Lecture 4.1

指令选择和调度优化

徐 辉

xuh@fudan.edu.cn

大纲

一、指令集和汇编代码

二、指令选择和翻译

三、指令调度算法

IR => 汇编代码

- IR指令=>汇编指令
- 变量=>寄存器和栈

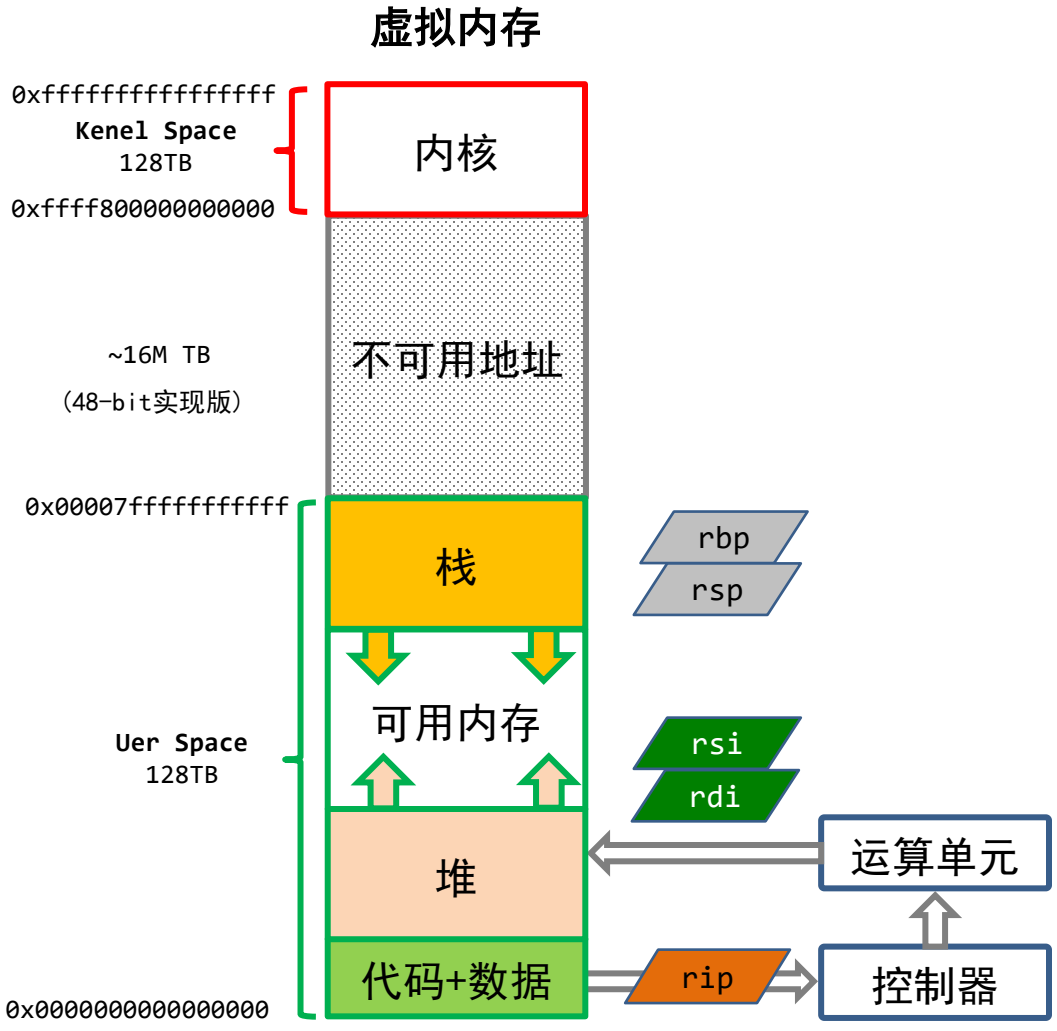
```
define i32 @foo(i32 %0a, i32 %0b) {
  %a = stackalloc i32;
  %b = stackalloc i32;
  store i32 %a0, %a;
  store i32 %b0, %b;
  %1 = load i32 %0a;
  %2 = icmp ne i32 %1, 0
  cjmp i1 %2, %BB1, %BB2

%BB1 :
  %3 = load i32 %b;
  %4 = add i32 %3, 1
  store i32 %4, %b
  jmp %BB2

%BB2:
  5% = load i32 %b;
  ret i32 %5;
}
```

```
# %bb.0:
     PUSHQ    %rbp
     PUSHQ     %rsp, %rbp
     MOVL     %edi, -4(%rbp)
     MOVL     %esi, -8(%rbp)
     CMPL     $0, -4(%rbp)
     JE       .LBB0_2
# %bb.1:
     MOVL     -8(%rbp), %eax
     ADDL     $1, %eax
     MOVL     %eax, -8(%rbp)
.LBB0_2:
     MOVL     -8(%rbp), %eax
     POPQ     %rbp
     RETQ
```

指令集架构：Instruction Set Architecture

- 精简指令集（RISC）
  - ARM架构（ARM公司）
- 复杂指令集（CISC）
  - X86、X86-64架构（Intel IA-32、AMD）
- 其它
  - very long instruction word (Intel IA-64)
    - 安腾处理器（Intel Itanium）
  - explicitly parallel instruction computing (EPIC)

X86-64内存空间图解



- rbp：基地址（栈底）寄存器
- rsp：栈地址（栈顶）寄存器
- rsi：指针（源）寄存器
- rdi：指针（目的）寄存器
- rip：指令地址
- rflags：ZF/SF/OF…
- rax：通用累加器
- rbx：基地址寄存器，指向数据
- rcx：计数器，用于循环等
- rdx：数据寄存器，用于IO
- r08-r15：通用寄存器

寻址模式（AT&T风格）

- 直接寻址：MOVL $1, 0x604892
- 间接寻址：MOVL $1, (%eax)
  - 带位移：MOVL $1, -24(%rbp)
    - 地址 = %rbp-24
  - 带索引：MOVL $1, (%rbp, %rcx, 8)
    - 地址 = %rbp+%rcx*8
  - 带索引和位移的：MOVL $1, 8(%rsp, %rdi, 4)
    - 地址 = %rbp+%rdi*4+8

不同的汇编语法风格

	AT&T 风格(Linux)	Intel风格(Windows)
寄存器前缀	`PUSHL %EAX`	`PUSH EAX`
立即操作数	`PUSHL $1`	`PUSH 1`
源目的顺序	`ADDL $1, %EAX`	`ADD EAX, 1`
操作数字长	`MOVB val, %AL`	`MOV AL, byte ptr val`
寻址方式	`MOVL -4(%EBP), %EAX`	`MOV EAX, [EBP - 4]`
	`MOVB $4, %FS:(%EAX)`	`MOV FS:EAX, 4`

主要X86-64指令：二元运算

- 将参数一和参数二对应的值运算后保存到参数二中
  - 参数一可以是立即数、寄存器或内存地址
  - 参数二可以是寄存器或内存地址
  - 两个参数不能同时是内存地址

```
ADD src, dst
AND src, dst
SHL count, dst
```

- 主要指令
  - 四则运算：ADD/SUB/IMUL/MUL/IDIV/DIV/…
    - 除法运算借助rax寄存器，只需要1个参数
  - 位运算：AND/OR/NOT/XOR
  - 位移运算：SHL/SHR/SAR
  - 浮点数运算有单独的指令集（X87），一般在前面加f表示

i:有符号的

主要X86-64指令：数据拷贝

- MOV：将数据从一个地址拷贝到另外一个地址
  - 参数可以是立即数、寄存器、或内存地址
  - 两个参数不能同时是内存地址
- 等量内容拷贝：
  - MOVB：1 byte
  - MOVW：2 bytes
  - MOVL：4 bytes
  - MOVQ：8 bytes

```
MOV $0, %EAX
MOV 8(%RSP), %EAX   <-- 对应IR load
MOVB %AL, 0x409892  <-- 对应IR store
```

- 拷贝到大空间：
  - MOVZBL：将1字节内容拷贝到4字节空间，使用0填充
  - MOVSBL：将1字节内容拷贝到4字节空间，符号扩展

主要X86-64指令：取地址

- LEA：将参数一的地址保存到参数二中。

```
LEA 0x20(%rsp), %rdi          # %rdi = %rsp + 0x20
LEA (%rdi,%rdx,1), %rax       # %rax = %rdi + %rdx
```


对应IR getptr

主要X86-64指令：比较和跳转

- 比较指令：CMPL/TEST，改写EFLAGS中对应的标志位
  - ZF：zero flag
  - SF：sign flag
  - OF：overflow flag，signed
  - CF：carry flag，unsigned

```
CMPL op2, op1    #op1-op2，设置SF
TEST op2, op1    #op1&op2，设置ZF
```

- 直接跳转：JMP
- 比较跳转：
  - JE：ZF = 1
  - JNE：ZF = 0
  - JZ：ZF = 1
  - JNZ：ZF = 0
  - JG：ZF = 0 and SF = OF
  - JGE：SF = OF;
  - JL：SF != OF
  - JLE：ZF = 1，SF != OF
  - JA：无符号大于，CF=0 and ZF=0
  - JAE：无符号大于等于，ZF=0
  - JB：无符号小于，CF=1
  - JBE：无符号小于等于，CF=1 or ZF=1

