Supplementary Information

MIPS R2000 Assembly Language

❇ Acknowledgements ❇

The information in this section on the MIPS processor was compiled by Bob Rinker. The format is based on the original version of these notes, based on the M68000, developed by Dr. Dale Grit, Colorado State University. The information about the MIPS processor and the SPIM simulator was obtained from the sources listed below.

References

[1] G. Kane and J. Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

[2] James Larus. *SPIM S20: A MIPS R2000 Simulator.* Manual and code available via anonymous ftp from `ftp.cs.wisc.edu/pub/spim`.

[3] D. A. Patterson and J. L. Hennessey. *Computer Organization & Design: the Hardware/Software Interface.* Morgan–Kaufmann, 1994.

CS 245 Computer Organization

Notes

How to Learn an Assembly Language

Each processor has its own architecture and assembly language. If you happen to use assembly language in the future, chances are you will use something other than the MIPS R2000. Thus it will be necessary for you to adapt the concepts you learned for the MIPS R2000 to another processor. The following are the set of steps involved in learning a new assembly language.

1. Study the **Programming Model** of the processor, consisting of the hardware features of a processor which are important to the programmer, including:
   - Number, size and type of the *registers*.
   - The *addressing modes* that are available. The MIPS has a relatively small number of addressing modes, typical of RISC processors.
   - Method for doing *conditional branches*. With MIPS, a comparison between values in registers is performed in the same instruction that does the branch. Another common method is to save the results of the last operation in a Condition Code (or Status) register, and then provide a set of instructions that branch based upon the values of the condition codes.
   - *Memory organization*, how addressed, little-endian or big-endian, any reserved or restricted areas.
   - Any *limitations* imposed by the processor organization. For example, some addressing modes can only be used with specific instructions.
2. Know the **Programming Conventions** used with the assembler. These are rules and guidelines that have been agreed to by the programming community for this processor. They are not hardware–enforced (i.e., part of the *Programming Model*), but they usually must be followed if your code is to interface and function properly with other code (subroutines written by others, for example) that you may wish to use. These conventions include:
   - The programming style used with the particular assembler code.
   - Any guidelines used for assigning, using, or allocating registers.
   - The common methods used for passing parameters between subroutines.
   - Any programming "tricks" that are commonly used with this particular assembler.
3. Learn the most common instructions thoroughly. Typically, 80% – 90% of an assembly language program uses only 20% of the instructions available in a processor.
4. Learn the assembler syntax and the most common assembler directives.
5. If possible, start with a working example program and modify it, rather than writing a program from scratch.
6. Have a reference guide available for the processor, in case you need to look up the syntax or operation of an uncommon instruction.

Notes — MIPS R2000

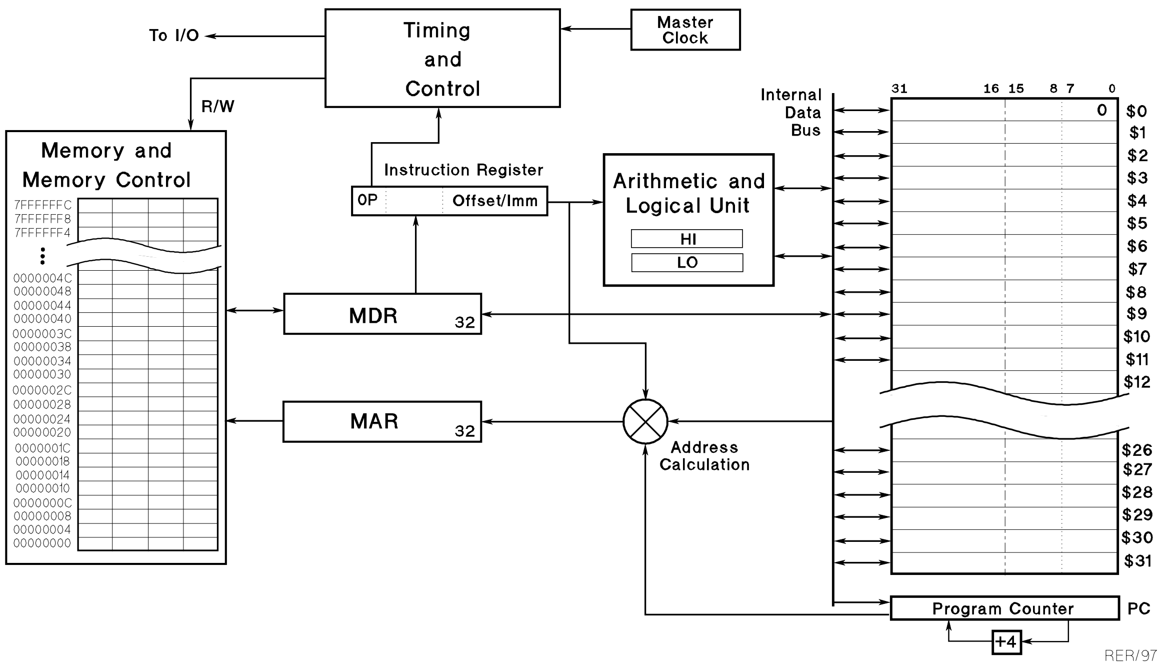
Programming Model

The *programming model* of a processor is that information about the resources and organization within a CPU that is needed by the assembly language programmer. It specifies the number of registers, the types of data they can hold, their size (width), function, and limitations on their use. The programming model is a simplification of the actual hardware, and is not intended to show the implementation details of the processor.

- **Computer Block Diagram**

ALU	- Arithmetic and Logic Unit.
IR	- Instruction Register. Holds the instruction text while it is being executed.
PC	- Program Counter. Contains the address of the next instruction.
\$0 - \$31	- General Purpose Registers. Hold 8, 16, or 32-bit data values or addresses.
MAR	- Memory Address Register. Holds *address* of item being read from/written to memory.
MDR	- Memory Data Register. Holds *value* read from/written to memory.

Programmer's Model of the MIPS R2000 Microprocessor



• Programmer Accessible Registers

- **General Purpose Registers:**

  Function: temporary, quickly accessible locations for holding data and addresses

  Sizes: word (32-bits)
halfword (16-bits)
byte (8-bits)

  Number: 32, specified by **\$0 - \$31**, **\$0** always contains the value 0, some of the other registers are reserved for special purposes by *convention*, **\$8 - \$25** are available for unrestricted use

- **HI and LO Registers** (we will not use these registers directly):

  Function: holds the (64 bit) result of a multiply instruction, and the quotient (LO) and remainder (HI) of a divide instruction

  Size: one word (32-bits) each, sometimes treated as a single 64 bit register

- **Program Counter:**

  Function: contains address of next instruction to be executed

  Size: one word (32-bits)

• Data Types and Sizes

- **Integer**
  - 8-bit **byte** - usually used for ASCII characters
  - 16-bit **halfword** - hold 2's complement (-32K to +(32K-1)) or unsigned (0 - (64K-1)) values
  - 32-bit **word** - used for *addresses* or larger 2's complement values ($\approx \pm$ 2 billion)
- **Floating Point** - 32 bit representation of real values (we will not use this type)

• Storage Organization

When a word or halfword is stored in memory, the bytes can be stored in one of two ways:

Big–endian - the most significant bits are stored at the lowest address

Little–endian - the least significant bits are stored at the lowest address

For example, if we store `0x01234567` (hex) starting at location 1000, bytes `1000 - 1003` will hold the values shown for each order:

Big – Endian	Little – Endian
`1000: 01 (hex)`	`1000: 67 (hex)`
`1001: 23 (hex)`	`1001: 45 (hex)`
`1002: 45 (hex)`	`1002: 23 (hex)`
`1003: 67 (hex)`	`1003: 01 (hex)`

The MIPS can store the values in either order, and is determined by system software. The SPIM simulator stores values in the same order as the host processor. In general, this is of little concern to the programmer, unless a file is going to be moved from one type of machine to the other.

