

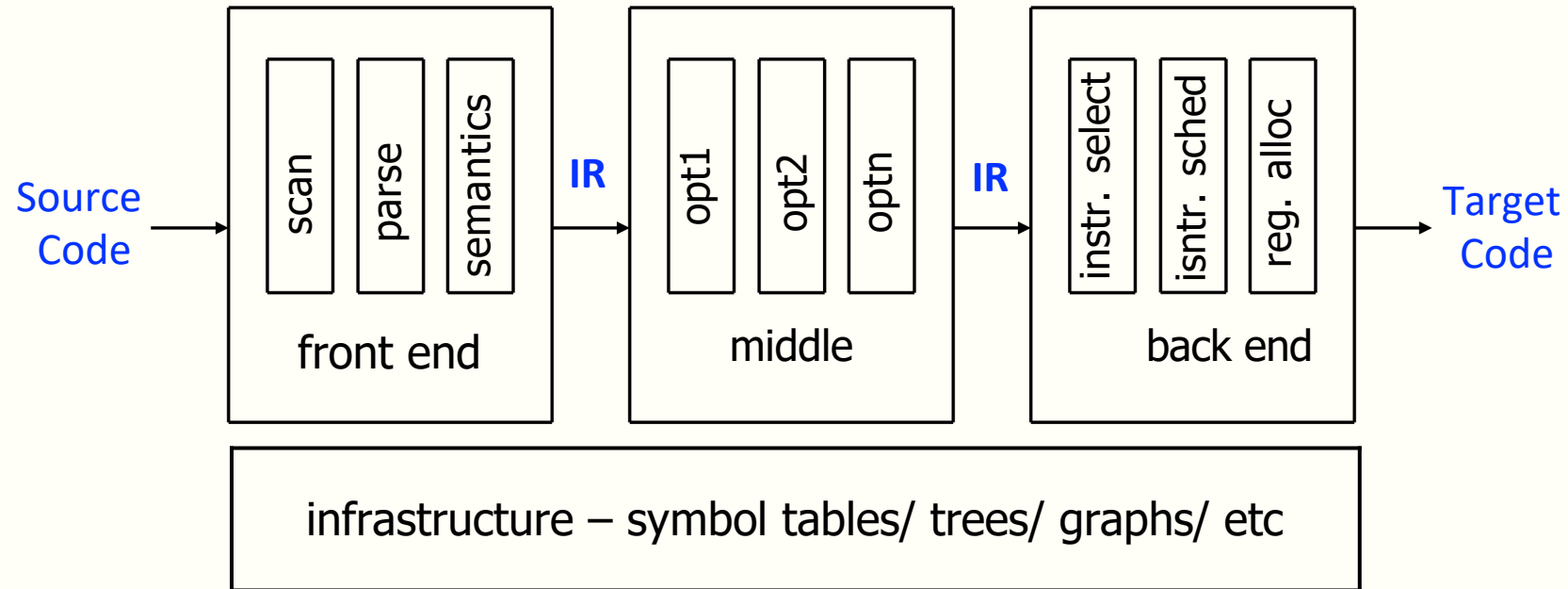


《编译原理》

第6讲：指令选择

何冬杰
重庆大学

Big Picture of Compiler



- ❑ **Front End:** produces an intermediate representation (IR)
- ❑ **Middle end:** transforms the IR into an equivalent IR that runs more efficiently
- ❑ **Back end:** transforms the IR into native code
- ❑ **IR:** encodes the compiler's knowledge of the program Middle end usually consists of several passes

Compiler Backend

❑ Instruction Selection: map IR into assembly code

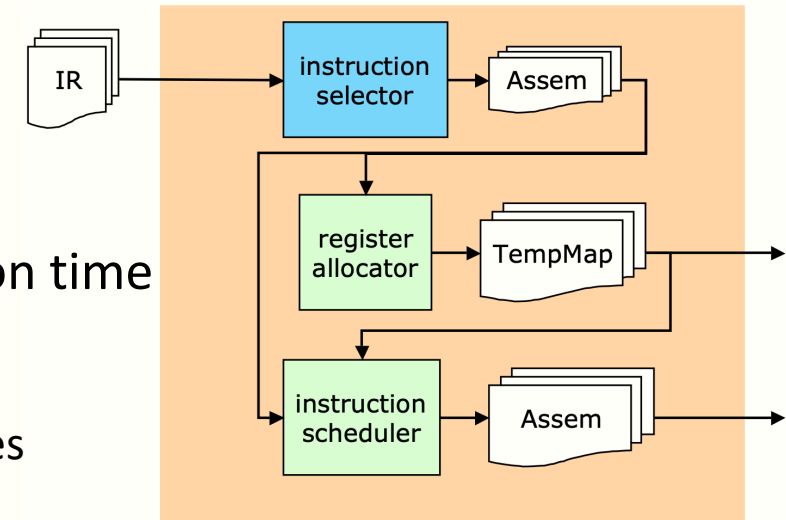
- Assume known storage layout and code shape
 - i.e., the optimization phases have already done their thing
- Combine low-level IR operations into machine instructions
 - Take advantage of addressing modes, etc.

❑ Instruction Scheduling: reorder instructions to minimize execution time

- hide latencies: processor function units, memory/cache stalls
- Originally invented for supercomputers (**1960s**)
- Required to get reasonable (or correct!) code on classic **RISC** architectures
- Still important on most machines
 - Even non-RISC machines, e.g., the **x86** family
 - Even if the processor **reorders on the fly** (处理器在程序运行时可以动态调整指令执行顺序以提高执行效率)
- Good schedules help processors do a better job

❑ Register Allocation: map values to actual registers

- Add code to spill values to temporaries in memory and reload as needed, etc.
- Usually worth doing another scheduling pass afterwards if spill code is inserted



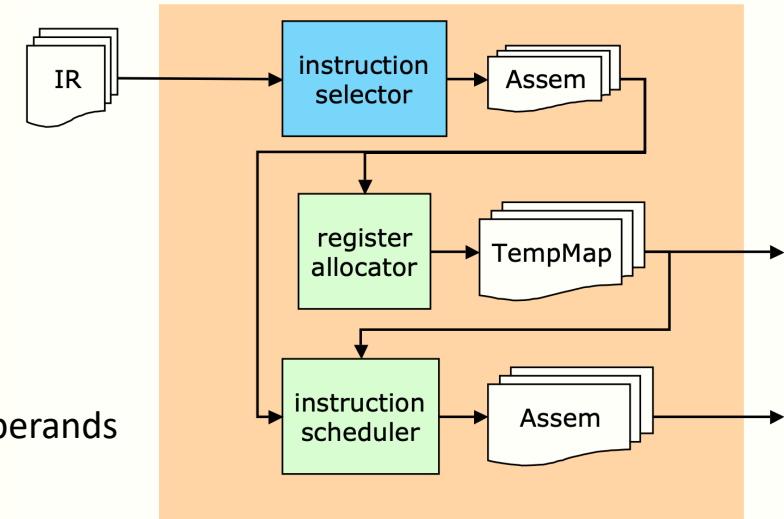
Conventional Wisdom

❑ Backend problems are almost **NP-Complete** in theory, including

- Instruction selection, Instruction scheduling, and Register allocation
- Frontend and Middle-end algorithms are usually fast:
 - Scanner & Parser: $O(n)$
 - Analysis & Optimization: $O(n \log n)$

