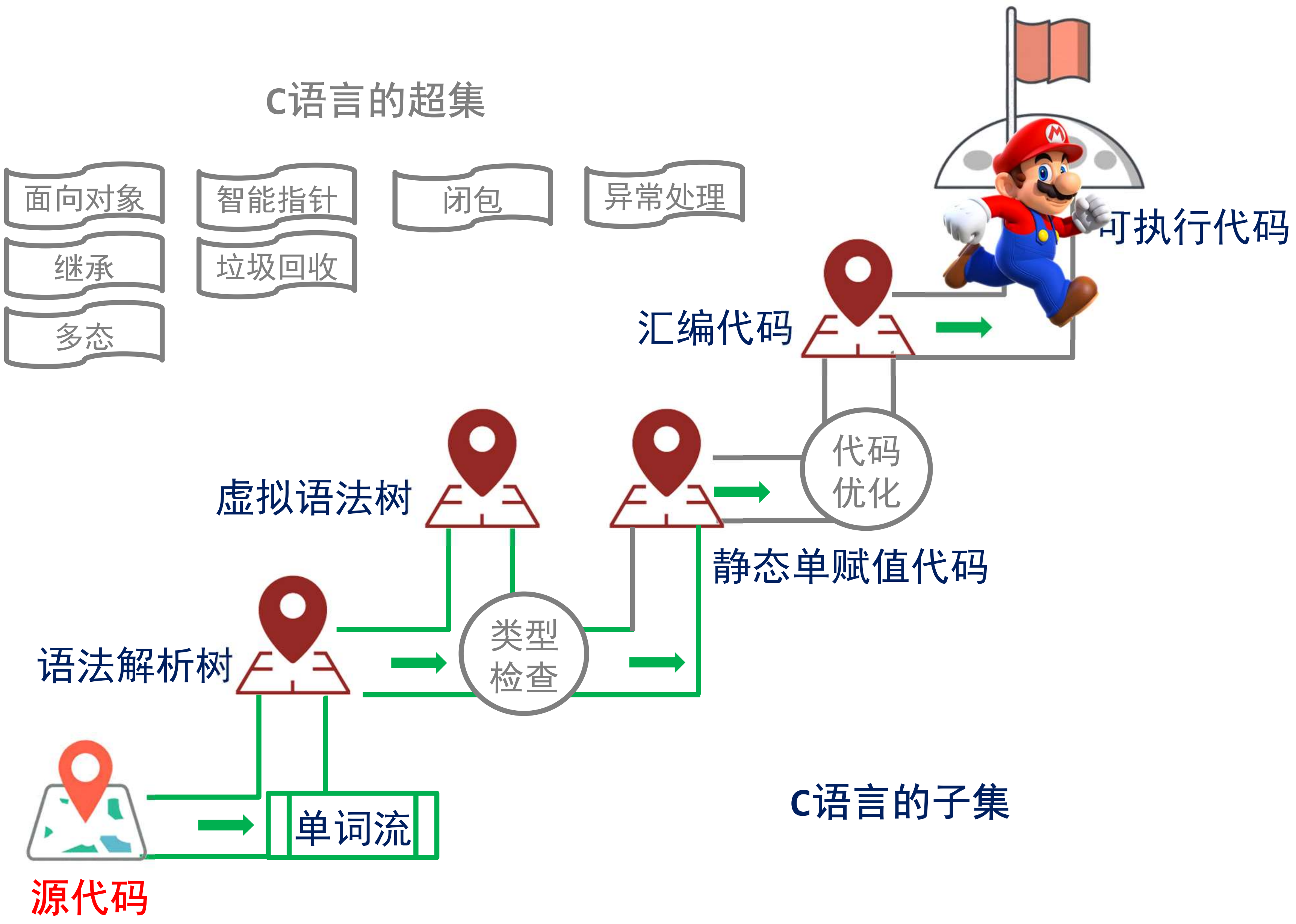
Lecture 5

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
源代码
C语言的子集

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

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

```llvm
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
}
```

```asm
# %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

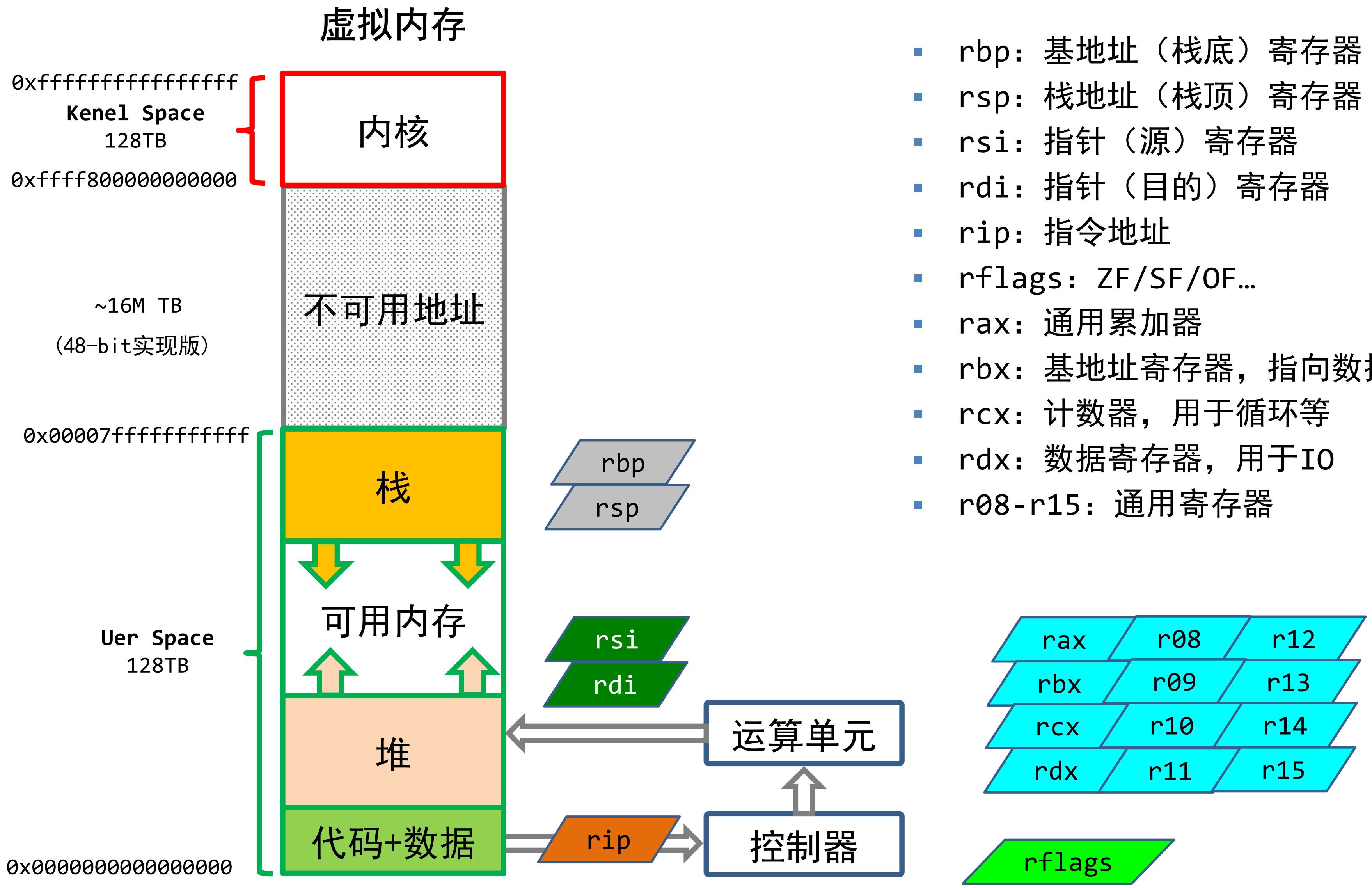
- 一、指令集和汇编代码
- 二、指令选择和翻译
- 三、指令调度算法
- 四、寄存器分配算法
- 五、LLVM案例参考
- 六、组装可执行程序

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, (%rbp, %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
- 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

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

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

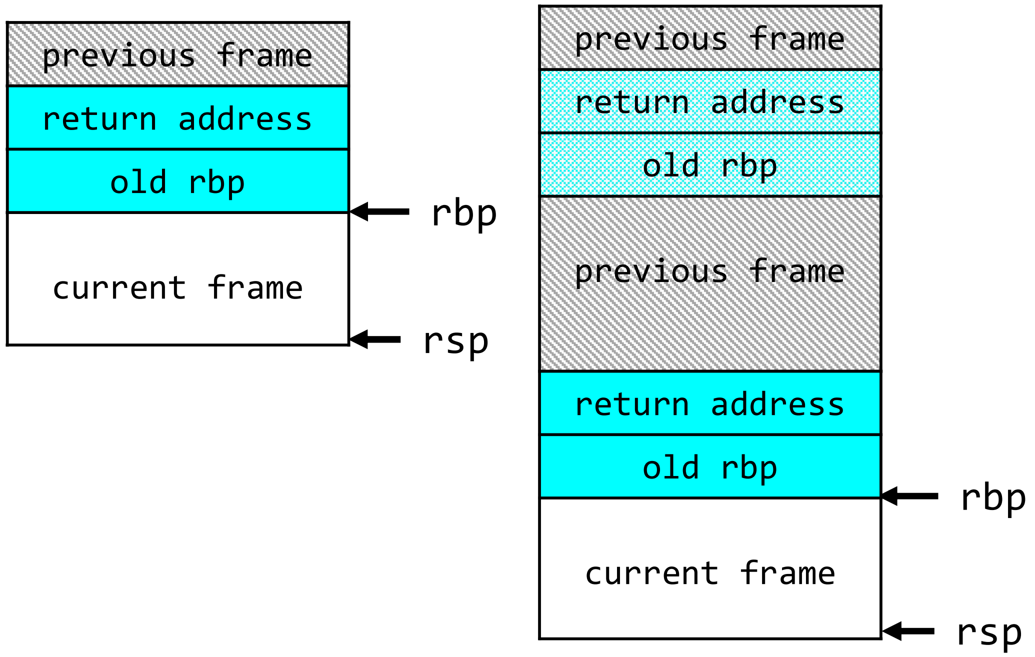
- 条件移动：
  - cmove/cmovne/cmovle/cmovl/cmovz/cmovnz/…
  - 源地址和目的地址必须都是寄存器
- 根据条件将目标寄存器设置0或1
  - sete/setne/setle/setl/setz/setnz/…
  - 寄存器只能是1byte的子寄存器，如al寄存器

```
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

```c
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

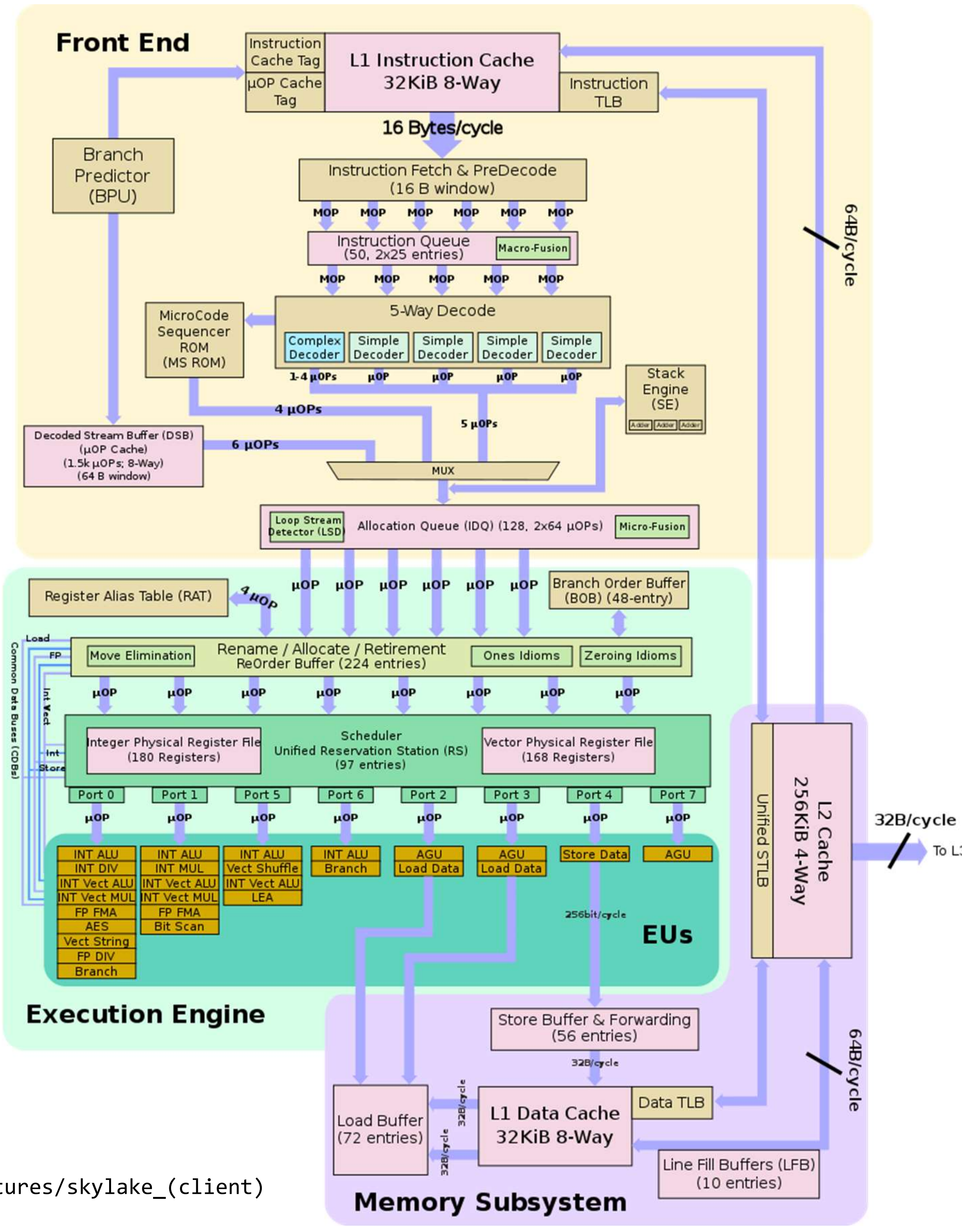
```
int val = 1;
do{
    __asm__("xchg %0, %1" : "+q" (val), "+m" (count));
} while(val - count == 0)
```

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）



Front End
Instruction Cache Tag
μOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP MOP MOP MOP MOP MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MOP MOP MOP MOP MOP
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Simple Decoder
1-4 μOPs μOP μOP μOP μOP
4 μOPs
5 μOPs
Stack Engine (SE)
Adder Adder Adder
Decoded Stream Buffer (DSB) (μOP Cache) (1.5k μOPs; 8-Way) (64 B window)
6 μOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 μOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
4 μOP
μOP μOP μOP μOP μOP μOP
Branch Order Buffer (BOB) (48-entry)
Load
FP
Common Data Buses (CDBs)
Int Vect
Int
Store
Move Elimination
Rename / Allocate / Retirement
ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
μOP μOP μOP μOP μOP μOP μOP μOP
Integer Physical Register File (180 Registers)
Scheduler
Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0 Port 1 Port 5 Port 6 Port 2 Port 3 Port 4 Port 7
μOP μOP μOP μOP μOP μOP μOP μOP
INT ALU INT DIV INT Vect ALU INT Vect MUL FP FMA AES Vect String FP DIV Branch
INT ALU INT MUL INT Vect ALU INT Vect MUL FP FMA Bit Scan
INT ALU Vect Shuffle INT Vect ALU LEA
INT ALU Branch
AGU Load Data
AGU Load Data
Store Data
AGU
256bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 4-Way
32B/cycle
To L3
Store Buffer & Forwarding (56 entries)
32B/cycle
Load Buffer (72 entries)
32B/cycle
32B/cycle
L1 Data Cache 32KiB 8-Way
Data TLB
64B/cycle
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

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：转化为铺树问题
  - 思路2：peephole优化
- 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

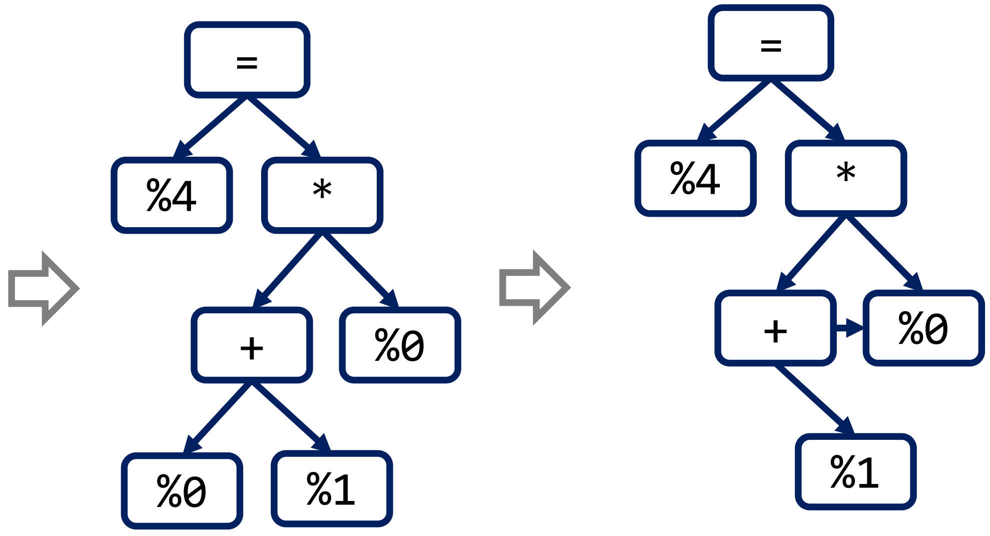
表达式树和DAG

- 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

```
int r = (a + b) * a;
```

```
%3 = add nsw i32 %0, %1
%4 = mul nsw i32 %3, %0
```

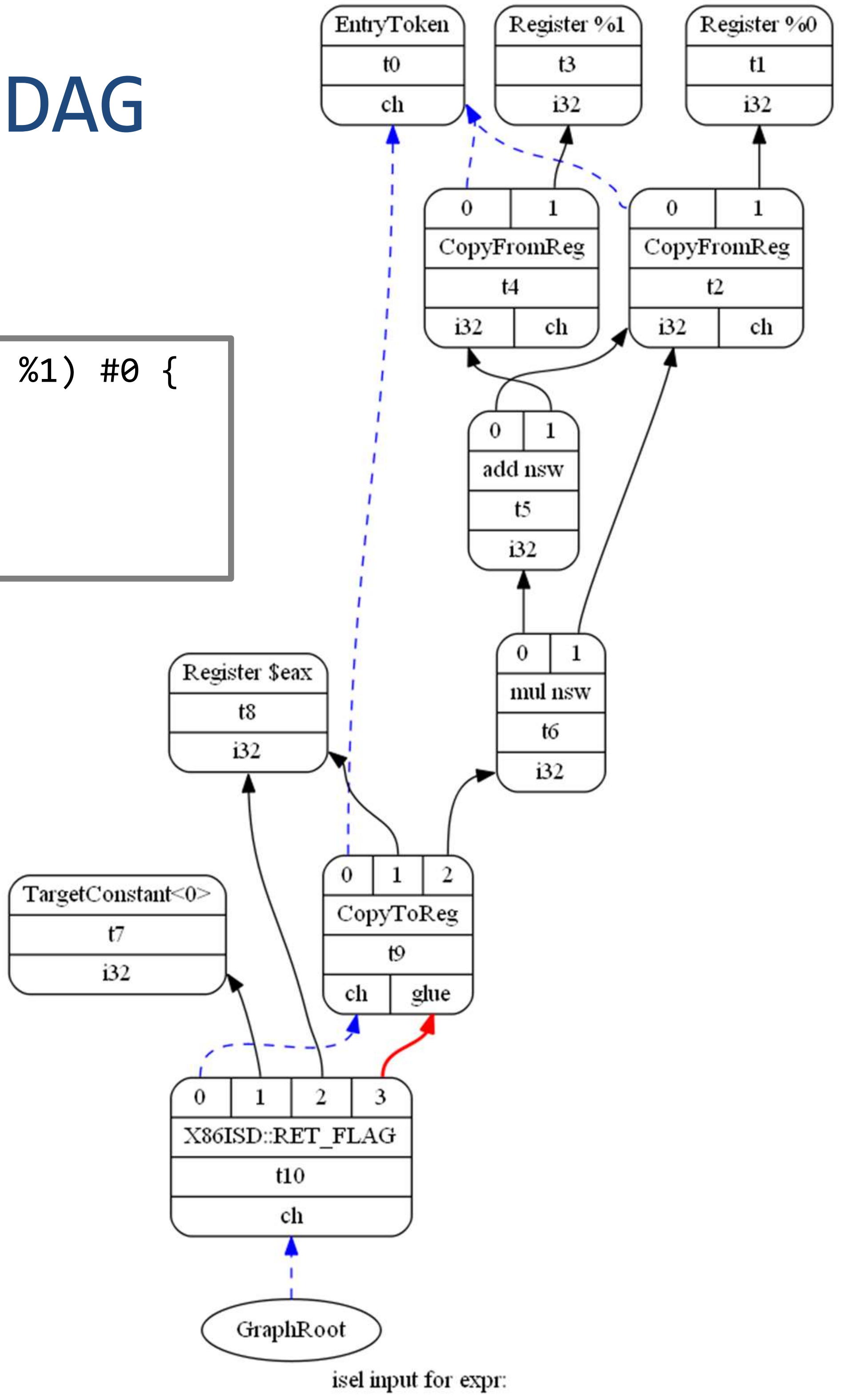
```
%4 = *(+(%0, %1),%0)
```



表达式树　　　　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
}
```



