Topic 7 ½: Instruction Selection

COS 320

Compiling Techniques

Princeton University
Spring 2015

Prof. David August
Instruction Selection

- Process of finding set of machine instructions that implement operations specified in IR tree.
- Each machine instruction can be specified as an IR tree fragment → tree pattern
- Goal of instruction selection is to cover IR tree with non-overlapping tree patterns.
Our Architecture

- Load/Store architecture
- Relatively large, general purpose register file
  - Data or addresses can reside in registers (unlike Motorola 68000)
  - Each instruction can access any register (unlike x86)
- $r_0$ always contains zero.
- Each instruction has latency of one cycle.
- Execution of only one instruction per cycle.
Our Architecture

Arithmetic:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>$r_d = r_{s1} + r_{s2}$</td>
</tr>
<tr>
<td>ADDI</td>
<td>$r_d = r_s + c$</td>
</tr>
<tr>
<td>SUB</td>
<td>$r_d = r_{s1} - r_{s2}$</td>
</tr>
<tr>
<td>SUBI</td>
<td>$r_d = r_s - c$</td>
</tr>
<tr>
<td>MUL</td>
<td>$r_d = r_{s1} \times r_{s2}$</td>
</tr>
<tr>
<td>DIV</td>
<td>$r_d = r_{s1} / r_{s2}$</td>
</tr>
</tbody>
</table>

Memory:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>$r_d = M[r_s + c]$</td>
</tr>
<tr>
<td>STORE</td>
<td>$M[r_{s1} + c] = r_{s2}$</td>
</tr>
<tr>
<td>MOVEM</td>
<td>$M[r_{s1}] = M[r_{s2}]$</td>
</tr>
</tbody>
</table>
Pseudo-op - An assembly operation which does not have a corresponding machine code operation. Pseudo-ops are resolved during assembly.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>$r_d = r_s$</td>
</tr>
<tr>
<td>ADDI</td>
<td>$r_d = r_s + 0$</td>
</tr>
<tr>
<td>MOV</td>
<td>$r_d = r_s$</td>
</tr>
<tr>
<td>ADD</td>
<td>$r_d = r_{s1} + r_0$</td>
</tr>
<tr>
<td>MOVI</td>
<td>$r_d = c$</td>
</tr>
<tr>
<td>ADDI</td>
<td>$r_d = r_0 + c$</td>
</tr>
</tbody>
</table>

(Pseudo-op can also mean assembly directive, such as .align.)
# Instruction Tree Patterns

<table>
<thead>
<tr>
<th>Name</th>
<th>Effect</th>
<th>Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>$r_i$</td>
<td>TEMP 6</td>
</tr>
<tr>
<td>ADD</td>
<td>$r_i$</td>
<td>$r_j + r_k$</td>
</tr>
<tr>
<td>MUL</td>
<td>$r_i$</td>
<td>$r_j \times r_k$</td>
</tr>
<tr>
<td>SUB</td>
<td>$r_i$</td>
<td>$r_j - r_k$</td>
</tr>
<tr>
<td>DIV</td>
<td>$r_i$</td>
<td>$r_j / r_k$</td>
</tr>
<tr>
<td>ADDI</td>
<td>$r_i$</td>
<td>$r_j + c$</td>
</tr>
<tr>
<td>SUBI</td>
<td>$r_i$</td>
<td>$r_j - c$</td>
</tr>
<tr>
<td>LOAD</td>
<td>$r_i$</td>
<td>$M[r_j + c]$</td>
</tr>
<tr>
<td>Instruction</td>
<td>Pattern 1</td>
<td>Pattern 2</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>STORE</td>
<td>$M[r_j + c]$</td>
<td>$r_i$</td>
</tr>
<tr>
<td></td>
<td>MEM</td>
<td>MEM</td>
</tr>
<tr>
<td></td>
<td>$+$</td>
<td>$+$</td>
</tr>
<tr>
<td></td>
<td>CONST</td>
<td>CONST</td>
</tr>
<tr>
<td>MOVEM</td>
<td>$M[r_j]$</td>
<td>$M[r_i]$</td>
</tr>
<tr>
<td></td>
<td>MEM</td>
<td>MEM</td>
</tr>
</tbody>
</table>

