Reverse Engineering of Real-Time Assembly Code

Jens Palsberg **Matthew Wallace**
Purdue University
Dept. of Computer Science
West Lafayette, IN 47907
{palsberg,wallacms}@cs.purdue.edu

Abstract

Much legacy real-time code is written in assembly language. Such code is often crafted to meet stringent time and space requirements so the high-level intent of the programmer may have been obscured. The result is code that is difficult to maintain and reuse. In this paper we present a tool for reverse engineering of real-time Z86 assembly code, together with a tool for validation of the output. Our experimental results are for a suite of commercial microcontrollers. For those benchmarks, our tool does the bulk of the reverse-engineering work, leaving just a few undisciplined uses of machine code to be handled manually. Our tool is designed to preserve programmer intent to the largest extent possible. Thus, the reverse engineered program is easier to understand and maintain than the original.

1 Introduction

1.1 Background

In most embedded systems, resource constraints play an important role. This is usually due to economic considerations that may dictate the use of a cheap and impoverished processor. Constraints on time, space, power, etc., can make it challenging to implement the desired functionality. For example, it may be required that the handling of a certain event terminates within one millisecond. If the event handler is programmed in C, then the programmer is at the mercy of the C compiler—will it generate code that is fast enough? If the event handler is programmed in assembly language, then it is easier to control the timing aspects, but programming, maintenance, and reuse become harder. Todays real-time systems are sometimes programmed using a hybrid approach: program in C, compile, and then hack the output from the compiler.

Much legacy real-time code is written in assembly language. For the purposes of maintenance and reuse, it is desirable to create representations at a higher level of abstraction. In this paper we focus on the following question.

Question: Can we automate the reverse engineering of real-time assembly code?

We are particularly interested in preserving programmer intent to the largest extent possible.

Our starting point is a suite of seven proprietary microcontroller systems. The code for these systems were kindly provided to us by Greenhill Manufacturing, Ltd. [1]. Greenhill has over a decade of experience producing environmental control systems for agricultural needs. The microcontrollers were carefully handwritten in Z86 assembly language.

Our goal is to automate the reverse engineering of the seven microcontrollers into ZIL (Z86 Intermediate Language). ZIL was designed by Naik and Palsberg [7] for use as an intermediate representation in compilers; they implemented a code-size-directed ZIL compiler. As part of their work, Naik, Palsberg, and their colleagues reverse engineered two of the microcontrollers by hand. It was a long and painful process and they decided that without some level of automation, no further reverse engineering would be done.

1.2 Our results

We have implemented a tool for reverse engineering of real-time Z86 assembly code, together with a tool for validation of the output. For six of the seven microcontrollers, our tool does the bulk of the reverse-engineering work, leaving just a few undisciplined uses of machine code to be handled manually. For those six benchmarks, we have successfully completed the reverse engineering task by hand with a minimal effort. The seventh benchmark remains problematic for us. Our tool is good at preserving programmer intent and, in our judgment, the reverse engineered programs seem easier to understand and maintain than the originals.

The main problems that are solved by our reverse engineering tool are: (1) creation of procedures and interrupt handlers with single entry and exit points, (2) declaration of variables that can be used in the place of registers, and (3) introduction of formal parameters to procedures. The basis for our validation tool is a Z86 assembly language interpreter [4] and a ZIL interpreter. The main problems that are solved by our validation tool are: (i) execution in a common environment of both Z86 assembly code and ZIL programs, and (ii) managing the wide range of possible compilations of a ZIL program.

The ideas underlying our tools are applicable beyond Z86 and ZIL. Our results suggest that reverse engineering of real-time assembly code can be automated to a large degree while preserving programmer intent. It remains to be seen whether it is feasible to continue the reverse engineering process to the level of, say, C. At the level of ZIL, our validation tool gives some assurance that all possible compilations of the ZIL program have the desired timing properties. Such an assurance is more difficult to get when targeting a higher-level language.

Related work on analyzing assembly code includes the safety checking of Xu, Miller, and Reps [10], and the typed assembly language of Morrisett, Walker, Crary, and Glew [6, 5]. These papers do not attempt to do reverse engineering.

