Interprocedural Data Flow Decompilation

Cristina Cifuentes*
Department of Computer Science, University of Tasmania
GPO Box 252C, Hobart TAS 7001, Australia
Email: C.N.Cifuentes@cs.utas.edu.au

Abstract

Traditional compiler data flow analysis techniques are used to transform the intermediate representation of a decompiled program to a higher representation that eliminates low-level concepts such as registers and condition codes, and reintroduces the high-level concept of expression.

Summary data flow information is collected on condition codes and registers, and is propagated across basic blocks and subroutine boundaries to find boolean and arithmetic expressions, register arguments, function return registers, actual arguments, and propagate data types whenever required. The elimination of condition codes is performed by an extension of a reach algorithm. The elimination of registers and intermediate instructions is performed by an extended copy propagation algorithm that is based on intra and interprocedural analysis of the program's control flow graph.

The methods presented in this paper have been implemented in dcc, a prototype decompiler for the Intel i80286 architecture. Experimental results have proved to reduce the number of intermediate instructions by over 75% on average for this particular CISC machine.

1 Introduction

A decompiler is a software tool that attempts to reverse the compilation process by translating an input pure binary program to an equivalent target high-level language (HLL) program. The input program does not have symbolic information within it, and the HLL used to compile this binary program does not need to be the same as the target HLL produced by the decompiler.

Although decompilers have not been greatly studied in the literature, there are a variety of applications that could benefit from them, including the obvious maintenance of old code and recovery of lost source code, but also the debugging of binary programs, migration of applications to a new hardware environment, verification of generated code by the compiler, and translation of code written in an obsolete language.

The structure of decompilers is based on that of compilers; similar principles and techniques are used to perform the analysis of programs. In theory, the grouping of phases in a decompiler makes it easy to write different decompilers for different machines and target languages, by writing different front-ends for different machines, and different back-ends for different target languages. Hence, the phases of our prototype decompiler, dcc, were grouped into the following modules: the front-end; the machine dependent module, the UDM or universal decompiling machine; the machine and language independent module, and the back-end; the target language dependent module. This prototype decompiler was designed for a CISC machine, the Intel i80286, and produces target C

*This research was partly funded by Australian Research Council (ARC) Grant No.A49130261 while the author was with the Queensland University of Technology, Brisbane, Australia.

programs as output.

The core of the decompilation analysis is done by the UDM in a two phase process: the data flow and the control flow analysis. The former translates the intermediate code to a higher representation available in HLLs, by removing all references to condition codes, registers, and low-level instructions not available in HLLs. The latter structures the underlying control flow graph of each subroutine into a generic set of HLL constructs available in any imperative language, making minimal use of the `goto` statement[1, 2]. This paper concentrates on the data flow analysis phase.

Conventional data flow analysis collects information about the way variables are used in a program, and summarizes it in the form of sets. In decompilation, this information is used to transform and improve the quality of the intermediate code, preserving the meaning of the program as with standard compiler data flow analysis[3].

1.1 Previous Work

Little work has been done in the area of data flow analysis of a decompiler, mainly due to the limitations placed on many of the decompilers available in the literature, including: decompilation of assembler source files[4, 5, 6, 7], decompilation of object files with symbolic debugging information[8], and the compiler specification requirements to build a decompiler[9, 10, 11]. Data flow analysis is essential when decompiling pure binary files, as a great amount of information is lost during the compilation and linking process.

In the area of flag or condition code analysis, DeJean and Zobrist formulated an optimization of flag definitions by means of a reach algorithm[12]. This method was used in a program which translated microprocessor object code for the i8085 into a behaviorally equivalent PL/1 program, and eliminated over 50% of the initial flag definitions; generating PL/1 programs that defined only the necessary flags used by a later condition.

In the area of register elimination, a method of *text compression* was presented by Housel[4] for the elimination of intermediate loads and stores. This method performed forward and backward substitution of registers in assignment instructions, if the result was not busy within the current basic block, and provided a reduction of instructions of up to 40% in assembly programs compiled with Knuth's MIXAL compiler. Hopwood described a method of *expression condensation* to combine two or more intermediate instructions into an equivalent expression by means of forward substitution. This method specified 5 necessary conditions and 6 sufficient conditions under which forward substitution of a variable or register can be performed, and was based on variable usage analysis[13]. No performance of the method was given.

The above mentioned methods were used to decompile toy languages which did not have any type of interprocedural information flowing across subroutines, hence, there was no need to check for register arguments, function return registers, or actual arguments, and how to incorporate this information in the analysis process. In addition, no mention to `PUSH` and `POP` instruction is given in those methods.

2 Structure of a Decompiler

Figure 1 illustrates the structure of a decompiler. Once the executable program has been loaded, the program is parsed to generate the control flow graph of the program and the low-level intermediate code. This low-level intermediate representation is very similar to the assembler of the particular machine; that is, machine instructions are mapped to assembler mnemonics. This means that the initial representation of the program is in terms of registers, condition codes, and offsets from the stack.

The semantic analysis phase checks the input code for known idioms, such as long variable addition, and HLL prologue code. In this way, useful information is saved, and the low-level instructions are modified (whenever needed) into one low-level instruction that represents the idiom. Figure 2 describes the transformations involved

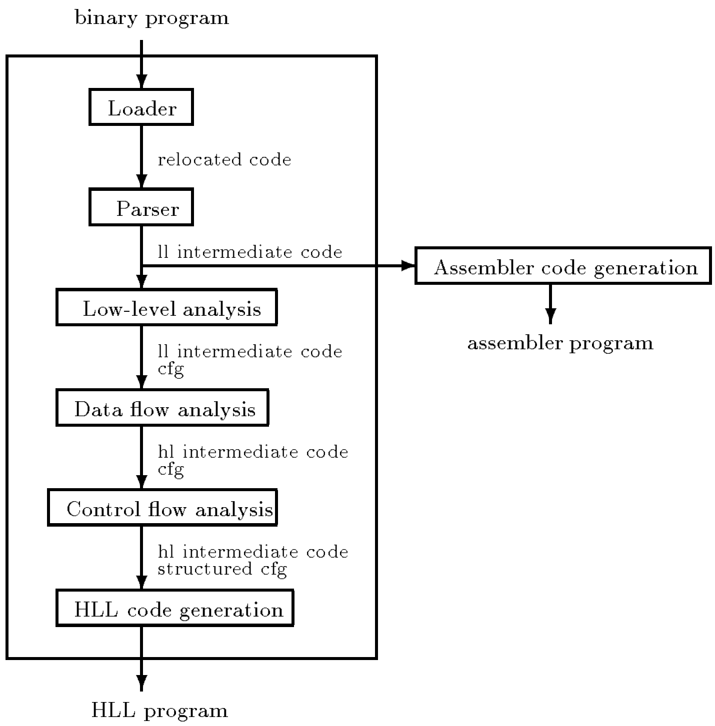

Figure 1: Structure of a Decompiler. The boxes represent the different stages of the decompiler, and the text next to the arrows represents the intermediate representation flowing between the stages.

