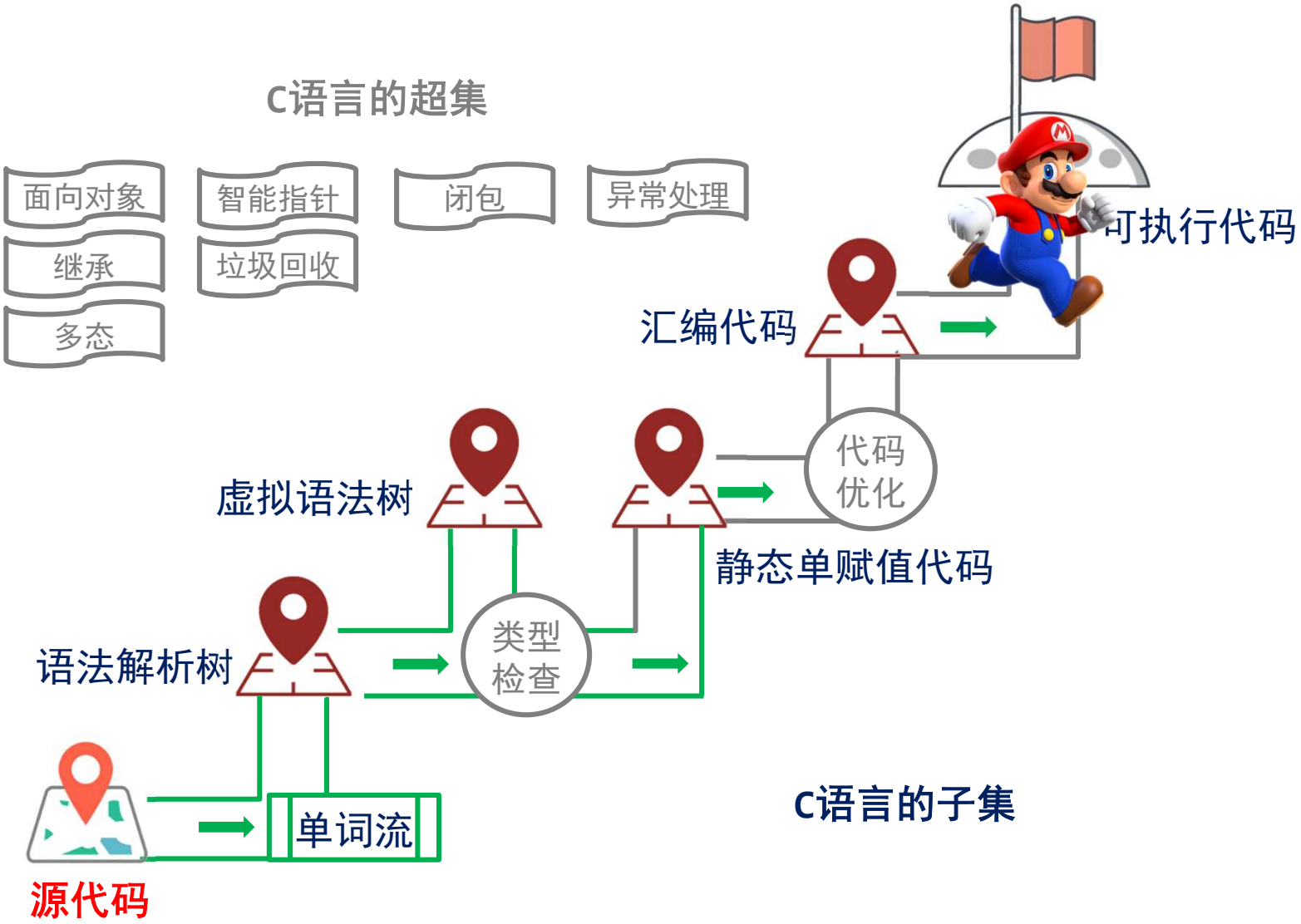
Lecture 5

汇编代码生成

徐 辉

xuh@fudan.edu.cn

学习地图


C语言的超集
面向对象
智能指针
闭包
异常处理
继承
垃圾回收
多态
可执行代码
汇编代码
代码
优化
虚拟语法树
静态单赋值代码
类型
检查
语法解析树
单词流
C语言的子集
源代码

LLVM SSA vs 汇编代码

- 还有哪些事要做？
  - IR指令=>汇编指令
  - 临时变量=>寄存器和栈

```
int phib(int a, int b){
    if(a) b++;
    return b;
}
```

```
define dso_local i32 @phib(i32 %0, i32 %1) #0 {
  %3 = icmp ne i32 %0, 0
  br i1 %3, label %4, label %6

4:
  %5 = add nsw i32 %1, 1
  br label %6

6:
  %.0 = phi i32 [ %5, %4 ], [ %1, %2 ]
  ret i32 %.0
}
```

```
# %bb.0:
        pushq   %rbp
        movq    %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
```

大纲

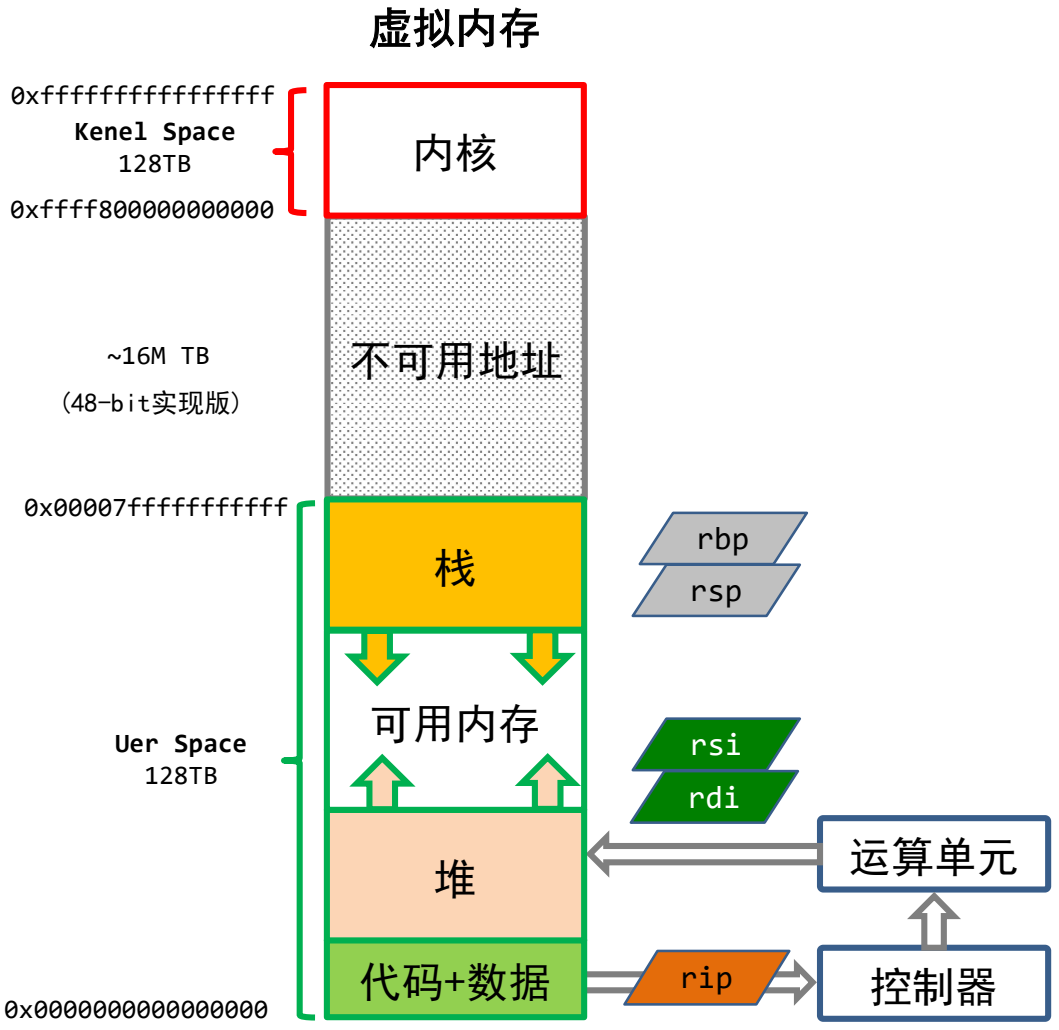
- 一、指令集和汇编代码
- 二、指令选择和翻译
- 三、指令调度算法
- 四、寄存器分配算法
- 五、LLVM的实现分析
- 六、组装可执行程序