其它与eflags标志位有关的指令

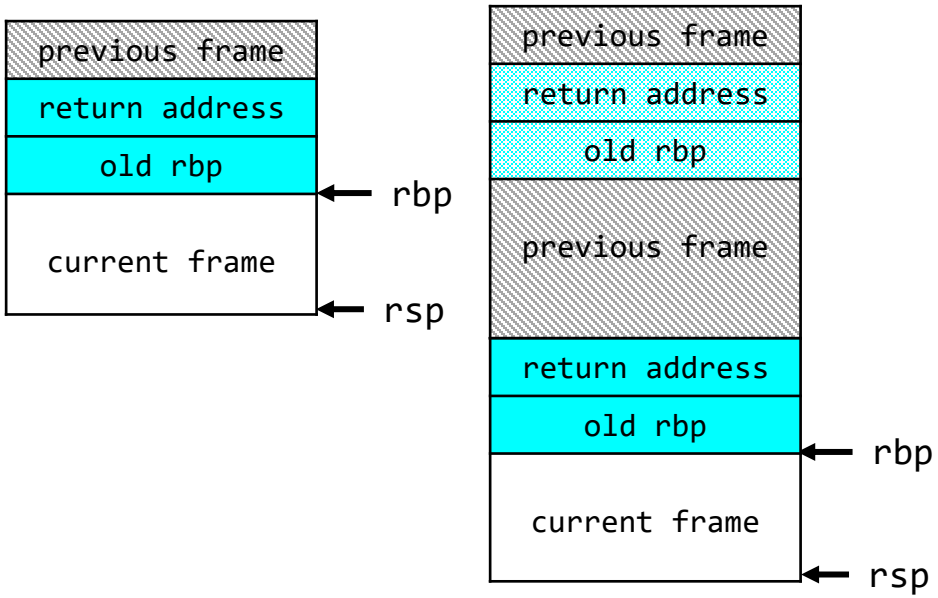
- 条件移动：
  - CMOV/CMOVNE/CMOVLE/CMOVL/CMOVZ/CMOVNZ/…
  - 源地址和目的地址必须都是寄存器
- 根据条件将目标寄存器设置0或1
  - SETE/SETNE/SETLE/SETL/SETZ/SETNZ/…
  - 寄存器只能是1byte的子寄存器，如al寄存器

```
SETE dst
SETGE dst
CMOVNS src, dst
CMOVLE src, dst
```

调用规约（System V AMD64 ABI）

```
PUSHQ   %rbp
MOVQ    %rsp, %rbp
SUBQ    $32, %rsp
MOVL    $0, -4(%rbp)
MOVL    %edi, -8(%rbp)
MOVQ    %rsi, -16(%rbp)
MOVL    $7, -20(%rbp)
MOVL    -20(%rbp), %edi
CALLQ   fibonacci
MOVL    %eax, -24(%rbp)
MOVL    -24(%rbp), %eax
ADDQ    $32, %rsp
POPQ    %rbp
RETQ
```

```
int main(int argc, char** argv){
  int n = 7;
  int r = fibonacci(n);
  return r;
}
```



- 传参使用的寄存器：
  - rdi/rsi/rdx/rcx/r8/r9
  - 超过6个放栈上
  - 浮点数参数使用xmm0-7
- 返回值使用的寄存器
  - rax/rdx（超过64bit）
  - 浮点数使用xmm0-1
- callee-saved寄存器
  - 用完必须还原
  - rbx/rbp/rsp/r12/r13/r14/r15

数据在内存中的管理

- 常量数据：放在数据区，可直访问
  - `int a[5] = {1,2,3,4,5}`
  - `char* s = "const chars"`
- 栈：函数中变量，函数退出自动销毁
  - `int i = 1;`
  - `int *j;`
  - `int a[5] = {1,2,3,4,5}`
  - `char* s = "const chars"`
- 堆：如malloc申请的空间，需主动free释放

例子

```
char *global_var = "global chars";
void mem(int x){
  int i = 1;
  int* j = &i;
  int a[] = {1,2,3,4,5};
  char *local_var = "local chars";
  local_var = global_var;
  int* k = (int *) malloc (sizeof(int));
  *k = 3;
}
```

```
PUSHQ   %rbp
MOVQ    %rsp, %rbp
SUBQ    $64, %rsp
MOVL    %edi, -4(%rbp)
MOVL    $1, -8(%rbp)
LEAQ    -8(%rbp), %rax
MOVQ    %rax, -16(%rbp)
MOVQ    .L__const.mem.a, %rax
MOVQ    %rax, -48(%rbp)
MOVQ    .L__const.mem.a+8, %rax
MOVQ    %rax, -40(%rbp)
MOVL    .L__const.mem.a+16, %ecx
MOVL    %ecx, -32(%rbp)
MOVABSQ $.L.str.1, %rax
MOVQ    %rax, -56(%rbp)
MOVQ    global_var, %rax
MOVQ    %rax, -64(%rbp)
MOVL    $4, %edi
CALLQ   malloc
MOVQ    %rax, -64(%rbp)
MOVQ    -64(%rbp), %rax
MOVL    $3, (%rax)
ADDQ    $64, %rsp
POPQ    %rbp
RETQ
```

数据分布

```
PUSHQ   %rbp
MOVQ    %rsp, %rbp
SUBQ    $64, %rsp
MOVL    %edi, -4(%rbp)
MOVL    $1, -8(%rbp)
LEQA    -8(%rbp), %rax
MOVQ    %rax, -16(%rbp)
MOVQ    .L__const.mem.a, %rax
MOVQ    %rax, -48(%rbp)
MOVQ    .L__const.mem.a+8, %rax
MOVQ    %rax, -40(%rbp)
MOVL    .L__const.mem.a+16, %ecx
MOVL    %ecx, -32(%rbp)
MOVABSQ $.L.str.1, %rax
MOVQ    %rax, -56(%rbp)
MOVQ    global_var, %rax
MOVQ    %rax, -64(%rbp)
MOVL    $4, %edi
CALLQ   malloc
MOVQ    %rax, -64(%rbp)
MOVQ    -64(%rbp), %rax
MOVL    $3, (%rax)
ADDQ    $64, %rsp
POPQ    %rbp
RETQ
```

```
    .type   .L.str,@object          # @.str
    .section  .rodata.str1.1,"aMS",@progbits,1
.L.str:
    .asciz  "global chars"
    .size   .L.str, 13

    .type   global_var,@object      # @global_var
    .data
    .globl  global_var
    .p2align        3
global_var:
    .quad   .L.str
    .size   global_var, 8

    .type   .L__const.mem.a,@object # @__const.mem.a
    .section  .rodata,"a",@progbits
    .p2align        4
.L__const.mem.a:
    .long   1                       # 0x1
    .long   2                       # 0x2
    .long   3                       # 0x3
    .long   4                       # 0x4
    .long   5                       # 0x5
    .size   .L__const.mem.a, 20

    .type   .L.str.1,@object        # @.str.1
    .section  .rodata.str1.1,"aMS",@progbits,1
.L.str.1:
    .asciz  "local chars"
    .size   .L.str.1, 12
```

函数调用和栈操作

- 函数调用：CALL
  - 64位：CALLQ
- 函数返回：RET
  - 64位：RETQ
- 压栈：PUSH/PUSHQ
  - 将参数压栈，同时修改rsp地址
  - SUB $0x04, %rsp
- 出栈：POP/POPQ
  - 将参数出栈，同时修改rsp地址
  - ADD $0x04, %rsp

交换指令

- 交换两个操作数：XCHG
  - 参数可以是2个寄存器或1个寄存器+1个内存地址
  - 原子操作
  - 用于实现锁

```
XCHG dst, src
```

- 比较并交换操作数：CMPXCHG
  - 将al\ax\eax\rax中的值与首操作数比较：
  - 相等则将参数2的装载到参数1，zf置1；
  - 不等则参数1装载到al\ax\eax\rax，并将zf清0；
  - 实现原子操作需要lock前缀：lock cmpxchg

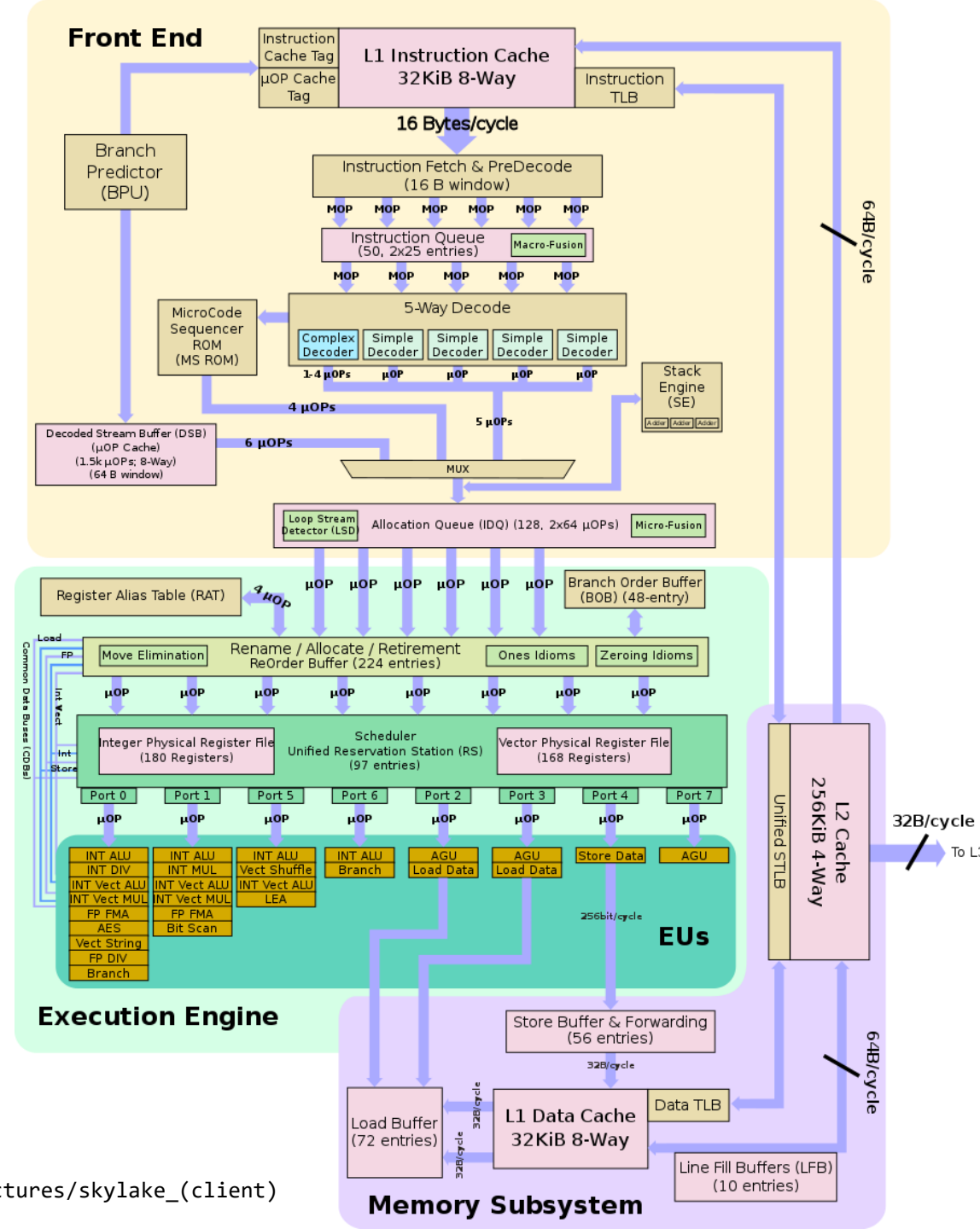
```
int val = 1;
do{
    __asm__("xchg %0, %1" : "+q" (val), "+m" (count));
} while(val - count == 0)
```

