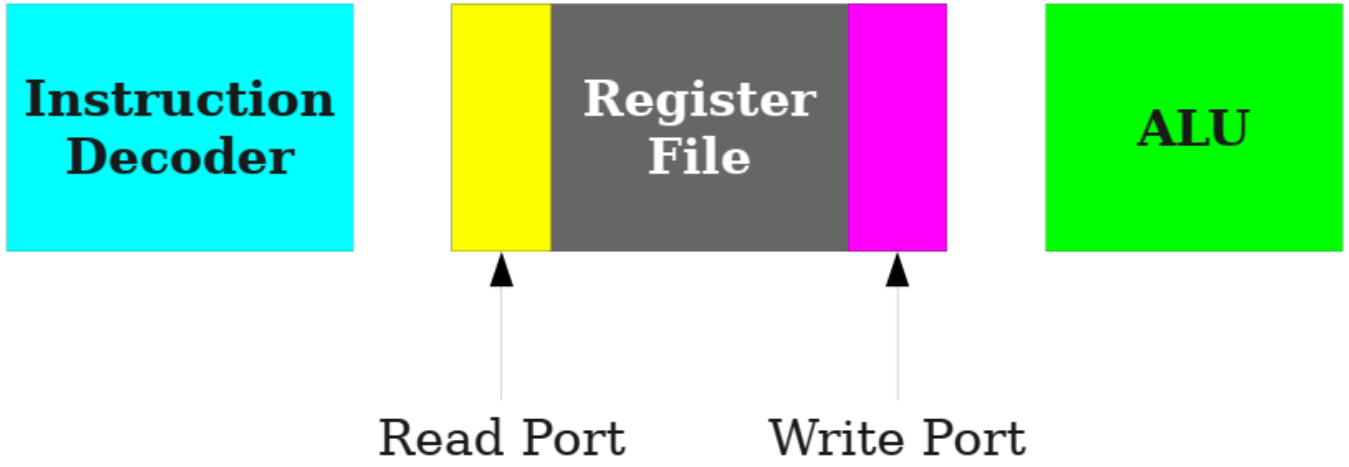
处理器流水线



ID	RR	ALU	RW

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

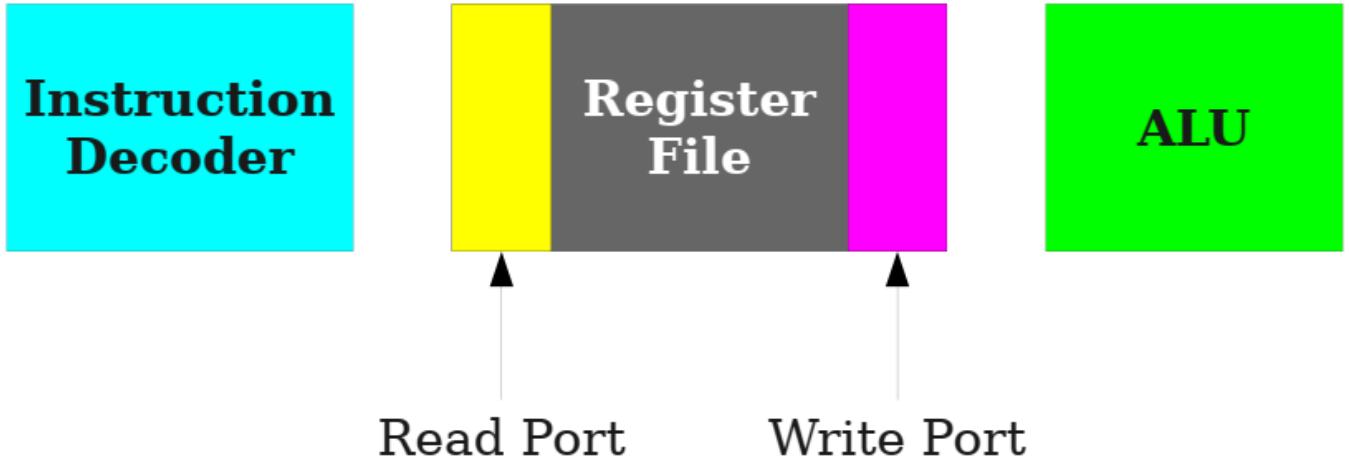
处理器流水线



ID	RR	ALU	RW

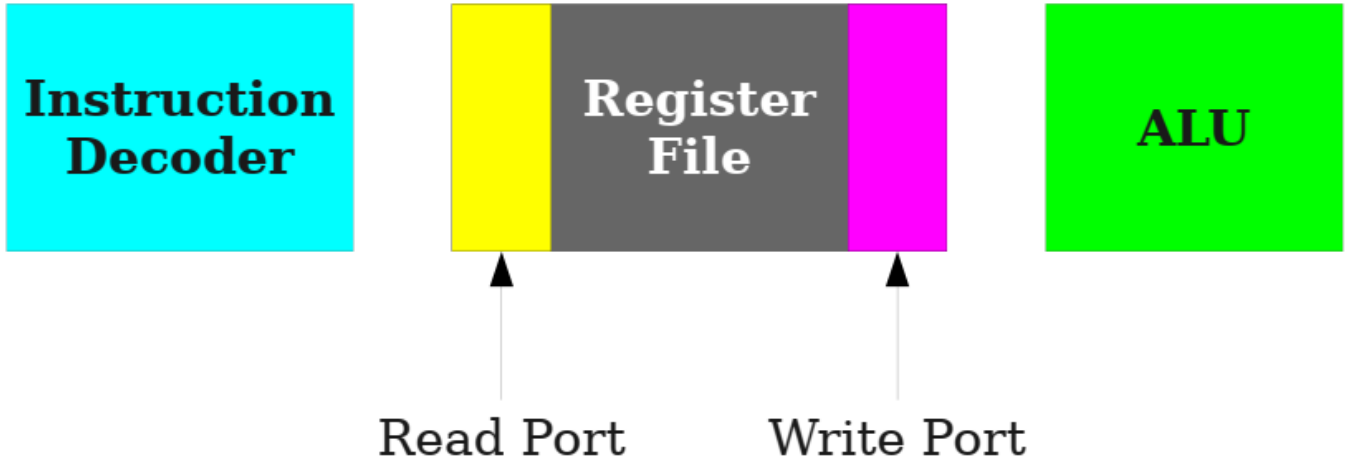
```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

复杂的流水线并行




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

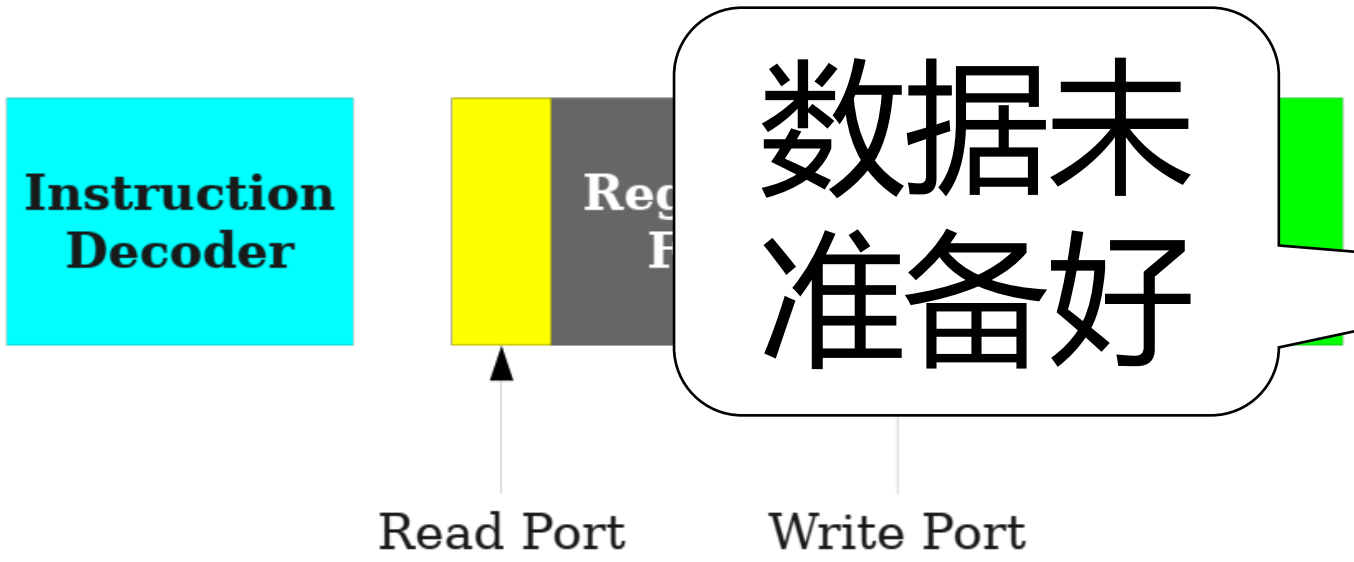
复杂的流水线并行




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

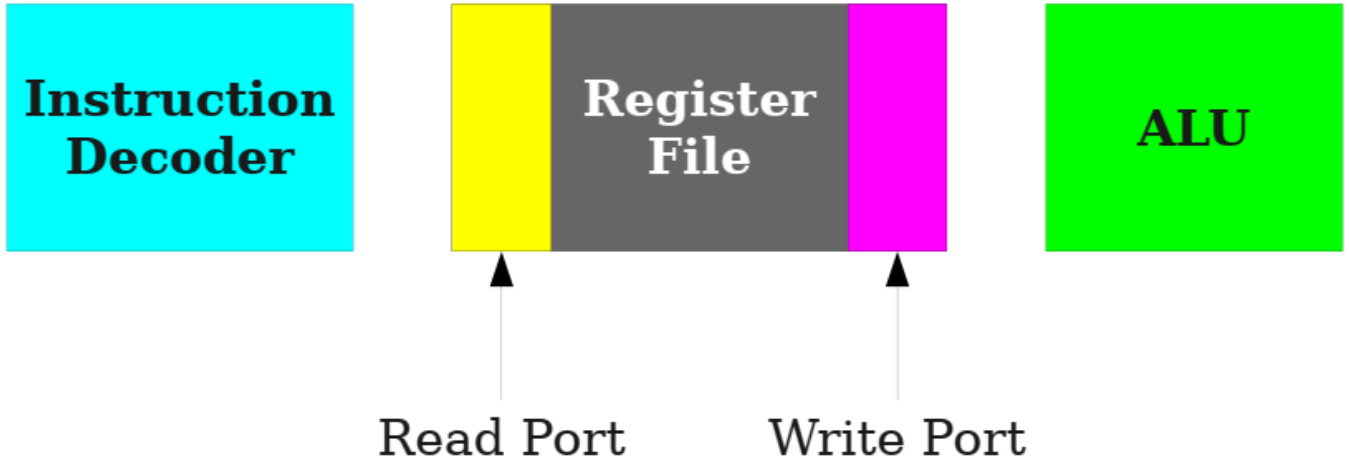
复杂的流水线并行



ID	RR	ALU	RW

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

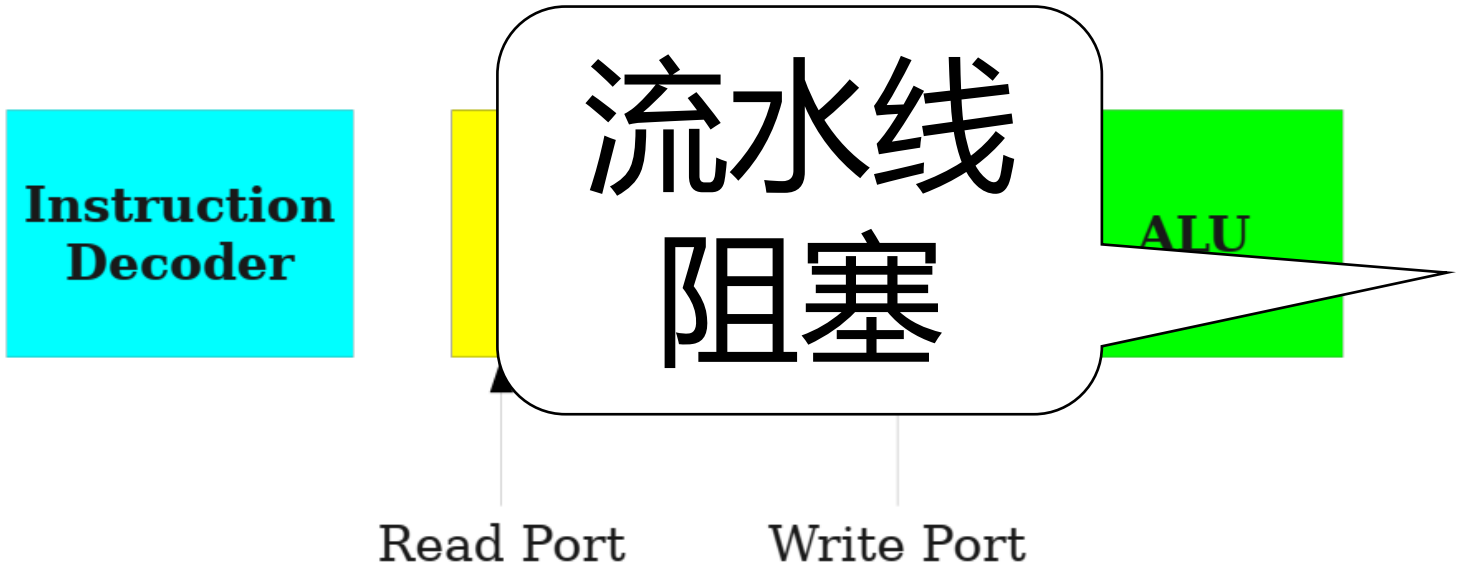
复杂的流水线并行

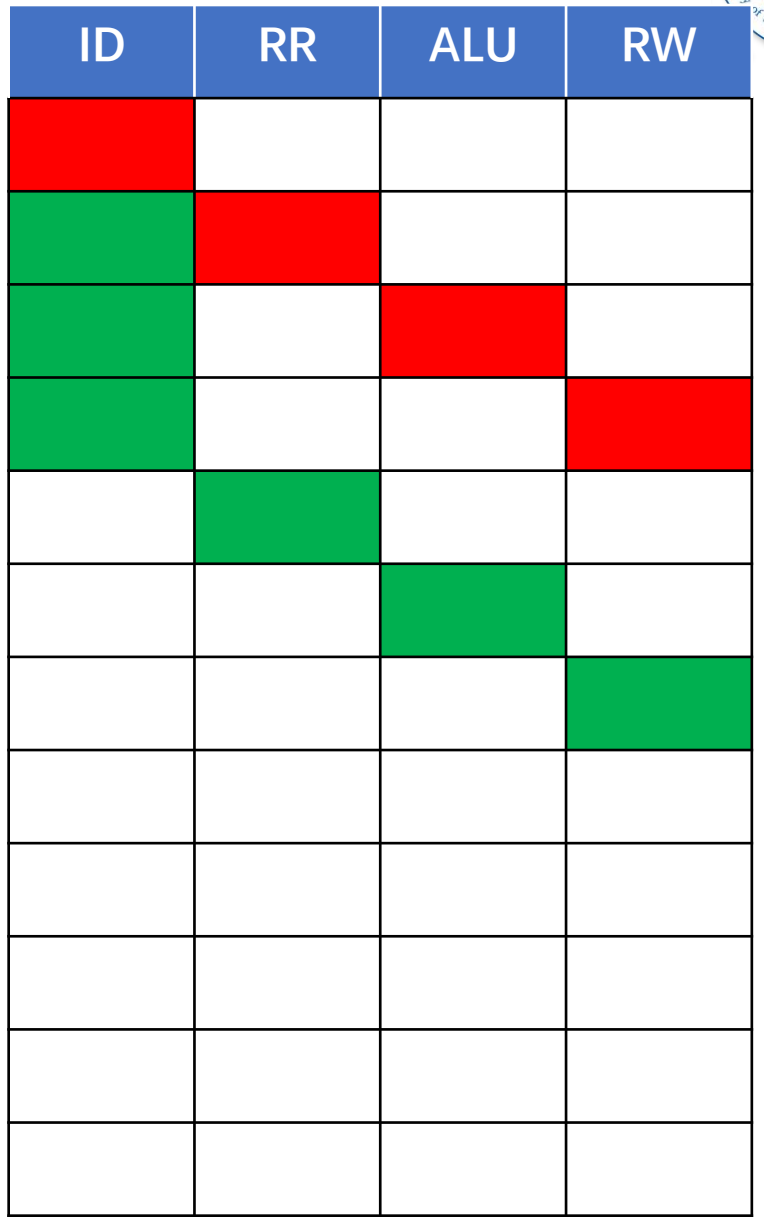