一、指令集和汇编代码

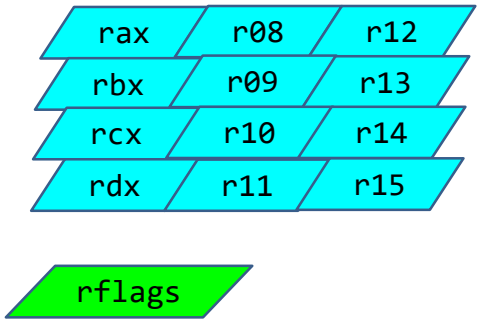
指令集

- 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, (%rax, %rcx, 8)
    - 地址 = %rbp-%rcx*8
  - 带位移和索引的：movl $1, 8(%rsp, %rdi, 4)
    - 地址 = %rbp+8+%rdi*4

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：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
movb %al, 0x409892
mov 8(%rsp), %eax
```

- 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
```

主要X86-64指令：整数运算

- 一般形式：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
- 主要运算
  - 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

```
add src, dst
and src, dst
shl count, dst
```

i:有符号的

主要X86-64指令：比较和跳转

- 比较指令：cmp/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

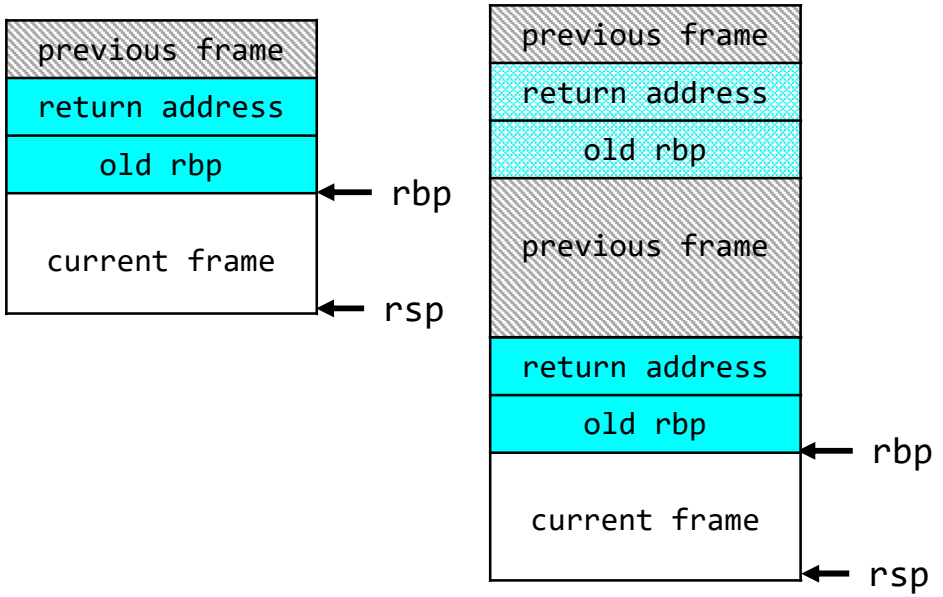
- 条件移动：
  - cmove/cmovne/cmovle/cmovl/cmovz/cmovnz/…
  - 源地址和目的地址必须都是寄存器
- 根据条件将目标寄存器设置0或1
  - sete/setne/setle/setl/setz/setnz/…
  - 寄存器只能是1byte的子寄存器，如al寄存器

```
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)
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
```

```
    .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

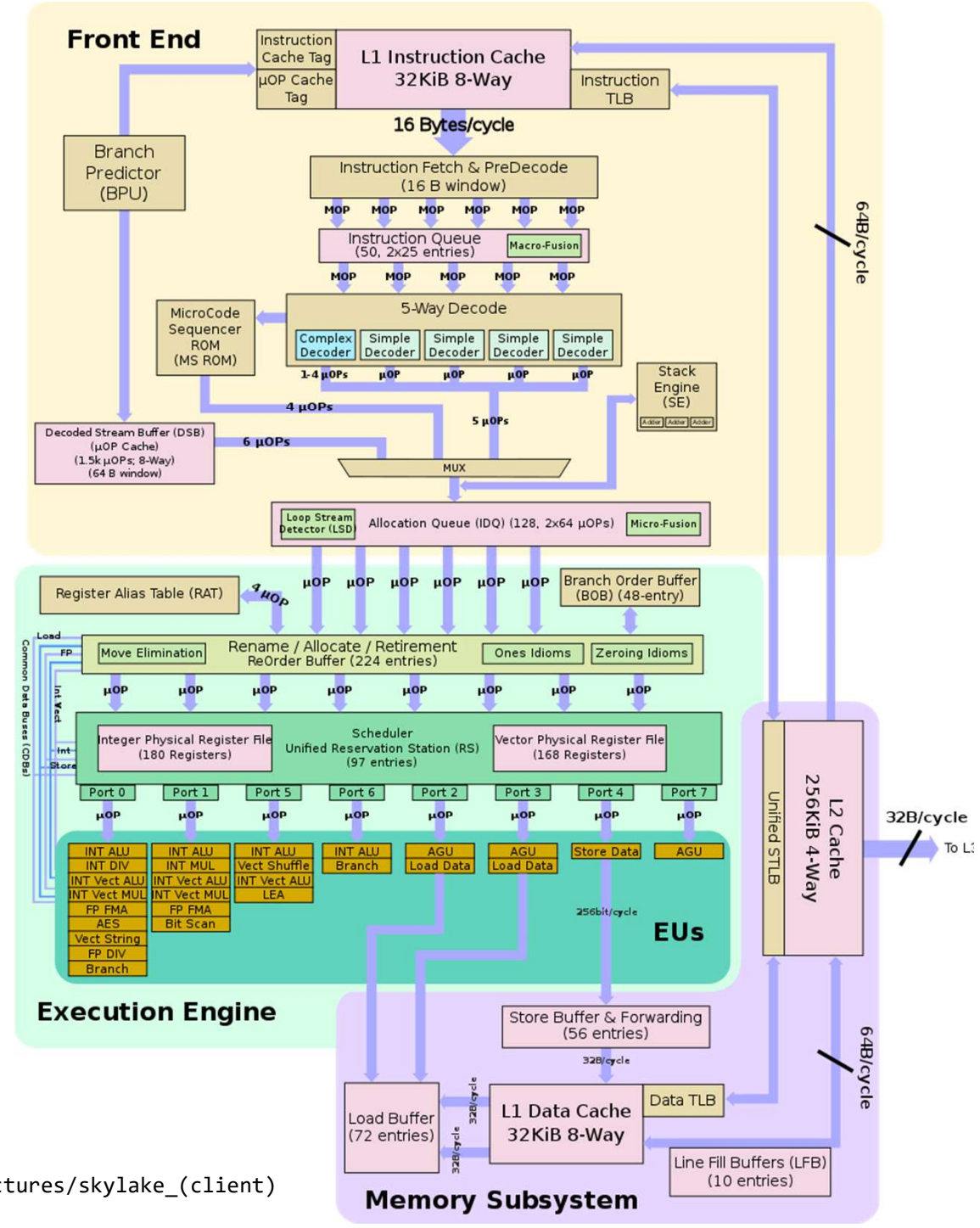
```
int val = 1;
do{
    __asm__("xchg %0, %1" : "+q" (val), "+m" (count));
} while(val - count == 0)
```

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)

二、指令选择和翻译

如何将中间代码翻译为汇编代码？

- 基本思路：模式匹配

```
define dso_local i32 @ident(i32 %0) #0 {
  %2 = load i32, i32* @global_var, align 4
  %3 = add nsw i32 %0, %2
  %4 = mul nsw i32 %3, 2
  ret i32 %4
}
```

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -4(%rbp)
movl    -4(%rbp), %eax
addl    global_var, %eax
shll    $1, %eax
movl    %eax, -8(%rbp)
movl    -8(%rbp), %eax
popq    %rbp
retq
```

```
define dso_local void @array(i32 %0) #0 {
  %2 = alloca [100 x i32], align 16
  %3 = sext i32 %0 to i64
  %4 = getelementptr inbounds [100 x i32],
          [100 x i32]* %2, i64 0, i64 %3
  store i32 99, i32* %4, align 4
  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
```

更多例子

```
int a[] = {1,2,3,4,5};
int* k = (int *) malloc (sizeof(int)*2);
k[1] = 3;
```

```
%2 = alloca [5 x i32], align 16
%3 = bitcast [5 x i32]* %2 to i8*
call void @llvm.memcpy.p0i8.p0i8.i64
    (i8* align 16 %3,
     i8* align 16 bitcast ([5 x i32]*
      @__const.mem.a to i8*),
    i64 20, i1 false)
%5 = call i8* @malloc(i64 8)
%6 = bitcast i8* %5 to i32*
%7 = getelementptr inbounds i32,
       i32* %6, i64 1
store i32 3, i32* %7, align 4
```

```
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)

movl    $8, %edi
callq   malloc
movq    %rax, -64(%rbp)

movq    -64(%rbp), %rax
movl    $3, 4(%rax)
```

