Compiler and Code Generation

SS 2009
Hw/Sw Codesign

Christian Plessl
Compiler and Code Generation

- Compiler structure, intermediate code
- Code generation
- Code optimization
- Code generation for specialized processors
- Retargetable compiler
Compiler Structure - Overview

• Compiler phases: analysis, synthesis
• Syntax tree (DAG)
• 3 address code
• Basic blocks (DAG)
• Control flow graphs
Translation Process

skeletal source program

preprocessor

source program

compiler

assembler program

assembler

relocatable machine code

linker / loader

absolute machine code

library
Compiler Phases

source program

lexical analysis

syntactic analysis

semantic analysis

symbol table

intermediate code generation

code optimization

code generation

target program

analysis

error handling

synthesis
Analysis

- **Lexical analysis**
  - scanning of the source program and splitting into symbols
  - regular expressions: recognition by finite automata

- **Syntactic analysis**
  - parsing symbol sequences and construction of sentences
  - sentences are described by a context-free grammar
    
    \[
    A \rightarrow \text{Identifier} := E \\
    E \rightarrow E + E | E * E | \text{Identifier} | \text{Number}
    \]

- **Semantic analysis**
  - make sure the program parts "reasonably" fit together, eg. type casts
Synthesis

• Generation of intermediate code
  – machine independent \(\rightarrow\) simplified retargeting
  – should be easy to generate
  – should be easily translatable into target program

• Optimization
  – GP processors: fast code, fast translation
  – specialized processors: fast code, short code (low memory requirements), low power consumption
  – both intermediate and target code can be optimized

• Code generation
Example (1)

Source program

\text{position} := \text{initial} + \text{rate} \times 60

Lexical analysis

\begin{align*}
\text{id1} & := \text{id2} + \text{id3} \times 60 \\
\text{assignment symbol} & \quad \text{operators} \\
\text{identifiers} & \quad \text{number}
\end{align*}
Example (2)

syntactic analysis

:=

id1

id2

id3

*

+

id1

id2

id3

60

Identifier

:=

IntToReal

60
**Example (3)**

**Intermediate code generation**

\[
\begin{align*}
tmp1 & := \text{IntToReal}(60) \\
tmp2 & := id3 \times tmp1 \\
tmp3 & := id2 + tmp2 \\
id1 & := tmp3
\end{align*}
\]

**Code optimization**

\[
\begin{align*}
tmp1 & := id3 \times 60.0 \\
id1 & := id2 + tmp1
\end{align*}
\]

**Code generation**

\[
\begin{align*}
\text{MOVF } & id3, R2 \\
\text{MULF } & \#60.0, R2 \\
\text{MOVF } & id2, R1 \\
\text{ADDF } & R2, R1 \\
\text{MOVF } & R1, id1
\end{align*}
\]
Syntax Tree and DAG

\[ a := b*(c-d) + e*(c-d) \]

syntax tree

DAG (directed acyclic graph)
3 Address Code (1)

• 3 address instructions
  – maximal 3 addresses (2 operands, 1 result)
  – maximal 2 operators

assignment instructions

x := y op z
x := op y
x := y
x := y[i]
x[i] := y
x := &y
y := *x
*x := y

control flow instructions

goto L
if x relop y goto L

subroutines

param x
call p,n
return y
• Generation of 3 address code

\[
\begin{align*}
    t1 & := c - d \\
    t2 & := e \times t1 \\
    t3 & := b \times t1 \\
    t4 & := t2 + t3 \\
    a & := t4
\end{align*}
\]
3 Address Code (3)

- **Advantages**
  - dissection of long arithmetic expressions
  - temporary names facilitate reordering of instructions
  - forms a valid schedule

