9. Code Generation

Eva Rose       Kristoffer Rose

NYU Courant Institute
Compiler Construction (CSCI-GA.2130-001)

April 13, 2015
1. Issues
2. Target Language: ARM32
3. Basic Blocks & Flow Graphs
4. Project Milestone 3
Sixth compilation phase
1 Issues

2 Target Language: ARM32

3 Basic Blocks & Flow Graphs

4 Project Milestone 3
What’s Important?

- Correctness.
- Speed of target program.
- Speed of code generator.
Target Architectures

- CISC.
- RISC.
- Stack.

Only sequential architectures considered in class—but see Dragon Book chapter 11...
Target Architectures

- CISC.
- RISC.
- Stack.

Only sequential architectures considered in class—but see Dragon Book chapter 11…
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**

\[
\begin{align*}
    a & = b + c \\
    d & = a + e
\end{align*}
\]

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]
a = b + c

Example

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

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

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

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

2. Target Language: ARM32

3. Basic Blocks & Flow Graphs

4. Project Milestone 3
ARM32 Machine Model

- Register machine with 13 general-purpose registers Rn + 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.
# ARM32 Register Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0–1</td>
<td>general purpose</td>
<td>parameter &amp; return value</td>
</tr>
<tr>
<td>R2–3</td>
<td>general purpose</td>
<td>parameter</td>
</tr>
<tr>
<td>R4–11</td>
<td>general purpose</td>
<td>preserved</td>
</tr>
<tr>
<td>R12</td>
<td>frame pointer</td>
<td>–</td>
</tr>
<tr>
<td>SP (R13)</td>
<td>stack pointer</td>
<td>address of lowest stack entry</td>
</tr>
<tr>
<td>LR (R14)</td>
<td>link register</td>
<td>–</td>
</tr>
<tr>
<td>PC (R15)</td>
<td>instruction address + 8</td>
<td>–</td>
</tr>
</tbody>
</table>
MinARM32 Memory Model

Static

Code

Static

Heap

Free Memory

Dynamic

↓

↑

Stack

lowest address

highest address
Assembler

- Allocate memory image (Code and Static) word by word.
- **OpCodes** allocate words for machine instructions.
- **Directives** allocate non-instruction words and placement.
- Maintain **labels** for relative locations in code.
### MinARM32 Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell$</td>
<td>the label $\ell$ is set to the next address in memory</td>
</tr>
<tr>
<td>$\ell = n$</td>
<td>the label $\ell$ is set to the value of the integer $n$ – makes $&amp; \ell = # n$</td>
</tr>
<tr>
<td>DCI $n_1, \ldots, n_k$</td>
<td>store the integers $n_1, \ldots, n_k$ ($k \geq 1$) into consecutive words</td>
</tr>
<tr>
<td>$op$</td>
<td>stores the word encoding of the instruction $op$</td>
</tr>
</tbody>
</table>
## MinARM32 Data Processing Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV $r_d, arg$</td>
<td>$r_d := arg$</td>
<td></td>
</tr>
<tr>
<td>MVN $r_d, arg$</td>
<td>$r_d := \sim arg$</td>
<td>bitwise not</td>
</tr>
<tr>
<td>ADD $r_d, r_1, arg$</td>
<td>$r_d := r_1 + arg$</td>
<td></td>
</tr>
<tr>
<td>SUB $r_d, r_1, arg$</td>
<td>$r_d := r_1 - arg$</td>
<td></td>
</tr>
<tr>
<td>RSB $r_d, r_1, arg$</td>
<td>$r_d := arg - r_1$</td>
<td></td>
</tr>
<tr>
<td>AND $r_d, r_1, arg$</td>
<td>$r_d := r_1 &amp; arg$</td>
<td>bitwise and</td>
</tr>
<tr>
<td>ORR $r_d, r_1, arg$</td>
<td>$r_d := r_1</td>
<td>arg$</td>
</tr>
<tr>
<td>EOR $r_d, r_1, arg$</td>
<td>$r_d := r_1 \hat{&amp;} arg$</td>
<td>bitwise exclusive or</td>
</tr>
<tr>
<td>MUL $r_d, r_1, r_2$</td>
<td>$r_d := r_1 \times r_2$</td>
<td></td>
</tr>
</tbody>
</table>
MinARM32 Data Processing Argument Forms