指令选择的目标和挑战

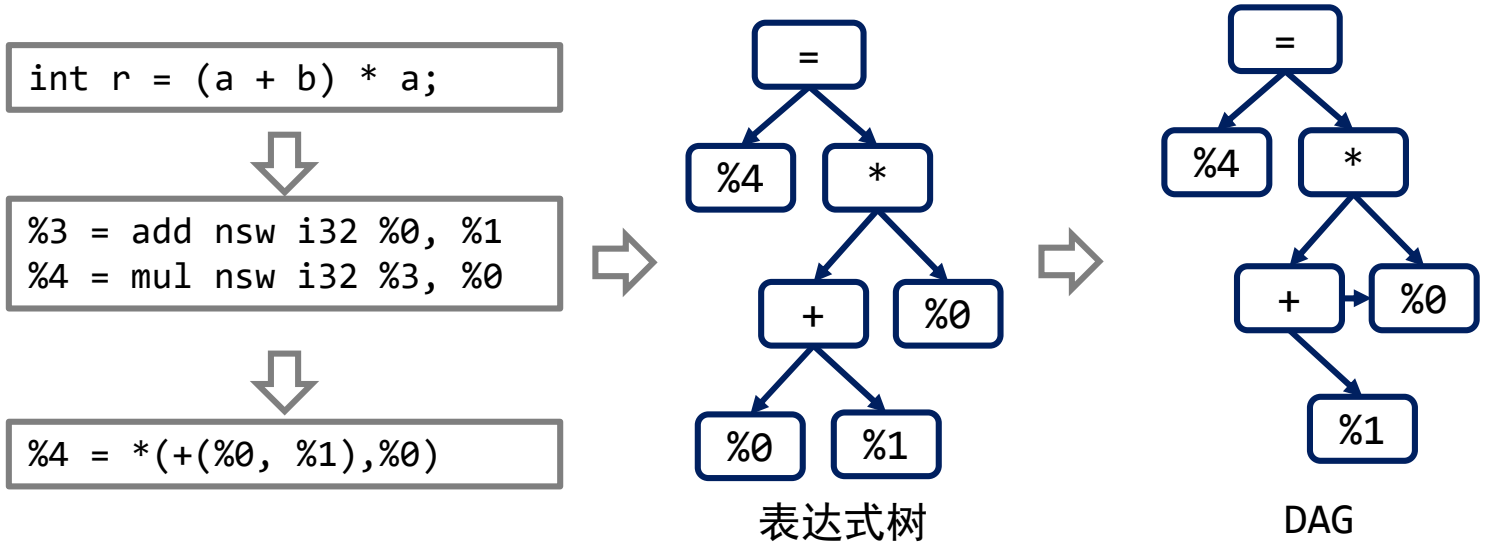
- 目标：
  - 汇编代码与IR代码语义等价
  - 性能：代码体积小，运行速度快
- 问题1：单条IR指令如何选择对应的汇编指令？
  - 有多种选择：如MUL vs SHL；
  - 相对容易选择。
- 问题2：一组IR指令应如何分割或合并翻译？
  - 如getelementptr + store
  - 思路1：peephole优化
  - 思路2：转化为铺树问题
- 暂时不考虑寄存器、流水线、乱序执行等机制带来的性能影响。

指令选择问题（问题2）

- 输入中间代码
  - 表达式树（expression tree）或有向无环图DAG
- 输出汇编代码，性能目标：
  - 体积小（指令数少）
  - 运算快

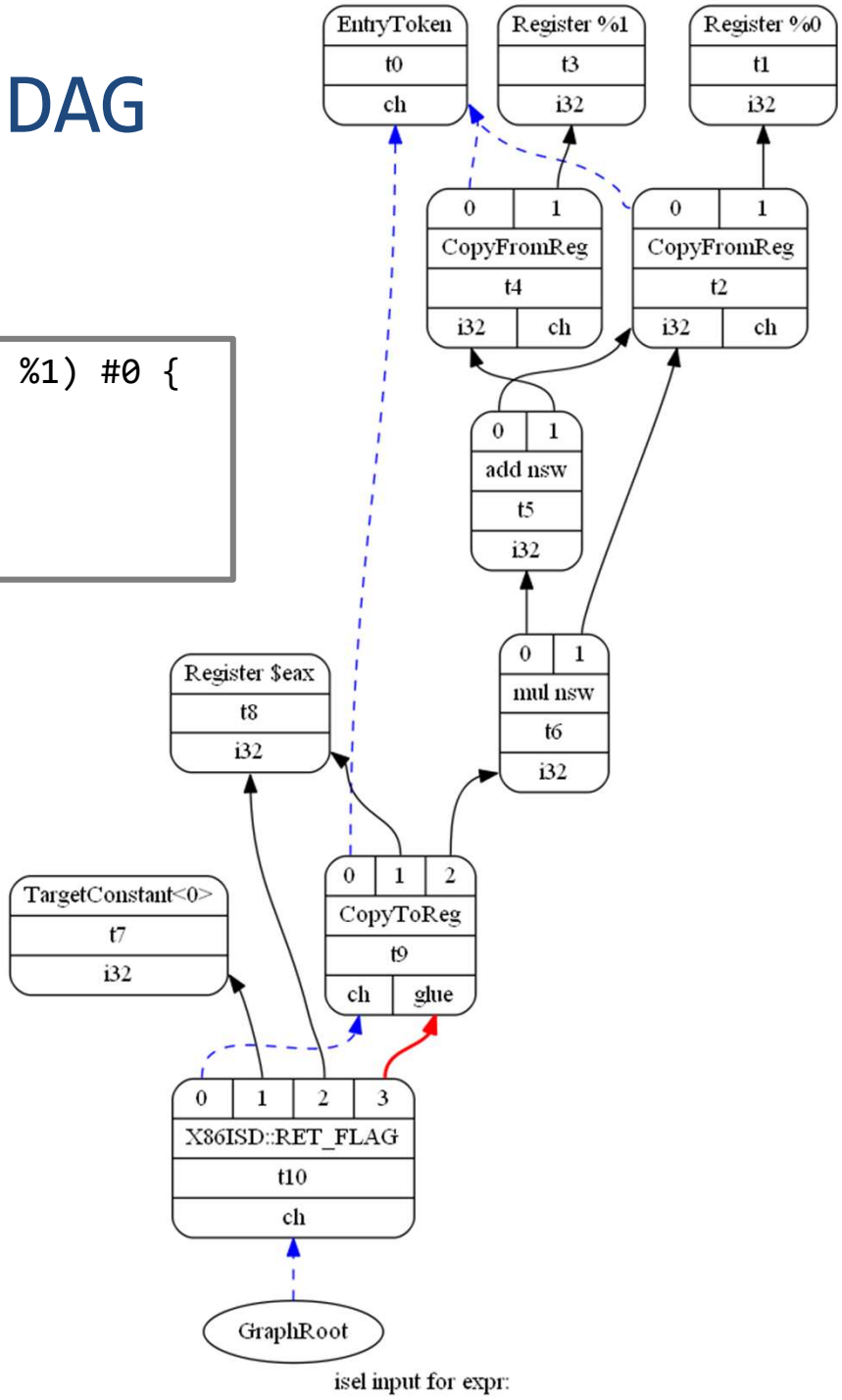
表达式树和DAG

- 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



LLVM用于指令选择的DAG

```
define dso_local i32 @expr(i32 %0, i32 %1) #0 {
  %3 = add nsw i32 %0, %1
  %4 = mul nsw i32 %3, %0
  ret i32 %4
}
```



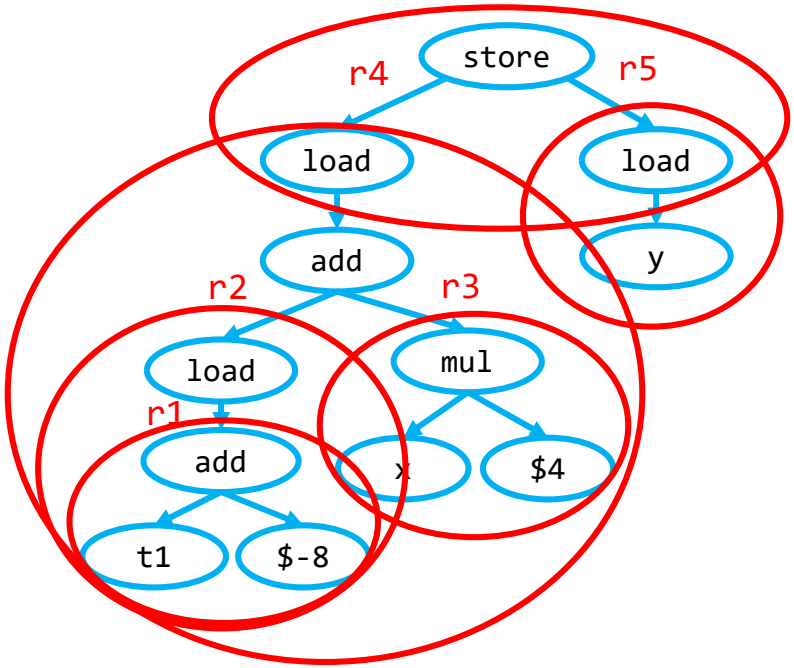
指令选择开销假设

IR指令模式	汇编指令	开销	备注
add(t1,t2)	LEAL (t1,t2), r	1	
add(t1,$i)	LEAL $i(t1), r	1	
mul(t1,t2)	MOVL t2, r IMUL t1, r	2	
mul(t1,$i)	LEAL (,t1,$i), r	1	$i=1/2/4
mul(t1,$i)	MOVL $i, r IMUL t1, r	2	
load(t1)	MOVL (t1), r	1	
store(load(t1), t2)	MOVL t2, (t1)	1	
store(load(t1), load(t2))	MOVL (t2), r MOVL r, (t1)	2	
load(+(t1,$i))	MOVL $i(t1), r	1	
store(load(+(t1,t2)), t3)	MOVL t3, (t1,t2)	1	t3是寄存器或立即数

