CPS311 - COMPUTER ORGANIZATION

A Brief Introduction to the MIPS Architecture

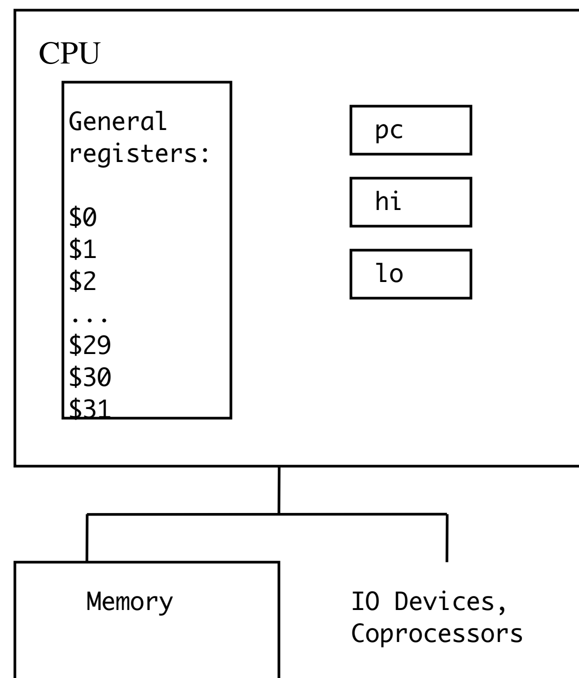
A bit of history

The MIPS architecture grows out of an early 1980's research project at Stanford University. 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. Though the architecture is a quarter of a century old, MIPS-architecture chips are widely used in current systems such as CISCO Routers,, TiVo Series 2 and many other embedded applications including set-top boxes for digital TV, cable modems, DVD recorders, and printers. We use it in this course, because of its clean and simple architecture.

The MIPS architecture itself has passed through a series of evolutions, known as MIPS I, MIPS II, MIPS III, and MIPS IV. 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. 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, and MIPS IV expanded on this. This handout deals only with a subset of the core MIPS I architecture.

Basic Architecture

The following diagram shows the basic architecture of a MIPS-based system:



<u>CPU Registers</u>

The CPU has 32 32-bit general registers that are usable in instructions. With two exceptions, from a hardware standpoint these have identical capabilities and any register can be used by the programmer for any purposes. The two exceptions are these:

· `$0` is "hardwired" to always contain 0. If used as a source register in an instruction, the value used will always be 0. If an instruction attempts to store a value into it, the operation is legal but will be ignored.

· `$31` plays a special role in the `jal` (jump and link) instruction. It is otherwise like any other general register.

While the rest of the general registers are interchangeable from a hardware standpoint, there are some well-established software conventions about how the general registers are used, which one must adhere to if writing a program that is to work correctly with the system libraries and/or code generated by standard software development tools, as well as with multi-user operating systems. Accordingly, The registers have both "generic" and "special" names - in fact, some have two similar special names. Only the generic names and the special names that begin with `$` can be used in assembly language code unless the assembly-language program contains `#include <regdef.h>` - then any form can be used.

Generic name	Special name(s)	Conventional use
$0	zero	Hardwired to always be 0
$1	at	Reserved for the assembler (do not use)
$2, $3	v0, v1	Return values from procedures
$4 .. $7	a0 .. a3	Arguments of procedure calls
$8 .. $15 $24, $25	t0 .. t7 t8, t9	Temporary registers - procedures may use these freely
$16 .. $23	s0 .. s7	Saved registers - a procedure which alters one of these must save its value on the stack and restore it before returning to the caller
$26, $27	k0, k1	Reserved for operating system kernel (do not use)
$28	$gp, gp	Points to memory region containing global variables
$29	$sp, sp	Points to memory region containing stack
$30	$fp, fp	Points to memory region containing stack frame for current procedure; also known as $s8 and treated as a saved register if not needed for this purpose
$31	ra	Holds return address from current procedure

Note that, by convention, general register numbers are specified in decimal.

In addition to the general registers, the CPU also contains three 32-bit special registers:

· `pc` - program counter - always contains the address of <u>next</u> instruction to be executed. Because all MIPS instructions are one 32-bit word long and must reside at an address which is a multiple of 4, this register always contains a value that is a multiple of 4.

