C functions

```c
main() {
    int a, b, c;
    ...
    c = sum(a, b); /* a, b, c: r0, r1, r2 */
    ...
}

/* really dumb sum function */
int sum(int x, int y) {
    return x + y;
}
```

What information must the compiler/programmer keep track of?

What instructions can accomplish this?
Registers play a major role in keeping track of information for function calls

- **Register conventions:**
  - Return address: \( lr \)
  - Arguments: \( r0, r1, r2, r3 \)
  - Return value: \( r0, r1, r2, r3 \)
  - Local variables: \( r4, r5, \ldots, r12 \)

- The stack is also used; more later
Register Usage

Arguments into function
Result(s) from function
otherwise corruptible
(Additional parameters passed on stack)

Register variables
Must be preserved

Scratch register (corruptible)

Stack Pointer
Link Register
Program Counter

The compiler has a set of rules known as a Procedure Call Standard that determine how to pass parameters to a function (see AAPCS).

CPSR flags may be corrupted by function call. Assembler code which links with compiled code must follow the AAPCS at external interfaces.

The AAPCS is part of the new ABI for the ARM Architecture.

- Stack base
- Stack limit if software stack checking selected

- SP should always be 8-byte (2 word) aligned
- R14 can be used as a temporary once value stacked
In ARM, all instructions are stored in memory just like data. So here we show the addresses of where the programs are stored.
C

```c
int sum(int x, int y) {
    return x+y;
}
```

ARM

```
address
1000  MOV  r0,r4 ; x = a
1004  MOV  r1,r5 ; y = b
1008  MOV  lr,1016 ; lr = 1016
1012  B    sum ; branch to sum
1016  ...
2000  sum: ADD r0,r0,r1
2004  BX lr ; MOV pc,lr i.e., return
```
Instruction Support for Functions

... sum(a,b);... /* a,b:$s0,$s1 */

C

```c
int sum(int x, int y) {
    return x+y;
}
```

Question: Why use BX here? Why not simply use B?

Answer: sum might be called by many functions, so we can’t return to a fixed place. The calling proc to sum must be able to say “return here” somehow.
Instruction Support for Functions

- Single instruction to jump and save return address: jump and link (BL)

- Before:
  1008 MOV lr, 1016 ; lr=1016
  1012 B sum ; goto sum

- After:
  1008 BL sum # lr=1012, goto sum

- Why have a BL? Make the common case fast: function calls are very common. Also, you don’t have to know where the code is loaded into memory with BL.
Instruction Support for Functions

- Syntax for BL (branch and link) is same as for B (branc):
  \[ \text{BL label} \]

- BL functionality:
  - Step 1 (link): Save address of next instruction into lr (Why next instruction? Why not current one?)
  - Step 2 (branch): Branch to the given label
Instruction Support for Functions

- Syntax for BX (branch and exchange):
  - BX register

- Instead of providing a label to jump to, the BX instruction provides a register which contains an address to jump to

- Only useful if we know exact address to jump

- Very useful for function calls:
  - BL stores return address in register (lr)
  - BX lr jumps back to that address
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}

- Something called `sumSquare`, now `sumSquare` is calling `mult`.
- So there’s a value in `lr` that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`.
- Need to save `sumSquare` return address before call to `mult`. 
Nested Procedures

- In general, may need to save some other info in addition to lr.

- When a C program is run, there are 3 important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes. E.g., C globals
  - Heap: Variables declared dynamically
  - Stack: Space to be used by procedure during execution; this is where we can save register values
C Memory Allocation

Address $\infty$

Stack

Heap

Static

Code

Space for saved procedure information

Explicitly created space, e.g., malloc(); C pointers

Variables declared once per program

Program
Using the Stack

- So we have a register $sp$ which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```c
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}
```
Using the Stack

- **Hand-compile**

```c
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}
```

```assembly
// sumSquare:
ADD sp, sp, #8           ; space on stack
STR lr, [sp, #4]         ; save ret addr
STR r1, [sp]             ; save y