❑ We typically lose little by solving these **independently**

- But not always, of course
 - **Iterating** phases on **x86-64** can help because of limited registers; use of memory operands
- Instruction selection
 - Use some form of pattern matching
 - ∞ virtual registers, create as needed
- Instruction scheduling
 - Within a block, list scheduling is close to optimal
 - Across blocks: extended basic blocks or trace scheduling if list scheduling not good enough
- Register allocation
 - Start with unlimited virtual registers and map to some subset of K real registers



Instruction Selection

❑ Translate low-level IR to assembly instructions on the target machine

❑ **Straightforward solution** (sometimes called **Macro-expansion**):

- Translate each low-level IR instruction to a sequence of machine instructions
- Example: $a[i+1] = b[j]$

Address of $b[j]$:	multi t0, s2, 4	→ {	t1 = j*4
	add t1, t0, s4		
Load value $b[j]$:	ld t2, 0(t1)	→ {	t2 = b+t1
	addi t3, s1, 1		t3 = *t2
Address of $a[i+1]$:	multi t3, t3, 4		t4 = i+1
	add t4, t3, s3		t5 = t4*4
Store into $a[i+1]$:	sd t2, 0(t4)		t6 = a+t5
			*t6 = t3

- Often introducing **Machine IR (MIR)** before emitting the assembly code

Machine IR (MIR)

Machine IR:

- 指令选择阶段产出的、和目标机器指令集架构绑定的中间表示
- 操作码：已经是目标 CPU 的指令（lw/add/slli这类 RISC-V 指令）
- 操作数：一开始是**虚拟寄存器**，后面寄存器分配替换为**物理寄存器**
- 本质就是**机器指令的抽象内部表示**，介于IR和汇编代码之间
- LLVM 里面叫MachineInstr

MIR程序 (**MIR.ml**)：结构上与IR一致

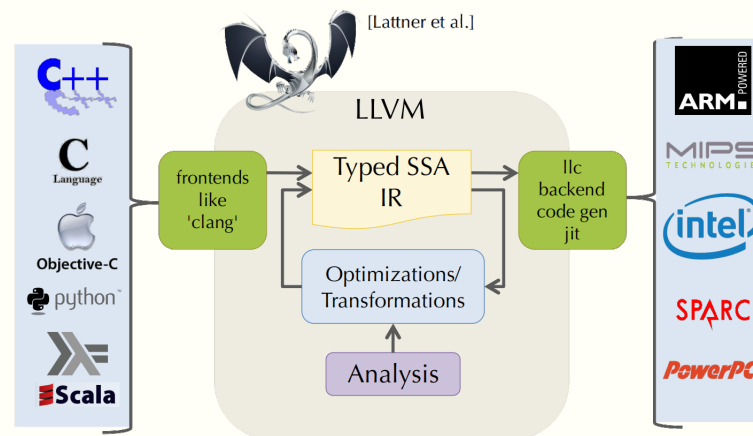
```
type block = { insns : insn list; term : terminator }
type cfg = block * (lbl * block) list
type fndef = { name : gid; param : vreg list; cfg : cfg; }
type data = ZeroBytes of int | Byte of int | Word of int64 | Address of gid
type gvdef = { size : int; align : int; init : data list; const : bool }
type program = { gvdefs : (gid * gvdef) list; fndefs : (gid * fndef) list }
```

- 寄存器：约定id ≥ 32 的为虚拟寄存器，0-31为RISC-V的物理寄存器

```
type vreg = VReg of int
```

```
type preg = X of int
```

```
type reg = R of int
```



Machine IR (MIR)

□MIR程序：指令与RISC-V汇编指令基本上——对齐

```
type insn = Add of reg * reg * reg | Sub of reg * reg * reg | Mul of reg * reg * reg
| Div of reg * reg * reg | Rem of reg * reg * reg | And of reg * reg * reg
| Or of reg * reg * reg | Sll of reg * reg * reg | Sra of reg * reg * reg
| Xor of reg * reg * reg | Slt of reg * reg * reg | Addi of reg * reg * int64
| Andi of reg * reg * int64 | Ori of reg * reg * int64 | Xori of reg * reg * int64
| Slli of reg * reg * int64 | Srli of reg * reg * int64 | Srai of reg * reg * int64
| Seqz of reg * reg | Li of reg * int64 | Mv of reg * reg | La of reg * gid
| Ld of reg * address | Lb of reg * address | Sd of address * reg | Sb of address * reg
| Call of gid (* direct call; args already placed in a0-a7 *)
| Jalr of reg (* indirect call (dispatch); target in reg *)
```

●分支跳转指令被融合进终止指令中

```
type cmpop = Eq | Ne | Lt | Gt | Le | Ge
type terminator = Ret | Br of lbl | Cbr of reg * lbl * lbl
| CmpBr of cmpop * reg * reg * lbl * lbl
```

type address = Stack of int
| RegOff of reg * int64

分为栈上的slot位置
和相对基地址的偏移

Instruction Selection (Straightforward solution)

□ IR转化为MIR (lowering, see [MIRGen.ml](#))

- 全局变量转为MIR全局数据
- 函数参数转为MIR虚拟寄存器
 - 参数按调用约定从a0-a7等物理寄存器传入, 目前限制函数最多只有8个参数
 - 可扩展支持更多参数, 超出参数可放置栈上
- IR指令转为MIR指令: MIR指令含虚拟寄存器%vN, 需做寄存器分配
- IR终结符转为MIR控制流终结符

□ IR的值如何映射到MIR?

- IR的UID映射到MIR的寄存器: e.g, Store (I64, Id "a", Id "ap") => Sd %v4, 0(%v3)
- 立即数和全局符号在需要时才具体化:
 - IR常量5 => li %v2, 5
 - IR全局变量@g => la %v3, @g

□ IR指令如何变成MIR?

- 算术指令和MIR几乎一一对应, 支持立即数运算的优先选择立即数形式
- 比较指令会生成布尔值: lcmp (Eq, I64, Id "a", Id "b") => xor %v1, %lhs, %rhs ; seqz result, %v1

Instruction Selection (Straightforward solution)

□ IR指令如何变成MIR (MIRGen.ml) ?

● Lower Alloca指令: (实际还有指令对齐)

- Alloca (ty, cnt) => %v1 = elem_size of ty; %v2 = Mul cnt * %v1; sp = sp - %v2; mv %dst, sp

● Lower Load/Store指令: Load (ty, opn) , Store (ty, opn, addr)

- 根据指令类型进行加载或存储, I64使用ld/sd, I8用lb/sb
- I1和I8类型的数据需要特别处理: 用掩码或移位, 让值保持规定的位宽或符号语义

● Cast类型转换: Cast (cop, s_ty*opn, d_ty)

- SExt (符号扩展) , ZExt (零扩展) , Trunc (截断)
- 根据转换种类生成相应截断、扩展和归一化操作

● 函数调用指令:

- 将参数搬到虚拟寄存器中: mv %arg_reg, %arg
- Global "callee"=> call @callee , 等价于auipc ra, %pcrel_hi(callee) ; jalr ra, %pcrel_lo(callee)(ra)
- 其他情况调用: jalr %tgt_reg
- 若有返回值, %dst_uid = a0

● Lower终止指令:

- Ret None: li a0, 0; ret
- Ret (ty, opn): mv a0, %opn_value; ret
- Br和Cbr 分别映射到MIR的无条件或条件分支

