MiniC 编译器实习项目

彭焯　王衍　华连盛　李春奇

PKU

code.google.com/p/minic

摘要 I

摘要 II

5. 编译优化
   - 数据流分析
   - 中间代码上的窥孔优化
   - 目标代码上的窥孔优化
   - 尾递归优化
   - 指令调度

摘要 I

摘要 II

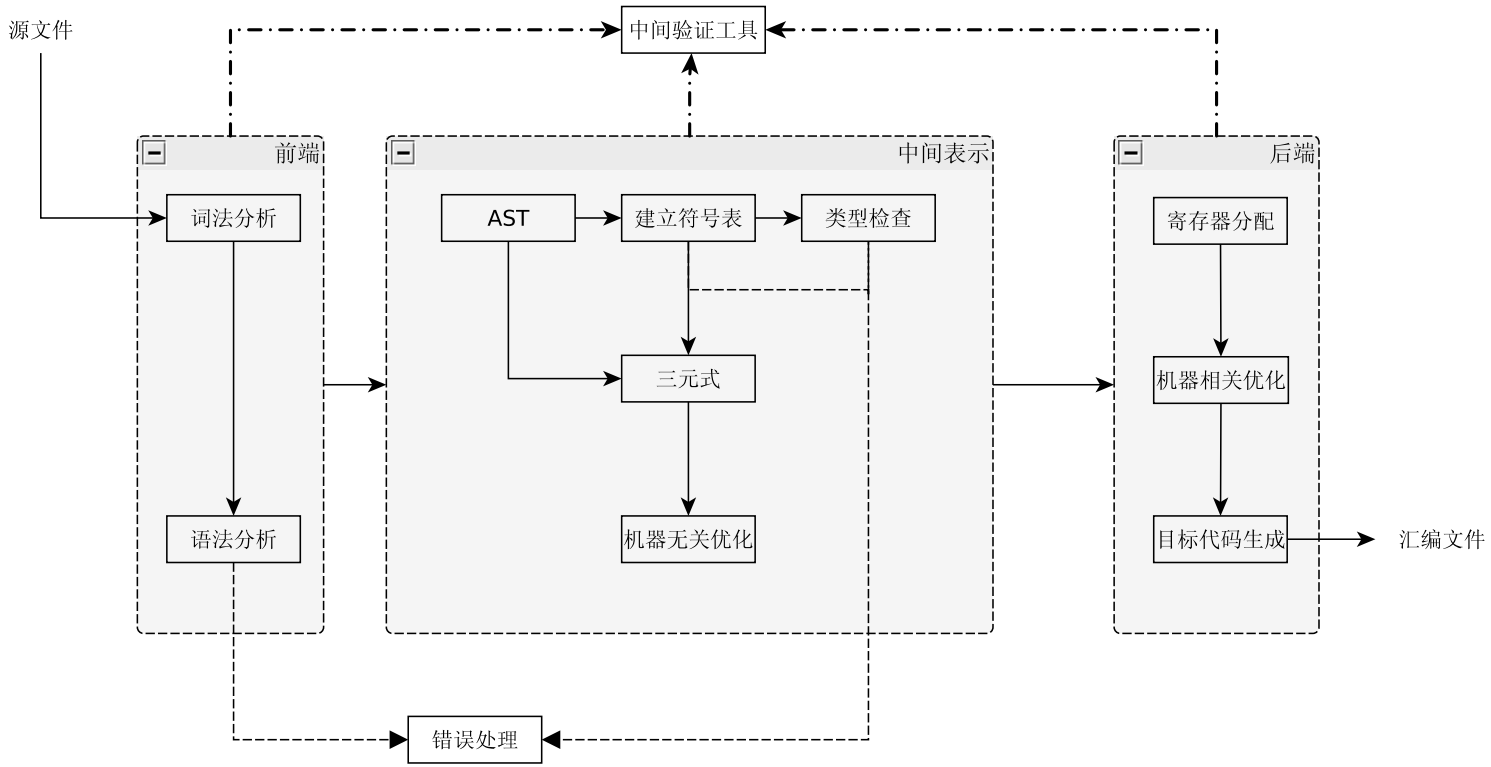
项目需求

实现一个 C 语言子集(MiniC)的编译器，要求：

- 目标机器为 Unicore32
- 输入 MiniC 源代码，生成 Unicore32 汇编代码。链接交由 Unicore32 二进制工具链完成
- 链接后的程序能在 Unicore32 实体机器上运行，且能够在同时设计的 Unicore32 模拟器上运行[1]
- 实现包括指令调度在内的编译优化
- 通过对模拟器相关参数的修改，考察编译-ISA 对性能的综合影响

