

CSc 453

Compilers and Systems Software

19 : Code Generation I

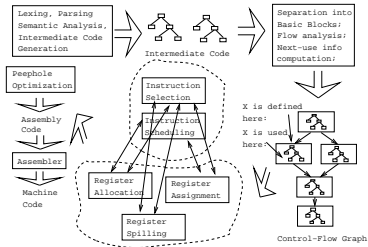
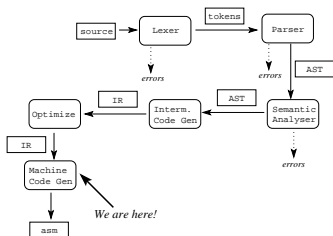
Department of Computer Science
University of Arizona

collberg@gmail.com

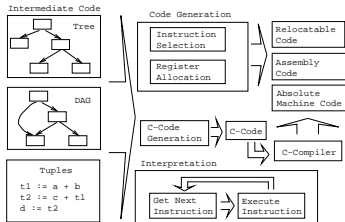
Copyright © 2009 Christian Collberg

Introduction

Compiler Phases



- The purpose of the code generation phase of the compiler is to transform the intermediate code produced by the front end into some other code that can be executed.
- Often the the code generator will produce assembly code or object code which (after assembly and linking) can be directly executed by the hardware.
- Alternatively, the code generator can generate C-code and use the native C-compiler as the "real" back-end.
- Or, the code generator can generate code for a "virtual machine", and use an **interpreter** to execute the code.
- We expect the code generator to produce code that is as efficient as possible.



- The input to the code generator can be any one of the intermediate representations we've discussed: Trees, Tuples, Graphs, . . . The work of the code generator consists of several (interdependent) tasks:

Instruction

- **selection:** Which instructions should be generated?
- **scheduling:** In which order should they be generated?

Register

- **allocation:** Which variables should be kept in registers?
- **assignment:** In which registers should they be stored?
- **spilling:** Which registers should be spilled when?

Architectures

- 3-Register: $\text{add } R1, R2, R3$
 $[R1 := R2 + R3]$ (MIPS,VAX,...).
- Register-Address: $\text{add } R, \text{Addr}$
 $[R := R + \text{Addr}]$ (VAX,x86,MC68k)
- 2-Register: $\text{add } R1, R2$
 $[R1 := R1 + R2]$ (VAX,x86,MC68k)
- 2-Address: $\text{add } \text{Addr1}, \text{Addr2}$
 $[\text{Addr1} := \text{Addr1} + \text{Addr2}]$ (VAX)
- 3-Address: $\text{add } \text{Addr1}, \text{Addr2}, \text{Addr3}$
 $[\text{Addr1} := \text{Addr2} + \text{Addr3}]$ (VAX)

General One set of register that can hold any type of data (VAX, Alpha).

Integer+Float Separate integer and floating point register sets (Sparc, MIPS).

Integer+Float+Address Separate integer, floating point, and address register sets (MC68k).

- Immediate: $\#X$ The value of the constant X . (All architectures.)
- Register Direct: R The contents of register R . (All.)
- Register Indirect: (R) The contents of the memory address in register R . (All.)
- Register Indirect with increment: $(R+)$ The contents of the memory address in register R . R is incremented by the **size** of the instruction (i.e. if $\text{MOVE.W } (R+)$, Addr moves two bytes, then R would be incremented by 2). (VAX, MC68k.)
- Register Ind. with Displacement: $d(R)$ The contents of the memory address $R+d$, where R is a register and d a (small) constant. (All architectures.)

- The Cost of an instruction is the number of machine cycles it takes to execute it.
- On RISCs, most instructions take 1 cycle to execute. Loads, stores, branches, multiplies, and divides may take longer.
- On CISCs, the number of cycles required to execute an instruction $\text{Instr } Op_1, Op_2$ is $\text{cost}(\text{Instr}) + \text{cost}(Op_1) + \text{cost}(Op_2)$. $\text{cost}(Op_i)$ is the number of cycles required to compute the addressing mode Op_i .

