ECE 486/586
Computer Architecture
Spring 2017

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-11/20 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.

The PDP-11/20 is one of a family of popular minicomputers from decades past. Despite its age, the PDP-11 architecture makes a good example because of its simplicity and regularity. For background on the PDP-11 architecture, please consult the following two Wikipedia articles:
http://en.wikipedia.org/wiki/PDP-11
http://en.wikipedia.org/wiki/PDP-11_architecture

For the definitive documentation, see the PDP-11/20 processor handbook (widely available on the web). Here's one:
http://research.microsoft.com/en-us/um/people/gbell/Digital/PDP%2011%20Handbook%201969.pdf

Some of these systems are still in use:
http://www.theregister.co.uk/2013/06/19/nuke_plants_to_keep_pdp11_until_2050/

PDP-11 Highlights

The PDP-11 was a 16-bit machine. All tools generally used octal for describing memory addresses and contents. Word references (and program counter values) are word aligned. You must support all addressing modes on all registers. The upper 4K words of memory space is used for I/O registers. The addresses from 0 to 400 (octal) are used for PCs and PSWs of interrupt service routines.

Note that your simulation of the PDP-11/20 will not require simulation of memory management or floating point operations. For this project you will also not need to simulate I/O (the PDP-11 used exclusively memory mapped I/O so it really wouldn't affect your simulation of the processor). You must accommodate a memory as large as 32K words. Moreover, many instructions were design to

have two variants: one that operates on words and another on bytes. You do not need to implement the versions that operate on byte operands.

Software

To test your simulator you'll write program fragments in PDP-11 assembly language and use the Macro-11 assembler to assemble them. Documentation for the Macro-11 assembler can be found at:

http://bitsavers.trailing-edge.com/pdf/dec/pdp11/rsx11/RSX11M_V2/DEC-11-OIMRA-A-D_MACRO_75.pdf

I've written a translator to convert these object files to an easily parsed ASCII representation your simulator can read.

Executable versions of the Macro-11 assembler and my translator can be found in my bin directory (~faustm/bin/macro11 and ~faustm/bin/obj2ascii) on the MCECS Linux systems (linux.cecs.pdx.edu).

Invoking macro11 without arguments will give you help. You'll most likely want to use a command line like:

```
% macro11 source.mac –o source.obj –l source.lst
```

where source.mac is your assembly language input file, source.obj is the object file created and source.lst is the list file created.

To convert the code from the object file to ASCII use

```
% obj2ascii source.obj source.ascii
```

Your simulator will then read the source.ascii memory image file.

Note that because your code will not be simulated along with an operating system, you cannot make system calls (e.g. PRINT).

Format of Code File

The program obj2ascii will read an object file produced by Macro-11 and generate an ASCII file containing a series of lines, each representing a 16-bit octal value. Each line begins with either a @, – , or * character. Lines beginning with an - indicate an octal value to be loaded at the current load address. The starting load address defaults to 0 and is incremented by two (one word) after each load. Any line that begins with an @ indicates a change to the current load address. There will be at most one line beginning with * to indicate the starting address of the program. Your simulator should use this if present, or allow the user to otherwise input the starting address (e.g. command line argument or prompt).

Simulator Output
Your simulator should produce the following outputs:

- A memory trace file like that used in ECE 485/585 showing each instruction fetch and memory data read and write
- A summary, displayed on the screen upon completion of the simulation (execution of a HALT instruction) indicating the total number of instructions executed
- An optional display that (at every instruction fetch) succinctly indicates the contents of each register, the contents of the NZVC flags, and the currently fetched instruction

Trace file format
The format of the trace file should 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
```

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 the byte variants of instructions that have them.

- Create a graphical user interface that allows the user to observe the simulated execution (e.g. by tracing value of the 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 that 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-11 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