Rest of the paper. In Section 2 we survey the Z86 architecture and discuss the ZIL language. In Section 3 we illustrate our tools and techniques with three examples. In Section 4 we present our reverse engineering tool, and in Section 5 we present our validation tool. In Section 6 we show our experimental results, and finally in Section 7 we conclude with some directions for future work.

2 Z86 and ZIL

On the Z86E30 architecture [2], the register file is partitioned into banks, and the Register Pointer (RP) designates a bank as the "current" or "working" bank. We can think of the register file as a matrix of registers, with RP pointing to the current row. For that reason, we sometimes refer to RP as the "row pointer."

A register address is either global or RP-relative (for a register in the current bank); a global address takes more space than an RP-relative address. With good data layout and RP manipulation, potentially, many instructions can be executed faster than without using RP. A significant fraction of the instructions used in real-world embedded programs qualifies for such an optimization.

The Z86E30 has 256 8-bit registers organized into 16 banks of 16 registers each. Of these, 236 are general-purpose, while the rest, namely, the first 4 registers in the 0^{th} bank and the 16 registers in the 15^{th} bank, are special-purpose. All special-purpose registers in the 0^{th} bank and 12 of the special-purpose registers in the 15^{th} bank are visible to the ZIL programmer. The RP is itself a special-purpose register in the 15^{th} bank that is invisible at the ZIL level.

The Z86E30 lacks a data/stack memory; all variables must be stored in registers. Distributing global variables among the various banks in a manner that reduces the execution time of the whole program is a major challenge [7].

A Z86 assembly instruction can address a register using an 8-bit or a 4-bit address. In the former case, the high nibble of the 8-bit address represents the bank number and the low nibble represents the register number within that bank. In the latter case, the high nibble of RP represents the bank number and the 4-bit address represents the register number within that bank. We shall refer to registers addressed using 4 and 8 bits as working and non-working registers respectively.

The execution time of certain Z86 assembly instructions depends upon whether they address registers using 4 or 8 bits. For instance, the binary add instruction **add** v_1, v_2 depends upon the value of RP: the execution time is 6 or 10 machine cycles depending on whether RP points to the bank in which v_1 and v_2 are stored or not. Nearly 30–40% of the instructions in our benchmark programs execute faster if they address registers using 4 bits.

ZIL [7] was designed to be a convenient intermediate language for a compiler that generates Z86 assembly code. A grammar for ZIL together with a few explanations are presented in Appendix A. As part of the work presented in this paper, we extended ZIL with read-only arrays and parameters to procedures. The new version of ZIL contains the old version as a subset, so we will refer to the new version simply as ZIL.

There are three main differences between ZIL and Z86 assembly language. First, registers are not used in ZIL. All data are stored in variables, of which there can be unboundedly many. Second, ZIL has a simple type system with the types int, string, proc_label, jump_label, and int[]. Intuitively, each label has one of the two label types, integer arrays have type int[], strings have type string, and all other values are of type int. Third, in ZIL, statements are organized into procedures and interrupt handlers with single entry and exit points. Thus, jumping from one procedure to the middle of another is not allowed. A procedure may have formal parameters; arguments can be passed either by value or by reference.

3 Examples

We now present three example programs, in both Z86 assembly and ZIL. Each example illustrates one aspect of our reverse engineering tool.

3.1 Procedure splitting

The example in Figure 1 illustrates the need for splitting procedures. Based strictly on the callees in the Z86 program, there are only two procedures. However, procedure P1 uses some of P2's code in its execution when it jumps to label P2A. We say that this label is contained inside of procedure P2 because it comes after the label P2 and before the next label that is the argument of a CALL instruction (or, in this case, the end of the program). We want each statement to be contained in just one procedure. Since the jump statement inside procedure P1 jumps to a label that is not contained inside the procedure, we call this a non-local jump. Such jumps are not allowed in ZIL. Instead we turn this jump into a call, and make the label into its own procedure. The process by which this is done is explained later in the paper; the result is shown in Figure 1.

3.2 RP analysis

The example in Figure 2 shows how working registers are changed into variables. In this program, the row pointer can be exactly determined at every point in the program. This means that RP-relative references can easily be transformed into global references by multiplying the row pointer by 0x10 and adding the register that is specified. Since RP is 0x10, then r1 becomes register 0x11. Since the programmer did not define a symbolic name for this register, we just make up an arbitrary variable name, R11.

