ECE 271 – Microcomputer Architecture and Applications
Lecture 6

Vince Weaver

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

vincent.weaver@maine.edu

7 February 2019

Announcements

- Read Chapter 14.9 for the Lab
- Read Chapters 3+4 to learn about ARM assembly

Lab#3, Keypad scanning

- Did you already do this in ECE177?
- Pre-lab already posted. Very straightforward.
  Do not be lulled into complacency! Lab itself a bit tricky.
- Actual part have to put wires on a breadboard, some resistors
  Assume everyone knows how breadboards work?
- Be sure to bring in a breadboard from previous classes and jumper wire.

It might be handy to have a second breadboard (give them out?)

- Keypads – some have 3 columns, some 4. Work the same, just can't type ABCD

Keypad Scanning

- With 16 buttons, how many GPIOs do you need? By scanning only need 4+4=8
- Column pulled high to 3.3V
- First set row to 0b1111, then read out. If all 0b1111 out it means nothing pressed
- If pressed, then need to try each row one at a time to see what is pressed
- What happens if two keys pressed?

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.

Back to Assembly Language

Other math operations

- Note: can use 'S' and immediate with all of these too
- 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

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)

Shift Instructions

- Note: carry and N/Z only updated if the 'S' variant used
- LSL r1,r2 – 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 – 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 – arithmetic shift right
  sign (high bit) shifted in (preserving sign)

low bit goes into carry

- 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
- Can also shift by immediate, LSR r1,#3
- Is there an ROL? Turns out it ROL by 5 is same as ROR by (32-5)
- Is there an ASL (arithmetic shift left?) Not needed
- 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.

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

Multiply

- Often relatively slow. Lots of ways to avoid using Shift/add
- How big is your result? 32bit * 32bit has potentially 64bit result
  What happens to the high bits?
- MUL RD,RN,RM = rd=rn*rm (signed)
- UMUL RD,rn,rm = unsigned

- MLA rd,rn,rm,ra = multiply/add rd=rn*rm+ra
- MLS rd,rn,rm,ra = multiply./sub rd=rn*rm-ra
- UMULL rdlo,rdhi,rm,rn
- MULL rdlo,rdhi,rm,rn

Divide

- `SDIV RD,rn,rm` = Signed divide rd=rn/rm
- `UDIV RD,rn,rm` = Unsigned divide rd=rn/rm
- Even slower than mul. Takes a lot of space, not used often, so some chips leave it off. For example, no divide on early Raspberry Pi
- For powers of two, can right-shift
- This gives you quotient: what if you want remainder?
  - If power of two, can use and with divisor - 1: 5/4 : R = 5&(4-1)

This is just masking off the bottom bits that get shifted off.

- ◦ If have multiply instruction, R = original - (Q * divisor):
  5/4 : Q = 5 - (1*4)
- Other ways to divide? Can multiply by reciprocal. x/10 = x* (1/10). Have to set up the value and rounding right, but is often faster than dividing.

Moves

- MOV r0,r1 – moves (copies) the value in r1 into r0
- MOVN r0,r1 – moves (copies) the 1's complement inverse of r1 into r0

Thumb-2 12-bit immediates

ADD and SUB can have a real 12-bit immediate (0..4095)
Or you can have flexible immediate (ADD and SUB can do this too):

- any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word
- any constant of the form 0x00XY00XY
- any constant of the form 0xXY00XY00
- any constant of the form 0xXYXYXYXY.

```
top 4 bits 0000 -- 00000000 00000000 00000000 abcdefgh
```

```
          0001 -- 00000000 abcdefgh 00000000 abcdefgh
          0010 -- abcdefgh 00000000 abcdefgh 00000000
          0011 -- abcdefgh abcdefgh abcdefgh abcdefgh
rotate bottom 7 bits|0x80 right by top 5 bits
          01000 -- 1bcdefgh 00000000 00000000 00000000
           ...
          11111 -- 00000000 00000000 00000001 bcdefgh0
```