for two idioms found on the Intel architecture. In the first case, long variable addition, we now know that registers `dx` and `ax` are temporarily used as a long register, and that local stack variables at offsets -2 and -4 are a long variable (i.e. its size is 4 bytes). On the second example, HLL prologue code, all HLL subroutines use the illustrated code on the Intel architecture; we now know that the subroutine is most likely a high-level routine, and that it uses 6 bytes of local variables.

```
                                   push bp
    add ax, [bp-4]                 mov  bp, sp
    adc dx, [bp-2]                 sub  bp, 6

                     ⇓

add dx:ax, [bp-2]:[bp-4]           enter 6,0
```

Figure 2: Sample Idioms and their Transformation

The data flow analysis phase performs two different analyses to improve the quality of the intermediate code. The first analysis eliminates the need for condition codes by transforming a group of instructions into one equivalent high-level instruction that preserves the meaning of the previous instructions. The second analysis removes all temporarily used registers and regenerates high-level expressions. These analyses are the purpose of this paper and are explained in detail in Section 4.

The control flow analysis phase structures each procedure's control flow graph into one that represents high-level control structures, such as `while()`, `repeat..until()`, `if..then[..else]`, and `loop`, as explained in [1, 2]. Once this is done, the code generator can generate code from the control flow graph and the high-level intermediate code, for the appropriate high-level language.

We have been working on a decompiler for the Intel i80286 architecture running under the DOS operating system, that takes as input **.exe** or **.com** files and produces C programs as output. This experimental decompiler has been named dcc, it is operational, and implements all of the ideas expressed in this paper.

The parsing phase of dcc classifies the low-level intermediate instructions into two different sets: the high-level instruction (HLI) set, which contains instructions that are likely to have been generated by a compiler, and the non high-level instruction (NHLI) set, which includes all other instructions (e.g. instructions that are likely to have been generated by hand-crafted assembly code, such as `SAHF`, and `AAM`). Instructions in the NHLI set are flagged as being so, as well as the subroutine that uses these instructions. In this way, we do not attempt to decompile subroutines that are untranslatable into a higher order representation, but produce assembler for them. From the 110 low-level intermediate instructions in dcc, 28 belong to the NHLI set, and 6 other instructions are sometimes non high-level (i.e. depending on their arguments).

dcc is part of a decompiling system that also checks for library and compiler signatures in order to eliminate the need to decompile routines that are part of libraries, given that many of these routines are written in assembler and are therefore hard or impossible to translate into C (or any other high-level language). **dccSign** generates unique signatures for library code from different compilers (Borland's Turbo C, Microsoft C, Borland's Turbo Pascal) and stores them in binary files, one for each combination of compiler vendor, memory model, and version of the compiler. The technique makes use of perfect minimal hashing, and thus is very efficient to check whether a given routine belongs to a library or not, if it does, the routine is not analysed any further and it is replaced by its real name from the library. This technique is further explained in [14].

3 Intermediate Code

The initial intermediate representation given by the front-end is a mnemonic-type intermediate code which resembles an assembler language, henceforth called low-level intermediate language (LLIL). The main characteristic of this intermediate representation is that each instruction performs only one function, for example, an assign instruction assigns the value of the right-hand side (rhs) to the left-hand side (lhs). Compound instructions in the machine language are therefore translated into two or more LLIL instructions. For example, `DIV di` divides the combined value of registers `dx`:`ax` by `di`, and places the result in `ax` and the remainder in `dx`. In LLIL code, this instruction makes use of a temporal register, `tmp`, and is translated as follows:

```
tmp = dx:ax
ax = tmp / di
dx = tmp % di
```

For each intermediate instruction, the following bitsets of information are collected during parsing: flags defined, flags used, registers defined, and registers used.

The LLIL code is transformed into a higher level representation which resembles a HLL, henceforth called high-level intermediate language (HLIL). This representation has the following 7 instructions:

- `asgn` arithExp, arithExp
  This instruction assigns an arithmetic expression to another arithmetic expression; normally an identifier. An identifier can be better described as a register, local variable, global variable or a parameter. A subroutine

that returns a value (i.e. a function), is also considered an identifier in this context, as its invocation returns a result in registers. The arithmetic expression represents a tree of binary operations such as addition, subtraction, multiplication, modulus, and, or, and xor. This instruction uses any identifiers in the right-hand side (rhs), and defines the identifier in the left-hand side (lhs).

- `jcond` boolExp
  This conditional jump instruction has a boolean expression associated with it, which determines whether the branch is taken or not. The target branch and the fall-through addresses are not part of the instruction as these have been coded in the graph (i.e. this is the last instruction of a basic block that has 2 out-edges). This instruction uses all identifiers in the boolean expression.

- `jmp` offset

- `call` procId {arithExp} (zero or more actual arguments)
  The `call` instruction represents a subroutine call. The procedure identifier (<procId>) is a pointer to the flow graph of the invoked procedure. The actual parameter list is constructed during data flow analysis. This instruction uses all identifiers present in the actual parameter list, and, in the case of a procedure, does not define any identifiers. If the subroutine called is a function, the function defines the registers that hold the returned value. In this case, the instruction is equivalent to an `asgn` <regs>, <procId> <actual parameters>.

- `ret` [arithExp] (zero or one return expression)
  The return instruction determines the end of a procedure along a path. If there is nothing to return, it means the subroutine is a procedure. Otherwise it is a function, in which case, the returned identifiers are used by this instruction.

- `push` arithExp
  The push instruction places the associated arithmetic expression or identifier on a temporary stack. It uses those identifiers.

- `pop` register
  The pop instruction takes the expression or identifier at the top of the temporary stack and assigns it to the identifier on hand. It defines the identifier with the expression at the top of the stack.

The last two instructions are considered pseudo-high level instructions since they are eliminated from the intermediate code by the end of the data flow analysis. The transformation of LLIL code to HLIL code is done implicitly while performing the data flow analysis. Flow of control HLL constructs, such as looping and conditional instructions, are determined by the control flow analysis phase.

4 Data Flow Analysis

This section describes the transformation of the intermediate code by performing code-improving optimizations. The aim of these optimizations is to eliminate all references to condition codes and registers as they do not exist in high-level languages, and to regenerate the high-level expressions available in the decompiled program; therefore transforming the LLIL code into HLIL code. This section makes references to the control flow graph in Figure 3; a sample program which illustrates all optimizations that are described in this paper.