Lvalue在左侧　　Lvalue在右侧

铺树问题（Tile an Expression Tree）

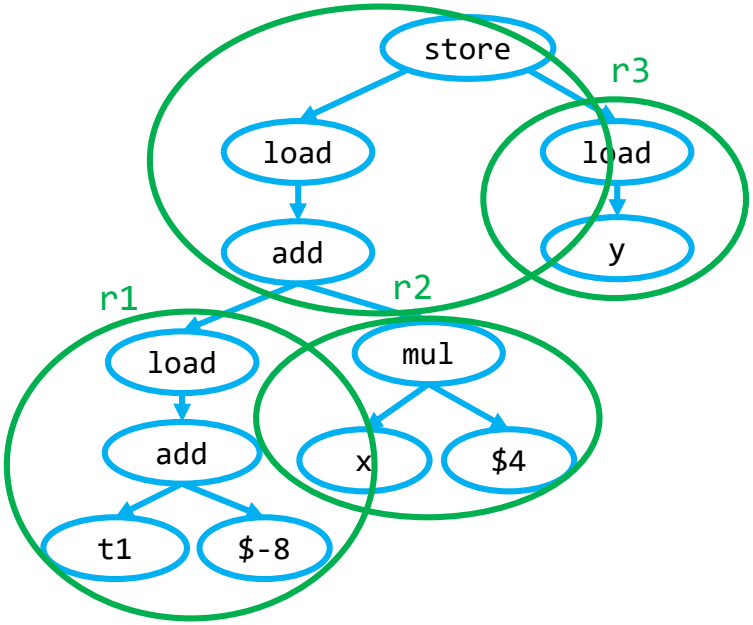
假设数组A的地址保存在t1+$-8的地址中，
A[x] = *y的表达式树如下图所示：




```
LEAL $-8(t1),r1
MOVL (r1),r2
LEAL (,x,$4),r3
LEAL (r2,r3),r4
MOVL (y),r5
MOVL r5,(r4)
```

指令数：6

另外一种铺树选择




```
LEAL $-8(t1),r1
MOVL (r1),r2
LEAL (,x,$4),r3
LEAL (r2,r3),r4
MOVL (y),r5
MOVL r5,(r4)
```

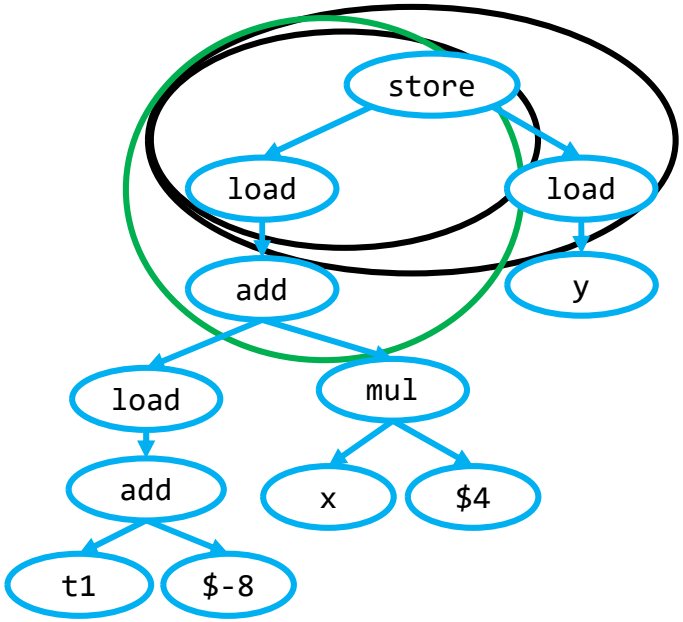
```
LEAL $-8(t1),r1
LEAL (,x,$4),r2
MOVL (y),r3
MOVL r3,(r1,r2)
```

指令数：4

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

Maximal Munch




```
store(load(t1), t2)
→ MOVL t2, (t1)
```

```
store(load(t1), load(t2))
→ MOVL (t2), r
  MOVL r, (t1)
```

```
store(load(+(t1,t2)), t3)
→ MOVL t3, (t1,t2)
```

```
LEAL $-8(t1),r1
LEAL (,x,$4),r2
MOVL (y),r3
MOVL r3,(r1,r2)
```

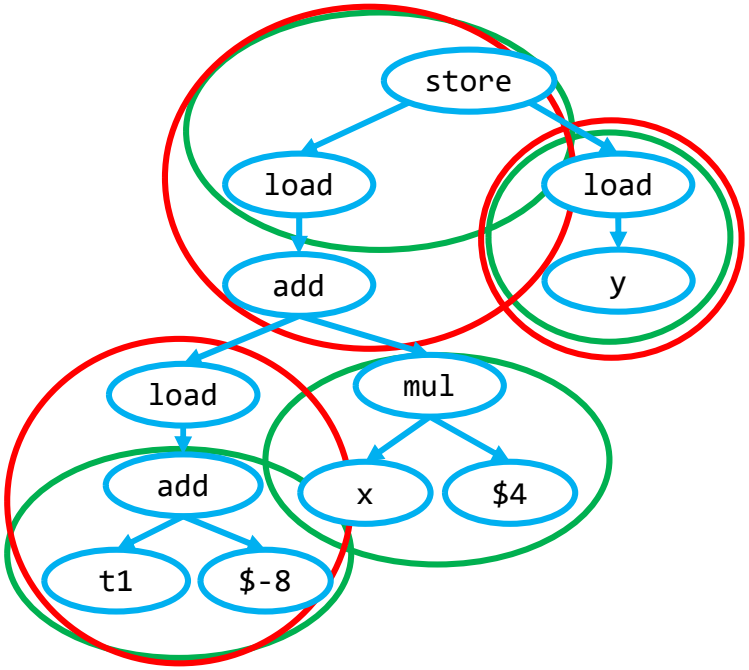
如果匹配规则改变

	IR指令模式	汇编指令	开销	备注
1	add(t1,t2)	LEAL (t1,t2), r	1	
2	add(t1,$i)	LEAL $i(t1), r	1	
3	mul(t1,t2)	MOVL t2, r IMUL t1, r	2	
4	mul(t1,$i)	LEAL (,t1,$i), r	1	$i=1/2/4
5	mul(t1,$i)	MOVL $i, r IMUL t1, r	2	
6	load(t1)	MOVL (t1), r	1	
7	store(load(t1), t2)	MOVL t2, (t1)	1	
8	store(load(t1), load(t2))	MOVL (t2), r MOVL r, (t1)	2	

9	load(+(t1,$i))	MOVL $i, r
		ADDL t1, r
		MOVL (r), r
			3	

10	store(load(+(t1,t2)), t3)	MOVL t2, r
		ADDL t1, r
		MOVL t3, (r)
			3	

Maximal Munch结果并非最优




```
MOVL $-8(t1),r1
ADDL %ebp,r1
MOVL (r1),r2
LEAL (,x,$4),r3
LEAL (r2,r3),r4
MOVL (y),r5
MOVL r5,(r4)
```

