Erik Laucks Project 1

CS203

Professor Pfaffman

23 October 2017

Table of Contents

ISA Design

Instruction Types

There are 5 instruction types in my Littlefinger ISA. These are similar to the instruction types in LEGv8, but are slightly modified. I decided to keep the R type instruction because I liked the format, but rearranged the instruction to make it more intuitive to use. I also recycled the I type instruction, but again rearranged to make it more intuitive. I changed the name of the D type instruction to M type, for Memory, and rearranged the structure to make it more intuitive. I combined the two branching instruction types in LEGv8 into just the B branch type. There is a space for a register to account for CBZ, CBNZ, BR, and BL, while keeping a large address field to allow for B and B.cond to use a large address space. Lastly, there is the O type, for other, that holds extraneous instructions like HALT, NOP, PUSH, and POP. The following is a diagram of the breakdown of the instruction types:

```
R - register:
0 1 2 3 4 5 | 6 7 8 9 10 | 11 12 13 14 15 | 16 17 18 19 20 | 21 22 23 24 25 26 27 28 29 30 31
   opcode   |    first   |     second     |     target     |              shift

I - immediate:
0 1 2 3 4 5 | 6 7 8 9 10 | 11 12 13 14 15 | 16 17 18 19 20 | 21 22 23 24 25 26 27 28 29 30 31
   opcode   |    first   |     target     |                   immediate

M - memory:
0 1 2 3 4 5 | 6 7 8 9 10 | 11 12 13 14 15 | 16 17 18 19 20 | 21 22 23 24 25 26 27 28 29 30 31
   opcode   |    first   |     second     |                     offset

B - branch:
0 1 2 3 4 5 | 6 7 8 9 10 | 11 12 13 14 15 | 16 17 18 19 20 | 21 22 23 24 25 26 27 28 29 30 31
   opcode   |    first   |                     address

O - other:
0 1 2 3 4 5 | 6 7 8 9 10 | 11 12 13 14 15 | 16 17 18 19 20 | 21 22 23 24 25 26 27 28 29 30 31
   opcode   |    value   |
```

Instruction List

All of the following instructions have been implemented in my ISA: the Arthmetic and Logical operations from textbook Figure 2.1, the first eight data transfer instructions from Figure 2.1, both the conditional and unconditional branching from Figure 2.1, the stack operators PUSH and POP, a HALT command that will cause the simulator to stop processing and print the state of the machine to the command-line, and a NOP command that will have the operation code value of all zeros and indicates that the machine will perform no operation other than loading the next operation. I redesigned all of the opcodes for all of the instructions in 6 bits. First, I broke the instructions up into 5 categories: Arithmetic, Logical, Memory, Branching, and Other. I assigned a unique 3 bit code to each category: 001, 010, 011, 100, and 000, respectively. Each category has a maximum of 8 instructions in it, so each of the instructions were indexed with 3 bits within their category. Then, by combining the category code with the instruction code, I got a 6 bit opcode. For example, ADD is 001 (Arithmetic category) and 000 (First instruction in category) to get 001000. The following tables lists all of the instructions in the ISA, their opcodes, types, and a brief description:

Mnemonic	Type	Opcode	Description
NOP	O	000000	Perform no operation
HALT	O	000001	Halt the execution

PUSH	O	000010	Push to stack
POP	O	000011	Pop from stack
ADD	R	001000	Add
SUB	R	001001	Subtract
ADDI	I	001010	Add immediate
SUBI	I	001011	Subtract immediate
ADDS	R	001110	Add & set flags
SUBS	R	001101	Subtract & set flags
ADDIS	I	001110	Add immediate & set flags
SUBIS	I	001111	Subtract immediate & set flags
AND	R	010000	Bitwise and
IOR	R	010001	Bitwise inclusive or
EOR	R	010010	Bitwise exclusive or
ANDI	I	010011	Bitwise and immediate
IORI	I	010100	Bitwise inclusive or immediate
EORI	I	010101	Bitwise exclusive or immediate
LSL	I	010110	Logical shift left
LSR	I	010111	Logical shift right
LDUR	M	011000	Load register
STUR	M	011001	Store register
LDURW	M	011010	Load word
STURW	M	011011	Store word
LDURH	M	011100	Load half
STURH	M	011101	Store half

