Compiler, Linker, Loader

Hardware Software CoDesign

November 2011

Agenda

Compiler, Linker, Loader

1. Basic Definitions of a Compiler, Linker, Loader
2. Challenges in Compiler design
3. Basic requirements a compiler must fulfill
4. Brief on Compiler Architecture
5. Challenges for Compiler design for Embedded processors
6. Concept of a ReTargettable Compiler
7. Discussion on the Answering Machine Assignment (Group wise)
8. If time permits, open up the ARM CortexM3 processor documentation and readup.

Compiler, Linker, Loader

Introduction to Compiler

1. **What is a Compiler?** :: A *compiler* is a program that transforms a source program written in some high-level programming language (such as C) into machine code for some computer architecture (such as ARM).

2. The generated machine code can be later executed many times against different data each time.

3. The translation (transformation) of the source code from a high-level programming language to an executable or machine code can happen in steps :-
   - conversion into the target language, often having a binary form called object code.
   - conversion from object code into machine code (executable)

4. If the compiled program can run on a computer whose CPU or OS is different from the one on which the compiler runs, the compiler is known as a *cross-compiler*.

5. Compare this with an *interpreter* which reads an executable source program written in a high-level programming language as well as data for this program, and it runs the program against the data to produce some results. One example is the Unix Shell intrepretor, which runs operating systems commands interactively.

6. GROUP DISCUSSION ::
   - Enumerate available compilers.
   - Why do we need compilers in the first place.

Compiler, Linker, Loader

What are the Challenges? – Many Variations

1. many programming languages (eg. FORTRAN, C, C++, Java)
2. many programming paradigms (eg. object-oriented, procedural, functional, logic)
3. many computer architectures (eg. MIPS, ARM, SPARC, Intel, SHARC)
4. many operating systems (eg. Linux, Solaris, Windows, Android)

Compiler, Linker, Loader

What are the Qualities of a Compiler?

1. the compiler itself must be bug-free
2. it must generate correct machine code
3. the generated machine code must run fast
4. the compiler itself must run fast (compilation time must be proportional to program size)
5. the compiler must be portable (ie., modular, supporting separate compilation)
6. it must print good diagnostics and error messages
7. the generated code must work well with existing debuggers
8. must have consistent and predictable optimization

Compiler, Linker, Loader

What is requirement for design of a Compiler?

Knowledge of :-

1. programming languages (parameter passing, variable scoping, memory allocation...)
2. automata theory
3. algorithms and data structures (hash tables, graph algorithms, dynamic programming...)
4. computer architecture (assembly programming)
5. software engineering

Compiler, Linker, Loader

Brief on Compiler Architecture

1. Suppose you want to build compilers for m programming languages and you want to run them on n different architectures.

2. If it is done natively, there is a requirement to write $m * n$ compilers, one for each language-architecture combination.

3. The HOLY GRAIL of portability in compilers is to do the same thing by writing $m + n$ programs only. HOW ?

4. Define and use a universal *Intermediate Representation (IR)* and make the compiler two phases. An IR is typically a tree-like data structure that captures the basic features of most computer architectures.

5. Eg. representation of say an instruction $d \leftarrow s_1 + s_2$, that gets two source numbers and produces one destination number.

6. The front-end of the compiler maps the source code into IR and the second phase, called back-end, maps IR into machine code.

7. If this is followed, ideally there will be only $m + n$ components.

8. The challenge is to define the IRs such that all language and machine features are captured properly.

Compiler, Linker, Loader

Brief on Compiler Architecture – Multiple Phases in Front-End and Back-End

A typical real-world compiler has multiple phases. This increases the compiler's portability and simplifies re-targetting.

The following phases in front-end :-

1. *scanning:* a scanner groups input characters into tokens;
2. *parsing:* a parser recognizes sequences of tokens according to some grammar and generates *Abstract Syntax Trees (ASTs)*;
3. *Semantic analysis:* performs *type checking* (ie., checking whether the variables, functions etc in the source program are used consistently with their definitions and with the language semantics0 and translates ASTs into IRs;
4. *Optimization:* optimization IRs.

