Embedded Systems Development

Lecture 8
Code Generation for Embedded Processors

Daniel Kästner
AbsInt Angewandte Informatik GmbH
kaestner@absint.com
Life Range and Register Interference

- A symbolic register (a variable) \( r \) is live at a program point \( p \), if there is a program path from the entry node of the procedure to \( p \) that contains a definition of \( r \) and there is a path from \( p \) to a use of \( r \) on which \( r \) is not defined. The life range of a symbolic register \( r \) is the set of program points at which \( r \) is live.

- Two life ranges of symbolic registers interfere, if one of them is defined during the life range of the other. The register interference graph is an undirected graph whose nodes are life ranges of symbolic registers and whose edges connect the nodes of interfering life ranges.

\[
\begin{align*}
  v_1 &= \text{Mem}[0xAFA0] \\
  v_2 &= \text{Mem}[0xAFC0] \\
  v_3 &= v_1 + v_2 \\
  v_4 &= v_1 \times v_2 \\
  v_5 &= v_3 + v_4 \\
  \text{return } v_5
\end{align*}
\]
Register Allocation by Graph Coloring

- If $k$ physical registers are available, the $k$-coloring problem must be solved.
- NP-complete for $k>2$ -> Use heuristics

Algorithm:
- If $G$ contains a node $n$ with degree $< k$:
  - $n$ and its neighbors can be colored with different colors
  - Remove $n$ from $G$, decreasing the size of $G$
  - $G$ is $k$-colorable, if we arrive at the empty graph.
- If $G$ is not empty and there exists no node with degree $< k$:
  - use heuristics to select one node to remove (spilling)
  - modify program inserting spills at definitions and loads at uses
  - reflect changes in graph.
Register Allocation by Graph Coloring

- Heuristics for node removal
  - Degree of the node: high degree of node causes many deletions of edges.
  - Costs of spilling.
  - Register constraints of operands.

- Enhancements:
  - Optimistic Coloring.
  - Coalescing.
  - Life range splitting.
  - Pre-colored nodes.
  - Support for multiple register sets.
  - Preserve topological information (loop nesting level).
Example: Optimistic Graph Coloring
Embedded Processors: Application Areas

- Automotive
- Avionics & Space
- Telecommunication
- Consumer electronics
- Healthcare technology
Design of Embedded Systems: Goal Conflict

High performance requirements
Increasing software complexity
High dependability

Low cost
Low power consumption
Short product and development cycles
Types of Microprocessors

- Complex Instruction Set Computer (CISC)
  - large number of complex addressing modes
  - many versions of instructions for different operands
  - different execution times for instructions
  - few processor registers
  - microprogrammed control logic

- Reduced Instruction Set Computer (RISC)
  - one instruction per clock cycle
  - memory accesses by dedicated load/store instructions
  - few addressing modes
  - hard-wired control logic
Example Instruction (IA-32)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 ib</td>
<td>ADD AL,imm8</td>
<td>Add imm8 to AL</td>
</tr>
<tr>
<td>05 iw</td>
<td>ADD AX,imm16</td>
<td>Add imm16 to AX</td>
</tr>
<tr>
<td>05 id</td>
<td>ADD EAX,imm32</td>
<td>Add imm32 to EAX</td>
</tr>
<tr>
<td>80 /0 ib</td>
<td>ADD r/m8,imm8</td>
<td>Add imm8 to r/m8</td>
</tr>
<tr>
<td>81 /0 iw</td>
<td>ADD r/m16,imm16</td>
<td>Add imm16 to r/m16</td>
</tr>
<tr>
<td>81 /0 id</td>
<td>ADD r/m32,imm32</td>
<td>Add imm32 to r/m32</td>
</tr>
<tr>
<td>83 /0 ib</td>
<td>ADD r/m16,imm8</td>
<td>Add sign-extended imm8 to r/m16</td>
</tr>
<tr>
<td>83 /0 ib</td>
<td>ADD r/m32,imm8</td>
<td>Add sign-extended imm8 to r/m32</td>
</tr>
<tr>
<td>00 /r</td>
<td>ADD r/m8,r8</td>
<td>Add r8 to r/m8</td>
</tr>
<tr>
<td>01 /r</td>
<td>ADD r/m16,r16</td>
<td>Add r16 to r/m16</td>
</tr>
<tr>
<td>01 /r</td>
<td>ADD r/m32,r32</td>
<td>Add r32 to r/m32</td>
</tr>
<tr>
<td>02 /r</td>
<td>ADD r8,r/m8</td>
<td>Add r/m8 to r8</td>
</tr>
<tr>
<td>03 /r</td>
<td>ADD r16,r/m16</td>
<td>Add r/m16 to r16</td>
</tr>
<tr>
<td>03 /r</td>
<td>ADD r32,r/m32</td>
<td>Add r/m32 to r32</td>
</tr>
</tbody>
</table>

