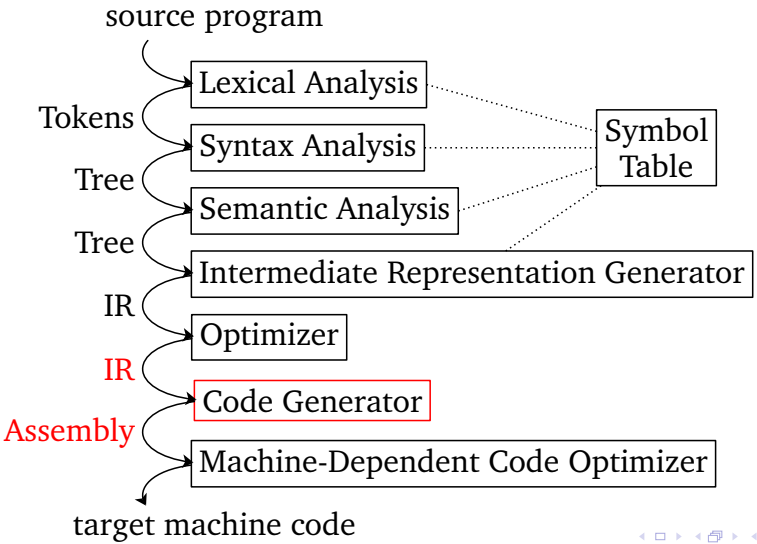
Code Generation

Kristoffer H. Rose Eva Rose

NYU Courant Institute

April 14, 2014

Sixth compilation phase



What's Important?

- *Correctness*.
- Speed of target program.
- Speed of code generator.

Target Architectures

- CISC.
- RISC.
- Stack.

Only sequential architectures considered!

Target Architectures

- CISC.
- RISC.
- Stack.

Only sequential architectures considered!

Instruction Selection

- How closely does the IR match the target language?
- How good should the code be?
- What kind of architecture are we considering?

Example

```
a = b + c
d = a + e
```

```
LDR  r0,[sp,&B]
ADD  r0,r0,[sp,&C]
STR  r0,[sp,&A]
LDR  r0,[sp,&A]
ADD  r0,r0,[sp,&E]
STR  r0,[sp,&D]
```

Example

```
a = b + c
d = a + e

LDR  r0,[sp,&B]
ADD  r0,r0,[sp,&C]
STR  r0,[sp,&A]
LDR  r0,[sp,&A]
ADD  r0,r0,[sp,&E]
STR  r0,[sp,&D]
```

Example

```
a = b + c
d = a + e

LDR  r0,[sp,&B]     ;sp-relative OK when no local stack
ADD  r0,r0,[sp,&C]
STR  r0,[sp,&A]
LDR  r0,[sp,&A]
ADD  r0,r0,[sp,&E]
STR  r0,[sp,&D]
```

Register Allocation

- Hard!
- Operating system conventions.
- Hardware restrictions:
  - Special purpose registers (stack, index, frame, *etc.*)
  - Register pairs.
- Allocation—which variables are in registers?
- Assignment—which specific registers are used?

Register Allocation

- Hard!
- Operating system conventions.
- Hardware restrictions:
  - Special purpose registers (stack, index, frame, *etc.*)
  - Register pairs.
- Allocation—which variables are in registers?
- Assignment—which specific registers are used?

ARM Machine Model

- Register machine with 13 general-purpose registers + sp/lr/pc.
- Load and store to byte-addressable memory.
- Computations.
- Jumps with optional return address save mechanism.
- Most operations may test and/or set condition codes.

Instruction (Sub)Set

```
MOV reg,arg             arg = #imm8 | reg | reg,LSL #imm5 | reg,LSR #imm5

ADD reg,reg1,arg2       reg = R0 | R1 | ... | R15 | SP | LR | PC
SUB reg,reg1,arg2
MUL reg,reg1,arg2       immn = n-bit constant

CMPS reg,arg

B   imm12
Bcd imm12               cd = EQ | NE | MI | PL | GE | LT | GT | LE
BL  imm12
                        b = B |
LDRb reg,mem
STRb reg,mem            mem = [reg,arg]
LDMFD reg!,mreg
STMFD reg!,mreg         mreg = { reg,...,reg }  (regs must be in order)
```

Load and Store

LDR r, mem Load: $r := *mem$.

STR r, mem Store: $*mem := r$.

where *mem* is one of

mem	Meaning
$[r, \&x]$	variable x in r frame
$[r, \pm r']$	char at $r \pm r'$
$[r, \pm r', shift]$	r'th member of array at r
$[r, \&x]!$	Same, with update

Load and Store

LDR r, mem Load: $r := *mem$.