3.3 Adding parameters

In the example in Figure 3 the Z86 program first sets the row pointer to the first bank. Then, values are loaded into registers 1 and 2 of this bank. Since the row pointer is 0x10, this means that registers 0x11 and 0x12 are used. In the procedures these registers are added together, storing the result in the first register. Control is returned to MAIN, and a few statements later the row pointer is set to 0x20. This means that when P1 is called in the next statement, it will be accessing different registers, 0x21 and 0x22, than in the first call. It is important to capture this in the ZIL program. In the ZIL version, the SRP statements are removed, because there is no register file into which to index. In order to access the correct variables, we introduce formal parameters into P1. In the first call to P1, the local variables var1 and var2 are passed as parameters. They are passed by reference to P1, as designated by the "&" in front of each parameter. This means that var1 will contain the sum of var1 and var2 after the call, which is the desired effect. In the next call the global variables glob1 and glob2, which have presumably been assigned to in some other part of the program, are passed as parameters. It is important to note that these variables have a direct mapping to the original Z86 program. Anywhere that register 0x21 is used in the original program, the variable glob1 will be used in the ZIL program.

```
// Z86 program:

MAIN:
            .
            .
            CALL P1
            CALL P2
            .
            JP MAIN
P1:
            SUB r1, #2h
            JP Z, P2A
            LD r2, #20h
            RET
P2:
            ADD r1, #1h
P2A:        LD r2, #10h
            .
            .
            RET
```

```
// ZIL program:

int var1
int var2

MAIN {
START:
            .
            .
            CALL P1
            CALL P2
            .
            JP START
}

PROCEDURES
P1() {
            SUB var1, 2h
            JP Z, CALLP2
            LD var2, 20h
            JP END
CALLP2:     CALL P2A
END:        RET
}

P2() {
            ADD var1, 1h
            CALL P2A
            RET
}

P2A() {
            LD r2, 10h
            .
            .
            RET
}
```

Figure 1: Procedure splitting

```
// Z86 program:

MAIN:
        SRP #10h
        LD IMR, #1h
        LD r1, #0h
        EI
LOOP:   INC r1
        .
        .
        CP r1, #0h
        JP NEQ, LOOP

IRQVC0:
        SRP #10h
        LD r1, #0h
        IRET
```

```
// ZIL Program:

int R11
IRQ 0 IRQVC0
MAIN {
        SRP 10h
        LDIMR, 1h
        LD R11, 0h
        EI
LOOP:   INC R11
        .
        .
        CP R11, 0h
        JP NEQ, LOOP
}

HANDLERS:
IRQVC0 {
        SRP 10h
        LD R11, 0h
        IRET
}
```

Figure 2: From registers to variables

```
// Z86 Program:

MAIN:
            SRP #10
            LD r1, #42h
            LD r2, #23h
            CALL P1
            .
            SRP #20
            CALL P1
            .
            JP MAIN
P1:
            ADD r1, r2
            RET
```

```
// ZIL Program:

int glob1
int glob2
MAIN {
int var1
int var2
START:
            LD var1, 42h
            LD var2, 23h
            CALL P1(var1, var2)
            .
            CALL P1(glob1, glob2)
            .
            JP START
}
PROCEDURES

P1(int &a, int &b) {
            ADD a, b
            RET
}
```

Figure 3: Adding parameters

4 Our Reverse Engineering Tool

There are three main tasks in creating an equivalent ZIL program from a Z86 program: procedure splitting, RP analysis, and adding parameters.

First is to split the code up into procedures with unique entry and exit points. The first step in this process is to go through all of the instructions in the program and make a list of all of the labels that are the argument of a CALL instruction. Each label that is CALLed constitutes the beginning of a procedure. The end of that procedure is temporarily defined as the instruction before the next label that is the argument of a CALL instruction. With temporary procedure boundaries in place, we can check for jumps outside of the procedure and procedures that overlap each other. Each label that is jumped to outside of its containing procedure is then split off to form a new procedure. A CALL to the new procedure is added to the original procedure; the location of the call is where the new procedure used to start. In the procedure where the offending jump occurred, a new label is added to end of the procedure. At this label, a CALL to the new procedure is added, followed by a jump to the end of the current procedure. If a procedure "runs-over" into another procedure, then that procedure is simply truncated at the overlap point and a CALL is inserted to the following procedure.

