# FP8-17: Software Programmable Signal Processing Platform Analysis

Andrzej Wąsowski

---

# Software Programmable DSP Platform Analysis

Episode 2, Tuesday 21 March 2006

Contents, Goals, and Administrivia

## Compilation environment

Preprocessor, Compiler, Assembler & Linker

## Compiler Architecture

## Lexical Analysis

Tokens, Regular Expressions

## Syntactical Analysis

Context Free Grammars, Derivations
Parse Trees

---

# Contents

- Structure of a compiler
- Architecture and instruction set of DSPs/VLIW
- Implementation of a compiler for DSPs
- Lexical analysis
- Parsing
- Diagnostics
- Register allocation
- Code selection
- Code optimization

---

# Goals

You will

- learn the C programming language better
- understand compilation error messages
- know abilities and limitations of compilers
- be able to program more efficiently by:
  - producing more efficient code.
  - using less time for development.
- understand compiler documentation
- be able to choose compiler options
- be able evaluate compiler's suitability for your application.
- learn objective functions for code optimization.

# Non objectives

You will not

- be able to modify existing compilers without excessive effort or additional introduction.
- be able to implement a compiler from scratch,
- know how to implement advanced features of contemporary languages like: objects, polymorphism, garbage-collectors, aspects, higher order functions, etc.
- learn programming languages theory (type systems, semantics, etc)
- learn mathematical linguistics (regular, context-free languages, etc)

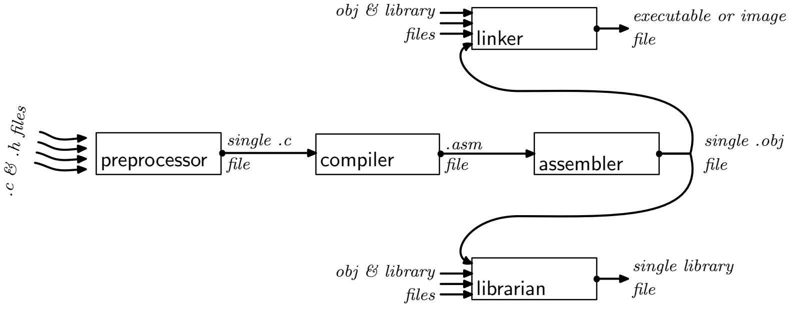
---

# The course, teachers, etc.

- Teachers:
  - Andrzej Wąsowski (compilers)
  - Ole Wolf (architecture)
  - Andreas and Michael (instructors)
- http://www.itu.dk/˜ wasowski/teach/dsp-compiler-06
  - schedule, exercise sheets, slides and news
- Text: Appel. *Modern Compiler Implementation in C* + website.
- Each module = 90 min. lecture + 90 min. tutorial
- Do ask questions during lectures.
- In depth understanding requires devoting more time to the exercises than 90min.

---

# Compilation Environment



---

# Compilation Environment (II)

- Preprocessor expands macrodefinitions (`#define`’s), joins continued lines, removes comments (in C), includes files (`#include`).
- Compiler translates a single source file into assembly file
- Assembler translates .asm file to a binary .o file
- Linker consolidates bits and pieces into a single program.
- Modern linkers can perform global program optimizations, too.

# Compilation Environment: Example

hello.c

```
#define MSG "Hello, world!\n"
extern int printf(const char *format, ...);
/* A comment before the main function */
int main(int argc, const char * argv[])
{
    printf( MSG );
    return 0;
}
```

requires: preprocessing, compiling, assembling and linking with the startup code and the C library.

# Example preprocessed

hello.c

```
extern int printf(const char *format, ...);

int main(int argc, const char * argv[])
{
    printf( "Hello, world!\n" );
    return 0;
}
```

Expanded macros, removed comments, included files (not in this example).

# Example compiled

Hello.c compiled wit GCC for x86, giving hello.s:

```
        .file    "hello.c"
        .section .rodata
.LC0:   .string  "Hello, world!\n"
        .text
.globl main
        .type main, @function
main:   pushl %ebp
        movl  %esp, %ebp
        pushl $.LC0
        call  printf
        leave
        movl  $0, %eax
        ret
```

# Example compiled (II)

- The compilation step is our main point of interest.
- The C program is translated into a flat list of simple instructions.
- Instructions and addresses are symbolic (mnemonics and labels).