Additionally, there are specific nodes labeled with numbers 13, 14, 15, and 16.
Example

\[ a[i] := x \] assuming \( i \) in register, \( a \) and \( x \) in stack frame.

\[
\text{MOVE}
\]

\[
\text{MEM} \quad \text{MEM}
\]

\[
\text{PLUS} \quad \text{PLUS}
\]

\[
\text{MEM} \quad \text{MULT} \quad \text{TEMP} \quad \text{CONST}
\]

\[
\text{PLUS} \quad \text{TEMP} \quad \text{CONST}
\]

\[
\text{MEM} \quad \text{MEM}
\]

\[
\text{PLUS} \quad \text{PLUS}
\]

\[
\text{TEMP} \quad \text{CONST}
\]

\[
\text{FP} \quad \text{offset-x}
\]

\[
\text{TEMP} \quad \text{const}
\]

\[
\text{FP} \quad \text{offset-a}
\]

\[
temp-i \quad 4
\]
Individual Node Selection

MOVE
  /   
MEM PLUS MEM
     /   
    MEM PLUS
       /   
      MEM MULT
         /   
        TEMP CONST
          /   
         FP offset-x
        /   
TEMP temp-i
  /   
TEMP CONST
 /   
FP offset-a
**Individual Node Selection**

ADDI  r1 = r0 + offset_a  
ADD  r2 = r1 + FP  
LOAD  r3 = M[r2 + 0]  

ADDI  r4 = r0 + 4  
MUL  r5 = r4 * r_i  
ADD  r6 = r3 + r5  

ADDI  r7 = r0 + offset_x  
ADD  r8 = r7 + FP  
LOAD  r9 = M[r8 + 0]  

STORE  M[r6 + 0] = r9  

9 registers, 10 instructions
Random Tiling
Random Tiling

ADDI  r1 = r0 + offset_a
ADD   r2 = r1 + FP
LOAD  r3 = M[r2 + 0]

ADDI  r4 = r0 + 4
MUL   r5 = r4 * r_i

ADD   r6 = r3 + r5

ADDI  r7 = r0 + offset_x
ADD   r8 = r7 + FP
MOVEM M[r6] = M[r8]

Saves a register (9 → 8) and an instruction (10 → 9).
Node Selection

- There exist many possible tilings - want tiling/covering that results in instruction sequence of *least cost*
  - Sequence of instructions that takes least amount of time to execute.
  - For single issue fixed-latency machine: fewest number of instructions.

- Suppose each instruction has fixed cost:
  - *Optimum Tiling*: tiles sum to lowest possible value - globally “the best”
  - *Optimal Tiling*: no two adjacent tiles can be combined into a single tile of lower cost - locally “the best”
  - Optimal instruction selection easier to implement than Optimum instruction selection.
  - Optimal is roughly equivalent to Optimum for RISC machines.
  - Optimal and Optimum are noticeably different for CISC machines.

- Instructions are not self-contained with individual costs.
Optimal Instruction Selection:
Maximal Munch

• Cover root node of IR tree with largest tile $t$ that fits (most nodes)
  – Tiles of equivalent size $\Rightarrow$ arbitrarily choose one.
• Repeat for each subtree at leaves of $t$.
• Generate assembly instructions in reverse order - instruction for tile at root emitted last.
Maximal Munch

```
MOVE
  / \  / \  /  \\
MEM  MEM
  |   |
  PLUS PLUS
  |
MEM  MULT  TEMP  CONST
  |
PLUS  TEMP  CONST  FP  offset-x
  |
TEMP Const temp-i 4
  |
FP  offset-a
```
Maximal Munch

LOAD   r3 = M[FP + offset_a]
ADDI   r4 = r0 + 4
MUL    r5 = r4 * r_i
ADD    r6 = r3 + r5
ADD    r8 = FP + offset_x
MOVEM  M[r6] = M[r8]

