

Topic 7 ½ : Instruction Selection

COS 320

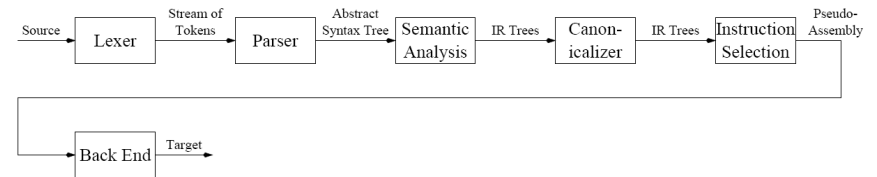
Compiling Techniques

Princeton University
Spring 2015

Prof. David August

1

Instruction Selection



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 \rightarrow *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:

ADD $r_d = r_{s1} + r_{s2}$
ADDI $r_d = r_s + c$
SUB $r_d = r_{s1} - r_{s2}$
SUBI $r_d = r_s - c$
MUL $r_d = r_{s1} * r_{s2}$
DIV $r_d = r_{s1} / r_{s2}$

Memory:

LOAD $r_d = M[r_s + c]$
STORE $M[r_{s1} + c] = r_{s2}$
MOVEM $M[r_{s1}] = M[r_{s2}]$

Pseudo-ops

Pseudo-op - An assembly operation which does not have a corresponding machine code operation. Pseudo-ops are resolved during assembly.

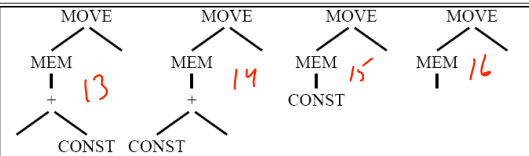
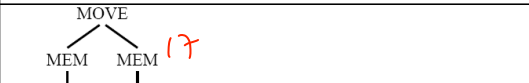
MOV	$r_d = r_s$	ADDI	$r_d = r_s + 0$
MOV	$r_d = r_s$	ADD	$r_d = r_{s1} + r_0$
MOVI	$r_d = c$	ADDI	$r_d = r_0 + c$

(Pseudo-op can also mean assembly directive, such as `.align`.)

Instruction Tree Patterns

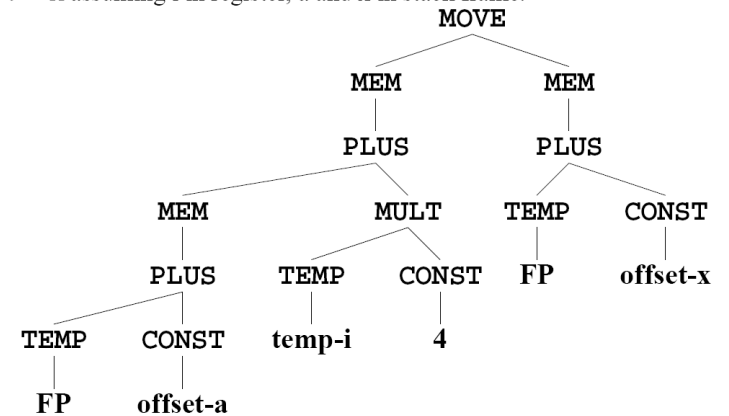
Name	Effect	Trees
—	r_i	TEMP 0
ADD	$r_i \quad r_j + r_k$	$\begin{array}{c} + \\ / \quad \backslash \\ 1 \quad 2 \end{array}$
MUL	$r_i \quad r_j \times r_k$	$\begin{array}{c} * \\ / \quad \backslash \\ 2 \quad 2 \end{array}$
SUB	$r_i \quad r_j \quad r_k$	$\begin{array}{c} - \\ / \quad \backslash \\ 2 \quad 2 \end{array}$
DIV	$r_i \quad r_j \ / \ r_k$	$\begin{array}{c} / \\ / \quad \backslash \\ 2 \quad 2 \end{array}$
ADDI	$r_i \quad r_j + c$	$\begin{array}{c} + \\ / \quad \backslash \\ \text{CONST} \quad \text{CONST} \end{array}$ 5 6 7
SUBI	$r_i \quad r_j \quad c$	$\begin{array}{c} - \\ / \quad \backslash \\ \text{CONST} \quad \text{CONST} \end{array}$ 8
LOAD	$r_i \quad M[r_j + c]$	$\begin{array}{c} \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\ \quad \quad \quad \\ + \quad + \quad \text{CONST} \quad \text{CONST} \\ / \quad \backslash \quad / \quad \backslash \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$ 9 10 11 12 2

