ECE 486/586
Winter 2015
Project
Instruction Set Architecture Simulator

Introduction

Simulation has a number of uses, but simulation at the instruction set architecture level is particularly useful for computer architects and designers. Without needing to simulate at a deeper machine organization or register-transfer level, you can explore the impact of instruction set architecture changes on CPI, look at the instruction counts for various program mixes (or benchmarks), and generate trace files useful for analyzing and comparing results of decisions in branch predictor algorithms and cache designs.

For this project, you are to write an instruction set architecture (ISA) level simulator for the PDP-8 minicomputer capable of generating memory trace files like those used in the cache simulations in ECE 485/585. You can write your simulator in Verilog, C, C++, or Java. If you want to use a language not on the list, check with me first.

A link to a brief overview of the PDP-8 architecture can be found on the course web site along with a document containing more detailed information on instruction formats and addressing modes. Despite its age the PDP-8 makes a good example because of its simplicity. It's easy to learn the instruction set architecture in a few hours or less, and its entire description can fit on a couple of pages.

Assembler and object code format

Your simulator should take as input either an ASCII or binary object code file. These files are produced by the PAL, the PDP-8 assembler which you can download from the course website where you can also find a sample assembly language program add01.as. The program runs in a DOS window and has a simple command line interface. Just type

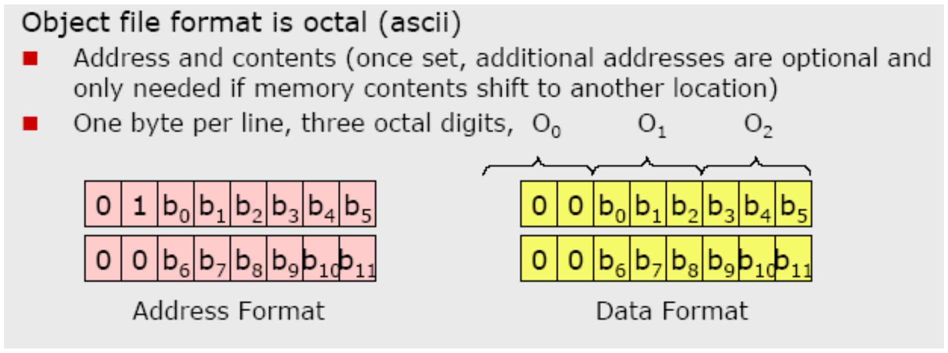
```
pal -<flag> <input filename>
```

If you omit a flag it generates a binary (.bin) object file. Other options are

-v	ASCII hexadecimal file in format readable by Verilog $readmemh()
-o	ASCII octal format as described further below

For example:

```
pal -v add01.as
```


Object file format is octal (ascii)

- Address and contents (once set, additional addresses are optional and only needed if memory contents shift to another location)
- One byte per line, three octal digits, O_0 O_1 O_2

0 1 b_0 b_1 b_2 b_3 b_4 b_5
0 0 b_6 b_7 b_8 b_9 b_{10} b_{11}
Address Format
0 0 b_0 b_1 b_2 b_3 b_4 b_5
0 0 b_6 b_7 b_8 b_9 b_{10} b_{11}
Data Format

If an address format is specified then the two lines are interpreted as a 12-bit address (note that the PDP-8 uses b0 to refer to the MSB) and subsequent data is loaded beginning at this address.

Functionality
Your simulator must correctly simulate the entire PDP-8 instruction set with the exception of I/O instructions and EAE options (group 3 microinstructions). You can treat these as NOPs and print a warning if you encounter them. Otherwise, the simulator must be "clock accurate". The number of simulated clock cycles for each instruction should be as follows:

Mnemonic	OpCode	Cycles
AND	0	2
TAD	1	2
ISZ	2	2
DCA	3	2
JMS	4	2
JMP	5	1
<IO>	6	0
μinstructions	7	1

Note that indirect addressing requires an additional cycle beyond those in the table above. Auto increment requires two additional cycles beyond those in the table above.

After loading the memory image you can either assume a start address of 200 (octal), or assume that the start address is the first address in the object file. At the end of execution, your simulator should generate a brief summary which includes:

- The total number of instructions executed
- The total number of clock cycles consumed
- The number of times each instruction type (by mnemonic) was executed

Unless you're simulating I/O instructions for extra credit, you should include them in the counts of instructions but assume they consume 0 cycles.

In addition, you must generate a trace file of all memory accesses (instructions and data) in order. Be particularly careful when dealing with addressing modes. While you do not need to do a simulation at the microarchitecture level, assume a non-pipelined implementation in which one instruction completes before another is fetched.

The format of the trace file will be

```
<type>  <address>
```

Where <type> of 0 indicates a data read, 1 indicates a data write, and 2 indicates an instruction fetch. The address should be displayed in octal. Do not display the data being read/written. The following example shows an instruction fetch from location 200_8 followed by a data read from location 167_8 and a data write back to that same location.

```
2    200
0    167
1    167
```

Your report must include the complete source code and the result of running your emulator on several object files that I will provide. Be sure to also include in your report a brief discussion of the design and implementation of your simulator as well as your testing procedures and results.

Grading
Your project will be graded as follows:

50%	Correctness (generates complete and correct output on all test cases)
25%	Code quality (organized, readable, maintainable, robust)
15%	Testing (sensible test strategy articulated and executed)
10%	Project report (concise summary of design and testing, presentation of results)

You can also earn extra credit (up to an additional 20%) as follows:

- Correctly implement I/O instructions, at least the ability to read the keyboard and write to the display.

- Create a graphical user interface that allows the user to observe the simulated execution (e.g. by tracing value of the L, AC, PC registers) and to interrupt the execution to display memory contents. You can go further and permit breakpoints, single-stepping, interactive assembly, etc.

- Create a branch trace file which contains the PC for each branch you encountered, the type of branch, the target address, and whether the branch was taken. Note that subroutine calls and returns are branches.

Plagiarism
Plagiarism on this assignment will not be tolerated. All code must be your own. You may make use of other tools to validate your assumptions about the PDP-8 architecture but you may not incorporate any outside code in your project. "Borrowed" code will result in a zero for the assignment.

Suggestions
- Start early
- Consider
  - Get the basic memory loading working
  - Implement memory read/write macros or functions along with trace file creation
  - Get the fetch/decode/execute framework working
  - Then work on the effective address calculation
  - Then implement each instruction type
- Have a written test strategy
- Test each aspect of the ISA (opcode, addressing modes, etc)
- Include useful debugging information
  - Consider debugging modes/levels
  - Useful, but not too verbose