· `hi, lo` - hold result of a multiplication or division. Note that multiplying two 32-bit numbers can produce a product that is up to 64 bits long, and division actually produces two results (a quotient and a remainder) - hence the need for two registers for this. `hi` holds the most significant 32 its of a 64 bit product from a multiplication, or the remainder from a division. (`lo` holds the least significant 32 bits of a product or the quotient.)

Memory

Memory is byte-addressable, using a 32-bit address (so it could consist of up to 4 GB of memory) Data in memory can be accessed 8, 16, or 32 bits at a time. 32 bits is referred to as a "word" and 16 bits as a "half word" and 8 bits is, of course, called a "byte". All instructions are one word long.

Words must have an address that is a multiple of 4, and half-words must have an address that is a multiple of 2. Words and half-words are stored in "big-endian" fashion. - e.g. given the memory contents below:

```
Address      Contents

0x100        0xaa
0x101        0xbb
0x102        0xcc
0x103        0xdd
```

The word whose address is `0x100` contains `0xaabbccdd`.
It would be illegal to try to access a word using addresses `0x101, 0x102, 0x103`.

The half word whose address is `0x100` contains `0xaabb`.
The half word whose address is `0x102` contains `0xccdd`.
It would be illegal to try to access a half word using addresses `0x101, 0x103`.

IO Devices

The CPU has two basic modes of operation: user mode and kernel mode. On a workstation, only the "kernel" of the operating system runs with the CPU in kernel mode. Most operating system code, and all user applications, run with the CPU in user mode. (On embedded systems, all operations may be performed in kernel mode.)

Access to IO devices is a function that can only be done with the CPU in kernel mode - i.e. it is an operating system function on workstations. We will not discuss kernel mode operations, including access to IO devices, in this handout. (Programs written in assembly language typically use system library routines for IO operations.)

Coprocessors

The MIPS ISA includes provision for attaching up to 4 coprocessors to the system (although more than 2 is rare). The main CPU handles all instruction fetching and decoding; but certain instructions are then dispatched to one of the coprocessors to be executed (i.e. all the CPU does is to recognize it as a coprocessor instruction and send it to the correct coprocessor). The CPU also handles transfer of data between CPU registers and the coprocessor. Coprocessors can either be separate chips, or they can be integrated into the CPU chips.

All MIPS systems include a coprocessor 0 on chip - this is what actually does the kernel mode operations such as memory management. Because the basic MIPS ISA only provides for integer operations, floating point operations are handled by a floating-point coprocessor (coprocessor 1), which has its own floating-point register set. For early implementations, this was a separate chip; current implementations either incorporate the floating point coprocessor on the main chip or omit it totally. We will not discuss floating point operations in this handout, nor will we discuss other coprocessors.

Basic MIPS Instructions

R-Type Instructions

Arithmetic, logical, and shift-type operations are performed on MIPS using R-Type instructions, which encode the operation to be performed and three general registers (two sources and a destination). Shift instructions may also encode the number of places to shift. It is conventional, in MIPS documentation, to refer to the two source registers as `rs` and `rt`, and the destination register as `rd`. Each of these is actually encoded in the instruction as a 5-bit integer that specifies one of the 32 general registers. In assembly language, the registers can be specified by using their generic names (e.g. `$8`) or their special names (e.g. `t0`) if the program includes `<regdef.h>`

```
MIPS Assembly Language      C Equivalent                        Notes

add    rd, rs, rt           rd = rs + rt;                       Note 1
addu   rd, rs, rt           rd = rs + rt;                       Note 1
sub    rd, rs, rt           rd = rs - rt;                       Note 1
subu   rd, rs, rt           rd = rs - rt;                       Note 1
and    rd, rs, rt           rd = rs & rt;                       Note 2
or     rd, rs, rt           rd = rs | rt;                       Note 2
xor    rd, rs, rt           rd = rs ^ rt;                       Note 2
nor    rd, rs, rt           rd = ~(rs | rt);                    Note 2
sll    rd, rt, amount       rd = rt << amount;                  Note 3
srl    rd, rt, amount       rd = (unsigned) rt >> amount;       Notes 3, 4
sra    rd, rt, amount       rd = (signed) rt >> amount;         Notes 3, 4
sllv   rd, rt, rs           rd = rt << rs;                      Note 3
srlv   rd, rt, rs           rd = (unsigned) rt >> rs;           Notes 3, 4
srav   rd, rt, rs           rd = (signed) rt >> rs;             Notes 3, 4
slt    rd, rs, rt           rd = (rs < rt);                     Note 5
sltu   rd, rs, rt           rd = (rs < rt);                     Note 5
mult   rs, rt               hilo = rs * rt;                     Notes 6, 7
multu  rs, rt               hilo = rs * rt;                     Notes 6, 7
div    rs, rt               hi = rs % rt; lo = rs / rt;         Notes 6, 7
divu   rs, rt               hi = rs % rt; lo = rs / rt;         Notes 6, 7
mfhi   rd                   rd = hi;                            Note 6
mflo   rd                   rd = lo;                            Note 6
```

