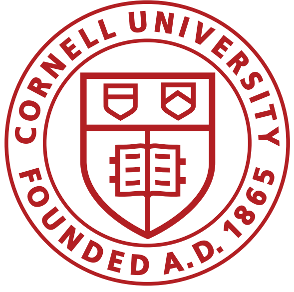


Implementing C Language Constructs

ECE 3140/CS 3420 — EMBEDDED SYSTEMS

© 2019 José F. Martínez. Unauthorized distribution prohibited.

Control flow: if-then-else

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

```
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;
}
```

```
ca1:  CMP R0,#1    ; x is R0
      BNE ca2
      ADD R1,R1,#1; y1 is R1
      B end        ; break
ca2:  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;
for(i=1;i<=n;i++)
  s += i;
```

```
      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:  ; ...
```

- Advantages:
  - Simplest
  - Fastest
- Disadvantages:
  - Code size

Approach 2.1: Subroutines

```
s = sumi(n);


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:  B cont       ; return addr
```

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

```
loc1: B sumi
cont: ; ...

loc2: B sumi
cnt2: ; ...
```

Approach 2.2: Recall return address

```
s = sumi(n);


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:  MOV PC,R14
```

- Second take: Recall return PC
  - Store return address in R14
  - Load PC with R14 to return
- Does it work?

```
      ADR R14,cnt1
      B sumi
cnt1: ; ...
```

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 = sumi(n);


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)”

```
      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);
}
```

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

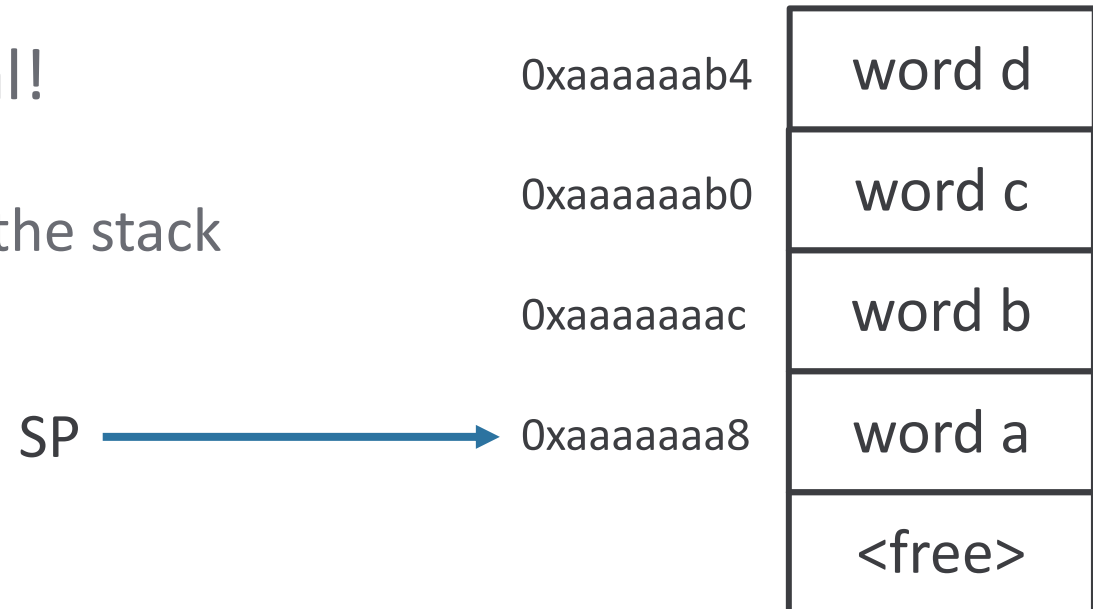
Approach 2.4: Stack

```
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 >= 0, v | 4, SP | 4
- SP is R13 in ARM



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 = foo(p0,p1,p2,p3,p4,p5)
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

```
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

```
int foo(<params>) {
  int x,y,z;
  /* ... */
  return <value>;
}
foo:   PUSH {R4-11,LR}
       SUB SP,#12   ; 3*4B
       ;...
       ADD SP,#12
       POP {R4-11,PC}
```

- 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>]`