Instruction selection

Simple approach:

- Macro-expand each IR tuple/subtree into machine instructions
- Expanding tuples/subtrees independently ⇒ poor quality code
- Sometimes mapping is many-to-one
- "Maximal munch": works reasonably well with RISC

Other approaches:

- Model target machine *state* as IR is expanded (*interpretive code generation*)

Register and temporary management

Temporaries hold data values relevant to current computation:

- Usually registers
- May be in-memory *storage* temporaries in local stack frame

Register allocation: assign registers to temporaries

- Limited number of hard registers
  ⇒ some temporaries may need to be allocated to storage
  - assume a *pseudo-register* for each temporary
  - register allocator chooses temporaries to spill
  - allocator generates corresponding mapping
  - allocator inserts code to spill/restore pseudo-registers to/from storage as necessary

We will deal with register allocation *after* instruction selection

Tree patterns

- Express each machine instruction as fragment of IR tree: a *tree pattern*
- Instruction selection means *tiling* IR tree with minimal set of tree patterns

MIPS tree patterns

Notation:

r_i	register i
Rd	destination register
Rs	source register
Rb	base register
I	32-bit immediate
I_{16}	16-bit immediate
label	code label

Addressing modes:

- register: R
- indexed: I_{16}(Rb)
- immediate: I_{16}

MIPS tree patterns

—	r_i			TEMP
—	r_0			CONST 0
li	Rd	I		CONST
la	Rd	label		NAME
move	Rd	Rs		MOVE(●, ●)
add	Rd	Rs$_1$	Rs$_2$	+(●, ●)
	Rd	Rs$_1$	I_{16}	+(●, CONST$_{16}$), +(CONST$_{16}$, ●)
mulo	Rd	Rs$_1$	Rs$_2$	×(●, ●)
	Rd	Rs	I_{16}	×(●, CONST$_{16}$), ×(CONST$_{16}$, ●)
and	Rd	Rs$_1$	Rs$_2$	AND(●, ●)
	Rd	Rs$_1$	I_{16}	AND(●, CONST$_{16}$), AND(CONST$_{16}$, ●)
or	Rd	Rs$_1$	Rs$_2$	OR(●, ●)
	Rd	Rs$_1$	I_{16}	OR(●, CONST$_{16}$), OR(CONST$_{16}$, ●)
xor	Rd	Rs$_1$	Rs$_2$	XOR(●, ●)
	Rd	Rs$_1$	I_{16}	XOR(●, CONST$_{16}$), XOR(CONST$_{16}$, ●)
sub	Rd	Rs$_1$	Rs$_2$	−(●, ●)
	Rd	Rs	I_{16}	−(●, CONST$_{16}$)
div	Rd	Rs$_1$	Rs$_2$	/(●, ●)
	Rd	Rs	I_{16}	/(●, CONST$_{16}$)
srl	Rd	Rs$_1$	Rs$_2$	RSHIFT(●, ●)
	Rd	Rs	I_{16}	RSHIFT(●, CONST$_{16}$)
sll	Rd	Rs$_1$	Rs$_2$	LSHIFT(●, ●)
	Rd	Rs	I_{16}	LSHIFT(●, CONST$_{16}$)
	Rd	Rs	I_{16}	×(●, CONST$_{2^k}$)
sra	Rd	Rs$_1$	Rs$_2$	ARSHIFT(●, ●)
	Rd	Rs	I_{16}	ARSHIFT(●, CONST$_{16}$)
	Rd	Rs	I_{16}	/(●, CONST$_{2^k}$)
lw	Rd	I_{16}(Rb)		MEM(+(●, CONST$_{16}$)), MEM(+(CONST$_{16}$, ●)), MEM(CONST$_{16}$), MEM(●)