The following phases in the back-end :-

1. *instruction selection:* maps IRs into assembly code;
2. *code optimization:* optimizes the assembly code using control-flow and data-flow analysis, register allocation, etc;
3. *code emission:* generates machine code from assembly code.

Compiler, Linker, Loader

Brief on Compiler Architecture – Multiple Phases in Front-End and Back-End

The generated machine code in written in an object file. This file is not executable since it may refer to external symbols (such as system calls). The operating system provides the following utilities to execute the code (in a computer system – this is slightly different in an embedded system).

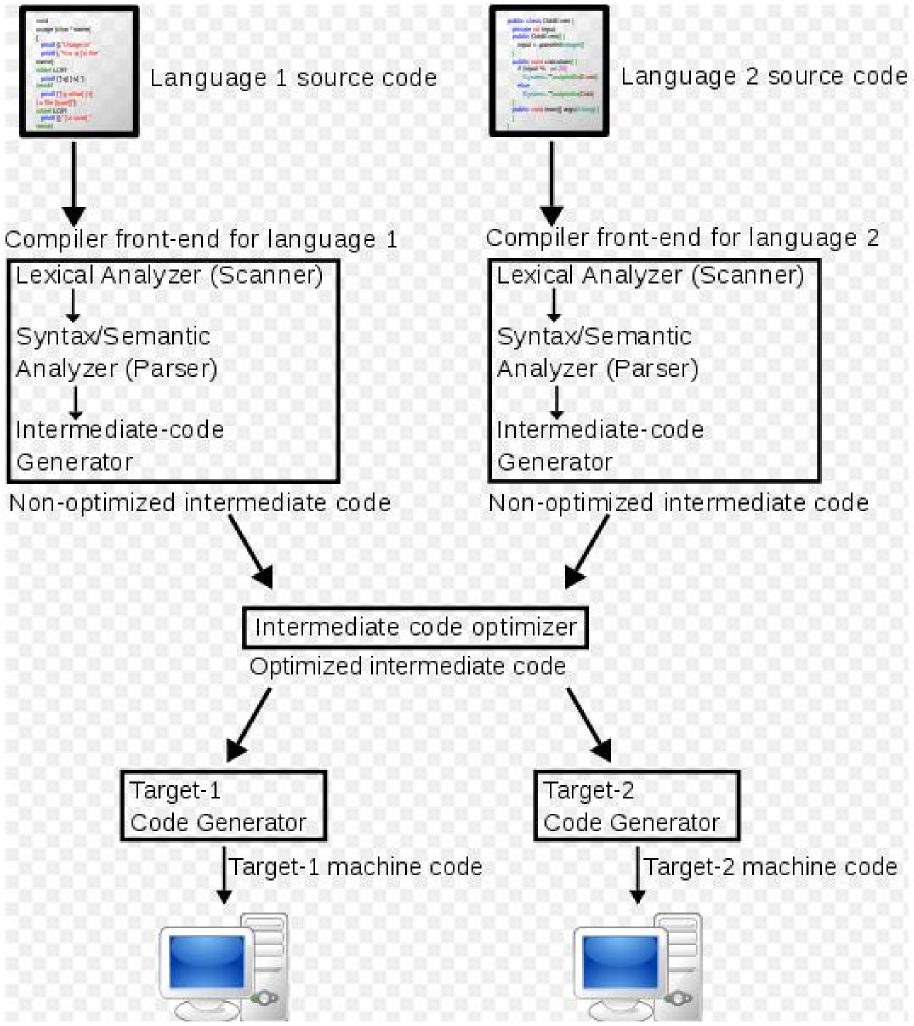
1. *linking:* A linker takes several object files and libraries as input and produces one executable object file.

   $\rightarrow$ It retrieves from the input files (and puts them together in the executable object file) the code of all the referenced functions / procedures and it resolves all external references to real addresses. The libraries include the operating system libraries, the language- specific libraries, and user-created libraries.

2. *loading:* A loader loads an executable object file into memory, initializes the registers, heap, data, etc and starts the execution of the program.