Notes:

1. There are two different versions of the add instruction (`add` and `addu`). Each instruction does the exact same computation. Where they differ is how they handle overflow. The `add` instruction regards its operands as two's complement signed integers, and throws an exception if there is two's complement overflow. The `addu` instruction does not check for overflow.

   The same distinction exists for the subtract instructions (`sub` and `subu`).

   The "u" in the name of two of the instructions stands for "unsigned" - since different overflow rules are used for unsigned numbers. However, the "u" instructions do not

throw an exception if there is unsigned overflow - they don't check for overflow at all. (Hence it may be better to think of the "u" as meaning "unchecked"). Compilers typically generate the "u" form of these instructions in all cases, regardless of the declared types of the operands, since the specifications for languages like C don't call for overflow checking. Assembly language programmers should likewise use the "u" form of the instructions unless one wants to explicitly test for and handle overflow - something which can only be done when code is written in assembly language.

2. These are bitwise logical operations on integers, not boolean operations (note the use of `&` and `|` in the C code, not `&&` or `||`). If used with the constants 0 and 1, they also correspond to the boolean operations, of course.

3. There are two versions of each shift instruction. The "non-v" form hard codes the number of places to shift into the instruction (in the `shamt` field); the "v" form uses the contents of a register to specify the number of places to shift. (Actually, only the rightmost 5 bits of the register are used, since shifting a 32-bit number more than 31 places is meaningless.) For the "non-v" form, only one source register is specified - the assembler fills the field in the instruction that would specify `rs` with 0's. Note carefully the order of the operands in the "v" form!

4. There are two versions of the right shift instructions - `sra/srav` and `srl/srlv`. These differ in how they handle what bit is shifted into the sign position. The "a" stands for "arithmetic" shift. This version treats its operand as a two's-complement signed number, and preserves its sign by sign propagation - e.g. if the sign if 1 before the shift, then a 1 is shifted into the sign position. The "l" stands for "logical" shift - a 0 is always shifted into the leftmost position. Compilers emit the "a" instruction when the operand being shifted is a signed number, and the "l" form when it is unsigned. Note that this distinction is only an issue for right shifts; the left shift always shifts in a 0.

   Example: if some register contains `0xfffffffe` (-2), sra would produce `0xffffffff` (-1), but srl would produce `0x7fffffff`.

5. `slt` means "set if less than". The destination register is set to 1 if the first source operand is less than the second; it is set to 0 otherwise. The "non-u" form treats the operands as signed numbers; the "u" form treats them as unsigned. To see why this is important, note that `0xffffffff < 0xfffffffe` if both are regarded as signed numbers, but `0xffffffff > 0xfffffffe` if they are unsigned. We will see in the discussion of conditional branching below where `slt/sltu` is useful.

6. The multiply and divide instructions do not specify a destination register, for two reasons:

   - The need to use the `hi` and `lo` registers to store the results (a 64-bit product or a quotient and a remainder).
   - Execution of these operations takes more than one cycle. The MIPS multiply unit is set up to run in parallel with the rest of the CPU once a multiplication or division is started, with the results being retrieved from `hi` and `lo` later.

   The `mult`/`multu` and `div`/`divu` instructions start the multiply-divide unit performing the operation. The CPU continues performing subsequent instructions in parallel with the multiplication/division.

