Retargetable Compilers

Daniel Karlsson

danka@ida.liu.se

ESLAB, IDA, Linköpings universitet

Introduction

- System on Chip
- Many different types of DSPs and embedded processors
  - ASIPs (replace by latest general purpose processor?)
  - no compromise, high speed, low cost, low power
- Compilation plays important role

Outline

- Introduction
- Basic compilation techniques
- Retargetable compilation issues
- Processor modelling and the CHESS compiler
- Summary

Basic Compilation Techniques (Overview)

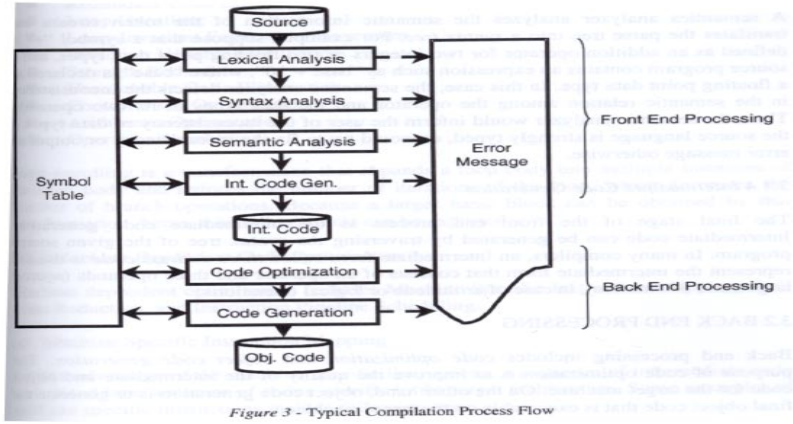


Figure 3 - Typical Compilation Process Flow

Front-End Processing

- Lexical analysis
  - Generating tokens
- Syntax analysis
  - Generating parse tree
  - Creating symbol table
- Semantic analysis
  - Generating syntax tree
- Intermediate code generation
  - Generating quadruples (virtual machine)

Back-End Processing (1/3)

- Machine independent code optimisation
  - Common sub-expression elimination
  - Loop unrolling
  - Loop-invariant expression movement
  - Induction variable elimination
  - Unreachable code elimination
  - Control flow optimisation
  - Arithmetic optimisation
  - Operation combining

Back-End Processing (2/3)

- Machine dependent code optimisation
  - Machine specific instruction mapping
    - Auto incr/decr indexed memory access instructions
    - Stack instructions
    - MAC (Multiply and ACcumulate) instruction
  - Spill code reduction
    - Too many pseudo registers -> memory space access
  - Instruction scheduling
    - Pipeline hazards

Back-End Processing (3/3)

- Object code generation
  - Code reordering
  - Instruction pattern matching
  - Register allocation
  - Register assignment

Questions

- What stages/transformations does the code go through in a compiler?
- Name a few optimisation strategies, both machine independent and machine dependent.
- What are the tasks of code generation?

Discussion:

- Is there one optimisation strategy which generally generates better results than others?

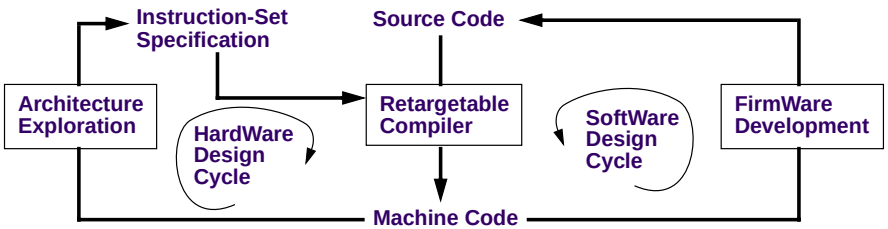
What's the Difference?

- Retargetability
- Register constraints
  - Special-purpose registers
  - Unusual wordlength
- Arithmetic specialisation
- Instruction-level parallelism
  - DCU and ACU
- Optimisations
  - Poor compilation unaffordable

About Retargetability

Rapid set-up of a compiler will boost for algorithm developers wishing to evaluate the efficiency of application code on different existing architectures.

Retargetability permits architecutre exploration. The processor designer is able to tune the architecture to run efficiently for a set of source applications in a particular domain.