- CmpBr指令: 对应 IR 中相邻且可融合的 ICmp + Cbr, 要求ICmp只被使用1次, 且是在Cbr中使用

● Gep指令: 逐层计算offset

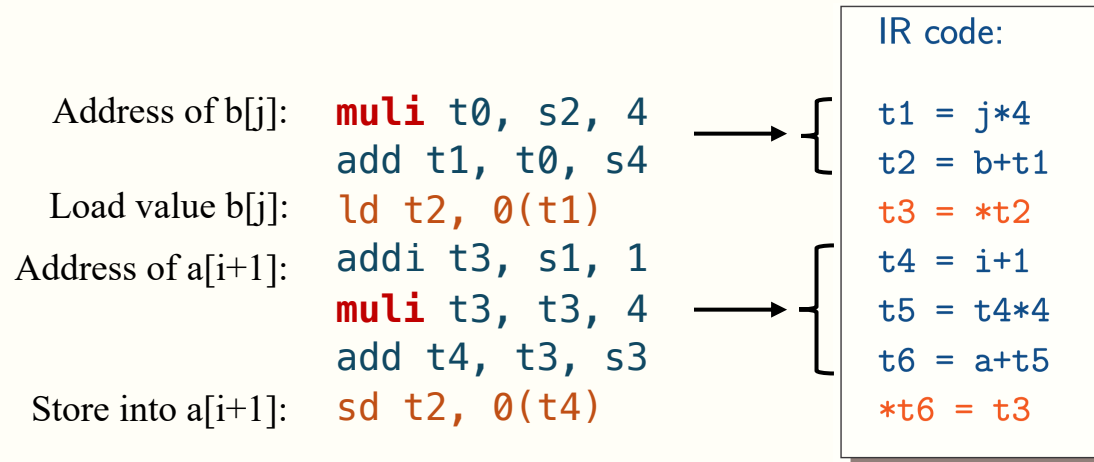
```
let lower_insn state (dst_uid, insn) =  
  match insn with  
  | Binop (op, opn_ty, lhs, rhs) ->  
    lower_binop state (dst_uid, op, opn_ty, lhs, rhs)  
  | ICmp (cmpop, opn_ty, lhs, rhs) ->  
    lower_icmp state (dst_uid, cmpop, opn_ty, lhs, rhs)  
  | Alloca (element_ty, count) ->  
    lower_alloca state (dst_uid, element_ty, count)  
  | Load (value_ty, address) ->  
    lower_load state (dst_uid, value_ty, address)  
  | Store (value_ty, value, address) ->  
    lower_store state (value_ty, value, address)  
  | Gep (elem_ty, base, indices) ->  
    lower_gep state (dst_uid, elem_ty, base, indices)  
  | Cast (cast_kind, (source_ty, value), target_ty) ->  
    lower_cast state (dst_uid, cast_kind, source_ty, value, target_ty)  
  | Call (call_sig, callee, args) ->  
    lower_call state (dst_uid, call_sig, callee, args)
```

```
insn ::= ... | getelementptr t* %val, t1 idx1, t2 idx2 ,...
```

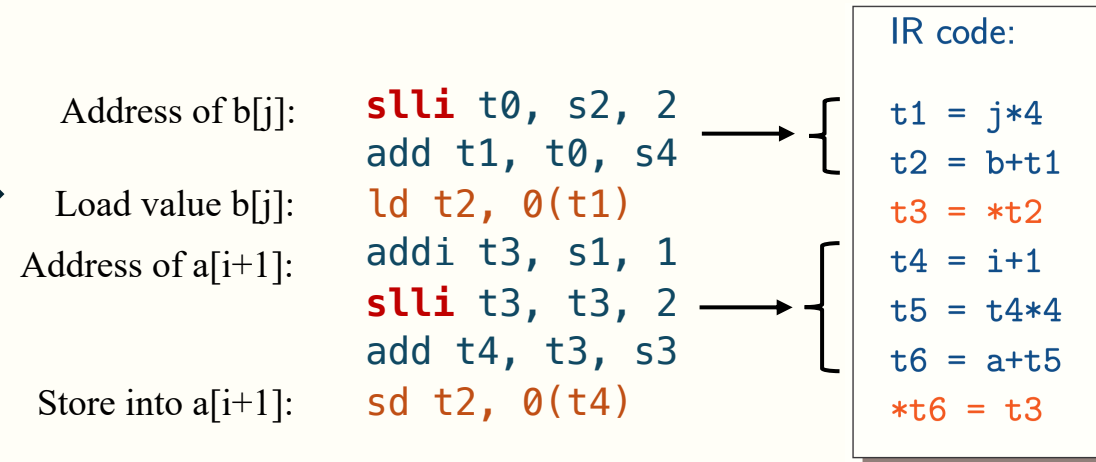
Instruction Selection

❑ Problems of Straightforward solution

- A straightforward translation is **inefficient**
- Prefer to use cheap instructions:
 - addi vs add; muli vs slli; ...
- Low-level IR instructions matched by machine instructions may **not be adjacent** (影响指令选择复杂度)
 - Example: `ld t2, 0(t1)` and `sd t2, 0(t4)`, x86系统中可以直接使用 `movem t2, t4`