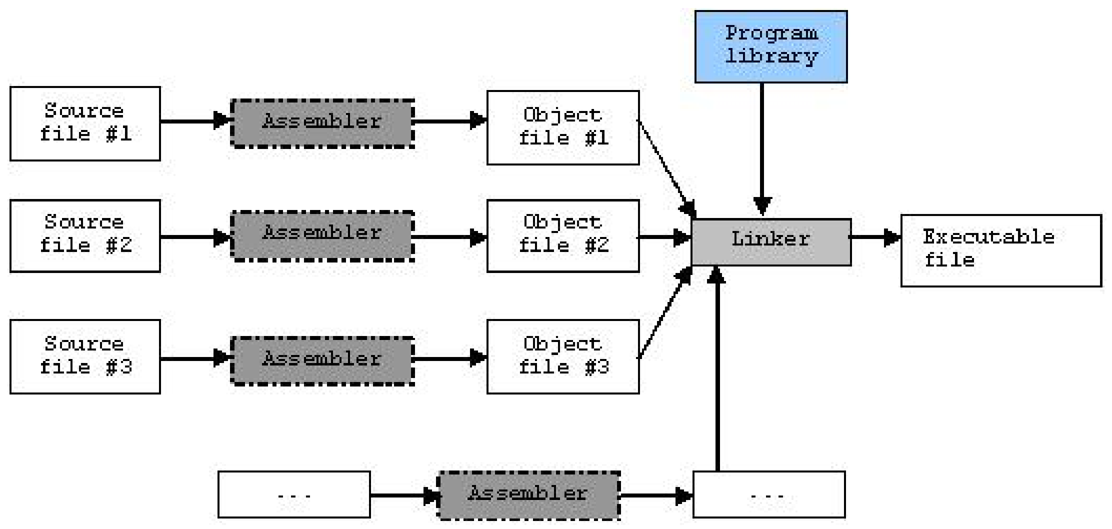
Compiler, Linker, Loader

Compiler Architecture – Multiple Phases in Front-End and Back-End


Language 1 source code
Language 2 source code
Compiler front-end for language 1
Compiler front-end for language 2
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Non-optimized intermediate code
Intermediate code optimizer
Optimized intermediate code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine code

