CPS311 Lecture: Introduction to the MIPS Architecture and Assembly Language

Last revised June 21, 2021

Objectives:

1. To introduce the MIPS architecture
2. To introduce MIPS R-Type, immediate, and load-store instructions

Materials:

1. Levels of Structure Diagram with Highlighted Levels
2. IPS ISA Handout (will have been distributed before class)
3. Projectables of diagrams in handout
4. Connection to MIPS to demo gcc

I. Introduction

A. At the start of the course, we introduced the notion of levels of computer structure.

PROJECT Levels diagram with highlighted layers

1. We first looked at the digital Components level

2. Then we moved up the the Building Blocks level

3. We now move all the way up to the Architecture level - then we will go back down to the microarchitecture level, since in order to study implementation we need to understand both the things we are building with and what it is we need to build

B. For the next few weeks, then we will be studying the Machine Language level of system description. At this level, a computer system can be viewed as a memory, a set of registers, and a set of instructions for manipulating the information in the memory and registers.

1. Programs written at a higher level of system description (e.g. in a language such as C) are translated into primitive operations at this level.

2. This level is, in turn, implemented directly by hardware - i.e. the registers are arrays of flip-flops, addition is performed by full adders, etc.

3. The architectural description of a machine at this level is often referred to an Instruction Set Architecture (ISA).

C. For this portion of the course, we will be focussing on a particular instruction set architecture (ISA) known as MIPS.

1. It is not the goal of these lectures that you should become proficient MIPS assembly or machine language programmers.

2. Rather, we want to use MIPS as an example of a typical instruction set architecture.

   a) The MIPS architecture belongs to the general category of Reduced Instruction Set Computer (RISC) architectures.

      (1) As such, it is easier to learn than a Complex Instruction Set Computer (CISC) architecture such as the x86 architecture used by most laptops and PCs.

      (2) Although the x86 architecture (and its 64-bit extension) is not a RISC architecture, the current practice is to implement this architecture on top of a RISC core (there is a RISC inside the CISC).

   b) When we get to the actual details of implementing a CPU (computer organization), the implementation of a RISC architecture like MIPS is more comprehensible - and, indeed, we will discuss the implementation of the MIPS architecture in later lectures.

3. It is also the case that once you have become familiar with one instruction set architecture, it is much easier to learn another. (Once you learn to drive a Ford, driving a Chevy is easy.)

D. A bit of history

1. The MIPS architecture grows out of an early 1980's research project at Stanford University.

2. In 1984, MIPS computer corporation was founded to commercialize this research. However, CPU chips based on the MIPS architecture have been produced by a number of different companies, including LSI Logic, Toshiba, Philips, NEC, IDT, and NKK.

3. The MIPS architecture has passed through a series of evolutions, known as MIPS I, MIPS II, MIPS III, and MIPS IV.

   a) Each successive ISA is a superset of the preceding one - so anything found in MIPS I is also found in MIPS II, III, and IV, etc.

   b) The MIPS I and II ISA's were 32 bit architectures. MIPS III added 64 bit capabilities - but with the core 32 bit architecture as a subset.

   c) We will confine our coverage to the core MIPS I architecture.

4. Note that the MIPS architecture itself is older than you are!

   a) That may seem surprising, given the rapid progress in the field of CPU performance. However, the changes have mostly come at the implementation level, not the architectural level.

      (Compare: Today's cars are much safer, longer lasting, and environmentally friendly than those of decades ago - however,

the basic architecture of engine, four wheels, a steering wheel, accelerator, brake and (optionally) clutch pedals has remained unchanged for decades.)

b) Though mips chips are not currently used in laptops or workstations, they are still widely used in embedded systems including many driver assistance systems in cars.

c) For pedagogical reasons, mips is still widely used in computer architecture courses like this one because the simplicity and regularity of its design makes in more easily understood both in terms of an ISA and in terms of implementation,

E. Note that, at this point, we are going to study the MIPS ARCHITECTURE. As is true of most successful architectures, There have been many implementations of this architecture, but all have the same basic architecture and support the basic instruction set that was present in the original R2000 implementation, and this is what we will study.

