LSU EE 4720 **Homework 5** Solution **Due: XX 13 April 2018**

This assignment consists of questions on the ARM A64 (AAarch64) ISA. (Not to be confused with ARM A32, which might be called classic ARM. Older information sources that refer to ARM are probably referring to A32, which is not relevant to this assignment.)

A description of the ARM ISA is linked to the course references page, at `http://www.ece.lsu.edu/ee4720/reference.html`. Feel free to seek out introductory material as a suppliment.

ARM A64 was used in EE 4720 Spring 2017 Homework 4 and Spring 2017 Midterm Exam Problems 2 and 3. It may be useful to see those assignments for code samples, but the questions themselves are different.

Appearing on the next page is a simple C routine, `lookup`, that returns a constant from a list. The routine appears to have been written with the expectation that its call argument, `i`, would be either 0, 1, or 2. Following the C code is ARM A64 code for `lookup` as compiled by gcc version 8.

Use the course reference materials and external sources to understand the ARM code below. The course references page has a link to the ARM ISA manual which should be sufficient to answer questions in this assignment. Feel free to seek out introductory material on ARM A64 (AArch64) assembly language, but after doing so use the ARM Architecture Reference Manual to answer questions in this assignment.

Full-length versions of the code on the next page, along with other code examples can be found at `http://www.ece.lsu.edu/ee4720/2018/hw05.c.html` and `http://www.ece.lsu.edu/ee4720/2018/hw05-arm.s.html`. These include the `pi` program and a simple copy program that was a part of the decompress program used in Homeworks 1, 2, and 3.

Code on next page, problems on following pages.

```
int lookup(int i)
{
  int c[] = { 0x12345678, 0x1234, 0x1234000 };
  return c[i];
 }


@ ARM A64 Assembly Code. C code appears in comments.
lookup:
@@
@
@    CALL VALUE
@     w0: The value of i (from the C routine above).
@
@    RETURN VALUE
@     w0: The value of c[i].
@
@    Note: The size of int here is 4 bytes.

@   const int c[] = { 0x12345678, 0x1234, 0x1234000 };

        adrp    x1, .LC1

        mov     w2, 0x4000

        ldr     d0, [x1, #:lo12:.LC1]

        movk    w2, 0x123, lsl 16

        str     w2, [sp, 8]

        str     d0, [sp]

@    return c[i];

        ldr     w0, [sp, w0, sxtw 2]

        ret


        .rodata.cst8,"aM",@progbits,8
.LC1:
        .word   0x12345678
        .word   0x1234
```

Problems start on next page.

Problem 1: The ARM code above uses three kinds of register names, those starting with `d`, `w`, and `x`.

(*a*) Explain the difference between each.

The `d` registers are registers in the SIMD & floating-point register file. The `w` and `x` are in the general-purpose register file. The GPR file contains 32 64-bit registers, `r0` to `r31`. In assembly language `x0` to `x30` refer to the entire 64 bits of `r0` to `r30`, while `w0` to `w30` refer to the lower 32-bits of `r0` to `r30`. In assembly language `w31` and `x31` refer to the constant zero and `SP` refers to register `r31`. The SIMD & FP register file contains 32 128-bit registers, `v0` to `v31`. In assembly language `d0` to `d31` refer to the low 64 bits. See section B1.2.1 of the ARM V8 Architecture Reference Manual (2017).

(*b*) MIPS has general-purpose registers and four sets of co-processor registers. Indicate the name of the register set for each of the three types of ARM registers above. Hint: two are part of the same set.

See answer above.

Problem 2: The `mov` moves constant `0x4000` into register `w2`. Actually, `mov` is a pseudo instruction.

(*a*) What are pseudo instructions called in ARM?

They are called *aliases*.

(*b*) What is the real instruction that the assembler will use in this particular case?

The real instruction is `movz` with the shift amount set to zero.

(*c*) Show the encoding for this use of `mov`. Be sure to show how `w2` and `0x4000` fit into the fields.

Solution appears below. The `sf` field is set to 0 because we are using a `w` register, which is 32 bits. The opcode fields are set based on the description in the architecture manual. The `hw` field is set to zero to indicate that the immediate is not shifted. The `imm16` is the value appearing in the assembly code and `Rd` is the register number.

	sf	opc	opc2	hw	imm16	Rd
`movz`:	0	10_2	100101_2	0	4000_{16}	2
	31	30 29	28 23	22 21	20 5	4 0

Problem 3: MIPS-I does not have an instruction like `adrp`.

