MIPS Instruction Set

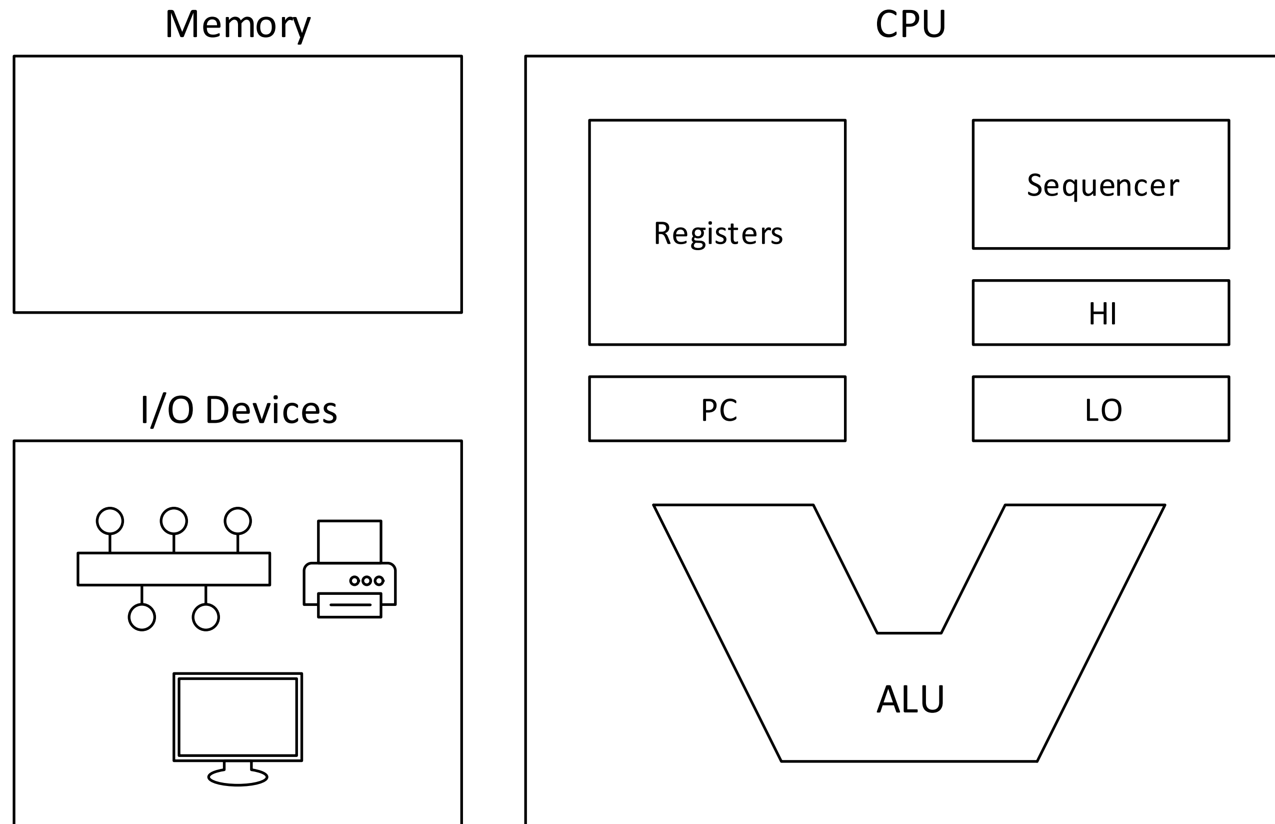
Prof. James L. Frankel
Harvard University

Version of 7:12 PM 3-Apr-2018
Copyright © 2018, 2017, 2016, 2015 James L. Frankel. All rights reserved.