# Example compiled III

Hello.c compiled with TI’s **cl6x** giving hello.asm (fragment):

```
SL1: .string "Hello, world!",10,0
     CALL .S1 _printf
     STW .D2T2 B3,*SP-(16)
     MVKL .S2 RL0,B3
     MVKL .S1 SL1+0,A3
     MVKH .S1 SL1+0,A3
     STW .D2T1 A3,*+SP(4)
     || MVKH .S2 RL0,B3 ;CALL OCCURS
RL0: LDW .D2T2 *++SP(16),B3
     ZERO .D1 A4
     NOP 3
     RET .S2 B3
```

---

# Example compiled (IV)

- The 67xx assembly is different from x86.
- Compiler translates a portable code to a platform specific one.
- Some instructions are put in parallel (STW||MVKH).
- NOP (no operation) instructions are inserted.
- Seemingly nonlinear execution (call place and parameter passing).

---

# Example assembled

- Assembler resolves symbolic addresses and translates symbolic instructions to binary values.
- External symbols remain unresolved.
- On the next slide statistics for the object file hello.o assembled from hello.s (GNU C/x86).

---

# Example assembled (II)

```
SYMBOL TABLE:
00000000 l df *ABS* 00000000 hello.cpp
00000000 l d .text 00000000
00000000 l d .data 00000000
00000000 l d .bss 00000000
00000000 l d .rodata 00000000
00000000 l d .eh_frame 00000000
00000000 l d .note.GNU-stack 00000000
00000000 l d .comment 00000000
00000000 g F .text 00000023 main
00000000 *UND* 00000000 printf
00000000 *UND* 00000000 __gxx_personality_v0
```

# Example assembled (III)

- This object (.o) file needs to be linked with the C library or another .o file that provides the `printf` function.
- In modern compilers the assembling stage is often incorporated in the compiler.

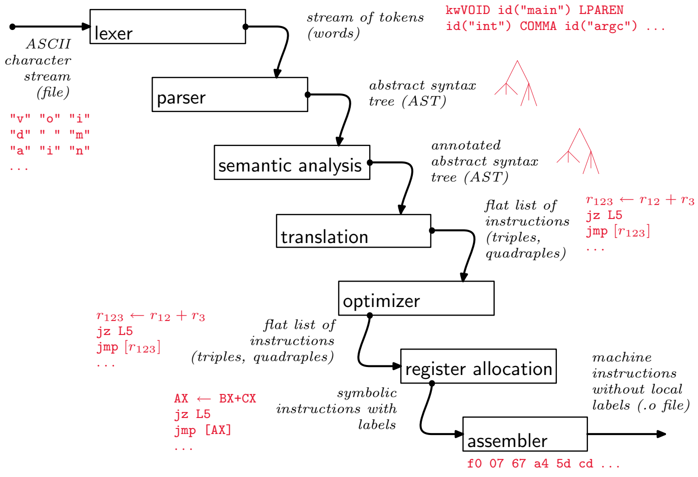
---

# Architecture of a compiler

- Compilers are divided into layers, called *stages* or *passes*.
- A stage inputs some program representation, processes it and outputs a another representation.
- The first stage typically inputs text files. The last stage typically outputs machine code, eg. an image that can be stored in EEPROM or a binary file that can be executed on a PC.
- The front stages perform analyses, while the late stages perform syntheses.

---

# Architecture of a compiler (II)



---

# Lexical analysis: Tokens

- A source program is represented as a sequence of characters
- A lexical analyzer (a lexer) breaks the sequence of characters into a sequence of corresponding tokens (like “words”).

| | |
|---|---|
| ID | `foo n14 last` |
| NUM | `73 0 00 515 082` |
| REAL | `66.1 .5 10.  1e67 5.5e-10` |
| IF | `if` |
| NOTEQ | `!=` |
| LPAR | `(` |
| RPAR | `)` |

# Lexical analysis: Tokens (continued)

The program

```
float match0(char *s)
{ /* find a zero */
    if (!strncmp(s,"0.0", 3))
    return 0.;
}
```

is translated to:

FLOAT ID(*match0*) LPAREN CHAR STAR ID(*s*)
RPAREN LBRACE IF LPAREN BANG ID(*strncmp*)
LPAREN ID(S) COMMA STRING(`0.0`) COMMA NUM(3)
RPAREN RPAREN RETURN REAL(`0.0`) SEMI RBRACE
EOF

---

# Lexical analysis: Lexer (continued)

- Lexer also removes comments (done by the preprocessor in C)
- Lexer removes white space from the code
- What are the words we need? How do we specify them?