• MIPS Addressing Modes

The MIPS is a *load/store* machine - all values must be moved into the CPU from memory and placed into one of the general purpose registers before it can be used. The location of a value, as specified by the addressing mode, is called its *effective address* (**EA**). The table below summarizes the addressing modes available in the MIPS processor. The first two modes are used by those instructions which perform calculations or other data manipulation. The last three are used to specify memory addresses for the load/store instructions. The next section shows pictorially how the operand is determined for each addressing mode.

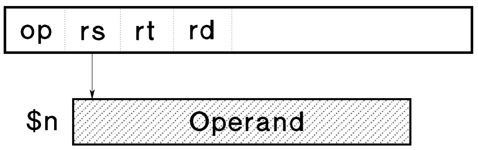
Mode	**Syntax**	**Effective Address**
Register	$\$n$	Operand = $\$n$
Immediate	number or ASYMB	Operand = offset
Relative (or PC relative)	RSYMB or RSYMB(PC)	EA = [PC] + offset
Register Indirect	($\$n$)	EA = [$\$n$]
Base	offset($\$n$)	EA = [$\$n$] + offset

Notes: The following symbols are used in the table above:

- **EA** - Effective Address
- **$\$n$** - General Purpose Register n
- **PC** - Program counter
- [ ] - "contents of"
- **ASYMB** - absolute symbol - a number (literal), or a symbol whose value is fixed
- **RSYMB** - relocatable symbol - a symbol (such as a label) whose actual value will be determined during assembly
- **offset** - value contained in the lower 16 bits of the instruction

• MIPS Addressing Modes - Pictorial View

Register

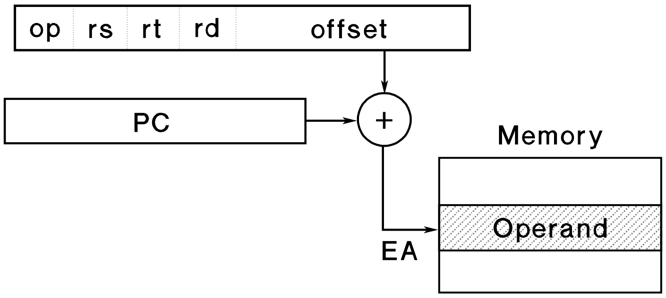


Register $n is the operand (contains the data)

Immediate - used in immediate instructions

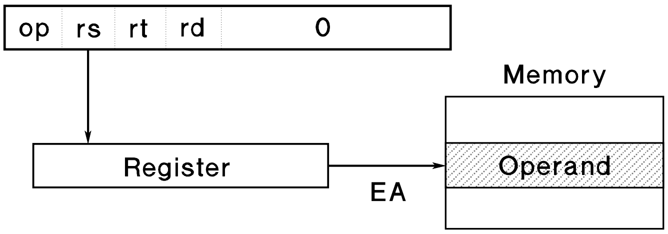
Operand (data) is contained in instruction

Relative (or PC–Relative)



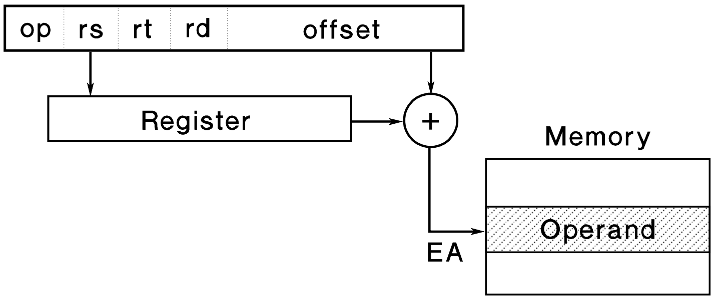
Address of operand is contents of PC plus offset from instruction

Register Indirect



Address of operand is contained in Rs

Base



Address of operand is contents of register Rs plus offset from instruction

- **Memory Operations** - these sequences are used anytime a memory operation (*load* or *store*) is to be performed:
  - Memory Read (or Fetch, or Load)
    1. Place *address* of item to be fetched into **MAR**
    2. Perform a memory read operation. Item fetched is placed into **MDR**
    3. Move contents of **MDR** to *destination*
  - Memory Write (or Store)
    1. Place *address* where item is to be written into **MAR**
    2. Move *value* of item into **MDR**
    3. Perform a memory write operation. Item to be stored is placed in memory.
  - Register-Transfer Notation

Memory Read	**Memory Write**
$MAR \leftarrow address$	$MAR \leftarrow address$
$R/W \leftarrow read$	$R/W \leftarrow write$
$MDR \leftarrow Mem[MAR]$	$MDR \leftarrow data$
$dest \leftarrow MDR$	$Mem[MAR] \leftarrow MDR$

- **Instruction Execution Cycle**
  1. Fetch instruction from memory address (i.e., do a *memory read*) specified by **PC**, place in **IR**.
  2. Decode Instruction
  3. Execute instruction:
     If instruction is a load/store:
     - Compute **E**ffective **A**ddress (**EA**) according to the addressing mode in instruction
     - Move **EA** of item into **MAR**
     - If instruction is a store, move value to **MDR**
     - Do a memory *read* (load) or *write* (store)
     - If instruction is a load, move value from **MDR** to register
     - Increment **PC** by 4 to point to next instruction

     If instruction is a branch:
     - If branch is conditional, perform the comparision specified by instruction
     - If branch is unconditional *OR* if branch condition is met, replace **PC** with **PC** + *offset* (from instruction).
     - If condition is NOT met, increment **PC** by 4 to point to next instruction.