   The `mfhi` and `mflo` instructions are later used to move the results to general registers. Of course, there is no good way of knowing when this is (and the extra time needed varies from 5 to 80 cycles depending on the operation and the implementation). Therefore, these instructions are interlocked - if one is executed before the result is available, execution of further instructions by the CPU is stalled until the result becomes available. For this reason, a new multiply or divide operation must not be started until the result of the previous one has been retrieved.

   Multiplication and division by powers of 2 can be done by shifting left or right, which is faster than using explicit multiply or divide instructions. (In fact, an optimizing compiler will often convert multiplication or division by a constant into shift and/or add instructions in the generated code.)

7. There are distinct versions of the multiply and divide instructions for signed and unsigned numbers. In this case, the computation done is different - e.g. `0xffffffff * 0xffffffff = 1` if the operands are signed numbers, but a very big number if they are unsigned!

Immediate Instructions

For certain operations, it is also possible to specify a constant as one of the two operands. The constant is coded into the instruction as a 16-bit integer; therefore, it must lie in the range -32678 ≤ constant ≤ 32767 (for addi, addiu, slti, sltiu) or 0 ≤ constant ≤ 65535 (for andi, ori, xori). The constant is turned into a 32-bit value either by filling the upper 16 bits with a copy of the sign bit (addi, addiu, slti, sltiu) or with 0's (andi, ori, xori). [Note that even the "unsigned" instructions sign-extend the immediate value!] (For larger constants, it is necessary to use several instructions).

MIPS Assembly Language	C Equivalent	Notes
addi rd, rs, constant	rd = rs + constant;	Notes 1,2,3
addiu rd, rs, constant	rd = rs + constant;	Notes 1,2
andi rd, rs, constant	rd = rs & (unsigned) constant	
ori rd, rs, constant	rd = rs | (unsigned) constant	Note 3
xori rd, rs, constant	rd = rs ^ (unsigned) constant	
slti rd, rs, constant	rd = (rs < constant)	Note 1
sltiu rd, rs, constant	rd = (rs < constant)	Note 1
lui rd, constant	rd = constant << 16;	Note 4

Notes:

1. The two forms of addi and slti are distinguished just like the two forms of add, slt.

2. There is no subi. One can subtract a constant by using addi with a negative value.

3. The standard way to load a (small enough) constant value into a register is to use either the addi or the ori instruction with $0 as the register source. (addi is used for a negative signed value, ori for a positive value, though addiu could also be used.)

   Example: Load 0x2a into $4 and -1 into $5:

```
ori    $4, $0, 0x2a
addi   $5, $0, -1
```

4 The lui instruction is used to allow 32-bit constants. It places the 16 bit constant value specified in the instruction into the upper half of the destination register, and clears the lower half of the register to all 0's. For example, suppose we wanted to place the constant 0xaabbccdd into $4. Since the constant requires more than 16 bits, we do this as follows.

```
lui    $4, 0xaabb
ori    $4, $4, 0xccdd
```

Note that the lower bits are loaded with ori; if addi were used, they would be sign-extended, filling all 16 upper bits of the immediate constant with 1's, and thus turning all the 16 upper bits of the $4 to 1's after orring! Note, too, that the ori is done second, since otherwise the lui would wipe out the value placed in the lower bits by ori!

Memory Reference (Load/Store) Instructions

RISC architectures like MIPS require that the operands of arithmetic and logical operations be in registers. Load and store instructions are included in the ISA to allow copying values between memory and registers. There are five different load instructions, and three different store instructions, to allow for accessing bytes, halfwords, or words in memory. (Values in registers are always words, of course)

```
lw        Copies a 32-bit word from memory into a register
lh        Copies a 16-bit signed halfword from memory into a
           register; the upper 16 bits of the register are filled
           with a copy of the sign bit
lhu       Same - but unsigned; the upper 16 bits of the register are
           filled with 0's
lb        Copies an 8-bit signed byte from memory into a
           register; the upper 24 bits of the register are filled
           with a copy of the sign bit
lbu       Same - but unsigned; the upper 24 bits of the register are
           filled with 0's
sw        Copies a 32-bit word from a register into memory
sh        Copies the rightmost 16 bits of a register into a halfword
           in memory (the other 16 bits of the register are ignored)
sb        Copies the rightmost 8 bits of a register into a byte
           in memory (the other 24 bits of the register are ignored)
```

