

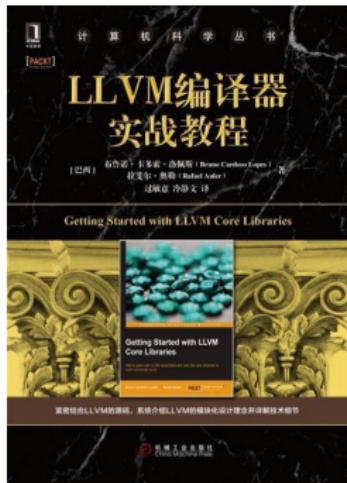
## 五、目标代码生成 (16. 指令选择)

魏恒峰

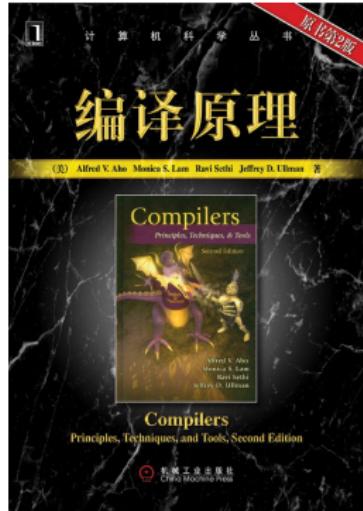
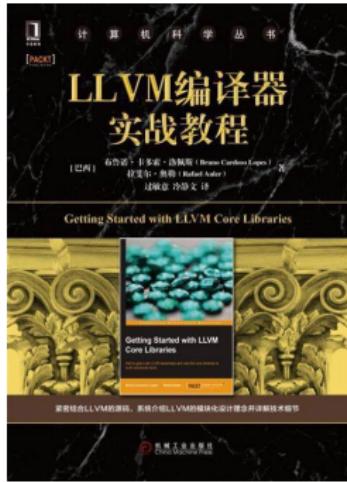
hfwei@nju.edu.cn

2024 年 06 月 12 日



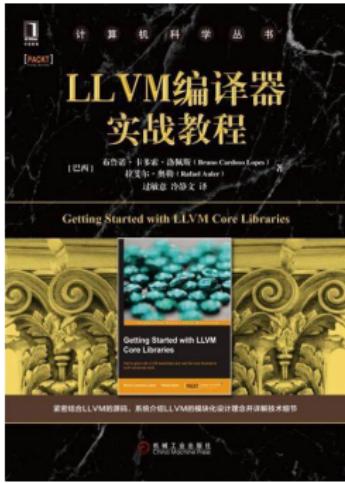


[https://llvm.org/docs/  
CodeGenerator.html](https://llvm.org/docs/CodeGenerator.html)