```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

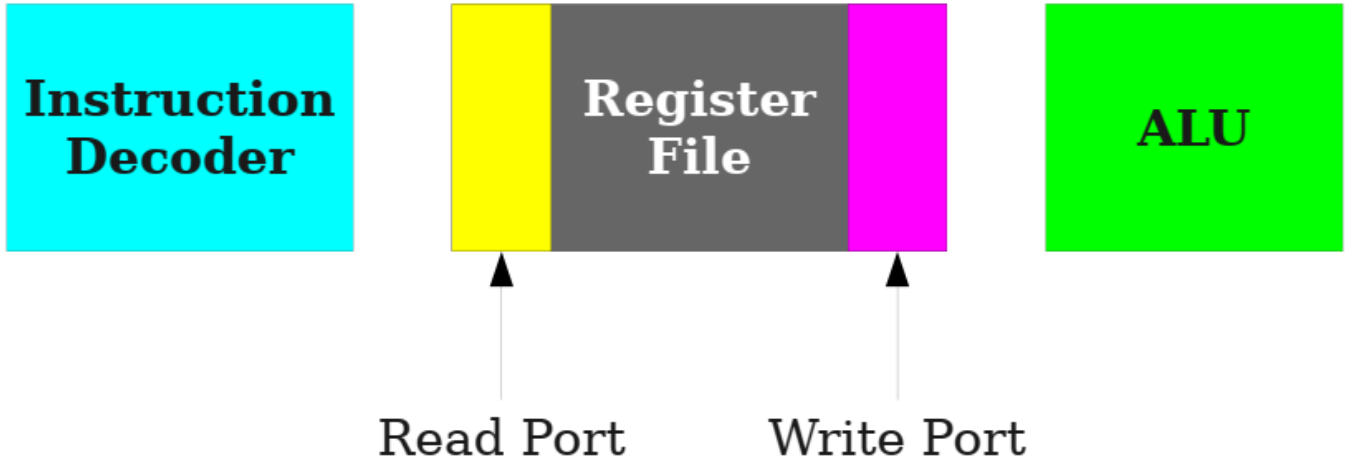
复杂的流水线并行



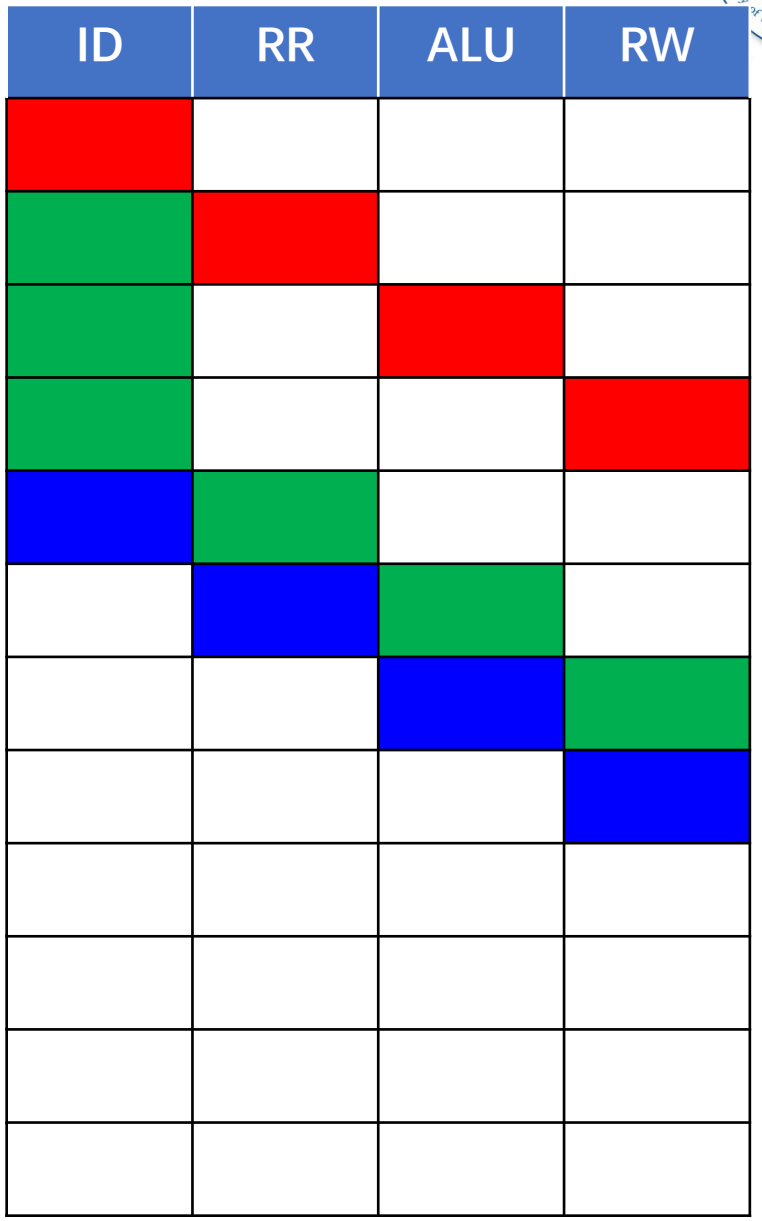



```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

复杂的流水线并行




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```



复杂的流水线并行

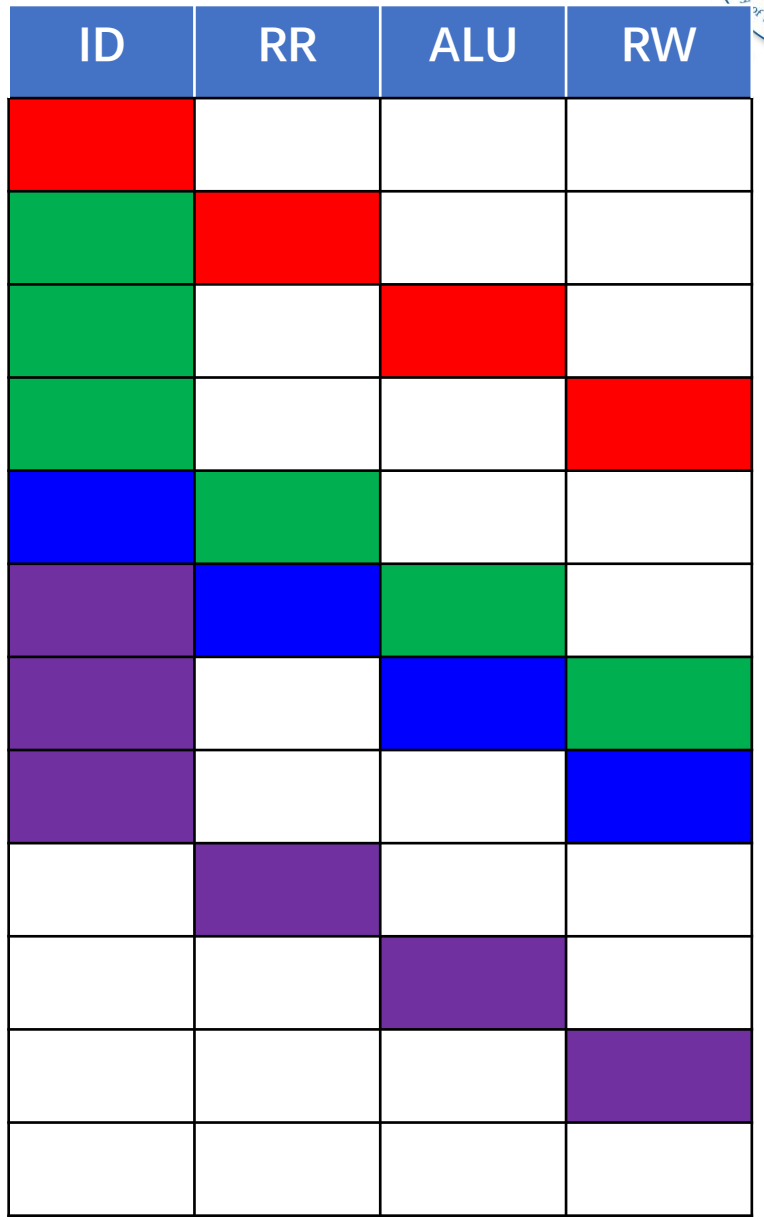
流水线阻塞

add $t2, $t0, $t + $t1

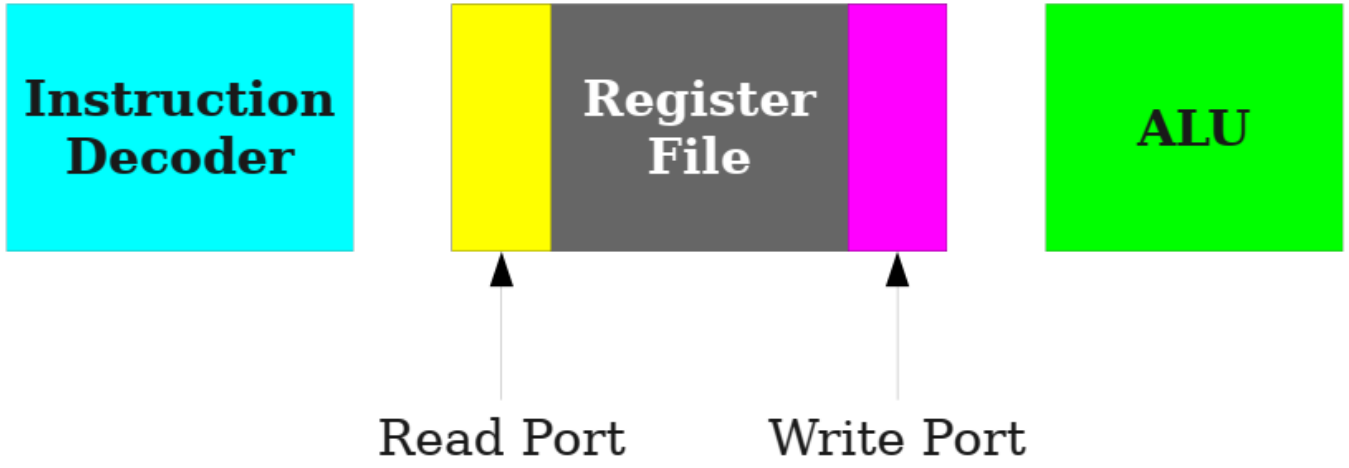
add $t4, $t3, $t + $t2

add $t7, $t5, $t6 # $t7 = $t5 + $t6

add $t0, $t0, $t7 # $t0 = $t0 + $t7



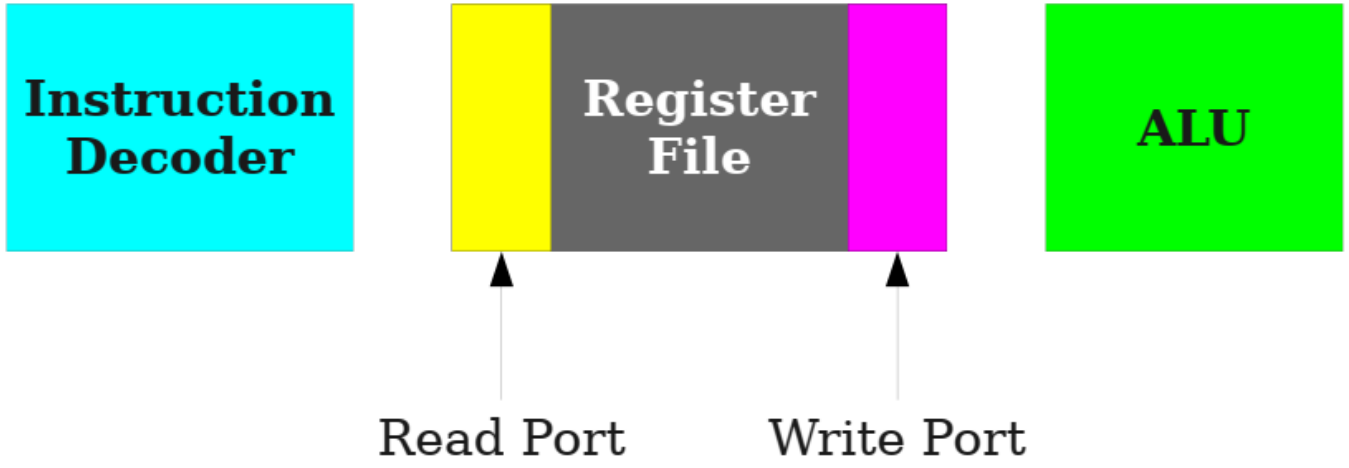
可能的优化




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

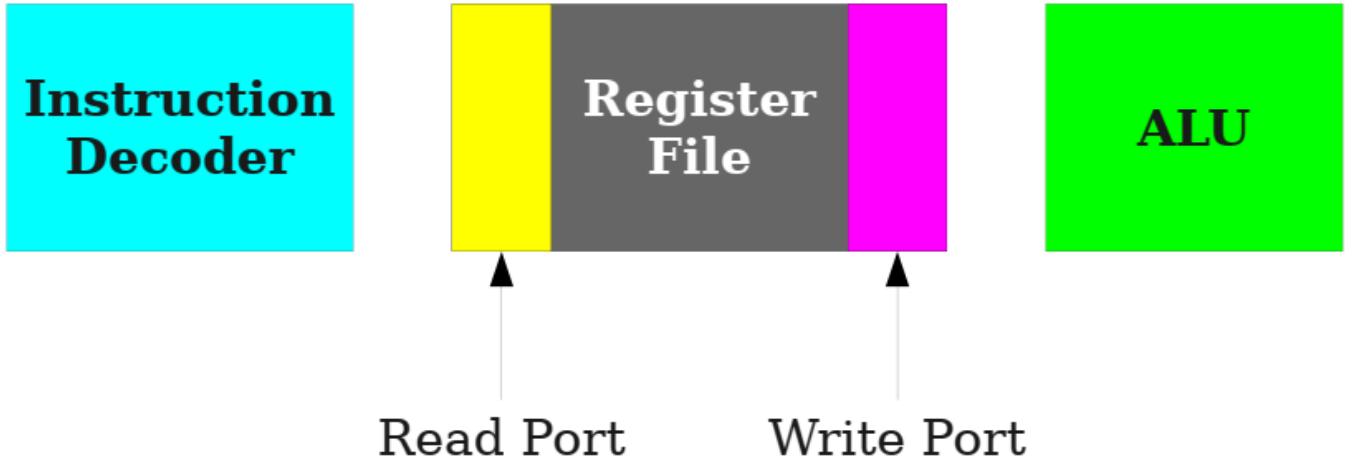
可能的优化




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

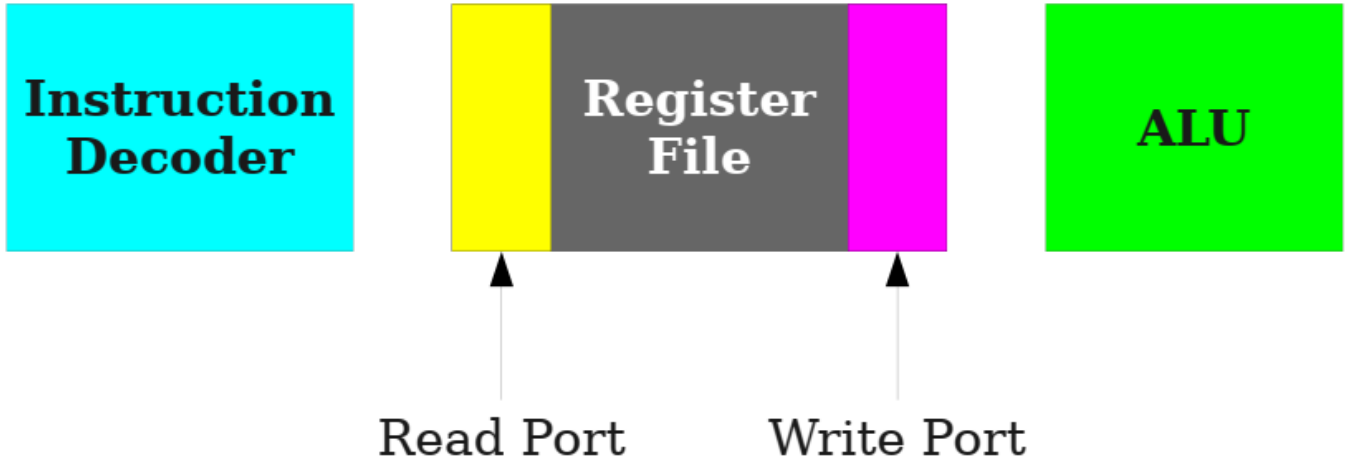
可能的优化




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

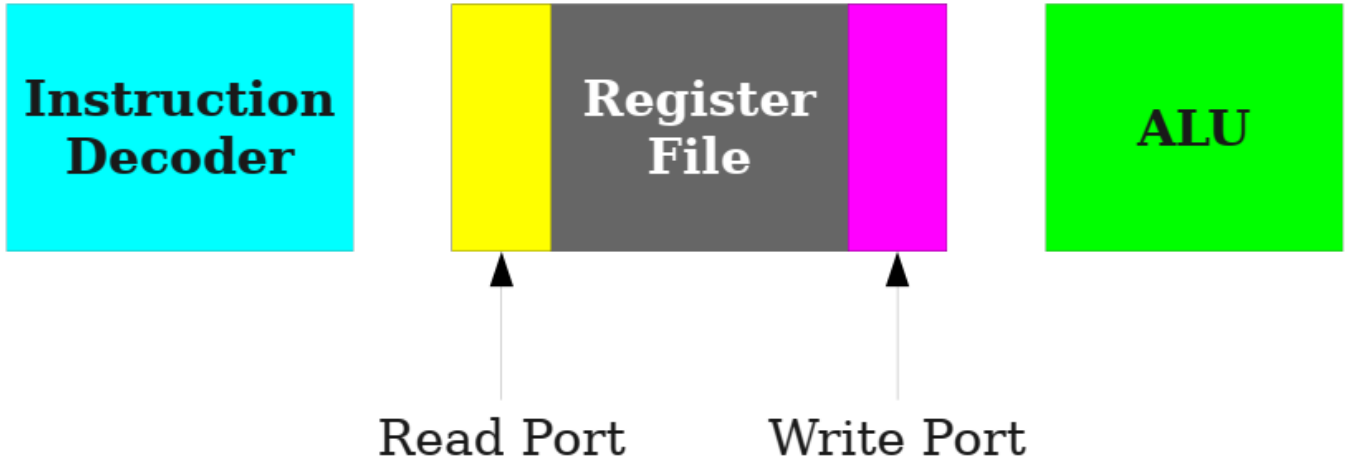
可能的优化



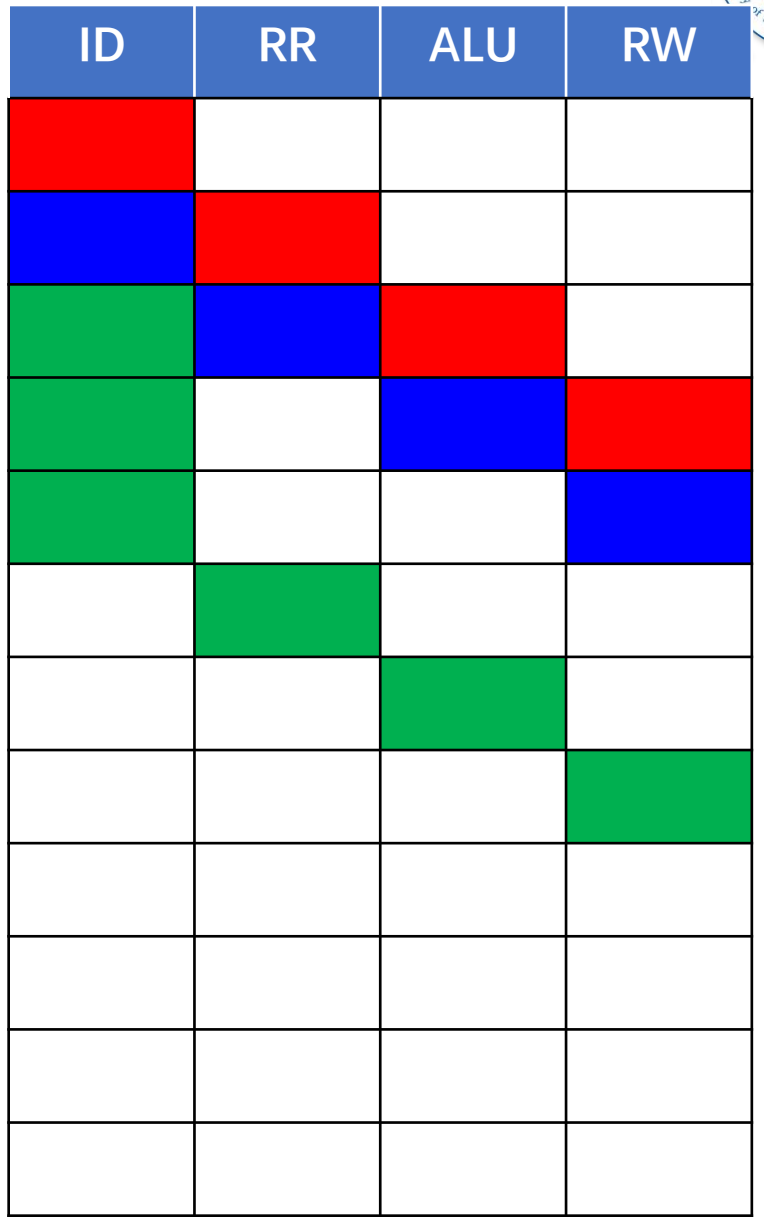

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

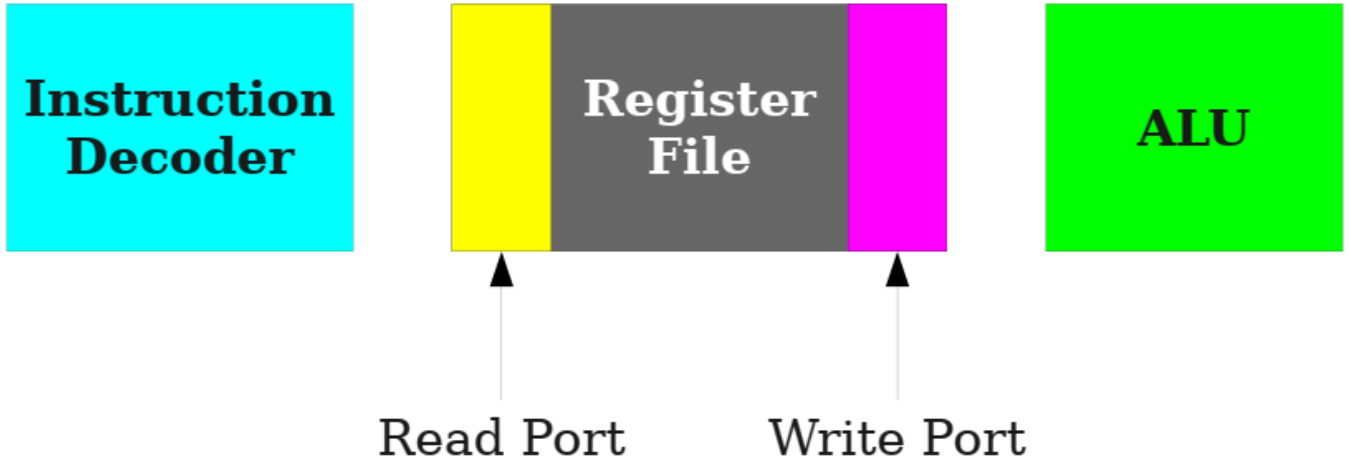
可能的优化




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```



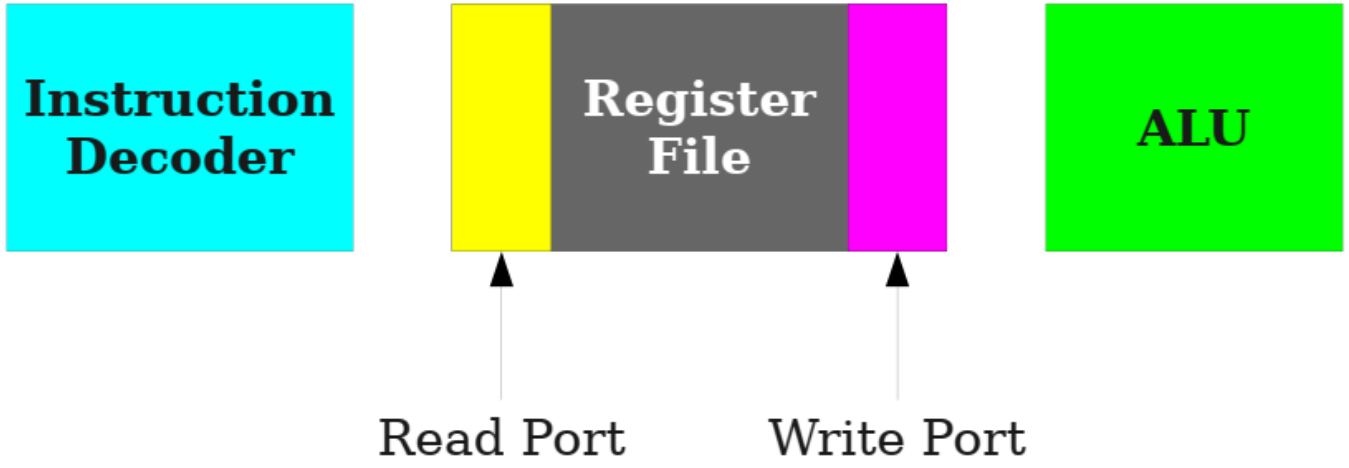
可能的优化




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

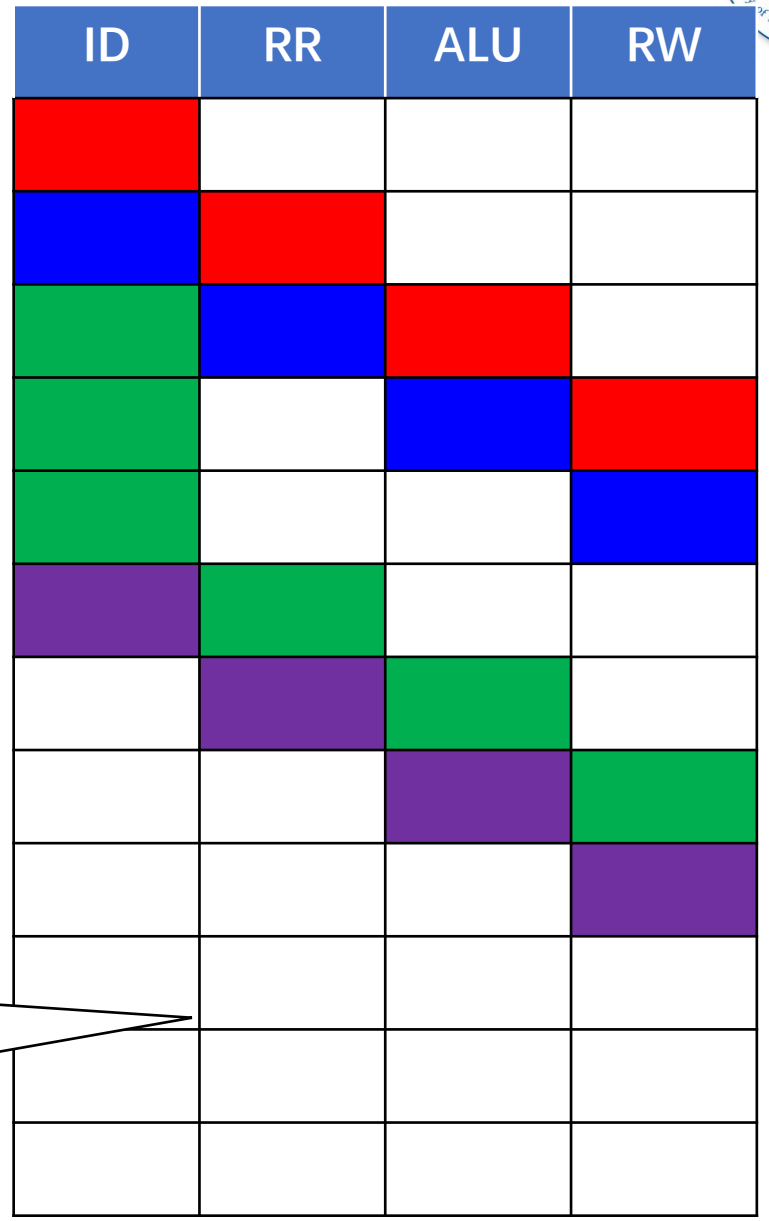
ID	RR	ALU	RW

可能的优化

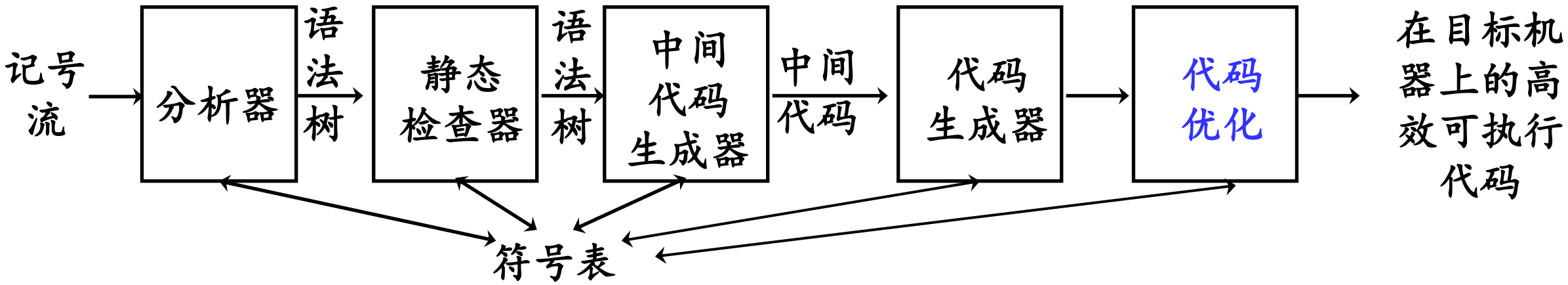



```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t       6
add $t4, $t3, $t       2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

节省两个时钟周期



本节提纲



- 现代处理器架构
- 流水线并行的例子
- **指令调度与数据依赖分析**
- **数据依赖指导下的指令调度**
- **科技前沿——大模型的流水并行训练**

指令调度(Instruction scheduling)

优化的起源

- 由于处理器流水线并行机制，指令的执行顺序对性能有较大影响。

指令调度

- 重排机器代码指令，旨在最小化执行特定指令序列所需的时钟周期数。
- 任意编译器均支持指令调度。

理论和技术挑战

- 然而，在处理器流水线上执行的顺序代码内含着一些指令之间的依赖关系，在指令调度期间执行的任何转换都必须保留这些依赖关系，以维护被调度代码的逻辑。

三种数据依赖关系

read-after-write，RAW

- 当一条指令读取另一条指令写入的结果时，会产生写后读相关性，读指令必须在写指令一定时钟周期后再读取而不会产生阻塞。

```
x = ...
... = x
```

write-after-read，WAR

- 当一条指令写在另一条指令的操作数上时，会产生反向依赖或称读后写依赖。 读指令必须在写指令之前经过适当的周期数才能安全读取，而不阻塞写指令。

```
... = x
x = ...
```

write-after-write，WAW

- 如果两条指令写入同一个目标，就会产生单个输出或写后写依赖关系

```
x = ...
x = ...
```

分析数据依赖关系

t0 = t1 + t2

t1 = t0 + t1

t3 = t2 + t4

t0 = t1 + t2

t5 = t3 + t4

t6 = t2 + t7

分析数据依赖关系

t0 = t1 + t2

t1 = t0 + t1

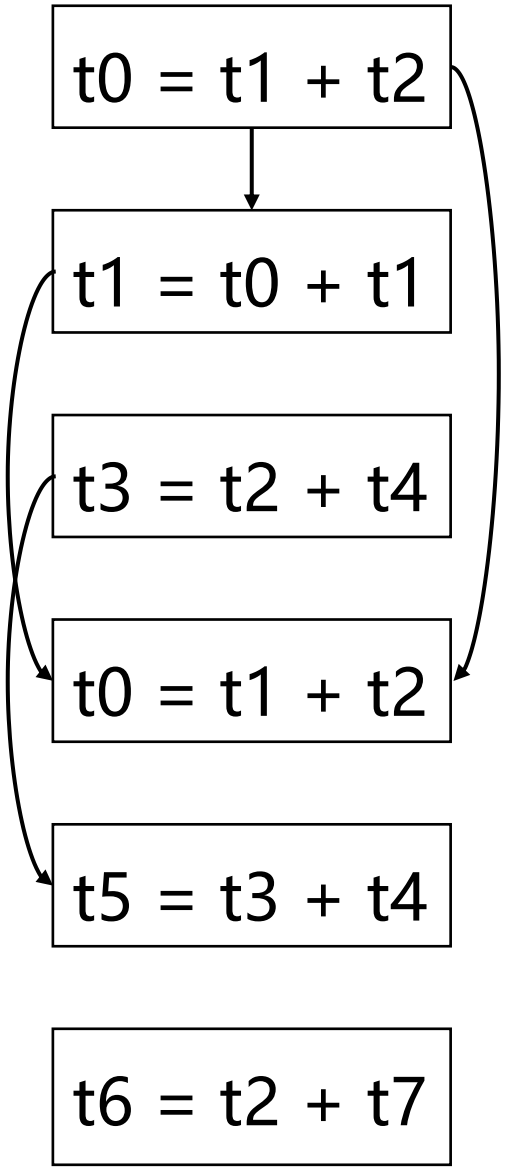
t3 = t2 + t4

t0 = t1 + t2

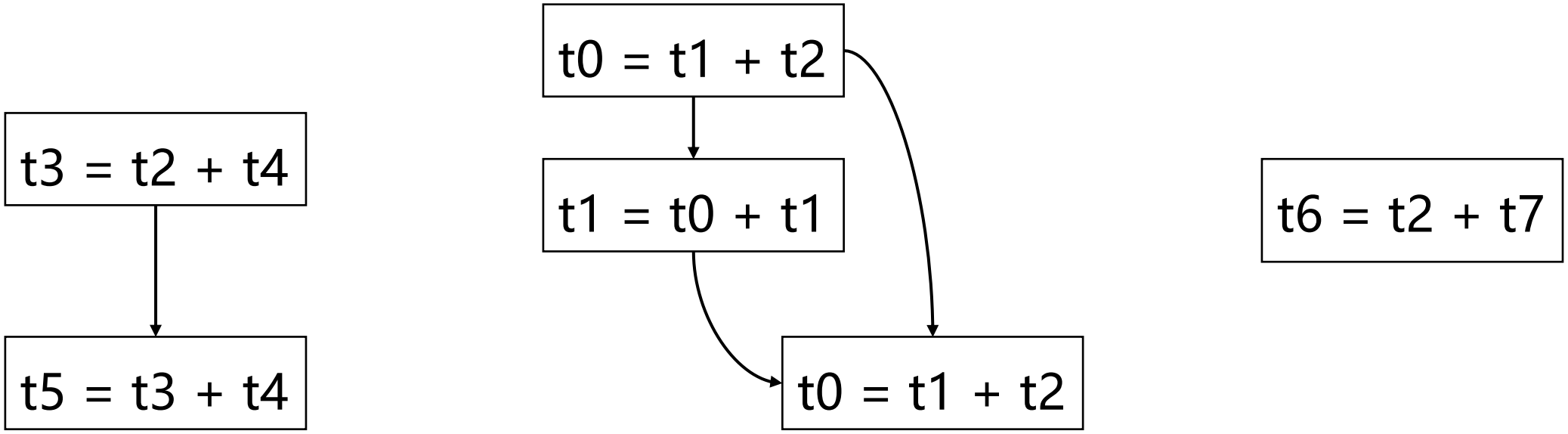
t5 = t3 + t4

t6 = t2 + t7

分析数据依赖关系



分析数据依赖关系

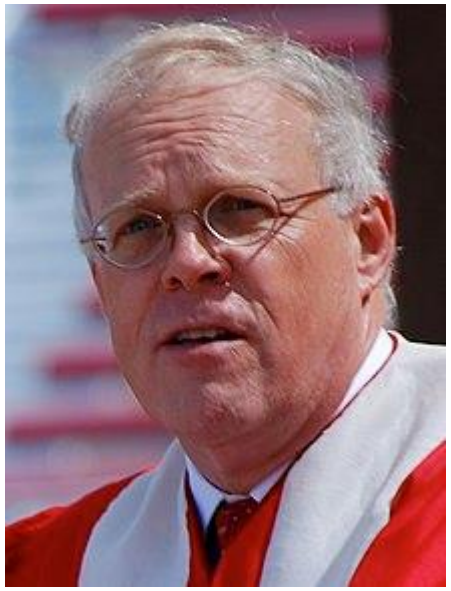


指令数据依赖图[HennessyGross，1983]

