Implementing C Language Constructs

ECE 3140/CS 3420 — EMBEDDED SYSTEMS
Control flow: if-then-else

```c
if(x == 1)
    y1 += 1;
else if (x == 2)
    y2 += 1;
else
    y3 += 1;
```

```assembly
if:   CMP R0,#1       ; x is R0
      BNE els1

  thn1: ADD R1,R1,#1; y1 is R1
      B end

 els1: CMP R0,#2
      BNE els2

  thn2: ADD R2,R2,#1; y2 is R2
      B end

 els2: ADD R3,R3,#1; y3 is R3
 end:    ; ...
```
Control flow: switch

switch(x) {
    case 1:
        y1 += 1;
        break;
    case 2:
        y2 += 1;
        break;
    default:
        y3 += 1;
}

cal:  CMP R0,#1 ; x is R0
      BNE ca2
      ADD R1,R1,#1; y1 is R1
      B end ; break

cal:
      CMP R0,#1
      BNE   ca2
      ADD R1,R1,#1; y1 is R1
      B end ; break

cal2:  CMP R0,#2
       BNE  def
        ADD R2,R2,#1; y2 is R2
       B end

def:  ADD R3,R3,#1; y3 is R3
end:   ; ...
Control flow: for

```
for(i=0;i < NUM;i++)
y += i;
```

```
for:  MOV R0,#1  ; i is R0
next: CMP R0,#NUM
      BGE end
      ADD R1,R1,R0; y is R1
      ADD R0,R0,#1
      B next
end:   ; ...
Code reuse

- Some code gets used a lot
  - Needs to execute multiple times
  - Needs to execute at multiple program locations

- Example:

\[ s = \text{sum\_ints}(n) = \sum_{i=1}^{n} i \]
Approach 1: Inline code as needed

\[ s = 0; \]
\[ \text{for}(i=1; i<=n; i++) \]
\[ \quad s += i; \]

\begin{verbatim}
MOV R0,#0    ; s is R0
MOV R1,#1    ; i is R1
next: CMP R1,R2    ; n is R2
       BGT end
       ADD R0,R0,R1
       ADD R1,R1,#1
       B next
end:     ; ...
\end{verbatim}

- **Advantages:**
  - Simplest
  - Fastest

- **Disadvantages:**
  - Code size
Approach 2.1: Subroutines

\[ s = \text{sumi}(n); \]

\text{sumi:} \quad \text{MOV} \ R0,\#0 ; \ s \ is \ R0  \\
\quad \text{MOV} \ R1,\#1 ; \ i \ is \ R1  \\
\text{next:} \quad \text{CMP} \ R1,R2 ; \ n \ is \ R2  \\
\quad \text{BGT} \ \text{end}  \\
\quad \text{ADD} \ R0,R0,R1  \\
\quad \text{ADD} \ R1,R1,\#1  \\
\quad \text{B} \ \text{next}  \\
\text{end:} \quad \text{B} \ \text{cont} ; \ \text{return} \ \text{addr}

- **Advantages:**
  - Can invoke from multiple locations
  - First take: Jump and back
  - Does it work? Consider:

\begin{align*}
\text{loc1:} \ & \text{B} \ \text{sumi} \\
\text{cont:} & \ ; \ldots \\
\text{loc2:} & \ \text{B} \ \text{sumi} \\
\text{cnt2:} & \ ; \ldots
\end{align*}
Approach 2.2: Recall return address

\[ s = \text{sumi}(n); \]

\textbf{sumi:} \text{MOV} R0,#0 ; s \text{ is } R0
\text{MOV} R1,#1 ; i \text{ is } R1
\textbf{next:} \text{CMP} R1,R2 ; n \text{ is } R2
\text{BGT} \text{ end}
\text{ADD} R0,R0,R1
\text{ADD} R1,R1,#1
\text{B next}
\textbf{end:} \text{MOV} PC,R14

- Second take: Recall return PC
  - Store return address in R14
  - Load PC with R14 to return

- Does it work?

\[ \text{ADR} \ R14,\text{cnt1} \]
\[ \text{B} \ \text{sumi} \]
\[ \text{cnt1:} \ ; \ldots \]
Problem: ARM vs. Thumb modes

- Cortex-M processors can run two types of code
  - ARM code = 32-bit instruction mode (à la Cortex-Ax)
  - Thumb code = (largely) 16-bit instruction mode

- `ADR R14, cnt1` and `MOV PC, R14` do not allow “interworking”
  - E.g., Thumb code branching to ARM subroutine and back

- `BX <Rx>` changes processor mode as it jumps, based on bit b0 of Rx
  - Never used anyway: ARM code word-aligned, Thumb halfword-aligned

- `BL <addr>` stores return address in R14 with b0 set to caller’s mode, then performs branch
Approach 2.3: Branch-and-link

\[ s = \text{sumi}(n); \]

- \( \text{sumi}: \) MOV R0,#0 ; s is R0
  MOV R1,#1 ; i is R1
  next: CMP R1,R2 ; n is R2
        BGT end
        ADD R0,R0,R1
        ADD R1,R1,#1
        B next
  end: BX LR

- R14 called “link register (LR)”
  \[ \text{BL sumi} \]
  cnt1: ; ...

- What about:
  - Nested subroutine calls?
  - Recursive subroutines?
Approach 2.4: Stack