     If instruction is arithmetic/logical:
     - Perform the operation between the two source registers
     - Place result in destination register
     - Increment **PC** by 4 to point to next instruction

- **Register-Transfer Notation for the Instruction Execution Cycle (Selected Instructions)**

$$
\begin{array}{rl}
fetch: & MAR \leftarrow PC \\
& MDR \leftarrow Mem[MAR] \\
& IR \leftarrow MDR \\
\\
execute: & if\ instr = Load \\
& \quad MAR \leftarrow EA \\
& \quad MDR \leftarrow Mem[MAR] \\
& \quad \$rd \leftarrow MDR \\
& \quad PC \leftarrow PC + 4 \\
\\
& if\ instr = Store \\
& \quad MAR \leftarrow EA \\
& \quad MDR \leftarrow \$rs \\
& \quad Mem[MAR] \leftarrow MDR \\
& \quad PC \leftarrow PC + 4 \\
\\
& if\ instr = b \text{ (branch unconditional)} \\
& \quad PC \leftarrow PC + offset \\
\\
& if\ instr = bcc \text{ (branch conditional)} \\
& \quad if\ \$rs\ cc\ \$rt\ PC \leftarrow PC + offset \\
& \quad else\ PC \leftarrow PC + 4 \\
\\
& if\ instr = arith/logical \\
& \quad ALU_1 \text{ (temporary ALU register)} \leftarrow \$rs \\
& \quad ALU_2 \leftarrow \$rt \\
& \quad ALU_3 \leftarrow ALU_1\ op\ ALU_2 \\
& \quad \$rd \leftarrow ALU_3 \\
& \quad PC \leftarrow PC + 4 \\
& \qquad \vdots \\
& \text{(all other instructions follow)}
\end{array}
$$

CS 245 Computer Organization

Notes — MIPS R2000

Instructions

• MIPS and RISC Architectures

The MIPS R2000 is a good example of a **R**educed **I**nstruction **S**et **C**omputer (**RISC**). **RISC** is a *design philosophy* characterized by:

- Fixed length, fixed format instructions that are easy to decode
- Load/store architecture, with a relatively small number of addressing modes
- Simple instructions that can be made to execute quickly

An important design maxim with RISC architectures is to *"make the common case fast"*. The most commonly executed instructions should be simple so that the hardware necessary to execute them can operate very quickly. This philosophy manifests itself in many of the features of the MIPS architecture. It is largely this philosophy that has allowed today's processors to achieve incredible performance gains over the more complicated, and much slower, processors of a few years ago.

• MIPS Machine Language (Binary) Instruction Formats

In keeping with the RISC philosophy, the MIPS has only three types of instructions. Since it is the assembler's job to translate assembly instructions to their binary equivalent, it is not essential that the assembly language programmer know the exact instruction format. However, some knowledge of instruction formats is useful, as it helps the programmer to remember what combinations of capabilities each instruction can have – *if an assembly language instruction cannot be coded into binary, it is illegal.*

All MIPS instructions are 32 bits in length. The *R-type* instructions are the most common; they specify an opcode, up to two source and one destination register, and (for the shift instructions) a shift amount. The *I-type* format is used for immediate instructions. The *J-type* is used for the unconditional jump instructions. The bit encodings for each type is:

opcode	*rs*	*rt*	*rd*	*shamt*	*funct*	R – Type
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

opcode	*rs*	*rd*	*offset/immediate*	I – Type
6 bits	5 bits	5 bits	16 bits	

opcode	*jump address*	J – Type
6 bits	26 bits	

The bit fields in the instruction formats are defined as:

opcode - (6 bits), specifies the operation the instruction is to perform
rs - (5 bits), the number of the first source register
rt - (5 bits), the number of the second source register
rd - (5 bits), the number of the destination register
shamt - (5 bits), shift amount (not used in most instructions)
funct - (6 bits), additional opcode bits for R-Type instructions
offset/immediate - (16 bits), the immediate value, or address offset
jump address - (26 bits), shifted left 2 bits, then used as the jump address

• Pseudoinstructions

Some of the MIPS assembler mnemonic codes are not translated directly into unique machine instructions, but rather are *pseudoinstructions.* These codes are translated into a sequence of one or more machine instructions which perform an operation equivalent to what the apparent "machine instruction" does. Pseudoinstructions allow the hardware to remain simple (in keeping with the "*keep the common case fast*" philosophy), yet provide the programmer with a richer and more understandable set of instructions to use. In the descriptions that follow, pseudoinstructions are marked with a dagger (†).

Most of the time, the programmer does not need to worry about whether a particular mnemonic code is a pseudoinstruction or not. However, the following should be kept in mind when using pseudoinstructions:

- Many pseudoinstructions translate into more than one machine instruction; when debugging or using a simulator, remember that there is not a one-to-one correspondence between the assembly instruction and the generated machine instructions. (For example, "single stepping" through a program with the simulator will only execute the first *machine* instruction generated for a pseudoinstruction, NOT the entire pseudoinstruction.
- By convention, register 1 (**$1**) is used by the assembler to expand pseudoinstructions into a sequence of machine instructions. Therefore, you must be careful when using **$1** – it is best to avoid using it.
- It is sometimes possible to re-code pseudoinstructions into the same number of "real" instructions; the resulting sequence will execute faster, since fewer machine instructions will be generated. This is generally only necessary when speed is the most critical factor.

• Assembler Syntax

Each instruction is placed on a separate line in the following format:

```
label:   mnemonic    operands    #comment
```

`label:`	- (Optional) Must start with a letter, be the first thing on a line, followed by a colon (:). The colon is NOT part of the label. Must be unique within the assembly language file. Cannot be an opcode.
`mnemonic`	- Opcode for a MIPS instruction or pseudoinstruction
`operands`	- Other information (registers, memory addresses, values, etc.) needed by the instruction
`# comment`	- Anything after the `#` on a line is ignored by the assembler

The following sections categorize and describe the most commonly-used MIPS instructions (and pseudoinstructions). In the following descriptions:

`rs` and `rt`	- specify source registers
`rd`	- specifies the destination register
`mem`	- specifies a memory address
`imm`	- is a constant (an immediate operand)
`sa`	- is a constant (a shift amount)
`label`	-is a memory address, usually a label on an instruction.

- ## Load/Store, Data Movement Instructions

Assembler Syntax:

Loads		Stores		Data Movement		Immediate	
`lw`	`rd, mem`	`sw`	`rs, mem`	`move`	`rd, rs`†	`la`	`rd, mem`†
`lh`	`rd, mem`	`sh`	`rs, mem`			`li`	`rd, imm`†
`lhu`	`rd, mem`	`sb`	`rs, mem`			`lui`	`rd, imm`
`lb`	`rd, mem`						
`lbu`	`rd, mem`						

Instruction format: I-type

These instructions move a value (`w` - word, 32 bits, `h` - halfword, 16 bits, `b` - byte, 8 bits), between a register and a memory location. The *load* instructions move the value from memory into a register, while the *store* instructions move the value from the register to memory. The **E**ffective **A**ddress of the memory location (i.e., the `mem` field) is specified by any of the three memory addressing modes. The `lh` and `lb` are *signed* loads, meaning that the sign is extended (i.e., replicated) to fill the upper bits of the 32 bit register. `lhu` and `lbu` are *unsigned* - the upper bits of the register are filled with zeroes.

The `la`, `li`, `lui`, and `move` instructions are NOT really load instructions – that is, they do not access memory. The `la` instruction computes the **E**ffective **A**ddress in the instruction and places the *address* (NOT the contents of the memory location) in the register. The `li` places the (16 bit) immediate value into the lower half of the register, while `lui` places the immediate value into the upper half of the register. The `move` instruction copies a value from one register to another.

