Chapter III
Instructions and assembly programming

III.1 Instruction-set architecture: the hardware-software interface

Processor instructions define the most important hardware-software interface: software programs use the processor instructions to carry out arbitrary data processing, while electronic circuits interpret those instructions and perform the requested actions in hardware. The *instruction set* abstracts away the details of the hardware in much the same way as an object in a high-level programming language hides its internal operation from its users.

There are potentially many ways to define this hardware-software interface depending on how much 'work' the hardware is allowed to perform on its own in order to carry out the actions required by an instruction. Different brands of processors have different instruction sets because their designers have made different decisions about this hardware-software boundary. Instruction sets are often classified into two groups: RISC (reduced instruction set computer) and CISC (complex instruction set computer).

A single program statement such as `a = b + c;` might compile to one CISC instruction, but to several RISC instructions. Compiled programs for CISC machines are therefore smaller. For that reason, CISC ruled the roost in the 70s when memory was expensive to built and, thus, limited in size. However, as memory got cheaper, and the compiler technology evolved, the advantages of CISC disappeared. Because RISC instructions are simpler, RISC processors are easier to design – very important in the competitive processor market. Despite compiled programs containing more instructions, RISC machines often run faster too, as the simple instructions execute very fast. For that reason, new processor designs are RISC designs. The big exception of course is the Intel 80X86 architecture used in the PC. Dating from the 70's, and still in use, the Intel architecture (IA32) is the only general purpose CISC design still around.

Recall that instructions (and data) are represented as binary numbers. Although using the hex equivalent would be shorter, a numerical representation would still be very tedious to use. Therefore a symbolic representation of instructions was developed and this symbolic language is called *assembly language*. Note that there is a one-to-one correspondence between assembly instructions and machine instructions, unlike high-level languages in which a single statement corresponds to a number of machine instructions[1].

For this course, the MIPS instruction set will be used, which is typical of a RISC processor. There is a separate handout with the MIPS instruction subset that we will use for this course. The full instruction set can be found in the appendix A of the textbook, which is also available at `http://www.cs.wisc.edu/~larus/HP_AppA.pdf`.

III.2 Some basic instructions

Computers were created to do calculations, therefore all computers have instructions to perform basic arithmetic operations. For example `add a, b, c` is the MIPS instruction that adds variables `b`, `c` and stores the result to `a`. Similarly `sub a, b, c` calculates $b - c$ and stores the result to `a`.

The above examples use the symbolic, assembly, representation of MIPS instructions; we will examine the binary representation of instructions later. In general MIPS arithmetic and other operations are of the form: "operation type" "destination variable", "first operand", "second operand". The distinction between first and second operand can be important for (non-commutative) operations such as subtraction.

Note that only two operands are allowed in MIPS (and most other processor) instructions. To perform more complex arithmetic operations multiple instructions are required and, sometimes, temporary variables hold intermediate results. For example to compute x=(a+b)-(c+d), the following program would be used[2]:

```
add t0, a, b    # add a to b, store to t0
add t1, c, d    # add c to d, store to t1
sub x, t0, t1   # Subtract t1 from t0, store to x
```

Although computers usually operate on full words, sometimes specific bits or bit fields need to be extracted. In order to perform these tasks, processors include instructions that perform *logical operations* and shifts. MIPS provides `and`, `or`, `nor` and `xor` instructions, which perform the AND, OR, NOR, XOR logical operation on a bit-by-bit basis, respectively.

There are two main types of shift operations: shift left logical, `sll a, b, shamnt` and shift right logical, `srl a, b, shamnt`. The first stores to `a` the result of shifting `b` to the left by as many bits as `shamnt`, a constant, defines. `srl` does the same in the opposite direction. Recall the distinction between arithmetic and logical shift right from the previous chapter.

[1]Modern assembly languages sometimes define pseudo-instructions that get translated to one or more real instructions.

[2]In MIPS assembly, # starts a comment which continues to the end of the line.

