$\boxed{13}$ Assembly Language

13.1 Example Subroutine

```
# int max (int x, int y)
#       { if (x < y) return y; else return x; }
max:
    slt $t0, $a0, $a1             # $t0 = ( $a0 < $a1 )
    bne $t0, $0, a0_lt_a1         # if ( $t0 != 0 )
                                  #     goto a0_lt_a1
    add $v0, $0, $a0              # $v0 = $a0
    jr $ra                        # return
a0_lt_a1:
    add $v0, $0, $a1              # $v0 = $a1
    jr $ra                        # return
```

$\boxed{13.2}$ Immediate Instructions

immediate-instruction ::= aiop-code rt, rs, integer
| `lui` rt, integer
aiop-code ::= `addi` | `addiu` | `slti` | `sltiu` | `andi` | `ori` | `xori`
rt, rs ::= general-register
general-register ::= `$0` | `$1` | . . . | `$31` | temporary-register
temporary-register ::= `$t0` | `$t1` | . . . | `$t8` | `$t9`

`$t0` ≡ `$8`, `$t1` ≡ `$9`, . . . , `$t7` ≡ `$15`
`$t8` ≡ `$24`, `$t9` ≡ `$25` [≡ means identical on input]

For `addi`, `addiu`, `slti`, `sltiu`: integer $\in [-2^{15} \ldots 2^{15}-1]$
For `andi`, `ori`, `xori`, `lui`: integer $\in [0 \ldots 2^{16}-1]$

Op Action: rt = rs op integer   op ::= `+` | `<` | `&` | `|` | `^`
`lui` Action: rt = integer $* 2^{16}$ [Load Upper Immediate]

$\boxed{13.3}$ Decimal and Hexadecimal Integers

integer ::= decimal-integer | hexadecimal-integer

decimal-integer ::= sign decimal-digit decimal-digit*

hexadecimal-integer ::=

    sign hexadecimal-digit hexadecimal-digit*

sign ::= + | - |

decimal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

hexadecimal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| A | B | C | D | E | F

$\boxed{13.4}$ Register Instructions

register-instruction ::= rop-code rd, rs, rt

rop-code ::= `add` | `addu` | `sub` | `subu` | `slt` | `sltu`
| `and` | `or` | `xor`

rd, rs, rt ::= general-register

Action: rd = rs op rt   op ::= `+` | `-` | `<` | `&` | `|` | `^`

`add` and `sub` do signed integer computation and trap if result is $< -2^{31}$ or $\geq +2^{31}$.

`addu` and `subu` do unsigned integer computation modulo 2^{32} and never trap.

`slt` does signed integer comparison; `sltu` does unsigned

Comments:

1. Hexadecimal numbers are base 16. They are denoted by writing "`0x`" in front of their digits. Their digits are `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a` (=10), `b` (=11), `c` (=12), `d` (=13), `e` (=14), and `f` (=15). Thus `0x345` $= 3 \cdot 256 + 4 \cdot 16 + 5 \cdot 1$; $100 = 6 \cdot 16 + 4 \cdot 1 =$ `0x64`; and `0xaf` $= 10 \cdot 16 + 15 \cdot 1$.

2. In `addiu` the `i` indicates that the last operand is an "immediate" value, i.e. a constant. The `u` indicates that the arithmetic operation is unsigned: i.e. unsigned addition on unsigned integers, instead of signed addition on signed integers.

3. The `addiu` and `sltiu` instructions have signed immediate values, in spite of the fact that they perform unsigned arithmetic operations. This means that the 16 bit immediate value is converted to a 32 bit integer using signed computation, and then the 32 bit integer is treated modulo 2^{32} as an unsigned integer.

4. The two instructions:

```
lui   $t0, 0x1234
ori   $t0, $t0, 0x5678
```

   used together can load a 32-bit constant, e.g. `0x12345678`, into a register, e.g. `$t0`.

5. A 'trap' or an 'exception' causes an instruction to abort and to change the program counter to `0x80000080`, which is the address where a subroutine to handle the trap is stored.

$\boxed{13.5}$ Labels and Branches

label-directive ::= symbol **:**
symbol ::= C-identifier

A label makes the symbol table entry:

(symbol, address-of-next-instruction-or-datum)

branch-instruction ::= bop-code rs**,** rt**,** symbol
bop-code ::= `beq` | `bne` rs, rt ::= general-register

Action: if (rs op rt) goto symbol op ::= `==` | `!=`

13.6 Example Loop

```
# int $t0 = 1;
# do $t0 = $t0 + $t0 while ( $t0 < 8 );

        addi $t0, $0, 1              # $t0 = 1
loop:
        add  $t0, $t0, $t0           # $t0 = $t0 + $t0
        slti $t1, $t0, 8             # $t1 = ( $t0 < 8 )
        bne  $t1, $0,  loop          # if ( $t1 != 0 )
                                     #     goto loop
```

$\boxed{13.7}$ SYSCALL Instructions

general-register ::= `$0` | `$1` | ... | `$31` | temporary-register
| value-register | argument-register

value-register ::= `$v0` | `$v1`

`$v0` $\equiv$ `$2`, `$v1` $\equiv$ `$3` [$\equiv$ means identical on input]

argument-register ::= `$a0` | `$a1` | `$a2` | `$a3`

`$a0` $\equiv$ `$4`, `$a1` $\equiv$ `$5`, `$a2` $\equiv$ `$6`, `$a3` $\equiv$ `$7`

syscall-instruction ::= `syscall`

Call to operating system; uses `$v0` contents as opcode.