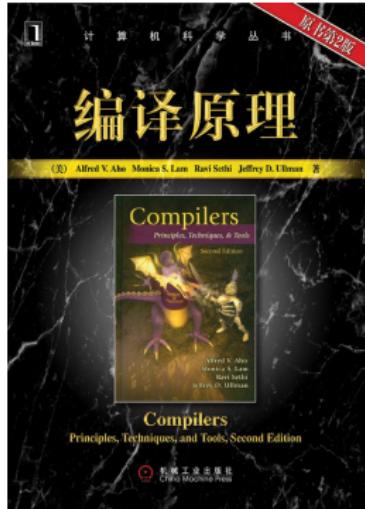


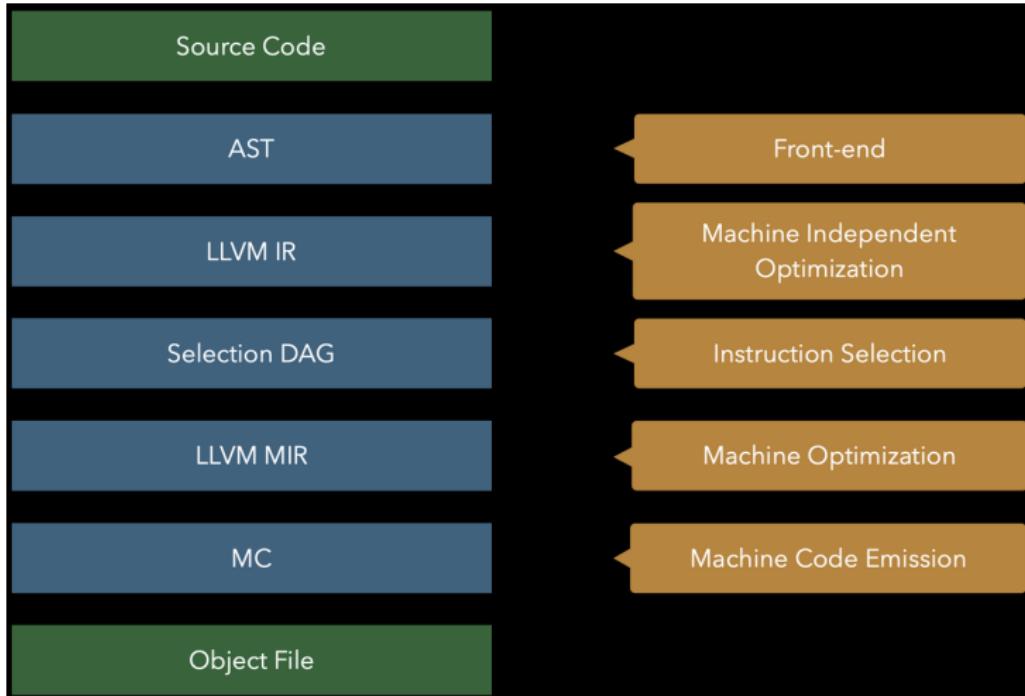
<https://llvm.org/docs/>

CodeGenerator.html

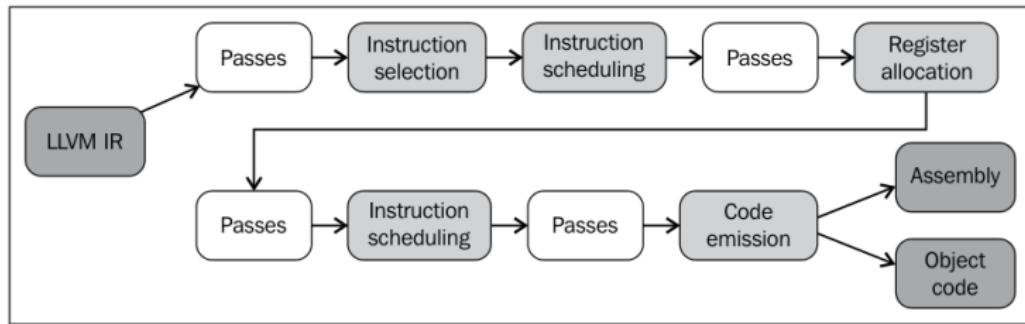


[https://llvm.org/docs/  
CodeGenerator.html](https://llvm.org/docs/CodeGenerator.html)

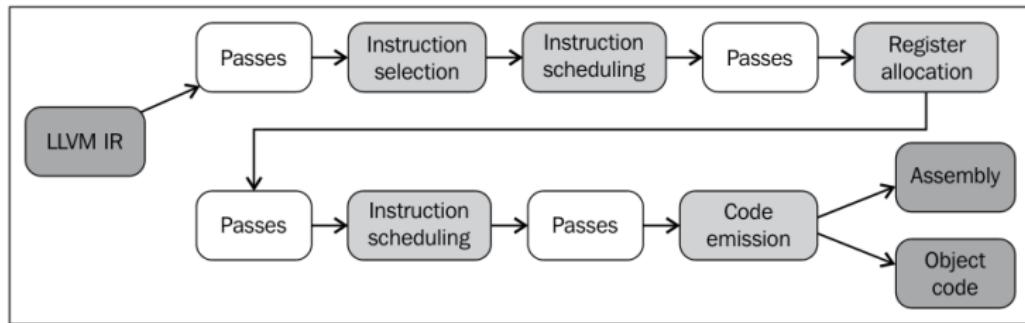




(in-memory) LLVM IR SelectionDAG MachineInstr MCInst



(in-memory) LLVM IR    SelectionDAG    MachineInstr    MCInst



Where is “Prologue/Epilogue Insertion”?

# f0-O0 @ Compiler Explorer

The screenshot shows the Compiler Explorer interface with three main panes:

- Left Pane:** C source code for a factorial function. The code is annotated with assembly instructions corresponding to each line:

```
1 int facto
2
3 int main()
4     return
5 }
```

Annotations include: 1: addi \$x1, \$zero, 1; 2: sw \$x1, 4(\$x1); 3: sw \$x1, 8(\$x1); 4: addi \$x1, \$x1, \$x1; 5: li \$x1, 1; 6: sw \$x1, 12(\$x1); 7: addi \$x1, \$x1, 1; 8: sw \$x1, 16(\$x1); 9: li \$x1, 2; 10: addi \$x1, \$x1, 1; 11: call factorial; 12: mv \$x1, \$x1; 13: slli \$x1, \$x1, 2; 14: sub \$x1, \$x1, 1; 15: addi \$x1, \$x1, 1; 16: seqz \$x1, 1; 17: lw \$x1, 12(\$x1); 18: lw \$x1, 16(\$x1); 19: addi \$x1, \$x1, 1; 20: ret;
- Middle Pane:** LLVM IR Viewer showing the LLVM IR for the RISC-V f0-O0 build. It includes a Control Flow Graph and a detailed list of optimization passes applied to the DAG:
  - RISC-V DAG -> DAG Pattern
  - Instruction Selection (riscv-isel)
  - Eliminate PHI nodes for register allocation (phi-node-elimination)
  - Two-Address instruction pass (twoaddressinstruction)
  - Fast Register Allocator (regallocfast)
  - Slot index numbering (slotindexes)
  - Live Interval Analysis (liveintervals)
  - Fast Register Allocator (regallocfast)
  - Prologue/Epilogue Insertion & Frame Finalization (prologueepilog)
  - Post-RA pseudo instruction expansion pass (postrapseudos)
- Right Pane:** Opt Pipeline Viewer showing the assembly output for the RISC-V f0-O0 build. The assembly code is color-coded by section:

```
14+ SW killed $x1, $x2, 28 :: (stor
15+ SW killed $x8, $x2, 24 :: (stor
16+ frame-setup CFI_INSTRUCTION off
17+ frame-setup CFI_INSTRUCTION off
18+ $x8 = frame-setup ADDI $x2, 32
19+ frame-setup CFI_INSTRUCTION def
1 20 renamable $x12 = COPY $x0
1 21 SW killed renamable $x12, %stack
1 22 SW killed renamable $x10, %stack
1 23 SW killed renamable $x11, %stack
1 24 ADJCALLSTACKDOWN 0, 0, implicit
1 25 SW killed renamable $x12, $x8,
21+ SW killed renamable $x10, $x8,
22+ SW killed renamable $x11, $x8,
23+ SW killed renamable $x11, $x8,
1 24 renamable $x10 = ADDI $x0, 2
1 25 PseudoCALL target-flags(riscv-c
1 26 ADJCALLSTACKUP 0, 0, implicit-c
1 27 renamable $x11 = COPY killed $x
1 28 renamable $x10 = SLLI renamable
2 29 renamable $x10 = SUB killed ren
2 30 renamable $x10 = SLTIU killed r
31+ $x1 = LW $x2, 28 :: (load ($s32)
32+ $x8 = LW $x2, 24 :: (load ($s32)
33+ $x2 = frame-destroy ADDI $x2, 3
2 34 PseudoRET implicit killed $x10;
2 35
```