A Simple Example

- A straight-forward code generator considers one tuple at a time, without looking at other tuples. The code generator is simple, but the generated code is sub-optimal.

_____ The Source Program: _____

```
int A[5], i, x;
main(){
    for(i=1;i<=5;i++)
        x=x*A[i]+A[i];
}
```

Example — Intermediate Code

```
int A[5], i, x;
main(){for(i=1;i<=5;i++) x=x*A[i]+A[i];}
```

The Tuple Code

| | |
|----------------------|------------------|
| (1) i := 1 | (9) T5 := i |
| (2) T0 := i | (10) T6 := A[T5] |
| (3) IF T0<6 GOTO (5) | (11) T7 := T4+T6 |
| (4) GOTO (17) | (12) x := T7 |
| (5) T1 := i | (13) T8 := i |
| (6) T2 := A[T1] | (14) T9 := T8+1 |
| (7) T3 := x | (15) i := T9 |
| (8) T4 := T2*T3 | (16) GOTO (2) |

Example – Unoptimized MIPS Code

```
(1) i := 1
li $2,0x1          # $2 := 1
sw $2,i           # i := $2

L2: (2) T0 := i
lw $2,i           # $2 := i
(3) IF i < 6 GOTO (5)
slt $3,$2,6       # $3 := i < 6
bne $3,$0,L5     # IF $3≠0 GOTO L5
(4) GOTO (17)
j L3              # GOTO L3
```

| | | | | |
|------|-------------------|---------------------|--------------------|--------------------------------|
| L5: | (5) T1 := i | | (9) T5 := i | |
| lw | \$2,i | # \$2 := CONT(i) | lw | \$2,i # \$2 := CONT(i) |
| | (6) T2 := A[T1] | | (10) T6 := A[T5] | |
| move | \$3,\$2 | # \$3 := \$2 | move | \$3,\$2 # \$3 := \$2 |
| sll | \$2,\$3,2 | # \$2 := \$3 * 4 | sll | \$2,\$3,2 # \$2 := \$3 * 4 |
| la | \$3,A | # \$3 := ADDR(A) | la | \$3,A # \$3 := ADDR(A) |
| addu | \$2,\$2,\$3 | # \$2 := \$2 + \$3 | addu | \$2,\$2,\$3 # \$2 := \$2 + \$3 |
| lw | \$2,0(\$2) | # \$2 := CONT(A[i]) | lw | \$3,0(\$2) # \$2 := CONT(A[i]) |
| | (7) T3 := x | | | |
| lw | \$3,x | # \$3 := CONT(x); | | |
| | (8) T4 := T2 * T3 | | (11) T7 := T4 + T6 | |
| mult | \$3,\$2 | # \$lo := \$3 * \$2 | addu | \$2,\$4,\$3 # \$2 := \$4 + \$3 |
| mflo | \$4 | # \$4 := \$lo | (12) x := T7 | |
| | | | sw | \$2,x # x := \$2 |

◀ ▶ ⏪ ⏩ 🔍 🔄

◀ ▶ ⏪ ⏩ 🔍 🔄

| | | |
|------|-------------------|------------------|
| | (13) T8 := i | |
| lw | \$3,i | # \$3 := CONT(i) |
| | (14) T9 := T8 + 1 | |
| addu | \$2,\$3,1 | # \$2 := \$3 + 1 |
| move | \$3,\$2 | # \$3 := \$2 |
| | (15) i := T9 | |
| sw | \$3,i | # i := \$3 |
| | (16) GOTO (2) | |
| j | L2 | # GOTO L2 |
| L3: | | |

Common Sub-expression Elimination

◀ ▶ ⏪ ⏩ 🔍 🔄

◀ ▶ ⏪ ⏩ 🔍 🔄

- The generated code becomes a lot faster if we perform Common Sub-Expression Elimination (CSE) and keep the index variable `i` in a register (`$6`) over the entire loop:

```

(1) i := 1
li    $6,0x1          # $6 := 1

L2:   (2) T0 := i
      (3) IF i < 6 GOTO (5)
      slt $3,$6,6      # $3 := i < 6
      bne $3,$0,L5     # IF $3≠0 GOTO L5
      (4) GOTO (17)
      j    L3           # GOTO L3