Execution time:
- 1 cycle: ADD EAX, EBX
- 2 cycles: ADD EAX, memvar32
- 3 cycles: ADD memvar32, EAX
- 4 cycles: ADD memvar16, AX

Instruction Width:
between 1 byte (NOP) and 16 bytes
Types of Microprocessors

- **Superscalar Processors**
  - subclass of RISCs or CISCs
  - multiple instruction pipelines for overlapping execution of instructions
  - parallelism not necessarily exposed to the compiler

- **Very Long Instruction Word (VLIW)**
  - statically determined instruction-level parallelism (under compiler control)
  - instructions are composed of different machine operations whose execution is started in parallel
  - many parallel functional units
  - large register sets
VLIW Architectures

![Diagram of VLIW Architecture]
Analog Devices SHARC

- 32-bit DSP for speech, sound, graphics, and imaging applications. ADSP-21060 used in Dolby Surround AV/R- Receivers.

- **Memory:**
  - 48-bit PM (program memory) bus (DAG1)
  - 40-bit DM (data memory) bus (DAG2)
  - PM and DM can be accessed simultaneously, but: parallelism is **restricted**

- Instruction width: 48 bit

- General purpose **register file:** 16 40-bit registers used for fixed-point and floating-point data

- Load/store architecture

- Three computation units: ALU, multiplier, shifter
  - **restricted** parallelism of ALU and multiplier
Analog Devices SHARC

- Four types of address registers each consisting of 8 registers in DAG1 (32 bit) and 8 registers in DAG2 (24 bit):
  - index registers (I): base address of memory accesses
  - modify registers (M): offset/increment value
  - base registers (B): base address of circular buffer
  - length registers (L): length of circular buffer

- **Addressing modes:**
  - direct addressing
  - indirect addressing with pre- and post-modify
  - circular addressing
  - bit-reverse addressing
Analog Devices SHARC

• Restricted parallelism between ALU and multiplier.

\[
R1 = R1 \times R4 \\
R2 = R8 + R12
\]

\[
R1 = R1 \times R3 \\
R2 = R8 + R12
\]
TriMedia TM1000

- Media processor for real-time processing of audio, video, graphics and communication.
- 32-bit VLIW processor with 128 general purpose registers and 27 functional units.
- Load/store architecture.
- Instruction set is composed of RISC-like operations, multimedia and DSP operations (SIMD; e.g. ifir)
- Each instruction word is composed of 5 microoperations.
TriMedia TM1000

- **Execution time** of operations is between 1 and 17 clock cycles.
- **Predicated execution**: each operation can optionally be guarded. The operation is executed depending on the value of the general-purpose register specified as guard operand.
- The **packing** of operations into instructions (assignment of operations to issue slots of the instruction word) is restricted:
  - E.g. no more than two load/store operations can be packed into a single instruction.
  - No more than 5 results can be written during any one cycle.