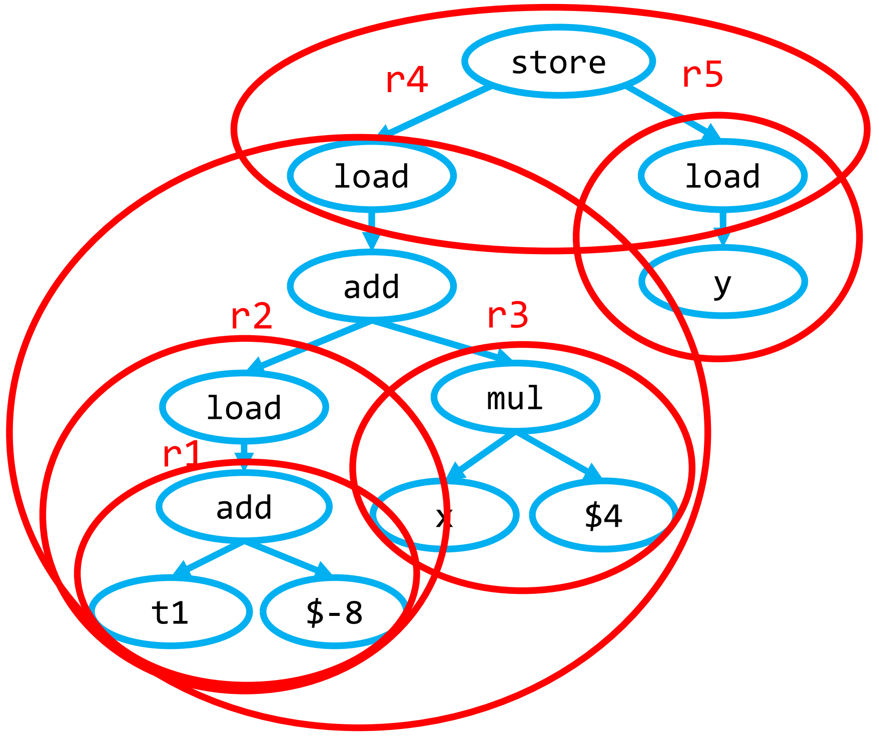
指令选择开销假设

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 $i, r IMUL t1, r	2	
load(t1)	MOVL (t1), r	1	
store(load(t1), t2)	MOVL t2, (t1)	1	
store(load(t1), load(t2))	MOVL (t2), r MOVL r, (t1)	2	
load(add(t1,$i))	MOVL $i(t1), r	1	
store(load(add(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）

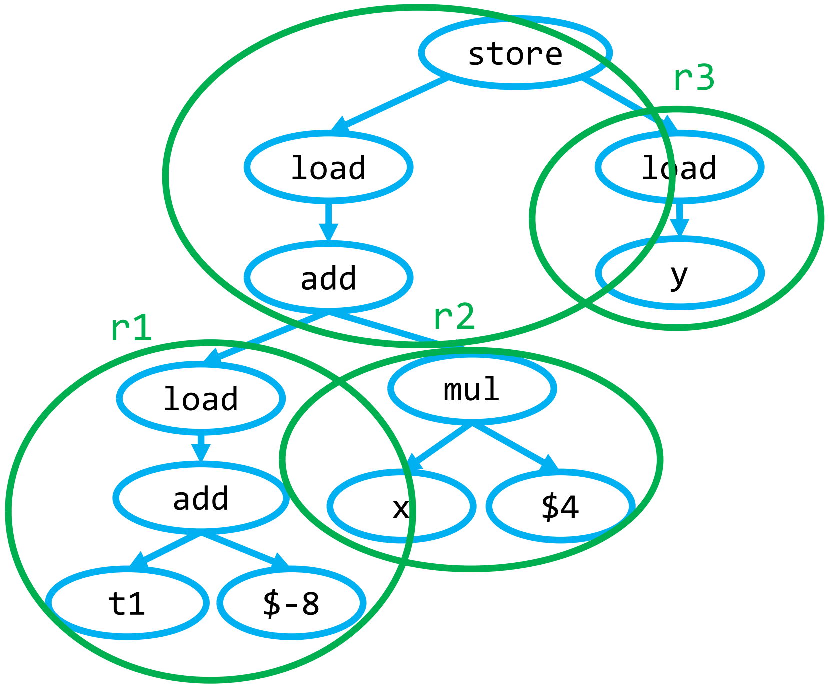
- 假设数组int A[]的地址保存在t1+$-8的内存中，
- A[x] = *y的表达式树如下图所示。



```
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



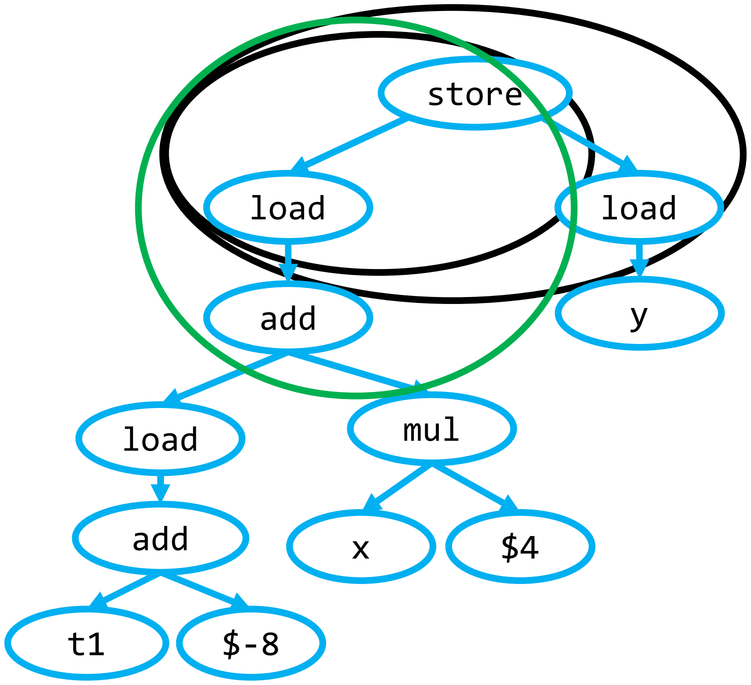
```
MOVL $-8(t1),r1
LEAL (,x,$4),r2
MOVL (y),r3
MOVL r3,(r1,r2)
```

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



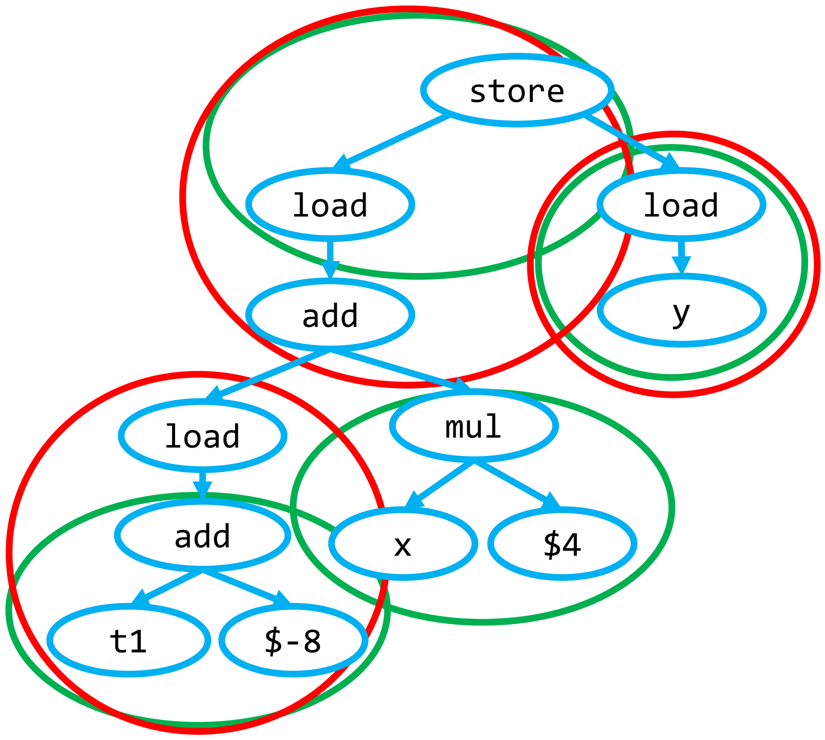
```
store(load(t1), t2)
→ MOVL t2, (t1)

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

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

```
MOVL $-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 $i, r IMUL t1, r	2	
6	load(t1)	MOVL (t1), r	1	
7	store(load(t1), t2)	MOVL t2, (t1)	1	
8	store(load(t1), load(t2))	MOVL (t2), r MOVL r, (t1)	2	
9	load(add(t1,$i))	MOVL $i, r ADDL t1, r MOVL (r), r	3	
10	store(load(add(t1,t2)), t3)	LEAL (t1,t2),r MOVL t3, (r)	3	

Maximal Munch结果并非最优



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

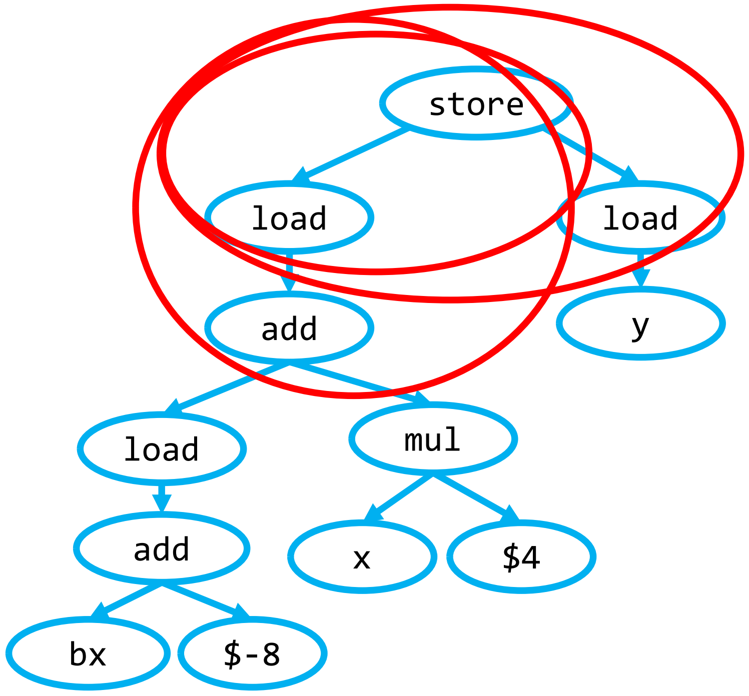
开销: 7

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

开销: 6

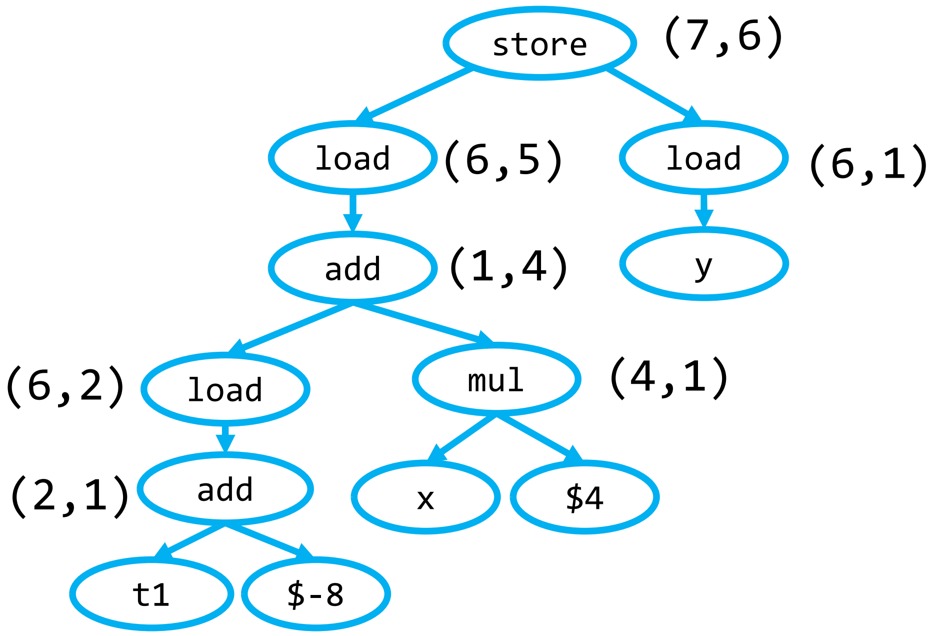
动态规划思路

- 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(bx,$i)
            - load(add(bx,$i))
          - t4的最优解方案选项：
            - mul(x,$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）

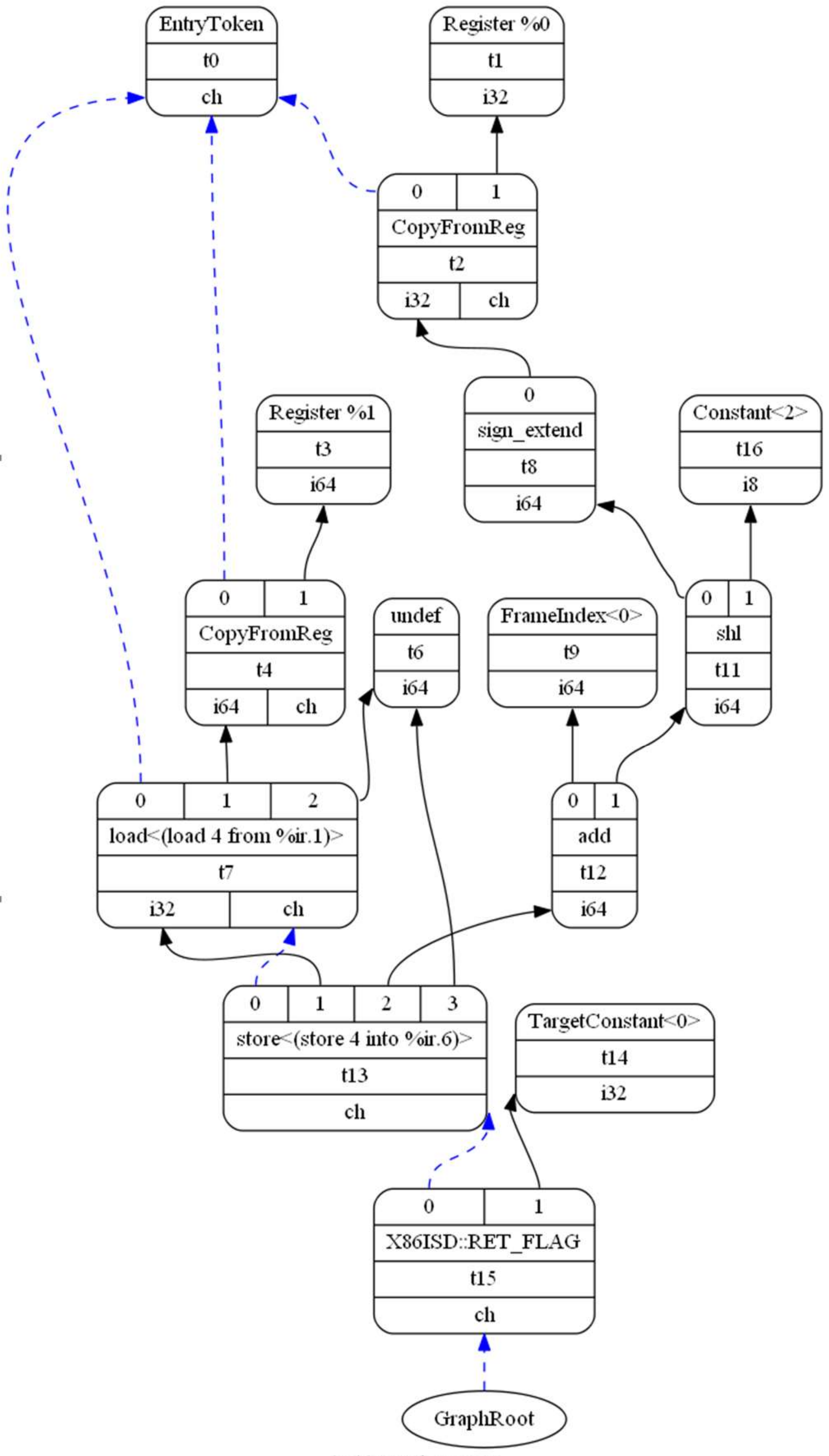


LLVM的例子

