

CSc 553

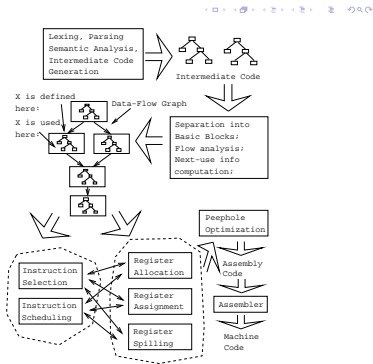
## Principles of Compilation

### 7: Code Generation I

Department of Computer Science  
University of Arizona

[collberg@cs.arizona.edu](mailto:collberg@cs.arizona.edu)

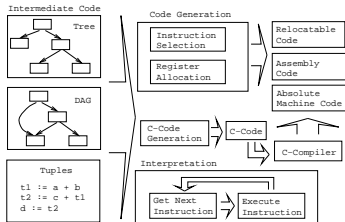
Copyright © 2011 Christian Collberg



## Introduction

### Code Generation Issues I

- 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

### Machine Architectures I

- \_\_\_\_\_ Kinds of Instructions: \_\_\_\_\_
- 3-Register: `add R1, R2, R3`  
 $[R1 := R2 + R3]$  (MIPS, VAX, ...).
- Register-Address: `add R, Addr`  
 $[R := R + Addr]$  (VAX, x86, MC68k)
- 2-Register: `add R1, R2`  
 $[R1 := R1 + R2]$  (VAX, x86, MC68k)
- 2-Address: `add Addr1, Addr2`  
 $[Addr1 := Addr1 + Addr2]$  (VAX)
- 3-Address: `add Addr1, Addr2, Addr3`  
 $[Addr1 := Addr2 + Addr3]$  (VAX)

#### \_\_\_\_\_ Kinds of Register Classes: \_\_\_\_\_

- 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).

## Kinds of Register Classes (cont):

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

## Kinds of Addressing Modes:

**Immediate:**  $\#X$  The value of the constant X. (All architectures.)

**Register Direct:**  $R$  The contents of register R. (All architectures.)

**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 MOVE.W  $(R+)$ , Addr moves two bytes, then R would be incremented by 2). (VAX, MC68k.)

## Kinds of Addressing Modes:

**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:

- 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  $\boxed{\text{Instr Op}_1, \text{Op}_2}$  is  $\text{cost}(\text{Instr}) + \text{cost}(\text{Op}_1) + \text{cost}(\text{Op}_2)$ .  $\text{cost}(\text{Op}_i)$  is the number of cycles required to compute the addressing mode  $\text{Op}_i$ .

## Code Generation Example I

- 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.

```
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)

## A Simple Example

## Code Generation Example II (A)

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
lw $2,i # $2 := CONT(i)

```

```

(11) T7 := T4 + T6
addu $2,$4,$3 # $2 := $4 + $3

```

```

(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:

```

(6) T2 := A[T1]
move $3,$2 # $3 := $2
sll $2,$3,2 # $2 := $3 * 4
la $3,A # $3 := ADDR(A)
addu $2,$2,$3 # $2 := $2 + $3
lw $2,0($2) # $2 := CONT(A[i])

```

```

(7) T3 := x
lw $3,x # $3 := CONT(x);

```

```

(8) T4 := T2 * T3
mult $3,$2 # $lo := $3 * $2
mflo $4 # $4 := $lo

```

```

(9) T5 := i
lw $2,i # $2 := CONT(i)

```

```

(10) T6 := A[T5]
move $3,$2 # $3 := $2

```

```

sll $2,$3,2 # $2 := $3 * 4
la $3,A # $3 := ADDR(A)

```

```

addu $2,$2,$3 # $2 := $2 + $3
lw $3,0($2) # $2 := CONT(A[i])

```

## Code Generation Example III (A)

- 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

```

```

L5:
(5) T1 := i

```

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

```

(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

(9) T5 := i
(10) T6 := A[T5]

```

- After the loop we need to store the value of \$6 which has been used to hold the loop index variable i.

```

(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

```

## Code Generation Example IV (A)

- 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
addu $6,$6,1 # $6 := $6 + 1
(16) GOTO (2)
j L2 # GOTO L2
L3:sw $6,i # i := $6
sw $7,x # x := $7

```

- 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 I

- 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

```

### Instruction Selection II

#### 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`).
- Integer multiplication by  $2^i$  can be implemented as a shift  $i$  steps to the left.

```

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

```

## 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

- 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.

```
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.

- Consider this example program:

```
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;}
```

- How will the generated code be different if the compiler takes advantage of parallel execution, or not?

## Instruction Scheduling II (B)

cc -02	cc -03
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;}	
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

## Why do we need registers?

- 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.

## Register Uses:

- 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:

- 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 (A)

- Some architectures pass procedure arguments in registers. 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.
- Sparc passes its first 6 arguments in registers `%o0,%o1,%o2,%o3,%o4,%o5`.
- See the next slide for an example.

## Register Assignment (B)

```
main () {
    int a,b;
    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 it's 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).