- **Definition:** A 3 address instruction
  \[ x := y \text{ op } z \]
  defines \( x \) and
  uses \( y \) and \( z \)
• **Definition:** A basic block is a sequence of instructions where the control flow enters at the beginning and exits at the end, without stopping in-between or branching (except at the end).

```plaintext
t1 := c - d
t2 := e * t1
t3 := b * t1
t4 := t2 + t3
if t4 < 10 goto L
```
Basic Blocks (2)

• Sequence of 3 address instructions → set of basic blocks:

1. determine the block beginnings:
   - the first instruction
   - targets of un/conditional jumps
   - instructions that follow un/conditional jumps

2. determine the basic blocks:
   - there is a basic block for each block beginning
   - the basic block consists of the block beginning and runs until the next block beginning (exclusive) or until the program ends
Control Flow Graphs

- "Degenerated" control flow graph (CFG)
  - the nodes are the basic blocks

```
i := 0
i := 0
L t2 := t2 + i
i := i + 1
if i < 10 goto L
x := t2
```

Diagram:

- Nodes represent basic blocks.
- Arrows indicate control flow.
- Conditions for loop continuation are illustrated.

```plaintext
i < 10
i >= 10
```
DAG of a Basic Block

• **Definition:** A DAG of a basic block is a directed acyclic graph with following node markings:

  - Leaves are marked with a variable / constant name. Variables with initial values are assigned the index 0.

  - Inner nodes are marked with an operator symbol. From the operator we can conclude whether the value or the address of the variable is being used.

  - Optionally, a node can be marked with a sequence of variable names. Then, all variables are assigned the computed value.
C program

```c
int i, prod, a[20], b[20];
...
prod = 0;
i = 0;
do {
    prod = prod + a[i]*b[i];
i++;
} while (i < 20);
```

3 address code

1. prod := 0
2. i := 0
3. t1 := 4 * i
4. t2 := a[t1]
5. t3 := 4 * i
6. t4 := b[t3]
7. t5 := t2 * t4
8. t6 := prod + t5
9. prod := t6
10. t7 := i + 1
11. i := t7
12. if i < 20 goto (3)
Example (2)

basic blocks

(1) prod := 0
(2) i := 0
(3) t1 := 4 * i
(4) t2 := a[t1]
(5) t3 := 4 * i
(6) t4 := b[t3]
(7) t5 := t2 * t4
(8) t6 := prod + t5
(9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i < 20 goto (3)

control flow graph

B1 ----> B2

B1

B2
Example (3)

basic block B2

\[
\begin{align*}
t1 & := 4 \times i \\
t2 & := a[t1] \\
t3 & := 4 \times i \\
t4 & := b[t3] \\
t5 & := t2 \times t4 \\
t6 & := \text{prod} + t5 \\
\text{prod} & := t6 \\
t7 & := i + 1 \\
i & := t7 \\
\text{if } i < 20 \text{ goto (3)}
\end{align*}
\]
## Compiler and Code Generation

- Compiler structure, intermediate code
  - Code generation
- Code optimization
- Code generation for specialized processors
- Retargetable compiler
Code Generation - Overview

- **Subproblems**: register binding, instruction selection, scheduling

- **Simple code generator** (simple machine model)

- **Register binding**: graph coloring, usage counters

- **Code generation for DAGs**

- **Code generation by dynamic programming** (extended machine model)
Code Generation

• Requirements
  – correct code
  – efficient code
  – efficient generation (?)

• Code generation = software synthesis
  – allocation: mostly the components are fixed
  – binding:
    ▪ register binding (register allocation, register assignment)
    ▪ instruction selection
  – scheduling:
    ▪ instruction sequencing
Register Binding

• Goal: efficient use of registers
  – instructions with register operands are generally shorter and faster than instructions with memory operands (CISC)
  – minimize number of LOAD/STORE instructions (RISC)

• Register allocation, register assignment
  – allocation: determine for each time the set of variables that should be held in registers
  – assignment: assign these variables to registers

• Optimal register binding
  – NP-complete problem
  – additionally: restrictions for register use by the processor architecture, compiler, and operating system
Instruction Selection