```
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
```

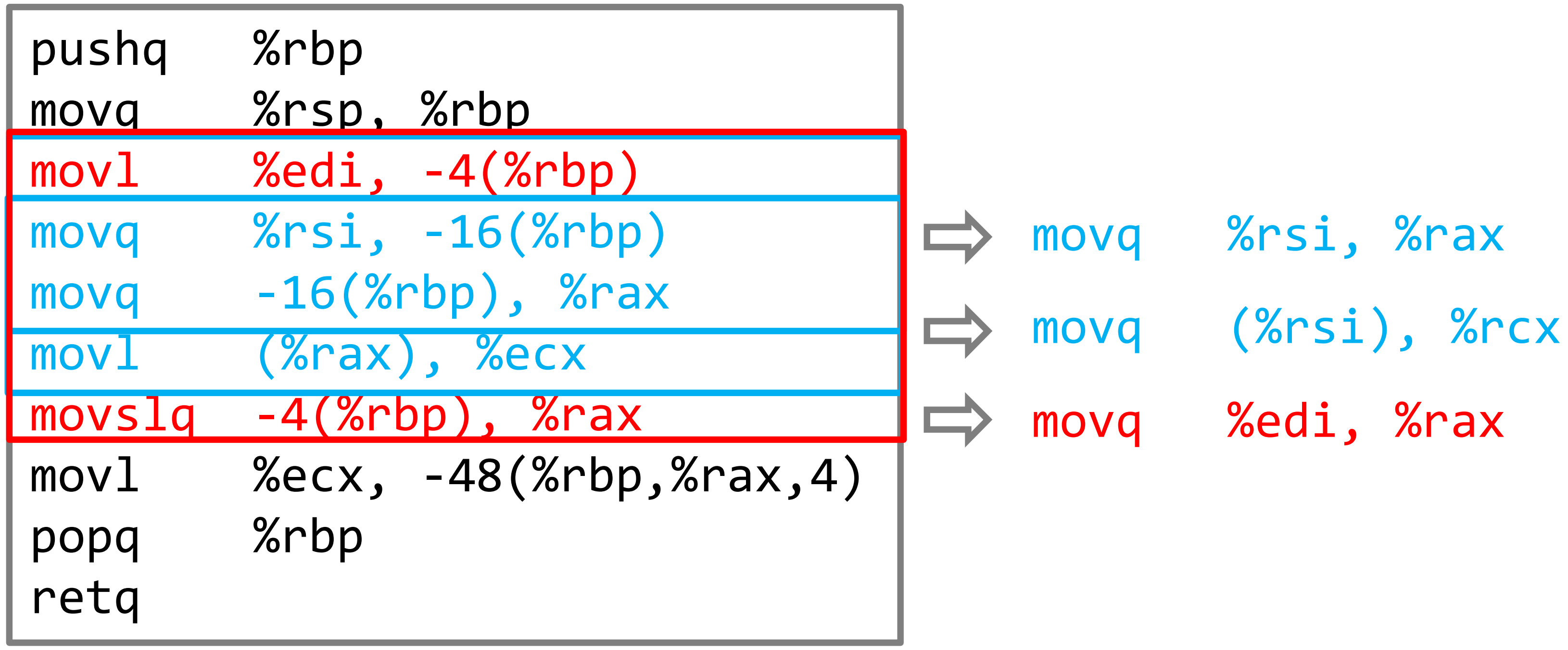


EntryToken t0 ch
Register %0 t1 i32
0 1 CopyFromReg t2 i32 ch
0 sign_extend t8 i64
Register %1 t3 i64
Constant<2> t16 i8
0 1 CopyFromReg t4 i64 ch
undef t6 i64
FrameIndex<0> t9 i64
0 1 shl t11 i64
0 1 2 load<(load 4 from %ir.1)> t7 i32 ch
0 1 add t12 i64
0 1 2 3 store<(store 4 into %ir.6)> t13 ch
TargetConstant<0> t14 i32
0 1 X86ISD::RET_FLAG t15 ch
GraphRoot

isel input for array:

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

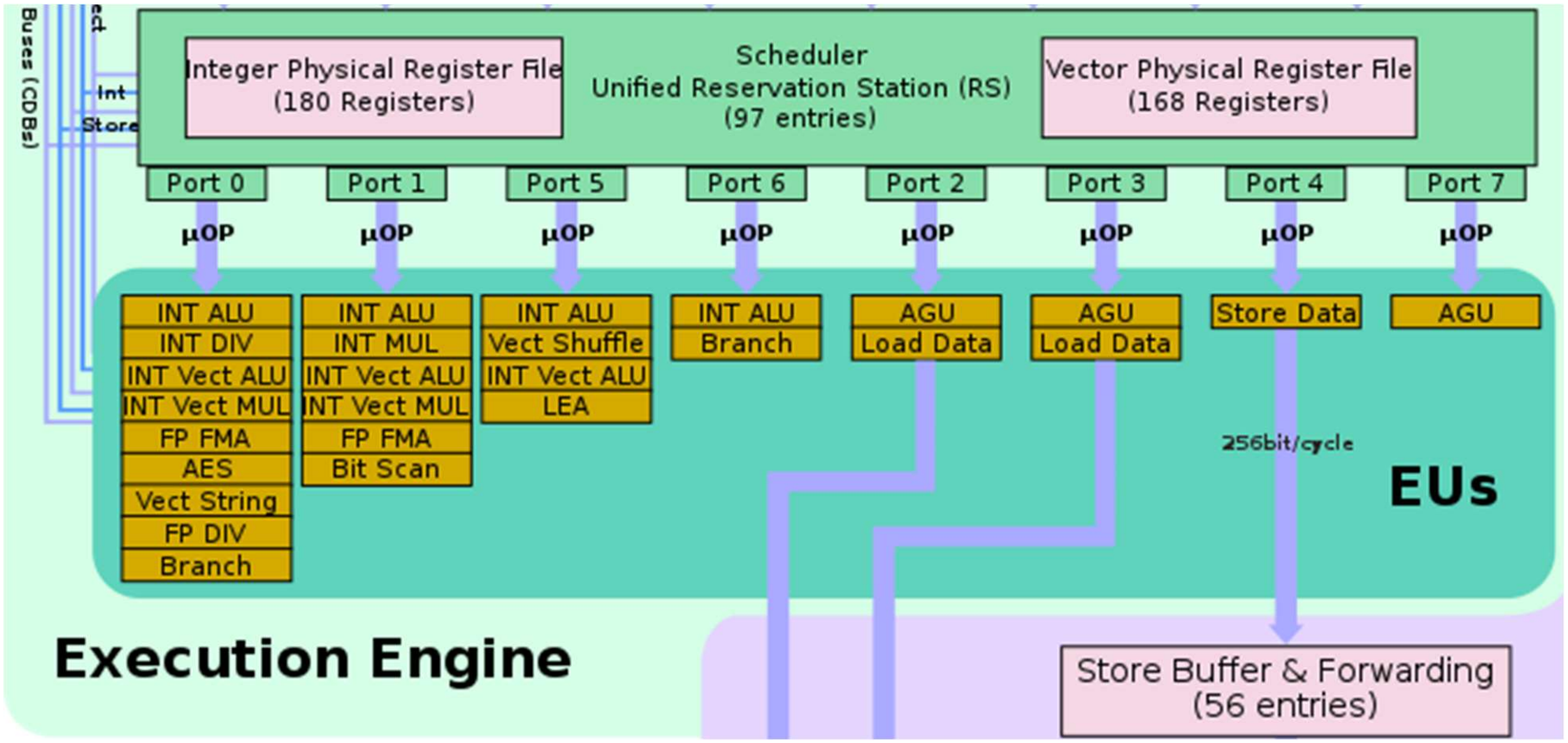
- 防止数据依赖造成的CPU空闲，可以优先执行后面的指令。
- CPU级别的指令调度机制

```
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 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

指令调度问题

- 指令的执行时间与其执行顺序密切相关。
- 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。
- 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)

$$\begin{tikzcd}
 & \boxed{I1} \arrow[d] & & \\
\boxed{I2} \arrow[r] & \boxed{I3} \arrow[d] & & \boxed{I5} \arrow[d] \\
\boxed{I8} \arrow[r] & \boxed{I9} \arrow[d] & \boxed{I4} \arrow[r] & \boxed{I6} \arrow[d] \\
 & \boxed{I10} & & \boxed{I7}
\end{tikzcd}$$

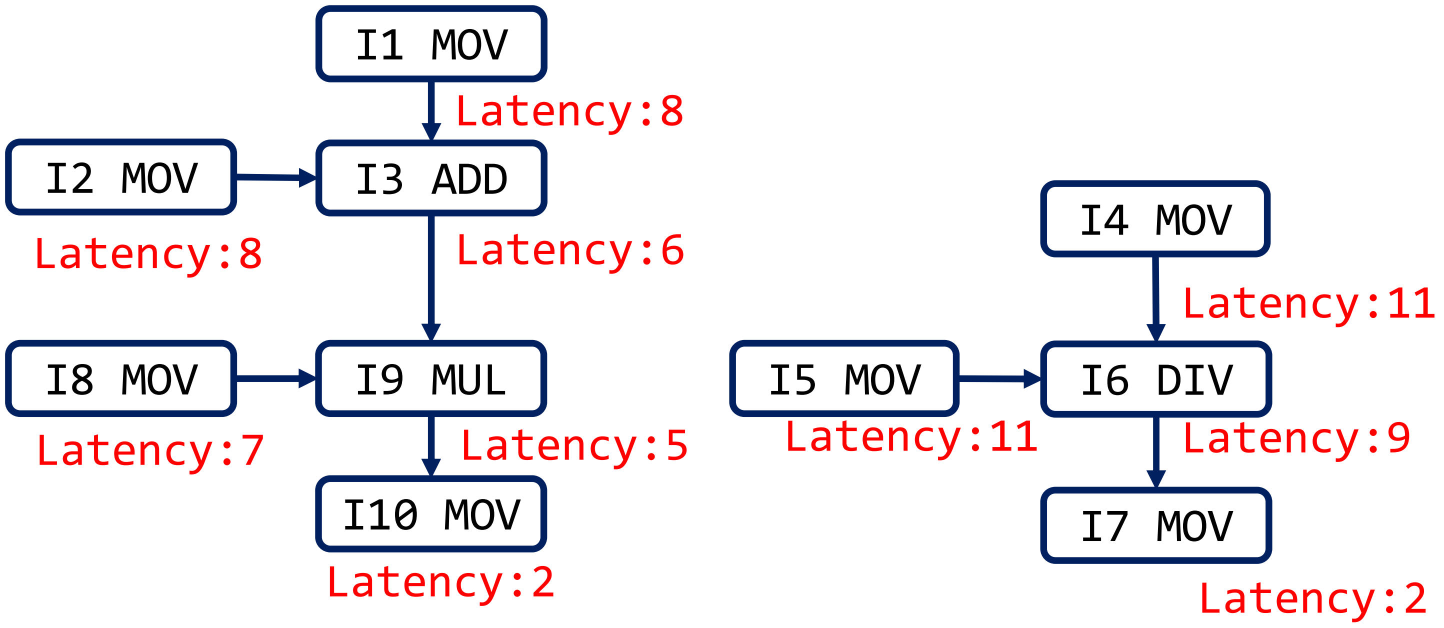
指令依赖关系

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
- 优先执行Latency大的指令。

Cost假设：

MOV	2
ADD	1
MUL	3
DIV	7

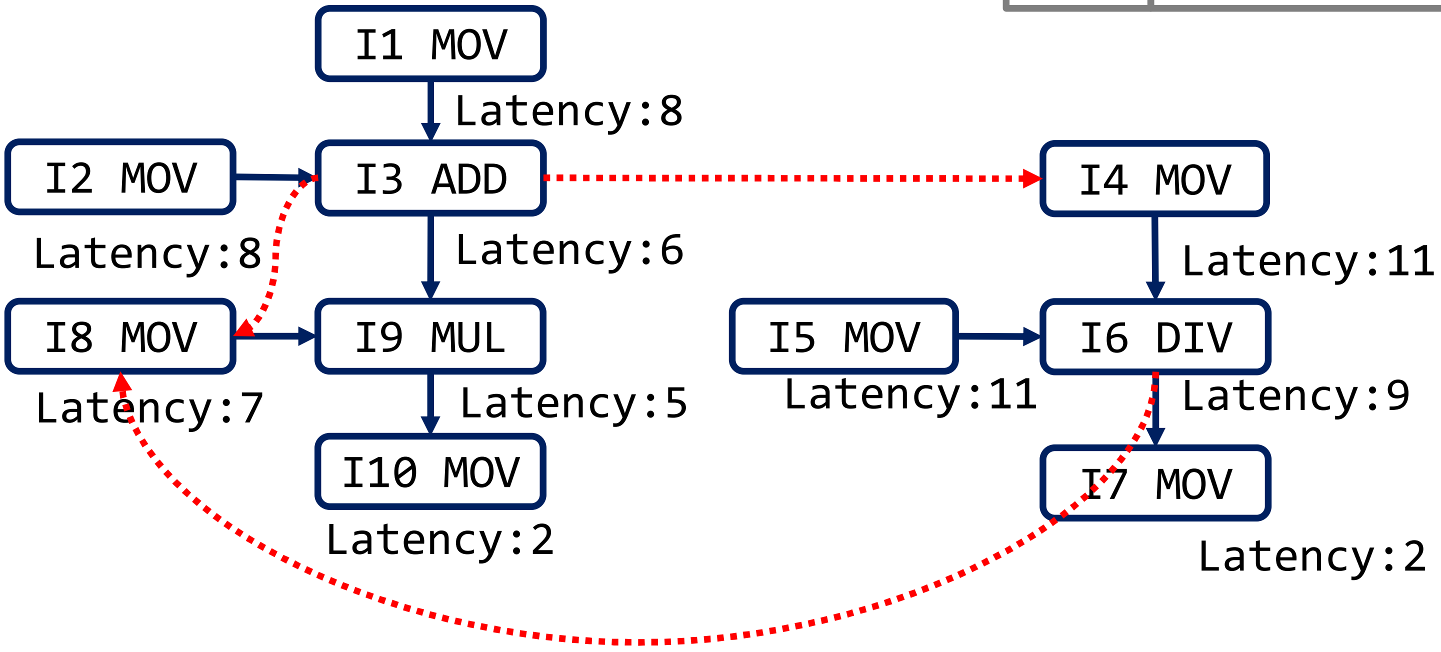


$$\begin{tikzcd}
 & \text{I1 MOV (Latency:8)} \arrow[d] & & \text{I4 MOV (Latency:11)} \arrow[d] \\
\text{I2 MOV (Latency:8)} \arrow[r] & \text{I3 ADD (Latency:6)} \arrow[d] & \text{I5 MOV (Latency:11)} \arrow[r] & \text{I6 DIV (Latency:9)} \arrow[d] \\
\text{I8 MOV (Latency:7)} \arrow[r] & \text{I9 MUL (Latency:5)} \arrow[d] & & \text{I7 MOV (Latency:2)} \\
 & \text{I10 MOV (Latency:2)} & &
\end{tikzcd}$$

反依赖问题

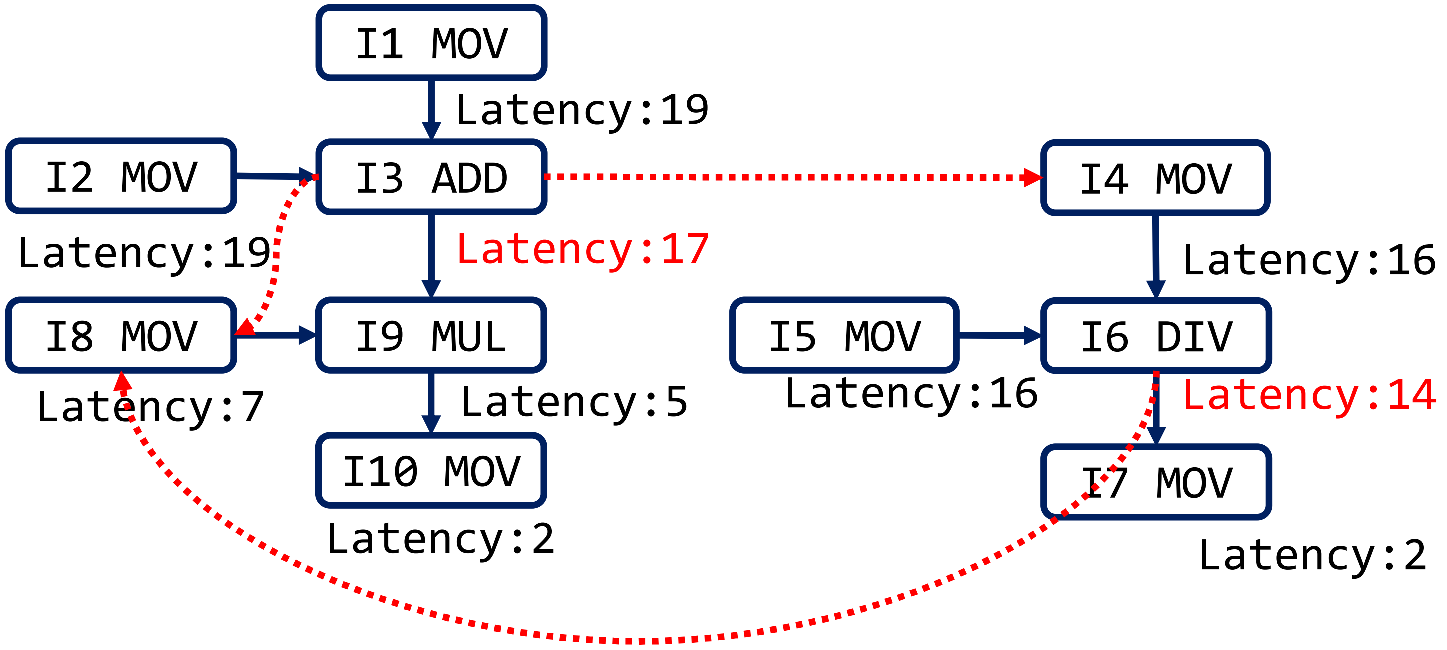
- 读-写反依赖（anti-denpendency）
  - I3执行完I4和I8才能执行；
    - 否则会影响I3的计算结果
  - I6执行完才能执行I8；

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？

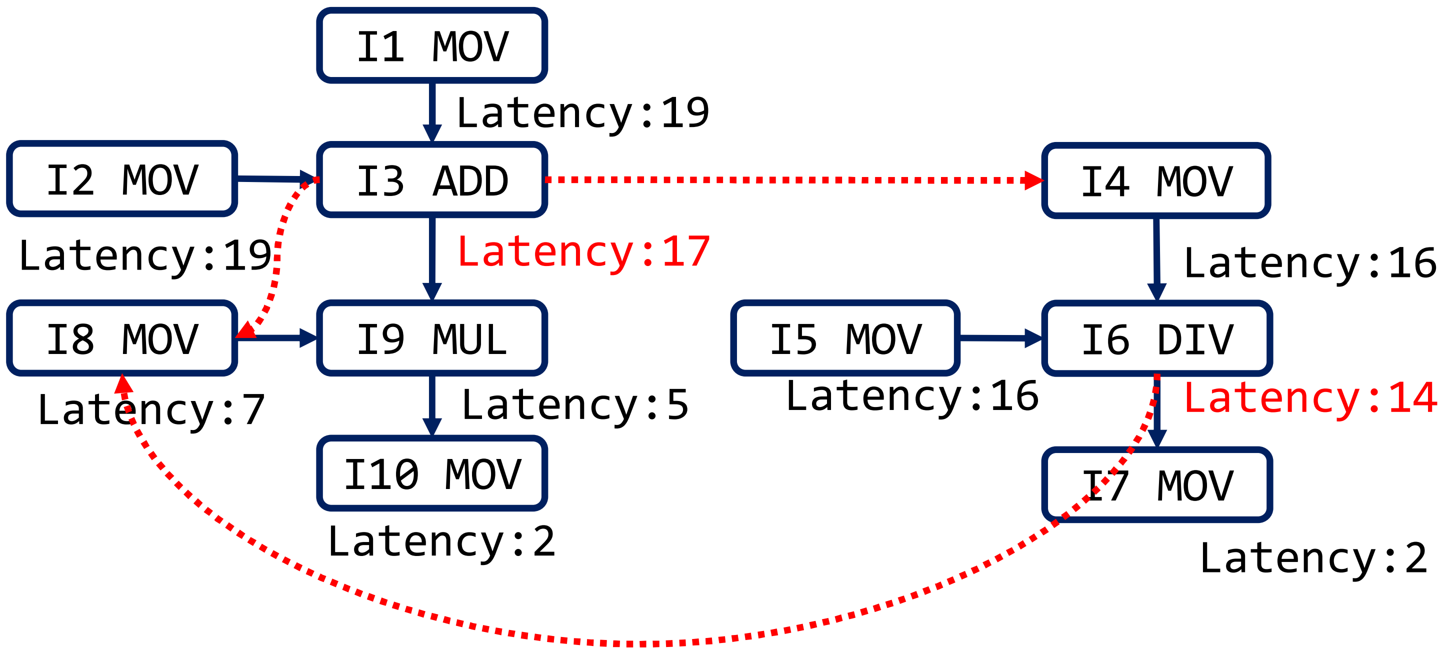
- Latency：I4=I5>I6>I1=I2>I8>I3>I9>I7=I10
  - I3早于I4、I8；
  - I6早于I8；
- 更新Latency：I1=I2>I3>I4=I5>I6>I8>I9>I7=I10



调度方案开销

- I1=I2>I3>I4=I5>I6>I8>I9>I7=I10
  - 开销：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
18	19	I7	MOV %eax, $-24(%rsp)
20	21	I10	MOV r2, $-28(%rsp)