III.2.1 Registers

Most processors place restrictions on which operands can be used in instructions and where the results can be stored. Nearly all RISC processors, incl. MIPS, allow their instructions to operate on (and store the results to) *registers* only. Registers are storage locations inside the processor which hold program variables. The size of a register (in bits) is equal to a word and the total number of registers is relatively small (32 for MIPS).

In view of the above, when writing assembly programs we use only register names instead of arbitrary variable names, as above. Although nearly any register can be used for any purpose, most MIPS programs follow the following convention: registers `$s0, $s1,...$s7` correspond to variables in C and Java programs, while temporary variables use registers `$t0,...$t9`. There are other registers with special roles which we will see later. For full details, consult figure A.6.1 (A.10 in 2nd edition) of appendix A of the textbook (also available on-line).

Note that the registers are 'visible' to the programmer, therefore constitute part of the hardware-software interface. Peculiar to MIPS, one register, `$zero` or `$0`, is hardwired to `0`, so MIPS ends up having only 31 registers available to programs. Of course the names given above to these registers are still symbolic names. The actual registers are simply numbered from `0` to 31 (i.e. 5 bits are enough to address them) and there is a well-known correspondence between the symbolic names and the register numbers. You can find this correspondence in the "MIPS reference data" handout, which comes from the course text book.

In addition to these general purpose registers, there is another register which is not directly accessible but essential for the operation of a processor. It is called the *program counter* or the *PC* for short and points to the current instruction that is being executed by the processor. We will see control transfer instructions that change the PC and therefore the program flow, later on.

III.2.2 Immediate operands

Sometimes one of the operands to an arithmetic operation is a constant known at compile time, for example when compiling `i++;`. The MIPS provides versions of the arithmetic instructions to support this, for example the *add immediate* instruction `addi r1, r2, n`. This adds the constant value `n`, encoded as part of the instruction itself, to the value in register `r2` and stores the result in register `r1`. `n` can be positive or negative and it is sign-extended to 32 bits before the addition. There are immediate types for the `and`, `or` and `nor` instructions too.

The `addi` instruction can also be used to load a small constant into a register, using `addi r1, $0, n`, because register `0` always contains `0`. What if we need to load a full 32-bit value into a register? MIPS provides an instruction *load upper immediate*, `lui r, n`, which loads the *top half* of register `r` with the 16-bit value `n`, setting the bottom half of `r` to `0`. The bottom half of the required 32-bit constant can then be added in with an `ori` instruction.

MIPS provides a convenient pseudoinstruction that does the above job. Simply write `li reg, immediate`; it gets translated to either one or two real MIPS instructions, depending on how many bits long the immediate is.

III.3 Getting at the data

Most programs use a lot more than the 32 variables a MIPS processor can store in its registers. Variables in a program are therefore stored in memory, not registers. But the MIPS, like most RISC processors, does not have instructions that do arithmetic operations on values stored in memory (CISC processors do usually have such instructions). The MIPS requires values to be loaded from memory into registers before they are operated on, and the result to be stored back into memory afterwards. RISC architectures are some times called *load-store* architectures for this reason.

In an object-oriented language, most variables are likely to be object instance variables – each object has its own copy of such variables. An object will be stored as a chunk of memory containing all its instance variables. This chunk of memory is set up when the object is constructed, i.e. when a statement such as `obj1 = new MyClass();` is executed. This allocation of a chunk of memory happens at *run time*, when the program is running, not at compile time – the memory chunk is said to be *dynamically allocated*. In fact, running the same program on different input may result in a different number of objects being constructed in a different order.

What this means is that if a program accesses an instance variable of an object, e.g. with an expression such as `obj1.avar`, the compiler cannot usually know at what addresses in memory that particular object's `avar` variable will be stored when the program is running. What it does know however is that the value `obj1` is a 32-bit pointer, or reference, to the object containing the variable wanted. This is because `obj1` is set to point at the newly allocated memory space for the object, wherever it might be, at the time the statement `obj1 = new MyClass()` is executed.