指令执行时间开销（SkylakeX）

	参数	时钟	参数	时钟	参数	时钟
MOV	r, r	0.25	m, r32/r64	0.5	r, m	1
LEA	m,r32/r64	0.5	m, r16	1		
ADD/SUB	r, r	0.25	m, r	0.5	r, m	1
AND/OR/XOR	r, r	0.25	m, r	0.5	r, m	1
SHL/SHR	i,r	0.5	i, m	2	cl,r	4
IMUL	r32	1	m32	2	r, r	1
MUL	r32	1	m32	2	r, r	
IDIV	r32	6	r64	24-90		
DIV	r32	6	r64	21-83		
SHL/SHR	i,r	1	i, m	2		
JMP	near/short	1-2	r	2	m	2
JE/JGE/JL	near/short	0.5-2				
CALL	near	3	r	2	m	3
RET		1	i	2		
PUSH	r/m/i	1				
POP	r	0.5	m	1	stack ptr	3
XCHG	r, r	1	m, r	20		

https://www.agner.org/optimize/instruction_tables.pdf

X86-64图解（Skylake）



https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

大纲

一、指令集和汇编代码

二、指令选择和翻译

三、指令调度算法

如何将中间代码翻译为汇编代码？

- 基本思路：模式匹配

```
define i32 @foo(i32 %a) {
  %a = stackalloc(i32);
  %1 = load i32 %a;
  %2 = load i32, @gvar;
  %3 = add i32 %1, %2
  %4 = mul i32 %3, 2
  ret i32 %4
}
```

```
PUSHQ   %rbp
MOVQ    %rsp, %rbp
MOVL    %edi, -4(%rbp)
MOVL    -4(%rbp), %eax
ADDL    .gvar, %eax
SHLL    $1, %eax
MOVL    %eax, -8(%rbp)
MOVL    -8(%rbp), %eax
POPQ    %rbp
RETQ
```

```
define void @bar(i64 %a) {
  %a = stackalloc(i32);
  %1 = load i32 %a;
  %b = stackalloca [100 x i32];
  %2 = getptr i32*, %b, %1;
  store i32 99, %2;
  ret void
}
```

```
PUSHQ   %rbp
MOVQ    %rsp, %rbp
SUBQ    $288, %rsp
MOVL    %edi, -4(%rbp)
MOVSLQ  -4(%rbp), %rax
MOVL    $99, -416(%rbp,%rax,4)
ADDQ    $288, %rsp
POPQ    %rbp
RETQ
```

指令选择的目标和挑战

- 将IR代码翻译为功能等价的汇编代码
  - 性能：代码体积小，运行速度快
- 单条IR指令如何选择对应的汇编指令？
- 一组IR指令应如何分割或合并翻译？
- 不考虑CPU流水线、乱序执行等因素

load指令

IR指令模式	汇编指令	开销	备注
load(%1)	MOV (%1), %r	1	
load(%a)	MOV -a(%rbp), %r	1	
load(@a)	MOV i, %r	1	
load(add(%1,i))	MOV i(%1), %r	1	
load(add(%1,%2))	MOV %1(%2), %r	1	
load(add(mul(%1,i),%2))	MOV (%2,%1,i), %r	1	i=1/2/4/8
load(add(add(mul(%1,i),%2),j))	MOV j(%2,%1,i), %r	1	i=1/2/4/8
...			

store指令

IR指令模式	汇编指令	开销	备注
store(i,%a)	MOV i, -a(%rbp)	1	
store(i,add(%1,j))	MOV i, j(%1)	1	
store(i,add(mul(%1,i),%2))	MOV i, (%2,%1,i)	1	
store(i,add(add(mul(%1,j),%2),k))	MOV i, k(%2,%1,j)	1	
store(%1,%a)	MOV %1, -x(%rbp)	1	
store(%1,add(%2,j))	MOV %1, j(%2)	1	
store(%1,add(mul(%2,i),%3))	MOV %1, (%3,%2,i)	1	
store(%1,add(add(mul(%2,j),%3),k))	MOV %1, k(%3,%2,j)	1	
store(load(%1),%a)	MOV (%1), -a(%rbp)	1	
...			

add指令

IR指令模式	汇编指令	开销	备注
add(%1,%2)	MOV %1, %r ADD %2, %r	2	
	LEA (%1,%2), %r	1	
add(%1,i)	MOV %1, %r ADD i, %r	2	
	LEAL i(%1), %r	1	
add(load(%a),%1)	MOV -a(%rbp), %r ADD %1, %r	2	
add(load(%a),i)	MOV -a(%rbp), %r ADD i, %r	2	
add(load(%a),load(%b))	MOV -a(%rbp), %r ADD -b(%rbp), %r	2	
add(mul(%1,i),%2)	LEA (%2,%1,i), %r	1	i=1/2/4/8
add(add(mul(%1,i),%2),j)	LEA j(%2,%1,i), %r	1	i=1/2/4/8
...			

mul/div指令

IR指令模式	汇编指令	开销	备注

mul(%1,%2)	MOV %1, %r1
	MOV %2, %r2
	MUL %r2
		5	低位：%r2
			高位：%r1

mul(%1,i)	MOV i, %r1
	MOV %1, %r2
	MUL %r2
		5	

div(%1,%2)	MOV %1, %r1
	MOV 0, %r2
	MOV %2, %r3
	DIV %r3
		10	商：%r1
			余数：%r2

...			

一组IR指令的翻译问题

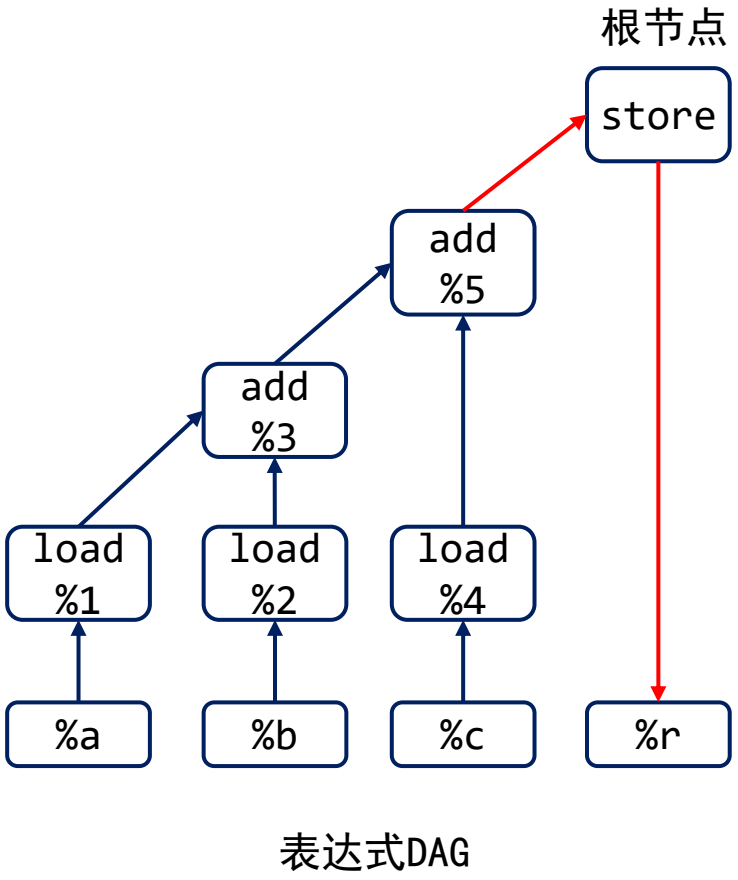
- 输入中间代码块
  - 表达式图：有向无环图DAG
- 输出汇编代码，性能目标：
  - 指令少
  - 运算快

IR=>DAG

- 将IR转换为表达式树
- 合并表达式树的共同节点得到表达式图DAG

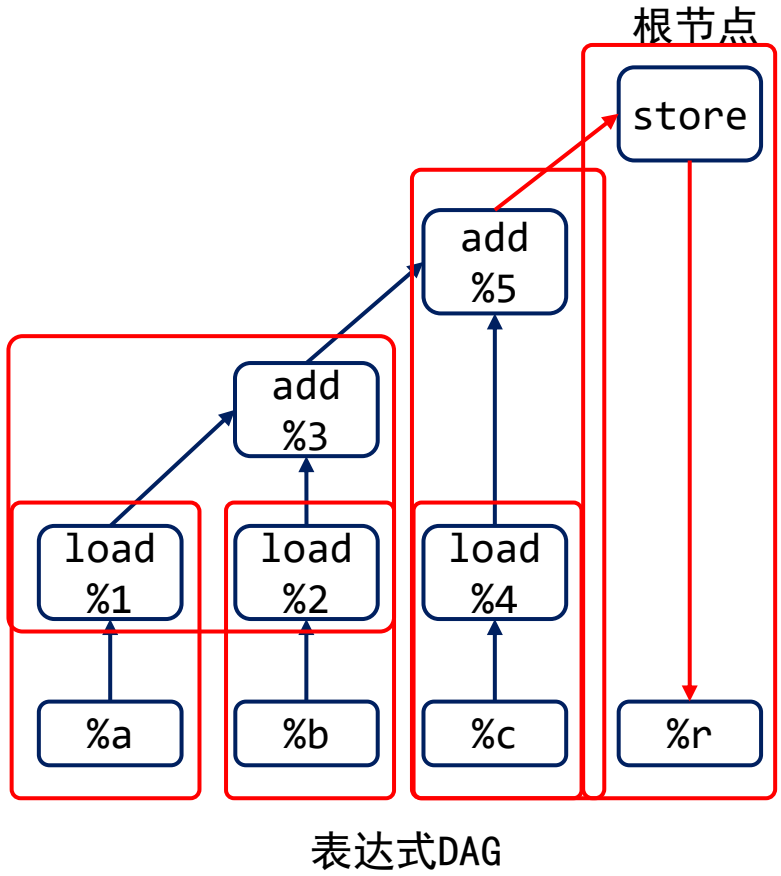
```
r = a + b + c;
```

```
%1 = load i32 %a;
%2 = load i32 %b;
%3 = add i32 %1, %2;
%4 = load i32 %c;
%5 = add i32 %3, %4;
store i32 %5, %r;
```