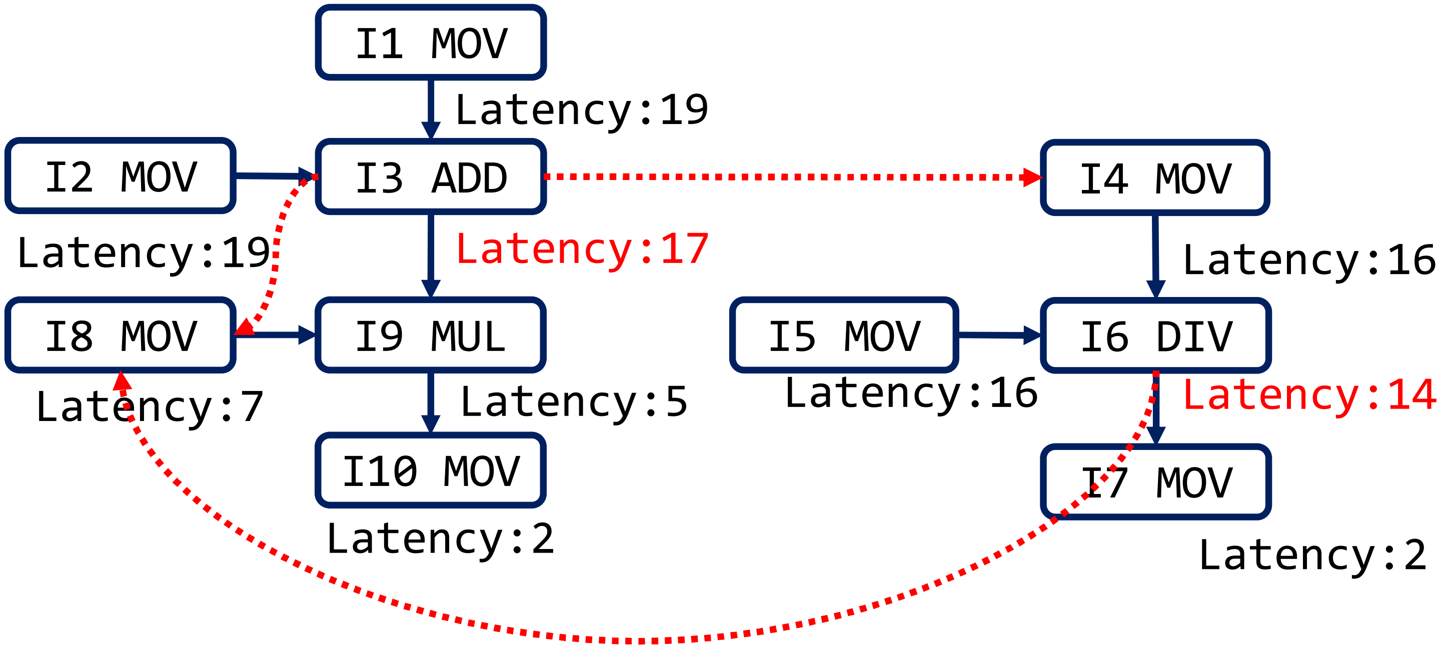


应对反依赖：重命名

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)

⇨

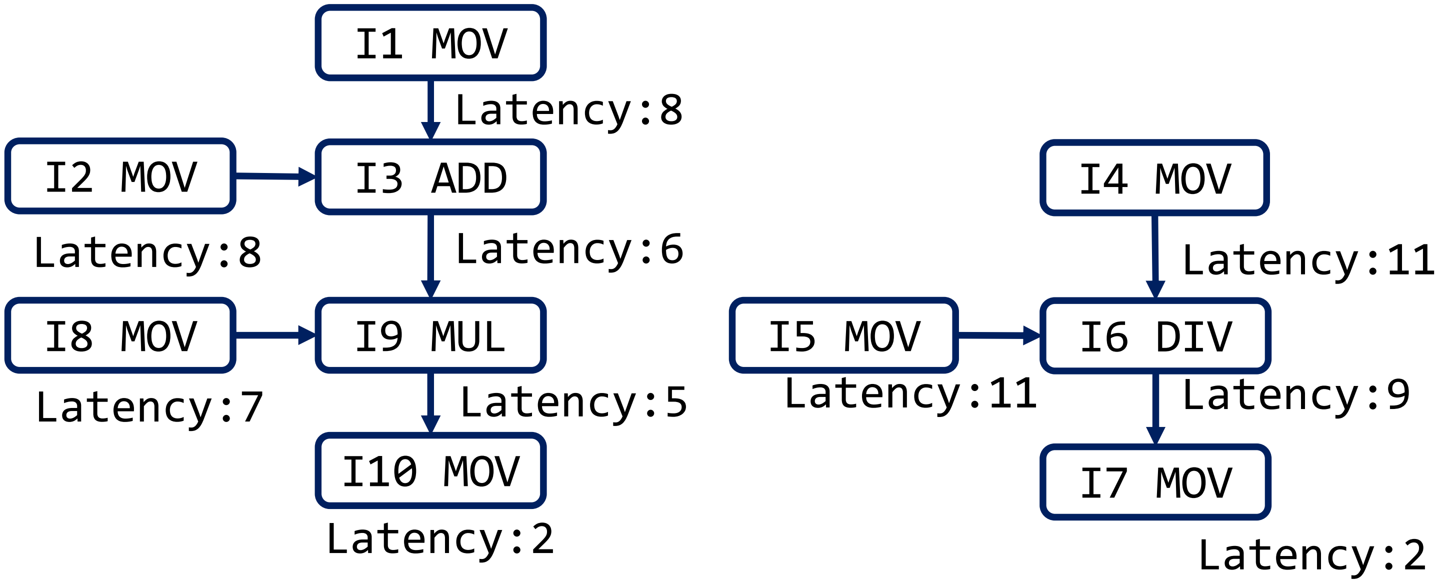
I1	MOV $-12(%rsp), r1
I2	MOV $-16(%rsp), r2
I3	ADD r2, r1
I4	MOV $-20(%rsp), r3
I5	MOV $-24(%rsp), %eax
I6	DIV r3, ~~%eax~~
I7	MOV %eax, $-24(%rsp)
I8	MOV $-28(%rsp), r4
I9	MUL r1, r4
I10	MOV r4, $-28(%rsp)



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)



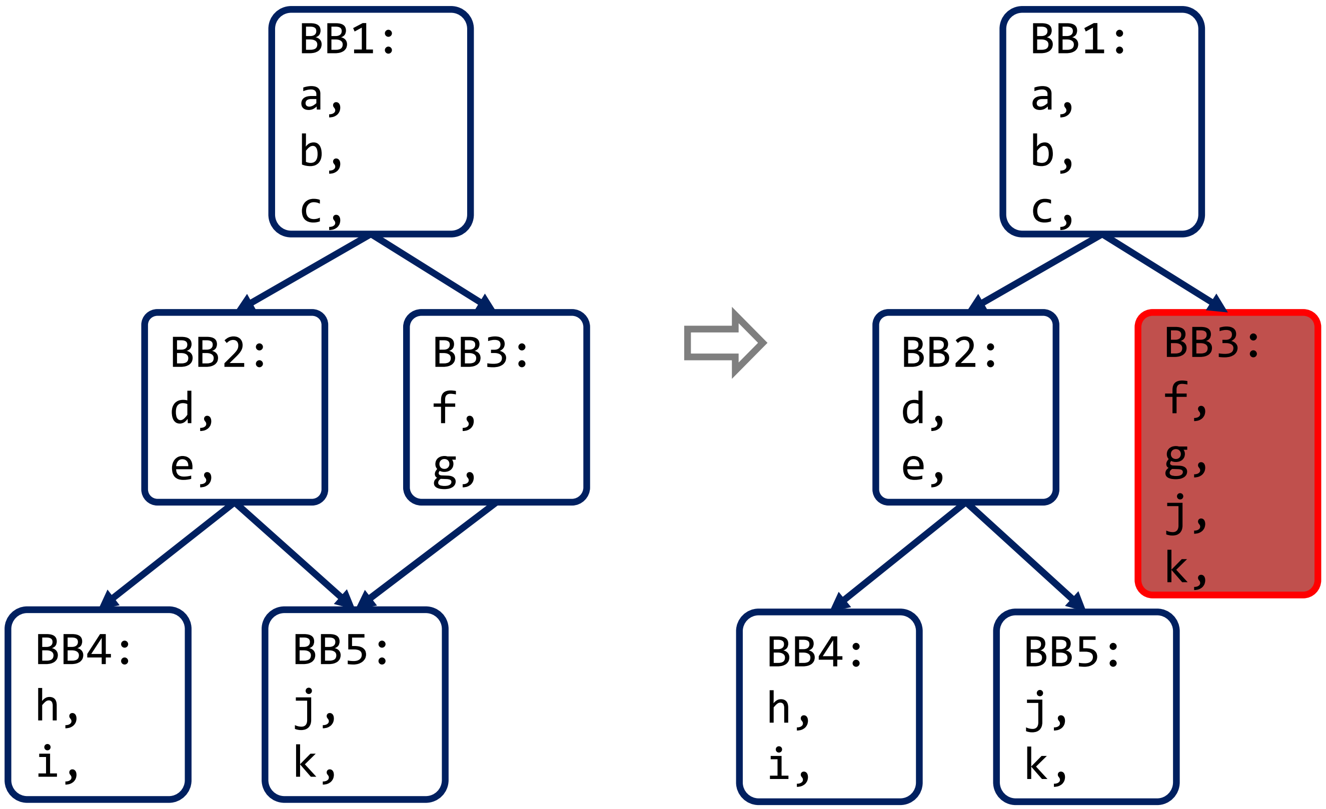
表调度算法

```
Clock = 1
Ready = {指令依赖图的所有叶子节点}
Active = {}
While (Ready ∪ Active ≠ Ø){
    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 ≠ Ø){
        Ready.remove(any I);
        Start(I) = Clock;
        Active.add(I);
    }
    Clock = Clock + 1;
}
```

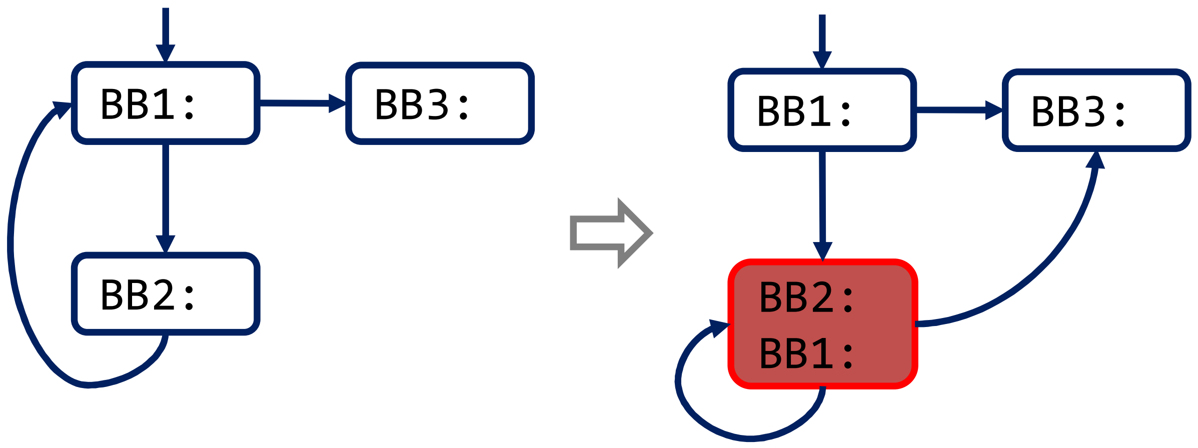
- 假设：
  - 线性代码；
  - 无反依赖。
- 两张表：
  - Ready表记录已满足数据依赖的指令；
  - Active表记录正在执行的指令。
- 方法：
  - 每个Clock尽量执行一条新的指令；
  - 如果Active表有指令执行完成，考虑将指令的next指令加入Ready。

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



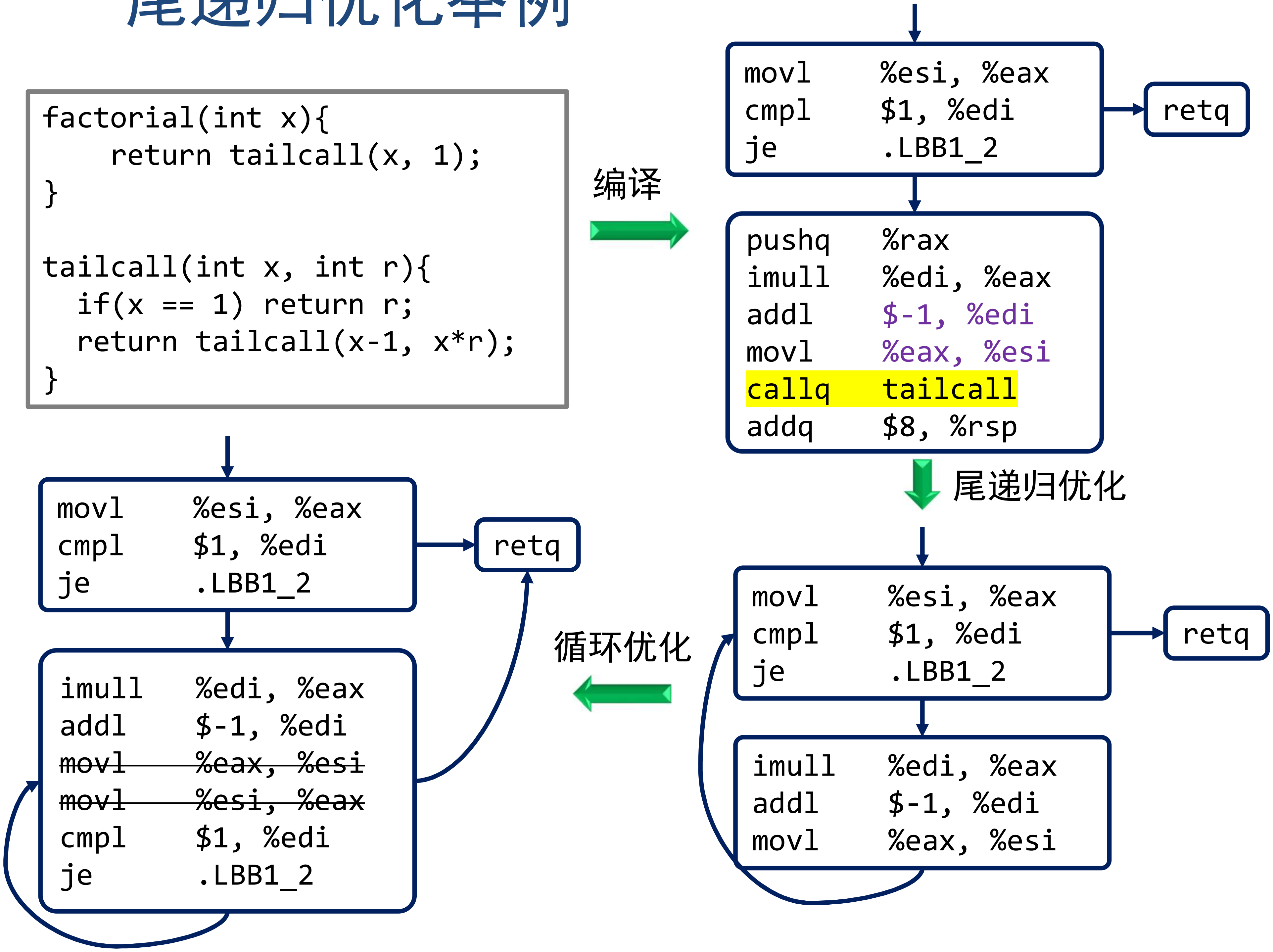
应用：循环优化



尾递归优化后的程序　　　　复制后

尾递归优化举例

```
factorial(int x){
    return tailcall(x, 1);
}

tailcall(int x, int r){
  if(x == 1) return r;
  return tailcall(x-1, x*r);
}
```



```
movl     %esi, %eax
cmpl     $1, %edi
je       .LBB1_2
```

```
retq
```

```
pushq    %rax
imull    %edi, %eax
addl     $-1, %edi
movl     %eax, %esi
callq    tailcall
addq     $8, %rsp
```

```
movl     %esi, %eax
cmpl     $1, %edi
je       .LBB1_2
```

```
retq
```

```
imull    %edi, %eax
addl     $-1, %edi
movl     %eax, %esi
```

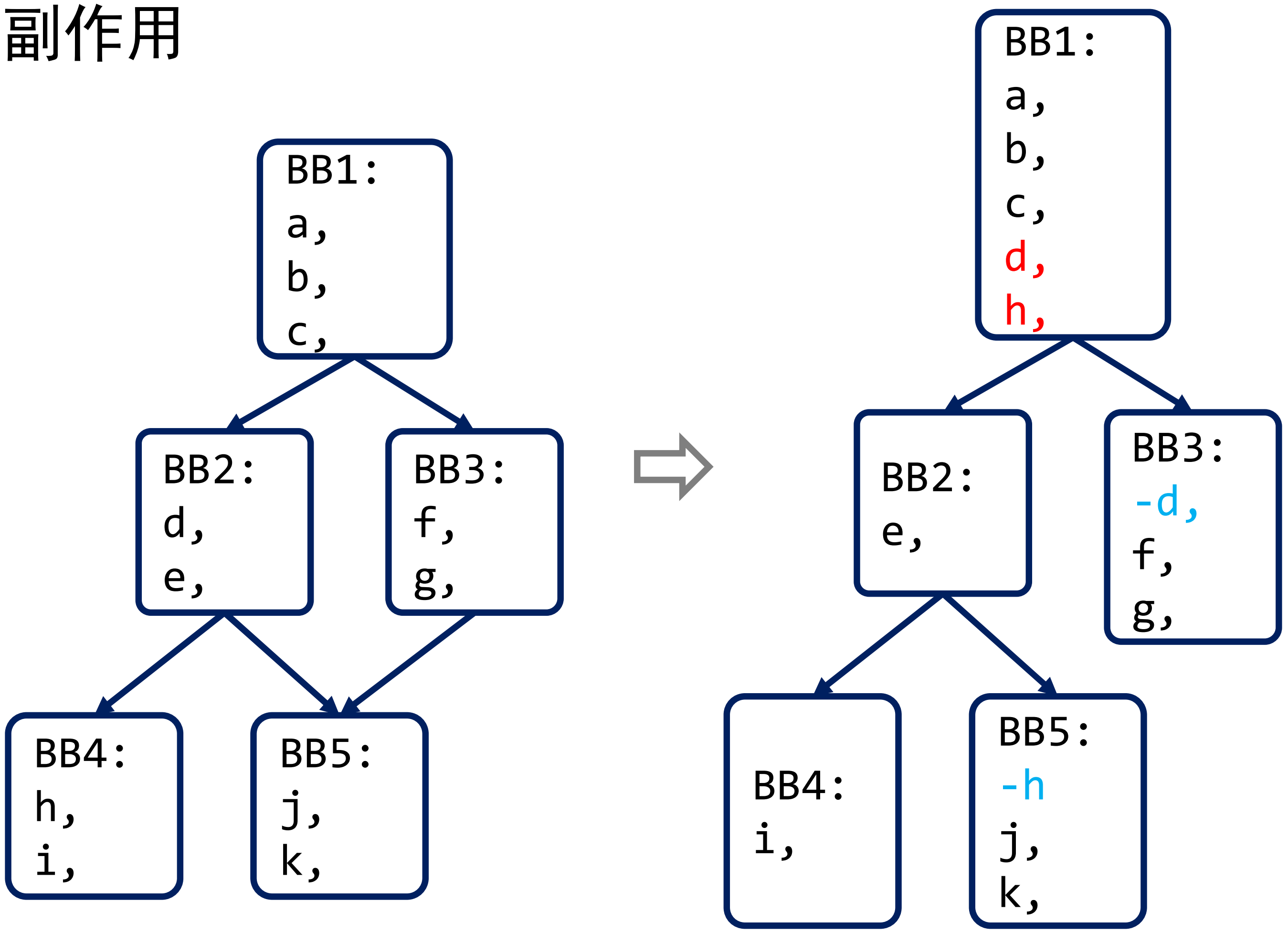
```
movl     %esi, %eax
cmpl     $1, %edi
je       .LBB1_2
```

```
retq
```

```
imull    %edi, %eax
addl     $-1, %edi
~~movl     %eax, %esi~~
~~movl     %esi, %eax~~
cmpl     $1, %edi
je       .LBB1_2
```

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



四、寄存器分配和优化

Register Allocation

寄存器分配问题

- 指令翻译没有限制寄存器的数量
  - 假设临时变量都是存在寄存器中的
  - 但通用寄存器的数量是有限的。
- 如何使用最少数量的寄存器？
  - 如最少数量仍然超过寄存器数量，则需将数据存入内存，使用时再读取，影响性能。

活跃性（Liveness）分析

- 一个变量如果接下来还会被使用，则这个变量是活跃的。

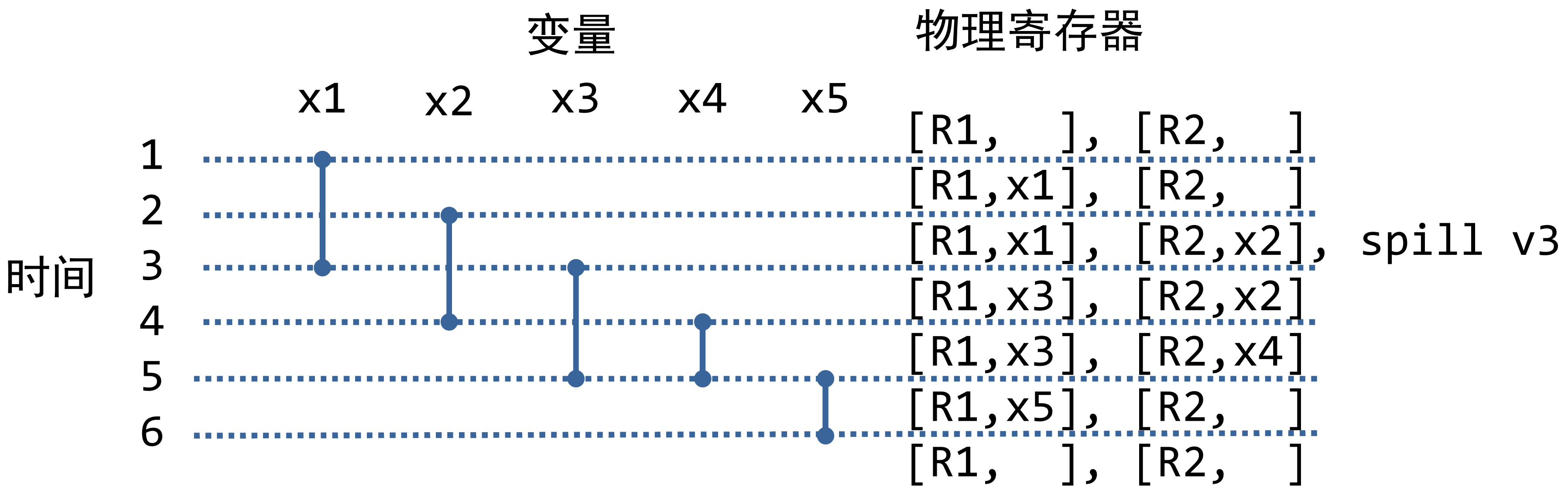
```
x1 = 0
x2 = 1
x3 = x1 + x2
x4 = x2 + x3
x5 = x3 + x4
%eax = x5
ret
```

活跃变量

```
∅
x1
x1,x2
x2,x3
x3,x4
X5
%eax
```

Linear Scan

- 输入每个变量的活跃周期，按时间顺序先到先得；
- 不考虑全局因素。

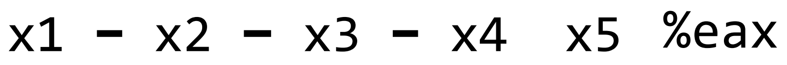


M. Poletto, and V. Sarkar, Linear scan register allocation. *TOPLAS*, 1999
O. Traub, *et al*. Quality and speed in linear-scan register allocation. *ACM SIGPLAN Notices, 1998*.