Instruction Tree Patterns

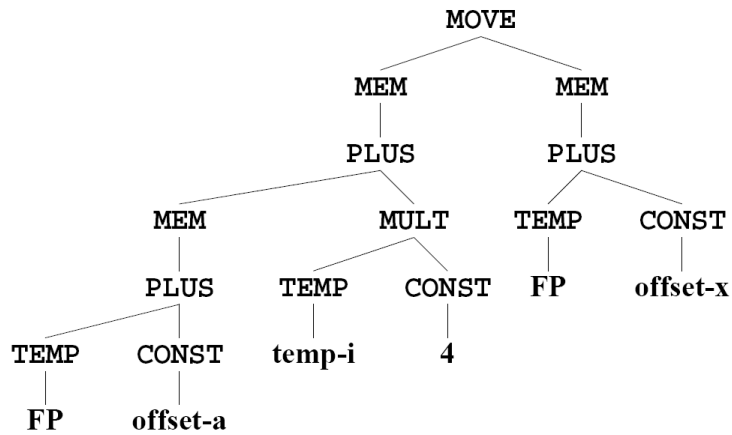
STORE	$M[r_j + c] \quad r_i$	
MOVEM	$M[r_j] \quad M[r_i]$	

Example

$a[i] := x$ assuming i in register, a and x in stack frame.



Individual Node Selection



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

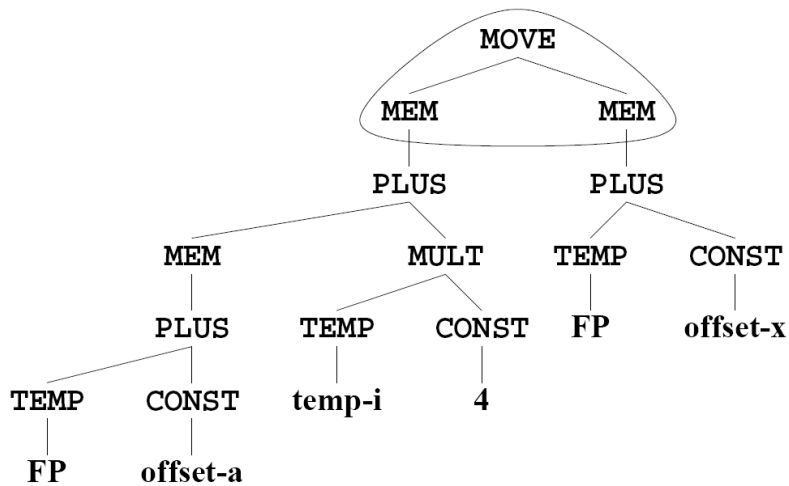
ADD r6 = r3 + r5

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).

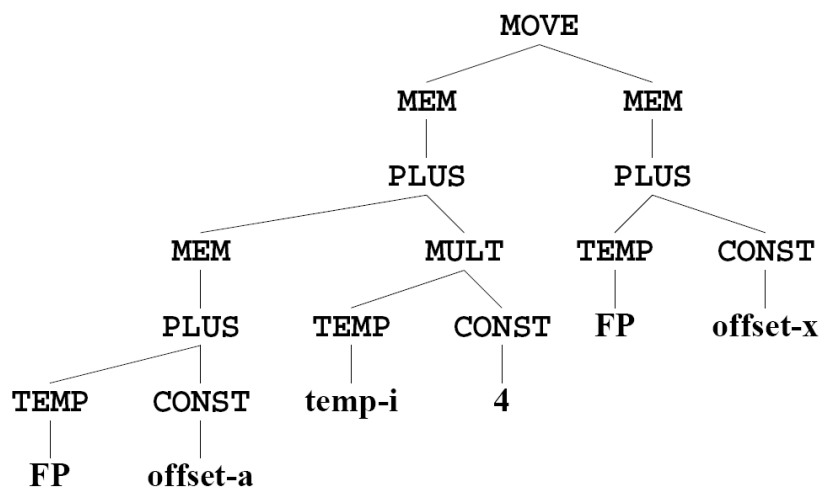
Node Selection

- 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



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

Maximal Munch



Assembly Representation

```
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 option}
  | ...
  ...
end
```

Codegen

```
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
```

Statement Munch

```
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})
  ...
```