5 registers, 6 instructions
structure Assem = struct
  type reg = string
  type temp = Temp.temp
  type label = Temp.label

  datatype instr = OPER of
    {assem: string,
     dst: temp list,
     src: temp list,
     jump: label list list option}
    | ...
  ...
end
fun codegen(frame)(stm: Tree.stm):Assem.instr list =
  let
    val ilist = ref(nil: Assem.instr list)
    fun emit(x) = ilist := x::!ilist
    fun munchStm: Tree.stm -> unit
    fun munchExp: Tree.exp -> Temp.temp
  in
    munchStm(stm);
    rev(!ilist)
  end
fun munchStm(
    T.MOVE(T.MEM(T.BINOP(T.PLUS, e1, T.CONST(c))), e2)
) =
    emit(Assem.OPER{assem="STORE M[\'s0 + " ^
        int(c) ^ "] = \'s1\n",
        src=[munchExp(e1), munchExp(e2)],
        dst=[],
        jump=None})
| munchStm(T.MOVE(T.MEM(e1), T.MEM(e2))) =
    emit(Assem.OPER{assem="MOVEM M[\'s0 = M[\'s1]\n"
        src=[munchExp(e1), munchExp(e2)],
        dst=[],
        jump=None})
| munchStm(T.MOVE(T.MEM(e1), e2)) =
    emit(Assem.OPER{assem="STORE M[\'s0 = \'s1\n"
        src=[munchExp(e1), munchExp(e2)],
        dst=[],
        jump=None})

...
and munchExp(T.MEM(T.BINOP(T.PLUS, e1, T.CONST(c))))) =
  let
    val t = Temp.newtemp()
  in
    emit(Assem.OPER{assem="LOAD ‘d0 = M[‘s0 +" ^
      int(c) ^ "]\n",
      src=[munchExp(e1)],
      dst=[t],
      jump=NONE});

  t
end
| munchExp(T.BINOP(T.PLUS, e1, T.CONST(c))) =
|     let
|         val t = Temp.newtemp()
|     in
|         emit(Assem.OPER{assem="ADDI 'd0 = 's0 +" ^
|                       int(c) ^ "\n",
|                       src=[munchExp(e1)],
|                       dst=[t],
|                       jump=NONE});
|         t
|     end
| ...
|     munchExp(T.TEMP(t)) = t
Optimum Instruction Selection

- Find optimum solution for problem (tiling of IR tree) based on optimum solutions for each subproblem (tiling of subtrees).
- Use Dynamic Programming to avoid unnecessary recomputation of subtree costs.
- \textit{cost} assigned to every node in IR tree
  - Cost of best instruction sequence that can tile subtree rooted at node.
- Algorithm works bottom-up (Maximum Munch is top-down) - Cost of each subtree $s_j$ ($c_j$) has already been computed.
- For each tile $t$ of cost $c$ that matches at node $n$, cost of matching $t$ is:
  \[ c_t + \sum_{\text{all leaves } i \text{ of } t} c_i \]
- Tile is chosen which has minimum cost.
Optimum Instruction Selection – Example

\[
\text{MEM(\text{BINOP(PLUS, CONST(1), CONST(2)))}}
\]

\[
\text{MEM(PLUS(CONST(1), CONST(2)))}
\]

```
MEM
   
PLUS
   
CONST 1
   
CONST 2
```
Step 1: Find cost of root node
(a,b): a is minimum cost, b is corresponding pattern number

Consider PLUS node:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Cost</th>
<th>Leaves Cost</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2) PLUS(e1, e2)</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(6) PLUS(CONST(c), e1)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(7) PLUS(e1, CONST(c))</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Consider MEM node:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Cost</th>
<th>Leaves Cost</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>(13) MEM(e1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(10) MEM(PLUS(e1, CONST(c))</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(11) MEM(PLUS(CONST(c), e1))</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Step 2: Emit instructions

ADDI $r1 = r0 + 1$
LOAD $r2 = M[r1 + 2]
Optimum Instruction Selection – Big Example

```
  MOVE
   /\  \\
  MEM PLUS MEM
   | /| |
  PLUS /|
  |
  MEM
  |  |
  PLUS
  |
  MEM
  |  |
  TEMP
  |  |
  CONST
  |  |
  temp-i
  |  |
  4
  |
  FP
  |  |
  offset-a
```

FP

offset-x
Optimum Instruction Selection – Big Example

LOAD  r3 = M[FP + offset_a]
ADDI  r4 = r0 + 4
MUL   r5 = r4 * r_i
ADD   r6 = r3 + r5

LOAD  r9 = M[FP + offset_x]
STORE M[r6] = r9

5 registers, 6 instructions
Optimal tree generated by Maximum Munch is also optimum...