Compiler Construction

Designing an Intermediate Representation

Format? Representation? Language?

Intermediate representation:

- a faithful model of the source program
- "written" in an abstract language, the *intermediate language*
- may have an external syntax
- may be interpreted/compiled (HAVM, byte code)
- may have different levels (GCC's Tree is very much like C).

What Language Flavor?

- **Imperative?**
  - ▸ **Stack Based?** (Java Byte-code)
  - ▸ Register Based? (GCC's RTL, tC's Tree)

- **Functional?**
  Most functional languages are compiled into a lower level language, eventually a simple λ-calculus.

- **Other?**

What Level?

A whole range of expressivities, typically aiming at making some optimizations easier:

- Keep array expressions?
  - Yes: adequate for dependency analysis and related optimizations,
  - No: Good for constant folding, strength reduction, loop invariant code motion, etc.
- Keep loop constructs?

What level of machine independence?

- Explicit register names?

Designing an Intermediate Representation

Intermediate-language design is largely an art, not a science.

–
Muchnick, 1997

Different Levels [Muchnick, 97]

```
float a[20][10];
...
a[i][j+2];
```

Different Levels [Muchnick, 97]

```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

Different Levels [Muchnick, 97]

```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```
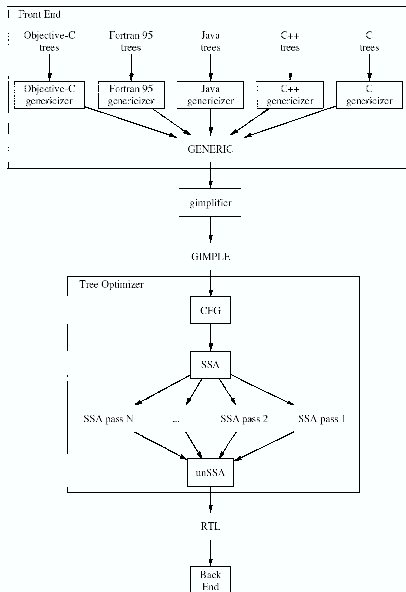
Different Levels [Muchnick, 97]

```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

```
r1 <- [fp - 4]
r2 <- r1 + 2
r3 <- [fp - 8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp - 216
f1 <- [r7 + r6]
```

Different Levels: The GCC Structure

Stack Based: Java Byte-Code [Edwards, 03]

```java
class Gcd {
  static public int gcd(int a,
                        int b){
    while (a != b)
      {
        if (a > b)
          a -= b;
        else
          b -= a;
      }
    return a;
  }
  static public
  int main(String[] arg){
    return gcd(12, 34);
  }
}
```

Stack Based: Java Byte-Code

```
% gcj-3.3 -c gcd.java
% jcf-dump-3.3 -c gcd
...
Method name:"gcd" public static
Signature: 5=(int,int)int
Attribute "Code", length:66,
max_stack:2, max_locals:2,
code_length:26
  0: iload_0
  1: iload_1
  2: if_icmpeq 24
  5: iload_0
  6: iload_1
  7: if_icmple 17
 10: iload_0
 11: iload_1
 12: isub
```

```
 13: istore_0
 14: goto 21
 17: iload_1
 18: iload_0
 19: isub
 20: istore_1
 21: goto 0
 24: iload_0
 25: ireturn
Attribute "LineNumberTable",
        length:22, count: 5
  line: 5 at pc: 0
  line: 7 at pc: 5
  line: 8 at pc: 10
  line: 10 at pc: 17
  line: 12 at pc: 24
```

Stack Based

Advantages

- Trivial translation of expressions
- Trivial interpreters
- No pressure on registers
- Often compact

Stack Based

Advantages

- Trivial translation of expressions
- Trivial interpreters
- No pressure on registers
- Often compact

Disadvantages

- Does not fit with today's architectures
- Hard to analyze
- Hard to optimize

Register Based: What Structure?

How is the structure coded?

Addresses   Expressions and instructions
            have names, or (absolute)
            addresses. (Stack based is a
            bit like a relative address).

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions
have names, or (absolute)
addresses. (Stack based is a
bit like a relative address).

- 2 address instructions?
  (**triples**)

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions
have names, or (absolute)
addresses. (Stack based is a
bit like a relative address).

- 2 address instructions?
  (**triples**)
- 3 address instructions?
  (**quadruples**)

Register Based: What Structure?

How is the structure coded?

Addresses   Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).

- 2 address instructions? (**triples**)
- 3 address instructions? (**quadruples**)

Tree   Expressions and instructions are unnamed, related to each other as nodes of trees

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions
have names, or (absolute)
addresses. (Stack based is a
bit like a relative address).

- 2 address instructions?
  (**triples**)
- 3 address instructions?
  (**quadruples**)

Tree Expressions and instructions
are unnamed, related to each
other as nodes of trees

DAG Compact, good for local value
numbering, but that's all.

Quadruples vs. Triples