(*a*) Describe what the `adrp` instruction does in general.

The `adrp` instruction writes its destination register with `( PC & bitwise not 0xfff ) + ( imm << 12 )`, where `PC & bitwise not 0xfff` is the address of the `adrp` instruction with the 12 least-significant bits set to zero. See the next part for an example.

(*b*) Explain what it is doing in the code above. (It might be easier to look at the documentation for `adr` first.)

It is setting register `x1` to the address `.LC1 & bitwise not 0xfff` (address `.LC1` with the 12 least-significant bits set to zero). Instruction `ldr d0, [x1, #:lo12:.LC1]` computes its address by adding the least significant bits of `.LC1`, which is the same as `.LC1 & 0xfff`, to `x1`, the result of which is `.LC1`. The immediate value in the `adrp` instruction was set to `(.LC1 >> 12 ) - ( PC >> 12 )`.

The alert student may wonder why the compiler didn't choose an instruction that would set `x1` to exactly `.LC1`, such as `adr`. It's hard to know for sure, but one possible reason was so the same register, `x1` could be used as a base for accesses to multiple addresses, say `.LC1`, `.LC2`, etc., all of which had the same values for bits 63 to 12.

(*c*) Show MIPS code that writes the same value to its destination as `adrp`. Do not use MIPS pseudo instructions other than `la`. Assume that MIPS integer registers are 64 bits.

```
subi r3, r0, 0x1000  # Write r3 with 0xfffff000
la r2, .LC1
and r1, r2, r3    # Write r1 with low 12 bits set to zero.
```

Problem 4: The `movk` instruction is sort of an improved version of `lui`.

(*a*) Describe what the `movk` instruction does in general.

It moves a 16-bit immediate into a 16-bit section of its destination register, leaving the other 16 or 48 bits unchanged.

(*b*) Explain why a single MIPS `lui` instruction could not do what the `movk` is doing in the code above.

Because the `lui` always zeros the low 16 bits.

Problem 5: Add comments to the ARM code above that explain what the code is doing, rather than what the individual instructions do.

```
// SOLUTION

int lookup(int i) { int c[] = { 0x12345678, 0x1234, 0x1234000 }; return c[i]; }



@ ARM A64 Assembly Code. C code appears in comments.
lookup:
@@
@
@    CALL VALUE
@     w0: The value of i (from the C routine above).
@
@    RETURN VALUE
@     w0: The value of c[i].
@
@    Note: The size of int here is 4 bytes.

@   const int c[] = { 0x12345678, 0x1234, 0x1234000 };

        @
        @ Read the first two elements of the c array from memory at
        @ address .LC1, and compute the third element using immediates.
        @
        adrp    x1, .LC1            @ Load x1 with base of c original array.
        mov     w2, 0x4000          @ Compute lower 16-bits of 3rd c array elt.

        ldr     d0, [x1, #:lo12:.LC1]   @ Load 1st and 2nd elts of c array.

        movk    w2, 0x123, lsl 16   @ Compute 3rd elt, 0x1234000

        @
        @ Write a copy of the c array to memory ...
        @ ... starting at the address in register sp.
        @
        str     w2, [sp, 8]         @ Write 3rd elt to a c array copy.
        str     d0, [sp]            @ Write 1st and 2nd elts to c array copy.

@    return c[i];

        ldr   w0, [sp, w0, sxtw 2]  @ Load the i'th (w0'th) element.
        ret


        .rodata.cst8,"aM",@progbits,8
.LC1:
        .word   0x12345678          @ First element of c array
        .word   0x1234              @ Second element of c array.
                                    @ This element intentionally left blank.
```

Problem 6: The `lookup` routine was compiled using `gcc` at optimization level 3, the highest. Nevertheless, the code appears more complicated than it need to be. Explain what about the code is excessively complicated and how it could be simplified.

Because the compiler, for whatever reason, decided to construct the `c` array using the first two elements stored at `.LC1` and an instruction that wrote the third array element.

Most humans would simply put the entire `c` array at `.LC1` so that the routine would consist of just two instructions, an `adr` instruction and an `ldr` instruction.