Software Programmable DSP Platform Analysis

Episode 4, Tuesday 19 April 2005, Ingredients

Intermediate Representation

- After initial analyses, abstract syntax tree is translated to an intermediate representation.
- Single back-end is used for several languages,
- and single front-end for various targets (important for companies like TI)
- IR is a form of a tree-like language with limited instruction set.
- Later the back-end shall translate IR to the target instruction set.

IR: Expressions

$\textsc{Const}\ i$	integer constant i
$\textsc{Name}\ n$	symbolic label n
$\textsc{Temp}\ t$	temporary (think abstract register)
$\textsc{Ope}(e_1, e_2)$	evaluate e_1, e_2, return $e_1\ \textsc{Ope}\ e_2$ $\textsc{Ope} \in \{+, -, \textsc{Xor}, *, /, \&, \vert, \gg, \ll\}$
$\textsc{Mem}(e, n)$	content of n cells at address e. Often drop n to avoid clutter
$\textsc{Call}(f, l)$	Call function at address f with arguments on list l
$\textsc{Eseq}(s, e)$	execute stmt s, evaluate expr e, return value of e.

Translating a Constant

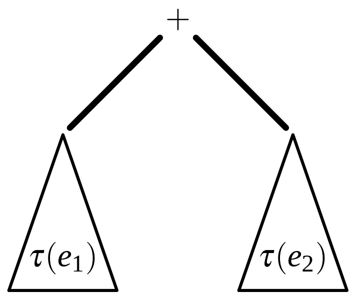
Each integer constant i is translated to $\textsc{Const}\ i$. For example:

$$\tau(1) = \textsc{Const}\ 1$$

Should we have more types of constants (for example floats), a distinct constructor for each of them should be included in the IR.

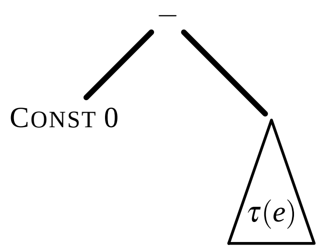
Translating Addition

$$\tau(e_1 + e_2) = +(\tau(e_1), \tau(e_2))$$



Unary Minus

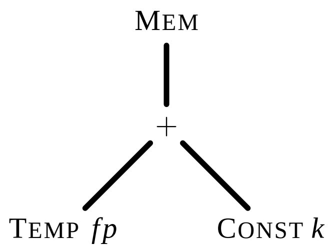
$$\tau(-e_1) = \textsc{Const}\ 0 - \tau(e_1)$$



Variable Access

A stack allocated variable v at offset k:

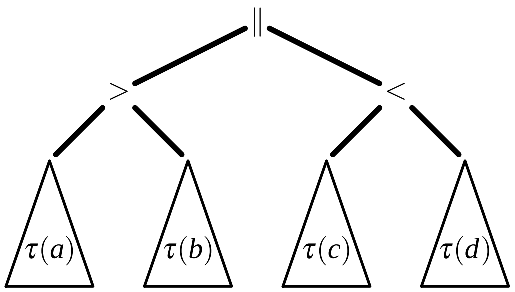
$$\textsc{Mem}(+, \textsc{Temp}\ fp, \textsc{Const}\ k)$$



- If v is allocated in register r_i then the translation is simply $\textsc{Temp}\ r_i$.
- Typically all variables that need explicit addresses would be allocated on the stack,
- and all the others in abstract registers (temporaries).
- Only at the later optimization steps abstract registers will be mapped to finite number of physical registers.

Translating Conditions (first attempt)

$$\tau(a > b \| c < d) = \|(>(\tau(a), \tau(b)), <(\tau(c), \tau(d))))$$



Does not preserve C semantics: no short circuit.
Needs control statements to achieve lazy evaluation.

IR: Statements

MOVE(TEMP t, e)	move value of e to register t
MOVE(MEM(e_1, n), e_2)	store value of e_2 in n cells at e_1
EXP e	compute value of e, discard it
JUMP e	jump to program location returned by e
CJUMP (o, e_1, e_2, t, f)	compare values of e_1, e_2 using operator o, jump to label t or f depending on the result. $o \in \{=, !=, <, >, \leq, \geq\}$
SEQ(s_1, s_2)	execute s_1 and then s_2
LABEL n	label n before next instruction