So the compiler can generate code to load the value of `obj1.avar` into a register by using code which accesses into the chunk of memory pointed at by the value in `obj1`, without the compiler actually knowing what the value of that pointer will be at run time.

The MIPS instruction that does this is called *load word*, and its symbolic form is `lw $s1, n($s2)`. The effect of this instruction is to take the value in register `$s2` and add to it the constant value `n`, which is coded into the instruction, and then to use this as an address into memory, finally loading four bytes from memory, from that address and the next three, into register `$s1`. The value in `$s2` is known as the *base address*, and `n` as the *offset*.

In our example, `$s2` would first be loaded with the pointer to the object, copied from the variable `obj1`. The constant `n` would be calculated by the compiler to be the correct number to offset into the object's memory chunk so that we fetch the variable `avar` rather than any of the object's other instance variables: each variable is stored at a different offset into the whole memory chunk, and the compiler knows these offsets as it knows what instance variables are declared for this object and in what order.

The MIPS also has a *store word* instruction, `sw $s1, n($s2)`, which stores the integer in register `$s1` into memory at the address `$s2 + n`. The instructions that access the memory are collectively called *data transfer instructions*.

Now, our original statement `a = b + c;` can be implemented by the following code (assuming that register `$s1` already points at a chunk of memory containing the three variables):

```
lw $t0, boffset($s1)
lw $t1, coffset($s1)
add $t2, $t0, $t1
sw $t2, aoffset($s1)
```

Note that the offset can be positive or negative as were the immediates in the arithmetic and logical instructions.

In most computer systems the memory is *byte-addressable*, i.e. each individual byte has a corresponding address. So, which one of the 4 bytes should be used as the address of a 32-bit word? We could decide to allow words to be stored at any byte address and use the address of the first byte as the word's address. This turned out to hinder the implementation of the memory sub-system, therefore, in most processors including MIPS, words must start at addresses which are multiples of 4. This is called the *memory alignment restriction*.

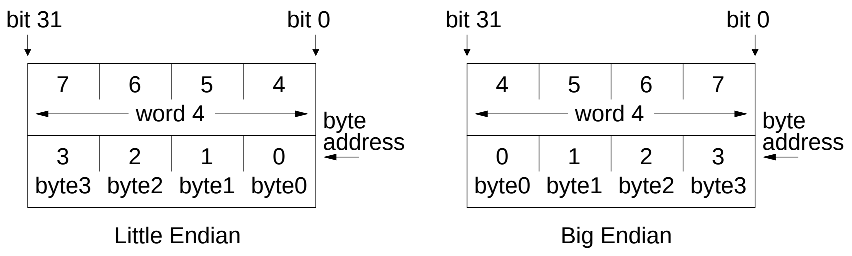

Figure III.1: Memory endianism.

There are two ways to store a word in a byte-addressable memory (whether aligned or not) depending on whether the least significant byte is stored at a lower or higher address than the least significant byte. Figure III.1 (from "ARM System Architecture", Steve B. Furber, Addison Wesley, 1997) shows the two cases. Note that in both cases a word is written with its most significant bit to the left, the difference is at which end the byte address `0` starts from. If it starts from the least significant end, the machine is called *little endian* (e.g. Intel x86), otherwise it is called *big endian* (e.g. Sun SPARC).

III.4 Instruction formats

So far we have only seen the symbolic representation of instructions, but in reality instructions are represented using bits. The layout of the binary representation of an instruction is composed of fields of binary numbers (*bit-fields*) and is called the *instruction format*.

The instructions of RISC processors have very rigid formats. For start all the instructions are of the same bit length, equal to one word. The same format and bit-fields are used for as many different instructions as possible. The rational behind this rigidness is that it makes it easier (and faster) to build the circuits that decode the instructions in order to perform the operations it commands.

The instructions we have seen so far use two formats: the R-format is used by the arithmetic and logical instructions (`add`, `or`, ...) and the I-format is used by data transfer and the immediate instructions.