[1]体系实习项目

项目框架


源文件
中间验证工具
前端
词法分析
语法分析
中间表示
AST
建立符号表
类型检查
三元式
机器无关优化
后端
寄存器分配
机器相关优化
目标代码生成
汇编文件
错误处理

摘要 I

摘要 II

前端

- 词法分析使用 flex 辅助完成
- 语法分析使用 bison 辅助完成，生成 AST
- 简单的预处理器，支持注释
- 简单的错误提示功能（见下图）

错误提示

```
1 int main()
2 {
3     int i,j;
4     i = i+j
5 }
```

```
>>> ../minic error_example.c
syntax error
Parsing error: line 4.0-5.1: bad lvalue
 Parsing error: line 4.0-5.1: bad lvalue
 Parser: terminated, 2 error(s).
```

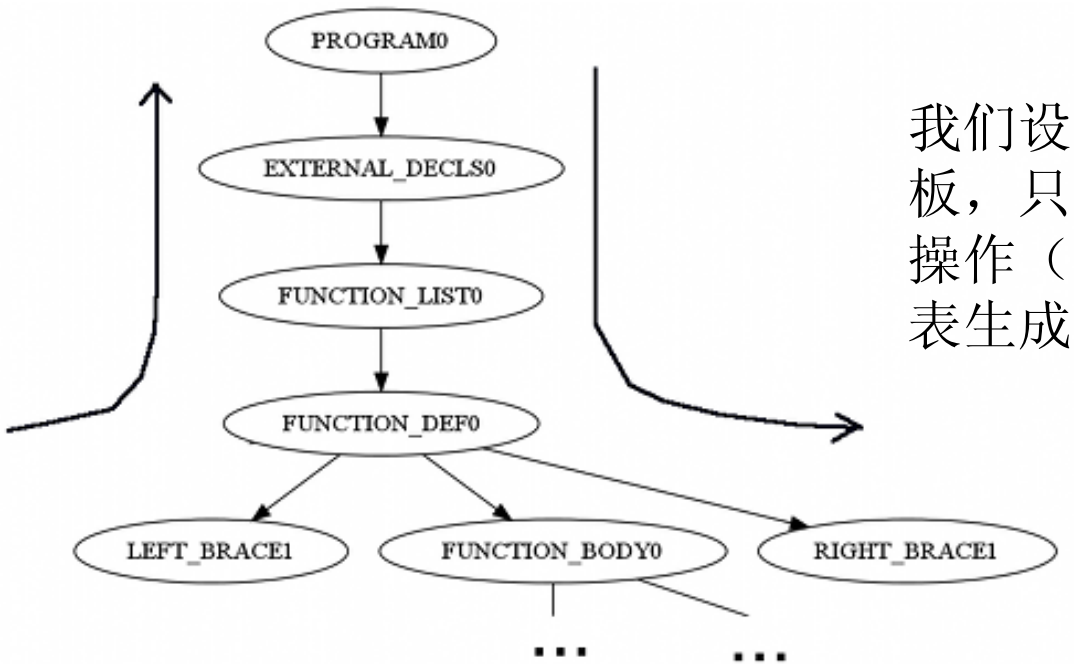
摘要 I

摘要 II

5 编译优化
- 数据流分析
- 中间代码上的窥孔优化
- 目标代码上的窥孔优化
- 尾递归优化
- 指令调度

抽象语法树 AST

AST 是 MiniC 的第一重中间表示，符号表生成和类型检查检查在 AST 上进行。

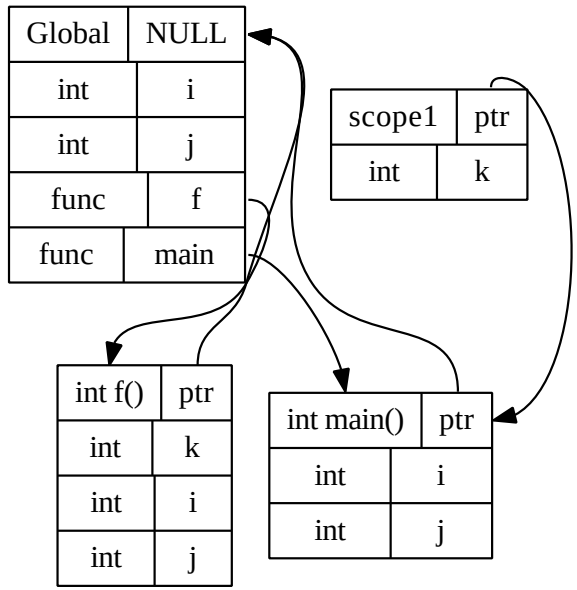


我们设计了通用的 AST 遍历模板，只需要在其上填入所需的操作（函数）就能够完成符号表生成/类型检查

符号表

为了支持支持复合语句中的声明，符号表采用树状组织，具体实现采用可扩张线性表。