表达式DAG

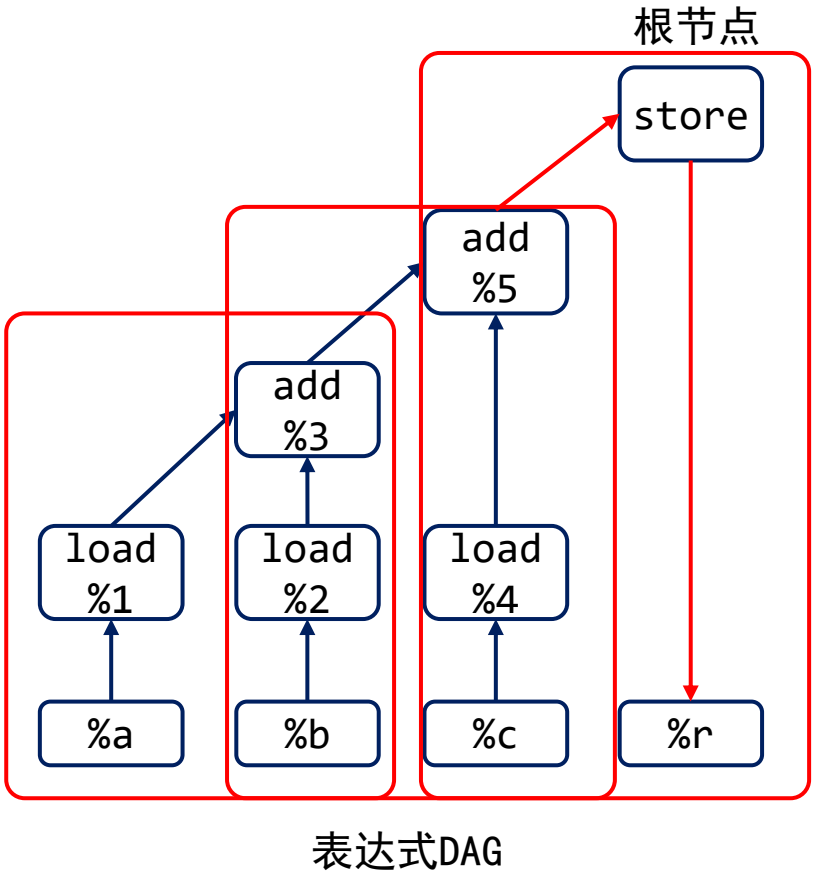
指令选择=>铺树问题



表达式DAG

```
MOV -a(%rbp), %r1
MOV -b(%rbp), %r2
ADD %r1, %r2
MOV -c(%rbp), %r3
ADD %r2, %r3
MOVE %r3, -r(%rbp)
```

指令选择=>铺树问题



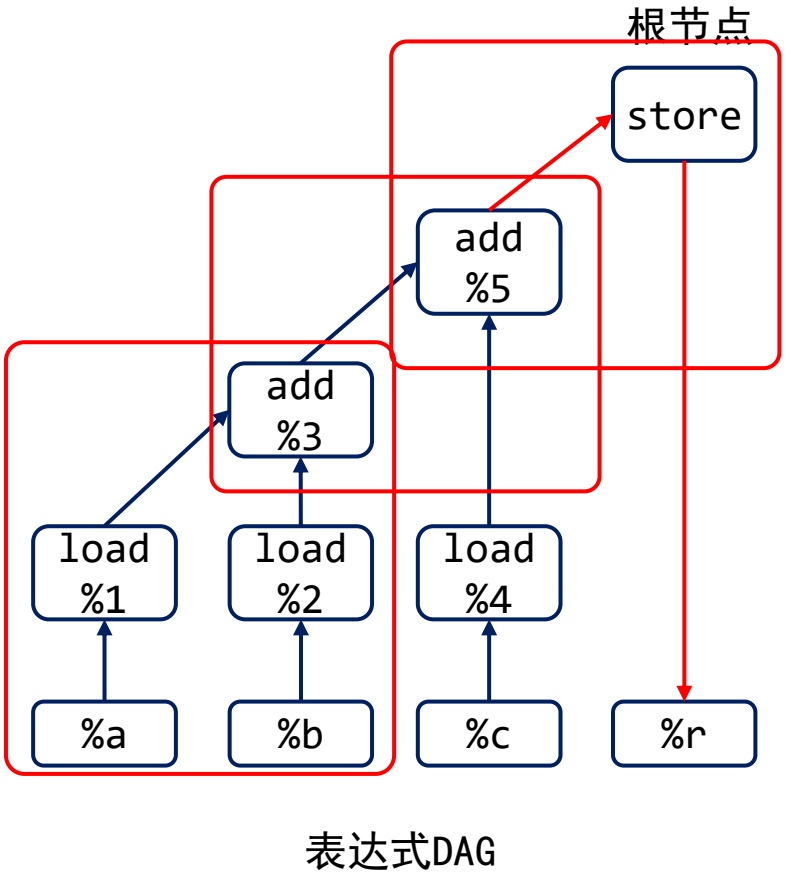
表达式DAG

```
MOV -a(%rbp), %r1
ADD -b(%rbp), %r1

MOV -c(%rbp), %r2
ADD %r1, %r2

MOV %r2, -r(%rbp)
```

铺树问题

- 如何铺树使得最终的汇编代码：
  - 体积小（指令数少）
  - 运算快
- 贪心算法：Maximal Munch
  - 从树根开始，每次选择覆盖节点最多、开销最低的规则
  - 逆序生成汇编指令
  - 局部最优
- 动态规划
  - 从树根开始，递归搜索每个节点的最优方案

Maximal Munch



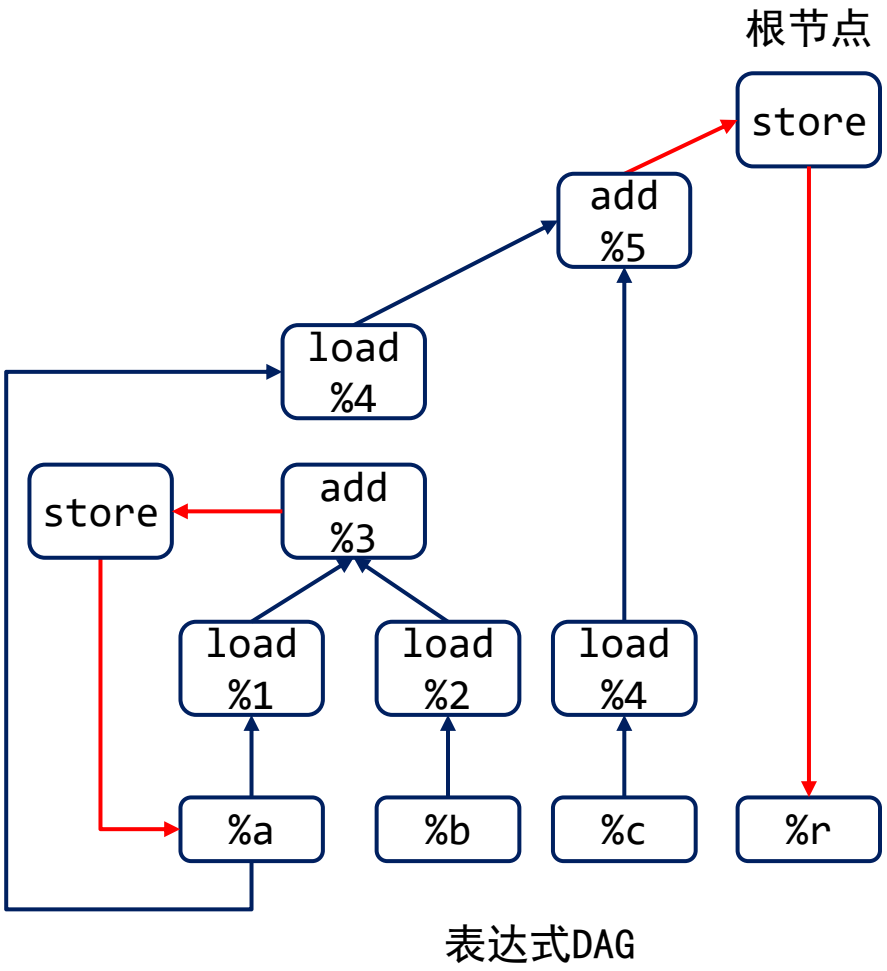
表达式DAG

```
MOV -a(%rbp), %r1
ADD -b(%rbp), %r1

MOV -c(%rbp), %r2
ADD %r1, %r2

MOV %r2, -r(%rbp)
```