---

# Describing Tokens

*An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs newlines, and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and constants.*

- How do we detect identifiers?
- We need a precise way to describe them first.
- **Regular expresssions** offer such a way.

---

# Regular Expressions

| | |
|---|---|
| **a** | An ordinary character stands for itself |
| $\varepsilon$ | The empty string. |
| $M \| N$ | Alternation, chosing from $M$ or $N$ |
| $M \cdot N$ | Concatenation, $M$ followed by an $N$ |
| $M^*$ | Repetition zero or more times, Kleene’s closure |
| $M^+$ | Repetition one or more times |
| $M?$ | Optional |
| $[a-zA-Z]$ | Character set |
| . | Any single character except newline |

The longest prefix of current input that can match any regular expression is taken as the next token.

# Examples of Regular Expressions

| | |
|---|---|
| `if` | an if keyword (IF) |
| $[a-z][a-z0-9]^*$ | a simple identifier (ID), note: no capital letters |
| $[0-9]^+$ | a decimal number (NUM) |
| $([0-9]^+"."[0-9]^*)\|\|([0-9]^*"."[0-9]+)$ | a real number (REAL) |
| $("//"[a-z]^*"\backslash n")\|\|(""" \|\| "\backslash n" \|\| "\backslash t")^*)^*$ | whitespace and one line comment |

How can we describe the C identifier token?

---

# Lexer Generators

- Lexer generator: given regular expressions for token types generate a lexer translating a stream of characters to a stream of tokens.
- by translating regular expressions to deterministic finite automata, similar to Mealy machines.
- The translation algorithm is standard (Appel, section 2.3–2.4)
- A popular free lexer generator targeting C is flex (see also lex in Appel, section 2.5).
- There exist such tools for any general purpose programming language.

---

# Straight-Line Programs

```
a := 5+3;
b := (print (a, a+1), 10+a);
print(b)
```

produces

```
8 9
18
```

---

# A Sample Straight-Line Programs

```
a := 5+3;
b := (print (a, a+1), 10+a);
print(b)
```

Token representation returned by a lexer:

ID(a) ASSGN DEC(5) PLUS DEC(3) SEMI ID(b) ASSGN LPAR PRINT LPAR ID(a) COMMA ID(a) PLUS DEC(1) RPAR COMMA DEC(10) PLUS ID(a) RPAR SEMI ...

- How do we decide whether this token stream constitutes a legal program?
- How do we translate it to a tree?

---

# Syntactical Analysis: Parsing

- A parser inputs the stream of tokens produced by the lexer.
- The tokens are analyzed and translated into an **Abstract Syntax Tree**
- This analysis is performed by finding a deriviation of the program with respect to a **context free grammar** of the source language.

---

# Syntactical Analysis: Parsing (II)

- A context free grammar is a set of production rules describing the language's syntax.
- A production:

$$\textit{symbol} \rightarrow \textit{symbol symbol} \ \ldots \ \textit{symbol}$$

- where *symbol* is either a token, called a **terminal** symbol now,
- or a **nonterminal** symbol.

---

# A Grammar for SL Programs

Stmnt → Stmnt SEMI Stmnt
Stmnt → ID ASSGN Expr
Stmnt → PRINT LPAR List RPAR

Expr → ID
Expr → DEC
Expr → Expr PLUS Expr
Expr → LPAR Stmnt COMMA Expr RPAR

List → Expr

List → List COMMA Expr

Terminals are capitalized. Nonterminals arex *Stmnt*, *Expr*, *List*. *Stmnt* is the start symbol. See also Grammar 3.1, p. 41 in Appel.