These memory-reference instructions specify an address in memory in terms of a register plus a 16-bit signed offset encoded into the instruction. The address in memory to access is calculated by adding the register and the offset. (The register is not changed - the calculated address is not stored anywhere.) Because any general register can be used, this supports various ways to specify an address - including the following:

· Access a specific address in the lowest-address part of memory: encode the address in the offset part of the instruction, and specify `$0` as the base register. (Only works for addresses ≤ `32867`).

Example: Load the word at address `0x100` into `$4`:

```
lw        $4, 0x100($0)
```

· Access a global variable. Many compilers use the "global pointer" register (`$gp - $28`) to point to a region of memory in which global variables are stored. Specific variables are then accessed as an offset relative to `$gp`. Because the hardware treats the offset appearing in an instruction as a 16-bit signed number, `$gp` is set to point to the middle of the area, and the first global variable is stored at the most negative possible offset relative to the `$gp`.

Example: Load the very first global variable in a program (assume its an int) into `$4`:

```
lw        $4, 0xfffc($0)
```

· Access a variable at any location in memory: use two instructions - one to load the high order bits of the address into some register, and one to access the data, using the low order bits of the address as an offset. (The assembler reserves `at - $1` - for this purpose.) This is a bit tricky, since the hardware interprets the offset as a signed number - thus if there is a "1" in bit position 15 of the address, then the register needs to contain the high order bits of the address plus one to compensate for the sign extension done by the CPU.

Example: Load a variable at an arbitrary location in memory into `$4`:

```
lui $1, hi 16 bits of address of somevariable (possibly + 1)
lw $4, lo 16 bits of address of somevariable($1)
```

· Access a local variable of the current procedure. Most compilers put the local variables of a procedure into a contiguous part of the run-time stack in memory, and and arrange for the frame pointer register (`$fp = $30`) to point to it. A specific variable is accessed at a constant offset relative to the base of this area. (Only works if the total size of all local variables for a given procedure is ≤ `65536`). (No example given because this depends on details of the structure of a procedure's frame which is determined by the compiler)Access a variable based on a pointer that is stored in a register: use an offset of 0 with the specified register.

Example: Suppose `$4` holds a pointer to an integer. Store `$5` into it:

```
    sw     $5, 0($4)     # Note that register being stored from is written
                         # first for consistency with loads - contrary to
                         # normal "destination first" order
```

· Access a specific field within a structure pointed to by a pointer that is stored in a register: use the distance of the field from the start of the structure as the offset relative to the specified register.

Example: Given C declarations as follows:

```
struct student
{
          char name[11];      // Occupies 12 bytes - one for terminating \0
          int age
};
student * s;
```

Store the value in `$4` into `s -> age`. Assume `$2` is not currently being used.

```
lw        $2, s
sw        $4, 0xc($2)
```

Conditional Branch Instructions

The basic MIPS conditional branch instructions compare two registers, and branch if they are (or are not) equal. The address to branch to is encoded in the instruction as a 16-bit signed integer which specifies an offset in words, relative to the address of the next instruction.

Because only 16 bits are available for the offset in the encoding, a conditional branch can reach forward up to 32767 instructions, or backwards up to 32768 instructions. Longer reaches must be encoded by using a conditional branch to skip over a jump.

MIPS Assembly Language	C Equivalent	Notes
`beq rs, rt, label`	`if (rs == rt) goto label`	Notes 1, 2
`bne rs, rt, label`	`if (rs != rt) goto label`	Notes 1, 2

Notes:

1. The assembler figures out the correct offset based on the symbolic label used.

   Example: Assume the following code fragment assembles into the memory addresses shown:

   MIPS Assembly Language / C Equivalent

```
      beq   rs, rt, else   #code goes into 0x1000             if (rs != rt)
      ... a series of instructions in 0x1004..0x10ff    { ... }
else: ...                  #code goes into 0x1100
```

   Then the offset encoded into the branch instruction would be calculated as the number of words between next instruction and target = `(0x1100 - 0x1004) / 4 = 0xfc/4 = 0x3f`

2. It might seem that the MIPS ISA is deficient in not including conditional branches based on relative order (e.g. "branch if less than"). Actually, any comparison between two values can by synthesized by using `slt` in conjunction with `beq` or `bne`.