多个load的情况

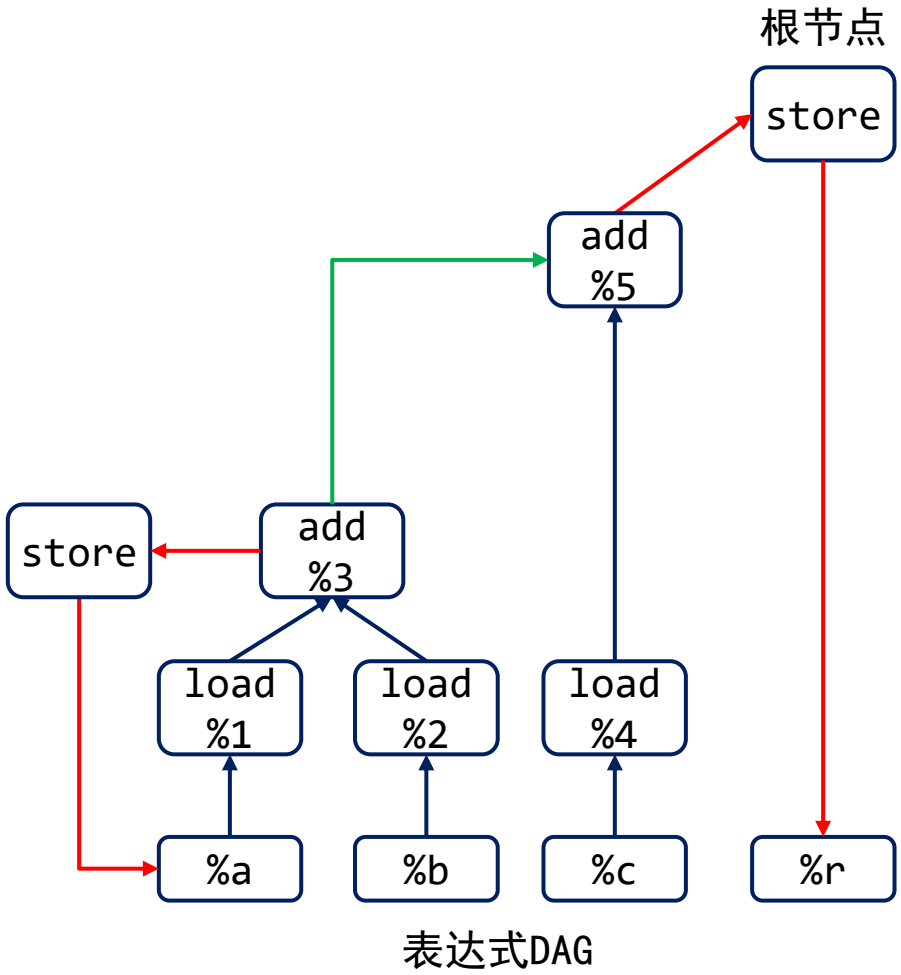
```
a = a + b;
r = a + c;
```

```
%1 = load i32 %a;
%2 = load i32 %b;
%3 = add i32 %1, %2;
store i32 %3, %a;
%4 = load i32 %a;
%5 = load i32 %c;
%6 = add i32 %4, %5;
store i32 %6, %r;
```



表达式DAG

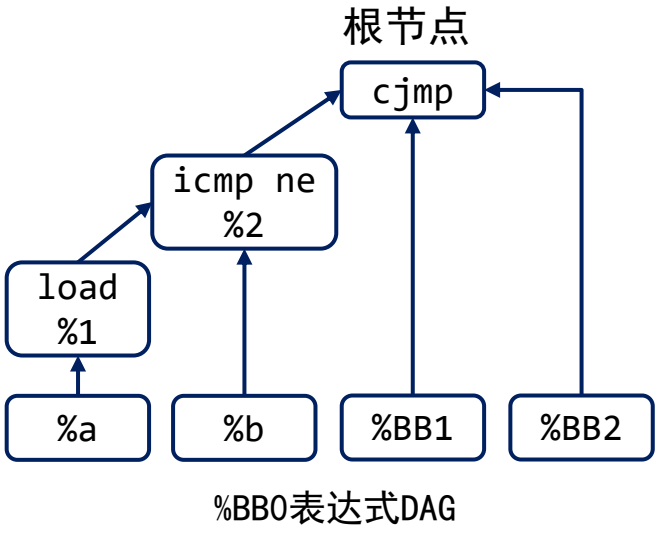
多个load的情况



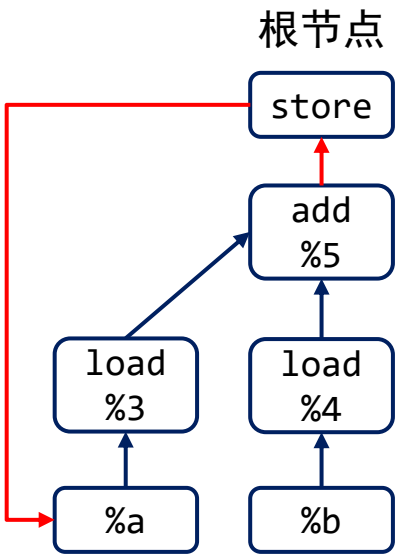
控制流

```
if(a==0)
    a = a + b;
int r = a + c;
```

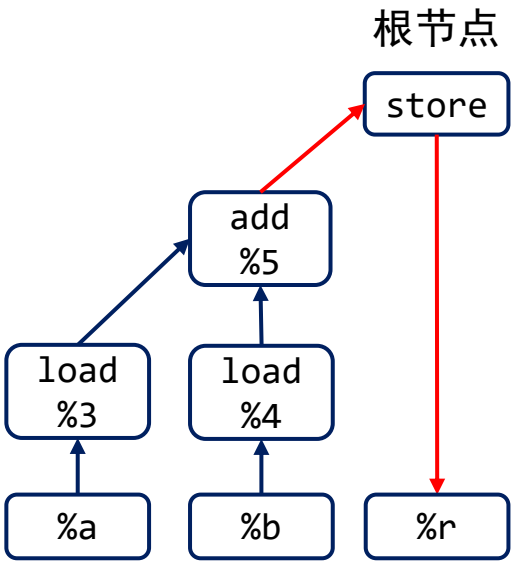
```
%BB0:
    %1 = load i32, i32* %a;
    %2 = icmp eq i32 %8, 0;
    cjmp i1 %2, %BB1, %BB2;
%BB1:
    %3 = load i32, i32* %a;
    %4 = load i32, i32* %b;
    %5 = add i32 %3, %4;
    store i32 %5, %a;
    jmp %BB2;
%BB2:
    %6 = load i32, %a;
    %7 = load i32, %b;
    %8 = add i32 %6, %7;
    store i32 %8, %r;
```



%BB0表达式DAG



%BB1表达式DAG



%BB2表达式DAG

指针

```
int a[];
int* b;
...
a[i] = *b;
```

```
%1 = load i32, %i;
%2 = getptr i32* %a, %1;
%3 = getptr i32* %b, 0;
%4 = load i32 %3
store i32 %4, %2;
```

IR指令模式	汇编指令	开销	备注
getptr i8* %a, %1;	MOV -a(%rbp), %r ADD %1, %r	2	

getptr i32* %a, %1;	MOV -a(%rbp), %r
	MUL %1, %r
	ADD %1, %r		

	MOV -a(%rbp), %r LEA (%r,%1,4), %r	2	
...			

大纲

一、指令集和汇编代码

二、指令选择和翻译

三、指令调度算法

指令调度问题

- 指令的执行时间与其执行顺序密切相关
  - 满足数据依赖
  - CPU并行处理能力
- 编译器的指令调度一般只考虑数据依赖关系
- 指令调度的目标：
  - 在单位时间内执行更多的操作
  - 使用更少的寄存器

流水线（Instruction pipelining）

- 经典5-stage流水线
  - Instruction Fetch
  - Instruction Decode
  - Execute
  - Memory Access
  - Write Back

```
ADD %r1, %r2
SUB %r1, %r3
```

Stage	Clock Cycles					
	1	2	3	4	5	6
Fetch	ADD	SUB				
Decode		ADD	SUB			
Execute			ADD	SUB		
Access				ADD	SUB	
Write					ADD	SUB

数据依赖和乱序执行

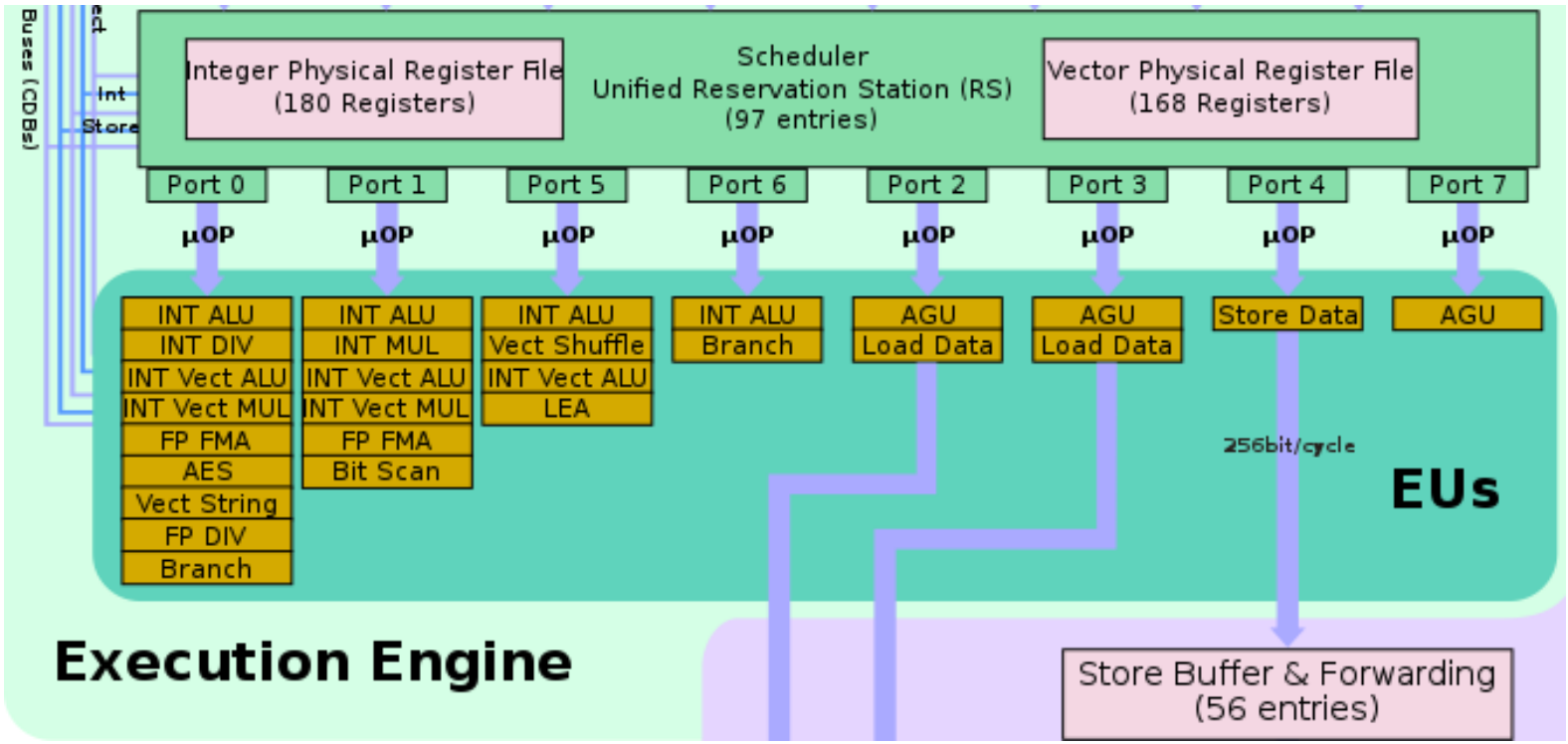
- 防止数据依赖造成的CPU空闲，可以优先执行后面的指令。
- CPU级别的指令调度机制

```
ADD %r1, %r2
SUB %r2, %r3
MUL %r1, %r4
```

