Compiler and Code Generation 2

SS 2010
HW/SW Codesign

Christian Plessl
Compiler and Code Generation

• Compiler structure, intermediate code

• Code generation

• Code optimization

• Code generation for specialized processors

• Retargetable compiler
Compilers for Embedded Processors

• Software for embedded systems
  – transition from assembly to high-level languages (HLLs)

• Main requirements: code has to be …
  – correct
  – fast
  – small
  – low power

• Further requirements on HLL and compiler
  – formal verification possible
  – specification of real-time constraints
  – support of DSP/multi-media algorithms and architectures
  – retargetable: quickly adaptable to new processors
Selected Problems

• Inhomogeneous register files, irregular data paths
  – tight phase coupling: register binding, instruction binding, instruction sequencing

• Allocation of memory addresses and address registers
  – efficient use of address registers and specialized address generation units

• Code compression
  – reduction of memory, important for cost sensitive applications
Inhomogeneous register files, irregular data paths – TMS320C25 DSP
TMS320C25 - Instructions (1)

- Addition

\[ \text{add} \quad \begin{array}{c}
+ \\
a \\
m \text{ (memory)} \\
a
\end{array} \quad \begin{array}{c}
apac \\
a \\
p \\
a
\end{array} \]

- Subtraction

\[ \text{spac} \quad \begin{array}{c}
- \\
a \\
a \\
p
\end{array} \]

• Addition

• Subtraction
TMS320C25 - Instructions (2)

- Multiplication

\[
\text{mpy} \quad t \rightarrow m \rightarrow p
\]

\[
\text{mpyk} \quad t \rightarrow \text{const} \rightarrow p
\]

- Data transfer

\[
\text{lack} \quad \text{const} \rightarrow a
\]

\[
\text{pac} \quad p \rightarrow a
\]

\[
\text{sac} \quad a \rightarrow m
\]

\[
\text{lac} \quad m \rightarrow a
\]

\[
\text{lt} \quad m \rightarrow t
\]
\[ X := A \times B - (C + D) \times E \]
DAG – Example (2)

; left subtree
; first: cost 12

\begin{verbatim}
lt   M_A
mpy  M_B
pac
sacl M_0
lac  M_C
add  M_D
sacl M_1
lt   M_E
mpy  M_1
lac  M_1
spac
sacl M_X
\end{verbatim}
DAG – Example (3)

; right subtree
; first: cost 14

lac \rightarrow M_C
add \rightarrow M_D
sacl \rightarrow M_0
lt \rightarrow M_E
mpy \rightarrow M_0
pac \rightarrow M_0
sacl \rightarrow M_1
lt \rightarrow M_A
mpy \rightarrow M_B
pac \rightarrow M_B
lt \rightarrow M_1
mpyk \rightarrow 1
spac
sacl \rightarrow M_X
previous method based on dynamic programming is no longer optimal!
Register Transfer Graph

- **Definition:** The register transfer graph for a processor architecture is a directed graph where each node represents a location in the data path at which data can be stored. An edge between two nodes $r_i$ and $r_j$ is labelled with the instructions that read operands from $r_i$ and write operands to $r_j$. 
• **Definition:** The RTG criterion is satisfied if for all node triples $(r_1, r_2, r_3)$ of the RTG for which
  
  1.) $r_3$ has incoming edges from register nodes $r_1$ and $r_2$ with identical labelling, and

  2.) there exists at least one cycle in the RTG including $r_1$ and $r_2$,

  the following holds:

  Each cycle between $r_1$ and $r_2$ includes a memory node.
1. add: \( a = a + m \)
2. apac: \( a = a + p \)
3. spac: \( a = a - p \)
4. mpy: \( p = t \times m \)
5. mpyk: \( p = t \times \text{const} \)
6. lack: \( a = \text{const} \)
7. pac: \( a = p \)
8. sacl: \( m = a \)
9. lac: \( a = m \)
10. lt: \( t = m \)

RTG - TMS320C25

RTG criterion satisfied!
For processor architectures that satisfy the RTG criterion, optimal code can be generated in $O(n)$ time.

- code generation in 2 phases
  - optimal instruction selection and register binding (based on dynamic programming)
  - optimal scheduling
- $n$ is the number of DAG nodes
Address Generation Units (AGUs)