<table>
<thead>
<tr>
<th>Type</th>
<th>Slot1</th>
<th>Slot2</th>
<th>Slot3</th>
<th>Slot4</th>
<th>Slot5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
TriMedia TM1000

- Example code sequence:

(* cycle 0 *)
IF r1 iaddi(0x2) r0 -> r38,        IF r1 isubi(0x4) r0 -> r7,
IF r1 isubi(0x3) r0 -> r8,        IF r1 isubi(0x2) r0 -> r36,
IF r1 isubi(0x1) r0 -> r37;

(* cycle 1 *)
IF r1 iaddi(0x3) r0 -> r39,        IF r1 uimm(0x190) -> r34,
IF r1 iaddi(0x8) r0 -> r35,        IF r1 h_st8d(0) r7 r6,
IF r1 h_st8d(4) r0 r6;
Architectural Valuation

- More efficient architectures will use less energy to complete the same task on the same generation CMOS solid state technology.

- Power consumption: \( P = CV^2 f \Delta N \)
  - \( C \): Capacitance
  - \( V \): CPU Core Voltage
  - \( f \): CPU clock frequency
  - \( D/N \): Number of gates changing state

- Architectural specialization as a measure for how well the architecture fits a given target application.

- Estimation of architectural specialization: Performance per power.
Architectural Valuation

- Observations:
  - Higher performance by increasing the clock frequency does not change the performance per power ratio
    \[
    \frac{\text{Performance}}{\text{Power}} = \frac{O}{CV^2 f\Delta N} = \frac{Of}{CV^2 f\Delta N} = \frac{O}{CV^2 \Delta N}
    \]
  - A voltage decrease improves performance per power non-linearly
    \[
    \frac{\text{Performance}}{\text{Power}} = \frac{O}{CV^2 f\Delta N}
    \]
Comparison of Performance Per Power Ratios

![Graph showing FFT MFLOPS/WATT](image)

- **Alpha 21064A**, 275 MHz, 275 MFLOP Peak, 3.3 Volts, 33+8 Watts
- **Alpha 21164**, 333 MHz, 666 MFLOP Peak, 2.2 Volt, 25.4+6 Watts
- **SHARC 21060**, 40 MHz, 120 MFLOP Peak, 3.3 Volt, 1.75 Watts

\[
\frac{Q_{\text{DSP32}}}{Q_{\text{RISC64}}} = \frac{58}{3.14} \approx 18
\]

\[
\frac{Q_{\text{RISC64}}}{Q_{\text{RISC64}}} = \frac{10.9}{4.8} = 2.3
\]
Classification of Microprocessors

Microprocessors

General Purpose Processors (GPP)
- GPP proper: general purpose applications
- Microcontrollers: industrial applications

Application Specific Processors (ASP)

DSP (Digital Signal Processor): programmable microprocessor for extensive numerical real-time computations

ASIC (Application Specific Integrated Circuit): algorithm completely implemented in hardware

ASIP (Application Specific Instruction Set Processor): programmable microprocessor where hardware and instruction set are designed together for one special application

Requirements:
- high performance
- low cost
- low power consumption

Specialization
The DSPStone Study

- Evaluation of the performance of **DSP compilers** and joint compiler/processor systems [1994]. Evaluated compilers:
  - Analog Devices ADSP2101,
  - AT&T DSP1610,
  - Motorola DSP56001,
  - NEC mPD77016,
  - TI TMS320C51.

- Hand-crafted assembly code is compared to the compiler-generated code.

- Result: overhead between **100%** and **1000%** of compiler-generated code is typical!
Characteristics of Embedded Processors

- **Multiply-accumulate units**: multiplication and accumulation in a single clock cycle (vector products, digital filters, correlation, fourier transforms, etc)

- **Multiple-access memory architectures** for high bandwidth between processor and memory
  - Goal: throughput of one operation per clock cycle.
  - Required: several memory accesses per clock cycle.
  - Separate data and program memory space: harvard architecture.
  - Multiple memory banks
  - Arithmetic operations in parallel to memory accesses. But often irregular restrictions.