- 一个基本块的指令数据依赖图：
  - 每个节点表示单个机器指令
  - 每一条边代表了两条指令间存在数据依赖，否则就没有依赖
- 依赖图是一个有向无环图，directed acyclic graph (DAG)
  - Directed: 代表了计算的顺序
  - Acyclic: 不能存在环状依赖（why？）
- 合法的指令调度
  - 条件：一条指令不能先于他的祖先节点执行
- 实现方法
  - 对依赖图进行拓扑排序(topological sort)

延伸阅读

- John L. Hennessy and Thomas Gross. 1983. **Postpass Code Optimization of Pipeline Constraints**. ACM Trans. Program. Lang. Syst. 5, 3 (July 1983), 422–448. https://doi.org/10.1145/2166.357217

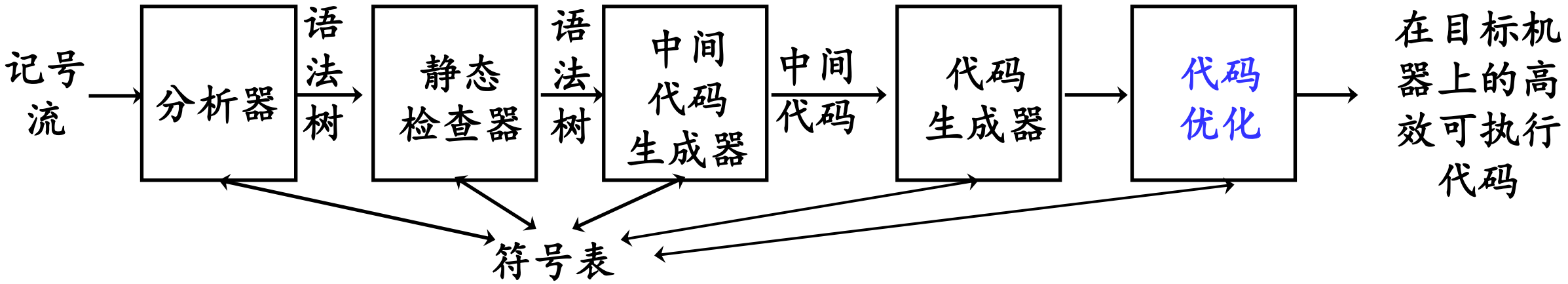


John L. Hennessy

David Patterson

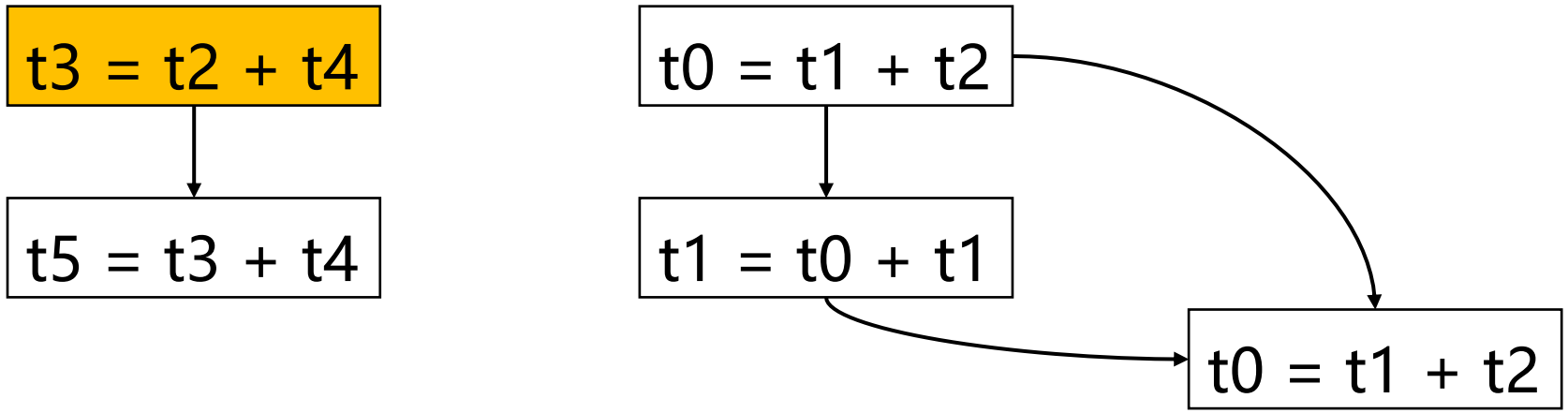
- 2017年，Hennessy和Patterson共同获得图灵奖。
- 获奖演说：
  - A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development

本节提纲

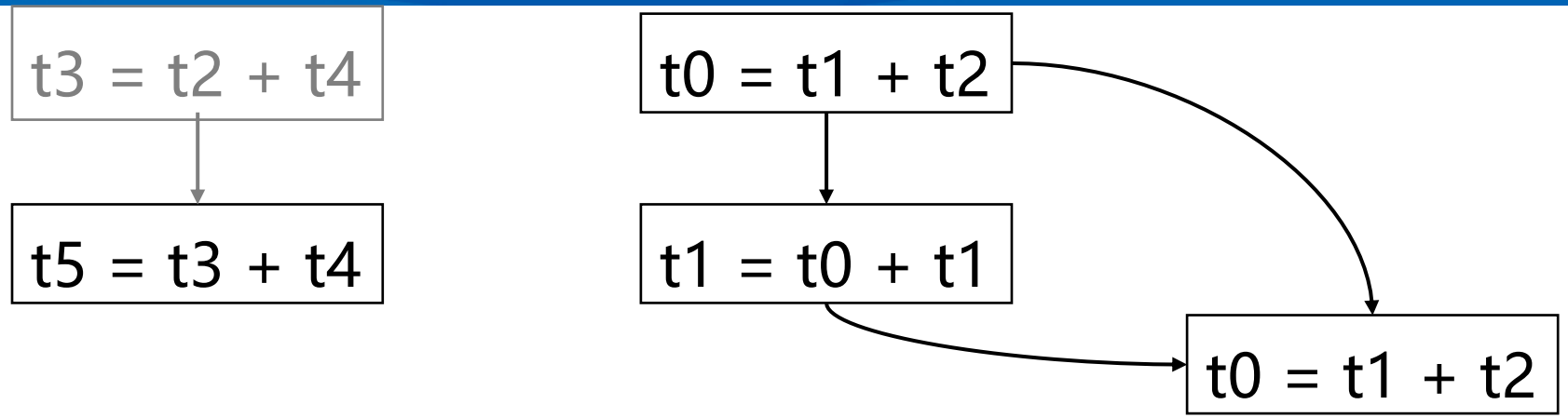


- 现代处理器架构
- 流水线并行的例子
- 指令调度与数据依赖分析
- **数据依赖指导下的指令调度**
- **科技前沿——大模型的流水并行训练**

数据依赖指导下的指令调度

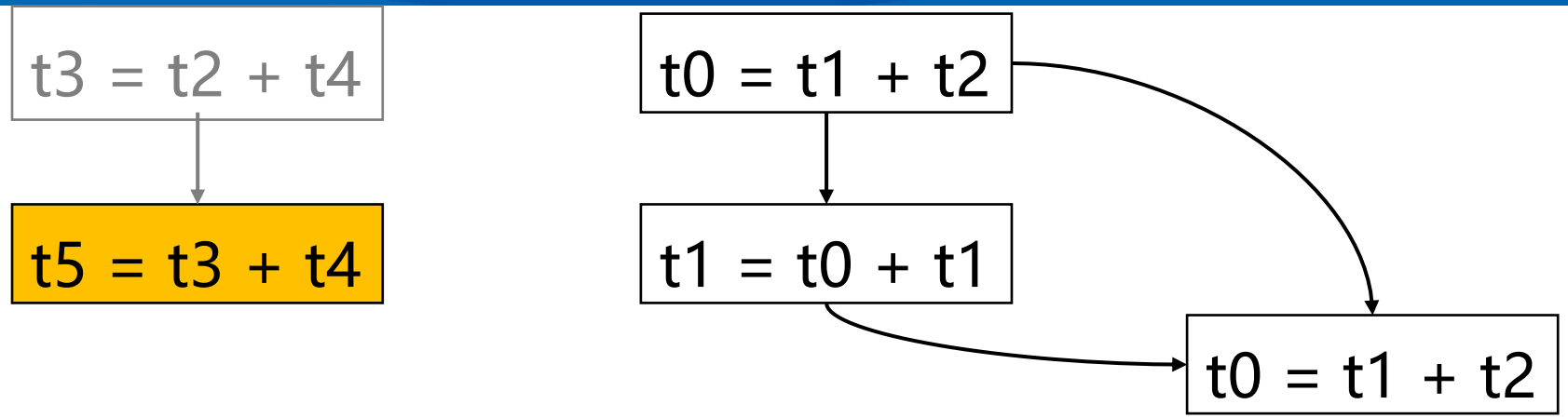


数据依赖指导下的指令调度



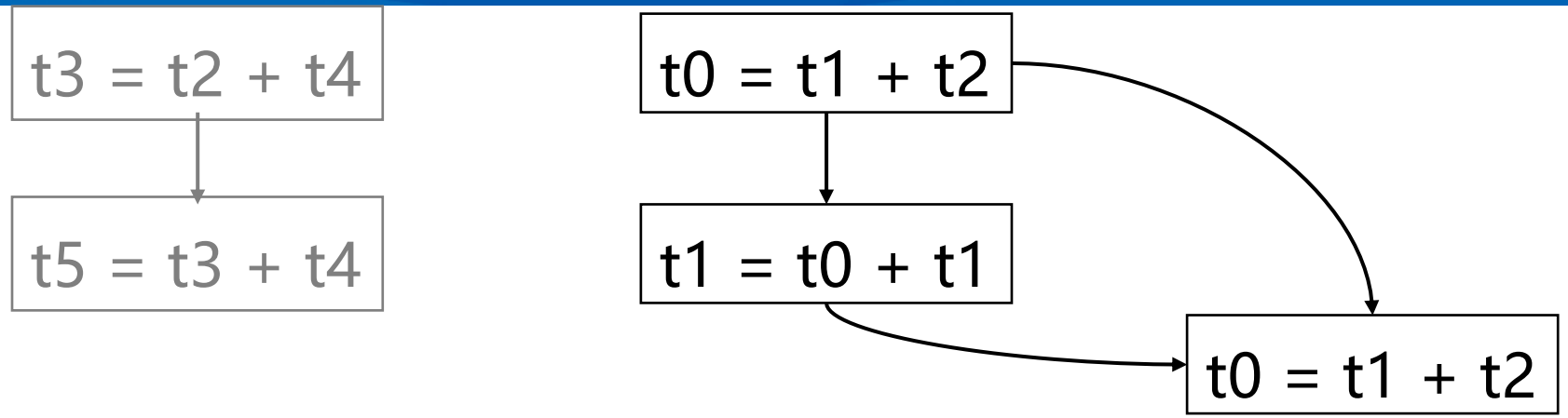
t3 = t2 + t4

数据依赖指导下的指令调度



t3 = t2 + t4

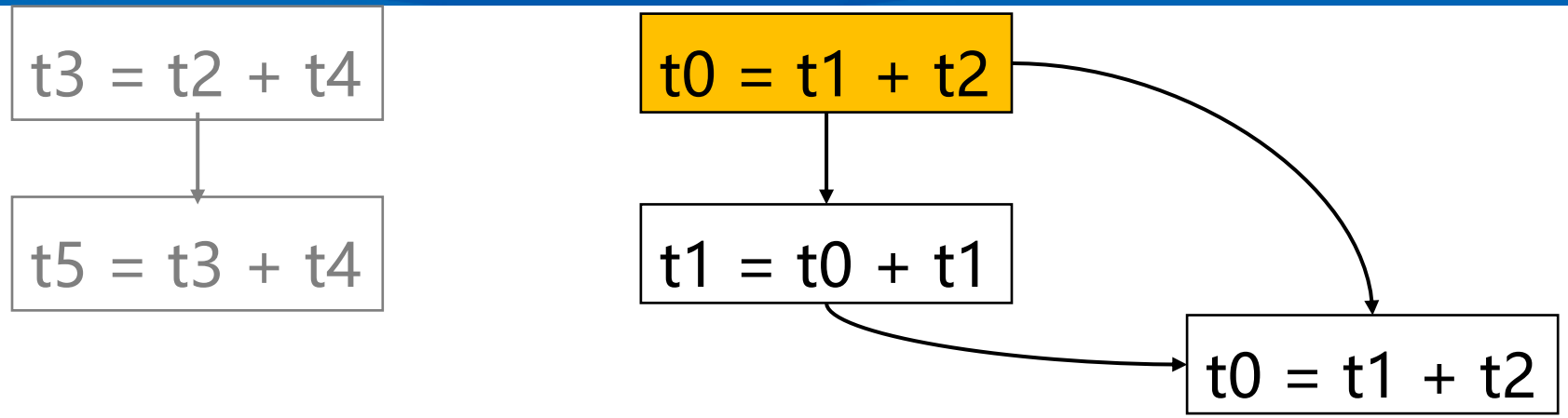
数据依赖指导下的指令调度



t3 = t2 + t4

t5 = t3 + t4

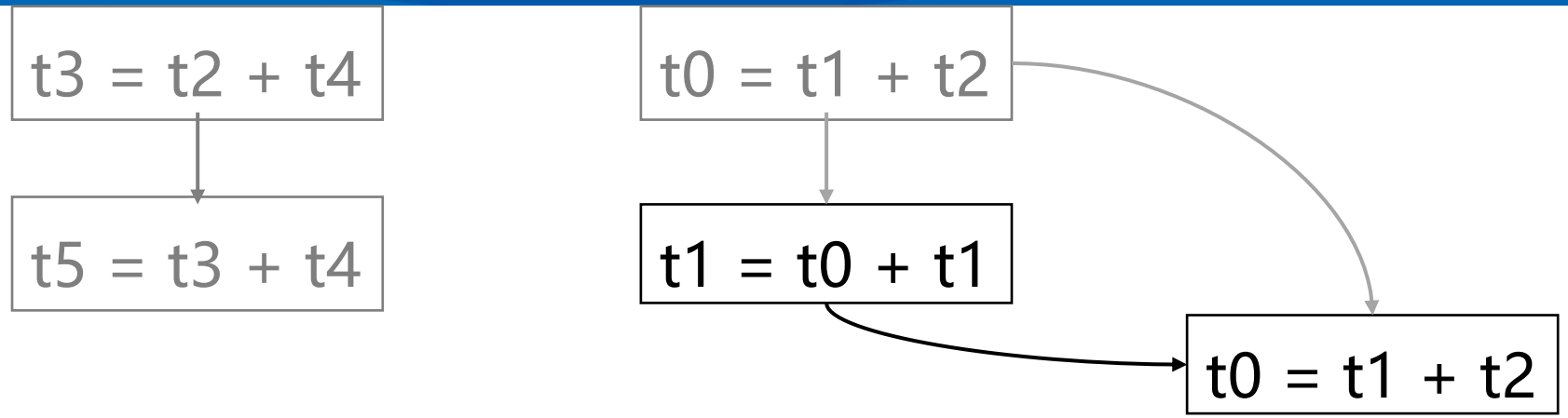
数据依赖指导下的指令调度



t3 = t2 + t4

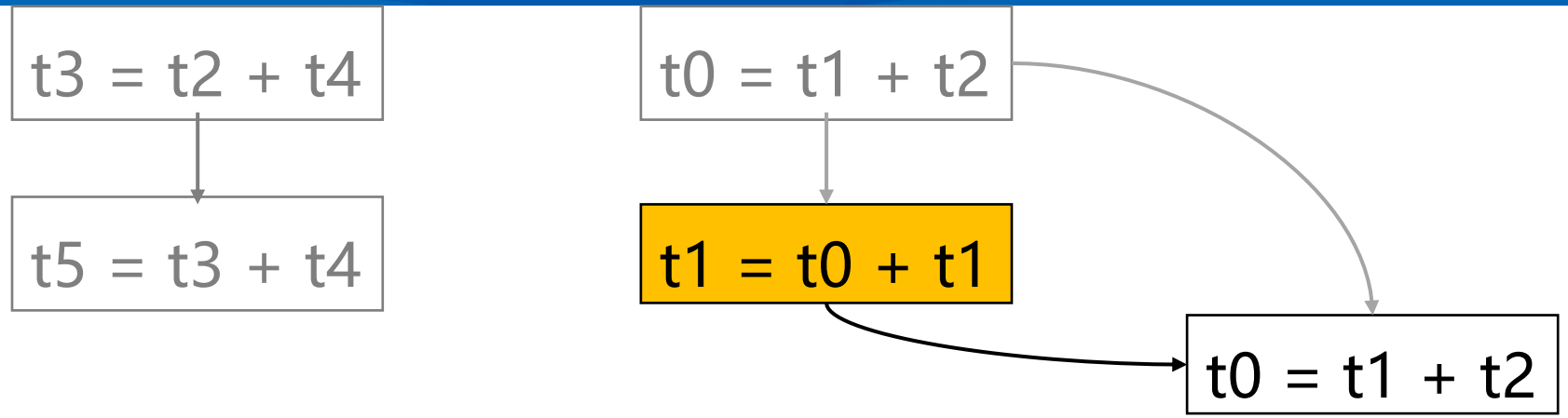
t5 = t3 + t4

数据依赖指导下的指令调度



t3 = t2 + t4
t5 = t3 + t4
t0 = t1 + t2

数据依赖指导下的指令调度

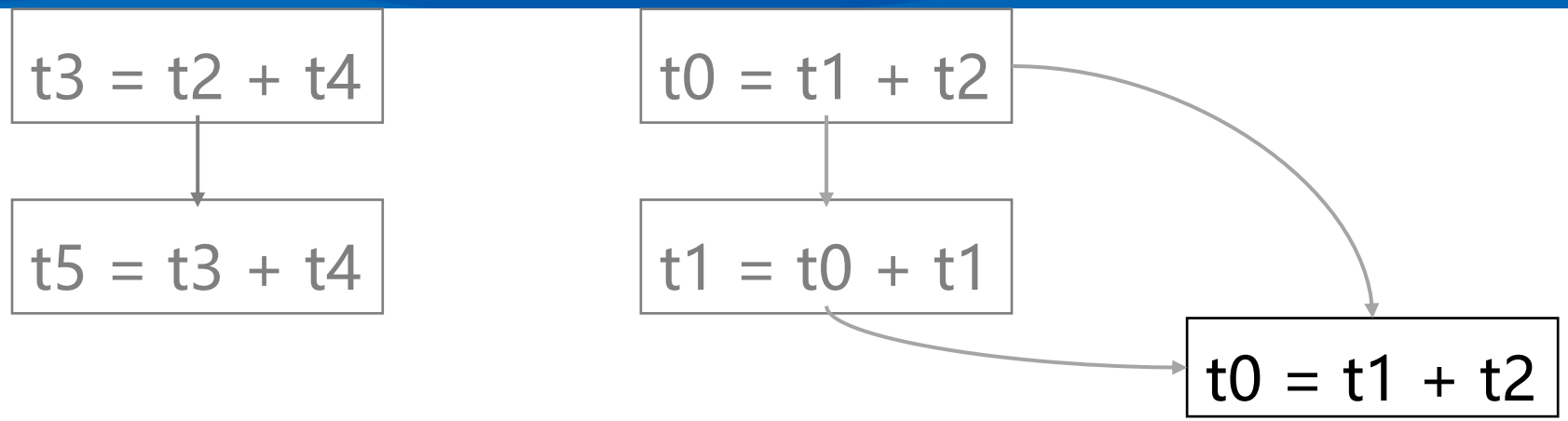


t3 = t2 + t4

t5 = t3 + t4

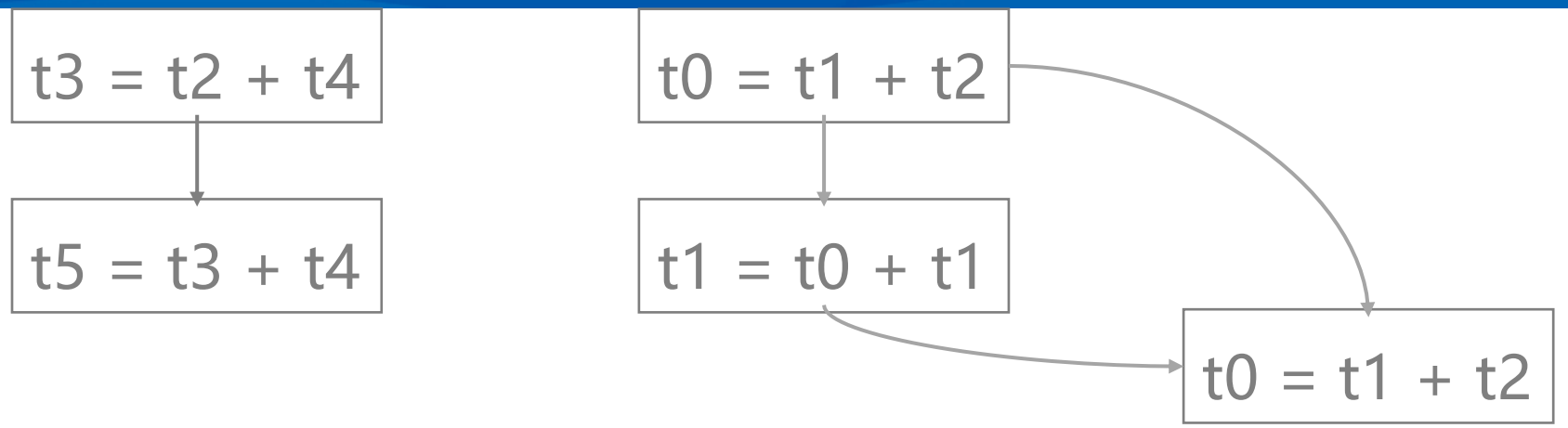
t0 = t1 + t2

数据依赖指导下的指令调度



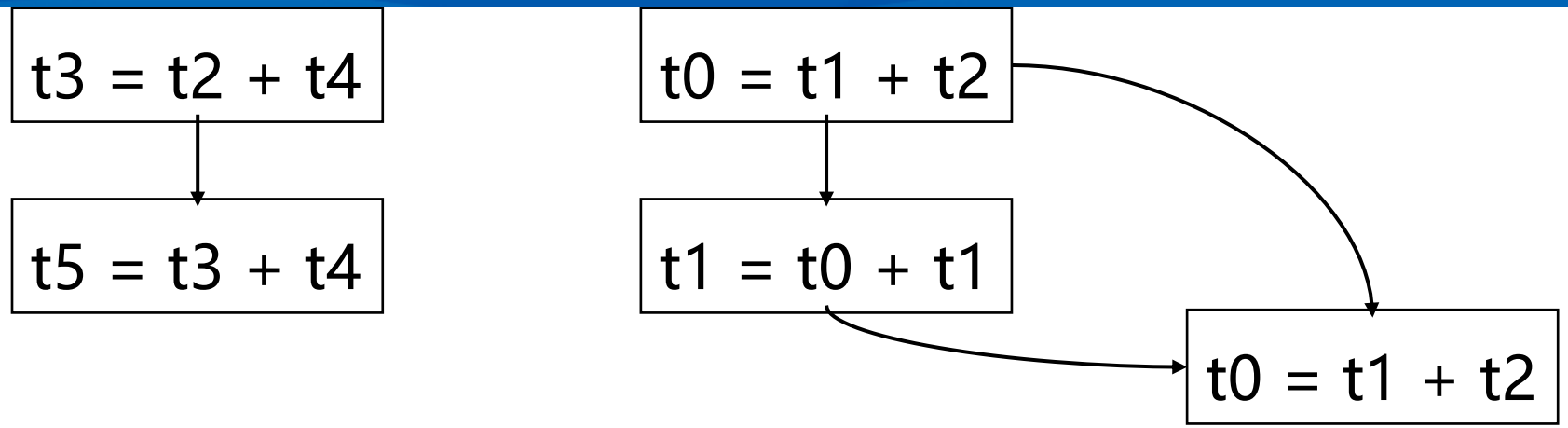
t3 = t2 + t4
t5 = t3 + t4
t0 = t1 + t2
t1 = t0 + t1

数据依赖指导下的指令调度



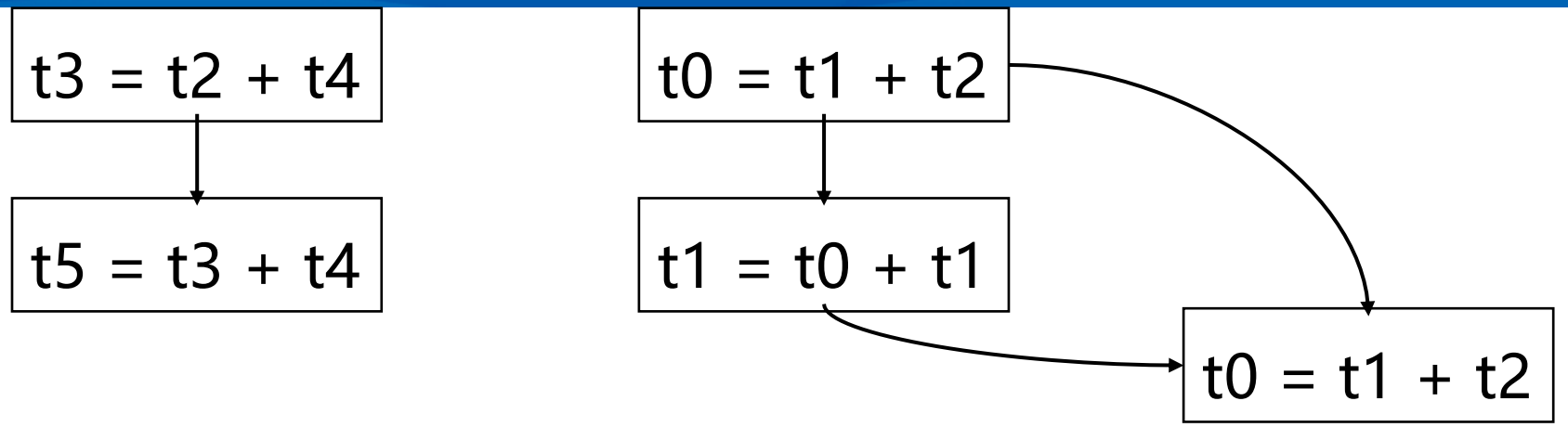
t3 = t2 + t4
t5 = t3 + t4
t0 = t1 + t2
t1 = t0 + t1
t0 = t1 + t2
t6 = t2 + t7