Levels of Retargetability

- Automatically retargetable
- Compiler user retargetable
- Compiler developer retargetable

Processor Modelling Languages

- Mimola (HDL)
  - Netlists display an explicit activation of functional components by bits in the instruction word.
- nML
  - Describes behavioural mechanics rather than structural detail.
  - Description of operations, storage elements, binary and assembly syntax, and an execution model.
  - Based on synchronous register-transfer model
- Instruction Set Graph (ISG)
  - Associates behavioural information with structural information.

Principal Compiler Tasks

- Instruction-set matching and selection
- Register allocation and assignment
- Instruction scheduling and compaction

Instruction-Set Matching and Selection

Instruction set matching: Determine wide set of target instructions which can implement the source code.

Instruction set selection: Choose the best subset of instructions from the matched set.

A pattern based approach:

```
1: Produce a template base of patterns, each member represents
an instruction.
2: Translate the source program to a forest of syntax trees.
3: Match the trees to the pattern set.
4: A subset of all the matched patterns are selected to form the
implementation in microcode.
```


Register Allocation and Assignment

Register allocation: Determine a set of registers which may hold the value of a variable.

Register assignment: Determine a physical register which is specified to hold the value of a variable.

Solution based on graph colouring:

```
1: Build interference graph. (nodes=variables, edges=overlap)
2: Assign colours to each node. Adjacent nodes may not have the
same colour.
```

Drawback: Can not handle control-flow constructs (if, case, function calls, ...)
Special purpose registers complicate the matter.

Instruction Scheduling

Scheduling: Determine an order of execution of instructions.
Huge interdependence with instruction selection and register allocation.

Mutation scheduling:
```
1: Implementations of instructions can be regenerated by means
of a mutation set.
2: After generation of quadruples, calculate critical path.
3: Improve speed by identifying the instructions which lie on
critical paths and mutating them to other implementations which
allow a rescheduling of the instructions.
```

Integer Linear Programming (ILP):
```
1: Consider the following aspects: pattern-matching, scheduling,
register assignment and spilling to memory.
2: Dynamically make trade-offs between these based on an objec-
tive function and a set of constraints.
```
Common obj. func.: minimise time.
Common constraints: architecture characteristics.

Instruction Compaction

Compaction: Fine-grained scheduling to support instruction-level parallelism.

```
1: Define pseudo microinstructions and sequences of micro-oper-
ations with source and destinations properties.
2: Pack and upward past pseudo microinstructions to form real
microinstructions.
```


Optimisation for Embedded Processors

”Optimisations” which could reduce efficiency:
Common subexpression elimination -> increase register pressure.
Constant propagation -> too narrow instruction word

Loop optimisations (unrolling, pipelining, ...) are important.

Take processor characteristics into account!

Memory optimisations may lead to cost reduction.
- Narrowing of instruction words
- Paged memory
  - Reduce number of page changes
  - Long subroutines broken into several pieces
- Multi-memory allocation

Questions

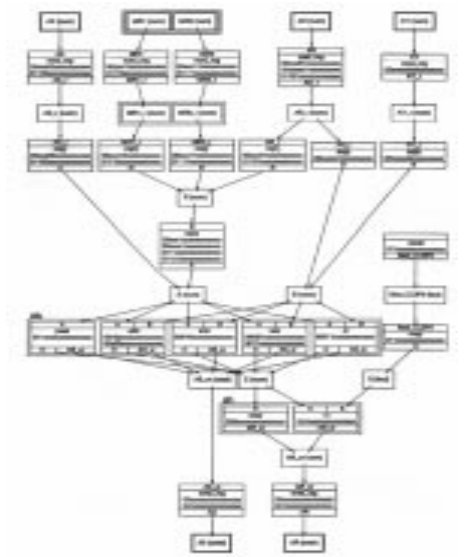
- What are the main differences between compilationfor general purpose processors and embedded processors?
- What are the principal compiler tasks?
- Can we just adopt optimisation techniques for general purpose compilers?

Instruction Set Graph (ISG) (1/2)

- Bipartite graph $G_{ISG} = \langle V_{ISG}, E_{ISG} \rangle$ with $V_{ISG} = V_S \cup V_I$ where V_S contains all vertices representing storage elements in the processor and V_I contains all vertices representing its operation types. Edges $E_{ISG} \subseteq (V_S \times V_I) \cup (V_I \times V_S)$ represent the connectivity of the processor.