A straightforward solution

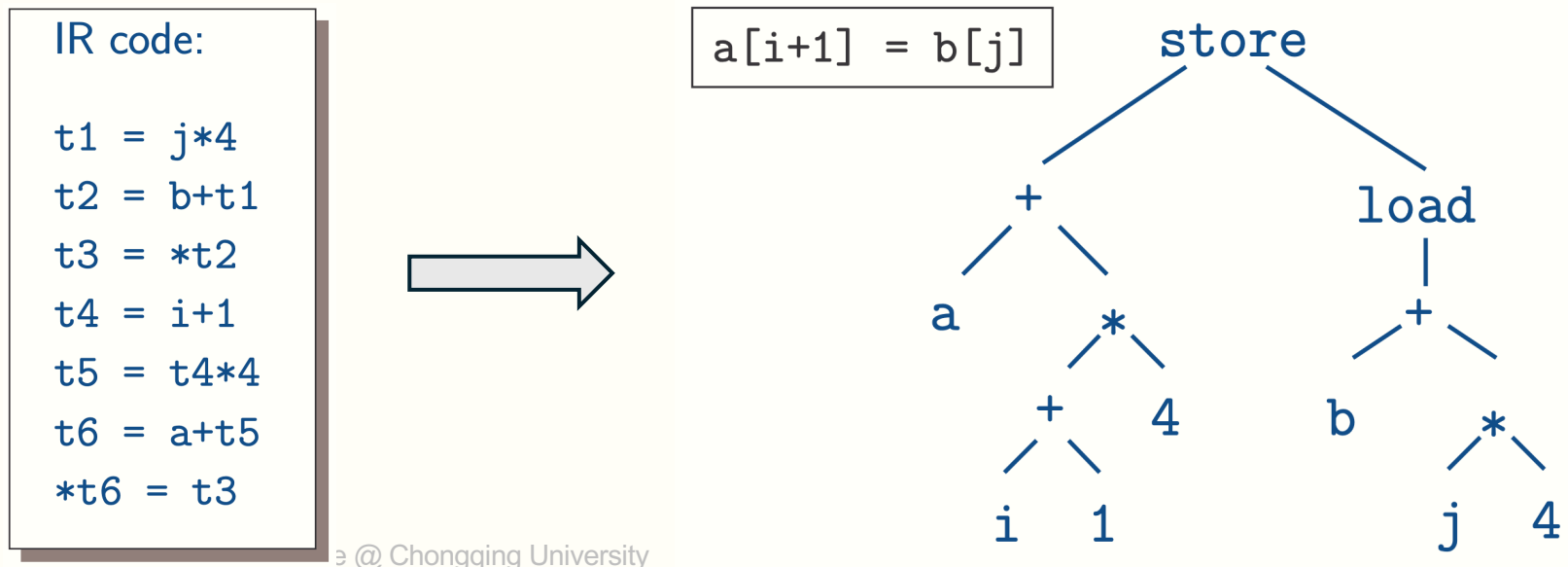


A better solution

❑ 其他指令选择算法：只讲原理，不给实现

Tree-IR-based Instruction Selection

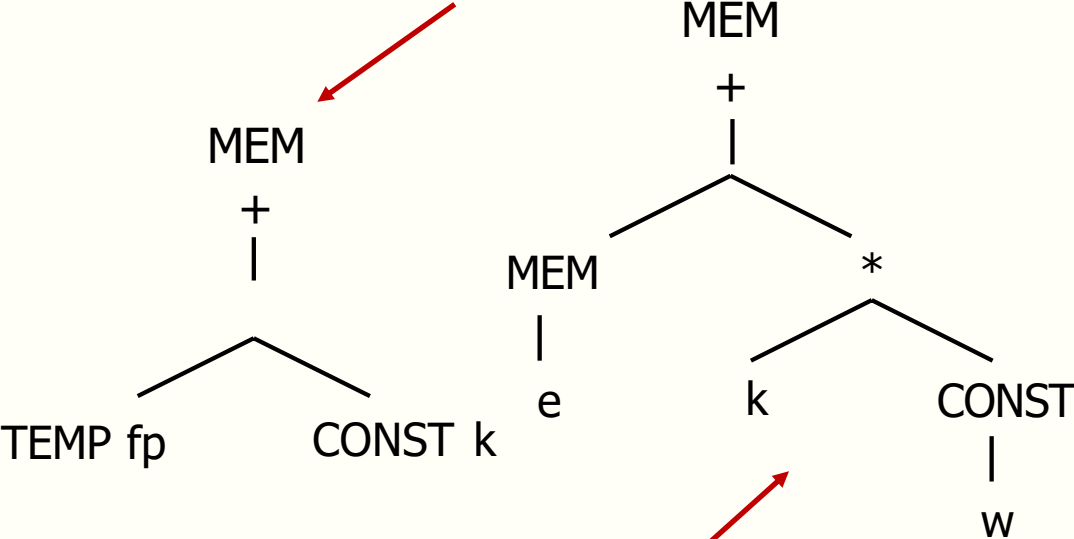
- ❑ Instruction selectors often use pattern-matching on IR trees
 - Each machine instruction can be specified as an IR tree fragment (pattern)
 - Code in each basic block is converted into an IR tree
 - Need to cover the IR tree with non-overlapping tree patterns
- ❑ Tree Representation (Tree IR)
 - Transform the IR code in a Basic Block to the Tree IR
 - Example: **a[i+1] = b[j]**



Instruction Tree Patterns

Examples:

● 访问Stack k: 访问相对于帧指针 **fp** 处于已知偏移量 **k** 处的局部变量

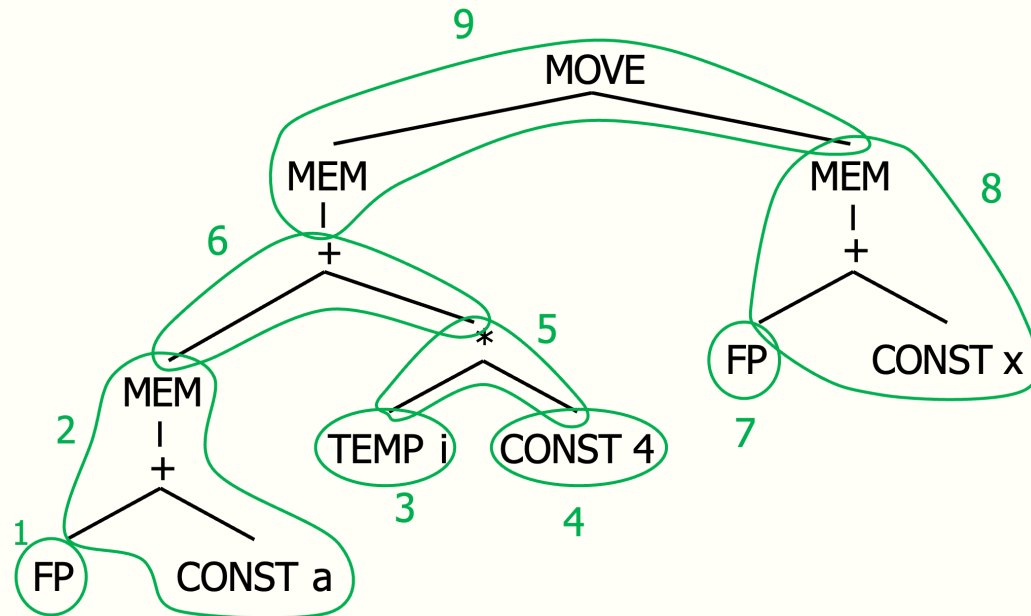


● 访问数组元素 $e[k]$, 其中每个元素占用 w 个存储位置

Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i \quad r_j + r_k$	$\begin{array}{c} + \\ \swarrow \quad \searrow \end{array}$
MUL	$r_i \quad r_j \times r_k$	$\begin{array}{c} * \\ \swarrow \quad \searrow \end{array}$
SUB	$r_i \quad r_j - r_k$	$\begin{array}{c} - \\ \swarrow \quad \searrow \end{array}$
DIV	$r_i \quad r_j / r_k$	$\begin{array}{c} / \\ \swarrow \quad \searrow \end{array}$
ADDI	$r_i \quad r_j + c$	$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
SUBI	$r_i \quad r_j - c$	$\begin{array}{c} - \\ \swarrow \quad \searrow \\ \text{CONST} \end{array}$
LOAD	$r_i \quad M[r_j + c]$	$\begin{array}{c} \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\ \quad \quad \quad \\ + \quad + \quad \text{CONST} \quad \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$
STORE	$M[r_j + c] \quad r_i$	$\begin{array}{c} \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \quad \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\ \quad \quad \quad \\ + \quad + \quad \text{CONST} \quad \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$
MOVEM	$M[r_j] \quad M[r_i]$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\ \quad \end{array}$

Tree Pattern Matching

□ A tiling for $a[i] := x$



- A **tiling** is a set of **<node,op>** pairs,
- **op** can implement the subtree at **node**
 - **Every node** is covered
 - Any two tiles overlap **at most one node**