**TMS320C2x**

**ADSP 210x**

**Motorola 56K**
General AGU - Characteristics

- Set of address registers $AR$ for indirect addressing modes

- Set of modification registers $MR$ for updating the $AR$

- Modification operations in parallel to instruction execution
  - eg. post-modify: auto increment/decrement by
    - one address (+/- 1) or
    - the content of an $MR$
<table>
<thead>
<tr>
<th>operation</th>
<th>function</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR load</td>
<td>AR[ARP] = value</td>
<td>1</td>
</tr>
<tr>
<td>MR load</td>
<td>MR[MRP] = value</td>
<td>1</td>
</tr>
<tr>
<td>ARP load</td>
<td>ARP = value</td>
<td>0</td>
</tr>
<tr>
<td>MRP load</td>
<td>MRP = value</td>
<td>0</td>
</tr>
<tr>
<td>immediate modify</td>
<td>AR[ARP] += value</td>
<td>1</td>
</tr>
<tr>
<td>auto-increment</td>
<td>AR[ARP] ++</td>
<td>0</td>
</tr>
<tr>
<td>auto-decrement</td>
<td>AR[ARP] --</td>
<td>0</td>
</tr>
<tr>
<td>auto-modify</td>
<td>AR[ARP] += MR[MRP]</td>
<td>0</td>
</tr>
</tbody>
</table>
AGU – Problems

- Allocation of memory addresses and address/modification registers
  - scalar variables:
    - What is an optimal ordering of variables in memory, given \( 1 \ AR/n \ AR \)?
    - How to efficiently use \( MR \)?
  - array variables:
    - How to efficiently use \( AR/MR \) for loops? (important for DSPs)
Addressing of Scalar Variables

- access sequence: \texttt{b, d, a, c, d, a, c, b, a, d, a, c, d}
- 1 \texttt{AR} available

\begin{itemize}
  \item \texttt{LOAD AR, 1}
    \begin{itemize}
      \item \texttt{AR += 2}
      \item \texttt{AR -= 3}
      \item \texttt{AR += 2}
      \item \texttt{AR += 2}
      \item \texttt{AR -= 2}
      \item \texttt{AR ++}
      \item \texttt{AR --}
    \end{itemize}
  \item \texttt{LOAD AR, 3}
    \begin{itemize}
      \item \texttt{AR --}
      \item \texttt{AR --}
      \item \texttt{AR =~= 3}
      \item \texttt{AR =~= 3}
      \item \texttt{AR =~= 3}
      \item \texttt{AR += 3}
      \item \texttt{AR += 3}
    \end{itemize}
\end{itemize}

addressing cost = 9

addressing cost = 5
Access Graph

- **Definition:** The access graph for a given set of variables and an access sequence consists of
  - nodes that model the variables,
  - edges that connect nodes if the corresponding variables are adjacent in the access sequence
  - edge weights that denote the number of transitions between the corresponding variables in the access sequence

An optimal address allocation is given by a Hamiltonian path in the access graph (path that visits each node exactly once) with maximal edge weight.

- NP-hard problem
Access Graph - Example (1)

access sequence: \texttt{b, d, a, c, d, a, c, b, a, d, a, c, d}
Access Graph - Example (2)

- additionally 1 MR available:

```plaintext
LOAD AR, 3
AR --
AR --
AR --
AR += 2
AR --
AR --
AR += 3
AR -= 2
AR ++
AR --
AR --
AR += MR
addressing cost = 5

LOAD AR, 3
AR --
AR --
AR --
LOAD MR, 2
AR += MR
AR --
AR --
AR += 3
AR -= MR
AR ++
AR --
AR --
AR += MR
addressing cost = 3
```
Addressing of Array Variables

• Typical assumptions for DSP code:
  – nested for loops with known number of iterations
  – each (nesting) level has a loop counter which is in/decremented by one
  – reference of an array element:

\[ A[f_1(i_1)][f_2(i_2)] \ldots [f_n(i_n)] \]

indexing function: \( f_j(i_j) = \{ c, i_j \pm c, c \pm i_j \} \)

\( c \in \mathbb{N}_0 \)

loop counter: \( i_j \)
Memory Layout

- \( n \)-dimensional array

\[
A[m_1][m_2] \cdots [m_n] \quad m_i \in [0, N_i - 1]
\]

\[
\text{adr}(A[m_1][m_2] \cdots [m_n]) = \text{base}(A) + \sum_{j=1}^{n} m_j a_j
\]

\[
a_j = \begin{cases} 
\prod_{k=j+1}^{n} N_k & 1 \leq j < n \\
1 & j = n
\end{cases}
\]

- Example: array definition \( A[10][4][5] \)

\[
\text{adr}(A[5][3][1]) = \text{base}(A) + 5*(4*5) + 3*5 + 1*1 = \text{base}(A) + 116
\]
• Simple loop: \[
\text{FOR } i=\text{low TO high DO}
\]
\[
\begin{align*}
\text{ref 1} \\
\text{ref 2} \\
\ldots \\
\text{ref } k
\end{align*}
\]
\[
\text{ENDFOR}
\]

• **Definition:** The update value \(UV\) of an array reference is:

\[
UV(A[f_1(i)][f_2(i)]\cdots[f_n(i)]) = adr(A[f_1(i_1)]\cdots[f_n(i_1)]) - \]
\[
adr(A[f_1(i_0)]\cdots[f_n(i_0)])
\]

where \(i_0\) is the index of an iteration and \(i_1\) the index of the next iteration.
Update Value - Example