冲突图（Interference Graph）

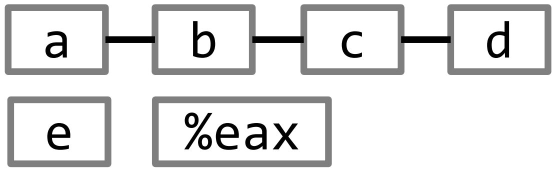
- 冲突：任意赋值指令def(t1)与在该指令后，use(t1)之前的活跃变量需要同时占用寄存器。
- 冲突图：创建从t1出发的边，连接其它在该指令后，use(t1)之前活跃的变量。
- 物理含义：有连线关系的两个变量在某一时刻同时存活，最好分配不同的寄存器。

	活跃变量
x1 = 0	∅
x2 = 1	x1
x3 = x1 + x2	x1,x2
x4 = x2 + x3	x2,x3
x5 = x3 + x4	x3,x4
%eax = x5	X5
ret	%eax



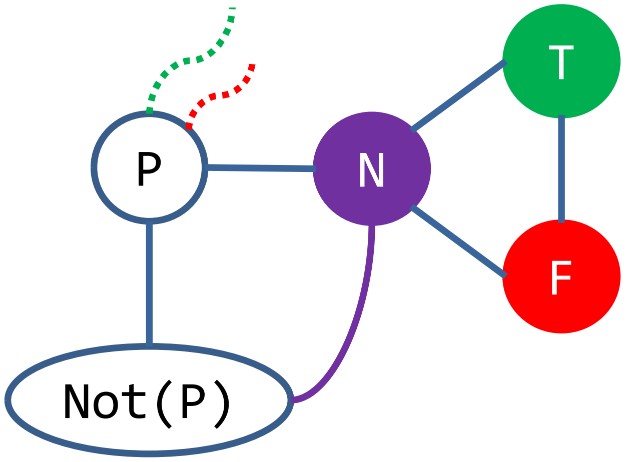
基于着色问题（Graph Coloring）的解法

- 最多使用K种颜色为冲突图着色，要求相邻节点颜色均不同；
- 当K≥3时，该问题是NP完全问题；
  - Chaitin的证明
- 解法：贪心算法。

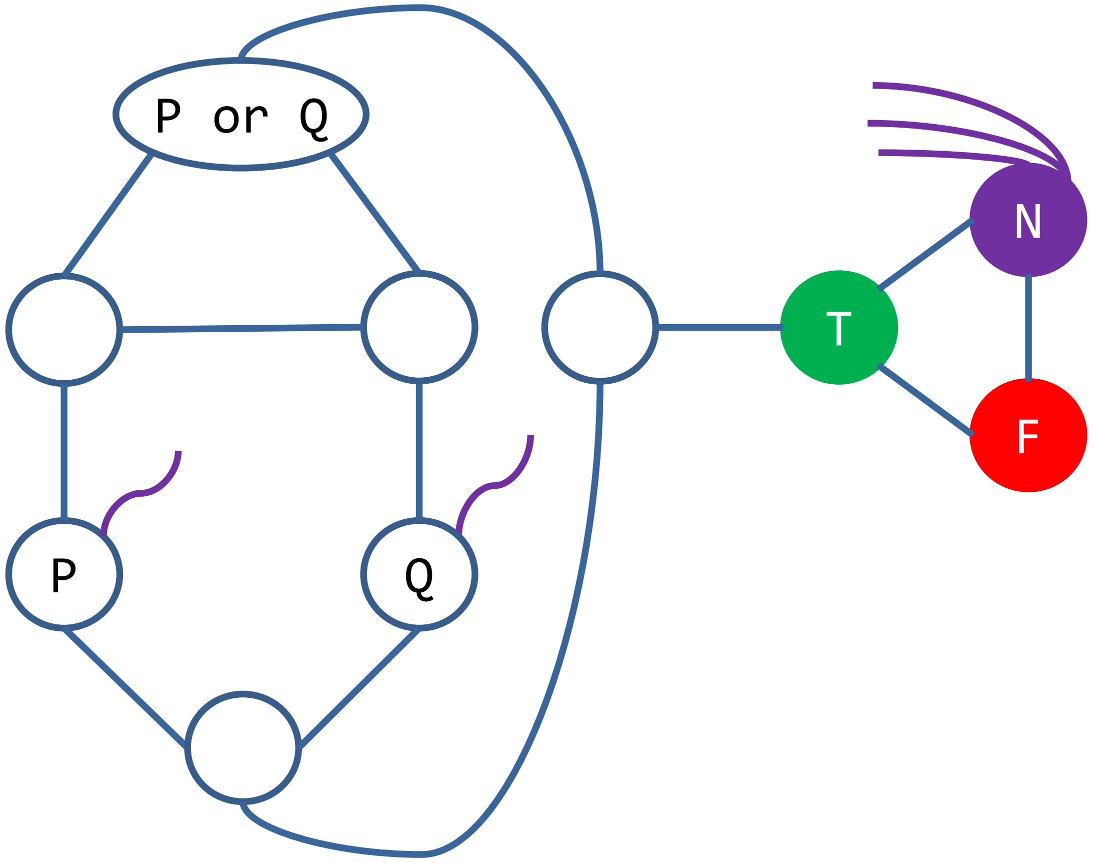


G.J. Chaitin, *et al*, Register allocation via coloring. *Computer languages*, 1981.

证明思路：SAT可以reduce到着色问题



构造not门

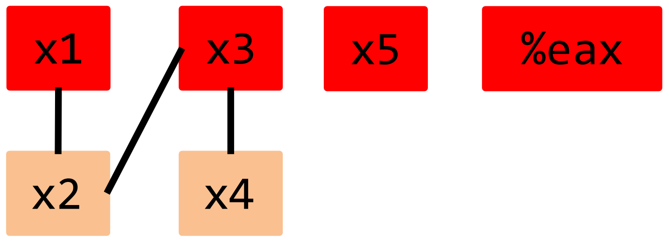


构造or门

Albert R. Meyer和他的学生，MIT

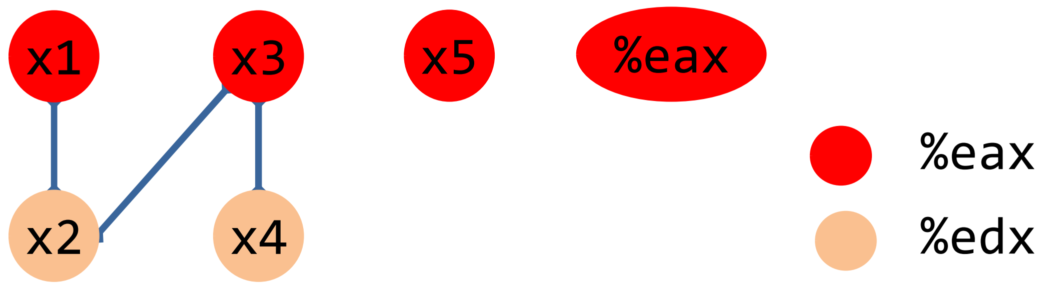
贪心算法着色

- 策略：根据邻居节点颜色，为当前节点选取可用的颜色；
- 假设变量的着色顺序是x1,x2,x3,x4,x5



```
贪心算法着色
Input: G=(V,E)
Output: Assignment of colors
For i = 1..n do
  Let c be the lowest color not used in Neighbor(vi)
  Set Col(vi) = c
```

寄存器分配后的程序



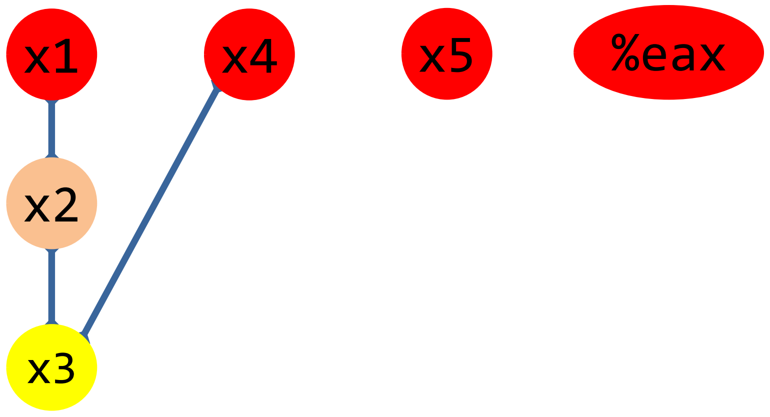
```
x1 = 0
x2 = 1
x3 = x1 + x2
x4 = x2 + x3
x5 = x3 + x4
%eax = x5
ret
```

⇨

```
MOV $0, %eax
MOV $1, %edx
ADD %edx, %eax
ADD %eax, %edx
ADD %edx, %eax

ret
```

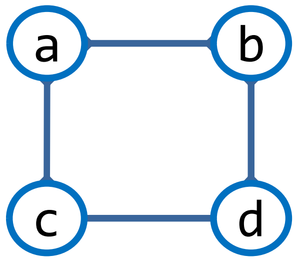
贪心着色算法有时不能求到最优解

- 假设变量的着色顺序是x1,x4,x2,x3,x5，需要使用3种颜色着色。

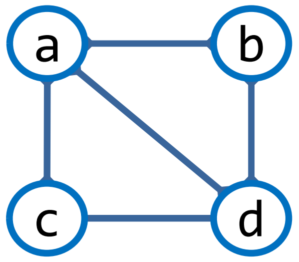


一类特殊的着色问题：弦图chordal graph

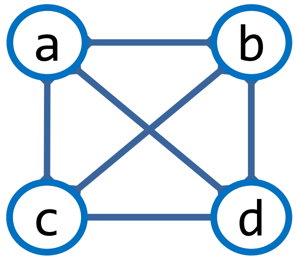
- 任意长度大于3个环都有弦（chord）；
- 多项式时间可解。



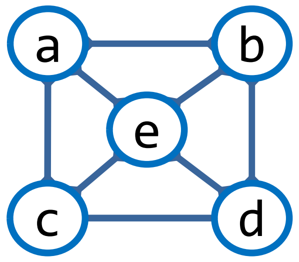
非弦图



弦图



弦图



非弦图

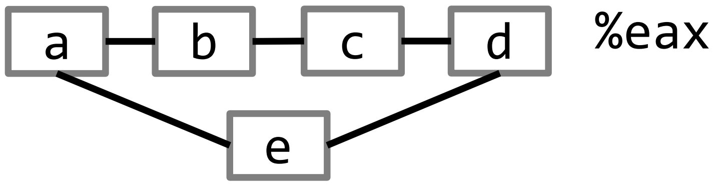
SSA程序的冲突图是弦图

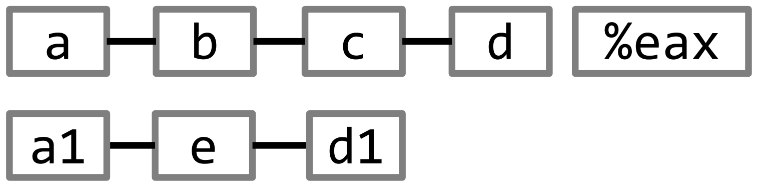
- 变量重命名会将环截断

```
                      活跃变量
a = 0                 Ø
b = 1                 a
c = a + b             a,b
d = b + c             b,c
a = c + d             c,d
e = 7                 a
d = a + e             a,e
%eax = d + e          d,e
ret                   %eax
```

⇒

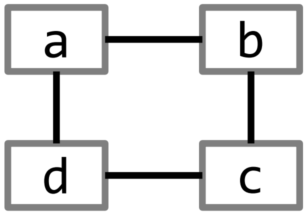
```
                      活跃变量
a = 0                 Ø
b = 1                 a
c = a + b             a,b
d = b + c             b,c
a1 = c + d            c,d
e = 7                 a1
d1 = a1 + e           a1,e
%eax = d1 + e         d1,e
ret                   %eax
```





为什么SSA程序的冲突图是弦图？

- 如何构造非弦图？

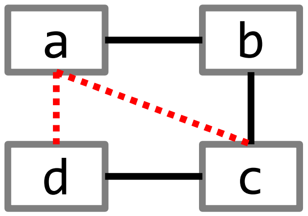


尝试1：

```
c = a + b
d = b + c
e = c + d
f = a + d
```

活跃变量

```
a,b
b,c a
c,d a
a,d
```



尝试2：

```
c = a + b
d = b + c
e = c + d
a = 1
f = a + d
```

活跃变量

```
a,b
b,c
c,d
d
a,d
```

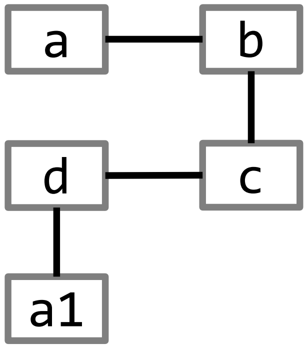
⇨

SSA

```
c = a + b
d = b + c
e = c + d
a1 = 1
f = a1 + d
```

活跃变量

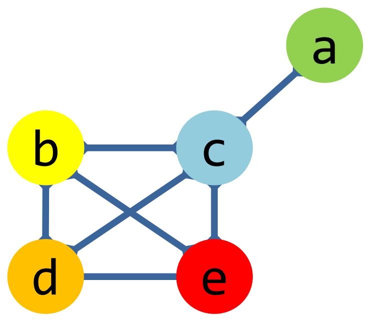
```
a,b
b,c
c,d
d
a1,d
```

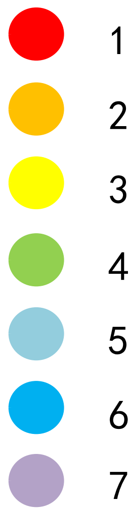


Sebastian Hack and Gerhard Goos, Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 2006.

着色思路

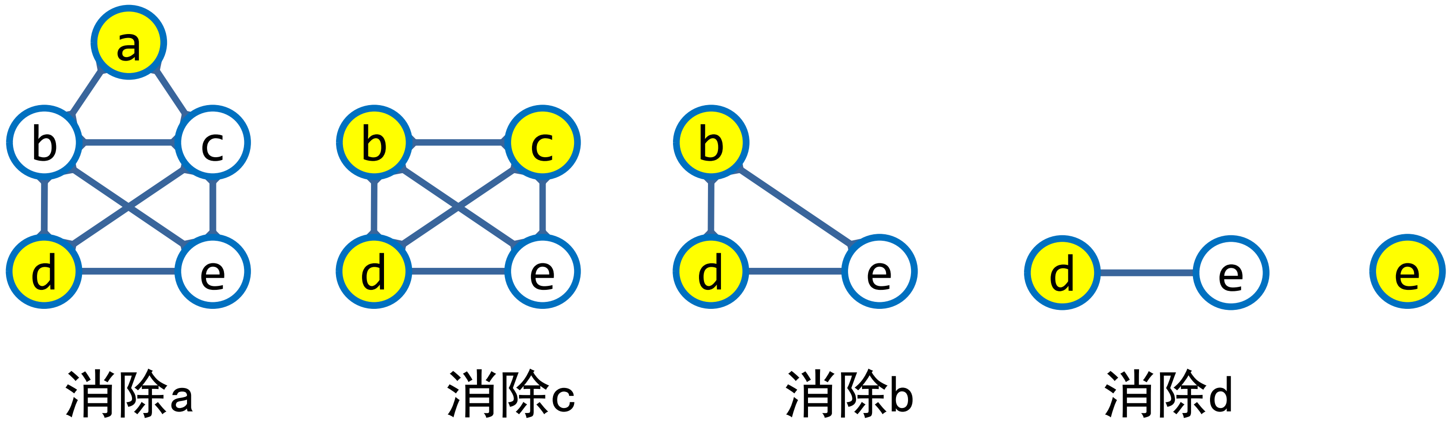
- 着色所需颜色数与最大团(clique)的大小一致。
  - clique：所有节点都互相连接；
  - 含义：需要不同的颜色；
  - 找最大团？也是np-hard问题。
- 着色顺序不应引入非团节点带来的颜色限制：
  - 单纯消除序列可达到上述目的。

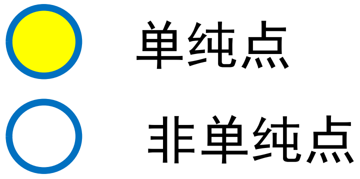




单纯点

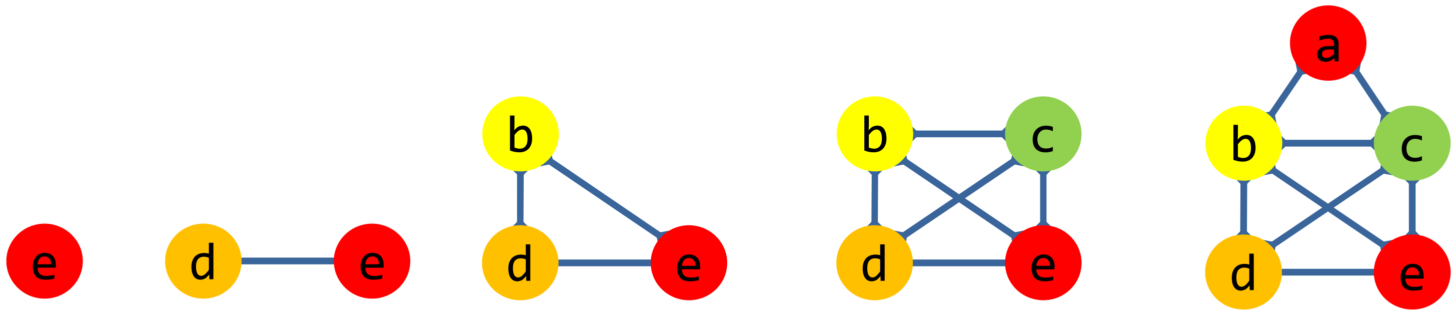
- 单纯点（simplicial）：邻居是一个团（clique）。
- 完美消除序列：图G的所有节点存在一个排列{v1,…,vn}，按照该序列消除的每一个点都是单纯点。
- 弦图：如果一个图是弦图，则该图存在完美消除序列。