II. **Basic MIPS-I Architecture**

A. Go over diagram in handout. (Note: IO Devices will be discussed later in the course and vary widely from installation to installation.)

PROJECT Diagram

B. Discuss handout material on CPU registers

1. The CPU has 35 user-visible REGISTERS (plus several typically used only by the operating system). Each register holds one word (32 bits).

2. Registers can be thought of as a very special kind of memory cell.

a) Registers are part of the CPU, not the memory system.

b) A register is referred to by name (e.g. $31, pc) instead of by address.

(1)The general registers have generic names ($0, $1 ...)

(2)But some also have special names as noted in the handout. Note that these are aliases - either the general name or the special name refers to the same register.

(3)The other registers (pc, hi, lo) only have special names.

c) Information in a register can be accessed in much less than one clock cycle (e.g. much less than a nanoseconds on a 1GHz + machines). In contrast, information in memory requires 10's of ns to access.

3. Discuss

a) 32 general registers. Two have special uses

(1) $0 is always zero (hardwired to this value - cannot be changed) Special name: zero

(2) $31 is used for the return address when a procedure call instruction is executed. It can be used for other things, but by convention it is not. Special name: ra

b) pc - always contains address of **next** instruction

c) hi and lo

d) Note that - in contrast to the VonNeumann machine, there is no IR. That's because MIPS uses a pipelined implementation in

which several instructions are at different stages of processing at any one time. There are several "IR's" that are part of the pipeline registers, as we shall see later.

C. Discuss Handout material on Memory

1. The amount of physical memory installed will vary from system to system Special hardware and software gives the user the appearance of a much larger VIRTUAL MEMORY by using disk as an extension of main memory to hold regions not currently being used.

2. Addresses whose leftmost bit is 1 (0x80000000 to 0xffffffff) are handled in special ways by the memory management hardware, and so are not used by ordinary user programs (though they are used by the operating system).

3. Individual regions of memory may be PROTECTED, so that a user program may be prohibited from writing it or reading or writing it. (This allows the system to protect multiple users of the same system from one another, and to protect itself from them.)

   (Memory management and protection are topics we will consider toward the end of the course. They are only an issue when writing user programs if one accidentally or intentionally uses an invalid address. The familiar "segmentation fault - core dumped" that you may have gotten due to pointer error in a C++ program is the operating system's typical response to attempted access to illegal memory addresses.)

4. Thus, regardless of the mount of physical memory actually installed, the application program view of memory is 2 Gigabytes, with addresses ranging from 00000000 to 0x7fffffff (on a 32-bit version of MIPS).

III. **Basic MIPS Instructions**

A. The Basic Execution Cycle

1. The CPU fetches and executes instructions from memory.

   a) Each instruction is one word (32 bits) long.

      PROJECT Basic instruction format

   b) The leftmost six bits of each instruction are the OPCODE which specifies what operation is to be performed. (Some instructions use additional bits elsewhere in the instruction to further specify the operation.)

   c) The remainder of the instruction specifies the OPERANDS - what values the operation is to be performed upon.

   d) The precise format of the rest of the instruction (what bits have what meaning) follows one of three patterns, depending on the opcode. (We will briefly introduce two today.)

2. Like all Von Neumann machines, the CPU repeatedly executes the following "fetch - execute" cycle:

```
while not halted
{
        fetch a word from the memory location specified by pc
        update pc (pc <- pc + 4 since instructions are one word long)
        decode instruction
        execute instruction
}
```

   PROJECT

3. MIPS instructions have one of three formats:

   a) R-Format

   b) I-Format

   c) J-Format

B. The add instruction as a typical R-Format instruction

PROJECT R-Format

1. The MIPS add instruction can be used to add the contents of two SOURCE registers, placing the result in some DESTINATION register (which can be the same as one of the source registers, or different.)

