ECE 486/586
Spring 2018
Project 1

This project is a little unusual in that we'll discuss and explore design alternatives, decisions, and approaches together, with the implementation to be done outside of class. In this project you'll get more in-depth experience with issues like instruction set architecture and encoding, floating point, and data representation. But the more crucial objective is to get experience in thinking as a designer.

One of the challenges facing systems designers is how to implement new functionality into existing architectures and systems. For example, Intel has extended the instruction set architecture of the original x86 numerous times, adding hundreds of additional instructions as well as adding to the number and extending the size of the registers while maintaining binary compatibility.

In this assignment we'll consider how to extend an older architecture to implement additional instructions and data types.

First we need to acquaint ourselves with the PDP-8 architecture, particularly instruction encoding and subroutine linkage.

The PDP-8 does not have floating point instructions or dedicated floating point hardware. We will explore three alternatives for adding floating point operations, two software based, and the other hardware based.

To prepare for in-class discussions, you should

- Be familiar with PDP-8 architecture and assembly language
    - Attend class for PDP-8 lecture material
    - Review the lecture slides on PDP-8
    - Review the PDP-8 related documentation the Resources page of the course web site
    - Pay particular attention to subroutine linkage
    - Download the DOS-compatible version or use the Linux version of the PDP-8 assembler to assemble the sample source code and examine the resulting output files
- Be familiar with IEEE Floating Point and floating point algorithms
    - Attend class for IEEE 754 lecture material
    - Review the lectures slides on IEEE 754 floating point
- Download and review the code for my PDP-8 simulator written in Verilog

Assembler and object code format

I've modified a PDP-8 assembler to produce object code output in the format used by Verilog's $readmemh() function. You can download a DOS executable from the course web site (where you can also find an example assembly language program add01.as), or use a Linux version in my home bin directory on MCECS.

To use the Linux version just type

```
~faustm/bin/pal -v <input filename>
```

For example:

```
~faustm/bin/pal -v add01.as
```

The assembler will also produce a listing file that shows the source code and an octal representation of the assembled code along with addresses. This can be very helpful in understanding what's happening.

In this assignment you are to implement the following floating point instructions:

- FPCLAC (clear floating point accumulator)
  - Clear the internal floating point accumulator
  - Do we need this?
- FPLOAD (floating point load)
  - Load the internal floating point accumulator with the floating point value which address follows the FPLOAD instruction
- FPSTOR (floating point store)
  - Store the contents of the internal floating point accumulator to the address which follows the FPSTOR instruction
- FPADD (floating point add)
  - Add to the floating point accumulator the floating point operand which address follows the FPADD instruction
- FPMULT (floating point multiply)
  - Multiple the floating point accumulator by the floating point operand which address follows the FPMULT instruction

You'll add support for these new instructions into an existing PDP-8 model written in Verilog. You can use Verilog integer addition, subtraction, and multiplication as well as logical and bit manipulation operators. You may not use Verilog floating point operators (except to verify your solution).

Floating Point Number Representation

Before we begin to look at alternative implementations we need to discuss representation.

- What will the format of the operands be?
  - PDP-8 uses 12-bit words, IEEE 754 uses 32-bit words, so can't be 100% IEEE 754 compliant
  - Three consecutive words (which bits unused?)
    - Could make 3*12 – 32 = 4 consecutive bits in a word unused
    - Make things simpler by storing 23-bit significant and sign bit in 24 bits of two words, 8-bit exponent field in least significant 8 bits of remaining 12-bit word.
  - Will address be of the least or most significant word (little or big-endian)?
    - PDP-8 package used big-endian

Software

We could just create subroutines for each floating point operation we want to support (e.g. add, subtract, multiply, divide, sqrt). However, performing even a simple evaluation of a floating point expression (e.g. evaluating a polynomial like $Y = AX^3 + BX^2 + CX + D$) would take many subroutine calls which generally have high overhead costs.

Consider for example, the polynomial

$$Y = AX^3 + BX^2 + CX + D$$

which could be evaluated as Y = X * (X * (A * X + B) + C) + D. The PDP-8 assembly code might look something like the following. Recall that arguments to subroutines appear immediately following the subroutine call instruction. Because we want to specify a destination, we need to pass a pointer to the destination location.