- **Specialized addressing modes**, e.g. bit-reverse addressing or auto-modify addressing.
Characteristics of Embedded Processors

- **Predicated/guarded execution**: instruction execution depends on the value of explicitly specified bit values or registers.

- **Hardware loops / zero overhead loops**: no explicit loop counter increment/decrement, no loop condition check, no branch back to top of loop.

- **Restricted interconnectivity** between registers and functional units -> phase coupling problems.

- **Strongly encoded instruction formats**: a throughput of one instruction per clock cycle requires one instruction to be fetched per cycle. Thus each instruction has to fit in one memory word => reduction of bit width of the instruction.
Characteristics of Embedded Processors

- **SIMD instructions:**
  - Focus on multimedia applications
  - SIMD: Single Instruction Multiple Data
  - SIMD-Instruction: instruction operating concurrently on data that are packed in a single register or memory location.

```
a = b + c*z[i+0]  \Rightarrow  \begin{array}{c}
  a \\
d \\
r \\
w
\end{array} = \begin{array}{c}
  b \\
  c \\
x \\
  y
\end{array} +_{\text{SIMD}} \begin{array}{c}
  z[i+0] \\
  z[i+1] \\
  z[i+2] \\
  z[i+3]
\end{array}
```
Characteristics of Embedded Processors

- Cost constraints, low power requirements and specialization:
  - Irregularity
  - Phase coupling problems during code generation
  - Need for specialized algorithms
The Phase Coupling Problem

- Main subtasks of compiler backends:
  - **code selection**: mapping of IR statements to machine instructions of the target processor
  - **register allocation**: map variables and expressions to registers in order to minimize the number of memory references during program execution
  - **register assignment**: determine the physical register used to store a value that has been previously selected to reside in a register
  - **instruction scheduling**: reorder an instruction sequence in order to exploit instruction-level parallelism and minimize pipeline stalls
  - **resource allocation**: assign functional units and buses to operations
The Phase Coupling Problem

- Classical approaches: isolated solution by heuristic methods (list scheduling, trace scheduling, graph coloring register allocation, etc).

  ⇒ Problem: interdependence of code generation phases.
  ⇒ Suboptimal combination of suboptimal partial results.
  ⇒ Inefficient code.

- Solution in a perfect world: Address all problems simultaneously in a single phase. BUT:
  - How to formulate this?
  - Code selection, register allocation/register assignment and instruction scheduling by themselves are NP-hard problems.
  - This means that in general there is no chance to optimally solve even one single of these tasks separately.
Code Selection and Register Allocation

- The goal of code selection is to determine the cheapest instruction sequence for a subgraph of the IR. However, the code selector does not know what the real overall costs of the instruction sequence will be; it can use only estimations.
- Register allocation usually is done after code selection so the code selector typically has to assume an infinite number of registers (virtual registers).
- In consequence when estimating the cost of an instruction sequence the code selector will mostly assume register references.
- Register allocation has to cope with a finite number of registers. If there are too few registers, spill code is generated.
- However, the cost of this spill code has not been considered during code selection, so the chosen operation sequence may in fact be a bad choice since another one might have worked without spill code.
Register Allocation and Instruction Scheduling

d₁ = s₁ * s₁;
d₂ = s₁ + s₂;

R₃ = R₁ * R₁
store R₃
R₃ = R₁ + R₂
store R₃

R₃ = r₁ * r₁, R₄ = R₁ + R₂
store R₃
store R₄

Register Allocation first
Instruction Scheduling first
Instruction Scheduling and Register Assignment

Analog Devices SHARC: Restricted Parallelism between ALU and multiplier.