In order to perform the data flow analysis, intraprocedural information is firstly summarized for each instruction in the form of definition-use (du) chains; the set of live uses associated with each definition of an identifier, and use-definition (ud) chains; the set of reaching definitions associated with each use of an identifier, for all flags and register identifiers. Variables flagged by the front-end as being register variables do not have a du-chain as they represent local variables rather than temporary registers. In Figure 3, both `si` and `di` are flagged as register variables by the idiom analyzer of the front-end.

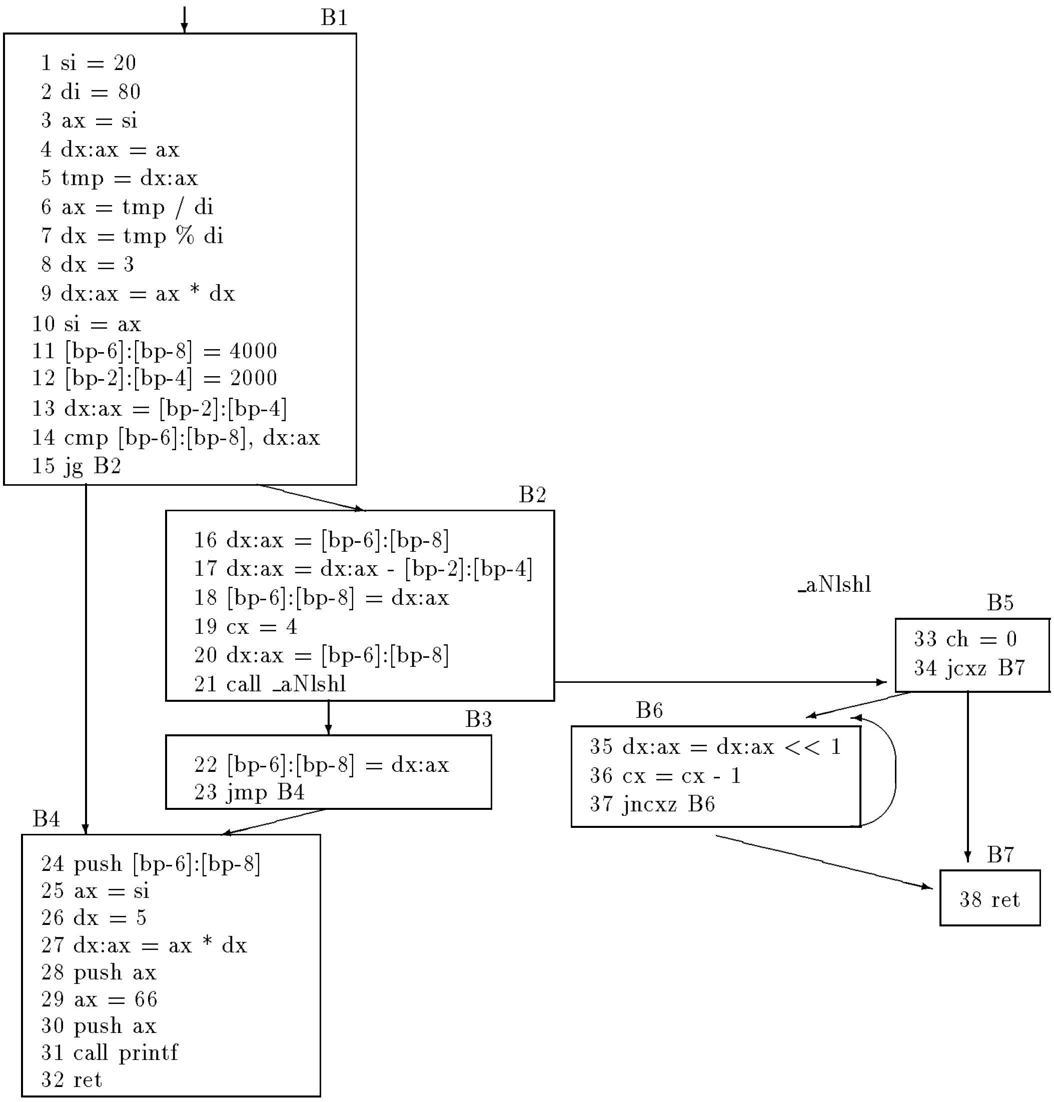

Figure 3: Flow Graph Before Optimization

For all data flow analysis, registers that can be used as both word and byte registers (e.g. `ax`, `ah`, `al`) are treated as different registers in the analysis. For example, whenever register `ax` is defined, it also defines registers `ah` and `al`, but, if register `al` is defined, it defines only registers `al` and `ax`, but not register `ah`. This is needed so that uses of part of a register can be detected and treated as a byte identifier rather than an integer identifier.

While computing du and ud chains, dead register elimination is done in the intermediate code. This analysis is needed even in binary code produced by an optimizing compiler given the nature of the LLIL, which performs one function per instruction, hence compound machine instructions are represented by several LLIL instructions. It is said that a register is dead if it is defined by an instruction and is not used before being redefined by a subsequent instruction. If the instruction that defines a dead register defines only this one register, it is said that the instruction is useless, and thus can be eliminated. On the other hand, if the instruction also defines other register(s), the instruction is still useful but should not define the dead register any more. In this case, the intermediate representation of the instruction is modified to reflect this fact. In the following code from basic block B1, Figure 3, register `dx` is dead at instructions 7 and 9 from inspection of the du chains :

```
6  ax = tmp / di   ; du(ax)={9}
7  dx = tmp % di   ; du(dx)={}
8  dx = 3          ; du(dx)={9}
9  dx:ax = ax * dx ; du(ax)={10} du(dx)={}
10 si = ax
```

Since instruction 7 defines only this register, it is redundant and can be eliminated. On the other hand, instruction 9 defines not only `dx` but `ax` as well, hence the instruction is not dead but is modified to reflect the fact that `dx` is no longer defined by this instruction, simplifying the code to the following:

```
6  ax = tmp / di   ; du(ax)={9}
8  dx = 3          ; du(dx)={9}
9  ax = ax * dx    ; du(ax)={10}
10 si = ax
```

If an instruction i is to be eliminated due to a dead register definition r defined in terms of other registers (i.e. $r = f(r_1, \ldots, r_n), n \geq 1$), the uses of these registers at instruction i no longer exist, and thus, the corresponding du-chains of the instructions that define the registers used at i are to be modified so that they no longer have a reference to i. This is done by checking the ud chain of i while performing dead register elimination.