   Example: Branch to `foo` if `$4 < $5`. Assume `2` is not currently being used.

```
      slt   $2, $4, $5
      bne   $2, $0, foo        # if $4 < $5, $2 contains 1 ≠ 0
```

   Example: Branch to `foo` if `$4 ≤ $5`. Assume `$2` is not currently being used.

```
      slt   $2, $5, $4         # $2 contains 0 iff $4 ≤ $5
                               # (i.e. !($5 < $4) )
      beq   $2, $0, foo
```

Jump Instructions

The branch instructions discussed above use 16 bits to encode a target address, relative to the current instruction, making most of memory "out of reach" from any given location. For most control structures that call for conditional branches, this is not a serious problem, since control structures such as `if`, `for`, and `while` are usually much smaller than 32768 instructions. There are times, though, when it is necessary to transfer control to an arbitrary location.

One of the instructions listed below uses a register to hold the target address, and so can specify any 32-bit target address. The other two use 26 bits in the encoding for a literal address. This is shifted left two places (i.e. multiplied by 4) since instruction addresses must be a multiple of 4; and the high order 4 bits of the target address are taken from the current value in the `pc`. (Thus, these instructions can reach any location within the 256 MB region of memory that contains the jump instruction - the `jr` instruction must be used to reach a completely arbitrary location in memory, but this is rarely necessary since must programs fit in 256 MB!)

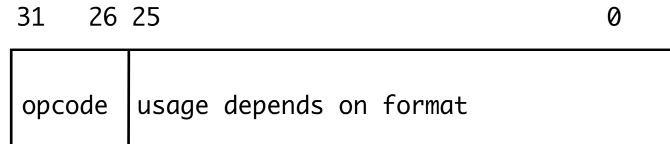
MIPS Assembly Language	C Equivalent	Notes
`jr rs`	`(Used w/$31 to translate return)`	
`j label`	`goto label`	
`jal label`	`label()`	Note 1

Notes:

1. This is the instruction used for procedure call. The address of the next instruction after the `jal` is saved in register `ra` (`$31`), which allows the called procedure to return to the caller by using `jr $31`.

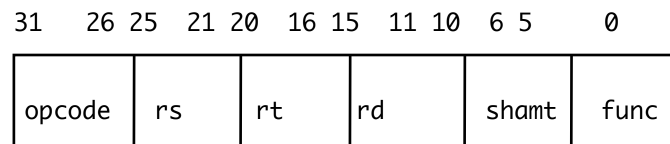
MIPS Instruction Encoding

MIPS instructions are always one word (32 bits) long. Basic MIPS instructions have one of three formats. The instruction format is always implied by the opcode, which is the leftmost 6 bits of the instruction.



The opcode determines the format. All R-Format instructions discussed in this handout have an opcode of 0 . J-Format instructions have opcodes of 2 or 3. All other values correspond to I–Format instructions, or to instructions not discussed in this handout (e.g. instructions that are passed to a coprocessor for execution)

R Format:



This format is used for all the R-Type instructions, as well as for the jr instruction. Note that this format provides fields for specifying the 3 registers used by R-Type instructions, as well as a shamt field used by “non-v” shift instructions. A number of instructions do not use one or more of the fields, as indicated in the table below. Fields that are unused should be 0.

Note well that the registers appear in most assembly language instructions in the order rd,rs,rt, but are physically encoded in the instruction in the order rs,rt,rd. For the “non-v” type shift instructions, the assembly language order is rd,rt,rs.