Conditions Revisitted

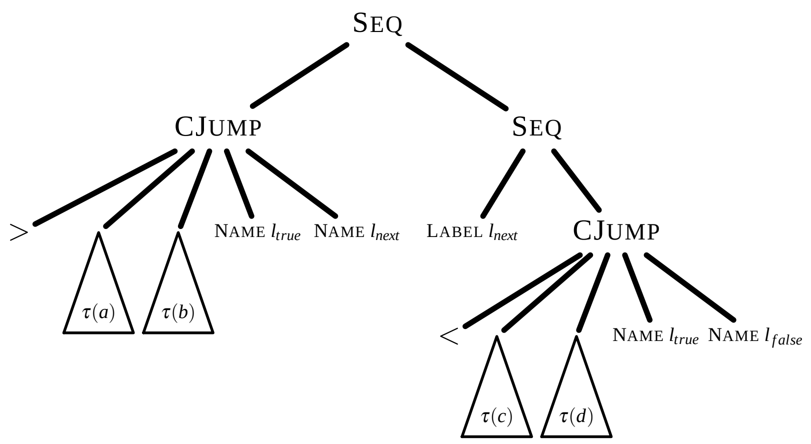
- Use conditional jump (CJUMP) to shortcut computation of disjunction.
- Only compute the right side, if the left side fails:
- Compute the left side,
- and if it is true, jump over the computation of the right operand.
- If the left side gives fall, jump to the computation of the right operand.

Let l_{true} be the label of the code to be executed if the condition is true, and l_{false} otherwise. Then:

$$\begin{aligned}\tau(a > b \| c < d) \quad = \\ \text{SEQ}(\ \text{CJUMP}(>, \tau(a), \tau(b), l_{true}, l_{next}), \\ \text{SEQ}(\text{LABEL } l_{next}, \\ \text{CJUMP}(<, \tau(c), \tau(d), l_{true}, l_{false})))\end{aligned}$$

where l_{next} is a fresh, local label.

$\tau(a > b \| c < d)$:



While Loops

A while loop: **`while`** (e) b;

Naturally expands to:

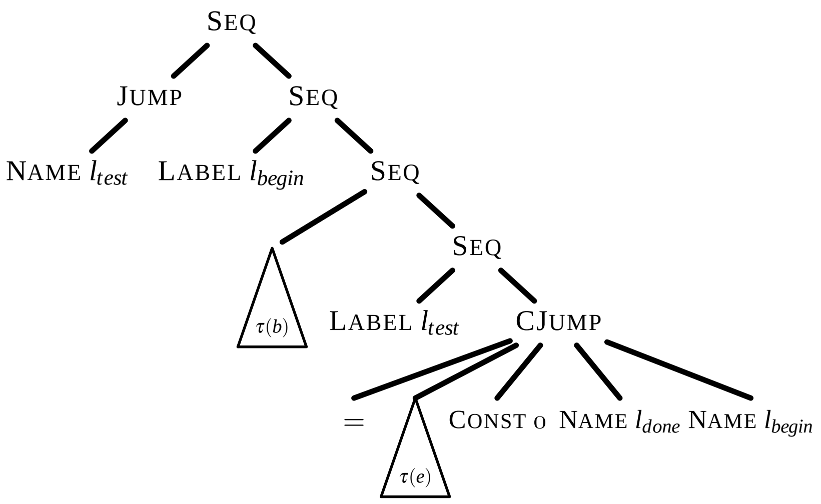
```
test:if (!e)
        goto done;
      b;
      goto test;
done:...
```

1 CJUMP per iteration
+ 1 JUMP per iteration

but more popular is:

```
      goto test;
beg: b;
test:if (e)
        goto
beg;
```

1 CJUMP per iteration
+ 1 JUMP to initialize



The rightmost variant translated to IR.

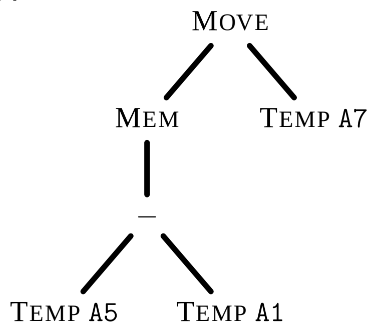
- More patterns of translation in Appel, section 7.2.
- The IR language does not have the construct for function definition (but it has calls).
- IR is suitable for representing function bodies.
- In this way platform dependent calling conventions (entry and exit code) do not pollute our IR, which should be general.
- This code is added by the compiler later on.

Instruction Selection

A node in the IR tree represents a single operation.
A target (VLIW) instruction represents many.

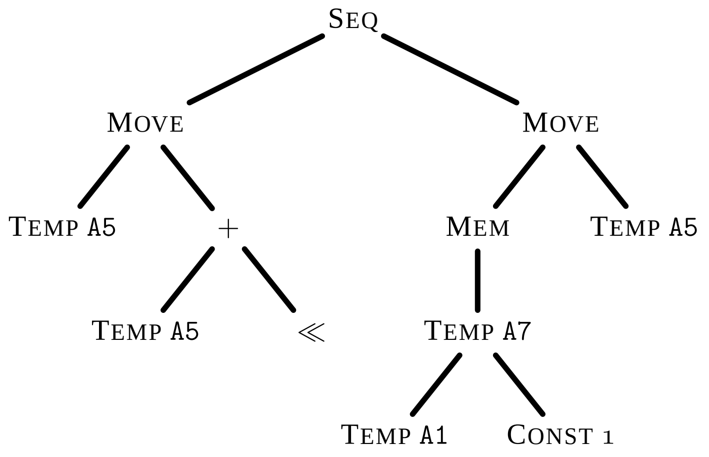
Example LDW on C67x: `LDW *-A5[A1],A7`

Corresponds (roughly) to:



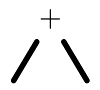
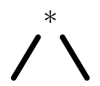
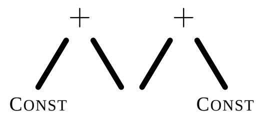
(spru189 pp. 3-68—3-71)

And `LDH *--A5[A1],A7` is even more complex

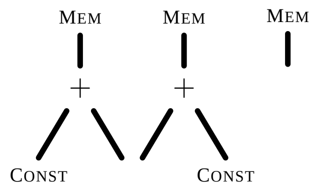


(source: spru189, pp. 3-68—3-71)

Target Instructions

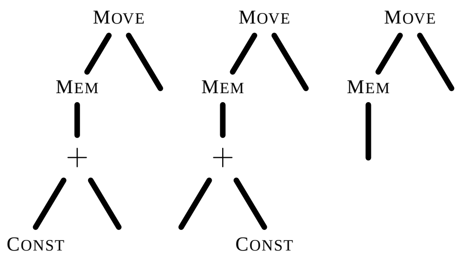
name	*semantics*	*c6xxx instr.*	*pattern*
ADD	$r_i \leftarrow r_j + r_k$	`ADD` r_j, r_k, r_i	
MUL	$r_i \leftarrow r_j * r_k$	`MPY` r_j, r_k, r_i	
ADDI	$r_i \leftarrow r_j + c$	`ADD` c, r_j, r_i	 CONST CONST

name	*semantics*	*c6xxx instr.*
LOAD	$r_i \leftarrow M[r_j + c]$	`LDW` $*r_j[c], r_i$



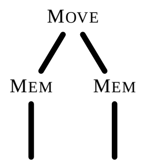
The last pattern matches for $c = 0$.

name	*semantics*	*c6xxx instr.*
STORE	$M[r_j + c] \leftarrow r_i$	`STW` $r_i, *r_j[c]$



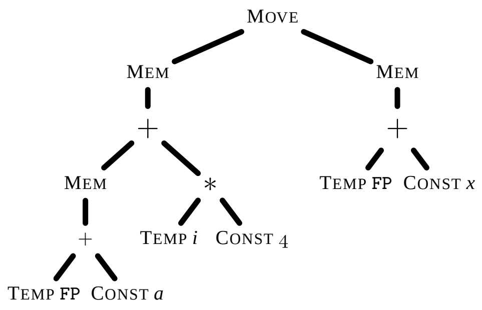
The last pattern matches for $c = 0$.

name	*semantics*	*c6xxx instr.*
MOVEM	$M[r_j] \leftarrow M[r_i]$	n/a

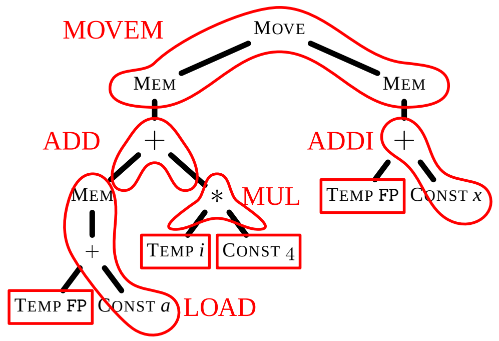


MOVEM does not seem to have a direct C6xxx counterpart, but we shall assume that we have it, for simplicity of the examples.

a[i*4] = x



Maximal Munch



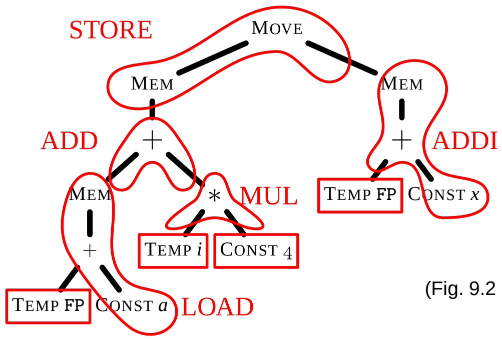
- Tile the tree with instruction patterns
- Always possible, but solutions is not unique.
- *Maximal Munch* finds the largest tile for the root
- and applies itself recursively to the subtrees.

Linearization of the Tree

- Maximal Munch did the tiling top down.
- Translation to a sequence of instructions proceeds bottom up.
- First instantiate leaves, then parents.
- The outcome:

LDW *FP[a], r_1
MPY 4, i, r_2
ADD r_1, r_2, r_3
ADDI x, FP, r_4
MOVEM $*r_1 \leftarrow *r_4$

Another Tiling of the Same Tree



(Fig. 9.2 left)

- Bigger by one instruction, but may be faster.
- Maximal Munch does not guarantee optimality.
- Optimal algorithm based on dynamic programming, Appel p. 197.