Examples:

```
lw    $8, val1      # load value of memory location val1 into register 8
sw    $10, 100($9)  # store contents of R10 into location [R9] + 100
la    $15, num      # place ADDRESS of num into register 15
li    $12, 4        # put constant value 4 into R12
move  $8, $4        # copy contents of $4 into $8
```

- ## Arithmetic and Logical Instructions

Assembler Syntax:

Signed Arithmetic Instructions

`add`	`rd, rs, rt`	`sub`	`rd, rs, rt`
`mul`	`rd, rs, rt`†	`div`	`rd, rs, rt`†
`rem`	`rd, rs, rt`† (remainder)		
`neg`	`rd, rs`†		

Unsigned Arithmetic Instructions

`addu`	`rd, rs, rt`	`subu`	`rd, rs, rt`
`mulu`	`rd, rs, rt`†	`divu`	`rd, rs, rt`†
`remu`	`rd, rs, rt`†		

Logical Instructions

`and`	`rd, rs, rt`	`or`	`rd, rs, rt`
`nor`	`rd, rs, rt`	`xor`	`rd, rs, rt` (exclusive or)
`not`	`rd, rs`†		

Instruction format: R-type

The arithmetic instructions compute the *2's complement* operation between the two source registers (`rs` and `rt`), and place the result in the destination register `rd`. The *unsigned* instructions (`addu`, `subu`, etc.) treat the values as unsigned, rather than 2's complement –

these instructions are usually used to compute addresses. The logical instructions compute the *bitwise* operation between the source registers, placing the result in the destination register. The two unary operations, `neg` ("take the 2's complement") and `not` ("take the 1's complement") perform their function on the value in the source register and place the result in the destination.

Examples:

```
add   $10, $8, $9   # compute R10 = R8 + R9
or    $16, $20, $9  # compute bitwise inclusive-OR    R16 = R9 OR R20
neg   $12, $13      # Take negative of value in R13, put into R12
add   $14, $0, $0   # TRICK 1: clears R14. REMEMBER: R0 is always zero
addu  $12, $13, $0  # TRICK 2: same as   mov  $12, $13
```

• Arithmetic/Logical Immediate Instructions

Assembler Syntax:

```
addi    rd, rs, imm            addiu   rd, rs, imm
andi    rd, rs, imm            ori     rd, rs, imm
xori    rd, rs, imm
```

Instruction format: I-type

Several of the arithmetic/logical instructions also have an immediate form. (Note that only the *symmetric* operations, those that compute the same value regardless of the operand order, have immediate forms).

Example:

```
addi  $10, $8, 9    # compute R10 = R8 + 9  Note the value 9, NOT R9
```

• Shift and Rotate Instructions

Assembler Syntax:

```
ror    rd, rs, rt†            rol    rd, rs, rt†

srl    rd, rs, sa             srlv   rd, rs, sa
sll    rd, rs, rt             sllv   rd, rs, rt
sra    rd, rs, sa             srav   rd, rs, rt
```

Instruction format: R-type

The `ror` and `rol` instructions *rotate* the bits within `rs` right (`ro`*r*) or left (`ro`*l*) by the number of positions specified in `rt`, and place the result in `rd`. The logical shift instructions shift the bits in `rs` the number of positions in either the constant `sa` (for `srl` and `sll`), or by a *variable* amount specified in register `rt` (for `srl`*v* and `sll`*v*). With the arithmetic shifts (`sra` and `srav`), the sign (MSB) is copied into the shifted bit positions. These instructions are often used for manipulating individual bits; the shift instructions are used to multiply and divide by powers of 2.

Examples:

```
ror   $10, $8, $9   # R10 = R8 rotated right by amount specified in R9
sll   $16, $20, 9   # R16 = R20 shifted left 9 bits (multiply by 2^9)
sllv  $16, $20, $9  # R16 = R20 shifted left by amount specified in R9
sra   $14, $10, 1   # R14 = R10 shifted right 1 bit (signed divide by 2)
srl   $14, $10, 1   # Unsigned divide by two
```

• Comparison Instructions

Assembler Syntax:

```
seq     rd, rs, rt†          sne     rd, rs, rt†
slt     rd, rs, rt           sle     rd, rs, rt†
sgt     rd, rs, rt†          sge     rd, rs, rt†
sltu    rd, rs, rt           sleu    rd, rs, rt†
sgtu    rd, rs, rt†          sgeu    rd, rs, rt†
slti    rd, rs, imm          sltiu   rd, rs, imm
```

Instruction format: R-type, (`slti`, `sltiu` are I-type)

These instructions perform a comparison `rs` *op* `rt`, where *op* is the comparison contained in the op-code of the instruction (`eq` - equal, `ne` - not equal, `lt` - less than, etc.), and *sets* the value of `rd` to 1 if the comparison is TRUE, and to 0 if false. Most commonly, these are used in conjunction with the conditional branch instructions (described below) to make decisions.

• Branch/Jump Instructions

Assembler Syntax:

```
j       label
b       label†

beq     rs, rt, label         bne     rs, rt, label
blt     rs, rt, label†        ble     rs, rt, label†
bgt     rs, rt, label†        bge     rs, rt, label†
bltu    rs, rt, label†        bleu    rs, rt, label†
bgtu    rs, rt, label†        bgeu    rs, rt, label†

beqz    rs, label†            bnez    rs, label†
bltz    rs, label             blez    rs, label
bgtz    rs, label             bgez    rs, label
```

Instruction format: jump – J-type; branch – I-type

The unconditional instructions (`j` and `b`) transfer control to the instruction at `label` when executed. The conditional branches first perform the indicated comparison, then branch if the result of the comparison is true. For the instructions with two registers, the comparison is performed between the contents of the two registers. The `rt` register can be replaced with a constant instead of a register designation; this produces a "branch immediate" type of pseudoinstruction. For the "`z`" instructions (`beqz`, `bltz`, etc.), the register contents are compared with 0. Most of the conditional branches are pseudoinstructions, composed of a "set" (comparison) instruction, followed by a `beq` or `bne` instruction.

Examples:

```
ex1:  b     down          # skip instructions to label "down"
      .       .
      .       .
down: add ....            # next instruction to execute after branch

ex2:  slt   $1, $8, $9    # Is R8 < R9? If so, set R10 = 1 and ...
      bnez  $1, less      # ... branch to label "less" if true

ex3:  blt   $8, $9, less  # The pseudoinstruction equivalent of ex2
```

• Miscellaneous Instructions

Assembler Syntax:

```
nop†
syscall     code
```

Instruction format: R-type

`nop` ("no-operation") is the instruction that does nothing! It is sometimes used to hold a place for an instruction that may be inserted later, or is sometimes used in debugging when an extra instruction has been found that needs to be "taken out." There are several machine codes that correspond to a no-operation; the "official" binary code for `nop` is zero (i.e., 0x00000000), which makes it easy to remember.

`syscall` is used to invoke system subroutines. It is used instead of the `jal` instruction, because it provides a mechanism that does not require the programmer to know the exact address of the subroutine being invoked – the primary advantage of this is that the system software (and thus, the addresses of system routines) can be changed without requiring all programs that call system subroutines to change. The `syscall` instruction is used within the SPIM simulator to invoke I/O routines, and to exit from a program.

• Subroutine Instructions

Assembler Syntax:

```
jal    label
jr     rs
```

Instruction format: `jal` - J-type; `jr` - R-type

The `jal` ("jump and link") instruction is similar to a jump, in that control is passed to the label specified in the instruction. Additionally, however, the address of the instruction following the jump instruction is *automatically* placed into register `$31`.

The `jr` instruction transfers control to the address contained in `rs`. Thus, if a `jr $31` instruction is executed at the end of a subroutine which was invoked by a `jal`, control will return to the instruction following the `jal`.