R1 = R1 * R4
R2 = R8 + R12

R1 = R1 * R3
R2 = R8 + R12

R1 = R1 * R3
R2 = R8 + R12
Code Selection and Complex Instructions

- Code selection usually is done by tree parsing (tree pattern matching) and dynamic programming.
- Usually the IR however takes the form of directed acyclic graphs, e.g. due to common subexpressions.
- Before code selection proper DAGs must be broken into trees for the code selection algorithm to work.
- Breaking the trees is done heuristically. Thus the resulting trees and the resulting use of temporary storage locations may destroy the opportunity of generating complex instructions which would correspond to larger expression trees.
Expression DAG

On TriCore: could be implemented by one maddsum.h instruction

Expression Trees

Requires sequences of mul, add, sub instructions
The Offset Assignment Problem

- Many embedded processors have few general purpose registers
- Program variables kept in memory
- **Address Registers** (AR) used to access variables
- **Modify Registers** (MR) can be used to update Address Registers.
- After a variable is accessed, the AR can be auto-incremented (or decremented) by one word in the same cycle.

Example: Texas Instruments TMS320C54X DSP family:
- Accumulator-based DSP
- 8 Address Registers
- Initializing an address register requires 2 cycles of overhead
- Explicit address computations require 1 cycle of overhead
- Using auto-increment (or auto-decrement) has no overhead.
Example

- add ‘A’ and ‘B’, store in accumulator

\[
\begin{array}{ccc}
A & C & B \\
0x1000 & 0x1001 & 0x1002 \\
\end{array}
\]

\[
\begin{align*}
$AR0 &= &\&A \\
$ACC &= &*AR0 \\
$AR0 &= &AR0 + 2 \\
$ACC &= &*AR0
\end{align*}
\]

Explicit address computation

\[
\begin{array}{ccc}
A & B & C \\
0x1000 & 0x1001 & 0x1002 \\
\end{array}
\]

\[
\begin{align*}
$AR0 &= &\&A \\
$ACC &= &*AR0++ \\
$ACC &= &*AR0
\end{align*}
\]

Auto-Increment
The Offset-Assignment Problem

- \((k,m,r)\)-OA is the problem of generating a memory layout which minimizes the cost of addressing variables, with
  - \(k\): number of address registers (AR)
  - \(m\): number of modify registers (MR)
  - \(r\): the offset range

- The case \((1,0,1)\) is called simple offset assignment (SOA).
  - Equivalent to finding a maximum weight path cover (NP-complete)

- The case \((k,0,1)\) is called general offset assignment (GOA).
  - Consists of SOA problems, and is at least NP-hard.
Effect of Memory Layout

- Variables in a basic block: \( V = \{a, b, c, d\} \)
- Access sequence: \( S = (b, d, a, c, d, c) \)
- Cost Model:
  - immediate AR load: 2
  - immediate AR modify: 1
  - auto-modify: 0
  - \( \text{AR} += \text{MR}: 0 \)

```
0       Load AR,1 ;b
     AR += 2 ;d
1       AR -= 3 ;a
     AR += 2 ;c
2       AR ++ ;d
     AR -- ;c
3
```

**cost: 5**

```
0       Load AR,0 ;b
     AR ++ ;d
1       AR +=2 ;a
     AR -- ;c
2       AR -- ;d
     AR ++ ;c
3
```

**cost: 3**
The Access Graph

- Access sequence: \( S = (b, d, a, c, d, c) \)
Solving the SOA

- Naive approach: access sequence defined by the order in which the variables are defined.

- Liao’s algorithm:
  - Similar to Kruskal’s spanning tree algorithms:
    1. Sort edges of access graph $G=(V,E)$ according to their weight
    2. Construct a new graph $G'=(V',E')$, starting with $E' = \emptyset$
    3. Select an edge $e$ of $G$ of highest weight. If this edge does not cause a cycle in $G'$ and does not cause any node in $G'$ to have a degree $> 2$ then add this node to $E'$. Otherwise discard $e$.
    4. Goto 3 as long as not all edges from $G$ have been selected and as long as $G'$ has less than the maximum number of edges ($|V| - 1$).
Example for Liao’s Algorithm

- Access sequence: a b c d e f a d a d a c d f a d

G

G'

\[
\begin{align*}
\text{G:} & \\
\text{G':} & \\
\end{align*}
\]
General Offset Assignment (GOA)

- Allow multiple address registers (k>1)
- Find an ordering of variables in memory (memory layout) that has minimum overhead.
- Assign each variable to an address register to form access sub-sequences.