# f0-O1 @ Compiler Explorer

The screenshot shows the Compiler Explorer interface with three main panes:

- Left Pane (Compiler Explorer):** Displays the C source code for a factorial function:

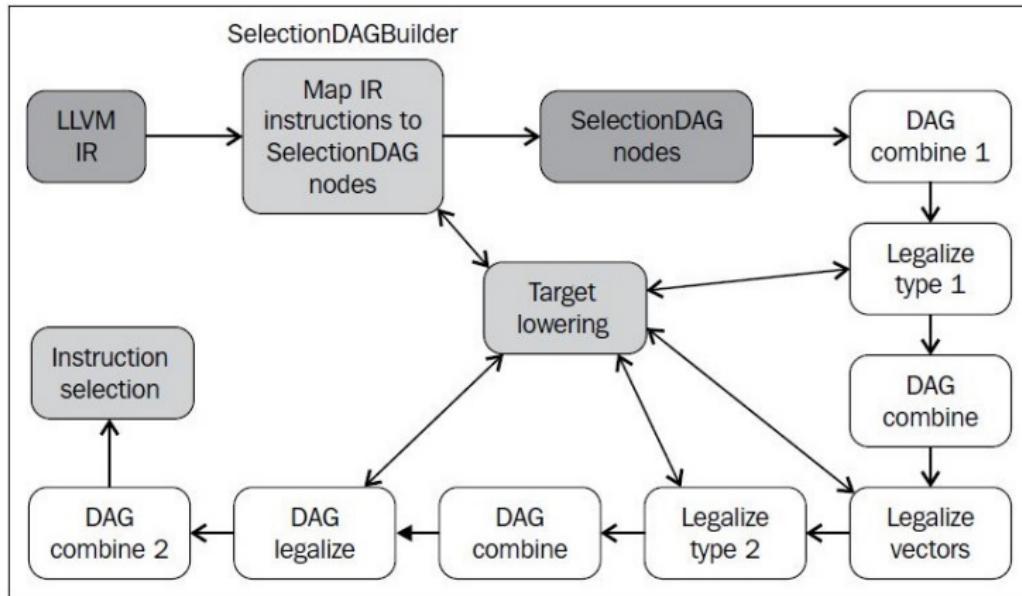
```
1 int facto
2
3 int main()
4     return
5 }
```

The `main` function is highlighted.
- Middle Pane (LLVM IR Viewer):** Shows the LLVM Intermediate Representation (IR) for the same code:

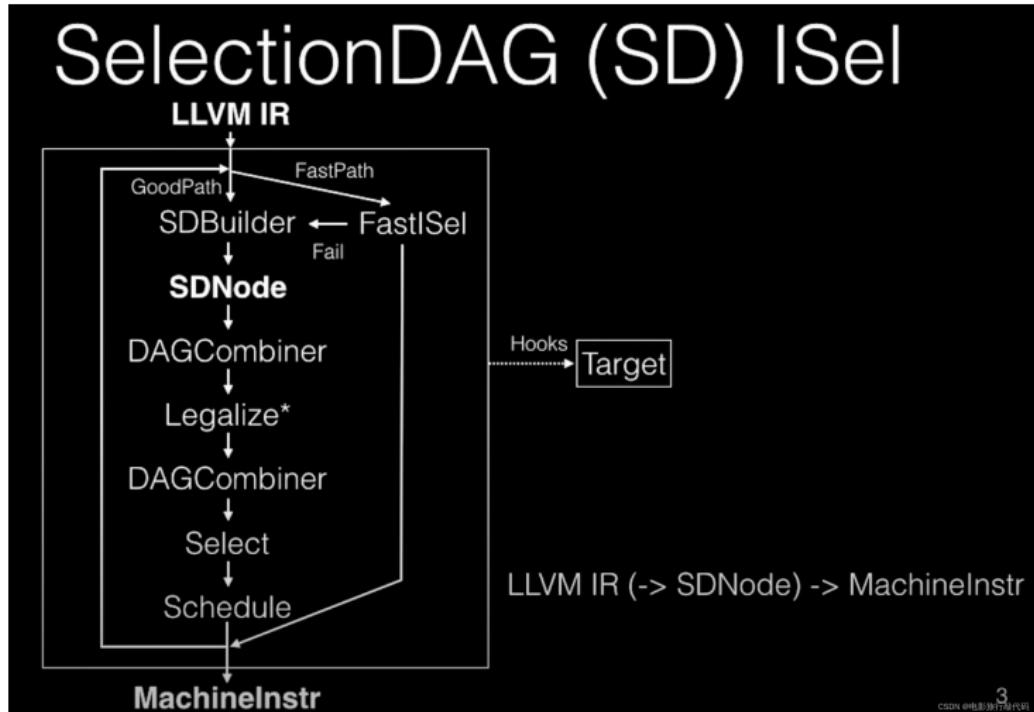
```
1 define dso_local range(i32 0, 2)
entry:
  %call = tail call i32 @_factoria
  %cmp = icmp eq i32 %call, 6
  %conv = zext i1 %cmp to i32
  ret i32 %conv
}
1# Machine code for function main:
2+
3 bb.0.entry:
4+ ADDCALLSTACKDOWN 0, 0, implicit
5+ %2:gpr = ADDI $x0, 2
6+ $x10 = COPY %2:gpr; example.c:4
7+ PseudoCALL target-flags(riscv-c
8+ ADDCALLSTACKUP 0, 0, implicit-c
9+ %3:gpr = COPY $x10; example.c:4
10+ %4:gpr = ADDI %3:gpr, -6; example.c:4
11+ %5:gpr = SLTIU killed %4:gpr, 1
12+ $x10 = COPY %5:gpr; example.c:4
13+ PseudoRET implicit $x10; example.c:4
14+
15# End machine code for function main:
```
- Right Pane (Opt Pipeline Viewer):** Displays the optimization pipeline passes for the LLVM IR:
  - AssignmentTrackingPass on [module]
  - SROAPass on main
  - IPSCCPPass on [module]
  - GlobalOptPass on [module]
  - InstCombinePass on main
  - PostOrderFunctionAttrsPass on (main)
  - TailCallElimPass on main
  - RISC-V DAG->DAG Pattern
  - Instruction Selection (riscv-isel)
  - Slot index numbering (slotindexes)
  - Merge disjoint stack slots (stack-coloring)
  - Machine code sinking (machine-sink)
  - Live Variable Analysis (livevars)
  - Eliminate PHI nodes for register allocation (phi-node-elimination)
  - Two-Address instruction pass (twoaddressinstruction)
  - Slot index numbering (slotindexes)
  - Live Interval Analysis