The second task is to convert register references to variable references. The important thing in this step is to make sure that a register is not split up into two abstract variables when it is really intended to be "live" over several procedure calls. This is made more complicated by the fact that registers can be specified using one of three addressing modes: global, RP-relative, and indirect. Our tool successfully translates all global references, most RP-relative references, and does not currently attempt to translate indirect references. Global references are the easiest to deal with because they always refer to the same register. Translation involves creating a variable for each register that is accessed via global addressing. To make the new program as close to the original as possible, if the programmer has given a symbolic name to a particular register (through .EQU directive), then that name is used as the variable for that register.

Local addressing poses more of a problem because the register that is referenced is dependent on the value of the row pointer (RP) register. In order to determine which register is being referenced, and therefore which variable should be used, it is necessary to determine the possible values of RP at each instruction in the program. In order to collect this information, we do a flow analysis which is polyvariant in the RP value. We use a style of polyvariance which closely matches those studied by Agesen [3] and Schmidt [9], see also [8]. Each procedure is characterized such that it either has the same RP value when it exits as when it enters or it has another definite RP value when it exits. Once this information has been collected, each instruction that uses RP-relative addressing is examined. If a unique RP value has been determined at this instruction, the register is computed and the corresponding variable is substituted.

If there are multiple possible RP values at an instruction, there is still a possibility that this instruction can be translated. If a procedure is called from two program points that each have different RP values, this will create a situation where the statements inside the procedure have two possible RP values. By introducing parameter passing into ZIL, we can take out the RP-relative references, thereby eliminating the problem of a polymorphic RP.

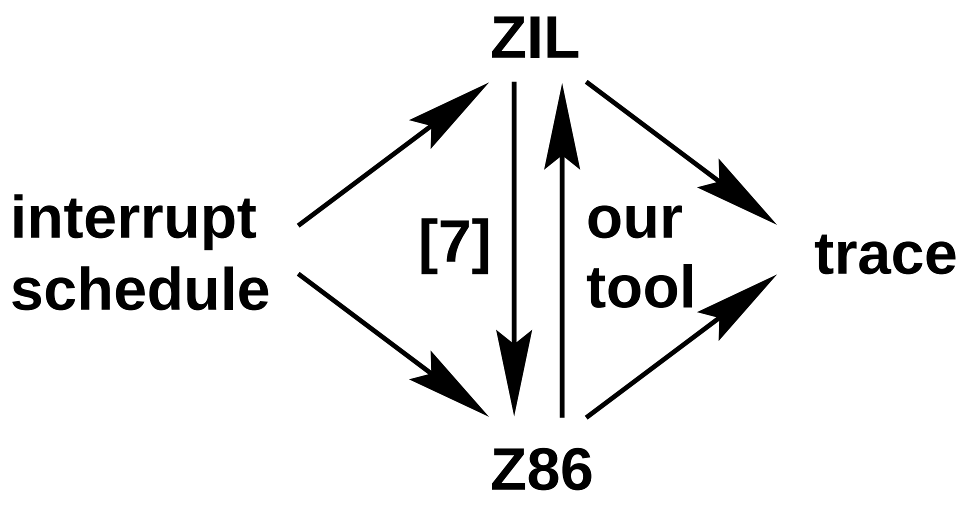

Figure 4: Our validation tool

To do this translation, we create one formal parameter for each unique RP-relative reference. Then, at the call site, the RP (which is known to be a unique value) is used to compute which register would be referenced by the RP-relative references inside the procedure. The variables corresponding to these registers are then inserted as parameters to the call. This is repeated for each call site of the procedure in question. This mechanism does not translate all RP-relative references, however. If the RP polymorphism was introduced by an if-else type construct, where the RP is set to one value for a true branch and to another for a false branch, the reference cannot be translated. These are left for the programmer to fix by hand.