Stage	Clock Cycles								
	1	2	3	4	5	6	7	8	9
Fetch	ADD	SUB	MUL						
Decode		ADD	SUB	MUL					
Execute			ADD			SUB	MUL		
Access				ADD			SUB	MUL	
Write					ADD			SUB	MUL

超标量处理器（superscalar）

- 指令级并行（Instruction-level Parallel）
  - 一个周期可以分派多条指令
  - 流水线stages数量15~20
- 每个指令由多个微指令（μOP）组成
- 通过调度器和一组ports实现
  - 不同ports支持的微指令存在一定区别



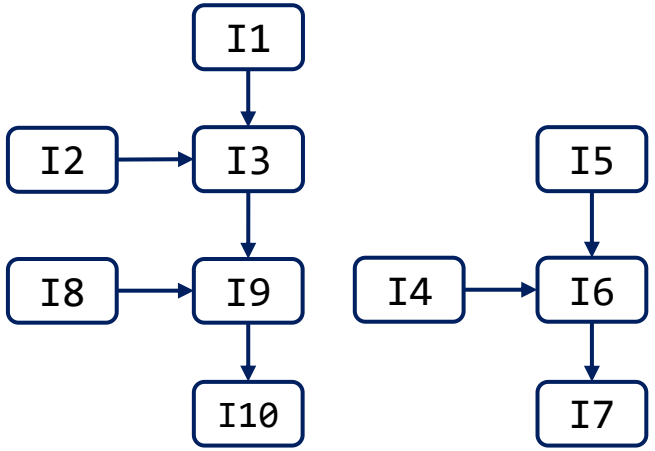
影响性能的因素

- 数据依赖关系（data ependency ）
  - 写-读依赖（true-dependency）
  - 读-写反依赖（anti-denpendency）
- 结构性影响（structural hazard）
  - 一条指令由多条微指令组成
  - 相邻指令的微指令可能会竞争ports的使用
- 控制流影响（control hazard）
  - 条件跳转或分支预测

指令依赖关系

- 场景：单个程序块，无跳转指令
- 如果指令I2使用I1的结果，那么I2依赖I1
- 叶子节点没有任何依赖，可以尽早执行
  - I1、I2、I4、I7

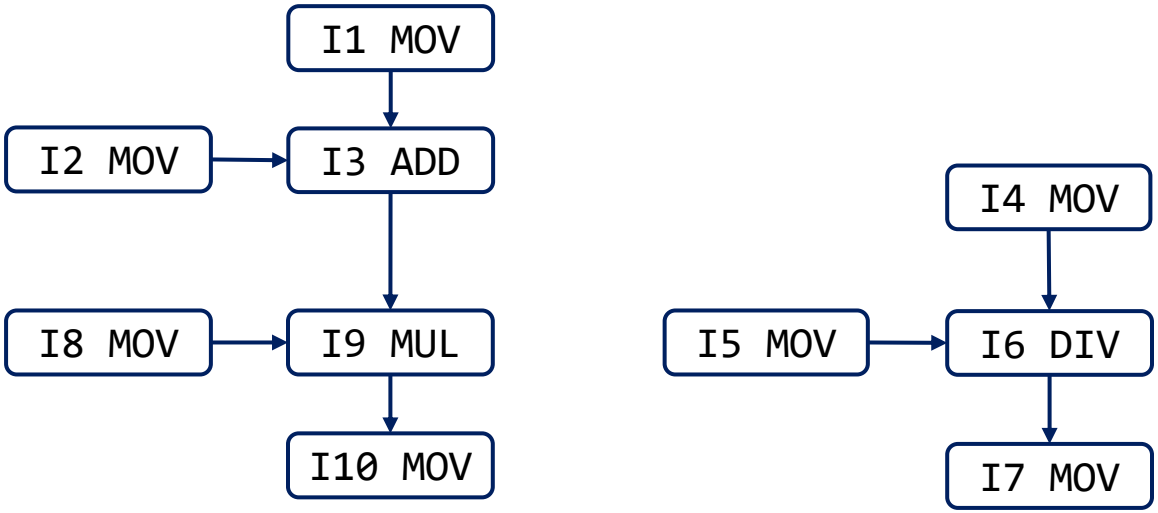
I1	MOV -12(%rsp), %r1
I2	MOV -16(%rsp), %r2
I3	ADD %r2, %r1
I4	MOV -20(%rsp), %r2
I5	MOV -24(%rsp), %eax
I6	DIV %r2, ~~%eax~~
I7	MOV %eax, -24(%rsp)
I8	MOV -28(%rsp), %r2
I9	MUL %r1, %r2
I10	MOV %r2, -28(%rsp)



指令依赖关系

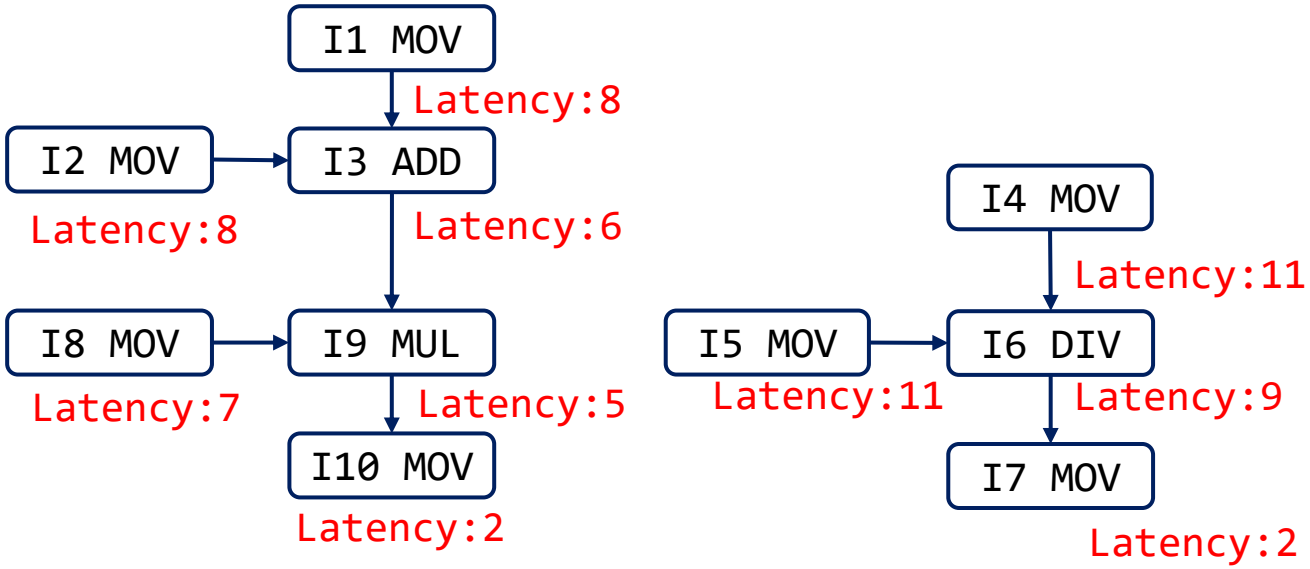
编译器的指令调度问题

- 假设每个cycle可以发射一条指令，多条指令可以并行
- 假设指令开销如下，应如何确定最佳的指令执行序列？
  - MOV 2
  - ADD 1
  - MUL 3
  - DIV 7
- 指令执行序列应满足指令依赖关系



指令调度思路

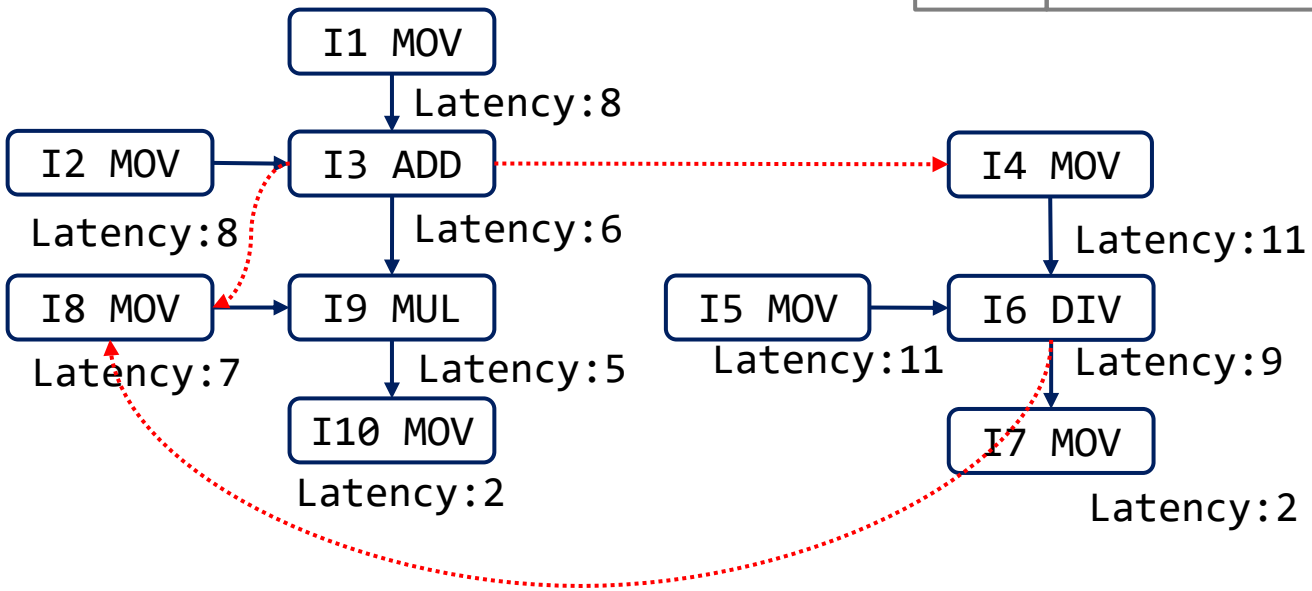
- 搜索需要尽早执行的关键指令
- 计算每条指令开始执行后，序列执行结束所需时间（latency）
  - 假设$i = v.next$， $L(v) = E_v + L(i)$
- 优先执行Latency大的指令
  - 根据latency从大到小对指令进行排序
  - I4=I5>I6>I1=I2>I8>I3>I9>I7=I10