## Instruction Selection

Instruction Selection is the process of translating LLVM code presented to the code generator into target-specific machine instructions. There are several well-known ways to do this in the literature. LLVM uses a SelectionDAG based instruction selector.



SDISel    FastISel (per basic block)    GlobalISel (per function)



GlobalISel (per function)

```
unsigned int MUL(unsigned long long int x, unsigned int y)
{
    return x * y;
}
```

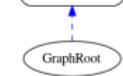
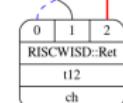
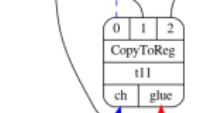
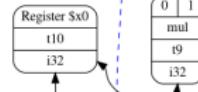
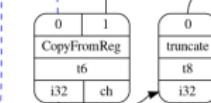
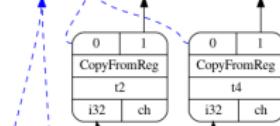
```
unsigned int MUL(unsigned long long int x, unsigned int y)
{
    return x * y;
}
```

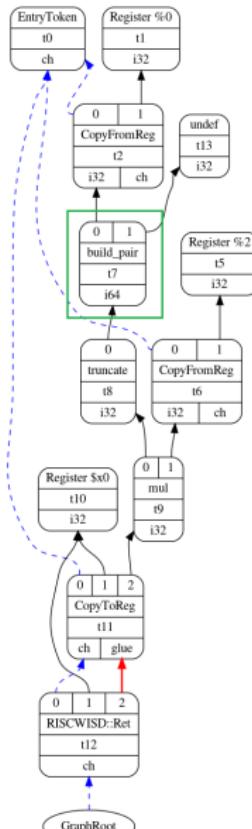
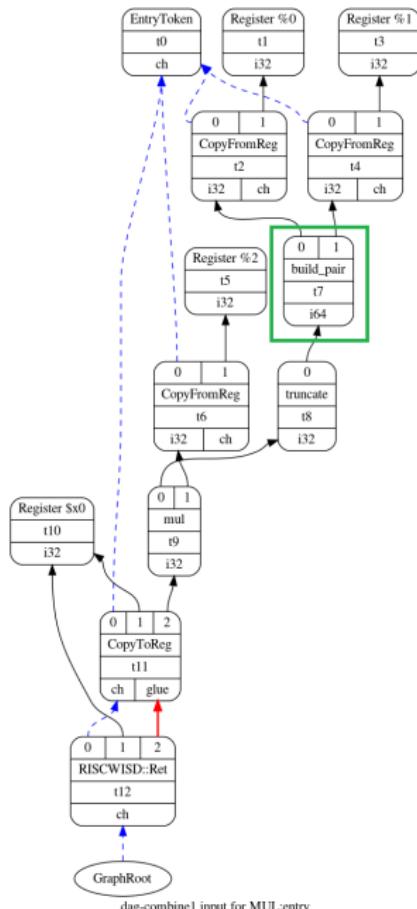
```
define dso_local i32 @MUL(i64 %x, i32 %y) local_unnamed_addr #0 {
entry:
    %0 = trunc i64 %x to i32
    %conv1 = mul i32 %0, %y
    ret i32 %conv1
}
```

```

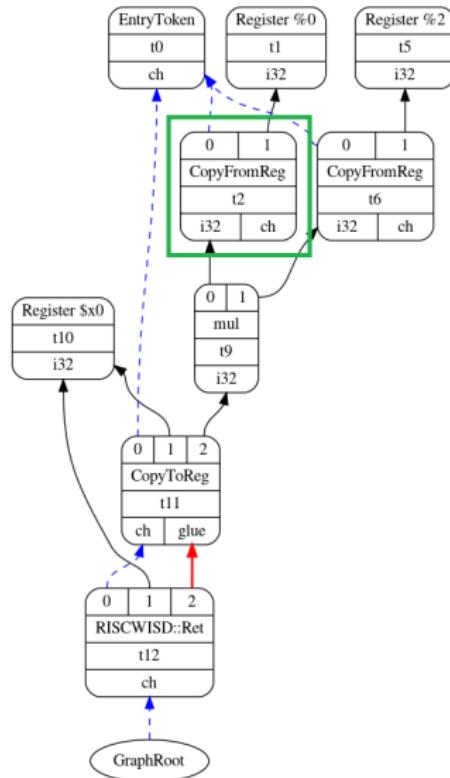
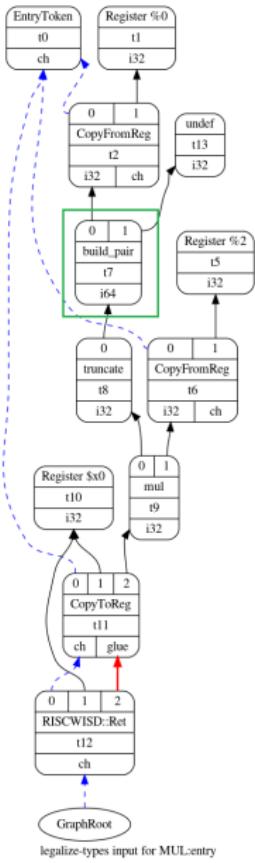
define dso_local i32 @MUL(i64 %x, i32 %y) local_unnamed_addr #0 {
entry:
    %0 = trunc i64 %x to i32
    %conv1 = mul i32 %0, %y
    ret i32 %conv1
}

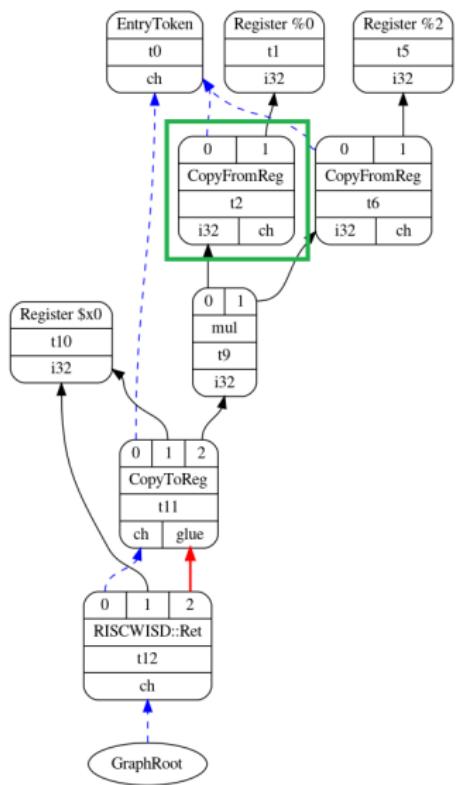
```