```

⏪ ⏩ ⏴ ⏵ 🔍

- `A[T1]` is computed once, and the result is kept in register `$5` until it's needed the next time.

```

L5:   (5) T1 := i
      (6) T2 := A[T1]
      move $3,$6       # $3 := $6
      sll $2,$3,2      # $2 := $3 * 4
      la  $3,A         # $3 := ADDR(A)
      addu $2,$2,$3    # $2 := $2 + $3
      lw  $5,0($2)     # $5 := CONT(A[i])
      (7) T3 := x
      lw  $3,x         # $3 := CONT(x);
      (8) T4 := T2 * T3
      mult $3,$5       # $lo := $3 * $5
      mflo $4         # $4 := $lo

```

⏪ ⏩ ⏴ ⏵ 🔍

- After the loop we need to store `$6` back into `i`.

```

(9) T5 := i
(10) T6 := A[T5]
(11) T7 := T4 + T6
addu $2,$4,$5       # $2 := $4 + $5
(12) x := T7
sw  $2,x            # x := $2
(13) T8 := i
(14) T9 := T8 + 1
(15) i := T9
addu $6,$6,1        # $6 := $6 + 1
(16) GOTO (2)
j    L2             # GOTO L2
L3:sw $6,i          # i := $6

```

⏪ ⏩ ⏴ ⏵ 🔍

More Optimization

⏪ ⏩ ⏴ ⏵ 🔍

Example — More Register Allocation

- Since `x` and `ADDR(A)` seem to be used a lot in the loop, we keep them in registers (`$7` and `$8`, respectively) as well.
- We also reverse the comparison, which allows us to remove one jump.
- The `move` instruction is unnecessary, so we remove it also.

```
(1) i := 1
li   $6,0x1          # $6 := 1

lw   $7,x           # $7 := CONT(x);
la   $8,A           # $8 := ADDR(A)
```

```
L2:   (2) T0 := i
      (3) IF i < 6 GOTO (5)
      (4) GOTO (17)

      sge $3,$6,6      # $3 := i >= 6
      bne $3,$0,L3    # IF $3≠0 GOTO L3

L5:   (5) T1 := i
      (6) T2 := A[T1]

      sll $2,$6,2      # $2 := $3 * 4
      addu $2,$2,$8    # $2 := $2 + $8
      lw  $5,0($2)     # $5 := CONT(A[i])

      (7) T3 := x
      (8) T4 := T2 * T3

      mult $7,$5       # $lo := $7 * $5
      mflo $4          # $4 := $lo
```

◀ ▶ ↻ 🔍

◀ ▶ ↻ 🔍

```
(9) T5 := i
(10) T6 := A[T5]
(11) T7 := T4 + T6
(12) x := T7

addu $7,$4,$5      # $7 := $4 + $5

(13) T8 := i
(14) T9 := T8 + 1
(15) i := T9
(16) GOTO (2)

j    L2            # GOTO L2
L3:sw $6,i         # i := $6
sw  $7,x         # x := $7
```

◀ ▶ ↻ 🔍

◀ ▶ ↻ 🔍

Example — Summary

- The unoptimized code (produced by `gcc -S -g`) was 28 instructions long. Our optimized code is 16 instructions. Improvement: 42%.
- More importantly, in the original code there were 26 instructions **inside the loop**, and 2 outside. Since the loop runs 5 times, **we will execute** $3 + 5 * 25 = 128$ instructions.
- In the optimized case, we have 11 instructions in the loop and 5 outside. We will execute only $5 + 5 * 11 = 60$ instructions. Improvement: 53%.

Instruction Selection

- Instruction selection is usually pretty simple on RISC architectures – there is often just one possible sequence of instructions to perform a particular kind of computation.
- CISC's like the VAX, on the other hand, leave the compiler with more choices: `ADD2 1, R1` `ADD3 R1, 1, R1` `INC R1` all add 1 to register R1.

