ECE 271 – Microcomputer Architecture and Applications Lecture 6

Vince Weaver

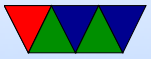
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

3 February 2022

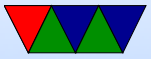
Announcements

- Read Chapter 14.9 for the Lab
- Read Chapters 3+4 to learn about ARM assembly
- We have gitlab just about ready to do, we'll send further directions when we have them

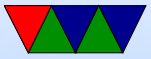


Lab#3, Keypad scanning

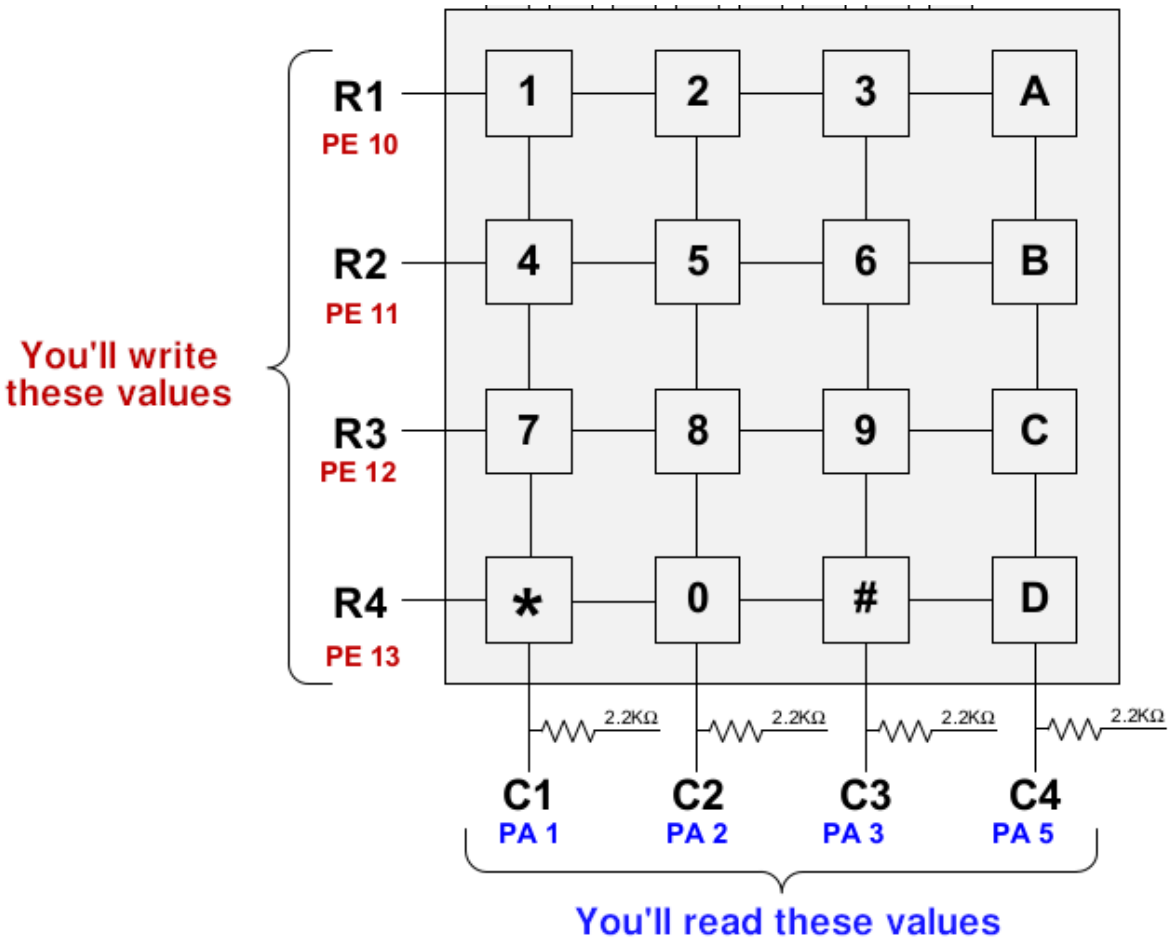
- Did you already do this in ECE177?
- Pre-lab will be posted. Very straightforward.
  Do not be lulled into complacency! Lab itself a bit tricky.
- Will hate to put some wires and resistors on a breadboard
  Hopefully everyone knows how breadboards work?
- Be sure to bring in a breadboard from previous classes and jumper wire.
  We will provide one, but it can be handy to have second

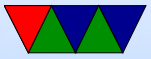


- Keypads – I think all of the ones this year have 4 columns, but if you get one with three it works just the same, just can't type ABCD