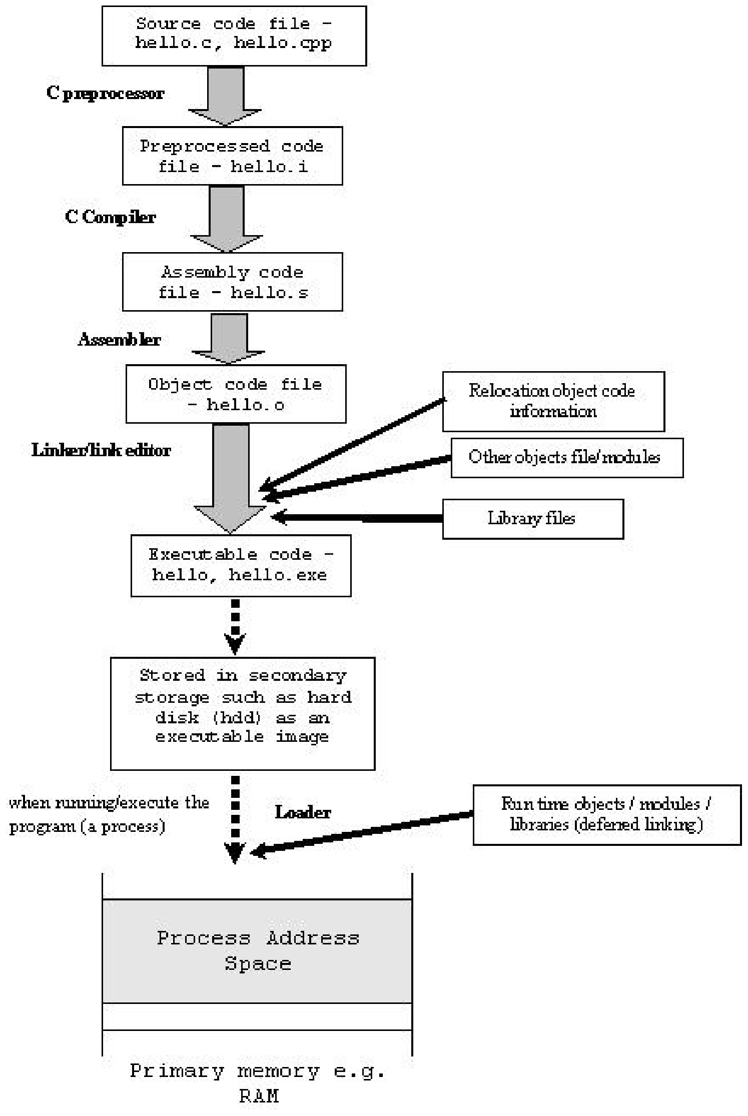
Compiler, Linker, Loader

Compiler Architecture – Multiple Phases in Front-End and Back-End



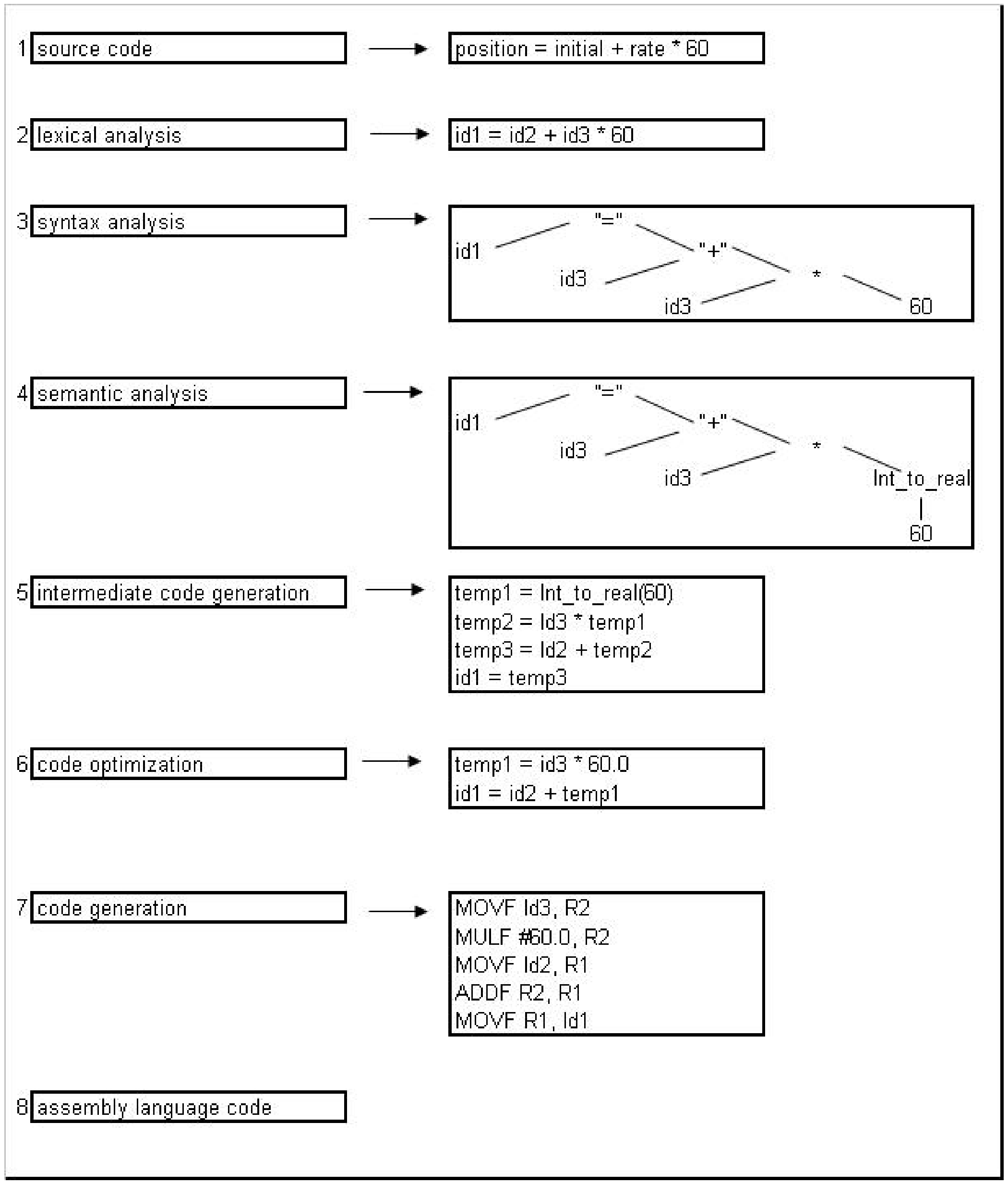
Compiler, Linker, Loader

Compiler Architecture – Multiple Phases in Front-End and Back-End


Source code file - hello.c, hello.cpp
C preprocessor
Preprocessed code file - hello.i
C Compiler
Assembly code file - hello.s
Assembler
Object code file - hello.o
Relocation object code information
Other objects file/modules
Library files
Linker/link editor
Executable code - hello, hello.exe
Stored in secondary storage such as hard disk (hdd) as an executable image
when running/execute the program (a process)
Loader
Run time objects / modules / libraries (deferred linking)
Process Address Space
Primary memory e.g. RAM

Compiler, Linker, Loader

Brief on Compiler Architecture – Steps in Compilation



Compiler, Linker, Loader

Compiler Architecture – Problem for Compilation to Embedded Processors

1. *ReTargetability:* Typically, retargetting to a new architecture is confined to the final code generation phase. This means that the IR (intermediate representation) must closely represent the final target machine to produce efficient code.

   → In a scenario where there are "Application Specific Instruction Sets" and where the embedded processor instruction sets vary widely in composition, a general (generic) form of IR can be difficult to conceptualize.

2. *Register Constraints:* Embedded processors contain a number of special purpose registers. Registers are reserved for special functionality to maintain low instruction widths. The instruction width directly correlates with the program space. The register constraints can affect all the phases of compilation.

3. *Arithmetic Specialization:* The typical 3-tupule code artificially decomposes data-flow operation into smaller pieces. Arithmetic operations which require more than 3 operands are not naturally handled with 3-tupule code. However, greater than 3 operands occur frequently in DSP architecture.

4. *Instruction-Level Parallelizm:* Architectures with parallel executing engines require different compilation techniques. eg. A DSP typically has both Data Calculation Unit (DCU) and Address Calculation Unit (ACU).

   → A compiler should take into account the possibility to perform operations on different functional units, as well as choose the most compact solution.

5. *Optimization:* Real Time embedded software cannot afford to have performance penalties as a result of poor compilation. Efficient compilation is only arrived upon by many optimization algorithms.

