Implementing C Language Constructs

ECE 3140/CS 3420 — EMBEDDED SYSTEMS

Control flow: if-then-else

```
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 = 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
```

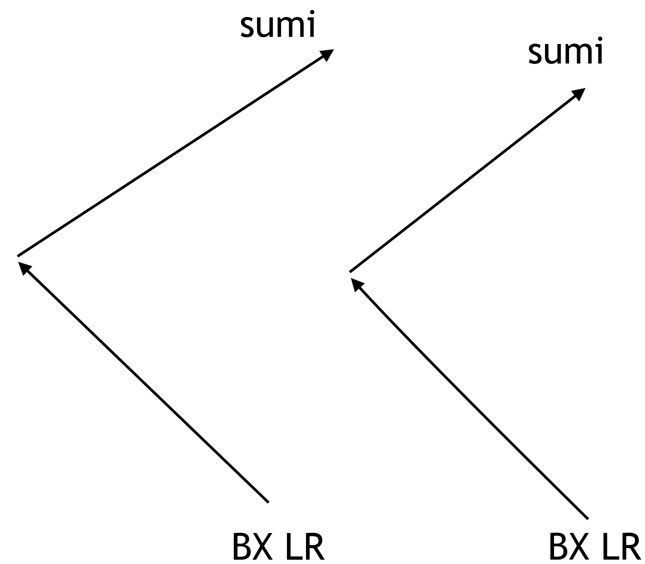
- R14 called "link register (LR)"

```
        BL sumi
   cnt1: ; ...
```

- What about:
  - Nested subroutine calls?
  - Recursive subroutines?

2.4 Stack

```
int sumi(int n) {
      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
else: …
        BX LR
```

```
else: Push LR                       // where came from
      *PUSH R2                      // remember n; caller-saved register
      SUB R2 R2 1                   // decrement n
      BL sumi
      *POP R2                       // LR pointing here; overriden?
      ADD R0 R2 R0                  // add to whatever result is passed
      POP LR (contains count)
      BX LR
```

BL: sets link register after
BX: just jumps to somewhere else
if values needs to be preserved; push to stack and then pop
Callee-saved register vs caller-saved register
- caller-saved registers are used to hold temporary quantities
- callee-saved registers are used to hold long-lived values that should be preserved across calls.

Implementing Stack
- Starts from high address and grows downward
- Need SP: stack pointer- reg that stores top word memory address
- Push: move SP forward and put in free space
- Pop: remove value and move SP backwards

Example

```
s = foo(p0,p1,p2,p3,p4,p5)
main:
      PUSH {R4, R5}          // R0-R3 are already in stack; pushing R4,R5 for the sake of foo
      BL foo
      ADD SP 8               // to return stack pointer to position in PUSH R4,R5 (back to beginning)
```

*For stack you must leave it the way it began