Acknowledgements

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

References


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.

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.
  80 - 831 - 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-bit)
          halfword (16-bit)
          byte (8-bit)
    Number: 32, specified by $80 - 831$, 80 always contains the value 0, some of the other registers are reserved for special purposes by convention, $88 - 825$ 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-bit) each, sometimes treated as a single 64-bit register
    - Program Counter:
      Function: contains address of next instruction to be executed
      Size: one word (32-bit)

• 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 (±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 0x12345678 (hex) starting at location 1000, bytes 1000 - 1003 will hold the values shown for each order:

<table>
<thead>
<tr>
<th>Big - Endian</th>
<th>Little - Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000: 01 hex</td>
<td>1000: 67 hex</td>
</tr>
<tr>
<td>1001: 23 hex</td>
<td>1001: 45 hex</td>
</tr>
<tr>
<td>1002: 45 hex</td>
<td>1002: 23 hex</td>
</tr>
<tr>
<td>1003: 67 hex</td>
<td>1003: 01 hex</td>
</tr>
</tbody>
</table>

  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.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Syntax</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$n$ or ASYMn</td>
<td>$EA = [n] + offset$</td>
</tr>
<tr>
<td>Relative</td>
<td>RSYMn or RSYMn(PC)</td>
<td>$EA = [PC] + offset$</td>
</tr>
<tr>
<td>Register Indirect</td>
<td>($n$)</td>
<td>$EA = [n]$ + offset$</td>
</tr>
<tr>
<td>Base</td>
<td>offset($n$)</td>
<td>$EA = [n]$ + offset$</td>
</tr>
</tbody>
</table>

  Notes: The following symbols are used in the table above:
  **EA** - Effective Address
  **$n$** - General Purpose Register
  **PC** - Program counter
  **RSYMn** - Relative Symbol
  **ASYMn** - Absolute Symbol
  **$EA$** - Effective Address

  **Immediate** - used in immediate instructions

  **Operand (data) is contained in instruction**

- **Pictorial View**

  Register

  $\text{op} \quad \text{rs} \quad \text{rt} \quad \text{rd}$

  $\text{rs}$ is the operand (contains the data)

  Immediate - used in immediate instructions
**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**

<table>
<thead>
<tr>
<th>Memory Read</th>
<th>Memory Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAR ← address</td>
<td>MAR ← address</td>
</tr>
<tr>
<td>R/W ← read</td>
<td>R/W ← write</td>
</tr>
<tr>
<td>MDR ← Mem[MAR]</td>
<td>MDR ← data</td>
</tr>
<tr>
<td>dest ← MDR</td>
<td>Mem[MAR] ← MDR</td>
</tr>
</tbody>
</table>

**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 Effective Address (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 (read) 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 comparison 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)

\[
\begin{array}{l}
\text{fetch: } \quad MAR \leftarrow PC \\
\quad MDR \leftarrow \text{Mem}[MAR] \\
\quad IR \leftarrow MDR \\
\text{execute: } \quad i/f \text{ instr = Load} \\
\quad MAR \leftarrow EA \\
\quad MDR \leftarrow \text{Mem}[MAR] \\
\quad \#d \leftarrow MDR \\
\quad PC \leftarrow PC + 4 \\
\quad i/f \text{ instr = Store} \\
\quad MAR \leftarrow EA \\
\quad MDR \leftarrow \#d \\
\quad Me[m][MAR] \leftarrow MDR \\
\quad PC \leftarrow PC + 4 \\
\quad i/f \text{ instr = 0 (branch unconditional)} \\
\quad PC \leftarrow PC + \text{ of } \text{ set} \\
\quad i/f \text{ instr = 0x (branch conditional)} \\
\quad \text{if } \#x \text{ cc } \#x \text{ PC} \leftarrow PC + \text{ of } \text{ set} \\
\quad \text{else } PC \leftarrow PC + 4 \\
\quad i/f \text{ instr = or r1 r2 logical} \\
\quad ALU1 (\text{temporary ALU register}) \leftarrow \#x8 \\
\quad ALU2 \leftarrow \#x4 \\
\quad ALU3 \leftarrow ALU1 \text{ op } ALU2 \\
\quad \#d \leftarrow ALU3 \\
\quad PC \leftarrow PC + 4 \\
\end{array}
\]