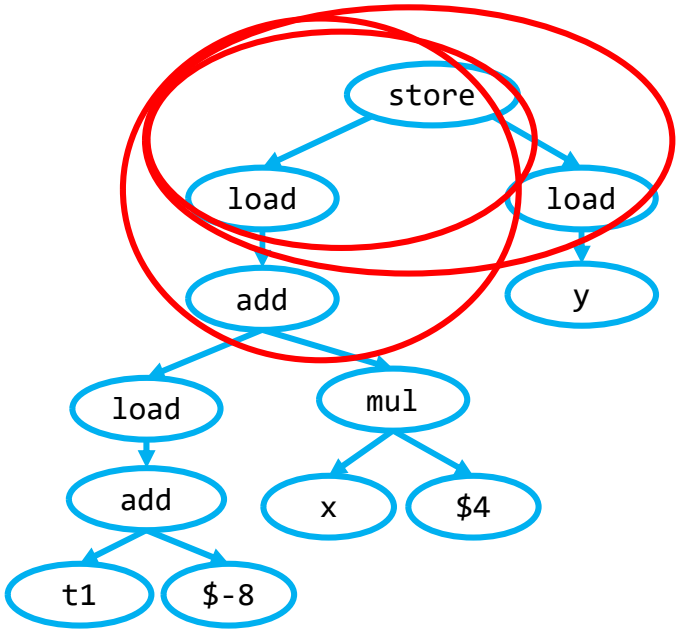
开销：7

```
LEAL $-8(t1),r1
MOVL (r1),r2
LEAL (,x,$4),r3
LEAL (r2,r3),r4
MOVL (y),r5
MOVL r5,(r4)
```

开销：6

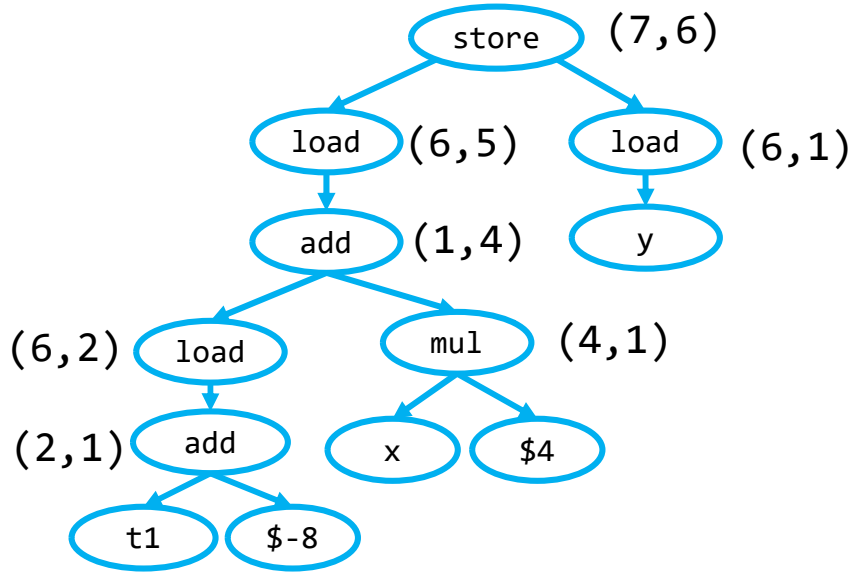
动态规划思路

- 从根开始计算子树的最优平铺解
  - 根节点的最优解方案选项：
    - store(load(t1), t2)
      - t1的最优解方案选项：
        - add(t3,t4)
          - t3的最优解方案选项：
            - load(t5)
              - t5的最优解方案选项：
                - add(t1,$i)
            - load(add(t1,$i))
          - t4的最优解方案选项：
            - mul(t1,$i)
      - t2的最优解方案选项：
        - load(y)
    - store(load(t1), load(t2))
      - …
    - store(load(add(t1,t2)), t3)
      - …



动态规划最优解

标记：(规则序号：开销)

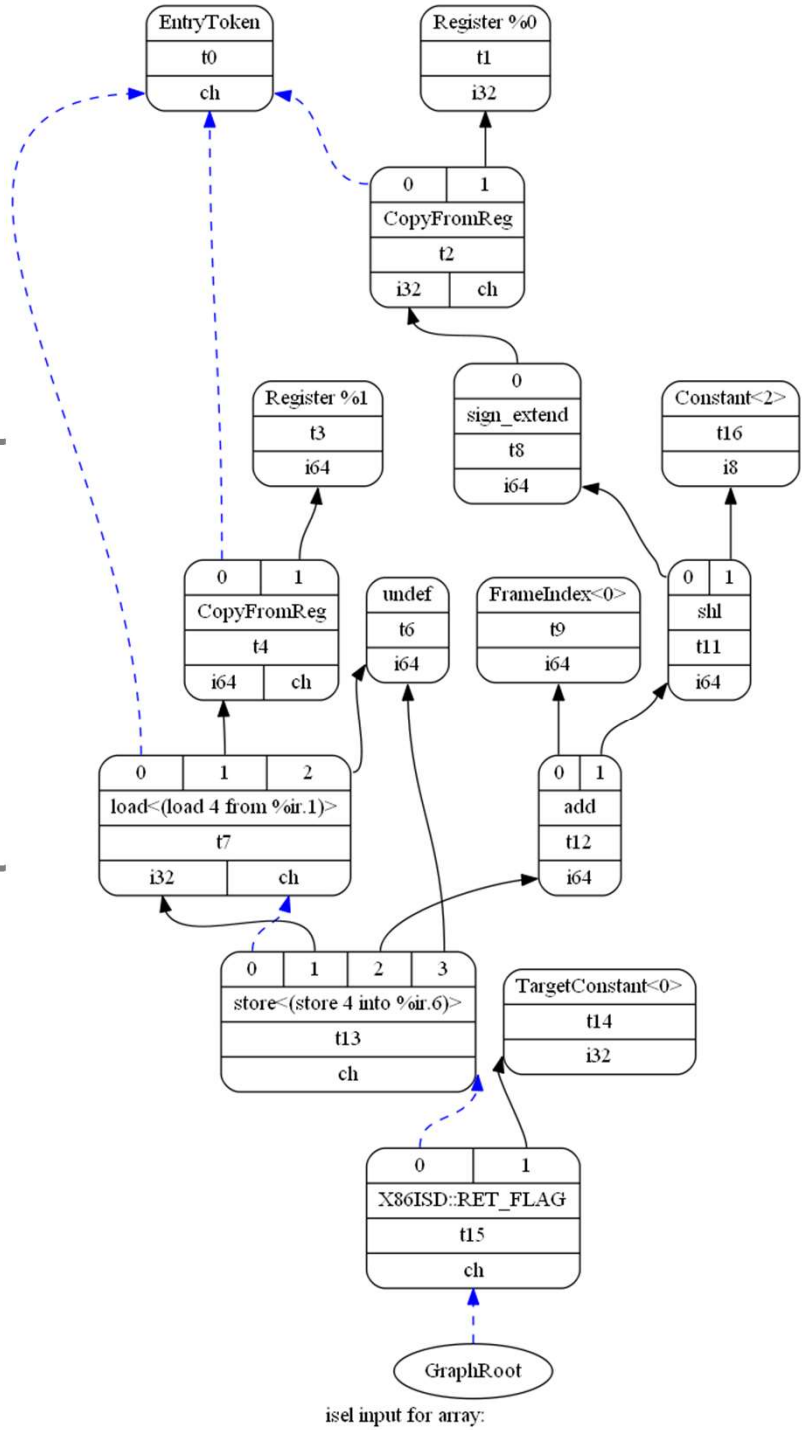


LLVM的例子

```
void array(int x, int* y){
    int A[5];
    A[x] = *y;
}
```

```
define dso_local void @array(i32 %0, i32* %1) #0 {
  %3 = alloca [5 x i32], align 16
  %4 = load i32, i32* %1, align 4
  %5 = sext i32 %0 to i64
  %6 = getelementptr inbounds [5 x i32],
            [5 x i32]* %3, i64 0, i64 %5
  store i32 %4, i32* %6, align 4
  ret void
}
```

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -4(%rbp)
movq    %rsi, -16(%rbp)
movq    -16(%rbp), %rax
movl    (%rax), %ecx
movslq  -4(%rbp), %rax
movl    %ecx, -48(%rbp,%rax,4)
popq    %rbp
retq
```



窥孔优化（Peephole Optimization）

- 基于滑动窗口匹配的指令优化思路
  - 编译器一般会先把IR转换成更小的IR；
  - 优化效果取决于窗口大小等因素。

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -4(%rbp)
movq    %rsi, -16(%rbp)
movq    -16(%rbp), %rax
movl    (%rax), %ecx
movslq  -4(%rbp), %rax
movl    %ecx, -48(%rbp,%rax,4)
popq    %rbp
retq
```

⇨ `movq %rsi, %rax`

⇨ `movq (%rsi), %rcx`

⇨ `movq %edi, %rax`

三、指令调度算法

流水线（Instruction pipelining）

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

```
ADD t1, t2
SUB t1, t3
```

