Heather Warman and David Merrick
April 21, 2014
CS 472 HW 3

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

The program counter is a pointer because it needs to point to the next instruction to be run. This keeps track of where we are in a program and allows for easy execution of the program. If the program counter was simply a counter, then the program wouldn't know where to go next to implement the instruction.

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

a. PC

The PC contains the address of the next instruction to be executed. So it points to the location in memory that holds the next instruction.

b. MAR

The MAR (memory address register) stores the address of the location in main memory that is currently being accessed by a read or write operation.

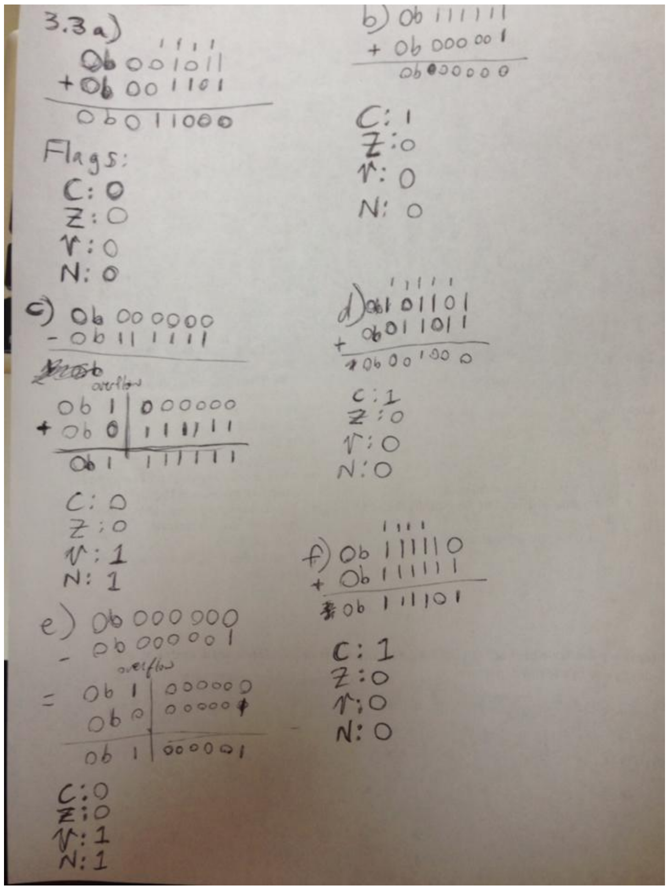
c. MBR

The MBR (memory buffer register) stores data that has just been read from main memory or data to be immediately written to main memory.

d. IR

The IR (instruction register) stores the instruction most recently read from main memory. This is the instruction currently being executed.

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



```
3.3 a)
      1111
  0b 001011
+ 0b 001101
-----------
  0b 011000
Flags:
 C: 0
 Z: 0
 V: 0
 N: 0

b) 0b 111111
 + 0b 000001
 -----------
   0b 000000
 C: 1
 Z: 0
 V: 0
 N: 0

c) 0b 000000
 - 0b 111111
 -----------
 ~~[illegible]~~
       overflow
   0b 1 | 000000
 + 0b 0 | 111111
 ---------------
   0b 1   111111
 C: 0
 Z: 0
 V: 1
 N: 1

      11111
d) 0b 101101
   0b 011011
 + ---------
  1 0b 001000
 C: 1
 Z: 0
 V: 0
 N: 0

e) 0b 000000
 - 0b 000001
        overflow
 = 0b 1 | 000000
   0b 0 | 000001
 ---------------
   0b 1 | 000001
 C: 0
 Z: 0
 V: 1
 N: 1

      1111
f) 0b 111110
 + 0b 111111
 -----------
  1 0b 111101
 C: 1
 Z: 0
 V: 0
 N: 0
```

(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?

The reason that RSB exists in ARM is that ARM does not have a negation instruction, so you could never just say NEG r1, or essentially just negate a register. With only SUB, it woud be impossible to get the negative value of a register, unless you already knew what the value held in that register was, then you could use SUB r0, r1, #(negative of r1). In most cases, we don't want to hardcode thsi value, or won't even know this value. With the addition of RSB into the instruction set, we can have #(some number) - r1, in other words, we can just subtract the value in a register from a number.

(3.17) ARM instructions have a 12-bit literal. Instead of permitting a word in the range 0 to 2Î2 - 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 to straight 12-bit integer?