- $r$ refers to any of the registers in the previous section.
- $arg$ refers to one of these value forms:
  - #$n$ – the “immediate” value $n$ ($0 \leq n \leq 255$).
  - $r$ – the value in the indicated register.
  - $r$, LSL #$n$ – the value in the indicated register shifted left by $n$ bits ($0 \leq n \leq 31$).
  - $r$, LSR #$n$ – the value in the indicated register shifted right by $n$ bits ($0 \leq n < 31$).
## MinARM32 Load and Store

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR $r, \text{mem}$</td>
<td>$r := \ast \text{mem}$</td>
<td></td>
</tr>
<tr>
<td>STR $r, \text{mem}$</td>
<td>$\ast \text{mem} := r$</td>
<td>data moves from left to right</td>
</tr>
</tbody>
</table>

- $[r, \#n]$ – address is $r + n$ for $-4096 \leq n \leq 4095$.
- $[r, \pm r']$ – address is $r \pm r'$ with $\pm$ meaning $+$ or $-$.
- $[r, \pm r', \text{LSL} \#n]$ – address is $r \pm (r' \times 2^n)$ for $1 \leq n \leq 31$. 
MinARM32 Load and Store Multiple

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDMFD r!,{mreg}</td>
<td>pop all registers in mreg from stack with r as stack pointer</td>
</tr>
<tr>
<td>STMFD r!,{mreg}</td>
<td>push all registers in mreg onto stack with r as stack pointer</td>
</tr>
</tbody>
</table>

▶ mreg stands for a set of registers separated by commas.
## ARM32 Comparisons

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP $r_1$, arg</td>
<td>$\text{cond} := r_1 \ ? \ arg$</td>
<td>Sets condition bits (See ARM Reference p. A3-4). The actual ? comparison to use is decided later.</td>
</tr>
</tbody>
</table>
## ARM32 Comparisons

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Notes</th>
</tr>
</thead>
</table>
| CMP $r_1$, arg | $cond := r_1 \text{ ? } arg$ | - Sets condition bits (See ARM Reference p. A3-4).  
- The actual ? comparison to use is decided later. |
## MinARM32 Branch and Conditional Branch

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>B ( \ell )</td>
<td>( \text{PC} := \ell )</td>
<td></td>
</tr>
<tr>
<td>BEQ ( \ell )</td>
<td>if ( \text{cond}(=) ) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with ( = ) for ?</td>
</tr>
<tr>
<td>BNE ( \ell )</td>
<td>if ( \text{cond}(\neq) ) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with ( \neq ) for ?</td>
</tr>
<tr>
<td>BGT ( \ell )</td>
<td>if ( \text{cond}(&gt;) ) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with ( &gt; ) for ?</td>
</tr>
<tr>
<td>BLT ( \ell )</td>
<td>if ( \text{cond}(&lt;) ) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with ( &lt; ) for ?</td>
</tr>
<tr>
<td>BGE ( \ell )</td>
<td>if ( \text{cond}(\geq) ) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with ( \geq ) for ?</td>
</tr>
<tr>
<td>BLE ( \ell )</td>
<td>if ( \text{cond}(\leq) ) then ( \text{PC} := \ell )</td>
<td>Tests last CMP with ( \leq ) for ?</td>
</tr>
<tr>
<td>BL ( \ell )</td>
<td>LR := PC; PC := ( \ell )</td>
<td></td>
</tr>
</tbody>
</table>

\( \ell \) denotes a label.
Example: $x = y - z$

LDR R1, [R12, &y] // R1 = y
LDR R2, [R12, &z] // R2 = z
SUB R1, R1, R2 // R1 = R1 - R2
STR R1, [R12, &x] // x = R1

- $x, y, z$ are defined activation record offsets
Example: \( b = a[i] \)

\[
\text{LDR} \ R4,[R12,&i] \quad // \ R4 = i \\
\text{ADD} \ R5,R12,&a \quad // \ R5 = &a \\
\text{LDR} \ R6,[R5,R4,LSL#2] \quad // \ R6 = *(a+i) \\
\text{STR} \ R6,[R12,&b] \quad // \ b = R2
\]
Example: $a[j] = c$

