

# The Chirp Intermediate Language and Code Generation

Li Xu

Department of Computer Science  
University of Massachusetts Lowell  
Lowell, MA 01854

October 2008

## 1 Introduction

Chirp is a simple C-like imperative language targeting embedded microcontrollers used in personal robot devices. The language itself is described in its language specification [1]. This document describes the Chirp Intermediate Language (ChirpIL) and compiler code generation from ChirpIL to target robot hardware.

The design of the Chirp compiler follows the typical compiler structure: the front-end lexer, parser and semantics checker will parse the input program, create an abstract syntax tree (AST) tree representation, and check semantics of program function and statements for type and semantics errors; the middle-end will convert program AST tree into the low-level ChirpIL which is close to the command set of the robot controller, the middle-end will also consist of multiple optimizing passes over ChirpIL to improve code quality and reduce memory size; finally, the back-end translates ChirpIL into instructions in microcontroller command set and generate the executable file to run on the robot device. The document will describe ChirpIL: the intermediate instructions, the program control flow graph formed from basic blocks of ChirpIL instructions, and the code generation process to generate target microcontroller code from ChirpIL. Although ChirpIL enables many opportunities for both code and data optimization, however, its optimization is not in the scope of this document and will be addressed separately in other Chirp compiler design documents.

## 2 Chirp Intermediate Language

The Chirp compiler will convert the AST from the front-end to the 3-address style intermediate representation specified by the following ChirpIL grammar.

### 2.1 ChirpIL Instructions

As listed in Figure 1, program in ChirpIL form consists of sequence of basic blocks. Block with labels are target of ChirpIL branch instructions. Basic blocks are sequence of ChirpIL instructions with control flow entering through the first instruction and exiting at the last instruction, there is no control transfer into and out of the basic blocks in the middle. The basic block can include the following ChirpIL instructions:

- Array load:  $dst = \text{aload } abase \text{ } adr$   
Load array element data from the memory address of  $adr$ . Array itself is represented by  $abase$  operand. The load result is in  $dst$  operand.

<i>il_prog</i>	→	( <i>il_func</i> )*
<i>il_func</i>	→	( <i>il_block</i> )*
<i>il_block</i>	→	(label : )? ( <i>il_inst</i> )*
<i>label</i>	→	STRING
<i>il_inst</i>	→	<i>aload</i>   <i>astore</i>   <i>binop</i>   <i>branch</i>   <i>call</i>   <i>jump</i>   <i>mov</i>   <i>retval</i>   <i>return</i>
<i>aload</i>	→	<i>rand</i> = <b>aload</b> <i>rand</i> <i>rand</i>
<i>astore</i>	→	<b>astore</b> <i>rand</i> <i>rand</i> <i>rand</i>
<i>binop</i>	→	<i>rand</i> = <i>rand</i> <i>op</i> <i>rand</i>
<i>op</i>	→	add   sub   mul   div   mod
<i>branch</i>	→	(bz   bnz ) <i>rand</i> <i>label</i>
	→	(beq   bne   blt   ble   bgt   BGE ) <i>rand</i> <i>rand</i> <i>label</i>
<i>call</i>	→	( <i>rand</i> = )? <b>call</b> FUNC_NAME ( <i>rand</i> )*
<i>jump</i>	→	<b>jmp</b> <i>label</i>
<i>mov</i>	→	<i>rand</i> = <i>rand</i>
<i>retval</i>	→	<b>retval</b> <i>rand</i>
<i>return</i>	→	<b>return</b>
<i>rand</i>	→	INT   STRING   VAR_NAME

Figure 1: ChirpIL grammar

- Array store: **astore** *abase* *adr* *src*  
Store the *src* value into array element at address of *adr*. Array itself is represented by **abase** operand.
- Binary instructions: *dst* = *src1* *op* *src2*  
Perform binary *op* operation on *src1* and *src2* and store result in *dst*. The operations include add, sub, mul, div and mod.
- Branch instructions: (bz/bnz ) *src* *label* or beq/bne/blt/ble/bgt/bge *src1* *src2* *label*  
The branch instructions check whether the src operand is 0 or non-0 (bz, bnz), or compare two operands (beq, bne, blt, ble, bgt, bge) and branch to the target label basic block in the condition is true.
- Call instruction: *retval* = **call** FUNC\_NAME arg-list  
Call the specified function with given argument operands and store return value (if any) into *retval* variable operand.
- Jump instruction: **jmp** *label*  
Jump to the target block.
- Move instruction: *dst* = *src*  
Copy *src* value to *dst* variable operand.
- Return value instruction: **retval** *val*  
**retval** sets *val* as return value of function.
- Function return: **return**  
**return** ends function call and transfers control to caller. The difference of *retval* and **return** is the former has semantics to set return value of the function and the latter does the transfer of control of function return. The code generation section will describe their use to implement function calling convention.

```

int a=1001;
int b;
int c;
int i;
int A[10];

int main()
{
  a = 3;
  c = 35;
  b = 5+3*a-b/2%100;

  for i (0:9) {
    A[i] = i+1;
  }

  return A[0];
}

```

Figure 2: Example Chirp program

The ChirpIL instructions operate on three kinds of operands:

- INT for integer values.
- STRING for string constants.
- VAR\_NAME for named variables. These include the variables from input program and temporary variables generated by the compiler. They should have symbol table entries and the code generator will use their type information to allocate storage and generate target code.