- An *operation type* is a primitive processor activity.

Instruction Set Graph (ISG) (2/2)



Enabling and Encoding

- The set of instructions E_i that enables operation type i is called its *enabling condition*. $enabling: V_I \rightarrow 2^B$.

- Given a subset of operation types $V_{I_o} \subseteq V_I$, $enabling(V_{I_o}) = \cap_{i \in V_{I_o}} enabling(i)$ is the enabling condition for the set V_{I_o}.

- The set V_{I_o} has an *encoding conflict* if $enabling(V_{I_o}) = \emptyset$.

Storage Elements

- Static storage
- Transitory storage

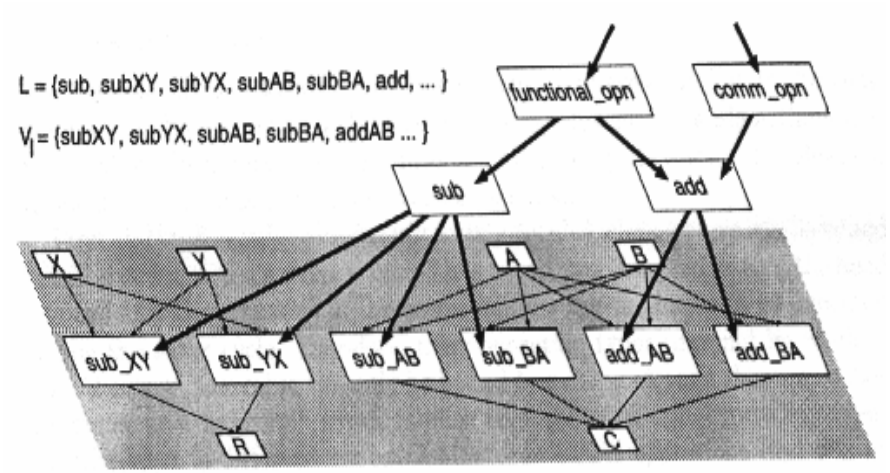
Memory: V_M
Registers: V_R
Transitories: V_T

Structural skeleton: $V_S = V_M \cup V_R \cup V_T$

Hardware Conflicts

- Hardware conflict = access conflict on transitory
- The function $resources: V_I \rightarrow 2^{V_T}$ returns the set of transitories that are written by operation type i.
- Operation types $V_{I_o} \subseteq V_I$ are free from structural hazards if $\forall i_i, i_j \in V_{I_o} \left| (i_i \neq i_j) \Rightarrow resources(i_i) \cap resources(i_j) = \varnothing \right.$.