Example:

```
      jal   sub  # jump to subroutine sub, R31 = return addr
ret:  ...        # instruction that will be executed following the sub
       .
       .
sub:  ...        # first instruction of subroutine
       .
       .
      jr    $31  # last instruction of subroutine -- performs return
```

Subroutines in Assembly Language

The high level language (C or FORTRAN) programmer expects several "features" or characteristics of subroutines:

1. A mechanism for returning back to the calling point after a subroutine is called.
2. Some means for passing arguments (values) to and from the routine.
3. Variables that are declared within the subroutine are accessible only by that routine (i.e., *local* variables).

With the MIPS, only the first feature is supported by hardware – the others are implemented "by convention" – that is, an agreement among programmers that certain groups of registers will be used in certain ways. These conventions must be followed by anyone wishing to write subroutines that will be used with other code (including code produced by compilers) – however, there is nothing inherent within the MIPS architecture that requires or enforces the use of these conventions.

The following table shows the 32 MIPS registers, their alternate names (conventional usage names), and a brief description of the register usage. Either the number or the alternate name can be used to specify registers in assembly language.

MIPS Register Usage Conventions

Name	Register	Description/Usage
	`$0`	Hardwired constant 0
`$at`	`$1`	Reserved for expanding pseudoinstructions by assembler
`$v0-$v1`	`$2-$3`	Return results from functions
`$a0-$a3`	`$4-$7`	Used for arguments 1-4 of subroutine. Any additional arguments are placed on the stack
`$t0-$t7`	`$8-$15`	"Temporary" - unrestricted use - BUT not saved during a call
`$s0-$s7`	`$16-$23`	"Saved" - will be saved across a call, BUT must be saved (usually on stack) before being used, restored before return
`$t8-$t9`	`$24-$25`	More temporary
`$k0-$k1`	`$26-$27`	"Kernel" - reserved for OS use
`$gp`	`$28`	Pointer to global data area
`$sp`	`$29`	Stack Pointer - contains address of top of stack
`$fp`	`$30`	Frame pointer - contains address of local variable space
`$ra`	`$31`	Return address of subroutine

The following are the guidelines for using the *temporary* and *saved* registers, according to the convention:

1. If your routine DOES NOT call any other routines, then you can use the *temporary* registers (`$t0-$t9` or `$8-$15`, `$24-$25`) without any restrictions.
2. If your routine DOES call another routine, then you cannot assume that values in the *temporary* registers will be left intact across the call (i.e., by convention, the called routine has a right to use these registers). Therefore, for any value stored in a *temporary* register that needs to be preserved across a call, you must either save the register before the call and restore the value afterwards (*caller–saved* register), or assign any such values to a *saved* register instead (but see below).

3. If you use any of the *saved* registers (`$s0-$s7` or `$16-$23`), you must save their contents before you use them (*callee–saved* registers), and restore the values before returning.

Register values are usually saved on the system stack (i.e., the memory pointed to by `$sp`). Since saving and restoring a register is a relatively expensive operation, it is a goal of the assembly language programmer to design a register allocation scheme that minimizes the number of save/restore operations.

Notes

Programming in Assembly Language

Both the advantage and the disadvantage of assembly language is the amount of flexibility the programmer has in coding a program. Any assembly language programmer can exploit this flexibility to write very fast, efficient code. However, the *successful* programmer will recognize the value of the structured programming techniques that are built into high level languages, and will use discipline in choosing coding practices in assembly language which still adhere to the principles of these structured techniques.

Structured programming is a programming philosophy, not a language feature!

- **Assembly Programming Standards**
  - Program Header: Include your name, the assignment number, a brief statement of purpose and a list of all registers used in the main program and how they are used. Enclose the header box in stars.
  - Subprogram Header: Include the name of the subprogram, its purpose, the input to and output from the subprogram, and a list of registers used and how used. Enclose the header box in stars.
  - Columns: The first *executable* statement of your program must have a label of `main`, which also must be declared global (with `.globl`). All mnemonics, operands, and comments are to be aligned. Tab characters are often used to perform this alignment. (See example programs.)
  - Comments: Inline comments (comments on each source line) are more frequently used than in high level languages like C++ or Fortran. Inline comments should be included as the program is typed in, NOT as an afterthought.
  - Program Blocks: Assembly language does not have block–structuring statements like high level languages, and the use of indenting to emphasize program blocks is not commonly used, so the use of comments and blank lines between program blocks are even more important than in high-level languages. The philosophy of programming and commenting is still the same – use comments to help emphasize the block–structure of the program.

- **Program Development**

  Writing a program in assembly language is not much different than writing in a high level language – only the final expression of the design (in assembly language rather than a high level language) is different. The following are the steps involved in the process:

  1. Develop the algorithm, or the sequence of steps, for the program. This is the “intellectual” part of the process. This algorithm may be expressed in some sort of pseudocode, or as a word description, or you can even use a high level language. Regardless of notation, the purpose is to form a clear description of exactly what steps will be required for the program to fulfill its purpose. You should use descriptive names for values, variables and labels that you use.
  2. Divide your program description into small partitions, each of which will become a subprogram. Carefully determine what each subprogram requires as input, and what will be returned as a result. It takes more statements in assembly language to perform an equivalent amount of work compared with a high level language; this means that smaller program units will be necessary to make the program understandable.
  3. Translate your algorithm into assembly language. Part of this process involves translating the program structures (data initializations, calculations, branches/decisions, loops, subprograms, etc.) in your algorithm into assembly language. The following pages show examples of the most common program control structures, written in C, along with the assembly language equivalents. Simply find the proper C sequence, then “plug in” the equivalent assembly code sequence.
  4. A more difficult part of the translation process involves “playing compiler” – performing the steps usually done by the compiler in a high level language. You must decide how to assign values and variables to registers and/or memory locations, how to use the primitive data types to implement more complex structures, how to pass arguments between subroutines, etc. Be sure to follow the programming conventions for register usage – while in a “stand-alone” program the proper conventional register usage doesn't matter, any program that performs a useful function eventually will become a part of a larger program, where the register conventions WILL make a difference.

     Be careful that the registers and memory locations are correctly initialized in each subprogram.
  5. Even though there is no formal concept of “local” and “global” variables in assembly (everything is actually *global*!), it is still an excellent idea to enforce the idea in your programs. Identify the values which should be locally accessible in each routine, and only allow that routine to access them. Make any necessarily global data obvious – identify them as global, using comments. Consider setting up an argument passing mechanism for those values which must be shared among routines.

     > Remember: the most difficult bugs in a program are usually data corruption errors, where one routine changes a value that another routine wasn't expecting to be changed.
  6. You can now create a file containing your program and either assemble it or use the simulator to execute it. You will probably have some syntax errors to correct before it will execute correctly.

Assembly Language Patterns

The following examples show a variety of translations from short `C` code sequences into MIPS assembly language. Assume all variables are `int`s, represented as 32-bit words, unless otherwise noted.