Compiler, Linker, Loader

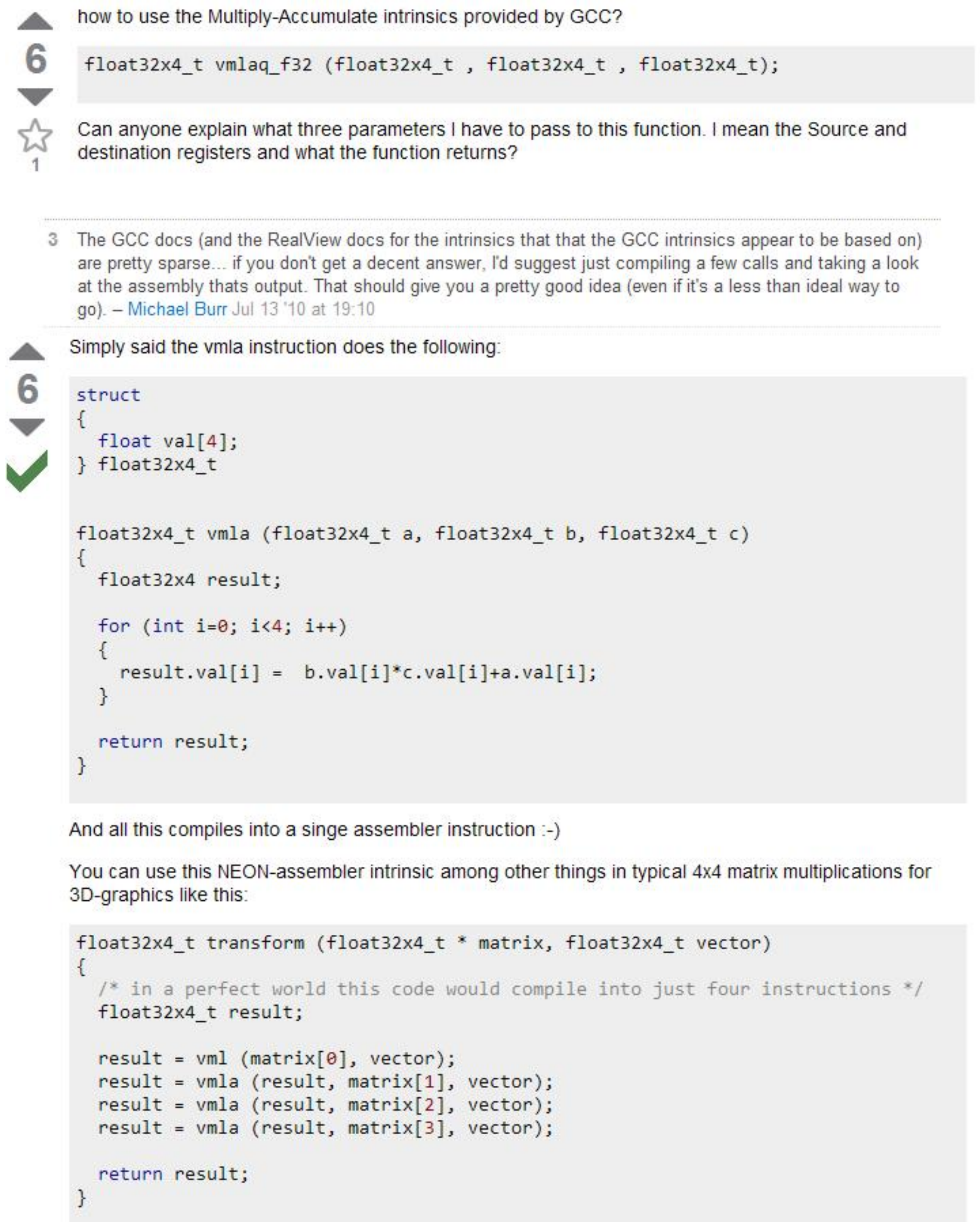
Compiler ReTargetability

Take an Example of a Typical compiler (say gcc)

1. Freely distributed through the FSF (Free Software Foundation)
2. With free C source code, its been ported to many machines – Intel, Sun..
3. Also its been retargetted to several DSPs like ADI2101, Lucent1610, Motorola56001, STD950,...
4. GCC has become a de-facto approach to develop compilers quickly from freely available sources.
5. GCC strengts lie in the complete set of architecture independent optimizations: common subexpression removal, dead code elimination, constant folding, constant propagation, basic code execution and others.
6. HOWEVER, for embedded processing, it is important that optimizations be applied according to the character of target architectures (and ability to exploit the architecture). GCC has little provisions for enabling optimizations to the target machine. Hence performance falls short of acceptable code quality.
7. Embedded processors have usually few registers, heterogeneous register structures, unusual wordlengths, and other architectural specializations.
8. Till Recently, a formal re-targetability model was not fully adopted primarily because general purpose CPUs have long lifetimes that do not justify the effort required to make the compilers re-targetable.

Compiler, Linker, Loader

Compiler ReTargetability – Example from recent times



6

1

how to use the Multiply-Accumulate intrinsics provided by GCC?

```
float32x4_t vmlaq_f32 (float32x4_t , float32x4_t , float32x4_t);
```

Can anyone explain what three parameters I have to pass to this function. I mean the Source and destination registers and what the function returns?

3 The GCC docs (and the RealView docs for the intrinsics that that the GCC intrinsics appear to be based on) are pretty sparse... if you don't get a decent answer, I'd suggest just compiling a few calls and taking a look at the assembly thats output. That should give you a pretty good idea (even if it's a less than ideal way to go). – Michael Burr Jul 13 '10 at 19:10

6

Simply said the vmla instruction does the following:

```
struct
{
  float val[4];
} float32x4_t


float32x4_t vmla (float32x4_t a, float32x4_t b, float32x4_t c)
{
  float32x4 result;

  for (int i=0; i<4; i++)
  {
    result.val[i] =  b.val[i]*c.val[i]+a.val[i];
  }

  return result;
}
```

And all this compiles into a singe assembler instruction :-)