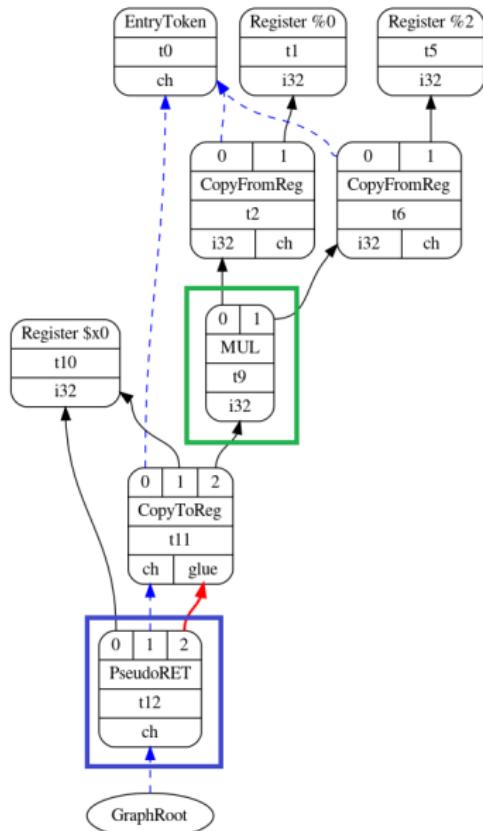


legalize-types input for MUL:entry

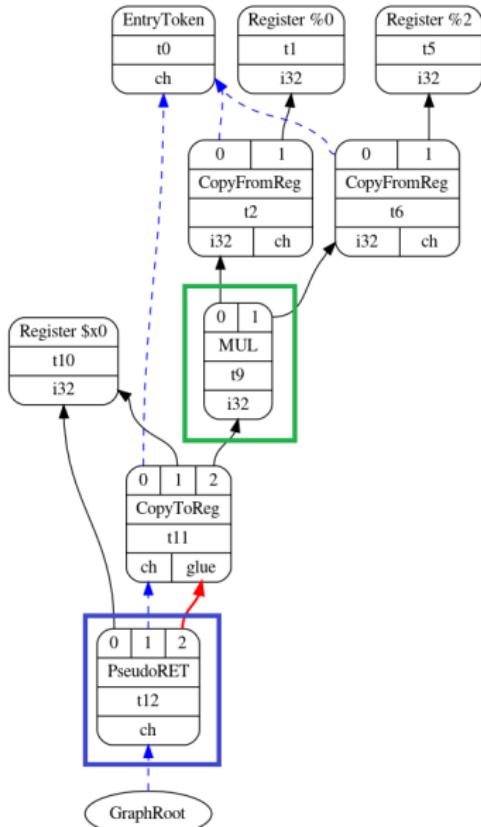




### **dag-combine2 input for MUL:entry**



scheduler input for MUL:entry



scheduler input for MUL:entry

bb. 0. entry:

liveins: \$x0, \$x2

%2:gpr = COPY \$x2

%0:gpr = COPY \$x0

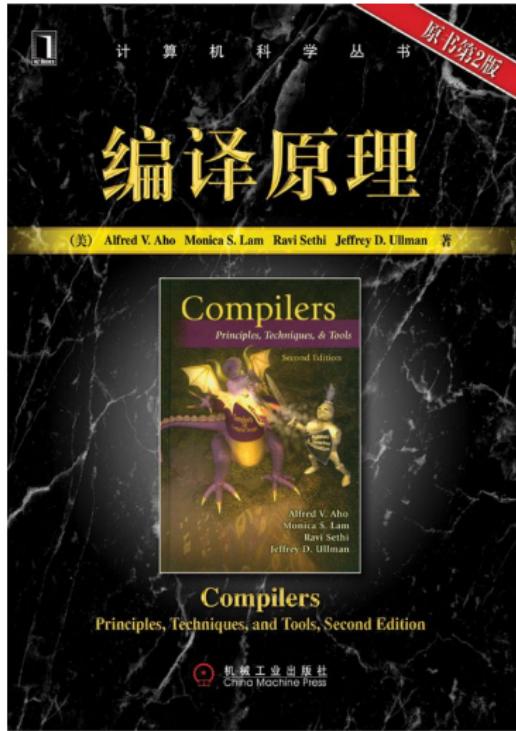
%3:gpr = MUL %0:gpr, %2:gpr

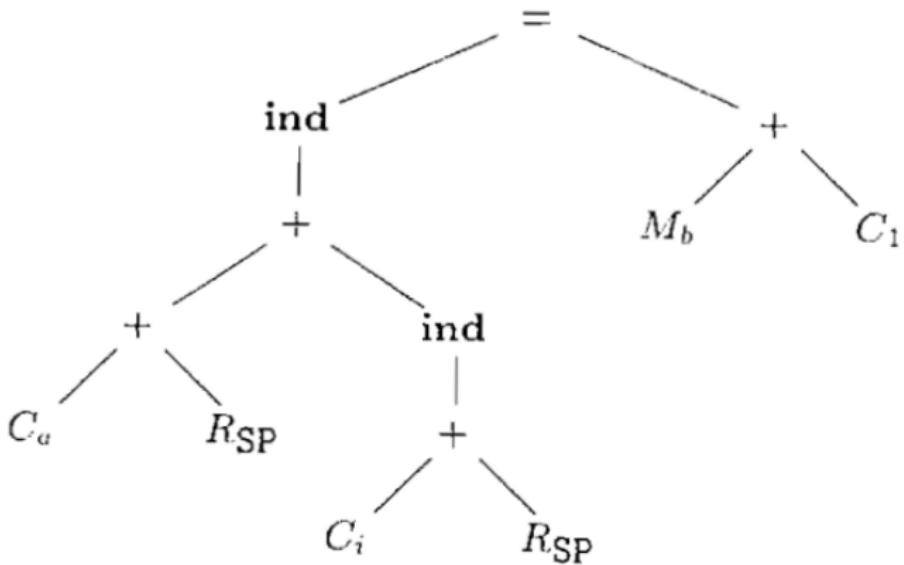
\$x0 = COPY %3:gpr

PseudoRET implicit \$x0

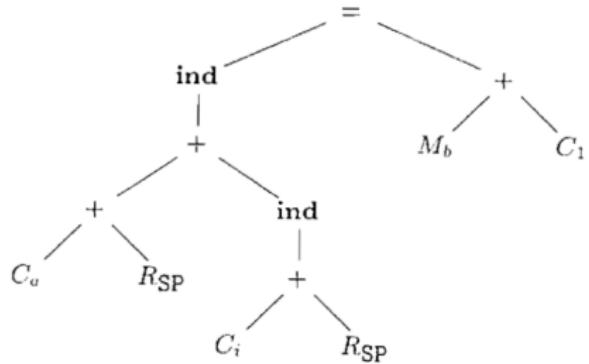
## SelectionDAG Select Phase

The Select phase is the bulk of the target-specific code for instruction selection. This phase takes a legal SelectionDAG as input, pattern matches the instructions supported by the target to this DAG, and produces a new DAG of target code. For example, consider the following LLVM fragment:



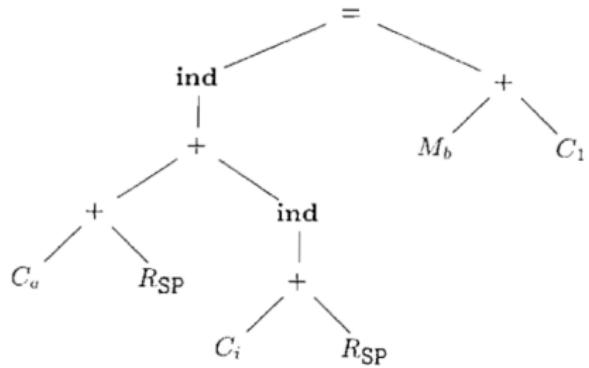


局部变量 a 与 i 的地址: 相对于 SP (栈指针) 的常数偏移量  $C_a$  和  $C_i$



LD R0, #a  
 ADD R0, R0, SP  
 ADD R0, R0, i(SP)  
 LD R1, b  
 INC R1  
 ST \*R0, R1

$R_i \leftarrow$   
 $R_i \quad / \backslash$   
 $\quad \quad \quad +$   
 $\quad \quad \quad \quad \quad C_a$   
 { if ( $a = 1$ )  
   INC Ri  
   else  
     ADD Ri, Ri, #a }



$= \text{ind} + + C_a R_{SP} \text{ ind} + C_i R_{SP} + M_b C_1$

1)	$R_i \rightarrow c_a$	{ LD Ri, #a }
2)	$R_i \rightarrow M_x$	{ LD Ri, x }
3)	$M \rightarrow = M_x R_i$	{ ST x, Ri }
4)	$M \rightarrow = \text{ind} R_i R_j$	{ ST *Ri, Rj }
5)	$R_i \rightarrow \text{ind} + c_a R_j$	{ LD Ri, a(Rj) }
6)	$R_i \rightarrow + R_i \text{ ind} + c_a R_j$	{ ADD Ri, Ri, a(Rj) }
7)	$R_i \rightarrow + R_i R_j$	{ ADD Ri, Ri, Rj }
8)	$R_i \rightarrow + R_i c_1$	{ INC Ri }
9)	$R \rightarrow \text{sp}$	
10)	$M \rightarrow m$	