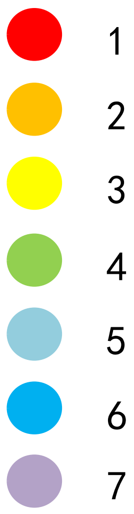


单纯消除序列simplicial elimination ordering

- 完美消除序列的逆序；
- 每次在（已染色）clique的基础上增加一个点连接所有点；
- 假设颜色的顺序为红、橙、黄、绿、青、蓝、紫

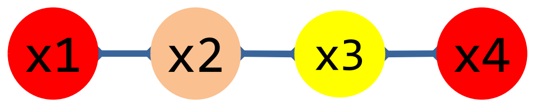


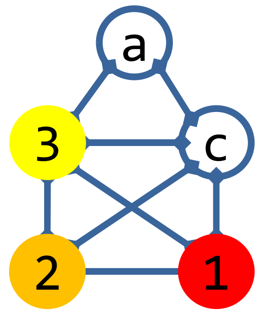
单纯消除序列和着色过程

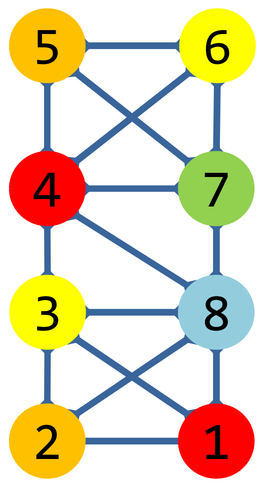


非单纯消除序列不一定是最优解

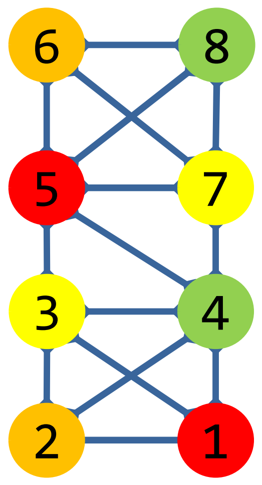
- x1、x4、x2、x3是否是单纯消除序列？
  - 否，逆序不是完美消除序列，
  - X3不是单纯点
- 下图何时先给a着色会导致所需颜色数超过最大团大小？

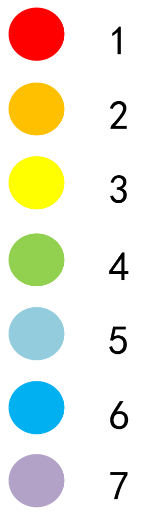






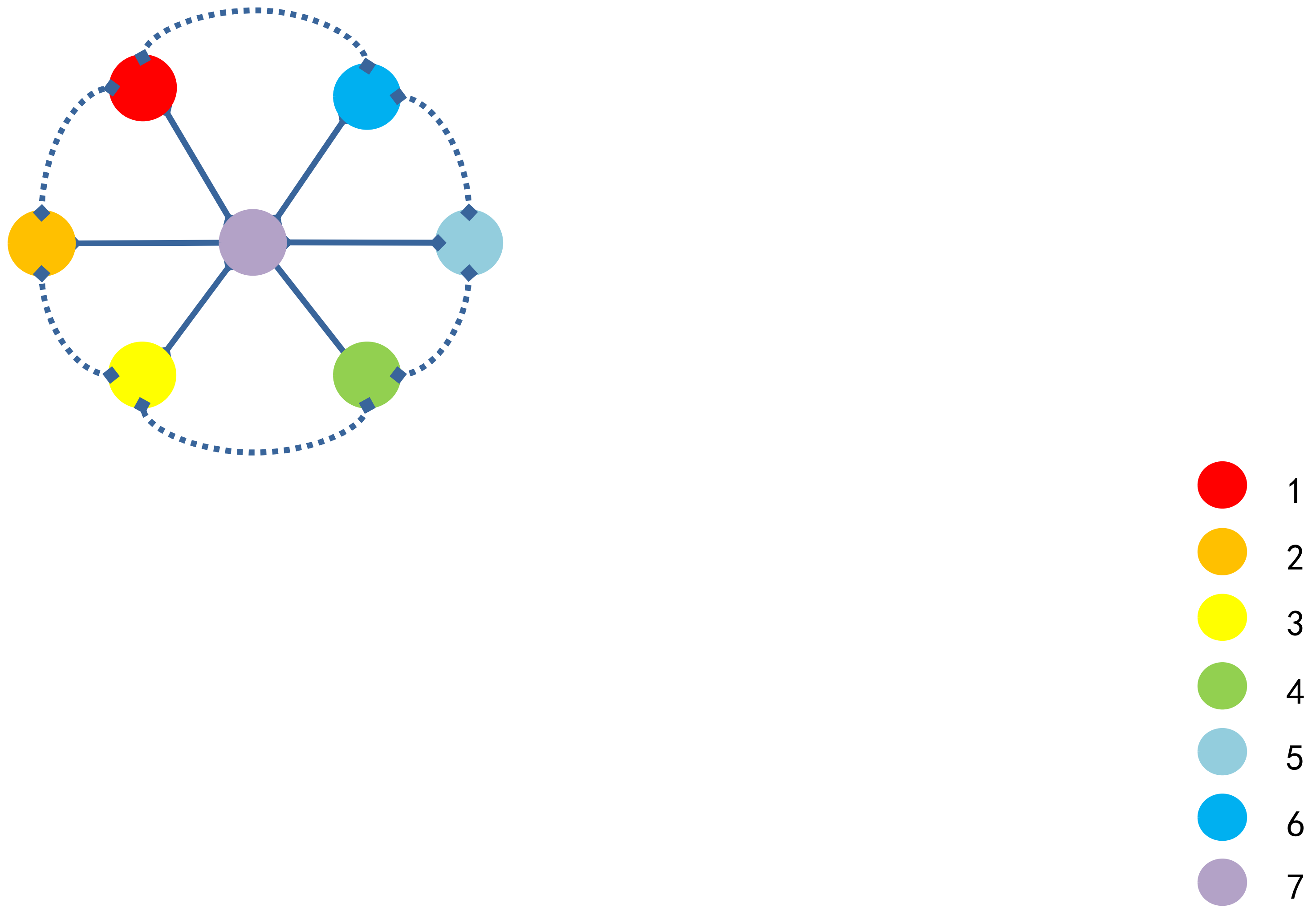
需要5个颜色





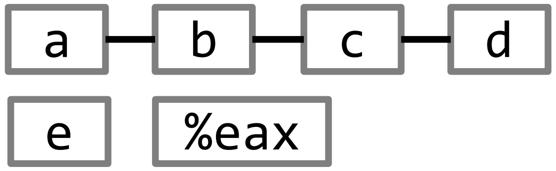
为什么单纯消除序列一定是最优解？

- 序列中存在非单纯点会导致颜色浪费；
- 单纯点的颜色可在其它区域复用。



最大势算法求单纯消除序列

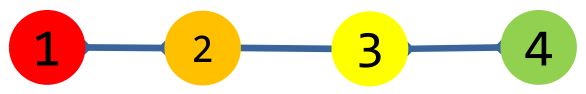
- Maximum Cardinality Search
- 思路：搜索与已着色节点邻居最多的点
  - 维护一个所有点的向量，每次选取值最大的点；
  - 选取一个点后，则其邻居计数加1。

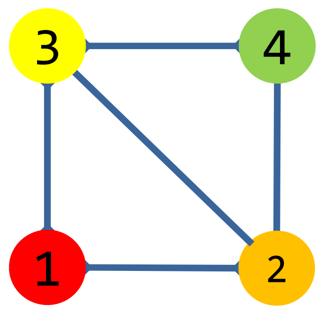


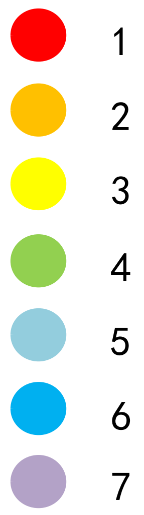
步骤	选取	a	b	c	d	e	%eax
		0	0	0	0	0	0
1	a		1	0	0	0	0
2	b			1	0	0	0
3	c				1	0	0
4	d					0	0
5	e						0
6	%eax						

为什么能得到单纯消除序列?

- 什么情况会引入单纯点?
  - 1-2-4-3





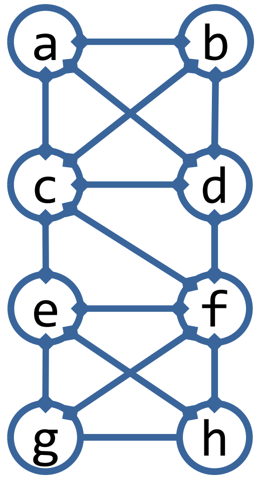


算法参考

```
**Maximum Cardinality Search**
Input: $G = (V, E)$
Output: Simplicial elimination ordering $v_1, \ldots, v_n$
For all $v_i \in V$
  $w(v_\mathrm{i}) = 0$
Let $W = V$
For i = 1,..,n do
  Let $v$ be a node with max weight in $W$
  Set $v_i = v$
  For all $u \in W \cap N(v)$
    $w(u) = w(u) + 1$
  $W = W \backslash \{v\}$
```

练习

- 求下列冲突图的单纯消除序列



步骤	选取	a	b	c	d	e	f	g	h
		0	0	0	0	0	0	0	0
1	a		1	1	1	0	0	0	0
2	b			2	2	0	0	0	0
3	c				3	1	1	0	0
4	d					1	2	0	0
5	f					2		1	1
6	e							2	2
7	g								3
8	h								

寄存器预着色

- 有些变量必须被分配到指定寄存器

X86-64寄存器	用途	注释
%rax	返回值	Caller-saved
%rdi	参数1	Caller-saved
%rsi	参数2	Caller-saved
%rdx	参数3	Caller-saved
%rcx	参数4	Caller-saved
%r8	参数5	Caller-saved
%r9	参数6	Caller-saved
%10-%11		Caller-saved
%ebp	栈基指针	Callee-saved
%rsp	栈顶指针	Callee-saved
%rbx		Callee-saved
%12-%r15		Callee-saved

寄存器分配步骤

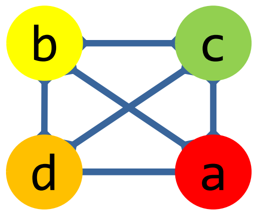
1) 生成冲突图（interference graph）
2) 计算单纯消除序列
3) 根据消除序列将冲突图着色
4) 如果颜色不够则怎么办？Spill
   - 临时存入内存，然后加载回寄存器

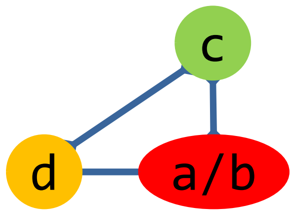
其它考虑：Coalesce

- 是否合并mov语句的两个寄存器参数？

```
a = 0
b = a + 1
c = a + b
d = a
e = a + c + d
```

活跃变量
∅
a
a,b
a,b,c
a,b,c,d





●	1
●	2
●	3
●	4
●	5
●	6
●	7

考虑控制流图？

- 通过数据流分析算法在提取变量活跃信息；
  - 以函数为对象
- 后面代码优化课会详细讲变量活跃分析算法。

流程小结：中间代码翻译为汇编代码

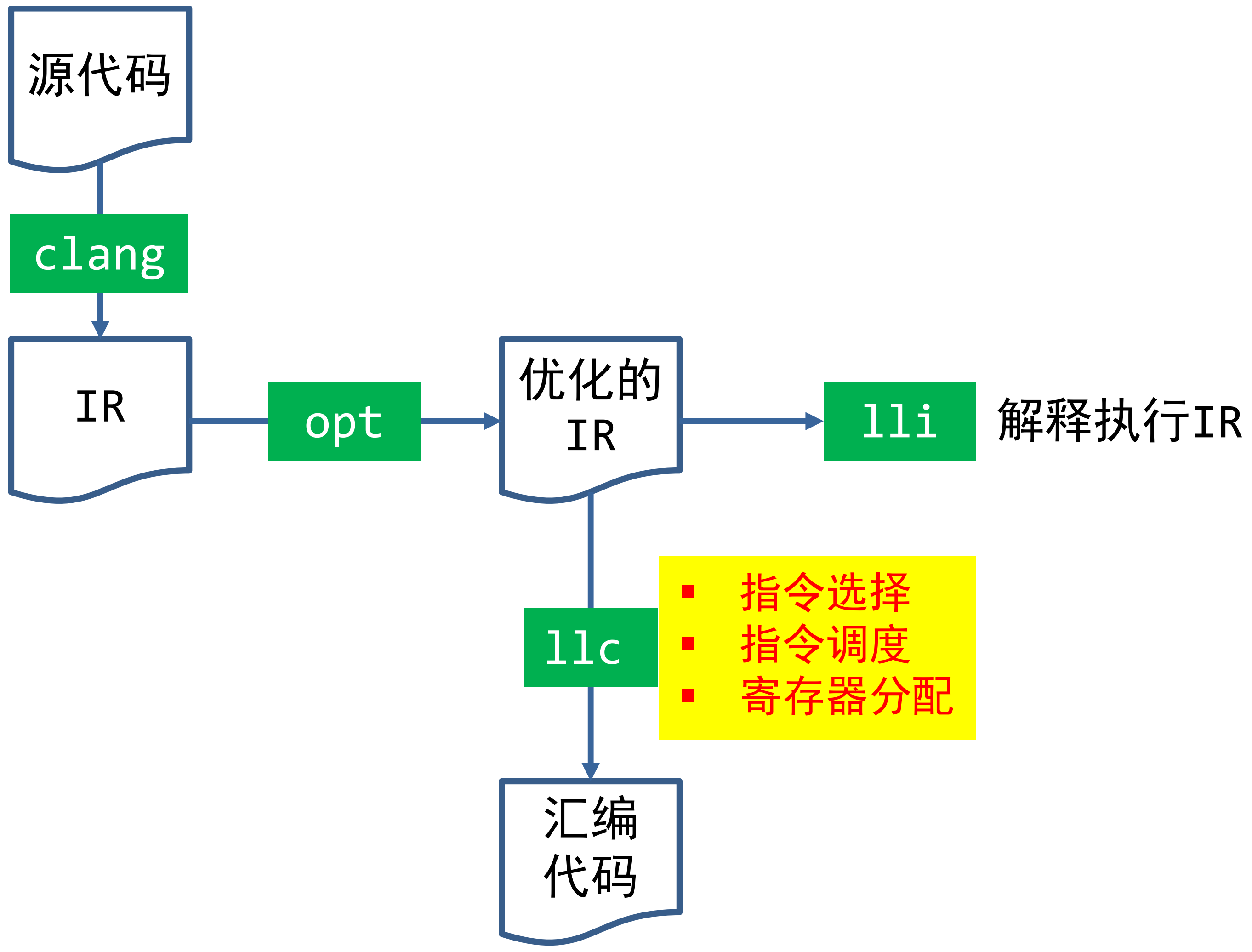
- 指令选择：铺树问题
  - 思考：汇编没有phi指令，应如何处理？
- 指令调度：不破坏数据依赖重排指令
- 寄存器分配：着色问题
  - 思考：是否需要将SSA还原？

思考

- 通过指令调度是否可以优化寄存器使用？
  - 构造一个程序说明

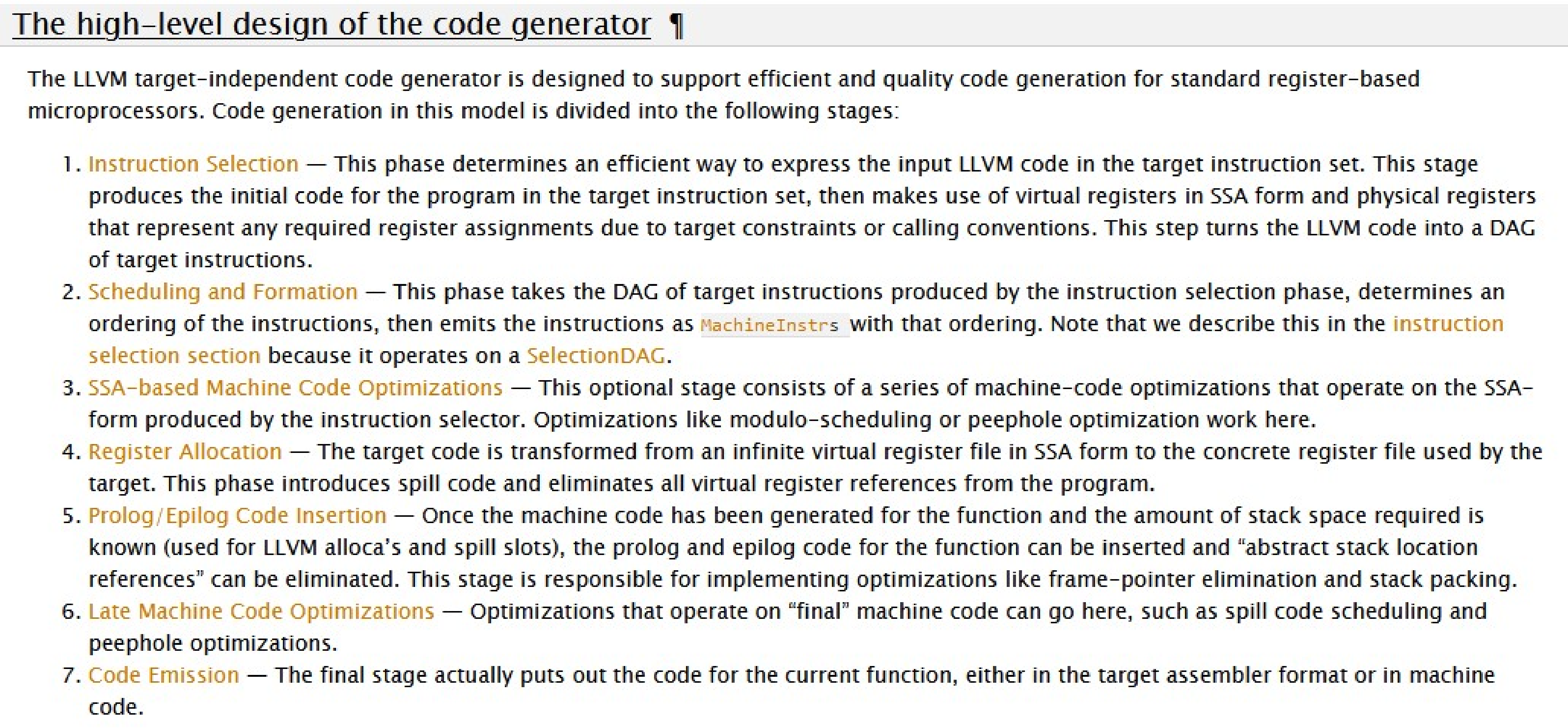
五、LLVM案例参考

基本框架和工具



https://llvm.org/docs/CommandGuide/index.html

LLVM代码生成的步骤



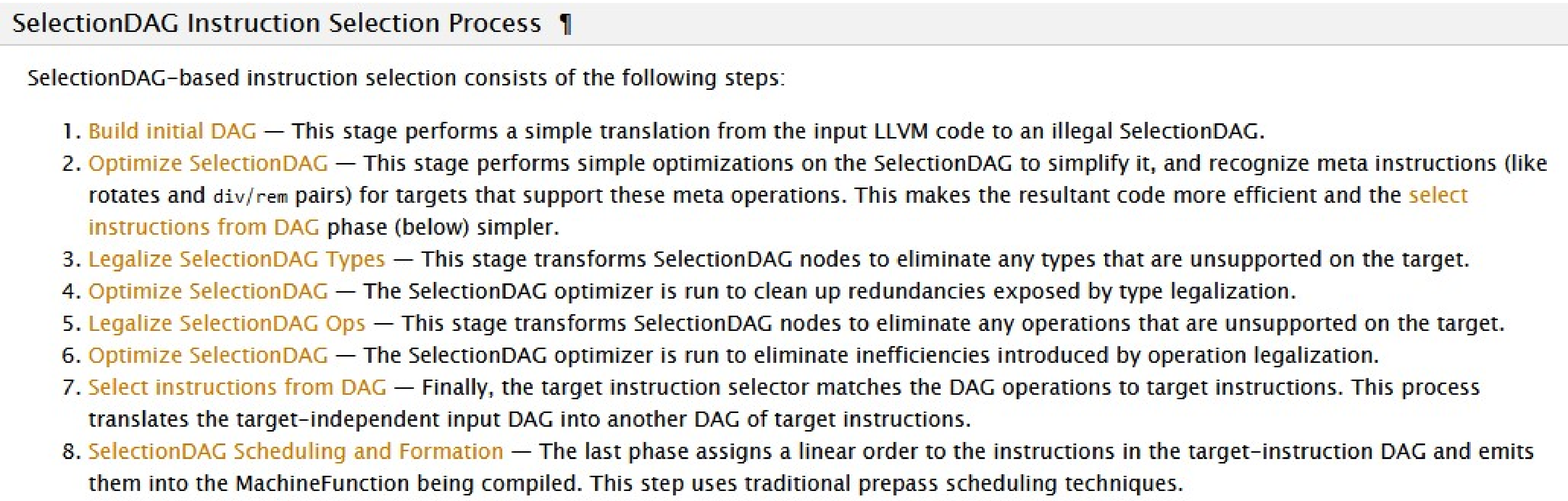
The high-level design of the code generator ¶