(all other instructions follow)

CS 245

Notes — MIPS R2000

**Instructs**

- **MIPS and RISC Architectures**

The MIPS R2000 is a good example of a Reduced Instruction Set Computer (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 L-type format is used for immediate instructions. The J-type is used for the unconditional jump instructions. The bit encodings for each type is:

\[
\begin{array}{l}
\text{opcode} \quad \text{rs} \quad \text{rt} \quad \text{shamt} \quad \text{funct} \quad \text{offset/immediate} \quad \text{jump address} \\
\text{R-type} \quad \text{L-type} \quad \text{J-type}
\end{array}
\]

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** - 6 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 in 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 *m*

  *dine* instruction generated for a pseudoinstruction, NOT the entire pseudoinstruction.

- By convention, register 1 (71) is used by the assembler to expand pseudoinstructions into a sequence of machine instructions. Therefore, you must be careful when using 71 – it is best to avoid using it.

- It is sometimes possible to recode 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 operand #comment
```

- *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.

- *Opcode* for a MIPS instruction or pseudoinstruction.

- 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
- *imm* - specifies a memory address
- *im* - is an immediate (as immediate operand)
- *sa* - is a constant (as immediate operand)
- *label* - is a memory address, normally a label on an instruction.
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 bitwise 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:

```assembly
add $10, $8, $9 # compute R0 = R8 + R9
or $16, $20, $3 # compute bitwise inclusive-or R16 = R8 OR R20
neg $12, $13 # Take negative of value in R13, put into R12
add $14, $0, $0 # TRUC 1: clear R14. REMEMBER: R0 is always zero.
add $12, $13, $0 # TRUC 2: make sure R12 = R13
```

### Arithmetic/Logical Immediate Instructions

**Assembler Syntax:**

```assembly
addi rd, rs, imm addiu rd, rs, imm ori rd, rs, imm
```

**Instruction formats:**

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:**

```assembly
add $10, $8, 9 # compute R0 = R8 + 9 Note the value 9, NOT R9
```

### Shift and Rotate Instructions

**Assembler Syntax:**

```assembly
ror rd, rs, rt ror rd, rs, rt
srl rd, rs, sa srlv rd, rs, sa
sll rd, rs, rt sllv rd, rs, rt
```

**Instruction formats:**

The ror and ror instructions rotate the bits within rs right (ror) or left (rol) 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 srlv and sllv). With the arithmetic shifts (sra and sral), the sign (SMB) 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:**

```assembly
ror $10, $8, 9 # R10 = R8 rotated right by amount specified in R9
sll $16, $20, 0 # R16 = R20 shifted left 0 bits (multiply by 2^0)
```

### Comparison Instructions

**Assembler Syntax:**

```assembly
seq rd, rs, rt? sge rd, rs, rt?
slt rd, rs, rt? sgt rd, rs, rt?
sleu rd, rs, rt? sgeu rd, rs, rt?
```

**Instruction formats:**

These instructions perform a comparison op of the operands contained in the operands 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:**

```assembly
j label
b label?
```

**Instruction formats:**

Jump - $type, Branch - $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 pseudo-instruction. For the "?" instructions (bgez, bgtz, etc.), the register contents are compared with 0. Most of the conditional branches are pseudo-instructions, composed of a "set" (comparison) instruction, followed by a beq or bne instruction.

**Examples:**

```assembly
add: b down # skip instructions to label 'down'
```

```assembly
down: add .... # next instruction to execute after branch
```

```assembly
seq: sge $1, $8, $9 # Is R8 < R9? If so, set R10 = 1 and...
```

```assembly
bnez $1, less # ... branch to label 'less' if true
```

```assembly
bnez: bnez $8, $9, less # The pseudo-instruction equivalent of bnez
```
Subroutines in Assembly Language

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

1. A mechanism for returning to the calling point after a subroutine is called.
2. Stack or local variables that are declared within a subroutine are accessible only by that subroutine.
3. Variables that are declared within the subroutine are accessible only by that subroutine.

With the MIPS only the first feature is supported by hardware - the other are implemented by the programmer. This means that the programmer must think in terms of calling subroutines by conventions - that is, an agreement among programmers that certain groups of registers will be used in certain ways. In this chapter we will be using conventions. We will use some variable use conventions for subroutines.

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

### Miscellaneous Instructions

#### Assembly Syntax

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jr</td>
<td>Jump to subroutine (but not to a function)</td>
</tr>
<tr>
<td>jal</td>
<td>Jump to subroutine and pass arguments</td>
</tr>
</tbody>
</table>

#### Examples