Example: \_\_\_\_\_

- Assume a machine with registers R1--R3.
- R1 holds variable a; R2 holds b, R3 holds c, and R4 holds d.  
Generate code for:  
 $x = a + b;$      #  $\Leftarrow$  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];
    END
  END
END
```

- 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).

## Register Spilling Example

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

\_\_\_\_\_ 1st Attempt (4 Regs available): \_\_\_\_\_

Allocation/Assignment: i in R<sub>1</sub>, j in R<sub>2</sub>, ADDR(A) in R<sub>3</sub>,  
 ADDR(A[5, \_]) in R<sub>4</sub>.

Spilling: Spill R<sub>4</sub> in the inner loop to get enough registers to evaluate the complicated expression.

\_\_\_\_\_ 2nd Attempt (4 Regs available): \_\_\_\_\_

Allocation/Assignment: i in R<sub>1</sub>, j in R<sub>2</sub>, ADDR(A) in R<sub>3</sub>.

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

## Basic Blocks and Flow Graphs

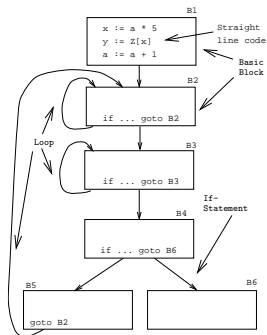
## Basic Blocks and Flow Graphs I

- We divide the intermediate code of each procedure into basic blocks. A basic block is a piece of straight line code, i.e. there are no jumps in or out of the middle of a block.
- The basic blocks within one procedure are organized as a *flow graph*.
- A flowgraph has
  - basic blocks  $B_1 \dots B_n$  as nodes,
  - a directed edge  $B_1 \rightarrow B_2$  if control can flow from  $B_1$  to  $B_2$ .
- Code generation can be performed on a small or large piece of the flow graph at a time (small=easy, large=hard):

Local Within one *basic block*.

Global Within one *procedure*.

Inter-procedural Within one *program*.



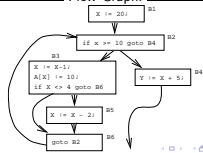
Source Code: \_\_\_\_\_

```
X := 20; WHILE X < 10 DO
  X := X-1; A[X] := 10;
  IF X = 4 THEN X := X - 2; ENDIF;
ENDDO; Y := X + 5;
```

Intermediate Code: \_\_\_\_\_

- |                       |                      |
|-----------------------|----------------------|
| (1) X := 20           | (5) if X<>4 goto (7) |
| (2) if X>=10 goto (8) | (6) X := X-2         |
| (3) X := X-1          | (7) goto (2)         |
| (4) A[X] := 10        | (8) Y := X+5         |

Flow Graph: \_\_\_\_\_



## Constructing Basic Blocks

- How do we identify the basic blocks and build the flow graph?
- Assume that the input to the code generator is a list of tuples. How do we find the beginning and end of each basic block?

Algorithm:

- First determine a set of **leaders**, the first tuple of basic blocks:
  - The first tuple is a leader.
  - Tuple L is a leader if there is a tuple `if ...goto L` or `goto L`
  - Tuple L is a leader if it immediately follows a tuple `if ...goto L` or `goto L`
- A basic block consists of a leader and all the following tuples until the next leader.

Navigation icons: back, forward, search, etc.

Block  $B_1$ : [(1) P:=0; (2) I:=1]

Block  $B_2$ : [(3) P:=P+I;

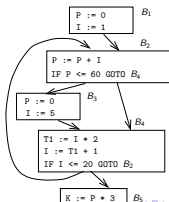
(4) IF P<=60 GOTO  $B_4$ ]

Block  $B_3$ : [(5) P:=0; (6) I:=5]

Block  $B_4$ : [(7) T1:=I\*2; (8) I:=T1+1;

(9) IF I<=20 GOTO  $B_2$ ]

Block  $B_5$ : [(10) K:=P\*3]



Navigation icons: back, forward, search, etc.

```

P := 0; I := 1;
REPEAT
  P := P + I;
  IF P > 60 THEN P := 0; I := 5 ENDIF;
  I := I * 2 + 1;
UNTIL I > 20;
K := P * 3
  
```

Tuples:

- P := 0      ← Leader (Rule 1.a)
- I := 1
- P := P + I      ← Leader (Rule 1.b)
- IF P <= 60 GOTO (7)
- P := 0      ← Leader (Rule 1.c)
- I := 5
- T1 := I \* 2      ← Leader (Rule 1.b)

Navigation icons: back, forward, search, etc.

## Summary

Navigation icons: back, forward, search, etc.

- Read the Tiger book:
  - [Instruction selection](#) pp. 205–216
  - [Taming conditional branches](#) pp. 185–188
- Or, 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.

## Summary II

- 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.

## Homework

- Translate the program below into quadruples.
- Identify beginnings and ends of basic blocks.
- Build the control flow graph.

```
PROGRAM P;
VAR X : INTEGER; Y : REAL;
BEGIN
  X := 1; Y := 5.5;
  WHILE X < 10 DO
    Y := Y + FLOAT(X);
    X := X + 1;
    IF Y > 10 THEN
      Y := Y * 2.2;
    ENDIF;
  ENDDO;
END.
```

<pre>int A[5],x,i,n; for (i=1; i&lt;=n; i++) {   if (i&lt;n) {     x = A[i];   } else {     while (x&gt;4) {       x = x*2+A[i];     };   };   x = x+5; }</pre>	<pre>(1) i := 1 (2) IF i&gt;n GOTO (14) (3) IF i&gt;=n GOTO (6) (4) x := A[i] (5) GOTO (11) (6) IF x&lt;=4 GOTO (11) (7) T1 := x*2 (8) T2 := A[i] (9) x := T1+T2 (10) GOTO (6) (11) x := x+5 (12) i := i+1 (13) GOTO (2)</pre>
---	--