读-写反依赖问题

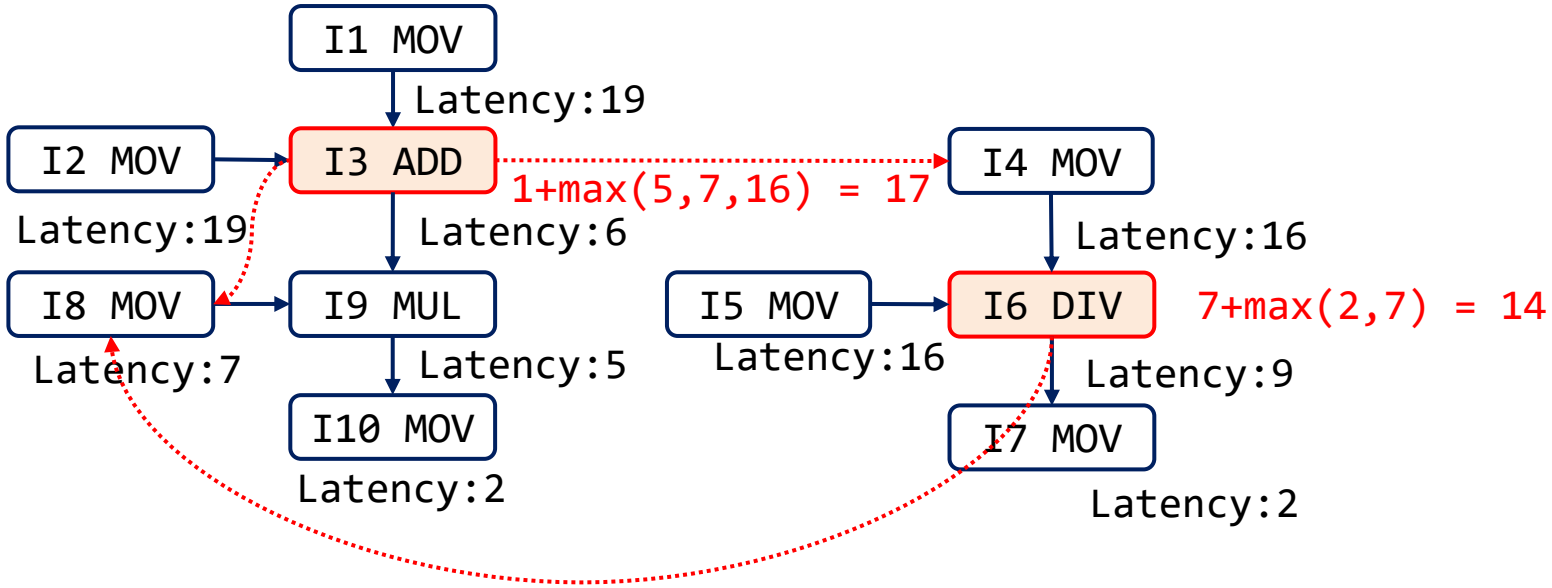
- I3执行完I4和I8才能执行
  - 否则会影响I3的计算结果
- I6执行完才能执行I8

I1	MOV -12(%rsp), %r1
I2	MOV -16(%rsp), %r2
I3	ADD %r2, %r1
I4	MOV -20(%rsp), %r2
I5	MOV -24(%rsp), %eax
I6	DIV %r2, ~~%eax~~
I7	MOV %eax, -24(%rsp)
I8	MOV -28(%rsp), %r2
I9	MUL %r1, %r2
I10	MOV %r2, -28(%rsp)



更新Latency和执行序列

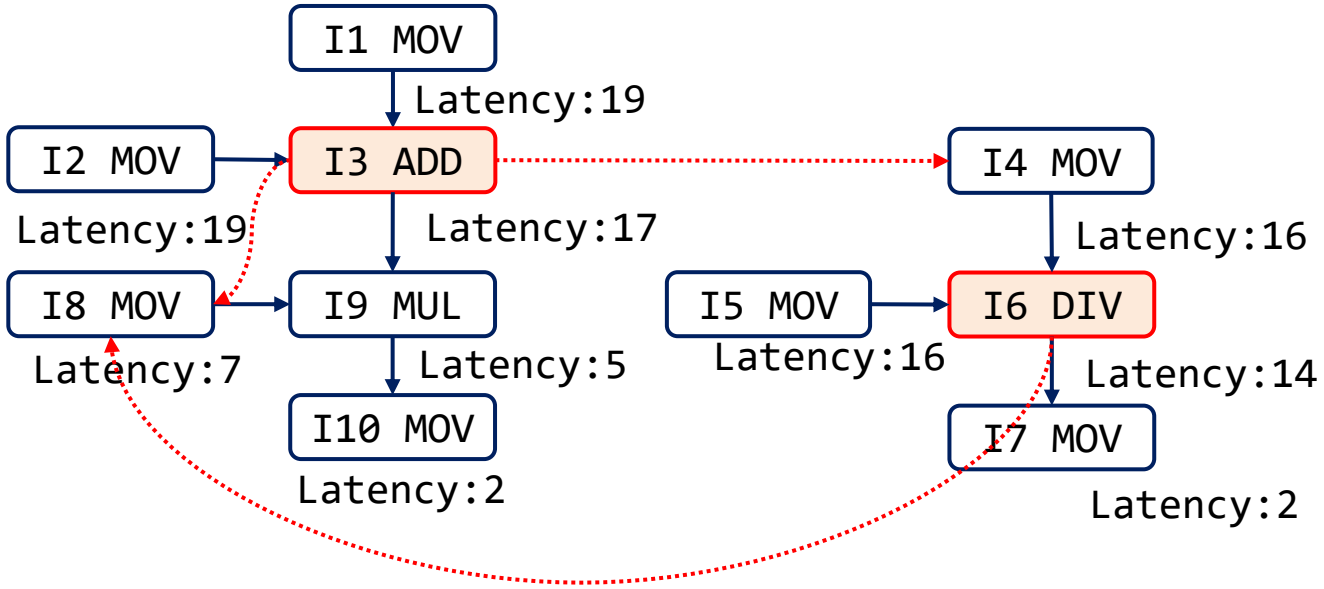
- $\forall i \in v.next, L(v) = E_v + Max(L(i))$
- 新序列：I1=I2>I3>I4=I5>I6>I8>I9>I7=I10



调度方案开销

- I1=I2>I3>I4=I5>I6>I8>I9>I7=I10
  - 开销：21

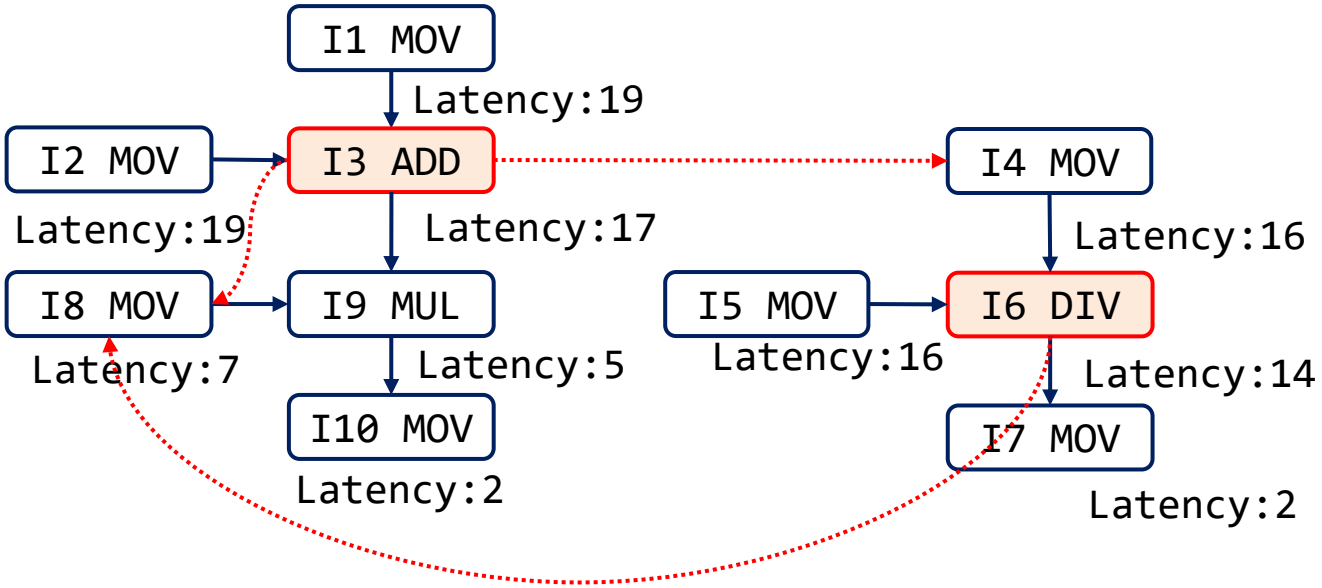
开始	结束	指令	
1	2	I1	MOV -12(%rsp), %r1
2	3	I2	MOV -16(%rsp), %r2
4	4	I3	ADD %r2, %r1
5	6	I4	MOV -20(%rsp), %r2
6	7	I5	MOV -24(%rsp), %eax
8	14	I6	DIV %r2, ~~%eax~~
15	16	I8	MOV %eax, -24(%rsp)
17	19	I9	MOV -28(%rsp), %r2
18	19	I7	MUL %r1, %r2
20	21	I10	MOV %r2, -28(%rsp)



