

A Look at the MIRV Compiler System

EECS 483

Supplementary Information for Project 2

October 9, 2006

Overview

- MIRV compiler
 - Frontend
 - Convert C to MIRV IR (Project 2)
 - Backend
 - Convert MIRV IR to assembly code (Project 3)
- Simplescalar simulator
 - Sim-fast
 - functional simulator
 - Sim-outorder
 - behavioral simulator

MIRV Overview

- High Level Intermediate Representation
- Textual Format
- A “sanitized C”
 - Type declarations, constant declarations, etc.
- Prefix Form AST
$$c = a+b \rightarrow = c + a b$$
- Developed by Dave Green and Matt Postiff

Fundamental Concepts

- Symbol
 - Abstract representation of a program object
 - Constant (Data value)
 - Variable (Storage location)
 - Function (Program text)
 - Type (Object characteristic – a “meta object”)
- Unid: UNique ID
 - Every symbol has one
 - Present a flat namespace
 - `[a-zA-Z][\ .a-zA-Z_0-9]*`

General Program Structure

< Package Header >

< Module Definition >

< Declarations > (before use)

< Type decl >

< Constant decl >

< Variable decl >

< Function decl >

<Definitions > (of things declared)

< Function def >

< Declarations > (flat namespace, no declarations in loops, if-else, etc)

< Statements >

Packages and Modules

```
Package version 0 0 1 unid
  __mirv_pack
  version 0 0 0 {
  name "__mirv_pack"
}
```

```
Module unid __mirv_pack.ml {
  <mirv program text>
}
```

Symbols

- All Symbols Are Declared
 - tsdecl: “type simple” decl (int, float, pointer, array)
 - tcdecl: “type complex” decl (struct, union)
 - tfdecl: “type function” decl (function signature)
 - vdecl: variable decl
 - cdecl: constant decl
 - fdecl: function decl
- Must declare a symbol before using it!

Type Declaration

- Integer Type
 - tsdecl unid <name> integer <sign> <bit_size>
 - <sign>: (signed | unsigned)
- Pointer Type
 - tsdecl unid <name> pointer unid <name>
- Array Type
 - tsdecl unid <name> array unid <name> { <dim_lst> }
- Function Type
 - tfdecl unid <name> { <interface_spec> }

Simple Type Example

- C

```
int *ptr[10];
```

- MIRV

- Type names can be anything, but need to be unique and consistent

```
tsdecl unid sint32_t integer signed 32
```

```
tsdecl unid sint32_t_p_t pointer unid sint32_t
```

```
tsdecl unid sint32_t_p_t_10_t
```

```
array unid sint32_t_p_t { 10 }
```

Constant String Type

- C (type declaration for “%d\n”)

```
printf(“%d\n”, x);
```

- MIRV

```
tsdecl unid sint8_t_4_t array unid sint8_t { 4 }
```

– ‘\n’ is a single char

– Leave room for the ‘\0’!

Function Type Example

- C

```
int foo(int);
```

- MIRV

```
tfdecl unid sint32_t_sint32_t_t {  
    retValType sint32_t  
    argType sint32_t  
}
```

Function Type Example

- C

```
int bar(void);
```

- MIRV

```
tfdecl unid sint32_t_sint0_t_t {  
    retValType sint32_t  
}
```

NOTE: No argType! (inconsistency)

Vararg Function Type

- C

```
extern int printf(char *a, ...);
```

- MIRV

```
tfdecl unid
  sint32_t_sint8_t_p_t_va_t {
  retValType sint32_t
  argType sint8_t_p_t
  varArgList
}
```

Symbol Linkage

- Specify Symbol Visibility
 - Global -> export (visible to anyone)
 - Extern -> import (visible from anyone)
 - Static -> internal (visible only within file)
 - Local -> internal (visible only within function)
- Affect Variables and Functions
 - Functions and global variables must retain their original C names for the linker to work properly!

Declarations Example

- C

```
int global = 10;  
int main(void);  
extern int printf(char *, ...);
```

- MIRV

```
<type decls...>  
vdecl export unid global unid sing32_t {}  
cdecl unid c10_c unid sint32_t 10  
fdecl export unid main unid sint32_t_sint0_t_t  
fdecl import unid printf unid sint32_t_va_t
```

Declaring Constant String

- C (declaration for “%d\n”)

```
printf( "%d\n" , x ) ;
```

- MIRV

```
cdecl unid const_8 unid sint8_t_4_t "%d\n\0"
```

Variable Declaration

- Unique names for all local variables

- C

```
void foo() { int a; int b; }  
void main() { int a; }
```

- MIRV

```
fdef unid fib {  
  vdecl internal unid a_1 unid sint32_t {}  
  vdecl internal unid b_2 unid sint32_t {}  
}  
fdef unid main {  
  vdecl internal unid a_3 unid sint32_t {}  
}
```

- Rename all local variables
 - Keep a global counter in Bison
- NOTE: all global variables and functions need to retain their original name!