Expression Munch

```
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
```

Expression Munch

```
| 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.
- *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:

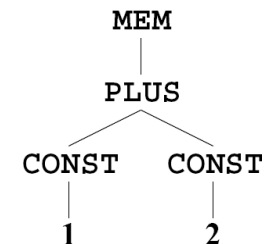
$$c_t + \sum_{\text{all leaves } i \text{ of } t} c_i$$

- Tile is chosen which has minimum cost.

Optimum Instruction Selection – Example

MEM(BINOP(PLUS, CONST(1), CONST(2)))

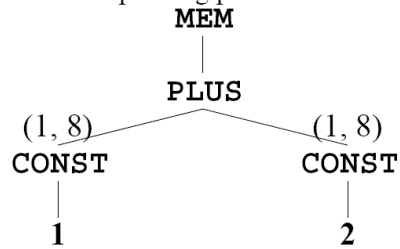
MEM(PLUS(CONST(1), CONST(2)))



Optimum Instruction Selection – Example

Step 1: Find cost of root node

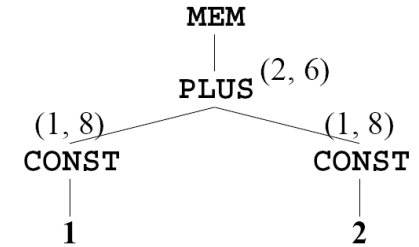
(a,b): a is minimum cost, b is corresponding pattern number



Consider 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

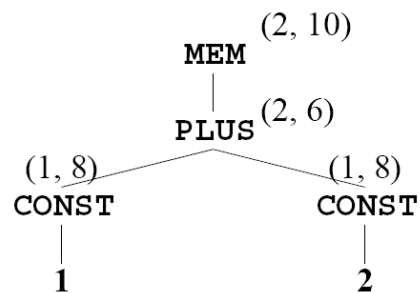
Optimum Instruction Selection – Example



Consider 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

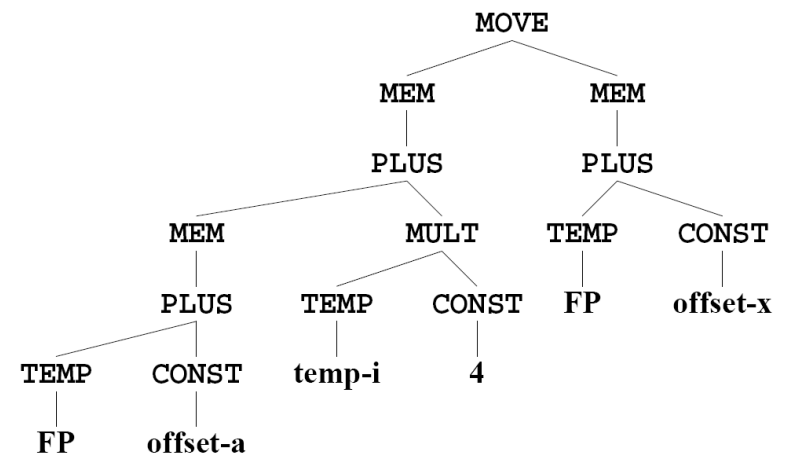


Step 2: Emit instructions

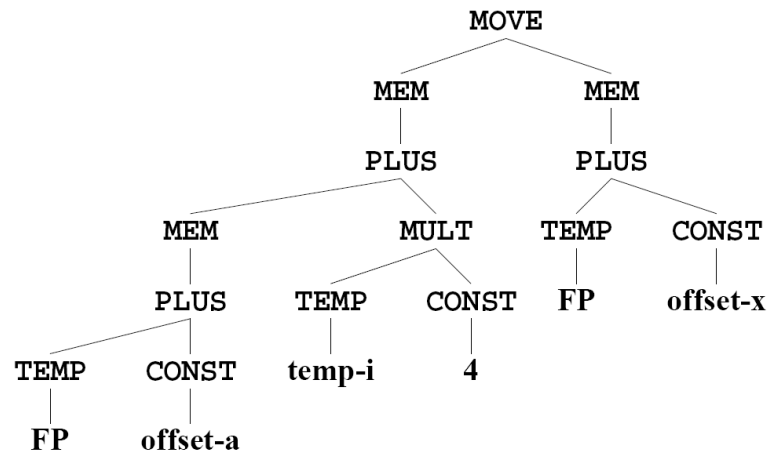
```

ADDI r1 = r0 + 1
LOAD r2 = M[r1 + 2]
  
```

Optimum Instruction Selection – Big Example



Optimum Instruction Selection – Big Example



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...