4.1 Elimination of Condition Codes

It is said that a condition code is dead if it is defined by an instruction and is not used before redefinition. Since the definition of a condition code is a side effect of an instruction, eliminating dead flags does not make an instruction redundant; therefore this analysis leads to removal of data flow information on the instructions. In the following code from basic block B1, Figure 3, the `CF` (carry) and `ZF` (zero) flags are dead at instruction 14 by inspection of their du chains:

```
14 cmp [bp-6]:[bp-8], dx:ax ; def={ZF,CF,SF}
                            ; du(SF)={15}
                            ; du(CF,ZF)={}
15 jg B2                    ; use={SF}
```

The simplified code after removal of this information is the following:

```
14  cmp [bp-6]:[bp-8], dx:ax ; def = {SF}
                             ; du(SF)={15}
15  jg B2                    ; use = {SF}
```

The remaining condition codes are used by subsequent instructions, and are eliminated from the intermediate representation after propagating the essence of the boolean condition: for a particular flag(s) use, we find the instruction that defined the flag(s) and merge them according to the implicit boolean condition of the instruction that uses the flag. In the following code from basic block B1, Figure 3, instruction 14 defines the `SF` flag which is used at instruction 15:

```
14  cmp [bp-6]:[bp-8], dx:ax ; def = {SF}
                             ; du(SF) = {15}
15  jg B2                    ; use = {SF}
                             ; ud(SF) = {14}
```

Instruction 15 implicitly checks for a greater-than boolean condition, and instruction 14 compares the first identifier (`[bp-6]:[bp-8]`) against the second identifier (`dx:ax`). If the first identifier is greater than the second identifier, the `SF` is set. It is obvious from these instructions that the condition propagated is greater than, therefore leading to the following HLIL code:

```
15  jcond ([bp-6]:[bp-8] > dx:ax) B2
```

eliminating instruction 14 and thus eliminating all flag references.

This propagation method works well on extended basic blocks, where the identifiers of the boolean condition are propagated to two or more conditions within the one extended basic block. The algorithm can be extended to propagate condition codes that are defined in two or more basic blocks (i.e. by doing an `and` of the individual boolean conditions), but it has not been required in practice, since it is almost unknown for even optimising compilers to attempt to track flag definitions across basic block boundaries[15].

4.2 Elimination of Registers and Regeneration of Arithmetic Expressions

The regeneration of arithmetic expressions is based on the elimination of registers and the propagation of expressions via registers. Preliminary information on register arguments and function return registers is collected first since the machine language does not provide us with this type of information.

The register calling convention is used by compilers to speed up the invocation of a subroutine. It is an option available in most contemporary compilers, and is also used by the compiler runtime support routines. Given a subroutine, register arguments translate to registers that are used by the subroutine before being defined in the subroutine; i.e. upwards exposed uses of registers in the subroutine. In the following code from basic blocks B5 and B6, Figure 3, subroutine `_aNlshl`, instruction 34 uses register `cx` which was partly defined at instruction 33, and instruction 35 uses registers `dx` and `ax`, neither of which were defined in that subroutine:

```
33  ch = 0
34  jcond (cx = 0) B7  ; ud(ch)={33}
                       ; ud(cl)={}
35  dx:ax = dx:ax << 1 ; ud(dx:ax)={}
```

Information on registers used before definition in a subroutine is summarized by an intraprocedural live register analysis: a register is live on entrance to the basic block that uses it. Standard live register equations are used to solve this problem. In our example, subroutine `_aNlshl` has the following LiveIn and LiveOut sets:

Basic Block	LiveIn	LiveOut
B5	{dx,ax,cl}	{dx,ax}
B6	{dx,ax}	{}
B7	{}	{}

The set of LiveIn registers summarized for the header basic block B5 is the set of register arguments used by the subroutine; `dx`, `ax`, and `cl`. The formal argument list of this subroutine is updated to reflect these two arguments:

```
formal_arguments(_aNlshl) = (arg1 = dx:ax,
                             arg2 = cl)
```

It is said that the `_aNlshl` subroutine *uses* these registers. In general, any subroutine that makes use of register arguments uses those registers, thus a `CALL` to one of these subroutines is also said to use those registers, as in the following instruction:

```
21 call _aNlshl     ; use={dx,ax,cl}
```

Functions return results in registers, and there is no machine instruction that specifies which registers are being returned by the function in CISC machines. After function return, the caller uses the registers returned by the function before they are redefined (i.e. these registers are live on entrance to the basic block that follows the function call). This register information is propagated across subroutine boundaries, and is solved with a reaching and live register analysis. In the following code from basic blocks B2 and B3, Figure 3, instruction 22 uses registers `dx` and `ax`, which could have been redefined in the subroutine called at instruction 21 or at instruction 20:

```
20  dx:ax = [bp-6]:[bp-8] ; def={dx,ax}
                          ; use = {}
21  call _aNlshl          ; def={}
```

```
                          ; use={dx,ax,cl}
22  [bp-6]:[bp-8] = dx:ax ; def={}
                          ; use={dx,ax}
```

Summary information in the form of intraprocedural reaching definitions on subroutine `_aNlshl` leads to the following ReachIn and ReachOut sets:

Basic Block	ReachIn	ReachOut
B5	{}	{ch}
B6	{ch}	{cx,dx,ax}
B7	{cx,dx,ax}	{cx,dx,ax}

This analysis states that the last definitions of registers `cx`, `dx`, and `ax` reach the end of the subroutine (i.e. ReachOut set of basic block B7). The caller subroutine uses only some of these reaching registers, thus it is necessary to determine which registers are upwards exposed in the successor basic block to the subroutine invocation; this information is summarized in the form of an interprocedural live register analysis, to cater for registers propagated across subroutine boundaries. Traditional live register equations are used for the call graph of the complete program, or the set of more precise live equations recently described by Srivastava and Wall in [16]. For the example of Figure 3, either set of equations produces the following results:

Basic Block	LiveIn	LiveOut
B1	{}	{}
B2	{}	{dx,ax}
B3	{dx,ax}	{}
B4	{}	{}
B5	{dx,ax,cl}	{dx,ax}
B6	{dx,ax}	{dx,ax}
B7	{dx,ax}	{dx,ax}

From the three registers that reach instruction 22 in basic block B3, only two of these registers are used (i.e. belong to LiveIn of B3): `dx` and `ax`, thus these registers are the only registers of interest once the called subroutine has been finished, and are the registers returned by the function. This condition is formally expressed by the intersection of the ReachOut set of the function and the LiveIn set of the basic block following the `CALL`, to eliminate propagated registers across subroutines:

$$\text{ReachOut(B7)} \bigcap \text{LiveIn(B3)} = \{\text{dx,ax}\}$$

Once a subroutine has been determined to be a function and the register(s) that the function returns has been determined, this information is propagated to two different places: the return instruction(s) from the function and the instructions that `CALL` this function. In the former case, all return basic blocks have a `ret` instruction; this instruction is modified to return the registers that the function returns. In our example, instruction 38 of basic block B7, Figure 3 is modified to the following code:

```
38  ret dx:ax
```

In the latter case, any function invocation instruction (i.e. `CALL` instruction) is replaced by an `asgn` instruction that takes as left-hand side the defined register(s), and takes the function call as the right-hand side of the instruction, as in the following code:

```
21  dx:ax = call _aNlshl ; def={dx,ax}
                         ; use={dx,ax,cl}
```

The instruction is transformed into an `asgn` instruction, and defines the registers on the left-hand side.