It is convenient to use literals instead of tokens:

$_1$ Stmnt → Stmnt ; Stmnt
$_2$ Stmnt → ID := Expr
$_3$ Stmnt → `print` ( List )

$_4$ Expr → ID
$_5$ Expr → DEC
$_6$ Expr → Expr + Expr
$_7$ Expr → ( Stmnt , Expr )

$_8$ List → Expr

$_9$ List → List , Expr

**A stream of tokens is a syntactically legal SL program if it can be derived using these rules.**

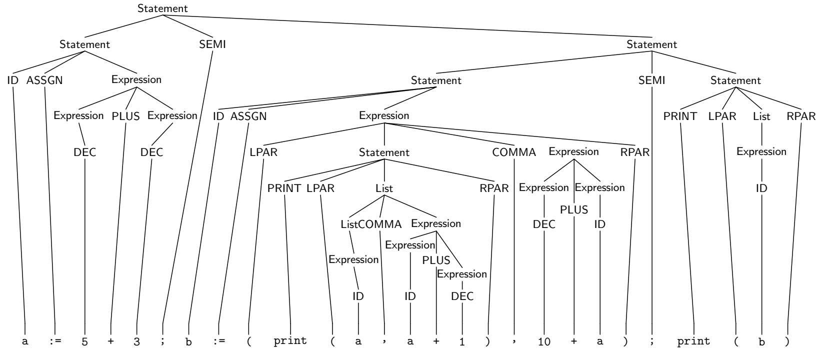
---

# Rightmost Derivation (example)

Stmnt $\rightarrow_1$ Stmnt ; Stmnt $\rightarrow_1$ Stmnt ; Stmnt ; Stmnt
$\rightarrow_3$ Stmnt ; Stmnt ; `print(` List )
$\rightarrow_8$ Stmnt ; Stmnt ; `print(` Expr )
$\rightarrow_4$ Stmnt ; Stmnt ; `print(b)`
$\rightarrow_2$ Stmnt ; `b:=`Expr ; `print(b)`
$\rightarrow_7$ Stmnt ; `b:=(`Stmnt, Expr `); print(b)`
$\rightarrow_6$ Stmnt ; `b:=(`Stmnt, Expr + Expr `); print(b)`
$\rightarrow_4$ Stmnt ; `b:=(`Stmnt, Expr + `a); print(b)`
$\rightarrow_5$ Stmnt ; `b:=(`Stmnt`,10+a); print(b)`
$\rightarrow_3$ Stmnt ; `b:=(print(` List `),10+a); print(b)`
$\rightarrow_9$ Stmnt ; `b:=(print(` List, Expr `),10+a); print(b)`
$\rightarrow_6$ Stmnt ; `b:=(print(`List, Expr+Expr `),10+a); print(b)`
$\rightarrow_5$ Stmnt ; `b:=(print(`List , Expr `+1),10+a); print(b)`
$\rightarrow_4$ Stmnt ; `b:=(print(`List`,a+1),10+a); print(b)`
$\rightarrow_8$ Stmnt ; `b:=(print(`Expr`,a+1),10+a); print(b)`
$\rightarrow_4$ Stmnt ; `b:=(print(a,a+1),10+a); print(b)`
$\rightarrow_2$ `a:=`Expr ; `b:=(print(a,a+1),10+a); print(b)`
$\rightarrow_6$ `a:=`Expr+Expr ; `b:=(print(a,a+1),10+a); print(b)`
$\rightarrow_5$ `a:=`Expr`+3; b:=(print(a,a+1),10+a); print(b)`
$\rightarrow_5$ `a:=5+3; b:=(print(a,a+1),10+a); print(b)`

---

# Parse Trees



A sanitized parse tree (also called abstract syntax tree, or AST) is the first, and perhaps most important form of the program representation in the entire compilation process.

---

# Parser Generators

- The process of parsing is a reverse of constructing a derivation.
- A parser is usually implemented as a push-down automaton (stack automaton).
- There exists several construction algorithms. See more in Appel, sections 3.2–3.3.
- Modern parsers are rarely hand-written.
- Parser generators translate grammars into programs that read tokens and build parse trees
- Popular parser generators are yacc, bison, JavaCC, jjtree, ANTLR, . . .
- Such tools exist for all popular languages.