$= \text{ind} + + c_a \text{ sp ind} + c_i \text{ sp} + m_b c_1$

前缀表示

二义性: 倾向于执行较大的归约, 而不是较小的归约

归约/归约冲突

移入/归约冲突

优先选择较长的归约

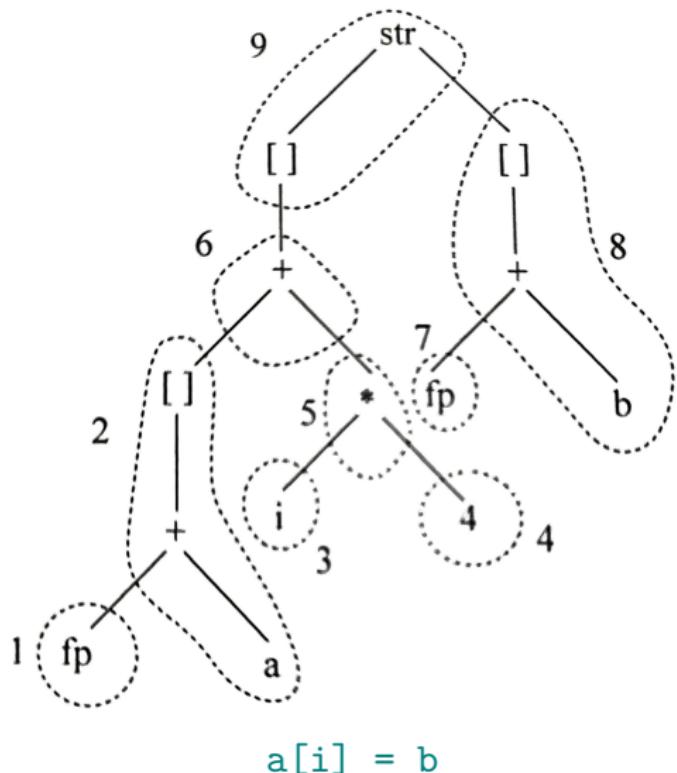
优先选择移入动作



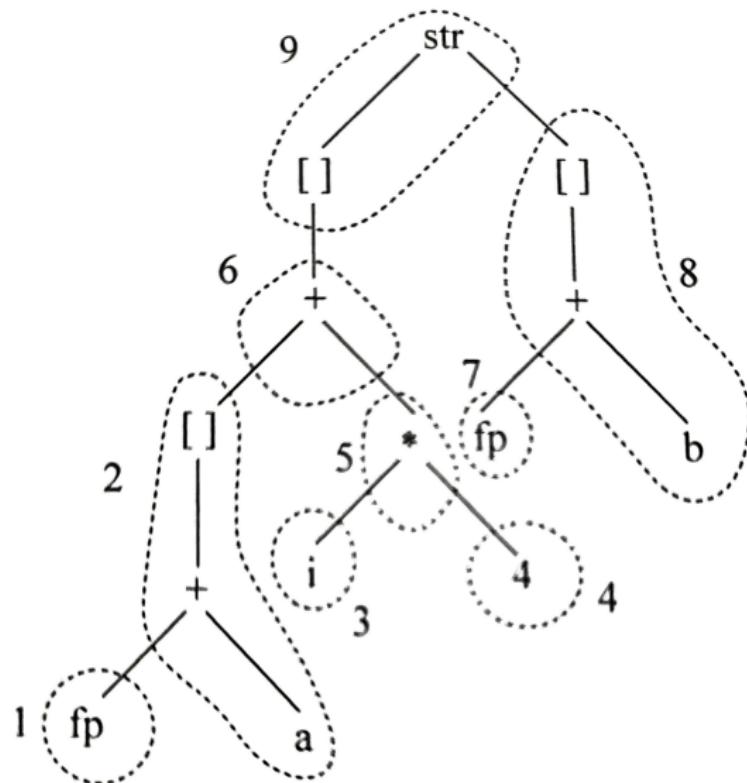
# 算术与存储指令及树型 (鲲鹏处理器)

指    令	树    型
$r_i$	x
add $r_i, r_j, r_k$	$+ \diagup \diagdown$
mul $r_i, r_j, r_k$	$* \diagup \diagdown$
sub $r_i, r_j, r_k$	$- \diagup \diagdown$
div $r_i, r_j, r_k$	$/ \diagup \diagdown$
addi $r_i, r_j, c$	$+ \diagup \quad + \diagup \quad c$ $c \diagdown \quad c \diagdown$
subi $r_i, r_j, c$	$- \diagup \quad - \diagup \quad c$ $c \diagdown \quad c \diagdown$
ldr $r_i, [r_j, c]$	$[] \quad [] \quad [] \quad []$ $c \quad + \quad c \quad + \quad c \quad  $
str $[r_j, c], r_i$	$str \quad str \quad str \quad str$ $[] \quad [] \quad [] \quad []$ $c \quad + \quad c \quad  $

## 瓦片覆盖 (tiling)



```
| ldr r1, [fp, a]  
| addi r2, r0, 4  
| mul r2, ri, r2  
| add r1, r1, r2  
| ldr r2, [fp, b]  
| str [r1, 0], r2
```



`a[i] = b`

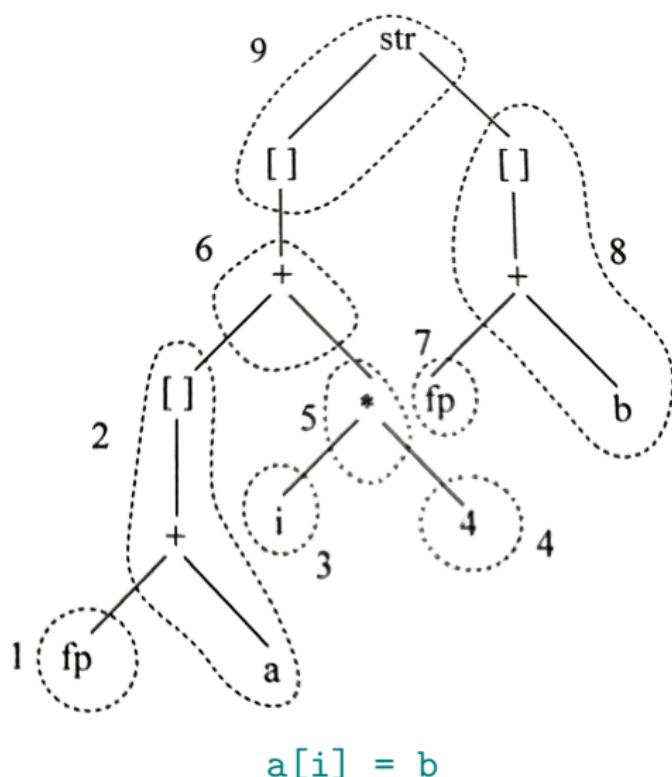
目标代码生成

魏恒峰 (hfwei@nju.edu.cn)