The LLVM target-independent code generator is designed to support efficient and quality code generation for standard register-based microprocessors. Code generation in this model is divided into the following stages:

1. Instruction Selection — This phase determines an efficient way to express the input LLVM code in the target instruction set. This stage produces the initial code for the program in the target instruction set, then makes use of virtual registers in SSA form and physical registers that represent any required register assignments due to target constraints or calling conventions. This step turns the LLVM code into a DAG of target instructions.
2. Scheduling and Formation — This phase takes the DAG of target instructions produced by the instruction selection phase, determines an ordering of the instructions, then emits the instructions as `MachineInstrs` with that ordering. Note that we describe this in the instruction selection section because it operates on a SelectionDAG.
3. SSA-based Machine Code Optimizations — This optional stage consists of a series of machine-code optimizations that operate on the SSA-form produced by the instruction selector. Optimizations like modulo-scheduling or peephole optimization work here.
4. Register Allocation — The target code is transformed from an infinite virtual register file in SSA form to the concrete register file used by the target. This phase introduces spill code and eliminates all virtual register references from the program.
5. Prolog/Epilog Code Insertion — Once the machine code has been generated for the function and the amount of stack space required is known (used for LLVM alloca's and spill slots), the prolog and epilog code for the function can be inserted and "abstract stack location references" can be eliminated. This stage is responsible for implementing optimizations like frame-pointer elimination and stack packing.
6. Late Machine Code Optimizations — Optimizations that operate on "final" machine code can go here, such as spill code scheduling and peephole optimizations.
7. Code Emission — The final stage actually puts out the code for the current function, either in the target assembler format or in machine code.

https://llvm.org/docs/CodeGenerator.html

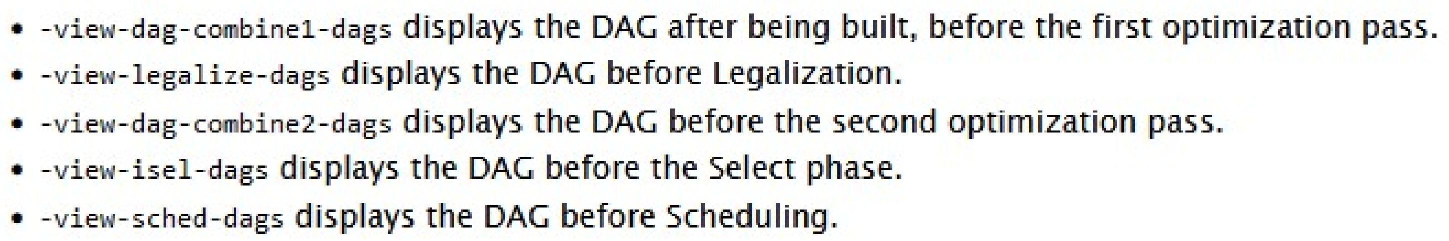
LLVM代码生成的步骤



SelectionDAG Instruction Selection Process ¶

SelectionDAG-based instruction selection consists of the following steps:

1. Build initial DAG — This stage performs a simple translation from the input LLVM code to an illegal SelectionDAG.
2. Optimize SelectionDAG — This stage performs simple optimizations on the SelectionDAG to simplify it, and recognize meta instructions (like rotates and `div`/`rem` pairs) for targets that support these meta operations. This makes the resultant code more efficient and the select instructions from DAG phase (below) simpler.
3. Legalize SelectionDAG Types — This stage transforms SelectionDAG nodes to eliminate any types that are unsupported on the target.
4. Optimize SelectionDAG — The SelectionDAG optimizer is run to clean up redundancies exposed by type legalization.
5. Legalize SelectionDAG Ops — This stage transforms SelectionDAG nodes to eliminate any operations that are unsupported on the target.
6. Optimize SelectionDAG — The SelectionDAG optimizer is run to eliminate inefficiencies introduced by operation legalization.
7. Select instructions from DAG — Finally, the target instruction selector matches the DAG operations to target instructions. This process translates the target-independent input DAG into another DAG of target instructions.
8. SelectionDAG Scheduling and Formation — The last phase assigns a linear order to the instructions in the target-instruction DAG and emits them into the MachineFunction being compiled. This step uses traditional prepass scheduling techniques.



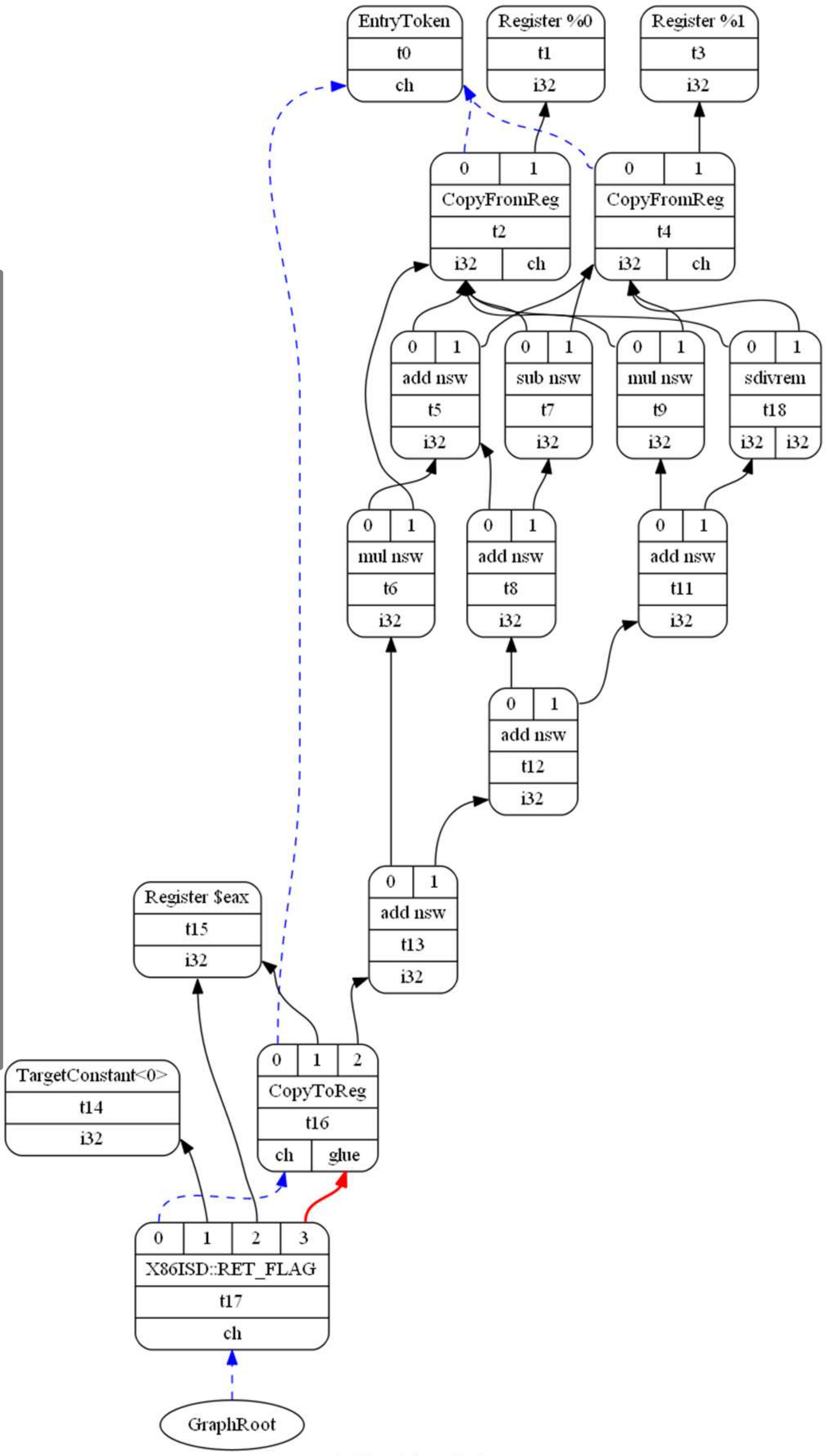
- `-view-dag-combine1-dags` displays the DAG after being built, before the first optimization pass.
- `-view-legalize-dags` displays the DAG before Legalization.
- `-view-dag-combine2-dags` displays the DAG before the second optimization pass.
- `-view-isel-dags` displays the DAG before the Select phase.
- `-view-sched-dags` displays the DAG before Scheduling.

https://llvm.org/docs/CodeGenerator.html

查看DAG

```
define dso_local i32 @sched(i32 %0,
i32 %1) #0 {
  %3 = add nsw i32 %0, %1
  %4 = mul nsw i32 %3, %0
  ~~%5 = add nsw i32 %0, %1~~
  %6 = sub nsw i32 %0, %1
  %7 = add nsw i32 %5, %6
  %8 = mul nsw i32 %0, %1
  %9 = sdiv i32 %0, %1
  %10 = add nsw i32 %8, %9
  %11 = add nsw i32 %7, %10
  %12 = add nsw i32 %4, %11
  ret i32 %12
}
```

```
#:llc -view-isel-dags ssa.ll
```

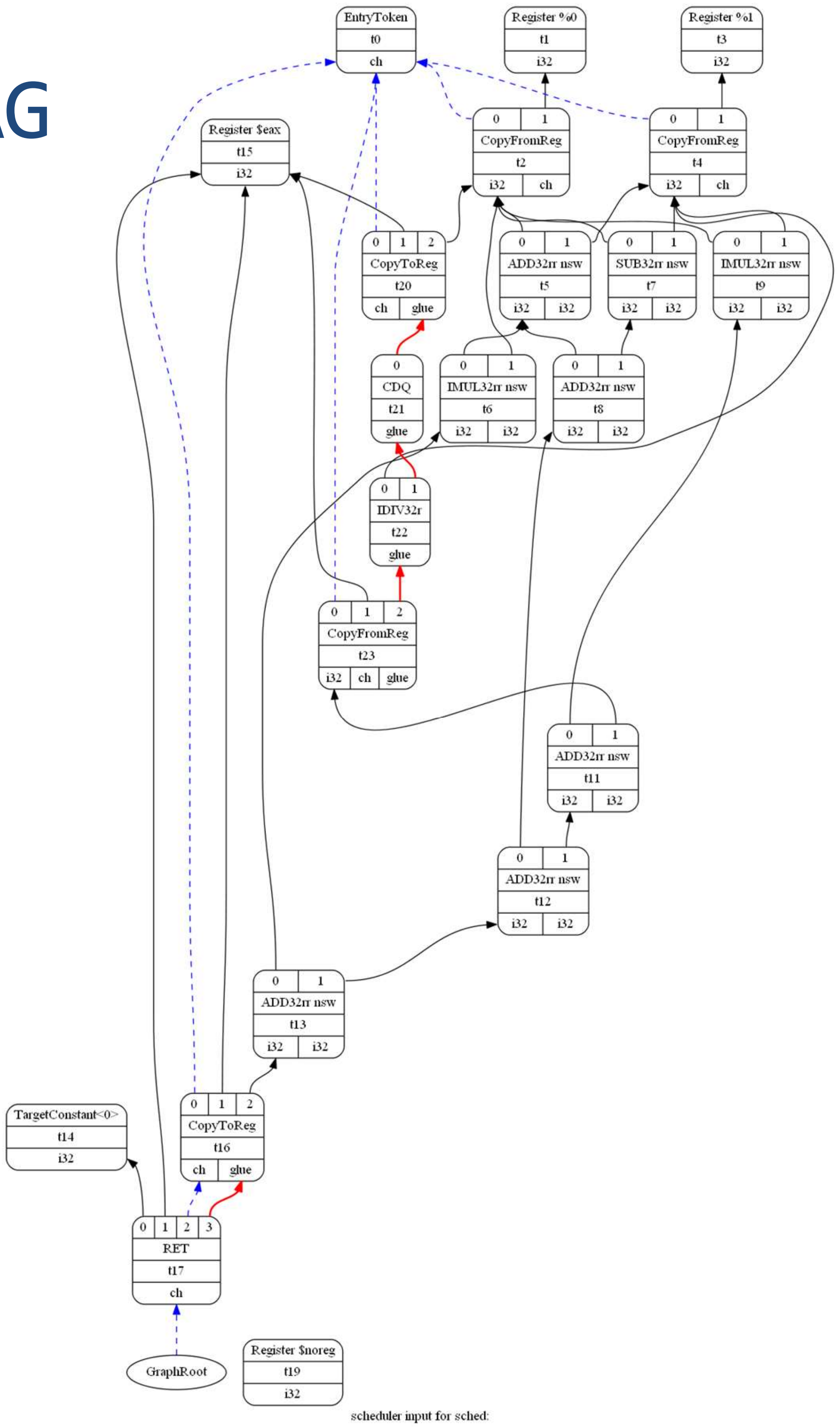


isel input for sched:

查看Schedule后的DAG

```
define dso_local i32 @sched(i32 %0,
i32 %1) #0 {
  %3 = add nsw i32 %0, %1
  %4 = mul nsw i32 %3, %0
  ~~%5 = add nsw i32 %0, %1~~
  %6 = sub nsw i32 %0, %1
  %7 = add nsw i32 %5, %6
  %8 = mul nsw i32 %0, %1
  %9 = sdiv i32 %0, %1
  %10 = add nsw i32 %8, %9
  %11 = add nsw i32 %7, %10
  %12 = add nsw i32 %4, %11
  ret i32 %12
}
```

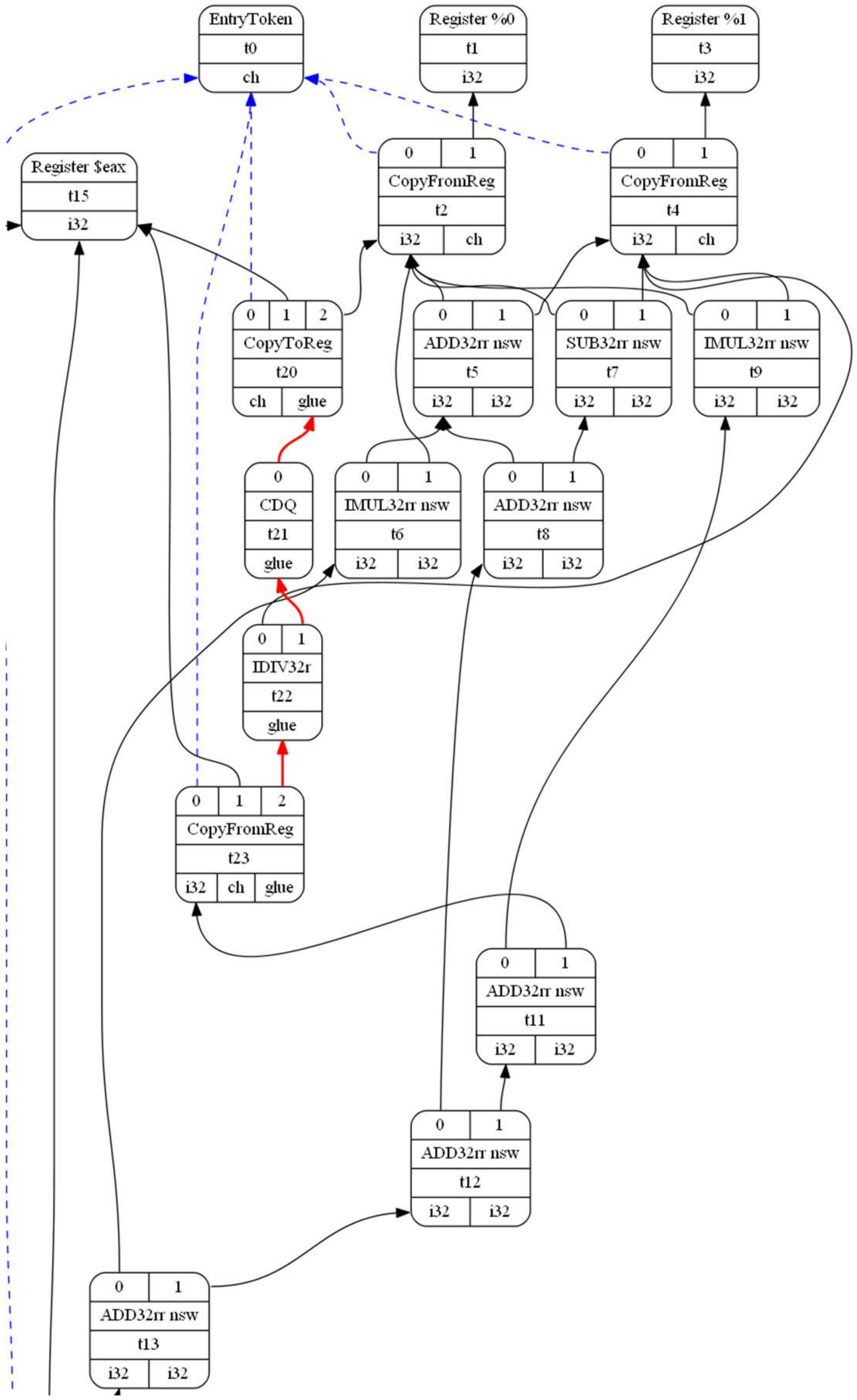
```
#:llc -view-sched-dags ssa.ll
```



scheduler input for sched:

生成汇编代码

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, %eax
leal    (%rax,%rsi), %ecx
subl    %esi, %edi
addl    %ecx, %edi
imull   %eax, %ecx
movl    %eax, %r8d
imull   %esi, %r8d
cltd
idivl   %esi
addl    %r8d, %eax
addl    %edi, %eax
addl    %ecx, %eax
popq    %rbp
retq
```

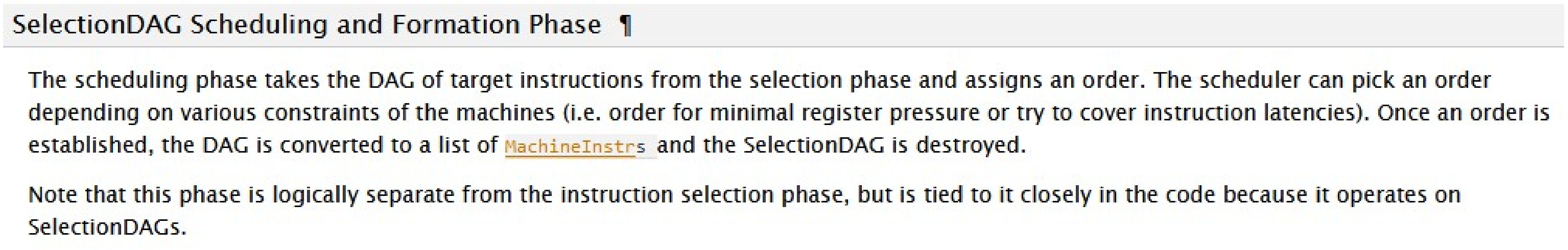


Schedule前后对比

```
define dso_local i32 @sched(i32
%0, i32 %1) #0 {
  %3 = add nsw i32 %0, %1
  %4 = mul nsw i32 %3, %0
  %6 = sub nsw i32 %0, %1
  %7 = add nsw i32 %3, %6
  %8 = mul nsw i32 %0, %1
  %9 = sdiv i32 %0, %1
  %10 = add nsw i32 %8, %9
  %11 = add nsw i32 %7, %10
  %12 = add nsw i32 %4, %11
  ret i32 %12
}
```