2. It looks like this (all values given in decimal)

# of bits	6	5	5	5	5	6
field	op	rs	rt	rd	shamt	funct
contents	op = 0 for most R type instructions	1st source reg	2nd source reg	dest reg	(not used - 0)	arith/logical function = 32 for add

   This general format of instruction is called R format (where R stands for "register", because all operands are in registers)

3. Thus, the instruction to add the contents of register 8 and register 9. placing the results in register 10, would look like this:

```
bits        31..26  25..21  20..16  15..11  10..6   5..0
            (6)     (5)     (5)     (5)     (5)     (6)

field       op      rs      rt      rd      shamt   func
values
(decimal)   0       8       9       10      0       32
(binary)    000000  01000   01001   01010   00000   100000

= 0000 0001 0000 1001 0101 0000 0010 0000
hexadecimal     =   0x01095020
```

PROJECT Machine language format for example add

4. Clearly, writing instructions in machine language is an error-prone and tedious process. For this reason, we normally use assembly language as a symbolic representation, relying on a program (the assembler) to translate into machine language for us.

Ex: The above instruction in assembly language

```
add $10, $8, $9 - corresponds to HLL $10 = $8 + $9
```

PROJECT

Three things to note:

a) The symbolic op code (add) represents values both in the op and the function fields

b) Order of specifying registers

(1)Machine language: source1, source2, destination

(2)Assembly language: destination, source1, source2

(corresponds to the way we would write a HLL assignment statement: destination = source1 + source2. The order of writing is not an issue, since we use a program to translate the symbolic form to machine language.)

(3)The shamt field that is not used by add (and many other instructions) is not specified at all in the assembly language form.

C. Other R-Format instructions

1. Go over list in handout.

   PROJECT

   a) Discuss distinction between add/sub and addu/subu

   b) Note two kinds of shift instructions (fixed amount and variable amount)

   c) Note two kinds of right shift instruction (arithmetic, logical)

   d) We will see uses for jr, jalr, slt, sltu later, so we won't discuss now.

2. You might think that MIPS would have multiply and divide instructions that look similar to add and subtract - but this is not the case.

   a) Multiply and divide are much more complex operations. An add or subtract can be done in one machine cycle, but a multiply or divide will take many cycles.

b) For this reason, early RISC architectures did not include multiply and divide instructions - they had to be synthesized by a software subroutine when needed.

c) MIPS does have hardware multiply and divide instructions, but they differ from most other R-type instructions in two ways

(1)They do not specify a destination register - the result of multiplication is placed as a double-length value in hi and lo; and division produces two results - quotient in lo, remainder in hi.

(2)These instructions START the execution of the operation, which is performed by the multiply divide unit in parallel with further ordinary computation.

The result is fetched from hi and lo by mfhi, mflo - which are interlocked - i.e. further execution of instructions by the CPU is suspended until the needed value is available.

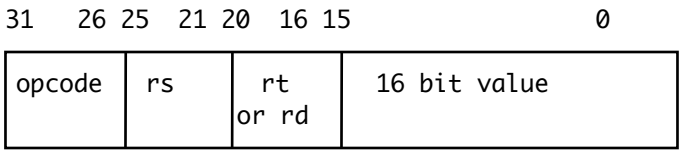
IV. **Working with Constants**

A. The add immediate instruction.

1. Many times, it is necessary to work with integer constants in a program - e.g. C/C++/Java i ++ translates into "add 1 to i"

2. One way to handle this would be to store the value 1 in a known location in memory, and then treat it like a memory variable when its value is needed.

3. However, because constants are needed so often, MIPS provides a special form of the add and several other instructions for dealing with them, called add immediate. These instructions are called I-Format instructions, because the instruction contains an IMMEDIATE VALUE as part of the instruction.



PROJECT

# of bits	6	5	5	16
field	op	rs	rt	immediate value
contents	op = 8 for addi	source reg	destination reg	value to add (two's complement signed number)

4. Example: to add 1 to register 8, and put the result in register 9, we could use the following instruction

bits	31..26 (6)	25..21 (5)	20..16 (5)	15..0 (16)
field values				
(decimal)	8	8	9	1
(binary)	001000	01000	01001	0000000000000001

= 0010 0001 0000 1001 0000 0000 0000 0001
hexadecimal = 0x21090001

PROJECT

5. The assembly-language way of writing the above would be

addi $9, $8, 1

PROJECT

(By now you're used to the fact that the MIPS machine language has the order source then destination, while the assembly language puts the destination first!)