```
L1:  i  <- i + 1

     t1 <- i + 1
     t2 <- p + 4
     t3 <- *t2
     p  <- t2
     t4 <- t1 < 10
     *r <- t3
     if t4 goto L1
```

Quadruples vs. Triples

```
L1:  i  <- i + 1

     t1 <- i + 1
     t2 <- p + 4
     t3 <- *t2
     p  <- t2
     t4 <- t1 < 10
     *r <- t3
     if t4 goto L1
```

```
(1)  i + 1
(2)  i sto (1)
(3)  i + 1
(4)  p + 4
(5)  *(4)
(6)  p sto (4)
(7)  (3) < 10
(8)  *r sto (5)
(9)  if (7), (1)
```

Register Based: GCC's RTL

```
int
gcd(int a, int b)
{
  while (a != b)
    {
      if (a > b)
        a -= b;
      else
        b -= a;
    }
  return a;
}
```

Register Based: GCC's RTL

```
;; Function gcd
(note 1 0 2 ("gcd.c") 3)
(note 2 1 3 NOTE_INSN_DELETED)
(note 3 2 4 NOTE_INSN_FUNCTION_BEG)
(note 4 3 5 NOTE_INSN_DELETED)
(note 5 4 6 NOTE_INSN_DELETED)
(note 6 5 7 NOTE_INSN_DELETED)
(insn 7 6 8 (const_int 0 [0x0]) -1 (nil)
    (nil))
```

Register Based: GCC's RTL cont'd

```
( note  8  7  9  ("gcd.c")  4)
( note  9  8  40  NOTE_INSN_LOOP_BEG)
( note  40  9  10  NOTE_INSN_LOOP_CONT)
( code_label  10  40  13  2  ""  ""  [0 uses])
( insn  13  10  14  ( set ( reg:SI 59)
        (mem/f:SI ( reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
    -1 ( nil) ( nil))
( insn  14  13  15  ( set ( reg:CCZ 17 flags)
        ( compare:CCZ ( reg:SI 59)
            (mem/f:SI ( plus:SI ( reg/f:SI 53 virtual-incoming-args)
                    ( const_int 4 [0x4])) [0 b+0 S4 A32]))) -1 ( nil)
    ( nil))
```

Register Based: GCC's RTL cont'd

```
(jump_insn 15 14 16 (set (pc)
        (if_then_else (ne (reg:CCZ 17 flags)
                (const_int 0 [0x0]))
            (label_ref 18)
            (pc))) -1 (nil)
    (nil))
(jump_insn 16 15 17 (set (pc)
        (label_ref 44)) -1 (nil)
    (nil))
(barrier 17 16 18)
(code_label 18 17 19 4 "" "" [0 uses])
(note 19 18 20 NOTE_INSN_LOOP_END_TOP_COND)
(note 20 19 21 NOTE_INSN_DELETED)
(note 21 20 22 NOTE_INSN_DELETED)
```

Register Based: GCC's RTL cont'd

```
(note 22 21 25 ("gcd.c") 6)
(insn 25 22 26 (set (reg:SI 60)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
     -1 (nil) (nil))
(insn 26 25 27 (set (reg:CCGC 17 flags)
        (compare:CCGC (reg:SI 60)
            (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                    (const_int 4 [0x4])) [0 b+0 S4 A32]))) -1 (nil)
    (nil))
(jump_insn 27 26 28 (set (pc)
        (if_then_else (le (reg:CCGC 17 flags)
                (const_int 0 [0x0]))
            (label_ref 34)
            (pc))) -1 (nil)
    (nil))
```

Register Based: GCC's RTL cont'd

```
(note 28 27 30 ("gcd.c") 7)
(insn 30 28 31 (set (reg:SI 61)
        (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                (const_int 4 [0x4])) [0 b+0 S4 A32])) -1 (nil)
    (nil))
(insn 31 30 32 (parallel[
            (set (mem/f:SI (reg/f:SI 53 virtual-incoming-args)
                [0 a+0 S4 A32])
                (minus:SI (mem/f:SI (reg/f:SI 53 virtual-incoming-args)
                [0 a+0 S4 A32])
                    (reg:SI 61)))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (expr_list:REG_EQUAL (minus:SI (mem/f:SI (reg/f:SI 53
                            virtual-incoming-args) [0 a+0 S4 A32])
            (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                (const_int 4 [0x4])) [0 b+0 S4 A32]))
        (nil)))
(jump_insn 32 31 33 (set (pc)
        (label_ref 39)) -1 (nil)
```

Register Based: GCC's RTL cont'd

