Machine Code Generation - 2

Y. N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Mach. code generation – main issues (in part 1)
- Samples of generated code
- Two Simple code generators
- Optimal code generation
  - Sethi-Ullman algorithm
  - Dynamic programming based algorithm
  - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations

Samples of Generated Code – Static Allocation (no JSR instruction)

Three Adress Code

// Code for function F1 action code seg 1 call F2 action code seg 2 Halt
// Code for function F2 action code seg 3 return

Activation Record for F1 (48 bytes)

0	return address
4	data array A
40	variable x
44	variable y

Activation Record for F2 (76 bytes)

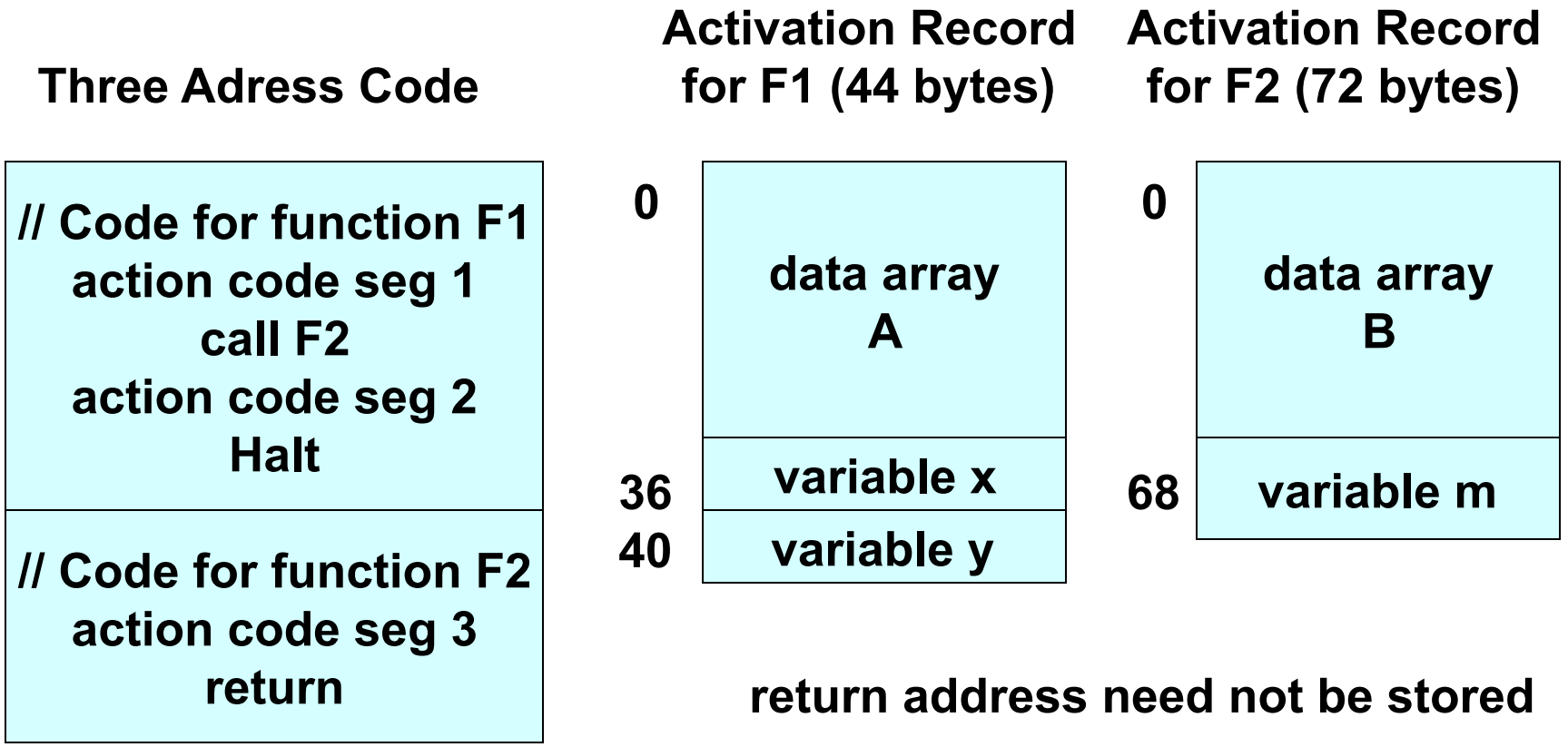
0	return address
4	parameter 1
	data array B
72	variable m