LDURB	M	011110	Load byte
STURB	M	011111	Store byte
CBZ	B	100000	Compare & branch on = 0
CBNZ	B	100001	Compare & branch on != 0
B.EQ	B	100010	Branch equal
B.LT	B	100011	Branch less than
B.GT	B	100100	Branch greater than
B	B	100101	Branch
BR	B	100110	Branch to register
BL	B	100111	Branch with link

Semantic Rules

When designing this Little Finger Assembly, certain design decisions were made to make it easier to parse the code. The following are the syntax and semantic specifications for the ISA:

-Only one instruction may be placed on a line

-A label must be on its own line

-A directive must be on its own line

-Semicolons (;) indicate a comment, and anything after one on a line is ignored

-All instructions are 32 bits

-All register values must be prefaced with a lowercase x (ex. x10)

-All immediate values must be in decimal

-Commas must be used to separate parameters in instructions (ex. ADD x0, x1, x1)

-Wordsize and register count must be specified in decimal

-Max memory must be specified in hex

-There must be a stack label somewhere in the code

-Labels can only be used for branching and stack, not memory operations (like LDUR)

Assembler

Directives

There are several directives that are implement in my Little Finger Assembly. Certain directives must be placed at the beginning of a file because they specify important information about the CPU. Others may be placed anywhere. Some, like data allocation directives, should be at the end of the file, after all instructions. If these are written before the HALT, then the CPU will try to read them as instructions. The following table lists all of the directives, where they may occur, and what they do:

Directive	Placement	Description
.wordsize	Top of file	Set wordsize of CPU
.regcnt	Top of file	Set number of registers in CPU
.maxmem	Top of file	Set memory size

.pos	Anywhere	Move write head to location
.align	Anywhere	Align all subsequent data to boundary
.double	End of file	Allocate space for doubleword
.single	End of file	Allocate space for word
.half	End of file	Allocate space for halfword
.byte	End of file	Allocate space for single byte

Labels

Labels are a feature of an ISA that effectively serve as pointers to memory locations. By placing a label in code, one can refer back to it when branching. This logic can be used to create loops and functions. Labels are also used to set the stack location in memory. Labels undergird key features of the Assembly language and become the gateway for higher level logical structures in code.

Formatting

In this assembler, the .o file can be written in either binary or hex, depending on the input to the command line. This data is written 4 bytes to a line and formatted to be easily readable.

Branching

Branching is a key feature of my Little Finger Assembly. Branching allows control over the program counter and gives the programmer a lot more access to implement logical functionality. Using labels and conditional branching, for loops and while loops can easily be implemented. Using the branch with link command, functions and function stacks can also be implemented. Push and pop can also be combined with this to allow powerful stack management in function calls and allow for instance variables to be passed between functions.

Simulator

Registers

Registers have been fully implemented in the simulator. When the simulator starts, all of the registers are initialized to zero (fulfilling requirement 1 from the project specification on the machine simulator). The number of registers is set by the .regcnt directive and the size of the registers is set by the .wordsize directive, both passed in on the first line of the .o file (fulfilling requirement 2). The values in the registers are stored as signed values, and are represented as character arrays of 1's and 0's and all operations on registers are performed on these character strings (fulfilling requirements 14 and 16). Lastly, instructions are stored in the instruction register before execution

and all processing on instructions is done from the instruction register (fulfilling requirement 15).

Memory

Memory in my simulator is implemented as an array of Java Byte objects (fulfilling requirement 13). The memory is byte addressable and is of the size of the .maxmem directive from the .o file (fulfilling requirement 3). When the program is loaded into the simulator memory, it is read character by character after the first line into the Byte objects starting at 0x0 and counting up, read as binary or hex as specified in the first line of the file (fulfilling requirement 4).