It is important to note that in the case of library functions whose return register(s) is not used, the call is not transformed into an `asgn` instruction but remains as a `CALL` instruction (e.g. `printf`).

4.2.1 Extended Register Copy Propagation

Register copy propagation is the method by which a defined register in an assignment instruction, say `ax = cx`, is replaced in a subsequent instruction(s) that references or uses this register, if neither register is modified after the assignment (i.e. neither `ax` nor `cx` is redefined). If this is the case, references to register `ax` are replaced by references to register `cx`, and, if all uses of `ax` are replaced by `cx` then `ax` becomes dead and the assignment instruction is eliminated. A use of `ax` can be replaced with a use of `cx` if the instruction `ax = cx` is the only definition of `ax` that reaches the use of `ax` and if no assignments to `cx` have occurred after the instruction `ax = cx`. The former condition is checked with ud chains, the latter condition is checked with an x-clear condition as described later. For example, in the following code from basic block B1, Figure 3, after dead-register elimination:

```
3  ax = si          ; du(ax)={4}
4  dx:ax = ax       ; du(dx:ax)={5}
                    ; ud(ax)={3}
5  tmp = dx:ax      ; du(tmp)={6}
                    ; ud(dx:ax)={4}
6  ax = tmp / di    ; du(ax)={9}
                    ; ud(tmp)={5}
8  dx = 3           ; du(dx)={8}
9  ax = ax * dx     ; du(ax)={10}
                    ; ud(ax)={6} ud(dx)={8}
10 si = ax          ; ud(ax)={9}
```

the use of register `ax` in instruction 4 is replaced with a use of the register variable `si`, making the definition of `ax` in 3 dead. The use of `dx:ax` in instruction 5 is replaced with a use of `si` (from instruction 4), making the definition of `dx:ax` dead. The use of `tmp` in instruction 6 is replaced with a use of `si` (from instruction 5), making the definition of `tmp` dead at 5. The use of `ax` at instruction 9 is replaced with a use of `(si / di)` from instruction 6, making the definition of `ax` dead. In the same instruction, the use of `dx` is replaced with a use of constant 3 from instruction 8, making the definition of `dx` at 8 dead. Finally, the use of `ax` at instruction 10 is replaced with a use of `(si / di) * 3` from instruction 9, making the definition of `ax` at 9 dead. Since the register defined in instructions $3 \rightarrow 9$ were used only once, and all these registers became dead, the instructions are eliminated, leading to the final code:

```
10 si = (si / di) * 3
```

Register copy propagation is not limited to `asgn` instructions only. As seen in Figure 4, two other HLIL instructions also define registers: `CALL` defines a register if the invoked subroutine is a function, and `POP` defines the register associated with that instruction. Also, several instructions use registers: `CALL` uses any register arguments passed to it, `jcond` uses any registers associated with its boolean conditional expression, `ret` uses any registers it returns from a function, and `PUSH` uses all registers it pushes onto the stack. Since `PUSH` and `POP` rely on an extra data structure, the stack, a stack of expressions is used to cater for values pushed and popped from the stack. Note that the saving and restoring of registers by a subroutine at pre and postamble have been removed from the intermediate representation by the front-end (these registers are flagged as being register variables within that subroutine), hence they are not considered true uses or definitions of registers. The front-end has also removed all `POP` instructions that restore the stack after a subroutine call or during subroutine return, hence these definitions of registers are not part of the HLIL code. Therefore, both `PUSH` and `POP` are used in conjunction with the spilling of a register, and are eliminated from the HLIL code in the following way: a `PUSH` copies the arithmetic expression associated with the register to the stack, and a `POP` translates to an `asgn` of the top of stack expression to the associated register with the `POP`. In this way, pseudo-HLIL instructions are removed from the final representation.

Most actual parameters to a subroutine are pushed on the stack before invocation to the subroutine. Since nested subroutine calls are allowed in most languages, the arguments pushed on the stack represent those arguments of one or more subroutines, thus it is necessary to determine which arguments belong to which subroutine. Whenever a `CALL` instruction is met, the necessary number of arguments are popped from the stack, based on the fixed size of argument bytes restored by the subroutine (and summarized by the front-end). In the following code from basic block B4, Figure 3, instructions 24, 28 and 30 `PUSH` the arguments for the `printf` call at instruction 31:

Define	Use
asgn (lhs) CALL (function) POP	asgn (rhs) CALL (register arguments) jcond ret (function return registers) PUSH

Figure 4: High-Level Instructions that Define and Use Registers

```
24  push [bp-6]:[bp-8]
28  push (si * 5)
30  push 66
31  call printf
```

When the call to `printf` is reached, information on this function is checked to determine how many bytes of arguments the function call takes; in this case it takes 8 bytes. Expressions are popped from the stack, adding up the size of their type, and are placed on the actual parameter list associated with the subroutine call using the calling convention determined by the front-end. In this example, 3 expressions are popped from the stack with type sizes of 2, 2, and 4, and are stored using the C calling convention, leading to the following code:

```
31  call printf (66, si * 5, [bp-6]:[bp-8])
```

In the case of register arguments, since these arguments are not pushed on the stack but remain in registers, when performing register copy propagation and reaching a `CALL` instruction that uses one or more registers, the expression associated with this register(s) is placed on the actual parameter list of the invoked subroutine. For example, in the following code from basic blocks B2 and B3, Figure 3, the `CALL` at instruction 21 uses registers `dx`, `ax`, and `cl`:

```
19  cl = 4                ; du(cl)={21}
20  dx:ax = [bp-6]:[bp-8] ; du(dx:ax)={21}
21  dx:ax = call _aNlshl  ; ud(dx:ax)={20}
                          ; ud(cl)={19}
```

The expressions associated with these registers are moved to the actual parameter list of `_aNlshl` in the order defined by the formal argument list, leading to the following code:

```
21  dx:ax = call _aNlshl ([bp-6]:[bp-8], 4)
```

Instructions 19 and 20 are eliminated since they now define dead registers.