Samples of Generated Code – Static Allocation (no JSR instruction)

```
// Code for function F1
200:    Action code seg 1
// Now store return address
240:    Move #264, 648
252:    Move val1, 652
256:    Jump 400 // Call F2
264:    Action code seg 2
280:    Halt
        ...
// Code for function F2
400:    Action code seg 3
// Now return to F1
440:    Jump @648
        ...
```

```
//Activation record for F1
//from 600-647
600:    //return address
604:    //space for array A
640:    //space for variable x
644:    //space for variable y
//Activation record for F2
//from 648-723
648:    //return address
652:    // parameter 1
656:    //space for array B
        ...
720:    //space for variable m
```

Samples of Generated Code – Static Allocation (with JSR instruction)



Samples of Generated Code – Static Allocation (with JSR instruction)

```
// Code for function F1
200:    Action code seg 1
// Now jump to F2, return addr
// is stored on hardware stack
240:    JSR 400 // Call F2
248:    Action code seg 2
268:    Halt
        ...
// Code for function F2
400:    Action code seg 3
// Now return to F1 (addr 248)
440:    return
        ...
```

```
//Activation record for F1
//from 600-643
600:    //space for array A
636:    //space for variable x
640:    //space for variable y
//Activation record for F2
//from 644-715
644:    //space for array B
        ...
712:    //space for variable m
```

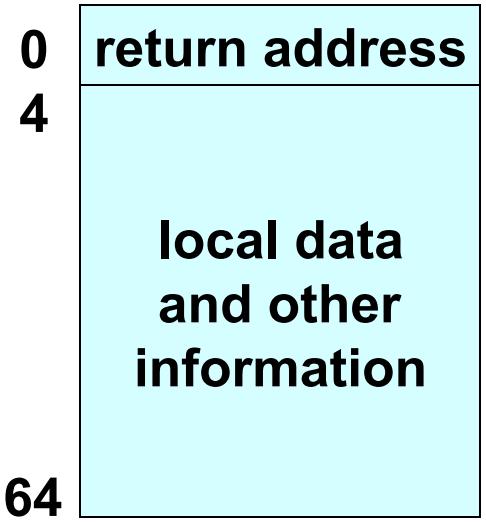
Samples of Generated Code – Dynamic Allocation (no JSR instruction)

Three Adress Code

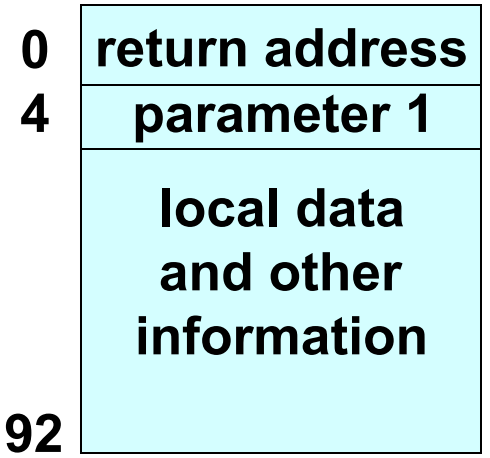
```
// Code for function F1
action code seg 1
call F2
action code seg 2
return

// Code for function F2
action code seg 3
call F1
action code seg 4
call F2
action code seg 5
return
```

Activation Record for F1 (68 bytes)



Activation Record for F2 (96 bytes)



Samples of Generated Code – Dynamic Allocation (no JSR instruction)

```
//Initialization
100:    Move #800, SP
        ...
//Code for F1
200:    Action code seg 1
230:    Add #96, SP
238:    Move #258, @SP
246:    Move val1, @SP+4
250:    Jump 300
258:    Sub #96, SP
266:    Action code seg 2
296:    Jump   @SP
```

```
//Code for F2
300:    Action code seg 3
340:    Add #68, SP
348:    Move #364, @SP
356:    Jump 200
364:    Sub #68, SP
372:    Action code seg 4
400:    Add #96, SP
408:    Move #424, @SP
416:    Move val2, @SP+4
420:    Jump 300
428:    Sub #96, SP
436:    Action code seg 5
480:    Jump @SP
```