V * 2 – Unoptimized Sparc Code

```

set  V, %o0           # %o0 := ADDR(V);
ld   [%o0], %o0      # %o0 := CONT(V);
set  2, %o1           # %o1 := 2;
call .mul, 2         # %o0 := %o0 * %o1;
nop                    # Empty delay slot

```

V * 2 – Better Instr. Selection

- The Sparc has a library function `.mul` and a hardware multiply instruction `smul`:

```

set  V, %o0
ld   [%o0], %o0
smul %o0, 1, %o0      # %o0 := %o0 * %o1;

```

V * 2 – Even Better Instr. Selection

- The Sparc also has hardware shift instructions (`sll`, `sr1`). To multiply by 2^i we shift i steps to the left.

```

set  V, %o0
ld   [%o0], %o0
sll  %o0, 1, %o0      # %o0 := %o0 * 2;

```

Instruction Scheduling

- Instruction scheduling is important for architectures with several functional units, pipelines, delay slots. I.e. most modern architectures.
- The Sparc (and other RISCs) have **branch delay slots**. These are instructions (textually immediately following the branch) that are "executed for free" during the branch.

V * 2 – Unoptimized Sparc Code

```

ld   [%o0], %o0      # %o0 := CONT(V);
set  2, %o1           # %o1 := 2;
call .mul, 2         # %o0 := %o0 * %o1;
nop                    # Empty delay slot

```


_____ V * 2 – Unoptimized Sparc Code _____

```
ld    [%o0], %o0           # %o0 := CONT(V);
set   2, %o1               # %o1 := 2;
call  .mul, 2              # %o0 := %o0 * %o1;
nop                                     # Empty delay slot
```

_____ V * 2 – Better Instr. Scheduling _____

```
ld    [%o0], %o0           # %o0 := CONT(V);
call  .mul, 2
set   2, %o1               # Filled delay slot
```

- The Sparc's integer and floating point units can execute in parallel. Integer and floating point instructions should therefore be reordered so that operations are interleaved.

```
int a, b, c; double x, y, z;
{
    a = b - c;
    c = a + b;
    b = a + c;
    y = x * x;
    z = x + y;
    x = y / z;
}
```

◀ ▶ ⏪ ⏩ 🔍 ↺

◀ ▶ ⏪ ⏩ 🔍 ↺

```
int a, b, c; double x, y, z;
{
    a = b - c; c = a + b; b = a + c;
    y = x * x; z = x + y; x = y / z;}
```

| cc -02 | cc -03 |
|----------------------|----------------------|
| set b,%o3 | fmuld %f30,%f30,%f28 |
| sub %o0,%o1,%o1 | set c,%o1 |
| set a,%o0 | ld [%o1],%o2 |
| add %o4,%o5,%o4 | faddd %f30,%f28,%f30 |
| add %o0,%o2,%o0 | set b,%o0 |
| set x, %o0 | ld [%o0],%o4 |
| fmuld %f0,%f2,%f0 | set z,%g1 |
| sethi %hi(z),%o2 | sub %o4,%o2,%o2 |
| faddd %f6,%f8,%f6 | fdivd %f28,%f30,%f2 |
| fdivd %f12,%f14,%f12 | add %o4,%o2,%o4 |
| | add %o2,%o4,%o5 |

Register Allocation/Assignment/Spilling

◀ ▶ ⏪ ⏩ 🔍 ↺

◀ ▶ ⏪ ⏩ 🔍 ↺

- 1 We only need 4–7 bits to access a register, but 32–64 bits to access a memory word.
- 2 Hence, a one-word instruction can reference 3 registers but a two-word instruction is necessary to reference a memory word.
- 3 Registers have short access time.

- 1 Instructions take operands in regs.
- 2 Intermediate results are stored in regs.
- 3 Procedure arguments are passed in regs.
- 4 Loads and Stores are expensive \Rightarrow keep variables in regs for as long as possible.
- 5 Common sub-expressions are stored in regs.