During the instantiation of actual arguments to formal arguments, data types for these arguments need to be verified, as if they are different, one of the data types needs to be modified. Consider the following partial code from basic block B4, Figure 3:

```
31  call printf (66, si * 5, [bp-6]:[bp-8])
```

where the actual parameter list has the following data types: integer constant, integer, and long integer. The formal argument list of `printf` has a pointer to a character string as the first argument, and a variable number of unknown data type arguments following it[1]. We can only verify the type of the first argument in this case, leading to a mismatch. Given that the data types used by the library subroutines must be right (i.e. they are trusted), it is safe to say that the actual integer constant must be an offset into memory, pointing to a character string. By checking virtual memory, it is found that at location `DS:0066` there is a string; thus, the integer constant is replaced by the string itself. For the next two arguments, since they have an unknown formal type, the type given by the caller is trusted, leading to the following code:

[1] Formal argument information is summarized by the library signature generator[14] in the front-end and is available for library calls only.

```
call printf ("c * 5 = %d, a = %ld\n",
                si * 5, [bp-6]:[bp-8])
```

Another case of type propagation is the conversion of two integers into one long variable, where the callee has determined that one of the arguments is a long variable, but the caller has so far used the actual argument as two separate integers.

The transformations presented here modify the initial graph in Figure 3 into the equivalent graph of Figure 5. In this graph, all identifiers are in terms of their local offset from the stack or a register variable. These identifiers are renamed during code generation and are assigned arbitrary names according to their location: local or argument.

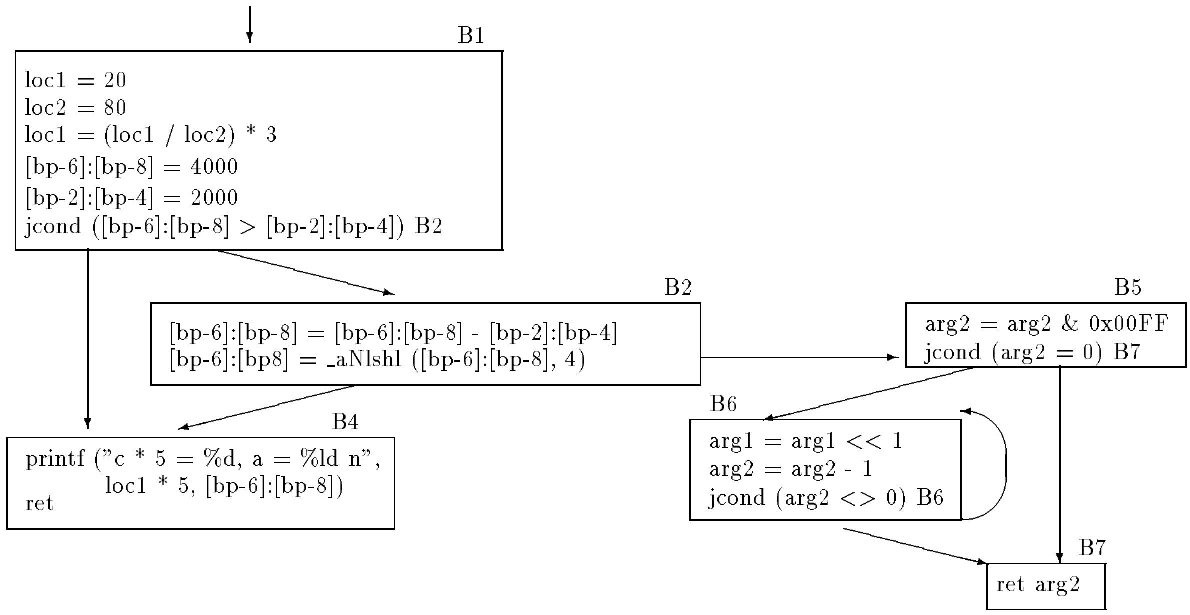

Figure 5: Control Flow Graph After Code Optimization

Having given all different types of examples where register copy and type propagation is possible, we now present the set of necessary conditions for performing such propagation of registers and associated expressions. From the definition, the instruction that uses the register to be propagated must be able to uniquely identify the instruction that defined that register, hence the *uniqueness condition*:

- For a given register use, the corresponding register definition must be unique, as registers that are used before being redefined translate to temporary registers that hold an intermediate result for the machine.

Redefinition of the involved registers cannot happen between the instructions. Even more important, the identifiers associated with the expression in the rhs of the defined register cannot be redefined along that path, hence the *rhs-clear path* condition:

- The identifiers x in an expression that defines a register r (i.e. the rhs of the instruction) that satisfies the uniqueness condition are checked to have an x-clear path to the instruction that uses the register r. The rhs-clear condition for an instruction j that uses a register r which is uniquely defined at instruction i is formally defined as:

$$\text{rhs-clear}_{i \rightarrow j} = \bigcap_{x \in \text{rhs}(i)} x\text{-clear}_{i \rightarrow j}$$

$$
\begin{array}{lll}
\text{where} & \text{rhs}(i) & \text{is the rhs of instruction i} \\
\text{and} & x & \text{is an identifier that belongs to} \\
 & & \text{the rhs}(i) \\
\text{and} & x\text{-clear}_{i \rightarrow j} & = \begin{cases} \text{True} & \text{if } x \text{ is not redefined} \\ & \text{along the path } i \rightarrow j \\ \text{False} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 6 is the algorithm used for extended register copy propagation. In this algorithm, the `propagate(r,exp1,exp2)` function propagates the use of register `r` in `exp2` with `exp1`, and `newRegArg(r,l)` places register `r` in the actual argument list `l`.

5 Experimental Results

dcc is a prototype decompiler written in C for the DOS operating system and the i80286 architecture that runs under DOS and Ultrix. dcc produces both C and assembler programs, its implementation is fully described in [2] and summarized in [17]. Compiler and library signatures were generated for several compilers, and dcc makes use of them if identified. At present, dcc determines base types (e.g. integers, longs) but is not able to determine compound types such as arrays or structures.

This section reports on results obtained from a test suite of .exe programs originally written in C and compiled under DOS. These programs make use of base type variables and illustrate different aspects of the decompilation process. The test suite was run in batch mode, generating the disassembly file `.a2`, the C file `.b`, the call graph of the program, and statistics on the reduction of intermediate code instructions. The statistics reflect the percentage of intermediate instruction reduction on all subroutines for which C is generated; subroutines which translate to assembler due to their low-level machine nature are not considered in the statistics. For each program, a total count on intermediate instructions before and after analysis, and a total percentage reduction is given.