CPU Overview

- CPU is an acronym for Central Processor Unit
- The CPU is the computation unit of the computer
  - The CPU does not include the memory or Input/Output (I/O) devices
- 32-bit Word Size (*i.e.,* there are four bytes per word)
- No Processor Status Word (PSW)
  - A PSW might contain flags (such as carry, overflow, negative), interrupt enable flag, privileged execution mode flag, current interrupt priority, etc.
- Registers can be accessed at instruction execution speed
- Access to memory is slower than access to registers
- User’s data and code reside in memory; Data is moved into registers before operations are performed on them

Simplified Block Diagram



CPU Registers

- General Purpose Registers (GPR)
  - Thirty-two 32-bit GPRs
  - Numbered 0 to 31; Designated $0 through $31
  - Some are used by the hardware
  - All have designated usage by software
- Multiply/Divide Registers
  - Used by hardware multiply and divide instructions
  - 32-bit HI register
  - 32-bit LO register
- Program Counter (PC)
  - 32-bit PC contains the address of the next instruction to be executed
  - During instruction fetch, the PC is incremented to point to the next instruction

General Purpose Registers

- $0 is always read as value 0; Values written to it are ignored
- $1 (named $at) is reserved for use by the assembler
- $2 and $3 (named $v0 & $v1, respectively) are used in expression evaluation and also for the return value from a function
- $4 through $7 (named $a0 through $a3) are used to pass actual parameters 1 through 4 to functions
- $8 through $15 and $24 and $25 (named $t0 through $t7 and $t8 and $t9) are used for temporaries that are not saved by a called function
- $16 through $23 (named $s0 through $s7) are used for temporaries that must be saved by a called function
- $26 and $27 (named $k0 and $k1) are reserved for operating system use
- $28 (named $gp) is used to point to global variables
- $29 (named $sp) is the stack pointer
- $30 (named $fp) is the frame pointer
- $31 (named $ra) contains the return address

Hardware Use of General Purpose Registers

- In the CPU, all registers are general purpose and can be used interchangeably *except*:
  - $0 is always read as value 0; Values written to it are ignored
  - $31 (named $ra) contains the return address
    - The instructions that call a subroutine place the address of the instruction following the subroutine call instruction (*i.e.,* the return address) in register $ra
    - The term *subroutine* is often used to refer to a procedure or function at the assembly language level

Memory Addresses

- Memory in MIPS is byte-addressable
  - That is, each byte in memory is sequentially numbered
- MIPS requires alignment for memory accesses
  - A 32-bit word must be located and accessed using a word aligned address
    - The address of a word is the address of the lowest numbered byte in that word
    - This implies that the low-order two bits of a word address must both be zeros
  - A 16-bit half-word must be located and accessed using a half-word aligned address
    - The address of a half-word is the address of the lower numbered byte in that half-word
    - This implies that the low-order bit of a half-word address must be zero
  - There is no alignment required for 8-bit byte accesses

Return Address

- Because instructions that call a subroutine overwrite register $ra with their return address, any subroutine that calls another subroutine must save register $ra prior to a nested call
    - In order to allow nested and recursive subroutines, any function that calls another subroutine generally saves register $ra on the stack (using the equivalent of *push* and *pop* actions)
    - We will delve into the calling conventions for functions and procedures later

Temporary and Saved GPRs

- The software convention differentiates between temporary (*i.e.,* \$t*n*) and saved (*i.e.,* \$s*n*) registers
- Following these conventions, a subroutine is required to save (on the stack) any \$s registers that it uses
  - A subroutine is **not** required to save any \$t registers that it uses
- Of course, a subroutine is allowed to use both \$t and \$s GPR registers
  - In a section of code in which a subroutine *is not* called, both \$t and \$s registers will retain their values
  - If a subroutine *is* called, the \$t registers are not guaranteed to retain their values across the call
    - Therefore, if a caller wants to maintain the values in \$t registers across a call, the caller will push the selected \$t registers' values before the call and pop those registers' values after the call

Onus of Responsibility for Temporary and Saved GPRs

- A *caller* is a code section that calls a subroutine
- A *callee* is a subroutine that is called
- Onus of responsibility for $t and $s registers
  - The caller is responsible for saving and restoring any $t registers whose values it needs to be retained across subroutine calls
  - The callee is responsible for saving and restoring any $s registers that it modifies

