Compilers

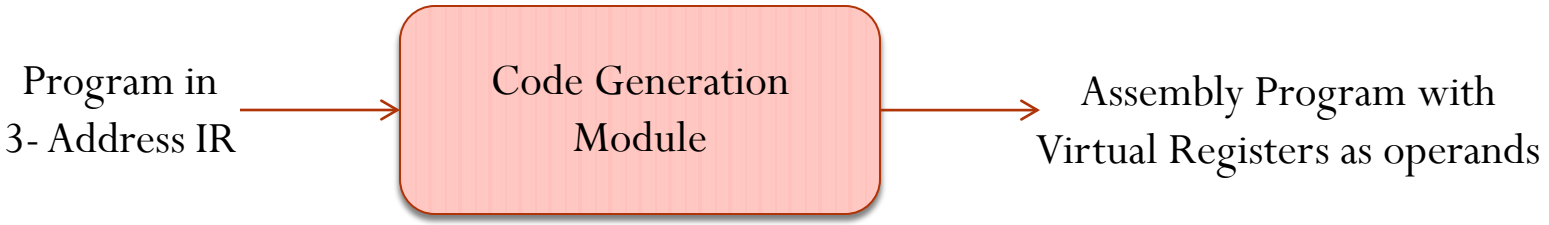
Topic: Simple Code Generation and Local Register Allocation

(ACK: Some of the Slides are based on Keith Cooper's Compilers Course)

Monsoon 2011, IIIT-H, Suresh Purini

Simple Code Generation

- Input: Program in 3-address code representation (and Symbol Table).
- Output: Program in Assembly Mnemonics where the instruction operands are virtual registers, registers, immediate values etc.

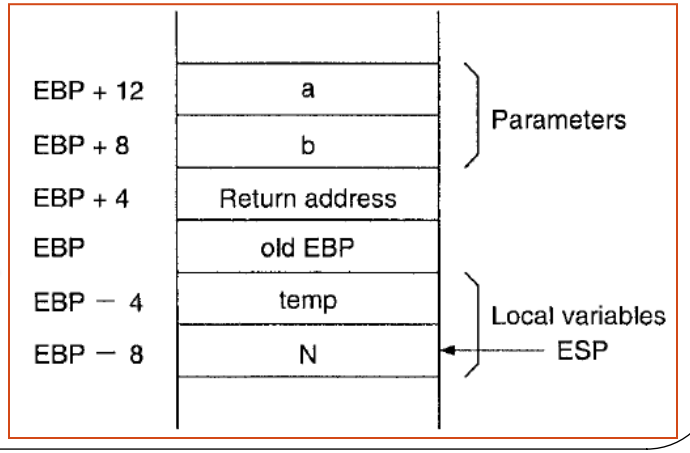


Super Simple Code Generation

- We use NASM syntax (x86/IA-32 is the target architecture).
- Map each 3-address tuple into equivalent assembly language sequence. No Virtual Registers (Memory-to-Memory Model)

3-address Instruction	Assembly Code
x, y and z are local variables and are of type integers	
x = y + z	mov eax, [ebp – 8] ; eax = y add eax, [ebp – 12] ; eax = y + z mov [ebp – 4], eax ; x = eax

Reference Picture



Super Simple Code Generation

3-address Instruction	Assembly Code
x, y and z are global variables and are of type integers	

x = y + z	.data
	x dw 0 ; we can use resw also
	y dw 0
	z dw 0
	.code
	mov eax, [y] ; eax = y
	add eax, [z] ; eax = y + z
	mov [x], eax ; x = eax

- Important Remark: NASM assembler automatically computes the offset of the global variables y, z, x and packs them in the instructions as it translates the assembly to object code.

Super Simple Code Generation

3-address Instruction	Assembly Code
x is local integer and ptr is global (of type int *)	

x = *ptr	mov eax, [ptr] ; eax = ptr
	mov ebx, [eax] ; ebx = *ptr
	mov [ebp-4], ebx ; x = ebx
	===why not the following?====
	mov eax, [ptr] ; eax = ptr
	mov [ebp-4], [eax] ; x = *ptr

x is boolean; (0 and 1 corresponds to true and false resp.)	
IfFalse x goto L	cmp 0, [ebp-8] je L

Super Simple Code Generator

- Super Simple Code Generator Super Under Utilizes the Registers.
- For every 3-address instruction of type x = y + z
  - In RISC Architectures (like MIPS) – 2 loads + 1 store.
  - In CISC architectures (like IA-32): 1 loads + 1 store (however 3 memory accesses!)
- Pro: The code generated is ready to run on the silicon.