Register Allocation/Assignment

Register Allocation:

- First we have to decide which variables should reside in registers at which point in the program.
- Variables that are used frequently should be favored.

Register Assignment:

- Secondly, we have to decide which physical registers should hold each of these variables.
- Some architectures have several different **register classes**, groups of registers that can only hold one type of data:
 - MIPS & Sparc have floating point and integer registers;
 - MC68k has address, integer, and floating point, etc.

Register Assignment

- Sparc passes its first 6 arguments in registers %o0, %o1, %o2, %o3, %o4, %o5.
- If a value is used twice, first in a computation and then in a procedure call, we should allocate the value to the appropriate procedure argument register.

```

a = b + 15;      /* ← b is used here */
P(b);           /* ← and here.  */
  ↓ ↓ ↓
ld    [%fp-8], %o0      # %o0 := CONT(b);
add   %o0, 15, %o1     # %o1 := %o0 + 15
st    %o1, [%fp-4]     # a := %o1;
call  P, 1             # P(%o0)
  
```

- We may have 8 | 16 | 32 regs available.
- When we run out of registers (during code generation) we need to pick a register to **spill**. I.e. in order to free the register for its new use, it's current value first has to be stored in memory.
- Which register should be spilt? Least recently used, Least frequently used, Most distant use, ... (take your pick).

- Assume a machine with registers R1--R3.
- R1 holds variable a; R2 holds b, R3 holds c, and R4 holds d. Generate code for:

```
x = a + b;           # ← Which reg for x?
y = x + c;
```

- Which register should be spilt to free a register to hold x?

```
FOR i := 1 TO n DO
  B[5,i] := b * b * b;
  FOR j := 1 TO n DO
    FOR k := 1 TO n DO
      A[i,j] := A[i,k] * A[k,j];
```

- 2 Registers Available: k and ADDR(A) in registers. (Prefer variables in inner loops).
- 4 Registers Available: k, ADDR(A), j, and i in registers. (Prefer index variables).
- 5 Registers Available: k, ADDR(A), j, i, and b in registers. (Prefer most frequently used variables).

```
FOR i := 1 TO 100000 DO
  A[5,i] := b;
  FOR j := 1 TO 100000 DO
    A[j,i] := <Complicated Expression>;
```

- _____ 1st Attempt (4 Regs available): _____
- Allocation/Assignment: i in R₁, j in R₂, ADDR(A) in R₃, ADDR(A[5, _]) in R₄.
- Spilling: Spill R₄ in the inner loop to get enough registers to evaluate the complicated expression.

```

FOR i := 1 TO 100000 DO
  A[S,i] := b;
  FOR j := 1 TO 100000 DO
    A[j,i] := <Complicated Expression>;

```

_____ 2nd Attempt (4 Regs available): _____
 Allocation/Assignment: i in R_1 , j in R_2 , ADDR(A) in R_3 .

Spilling: No spills. But ADDR(A[5,i]) must be loaded every time in the outer loop.

Summary

- Read Louden:
 - Basic code generation 407–416
 - Data structures 416–428
 - Control structures 428–436
 - Procedure calls 436–443
- Read the Dragon book:
 - Introduction 513–521
 - Basic Blocks 528–530
 - Flow Graphs 532–534
- Instruction selection picks which instruction to use, instruction scheduling picks the ordering of instructions.
- Register allocation picks which variables to keep in registers, register assignment picks the actual register in which a particular variable should be stored.
- We prefer to keep index variables and variables used in inner loops in registers.
- When we run out of registers, we have to pick a register to *spill*, i.e. to store back into memory. We avoid inserting spill code in inner loops.

- Code generation checklist:
 - 1 Is the code correct?
 - 2 Are values kept in registers for as long as possible?
 - 3 Is the cheapest register always chosen for spilling?
 - 4 Are values in inner loops allocated to registers?
- A basic block is a *straight-line* piece of code, with no jumps in or out except at the beginning and end.
- *Local* code generation considers one basic block at a time, *global* one procedure, and *inter-procedural* one program.