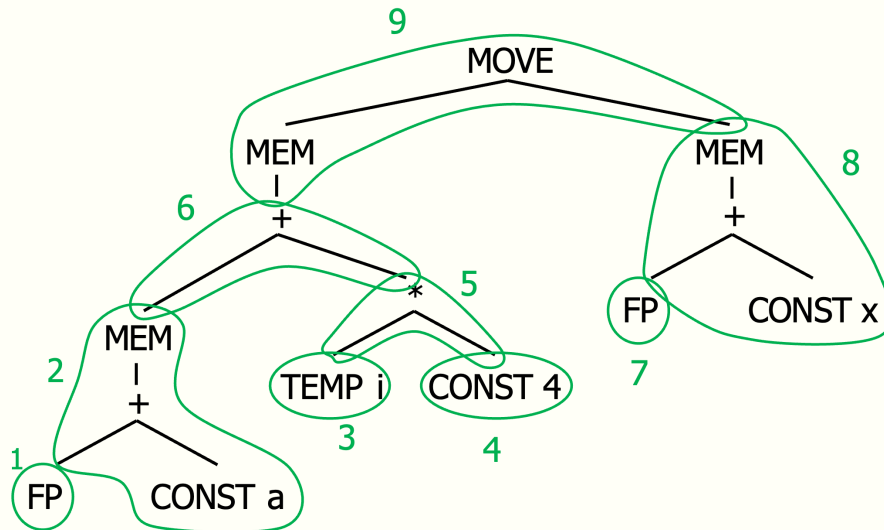
Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i \quad r_j + r_k$	$\begin{array}{c} + \\ \swarrow \quad \searrow \end{array}$
MUL	$r_i \quad r_j \times r_k$	$\begin{array}{c} * \\ \swarrow \quad \searrow \end{array}$
SUB	$r_i \quad r_j - r_k$	$\begin{array}{c} - \\ \swarrow \quad \searrow \end{array}$
DIV	$r_i \quad r_j / r_k$	$\begin{array}{c} / \\ \swarrow \quad \searrow \end{array}$
ADDI	$r_i \quad r_j + c$	$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$ $\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \end{array}$ CONST
SUBI	$r_i \quad r_j - c$	$\begin{array}{c} - \\ \swarrow \quad \searrow \\ \text{CONST} \end{array}$
LOAD	$r_i \quad M[r_j + c]$	$\begin{array}{c} \text{MEM} \\ \\ + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$ $\begin{array}{c} \text{MEM} \\ \\ + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$ $\begin{array}{c} \text{MEM} \\ \\ \text{CONST} \end{array}$ $\begin{array}{c} \text{MEM} \\ \end{array}$
STORE	$M[r_j + c] \quad r_i$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\ \quad \\ + \quad + \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$ $\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\ \quad \\ + \quad + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$ $\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\ \quad \\ + \quad + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$ $\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\ \quad \\ + \quad + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
MOVEM	$M[r_j] \quad M[r_i]$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\ \quad \end{array}$

Code Generating

□ Given a tiled tree, generate code:

- **Postorder** tree traversal with node-dependent child order.
- Child order depends on the node
- Emit each tile's code sequence **in order**
- Use the **same register** to connect tie boundaries

Tree for $a[i] := x$



2. LOAD $r1 \leftarrow M[fp+a_{off}]$
 4. ADDI $r2 \leftarrow 4 + r0$
 5. MUL $r2 \leftarrow r2 * r_i$
 6. ADD $r1 \leftarrow r1 + r2$
 8. LOAD $r2 \leftarrow M[fp+x_{off}]$
 9. STORE $M[r1+0] \leftarrow r2$

Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i \quad r_j + r_k$	$\begin{array}{c} + \\ \swarrow \quad \searrow \end{array}$
MUL	$r_i \quad r_j \times r_k$	$\begin{array}{c} * \\ \swarrow \quad \searrow \end{array}$
SUB	$r_i \quad r_j \quad r_k$	$\begin{array}{c} - \\ \swarrow \quad \searrow \end{array}$
DIV	$r_i \quad r_j / r_k$	$\begin{array}{c} / \\ \swarrow \quad \searrow \end{array}$
ADDI	$r_i \quad r_j + c$	$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
SUBI	$r_i \quad r_j \quad c$	$\begin{array}{c} - \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
LOAD	$r_i \quad M[r_j + c]$	$\begin{array}{c} \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\ \quad \quad \quad \\ + \quad + \quad \text{CONST} \quad \text{CONST} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$
STORE	$M[r_j + c] \quad r_i$	$\begin{array}{c} \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\ \quad \quad \quad \\ + \quad + \quad \text{CONST} \quad \text{CONST} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$
MOVEM	$M[r_j] \quad M[r_i]$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\ \quad \end{array}$

Tiling choices

❑ In general, for any given tree, many tilings are possible

- each resulting in a different instruction sequence
- E.g., “a[i+1] = b[j]” in x86

❑ What is the best tiling?

- Fastest, Smallest, or Minimize power consumption, ...

❑ We want the “**lowest cost**” tiling

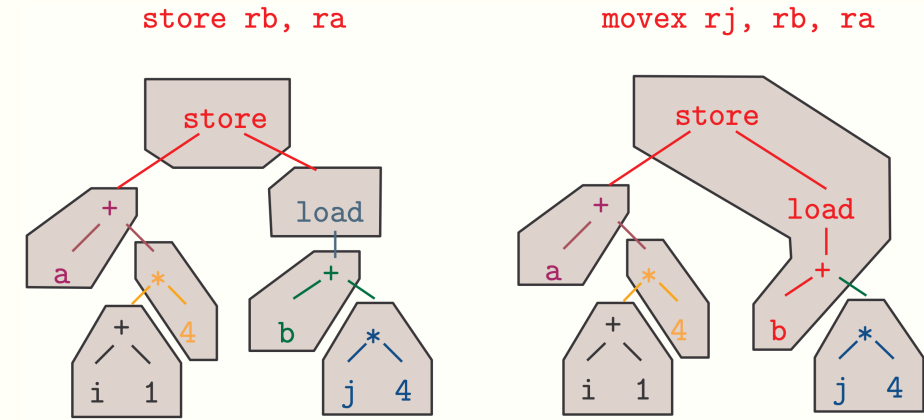
- Usually, the shortest sequence
- but can also take into account the cost/delay of each instruction

❑ **Locally Optimal tiling**: no two adjacent tiles can be combined into one tile of lower cost

- Maximal munch algorithm (a simple greedy algorithm)
 - Top-down tree walk and find the largest tile that fits each node

❑ **Optimum tiling**: lowest-cost tiling

- Dynamic programming algorithm (TOPLAS’1989)
 - Assign costs to each node in the tree, where **cost = cost of individual node + subtree costs**
 - Try all possible combinations bottom-up and pick the cheapest



Locally Optimal tiling: Maximal munch Algorithm

❑ Idea: choose the **largest** pattern with the **lowest** cost

- 优先选匹配范围最大的pattern, 多个候选则选代价最小的
- start from the root, then repeat for each subtree
- Works well for current instruction-set architectures