The fields of the R-format are:

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	shamt	funct

opcode The basic operation of the instruction.

rs The first register operand. Zero for shift instructions.

rt The second register operand.

rd The destination register which stores the result.

shamt Shift amount for shift instructions. Zero for non shift instructions.

funct The specific function of the instruction.

The I-format looks like:

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate

opcode The basic operation of the instruction.

rs The first register operand for immediate instructions or the base register for data transfer instructions.

rt The destination register for immediate and load instructions; the register to be stored for store instructions

immediate A 16-bit 2's complement constant used as an operand in immediate instruction or as the offset for data transfers.

Opcodes and funct codes for MIPS instructions are given in a separate handout: "MIPS reference data".

III.5 Instructions for making decisions

Programs which simply consist of a straight-line sequence of statements operating on data are not very interesting. To be useful, programs need a way of choosing which statements to execute, depending on data values. Java provides a variety of statements to do this, including the `if`, `switch`, `while`, `do` and `for` statements.

Let's start with the java statement `if (i != j)` *thenStmnt* `else` *elseStmnt*. The MIPS provides a *conditional branch* instruction which can be used to implement the control choice here. The `beq r1, r2, label` instruction compares the values in the two registers `r1` and `r2`, and if they are equal, transfers control to the instruction labelled `label`, otherwise control proceeds to the next instruction in sequence. There is also a related instruction `bne r1, r2, label` which branches if the values in `r1` and `r2` are not equal.

The `beq` and `bne` instructions follow the I-format and the 16-bit immediate field of the instruction contains the *branch offset*, that is, the distance to branch to reach `label`, in 32-bit words, from the instruction following the `beq`. This 16-bit number is multiplied by 4, to turn it into a word offset[3], then sign-extended and added to the Program Counter if the branch should be taken.

Our `if` statement could be implemented as follows, assuming the variable `i` is in register `$s1`, and `j` in register `$s2`:

```
      beq $s1, $s2, L1 # branch if i and j are equal
      thenStmnt
L1:   elseStmnt
```

[3]Recall that instructions are words, therefore aligned at byte addresses which are multiples of 4.

There is actually a mistake in the above program; it is not a correct translation of the Java statement. The problem is that after the *thenStmnt* is executed, *elseStmnt* will also be executed if the condition was true. To correct this we must add an instruction following the *thenStmnt* which unconditionally branches over the *elseStmnt*. MIPS provides the *jump* instruction for this purpose, `j label`, which transfers control to the instruction labelled `label`. The correct translation of the above Java statement can now be expressed as:

```
      beq r1, r2, L1   # branch if i and j are equal
      thenStmnt
      j L2             # an unconditional branch
L1:   elseStmnt
L2:
```

The jump instruction has a different format to the instructions we have seen so far; it does not use any registers for operands or to store a result. This format is called the J-format:

31 26	25 0
opcode	address

The 26-bit address field defines the *target* of the jump. A minor detail here is that the remaining 4 bits of the target address come from the upper 4 bits of the PC. This is an example of an *absolute jump* instruction, where the instruction contains the actual address of the destination[4]. In contrast, the branch instructions are *relative branches*; the instruction contains a positive or negative offset to the destination of the branch.

While loops

The loop `while (count != 0)` *stmnt* can be implemented as follows, assuming that the variable `count` is kept in register `$s1`:

```
loop:  beq $s1, $zero, end   # branch out of loop if count == 0
       stmnt
       j loop                # unconditional branch to loop test
end:
```

If the expression in the loop test contains `boolean` variables, these can easily be handled, assuming they are stored in bytes (or words) with value 1 (`true`) or 0 (`false`). For example, the loop `while (flag1 && flag2)` *stmnt* would become (assuming the variables `flag1` and `flag2` are kept in registers `$s1` and `$s2` respectively):

```
loop:  beq $s1, $zero, end   # branch if flag1 is false
       beq $s2, $zero, end   # branch if flag2 is false
       code for stmnt
       j loop                # unconditional branch to loop test
end:
```

[4]Well, almost!