```
  jr   jr        # Jump to subroutine
  jal  jr        # Jump to subroutine and pass arguments
```

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

1. If your subroutine uses registers other than the 7 (R0-R6), then you can use the temporary (T1-T32) and saved (S0-S31) registers. However, you cannot use the register that the system associates with your subroutine.
2. If your subroutine uses registers other than the 7 (R0-R6), then you cannot use the register that the system associates with your subroutine. Instead, you must either use the saved registers (R7-R31) or allocate registers to your subroutine.

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

1. If your subroutine uses registers other than the 7 (R0-R6), then you can use the temporary (T1-T32) and saved (S0-S31) registers. However, you cannot use the register that the system associates with your subroutine.
2. If your subroutine uses registers other than the 7 (R0-R6), then you cannot use the register that the system associates with your subroutine. Instead, you must either use the saved registers (R7-R31) or allocate registers to your subroutine.

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

1. If your subroutine uses registers other than the 7 (R0-R6), then you can use the temporary (T1-T32) and saved (S0-S31) registers. However, you cannot use the register that the system associates with your subroutine.
2. If your subroutine uses registers other than the 7 (R0-R6), then you cannot use the register that the system associates with your subroutine. Instead, you must either use the saved registers (R7-R31) or allocate registers to your subroutine.
3. If you use any of the save registers (8x0-8x7 or 816-823), you must save their contents before you use them (callee-save 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.

Computer Organization

CS 245

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. Enforce 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. Enforce 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 equivalent. 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 program. Identify the values which should be locally accessible in each routine, and only allow that routine to access them. Make any necessary global data visible—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 ints, represented as 32-bit words, unless otherwise noted.

#### Storage Allocation and Assignment

<table>
<thead>
<tr>
<th>C Statements</th>
<th>MIPS Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x, a = 27, b = 0x36, y; x</td>
<td>.data # Variables go in data segment</td>
</tr>
<tr>
<td>y = a; ma: sw $0, x</td>
<td>.text # Label $0 = 0</td>
</tr>
<tr>
<td>y = b; ma: sw $0, y</td>
<td>lw $0, a # Get a from memory and ...</td>
</tr>
<tr>
<td>x = b; ma: sw $0, b</td>
<td>lw $0, x # ... put it into y</td>
</tr>
<tr>
<td>x = y</td>
<td>lw $0, x # Get b ...</td>
</tr>
<tr>
<td>sw $0, x</td>
<td>add $0, $0, $0 # Compute x = b</td>
</tr>
<tr>
<td>ma: sw $0, x</td>
<td># Put x back into memory</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>C Statements</th>
<th>MIPS Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>int r, s, t;</td>
<td>li $0, 3 # Load Values for r, ...</td>
</tr>
<tr>
<td>li $1, 4</td>
<td>li $12, 5 # ... t using load immediate</td>
</tr>
<tr>
<td>to $1 = r + s - t;</td>
<td>add $13, $10, $11 # compute total</td>
</tr>
<tr>
<td>to $1 += 10;</td>
<td>sub $13, $13, $12</td>
</tr>
<tr>
<td>to $1 += 10;</td>
<td>add $13, $13, $10</td>
</tr>
</tbody>
</table>

#### Decision Statements

• The if-then Statement

There are no "program blocks" (i.e., the equivalent of 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 an assembly language implementation—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  

The if-then-else statement 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  

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 $k - k$ are allocated to registers $16 - 21$ respectively.