STR r, mem Store: $*mem := r$.

where *mem* is one of

mem	Meaning	C
$[r, \&x]$	variable x in r frame	`x`
$[r, \pm r']$	char at $r \pm r'$	
$[r, \pm r', shift]$	r'th member of array at r	`r[r']`
$[r, \&x]!$	Same, with update	r[r'++]

Instructions

$OP\ dst, reg_1, arg_2$	Operation: $dst := reg_1\ OP\ arg_2$.
$OP\ dst, reg$	Operation: $dst := OP(reg)$.
$\texttt{B}\ L$	Jump: "goto" L.
$\texttt{CMPS}\ reg_1, arg_2$	Compute $reg_1 - arg_2$, set *cond*.
$\texttt{B}cond\ L$	Conditional jump: "if" *cond* "goto" L; $cond \in \{\texttt{MI}, \texttt{PL}, \texttt{LT}, \texttt{GT}, \dots\}$.

$x = y - z$

```
LDR  r1,[r11,&y]   // R1 = y
LDR  r2,[r11,&z]   // R2 = z
SUB  r1,r1,r2      // R1 = R1 - R2
STR  r1,[r11,&x]   // x = R1
```

$b = a[i]$

```
LDR r4,[r11,&i]         // r4 = i
ADD r5,r11,&a           // r5 = &a
LDR r6,[r5,r4,LSL#2]    // r6 = *(a[i])
STR r6,[r11,&b]         // b = R2
```

$a[j] = c$

```
1 LDR r4,[r11,&j]        // r4 = j
2 ADD r5,r11,&a          // r5 = &a
3 LDR r6,[r11,&c]        // r6 = c
4 STR r6,[r5,r4,LSL#2]   // *(a[i]) = c
```

$*x = *y$

```
1  LDR r4,[r11,&y]     // r4 = &y
2  LDR r5,[r4,#0]      // r5 = *y
3  LDR r6,[r11,&x]     // r6 = &x
4  STR r5,[r6,#0]      // *x = *y
```

if $x < y$ goto L

```
1  LDR r4,[r11,&x]     // r4 = x
2  CMPS r4,[r11,&y]    // set flags from (x-y)
3  BLT  L              // if x<y goto L
```

Cost?

- Instruction size.
- Instruction memory accesses.
- Instruction execution time.

Segments

Code	where the code is stored
Static	for constants that do not fit in instructions
Stack	for activation records with scoped values
Heap	for dynamically allocated and freed data objects

Calls

param 1;
param "Bar";
i = *call foo*;

...

function *foo*(a : *int*, b : *string*) : *int* =
return a;

Static Allocation

```
  LD R1,#1  //a
  ST foo.staticArea, R1
  LD R1,#Bar.address  //b
  ST foo.staticArea+8, R1
  ST foo.staticArea+16, #ReturnHere
  BR FOO
ReturnHere:
  LD R1,foo.staticArea+24
...

FOO:
  LD R1,foo.staticArea
  ST foo.staticArea+24,R1
  BR *(foo.staticArea+16)
```

Stack Allocation

```
  ADD SP,SP,#my.recordSize+foo.paramsize
  LD R1,#1  //a
  ST -16(SP) R1
  LD R1,#Bar.address  //b
  ST -8(SP), R1
  ST  0(SP), #ReturnHere
  BR FOO
ReturnHere:
  LD  R1,-16(SP)
  SUB SP,SP,#my.recordSize

FOO:
  LD R1,-16(SP) // a
  ST -16(SP),R1
  BR *0(SP)
```

Idea

1. Basic Blocks are maximal sequences of consecutive three-address instructions,
   1. the control flow can only enter through first instruction,
   2. all branch instructions leave the block
2. Flow Graph is the the graph with basic blocks as nodes and branches as directed edges

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto 3
10) i = i + 1
11) if i <= 10 goto 2
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto 13
```

Reconstruct Dragon Book Figure 8.9 on the blackboard from this.

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto 3
10) i = i + 1
11) if i <= 10 goto 2
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto 13
```

Reconstruct Dragon Book Figure 8.9 on the blackboard from this.

Next-Use

How many registers are needed inside a basic block?

Idea

Now generate the instructions but. . .

- *Maintain mapping between variables and registe.*
- Avoid loading values already in a register.
- Store only as needed.
- Reuse registers that have no next use.

Let's translate

```
1 t = a - b
2 u = a - c
3 v = t + u
4 a = d
5 d = v + u
```

Reconstruct Dragon Book Figure 8.16 from this.

Let's translate

```
1 t = a - b
2 u = a - c
3 v = t + u
4 a = d
5 d = v + u
```

Reconstruct Dragon Book Figure 8.16 from this.

Register Allocation: *getReg*

We have value V how do we get it in a register?

1. If we got it all is already well!
2. If we have an empty register then use that.
3. If there is no free register we have to make one—
   1. If we have a redundant one, use that;
   2. If we know our instruction will destroy a register, use it;
   3. If we have no next-use then it is as free;
   4. Otherwise we have to spill.

Questions?

krisrose@cs.nyu.edu