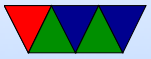
Keypad Setup





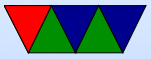
Keypad Scanning – Seeing if Any Pressed

- With 16 buttons, how many GPIOs do you need? By scanning only need 4+4=8
- Column pulled high to 3.3V
- First set all PE gpio (row) pins low to (0b0000)
- Next read out PA gpio (column) pins. If all still high (0b1111) then it means nothing was pressed (as we are pulling the lines high with the resistors, so they will only be pulled low if a button is pressed)



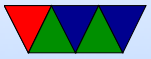
Keypad Scanning – Finding which one

- If key press detected, need to try each row at a time and see which one it was.
- You can try each row by writing out the pattern 0b1110, 0b1101, 0b1011, 0b0111
- Once you know the row and the column, you can translate this to the button pressed. How? Big if/then statement? switch/case? Lookup table?



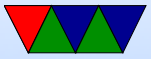
Keypad Scanning – Pressing Multiple Keys

- What happens if two keys pressed at once?
- Try not to do this, as you could short 3V to GND.
- Textbook goes on at length about this and describes ways to handle this.

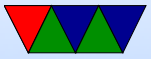


Keypad Scanning – Debouncing

- Debouncing – if your keypresses are noisy they will register as extra keypresses.
- Can you do this in hardware, maybe add capacitor? Maybe, but adds latency, also you need extra parts.
- One way is to read multiple times and only register keypress if multiple reads are all the same (say three in a row)
- Can also do thinks by having a delay to see if signal stabalizes, but this can have latency



- Also, what happens if someone holds a key down? Should it auto-repeat? Should you only register as one keypress (maybe on release of key?)

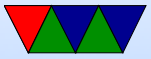


LCD Output

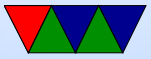
- Use your working code from Lab#2, specifically the `LCD_Display_String()` code
- First step is to wire up things and just read out.
  I made a first step where I printed the binary values to be sure switch hooked up right.
- How do you do that? Lots of ways

```
char string[7]; // why 7?
string[6]=0;    // why?
string[0]='X';
string[1]=(((GPIOA->IDR)&(1<<5))>>5)+'0';
string[2]=(((GPIOA->IDR)&(1<<5))>>5)?'1':'0';
```

- Then once you can see the keypad is working, go in and



write the code that scans rows/columns and prints the proper character to the LCD.



LCD Output

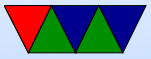
- Want to display keypresses as they occur, then scroll them to the left.
- If you have a character array:

```
char buffer[6]="␣␣1234";
```

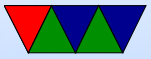
- One way is to just copy with a loop:

```
for(i=0;i<5;i++) buffer[i]=buffer[i+1];
buffer[5]='5';
```

- You might think of C library `memcpy()` or `memmove()` but on a small embedded board we might have those available.

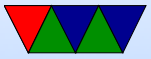


Back to Assembly Language



Math operations

- Note: can use 'S' and immediate with all of these too
- add r0,r1,r2 – add r0=r1+r2
- adc r0,r1,r2 – add with carry: r0=r1+r2+C
- sub r0,r1,r2 – subtract: r0=r1-r2
- sbc r0,r1,r2 – subtract with carry (borrow): r0=r1-r2-(NOT carry)
- rsb r0,r1,r2 – reverse subtract: r0=r2-r1
- rsc r0,r1,r2 – reverse subtract: r0=r2-r1

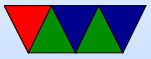


Add with Carry (8-bit example)

Say we want to add 16-bit value 0x0AFF + 0x8101 but we only have 8-bit wide add instruction. You can chain together 8-bit adds using the carry flag.

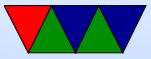
Use `adds` to add the low bytes and update flags

		1	1	1	1		1	1	1	1
	+	0	0	0	0		0	0	0	1
1	←	0	0	0	0		0	0	0	0
(C)										



Add the upper bytes using `adc` which adds in the existing carry flag (from the previous `adds`)

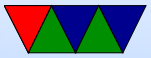
$$
\begin{array}{cccc cccc l}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \\
 & & & & & & + & 1 & \leftarrow \text{(C)} \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 &
\end{array}
$$



Subtract with Borrow (8-bit example)

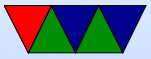
Say we want to subtract 16-bit 0x8000 - 0x0001 but we only have 8-bit wide ALU. You can chain together 8-bit subtracts using the carry (borrow) flag.

Use `subs` to subtract the low bytes and update flags. Carry is 1 if no borrow, 0 if there is a borrow.