```c
switch (k) {
  case 0: f = i + j; break;
  case 1: f = g + h; break;
  case 2: f = g + h; break;
  case 3: f = i - j; break;
  case 4: f = g + i; break;
  default: ...
}
```

• Loop Structures

Pre-Test Loop  

The following code sequence calculates the value $res = b*c*d*e$ 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
C Statements  

MIPS Assembly Code
res = 1;  
while ($ctr > 0$) {
  res = res * base;
  $ctr$--; 
}
```

Post-Test Loop  

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

```c
C Statements  

MIPS Assembly Code
```

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.

```c
C Statements  

MIPS Assembly Code
```

```
21
```
• 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.

```c
int array[10] =
    {10, 5, 30, 8, 7, 14, 22, 31, 3, 6};
array: .word 10, 5, 30, 8, 7, 14, 22, 31, 3, 6
byt: .asciiz 'The sum is ';
int i, sum;
.text
.globl main
void main() {
    i = 0;
    sum = 0;
    while (i < 10) {
        li $i, 0  # i = 0
        li $i, 0  # sum = 0
        addi
        bge $i, 10, done
        add $i, $i, 2  # Compute i+4
        lw $i2, array($i)  # Get a[i]
        add $i2, $i2, $i  # Add element to sum
        add $i, $i, 1  # i++
        b loop
    } done:
    li $i2, 4  # Call code for print_int
    la $i4, bye  # Addr of output string
    syscall
    li $i2, 1  # Call code for print_int
    move $i4, $i  # Move ans to $i4 to print
    syscall
    jr $i3  # return
}

print('The sum is ');
syscall
print(sum);
}
```

• 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.

```c
int array[10] =
    {10, 5, 30, 8, 7, 14, 22, 31, 3, 6};
array: .word 10, 5, 30, 8, 7, 14, 22, 31, 3, 6
int ep, sump;
.text
.globl main
void main() {
    li $i9, 0  # sum = 0
    la $i8, array  # p = array
    la $i10, array  # Addr of end of array
    lw $i12, array($i8)  # Get a[i]
    add $i9, $i9, $i10  # add element to sum
    add $i8, $i8, 4  # p++ (increments by 4)
    b loop
} done:
    li $i2, 4  # Call code for print_int
    la $i4, bye  # Addr of output string
    syscall
    li $i2, 1  # Call code for print_int
    move $i4, $i  # Move ans to $i4 to print
    syscall
    jr $i3  # return
}
```

• Subroutine Call Example

```c
   .text
   .globl max
max:
    bge $i4, $i5, xinit
xinit: move $i2, $i5
b return
xinit: move $i2, $i4
return: jr $i3

# The main program prompts the user for first one number, 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 $i3 can be used as return
    # address for subroutine call
    move $i31, $ra  # Save away our return address

    # Prompt for first number, input it, and put it into $i4
    li $i2, 4  # print_int
    move $i4, $i2
    syscall
    li $i2, 5  # print_int
    syscall
    move $i5, $i2  # Put input value into $i5 (MSB of 2nd
    # argument position is used to avoid
    # conflict with $i4 !)

    # Get second number, put it into $i4
    li $i2, 4  # print_int
    syscall
    li $i4, $i2
    syscall
    li $i2, 5  # print_int
    syscall
    move $i5, $i4  # Put value into 1st arg position

    # Call the subroutine: $i5 = max($i4, $i5)
    jal max
    la $i5, max
    syscall
    li $i2, 4  # print_int
    syscall
    li $i4, $i3
    syscall
    move $i5, $i4
    syscall
    li $i2, 4  # print_int
    syscall
```
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 MIPS handout)

.data
msg: .asciz "Hello, World"
bytes "$m",0
#
# The above can more easily be done with:
# .asciz "Hello, World"

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

CS 245

Computer Organization

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 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

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 variables 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) storage for each value listed.

Example:
vals: .word 10, 20, 30 # Allocate three words of memory with...
# ...the values 10, 20, and 30 in them
halfs: .half 0x10, 0x20 # Allocate two half-words, 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"
.ascii "string"

Allocates storage for the ASCII string enclosed in double quotes, that follows the directive. .ascii specifies that the string is to be terminated with a null 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" # use NULL terminated string
# H2L byte added explicitly
str2: .ascii "this is a string" # 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 alignment after character data - MIPS requires that values start on a multiple of the size of the data item.

Example:
str: .ascii "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

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 synchrony 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 format as 'regular' MIPS instructions (the J-type format is not used.)