```
(note 35 34 37 ("gcd.c") 9)
(insn 37 35 38 (set (reg:SI 62)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
    -1 (nil) (nil))
(insn 38 37 39 (parallel[
            (set (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                        (const_int 4 [0x4])) [0 b+0 S4 A32])
                (minus:SI (mem/f:SI (plus:SI (reg/f:SI 53
                                                virtual-incoming-args)
                            (const_int 4 [0x4])) [0 b+0 S4 A32])
                    (reg:SI 62)))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (expr_list:REG_EQUAL (minus:SI (mem/f:SI (plus:SI (reg/f:SI
                                                53 virtual-incoming-args)
                    (const_int 4 [0x4])) [0 b+0 S4 A32])
            (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
        (nil)))
(code_label 39 38 41 6 "" "" [0 uses])
(jump_insn 41 39 42 (set (pc)
```

Register Based: GCC's RTL cont'd

```
(note 45 44 46 ("gcd.c") 11)
(note 46 45 47 NOTE_INSN_DELETED)
(note 47 46 49 NOTE_INSN_DELETED)
(insn 49 47 51 (set (reg:SI 64)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32])) -1
    (nil))
(insn 51 49 52 (set (reg:SI 58)
        (reg:SI 64)) -1 (nil)
    (nil))
(jump_insn 52 51 53 (set (pc)
        (label_ref 56)) -1 (nil)
    (nil))
(barrier 53 52 54)
(note 54 53 55 NOTE_INSN_FUNCTION_END)
(note 55 54 59 ("gcd.c") 12)
(insn 59 55 60 (clobber (reg/i:SI 0 eax)) -1 (nil)
    (nil))
(insn 60 59 56 (clobber (reg:SI 58)) -1 (nil)
    (nil))
(code_label 56 60 58 1 "" "" [0 uses])
```

Register Based

Advantages

- Suits today's architectures
- Clearer data flow

Register Based

Advantages
- Suits today's architectures
- Clearer data flow

Disadvantages
- Harder to synthesize
- Less compact
- Harder to interpret

Tree [Appel, 98]

A simple intermediate language:

- Tree structure (no kidding...)
- Unbounded number of registers (temporaries)
- Two way conditional jump

Tree: Grammar

```
<Exp>  ::= "const" int
         | "name" <Label>
         | "temp" <Temp>
         | "binop" <Oper> <Exp> <Exp>
         | "mem" <Exp>
         | "call" <Exp> [{<Exp>}] "call end"
         | "eseq" <Stm> <Exp>

<Stm>  ::= "move" <Exp> <Exp>
         | "sxp" <Exp>
         | "jump" <Exp> [{<Label>}]
         | "cjump" <Relop> <Exp> <Exp> <Label> <Label>
         | "seq" [{<Stm>}] "seq end"
         | "label" <Label>

<Oper>  ::= "add" | "sub" | "mul" | "div" | "mod"
<Relop> ::= "eq" | "ne" | "lt" | "gt" | "le" | "ge"
```

Tree Samples

```
% echo '1 + 2 * 3' | tc -H -
```

```
/* == High Level Intermediate representation. == */
# Routine: Main Program
label Main
# Prologue
# Body
sxp
    binop add
        const 1
        binop mul
            const 2
            const 3
# Epilogue
label end
```

Tree Samples

```
% echo 'if 1 then print_int (1)' | tc -H -
```

```
# Routine: Main Program
label Main
# Prologue
# Body
seq
    cjump ne, const 1, const 0, name l1, name l2
    label l1
    sxp call name print_int, const 1
    jump name l3
    label l2
    sxp const 0
    label l3
seq end
# Epilogue
label end
```

Tree samples

```
let function gcd(a: int, b: int) : int =
    (
      while a <> b
        do if a > b then a := a - b
                    else b := b - a;
      a
    )
in
  print_int(gcd(42, 51))
end
```

Tree samples (1/4)

```
/* == High Level Intermediate representation. == */
# Routine: gcd
label l0
# Prologue
move temp t0 temp fp
move temp fp temp sp
move
  temp sp
  binop sub temp sp const 12
move
mem temp fp
    temp i0
move
  mem binop add temp fp const -4
  temp i1
move
  mem binop add temp fp  const -8
```

Tree samples (2/4)

```
# Body
move temp rv
  eseq
  seq
    label l2
    cjump ne mem binop add temp fp const −4
             mem binop add temp fp const −8
             name l3 name l1
    label l3
    seq
      cjump gt mem binop add temp fp const −4
               mem binop add temp fp const −8
               name l4 name l5
      label l4
      move mem binop add temp fp const −4
           binop sub mem binop add temp fp const −4
                     mem binop add temp fp const −8
      jump name l6
```

Tree samples (3/4)

```
        label l5
        move mem binop add temp fp const -8
              binop sub mem binop add temp fp const -8
                        mem binop add temp fp const -4
        label l6
     seq end
     jump name l2
     label l1
   seq end
   mem  binop add temp fp const -4

# Epilogue
move temp sp temp fp
move temp fp temp t0
label end
```

Tree samples (4/4)

```
# Routine: _main
label main
# Prologue
# Body
seq
  sxp
    call
      name print_int
      call name l0 temp fp const 42 const 51
      call end
    call end
  sxp
    const 0
seq end
# Epilogue
label end
```

Summary

Register based

Stack based

Tree

Triples / quadruples