ECE 271 – Microcomputer Architecture and Applications Lecture 5

Vince Weaver

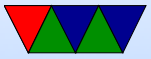
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

1 February 2022

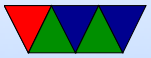
Announcements

- Read Chapter #3 and #4 of the book.
- We have a grad TA, Colin Leary, who will be having office hours on Wednesday at 2pm.
  If earlier/later/different day might work better for everyone, let me know.
- Reminder: no food or drink in the labs



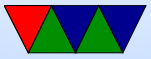
More Lab Notes

- Provided character translation code isn't the best
- Difference between uint8_t vs char?
- Something else the code does, copying data and bss segments
- Strings in C, pointers
- Commenting styles, Doxygen
- Using the predefined constants in stm32l476xx.h
- How do you make a delay? For loop? Don't forget the volatile.

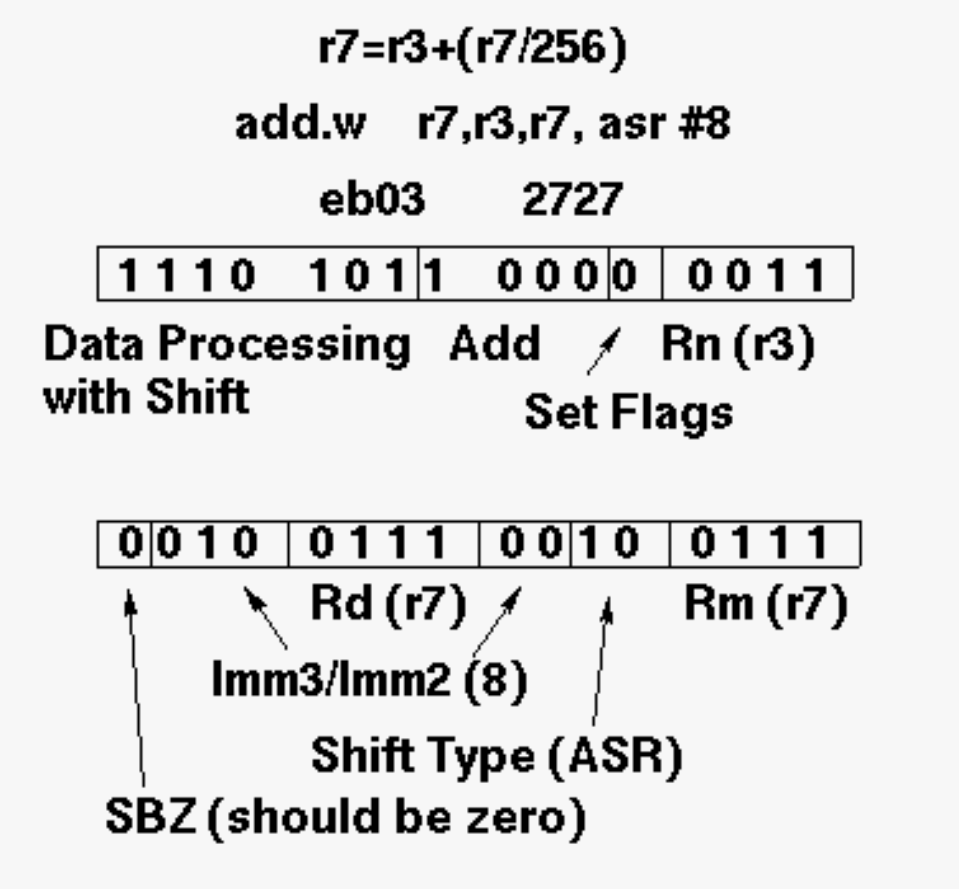


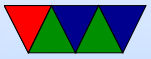
Assembly Language: What's it good for?

- Understanding your computer at a low-level
- Shown when using a debugger
- It's the eventual target of compilers
- Operating system writers (some things not expressible in C)
- Embedded systems (code density)
- Research. Computer Architecture. Emulators/Simulators.
- Video games (or other perf critical routines, glibc, kernel, etc.)