## 2.2 Control Flow Graph

Chirp functions are represented as control flow graph (CFG) in ChirpIL form. The CFG for each function consists of basic blocks and each block has CFG edges indicating successor blocks. Furthermore, each function CFG has a unique entry block node and exit block node to represent function entry and return points. The exit block contains a single `return` instruction to return control flow. The `retval` instruction should be followed by a `jmp` instruction to the exit block.

## 2.3 IRGen: Translation from AST to ChirpIL

The AST representation created by parser is converted into ChirpIL CFG by the IRGen intermediate representation generator. IRGen will translate Chirp statement and expression subtrees with sequence of ChirpIL instructions and generate basic blocks and CFG representation.

To translate expressions, IRGen will create temporary variables to store intermediate results. Arithmetic operations can be translated directly into ChirpIL binary instructions, while logic and comparison expressions are translated through equivalent blocks and branches. Chirp assignments are translated into move and array load and store instructions; control flow statements are translated into basic blocks, branches and jumps.

The following example shows Chirp source program and its ChirpIL IR.

```

#Function: main
main:  a = 3
      c = 35
      t0 = 3 mul a
      t1 = 5 add t0
      t2 = b div 2
      t3 = t2 mod 100
      t4 = t1 sub t3
      b = t4
      i = 0
BB0:  bgt i 9 BB2
BB1:  t5 = i add 1
      t6 = i
      t6 = t6 mul 2
      t6 = A add t6
      astore A (t6) t5
      i = i add 1
      jmp BB0
BB2:  t7 = 0
      t7 = t7 mul 2
      t7 = A add t7
      t8 = aload A (t7)
      retval t8
      jmp BB3
BB3:  return

```

Figure 3: ChirpIL IR for the example

### 3 Code Generation

The back-end code generator will translate ChirpIL to the target microcontroller's command set. In general, the ChirpIL instructions are close to instructions available on the target devices, as most provide similar instructions on arithmetic and control flow operations. The code generator should manage the following:

- Data storage for ChirpIL operands. The variable operands (both from the source or generated by compiler) will be allocated into register and memory storage. Arrays should be allocated into memory and array names are mapped to the base address.
- Translate ChirpIL instructions to the target microcontroller commands.
- Translate System intrinsic function call into hardware I/O control command sequence.
- Generate appropriate start-up code to initialize program state and run the main function.

We use Parallax Scribbler robot as concrete target. The code generator in general needs to do the following: 1) map variable names to STAMP controller RAM variables, and arrays as ROM variables; 2) as there is no stack on STAMP microcontroller, the code generator should do name mangling for formal and local variables and allocate in the global storage; 3) translate ChirpIL to PBASIC; 4) implement System.Scribbler functions as PBASIC I/O commands (see [3] for details); 5) generate start-up code to initialize variables and jump to the main function entry block.

The generated PBASIC code for the example in Section 2 is shown in Figure 4.

### 4 Acknowledgments

This work has been supported by NSF grant DUE-0737054.

### References

- [1] The Chirp Language Specification, Version 2. [http://www.cs.uml.edu/~xu/cs406/chirp\\_spec.v2.html](http://www.cs.uml.edu/~xu/cs406/chirp_spec.v2.html).
- [2] Parallax, Inc. *BASIC Stamp Reference Manual, version 2.1*.
- [3] Parallax, Inc. Hack's Hints for the Scribbler Robot.

```

'global
a VAR word
b VAR word
c VAR word
i VAR word
A DATA word 0(10)
main_ret VAR word
'func main:local
main_l_t0 VAR word
main_l_t1 VAR word
main_l_t2 VAR word
main_l_t3 VAR word
main_l_t4 VAR word
main_l_t5 VAR word
main_l_t6 VAR word
main_l_t7 VAR word
main_l_t8 VAR word

start_main:
    a = 1001
    GOSUB main
end

'func: main
main:
    a = 3
    c = 35
    main_l_t0 = 3 * a
    main_l_t1 = 5 + main_l_t0
    main_l_t2 = b / 2
    main_l_t3 = main_l_t2 // 100
    main_l_t4 = main_l_t1 - main_l_t3
    b = main_l_t4
    i = 0
BB0:
    if (i > 9) then goto BB2
BB1:
    main_l_t5 = i + 1
    main_l_t6 = i
    main_l_t6 = main_l_t6 * 2
    main_l_t6 = A + main_l_t6
    WRITE main_l_t6, word main_l_t5
    i = i + 1
    goto BB0
BB2:
    main_l_t7 = 0
    main_l_t7 = main_l_t7 * 2
    main_l_t7 = A + main_l_t7
    READ main_l_t7, word main_l_t8
    main_ret = main_l_t8
    goto BB3
BB3:
    RETURN

```

Figure 4: PBASIC code generated from ChirpIL IR