Processing

Processing in the simulator is done according to the fetch-execute cycle. A new instruction is fetched and is decoded and loaded into the instruction register and then is executed by the processor and the state changes are updated in the simulator (fulfilling requirement 5). If the noisy mode is set in the command line, output will be sent to the command line every time a register is changed and a new instruction is loaded (fulfilling requirement 7). The processor will execute commands from the entirety of the ISA defined above (fulfilling requirement 12). A data stack has been fully implemented using the PUSH and POP commands and empty memory has the ability to be used as a heap (fulfilling requirement 6). After execution finishes, the output is written to external files in

a nicely formatted report of the processor and memory states (fulfilling requirement 8). Additionally, the four CPU flags (Z, N, C, and V) are set after every arithmetic operation and are fully utilized in conditional branching commands in the ISA.

GUI

The simulator runs using a graphical user interface created using Java AWT. There are two windows present: a processor window that displays the registers and their values, the program counter, the current instruction (which isn't executed yet), and the CPU flags, along with a memory window that shows the current memory state (fulfilling requirement 9). The processor window will also have buttons that allow the user to step one instruction, continuously execute instructions at 3 different speeds, stop the execution, reset the simulator from the original image file, and write the current memory and processor state to separate files (fulfilling requirement 10). The memory window displays memory addresses and values from 0x0 down, and bytes are grouped by the wordsize (fulfilling requirement 11).

Visualizer

The visualizer is a separate tool that is subset of the processor simulator. There is a GUI that displays the memory in groups of bytes from 0x0 down. There are inputs to adjust the range of memory to show in the visualizer and the memory can be displayed

in hex or binary through the command line.

Project Organization

The directory structure of the project is organized to be very easy to navigate. There are separate directories for each of the tools: Assembler, Visualizer, and Simulator. These folders contain the code that is specific to each of those tools. There are also the Instructions and Operations directories. These contain classes and methods that are used by multiple tools and are thus in separate packages to improve code reuse. The data folder contains the testing Assembly files as well as the assembled memory images and the data dumps from the simulator GUI. The doc folder contains the entirety of the javadoc for the project. The compiled folder contains all of the compiled java classes and is where the project is executed from.

Test Programs

Test.as

This assembly file is provided to test 24 of the instructions in the ISA. This file implements all of the arithmetic, logical, and memory operations and can be tested

using the simulator GUI. All of the directives are also being tested here, including .pos to create space and the data allocation directives, which can be seen in the visualizer.

Branching.as

This assembly file tests the rest of the instruction set. It implements the branching operations, which can be tested through the simulator GUI by watching the instructions that are executed and seeing if the SUBIS on line 13 is skipped. The B command on line 12 can be changed to any of the branching commands to demonstrate this functionality. The stack is also tested in this file, with the stack label setting the stack pointer and push and pop being used to control the data stack.

User Manual

Compilation

To compile the project, simply untar the provided file, cd into the provided directory, and execute the make command. This will build all of the class files into the compiled/ directory and will build all of the javadoc to the doc/ directory.

Execution

To execute the program, use one of the provided make commands, which are as follows:

Make command	Description
make assemble1	Assemble test.as
make assemble2	Assemble branching.as
make visualize1	Open visualizer for test.o
make visualize2	Open visualizer for branching.o
make simulate1	Open simulator for test.o
make simulate2	Open simulator for branching.o
make simulate1noisy	Open simulator in noisy mode for test.o
make simulate2noisy	Open simulator in noisy mode for branching.o

If you would like to execute custom commands, here are the structures for executing each tool from the compiled/ directory:

-Assembler: java Assembler.Assembler <file>

-Visualizer: java Visualizer.Visualizer <file> <hex: type “hex” for hex, anything else or nothing for binary>

-Simulator: java Simulator.Simulator <file> <noisy: type “noisy” for noisy mode, anything else or nothing for quiet mode>

Bibliography

Pfaffmann, Jeffrey O. “Project 1 Description.” Jeffrey O Pfaffmann PhD - Homepage, Lafayette College, 139.147.9.182/~pfaffmaj/courses/f17/cs203/lectures/w4_d1/.