In addition to the three major tasks there is other minor work that the tool does. String constants are created from the .ASCII directives in the Z86 program. These take the form of LABEL: .ASCII ”String”. LABEL is taken as the name of the string constant in ZIL and ”String” is the value. Types are partially handled by the tool, but some work is left for the programmer. All variables are assumed to be of type int, which is a valid assumption for all but a few variables in the programs we have. Variables of type string and the label types must be declared by hand. The rest of the work done by the tool is syntactic changes, such as changing LD IMR, to LDIMR and removing SRP statements.

5 Our Validation Tool

The functionality of our validation tool is illustrated in Figure 4. The arrow from ZIL to Z86 denotes the code-size-directed compiler presented by Naik and Palsberg [7]. We now present our tool in more detail.

Validation of a reverse-engineered program amounts to ensuring that the reverse-engineered program and the original have the same behavior. In general, this is a difficult problem. It would not be feasible to track all of the register changes for both programs. However, if we can limit the number of registers we track, the problem is made significantly easier. We

```
0,3; 2,48; 2,56; 2,48; 2,56; 0,1; 3,98
Enter IRQ 0
1,4; 0,2
Enter IRQ 1
0,0; 0,2
Exit IRQ 1
0,1
Exit IRQ 0
2,48
```

Figure 5: Trace before reordering

```
0,3; 2,48; 2,56; 2,48; 2,56; 0,1; 3,98;
2,48; 1,4; 0,2; 0,1; 0,0; 0,2
Enter IRQ 0
Enter IRQ 1
Exit IRQ 1
Exit IRQ 0
```

Figure 6: Trace after reordering

observe that the function of a microcontroller is to give input to and receive output from devices that are attached to it. For the Z86 microprocessor, there are 4 registers (named P0-P4, also called port registers) that are reserved for this I/O. Since a microcontroller's functionality is wholly described by the input and output on these ports, it is reasonable to say that two programs are the same if they produce the same values in the port registers.

We have created two simulators, one for Z86 assembly and one for ZIL. These simulators execute each instruction of the given program and keep a store of the registers that would be present in an actual microprocessor. Each time a new value is written to one of the port registers, the number of the register and the value that was written is outputted to produce an execution trace. For two programs, one in ZIL and one in Z86, if their execution traces are identical, then we get some assurance that they have the same behavior.

Requiring exact matching of execution traces is actually a bit strict. It is possible that two programs will produce the same output, but that output might be ordered slightly differently. This is because these programs are heavily driven by interrupts, which may occur at any point in the program. We simulate interrupts that would normally be triggered by devices attached to the microcontroller by specifying fixed times at which interrupts occur. This specification is given in the form of 4 integers. These integers represent a modulo time for each interrupt, for example:

```
543   // Fire Interrupt 0 every 543 cycles
 32   // Fire Interrupt 1 every 32 cycles
 -1   // Don't fire Interrupt 2
 42   // Fire Interrupt 3 every 42 cycles
```

Any time the current number of cycles that have been executed is a multiple of the number specified, that interrupt is triggered. Whether the interrupt is actually handled still depends on whether the interrupt's bit in the interrupt mask register (IMR) is enabled. Only 4 interrupts must be specified because IRQ 4 and 5 are triggered by the processor's real-time clock. This clock is handled by our simulators.

Even if the intervals for Z86 and ZIL programs are set to be the same, interrupts can occur at different program points, because there are slight timing differences between Z86 and ZIL. These differences arise because in the Z86 processor, certain instructions have variable execution time depending on whether global or RP-relative register addressing is used. Since there are no registers in ZIL, this variability cannot be exactly modeled. We have achieved fairly good timing parity by simply choosing *randomly* between the slow and fast versions of each instruction when this variability would be present in the Z86.

In order to account for difference in where interrupts occur, along with the values for the port register, each execution trace contains statements to show when interrupts are entered and exited. An example of this trace is given in Figure 5. These statements are then used to reorder the execution trace such that all port values that occur while executing the main program are listed first, followed by values that occurred while executing each of the interrupt handlers. Finally the order in which interrupts occurred is listed. An of example of the reordered trace is shown in Figure 6. If, after this reordering, two traces are the same, then programs which generated them are also the same. This reordering works because the exact timing of the communications on the port registers does not matter, only that the values occur in the same order. Since the programmer cannot know in advance where in the program an interrupt will occur, the exact placement of values that occur in interrupt handlers cannot be considered to describe the program. Thus we can consider values written to port registers inside an interrupt handler to be local to that handler and analyze them separately.