`$v0` = 5: Read integer into `$v0`.

`$v0` = 1: Print integer that is in `$a0`.

`$v0` = 10: Exit program.

13.8 Example Using SYSCALL

```
#   $t0 = read_int();
#   $t0 = $t0 + read_int();
#   print_int ( $t0 );

    addi $v0, $0, 5           # $v0 = 5
    syscall                   # $v0 = read_int()
    add  $t0, $0, $v0         # $t0 = $v0
    addi $v0, $0, 5           # $v0 = 5
    syscall                   # $v0 = read_int()
    add  $t0, $t0, $v0        # $t0 = $t0 + $v0
    addi $v0, $0, 1           # $v0 = 1
    add  $a0, $0, $t0         # $a0 = $t0
    syscall                   # print_int ( $a0 )
```

$\boxed{13.9}$ Subroutine Calls

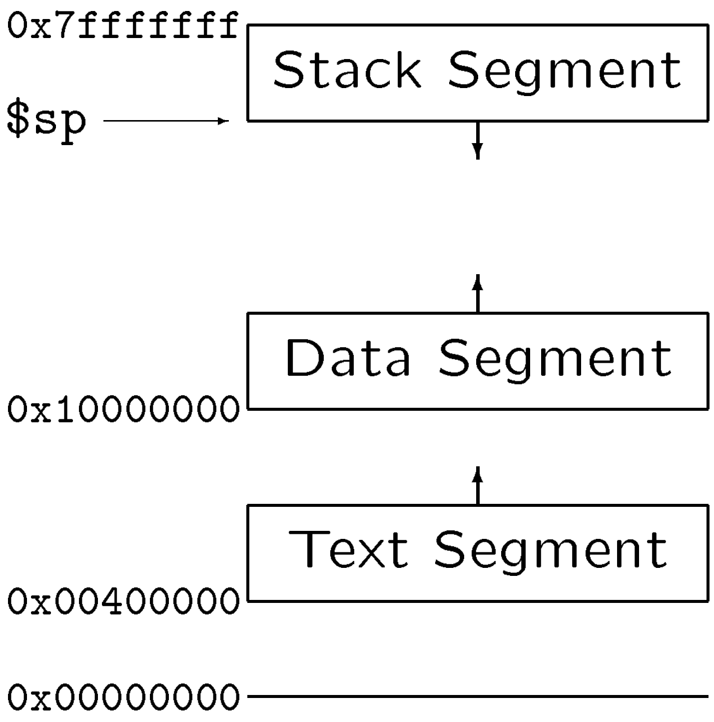
jump-instruction ::= `jal` symbol [Jump and Link]
| `jr` rs [Jump to Register]
| `jalr` rs [Jump and Link to Reg.]

`jal` Action: `$ra` = `$pc` + 4; `$pc` = symbol [`$ra` ≡ `$31`]
`jr` Action: `$pc` = rs
`jalr` Action: `$ra` = `$pc` + 4; `$pc` = rs

E.g.

```
addi $a0, $0, 5     # $a0 = 5
addi $a1, $0, 3     # $a1 = 3
jal  max            # $v0 = max ( $a0, $a1 )
```

13.10 UNIX Process Memory Layout




```
# print_string ( "HI\n" );

        .data
hello:
        .ascii "HI\n\0"
        .text
        addi $v0, $0, 4
        la $a0, hello
        syscall
```

13.11 Data Directives

data-directive ::= `.data` | `.text`
| `.ascii` C-quoted-string-constant
| `.align` 2
| `.word` integer
| `.globl` symbol

sbrk syscall: Allocates a memory block in the data segment of some given size in bytes.

```
addi $v0, $0, 9          # $v0 = 9
addi $a0, $0, size       # $a0 = size in bytes
syscall                  # $v0 = address of new block
```

Comments:

6. At any point in an assembly language file either the text segment or the data segment is current. The next instruction or data item read is placed at the end of the current segment.

   At the beginning of the file the text segment is current.

   The `.data` directive makes the data segment current. The `.text` directive makes the text segment current.

7. When the assembler reads an instruction, it assembles the instruction into a 32-bit word, and stores that instruction word at the end of the current segment.

   Note that some assembler instructions are pseudoinstructions that generate more than one actual machine instruction and hence more than one actual word.

8. When the assembler reads an `.ascii` directive, it stores the following character string at the end of the current segment. To use this string with the `print_string syscall`, the user must end it with a `\0` character. Several `.ascii` directives in a row with a `\0` only at the end of the last may be used to break up a long character string into several lines of code.