6. Actually, the addi instruction can be used for much more than just adding a value to a register.

   a) Suppose we wanted to SUBTRACT 1 - e.g. to do something like

      i --

      We can do this with addi, using a negative value. (Assume i is in register 8):

      addi $8, $8, -1

   b) Suppose we wanted to LOAD 1 into a register - e.g. to do something like:

      i = 1;

      We can do this with addi, taking advantage of the fact that register 0 always contains 0. (Assume i is in register 8)

      addi $8, $0, 1

   c) Other architectures might include several different instructions - e.g. add immediate, subtract immediate, and load immediate. In keeping with the RISC philosophy, MIPS has just one that can be used to perform multiple jobs.

B. Other Immediate instructions - Handout. Note that some treat the 16 bit constant as a signed number (and therefore sign extend to 32 bits) while others treat it as unsigned (and therefore append 16 leading 0's)

   PROJECT

C. The load upper immediate Instruction

   1. The I-Format instructions allocate 16 bits in the instruction to hold the immediate value to be used. For those instructions which sign extend

the immediate value, we can represent any value between -32768 and + 32767; for those which don't sign extend, any value between 0 and + 65535.

2. What do we do if we need a value outside this range? The MIPS architecture includes a "load upper immediate" instruction (lui), that can be used to place a 16 bit value into the UPPER half of a 32 bit register. When followed by an ordinary immediate instruction (typically ori to avoid sign extension), this can be used to put any 32 bit value in a register.

# of bits	6	5	5	16
field	op	0	rt	immediate value
contents	op = 15 for lui	(not used)	destination reg	value to load into upper 16 bits

3. Example:

```
C:      x = 0x12345678

MIPS:   lui     $2,0x1234
        ori     $2,$2,0x5678
        sw      $2,x
```

V. **MIPS Load and Store Instructions**

A. An important architectural characteristic of RISCs is that all computational instructions operate on values contained in registers, and put their result in a register.

1. If we want to do computation on variables contained in memory, we need to first load them into registers, do the computation there, and then (if necessary) store the result back into memory. (NOTE: in many ISA's, the term LOAD is used to mean "copy a value from a location in memory into a register", and STORE is used to mean "copy a value from a register into a location in memory" - but these terms are not used with 100% consistency! We will always use them in this way, though.

Ex: Assume that the variables x, y, and z are stored in memory, and we want to compute

x = y + z

a) Would be translated by four MIPS instructions

Load y into some register (say $8)
Load z into some other register (say $9)
Add the registers, putting the result into some register (say re-use $8)
Store the result register into x

b) In assembly language

```
lw $8, --- address of y
lw $9, --- address of z
add $8, $8, $9
sw $8, --- address of x
```

2. There are major two reasons why RISCs use this approach (known as load store architecture).

a) It facilitates using a speed-improving technique known as PIPELINING (to be discussed in detail later.)

b) It allows an arithmetic instruction to be represented in a single word - note that it takes only 5 bits to specify a register, but could take as many as 32 to specify a memory address (so if we could do x = y + z in one instruction, the instruction could need 96 bits just to specify the addresses of the three operands, plus more for the opcode! - which would amount to the same total length as four one-word instructions)

3. Obviously, when a value is stored in a register it is much easier to manipulate than when it is stored in memory. For this reason, good compilers (and smart human programmers) try to take advantage of

the large number of available registers to store frequently-used variables in registers, rather than memory.

a) For example, if a function contains a few local variables, these will most likely be kept in registers, and will never exist in memory - since they come into existence when the function is entered, and cease to exist when it terminates.

b) The C language includes a register directive which can be used, when declaring a variable, to tell the compiler that the variable should "live" in a register if at all possible.

Example:

register int i;

The compiler will try to set aside a register to hold the value of i, and will not put it in memory (unless it is unable to reserve a register.)

c) However, good compilers incorporate register allocation algorithms that accomplish the same result - but often more efficiently than humans can do with register "hints" - so most programmers leave it to the compiler to handle this issue.

d) Nonetheless, in any program having more than a very small number of variables, there will be a need to keep many variables in memory.