- **Storage Allocation and Assignment**

<u>C Statements</u> / <u>MIPS Assembly Code</u>

```
                                        .data            # variables go in data seg

int x, a = 27, b = 0x35, y;       x:    .space 4         # allocate 4 bytes for x
                                  a:    .word  27        # initialize a to 27
                                  bee:  .word  0x35      # note, 'b' is NOT legal ...
                                  y:    .space 4         # ... since it's an opcode!

  void main()                           .text            # switch to text segment
  {                                     .globl main      # make main a global symbol

    x = 0;                        main: sw     $0, x        # Register $0 = 0
    y = a;                              lw     $8, a        # Get 'a' from memory and ...
                                        sw     $8, y        # ... put it into y
    x += b;                             lw     $8, bee      # Get b ...
                                        lw     $9, x        # ... and x from memory
                                        add    $9, $9, $8   # Compute x + b
                                        sw     $9, x        # Put x back into memory
```

The above was a direct translation of C code into assembly, where the values of variables were kept in memory. With a Load/Store machine like the MIPS, it is much easier and efficient to keep variables mostly in registers. In the following sequence, the variables `r`, `s`, `t` and `total` are allocated to registers `$10 - $13` instead of to memory.

```
  int r=3, s=4, t=5, total;             li     $10, 3       # Load values for r, ...
                                        li     $11, 4       # ... s, and ...
                                        li     $12, 5       # ... t using load immediate
  total = r + s - t;                    add    $13,$10,$11  # compute total
                                        sub    $13,$13,$12
  total += 10;                          addi   $13,$13, 10
```

- **Decision Statements**

<u>The `if-then` Statement</u>

There are no "program blocks" (i.e., the equivalent to C statements enclosed in { }) in assembly language – we must use branch statements to jump around code blocks we don't want to execute. Below are two versions of the `if-then` statement, and the assembly language implementations – the first version is the direct translation from C, while the second is a better implementation in assembly language (Note that we check for the *opposite* condition in the second case.) Assume that the variables `f`, `g`, `h`, `i`, and `j` are in registers `$16 - $20`, respectively.

C Statements

```
/* Version 1 */
 if (i == j)
  {
    g += 5;
    f = g + h;
  }
 f = f - i;

/* Version 2 */
 if (i != j) goto L2
    g += 5;
    f = g + h;
L2: f = f - i;
```

MIPS Assembly Code

```
#  Version 1
       beq    $19,$20,L1   # go to L1 if i equals j
       b      L2           # skip program block if not
 L1:   addi   $17,$17,5    # add 5 to g
       add    $16,$17,$18  # f = g + h

 L2:   sub    $16,$16,$19  # Both paths get here

 # Version 2
       bne    $19,$20,L2   # go to L1 if i equals j
       addi   $17,$17,5    # add 5 to g
       add    $16,$17,$18  # f = g + h
 L2:   sub    $16,$16,$19  # Both paths get here
```

The if-then-else Statement

The **if-then-else** is implemented either (1) with the *else* portion immediately following the conditional branch, or (2) by checking for the *opposite* condition in the branch:

C Statements

```
 if (i == j)
    f = g + h;
 else
    f = g - h;
```

Ex. 1) *else* follows branch

```
       beq   $19,$20,then
else:  sub   $16,$17,$18
       b     endif
then:  add   $16,$17,$18
endif: ...
```

Ex. 2) *then* follows branch

```
       bne   $19,$20,else
then:  add   $16,$17,$18
       b     endif
else:  sub   $16,$17,$18
endif: ...
```

The switch-case Statement

The **switch-case** statement is efficiently implemented by using a *jump table.* In the **.data** section, a table containing the *addresses* of each **case** label is created. Then, the case variable is used to select the proper address in the table. In the following code, assume that variables **f - k** are allocated to registers **$16 - $21** respectively.

```
switch (k)                      li      $10,4          # put 4 into R10
                                mul     $9,$10,$21     # R9 = k * 4, Convert
                                                       # k to jump table offset
                                lw      $8,Jmptbl($9)  # Get address from table
{                               jr      $8             # Jump to proper place
 case 0: f = i + j;     k0:     add     $16,$19,$20    # case 0:
         break;                 j       endsw
 case 1: f = g + h;     k1:     add     $16,$17,$18    # case 1:
         break;                 j       endsw
 case 2: f = g - h;     k2:     sub     $16,$17,$18    # case 2:
         break;                 j       endsw
 case 3: f = i - j;     k3:     sub     $16,$19,$20    # case 3:
}                       endsw:  ...                    # done with case stmt

                                .data
                        Jmptbl: .word  k0    # Jump table. Each word ...
                                .word  k1    # ... contains the address ...
                                .word  k2    # ... of one of the case ...
                                .word  k3    # ... labels
```

- **Loop Structures**

Pre-Test Loop

The following code sequences calculate the value $res = base^{ctr}$ by using successive multiplication. The loop condition is checked *before* executing the loop each time through. The variable `base` is in `$10`, `ctr` is in `$9`, and `res` is computed in `$8`.

C Statements	MIPS Assembly Code

```
res = 1;                      li     $8,1
ctr = exp;                    lw     $9,exp
while (ctr > 0) {       loop: blez   $9,done
   res = res * base;          mul    $8,$8,$10
   ctr = ctr -1;              addi   $9,$9,-1
 }                            b       loop
                        done: ...
```

Post-Test Loop

The following code inputs one character at a time and echos it, until a capital `'Z'` is input. The assembly version uses the SPIM simulator system routines to input and output characters.

```
 char ch;                        .data
                         ch:     .byte   0,0         # first position will hold ...
                                                     # ... char, second is null byte
                                 .text
 do {                    loop:   li      $5,2        # Tell read_str to read 1 char
                                 la      $4,ch       # Address of character buffer
                                 li      $2,8        # Call code for read_str
     read(ch);                   syscall             # call read_str
                                 li      $2,4        # Call code for print_str
     write(ch);                  syscall             # call print_str
                                 lb      $8,ch       # get character from memory
  } while (ch != 'Z');           bne     $8,'Z',loop # Is it 'Z'? No, loop
                         done:   ...                 # Yes? Done.
```

Counting Loop

The following code counts and then outputs the number of bits in a word of memory. A loop which executes 32 times is used for the counting.

```
int i, count;                      .data
unsigned val = 0x2a49b175;   val:  .word   0x2a49b175

                                   .text
                                   li      $8,val         # get value
count = 0;                         li      $9,0           # count = 0
                                   li      $10,0          # i = 0
for(i = 0; i < 32; i++)      loop: bge     $10,32,fin     # Done? If so, leave
 {                                 andi    $11,$8,1       # isolate LSB
  if(val & 0x1 == 1)               beqz    $11,skip       # if LSB = 0, skip count
     count++;                      addi    $9,$9,1        # count++
  val = val >> 1;            skip: srl     $8,$8,1        # shift word right
                                   addi    $10,$10,1      # i++
 }                                 b       loop
                             fin:  ...
```