• MIPS Floating Point Processor - Programmer's Model

The FPU contains 36 registers, each of which is 64 bits (double) wide. They are numbered as if they were organized in 32 registers of 32 bits each - in fact only the even numbered registers can be specified in most instructions. They are specified as $f0, f27, etc.

Like the CPU, the FPU is a Harvard 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 beq or beqz (Branch on equaler register) 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:

- $f$, $ft$, $fd$ - represent FPU source/destination registers
- $mt$ - specifies the data format:
  - $s$ - single precision floating pt
  - $d$ - double precision floating pt
  - $l$ - integer (convert instructions only)
• Load/Store, Data Movement Instructions

**Assembler Syntax:**

<table>
<thead>
<tr>
<th>Location</th>
<th>Storage</th>
<th>Data Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>fd, mm</td>
<td>mov.f $t0, $f0</td>
</tr>
<tr>
<td>store</td>
<td>fs, mm</td>
<td>mov.f $t0, fd</td>
</tr>
</tbody>
</table>

**Instruction format:** Load/store: I-type, Move: I-type

These instructions move floating-point values from memory to/from FPU registers, and between registers. The load and store instructions perform similar to other FPU instructions. The mov.f instruction moves data between FPU registers, while movl and movt move values between the CPU and FPU registers. Note that when a memory location is specified using base address (if of the form offset(reg)), a CPU register (not an FPU register) is used in the address operation.

**Examples:**

- load $t0, $v0  # load value from memory location into FPU register $t0
- store $t0, $v0  # store value from FPU register to memory location
- mov.f $t0, $f0  # move value from FPU register to FPU register
- movl $t0, $f0  # move value from CPU register to FPU register

• Arithmetic and Conversion Instructions

**Assembler Syntax:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>add.f</td>
<td>add.f $t0, $f1, $f2</td>
</tr>
<tr>
<td>sub.f</td>
<td>sub.f $t0, $f1, $f2</td>
</tr>
<tr>
<td>mul.f</td>
<td>mul.f $t0, $f1, $f2</td>
</tr>
<tr>
<td>div.f</td>
<td>div.f $t0, $f1, $f2</td>
</tr>
<tr>
<td>cvt.f.f</td>
<td>cvt.f.f $t0, $f1</td>
</tr>
</tbody>
</table>

**Instruction format:** R-type

The arithmetic instructions perform the indicated operation between floating-point values. Note that there is no problem with real and imaginary operands, 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 from the first fpt to the second fpt. Note that either fpt can be v, indicating a conversion to/from integer (vector).

**Examples:**

- add.f $f0, $f1, $f2  # add two floating-point values
- sub.f $f0, $f1, $f2  # subtract two floating-point values
- mul.f $f0, $f1, $f2  # multiply two floating-point values
- div.f $f0, $f1, $f2  # divide two floating-point values
- cvt.f.f $f0, $f1  # convert from double to single precision

• Comparison and Branch Instructions

**Assembler Syntax:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.eq.f</td>
<td>c.eq.f $f1, $f2</td>
</tr>
<tr>
<td>c.lt.f</td>
<td>c.lt.f $f1, $f2</td>
</tr>
<tr>
<td>c.le.f</td>
<td>c.le.f $f1, $f2</td>
</tr>
</tbody>
</table>

**Instruction format:** Compare: I-type, Branch: I-type

The compare instructions set the condition code in the FPU Status Word Register, based on the result of the specified comparison. The branch instructions perform a branch depending if the result of the last floating point comparison was TRNE (for bteq) or TFLZ (for bleq).

**Examples:**

- c.eq.f $f0, $f1  # if $f0 == $f1 then set condition bit = 1
- beq label  # if equal, branch to label
- bleq label  # if less than or equal, branch to label

• Additional Assembler Directives

**Assembler Syntax:**

... float $v0, $v1, ...
... double $v0, $v1, ...

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.

<table>
<thead>
<tr>
<th>MIPS FPU Register Usage Conventions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
</tr>
<tr>
<td>$f0-$f15</td>
</tr>
<tr>
<td>$f16-$f19</td>
</tr>
<tr>
<td>$f20-$f23</td>
</tr>
<tr>
<td>$f24-$f31</td>
</tr>
<tr>
<td>$f32-$f39</td>
</tr>
</tbody>
</table>