The advantage of the ARM implementation of literals is that is provides range, the disadvantage is that we lose a bit of precision. This implementation is essentially how floating point numbers are stored, where we have a store values in smaller forms with an exponent to multiply it by. In doing this, we lose the precision because our 8-bit literal can only be in the range of 0 to 255 instead of having the full 12-bits to specify a literal, which would result in a range of 0 to 4095. So although we gain the ability to shift bits and extend the range of the 8-bit value, we lose 4 bits of precision.

(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.

1111 1100 0000 1111 1111 1111 1111 1111 1111

0xFC0FFFFFF

AND r0,r0,0x0xFC0FFFFFF

(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).

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 0101 1100 0010 0001 0000 0000 0000

b. LDR r3, [r4,r5] !

1110 0111 1011 0100 0011 0000 0000 0101

c. LDR r3, [r4], r5

1110 0110 1001 0100 0011 0000 0000 0101

d. LDR r3, [r4, #6] !

1110 0111 1100 0001 0010 0000 0000 0011

(3.39) Write an ARM assembly language program that scans a string from a source location pointed at by r0 to a destination pointed at by r1.

AREA Homework3, CODE, READWRITE

ENTRY

CR EQU 0x0D

Start

```
LDR r0,Source
;ADR r0,Source
;LDR r1,Destination
ADR r1,Destination
;STR r0,[r1]
Loop
LDRB r2,[r0,r3] ;load r3th byte from r0 into r2
STRB r2,[r1,r3] ;store byte from r2 in r1 at r3th position
CMP r2,#CR ;break if end of string
BEQ Complete
ADD r3,r3,#1 ;increment counter
BAL Loop ;loop again
Source
ALIGN 4
DCB ”This is a string”,CR
Destination
ALIGN 4
DCB ””
Complete
END
```

(3.51) Write an ARM assembly language program to determine whether a string of characters with an odd length is a palindrome (for example, mom) under the following constraints.

a. The string of ASIC-encoded characters is stored in memory.

b. 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 of exit from the program, register r0 contains a 0 is the string is not a palindrome, and 1 if it is.

```
AREA Lab2, CODE, READWRITE
ENTRY
CR EQU 0x0D
Start MOV r0,#0 ;Point the registers to the memory locations
MOV r1,#0 ;Make sure these are initially zero’d out
MOV r2,#0
MOV r3,#0 ;Use this as loop counter and str length
MOV r5,#0 ;Holds the current position of char to be compared
MOV r6,#0 ;Hold the length of the string
MOV r7,#0 ;Use as temp register?
ADR r4,hello
Loop ;While not last char, get another byte
LDRB r1,[r4,r3] ;Read n-th byte
CMP r1,#CR ;Compare to the string terminator
BEQ Loop_End ;If == to CR, stop
```

```
ADD r3,r3,#1 ;If =/= to CR, increment count
BAL Loop ;Loop again
Loop_End
LDR r6,[r3] ;Store length of string in r6 (value of r3, not address,
;specified with the []
SUB r3,r3,#1
CMP r6,#0 ;Compare str length to 0
BEQ Zero_Case ;Break if length of string = 0
CMP r6,#1 ;Compare str length to 1
BEQ One_Case ;Break if length of string = 1
B Div_Two
Div_Two ;Finds half the length of the string, so we know how many times to check bytes
CMP r3,#0
BEQ Load_Half ;If r3 == 0, then length of array is 0, break to Zero_Case
SUB r3,r3,#2 ;Else, subtract 2 from value of r3
ADD r7,r7,#1 ;Increment counter
BAL Div_Two
Load_Half
LDR r3,[r7] ;Load the value of r7 into r3, which is half the length of the array
;In other words, we now know how many times to loop through and compare
;bytes of string
B Cmp_Bytes
Cmp_Bytes
CMP r3,r5
BEQ One_Case ;If r3 and r5 hold the same value, then we've gone through the entire list, and
;everything matches, send to One_Case
LDRB r1,[r4,r5] ;Point r1 at the first byte of string
LDRB r2,[r4,r6] ;Point r2 at the last byte of the string
CMP r1,r2
BNE Zero_Case ;If not equivalent, exit and set r0 to 0
SUB r6,r6,#1 ;Subtract one (because we already checked the last bit)
ADD r5,r5,#1 ;Add one (because we already checked first bit)
BAL Cmp_Bytes ;Keep looping
Zero_Case ;Set r0 to 0, meaning string was not a palindrome
MOV r0,#0
One_Case ;Set r0 to 1, meaning string was a palindrome
MOV r0,#1
hello ;AREA Data, DATA, READWRITE
DCB ”MOMOM”,CR ;Assign labels for each memory location
Done
END ;Done
```