Simple Code Generation (Register-to-Register Model)

- Step 1: Construct a CFG for the 3-address representation of the program.
- Step 2: Identify which variables can safely reside in registers.
- Step 3: Make a pass over the CFG generating assembly code assuming that the target machine contains infinitely many registers (we call them virtual registers).
- Step 4: Run a Register Allocator Algorithm.
- Step 5: Generate code with the actual registers in place.

Example

3-address Code	Assembly Code
All variables are 4 byte signed integers and globals. vr1 ↔ a, vr2 ↔ b, vr3 ↔ c, vr4 ↔ d vr5 ↔ e, vr6 ↔ t1, vr7 ↔ t2	

t1 = a + b
t2 = c – d
e = t1 * t2
	mov vr1, [a]
	mov vr2, [b]
	mov vr6, vr2
	add vr6, vr1
	mov vr3, [c]
	mov vr4, [d]
	mov vr7, vr3
	sub vr7, vr4
	mov vr5, vr6
	imul vr5, vr7

Example

3-address Code	Assembly Code
All variables are 4 byte signed integers and globals. vr1 ↔ a, vr2 ↔ ptr, vr3 ↔ t1 (Is the following assembly code correct?)	

ptr = &a
a = a + 1
t1 = *ptr
t1 = t1 + 1
*ptr = t1
	mov vr2, a
	mov vr1, [a]
	inc vr1
	mov vr3, [vr2]
	inc vr3
	mov [vr2], vr3

- Don't promote variable a to a register?

Example

3-address Code	Assembly Code
All variables are 4 byte signed integers and globals. Memory ↔ a, vr2 ↔ ptr, vr3 ↔ t1 (Is the Code Correct Now?)	

ptr = &a
a = a + 1
t1 = *ptr
t1 = t1 + 1
*ptr = t1
	mov vr2, a
	inc [a]
	mov vr3, [vr2]
	inc vr3
	mov [vr2], vr3

Example

Key Idea

- ptr points to either x or y only.
- At compile time we have no clue as to what ptr points to.
- Be conservative and force both x and y to reside in memory only.
- Possible Solution: Dynamic Compilation (On the Fly Register Allocation at Run-Time)

```
x = y + z;
if ( x > 0 ) {
   ptr = &y ;
} else {
    ptr = &z ;
}
x  = *ptr + 1;
```

Register Safe Values

- When can a variable or a compiler temporary be safely allocated to a register?
  - When only 1 name can reference its value (no aliases)
  - Pointers, Call-by-Reference parameters, aggregates & arrays all cause trouble
  - Variables with global scope
    - Procedure calls could effect their values.
- When should a value be allocated to a register?
  - When it is both safe & profitable

Register Safe Values

Example 1:

```
p = &x
x = x + 1
*p = *p + 1
```

Few Simple Rules:

1. Never allocate a register for a variable whose address is taken.
2. Never allocate a register for an array variable.

- Too conservative rule but works.
- Pointer Arithmetic is a Night Mare for Register allocator.

Example 2:

```
x = x + 1
temp = call foo   // foo can have side effects
x = 2 * x
```

Example 3:

```
a[i] = a[i] + 1
a[j] = a[j] + 1
```

Code Generation for Infinite Register Machine

Consider x = y + z:

X	Y	Z	RISC Code	CISC Code
VRi	VR_j	VR_k	ADD VR_i, VR_j,VR_k	ADD VR_i, VR_j,VR_k
VR_i	mem(local)	VR_k	LD Rt, @y($bp) ADD VR_i, R_t, VR_k	ADD VR_i, VR_k, @y($bp)
VR_i	mem(global)	VR_k	LD Rt, @y($gp) ADD VR_i, R_t, VR_k	ADD VR_i, VR_k, @y($gp)
mem(local)	VR_i	VR_k	ADD R_t, VR_j, VR_k ST @x($bp), R_t	ADD @x($sp), VR_j, VR_k