• Code pattern for each 3 address instruction

\[
\begin{align*}
x &:= y + z \\
u &:= x - w
\end{align*}
\]

\[
\begin{align*}
\text{MOV } &y, R0 \\
\text{ADD } &z, R0 \\
\text{MOV } &R0, x \\
\text{MOV } &x, R0 \\
\text{SUB } &w, R0 \\
\text{MOV } &R0, u
\end{align*}
\]

• Problems
  – often inefficient code generated \(\rightarrow\) code optimization
  – there might be several matching target instructions
  – some instructions work only with particular registers
  – exploitation of special processor features, eg. autoincrement / decrement addressing
Scheduling (1)

• Optimal instruction sequence
  – minimal number of instructions for the given number of registers
  – NP-complete problem

\[
\begin{align*}
t_2 &:= c + d \\
t_3 &:= e - t_2 \\
t_1 &:= a + b \\
t_4 &:= t_1 - t_3
\end{align*}
\]
with 2 registers:

\[
\begin{align*}
  t2 & := c + d \\
  t3 & := e - t2 \\
  t1 & := a + b \\
  t4 & := t1 - t3
\end{align*}
\]

\[
\begin{align*}
  t1 & := a + b \\
  t2 & := c + d \\
  t3 & := e - t2 \\
  t4 & := t1 - t3
\end{align*}
\]
Machine Model

• $n$ registers R0 … Rn-1

• Byte addressable, word = 4 byte

• Instruction format

  \[ \text{op source, destination} \]

  – eg. MOV R0, a
  \[ \text{ADD R1, R0} \]
  – each instruction has cost 1
  – addresses of operands follow the instructions (in the code)
## Addressing Modes

<table>
<thead>
<tr>
<th>mode</th>
<th>form</th>
<th>address</th>
<th>added cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute</td>
<td>M</td>
<td>M</td>
<td>1</td>
</tr>
<tr>
<td>register</td>
<td>R</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>indexed</td>
<td>c(R)</td>
<td>c + contents(R)</td>
<td>1</td>
</tr>
<tr>
<td>indirect register</td>
<td>*R</td>
<td>contents(R)</td>
<td>0</td>
</tr>
<tr>
<td>indirect indexed</td>
<td>*c(R)</td>
<td>contents(c+contents(R))</td>
<td>1</td>
</tr>
<tr>
<td>immediate</td>
<td>#c</td>
<td>c</td>
<td>1</td>
</tr>
</tbody>
</table>
Life Time of Variables

- **Definition:** A name (variable) is active at a certain point inside a basic block if its value is used afterward (possibly in another basic block).

- **Determining life times**
  1. go to the end of the basic block and note names which must be active at the block exit

  2. go back to the block entry instruction by instruction; for each instruction $(I) \ x := y \ op \ z$
     - bind the information (active, next-use) for $x, y, z$ to $(I)$; if $x$ is inactive, remove the instruction
     - set $x$ to inactive and next-use to none
     - set $y, z$ to active and next-use to $(I)$
Simple Code Generator (1)

• For each 3 address instruction \( x := y \ op \ z \)
  1. call \( \text{getreg}(x) \) to determine the location \( L \), to which the result \( x \) will be written

  2. determine \( y' \), the current location of \( y \);
     if \( y' \) is not \( L \), generate \( \text{MOV} \ y', L \)

  3. determine \( z' \), the current location of \( z \), and generate \( \text{op} \ z', L \)

• Current location of a variable \( t \)
  – register or memory; if \( t \) is in both memory and register, prefer the register
Simple Code Generator (2)

- \( L = getreg(x) \) for the instruction \( x := y \ op \ z \)
  
  1. if \( y \) is in a register \( R \), which contains no other variables, and \( y \) 
  
  has no next-use: \( \text{return}(R) \)
  
  2. if there exists an empty register \( R \): \( \text{return}(R) \)
  
