Writing Assembly Programs

- You generally shouldn’t need to do this, but we spend time learning it in this course. Why?
- We use an R2000/3000 simulator (SPIM), running on tahiti, fiji, etc.
- SPIM simulates the execution of R2000/3000 assembly programs.
- Basic guidelines:
  1. Use lots of comments
  2. Don’t be too fancy, keep it simple
  3. Don’t get obsessed with performance
  4. Use words (rather than cryptic labels, for instance)
  5. Remember: the address of a word is evenly divisible by 4
  6. Use lots of comments

The SPIM Assembler

- Mostly, the SPIM assembler is pretty faithful to the definition of MIPS assembly language (it only implements a subset of the assembler directives, and includes macros, for instance).
- Because MIPS instructions and addressing modes are quite primitive, the assembler provides several mechanisms for making your programming life easier:
  - Relocatable symbols (labels)
  - Pseudo-instructions: it looks like a normal machine instruction, but it isn’t: the assembler converts it into a sequence of lower level instructions that the machine can execute
  - Additional addressing modes
  - Macros
## Important Pseudo-instructions

- **Some useful pseudo-instructions:** *(src can be reg or immediate)*
  - `mul rd, rs, src`  move rd, src
  - `bgt rs, src, label`  bge rs, src, label
  - `blt rs, src, label`  ble rs, src, label

- **Examples:**
  - `mul $t1, $t2, $t3`  ->  `mult $t2, $t3`
  - `mflo $t1`
  - `mul $t1, $t2, 100`  ->  `mult $t2, 1000`
  - `move $t0, $t1`  ->  `add $t0, $t1, $0`
  - `blt $t1, $t2, foo`  ->  `slt $at, $t1, $t2`
  - `bne $at, $0, foo`
  - `blt $t1, 32, foo`  ->  `subi $at, $t1, 32`
  - `bltz $at, foo`

- **... plus lots more (see the appendix)**

## Summary of Addressing Modes

- **Each ISA specifies a number of addressing modes.**
  - MIPS supports very few addressing modes, namely
    1. **based/displacement/indexed mode:** the address specified by “register + 16-bit signed offset” *(e.g. LW)*
    2. **register mode:** the address is in a register *(e.g. JR)*
    3. **immediate mode:** the address is a constant in the instruction *(e.g. J)*
    4. **PC-relative mode:** the address is calculated by ‘PC + 16-bit signed offset*4’. *(Very similar to base.)* *(e.g. BEQ)*

- **If we use relocatable symbols to specify immediate values, the assembler/linker will do the right thing when the program is relocated.**

- **We’ll see other addressing modes later, when we look at different architectures.**

## Putting a base address into a register

- **Method 1. Leave it up to the assembler:**
  ```
  .data # define program data section
  xyz: .word 1 # allocate some space
  ... # other junk
  .text # define program code section
  ...
  lw $5, xyz # loads contents of xyz to r5
  ```

- **Method 2:** Do it yourself using the **LA pseudo-instruction** that loads an address rather than the contents at that address:
  ```
  la $6, xyz # r6 holds addr of xyz
  lw $5, 0($6) # rf contains contents of xyz
  ```

## Macros

- **Macros are similar to #define macros in C. Example:**
  ```
  # Macro: print_int
  # Implicit argument: an integer in $a0
  # Side-effect: modifies $v0
  .macro print_int(op)
  move $a0, op
  li $v0, 1
  syscall
  .end_macro
  ...
   .text
   print_int($t0)
  ...
  ```

- **In the above code, the assembler will produce:**
  ```
  move $a0, $t0
  li $v0, 1
  syscall
  ```
SPIM Convention

- SPIM lists memory words from left to right
- Bytes within words are listed from most significant to least significant (just as we would read/write them)

SPIM:

```
[0x00001000] 0x09000001 0x01000300 0x04050000
```

Memory:

```
0x1000 0x01 0x00 0x00 0x09
0x1004 0x00 0x03 0x00 0x01
0x1008 0x00 0x00 0x05 0x04
```