In practice, this technique of reordering does not usually have to be used. Because the timing in between Z86 and ZIL remains relatively close, interrupts do not generally cause traces that should be the same to be different. So, most programs can be validated simply by directly comparing the output of the two simulators using diff or some similar tool. We have implemented the reordering technique as a separate tool that can be optionally used if differences between the two programs are believed to be caused by interrupt timing differences.

6 Experimental Results

6.1 Benchmark characteristics

For our experiments, we have used the seven microcontrollers provided by Greenhill Manufacturing, Ltd. We have also used a small example program (Example) that was used as a running example by Naik and Palsberg [7]. Some characteristics of our benchmark programs are presented in Figure 7. Figure 7 also shows similar numbers for the reverse-engineered ZIL programs.

Program	Number of Z86 instructions	Number of ZIL instructions	Number of Z86 callees	Number of ZIL procedures
CTurk	642	648	30	34
Gturk	800	836	35	49
ZTurk	750	778	35	47
DRop	531	514	25	27
Rop	516	500	26	27
Fan	1124	1268	36	39
Serial	195	187	9	10
Example	53	52	6	9

Figure 7: Benchmark characteristics

Program	After procedure splitting	After RP analysis	After adding parameters
CTurk	146	15	10
GTurk	178	25	20
ZTurk	163	18	14
DRop	83	2	2
Rop	81	2	2
Fan	329	10	6
Serial	24	0	0
Example	18	0	0

Figure 8: Number of 4-bit addresses to be resolved

Program	Number of parameters added	Number of call sites that pass parameters	Number of procedures with parameters
CTurk	6	3	1
GTurk	4	2	1
ZTurk	4	2	1
DRop	0	0	0
Rop	0	0	0
Fan	1	5	1
Serial	0	0	0
Example	0	0	0

Figure 9: Adding parameters to procedures

6.2 Measurements

Figure 8 shows the number of 4-bit addresses to be resolved after each of the three stages our reverse engineering tool.

Figure 9 shows the number of parameters added, the number of call sites that pass parameters, and the total number of procedures with parameters.

The benchmarking was run on an Athlon XP 1700 with 512 MB of DDR RAM using Debian GNU/Linux, kernel version 2.4.17. For all programs the time to run the reverse engineering tool was well under 1 second. At least 75% of this time was spent outputting the code.

6.3 Assessment

After running the original code through the reverse-engineering tool, there was still some work to be done to get the code running correctly. The biggest task was changing indirect addresses into parameters or some other appropriate abstraction. The *Turk programs each required about 5 of these types of changes, the Fan program required about 20 and the rest did not need this work done. Also the way that strings were handled had to be changed. In the Z86 programs a string was manipulated by loading the address in ROM where the string was stored into a register, then using indirect addressing to read a character at a time. ZIL supports a similar concept, but instead of ROM addresses, ZIL uses variables of type string. Also in Z86 there is a load-and-increment instruction, which is often used for strings. In ZIL this must be changed to a load, then an increment of the string variable. This work could possibly be automated, but we decided against it because it is possible for a programmer to use this same type of construct in Z86 to load generic data, rather than a string. We wanted the translation to always be correct, so that the programmer would only have to fix what the translator could not handle, not go back and fix its mistakes. Overall the work left to be done on each program was minimal. The Serial and *Rop programs had no more than 3 lines that needed to be changed, and the rest of the programs did not require more than about 30 changes total.

The biggest gain in effort saved definitely came from the RP analysis, as suggested by Figure 8. This step reduced the work in most of the programs to a manageable level, and completely eliminated RP-relative addressing for some. The addition of formal parameters produced a less dramatic reduction, but was still worthwhile. While the number of RP-relative references that were eliminated was quite small, it is important that we were able to capture the programmer's intent to pass parameters using RP-relative addressing. Furthermore, parameter passing was quite useful in translating the remaining parts of the programs in cases where indirect addressing was used.

