CMU 18-447 Introduction to Computer Architecture, Spring 2014
HW 1: Instruction Set Architecture (ISA)

Instructor: Prof. Onur Mutlu
TAs: Rachata Ausavarungnirun, Varun Kohli, Xiao Bo Zhao, Paraj Tyle

Assigned: Wednesday 15th January, 2014
Due: **Wednesday 29th January, 2014**

1 The ARM Simulator [5 points]

As you work through this homework assignment and Lab 1, you may want to examine the execution of ARM programs you write with a known good reference. For this purpose, it will be helpful to learn the basic operation of the ARM simulator. The ARM simulator is described similarly to what we expected you to do in Lab 1 and is available as a binary file called `arm-sim` on the ECE Linux workstations (`/afs/ece/class/ece447/bin`). Get familiar with the simulator. There is nothing to turn in for this question, but it is up to you to learn and use the ARM simulator.

2 Big versus Little Endian Addressing [5 points]

Consider the 32-bit hexadecimal number 0x21d3ea7d.

1. What is the binary representation of this number in *little endian* format? Please clearly mark the bytes and number them from low (0) to high (3).
2. What is the binary representation of this number in *big endian* format? Please clearly mark the bytes and number them from low (0) to high (3).

3 Instruction Set Architecture (ISA) [25 points]

Your task is to compare the memory efficiency of five different styles of instruction sets for the code sequence below. The architecture styles are:

1. A zero-address machine is a stack-based machine where all operations are done using values stored on the operand stack. For this problem, you may assume that its ISA allows the following operations:
   - PUSH M - pushes the value stored at memory location M onto the operand stack.
   - POP M - pops the operand stack and stores the value into memory location M.
   - OP - Pops two values off the operand stack, performs the binary operation OP on the two values, and pushes the result back onto the operand stack.

   Note: To compute A - B with a stack machine, the following sequence of operations are necessary: PUSH A, PUSH B, SUB. After execution of SUB, A and B would no longer be on the stack, but the value A-B would be at the top of the stack.
2. A one-address machine uses an accumulator in order to perform computations. For this problem, you may assume that its ISA allows the following operations:
   - LOAD M - Loads the value stored at memory location M into the accumulator.
   - STORE M - Stores the value in the accumulator into memory location M.
   - OP M - Performs the binary operation OP on the value stored at memory location M and the value present in the accumulator. The result is stored into the accumulator (ACCUM = ACCUM OP M).
3. A two-address machine takes two sources, performs an operation on these sources and stores the result back into one of the sources. For this problem, you may assume that its ISA allows the following operation:
   - OP M1, M2 - Performs a binary operation OP on the values stored at memory locations M1 and M2 and stores the result back into memory location M1 (M1 = M1 OP M2).

4. A three-address machine, in general takes two sources, performs an operation and stores the result back into a destination different from either of the sources.

   Consider

   (a) A three-address memory-memory machine whose sources and destination are memory locations. For this problem, you may assume that its ISA allows the following operation:

   - OP M3, M1, M2 - Performs a binary operation OP on the values stored at memory locations M1 and M2 and stores the result back into memory location M3 (M3 = M1 OP M2).

   (b) A three-address load-store machine whose sources and destination are registers. Values are loaded into registers using memory operations (The ARM is an example of a three-address load-store machine). For this problem, you may assume that its ISA allows the following operations:

   - OP R3, R1, R2 - Performs a binary operation OP on the values stored at registers R1 and R2 and stores the result back into register R3 (R3 = R1 OP R2).
   - LD R1, M - Loads the value at memory location M into register R1.
   - ST R2, M - Stores the value in register R2 into memory location M.

   To measure memory efficiency, make the following assumptions about all five instruction sets:

- The opcode is always 1 byte (8 bits).
- All register operands are 1 byte (8 bits).
- All memory addresses are 2 bytes (16 bits).
- All data values are 4 bytes (32 bits).
- All instructions are an integral number of bytes in length.

There are no other optimizations to reduce memory traffic, and the variables A, B, C, and D are initially in memory.

(a) Write the code sequences for the following high-level language fragment for each of the five architecture styles. Be sure to store the contents of A, B, and D back into memory, but do not modify any other values in memory.

```
A = B + C;
B = A + B;
D = A + B;
```

(b) Calculate the instruction bytes fetched and the memory-data bytes transferred (read or written) for each of the five architecture styles.

(c) Which architecture is most efficient as measured by code size?

(d) Which architecture is most efficient as measured by total memory bandwidth required (code+data)?

4 The ARM ISA [40 points]

4.1 Warmup: Computing a Fibonacci Number [15 points]

The Fibonacci number F_n is recursively defined as

$$F(n) = F(n-1) + F(n-2),$$

where $F(1) = 1$ and $F(2) = 1$. So, $F(3) = F(2) + F(1) = 1 + 1 = 2$, and so on. Write the ARM assembly for the `fib(n)` function, which computes the Fibonacci number $F($`n`$)$:

```
int fib(int n)
{
  int a = 0;
```

```
  int b = 1;
  int c = a + b;
  while (n > 1) {
    c = a + b;
    a = b;
    b = c;
    n--;
  }
  return c;
}
```

Remember to follow ARM register usage convention (just for your reference, you may not need to use all of these registers):

- ARM uses R13 as a Stack Pointer (SP). R13 is used by the PUSH and POP instructions.
- R14 is the Link Register (LR). This register holds the address of the next instruction after a Branch and Link (BL or BLX) instruction, which is the instruction used to make a subroutine call. It is also used for return address information on entry to exception modes. At all other times, R14 can be used as a general-purpose register.
- R15 is the Program Counter (PC). It can be used in most instructions as a pointer to the instruction which is two instructions after the instruction being executed. In ARM state, all ARM instructions are four bytes long (one 32-bit word) and are always aligned on a word boundary. This means that the bottom two bits of the PC are always zero, and therefore the PC contains only 30 non-constant bits.
- The remaining 13 registers are general purpose registers.

