Computer Design

Computer Science (2nd year B.Sc.)

Pseudo-assembly instructions and memory addressing in MIPS

Seminar #4

1. Pre-check

This section is designed as a check to help you determine whether you understand the concepts covered in class (refer to P&H 2.12, A.1 – A.4). Please answer "true/false" to the following questions and include an explanation.

1.1. High-level language compilers (Fortran, C, ...) may generate pseudo-assembly instructions.

1.2. The main purpose of the Assembler tool is to generate optimized machine code.

1.3. Destination addresses of all jump instructions are completely determined after linking.

2. Program compilation

2.1. How many passes in the source text should the Assembler tool make and why?

2.2. Describe the six main parts of assembler-generated object file (Header, Text, Data, Relocation Table, Symbol Table, Debug Information).

3. MIPS assembler

Consider a C program that contains a single `sum` function that computes the sum of the elements in an array. Below is a compiled version of this program in MIPS.

```
1 .import print.s        # print.s is an external file
2 .data
3  array:  .word 1,2,3,4,5
4 .text
5  sum:    la   $t0, array
6          li   $t1, 4
7          move $t2, $0
8  loop:   beq  $t1, $0, end
9          addi $t1, $t1, -1
10         sll  $t3, $t1, 2
11         add  $t3, $t0, $t3
12         lw   $t3, 0($t3)
13         add  $t2, $t2, $t3
14         j    loop
15 end:    move $a0, $t2
16         jal  print_int  # function defined in print.s
17         ...
```

3.1. Which lines contain pseudo-instructions that the assembler should convert to actual MIPS instructions (i.e. ISA instructions)?

3.2. For branch/jump instructions, which labels will be resolved on the first pass of the assembler? and which others in the second pass?

3.3. Suppose the machine code for this program, once loaded into memory, starts at address `0x00400000`. Why is there a jump of 8 between the first and second line?

```
0x00400000:  sum:    la   $t0, array
0x00400008:          li   $t1, 4
0x0040000C:          move $t2, $0
0x00400010:  loop:   beq  $t1, $0, end
0x00400014:          addi $t1, $t1, -1
0x00400018:          sll  $t3, $t1, 2
0x0040001C:          add  $t3, $t0, $t3
```

```
0x00400020:          lw   $t3, 0($t3)
0x00400024:          add  $t2, $t2, $t3
0x00400028:          j    loop
0x0040002C:  end:    move $a0, $t2
0x00400030:          jal  print_int
```

3.4. Give the Symbol Table once the Assembler tool has made all its passes.

4. Memory Addressing in MIPS

There are several addressing modes for accessing memory in MIPS:

- **Register mode**: the address is contained in the register. e.g. "`jr $ra`"
- **Indexed**: The target address is computed from a base address contained in a register to which an offset value is added. e.g. « `lw $a1, 0($s0)` » , « `sb $t1, 3($a0)` » .
- **PC Indexed**: Uses the $pc (actually $pc + 4) to which an immediate (multiplied by 4) is added to create the target address. This mode is used by instructions like `beq` and `bne`. The immediate in this type of instructions encodes the number of instructions to jump (positive values for forward jumps and negative values for backward jumps).
- **Pseudo-direct** : The target address in this mode is obtained by concatenating the four MSB bits of the $pc register and the 26 bits [25:0] of the jump instruction (with two implicit LSB bits of 00 – can you tell why?). This mode is used by J-type instructions.

4.1. An assembler tool you're writing needs to encode jumps to addresses $2^{28} + 4$ bytes more than the current $pc. How to do it? For this question, the exact destination address is assumed to be known at compile time (Hint: you need several assembly instructions).

4.2. You now need to write some MIPS instructions to encode branching (i.e. conditional jumps) to instructions $2^{17} + 4$ bytes higher than the current $pc. You may assume for this question that the jump address does not belong to a new 2^{28} bytes block.

4.3. Consider the following MIPS instructions and their memory addresses. Give the machine code associated with each instruction by filling in the empty fields on the right (you will need your mips sheet!). Note: The **OpCode** fields are already pre-populated for you.

```
0x002cff00:  loop:  addu $t0, $t0, $t0   | 0
0x002cff04:         jal foo              | 3
0x002cff08:         bne $t0, $0, loop    | 5
      ⋮                       ⋮
0x00300004:  foo:   jr $ra               # $ra =
```