To validate the output of the reverse-engineered programs, we ran them through our validation tool. The Z86 simulator runs at 4-5 times what the speed of the actual Z86 processor would be. Most programs needed about 10 minutes of Z86 processor time to go through all of their functions.

For all of the programs, except Fan and Example, the traces were identical *without* re-ordering. For Example, the traces were identical *after* reordering. Further work is needed to complete the reverse engineering of Fan.

7 Conclusion and Future Work

Reverse engineering of real-time assembly code can be automated to a large degree while preserving programmer intent.

A key idea for further improvement is to implement liveness analysis. There are several instances in our test suite where the programmer has pushed a register onto the stack using RP-relative addressing so that register can be used locally in the procedure and then popped off before returning. This is done because RP-relative addressing is faster than global addressing and it can not be known by the programmer whether that register is being used at the time or not. This idea is accommodated perfectly by local variables in procedures in ZIL. Currently the tool makes all variables global and ignores these pushes and pops. By analyzing this type of construct, the analysis could be made more precise and better preserve the programmer's intent.

Also, our tool cannot analyze interrupt handlers without some programmer intervention. Since interrupts can occur at any point in the program, if RP-relative addressing is used inside an interrupt handler, it is not possible for our tool to know what the possible values of the RP are without further analysis. In addition to knowing the RP at every point in the program, it is necessary to know the IMR. In this way, we can track only the RP values that occur while a particular handler is enabled. In practice, the RP almost always seems to be limited to a single value while a handler is enabled.

Finally, indirect addressing poses a problem. In order to know which register a statement that uses indirect addressing is accessing, the tool must know what the value of the register containing the indirect address is. This leads to a more general flow analysis problem, similar to the RP analysis. Combined with the liveness analysis, we plan to implement a "local" flow analysis for each occurrence of indirect addressing. In most cases the register being addressed is assigned to in the 2 or 3 statements preceding, so we expect this will not slow down the analysis significantly.

Acknowledgments. We thank James Rose for rewriting the Z86 version of cturk into ZIL. We thank Dennis Brylow for help in numerous situations. Palsberg is supported by a National Science Foundation Information Technology Research Award number 0112628.

References

[1] Greenhill Manufacturing. http://www.greenhillmfg.com.

[2] Zilog, Inc. http://www.zilog.com.

[3] Ole Agesen. The Cartesian product algorithm. In *Proceedings of ECOOP'95, Seventh European Conference on Object-Oriented Programming*, pages 2–26. Springer-Verlag (*LNCS* 952), 1995.

[4] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proceedings of ICSE'01, 23rd International Conference on Software Engineering*, pages 47–56, Toronto, May 2001.

[5] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. Presented at 1999 ACM Workshop on Compiler Support for System Software, May 1999.

[6] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 85–97, 1998.

[7] Mayur Naik and Jens Palsberg. Compiling with code-size constraints. In *LCTES'02, Languages, Compilers, and Tools for Embedded Systems joint with SCOPES'02, Software and Compilers for Embedded Systems*, Berlin, Germany, June 2002. To appear.

[8] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, May 2001. Preliminary version in Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 197–208, San Diego, California, January 1998.

[9] David Schmidt. Natural-semantics-based abstract interpretation. In *Proceedings of SAS'95, International Static Analysis Symposium.* Springer-Verlag (*LNCS* 983), Glasgow, Scotland, September 1995.

[10] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 70–82, 2000.

Appendix A: ZIL

The grammar for ZIL is shown in Figure 10. It is an extended version of the grammar given in [7], as explained below.

In ZIL, variables can be declared globally, locally to the main program, or locally to any procedure or interrupt handler. There are 16 predefined global variables in ZIL: P0, P1, P2, P3, FLAGS, T0, T1, P01M, P2M, P3M, TMR, PRE0, PRE1, IMR, IPR, IRQ

These correspond to special purpose registers defined in Z86. P0-P4 are located in Bank 0 of the register file and the rest are located in Bank 15.

Each ZIL routine (procedure or interrupt handler) has a single entry point and a single exit point. Each procedure has a single **ret** instruction and each interrupt handler has a single **iret** instruction. A jump from one routine to an instruction within another routine is not allowed.