B. This leads us to a consideration of the basic load and store instructions.

1. As indicated in the example above, each load or store must specify the operation to be performed, a register to be loaded or stored, and a memory address.

2. An astute observer will note that this appears to need more than the 32 bits available in an instruction word: some number of bits to specify the

operation, 5 to specify the register, and 32 to specify the memory address!

3. To avoid this problem, MIPS uses a format for these instructions that specifies the address in terms of a BASE REGISTER and a 16 bit OFFSET. The address is computed by adding the base register and the offset together. The instruction format used is I-Format, similar to that of the immediate instructions we looked at earlier.

# of bits	6	5	5	16
field	op	rs	rt	immediate value
contents	op =	source	transfer	offset
	35 for lw	(base)	(to load)	(16 bit two's
	43 for sw	reg	reg	complement
				signed number)

4. Example: to load the contents of memory location 100 (decimal) into register 8, we could use the following instruction - taking advantage of the fact that $0 always contains zero:

PROJECT

bits	31..26	25..21	20..16	15..0
	(6)	(5)	(5)	(16)
field values				
(decimal)	35	0	8	100
(binary)	100011	00000	01000	0000000001100100

= 1000 1100 0000 1000 0000 0000 0110 0100
hexadecimal = 0x8c080064

5. The assembly-language way of writing the above would be

```
lw    $8, 100($0)
```

(Note, once again, that the order of the two register operands is the opposite of the machine-language order. The load and store instructions always specify the transfer register first, then the offset and base register. As usual, the assembler takes care of the order issue for us.)

6. It might seem that the limitation to using a 16 bit offset would "cramp our style" in terms of accessing memory - i.e. a 16 bit offset can assume values in the range -32768 .. + 32767 if we regard the offset as a signed number.

   a) If the address we wish to access is in low memory (up to 32767), we can specify it directly, using $0 as the base register.

   b) It is common for programs to group variables into regions of memory, and to use a register to point to the beginning o that region.

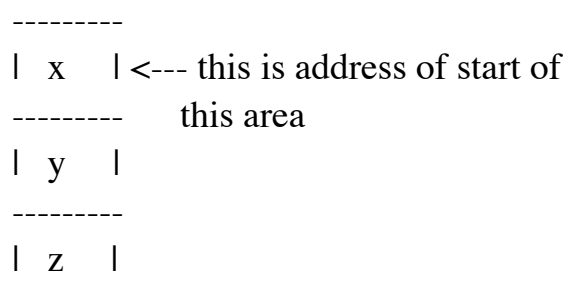
      (1)The fields of an object are allocated storage in successive locations of memory, and the "this" pointer of methods is set to point to the first such location.

      Example: Suppose we have a declared as follows:

```
class SomeClass
{
    int x, y, z;

    void foo()
    {
       x = y + z;
    }
    ...
```

      When foo() is executing, the following situation might exist in memory:



PROJECT

Assuming that the value of this is placed in register 2, the code for the assignment statement in foo might translate as follows (actual code generated by g++ on our MIPS machine)

```
lw      $3, 4($2)
lw      $4, 8($2)
add     $3, $3, $4
sw      $3, 0($2)
```

(Since ints are stored as words (4 bytes long), y is at an offset of 4 relative to this, and z is at an offset of 8.)

PROJECT

(2)Wherever possible, local variables of a function are kept in registers, rather than memory. However, if variables need to be in memory, most compilers put the local variables of a function into a single region of memory called the "stack frame" of the function, and set register 29 (known by the special name $fp) to point to it.

Example: a function might declare local variables as follows:

int a, b;

Assume, for the sake of discussion, that these need to be kept in memory. Then the compiler might generate code that would create the following environment when the function is called:

```
---------
|   a     | <--- $fp holds address of beginning of
---------        this area
|   b     |
---------
```

Then we might load b into register 8 by the following code:

lw $8, 4($fp)

(Since ints are one word (4 bytes) long, b is at an offset of 4 from the beginning of the frame area.) (Note: actual compilers store additional information in the frame, so if you looked at actual compiled code the offset would be more than 4, reflecting this.