```
define dso_local i32 @sched(i32
%0, i32 %1) #0 {
  %3 = add nsw i32 %0, %1
  %6 = sub nsw i32 %0, %1
  %7 = add nsw i32 %3, %6
  %4 = mul nsw i32 %3, %0
  %8 = mul nsw i32 %0, %1
  %9 = sdiv i32 %0, %1
  %10 = add nsw i32 %8, %9
  %11 = add nsw i32 %7, %10
  %12 = add nsw i32 %4, %11
  ret i32 %12
}
```

简少了寄存器使用？消除了%6和%4的conflict



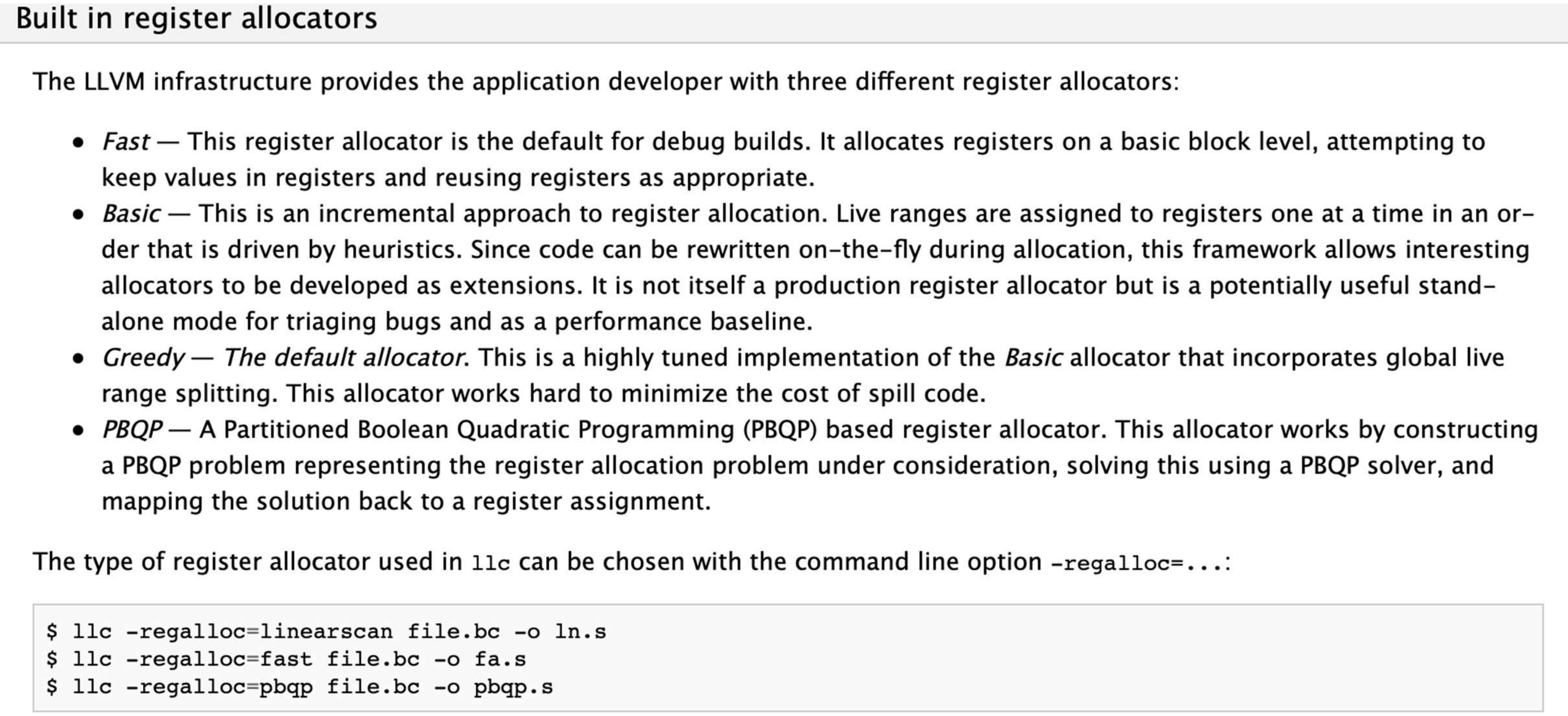
SelectionDAG Scheduling and Formation Phase ¶

The scheduling phase takes the DAG of target instructions from the selection phase and assigns an order. The scheduler can pick an order depending on various constraints of the machines (i.e. order for minimal register pressure or try to cover instruction latencies). Once an order is established, the DAG is converted to a list of `MachineInstrs` and the SelectionDAG is destroyed.

Note that this phase is logically separate from the instruction selection phase, but is tied to it closely in the code because it operates on SelectionDAGs.

https://llvm.org/docs/CodeGenerator.html

寄存器分配选项



Built in register allocators

The LLVM infrastructure provides the application developer with three different register allocators:

- *Fast* — This register allocator is the default for debug builds. It allocates registers on a basic block level, attempting to keep values in registers and reusing registers as appropriate.
- *Basic* — This is an incremental approach to register allocation. Live ranges are assigned to registers one at a time in an order that is driven by heuristics. Since code can be rewritten on-the-fly during allocation, this framework allows interesting allocators to be developed as extensions. It is not itself a production register allocator but is a potentially useful stand-alone mode for triaging bugs and as a performance baseline.
- *Greedy* — *The default allocator*. This is a highly tuned implementation of the *Basic* allocator that incorporates global live range splitting. This allocator works hard to minimize the cost of spill code.
- *PBQP* — A Partitioned Boolean Quadratic Programming (PBQP) based register allocator. This allocator works by constructing a PBQP problem representing the register allocation problem under consideration, solving this using a PBQP solver, and mapping the solution back to a register assignment.

The type of register allocator used in `llc` can be chosen with the command line option `-regalloc=...`:

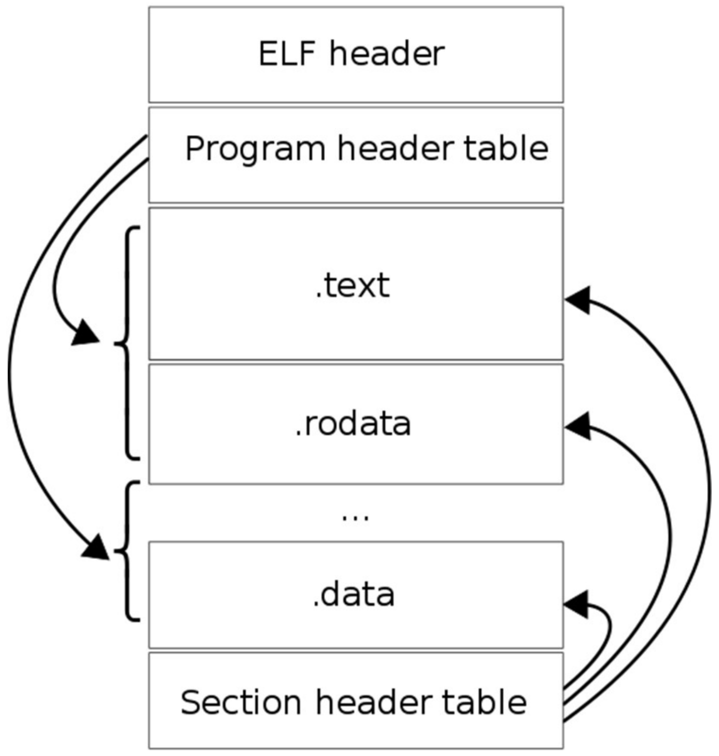
```
$ llc -regalloc=linearscan file.bc -o ln.s
$ llc -regalloc=fast file.bc -o fa.s
$ llc -regalloc=pbqp file.bc -o pbqp.s
```

六、组装可执行程序

常见的二进制程序类型

- ELF（Executable and Linkable Format）
  - Linux、Android等
  - ELF文件类型：
    - 可执行文件（Executable File）
    - 可重定位文件（Relocatable File）
      - 后缀：.o
      - ret.text、rel.data
    - 共享目标文件（Shared Object File）
      - 后缀：.so
      - dynsym，dynstr
- PE（Portable Executable）
  - Windows
- Mach-O:
  - MacOS、iOS

ELF文件组成和启动



- **Program Header**：提供运行时所需的段（segment）信息；
- **Section Header**：程序的实际内容
- 可通过hexdump、readelf、 objdump、IDA pro、GDB等工具查看：
- **ELF文件启动**：
  1) 运行系统调用execve()，内核态执行load_elf_binary()函数；
  2) 解析ELF Header和Program Header；
  3) 根据段信息将文件映射到内存；
  4) 从程序入口开始运行。

ELF文件头

```
#: readelf a.out -h
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401040
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14600 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         11
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

Section Headers

```
#: readelf -S a.out
There are 31 section headers, starting at offset 0x1a30:
Section Headers:
  [Nr] Name              Type         Address    Offset Size   EntSize  Flags   Link   Info   Align
  [ 0]                   NULL         ..00000000  0000 0000   0000              0      0      0
  [ 1] .interp           PROGBITS     ..004002a8  02a8 001c   0000     A        0      0      1
  [ 2] .note.gnu.build-i NOTE         ..004002c4  02c4 0024   0000     A        0      0      4
  [ 3] .note.ABI-tag     NOTE         ..004002e8  02e8 0020   0000     A        0      0      4
  [ 4] .gnu.hash         GNU_HASH     ..00400308  0308 001c   0000     A        5      0      8
  [ 5] .dynsym           DYNSYM       ..00400328  0328 0060   0018     A        6      1      8
  [ 6] .dynstr           STRTAB       ..00400388  0388 003f   0000     A        0      0      1
  [ 7] .gnu.version      VERSYM       ..004003c8  03c8 0008   0002     A        5      0      2
  [ 8] .gnu.version_r    VERNEED      ..004003d0  03d0 0020   0000     A        6      1      8
  [ 9] .rela.dyn         RELA         ..004003f0  03f0 0030   0018     A        5      0      8
  [10] .rela.plt         RELA         ..00400420  0420 0018   0018     AI       5     22      8
  [11] .init             PROGBITS     ..00401000  1000 001b   0000     AX       0      0      4
  [12] .plt              PROGBITS     ..00401020  1020 0020   0010     AX       0      0     16
  [13] .text             PROGBITS     ..00401040  1040 01c5   0000     AX       0      0     16
  [14] .fini             PROGBITS     ..00401208  1208 000d   0000     AX       0      0      4
  [15] .rodata           PROGBITS     ..00402000  2000 0010   0000     A        0      0      4
  [16] .eh_frame_hdr     PROGBITS     ..00402010  2010 003c   0000     A        0      0      4
  [17] .eh_frame         PROGBITS     ..00402050  2050 00e8   0000     A        0      0      8
  [18] .init_array       INIT_ARRAY   ..00403e10  2e10 0008   0008     WA       0      0      8
  [19] .fini_array       FINI_ARRAY   ..00403e18  2e18 0008   0008     WA       0      0      8
  [20] .dynamic          DYNAMIC      ..00403e20  2e20 01d0   0010     WA       6      0      8
  [21] .got              PROGBITS     ..00403ff0  2ff0 0010   0008     WA       0      0      8
  [22] .got.plt          PROGBITS     ..00404000  3000 0020   0008     WA       0      0      8
  [23] .data             PROGBITS     ..00404020  3020 0010   0000     WA       0      0      8
  [24] .bss              NOBITS       ..00404030  3030 0008   0000     WA       0      0      1
  [25] .comment          PROGBITS     ..00000000  3030 0049   0001     MS       0      0      1
  [26] .symtab           SYMTAB       ..00000000  3080 05b8   0018             27     43      8
  [27] .strtab           STRTAB       ..00000000  3638 01ca   0000              0      0      1
  [28] .shstrtab         STRTAB       ..00000000  3802 0103   0000
```

动态链接符号

函数链接代码

代码段

只读数据

异常处理信息

全局偏移地址
（动态链接）

全局变量（初始化）

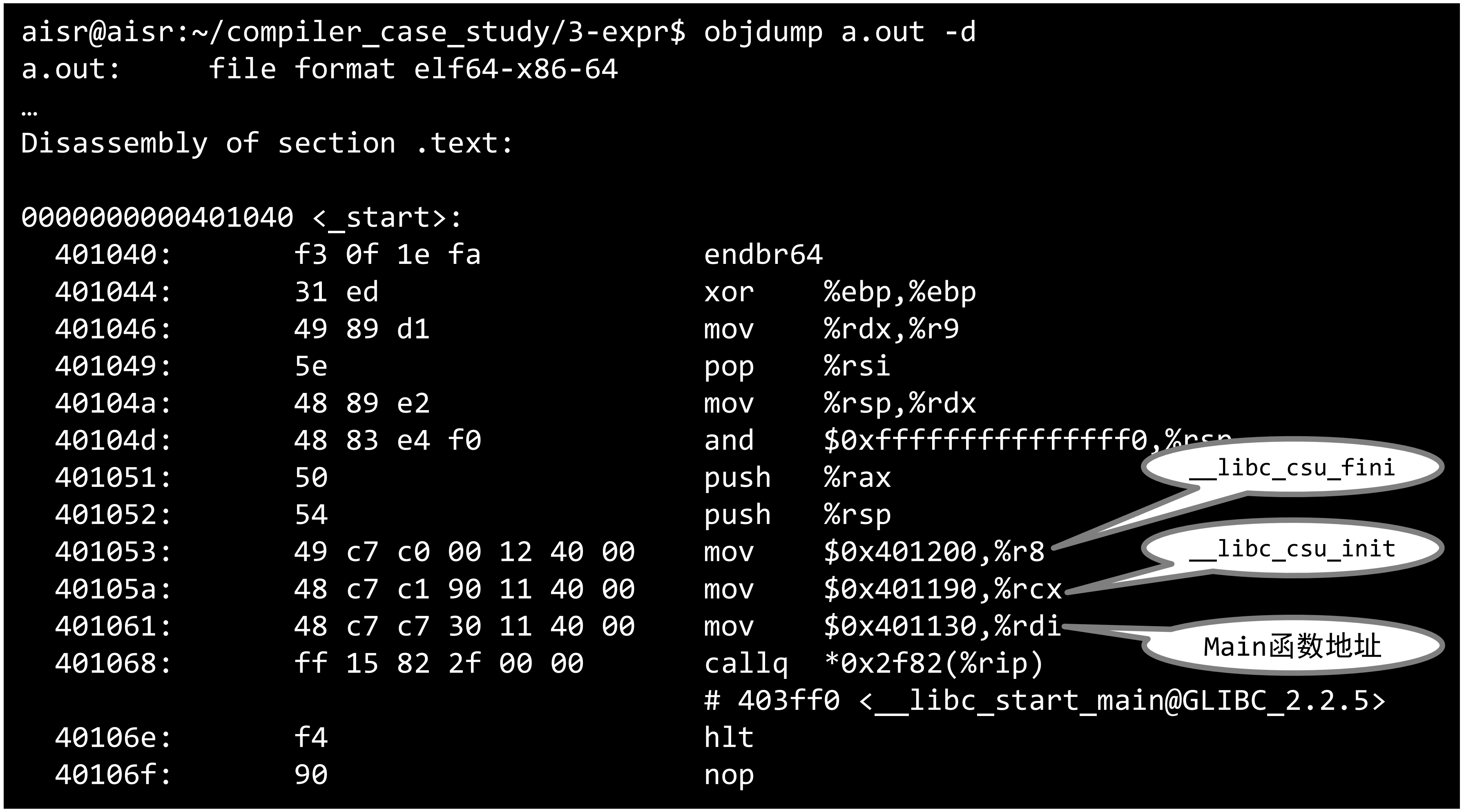
全局变量（未初始化）

Program Headers：运行视图

```
#: readelf -l a.out
Elf file type is EXEC (Executable file)
Entry point 0x401040
There are 11 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr     PhysAddr     FileSiz     MemSiz    Flags  Align
  PHDR           0040     0x..00400040 0x..00400040 0268        0268      R      0x8
  INTERP         02a8     0x..004002a8 0x..004002a8 001c        001c      R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0000     0x..00400000 0x..00400000 0438        0438      R      0x1000
  LOAD           1000     0x..00401000 0x..00401000 0215        0215      R E    0x1000
  LOAD           2000     0x..00402000 0x..00402000 0138        0138      R      0x1000
  LOAD           2e10     0x..00403e10 0x..00403e10 0220        0228      RW     0x1000
  DYNAMIC        2e20     0x..00403e20 0x..00403e20 01d0        01d0      RW     0x8
  NOTE           02c4     0x..004002c4 0x..004002c4 0044        0044      R      0x4
  GNU_EH_FRAME   2010     0x..00402010 0x..00402010 003c        003c      R      0x4
  GNU_STACK      0000     0x..00000000 0x..00000000 0000        0000      RW     0x10
  GNU_RELRO      2e10     0x..00403e10 0x..00403e10 01f0        01f0      R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01      .interp
   02      .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version
           .gnu.version_r .rela.dyn .rela.plt
   03      .init .plt .text .fini
   04      .rodata .eh_frame_hdr .eh_frame
   05      .init_array .fini_array .dynamic .got .got.plt .data .bss
   06      .dynamic
   07      .note.gnu.build-id .note.ABI-tag
   08      .eh_frame_hdr
   09
   10      .init_array .fini_array .dynamic .got
```

程序入口

- _start()函数
- 调用glibc的__libc_start_main()函数



```
aisr@aisr:~/compiler_case_study/3-expr$ objdump a.out -d
a.out:     file format elf64-x86-64
…
Disassembly of section .text:

0000000000401040 <_start>:
  401040:       f3 0f 1e fa             endbr64
  401044:       31 ed                   xor    %ebp,%ebp
  401046:       49 89 d1                mov    %rdx,%r9
  401049:       5e                      pop    %rsi
  40104a:       48 89 e2                mov    %rsp,%rdx
  40104d:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
  401051:       50                      push   %rax
  401052:       54                      push   %rsp
  401053:       49 c7 c0 00 12 40 00    mov    $0x401200,%r8
  40105a:       48 c7 c1 90 11 40 00    mov    $0x401190,%rcx
  401061:       48 c7 c7 30 11 40 00    mov    $0x401130,%rdi
  401068:       ff 15 82 2f 00 00       callq  *0x2f82(%rip)
                                        # 403ff0 <__libc_start_main@GLIBC_2.2.5>
  40106e:       f4                      hlt
  40106f:       90                      nop
```

程序启动

```
.text:
401068:        ff 15 82 2f 00 00        callq  *0x2f82(%rip)
                                        # 403ff0 <__libc_start_main@GLIBC_2.2.5>
```

got的地址

```
(gdb) start
Temporary breakpoint 2 at 0x401114
Starting program: /home/aisr/compiler_case_study/3-expr/a.out

Temporary breakpoint 2, 0x..00401114 in main ()
(gdb) x/32wx 0x403ff0
0x403ff0:       0xf7debfc0      0x00007fff      0x00000000      0x00000000
```

静态链接

- 外部函数实现一般编译为字节码（文件后缀.o）、或静态库（文件后缀.a）
- 将函数实现全部集成在一个可执行文件中
  - 优点：运行速度快
  - 缺点：文件较大

```
#clang -static hello.c
```

动态链接

- 外部函数实现一般编译为或动态库（文件后缀.so）
- 可执行程序中不包含该函数的实现，仅保留其函数指纹。
- 程序加载或运行时通过plt和got预完成函数寻址和调用。
  - 加载时寻址：影响启动性能；
  - 运行时寻址：延迟绑定技术

```
#clang hello.c
```

参考资料

- 《编译原理（第2版）》
  - 第11章：指令选择
  - 第12章：指令调度
  - 第13章：寄存器分配
- https://llvm.org/docs/CodeGenerator.html
- G.J. Chaitin, *et al*, Register allocation via coloring. *Computer languages*, 1981.
- O. Traub, *et al*. Quality and speed in linear-scan register allocation. *ACM SIGPLAN Notices, 1998*.
- M. Poletto and V. Sarkar, Linear scan register allocation. *TOPLAS*, 1999
- Blindell, "Survey on instruction selection: An extensive and modern literature review." arXiv preprint arXiv:1306.4898 (2013).
- Agner Fog, Software optimization resources-4-Instruction table, https://www.agner.org/optimize/instruction_tables.pdf