MOV r1, r0               ; mult(x,x)
BL mult                  ; call mult

LDR r1, [sp]             ; restore y
ADD r0, r0, r1           ; mult()+y

LDR lr, [sp, #4]         ; get ret addr
ADD sp, sp, #8           ; restore stack
BX lr
```

**“push”**

**“pop”**
Steps for Making a Procedure Call

1) Save necessary values onto stack
2) Assign argument(s), if any
3) BL call
4) Restore values from stack
Rules for Procedures

- Called with a `BL` instruction, returns with a `BX lr` (or `MOV pc, lr`)
- Accepts up to 4 arguments in r0, r1, r2 and r3
- Return value is always in r0 (and if necessary in r1, r2, r3)
- Must follow register conventions (even in functions that only you will call)!

So what are they?
## ARM Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Synonym</th>
<th>Role in Procedure Call Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0-r1</td>
<td>a1-a2</td>
<td>Argument/Result/Scratch Register</td>
</tr>
<tr>
<td>r2-r3</td>
<td>a3-a4</td>
<td>Argument/Scratch Register</td>
</tr>
<tr>
<td>r4-r8</td>
<td>v1-v5</td>
<td>Variable Register</td>
</tr>
<tr>
<td>r9</td>
<td>v6/sb/tr</td>
<td>Platform Register</td>
</tr>
<tr>
<td>r10-r11</td>
<td>v7-v8</td>
<td>Variable Register</td>
</tr>
<tr>
<td>r12</td>
<td>ip</td>
<td>Intra-Procedure Call Scratch Register</td>
</tr>
<tr>
<td>r13</td>
<td>sp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>r14</td>
<td>lr</td>
<td>Link Register</td>
</tr>
<tr>
<td>r15</td>
<td>pc</td>
<td>Program Counter</td>
</tr>
</tbody>
</table>
Register Conventions

- **CalleR**: the calling function
- **CalleE**: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- **Register Conventions**: A set of generally accepted rules as to which registers will be unchanged after a procedure call (BL) and which may be changed.
Saved Register Conventions

- **r4-r11 (v1-v8):** *Restore if you change.* Very important. If the *callee* changes these in any way, it must restore the original values before returning.

- **sp:** *Restore if you change.* The stack pointer must point to the same place before and after the `BL` call, or else the caller won’t be able to restore values from the stack.
Volatile Register Conventions

- **lr**: Can Change. The BX call itself will change this register. Caller needs to save on stack if nested call.

- **r0-r3 (a1-a4)**: Can change. These are volatile argument registers. Caller needs to save if they’ll need them after the call. E.g., r0 will change if there is a return value.

- **r12 (ip)** may be used by a linker as a scratch register between a routine and any subroutine it calls. It can also be used within a routine to hold intermediate values between subroutine calls.
Register Conventions

- What do these conventions mean?
  - If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a BL call.
  - Function E must save any saved registers it intends to use before garbling up their values.
  - Remember: Caller/callee need to save only volatile/saved registers they are using, not all registers.
Basic Structure of a Function

**Prologue**

entry_label:
ADD sp,sp, -framesize
STR lr, [sp, #framesize-4]; save lr
save other regs if need be

**Body**  ... (call other functions...)

**Epilogue**

restore other regs if need be
LDR lr, [$sp, framesize-4]; restore lr
ADD sp,sp, #framesize
BX lr
Example

```c
main() {  
    int i, j, k, m; /* i-m: v0-v3 */
    ...
    i = mult(j, k); ...
    m = mult(i, i); ...  
}

int mult (int mcand, int mlier) {  
    int product;
    product = 0;
    while (mlier > 0) {  
        product += mcand;
        mlier -= 1;  }
    return product; }
```
Example

```c
main() {
    int i, j, k, m; /* i-m:v0-v3 */
    ...
    i = mult(j,k); ...
    m = mult(i,i); ...
}

__start:
    MOV a1,v1 ; arg1 = j
    MOV a2,v2 ; arg2 = k
    BL mult ; call mult
    MOV v0,r0 ; i = mult()
    ...
    MOV a1,v0 ; arg1 = i
    MOV a2,v0 ; arg2 = i
    BL mult ; call mult
    MOV v3,r0 ; m = mult()
    ...
    done
```
Example

Notes:

- `main` function ends with `done`, not `BX lr`, so there’s no need to save `lr` onto stack.
- All variables used in `main` function are saved registers, so there’s no need to save these onto stack.
Example

```c
int mult (int mcand, int mlier){
    int product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;
}
```

```
mult:  MOV a3,#0; prod=0
```

```
Loop:  CMP a2, #0 ; mlier == 0?
       BLE Fin   ; if mlier <= 0 goto Fin
       ADD a3,a3,a1 ; product += mcand
       ADD a2,a2,#-1 ; mlier -= 1
       B Loop     ; goto Loop
```

```
Fin:    MOV a1,a3 ; setup return value
        BX lr     ; return
```
Example

Notes:

- No BL calls are made from `mult` and we don’t use any saved registers, so we don’t need to save anything onto stack.
- Temp registers are used for intermediate calculations (could have used saved registers, but would have to save the caller’s on the stack.)
- `a2` is modified directly (instead of copying into a temp register) since we are free to change it.
- Result is put into `a1 (r0)` before returning.
Functions are called with BL, and return with BX lr.

The stack is your friend: Use it to save anything you need. Just be sure to leave it the way you found it.

Register Conventions: Each register has a purpose and limits to its usage. Learn these and follow them, even if you’re writing all the code yourself.
Conclusion

Instructions so far:
- Previously:
  - ADD, SUB, MUL, MULA, [U|S]MULL, [U|S]MLAL, RSB
  - AND, ORR, EOR, BIC
  - MOV, MVN
  - LSL, LSR, ASR, ROR
  - CMP, B{EQ,NE,LT,LE,GT,GE}
  - LDR, LDR, STR, LDRB, STRB, LDRH, STRH
- New:
  - BL, BX

Registers we know so far
- All of them!