- **Array Operations - Example using Subscripts**

  The following is a complete program which computes and outputs the sum of an array. This first version uses subscripts to access the array elements.

```
int array[10] =                     .data
   {10,  5, 30, 8, 7,
    14, 22, 31, 3, 6};      array:  .word   10, 5, 30, 8, 7, 14, 22, 31, 3, 6
                            bye:    .asciiz "The sum is "
int i, sum;                         .text
                                    .globl  main
void main() {               main:
  i = 0;                            li      $8, 0          # i = 0
  sum = 0;                          li      $9, 0          # sum = 0
  while (i < 10)            loop:   bge     $8,10,done
   {                                sll     $11,$8,2       #compute i*4
                                    lw      $12,array($11) # get a[i]
    sum += array[i];                add     $9,$9,$12      # add element to sum
    i++;                            addi    $8,$8,1        # i++
   }                                b       loop
                            done:   li      $2,4           # Call code for print_str
                                    la      $4,bye         # Addr of output string
  print("The sum is ");             syscall                # call print_str
                                    li      $2,1           # Call code for print_int
                                    move    $4,$9          # move ans to $4 to print
  print(sum);                       syscall                # call print_int
 }                                  jr      $31            # return
```

- **Array Operations - Example using Pointers**

  The following program performs the same array operation as the previous example, except that it uses pointers (instead of subscripts) to access the array elements.

```
int array[10] =                     .data
   {10,  5, 30, 8, 7,       array:  .word   10, 5, 30, 8, 7, 14, 22, 31, 3, 6
    14, 22, 31, 3, 6};      arrend:
                            bye:    .asciiz "The sum is "
int *p, sum;                        .text
                                    .globl  main
void main()  {              main:
  sum = 0;                          li      $9,0        # sum = 0
  p = array;                        la      $8,array    # p = array
                                    la      $10,arrend  # addr of end of array
  while (p < array+10)      loop:   bgeu    $8,$10,done # compare ptr w/ end addr
   {                                lw      $12,($8)    # get a[i]
     sum += *p;                     add     $9,$9,$12   # add element to sum
     p++;                           addi    $8,$8,4     # p++ (increments by 4)
   }                                b       loop
                            done:   li      $2,4        # Call code for print_str
                                    la      $4,bye      # Addr of output string
  print("The sum is ");             syscall             # call print_str
                                    li      $2,1        # Call code for print_int
                                    move    $4,$9       # move ans to $4 to print
  print(sum);                       syscall             # call print_int
 }                                  jr      $31         # return
```

- **Subroutine Call Example**

```
        .text
#
#  This routine returns (via the standard calling convention, using $2) the
#  larger of the two values passed as arguments (in $4 and $5)
#
        .globl  max

max:    bge     $4, $5, xisit
yisit:  move    $2, $5
        b       return
xisit:  move    $2, $4
return: jr      $31
#
# The main program prompts the user for first one number, than a second,
# and prints out the larger of the two. It calls a routine called max, and
# both uses and expects standard register calling conventions.
#
        .globl  main
main:
     # Save away our return address so that $31 can be used as return
     # address for subroutine call
        sw      $31, retaddr    # save away our return address

     # Prompt for first number, input it, and put it into $5
        li      $2, 4           # print_str
        la      $4, str1
        syscall
        li      $2, 5           # read_int x
        syscall
        move    $5, $2          # put input value into $5 (NOTE: 2nd
                                # argument position is used to avoid
                                # conflict with $4 !!)
     # Get second number, put it into $4
        li      $2, 4           # print_str
        la      $4, str2
        syscall
        li      $2, 5           # read_int y
        syscall
        move    $4, $2          # put value into 1st arg position

     # Call the subroutine:  $8 = max(y,x)
        jal     max             # call max function
        move    $8, $2          # put answer in a safe place

     # Print results
        li      $2, 4           # print_str
        la      $4, str3
        syscall
        move    $4, $8          # print_int
        li      $2, 1
        syscall
        li      $2, 4           # print_str
        la      $4, str4
        syscall
```

```
        lw      $31, retaddr    # restore return address
        jr      $31             # return

        .data
retaddr:.space  4
str1:   .asciiz "Enter a value for x: "
str2:   .asciiz "Now enter a y value: "
str3:   .asciiz "The larger number is "
str4:   .asciiz "\n"
        .end
```

- **The "Hello, World" Program**

```
#
# Printing the message "Hello, World" is usually everybody's
#           first program in a new language!
# (This program is used as the example in the SPIM handout)
#
        .data
msg:    .ascii  "Hello, World"
        .byte   '\n',0
#
# The above can more easily done with:
#       .asciiz "Hello, World\n"

        .text
        .globl  main
main:   li      $2, 4  # system call code for "print string"
        la      $4, msg        # addr of string into $a0
        syscall
     # use the "exit" syscall to terminate this time
        li      $2, 10 # call code for exit
        syscall
```

Notes — MIPS R2000

Assembly Directives

An assembly directive is an assembly language statement that does not directly generate a machine operation, but does tell the assembler to perform some action. Most of the directives tell the assembler how to set up data values in memory.

`.text`

Specifies that the code that follows it is to be treated as machine instructions (placed into the `text` program segment). If there is more than one `text` segment in a program, they are all appended together into a single contiguous segment.

> **NOTE:** The word "text" might imply something that is readable by humans. In fact, `text` here means the binary code for the machine program, which is definitely NOT readable!

`.data`

Specifies that the code that follows is to be treated as data (placed into the `data` program segment. This is conventionally placed at the end of the assembly language program, but does not need to be. If there is more than one `data` segment in a program, they are all appended together into a single contiguous segment.

`.globl` *name*

Specifies that the specified name should be global, and therefore can be referenced from other files. In particular, the symbol `main` should be the label on the start of any stand-alone program, and should be made global.

Example - Typical program structure:

```
        .text
        .globl   main
 main:  .....          # program code starts here
          .
          .
        .data
    a:  .word   0      # variables (data) for the program
    b:  .word   1
    c:  .word   10
```

`.space` *n*

Reserves an area of bytes of (uninitialized) memory. Must be in the data segment. Used to allocate uninitialized variable and array space.

Example:

```
arr: .space  400  # Allocate 400 bytes (100 words) of memory, at the
                  # address known symbolically as "arr"
```

`.word` *val1*, *val2*, ...
`.half` *val1*, *val2*, ...
`.byte` *val1*, *val2*, ...

Reserves a word (32 bits), halfword (16 bits), or byte (8 bits) of storage for each value listed.

Example:

```
wvals: .word  10, 20, 30     # Allocate three words of memory with...
                             #   ...the values 10, 20, and 30 in them
hvals: .half  0x10, 0x20     # Allocate two halfwords, values specified in hex
arr:   .word  1,2,3,4,5,6,7,8,9,10 # Ten element array, initialized with values
str:   .byte  65, 66, 67,0   # Allocates space for the string "ABC"
                             # (for a better way, see below)
```

`.ascii` "*string*"
`.asciiz` "*string*"

Allocates storage for the ASCII string, enclosed in double quotes, that follows the directive. `.asciiz` specifies that the string is to be terminated with a zero byte, as in C. Special characters can be specified C-style (`'\n'` - newline, `'\t'` - tab, `'\0'` - NULL byte, etc.)

Example:

```
str1: .ascii "this is a string"  # non NULL terminated string
      .byte 0                    # NULL byte added explicitly
str2: .asciiz "this is a string" # does the same as the two previous lines
```

`.align` *n*

Tells the assembler to start the next field on a 2^n byte boundary. Value of n should be 1, 2, or 3. Used to insure that values are lined up in memory properly. This is especially useful to insure proper alignmemt after character data – MIPS *requires* that values start on a multiple of the size of the data item.