  3. if \( x \) has a next-use in the basic block or \( op \) is an operator that needs a 
  
  register (eg. indirect addressing), find a used register \( R \), save the 
  
  register (\text{MOV} \ R, M) and \( \text{return}(R) \)
  
  4. else: \( \text{return}(M_x) \)
Register Allocation

- Global register allocation
  - reserve a number of registers
    - for global variables
    - for loop variables
    - for variables in basic blocks
  - user-defined register allocation
    - eg. in the programming language C:
      ```c
      register int i;
      ```

- Graph coloring (inside basic blocks)

- Usage counters (loops)
Register Allocation - Graph Coloring

• Procedure
  1. code generation with unlimited number of registers (symbolic registers), ie. each variable name is a symbolic register
  2. determine the life time of the variables (symbolic registers)
  3. construction of the register conflict graph
  4. mapping of the symbolic registers to physical registers by graph coloring
Register Conflict Graph

• **Definition:** A register conflict graph \( G(V, E) \) is an undirected graph, where the nodes \( V \) represent the variables (symbolic registers) and the edges \( E \) represent the conflicts between the variables.

- An edge between the nodes \( v_i \) and \( v_j \) indicates that the life times of \( v_i \) and \( v_j \) overlap. Then, \( v_i \) and \( v_j \) cannot share a register.
Register Conflict Graph - Example

loop body (basic block)

(1) \( t_1 := t_3 \times 10 \)
(2) \( t_2 := t_4 \times 20 \)
(3) \( t_3 := t_1 + 5 \)
(4) \( t_4 := t_2 + t_3 \)
Graph Coloring (1)

• Color the nodes of $G$ with $l$ colors such that no two adjacent nodes get the same color.

  – graph coloring is NP-complete
Heuristics: Can we color a graph $G$ with $l$ colors?

Algorithm:
1. find a node $v_i$ in $G$ with $\text{deg}(v_i) < l$
2. remove $v_i$ and all its edges; this results in $G'$
3. if $G' = \{ \}$:
   - $l$ - coloring possible; stop.
   - if all nodes $v_i$ in $G'$ have a $\text{deg}(v_i) \geq l$:
     - $l$ - coloring not possible; stop.
else: $G = G'$; goto 1.
Graph Coloring - Example (1)

\[ k = 3 \]

\[ a \quad b \quad c \quad d \quad e \]

Remove \( d \)

\[ a \quad b \quad c \quad e \]
Graph Coloring - Example (2)

k = 3

\[
\begin{array}{c}
\text{remove } a \\
\text{remove } b \\
\text{remove } c \\
\text{remove } e
\end{array}
\]

\[
\begin{array}{c}
\text{remove } e
\end{array}
\]
Graph Coloring - Example (3)

sequence of removed nodes:
d, a, b, c, e
Usage Counters (1)

- A loop $L$ consists of several basic blocks

- If a variable $a$ stays in a register during the complete execution of $L$, we save
  - 1 cost unit for each use of $a$
    - $\text{ADD R0, R1}$ (cost 1) instead of $\text{ADD a, R1}$ (cost 2)
  - 2 cost units at the end of the basic block, if $a$ has been defined in the basic block and is active afterward
    - no save instruction $\text{MOV R0, a}$
Usage Counter (2)

• Cost savings for the loop

\[
\sum_{B \in L} \left( \text{used}(a, B) + 2 \cdot \text{active}(a, B) \right)
\]

  – \text{used}(a, B) counts the number of usages of \( a \) in basic block \( B \) before \( a \) is defined
  
  – \text{active}(a, B) is 1, if \( a \) has been defined in basic block \( B \) and is active at the end of the basic block; otherwise 0

• Approximation, because we assume that

  – all basic blocks in \( L \) are executed equally often
  
  – \( L \) is executed for many times
Usage Counter - Example

active(a, B1) = 1  
used(a, B2) = 1  
used(a, B3) = 1

cost savings

for 3 registers:
  b ➞ R0, d ➞ R1, a ➞ R2
Code Generation for DAGs (1)

