Jacob Branaugh, Brenn Kucey
October 21, 2013
CS 472 HW 3

(3.1) Why is the program counter a *pointer* and not a *counter*?

- If the program counter was a *counter*, the instructions would have to be sequential. Since the program counter is a *pointer*, the instructions can be located in any order since the PC simply gets the next address, enabling branching.

(3.2) Explain the function of the following registers in a CPU:

- PC (program counter): The PC holds the address of the *next* instruction to be fetched from memory
- MAR (memory address register): the MAR holds the address of the memory location that is being accessed for reading/writing during the current execute cycle
- MBR (memory buffer register): the MBR holds the data that was read from or is about to be written to the location pointed to by the address in the MAR
- IR (instruction register): the IR holds the instruction being currently executed

(3.3) For each of the following 6-bit operations, calculate the values of the C, Z, V, and N flags.

(3.10) Why does the ARM provide a reverse subtract instruction *RSB r0, r1, r2* that implements *[r0] = [r2] - [r1]* when the normal subtraction instruction *SUB r0, r2, r1* will do exactly the same job?

- Both operations only allow the last operand to be a flexible value. Flexible operands can be constants or registers with applied shifts, instead of just registers. In this example, SUB allows r1 to be flexible, while RSB allows r2 to be flexible.

(3.17) ARM instructions have a 12-bit literal. Instead of permitting a word in the range 0 to 2^{12}-1, the ARM uses an 8-bit format for the integer and a 4-bit alignment field that allows the integer to be shifted in steps of 2. What are the advantages and disadvantages of this mechanism in comparison with a straight 12-bit integer?

- The advantage is that the form is $(8-bit) \times 2^{4-bit}$ allows for numbers larger than 2^{12}-1. The disadvantage is that not all numbers between 255 and 255×2^{15} cannot be represented because not all numbers are multiples of this form.

(3.18) Write one or more ARM instructions that will clear bits 20 to 25 inclusive in register r0. All the other bits of r0 should remain unchanged.

- AND r0, r0, #0xFC0FFFFF
- ANDS r0, r0, #0xFC0FFFFF (sets status register)

(3.19) This is a classic problem of assembly language programming. Write a sequence of ARM instructions that swap the contents of registers r0 and r1 without using any additional registers or memory storage (that is, you can't move r1 to a temporary location).

- This can be done using the XOR-swap algorithm:
  - EOR r0, r0, r1
  - EOR r1, r0, r1
  - EOR r0, r0, r1

(3.25) What is the binary encoding of the following instructions?

a) STRB r1, [r2]

- 1110 01 0 0 0 1 0 0 0010 0001 000000000000

b) LDR r3, [r4, r5]!

- 1110 01 1 1 1 0 1 1 0100 0011 00000 00 0 0101

c) LDR r3, [r4], r5

- 1110 01 1 0 1 0 1 1 0100 0011 00000 00 0 0101

d) LDR r3, [r4, #-6]!

- 1110 01 0 1 0 0 1 1 0100 0011 111111111010

(3.39) Write an ARM assembly language program that scans a string terminated by the null byte 0x00 and copies the string from a source location pointed at by r0 to a destination pointed at by r1.

```
START:                  ; start of loop
LDR   r2, [r0], #1      ; load byte into r2, increment r0 pointer
CMP   r2, #0x00         ; check if byte is null
BEQ   END               ; jump to end of loop if null byte
STRB  r2, [r1], #1      ; store read byte into r1, increment r1 pointer
B     START             ; jump to start of loop
END:                    ; end of loop, continue with program
```

(3.51) Write an ARM assembly language program to determine whether a string of characters with and odd length is a palindrome under the following constraints: The string of ASIC-encoded characters is stored in memory, At the start of the program, register r1 contains the address of the first character in the string, and r2 contains the address of the last character. On exit from the program, register r0 contains a 0 if the string is not a palindrome, and 1 if it is.

```
START:                  ; start of loop
CMP    r1, r2           ; check if registers are pointing to same location
MOVEQ  r0, #1           ; load immediate 1 into r0 if same location
BEQ    END              ; jump to end of loop
CMP    [r1], [r2]       ; compare data at those locations
MOVNE  r0, #0           ; store immediate 0 to r0 if not same character
BNE    END              ; jump to end of loop
ADD    r1, r1, #1       ; increment r1 pointer
SUB    r2, r2, #1       ; decrement r2 pointer
B      START            ; jump to start of loop
END:                    ; end of loop, continue with program
```