New features in ZIL are as follows. Procedures may define formal parameters which may be passed by value or by reference. Formal parameters are considered to be local to the procedure, but may be passed to other procedures. ZIL also introduces array datatypes. These are statically defined arrays that can hold integer values. Arrays must be initialized with data when defined and immutable. The syntax is similar to Java. Arrays may be referenced into using integer constants or variables.

The ZIL instructions are, essentially, Z86E30 instructions, except that they operate on variables instead of registers.

```
              Goal ::= ( GlobalDef )* "MAIN" MainBlock "PROCEDURES" ( ProcDef )*
                       "HANDLERS" ( HandlerDef )*
         GlobalDef ::= ConstDef | VarDef
          ConstDef ::= "static" "final" Type Id "=" Literal
            VarDef ::= Type Variable
              Type ::= "int" | "string" | "proc_label" | "jump_label" | "int[]"
           ProcDef ::= Label "(" (ParamList)? ")" ProcedureBlock
         ParamList ::= ParamDef ("," ParamDef)*
          ParamDef ::= Type ("&")? Id
        HandlerDef ::= Label "(" ")" HandlerBlock
         MainBlock ::= "{" ( VarDef )* ( Stmt )* "}"
    ProcedureBlock ::= "{" ( VarDef )* ( Stmt )* ( Label ":" )? "RET" "}"
      HandlerBlock ::= "{" ( VarDef )* ( Stmt )* ( Label ":" )? "IRET" "}"
              Stmt ::= ( Label ":" )? Instruction
       Instruction ::= ArithLogic1aryOPC Variable
                    |  ArithLogic2aryOPC Variable "," Expr
                    |  CPUControl1aryOPC Expr
                    |  CPUControl0aryOPC
                    |  "LD" Variable "," LDExpr
                    |  "DJNZ" Variable "," Label
                    |  "JP" ( Condition "," )? LabelExpr
                    |  "CALL" LabelExpr (ArgList)?
                    |  "preserveIMR" "{" ( Stmt )* ( Label ":" )? "}"
            LDExpr ::= "@" Id | Expr | "LABEL" Label
              Expr ::= "!" Expr | "(" Expr "&" Expr ")" | "(" Expr "|" Expr ")"
                    |  Prim
              Prim ::= Id | IntLiteral | ArrayReference
    ArrayReference ::= Id "[" IntLiteral "]" | Id "[" Id "]"
         LabelExpr ::= Label | "@" Variable
           ArgList ::= "(" Id ("," Id)* ")"
          Variable ::= Id
             Label ::= Id
 ArithLogic1aryOPC ::= "CLR" | "COM" | "DA"  | "DEC" | "INC" | "POP" | "PUSH"
                    |  "RL"  | "RLC" | "RR"  | "RRC" | "SRA" | "SWAP"
 ArithLogic2aryOPC ::= "ADC" | "ADD" | "AND" | "CP"  | "OR"  | "SBC"
                    |  "SUB" | "TCM" | "TM"  | "XOR"
 CPUControl0aryOPC ::= "CLRIMR" | "CLRIRQ" | "EI"    | "DI"    | "HALT"  | "NOP"
                    |  "RCF"    | "SCF"    | "STOP"  | "WDH"   | "WDT"
 CPUControl1aryOPC ::= "ANDIMR" | "ANDIRQ" | "LDIMR" | "LDIPR" | "LDIRQ"
                    |  "ORIMR"  | "ORIRQ"  | "TMIMR" | "TMIRQ"
         Condition ::= "F"   | "C"   | "NC"  | "Z"  | "NZ" | "PL" | "MI" | "OV"
                    |  "NOV" | "EQ"  | "NE"  | "GE" | "GT" | "LE" | "LT" | "UGE"
                    |  "ULE" | "ULT" | "UGT" | "GLE"
           Literal ::= IntLiteral | StrLiteral | ArrayLiteral
        IntLiteral ::= <HEX_H> | <BIN_B> | <DEC_D>
        StrLiteral ::= <STRING_CONSTANT>
      ArrayLiteral ::= "{" IntLiteral ("," IntLiteral)* "}"
                Id ::= <IDENTIFIER>
```

Figure 10: The ZIL grammar