Ex.

Access Sequence: ‘a d b e c f b e c f a d’
Sub-sequence1: ‘a b c b c a’
Sub-sequence2: ‘d e f e f d’
General Offset Assignment (GOA)

- Each sub-sequence can be viewed as an independent SOA problem.
- Solve each sub-sequence as independent SOA problems.
- Requires solving SOA instances, so is at least NP-hard.

Ex.

Access Sequence: ‘a d b e c f b e c f a d’
Sub-sequence1: ‘a b c b c a’
Sub-sequence2: ‘d e f e f d’
General Offset Assignment (GOA)

- Each sub-sequence can be viewed as an independent SOA problem.
- Solve each sub-sequence as independent SOA problems.
- Requires solving SOA instances, so is at least NP-hard.

Ex.
Access Sequence: ‘a d b e c f b e c f a d’
Sub-sequence1: ‘a b c b c a’
Sub-sequence2: ‘d e f e f d’
General Offset Assignment (GOA)

- Each sub-sequence can be viewed as an independent SOA problem.
- Solve each sub-sequence as independent SOA problems.
- Requires solving SOA instances, so is at least NP-hard.

Ex.

Access Sequence: ‘a d b e c f b e c f a d’
Memory layout: a b c d e f
General Offset Assignment (GOA)

- Variables are assigned to address registers.
- There is nothing left to decide – each address register has a defined sequence of accesses.
- Imposes a restriction that all access to a variable is done by a single address register.

Ex.

Access Sequence: ‘a d b e c f b e c f a d’
Memory layout: 

```
  a b c d e f
  AR0 AR1
```

Diagram:

```
A --3-- B
    
C --3-- F
    
D --3-- E
```
General Offset Assignment (GOA)

- Variables are assigned to address registers.
- There is nothing left to decide - each address register has a defined sequence of accesses.
- Imposes a restriction that all access to a variable is done by a single address register.

Ex.

Access Sequence: ‘a d b e c f b e c f a d’

Memory layout:

| a | b | c | d | e | f |

AR0 AR1
General Offset Assignment (GOA)

- Variables are assigned to address registers.
- There is nothing left to decide - each address register has a defined sequence of accesses.
- Imposes a restriction that all access to a variable is done by a single address register.

Ex.

Access Sequence: ‘a d b e c f b e c f a d’
Memory layout: 

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
</tbody>
</table>

AR0

AR1
General Offset Assignment (GOA)

- Variables are assigned to address registers.
- There is nothing left to decide – each address register has a defined sequence of accesses.
- Imposes a restriction that all access to a variable is done by a single address register.

Ex.

Access Sequence: ‘a d b e c f b e c f a d’

Memory layout: 

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR0</td>
<td>AR1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
General Offset Assignment (GOA)

- Variables are assigned to address registers.
- There is nothing left to decide – each address register has a defined sequence of accesses.
- Imposes a restriction that all access to a variable is done by a single address register.

Ex.

Access Sequence: ‘a d b e c f b e c f a d’

Memory layout: 

```
 a b c
 d e f
```

AR0   AR1
General Offset Assignment (GOA)

- Variables are assigned to address registers.
- There is nothing left to decide – each address register has a defined sequence of accesses.
- Imposes a restriction that all access to a variable is done by a single address register.

Ex.

Access Sequence: ‘a d b e c f b e c f a d’
Memory layout: 

```
| a | b | c | d | e | f |
```

AR0 AR1
General Offset Assignment (GOA)

- Variables are assigned to address registers.
- There is nothing left to decide - each address register has a defined sequence of accesses.
- Imposes a restriction that all access to a variable is done by a single address register.

Ex.

Access Sequence: ‘a d b e c f b e c f a d’

Memory layout: \[
\begin{array}{cccc}
  a & b & c & d \\
  e & f & & \\
\end{array}
\]

*Requires Explicit Address Computations