数据依赖指导下的指令调度



t3 = t2 + t4
t5 = t3 + t4
t0 = t1 + t2
t1 = t0 + t1
t0 = t1 + t2
t6 = t2 + t7

数据依赖指导下的指令调度



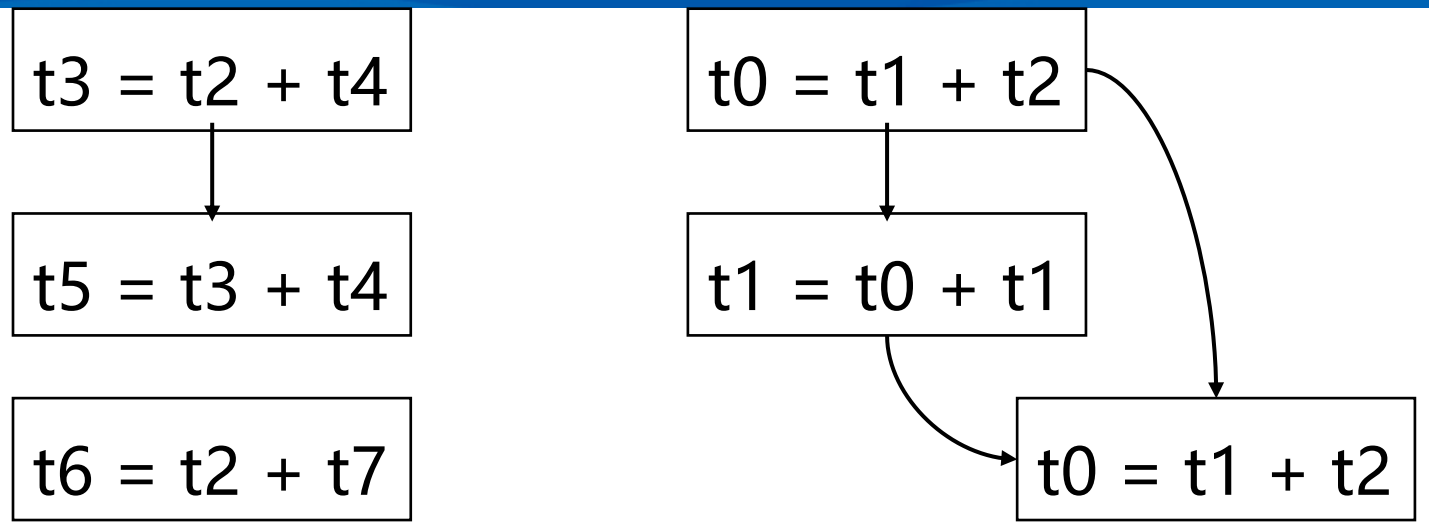
t3 = t2 + t4
t5 = t3 + t4
t0 = t1 + t2
t1 = t0 + t1
t0 = t1 + t2
t6 = t2 + t7

t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
t5 = t3 + t4
t0 = t1 + t2

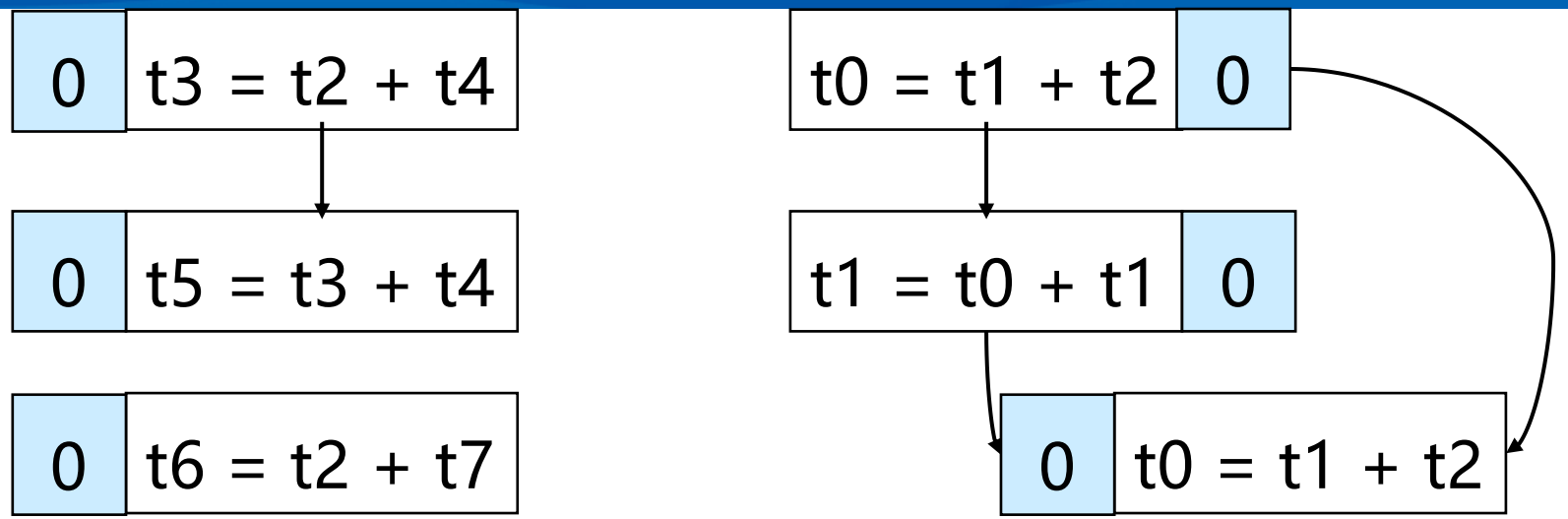
指令调度的空间

- 数据依赖图可能有许多有效的拓扑排序。
  - 该如何选择一种能与流水线完美配合的排序方式呢？
- 寻找最快的指令时间表是众所周知的 NP 难题。
  - 不要指望很快就能找到多项式时间算法！
- 在实践中使用启发式方法
  1. 将可以不受干扰地运行完成的指令安排在会造成干扰的指令之前。
  2. 将依赖关系较多的指令安排在依赖关系较少的指令之前。
  3. 对 DAG 进行加权调整！(边的权重为指令等待时间)