```
1 int i,j;
2 void f(int k)
3 {
4     int i,j;
5 }
6 int main()
7 {
8     int i,j;
9     {int k;}
10 }
```



类型检查

类型检查负责报告：

- 使用未声明的变量；调用未声明的函数
- 函数调用传参个数不符
- 函数调用传参类型不匹配（警告或错误）
- 函数返回值同表达式中算符/操作数类型不匹配(警告或错误)
- 非法的指针运算，如相加、相乘和不同类型指针相减
- 不同存储类型的变量运算，如`char`和`int`相加（警告）

三元式

三元式是 MiniC 的第二重中间表示，它由 AST 经过变换而来，在表达能力上同源语言等价。机器无关的优化、目标代码生成均以它为基础进行，下表是三元式指令及功能。

除了运算指令和条件跳转、无条件跳转指令外，三元式中有如下的特殊指令：

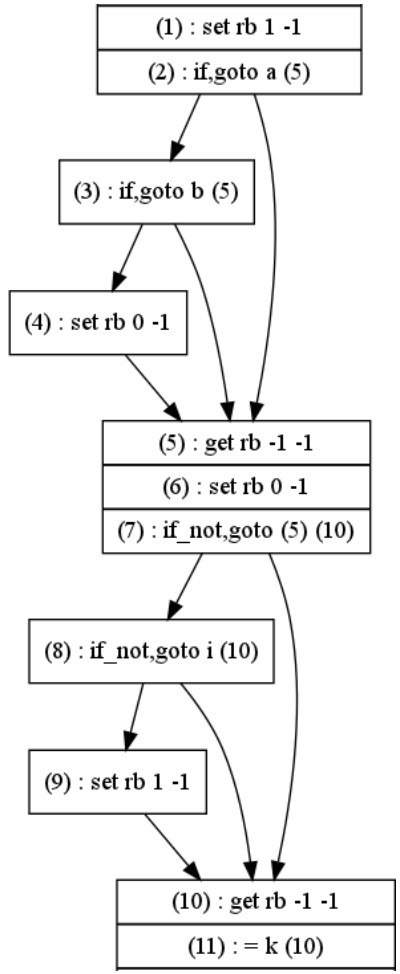
指令类型	举例	作用
比较指令	less	关系成立值为 1，否则 0
rb 操作指令	setrb, getrb	设置或读取 rb 的值
函数调用指令	param, call	传参，函数调用
常量表示指令	Imm, c_str	表示一个常量或字符串

三元式（续）：布尔表达式的翻译

下面主要介绍布尔表达式的翻译：

由于三元式中同一个临时变量不能有两次赋值，我们假设有一个 rb 寄存器，并且用 setrb 和 getrb 来实现给同一临时变量赋值。

右图是 k=(a||b)&&i 的基本块。在目标代码生成时，我们没有采取直接翻译的方式处理 setrb 和 getrb.