Figure 7 presents summary results of the 10 programs of the test suite. The first three programs deal with operations on the different three base types (byte, integer, long). The initial C programs had the same code, but their variables were defined of a different type. The next four programs are benchmark programs from the Plum-Hall benchmark suite; the suite is freely available on the network [18]. These programs were modified to ask for the arguments to the program with `scanf()` rather than scanning for them in the `argv[]` command line array since arrays are not supported by dcc. Finally, the last three programs calculate Fibonacci numbers, compute the cyclic redundancy check for a character, and multiply two matrixes. The latter program was used to illustrate how array address computation is represented in dcc in terms of an expression, rather than being further analyzed and type propagated as an array.

The total number of intermediate instructions before the analysis is 963, compared with the final 306 intermediate instructions, which gives a reduction of instructions of 76.25%. This reduction of instructions is mainly due to the optimizations performed during data flow analysis, particularly the elimination of registers across subroutines. The recognition of idioms in the LLIL code also reduces the number of instructions and helps in the determination of data types such as long integers. Decompiled programs have the same number of user subroutines, plus any runtime support routines used by the program, and any library functions not recognized by the library signature. Runtime routines are sometimes translatable into a high-level representation; assembler is generated whenever they are untranslatable.

6 Conclusions

The methods presented in this paper demonstrate how traditional data flow analysis techniques can be used in a decompiler to optimize the intermediate representation of the decompiled program, and transform it into a higher

```
procedure ExtRegCopyProp (p: subroutineRecord)

 initExpStk().
 for (all basic blocks b of subroutine p in postorder) do
   for (all instructions j in b) do
     for (all registers r used by instruction j) do
       if (ud(r) = {i}) then  /* uniquely defined at instruction i */
          case (opcode(i))
            asgn: if (rhsClear (i, j))
                     case (opcode(j))
                       asgn:  propagate (r, rhs(i), rhs(j)).
                       jcond, push, ret:  propagate (r, rhs(i), exp(j)).
                       call:  newRegArg (r, actArgList(j)).
                     end case
                  end if
            pop:  exp = popExpStk().
                  case (opcode(j))
                    asgn:  propagate (r, exp, rhs(j)).
                    jcond, push, ret: propagate (r, exp, exp(j)).
                    call:  newRegArg (exp, actArgList(j)).
                  end case
            call: case (opcode(j))
                    asgn:  rhs(j) = i.
                    push, ret, jcond:  exp(j) = i.
                    call:  newRegArg (i, actArgList(j)).
                  end case
          end case
       end if
     end for

     if (opcode(i) == push) then
        pushExpStk (exp(i)).
     elsif (opcode(i) == call) and (invoked routine uses stack arguments) then
        pop arguments from the stack.
        place arguments on actual argument list.
        propagate argument type.
     end if
   end for
 end for
end procedure
```

Figure 6: Extended Register Copy Propagation Algorithm

level representation that is available in any imperative procedural language. Specifically, these methods aim at the elimination of low-level concepts such as condition codes and registers from the intermediate representation, and to reintroduce the high-level concept of expression into the intermediate representation. This last feature comprises not only simple boolean and arithmetic expressions, but expressions that involve function calls and parameter passing. Short circuit evaluated expressions are determined during the control flow analysis phase of the decompiler.

In the field of condition code elimination, the method presented in this paper goes beyond the optimization of flag definitions, in that it not only determines which flag definitions are extraneous and therefore unnecessary,

Program	Before	After	% Reduction
intops	45	10	77.78
byteops	58	10	82.76
longops	117	48	58.97
benchsho	101	25	75.25
benchlng	139	28	79.86
benchmul	88	12	86.36
benchfn	82	36	56.10
fibo	78	15	80.77
crc	171	38	77.78
matrixmu	84	11	86.90
total	963	306	76.25

Figure 7: Results for Test Suite Programs

but also determines which boolean conditional expression is represented by the combined set of instructions that define and use the flag. In this way, the target HLL program does not rely on the use and concept of flags, as any real HLL program does not.

In the field of elimination of registers, this paper presents an extended register copy propagation algorithm which works well for the HLIL used and the propagation of expressions between instructions. This algorithm also eliminates the pseudo-intermediate instructions `PUSH` and `POP`. Two necessary conditions are presented for this method to be applied. Extensions to the method make it feasible to determine actual parameters, and expressions or identifiers returned from functions.

We are currently working on a RISC decompiler for the SPARC architecture and testing the extensibility of both data and control flow analyses. In particular, the higher optimization performed in RISC code will be tested against the reconstruction of high-level code. In regards to the intermediate language, as expected, a more general LLIL language is required, although the choice of HLIL language remains the same. Also, better high-level data abstraction (e.g. arrays) is needed – this is not a simple task. For further information refer to http://crg.cs.utas.edu.au/

Acknowledgements

I would like to thank Professor John Gough for several discussions of live register analysis and compiler data flow analysis, and Vishv Malhotra for suggestions on how to improve this paper.

A Graph-Theoretic Terminology

A *basic block* is a sequence of instructions that has a single entry point and a single exit point. These requirements give the basic block the property that, if one instruction is executed, then all other instructions are executed as well.

An *extended basic block* is a sequence of basic blocks $B_1, \ldots, B_n$ such that for $1 \leq i < n$, B_i is the only predecessor of B_{i+1}, and for $1 < i \leq n$, B_i has only a conditional jump instruction.

A *control flow graph* G is a tuple (N, E, h), where N is the set of nodes, E is the set of directed edges, and h is the root of the graph. A node $n \in N$ represents a basic block. A *path* from n_1 to n_m, represented $n_1 \rightarrow n_m$, is a sequence of edges $(n_1, n_2), (n_2, n_3), \ldots, (n_{m-1}, n_m)$.

Let $\mathcal{P} = \{p_1, p_2, \ldots\}$ be the finite set of procedures of a program. A *call graph* C is a tuple (N, E, h), where N is the set of procedures and $n_i \in N$ represents one and only one $p_i \in \mathcal{P}$, E is the set of edges and $(n_i, n_j) \in E$ represents one or more references of p_i to p_j, and h is the main procedure.

A variable or register is *defined* when a new value is assigned to the variable/register. A variable or register is *used* when the value of the variable/register is used but not modified.

A du-chain for variable x at statement i is the set of statements $j > i$ where x could be used, given that x is defined at statement i. There are known algorithms[19, 3, 20] to solve this backward-flow, any-path data flow problem (du-chains).

A ud-chain for variable x at statement j is the set of statements $i < j$ where x was defined, given that x was used at statement j. There are known algorithms[19, 3, 20] to solve this forward-flow, any-path data flow problem.

A path is *X-clear* if there is no definition of variable X along that path.