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

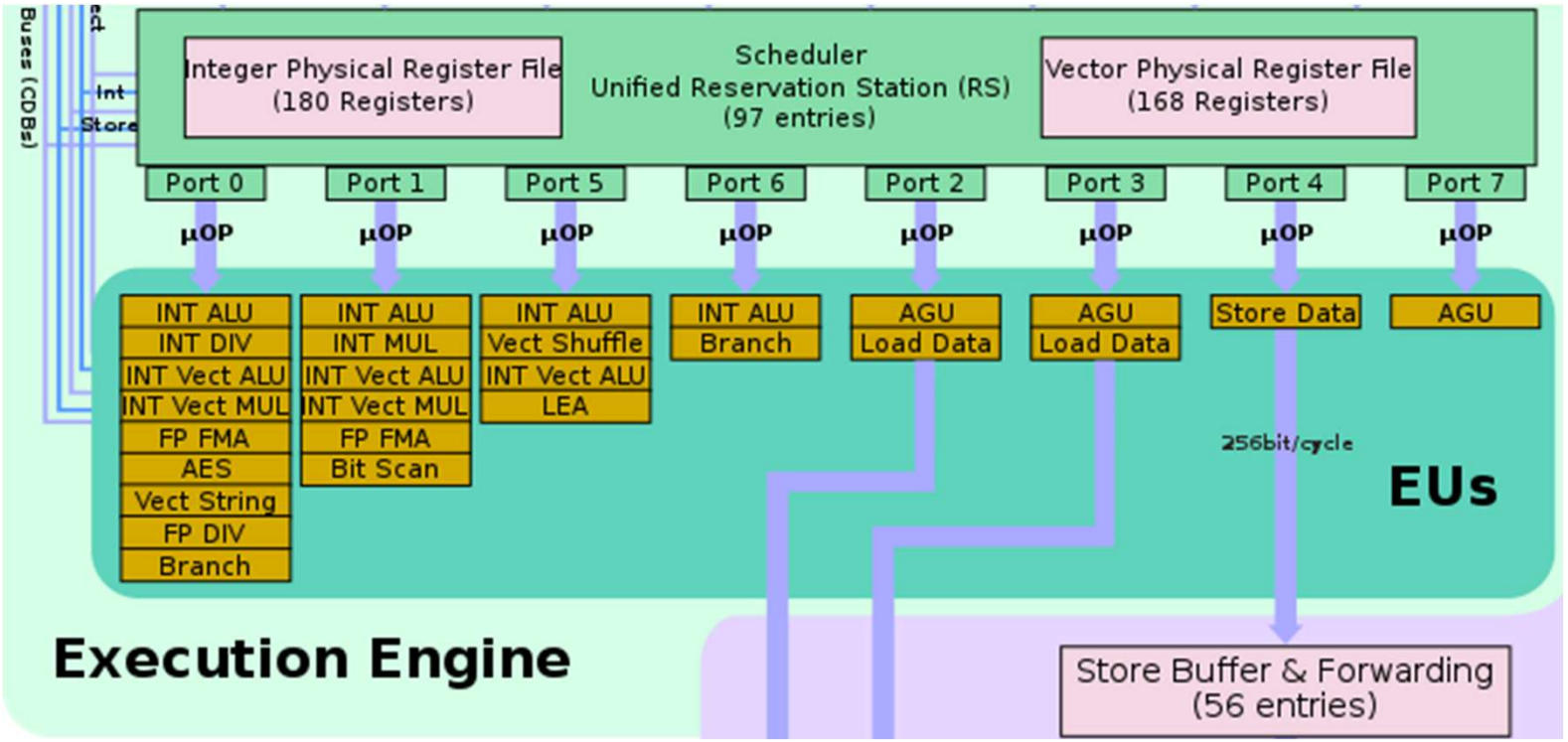
- 防止数据依赖造成的CPU空闲，可以优先执行后面的指令。
- CPU级别的指令调度机制

```
ADD t1, t2
SUB t2, t3
MUL t1, t4
```

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 harzard）
  - 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 harzard）
  - 条件跳转或分支预测

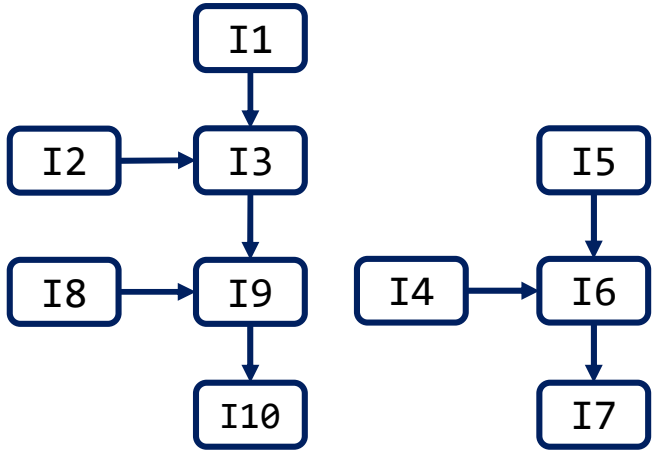
指令调度问题

- 指令的执行时间与其执行顺序密切相关。
- 如何在单位时间内执行更多的操作？
- 编译器的指令调度一般只考虑数据依赖关系。

指令依赖关系

- 对于程序块（无跳转指令），如果指令I2使用了I1的结果，那么指令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)

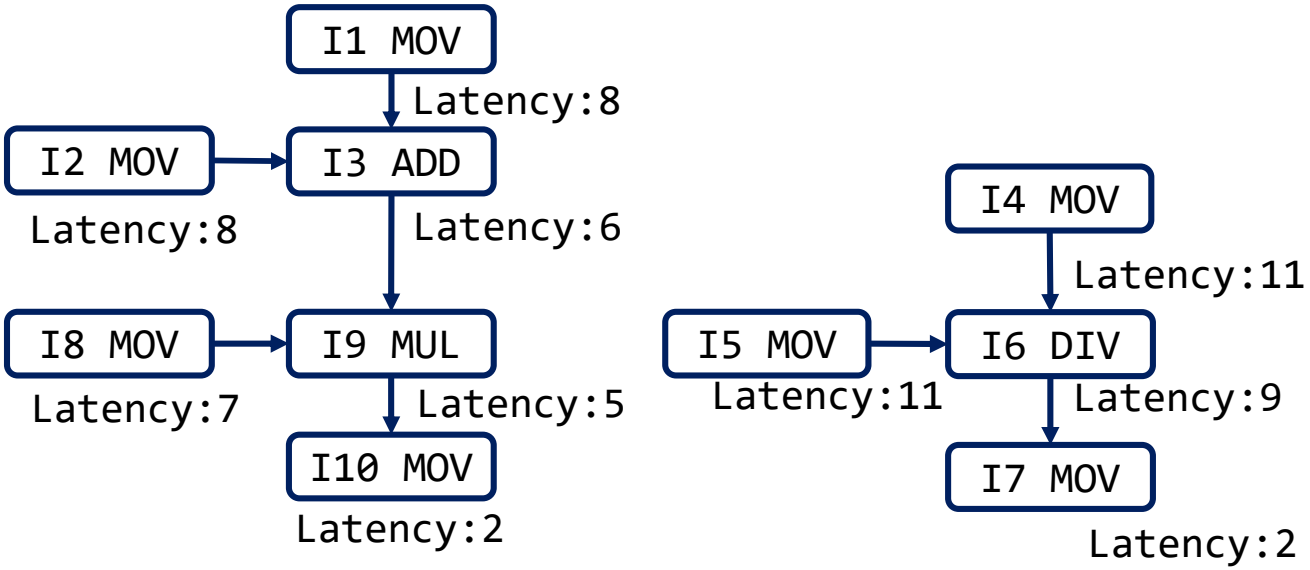


指令依赖关系

指令调度约束

- 倒序推导每条指令开始后的最早结束时间latency。
- 根据latency从大到小对指令进行排序：
  - I4=I5>I6>I1=I2>I8>I3>I9>I7=I10

