University of Arizona, Department of Computer Science

CSc 340 Foundations of Computer Systems — Assignment 4

Christian Collberg

Due Tuesday April 3, at 11:59pm

1 Overview

Your assignment is to write a Mips disassembler. This is a program that takes a 32-bit integer as input, decodes it as a Mips instruction, and prints out the resulting assembly code.

2 Assignment

`/home/cs340/spring01/prog4/dis.c` contains the following template:

```
int   mask[]  = {MASK};
int   value[] = {VALUE};
int   class[] = {CLASS};
char* instr[] = {NAME};
char* regs[]  = {REGS};
int   test[] = {0x210affff}; // addi $10, $8, -1

   //  ADD YOUR OWN FUNCTIONS HERE!

void disassemble(int x) {
   //  ADD YOUR OWN CODE HERE!
}

int dis (int code[], int length) {
  int i;
  for(i=0; i<length; i++)
    disassemble(code[i]);
}

int main () {
  dis(test,sizeof(test)/sizeof(int));
}
```

You should add code to

```
   void disassemble(int x) {
   }
```

that decodes `x` as a Mips instruction and prints out the resulting assembly code. For example, the call

```
disassemble(0x0000000c)
```

should print out ⌜`syscall`⌝ and

```
disassemble(0x210affff)
```

should print out ⌜`addi $t2,$t0,-1`⌝.

3 Algorithm

To disassemble an instruction-word X we need to do three things:

1. We must *classify* X, i.e. determine if X represents an `add`-instruction, a `beq`-instruction, etc.
2. We must *extract* the arguments from X. For example, if X represents an `add`-instruction, we must extract the three registers in the instruction.
3. We must *format* the disassembled instruction and print it out. For example, if we have classified X as a `sw` instruction and have extracted the register to be stored as `$a0`, the offset as `-24` and the indirect-register as `$sp` then we should format the instruction as

   ```
   sw $a0,-24($sp).
   ```

3.1 Classification

The header-file `mips.h` contains enough information about the Mips ISA to be able to classify every instruction. For example, the entry for `syscall` looks like this:

```
//  000000 00000 00000 00000 00000 001100 syscall
#define syscall_MASK 0xffffffff
#define syscall_VALUE 0xc
#define syscall_CLASS 0
#define syscall_NAME "syscall"
```

The values in `mips.h` are collected together into four arrays in `dis.c`: `mask[]`, `value[]`, `class[]`, and `instr[]`. The i:th entry in these arrays (`mask[`i`]`, `value[`i`]`, `class[`i`]`, and `instr[`i`]`) contains all the information necessary to disassemble the i:th instruction.

For example, the 0:th instruction ("`syscall`") has the entries

```
mask[0]  = 0xffffffff
value[0] = 0xc
class[0] = 0
instr[0] = "syscall"
```

`mask[i]` contains a "`1`" in every place where the instruction encoding for the instruction has a "`0`" or a "`1`". In other words, `mask[i]` points out to us which bits in an instruction can be used to identify it. In the case of `syscall`, `mask` is `0xffffffff` (which is $11111111111111111111111111111111_2$ in binary) since all the bits in the encoding of `syscall` should be used for classification.

`value[i]` is the hex value of those identifying bits, in the case of `syscall 0xc` (binary `1100`). This means that doing a bitwise-and with `mask[i]` and an instruction word X and then comparing the result to `value[i]` is enough to tell us if X is an i-instruction.

If it is, the opcode of the instruction can be found in `instr[i]`.

`class[i]` divides the instruction set into groups, such that all instructions with the same class-number are decoded the same. There are 15 groups.

3.2 Argument Extraction

The entry for `mfhi` is:

```
mask[1]  = 0xffff07ff
value[1] = 0x10
class[1] = 1
instr[1] = "mfhi"
```

The mask is ⌜`1111 1111 1111 1111 0000 0111 1111 1111`⌝ (binary for `0xffff07ff`). This tells us that bits 11-15 should not be used to classify `mfhi`, since they contain a register number:

```
000000 00000 00000 rd 00000 010000 mfhi
```

Instead we should extract bits 11-15 from the instruction word and format them as a register name.

This can be done by casting the instruction word to the structure `CLASS1` which has been set up so that the field `rd` corresponds to the relevant bits:

```
typedef struct {
  unsigned int bitsA:16;
  unsigned int rd:5;
  unsigned int bitsB:11;
} CLASS1;
```

The name of register number r is in `regs[r]`.

3.3 Bit-Manipulation in C

C has several useful operators that manipulate bits in a word. `word << s` shifts `word` (which should be an integer) `s` steps to the left. `word >> s` shifts `word` `s` steps to the right. `word1&word2` performs the bitwise-and on two words, and `word1|word2` performs the bitwise-or. `~word` complements the `word`, i.e. turns all 0's to 1's and vice versa.

4 Development Strategy

As always, it's a good idea to develop the program in stages. For example, you could start by writing a function `class0(x)` which formats and prints an instruction belonging to class 0. This is easy, since there's only one such instruction (`syscall`). Next write a function `class1(x)` that handles `mfhi` and `mflo`. When these work satisfactorily go back and code `disassemble(x)` which uses `mask` and `value` to classify an instruction. When `disassemble` has classified an instruction it can call on `class0` and `class1` to format and print it out. When you've gotten this far your program is almost done: all that is left to do is to write additional $\texttt{class}_i$ functions to decode the remaining classes of instructions.

5 Extension 1 [10% extra credit]

For extra credit, create symbolic labels for branch instructions. That is, instead of generating the following code

```
      beq    $zero,$at,12
      add    $t0,$a1,$a2
      srl    $t0,$t0,1
      addi   $a1,$t0,1
      lui    $at,$zero,4097
```

you should generate

```
      beq    $zero,$at,Lab1
      add    $t0,$a1,$a2
      srl    $t0,$t0,1
      addi   $a1,$t0,1
Lab1:
      lui    $at,$zero,4097
```

You should be able to handle both forward and backwards branches. This extension will probably require you to rewrite parts of the template.

6 Extension 2 [10% extra credit]

As you know, the SPIM assembler for MIPS will will allow you to write *synthetic* instructions, i.e. instructions that do not really exist on the MIPS cpu. The assembler will take such instructions and translate them into one (or sometimes two) "real" instructions.

For this extension you should:

1. Examine the SPIM documentation and output to determine exactly which synthetic instructions it supports, and how they are translated into real instructions.
2. Extend your disassembler to try to recover the synthetic instructions from the real ones.

For example, after having disassembled the following three instructions

```
addiu $29,$29,2
lui $1,6
ori $8,$1,-23407
```

you might translate them into

```
addu   $sp,2
li     $t0,435345
```

To simplify grading, this translation should only take place when the disassembler is run with the flag `-s`:

```
dis -s
```

7 Honors Section

You should do one of the extension for extra credit.

8 Turnin

When you have completed the program, submit your `dis.c` file by typing make turnin. You may turn in as many times as you want; `turnin` will always replace the previously turned-in version with the new version.

If you have implemented any of the extensions you should describe them in the README file, so that the graders know what to test for.

This is an individual assignment. Don't show your code to anyone, don't read anyone's code, don't discuss the details of the code with anyone. If you need help with the assignment see the TA or the instructor.