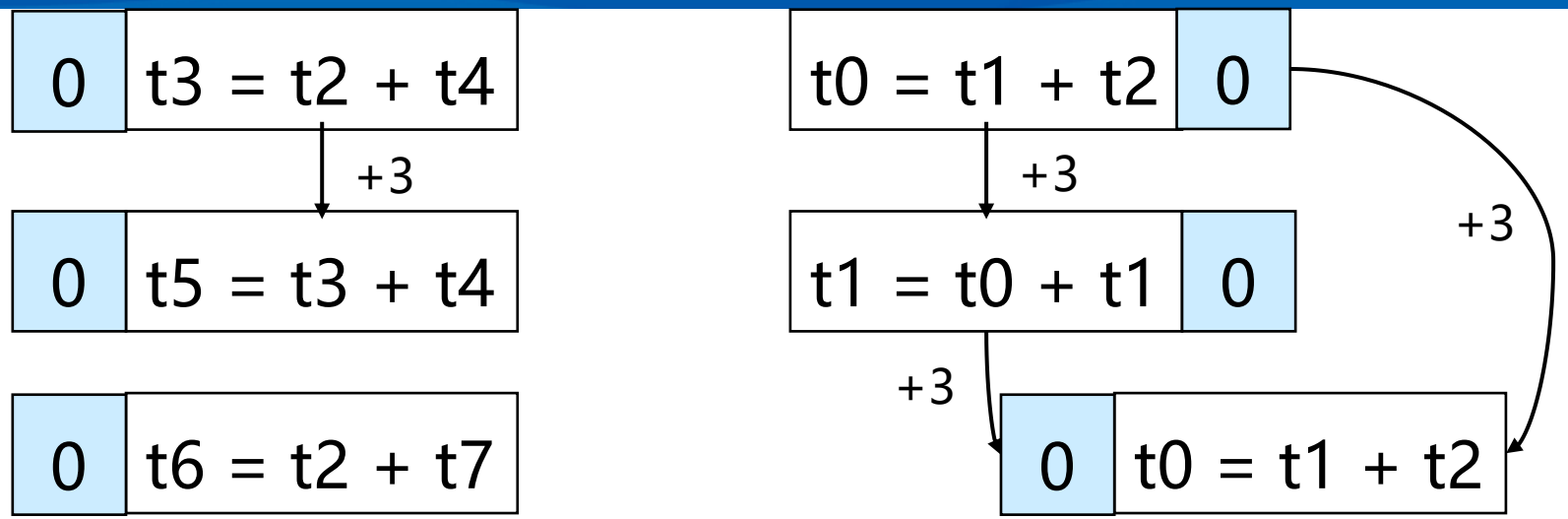
升级版的指令调度



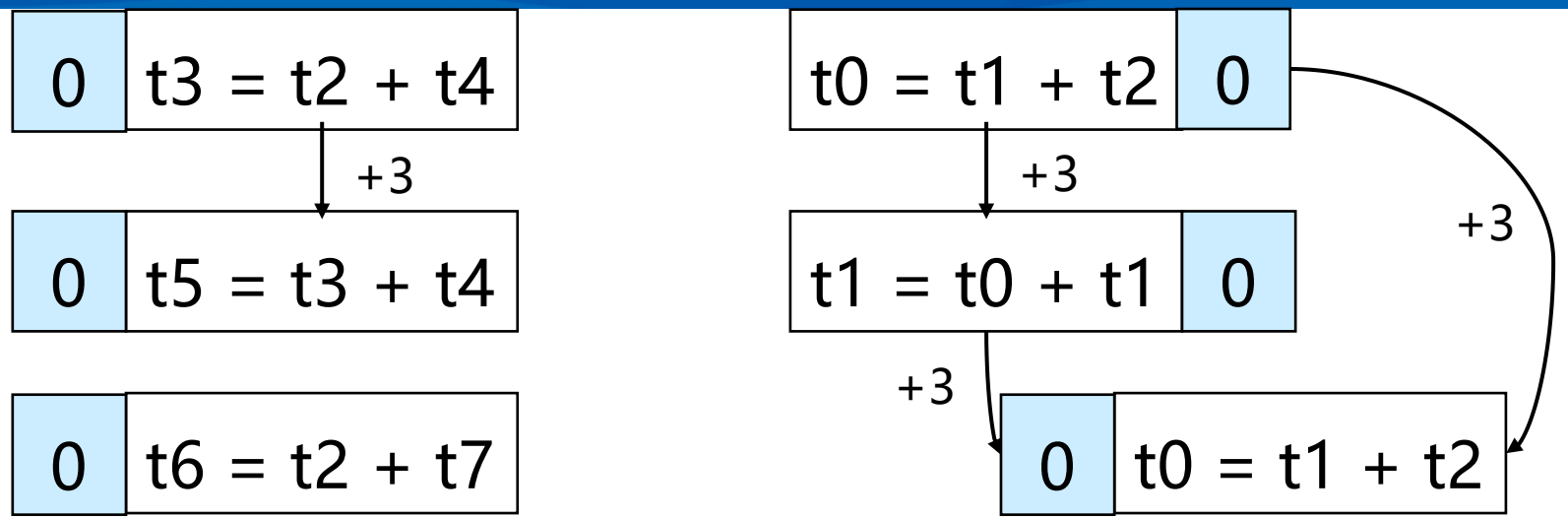
升级版的指令调度



升级版的指令调度

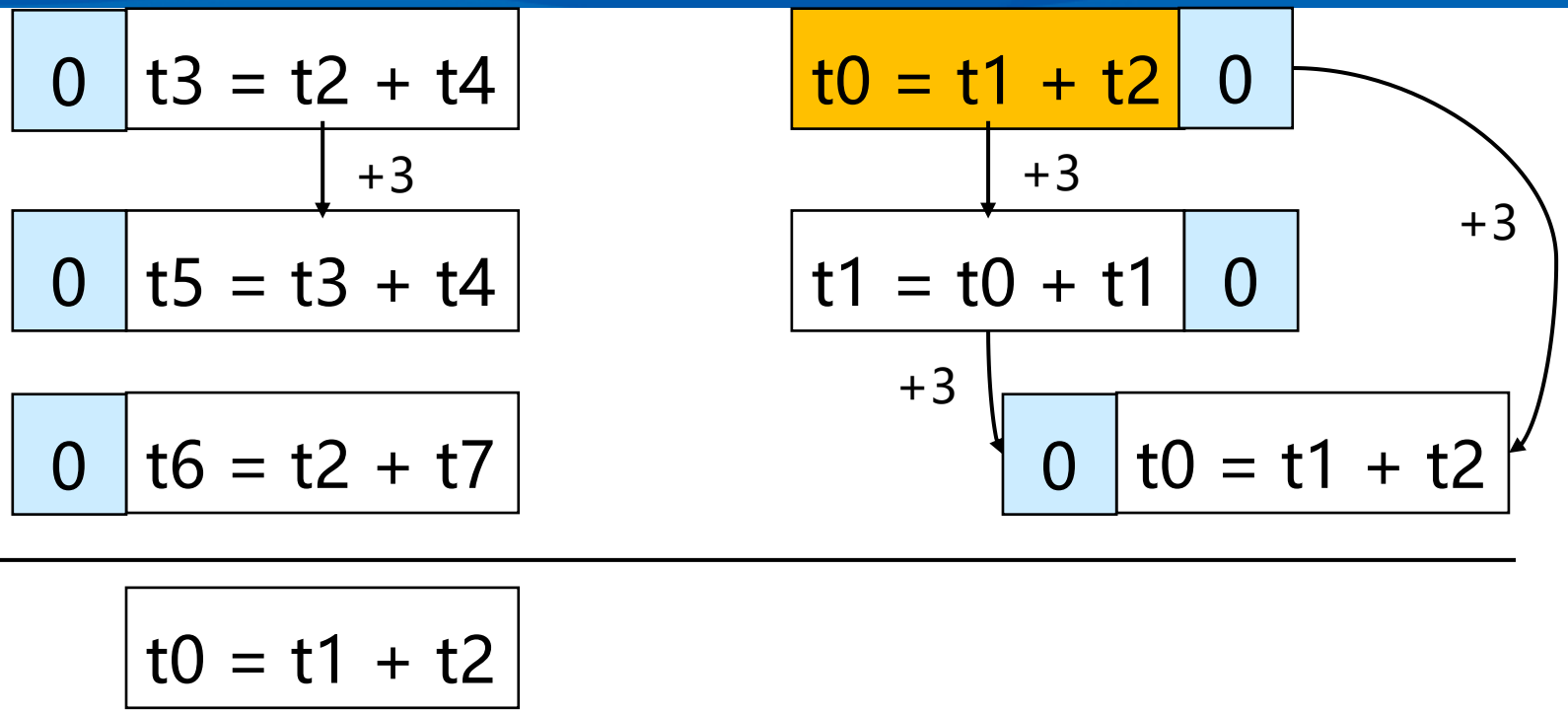


升级版的指令调度



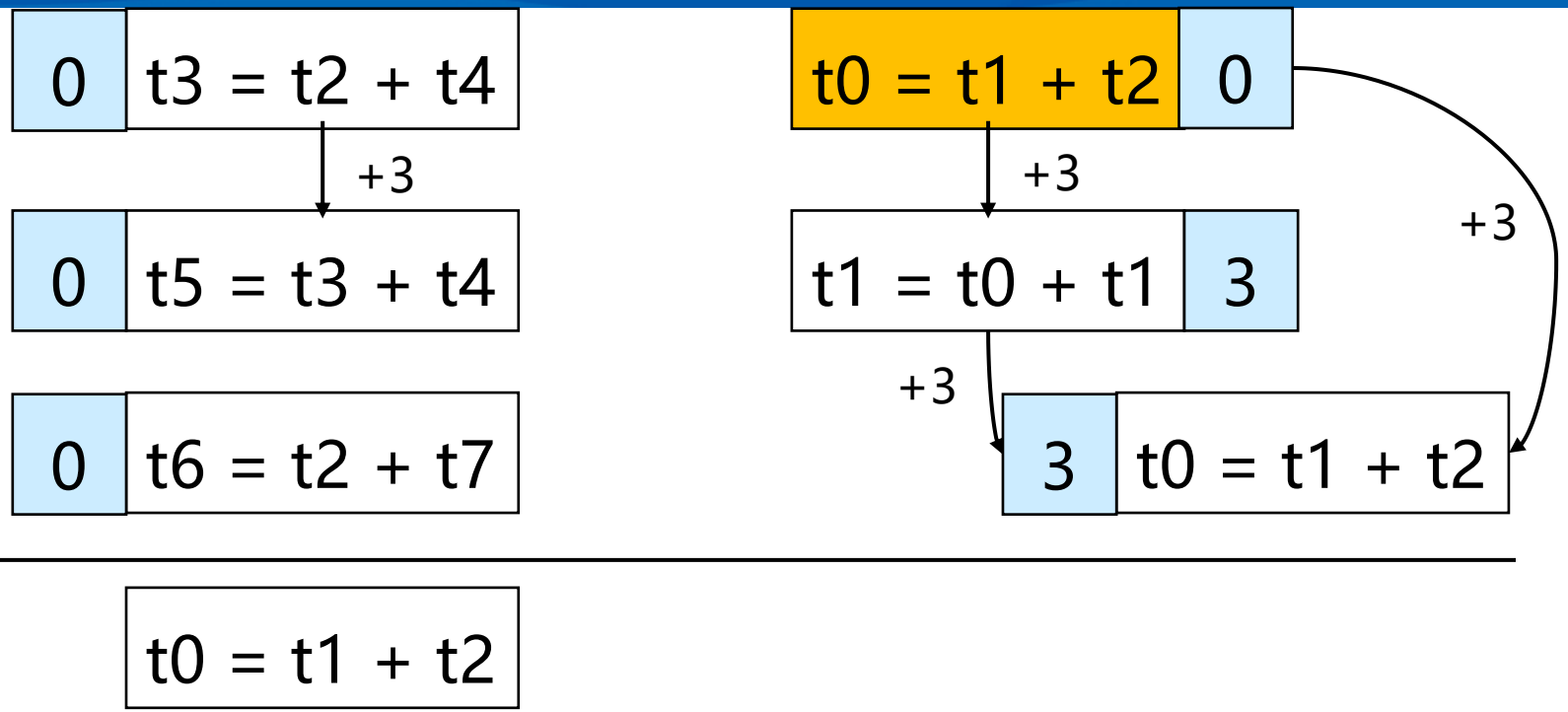
ID	RR	ALU	RW

升级版的指令调度



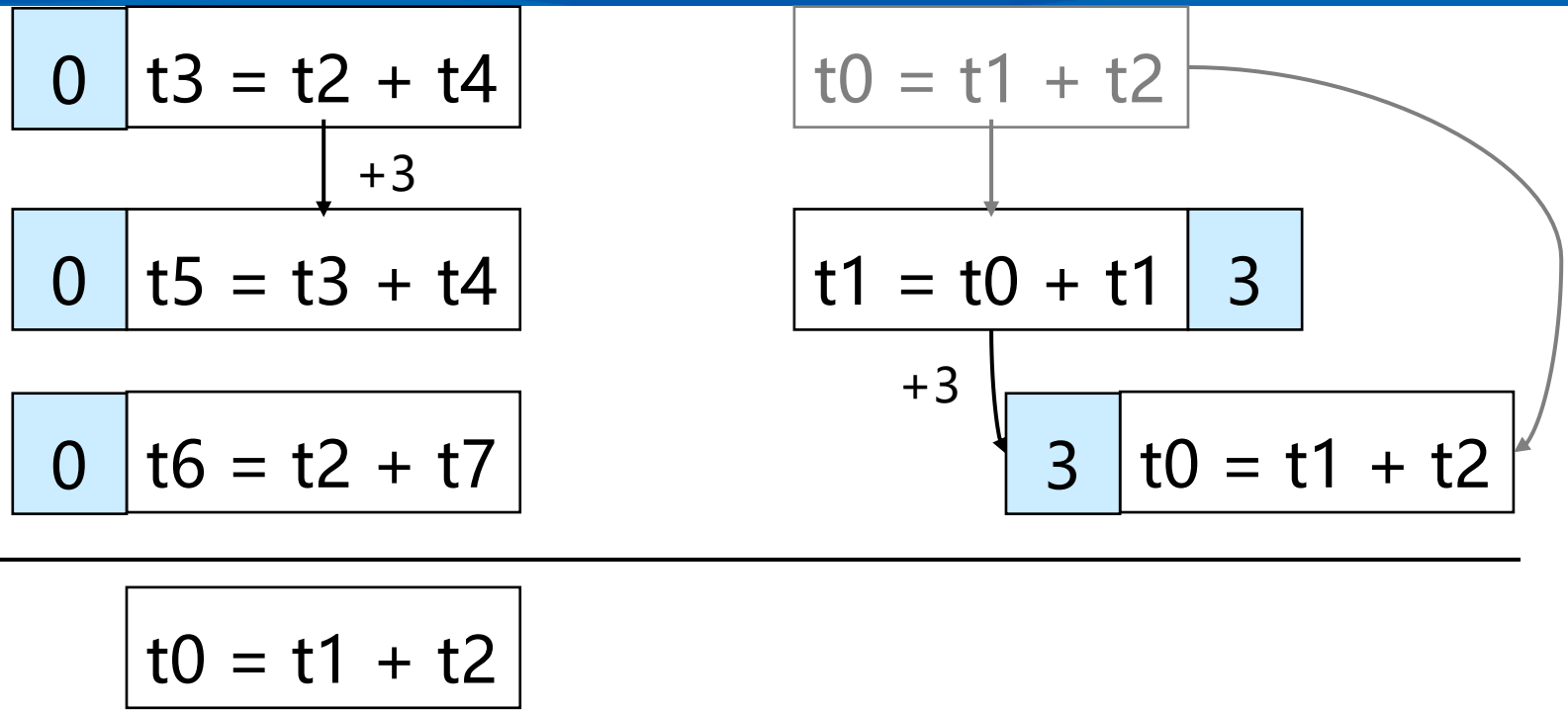
ID	RR	ALU	RW

升级版的指令调度



ID	RR	ALU	RW

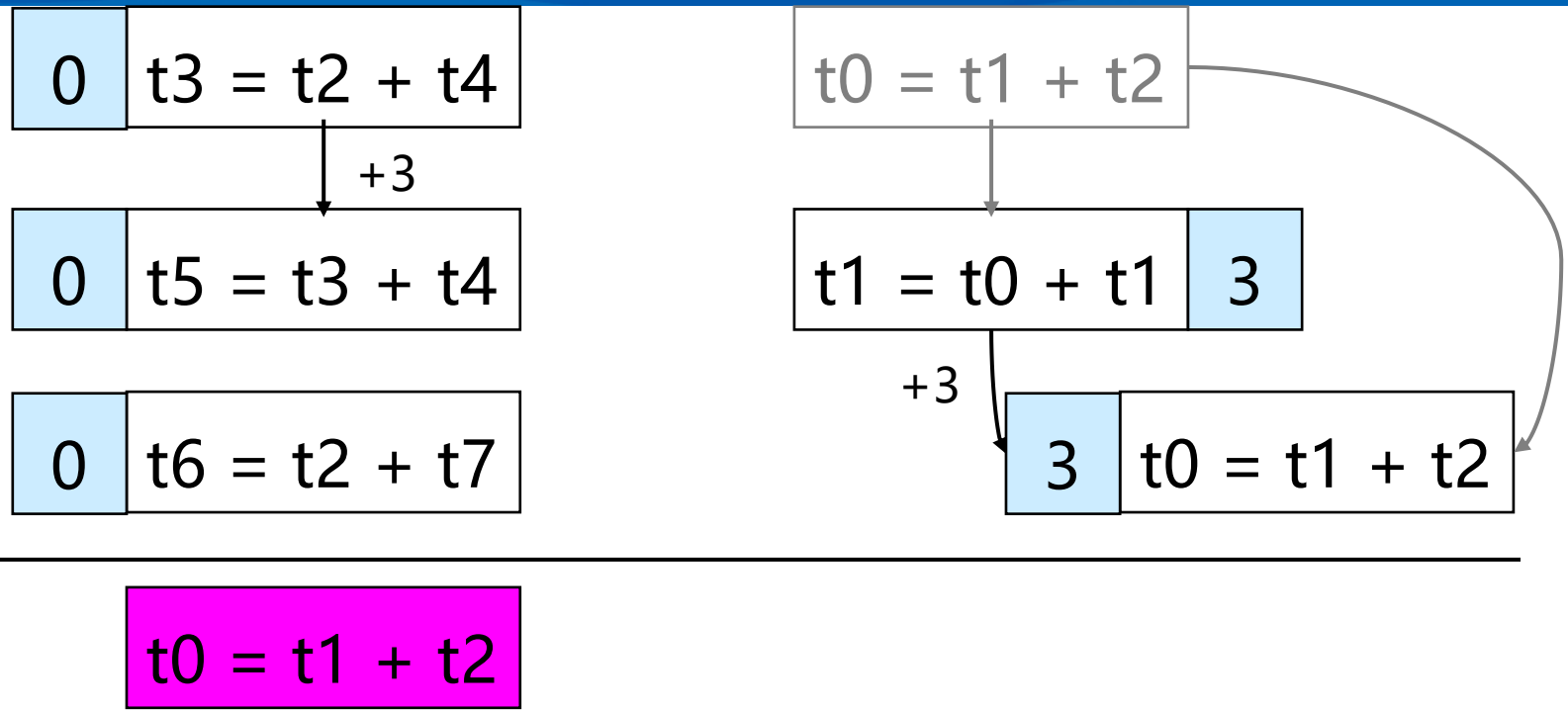
升级版的指令调度



t0 = t1 + t2

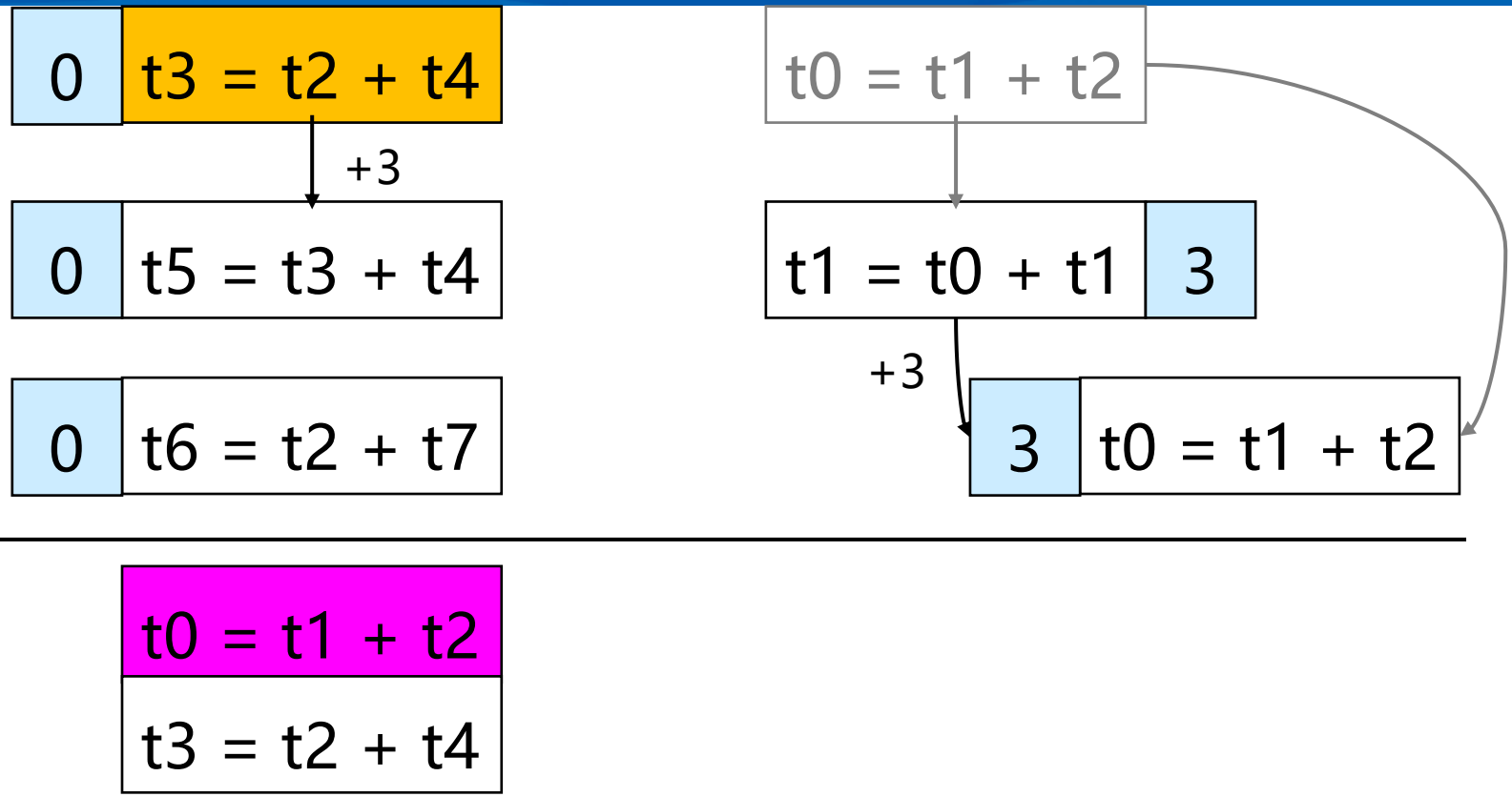
ID	RR	ALU	RW

升级版的指令调度



ID	RR	ALU	RW

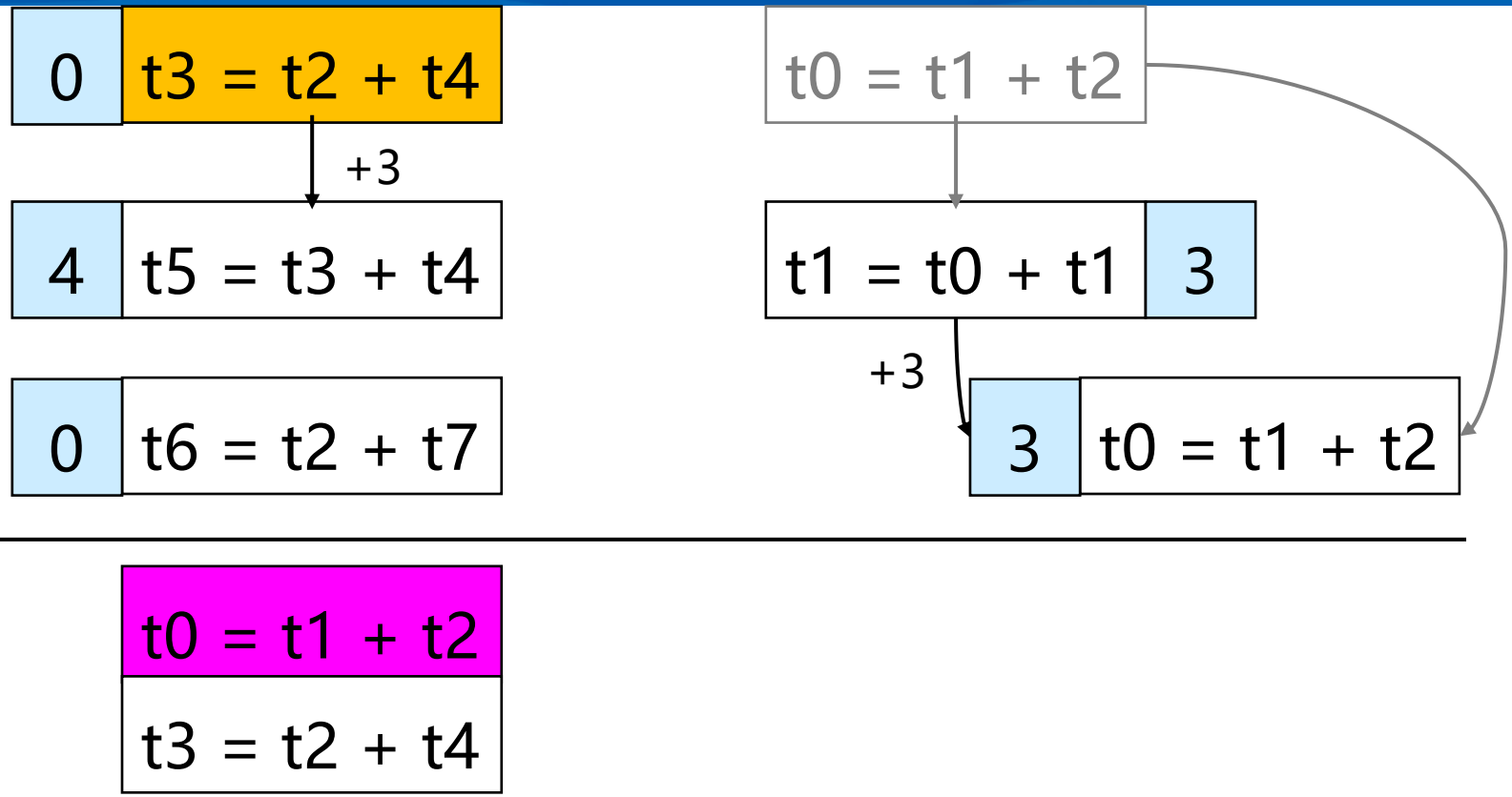
升级版的指令调度



t0 = t1 + t2
t3 = t2 + t4

ID	RR	ALU	RW

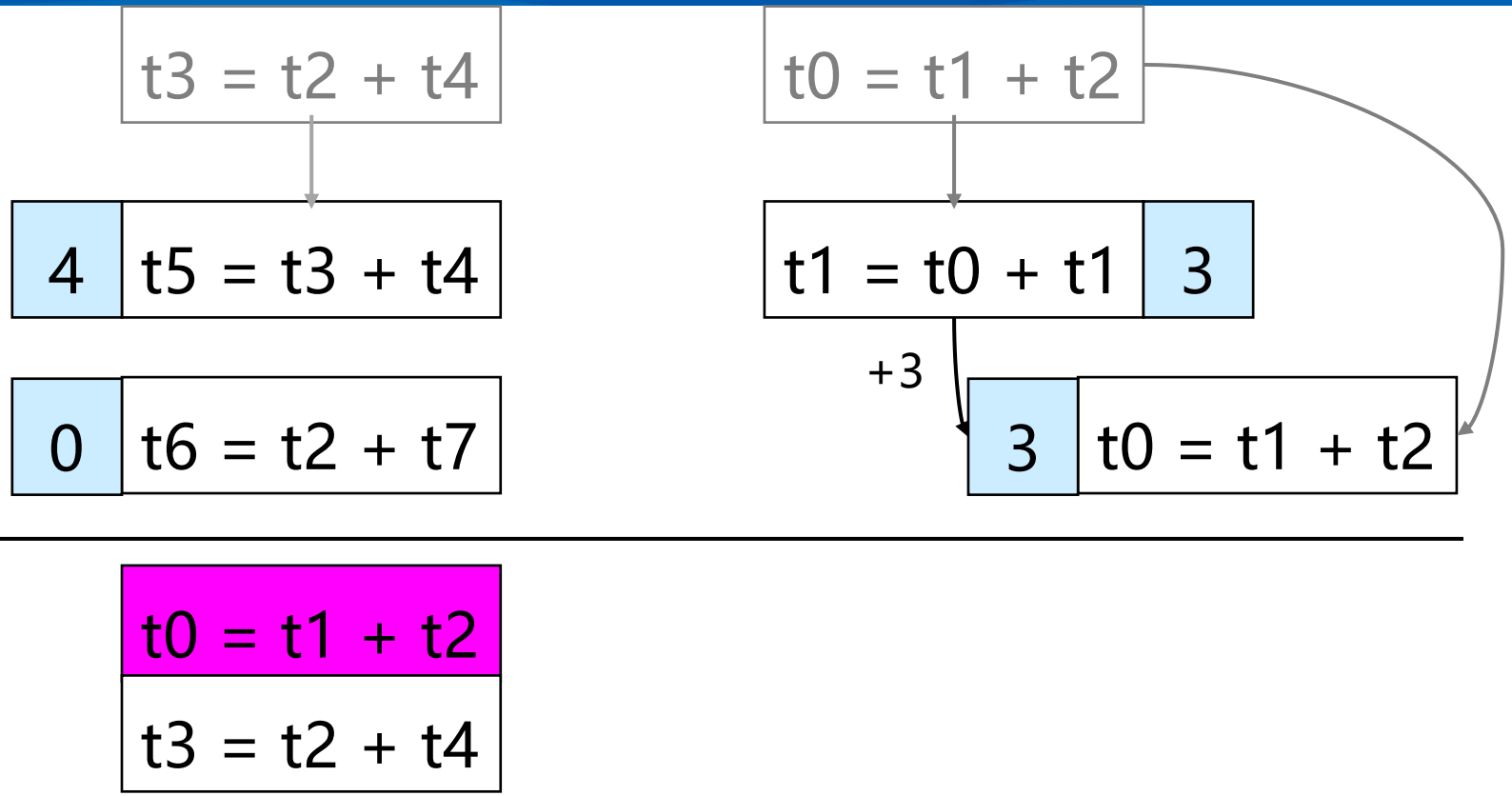
升级版的指令调度



t0 = t1 + t2
t3 = t2 + t4

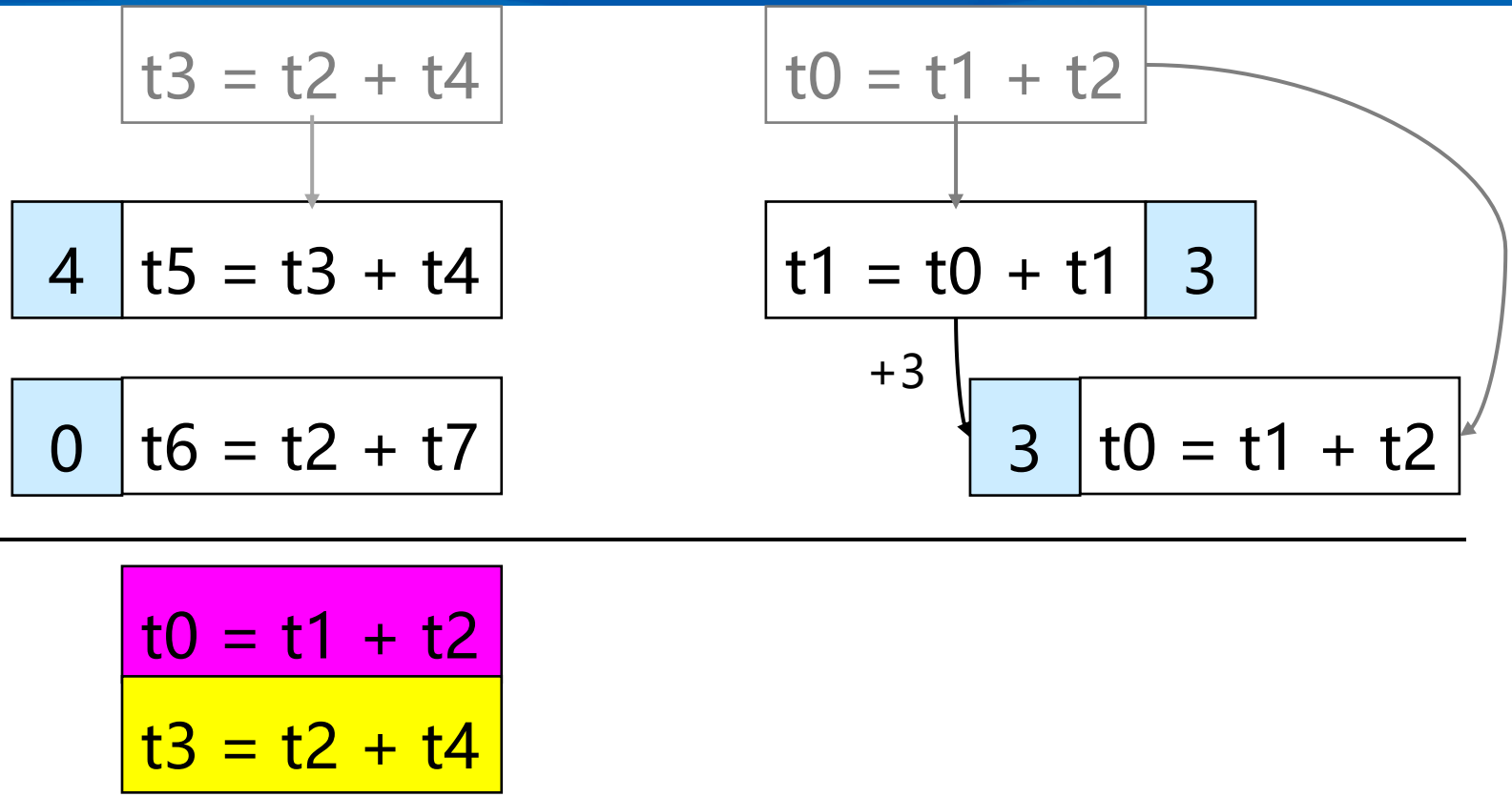
ID	RR	ALU	RW

升级版的指令调度



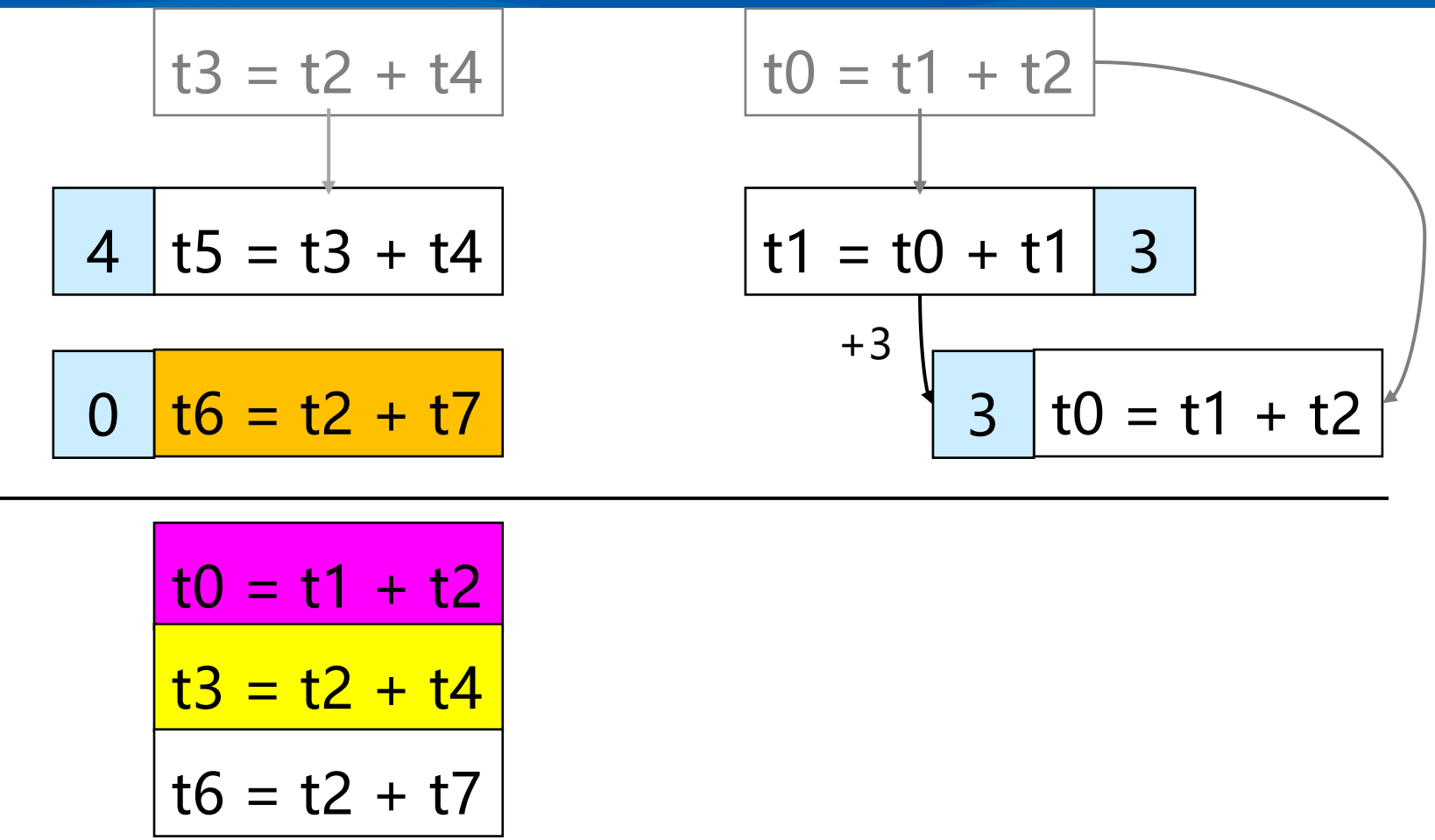
ID	RR	ALU	RW

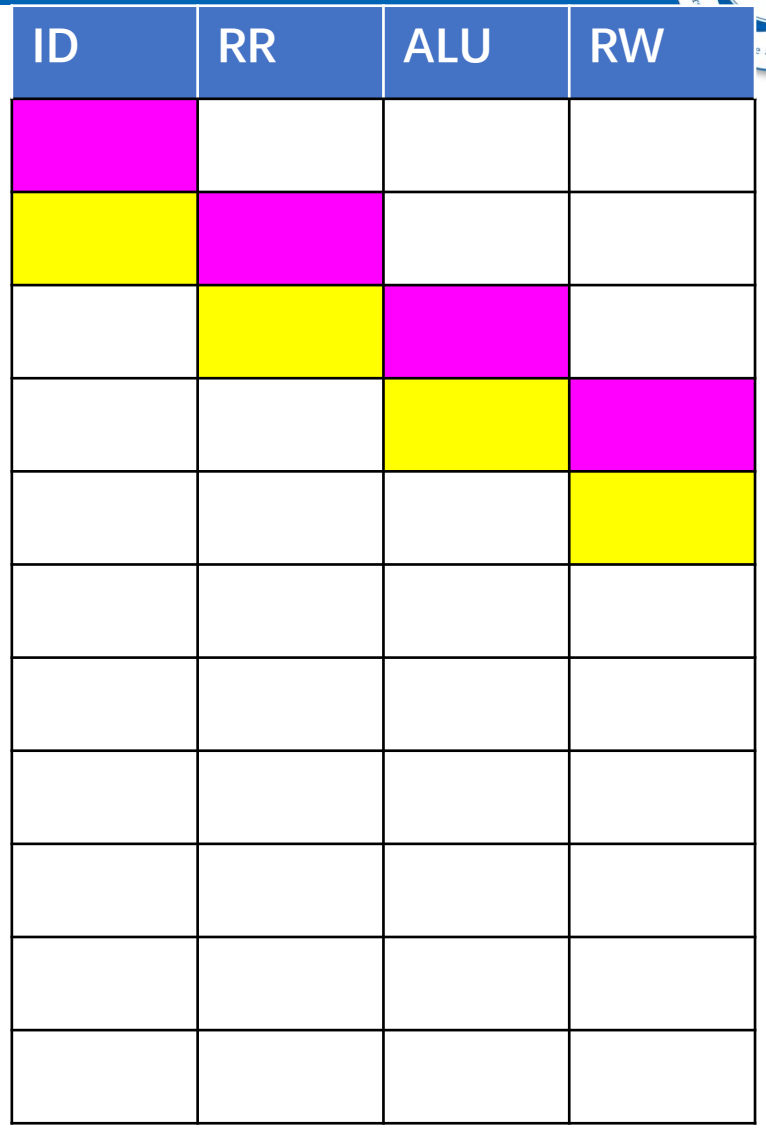
升级版的指令调度



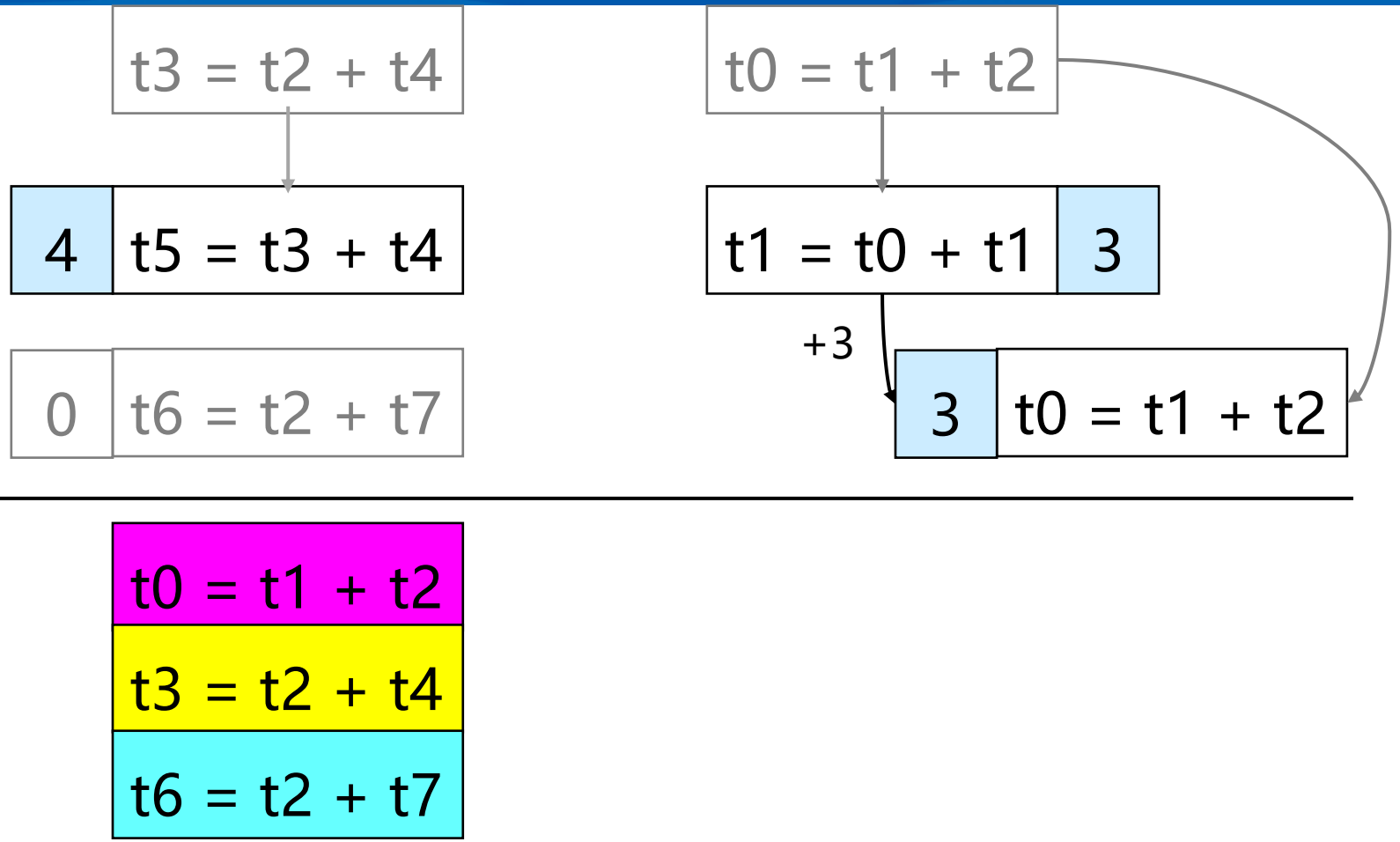
ID	RR	ALU	RW

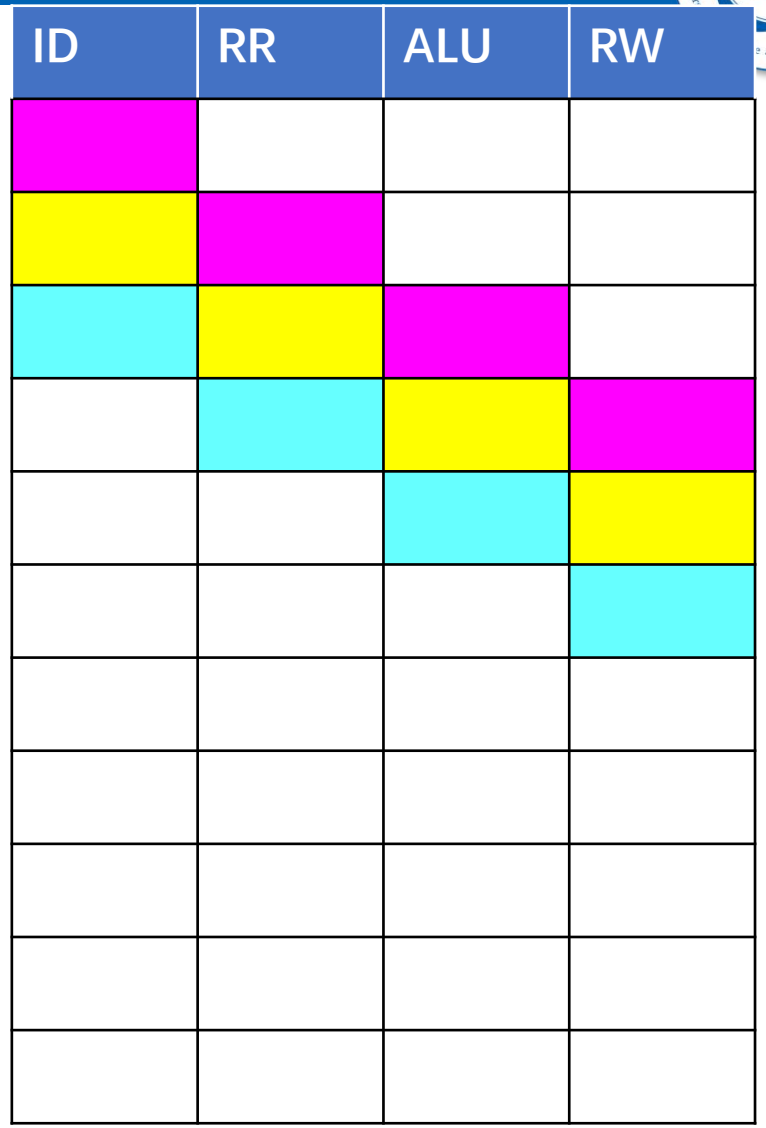
升级版的指令调度


t3 = t2 + t4
t0 = t1 + t2
4
t5 = t3 + t4
t1 = t0 + t1
3
+3
0
t6 = t2 + t7
3
t0 = t1 + t2
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7