For all of the R-Type instructions considered in this handout, plus jr, the opcode is always 0. The specific operation to be performed is specified by the func field, as follows:

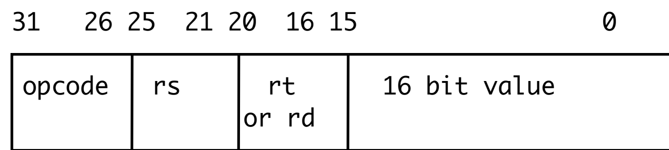
Assembler mnemonic	func	(Note)
sll	0	(1)
srl	2	(1)
sra	3	(1)
sllv	4	
srlv	6	
srav	7	
jr	8	(2)
mfhi	0x10	(3)
mflo	0x12	(3)
mult	0x18	(4)
multu	0x19	(4)
div	0x1a	(4)
divu	0x1b	(4)
add	0x20	
addu	0x21	
sub	0x22	
subu	0x23	

Assembler mnemonic	func	(Note)
and	0x24	
or	0x25	
xor	0x26	
nor	0x27	
slt	0x2a	
sltu	0x2b	

Notes:

1. rs is not used - encoded as 0.
   shamt is used to specify amount.
   (all other instructions encode
   shamt 0).
2. rt, rd not used - encoded as 0
3. rs, rt not used - encoded as 0
4. rd not used - encoded as 0

I Format:



This format is used for three different types of MIPS instruction

- Immediate instructions (the registers are called `rs` and `rd` above; the 16 bit value is the constant)

  Note well that the registers appear in assembly language instructions in the order `rd,rs`, but are physically encoded in the instruction in the order `rs,rd`.

- Memory Reference instructions (the registers are called `rs` and `rt` above; the 16 bit value is the offset to be added to `rs` to compute the memory address)

  Note well that the registers appear in assembly language instructions in the order `rt,rs`, but are physically encoded in the instruction in the order `rs,rt`.

- Conditional branch instructions (the registers are called `rs` and `rt` above; the 16 bit value is shifted left two places and added to the `pc` if the branch is taken)

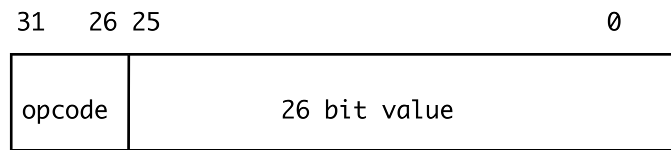
  Note well that the order of listing the registers is the same in both assembly language and the physical encoding. (Not that it would not matter anyway since == and != are commutative!)

The `opcode` determines what operation is to be performed, as follows:

Assembler mnemonic	opcode
beq	0x4
bne	0x5
addi	0x8
addiu	0x9
slti	0xa
sltiu	0xb
andi	0xc
ori	0xd
xori	0xe
lui	0xf

Assembler mnemonic	opcode
lb	0x20
lh	0x21
lw	0x23
lbu	0x24
lhu	0x25
sb	0x28
sh	0x29
sw	0x2b

J Format



This format is used by the Jump instructions - except `jr`, which uses the R Format.

Assembler mnemonic	opcode
`j`	2

Assembler mnemonic	opcode
`jal`	3

Some Examples of Instruction Encoding

1.Subtract `$8` from `$9`, storing the result in `$10`. (Equivalent to C: `$10 = $9 - $8`)

Assembly language:

```
sub      $10, $9, $8
```

Machine language encoding.

```
opcode = 0 (R-Format)
rs = 9; rt = 8; rd = 10; shamt unused (= 0); func = 0x22

000000 01001 01000 01010 00000 100010         (binary)
0000 0001 0010 1000 0101 0000 0010 0010       (regrouping)
0x01285022                                    (hexadecimal)
```

2.Add `0x2a` to `$8`, storing the result in `$10` (Equivalent to C: `$10 = $8 + 0x2a`)

Assembly language:

```
addi     $10, $8, 0x2a
```

Machine language encoding.

```
opcode = 8 (I-Format - immediate type)
rs = 8; rd = 10; constant = 0x2a

001000 01000 01010 0000000000101010           (binary)
0010 0001 0000 1010 0000 0000 0010 1010       (regrouping)
0x0210a002a                                   (hexadecimal)
```

3.Load the `[9]` element of an array of ints into register `$10`. Assume `$8` has previously been loaded with the starting address of the array. (Equivalent to C: `$10 = * ($8 + 9)`, if `$8` is a C variable declared as `int *`)