消除反依赖：重命名

I1	MOV -12(%rsp), %r1
I2	MOV -16(%rsp), %r2
I3	ADD %r2, %r1
I4	MOV -20(%rsp), %r2
I5	MOV -24(%rsp), %eax
I6	DIV %r2, ~~%eax~~
I7	MOV %eax, -24(%rsp)
I8	MOV -28(%rsp), %r2
I9	MUL %r1, %r2
I10	MOV %r2, -28(%rsp)

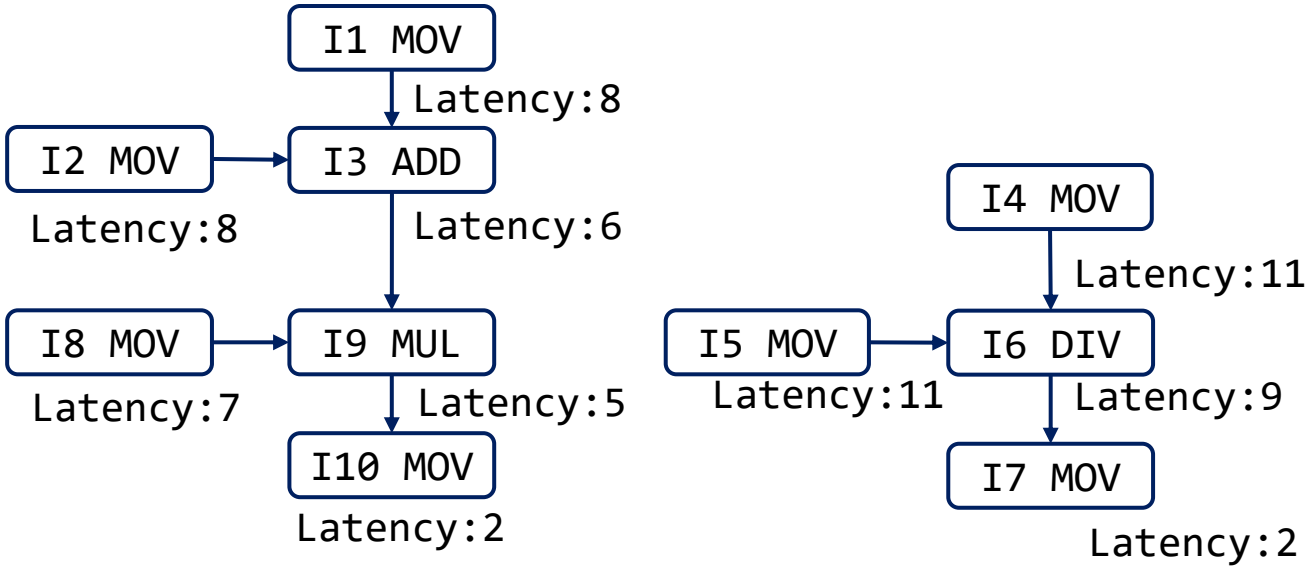
I1	MOV -12(%rsp), %r1
I2	MOV -16(%rsp), %r2
I3	ADD %r2, %r1
I4	MOV $-20(%rsp), %r3
I5	MOV $-24(%rsp), %eax
I6	DIV %r3, ~~%eax~~
I7	MOV %eax, -24(%rsp)
I8	MOV -28(%rsp), %r4
I9	MUL %r1, %r4
I10	MOV %r4, -28(%rsp)



调度方案开销

- I4=I5>I6>I1=I2>I8>I3>I9>I7=I10
  - 开销：14

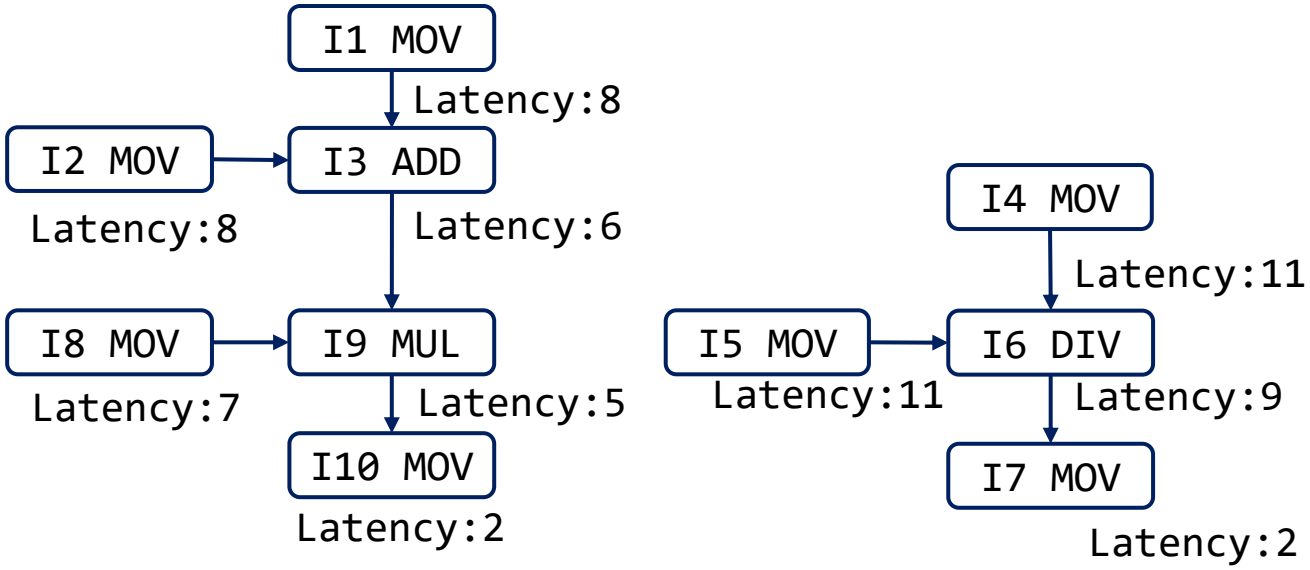
开始	结束	指令	
1	2	I4	MOV -20(%rsp), %r3
2	3	I5	MOV -24(%rsp), %eax
4	10	I6	DIV %r3, ~~%eax~~
5	6	I1	MOV -12(%rsp), %r1
6	7	I2	MOV -16(%rsp), %r2
8	8	I3	ADD %r2, %r1
9	10	I8	MOV -28(%rsp), %r4
11	13	I9	MUL %r1, %r4
12	13	I7	MOV %eax, -24(%rsp)
13	14	I10	MOV %r4, -28(%rsp)



进一步优化

- 可尽早执行已经满足了依赖的指令
- I1和I6互换，I7和I9互换
  - 开销：12

开始	结束	指令	
1	2	I4	MOV -20(%rsp), %r3
2	3	I5	MOV -24(%rsp), %eax
4	10	I6	DIV %r3, ~~%eax~~
5	6	I1	MOV -12(%rsp), %r1
6	7	I2	MOV -16(%rsp), %r2
8	8	I3	ADD %r2, %r1
9	10	I8	MOV -28(%rsp), %r4
11	13	I9	MUL %r1, %r4
12	13	I7	MOV %eax, -24(%rsp)
13	14	I10	MOV %r4, -28(%rsp)



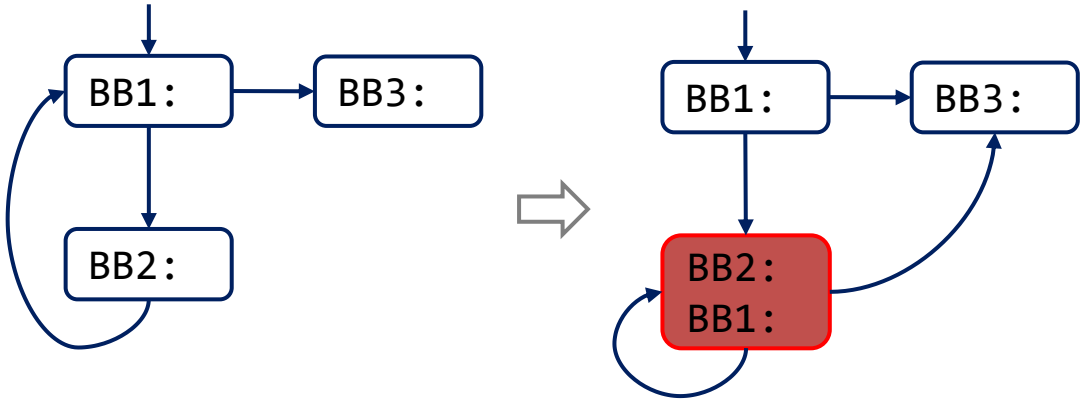
表调度算法

- 假设：
  - 线性代码
  - 无反依赖
- 两张表：
  - Ready表记录已满足数据依赖的指令
  - Active表记录正在执行的指令
- 算法：
  - 每个Clock尽量执行一条新的指令
  - 如果Active表有指令执行完成，将指令的next指令加入Ready

```
Clock = 1
Ready = {指令依赖图的所有叶子节点}
Active = {}
While (Ready ∪ Active ≠ ∅){
    foreach I in Active {
        if Start(I) + Cost(I) < Clock {
            remove I;
            foreach C in I.next {
                if C isReady
                    Ready.add(C);
            }
        }
    }
    if (Ready ≠ ∅){
        Ready.remove(any I);
        Start(I) = Clock;
        Active.add(I);
    }
    Clock = Clock + 1;
}
```

算法局限性

- 局部优化：代码块内部，未考虑全局信息
- 跨代码块的指令调度？
  - 思路举例：复制代码块，牺牲程序体积，提升运行速度



总结

- 中间代码翻译为汇编代码
- 指令选择：铺树问题
- 指令调度：不破坏数据依赖重排指令