(3)Many compilers put global variables into a single region of memory, and set register 28 (known by the special name $gp) to refer to point to it. This allows global variables to be referenced by loads of the form:

lw register, some-offset($gp)

Note: The gnu compilers we have on our mips machine doesn't actually do this.

c) If all else fails, it is always possible to access a variable in memory by putting its address into a register, and then using an address of the form

0(the register)

The assembler is capable of generating code to accomplish this, and uses a specific register that is set aside by software convention for this: register 1 - known by the special name at (assembler temporary.)

7. Go over list of load-store instructions

   PROJECT

C. In addition to accessing scalar variables, it is also possible to use load/store to access elements of an array. In this case, two approaches are possible:

1. If the element number is a constant, we can put the address of the array in a register and encode the element number in the offset.

   Example: Given an array of integers x, load x[4] into register 9, assuming that register 8 holds the address of the array (the address of x[0])

   ASK

   lw $9, 16($8) # 16 because each element is 4 bytes long

2. If the element number is a variable, we can compute the address of the desired element in a register and then use it with offset 0.

   Example: Given an array of integers x, load x[i] into register 9, assuming that register 8 holds the address of the array, and register 10 holds the value of i. Use register 2 as a temporary.

   ASK

```
add    $2, $10, $10    # $2 = 2 * i
add    $2, $2, $2      # $2 = 4 * i
add    $2, $2, $9      # $2 = address of x[i]
lw     $9, 0($2)
```

   PROJECT

3. Note that in C/C++ and Java it is possible to use very similar statements to allocate storage for an array:

   C/C++ int * x = new int [10];
   Java int [] x = new int [10];

In both languages, x now is a variable that holds the ADDRESS of the first element of the array (x[0]), and access to an array element x[i] is obtained by adding the value of x and the value of i (times the size of an element). This is the way that the underlying hardware accesses arrays.

VI. MIPS Conditional Branch Instructions

A. The original version of the MIPS ISA defined two conditional branch instructions, which change the value in the program counter (and thus alter the flow of the program) if some condition is true. (Later versions of the ISA defined additional such instructions, but we will limit ourselves to these two now).

1. beq - branch if the two registers are equal

2. bne - branch if the two registers are not equal

3. Both can be used for comparison to zero if $0 is used as one of the registers

B. Both conditional branches are I format instructions, and look like this

# of bits	6	5	5	16
field name	op	rs	rt	immediate value
contents	op = 4 for beq 5 for bne	first reg to compare	second reg to compare	offset (two's complement signed number)

(where rs and rt specify the registers to compare)

PROJECT

C. Both conditional branches specify the destination of the branch as an offset relative to the value currently in the PC.

1. The offset is sign-extended to yield a signed number, and then is multiplied by 4 (because all instruction addresses are a multiple of 4) and then added to the value currently in he pc, which is by this time the address of the NEXT instruction to be executed. (So the offset if effect specifies the distance in instructions - not bytes.)

2. The offset can range from -32678 to +32767. After multiplication by 4, and adding to the address of the next instruction, this means that conditional branches can "reach" to an instruction in the range

   addr of branch instruction - 131068 .. addr of branch instruction + 131072

D. An important quirk: RISC computers (including MIPS) achieve impressive performance in part by overlapping the execution of several instructions.

1. We will see, when we get to the implementation of MIPS, that the ISA was designed to allow the CPU to actually be working on different parts of up to 4 successive instructions at the same time.

2. This poses an interesting problem in the case of conditional branches: by the time that a decision has been made about whether or not to branch, the next instruction has already been fetched from memory.

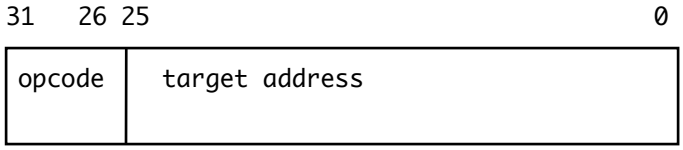
   a) This doesn't pose a problem if the branch is not taken (the condition is false). But what if the condition is true? In this case, we have fetched an instruction from memory that we don't want to execute.

   b) We could just nullify the instruction - causing a "bubble" in the pipeline. However, what many RISCs - including MIPS - do is to execute the instruction anyway. Often, it is possible to fill this "branch slot" with a useful instruction that needs to be done regardless of whether or not we branched; but - absent this - it is standard practice to fill this slot with a "nop" (no-operation) instruction.