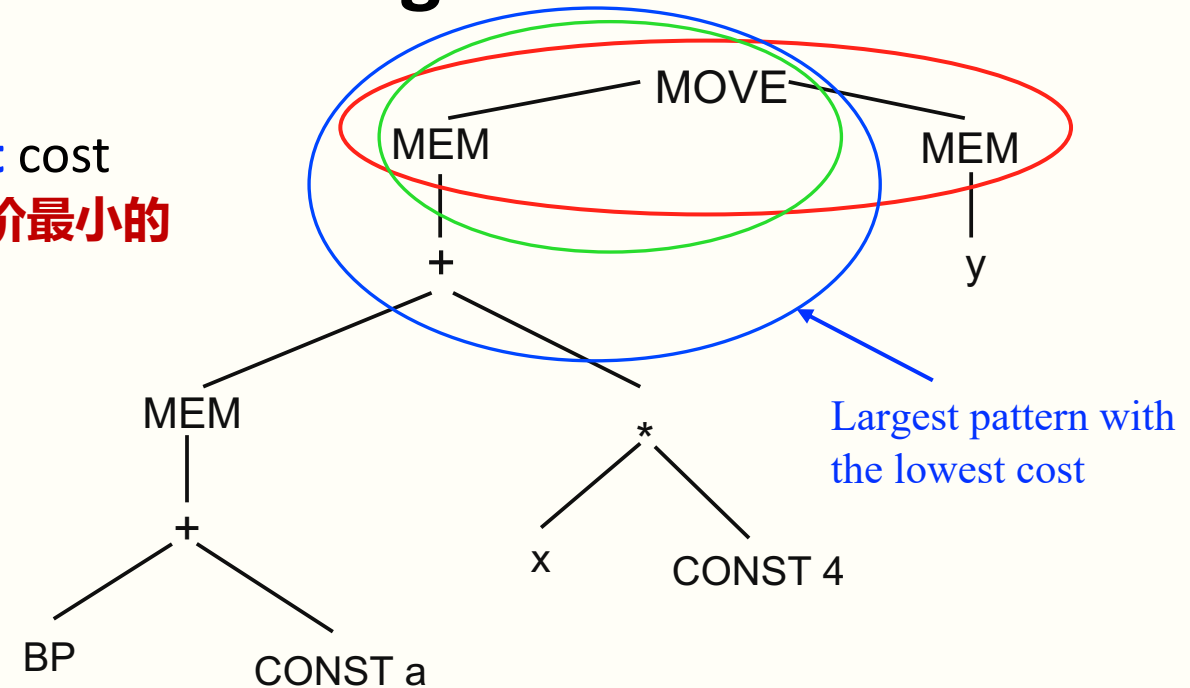
```
// tiles ordered from largest to smallest cost
List<Pattern> tiles;

Temp munchExpr (Tree.Exp e) {
  foreach (p : tiles)
    if (matches(p, e)) { pattern = p; break; }

  // wildcard(pattern, e) returns the list of
  // subexpressions of e matched to wildcards
  foreach (e_i : wildcard (pattern, e))
    recursively invoke temp_i = munchExpr (e_i)

  emit INS using temp_i as arguments
  putting result into new temp_0

  return temp_0
}
```



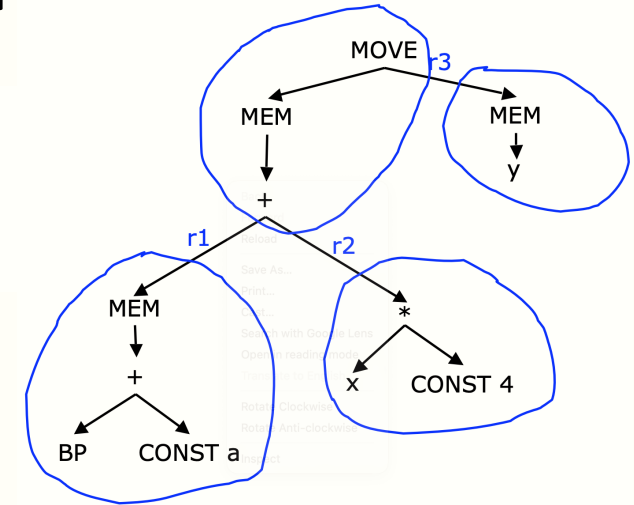
IR pattern	code	cost
MOVE (MEM (BINOP (PLUS, i, j)), k)	movl k, (i, j)	1
MOVE (MEM (i), j)	movl j, (i)	1
MOVE (MEM (i), MEM (j))	movl (j), t movl t, (i)	2

Example: Maximal munch is not optimum

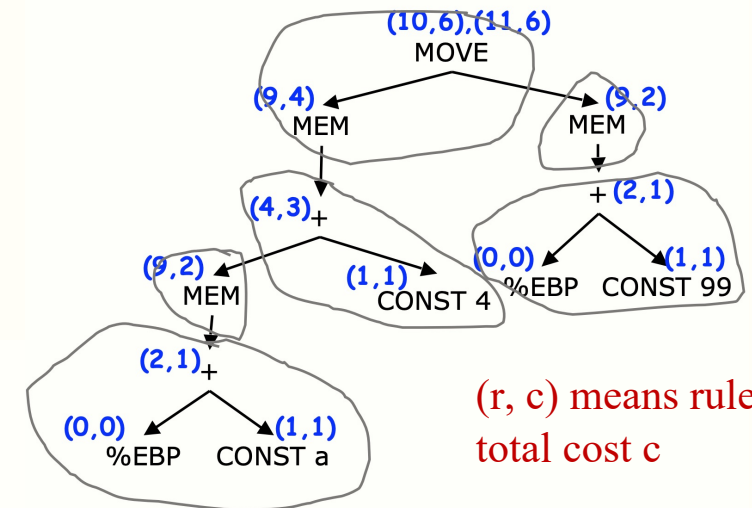
□ Tiling an IR tree with new rules: $\mathbf{a}[\mathbf{x}] = * \mathbf{y}$

Rule #	IR pattern	cost
0	TEMP t	0
1	CONST c	1
2	BINOP (PLUS, i, j)	1
3	BINOP (TIMES, i, j)	2
4	BINOP (PLUS, i, CONST c)	1
5	MEM (BINOP (PLUS, i, CONST c))	3
6	MOVE (MEM (BINOP (PLUS, i, j)), k)	3
7	BINOP (TIMES, i, CONST c)	1
8	BINOP (TIMES, i, CONST c)	2
9	MEM (i)	1
10	MOVE (MEM (i), j)	1
11	MOVE (MEM (i), MEM (j))	2

If c is 1, 2, or 4



A tiling using maximal munch, not optimum



A better tiling, which is optimum

Optimum Instruction Selection: Tree-based + DP

- ❑ Find an optimum tiling of the IR tree based on the optimum tiling of each of subtrees
- ❑ Use Dynamic Programming (DP) to avoid unnecessary re-computation of subtree costs
 - works in two passes: bottom-up **labeling** pass, and top-down **emitter** pass
- ❑ The bottom-up **labeling** pass: choose a tile with the minimum total cost for current root
 - Given a tile, cost of each subtree s_j (c_j) has already been computed
 - For each tile t of cost c that matches at node n , the cost of matching t is:
$$c_t + \sum_{\text{all leaves } i \text{ of } t} c_i$$
- ❑ The top-down **emitter** pass: emit the assembler code

```
void matchExpr (Tree.Exp e) {  
    for (Tree.Exp kid : e.kids())  
        matchExpr (kid);  
  
    cost = INFINITY  
    for each pattern P_i  
        if (P_i.matches (e)) {  
            cost_i = cost(P_i)  
                + sum ((wildcard (P_i, e)).mincost)  
            if (cost_i < cost) { cost = cost_i; choice = i; }  
        }  
    e.matched = P_{choice}  
    e.mincost = cost  
}
```