A summary of the ARM ISA is provided at the end of this handout, and an ARM reference sheet and ARM architecture reference manual is on the wiki at `http://www.ece.cmu.edu/~ece447/s14/lib/exe/fetch.php?media=arm-instructionset.pdf`.

4.2 ARM Assembly for REP MOVSB [25 points]

Recall from lecture that ARM is a Reduced Instruction Set Computing (RISC) ISA. Complex Instruction Set Computing (CISC) ISAs—such as Intel's x86—often use one instruction to perform the function of many instructions in a RISC ISA. Here you will implement the ARM equivalent for a single Intel x86 instruction, REP MOVSB, which we will specify here.[1]

The REP MOVSB instruction uses three fixed x86 registers: ECX (count), ESI (source), and EDI (destination). The "repeat" (REP) prefix on the instruction indicates that it will repeat ECX times. Each iteration, it moves one byte from memory at address ESI to memory at address EDI, and then increments both pointers by one. Thus, the instruction copies ECX bytes from address ESI to address EDI.

(a) Write the corresponding assembly code in ARM ISA that accomplishes the same function as this instruction. You can use any general purpose register. Indicate which ARM registers you have chosen to correspond to the x86 registers used by REP MOVSB. Try to minimize code size as much as possible.

(b) What is the size of the ARM assembly code you wrote in (a), in bytes? How does it compare to REP MOVSB in x86 (note: REP MOVSB occupies 2 bytes)?

(c) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0xcafebeef
```

[1]The REP MOVSB instruction is actually more complex than what we describe. For those who are interested, the Intel architecture manual (found on the wiki at `http://www.ece.cmu.edu/~ece447/s14/doku.php?id=techdocs`) describes the MOVSB instruction on page 1327 and the REP prefix on page 1682. We are assuming a 32-bit protected mode environment with flat addressing (no segmentation), and a direction flag set to zero. You do not need to worry about these details to complete the homework problem.

```
EDX: 0xfeed4444
ESI: 0xdecaffff
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, consider the ARM assembly code you wrote in (a). How many total instructions will be executed by your code to accomplish the same fuction as the single REP MOVSB in x86 accomplishes for the given register state?

(d) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

```
EAX: 0xccccaaaa
EBP: 0x00002222
ECX: 0x00000000
EDX: 0xfeed4444
ESI: 0xdecabeef
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
```

Now, answer the same question in (c) for the above register values.

5 Data Flow Programs [15 points]

Draw the data flow graph for the `fib(n)` function from Question 4.1. You may use the following data flow nodes in your graph:

- + (addition)
- > (left operand is greater than right operand)
- `Copy` (copy the value on the input to both outputs)
- `BR` (branch, with the semantics discussed in class, label the True and False outputs)

Clearly label all the nodes, program inputs, and program outputs. Try to the use fewest number of data flow nodes possible.

6 Performance Metrics [10 points]

- If a given program runs on a processor with a higher frequency, does it imply that the processor always executes more instructions per second (compared to a processor with a lower frequency)? (Use less than 10 words.)
- If a processor executes more of a given program's instructions per second, does it imply that the processor always finishes the program faster (compared to a processor that executes fewer instructions per second)? (Use less than 10 words.)

7 Performance Evaluation [15 points]

Your job is to evaluate the potential performance of two processors, each implementing a different ISA. The evaluation is based on its performance on a particular benchmark. On the processor implementing ISA A, the best compiled code for this benchmark performs at the rate of 6 IPC. That processor has a 400 MHz clock. On the processor implementing ISA B, the best compiled code for this benchmark performs at the rate of 2 IPC. That processor has a 800 MHz clock.

- What is the performance in Millions of Instructions per Second (MIPS) of the processor implementing ISA A?
- What is the performance in MIPS of the processor implementing ISA B?
- Which is the higher performance processor: A B Don't know

  Briefly explain your answer.

8 Fixed Length vs. Variable Length Instructions [15 points]

Table below shows a subset of MIPS instruction set, which we will call this a mini-MIPS. In this problem, we will examine the tradeoffs between a fixed length and variable length instruction set.

In a traditional MIPS ISA, each instruction is 32-bit long. Assuming that a register field is 3 bits and an immediate field is 8 bits.

- What is the smallest code size for a program that contains 5 ADD instructions, 10 SUB instructions, 2 MULT instructions, 3 BEQ instructions using the mini-MIPS ISA and variable length instructions? Please explain how you encode ADD, SUB, MULT and BEQ instructions.
- What is the difference in terms of code size comparing to a traditional MIPS ISA?

  Briefly explain your answer.

Opcode	Example Assembly	Semantics
add	`add $1, $2, $3`	`$1 = $2 + $3`
sub	`sub $1, $2, $3`	`$1 = $2 - $3`
multiply	`mult $2, $3`	`hi, lo = $2 * $3`
divide	`div $2, $3`	`lo = $2/$3, hi = $2 mod $3`
load word	`lw $1, 100($2)`	`$1 = memory[$2 + 100]`
store word	`sw $1, 100($2)`	`memory[$2 + 100] = $1`
branch on equal	`beq $1, $2, label`	`if ($1 == $2) goto label`
jump	`j label`	`goto label`

ARM Instruction Summary

ARM Instruction Summary is available on page 3-4 of the ARM Instruction Manual at `http://www.ece.cmu.edu/~ece447/s14/lib/exe/fetch.php?media=arm-instructionset.pdf`.