Function Argument Declaration

- C

```
int foo (int a, int b) { ... }
```

- MIRV

```
fdef unid foo {  
    retDecl unid foo.__retval unid sint32_t  
    argDecl unid a_1 unid sint32_t  
    argDecl unid b_2 unid sint32_t  
    ...  
}
```

Attributes

- Not used in this class, but MIRV backend needs them
- Defined before every type, function and constant declaration. (technically not necessary, but ...)

– i.e.

```
attribute {  
    name "my_global"  
}
```

```
vdecl uid my_global uid sint32_t {}
```

Attributes

- Variables need more (these are safe defaults)

```
attribute {  
    name "my_local"  
    register false  
    addrof true  
    temp false  
}
```

- Return (`retdecl`) attribute is a bit different, please refer to the sample MIRV output

Definitions

- Specify the “Value” of a Symbol
 - Types: none
 - Constants: none (in declaration)
 - Functions: body (the “guts” of the program)
 - Variables: assignment (in function definition)

Definition Example

- C

```
void foo(void) { int local; }
```

- MIRV

```
<declarations>
```

```
fdef unid main {
```

```
    vdecl internal unid foo.local_1 unid sint32_t
```

```
    return
```

```
}
```

Expressions

- Reference values:
 - By symbol name: (vref, cref, aref)
 - Through indirection: (deref, airef)
- Specify computation:
 - Arithmetic (+, -, /, *)
 - Comparison (<, <=, ==, !=, >=, >)
 - Boolean evaluation (cand, cor, not)
 - Casting/Addressing (cast, addrOf)
- Prefix Form (“Easy” to Translate)

Referencing Data

- **C**

```
int scalar; int *pointer;  
scalar  
pointer  
*pointer  
10
```

- **MIRV**

```
vref unid scalar  
vref unid pointer  
deref vref unid pointer  
cref unid c10_c
```

Array References

- C

```
int array[10];  
int *p;  
array[2];  
p[i];
```

- MIRV

```
aref unid array {  
    cref unid c8_c  
}  
airef unid p {  
    #Remember, we address bytes!  
    * vref unid i  
    cref unid c4_c  
}
```

Statements

- Specify Machine State Changes
 - Data value update (assignment)
 - Control sequencing (program counter update)
 - Block ({ }, a list of sequential statements)
 - Conditionals (if, ifElse)
 - Loops (while)
 - Function calls (fcall)
 - Function returns (return)
- NOTE: Function calls are not expressions as in C—
 - Need to create temporary variables to handle them
- Also “prefix” form

Statement Grammar

- assign <expr> <expr>
- { <stmt_list> }
- if <expr> <true_block>
- ifElse <expr><true_block><>false_block>
- while <expr> <body_block>
 - NOTE: for “For” loop, initialization before while statement, update in <body_block>
- fcall unid <name> { <param_list> }
- return

Assignments

- **C**

```
int foo(int a, char b) { return (a+b); }
```

- **MIRV**

```
<declarations>
```

```
fdef unid foo {
```

```
  <declarations>
```

```
  assign vref unid foo.__retval
```

```
    + vref unid foo.a_1
```

```
      cast unid sint32_t vref unid foo.b_2
```

```
  return
```

```
}
```

Conditionals

- C

```
if (x<10)
    x = x + 1;
else
    x = 0;
```

- MIRV

```
ifElse < vref unid x
    cref unid c10_c {
    assign vref unid x
        + vref unid x
        cref unid c1_c
    }{
    assign vref unid x
        cref unid c0_c
    }
```

Loops

- **C**

```
for (x=0; x<10; x=x+1) { <stmts> }
```

- **MIRV**

```
assign vref unid x
      cref unid c0_c
while < vref unid c
      cref unid c10_c {
  <stmts>
  assign vref unid x
        + vref unid x
        cref unid c1_c
}
```

Function Calls

- C

```
void foo(void) {  
    bar();  
    a = foo() + 1;  
}
```

- MIRV

```
fcall unid bar {}  
assign vref unid foo.__tmp_2  
    fcall unid foo {}  
assign vref unid a  
    + vref unid foo.__tmp_2  
    cref unid c1_c  
}
```

Vararg Function Calls

- C

```
char c;  
printf("%c", c);
```

- MIRV

```
fcall unid printf {  
  cast unid sint8_t_p_t  
  addrOf unid sint8_t_3_t_p_t  
  cref unid string_l  
  cast unid sint32_t  
  vref unid c  
}
```

NOTE: remember the promotions!

Gotchas: “Array” Assignment

- C

```
int array[10];  
int *p;  
p = array;
```

- MIRV

```
assign vref unid p  
  cast unid sint32_t_p_t  
    addrOf unid sint32_t_10_t_p_t  
      vref unid array
```

– Arrays are “first-class” objects!

Gotchas: Pointer Arithmetic

- C

```
int *p;
```

```
p = p + i;
```

- MIRV

```
assign vref unid p
```

```
    + vref unid p
```

```
        * vref unid i
```

```
        cref unid c4_c
```

– This is like `&p[i]`, so we must scale the address

Questions

- Please post questions on the newsgroup so other people can benefit from them too
- Get started on the project now!