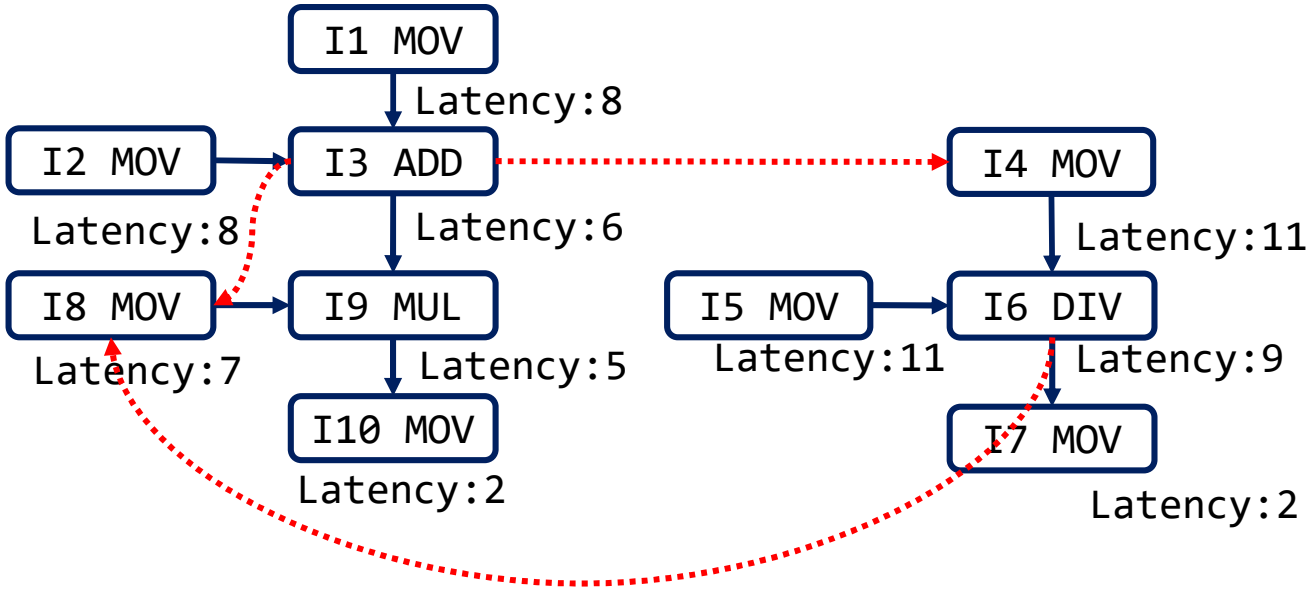
Cost假设：
MOV 2
ADD 1
MUL 3
DIV 7



反依赖问题

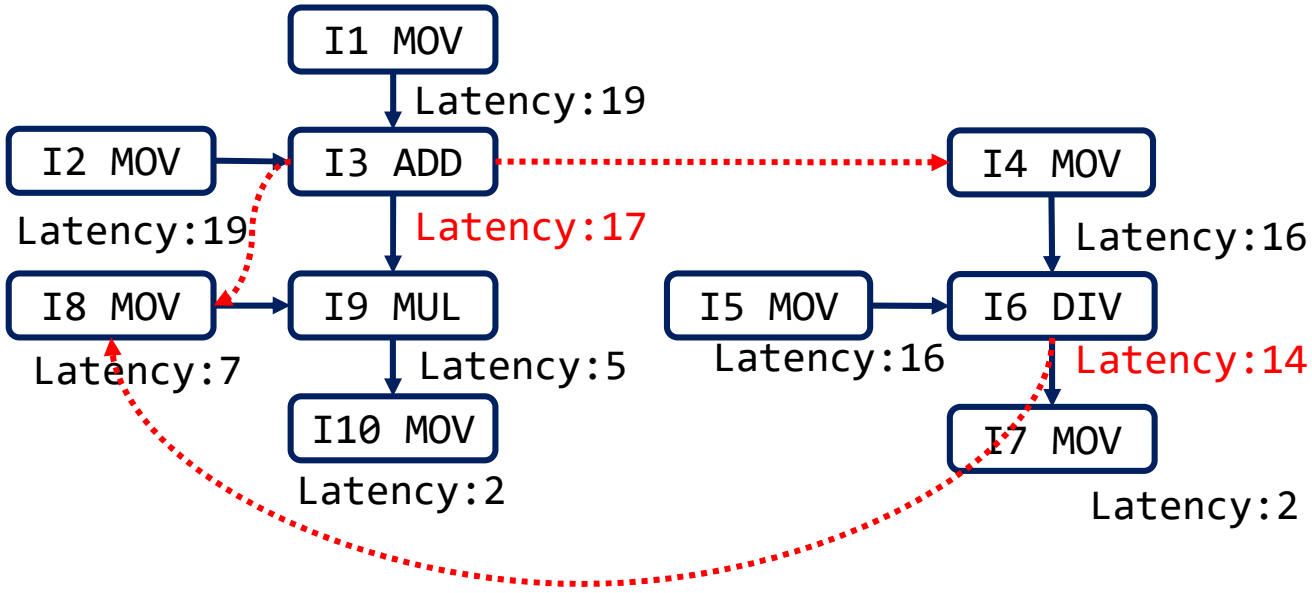
- 读-写反依赖（anti-denpendency）
  - I3执行完I4和I8才能执行；
    - 否则会影响I3的计算结果
  - I6执行完才能执行I8；

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)



如何调度？

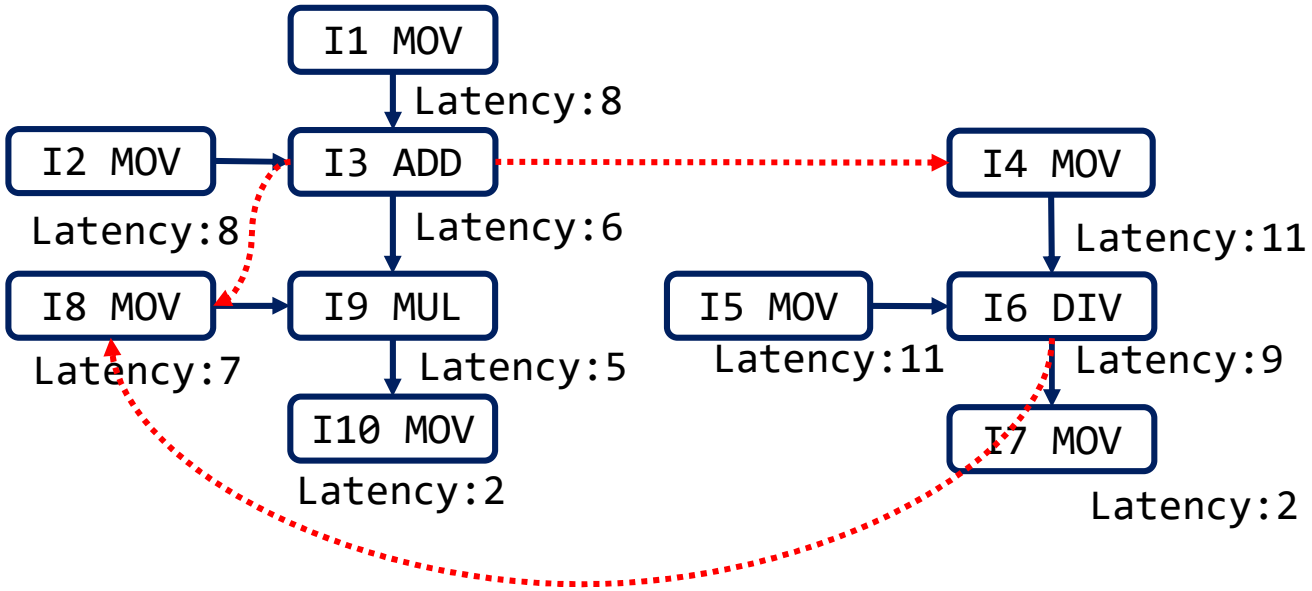
- Latency：I4=I5>I6>I1=I2>I8>I3>I9>I7=I10
  - I3早于I4、I8；
  - I6早于I8；
- 更新Latency：I1=I2>I3>I4=I5>I6>I8>I9>I7=I10



调度方案开销

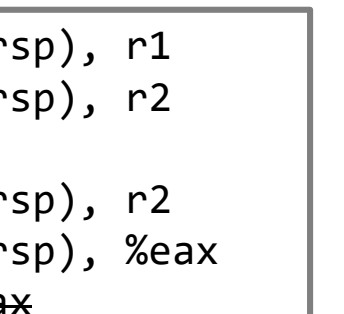
- I1=I2>I3>I4=I5>I6>I8>I9>I7=I10
  - 开销：22
- I7可以提前执行
  - I1, I2, I3, I4, I5, I6, I8, I7, I9, I10
  - 开销：21

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 $-28(%rsp), r2
17	19	I9	MUL r1, r2
20	21	I7	MOV %eax, $-24(%rsp)
21	22	I10	MOV r2, $-28(%rsp)