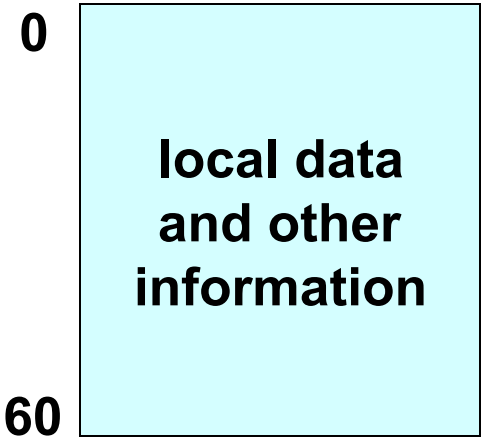
Samples of Generated Code – Dynamic Allocation (with JSR instruction)

Three Adress Code

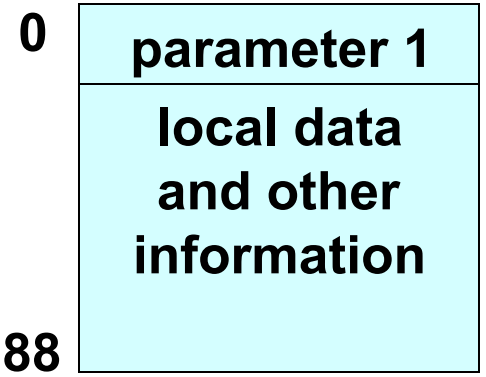
```
// Code for function F1
action code seg 1
call F2
action code seg 2
return

// Code for function F2
action code seg 3
call F1
action code seg 4
call F2
action code seg 5
return
```

Activation Record for F1 (64 bytes)



Activation Record for F2 (92 bytes)



Samples of Generated Code – Dynamic Allocation (with JSR instruction)

```
//Initialization
100:    Move #800, SP
        ...
//Code for F1
200:    Action code seg 1
230:    Add #92, SP
238:    Move val1, @SP
242:    JSR 290
250:    Sub #92, SP
258:    Action code seg 2
286:    return
```

```
//Code for F2
290:    Action code seg 3
330:    Add #64, SP
338:    JSR 200
346:    Sub #64, SP
354:    Action code seg 4
382:    Add #92, SP
390:    Move val2, @SP
394:    JSR 290
402:    Sub #92, SP
410:    Action code seg 5
454:    return
```

A Simple Code Generator – Scheme A

- Treat each quadruple as a ‘macro’
  - Example: The quad *A := B + C* will result in

Load B, R1	**OR**	**Load B, R1**
Load C, R2		
Add R2, R1		**Add C, R1**
Store R1, A		**Store R1, A**

  - Results in inefficient code
    - Repeated load/store of registers
  - Very simple to implement

A Simple Code Generator – Scheme B

- Track values in registers and reuse them
  - If any operand is already in a register, take advantage of it
  - Register descriptors
    - Tracks <register, variable name> pairs
    - A single register can contain values of multiple names, if they are all copies
  - Address descriptors
    - Tracks <variable name, location> pairs
    - A single name may have its value in multiple locations, such as, memory, register, and stack

A Simple Code Generator – Scheme B

- Leave computed result in a register as long as possible
- Store only at the end of a basic block or when that register is needed for another computation
  - A variable is live at a point, if it is used later, possibly in other blocks – obtained by dataflow analysis
  - On exit from a basic block, store only live variables which are not in their memory locations already (use address descriptors to determine the latter)
  - If liveness information is not known, assume that all variables are live at all times

Example

- A := B+C
  - If B and C are in registers R1 and R2, then generate
    - *ADD R2,R1* (cost = 1, result in R1)
      - legal only if B is *not live* after the statement
  - If R1 contains B, but C is in memory
    - *ADD C,R1* (cost = 2, result in R1) **or**
    - *LOAD C, R2*

      ADD R2,R1 (cost = 3, result in R1)
      - legal only if B is *not live* after the statement
      - attractive if the value of C is subsequently used (it can be taken from R2)

Next Use Information

- Next use info is used in code generation and register allocation
- Next use of *A* in quad *i* is *j* if

Quad *i* : A = ... (assignment to A)

(control flows from *i* to *j* with no assignments to A)

Quad *j* : = A op B (usage of A)