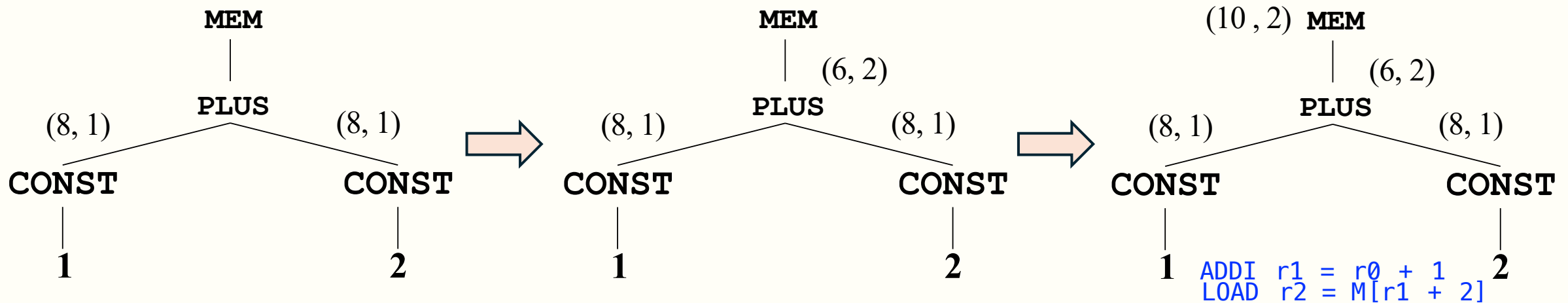
Labeling Pass

```
Temp emission (Tree.Exp e) {  
    foreach (e_i : wildcard (e.matched, e)) {  
        temp_i = emission (e_i)  
    }  
  
    emit INS using temp_i as arguments  
        putting result into new temp_0  
  
    return temp_0  
}
```

Emitter Pass

Optimum Instruction Selection: Example 1

(a, b) : a is the corresponding pattern number, b is the minimum cost.



Consider the **PLUS** node:

Pattern	Cost	Leaves Cost	Total
(2) PLUS(e1, e2)	1	2	3
(6) PLUS(CONST(c), e1)	1	1	2
(7) PLUS(e1, CONST(c))	1	1	2

Consider the **MEM** node:

Pattern	Cost	Leaves Cost	Total
(13) MEM(e1)	1	2	3
(10) MEM(PLUS(e1, CONST(c)))	1	1	2
(11) MEM(PLUS(CONST(c), e1))	1	1	2

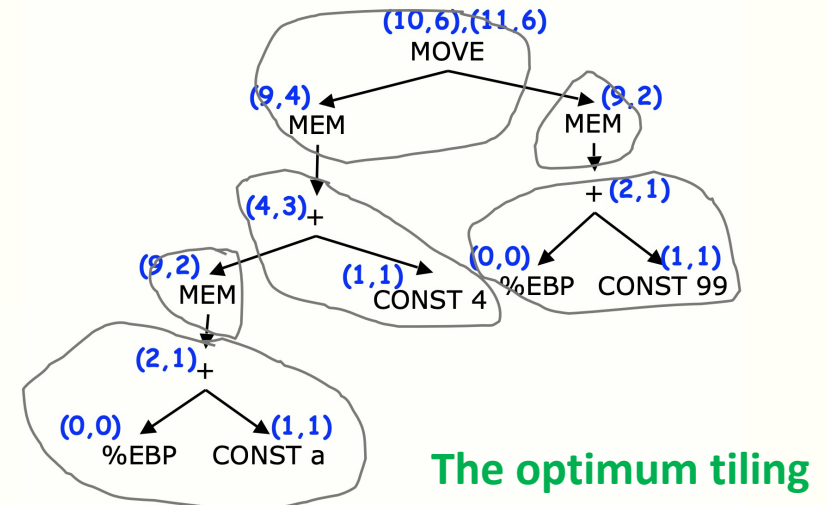
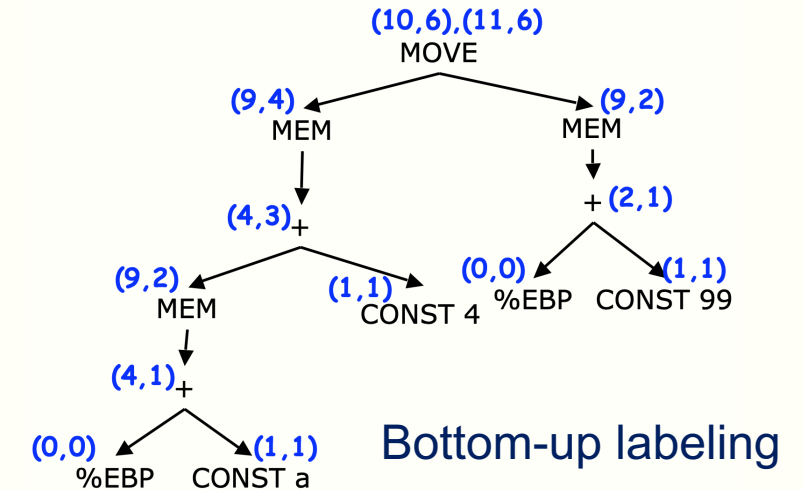
Optimum Instruction Selection: Example 2

□ Tiling an IR tree with new rules: $a[x] = *y$

(r, c) means rule r, total cost c

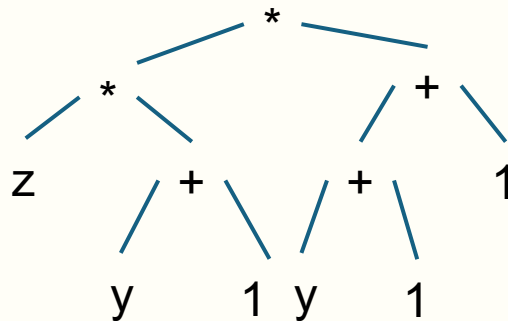
Rule #	IR pattern	cost
0	TEMP t	0
1	CONST c	1
2	BINOP(PLUS, i, j)	1
3	BINOP(TIMES, i, j)	2
4	BINOP(PLUS, i, CONST c)	1
5	MEM(BINOP(PLUS, i, CONST c))	3
6	MOVE(MEM(BINOP(PLUS, i, j)), k)	3
7	BINOP(TIMES, i, CONST c)	1
8	BINOP(TIMES, i, CONST c)	2
9	MEM(i)	1
10	MOVE(MEM(i), j)	1
11	MOVE(MEM(i), MEM(j))	2

If c is 1, 2, or 4



DAG-based Instruction Selection

- ❑ Tree IR cannot share computations of common sub-expressions
- ❑ **Directed Acyclic Graphs (DAG):** Common sub-expressions represented by the same node
 - one possible **solution**: split the DAG into multiple trees with shared nodes as new roots

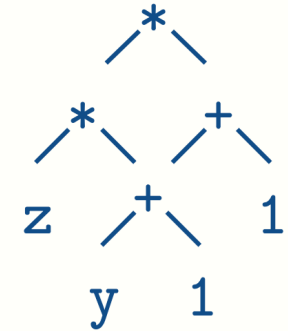


t = y+1

y = z*t

t = t+1

z = t*y



● DAG-based Instruction Selection

❑ How to construct a DAG?

