Computer Design

Computer Science (2nd year B.Sc.)

Pseudo-assembly instructions and memory addressing in MIPS

Seminar #5

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.
TRUE! It is the Assembler tool that must transform the pseudo-instructions into real instructions that conform to the ISA of the target processor.

1.2. The main purpose of the Assembler tool is to generate optimized machine code.
FALSE! This is the function of the compiler. The assembler is primarily responsible for translating pseudo-instructions into actual ISA instructions and calculating relative addresses and offsets for jump and branch instructions.

1.3. Destination addresses of all jump instructions are completely determined after linking.
FALSE! Register-related jumps (i.e., from instructions like `jr` and `jalr`) are only known at runtime (i.e. when the program is executed).

2. Assembling a program

2.1. How many passes in the source text should the Assembler tool make and why?
Twice. The first pass encodes simple instructions and determines the addresses of all the labels in the program. The second pass is used to encode instructions containing forward references in the program.

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

- Header: The sizes and positions of the other parts of the object file.
- Text: The (almost complete) machine code

1. Data: The binary representation of the static data present in the source file.

- Relocation Table: Identifies the lines of code that need to be handled by the linker (jumps to external labels, references to static data, ...).
- Symbol Table: A list of referenced labels and data (i.e. globally accessible) in the source files.
- Debug information: Additional information used by debuggers.

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)?

5, 6, 7 and 15. The pseudo instruction "`la` " is translated into two instructions: "`lui`" and "`ori`". "`li` " in 6 becomes an "`addi`" instruction, and "`move`" in 7 and 15 is converted to an "`add`" instruction.

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?

The jump to "loop" will be resolved in the first pass because the label is defined (line 8) before it is referenced (line 14). The "end" label, on the other hand, will be resolved (line 8) in a second pass of the assembler because it is defined later in the program (line 15). The first pass will have made it possible to identify where this late reference is located. The "print_int" label (on line 16) refers to a symbol in another file, and will in this case be resolved by the linker.

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
```

There is a jump of 8 because "`la`" is a pseudo-instruction that translates into two regular MIPS instructions (see answer to question 3.1)!

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

Label	Address
`sum`	`0x00400000`
`loop`	`0x00400010`
`end`	`0x0040002C`

Including the `loop` and `end` labels in the symbol table is a perfectly valid answer if using an assembler alone, as it would have no way of telling the difference between the three labels. In reality, however, we very often use tools that integrate a compiler, an assembler and a linker at the same time to produce object/executable files (e.g. `gcc`). This kind of tools will be able to determine from the compilation phase which labels are for functions and which are not. As such, it will only put function labels in the symbol table since those are the only ones that other files can reference.

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).
J-type jump instructions can only reach addresses that have the same four MSB as the $pc. A jump of $2^{28} + 4$ bytes would require changing the $pc's fourth most significant bit. Therefore, a J-type jump instruction is not suitable here. We need to load the address into a register and use a "`jr`" instruction

```
lui $at, { the upper 16 bits of the address }
ori $at, $at, { the lower 16 bits of the address }
jr $at
```

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.

The largest address a branch instruction can reach is `$pc + 4 + SgnExt(Imm)`. Since the **Imm**ediate field for this type of instructions is 16 bits and signed, the furthest positive jump one can make is $2^{15} - 1$ words (i.e. $2^{17} - 4$ bytes). So we cannot use a simple branching instruction to encode jumps to addresses $2^{17} + 4$ bytes higher than the $pc. However, considering the assumption made on branching into the same 2^{28} block of bytes, we can use a J-type jump instruction. Ex: to encode "`beq $t0, $0, Foo`" where `Foo` is $2^{17} + 4$ bytes further in memory:

```
                 bne $t0 , $0, DoNotBranch
                 j  Foo
DoNotBranch:     ...     # next instruction of the program
```

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  |  8 |  8 |  8 |  0 |0x21|
0x002cff04:         jal foo             | 3  |        0xc0001         |
0x002cff08:         bne $t0, $0, loop   | 5  | 0  | 8  | -3 = 0xfffd  |
      ⋮                       ⋮
0x00300004:  foo:   jr $ra              # $ra =   0x002cff08
```