- In computing next use, we assume that on exit from the basic block
  - All temporaries are considered non-live
  - All programmer defined variables (and non-temps) are live
- Each procedure/function call is assumed to start a basic block
- Next use is computed on a backward scan on the quads in a basic block, starting from the end
- Next use information is stored in the symbol table

Example of computing Next Use

3	T1 := 4 * I	T1 – (nlv, lu 0, nu 5), I – (lv, lu 3, nu 10)
4	T2 := addr(A) – 4	T2 – (nlv, lu 0, nu 5)
5	T3 := T2[T1]	T3 – (nlv, lu 0, nu 8), T2 – (nlv, lu 5, nnu), T1 – (nlv, lu 5, nu 7)
6	T4 := addr(B) – 4	T4 – (nlv, lu 0, nu 7)
7	T5 := T4[T1]	T5 – (nlv, lu 0, nu 8), T4 – (nlv, lu 7, nnu), T1 – (nlv, lu 7, nnu)
8	T6 := T3 * T5	T6 – (nlv, lu 0, nu 9),T3 – (nlv, lu 8, nnu), T5 – (nlv, lu 8, nnu)
9	PROD := PROD + T6	PROD – (lv, lu 9, nnu), T6 – (nlv, lu 9, nnu)
10	I := I + 1	I – (lv, lu 10, nu 11)
11	if I ≤ 20 goto 3	I – (lv, lu 11, nnu)

Scheme B – The algorithm

- We deal with one basic block at a time
- We assume that there is no global register allocation
- For each quad *A := B op C* do the following
  - Find a location *L* to perform *B op C*
    - Usually a register returned by *GETREG*() (could be a mem loc)
  - Where is *B*?
    - *B'* , found using address descriptor for *B*
    - Prefer register for *B'* , if it is available in memory and register
    - Generate *Load B' , L* (if *B'* is not in *L*)
  - Where is *C*?
    - *C'* , found using address descriptor for *C*
    - Generate *op C' , L*
  - Update descriptors for *L* and *A*
  - If *B/C* have no next uses, update descriptors to reflect this information

Function *GETREG( )*

Finds *L* for computing *A := B op C*

1. If *B* is in a register (say *R*), R holds no other names, and
   - *B* has no next use, and *B* is not live after the block, then return *R*
2. Failing (1), return an empty register, if available
3. Failing (2)
   - If *A* has a next use in the block, OR
     if *B op C* needs a register (*e.g., op* is an indexing operator)
     - Use a *heuristic* to find an occupied register
       - **a register whose contents are referenced farthest in future, or**
       - **the number of next uses is smallest etc.**
     - *Spill* it by generating an instruction, *MOV R, mem*
       - ***mem* is the memory location for the variable in *R***
       - **That variable is not already in *mem***
     - Update *Register* and *Address* descriptors
4. If *A* is not used in the block, or no suitable register can be found
   - Return a memory location for *L*

Example

T,U, and V are temporaries - not live at the end of the block
W is a non-temporary - live at the end of the block, 2 registers

Statements	Code Generated	Register Descriptor	Address Descriptor
T := A * B	Load A,R0 Mult B, R0	R0 contains T	T in R0
U := A + C	Load A, R1 Add C, R1	R0 contains T R1 contains U	T in R0 U in R1
V := T - U	Sub R1, R0	R0 contains V R1 contains U	U in R1 V in R0
W := V * U	Mult R1, R0	R0 contains W	W in R0
	Store R0, W		W in memory (restored)

Optimal Code Generation - The Sethi-Ullman Algorithm

- Generates the shortest sequence of instructions
  - Provably optimal algorithm (w.r.t. length of the sequence)
- Suitable for expression trees (basic block level)
- Machine model
  - All computations are carried out in registers
  - Instructions are of the form *op R,R* or *op M,R*
- **Always computes the left subtree into a register and reuses it immediately**
- Two phases
  - Labelling phase
  - Code generation phase

The Labelling Algorithm

- Labels each node of the tree with an integer:
  - fewest no. of registers required to evaluate the tree with no intermediate stores to memory
  - Consider binary trees
- For leaf nodes
  - ***if*** **n** is the leftmost child of its parent ***then***

    label(n) := 1 *else* label(n) := 0
- For internal nodes
  - **label(n) = max (l_1, l_2), if $l_1 <> l_2$**

    = l_1 + 1, if $l_1 = l_2$

Labelling - Example