You can use this NEON-assembler intrinsic among other things in typical 4x4 matrix multiplications for 3D-graphics like this:

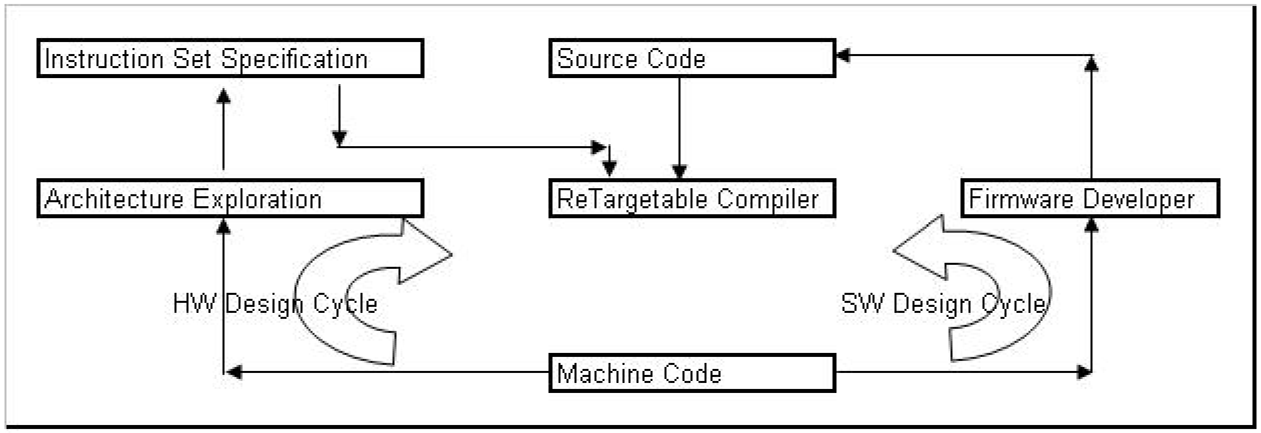
```
float32x4_t transform (float32x4_t * matrix, float32x4_t vector)
{
  /* in a perfect world this code would compile into just four instructions */
  float32x4_t result;

  result = vml (matrix[0], vector);
  result = vmla (result, matrix[1], vector);
  result = vmla (result, matrix[2], vector);
  result = vmla (result, matrix[3], vector);

  return result;
}
```

Compiler, Linker, Loader

ReTargetable Compiler – concept



1. Retargetability allows the rapid setup of a compiler to a specific processor. This can be an enormous boost for algorithm developers wishing to evaluate the efficiency of application code on different existing architectures.

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

3. This however, requires the need for *instruction set specification languages* where the user can describe the functionality of a processor in a formal fashion.

4. Then, the transformations of a compiler may be returned according to the architecture model (or definition).

5. One example of a retargetable compiler is MIMOLA - Machine Independent Micro-programming LAnguage.

Compiler, Linker, Loader

Example of Register Allocation – concept

Example - For the following 2 statement program segment, determine a smart register allocation scheme:

$$A = B + C * D$$
$$B = A - C * D$$

Simple Register Allocation
LOD (R1,C)
MUL (R1,D)
STO (R1,Temp)
LOD (R1,B)
ADD (R1,Temp)
STO (R1,A)
LOD (R1,C)
MUL (R1,D)
STO (R1,Temp2)
LOD (R1,A)
SUB (R1,Temp2)
STO (R1,B)
Net Result
12 instructions and memory ref.

Smart Register Allocation	
LOD (R1,C)	
MUL (R1,D)	**C*D**
LOD (R2,B)	
ADD (R2,R1)	**B+C*D**
STO (R2,A)	
SUB (R2,R1)	**A-C*D**
STO (R2,B)	
Net Result	
7 instruc. And 5 mem. refs.	

Compiler, Linker, Loader

Acknowledgements

1. Hardware / Software Co-Design - Principles and Practice :: J. Staunstrup & W. Wolf :: Ch 5

2. http://lambda.uta.edu/cse5317/notes/notes.html

3. A retargetable compiler for a high-level microprogramming language :: Peter Marwedel :: Proceedings of the 17th annual workshop on Microprogramming, 1984.

4. ARM Assembler Reference :: www.arm.com