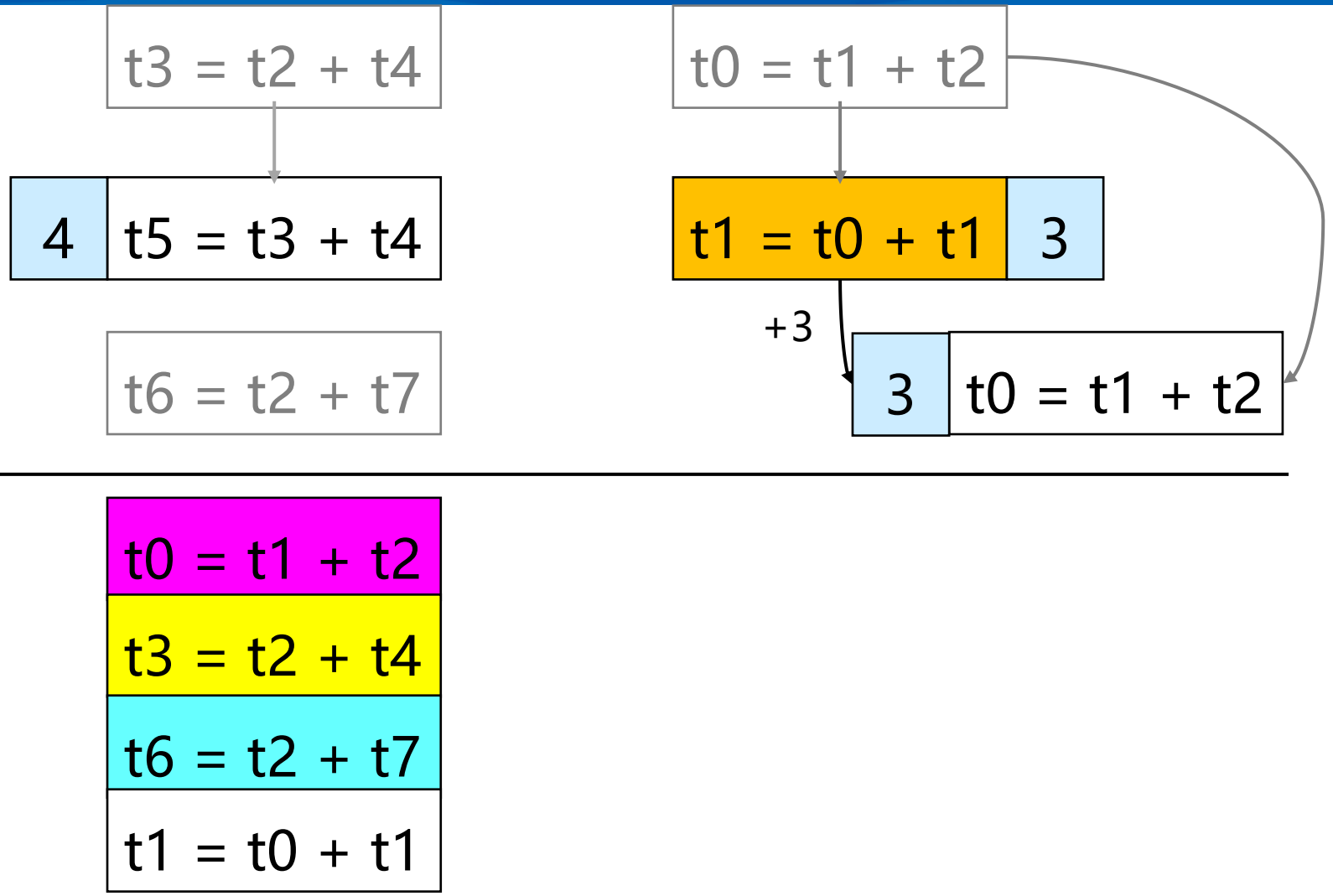

ID
RR
ALU
RW

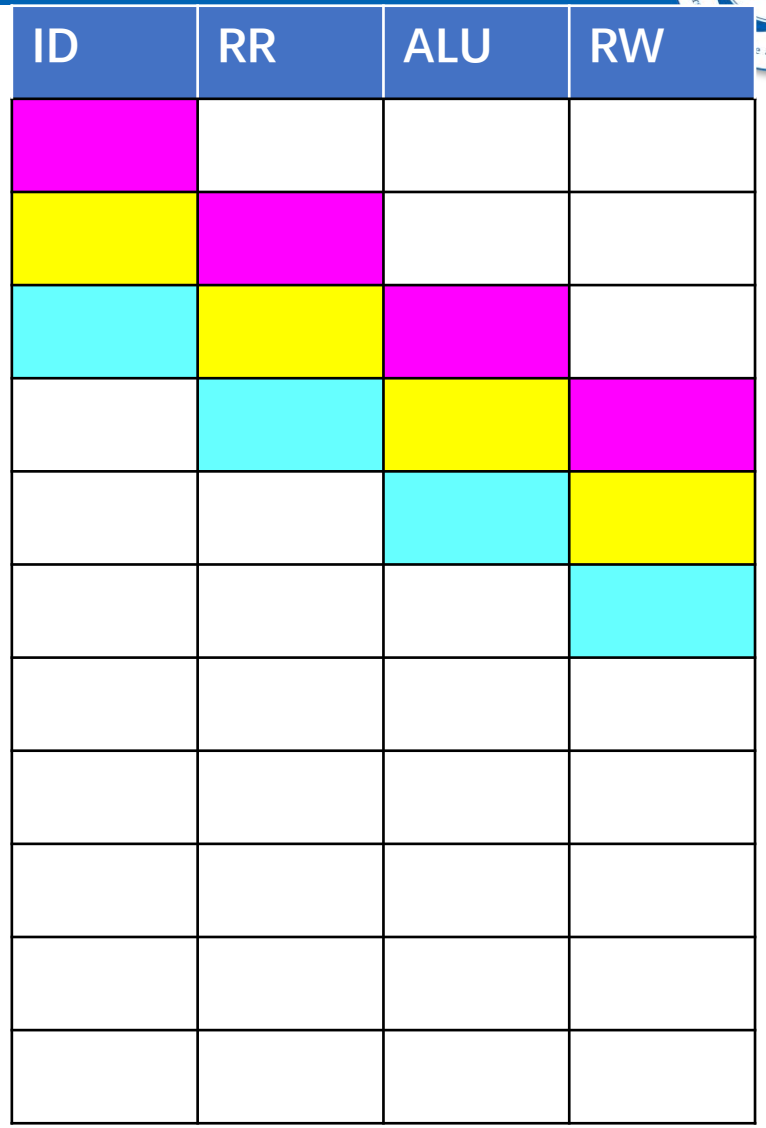
升级版的指令调度


t3 = t2 + t4
t0 = t1 + t2
4
t5 = t3 + t4
t1 = t0 + t1
3
+3
0
t6 = t2 + t7
3
t0 = t1 + t2
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7

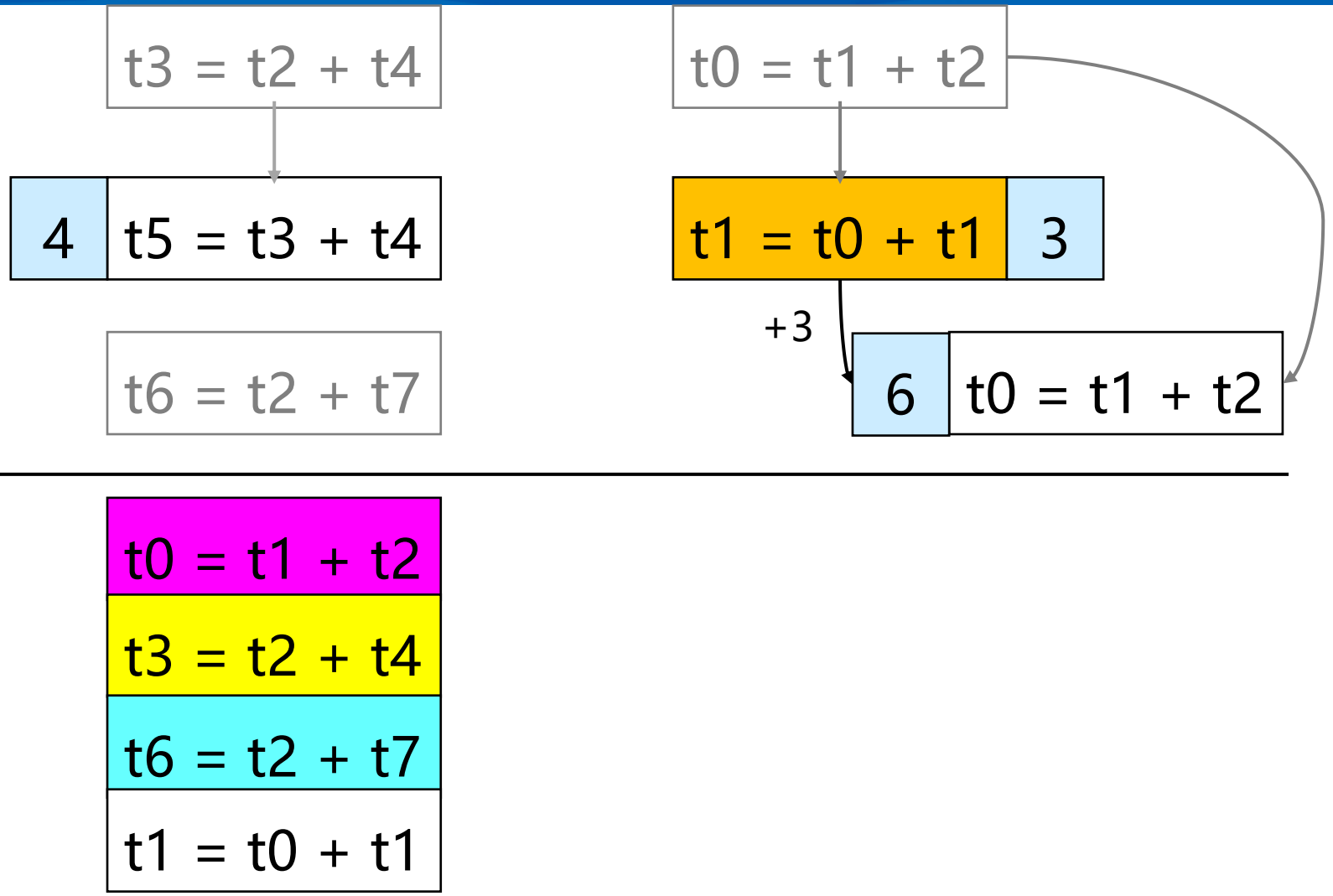

ID
RR
ALU
RW

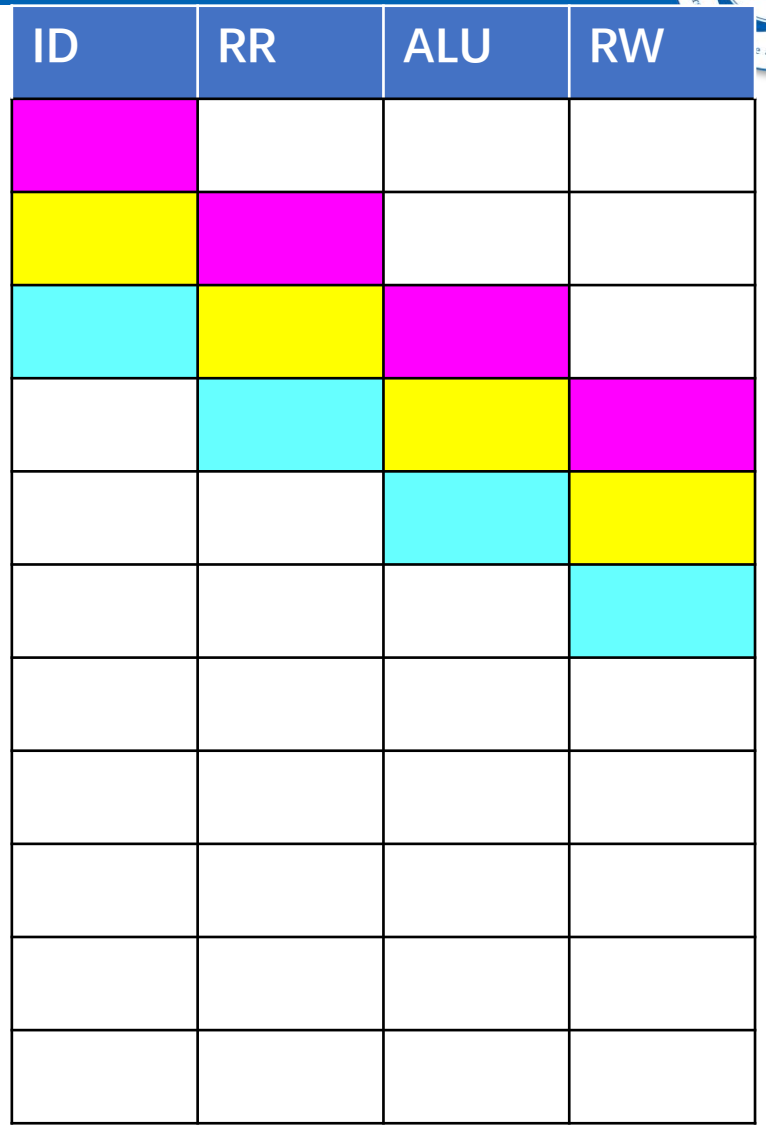
升级版的指令调度


t3 = t2 + t4
t0 = t1 + t2
4
t5 = t3 + t4
t1 = t0 + t1
3
+3
t6 = t2 + t7
3
t0 = t1 + t2
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1

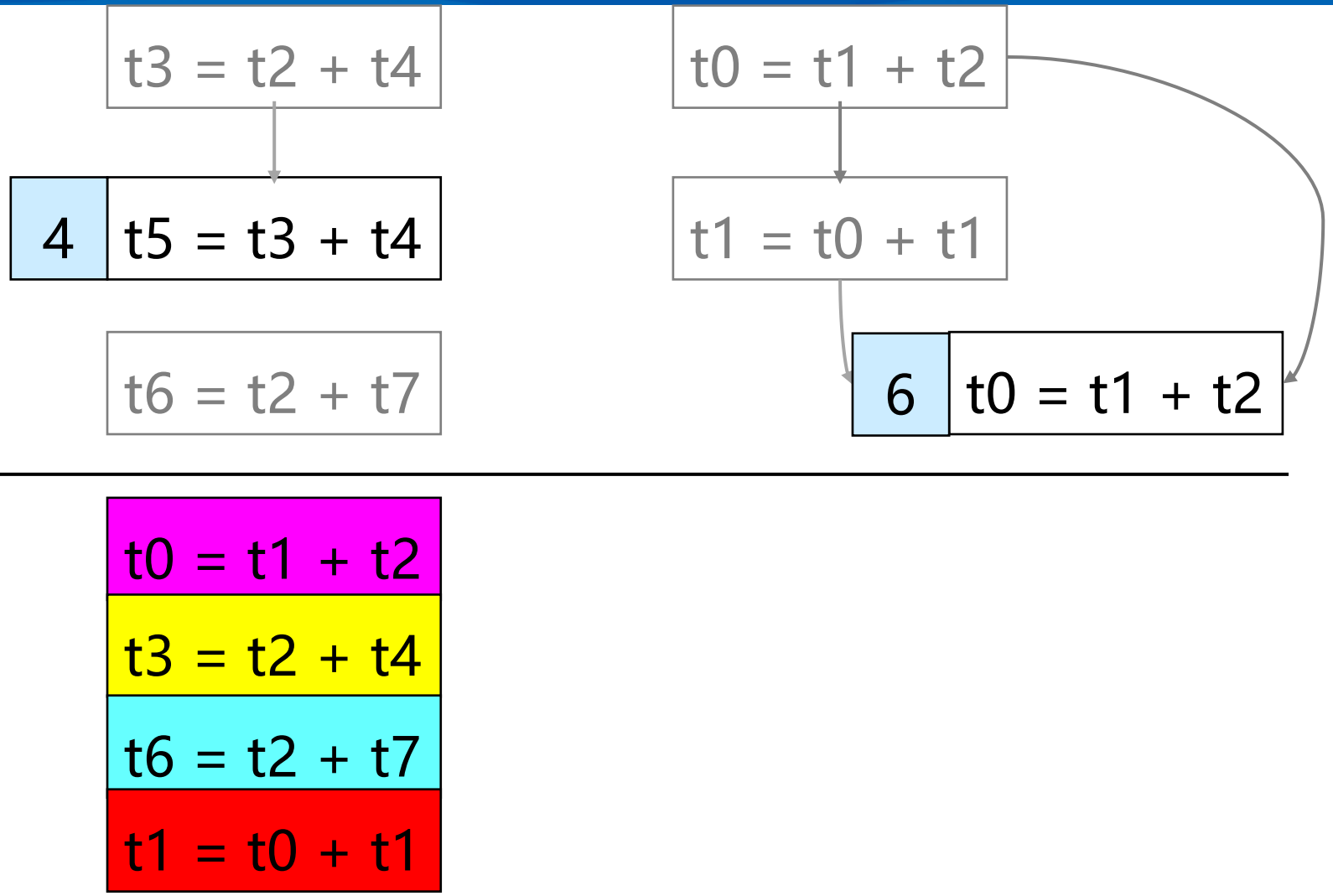

ID
RR
ALU
RW

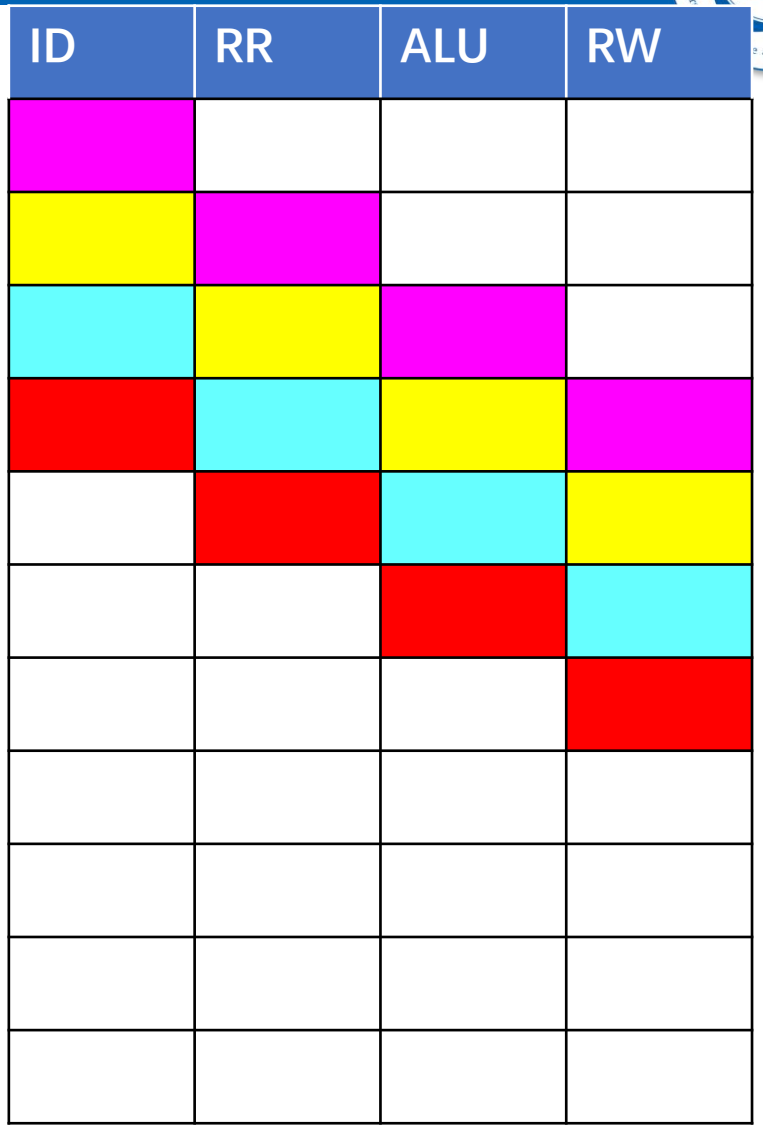
升级版的指令调度


t3 = t2 + t4
t0 = t1 + t2
4
t5 = t3 + t4
t1 = t0 + t1
3
+3
t6 = t2 + t7
6
t0 = t1 + t2
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1


ID
RR
ALU
RW

升级版的指令调度


t3 = t2 + t4
t0 = t1 + t2
4
t5 = t3 + t4
t1 = t0 + t1
t6 = t2 + t7
6
t0 = t1 + t2
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1