Assembly language: (Note that the `[9]` element of the array is 9 x 4 = 36_{10} = 0x24 bytes from the beginning of the array)

```
lw       $10, 0x24($8)
```

Machine language encoding.

```
opcode = 0x23 (I-Format - memory reference)
rs = 8; rt = 10; constant = 0x24

100011 01000 01010 0000000000100100           (binary)
1000 1101 0000 1010 0000 0000 0010 0100       (regrouping)
0x8d0a0024                                    (hexadecimal)
```

4.Branch to some location in the program if registers `$8` and `$9` are not equal to each other. Assume that this "somewhere" is 100 bytes beyond the branch instruction. (Equivalent to C: `if ($8 != $9) goto somewhere`)

Assembly language:

```
bne $8, $9, somewhere # Note: assembler figures out the offset
```

Machine language encoding.

```
opcode = 0x5 (I-Format - conditional branch)
rs = 8; rt = 9
```

Constant is calculated as follows: target is 100_{10} ahead of the branch instruction, so 96_{10} ahead of the next instruction. This is encoded as a word offset - $96_{10}/4 = 24_{10}$ = 0x18.

```
000101 01000 01001 0000000000011000           (binary)
0001 0101 0000 1001 0000 0000 0001 1000       (regrouping)
0x15090018                                    (hexadecimal)
```

5.Call some procedure. Assume that it starts at address `0x2000`, which is in the same segment of memory as the code calling it. (Equivalent to C: `someprocedure()`)

Assembly language:

```
jal someprocedure # Note: assembler plugs in address
```

Machine language encoding.

```
opcode = 0x3 (J-Format)
Constant is calculated by expressing target address in words.
0x2000/4 = 0x800

000011 00000000000000100000000000             (binary)
0000 1100 0000 0000 0000 1000 0000 0000       (regrouping)
0x0c000800                                    (hexadecimal)
```

6.Return from current procedure. (Equivalent to C: `return` or falling off end of the code).

Assembly language:

```
jr $31
```

Machine language encoding.

```
opcode = 0 (R-Format)
rs = 0x1f; rt, rd, shamt unused (= 0); func = 0x8

000000 11111 00000 00000 00000 001000         (binary)
0000 0011 1110 0000 0000 0000 0000 1000       (regrouping)
0x03e00008                                    (hexadecimal)
```

MIPS Assembler Pseudo-Instructions

In addition to tasks normally performed by assemblers (translating mnemonic opcodes and symbolic labels, calculating branch addresses, etc.), MIPS assemblers recognize a large number of psuedo-instructions which the assembler translates into an appropriate sequence of actual machine instructions. The following are just a few examples.

`nop` (no operation)

There are times when one needs a machine instruction that does nothing (e.g. to deal with certain timing issues connected with the pipeline, which will be discussed later.) There are a wide variety of instructions that could be used for this purpose - e.g. any R-Type instruction with a `rd = $0` would work, since attempting to store a result in `$0` is legal but ignored. The assembler translates `nop` as `sll $0, $0, 0` - which is a word of all 0's.

Memory loads and stores from/to a variable (`lw, lh, lhu, lb, lbu, sw, sh, sb`).

The I-Format used to encode these instructions only allows for a 16 bit offset relative to a base register. To access a variable at an arbitrary location in memory, it is necessary to first store the upper bits of the address (possibly plus 1) in a register.

Clearly, calculating addresses like this could be quite painful! For this reason, whenever the assembler sees a load/store instruction accessing a variable, it generates the appropriate two instruction sequence. By software convention, register `$1 (at)` is reserved for this purpose.

Example: If the program contains

```
lw    $4, somevariable
```

The assembler generates code as if the programmer had written:

```
lui $1, hi 16 bits of address of somevariable (possibly + 1)
lw  $4, lo 16 bits of address of somevariable($1)
```

`la` (load address)

There are times when it is necessary to place the address of a variable into some register - e.g. to access an array element (register contains the base address of the array). The assembler translates the pseudo-instruction into a two instruction sequence that accomplishes this.

Example: If the program contains

```
la    $4, somevariable
```

The assembler generates code as if the programmer had written:

```
lui $4, hi 16 bits of address of somevariable
ori $4, $4, lo 16 bits of address of somevariable
```