FOR i = 2 TO 1024 DO
(1) ref A[i+1]
(2) ref A[i]
(3) ref A[i+2]
(4) ref A[i-1]
(5) ref A[i+1]
(5) ref A[i]
(7) ref A[i-2]
ENDFOR

AR1 = adr(A[3])
FOR i = 2 TO 1024 DO
  AR1 --
  AR1 += 2
  AR1 -= 3
  AR1 += 2
  AR1 --
  AR1 -= 2
  AR1 += 4
ENDFOR

references with identical UVs may share an address register:
**Code Compression**

- Target code is redundant and can be compressed
  - GP systems: decompression at program loading time
  - for embedded systems the reduction of program ROMs, RAMs is important

- Decompression in the cache

- External Pointer Macro (EPM) model
  - dictionary: contains frequently used code sequences (mini-subroutines)
  - skeleton: contains instructions and pointers to the dictionary
  - implementation in SW or HW
EPM Model in Hardware
Compiler and Code Generation

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

• Portable Compiler
  – developer retargetable
  – code generation by tree pattern matching

• Compiler-compiler
  – user retargetable (semi-automatic)
  – compiler is generated from a description of the target architecture (processor model)

• Machine independent compiler
  – automatically retargetable
  – compiler generates code for several processors / processor variants
  – for parametrizable architectures
Tree Pattern Matching

- Rules for transforming a syntax tree (DAG) are given as tree patterns

replacement \rightarrow pattern \{ action \}

example: \( \text{reg i} \leftarrow + \{ \text{ADD Rj, Ri} \} \)

\[
\begin{array}{c}
\text{reg i} \\
\text{+} \\
\text{reg i} \\
\text{reg j}
\end{array}
\]

stepwise replacement by tree pattern matching until the tree contains only one node
Target Instructions (1)

(1) \( \text{reg } i \leftarrow \text{const } c \{ \text{MOV } \#c, \text{Ri} \} \)

(2) \( \text{reg } i \leftarrow \text{mem } a \{ \text{MOV } a, \text{Ri} \} \)

(3) \( \text{mem} \leftarrow := \{ \text{MOV } \text{Ri}, a \} \)

(4) \( \text{mem} \leftarrow := \{ \text{MOV } \text{Rj}, *\text{Ri} \} \)

(5) \( \text{reg } i \leftarrow \text{ind} \{ \text{MOV c(Rj), Ri} \} \)
Target Instructions (2)

(6) \[ \text{reg } i \leftarrow + \{ \text{ADD } c(R_j), R_i \} \]

\[
\begin{array}{c}
\text{reg } i \\
\text{ind} \\
+ \\
\text{const } c \\
\text{reg } j
\end{array}
\]

(7) \[ \text{reg } i \leftarrow + \{ \text{ADD } R_j, R_i \} \]

\[
\begin{array}{c}
\text{reg } i \\
\text{reg } j
\end{array}
\]

(8) \[ \text{reg } i \leftarrow + \{ \text{INC } R_i \} \]

\[
\begin{array}{c}
\text{reg } i \\
\text{const } 1
\end{array}
\]
Tree Pattern Matching - Example (1)

\[ a[i] := b + 1 \]
Tree Pattern Matching - Example (2)

\[ a[i] := b + 1 \]

(7) \{ \text{ADD SP, R0} \}
Tree Pattern Matching - Example (3)

\[ a[i] := b + 1 \]

\begin{align*}
(6) \{ \text{ADD } i(\text{SP}), \ R0 \} \\
\text{ind} & \quad + \\
\text{reg } 0 & \quad \text{ind} \\
\text{mem } b & \quad \text{const } 1 \\
\text{const } i & \quad \text{reg } SP
\end{align*}
Tree Pattern Matching - Example (4)

\[ a[i] := b + 1 \]

(2) \{ MOV b, R1 \}
a[i] := b + 1

Tree Pattern Matching - Example (5)

(8) { INC R1 }
Tree Pattern Matching - Example (6)

\[ a[i] := b + 1 \]

(4) \{ MOV R1, *R0 \}
Compiler Compiler:

Instruction Set Extraction

**Instruction:** \( xx011zz \)

**Operation:** \( \text{reg}[zz] \leftarrow \text{reg}[xx] + \text{acc} \)

**Pattern:**
```
+  
\downarrow  
\text{acc}  \rightarrow  \text{reg}  
\uparrow  
\downarrow  
\text{reg}  
```

**Control Bits:**
```
\( xx \)  
\downarrow  
\text{reg}  
\uparrow  
\text{acc}  
\downarrow  
\( 01 \)  
\downarrow  
\( 1 \)  
\downarrow  
\( zz \)  
\downarrow  
\text{reg}  
```
Processor Models

- Approaches for evaluating performance of processors without hardware implementation

- Behavioral models
  - describe the instruction set
  - simulation relatively fast (100-1000 times slower than target machine)
  - not very accurate (no pipelining effects)

- Structural models
  - describe the processor on the register transfer level
  - accurate
  - simulation rather slow

- Mixed models