摘要 I

摘要 II

5 编译优化
- 数据流分析
- 中间代码上的窥孔优化
- 目标代码上的窥孔优化
- 尾递归优化
- 指令调度

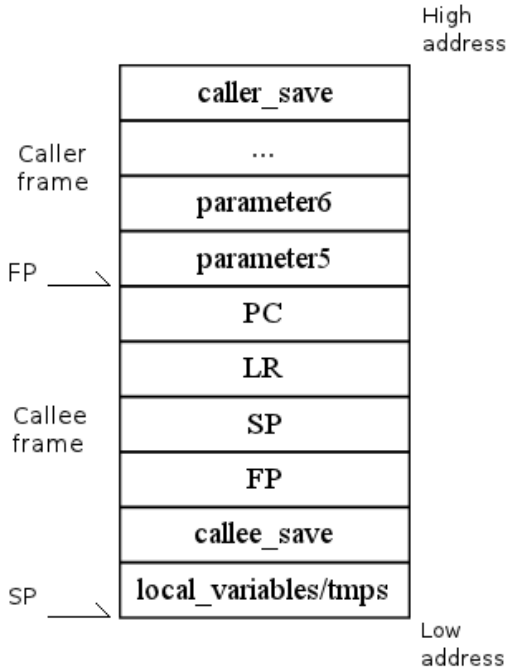
目标机器寄存器使用规范

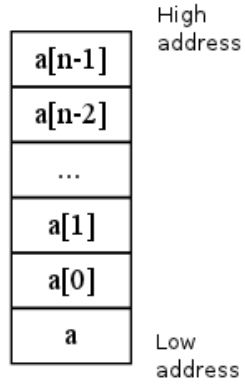
寄存器	Unicore32	MiniC
r0-r3	传递参数和返回值	传递参数和返回值；在函数内用于无寄存器变量的装入和运算
r4-r15	caller save	caller save*
r17-r25	callee save	callee save*
r26	静态基址	用于读取大立即数地址
r27	栈帧基址	栈帧基址
r28	调用者 SP	传参时用于装入和运算；生成立即数
r29	栈基址	栈基址
r30	返回地址	返回地址
r31	PC	PC

* 数目可调

栈帧及全局区

- 栈帧布局和数组存放规则见右图
- 无法使用立即数寻址的大立即数保存在代码段，使用 PC 相对寻址得到
- 全局变量、字符串常量保存在全局区，并将其指针放在代码段

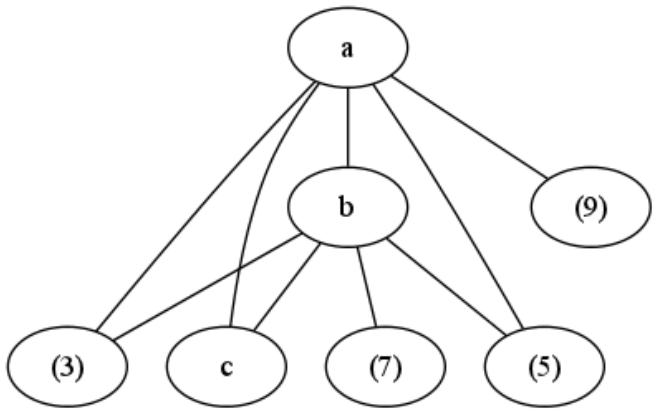




寄存器分配

- 依赖于活跃变量分析
- 采用图染色法
- 小的删点策略改进，见右例

(1) : = a 1
(2) : = b 1 {a}
(3) : + b 1 {a,b}
(4) : = c (3) {a,b,(3)}
(5) : + c 1 {a,b,c}
(6) : = d (5) {a,b,(5)}
(7) : + a 1 {a,b}
(8) : = a (7) {b,(7)}
(9) : + b 1 {b}
(10) : = b (9){(9)}



寄存器保护-推栈策略

- 局部变量
- 临时变量
- 全局变量

setrb/getrb 的处理

```
1 if(!(a&&(b||c)))
2     if(a||(b||c))
3         a = 1;
4     else
5         a = 2;
6 else
7     a = 3;
```

```
1          cmpsub.a    r7, #0
2          beq .L4
3 .L2:     cmpsub.a    r6, #0
4          bne .L5
5 .L3:     cmpsub.a    r5, #0
6          bne .L5
7 .L4:     cmpsub.a    r7, #0
8          bne .L8
9 .L6:     cmpsub.a    r6, #0
10         bne .L8
11 .L7:    cmpsub.a    r5, #0
12         beq .L9
13 .L8:    mov r7, #1
14         b   .L10
15 .L9:    mov r7, #2
16 .L10:   b   .L11
17 .L5:    mov r7, #3
18 .L11:
```

摘要 I

摘要 II

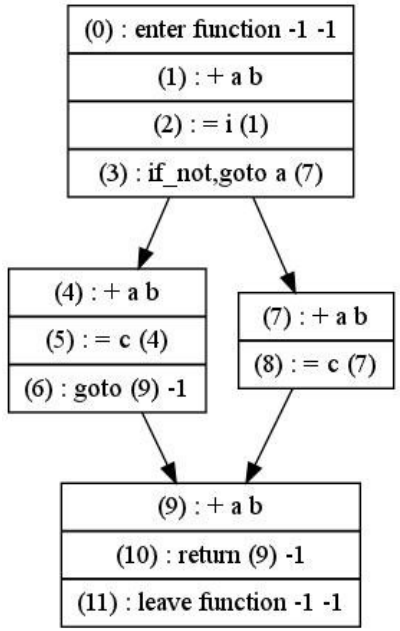
5 编译优化
- 数据流分析
- 中间代码上的窥孔优化
- 目标代码上的窥孔优化
- 尾递归优化
- 指令调度