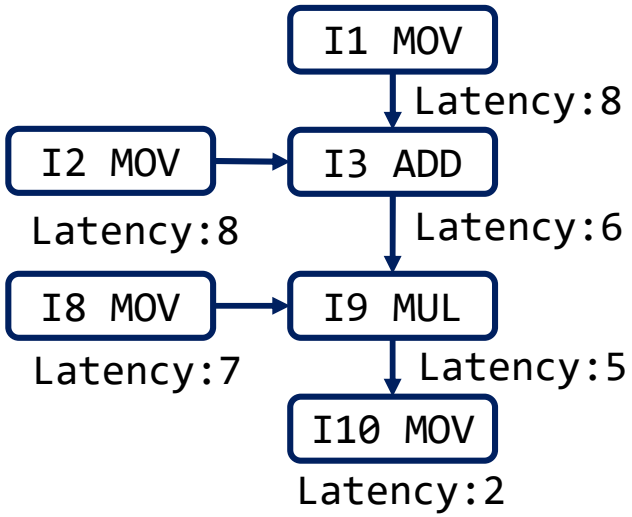


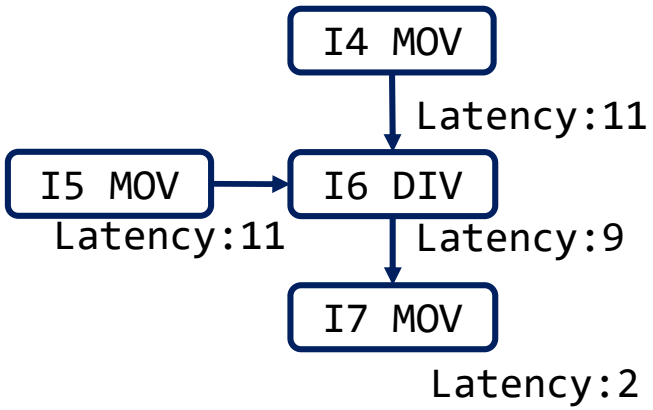
应对反依赖：重命名

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)

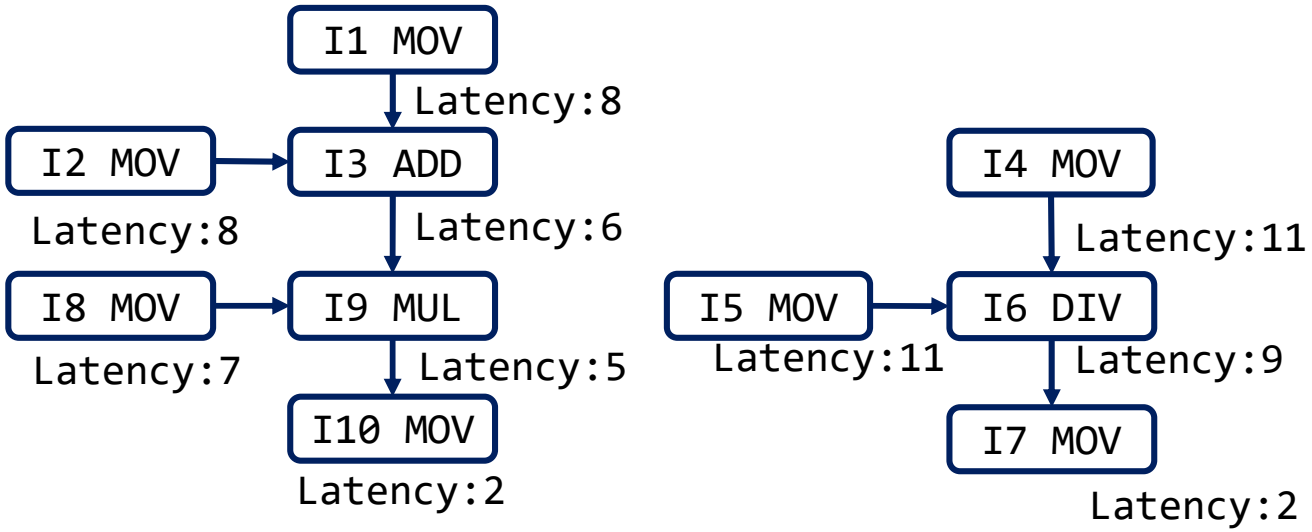




调度方案开销

- I4=I5>I6>I1=I2>I8>I3>I9>I7=I10
  - 开销：14
- 如果I1和I6互换顺序，I7和I9互换
  - 开销：12
- 应尽早执行已满足了数据依赖的指令

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)



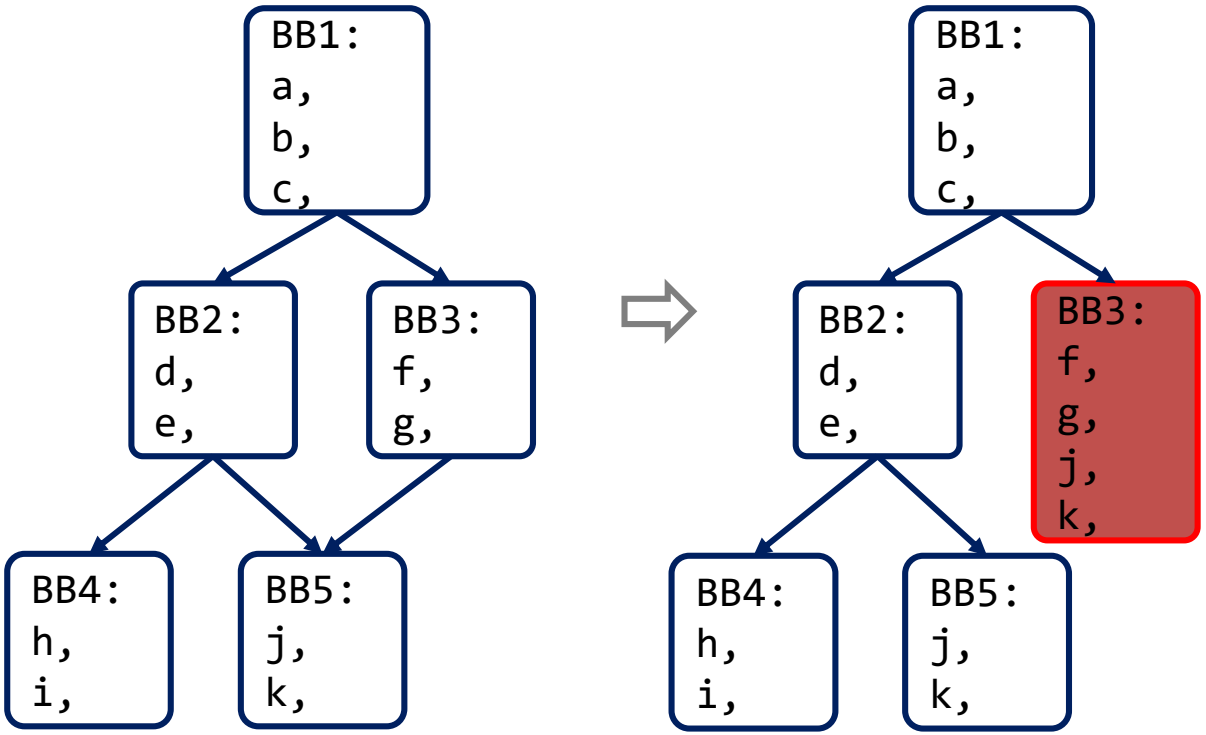
表调度算法

```
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;
}
```

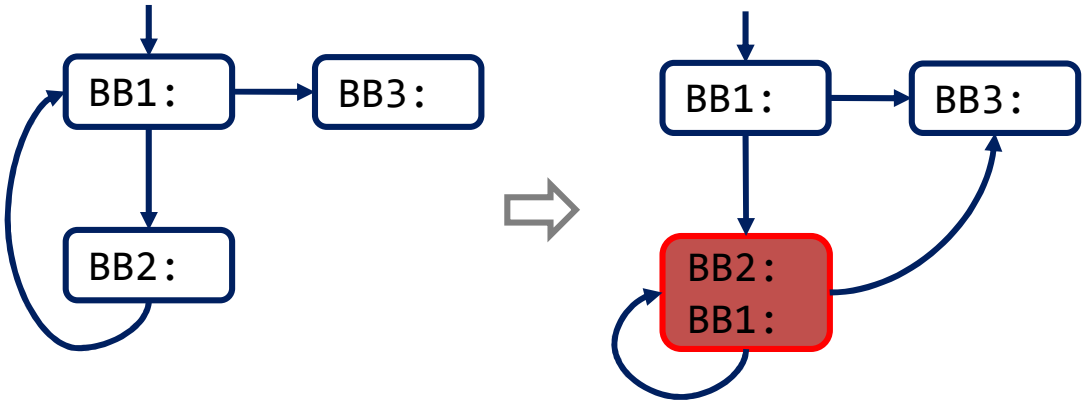
- 假设：
  - 线性代码；
  - 无反依赖。
- 两张表：
  - Ready表记录已满足数据依赖的指令；
  - Active表记录正在执行的指令。
- 方法：
  - 每个Clock尽量执行一条新的指令；
  - 如果Active表有指令执行完成，考虑将指令的next指令加入Ready。

如何解决跨代码块的指令调度？

- 核心思想：牺牲程序体积，提升运行速度
  - 复制代码块



应用：循环优化



尾递归优化后的程序　　复制后

其它跨区域调度思路

- 合并代码块，在其它受影响的块中插入补偿指令
  - 如将BB2中的部分代码块移入BB1，同时需要在BB3中消除副作用