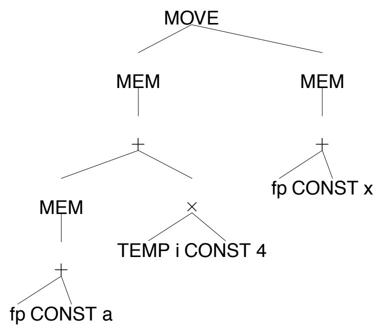
MIPS tree patterns

sw	Rs	I_{16}(Rb)		MOVE(MEM(+(●, CONST$_{16}$)), ●), MOVE(MEM(+(CONST$_{16}$, ●)), ●), MOVE(MEM(CONST$_{16}$), ●), MOVE(MEM(●), ●)
b	label			JUMP(NAME, [●])
jr	Rs			JUMP(●, [●])
beq	Rs$_1$	Rs$_2$	label	CJUMP(EQ, ●, ●, label, ●)
	Rs$_1$	I_{16}	label	CJUMP(EQ, ●, CONST$_{16}$, label, ●) CJUMP(EQ, CONST$_{16}$, ●, label, ●)
bne	Rs$_1$	Rs$_2$	label	CJUMP(NE, ●, ●, label, ●)
	Rs$_1$	I_{16}	label	CJUMP(NE, ●, CONST$_{16}$, label, ●) CJUMP(NE, CONST$_{16}$, ●, label, ●)
blt	Rs$_1$	Rs$_2$	label	CJUMP(LT, ●, ●, label, ●)
	Rs$_1$	I_{16}	label	CJUMP(LT, ●, CONST$_{16}$, label, ●)
bgt	Rs$_1$	Rs$_2$	label	CJUMP(GT, ●, ●, label, ●)
	Rs$_1$	I_{16}	label	CJUMP(GT, ●, CONST$_{16}$, label, ●)
ble	Rs$_1$	Rs$_2$	label	CJUMP(LE, ●, ●, label, ●)
	Rs$_1$	I_{16}	label	CJUMP(LE, ●, CONST$_{16}$, label, ●)
bge	Rs$_1$	Rs$_2$	label	CJUMP(GE, ●, ●, label, ●)
	Rs$_1$	I_{16}	label	CJUMP(GE, ●, CONST$_{16}$, label, ●)
bltu	Rs$_1$	Rs$_2$	label	CJUMP(ULT, ●, ●, label, ●)
	Rs$_1$	I_{16}	label	CJUMP(ULT, ●, CONST$_{16}$, label, ●)
bleu	Rs$_1$	Rs$_2$	label	CJUMP(ULE, ●, ●, label, ●)
	Rs$_1$	I_{16}	label	CJUMP(ULE, ●, CONST$_{16}$, label, ●)
bgtu	Rs$_1$	Rs$_2$	label	CJUMP(UGT, ●, ●, label, ●)
	Rs$_1$	I_{16}	label	CJUMP(UGT, ●, CONST$_{16}$, label, ●)
bgeu	Rs$_1$	Rs$_2$	label	CJUMP(UGE, ●, ●, label, ●)
	Rs$_1$	I_{16}	label	CJUMP(UGE, ●, CONST$_{16}$, label, ●)
jal	label			CALL(NAME, [●])
label:				LABEL

Tiling

- *Tiles* are a set of tree patterns for the target machine
- Goal is to cover the IR tree with nonoverlapping tiles

e.g., a[i] := x



```
lw   r1 a($fp)     | add r1 $fp a
sll  r2 ri      2  | lw  r1 (r1)
add  r1 r1      r2 | sll r2 ri  2
lw   r2 x($fp)     | add r1 r1  r2
sw   r2 (r1)       | add r2 $fp x
                   | lw  r2 (r2)
                   | sw  r2 (r1)
```

Optimal and optimum tilings

Optimum tiling: least cost instruction sequence

- shortest
- fewest cycles

Optimum tiling has tiles whose costs sum to lowest possible value

Optimal: no two adjacent tiles combine into single tile of lower cost

optimum $\Rightarrow$ optimal
optimal $\not\Rightarrow$ optimum

CISC instructions have complex tiles $\Rightarrow$ optimal $\not\approx$ optimum
RISC instructions have small tiles $\Rightarrow$ optimal $\approx$ optimum

Optimal tiling

Maximal "munch":

1. Start at root of tree
2. Tile root with largest tile that fits
3. Repeat for each subtree

Optimum tiling

Dynamic programming

- Assign a cost to every tree node: sum of instruction costs of best tiling for that node (including best tilings for children)

Example:

$$\begin{tikzcd}
 & \text{MEM} \arrow[d, dash] & \\
 & + \arrow[dl, dash] \arrow[dr, dash] & \\
\text{CONST 1} & & \text{CONST 2}
\end{tikzcd}$$

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
$+$(•, •)	add	1	1+1	3
$+$(•, CONST 2)	add	1	1+0	2
$+$(CONST 1, •)	add	1	0+1	2

CISC machines

- few registers (Pentium has 6 general, SP and FP)
  allocate TEMP nodes freely, assume good register allocation
- different register classes, some operations only on certain registers (Pentium allows mul/div only on eax, high-order bits into edx)

  $$t_1 \leftarrow t_2 \times t_3 \equiv \begin{array}{ll} \text{eax} & \leftarrow t_2 \\ \text{eax} & \leftarrow \text{eax} \times t_3;\ \text{edx} \leftarrow \\ t_1 & \leftarrow \text{eax} \end{array}$$

  register allocator removes redundant moves
- 2-address instructions

  $$t_1 \leftarrow t_2 + t_3 \equiv \begin{array}{l} t_1 \leftarrow t_2 \\ t_1 \leftarrow t_1 + t_3 \end{array}$$

  register allocator removes redundant moves
- arithmetic operations can address memory
  spill phase of register allocator will handle as

  $$\begin{array}{ll} \text{eax} & \leftarrow \text{[ebp-8]} \\ \text{eax} & \leftarrow \text{eax} + \text{ecx} \\ \text{[ebp-8]} & \leftarrow \text{eax} \end{array} \equiv \text{[ebp-8]} \leftarrow \text{[ebp-8]} + \text{ecx}$$

- several memory addressing modes
- variable-length instructions
- instructions with side-effects such as "auto-increment" addressing