- **Input:** a sequence of IR instructions in a basic block
- **Output:** an expression DAG for the block
- **Value Numbering Algorithm:** build DAG bottom up
 - Label each DAG node with a variable holding its value
 - Use a distinct variable for each value: t_0, t_1, t_2 , etc

index[v] = 0 for each variable v

for each instruction I (in the order they appear)

for each $v \in \text{use}[I]$, with $n = \text{index}[v]$

if node v_n doesn't exist create node v_n

Create node for instruction I, with children $\{ v_n \mid v \in \text{use}[I] \}$

for each $v \in \text{def}[I]$, index[v] = index[v] + 1

if I is of the form $x = \dots$ and $n = \text{index}[x]$

label the new node with x_n

SSA constructing Algorithm in a Basic Block

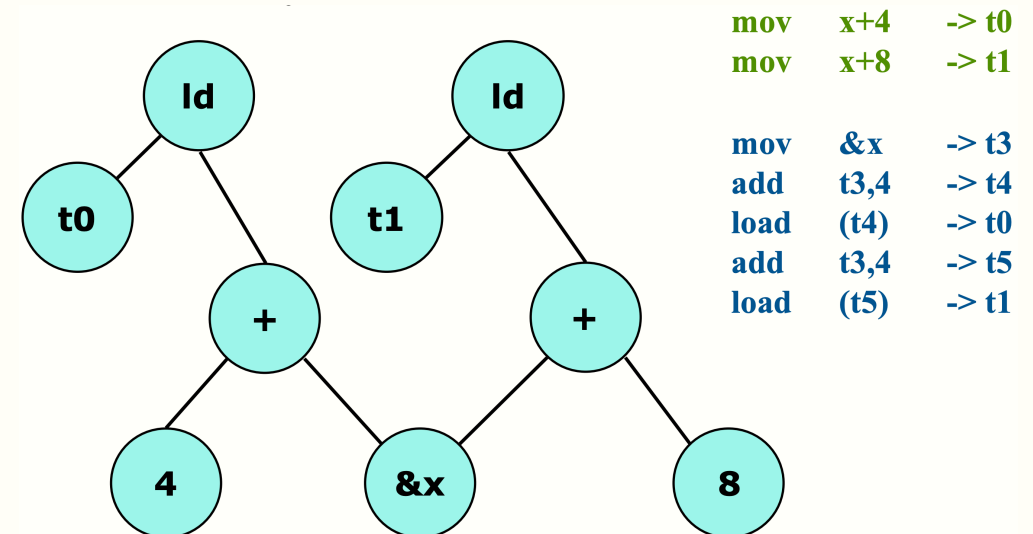
DAG-based Instruction Selection

❑ Computing a minimal cost covering (with tree patterns) for DAGs

- NP-complete [Proebsting, 98] but often works in practice
- Translate low-level (three-address) IR code into a DAG representation
- Then find a good tiling of the DAG similar to Tree-IR-based IS
 - Maximal munch algorithm
 - The dynamic programming (DP) algorithm

The dynamic programming algorithm for trees is OK as a heuristic for regular processor architectures. The algorithm for trees may create optimal results for DAGs for special tree grammars (usually for regular register sets like RISC rather than x86).

❑ Tiling a DAG: how would you tile a DAG?



Instruction Selection: Evolutionary Path

❑ Tree IR + Tree Tiling:

- An optimal algorithm exists, very simple, suitable for teaching
- For a single basic block, it can not share a common sub-expression

❑ DAG IR + DAG Tiling:

- E.g., LLVM SelectionDAG, allows the sharing of common sub-expressions
- But still limited to a basic block and relies on the CFG for global selection
- SSA IR => DAG + DAG Tiling:
 - Allow global optimisations, need to reconstruct DAG

❑ SSA IR => Machine IR:

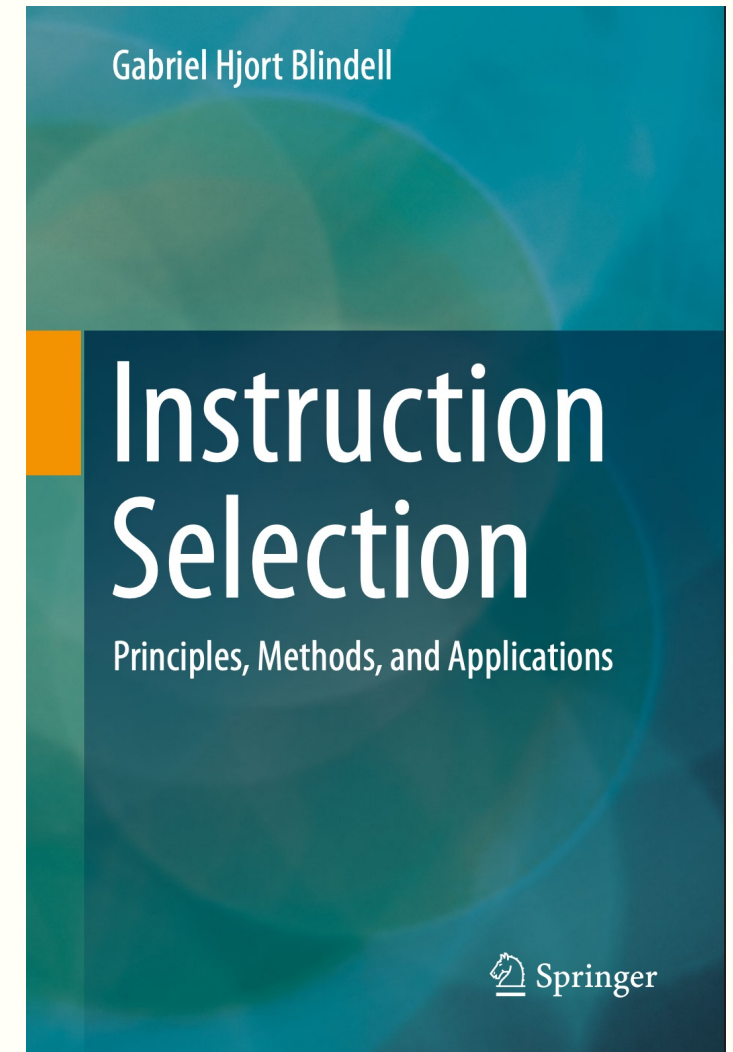
- no-longer constructing a DAG
- Support across basic-block selection, with global information
- In process in LLVM (GlobalISel)

❑ Sea-of-Nodes:

- Used in HotSpot C2, GraalVM, Suit for JIT

❑ Still a valuable research area:

- Interested students are encouraged to read this book:
 - <https://link.springer.com/content/pdf/10.1007/978-3-319-34019-7.pdf>



References

- ❑ A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, “Code Generation Using Tree Matching and Dynamic Programming,” *ACM Transactions on Programming Languages and Systems*, 11(4), 491–516, 1989. <https://dl.acm.org/doi/10.1145/69558.75700>
- ❑ Koes, David Ryan, and Seth Copen Goldstein. "Near-optimal instruction selection on DAGs." *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 2008. <https://dl.acm.org/doi/abs/10.1145/1356058.1356065>
- ❑ Blindell, Hjort. *Instruction selection*. Cham, Switzerland: Springer International Publishing, 2016. <https://link.springer.com/content/pdf/10.1007/978-3-319-34019-7.pdf>

作业：指令选择练习

□熟悉MIR，写一个MIR program

- 用 OCaml 构造 MIR 程序，实现斐波那契函数

□将IR中的GEP指令翻译到MIR：

```
insn ::= ... | getelementptr t* %val, t1 idx1, t2 idx2 ,...
```

- lower_gep state (dst_uid, elem_ty, base, indices)

□Maximal munch和动态规划算法练习

- 用Maximal munch和动态规划算法，给出如下IR的tiling划分
- 并手动生成其RISC-V汇编

```
%n2 = add i64 %a, 4          ; PLUS(n3, CONST 4)
%n6 = add i64 %c, 2          ; PLUS(n7, CONST 2)
%n5 = add i64 %n6, %d        ; PLUS(n6, n9)
store i64 %n5, ptr %n2      ; MOVE(MEM(n2), n5)
```