Example:

```
str:  .asciiz "Odd string"  # allocates 11 bytes (incl. NULL), an odd number
      .align  2             # insure next value is on a word boundary
val:  .word   27            # allocate one word on word boundary
```

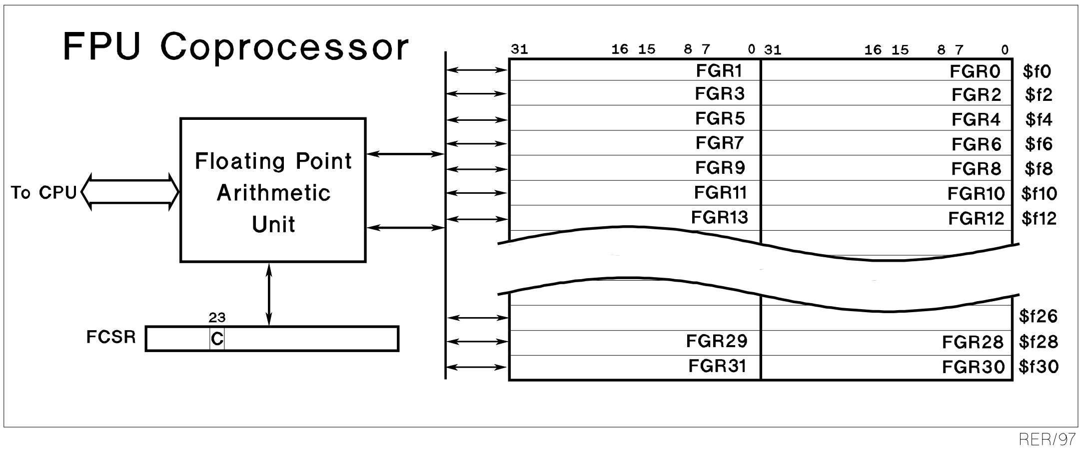
CS 245 Computer Organization

Notes — MIPS R2000

Floating Point Instructions

The MIPS processor implements floating point operations using a *coprocessor* called the FPU. It operates independently from, but in synergy with, the CPU. The FPU (in some places referred to as "Coprocessor 1") has its own set of registers. Instructions which specify floating point operations are passed by the CPU to the coprocessor for execution. The instructions use the same binary formats as "regular" MIPS instructions (the *J-type* format is not used.)

• MIPS Floating Point Processor - Programmer's Model



The FPU contains 16 registers, each of which is 64 bits (double) wide. They are numbered as if they were organized as 32 registers of 32 bits each – in fact only the even numbered registers can be specified in most instructions. They are specified as `$f0`, `$f2`, etc.

Like the CPU, the FPU is a load/store architecture. One difference is the presence of a Control/Status register – the result of a compare instruction is kept in this register instead of in a general purpose register. To perform a "branch on condition" operation, you first do a compare instruction, then use either a `BC1T` or `BC1F` (Branch on Coprocessor 1 True/False) instruction.

• Assembler/Instruction Syntax

The FPU instructions follow the same form as other MIPS instructions. The following additional abbreviations are used in the instruction descriptions below:

`fs`, `ft`, `fd` - represents FPU source/destination registers
`fmt` - specifies the data format:
- `s` - single precision Floating pt
- `d` - double precision Floating pt
- `w` - integer (convert instructions only)

• Load/Store, Data Movement Instructions

Assembler Syntax:

Loads		Stores		Data Movement	
`lwc1`	`fd, mem`	`swc1`	`fs, mem`	`mov.`*fmt*	`fd, fs`
`ldc1`	`fd, mem`	`sdc1`	`fs, mem`	`mtc1`	`rd, fs`
`l.`*fmt*	`fd, mem`†	`s.`*fmt*	`fs, mem`†	`mfc1`	`rt, fd`

Instruction format: Load/stores: I-type, Moves: R-type

These instructions move floating point values from memory to/from FPU registers, and between registers. The `l.`*fmt* and `s.`*fmt* are pseudoinstructions which provide a syntax for load/store similar to other FPU instructions. The `mov.`*fmt* instruction moves data between FPU registers, while `mtc1` and `mfc1` move values between the CPU and FPU registers. Note that when a memory location is specified using *base* addressing (of the form `offset(reg)`), a CPU register (**NOT** an FPU register) is used in the address operation.

Examples:

```
lwc1  $f8, val1     # load value of memory location val1 into FPU register 8
l.s   $f8, val1     # pseudoinstruction version of above
s.s   $f10, 100($9) # store single precision fp value at [R9] + 100
mov.s $f8, $f4      # copy between two FPU registers
mtci  $8, $f6       # copy from CPU reg 8 to FPU reg 6
```

• Arithmetic and Conversion Instructions

Assembler Syntax:

`add.`*fmt*	`fd, fs, ft`	`sub.`*fmt*	`fd, fs, ft`
`mul.`*fmt*	`fd, fs, ft`	`div.`*fmt*	`fd, fs, ft`
`neg.`*fmt*	`fd, fs`		
`cvt.`*fmt*`.`*fmt*	`fd, fs`		

Instruction format: R-type

The arithmetic instructions perform the indicated operation between floating point values. Note that there is no problem with `mul` and `div` with operand sizes, as there is with integers – the result is simply re-normalized (exponent adjusted) to accommodate any change in magnitude. The `cvt` instructions perform a conversion **to** the first `fmt` **from** the second `fmt`. Note that either `fmt` can be `w`, indicating a conversion to/from integer (word).

Examples:

```
add.s   $f10, $f8, $f6   # compute F10 = F8 + F6 (single precision)
neg.d   $f12, $f10       # F12 = - F10 (double precision)
cvt.d.s $f18, $f16       # F18 = (double precision) F16
cvt.s.w $f20, $f24       # convert INTEGER in F24 to single...
                         # ... precision floating pt in F20
```

• Comparison and Branch Instructions

Assembler Syntax:

```
c.eq.fmt   fs, ft
c.lt.fmt   fs, ft
c.le.fmt   fs, ft

bc1t       label
bc1f       label
```

Instruction format: Compares: R-type, Branches: I-type

The compare instructions set the *Condition* bit in the FPU *Control/Status* Register, based on the results of the specified comparison. The branch instructions perform a branch depending if the result of the last floating point comparison was `TRUE` (for `bc1t`) or `FALSE` (for `bc1f`).

Examples:

```
c.eq.s   $f8, $f10    # Is F8 == F10? If so, set Condition bit = 1
bc1t     equl         # if so, then branch to label equl

c.le.s   $f12, $f14   # This sequence implements a "branch if ...
bc1f     greater      # ... greater than" operation
```

• Additional Assembler Directives

Assembler Syntax:

```
.float   val1, val2, ...
.double  val1, val2, ...
```

These assembler directives allocate 4 and 8 bytes, respectively, for each value in the list.

• FPU Register Conventions

As with the CPU registers, a set of conventions has been established for using the FPU registers. The table below summarizes these conventions.

MIPS FPU Register Usage Conventions

Register	Description/Usage
`$f0-$f2`	Return results from functions
`$f4-$f10`	"Temporary" - unrestricted use - not saved during a call
`$f12-$f14`	Used to pass the first two floating point arguments to a function
`$f16-$f18`	More temporary
`$f20-$f30`	"Saved" - will be saved across a call, BUT must be saved (usually on stack) before being used, restored before return