E. We will discuss how comparisons other than for equality (e.g. <, <=, >, >=) are handled later.

VII. J-Format Instructions

A. The last MIPS instruction format we need to discuss is the J-Format - which is only used for two instructions.



PROJECT

1. The J instruction is used for unconditional branches in a program. It places the address specified into the pc, which causes the instruction at that address to be the next one executed.

2. The JAL instruction is used for calling a function/procedure. It does the same thing as J, except that it also places the value in the pc at the time it is executed (which is the address of the next instruction after the JAL) into $31 - so that when the function or procedure completes it can return to the instruction just after the one that called it.

B. The target address field in the instruction is only 26 bits long, which isn't long enough to specify an address anywhere in memory (for which a 32 bit address would required). Two measures are used to deal with this.

1. The target address specified in the instruction is multiplied by 4. Multiplication by 4 is done because instructions are words, and therefore must have addresses that are a multiple of 4. Doing this allows a 26 bit address to encode information that corresponds to the lower 28 bits of the pc.

2. The target address is loaded into the lower 28 bits of the pc, nd the upper 4 bits are left alone. This means that the jump must always be to an address in the same 256 MB chunk of memory as the instruction is in. Since most programs are much smaller than 256 MB, this hasn't proved to be a serious problem.

C. It a longer "reach" is needed, the R-Format jr and jalr instructions introduced earlier can be used. They use the 32-bit (or 64-bit) address in the rs register as the target address. (rt and rd are ignored in this case.)

VIII. **The Assembler**

A. From the examples above, it should be clear that writing MIPS programs in machine language would be tedious and error prone, and that such programs could be very hard to modify. This is true of any machine language. Thus, from the earliest days of computing (the 1950's), when it is necessary to program at this level it has been common to write programs in a symbolic ASSEMBLY LANGUAGE and then use a special program called an ASSEMBLER to translate the symbolic program into actual binary machine code.

B. Today, of course, it is relatively uncommon to write programs in assembly language - though assembly language must still be used for writing some low-level components of system software such as operating systems. Today, most assembly language is actually written by compilers - many compilers translate a HLL into assembly language, which then translates it into machine language - though some compilers compile directly to machine language.

1. Example: The C and C++ compilers on most Unix systems work this way.

   E.g. if we compile a C++ program called foo.cc into an object program foo.o using the command|

   g++ -c foo.cc

We actually get an intermediate file called foo.s which is then translated to get foo.o

2. Normally, the assembly code is deleted after it is assembled. However, you can use the -S command line switch to stop the process after the assembly code is produced.

   DEMO:

   Create the following simple program demo.cc:

   ```
   int x, y, z;

   void foo()
   {
      x = y + z;
   }
   ```

   Compile using gcc -S demo.cc

   Show demo.s.

   Note that there is a _lot_ of overhead code associated with entering and exiting a function.

   Note several lines that are immediately recognizable:

   ```
   lw      $2,y
   lw      $3,z
   addu    $2,$2,$3
   sw
   ```

   (Note: the addu instruction is very similar to the add instruction we have considered, except that it does not do any checking for overflow. We will consider how MIPS add handles overflow later; note that C programs simply ignore overflow!)

C. Historically, there has been a one-to-one correspondence between lines of assembly language code and machine instructions - e.g. the symbolic operation code corresponds directly to a machine language op code, and each line of assembly language produces exactly one machine instruction.

1. However, on RISCs this is not necessarily true - many RISC assemblers will accept some constructs that assemble into more than one machine instruction, and will synthesize certain RISC instructions from assembly language instructions that do not correspond directly to a machine instruction.

2. Example: in the above - the various load and store instructions may require more than one machine instruction.

   Ex: lw $2,y

   If y's address is <= 32767, this can be encoded in a single I ormat lw instruction. However, if y's address is >= 32768, it may be necessary to synthesize a sequence of instructions that put the address of y into some temporary register (at), and then the instruction lw $2, 0(at). Thus, this one line could translate into as many as three machine language instructions.