```
int sumi(int n) { /* recursive */
    if(n == 0)
        return 0;
    else
        return n+sumi(n-1);
}
```

- **Stack: Data structure**
  - Last-In First-Out (LIFO)
  - **PUSH Rx**: Grow stack by one word, store value Rx in it
  - **POP Rx**: Read out value at top of stack and into Rx, shrink stack by one word

```
sumi:   CMP R2,#0 ; n is R2
        BNE else
then:   MOV R0,#0 ; result in R0
        BX LR
else:
    ; ???
        BX LR
```
Approach 2.4: Stack

sumi: CMP R2,#0 ; n is R2
BNE else
then: MOV R0,#0 ; result in R0
BX LR
else: PUSH LR
PUSH R2
SUB R2,R2,#1
BL sumi
POP R2
ADD R0,R2,R0
POP LR
BX LR

- **Stack**: Data structure
  - Last-In First-Out (LIFO)
  - **PUSH Rx**: Grow stack by one word, store value Rx in it
  - **POP Rx**: Read out value at top of stack and into Rx, shrink stack by one word
Implementing a stack

- **Stack Pointer (SP):** Register that stores memory address of stack’s top word
  - `PUSH Rx` (pre-)decrements SP, then stores value Rx in (new) top
  - `POP <Rx>` stores top of stack in `<Rx>`, then *increments* SP

- **Note:** `LDR Rx, [SP, #v]` perfectly legal!
  - Non-destructive (unlike `POP`)
  - Allows *direct* access to variables deeper in the stack
  - Also note: $v \geq 0$, $v \mid 4$, $SP \mid 4$

- **SP is R13 in ARM**

<table>
<thead>
<tr>
<th>word d</th>
<th>0xaaaaaab4</th>
</tr>
</thead>
<tbody>
<tr>
<td>word c</td>
<td>0xaaaaaab0</td>
</tr>
<tr>
<td>word b</td>
<td>0xaaaaaaac</td>
</tr>
<tr>
<td>word a</td>
<td>0xaaaaaaa8</td>
</tr>
<tr>
<td>&lt;free&gt;</td>
<td></td>
</tr>
</tbody>
</table>
ARM syntax

- Pseudo-instructions `PUSH`, `POP` can list multiple registers

- `PUSH {Ra, Rx-Rz}`
  - Push registers in increasing ID order

- `POP {Ra, Rx-Rz}`
  - Pop registers in reverse order

- `POP {PC}` is equivalent to
  - `POP {Rx}`
  - `BX Rx`
  - Handles interworking correctly
Calling convention: Caller

\[ s = \text{foo}(p_0,p_1,p_2,p_3,p_4,p_5) \]

main: ;...
    ; assume px in Rx
PUSH \{R4,R5\}
BL foo
    ; result in R0
ADD SP,#8

- First four arguments in R0-3
- Excess arguments in the stack, in order
  - Reverse for variadic subroutines
- Return value will be in R0
Calling convention: Callee

```c
foo(int p0-p5) {
    /* ... */
    return x;
}

foo:
PUSH {R4-R11,LR}
; ...
; save result in R0
POP {R4-R11,PC}
```

- First four arguments in R0-3
- Excess arguments in the stack, in order
  - Reverse for *variadic* subroutines
- Return value will be in R0
- R4-11 *callee*-saved (as needed!)
- R12 assumed *scratch* register
Local variables

```c
int foo(<params>) {
    int x,y,z;
    /* ... */
    return <value>;
}
```

- Local variables allocated in the stack
  - Space reserved after pushing callee-saved registers and LR
  - Space recovered before popping callee-saved registers and PC

- Padded to be size | 4
  - Recall: SP | 4
Activation record

- Portion of the stack relevant to a particular subroutine during execution
- Accessed using `LDR/STR Rx, [SP,<depth>]`

```
main
(foo1)
(foo2)
```

```
<table>
<thead>
<tr>
<th>SP</th>
<th>higher addr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>main’s stuff</td>
</tr>
<tr>
<td></td>
<td>foo1’s parameters</td>
</tr>
<tr>
<td></td>
<td>foo1’s return address</td>
</tr>
<tr>
<td></td>
<td>foo1’s callee-saved registers</td>
</tr>
<tr>
<td></td>
<td>foo1’s local variables</td>
</tr>
<tr>
<td></td>
<td>foo2’s parameters</td>
</tr>
<tr>
<td></td>
<td>foo2’s return address</td>
</tr>
<tr>
<td></td>
<td>foo2’s callee-saved registers</td>
</tr>
<tr>
<td></td>
<td>foo2’s local variables</td>
</tr>
<tr>
<td></td>
<td>&lt;free&gt;</td>
</tr>
</tbody>
</table>
```