ECE 271 – Microcomputer Architecture and Applications Lecture 13

Vince Weaver

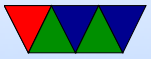
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

1 March 2022

Announcements

- Read Chapters 7 and 8
- Midterm, Tuesday, 8 March
  will send out worksheet with flags sample problem



Some C Gotchas

- Many errors are just obscure C coding issues
- Index variable scoping

```
int j;
for(j=0;j<100;j++) {
    ...

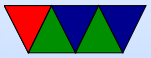
    for(j=0;j<1000;j++) ; // delay

}
```

- Operator precedence

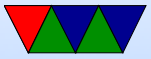
```
if (x & 0x10==0) {
// actually is
if (x & (0x10==0))
```

A safe bet is to just use extraneous parenthesis

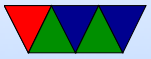


Coding Style

- Some people were asking if we were enforcing coding style in this class?
- Coding style is one of those things that varies from person to person and project to project and is hard to quantify.
- Does C have any rules about code formatting? Very few, see `https://www.ioccc.org/`
- People can be very opinionated.
- Some examples of coding style:

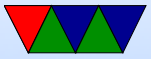


- Indentation: tabs vs spaces (and how many)
- Width of screen/wrapping: 80 col or more?
- Variable names: `new_x_size`, `NewXSize` (camel case), Hungarian Notation (strName) with type info
- Length of identifiers (old compilers ignored any past 6)
- Curly bracket on same line or next
- Header files include bare minimum, or include all
- Header files alphabetical, at end, Christmas tree (for git collision reasons)
- ```
  if (x==0)
  if (0==x)
  ```