LDR R4,[R12,&j] // R4 = j
ADD R5,R12,&a // R5 = &a
LDR R6,[R12,&c] // R6 = c
STR R6,[R5,R4,LSL#2] // *(a+i) = c
Example: \( *x = *y \)

\[
\begin{align*}
\text{LDR} & \quad \text{R4}, [\text{R12}, \&y] \quad \text{// R4} = y \\
\text{LDR} & \quad \text{R5}, [\text{R4}, \#0] \quad \text{// R5} = *y \\
\text{LDR} & \quad \text{R6}, [\text{R12}, \&x] \quad \text{// R6} = x \\
\text{STR} & \quad \text{R5}, [\text{R6}, \#0] \quad \text{// } *x = *y
\end{align*}
\]
Example: if $x < y$ goto $L$

```
LDR R4,[R12,&x]      // R4 = x
LDR R5,[R12,&y]      // R5 = y
CMPS R4,R5           // set flags from x?y
BLT L                // if x<y goto L
```
Example: Call

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

function int foo(int a, string b)
{
    return a;
}
```
ARM Call and Return Code Example

Caller:

```
bar       DCB "Bar"
MOV R0, #1
MOV R1, &bar
BL Callee
```

Callee:

```
STMFD sp!,{R4-R11,lr}
LDMFD sp!,{R4-R11,pc}
```

Continue...
ARM Call and Return Code Example

**Caller:**

```
bar       DCB "Bar"
MOV R0, #1
MOV R1, &bar
BL Callee
```

**Callee:**

```
STMFD sp!,{R4-R11,lr}
LDMFD sp!,{R4-R11,pc}
```

*Continue...*
ARM Call and Return Code Example

Caller:

bar DCB "Bar"

MOV R0, #1
MOV R1, &bar
BL Callee

Callee:

STMFD sp!,{lr}
LDMFD sp!,{pc}

Continue...
ARM Call and Return Code Example

Caller:

bar  DCB "Bar"

MOV R0, #1
MOV R1, &bar
BL Callee

Callee:

MOV pc, lr

Continue...
1. Issues
2. Target Language: ARM32
3. Basic Blocks & Flow Graphs
4. Project Milestone 3
### Idea

1. **Basic Blocks** are maximal sequences of consecutive three-address instructions,
   - control flow can only enter through first instruction,
   - control flow can only leave from last instruction.

2. **Flow Graph** is the graph with basic blocks as nodes and branches as directed edges.
Algorithm: Determine Basic Blocks

Input: A (numbered) sequence of three-address instructions.

Output: A mapping from the instruction (numbers) to basic blocks.

Method: Generate set of leaders.

1. The first instruction in sequence is leader.
2. Any instruction that is the target of branch is a leader.
3. Any instruction that follows a branch is a leader.

Now create a basic block per leader, and associate the leader with it. Associate all non-leaders with the basic block of the nearest preceeding leader.
Example: \( 10 \times 10 \) Identity Matrix

```plaintext
for i from 1 to 10 do
  for j from 1 to 10 do
    a[i,j] = 0.0;
  end for
end for

for i from 1 to 10 do
  a[i,i] = 1.0;
end for
```

- Numbers are double of size 8 bytes.
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.
Basic Block Next-Use

- A statement that assigns to a variable $x$ is said to define $x$.
- A statement that accesses a variable $x$ is said to use $x$.
- When a variable may still be used it is live.
Algorithm: Determine Liveness and Next-Use

Input: One basic block $B$ of three-address statements.

Output: Association from statements to liveness and next-use information.

Method: Start from last statement in $B$ and scan backwards to the beginning. Now, for each statement $i$ do the following:

1. Attach current information from table for all used variables.
2. If the statement table assigns to $x$ then change the state of $x$ to not live and no next use.
3. If the statement reads $y$ then change the state of $y$ to live and the next use information for $y$ to $i$. 
Flow Graphs

- Picture of basic blocks with arrows for possible control flows.
- A **loop** is sequence of distinct basic blocks where
  1. Entry to first block is **loop entry** is only label where control comes from basic blocks outside the sequence.
  2. It is possible to reach the loop entry from any block in the sequence.
Issues

Target Language: ARM32

Basic Blocks & Flow Graphs

Project Milestone 3
Project Milestone 3 Released!

- Includes **intermediate document** due 4/20.
- Main assignment due 5/1.
- Submit proper files!
- Hints...
Project Milestone 3 Released!

- Includes **intermediate document** due 4/20.
- Main assignment due 5/1.
- Submit proper files!
- Hints...
Project Milestone 3 Released!

- Main assignment due 5/1.
- Submit proper files!

Hints...
Project Milestone 3 Released!

- Main assignment due 5/1.
- Submit proper files!
- Hints...
Questions?