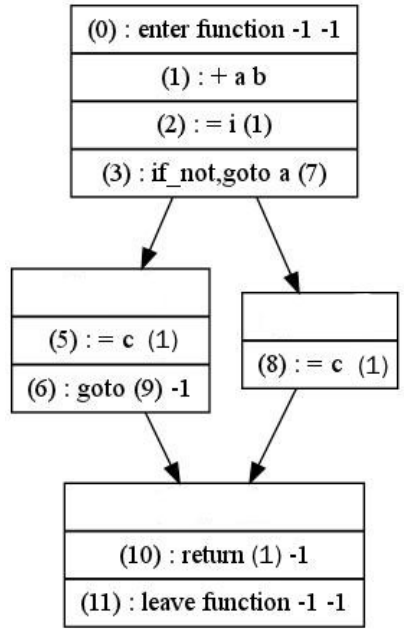
数据流分析

- 在三元式上进行数据流分析
- 设计了适用于迭代算法的通用的数据结构
- 进行了如下分析：
  - 活跃变量分析
  - 可用表达式传播
  - 指针分析

可用表达式传播

- 依赖于可用表达式分析
- 如果某三元式计算的表达式可用且所有可用表达式交汇于一点，则用该点替换下面所有用到该三元式标号的三元式，并删除该表达式。
- 迭代进行，直到没有三元式被删除
- 类似于，但性能劣于复写传播





指针分析

- 用于解决指针操作时寄存器-栈帧同步的问题
- 算法最终能得到在每个程序点，每个指针的指向状态。可以根据该状态比较精确地进行同步
- 基于数据流分析：
  - 对象：序对 (p, v)，代表指针 p 指向 v
  - $GEN(B)$ 基本块 B 中产生的序对集合
  - $KILL(B)$ 基本块 B 中注销的序对集合
  - 迭代方向：正向
  - 迭代方程：$OUT(N) = GEN(N) + (IN(N) - KILL(N))$
    $IN(N) = \bigcup_{P \in predecessor(N)} OUT(P)$

指针分析（续）——性能

对于右图的例子，下面三种方法在第 9 行、第 12 行、第 14 行需要的同步次数如下：

- 每次同步所有变量：3 次；3 次；3 次
- 扫描一次记录每个指针指过的所有变量：2 次；3 次；3 次
- 利用指针分析的结果：2 次；3 次；1 次

```
1  int f()
2  {
3      int a,b,c,*p,*q;
4      ...
5      if(some_condition)
6          p = &a;
7      else
8          p = &b;
9      *p = 1;
10     while(some_condition){
11         q = p;
12         *q = 1;
13         q = &c;
14         *q = 2;
15     }
16     ...
17
18 }
```

中间代码上的窥孔优化

- 常量表达式计算：计算出两个操作数都是立即数的三元式的值，如果用到该三元式值的三元式又能够直接计算，那么迭代地进行这个过程，直到结果是编译时不可计算的为止。（不考虑对于带有常数的布尔运算。）
- 简单的强度削减：乘以 2 的幂次改为左移（乘法 2 cycle 而左移 1 cycle）

(1) : + 1 2
(2) : = a (1)
(3) : ! (1)
(4) < (1) 15
(5) if (4) goto (_)

(2) : = a 3
(3) : ! (1)
(5) if 1 goto (_)

合并地址计算和访存语句

- 目标代码由三元式翻译而来，不能直接利用 Unicore32 的寄存器位移读取/写入指令
- 发现计算地址和访存指令，将它们合并

下例是a[i] = 1的三元式、优化前汇编码和优化后汇编码

(1) : add_shift a i
(2) : *= (1) 1

```
add r4, r5, r4<<#2
mov r3, #1
stw r3, [r4+], #0
```

```
mov r3, #1
stw r3, [r5+], r4<<#2
```