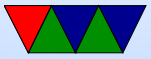
		0	0	0	0		0	0	0	0
	-	0	0	0	0		0	0	0	1
0 (borrow)	$\leftarrow$	1	1	1	1		1	1	1	1
(C)										

Subtract the upper bytes using `sbc` which uses the existing carry flag (from the previous `subs`). Note the *inverse* of the carry is subtracted off.



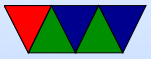
	1	0	0	0		0	0	0	0	
	0	0	0	0		0	0	0	0	
	1	0	0	0		0	0	0	0	
								-	1	← inverse of (C)
	0	1	1	1		1	1	1	1	

Why use inverted borrow? It makes the ALU easier if using twos complement. To SUB you just invert the second term, add them, then add 1. To SBC you add in the carry instead of the 1.



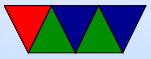
Complicated Operands

- add r0,r1,#5 – add constant, r=r1+5
- adds r0,r1,r2 – update flags
- add.w r0,r1,r2 – wide (use 32-bit rather than 16 bit encoding)
- addeq r0,r1,r2 – conditional execution, more on that later
- addeq r0,r1,r2,asr #8 – barrel shift more on that later
- addseq.w r0,r1,r2,asr #8 – you can combine them all

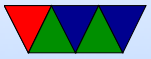


Moves

- mov r0,r1 – moves (copies) the value in r1 into r0
- mvn r0,r1 – moves (copies) the 1's complement inverse of r1 into r0
- ADR r0,symbol – move address of symbol into r0
- movw r0,#value – move 16-bit value into bottom, clear top
- movt r0,#value – move 16-bit value into top, leave

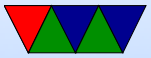


bottom alone



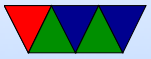
Bitwise

- and r0,r1,r2 – bitwise and: r0=r1 AND r2
- orr r0,r1,r2 – bitwise or: r0=r1 OR r2
- eor r0,r1,r2 – exclusive or: r0=r1 XOR r2
- orn r0,r1,r2 – or with inverse: r0=r1 OR (1's complement r2)
- bic r0,r1,r2 – bit clear (and not): r0=r1 and not r2



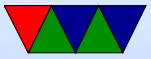
Test/Compare

- tst r1,r2 – test bit (does and, but only updates flags)
- teq r1,r2 – test equal (does eor, but only updates flags)
- cmp r1,r2 – compare (does subtract, only updates flags)
- cmn r1,r2 – compare negative (does add, only updates flags)



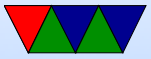
Shifting

- Shift bits one way or another
- Note: carry and N/Z only updated if the 'S' variant used
- The bit that gets shifted out is put into the carry.
- Why into carry? What if want to do 64-bit shift?
  Also can be clever and do things that are hard in C, like shift right and test C to see if low bit was 1.
- LSL r1,r2,r3 – logical shift left r2 by amount in r3 (or immediate) and store in r1.
- What happens if shift value larger than 32? It saturates



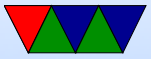
(goes to 0)

note that x86 is different, does mod (so shift by 32 same as shift by 0). Undefined behavior in C, be careful!

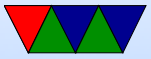


Shift instructions

- LSL r1,r2,r3 – logical shift left (shift in zeros)
  a shift left by one is the same as multiply by 2
  high bit shifted off goes into carry flag
- LSR r1,r2,r3 – logical shift right
  a shift right by one is the same as divide by 2
  0 shifted in on left, low bit shifted out into carry
- ASR r1,r1,r3 – arithmetic shift right
  sign (high bit) shifted in (preserving sign)
  low bit goes into carry

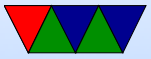


- Is there an ASL (arithmetic shift left?) Not needed



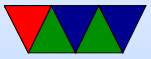
Rotate Instructions

- ROR r1,r2 – rotate right
  lo bit into carry and into hi
- RRX r1,r2 – rotate right, extended, so through carry lo to carry, carry to hi
- Is there an ROL? Turns out it ROL by 5 is same as ROR by (32-5)



Barrel Shifts

- For ALU instructions, and some others.
- The third argument can optionally be shifted by a constant
  - `add r1,r2,r3 LSR #2`
    r1=r2+(r3¡¡2)
  - LSL, LSR, ASR, ROR, RRX
  - on arm32 could have a 4th register instead of a constant as shift amount
- Why would you want to do this?



Accessing 32-bit values in an array

Hack, really fast multiplies

Example: `add r0,r1,r1 LSL #2` is same as r0=r1*5