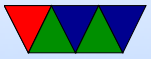
Thumb-2 Encoding Example





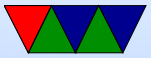
Registers

- How are registers designed?
  SRAM (static RAM: flip-flops)
  Aside: main memory on a desktop/laptop is DRAM (dynamic RAM) with one transistor and a capacitor, which drains quickly and has to be constantly refreshed.
- Three ports: two output and one input
- The rules for what goes in what register are part of the ABI (Application Binary Interface)



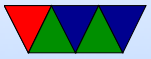
ARM32 registers

- Has 16 GP registers
- r0 - r12 are general purpose
- r11 is sometimes the frame pointer (fp) [iOS uses r7]
- r12 is inter-procedure scratch reg (ip)
- r13 is stack pointer (sp)
- r14 is link register (lr)
- r15 is program counter (pc)
  reading r15 usually gives PC+8



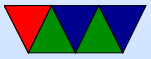
Application Binary Interface (ABI)

- There are no hardware rules about what to put in a register
- There are software guidelines though
- This helps in large programs where it can be trouble remembering what register does which
- It can help when working in a large project and you want your code to work with others
- It helps when you want to call standard libraries or functions from your code (e.g. `printf()`



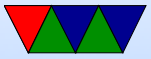
ARM ABI

- r0 - r3 are arguments to functions (a1-a4)
  these are caller saved (the caller must save the values, the function is allowed to change them)
- r0 is return value from function
- r4 - r10 are general purpose (v1-v8)
  these are callee saved, the function must save them if it wants to change them and restore them before returning
- There are additional corner cases (values more than 32-bits, floating point values, etc)



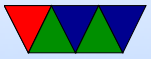
The Stack

- A stack is a LIFO (last-in first-out) data structure common on most processors
- A stack pointer (r13 on ARM) points to a region of memory
- Often stacks grow down (meaning, the stack pointer starts near the top of memory and when items are pushed the SP decreases toward 0)
- You can PUSH a 32-bit register value onto the stack This will place the item in memory and decrement the



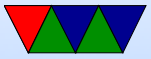
stack pointer by 4 bytes

- You can POP a 32-bit value off the stack into a register This will take the item from memory and increment the stack pointer by 4 bytes
- Values are stored on the stack with no additional identifying info. If you are pushing/poping values it's up to you to make sure you do things in the right order to get the proper values back.

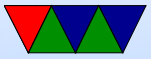


Use of The Stack

- If you run out of registers and need more for your computation, you can temporarily save values on the stack
- At entry to a function, any callee saved registers are usually stored on the stack, and they are restored at the end before returning.
- In non-leaf functions, often the Link Register return value is saved on the stack as well
- Any arguments beyond a1-a4 are passed on the stack

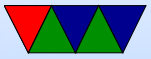


- In C, room for local variables is allocated on the stack. Often the Frame Pointer helps in accessing these. When the function ends, the stack is restored and the variables go away



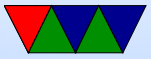
ARM Status Register

- 1 status register (more in system mode).
  NZCVQ (Negative, Zero, Carry, oVerflow, Saturate)



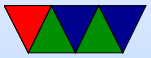
Let's start with ALU instructions

- a=b+c;
- How is ALU designed? Adder/subtractor/logic?



Add instruction

- `add r1,r2,r3` – r1=r2+r3
- Gets the values, adds two, stores in third

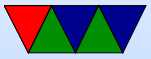


What does an assembly line look like

- annoyingly this can vary by platform, and even by assembler program on the same platform. (could be worse, intel vs at&t on x86)
- GNU asm style:

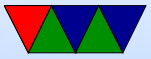
```
label: opcode  dest, src1, src2 ; comment
/* comment */
```

- Keil style:



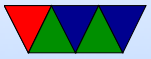
```
label
 opcode  dest, src1, src2 ; comment
```

- Label marks a point in the program. If you reference it the assembler will turn it to an address. You can do things like jump/branch/goto it. You can load from/store to it.
- The opcode or mnemonic says what you want to do. add/sub/eor, etc



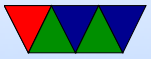
Assembly Directives: Keil / GNU

- Put this in your code to give the assembler directions
- Things like where to reserve memory, where functions start, etc.
- Slightly different from Keil to GNU (GNU starts with a .)



Add instruction

- `add r1,r2,r3` – r1=r2+r3
- `add r1,r2,#immediate` r1=r2+constant
  There are limits to constant size. Why?
  The thumb2 constants are exciting, will get to later



Settings Flags

- `adds r1,r2,r3` – set condition flag
  flags NZCV
  - N = negative (how can you tell if negative?)
  - Z = zero (how can you tell if zero?)
  - C = carry (how can you tell if carry? Why is it useful?)
  - V = overflow (will get to this later), signed overflow
  - Q = saturate