ID
RR
ALU
RW

升级版的指令调度

t3 = t2 + t4

4 | t5 = t3 + t4

t6 = t2 + t7

t0 = t1 + t2

t1 = t0 + t1

6 | t0 = t1 + t2

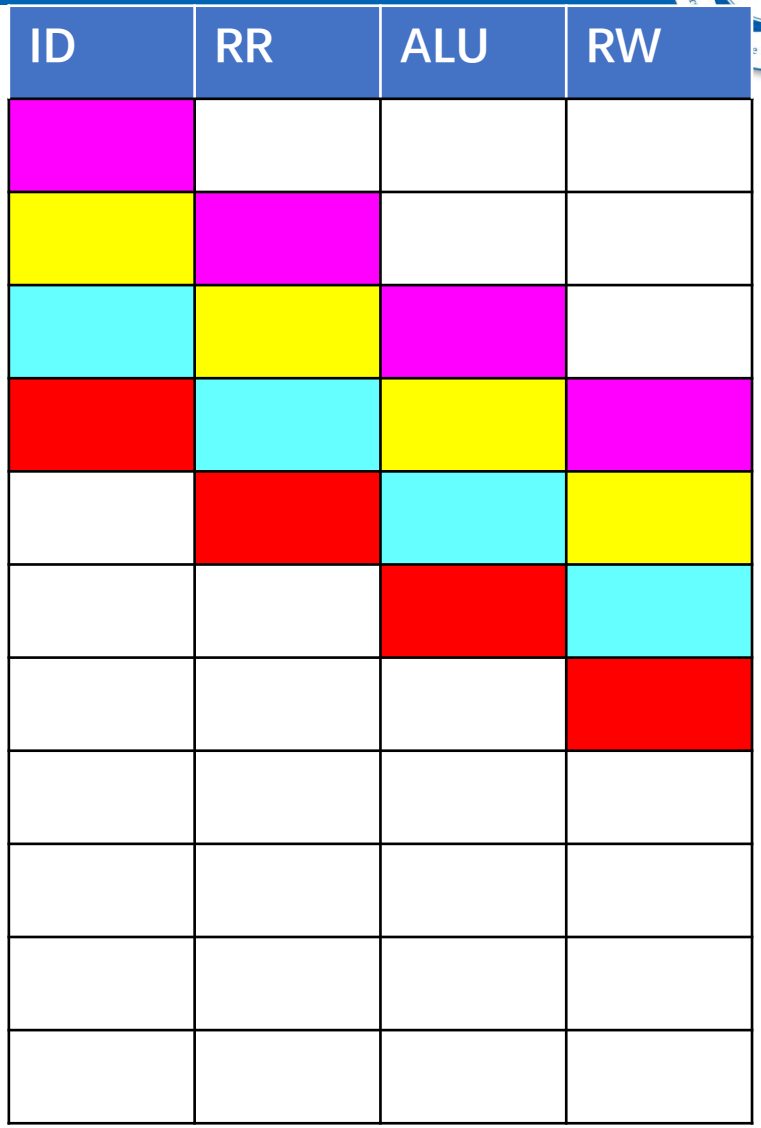
t0 = t1 + t2

t3 = t2 + t4

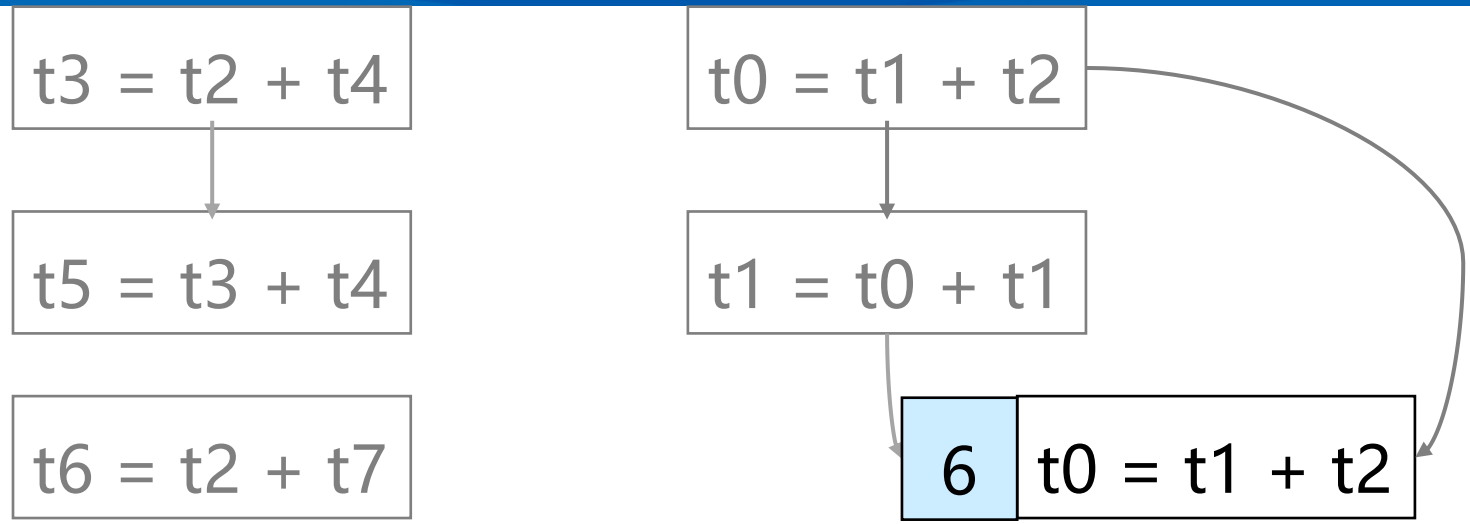
t6 = t2 + t7

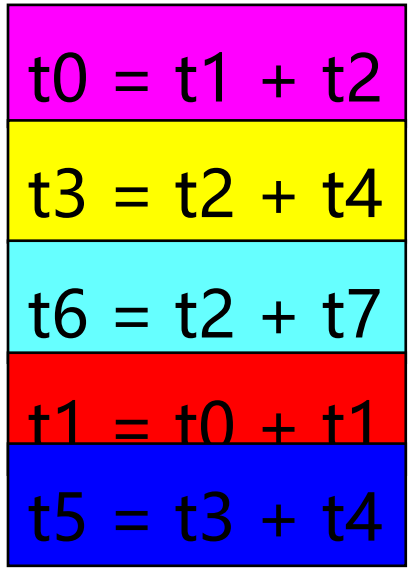
t1 = t0 + t1

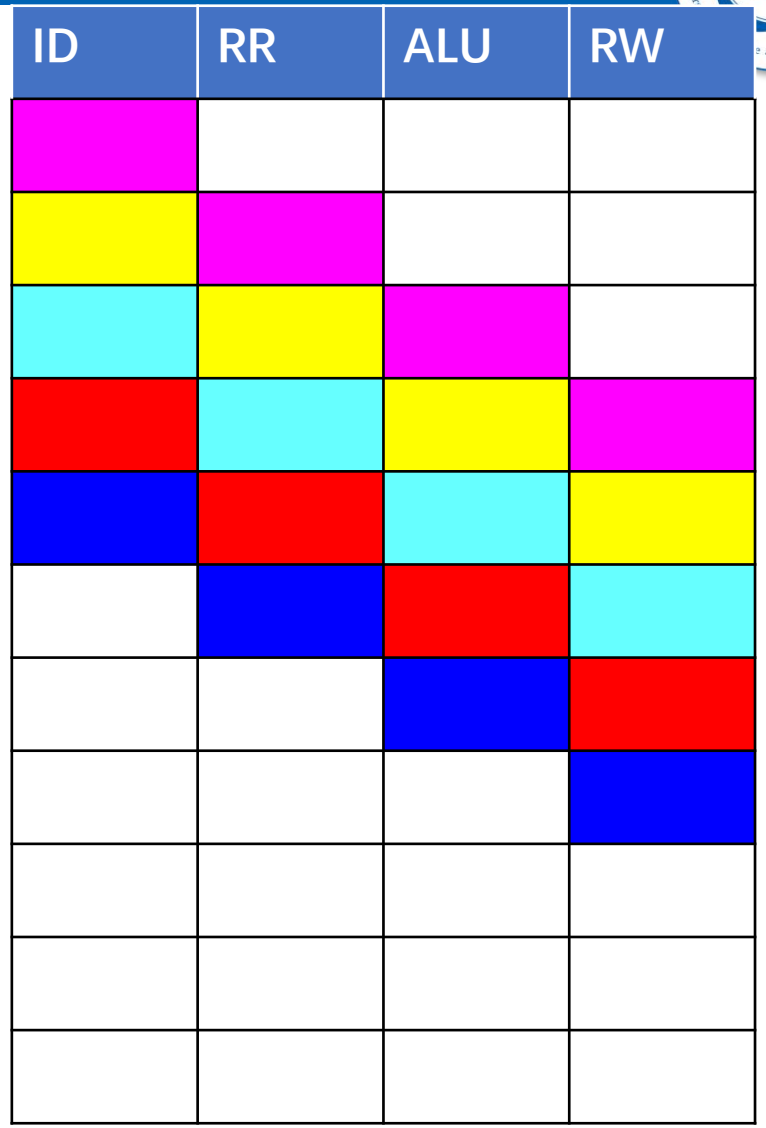
t5 = t3 + t4



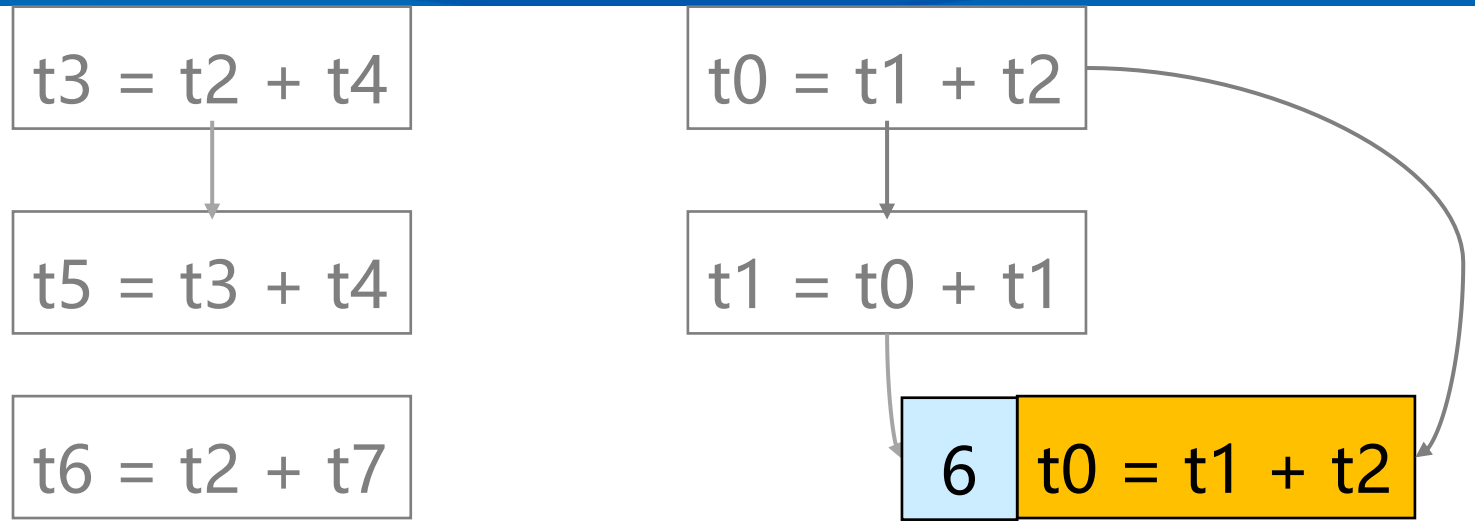
升级版的指令调度

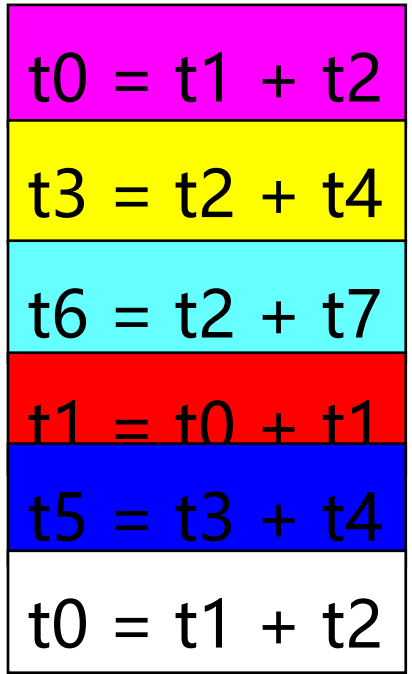

t3 = t2 + t4
t5 = t3 + t4
t6 = t2 + t7
t0 = t1 + t2
t1 = t0 + t1
6
t0 = t1 + t2

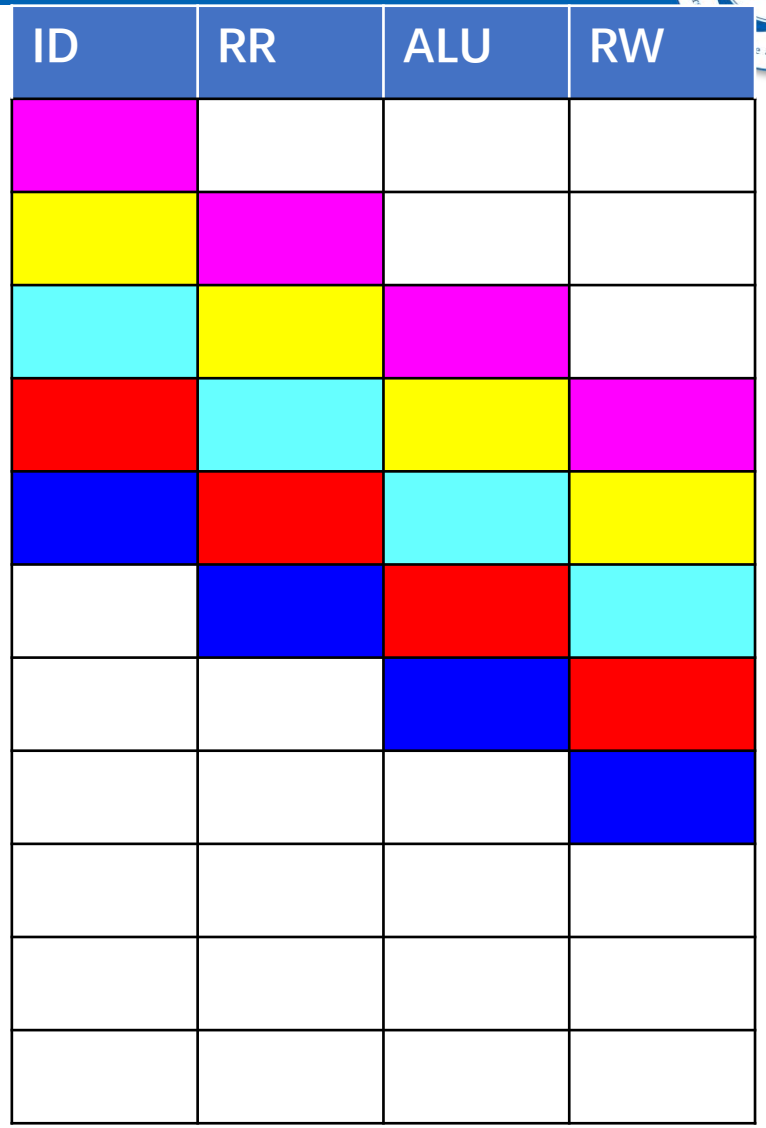

t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
t5 = t3 + t4


ID
RR
ALU
RW

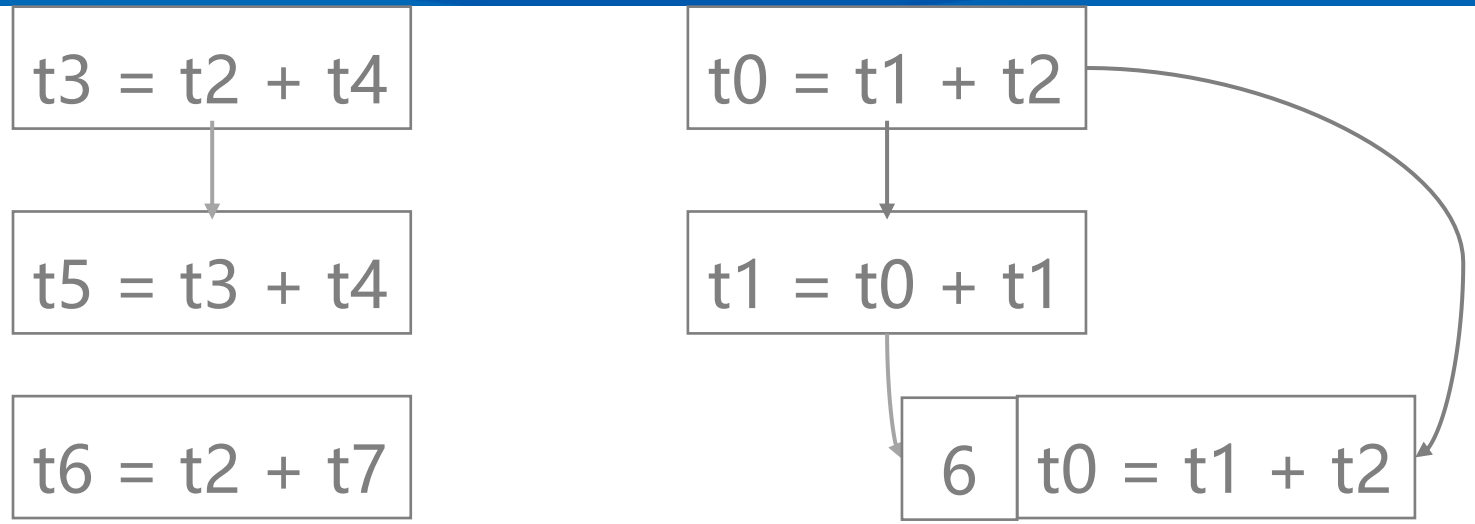
升级版的指令调度

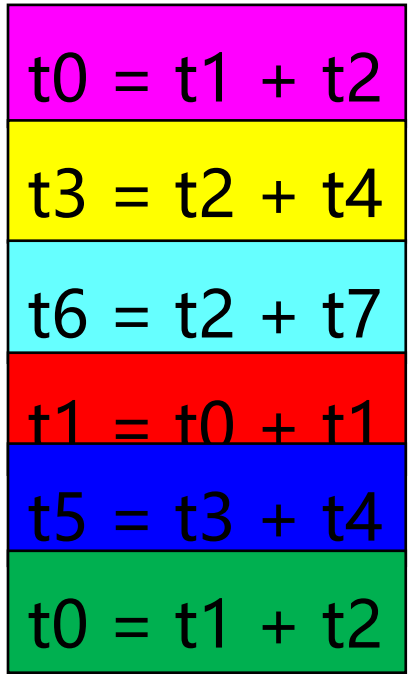

t3 = t2 + t4
t5 = t3 + t4
t6 = t2 + t7
t0 = t1 + t2
t1 = t0 + t1
6
t0 = t1 + t2

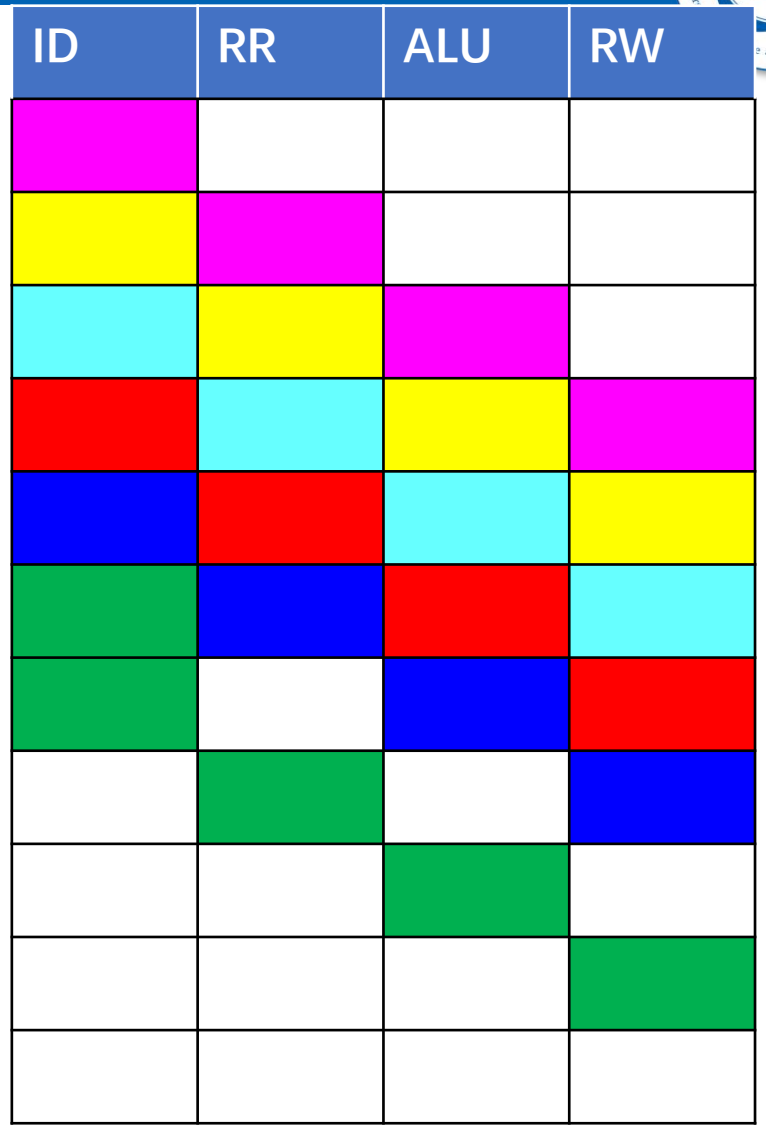

t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
t5 = t3 + t4
t0 = t1 + t2


ID
RR
ALU
RW

升级版的指令调度


t3 = t2 + t4
t5 = t3 + t4
t6 = t2 + t7
t0 = t1 + t2
t1 = t0 + t1
6
t0 = t1 + t2


t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
t5 = t3 + t4
t0 = t1 + t2


ID
RR
ALU
RW

不同调度之间的性能差距

t0 = t1 + t2

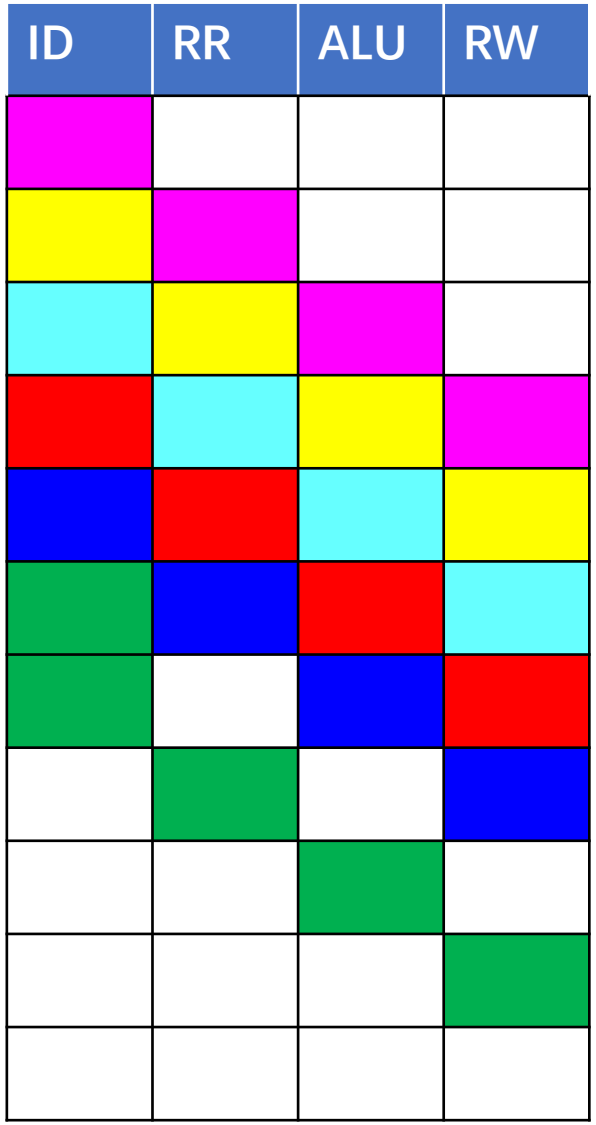
t3 = t2 + t4

t6 = t2 + t7

t1 = t0 + t1

t5 = t3 + t4

t0 = t1 + t2



t0 = t1 + t2

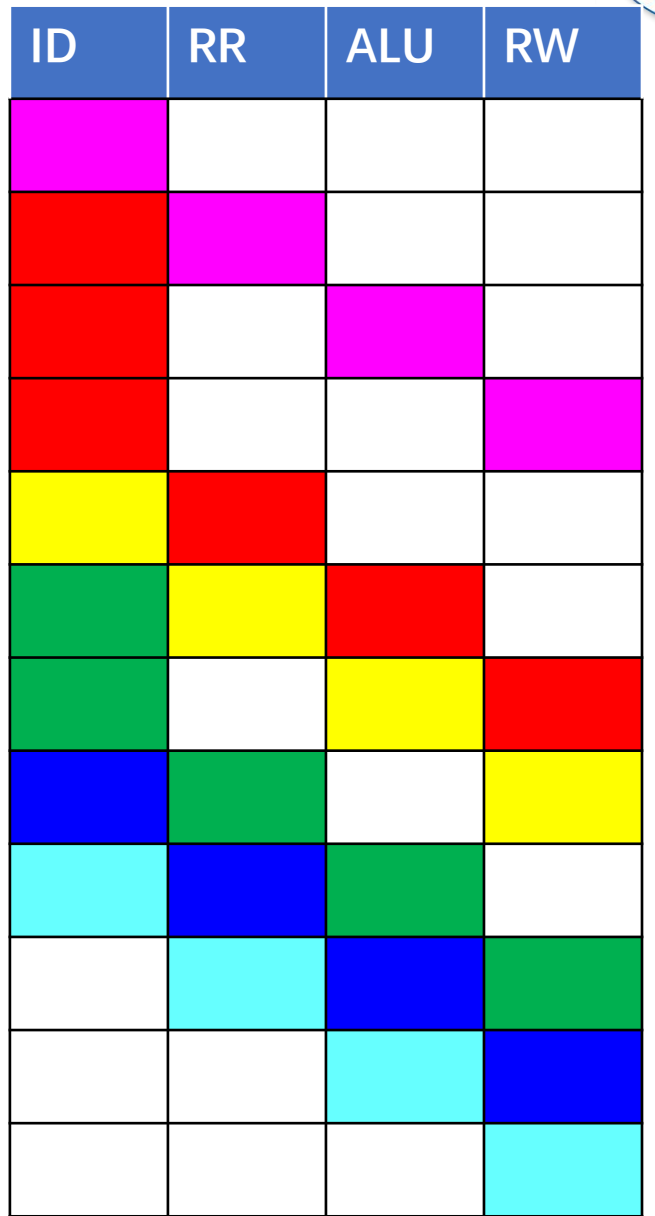
t1 = t0 + t1

t3 = t2 + t4

t0 = t1 + t2

t5 = t3 + t4

t6 = t2 + t7



更多高级的调度

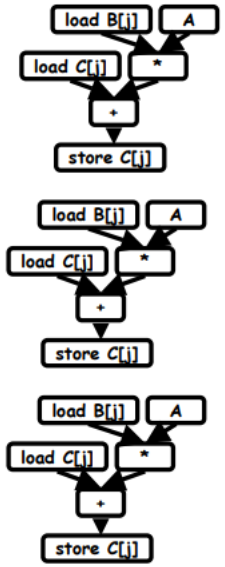
- 现代优化编译器可以进行更积极的调度，从而获得惊人的性能提升。
- 一种强大的技术：循环展开(loop unrolling)
  - 一次展开多个循环迭代。
  - 使用前面介绍的调度算法更智能地调度指令。
  - 可以在循环迭代中找到流水线级并行性。

软件流水线(Software pipeline)

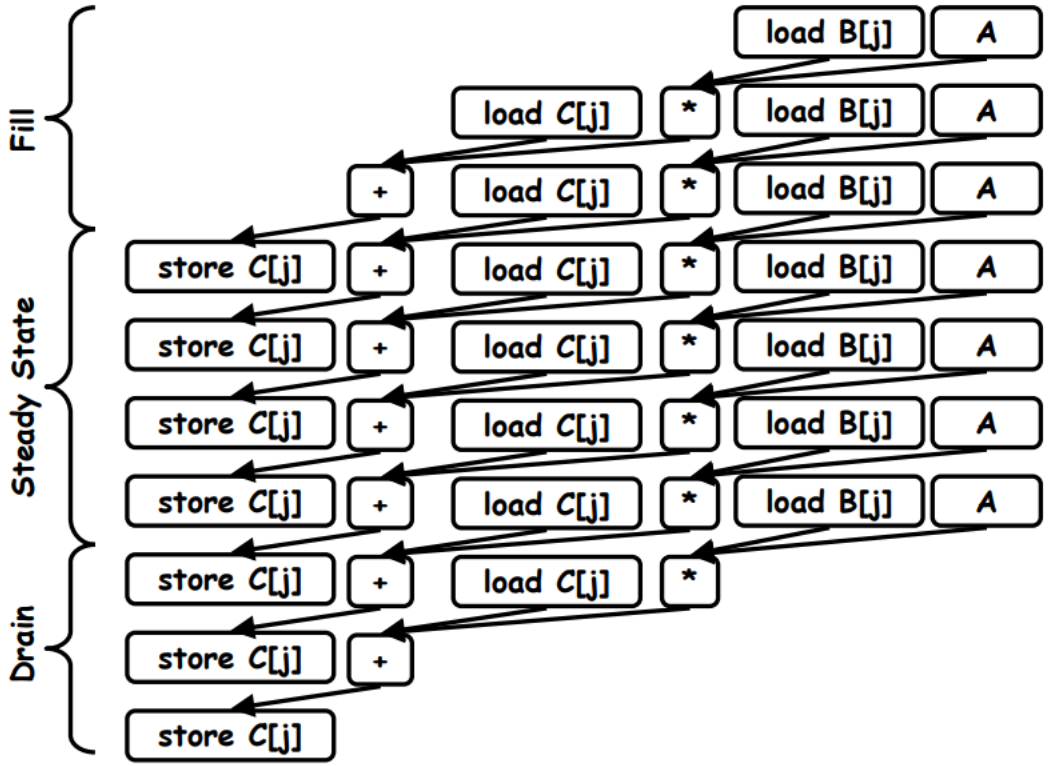
- 通过并行执行来自不同循环体的指令来加快循环程序的执行速度;
- 在前一个循环体未结束前启动下一个新的循环体，来达成循环体时间上的并行性;
- 相比于简单的展开循环,软件流水线在优化资源使用的同时保持代码的简洁。

```
for(j = 0; j < MAX; j++)
  C[j] += A * B[j];
```

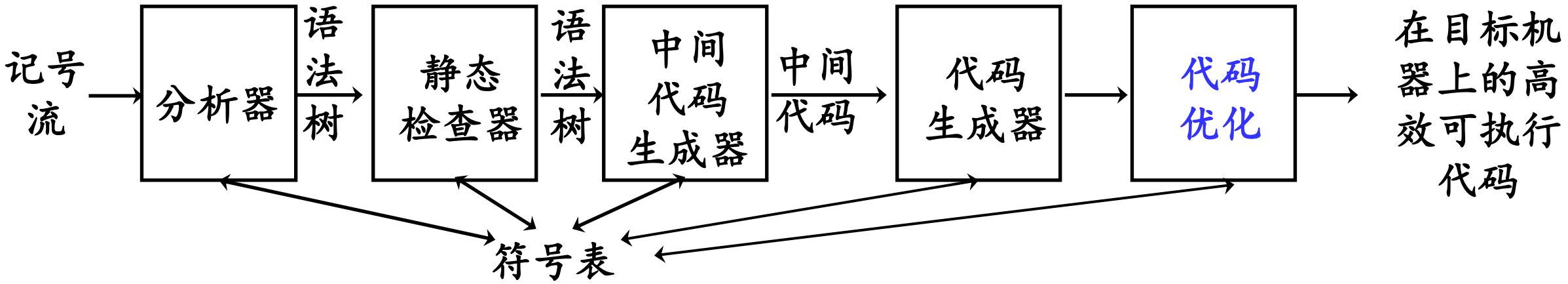
Not pipelined:



Pipelined:

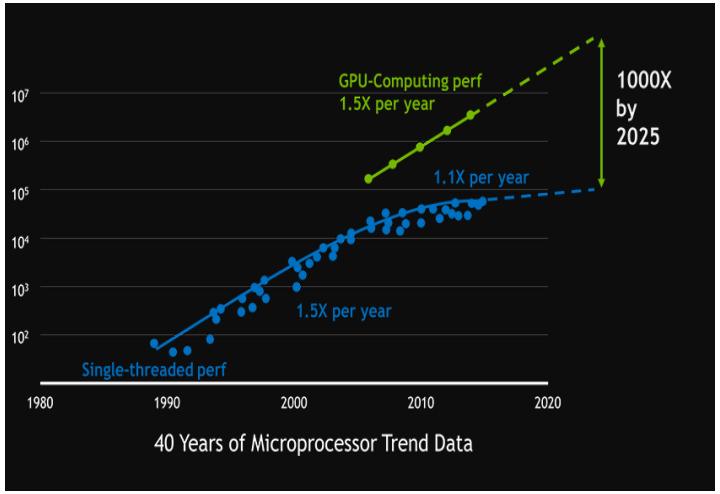


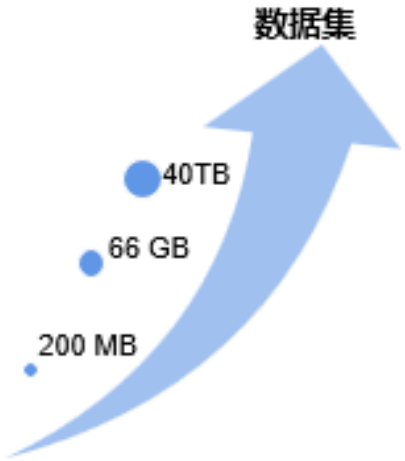
本节提纲

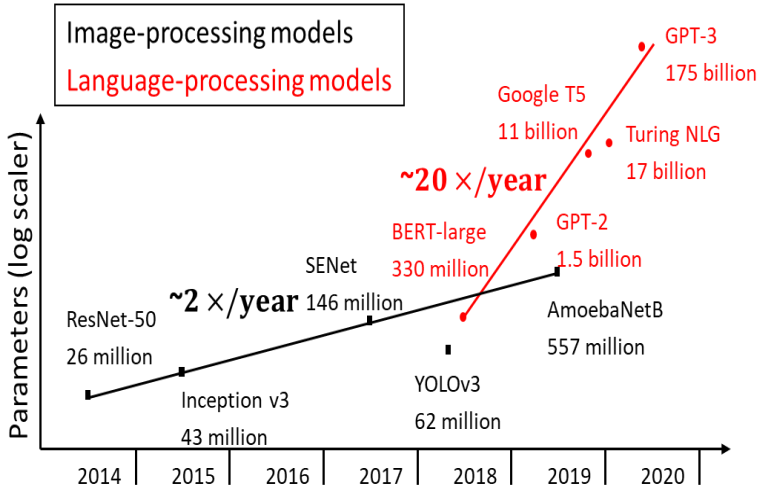


- 现代处理器架构
- 流水线并行的例子
- 指令调度与数据依赖分析
- 数据依赖指导下的指令调度
- **科技前沿——大模型的流水并行训练**

科技前沿——大模型并行训练







持续增长的算力

在后摩尔时代，GPU依然保持每年50%的算力增长幅度

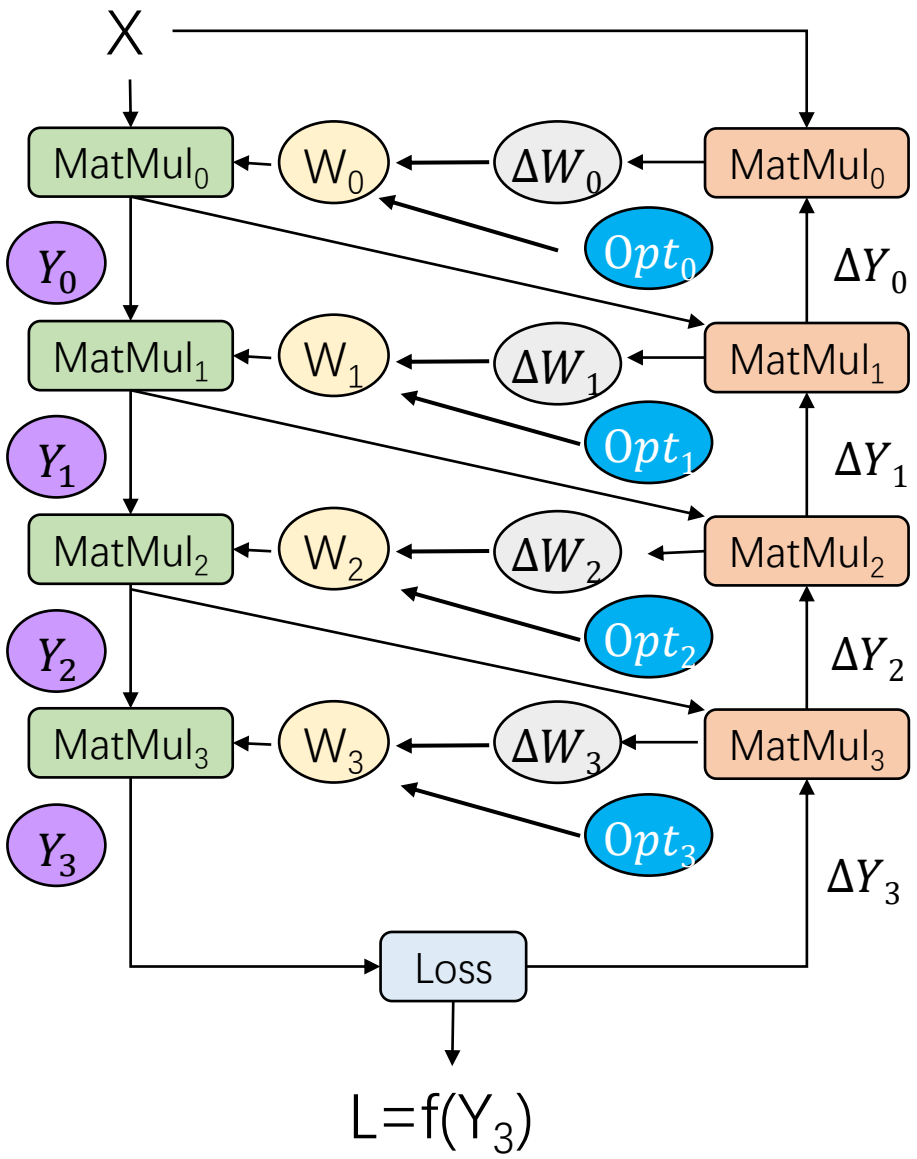
爆发式增长的数据

自然语言处理领域的训练数据集，从200MB增长到40TB

爆发式变大的模型

自然语言处理领域的模型，大小以每年20倍的速度增长

训练过程中“四大”内存占用



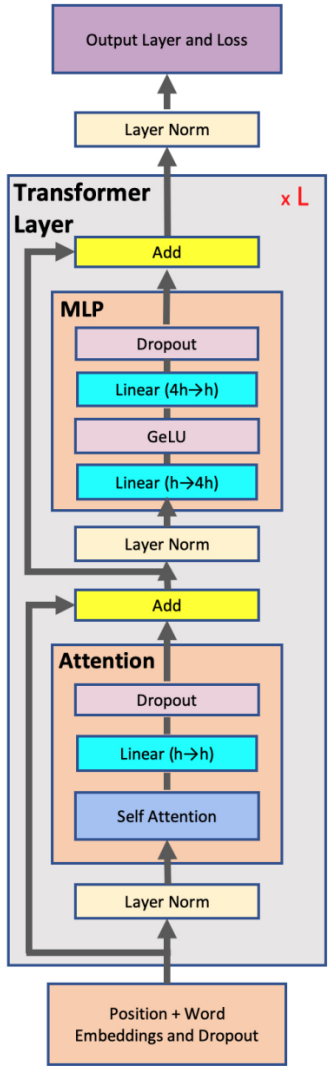
参数

梯度

中间数据

优化器状态

以GPT为例分析训练内存占用情况

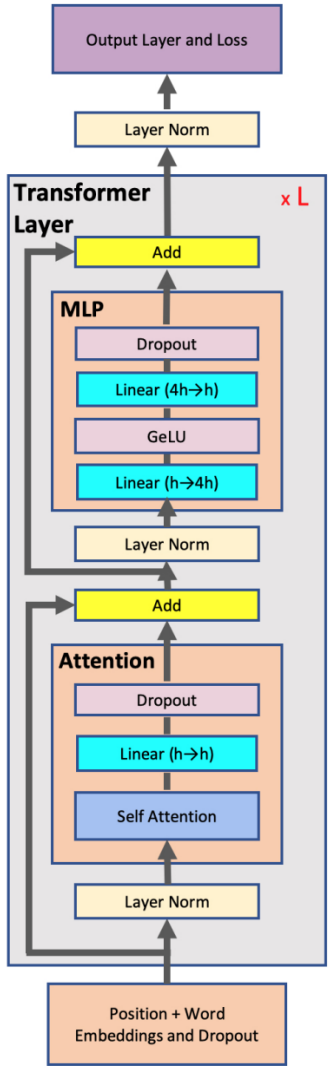


用于计算Loss的输出层

重复堆叠的Transformer层

→ Embedding层

以GPT为例分析训练内存占用情况



用于计算Loss的输出层

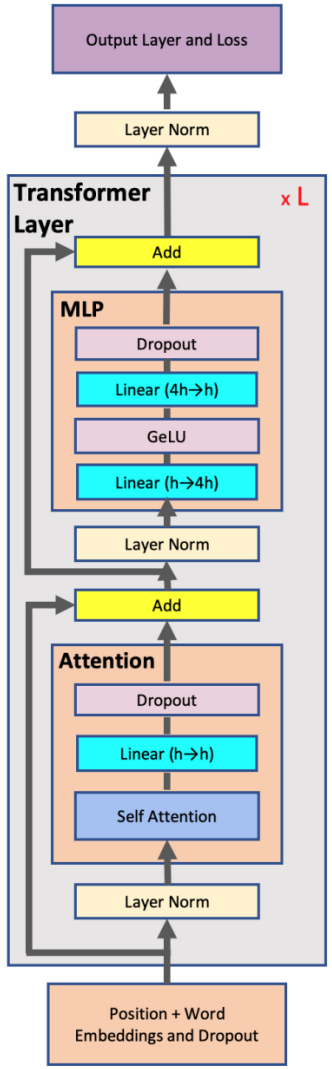
重复堆叠的Transformer层

→ Embedding层

训练1750亿参数的GPT-3的配置

参数	缩写	参考值
hidden size	h	12288
sequence length	s	2048
number of layers	L	96
number of attention heads	a	96
mini-batch size	b	128
vocabulary size	v	50000

以GPT为例分析训练内存占用情况



用于计算Loss的输出层

重复堆叠的Transformer层

→ Embedding层

训练1750亿参数的GPT-3的配置

参数	缩写	参考值
hidden size	h	12288
sequence length	s	2048
number of layers	L	96
number of attention heads	a	96
mini-batch size	b	128
vocabulary size	v	50000

训练1750亿参数的GPT-3的内存占用

内存占用类型	占用大小	参考值
参数 + 梯度	$(L(12h^2+13h)+hv+h(s+1))\times 4$	650 GB
优化器状态量	$(L(12h^2+13h)+hv+h(s+1))\times 12$	1950 GB
中间数据	$(L(5as^2+34hs)+5hs+4sv)\times 4b$	32895 GB

以GPT为例分析训练内存占用情况

显卡型号	发售年份	显存容量	参考价格
H100	2023	80GB	$36550
A100	2020	40/80GB	$9745 (40GB)
V100	2017	16/32 GB	$4392 (16GB)
P100	2016	16GB	$557

➡ **需要至少440/880张A100才能满足GPT-3训练过程中的内存占用**

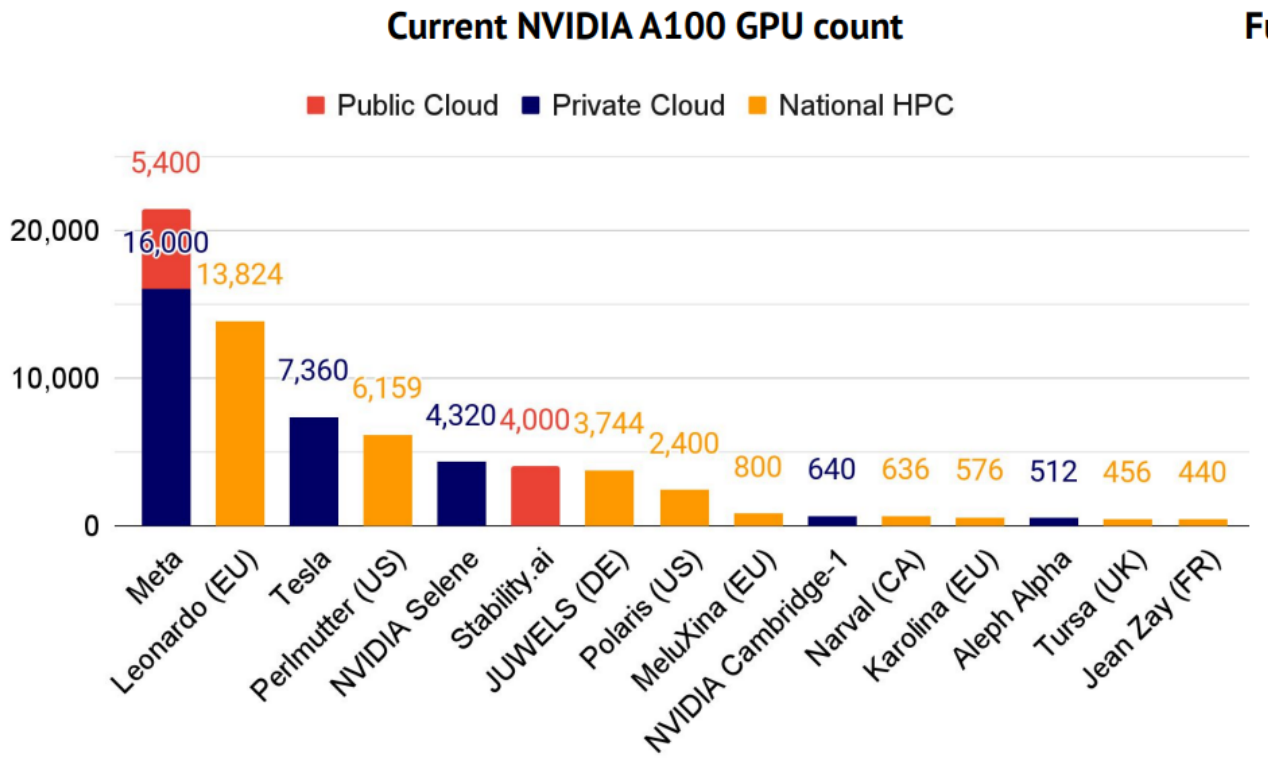
训练1750亿参数的GPT-3的内存占用

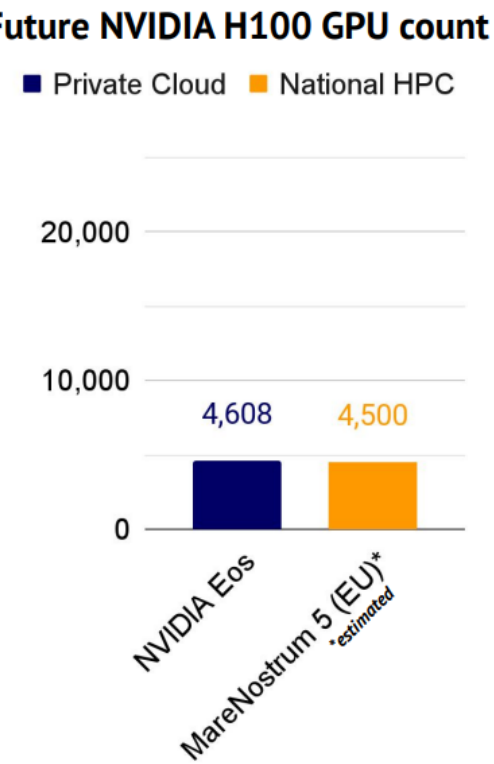
内存占用类型	占用大小	参考值
参数 + 梯度	$(L(12h^2+13h)+hv+h(s+1))\times 4$	650 GB
优化器状态量	$(L(12h^2+13h)+hv+h(s+1))\times 12$	1950 GB
中间数据	$(L(5as^2+34hs)+5hs+4sv)\times 4b$	32895 GB

国外各大机构的A100卡数

In a gold rush for compute, companies build bigger than national supercomputers

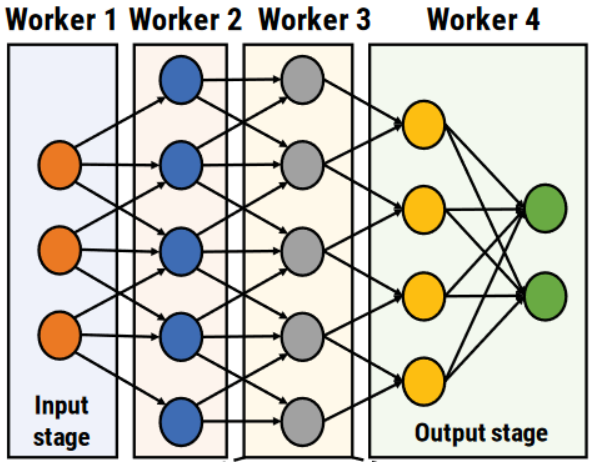
"We think the most benefits will go to whoever has the biggest computer" – Greg Brockman, OpenAI CTO

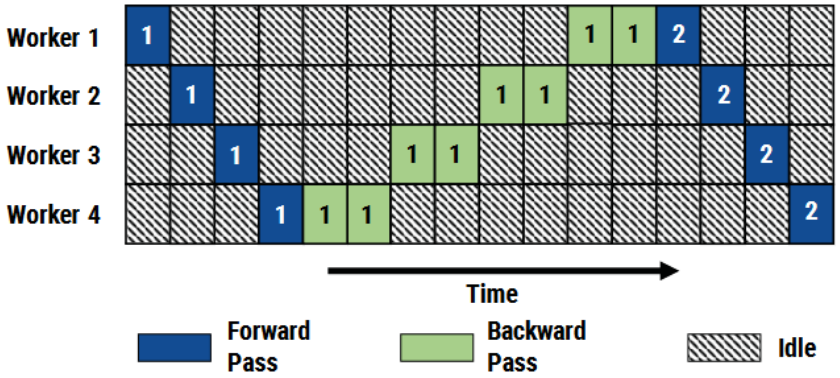




如何用更少的硬件资源训练大模型成为关键问题之一?

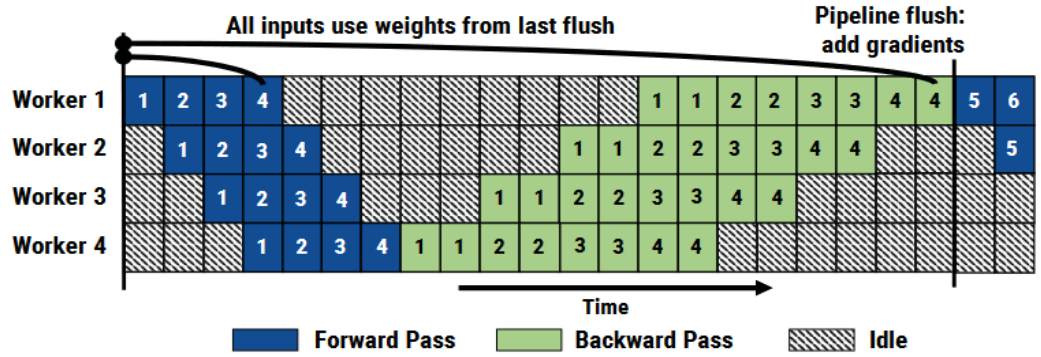
流水线并行: pipeline parallelism



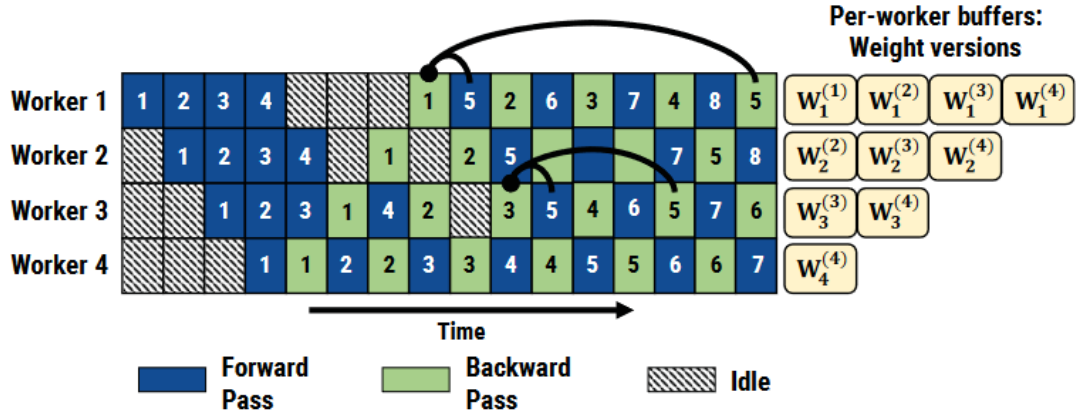


Intuitive strategy: devices are mostly idle

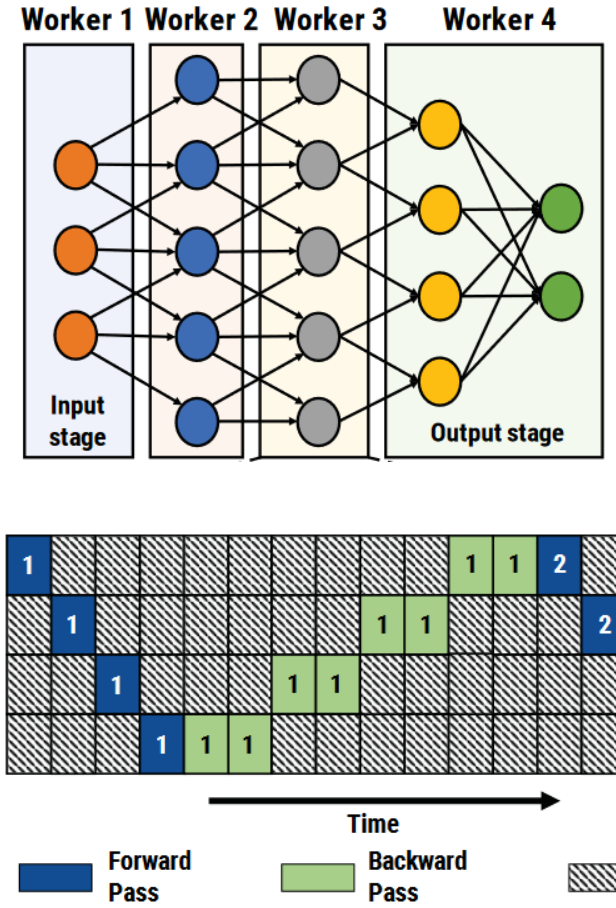
Gpipe, NIPS 2019, Google：将mini-batch进一步拆分为若干micro-batch，但仍有大量bubble



PipeDream, SOSP 2019, MicroSoft：允许后续micro-batch异步提前执行，但使用的参数比较陈旧



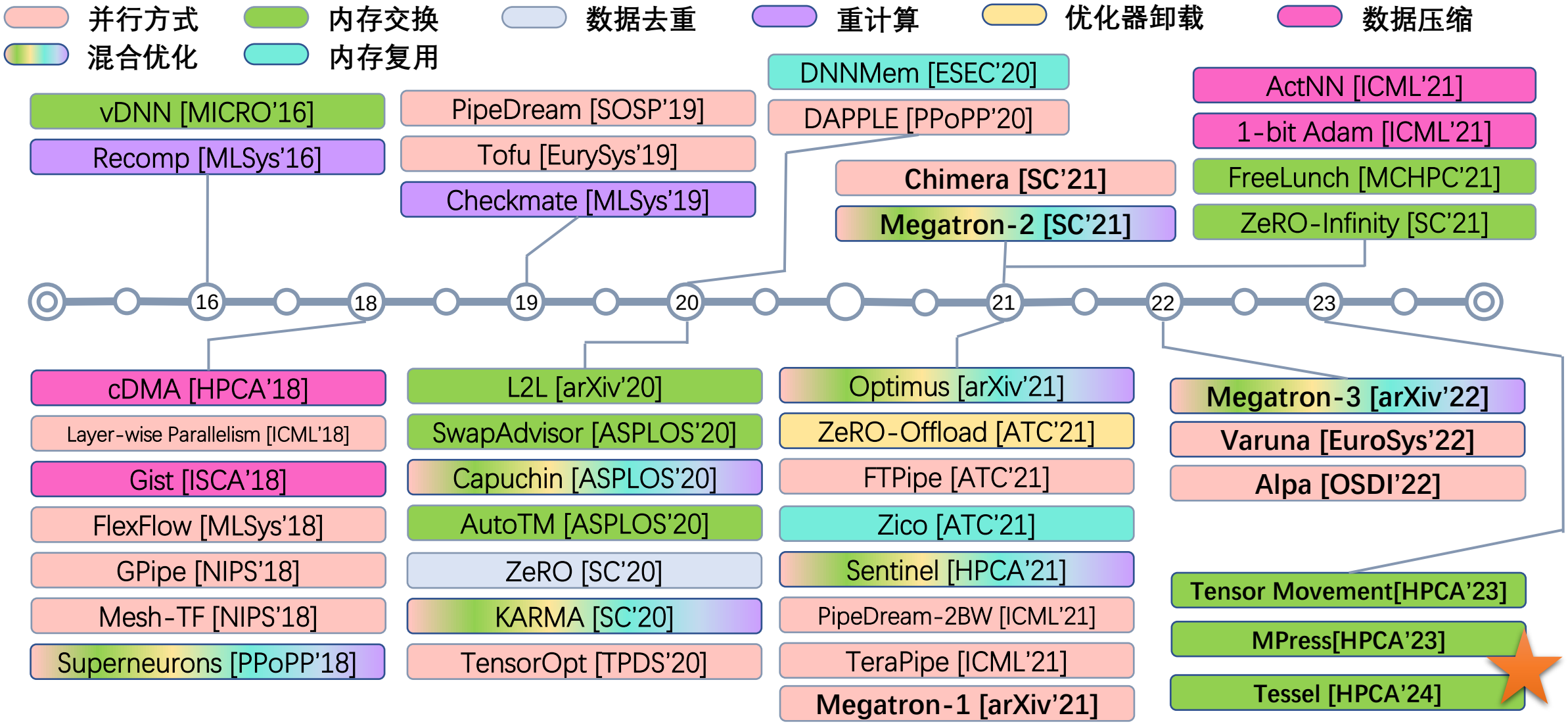
流水线并行: pipeline parallelism



Intuitive strategy: devices are mostly idle

DAPPLE, PPoPP 2021, Alibaba：修改执行序，交叉执行不同micro-batch的前向和反向计算，减少bubble，减少保存的activation数量

面向AI训练的内存腾挪技术



一起努力
打造国产基础软硬件体系！

李 诚

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2023年11月29日