[Note: these only allocate a single word for each operand. How do we allocate three? Could do:

```
A,    0
      0
      0
B,    0
      0
      0
```

Some assemblers permit

```
A,    ZBLOCK 3
```

or something similar to reserve three words of storage beginning at location A.]

```
A,          0
B,          0
C,          0
D,          0
X,          0
TMP,        0
TMPPTR,     TMP
      .
      .
      .
            JMS I FPMULT
            A
            X
            TMPPTR
            JMS I FPADD
            TMP
            B
            TMPPTR
      .
      .
      .
```

This requires specifying and passing three arguments for each function.
We could instead assume the existence of a floating point accumulator (which needn't be a specialized hardware accumulator, but could just be specific memory locations used for this purpose by the package). This simplifies code (and follows the PDP-8 model) by allowing the destination and one source operand to be the floating point accumulator, eliminating the need to specify them/pass them to subroutines. It does require something like LOAD and STORE instructions (similar to PDP-8 CLA; TAD and DCA instructions) that operate on the floating point accumulator. Let's call those FPLOAD and FPSTORE.

```
      JMS I FPLOAD     / load the floating point (FP) AC with A
      A
      JMS I  FPMULT    / multiply the FP AC by X
      X
      JMS I FPADD      / add B to the FP AC
      B
      JMS I FPMULT     / multiply the FP AC by X
      X
      JMS I FPADD      / add C to the FP AC
      C
      JMS I FPMULT     / multiply the FP AC by X
```

```
X
JMS I FPADD     / add D to the FP AC
D
JMS I FPSTORE   / store the FP AC to Y
Y
```

Floating point values require three words on the PDP-8. Should we copy three words for each operand? Why not always use a pointer instead?

A more efficient approach might be to create a "floating point system" – a software system that interprets floating point operations. A single subroutine call enters the system which then interprets special floating point operations until the system is exited (by executing an exit operation).

```
......
JMS  I  7      /ENTER FLOATING PACKAGE
FGET  A        /LOAD PSEUDO AC
FMPY  X        /MULTIPLY
FADD  B        /ADD
FMPY  X        /MULTIPLY
FADD  C        /ADD
FMPY  X        /MULTIPLY
FADD  D        /ADD
FPUT  Y        /STORE
FEXT           /EXIT FLOATING PACKAGE
......
```

Where memory location 7 contains a pointer to the floating point package.

ARITHMETIC

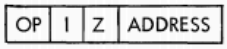
Since floating-point numbers are stored in a three-register format, the floating-point system uses a "psuedo" floating accumulator (FAC) which consists of three registers in the floating-point package: 44, 45, and 46. Register 44 contains the exponent; 45 and 46 contain the high and low order parts of the mantissa, respectively.

BASIC FLOATING-POINT COMMANDS

The basic floating-point commands include the following:

load floating accumulator
store floating accumulator
add to floating accumulator
subtract from floating accumulator
multiply by floating accumulator
divide into floating accumulator
normalize floating accumulator

All arithmetic operations are called through an interpreter. The command codes have a format that is almost identical to the format of the memory reference instruction, namely:



where the op code is from 000 to 111 or $0_8 - 7_8$.

Hardware

First issues:

- What operations should we support?
  - Easiest to restrict ourselves to ADD, SUB, MULT, DIV for now
  - But approach should permit additional operations in the future
  - Should be (mostly) IEEE 754 compliant – range/behavior, fields
- Where will the operands be?
  - PDP-8 is an accumulator based architecture – might want to follow
  - But existing 12-bit accumulator can't accommodate 32-bit IEEE 754 floating point representation
  - Why not just make accumulator 32-bits?
    - Need to avoid breaking existing code (consider shifts, Link, etc)
  - Why not create (at least) one floating point accumulator?
    - Just like rest of PDP-8, operations will be on the accumulator and memory
    - But how do we get things into (or out of) the accumulator?
      - We'll need additional instructions to move operands between accumulator and memory
      - Unlike other instructions that operate on single word operands, these must deal with multiple word operands (32-bits)
      - How would software have done this?
        - Reserve regular memory location(s) to act as floating point accumulator

Harder issues:

- How will we represent/encode new instructions?
  - The PDP-8 has limited opcode space
  - Let's use IOT instructions
  - A little help from the assembler to define new "instructions"
  - Use device code 55_8 for floating point instructions
  - Use remaining three "opcode" bits to specify the floating point operation
    - FPCLAC (opcode 0)
    - FPLOAD (opcode 1)
    - FPSTOR (opcode 2)
    - FPADD (opcode 3)
    - FPMULT (opcode 4)

Discuss with your teams – we'll tackle these in class on Tuesday, May 1st:

- How will we form floating point literals in assembler?

- How will we test it?