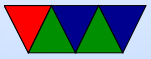

  ```

Register Allocation

- Should you leave your variables in RAM?
- Modern machines RAM is slow (though not as bad on the boards in class)
- Try to have frequently used values in registers if at all possible
- Long-running values left in as long as possible
- Once done using register, can re-use it for another variable
- Live range

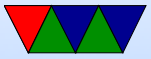


- Compilers try to do this automatically



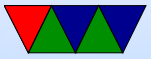
Subroutines / Functions

- Why use them?
  - Understandability (easier to follow short blocks of code)
  - Disk/RAM space (takes up less room than cut/pasting)
  - Centralizing code (find a mistake, only need to fix it once)
  - Re-usability (can reuse code again)
  - Performance? This is complex. Can lead to smaller



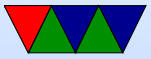
code fitting in caches, but at same time branches are expensive and have overhead

- Avoiding overhead
  - What if function only used once, should we just include it in main code to avoid function call overhead?
  - Modern compilers will try to do this for you, *inlining*
  - Some C compilers have an `inline` keyword where you can try to force the function to be inlined for performance



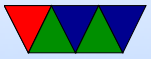
Putting Common Subroutines in Own File

- For example, what if want to put `print_lcd()` function into `lcd.c`
- How could you call it from `main.c`?
- You can pre-declare it `extern` at the top of your file (in C actually extern is the default for all function definitions)
- Often if you want to share code like this, you put the extern definition into a header file, something like `lcd.h` and then `#include "lcd.h"`
- To make a function only visible in the file it's in (the



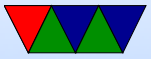
opposite of `extern`) declare it `static`

- When compiling, the C compiler makes assembler and then assembles to .o object files. These have placeholders for any external routines. Finally it calls the linker which goes through and joins all the files together and patches up all the calls to external routines to point to the write place, finally making the executable.
- Note: could you just put your C code in a header file (not just the definition, but all the code) and include that? You could, usually frowned upon. Try to avoid having code in header files.



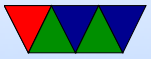
The Stack – Review

- Chunk of memory, LIFO.
- On ARM by default grows down, "Full"
  (full means points to last value pushed, empty would point to next)
- Why does it grow down? Can it grow up?
  Most processors assume you will grow down as it tends to maximize usable space. You often put the stack as high up as possible so it has lots of room to grow down.

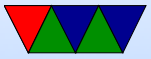


Global vs Local vars

- Memory layout diagram again
- Code/text (usually read only)
- Data (globals) initialized variables
- BSS (uninitialized / zeroed global variables)
  usually the OS clears these out, on our system we include startup code that does this
- Heap (dynamically allocated: `malloc` or `new`), grows "up"
- Stack, typically toward the top of memory, grows down.



Temporary variables and local variables

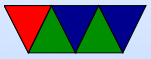


Stack Instructions

- ARM has complex stack manipulation instructions LDM/STM but these days people tend to just use push/pop
- `push {R4}`

  SP=SP-4. [SP]=R4
- `pop {R4}`

  R4=[SP]. SP=SP+4
- Push/Pull multiple.



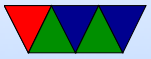
```
push {r1,r2,r4,lr}
```

can push multiple registers in one instruction, equivalent to pushing one at a time

- Returning shortcut

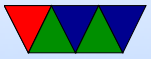
```
push {lr}
pop  {pc}
```

special case, can pop link register into PC which will return from a function



Function Prolog

- First thing an assembly language function does is save the link-register (if it's not a leaf function)
- It also needs to save any callee saved registers that will be used by the function (r4 - r11)
- It will also need to reserve room for local variables

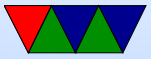


Local Variables

- In C, local variables are stored on the stack
- This gives them a lifetime of only while function is running
- Space is allocated in prolog by moving stack pointer
- i.e., to allocate an integer (32-bit) array with 100 entries

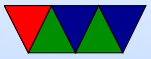
```
sub sp,sp,\#400
```

- To find the address you need to index into the stack properly
  Often a "frame pointer" (r11) holds a pointer to where



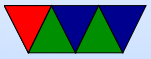
the stack begins to make this easier

- Otherwise as you use the stack in the program it can be hard to remember when the local variables live
- Is the stack initialized to any value when you allocate?
  - In C, usually no
  - So if you forget to initialize a local variable it can end up having whatever value was left of the stack from earlier code.
  - This can be really hard to debug, as doing things like printf() can move the stack and change the values there and your code might accidentally work



- Is this a security problem?

- Stack overflow bugs
  - What happens if you accidentally index off the end of the stack?
  - can you accidentally over-write the saved return-address?
  - What happens then? best-case crash. worst case, clever hacker can return to somewhere they can control, take over system

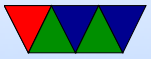


Function Epilog

- First deallocate any local variable space on stack

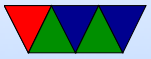
```
add sp,sp,\#400
```

- Next restore any registers that were saved
- Finally pop off the link register (if we weren't a leaf function) and return to the calling routine



Recursion

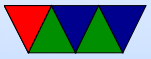
- Very CS thing to do
- Function calls itself
- ECE / embedded not like to do it much. Why? What happens when run out of stack?
- Can be useful. Think compilers?
- You'll see it in Google interviews



Factorial Example

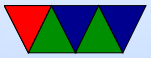
- n! = n * (n-1) * (n-2) ... * 1
- A normal person would implement it like

```
int factorial(int n) {
   int result=1;
   for(i=1;i<=n;i++) result*=i;
   return result;
}
```



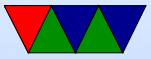
Factorial via Recursion

- factorial(0) = 1
- factorial(1) = 1 = 1*factorial(0)
- factorial(2) = 2 = 2*factorial(1)
- factorial(3) = 6 = 3*factorial(2)



Factorial Example – C

```
int factorial(int n) {
   if (n<2) return 1;
   return (n*Factorial(n-1));
}
```



Factorial Example – Assembler

```
factorial
   push {r4,lr} // save r4 (why?) save lr (why?)
   mov  r4,r0   // copy input arg to r4
   cmp  r4,#2
   bge  else    // if 2 or greater skip ahead
   mov  r0,#1   // otherwise return 1
   b    factorial_exit
else
   sub  r0,r4,#1   // arg is oldarg-1
   bl   factorial
   mul  r0,r4,r0   // return value in r0
                   // multiply by r4 (which was saved across call)
factorial_exit
   pop  {r4,pc}    // why have only one exit to function?


_start
   mov r0,\#0x3
   bl factorial
stop
   b stop
```