Add back to reading list bin

note @76 130 views

Actions

Programming Assignment 2

Due Friday, April 28 @ 10pm

Updated Fri Mar 31 11:05:05 CDT 2023 to include the note about bit manipulations.

You may work in teams of 1 to 2 on this assignment. EXACTLY ONE MEMBER OF YOUR TEAM SHOULD SUBMIT THE ASSIGNMENT IN CANVAS! THE TAs WILL ASSESS A PENALTY IF MORE THAN ONE MEMBER OF YOUR GROUP SUBMITS THE ASSIGNMENT, AS THIS CREATES A SIGNIFICANT OVERHEAD FOR THEM WHILE GRADING. Your submission should include a file, "authors.txt" giving the names and NetIDs of each member of your team.

For this assignment, you will be implementing a disassembler for the binaries that run on our LEGv8 emulator in binary mode. Your disassembler will handle input files containing any number of contiguous, binary LEGv8 instructions encoded in big-endian byte order. The input file name will be given as the first command line parameter. Your output, printed to the terminal, should be--modulo some caveats discussed below--the original LEGv8 assembly code that generated the binary.

Except that it ignores the PC and flow control, a disassembler essentially implements the first two stages (fetch and decode) of the five-stage pipeline described in lecture and the textbook. A working disassembler requires roughly half of the total work of building a binary emulator.

Your disassembler should fully support the following set of LEGv8 instructions:

- ADD
- ADDI
- AND
- ANDI
- B
- B.cond: This is a CB instruction in which the Rt field is not a register, but a code that indicates the condition extension. These have the values (base 16):
  - 0: EQ
  - 1: NE
  - 2: HS
  - 3: LO
  - 4: MI
  - 5: PL
  - 6: VS
  - 7: VC
  - 8: HI
  - 9: LS
  - a: GE
  - b: LT
  - c: GT
  - d: LE
- BL

- BR: The branch target is encoded in the Rn field.
- CBNZ
- CBZ
- EOR
- EORI
- LDUR
- LSL: This instruction uses the shamt field to encode the shift amount, while Rm is unused.
- LSR: Same as LSL.
- ORR
- ORRI
- STUR
- SUB
- SUBI
- SUBIS
- SUBS
- MUL
- PRNT: This is an added instruction (part of our emulator, but not part of LEG or ARM) that prints a register name and its contents in hex and decimal. This is an R instruction. The opcode is 11111111101. The register is given in the Rd field.
- PRNL: This is an added instruction that prints a blank line. This is an R instruction. The opcode is 11111111100.
- DUMP: This is an added instruction that displays the contents of all registers and memory, as well as the disassembled program. This is an R instruction. The opcode is 11111111110.
- HALT: This is an added instruction that triggers a DUMP and terminates the emulator. This is an R instruction. The opcode is 11111111111

You may implement your disassembler in any language you like so long is it builds (if a compiled language) and runs on pyrite. This restriction is necessary in order to give the TAs a common platform to test and evaluate your solution. Most important, popular, or trendy languages are already installed on pyrite, including C, C++, Java, and Python. If your language of choice is not installed, you are welcome to contact the system administrators <coms-ssg@iastate.edu> to request its installation; however, I can make no guarantees about whether or not they will do the installation for you, and--if they do--how timely their service will be.

In the past few semesters, students have stumbled upon the unfortunate idea of converting bit-strings to character strings and doing string manipulations instead of bit manipulations. Last semester at least 90% of the class chose this lamentable "solution". This is no longer acceptable. Any efforts to avoid bit manipulations will be rewarded with a static 50% penalty to your assignment grade. You are required to use bitwise operations.

In order to minimize the burden on the TAs, your solution should include two bourne shell scripts in the top level directory: build.sh and run.sh. build.sh should include the executable command(s) necessary to build your program; if your program is in an interpreted language, this file may be empty (but it must exists, nonetheless!). run.sh should take one parameter, the name of a LEGv8 binary file, and pass it to your disassembler. For instance, if I were implementing my disassembler in C and had the program in a source file named disasm.c, my build.sh would contain exactly:

```
gcc disasm.c -o disasm
```

And my run.sh would contain exactly:

```
./disasm $1
```

Bourne shell scripts are simply text files, and *$1* is interpreted as the first positional parameter passed to the script on the command line. To run your scripts, execute `sh build.sh` or `sh run.sh <legv8 assembly file>`.

The data lost in converting from assembly to machine code are comments and label names. Both of these are completely irretrievable, but new label names can be generated, even if these are devoid of the semantic meanings imparted by the original program author. For example, you can simply number the labels: "label1", "label2", etc. Your disassembled output should generate new label names such that our emulator can execute or assemble your generated code. Your "reassembled disassembly" should be byte-for-byte identical to the input.

To use the emulator as an assembler (the output file will have the same name is the input with ".machine" concatenated onto the end): `./legv8emul <legv8 assembly file> -a`
To run the emulator in binary emulation mode: `./legv8emul <legv8 binary file> -b`

Here is example C code to load the binary program from disc and correctly handle endianness:

```
//This is in main():
if (binary) {
  fd = open(argv[1], O_RDONLY);
  fstat(fd, &buf);
  program = mmap(NULL, buf.st_size, PROT_READ | PROT_WRITE,
                 MAP_PRIVATE, fd, 0);
  bprogram = calloc(buf.st_size / 4, sizeof (*bprogram));
  for (i = 0; i < (buf.st_size / 4); i++) {
    program[i] = be32toh(program[i]);
    decode(program[i], bprogram + i);
  }
  emulate(bprogram, buf.st_size / 4, &m);
  return 0;
}
```

be32toh() is a function found in endian.h which converts 32 bit integers from big endian to host endian. I will talk more about this, and show some other, related code in lecture.

Here is a list of the opcodes, extracted from the emulator sources: opcodes.txt

The 0b prefix is a GNU extension (supported by GCC and some other compilers, but not an official part of C) that signifies the value that follows is given in binary.

#pin

programming2

good note | 0 Updated 6 months ago by Jeremy Sheaffer

followup discussions, *for lingering questions and comments*

Start a new followup discussion

Compose a new followup discussion