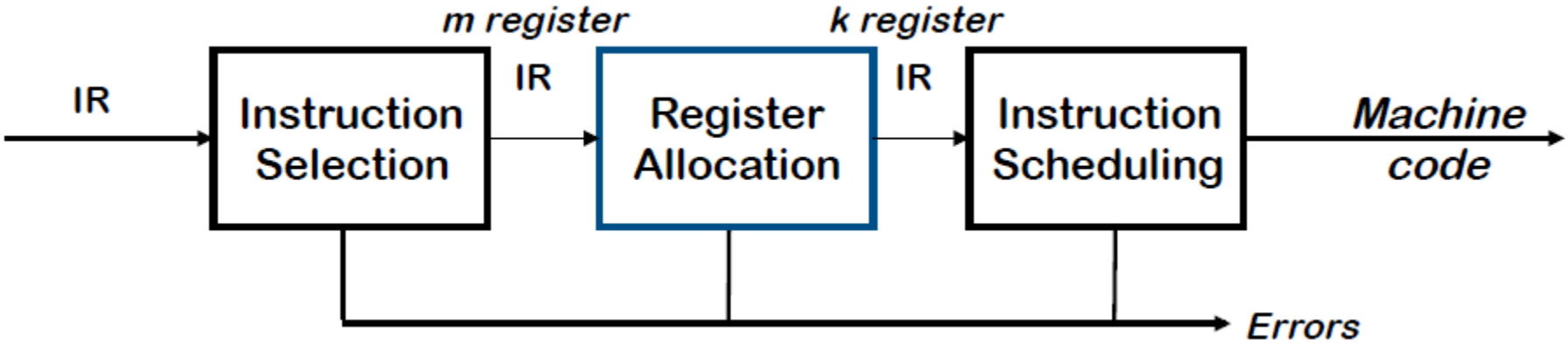
mem(local)	mem(local)	mem(local)	LD R_t, @y($bp)
			LD R_u, @z($bp)
			ADD R_s, R_t, R_u
			ST @x($sp), R_s
				LD R_t, @y($bp)
				ADD R_s, R_t, @z($bp)
				ST @x($sp), R_s

More cases follow…….

Key Remark: Physical Registers sp, bp, R_t and possible any other registers used in code generation are not available during the register allocation phase.

Register Allocation Problem

Part of the Compilers Back End



Critical properties

- Produce correct code that uses no more than k registers
- Minimize added work from loads and stores that spill values
- Minimize space used to hold spilled values
- Operate efficiently
  - O(n), O(n logn), maybe $O(n^2)$, but not $O(2^n)$

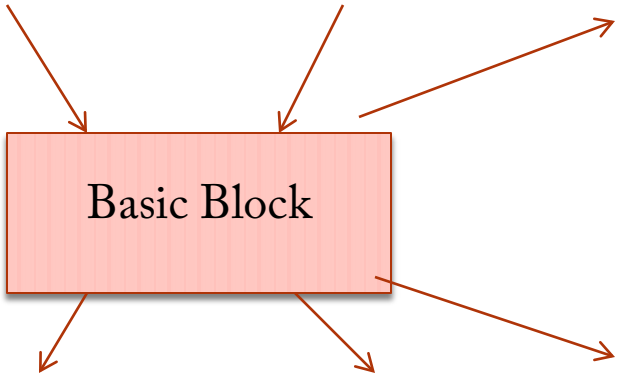
Register Allocation for Straight Line Code

1. Out of the K machine registers, reserve F machine registers.
   - F contains
     - Special purpose registers (like SP, BP)
     - Registers to handle computations involving variables which cannot be promoted to registers
     - Registers to handle spill code
2. Top-Down Register Allocator
   1. Assign priority to the Virtual Registers based on their usage frequency.
   2. Assign Physical Registers to the top K-F Virtual Registers.
   3. Rewrite the generated code to reflect changes (typically have to add spill code)

(Top-Down Register Allocator Example) u = z * (x – y) – 20 * (x + y) + x + y			
3-address code	Code with Virtual Registers x ↔ vr1, y ↔ vr2, t1 ↔ vr3, z ↔ vr4, t2 ↔ vr5, t3 ↔ vr6, 20 ↔ vr7, t4 ↔ vr8, t5 ↔ vr9, t6 ↔ vr10, u ↔ vr11,	Code Assuming 4 available Register (|K – F| = 4) What is K ?	Virtual Register Occurrence Frequency
t1 = x - y	load vr1, @x($fp) load vr2, @y($fp) sub vr3, vr1, vr2	load r1, @x($fp) load r2, @y($fp) sub r3, r1, r2	VR1 – 4 VR2 – 4 VR3 – 2 VR4 – 2 VR5 – 2 VR6 – 2 VR7 – 2 VR8 – 2 VR9 – 2 VR10 – 2 VR1 1– 1 Register Assignment x ↔ vr1 ↔ r1 y ↔ vr2 ↔ r2 t1 ↔ vr3 ↔ r3 z ↔ vr4 ↔ r4 Spill Code r5 and r6 are reserved to handle spill code
t2 = z * t1	load vr4, @z($fp) mul vr5, vr4, vr3	load r4, @z($fp) mul r5, r4, r3 store @t2($fp), r5	
t3 = x + y	add vr6, vr1, vr2	add r5, r1, r2 store @t3($fp), r5	
t4 = 20 * t3	loadi vr7, 20 mul vr8, vr7, vr6	loadi r5, 20 load r6, @t3($fp) mul r5, r5, r6 store @t4($fp), r5	
t5 = t2 – t4	sub vr9, vr5, vr8	load r5, @t2($fp) load r6, @t4($fp) sub r5, r5, r6 store @t5($fp), r5	
t6 = t5 + x	add vr10, vr9, vr1	load r5, @t5($fp) add r5, r5, r1 store @t6($fp), r5	
u = t6 + y	add vr11, vr10, vr2	load r5, @t6($fp) add r5, r5, r2 store @u($fp), r5	

Top-Down Register Allocator

- What if the code is not straight line (contains branches) ?
- Question: How to use the Top-Down Register Allocator for programs with branch instructions?
- Key Idea: Construct a CFG of the program and apply a Top-Down Register Allocation Algorithm over Basic Blocks



At the beginning of a basic block we assume the latest values of variables are reflected in their main memory locations.