• The node computation order for DAGs has great impact on the number of required instructions

• Heuristics for the node computation order
  1. select a node $n$ with already scheduled predecessors and schedule it
  2. as long as the left-most direct successor node $m$ of $n$ has no unscheduled predecessors and is no leaf:
     (a) schedule $m$
     (b) $n \leftarrow m$
     (c) goto 2.
  3. goto 1.
Code Generation for DAGs - Example

reversed order: 1 2 3 4 5 6 7
Code Generation for DAGs (2)

• For DAGs with \( n \) nodes that are trees, there exists an algorithm that generates optimal code in \( O(n) \) time (dynamic programming).

• Common subexpressions (the DAG is not a tree)

  1. split the DAG into trees at nodes which express common subexpressions

  2. generate optimal code for each tree (not necessarily optimal for the DAG)
Dynamic Programming (1)

- Machine model extended to complex instructions
  - \( n \) registers \( R_0 \ldots R_{n-1} \)
  - instructions \( R_i := E \)
    \( E \) is an expression with arbitrarily many registers and memory addresses
  - if \( E \) contains more than one register, \( R_i \) has to be one of it
  - load: \( R_i := M \), store: \( M := R_i \), copy: \( R_i := R_j \)
  - simplification (for the following): all instructions (all addressing modes) have cost 1

- Examples
  - \( \text{ADD } R_0, R_1 \quad \Rightarrow \quad R_1 := R_1 + R_0 \)
  - \( \text{ADD } *R_0, R_1 \quad \Rightarrow \quad R_1 := R_1 + \text{ind } R_0 \)
  - \( \text{SUB } a, R_0 \quad \Rightarrow \quad R_0 := R_0 - a \)
Dynamic Programming (2)

- Principle of dynamic programming applied to code generation for DAGs

optimal code for $E = (T_1 \text{ op } T_2)$:

1. optimal code for $T_1$, $T_2$
2. optimal code for $E$:
   - either $T_1$, $T_2$, $\text{op}$
   - or $T_2$, $T_1$, $\text{op}$
Dynamic Programming (3)

• Method has 3 phases
  1. Computation of cost vectors
  2. Determination of the instruction sequence
  3. Generation of target code

• Computation of cost vectors for each node $n$ (bottom up):
  - $C[i]$ optimal cost for computing $n$ with $i$ registers
  - $C[0]$ optimal cost for computing $n$, if the result is stored in memory
Example (1)

DAG:

machine model:
  • instruction set
    \[
    \begin{align*}
    Ri &:= Mj \\
    Ri &:= Ri \text{ op } Rj \\
    Ri &:= Ri \text{ op } Mj \\
    Ri &:= Rj \\
    Mj &:= Ri
    \end{align*}
    \]
  • two registers available
Example (2)

cost for computing \( a \):

- into memory \( C[0] = 0 \) (\( a \) is already there)
- with one register \( C[1] = 1 \) (\( R_i := M \))
- with two registers \( C[2] = 1 \)
Example (3)

• with one register: \( R_i := R_i - M \)
  
  left subtree with 1 register,
  right subtree into memory: \( C[1] = 2 \)

• with two registers: \( R_i := R_i - M \)

  cost = 2 = \( C[2] \)

  or \( R_i := R_i - R_j \)

  left subtree with 2 registers,
  right subtree with 1 register:
  cost = 3

• into memory: \( C[0] = 3 \)
Example (4)

- with one register: \( Ri := Ri \times M \)
  left subtree with 1 register, right subtree into memory: \( C[1] = 5 \)

- with two registers: \( Ri := Ri \times M \)
  or \( Ri := Ri \times Rj \)
  left subtree with 2 registers, right subtree with 1 register: cost = 4 = \( C[2] \)
  right subtree with 2 registers, left subtree with 1 register: cost = 4