3. Example: The MIPS assemblers recognize certain pseudo-instructions that can be synthesized from other actual machine instructions.

   a) Example: Suppose we want to copy the value in $2 into $3. How can we do this using MIPS instructions we already know?

      ASK

      add $3, $2, $0

      The MIPS assembler will accept the pseudo instruction:

move $3, $2

which does not directly correspond to any MIPS machine instruction, and will synthesize an instruction like the above when it occurs.

b) Example: Suppose we want to load a constant (say 42 = 2a hex) into register 2. How can we do this using instructions we know?

ASK

ori $2, $0, 42

The MIPS assembler will accept the pseudo instruction:

li $2, 42

which does not directly correspond to any MIPS machine instruction, and will synthesize an instruction like the above when it occurs.

4. The pipelined implementation of MIPS requires that we cannot use a value loaded from memory on the very next instruction. (This has to do with the fact that execution of successive instructions is overlapped in time.) Sometimes the assembler needs to insert a "do nothing" instruction to ensure that values are valid. (We will discuss this later in the course, and will ignore it for now.)

D. You will get some experience working with the MIPS assembler in lab.

IX. **An Example**

A. To put everything together, consider the compilation of the following C/C++ assignment:

```
int answer;
int x, y;
int a[10];
int i;

answer = (x + a[i]) - (y + 1);
```

B. MIPS assembly language (ignoring load delays). Assume, for simplicity, that all variables are in a low address region of memory, so they can be accessed by a 16-bit address. (A more realistic situation would involve 32-bit addresses, but that's not the point here.)

```
lw $8, i        # $8 = value of i
sll $8, $8, 2   # $8 = 4*i
lw $8, a($8)    # $8 = word at address of a + 4 * i = a[i]
lw $9, x        # $9 = x
add $8, $9, $8  # $8 = x + a[i]
lw $9, y        # $9 = value of y
addi $9, $9, 1  # $9 = y + 1
sub $8, $8, $9  # $8 = (x + a[i]) - (y + 1)
sw $8, answer
```

C. MIPS machine language. Assume, for simplicity, that answer is at 100, x at 104, y at 108, a at 112, and i at 152 - all decimal.

ASK class to develop

lw $8, i # $8 = value of i

35 0 8 152 (values in decimal)
100011 00000 01000 0000000010011000 (values in binary)
1000 1100 0000 1000 0000 0000 1001 1000 = 0x8c080098

sll $8, $8, 2 # $8 = 4*i

0 0 8 8 2 0
000000 00000 01000 01000 00010 000000
0000 0000 0000 1000 0100 0000 1000 0000 = 0x00084080

lw $8, a($8) # $8 = word at address of a + 4 * i = a[i]

35 8 8 112
100011 01000 01000 0000000001110000
1000 1101 0000 1000 0000 0000 0111 0000 = 0x8d080070

lw $9, x # $9 = x

35 0 9 104
100011 00000 01001 0000000001101000
1000 1100 0000 1001 0000 0000 0110 1000 = 0x8c090068

add $8, $9, $8 # $8 = x + a[i]

0 8 9 8 0 32
000000 01000 01001 01000 00000 100000
0000 0001 0000 1001 0100 0000 0010 0000 = 0x01094020

lw $9, y # $9 = value of y

35 0 9 108
100011 00000 01001 0000000001101100
1000 1100 0000 1001 0000 0000 0110 1100 = 0x8c09006c

addi $9, $9, 1 # $9 = y + 1

8 9 9 1
001000 01001 01001 0000000000000001
0010 0001 0010 1001 0000 0000 0000 0001 = 0x21290001

sub $8, $8, $9 # $8 = (x + a[i]) - (y + 1)

0 8 9 8 0 34
000000 01000 01001 01000 00000 100010
0000 0001 0000 1001 0100 0000 0010 0010 = 0x01094022

sw $8, answer

43 0 8 100
101011 00000 01000 0000000001100100
1010 1100 0000 1000 0000 0000 0110 0100 = 0xac080064