The `beq` and `bne` instructions are fine for comparing two values for equality, or a value for equality with `0`, but what about more complex arithmetic comparisons? The MIPS has an instruction *set if less than*, whose mnemonic form is `slt r1, r2, r3`, the effect of which is to set register `r1` to 1 if the value in `r2` is less than the value in `r3` and to set `r1` to `0` otherwise. In other words it returns the boolean value `r2 < r3` in `r1`. For example, the loop `while (i > j)` *stmnt* can be implemented (assuming `i` and `j` are kept in registers `$s1` and `$s2` respectively): *Comparisons*

```
loop:  slt $t0, $s2, $s1    # put the boolean value i > j in $t0
       beq $t0, $zero, end  # branch if that value is false
       stmnt
       j loop               # an unconditional branch
end:
```

There are a number of *branch on* … and *set if* … instructions in MIPS to simplify translation of high-level language boolean expressions and conditional statements into assembly code.

III.6 Method calls

A different kind of control transfer takes place when a method is called. The key point here is that the same method can be called from many different places in a program, and each time, we need to return to the statement after the method call when the method is complete. We need a way to remember from where we transferred control to the method.

The MIPS provides an instruction called *jump and link* to do this. Its mnemonic form is `jal label`, and its effect is to save the address of the following instruction in register `$ra` (r31), and to unconditionally transfer control to the instruction labelled `label`. This instruction uses the J-format and the target address is calculated in the same way as with the *jump* instruction above.

When the jump and link instruction is used to call a method, we then need an instruction to return from the method. The *jump register* instruction does this. Its mnemonic form is `jr r1`, and it simply sets the Program Counter to the value in register `r1`. We can now return from a method by using `$ra`.

III.6.1 Using a stack for method calls

What happens if we have nested method calls, i.e. we call one method, which then calls another? The jump and link instruction implementing the second call will put its return address into `$ra`, overwriting the return address for the first method call. We need a way to keep the first return address safe while we call the second method, and then put it back into `$ra` before we use a `jr $ra` instruction to return from the first method. If method calls are nested to several levels, we will need to save a sequence of return addresses, restoring them into `$ra` at the right times for the returns from each call.

Because the saved values are restored in the opposite order to which they are saved, an obvious data structure to save them in is a *stack*[5]. We can push each return address onto the stack as we arrive in a method and then pop it off the stack back into `$ra`, before we return from the method with a `jr r31` instruction.

A stack can be implemented as a block of memory words, which grows as we push new words onto it, and shrinks as we pop words. Often, programs use a stack which grows *downwards* in memory, so that each new word pushed occupies an address 4 bytes lower than the previous word. One register in the processor is usually set to point at the "top" of this stack, i.e. at the last word pushed on it. Pushing the contents of `$ra` can then be implemented as follows, using the notation `$sp` to represent the stack pointer register:

```
addi $sp, $sp, -4   # move stack pointer down by 4 bytes
sw   $ra, 0($sp)    # store $ra where the stack pointer points
```

Popping the value is done with:

```
lw   $ra, 0($sp)   # pop top of stack into $ra
addi $sp, $sp, 4   # move stack pointer up by 4 bytes
```

Method calls may use the stack for other things. For example the compiler may compile a method so that it saves the contents of processor registers that the method itself uses on the stack on entry to the method, and restores those values before returning to the calling code. That way, as far as the calling method is concerned, none of the registers are changed by the method call.

Most methods have *parameters*, the values passed into the method, and these can be passed by the calling code either placing them in a few of the processor registers (MIPS uses `$a0`,..., `$a3` for this purpose), or, if there are too many parameters to fit in registers, placing them on the stack. In either case, the method code can easily pick them up. If the method returns a value, that value can either be returned in a register (`$v0`, `$v1` for MIPS), or on the stack.

[5]See *Inf1B, Object Oriented Programming, lecture on data structures.*