2024 年 06 月 12 日

25 / 29

## 瓦片覆盖 (tiling)



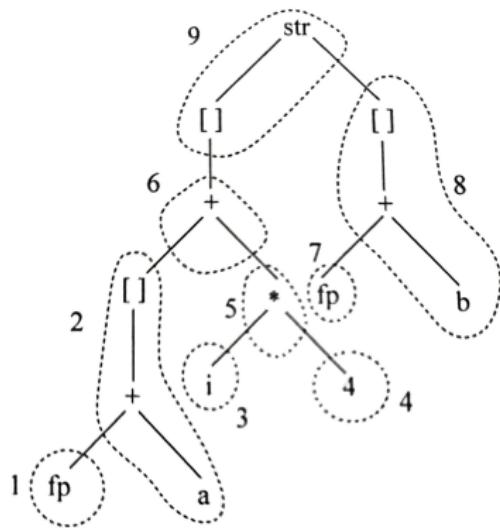
最大吞进算法  
(maximal munch)

```
ldr r1, [fp, a]
addi r2, r0, 4
mul r2, ri, r2
add r1, r1, r2
ldr r2, [fp, b]
str [r1, 0], r2
```

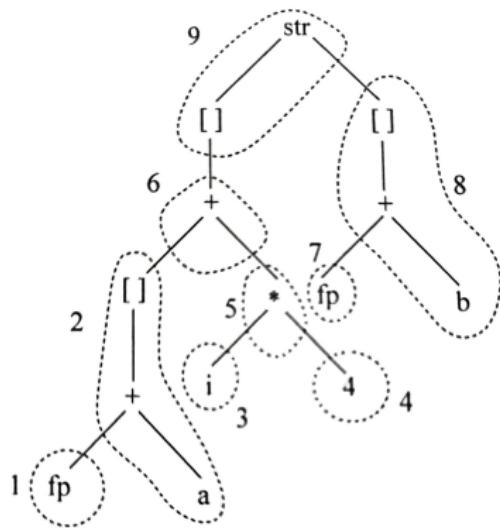
最佳覆盖方案

(不存在可以合并成更小代价树型的相邻树型)

## 使用动态规划思想计算最优覆盖方案

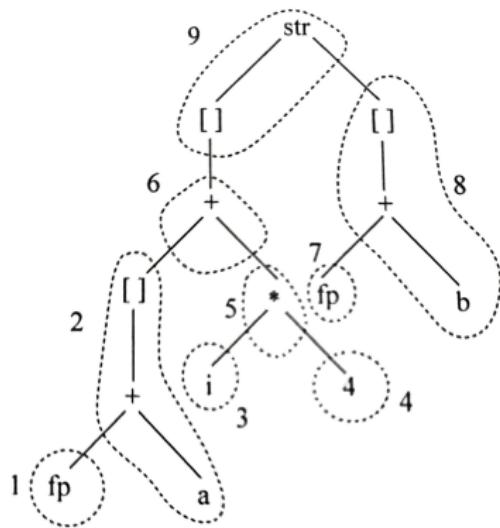


## 使用动态规划思想计算最优覆盖方案



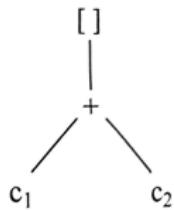
$$\forall n \in Node. c(n) = \min_{t \in tiles(n)} \left( c(t) + \sum_{n_i \in descendants(n)} c(n_i) \right)$$

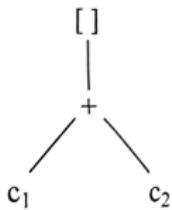
## 使用动态规划思想计算最优覆盖方案



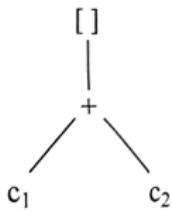
$$\forall n \in Node. c(n) = \min_{t \in tiles(n)} \left( c(t) + \sum_{n_i \in descendants(n)} c(n_i) \right)$$

$$\text{Opt} = c(r) \quad (r \text{ is the root})$$





瓦片	指令	瓦片代价	叶子节点代价	总代价
	add	1	$1+1$	3
	addi	1	1	2
	addi	1	1	2



瓦片	指令	瓦片代价	叶子节点代价	总代价
	add	1	1+1	3
	addi	1	1	2
	addi	1	1	2

瓦片	指令	瓦片代价	叶子节点代价	总代价
	ldr	1	2	3
	ldr	1	1	2
	ldr	1	1	2

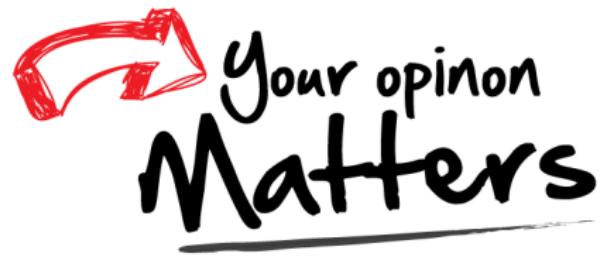
Gabriel Hjort Blindell

# Instruction Selection

Principles, Methods, and Applications

 Springer

# Thank You!



Office 926

hfwei@nju.edu.cn