Comments:

9. When the assembler reads a `.word` directive it stores the following integer as a 32 bit word at the end of the current segment. However, if the current segment has a length that is not a multiple of 4 bytes, enough extra bytes are added to the segment before the word is stored so that the word's address will be a multiple of 4. That is, the word will be stored at a word boundary in RAM.

   Unfortunately this means that a label immediately preceding the `.word` directive will not address the word if extra bytes were added. To fix this put an `.align 2` directive just in front of the label as in:

   ```
           .align 2
   my_word:
           .word 0
   ```

   The `.align 2` directive puts in the extra bytes to make the next address be a multiple of $2^2 = 4$.

10. The `.globl` directive must be used to indicate which symbols are global, i.e. are to be visible outside the file in which they occur. You cannot set breakpoints on non-global symbols.

11. To be safe when using the sbrk `syscall`, always request a number of bytes that is a multiple of 4. Then word data will always be aligned properly.

13.12 Recursive Subroutines

```
# Prolog:
#   Push $ra, $s0
#   $s0 = $a0
  addiu $sp, $sp, -4
  sw    $ra, 0($sp)
  addiu $sp, $sp, -4
  sw    $s0, 0($sp)
  add   $s0, $0, $a0
```

```
# Epilog:
#   Pop $s0, $ra
#   Jump to $ra
  lw    $s0, 0($sp)
  addiu $sp, $sp, 4
  lw    $ra, 0($sp)
  addiu $sp, $sp, 4
  jr    $ra
```

Save Registers: `$s0` $\equiv$ `$16`, `$s1` $\equiv$ `$17`, . . . , `$s7` $\equiv$ `$23`

Subroutine must save and restore `$ra` and `$s0`, . . . , `$s7`

13.13 Register Usage

register name	software name	usage
$0	$0	Always has value 0.
$2, $3	$v0, $v1	Values returned by subroutines.
$4...$7	$a0...$a3	Arguments to subroutines.
$8...$15	$t0...$t7	Temporary registers.
$16...$23	$s0...$s7	Registers saved and restored by subroutines.
$24, $25	$t8, $t9	More temporary registers.
$29	$sp	Stack pointer.
$31	$ra	Return address register.

Unmentioned general registers have other uses.

Comments:

12. Subroutines are required to save and restore the save registers `$s0`, ..., `$s7` and the stack pointer `$sp`, but they may destroy the contents of the temporary registers, `$t0`, ..., `$t9`, the argument registers, `$a0`, ..., `$a3`, the value registers, `$v0` and `$v1`, and the return address register, `$ra`.

13. A `syscall` instruction calls an operating system subroutine that must save and restore all registers except any value registers, `$v0` and `$v1`, that the particular `syscall` subroutine uses to return results.

14. When a program is started in the normal way, it jumps to location `0x400000`, where there is some code provided by the system to initialize the stack pointer, `$sp`, and then `jal` to the subroutine whose address is the global symbol `main`. The program will exit if the `main` subroutine returns.

13.14 Memory and Multiply Instructions

memory-instruction ::= mem-op rt, offset(base)
mem-op ::= `lw` | `sw`
rt, base = general-register   offset ::= integer

`lw` Action: rt = W(base + offset)   [W(address) denotes
`sw` Action: W(base + offset) = rt   word at address.]

multiply-instruction ::= mul-op rs, rt   op ::= `mult` | `multu`
Action: `$hi$lo` = rs * rt   signed/`mult` or unsigned/`multu`

`$hi$lo` denotes the 64 bit integer whose high order 32 bits are the `$hi` register and low order 32 bits are the `$lo` register.

The instruction '`mfhi` rd' has action: rd = `$hi`.

The instruction '`mflo` rd' has action: rd = `$lo`.

Comments:

15. The 'load word' (`lw`) and 'store word' (`sw`) instructions must be given word addresses that are an exact multiple of 4. Otherwise an alignment trap will occur.

16. The product of two 32-bit integers is in general a 64-bit integer. This is stored in two parts: the high order 32 bits are stored in the `$hi` register and the low order 32-bits in the `$lo` register. Special instructions (`mflo` and `mfhi`) can copy these registers to general registers.

13.15 Pseudo-Instructions

Some assembly language instructions are 'compiled' by the assembler into one or more different instructions.

Load Immediate (Load Address, `la`, is similar):

`li` rt, `0xZZZZ`	$\Longrightarrow$	`addi` rt, `$0`, `0xZZZZ`
`li` rt, `0xYYYY0000`	$\Longrightarrow$	`lui` rt, `xYYYY`
`li` rt, `0xYYYYZZZZ`	$\Longrightarrow$	`lui` rt, `xYYYY`
		`ori` rt, rt, `xZZZZ`

Move and Branches:

`move` rd, rt	$\Longrightarrow$	`add` rd, \$0, rt
`beqz` rs, symbol	$\Longrightarrow$	`beq` rs, \$0, symbol
`bnez` rs, symbol	$\Longrightarrow$	`bne` rs, \$0, symbol
`b` symbol	$\Longrightarrow$	`beq` \$0, \$0, symbol