- into memory: \( C[0] = 5 \)
Example (5)

\[
\begin{align*}
R0 &:= R0 + R1 \\
R0 &:= R0 - b \\
C &:= (3, 2, 2) \\
R0 &:= a \\
a &:= (0, 1, 1) \\
R1 &:= c \\
c &:= (0, 1, 1) \\
C &:= (0, 1, 1) \\
R0 &:= R0 / e \\
e &:= (0, 1, 1) \\
R0 &:= R0 * R0 \\
R1 &:= R1 * R0 \\
C &:= (8, 8, 7) \\
C &:= (5, 5, 4) \\
C &:= (3, 2, 2) \\
C &:= (0, 1, 1) \\
C &:= (0, 1, 1) \\
C &:= (0, 1, 1)
\end{align*}
\]
Compiler and Code Generation

• Compiler structure, intermediate code
• Code generation
  • Code optimization
• Code generation for specialized processors
• Retargetable compiler
Code Optimization

- Transformations on the intermediate code and on the target code

- Peephole optimization
  - small window (peephole) is moved over the code
  - several passes, because an optimization can generate new optimization opportunities

- Local optimization
  - transformations inside basic blocks

- Global optimization
  - transformations across several basic blocks
Optimizations (1)

• Deletion of unnecessary instructions

(1) MOV R0, a
(2) MOV a, R0

if (1) and (2) are in the same basic block

• Algebraic simplifications

\[ x := y + 0 \times (z^4/(y-1)) \]

\[ x := y; \]

\[ x := x \times 1; \]

\[ x := x + 0; \] delete
Optimizations (2)

• Control flow optimizations

(1) \( \text{goto L1} \) \hspace{2cm} (1) \( \text{goto L2} \)

\( \quad \quad \quad \cdots \)

(2) \( \text{L1 goto L2} \) \hspace{2cm} (2) \( \text{L1 goto L2} \)

\text{if L1 is not reachable:}
\text{delete (2) (dead code elimination)}

• Operator reductions

\( x := y \times 8; \) \hspace{2cm} \( x := y \ll 3; \)

\( x := y^{\ast\ast}2; \) \hspace{2cm} \( x := y \times y; \)
Local Optimizations

- Common subexpression elimination

\[
\begin{align*}
(1) & \quad a := b + c \\
(2) & \quad b := a - d \\
(3) & \quad c := b + c \\
(4) & \quad d := a - d \\
\end{align*}
\]

(1) \( a := b + c \)
(2) \( b := a - d \)
(3) \( c := b + c \)
(4) \( d := a - d \)

- Variable renaming

\[
\begin{align*}
t & := b + c \\
u & := b + c \\
t_1 & := b + c \\
t_2 & := x + y \\
t_2 & := x + y \\
t_1 & := b + c \\
\end{align*}
\]

normal form of a basic block: each variable is defined only once

- Instruction interchange

\[
\begin{align*}
t_1 & := b + c \\
t_2 & := x + y \\
t_2 & := x + y \\
t_1 & := b + c \\
\end{align*}
\]
Global Optimizations (1)

- Passive code elimination
  - an instruction that defines $x$ can be deleted if $x$ is not used afterward

- Copy propagation

(1) $x := t_1$
(2) $a[t_2] := t_3$
(3) $a[t_4] := x$
(4) goto L

if $x$ is not used after (1), (1) is passive code
Global Optimizations (2)

- Code motion

```plaintext
while (i <= limit*4+2)
{
    ....
}
```

if `limit` is not modified in the loop body

- Induced variables and operator reduction

```plaintext
j := n
(1) j := j - 1
(2) t4 := 4 * j
(3) t5 := a[t4]
(4) if t5 > v goto (1)
```

```plaintext
j := n
t4 := 4 * j
(1) j := j - 1
(2) t4 := t4 - 4
(3) t5 := a[t4]
(4) if t5 > v goto (1)
```