Operation Type Hierarchy

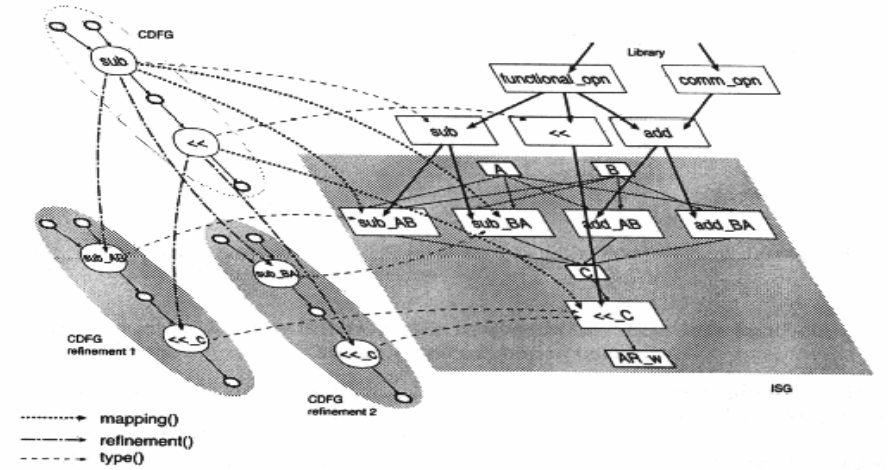


Code Generation

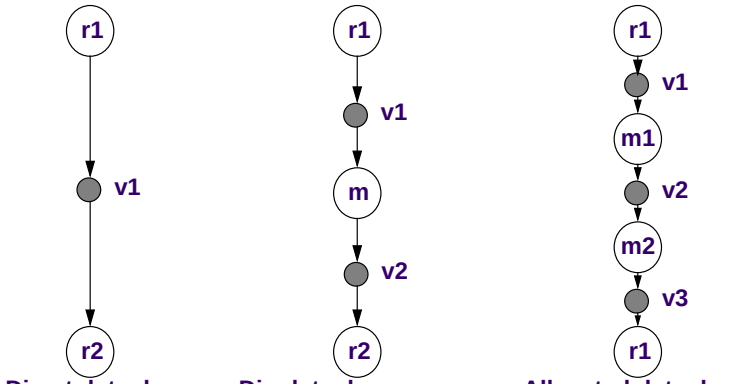
The source code is given as a dataflow graph (DFG):

- A dataflow graph is a bipartite graph $G_{DFG} = \langle V_{DFG}, E_{DFG} \rangle$, where $V_{DFG} = V_O \cup V_V$ with V_O representing CDFG operations and V_V representing the values they can produce and consume. The edges represent the dataflow.
- Code generation is mapping G_{DFG} onto G_{ISG} with values in V_V mapped on V_S and V_O mapped on V_I.

Refinement



Data Dependencies



Questions

- What is an ISG?
- How are hardware conflicts detected?
- How does code generation work?

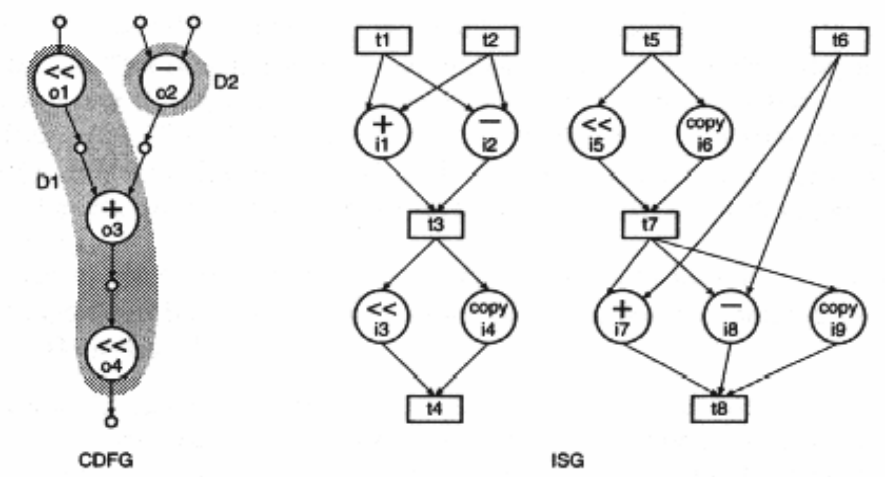
Bundles

- CDFG operations are grouped into bundles. Operations in a bundle have direct data dependency.
- All operations in a bundle are executed in the same clock cycle.

Code Selection

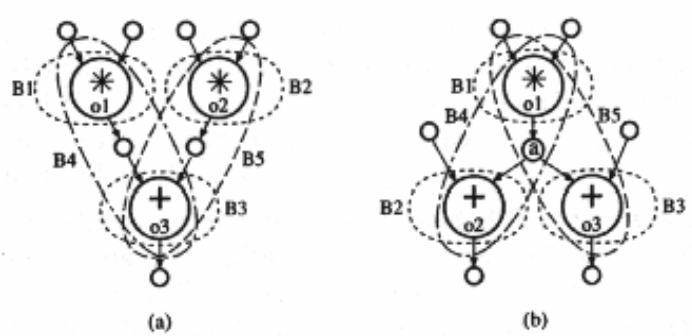
- Partition the CDFG into DAG patterns that can be implemented by a single instruction.
- Two subtasks:
  - Matching template patterns (NP-complete). Patterns may overlap.
  - Covering (NP-complete)

Matching



Covering

Cost function: Minimise cost = number of clock cycles. E.g. minimise number of extra moves.



{B4, B5} yields an illegal covering.

Questions

- What are the subtasks of code selection?

Summary

- SoC puts a challenge on retargetable compielers.
- Truths for general purpose no longer true.
- Examples of retargetable compilers, modelling processor with ISG.
- A lot more details, not brought up here..