Update the main memory locations of variables promoted to registers

Top Down Register Allocator

- Question: Does a Top-Down Register Allocator gives optimal Register Allocation? (let's say with respect to the main memory traffic generated)
- Problem: What happens if a Virtual Register is heavily used in the first half but not at all used in the second half of a Basic Block?
- How can we solve this problem?

Bottom-up Register Allocator

```
/* code for the allocator */
for each operation, i, in order from 1
    to N where i has the form
        op vr_i1 vr_i2 ⇒ vr_i3
    r_x ← ensure(vr_i1, class(vr_i1))
    r_y ← ensure(vr_i2, class(vr_i2))
    if vr_i1 is not needed after i
      then free(r_x,class(r_x))
    if vr_i2 is not needed after i
      then free(r_y,class(r_y))
    r_z ←allocate(vr_i3, class(vr_i3))
    rewrite i as op_i r_x, r_y ⇒ r_z
    if vr_i1 is needed after i
      then class.Next[r_x] ← dist(vr_i1)
    if vr_i2 is needed after i
      then class.Next[r_y] ← dist(vr_i2)
    class.Next[r_z] ← dist(vr_i3)

free(i,class)
    if (class.Free[i] ≠ true) then
      push(i,class)
      class.Name[i] ← −1
      class.Next[i] ← ∞
      class.Free[i] ← true

ensure(vr,class)
  if (vr is already in class)
    then result ← vr's physical register

  else
    result ← allocate(vr,class)
    emit code to move vr into result
  return result

allocate(vr,class)
  if (class.StackTop ≥ 0)
    then i ← pop(class)
  else
    i ← j that maximizes class.Next[j]
    store contents of j
  class.Name[i] ← vr
  class.Next[i] ← −1
  class.Free[i] ← false
  return i
```

```
struct Class {
  int Size;
  int Name[Size];
  int Next[Size];
  int Free[Size];
  int Stack[Size];
  int StackTop;
}
```

```
initialize(class,size)
  class.Size ← size
  class.StackTop ← −1
  for i ← 0 to size −1
      class.Name[i] ← −1
      class.Next[i] ← ∞
      class.Free[i] ← true
      push(i,class)
```

Local Graph Coloring Register Allocator

Techniques to improve Register Allocators

- Allocate Registers for Values rather than variables
- Use Live Ranges
- Construct an Interference Graph
  - Nodes correspond to Live Ranges
  - An edge between two nodes if the corresponding live ranges interfere.
- Do a k-coloring of the graph
- We may have to generate spill code to handle live ranges that are not assigned registers.
- Question: How to minimize the Spill Cost?

Live Ranges Example

3 – Address Code for a Basic Block	Rewritten 3-Address Code	$x_0 \leftrightarrow$ vr1, $x_1 \leftrightarrow$ vr2, $z_0 \leftrightarrow$ vr3, $t1_1 \leftrightarrow$ vr4, $t2_1 \leftrightarrow$ vr5, $y_1 \leftrightarrow$ vr6, $t3_1 \leftrightarrow$ vr7, $t4_1 \leftrightarrow$ vr8, $x_2 \leftrightarrow$ vr9	Live Range
$x = x + z$	$x_1 = x_0 + z_0$	1. vr2 = vr1 + vr3	vr1 – [Start, 1] vr2 – [1, 2] vr3 – [Start, 1] vr4 – [2, 3] vr5 – [4, 5] vr6 – [3, 4] vr7 – [6, 7] vr8 – [7, end] vr9 – [5, 7] Important: Remember to update memory locations at the end of a BB.
$t1 = 2 * x$	$t1_1 = 2 * x_1$	2. vr4 = 2 * vr2	
$y = t1 + 1$	$y_1 = t1_1 + 1$	3. vr6 = vr4 + 1	
$t2 = y + 4$	$t2_1 = y_1 + 4$	4. vr5 = vr6 + 4	
$x = t2 + 2$	$x_2 = t2_1 + 2$	5. vr9 = vr5 + 2	
$t3 = x + 1$	$t3_1 = x_2 + 1$	6. vr7 = vr9 + 1	
$t4 = t3 + x$	$t4_1 = t3_1 + x_2$	7. vr8 = vr7 + vr9	