删除冗余的 mov

- 由于三元式直接翻译到 Unicore32 汇编不能很好的利用三个操作数的汇编指令，会产生出很多冗余的mov
- 我们做了窥孔优化来找到符合以下条件的mov，删除它，并修改其后用到它的目标寄存器的语句的源寄存器：
  - 从mov指令的源寄存器最近的一次定值开始，到mov指令之间没有对mov的目标寄存器引用。
  - 从mov指令开始，将mov指令的源寄存器替换为目标寄存器时不会产生冲突。

```
ldw r4, [r5+], #0
mov r6, r4
add r5, r4, 1
ldw r4, [r27+], #0
```

```
ldw r6, [r5+], #0
add r5, r6, 1
ldw r4, [r27+], #0
```

尾递归优化

尾递归优化能将符合如下条件的函数调用转换为迭代（即免去保存和回复现场、建立栈帧等操作）：

1. 递归调用
2. 该调用语句是本函数除返回语句以外的最后一条语句
3. 该函数没有返回值

尾递归优化-续：一个例子

```c
void qsort(int* data, int begin, int end)
{
    int i,j,tmp;
    if(end <= begin + 1)
        return;
    /* partition routine */
    ...
    qsort(data, begin, j - 1);
    qsort(data, j, end);
    return;
}
```

```
qsort:
    @build stack frame
    @transmit parameters
.L1:
    add r4, r7, #1
    cmpsub.a r6, r4
    bsg .L3
.L2:
    @return
    b .L9
.L3:
    @@partition routine
    ...
    sub r9, r5, #1
    @protect caller save registers
    ...
    @call qsort(data, begin, j-1)
    mov r0, r8
    mov r1, r7
    mov r2, r9
    b.l qsort
    @resume caller save registers
    ...
    @protect caller save registers
    ...
    @call qsort(data, j, end)
    mov r0, r8
    mov r1, r5
    mov r2, r6
    b.l qsort
    @resume caller save regusters
    ...
.L9:
    @exit point
```

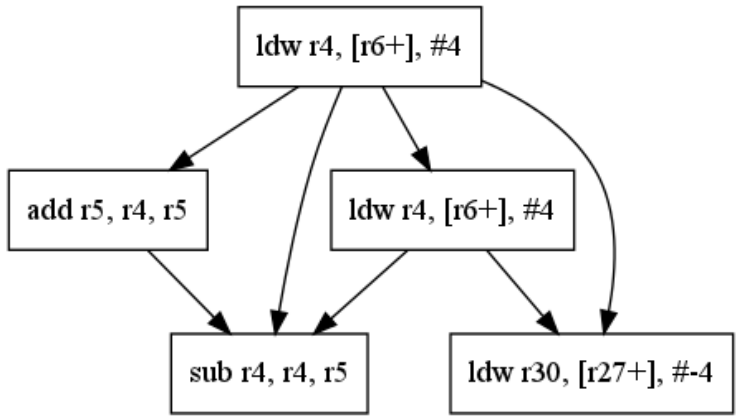
```
1  qsort:
2      @build stack frame
3      @transmit parameters
4  .L1:
5      add r4, r7, #1
6      cmpsub.a r6, r4
7      bsg .L3
8  .L2:
9      @return
10     b .L9
11 .L3:
12     @@partition routine
13     ...
14     sub r9, r5, #1
15     @protect caller save registers
16     ...
17     @call qsort(data, begin, j-1)
18     mov r0, r8
19     mov r1, r7
20     mov r2, r9
21     b.l qsort
22     @resume caller save registers
23     ...
24     mov r7, r5
25     b .L1
       ----------
       ----------
       ----------
       ----------
       ----------
       ----------
26     ...
27 .L9:
28     @exit point
```

指令调度

根据 Unicore32 指令系统体系结构，只有载入指令和运算指令可能产生的数据相关无法转发，需要等待一个 cycle。指令调度的目的就是尽量让这种数据相关不发生。

算法作用范围是目标代码上的基本块，基于一个“数据依赖图”：若指令i的源操作数依赖于指令j的执行结果，则建立j到i的一条边；算法的思想是先按顺序对ldw执行一种带优先级的拓扑排序：遇到ldw则安排执行所有它依赖的指令（包括反依赖关系），然后从未安排执行过的指令中，取一条入度为 0 的指令，要求它尽量不依赖于ldw的结果，排在ldw之后；最后对还未被安排的非ldw指令再进行一次拓扑排序即可。

```
ldw r4, [r6+], #4
add r5, r4, r5
ldw r4, [r6+], #4
sub r4, r4, r5
ldw r30, [r27+], #-4
```




```
ldw r4, [r6+], #4
add r5, r4, r5
ldw r4, [r6+], #4
ldw r30, [r27+], #-4
sub r4, r4, r5
```

演示与 Q&A