B Example

This section illustrates an example of the decompilation of a simple C program. The sample program (see Figure 10) was chosen not for its content but because it illustrates several of the concepts that were introduced in this paper. Figure 8 illustrates the LLIL code for the `main()` procedure of this program. All calls to library routines were detected by **dccSign**, and thus not included in the analysis. Figure 9 is the final output from dcc. This C program can be compared with the original C program in Figure 10. The decompiled program is functionally equivalent to the original C program, although it uses a different looping construct: a `while()` rather than a `for()` loop. The rest of the code is the same, with the use of different variable names. In this program, 52 LLIL instructions were converted into 10 HLIL instructions by means of interprocedural data flow analysis (only type propagation between procedures was done in this example); this is equivalent to an 80.76% reduction on the number of intermediate instructions.

References

[1] C. Cifuentes. A structuring algorithm for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Informática*, pages 267–276, Buenos Aires, Argentina, 2-6 August 1993. Centro Latinoamericano de Estudios en Informática.

[2] C. Cifuentes. *Reverse Compilation Techniques.* PhD dissertation, Queensland University of Technology, School of Computing Science, July 1994.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 10, pages 585–722. Addison-Wesley Publishing Company, 1986.

[4] B.C. Housel. *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages.* PhD dissertation, Purdue University, Computer Science, August 1973.

[5] F.L. Friedman. *Decompilation and the Transfer of Mini-Computer Operating Systems.* PhD dissertation, Purdue University, Computer Science, August 1974.

[6] D.A. Workman. Language design using decompilation. Technical report, University of Central Florida, December 1978.

```
main  PROC  FAR
000 0004C2 55                  PUSH           bp
001 0004C3 8BEC                MOV            bp, sp
002 0004C5 83EC02              SUB            sp, 2
003 0004C8 56                  PUSH           si
004 0004C9 57                  PUSH           di
005 0004CA 1E                  PUSH           ds
006 0004CB B89400              MOV            ax, 94h
007 0004CE 50                  PUSH           ax
008 0004CF 9A0B004901          CALL   far ptr printf
009 0004D4 59                  POP            cx
010 0004D5 59                  POP            cx
011 0004D6 16                  PUSH           ss
012 0004D7 8D46FE              LEA            ax, [bp-2]
013 0004DA 50                  PUSH           ax
014 0004DB 1E                  PUSH           ds
015 0004DC B89900              MOV            ax, 99h
016 0004DF 50                  PUSH           ax
017 0004E0 9A0400FD01          CALL   far ptr scanf
018 0004E5 83C408              ADD            sp, 8
019 0004E8 BE0400              MOV            si, 4
020 0004EB BF0100              MOV            di, 1
022 000510 83FF28         L1:  CMP            di, 28h
023 000513 7EDB                JLE            L2
024 000515 FF76FE              PUSH  word ptr [bp-2]
025 000518 1E                  PUSH           ds
026 000519 B89C00              MOV            ax, 9Ch
027 00051C 50                  PUSH           ax
028 00051D 9A0B004901          CALL   far ptr printf
029 000522 83C406              ADD            sp, 6
030 000525 5F                  POP            di
031 000526 5E                  POP            si
032 000527 8BE5                MOV            sp, bp
033 000529 5D                  POP            bp
034 00052A CB                  RETF
035 0004F0 8BC7           L2:  MOV            ax, di
036 0004F2 BA0700              MOV            dx, 7
037 0004F5 F7E2                MUL            dx
038 0004F7 8BF0                MOV            si, ax
039 0004F9 0376FE              ADD            si, [bp-2]
040 0004FC 8BC6                MOV            ax, si
041 0004FE B104                MOV            cl, 4
042 000500 D3F8                SAR            ax, cl
043 000502 8BF0                MOV            si, ax
044 000504 8BC6                MOV            ax, si
045 000506 BB0A00              MOV            bx, 0Ah
046 000509 99                  CWD
047                            MOV            tmp, dx:ax         ;Synthetic inst
048 00050A F7FB                IDIV           bx
049                            MOD            bx                 ;Synthetic inst
050 00050C 8956FE              MOV            [bp-2], dx
051 00050F 47                  INC            di
052                            JMP            L1                 ;Synthetic inst
main  ENDP
```

Figure 8: Low-level Intermediate Code

```
/*
 * Input file : testfile.exe
 * File type : EXE
 */


void main ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
int loc1;
int loc2;
int loc3;

    printf ("a = ");
    scanf ("%d", &loc1);
    loc2 = 4;
    loc3 = 1;
    while ((loc3 <= 40)) {
        loc2 = ((loc3 * 7) + loc1);
        loc2 = (loc2 >> 4);
        loc1 = (loc2 % 10);
        loc3 = (loc3 + 1);
    } /* end of while */
    printf ("a = %d\n", loc1);
}
```

Figure 9: Final C Program

```
main()
{ int a, b, c;

   printf ("a = ");
   scanf ("%d", &a);
   b = 4;
   for (c = 1; c <= 40; ++c)
   {
      b = a + c * 7;
      b = b >> 4;
      a = b % 10;
   }
   printf("a = %d\n", a);
}
```

Figure 10: Original C Program

[7] D.L. Brinkley. Intercomputer transportation of assembly language software through decompilation. Technical report, Naval Underwater Systems Center, October 1981.

[8] J. Reuter. URL: ftp//cs.washington.edu/pub/decomp.tar.z. Public domain software, 1988.

[9] J. Bowen and P. Breuer. Decompilation techniques. Internal to ESPRIT REDO project no. 2487 2487-TN-PRG-1065 Version 1.2, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, March 1991.

[10] J. Bowen. From programs to object code and back again using logic programming: Compilation and decompilation. *Journal of Software Maintenance: Research and Practice*, 5(4):205–234, 1993.

[11] P.T. Breuer and J.P. Bowen. Decompilation: the enumeration of types and grammars. *Transaction of Programming Languages and Systems*, 16(5):1613–1647, September 1994.

[12] D.M. Dejean and G.W. Zobrist. A definition optimization technique used in a code translation algorithm. *Communications of the ACM*, 32(1):94–104, January 1989.

[13] G.L. Hopwood. *Decompilation*. PhD dissertation, University of California, Irvine, Computer Science, 1978.

[14] M. Van Emmerik. Signatures for library functions in executable files. Technical Report 2/94, Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia, April 1994.

[15] K.J. Gough. Private communication, 1993.

[16] A. Srivastava and D.W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[17] C. Cifuentes and K.J. Gough. Decompilation of binary programs. *Software – Practice and Experience*, 25(7):811–829, July 1995.

[18] E.S. Raymond. Plum-hall benchmarks. URL: ftp//plaza.aarnet.edu.au/usenet/comp.sources.unix/volume20/plum-benchmarks.gz, 1989.

[19] F.E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[20] C.N. Fischer and R.J. LeBlanc Jr. *Crafting a Compiler*, chapter 16, pages 609–680. Benjamin Cummings, 2727 Sand Hill Road, Menlo Park, California 94025, 1988.