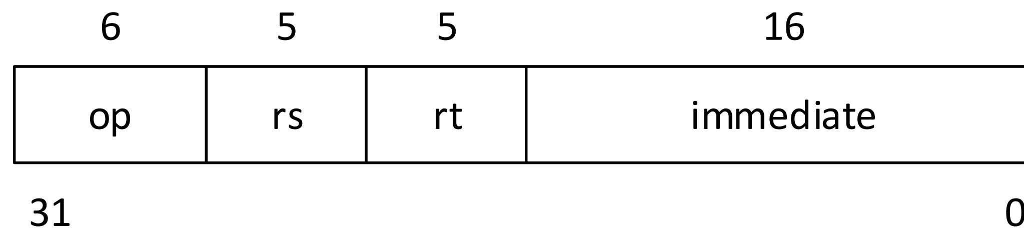
Stack

- The $sp register is used to maintain a stack
- $sp points to the word on top-of-stack
- The stack grows toward lower addresses
- Therefore, a *push* operation is implemented by
  - First, decrementing the $sp by four
  - Then, storing the word to be pushed on the stack into memory at the location pointed to by $sp
- And similarly, a *pop* operation is implemented by
  - First, reading the word on the top of stack by reading the word at the location pointed to by $sp
  - Then, incrementing the $sp by four

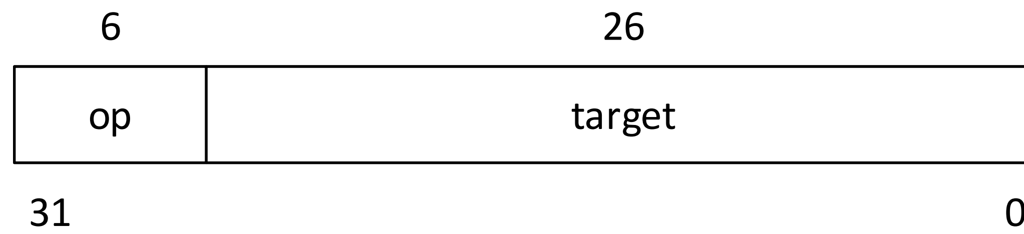
CPU Instruction Formats

- I-Type (Immediate)
- J-Type (Jump)
- R-Type (Register)

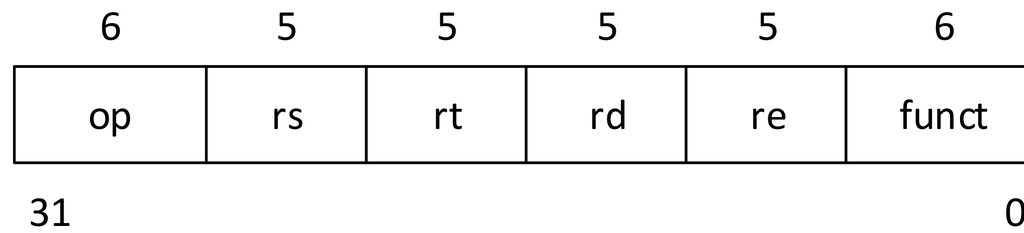
I-Type



J-Type



R-Type



Instruction Summary

- Load & Store instructions move data between memory and registers
  - All are I-type
- Computational instructions (arithmetic, logical, shift) operate on registers
  - Both R-type and I-type exist
- Jump & Branch instructions affect control flow (*i.e.,* may change the value in the PC register)
  - Jumps are J-type or R-type
  - Branches are I-type

Immutable Instructions

- In all modern computers, once **instructions are** loaded into memory for execution, they are **immutable**
  - That is, they cannot be modified
  - This doesn’t have to be true in your new instruction set

- This implies that in all modern computers, data is not intermingled with instructions *in memory*
  - Data and instructions may be intermingled in an assembler program so long as the assembler is able to separate all the instructions and all the data words into separate ***segments*** (e.g., the **text segment** and the **data segment**)

Assembly Language Format

- Instructions begin with an opcode
- The opcode is usually indented (usually by a single tab character)
- The opcode is followed by white space (usually a single tab character)
- The tab is followed by the operands that are appropriate for that opcode
- Most instructions take the destination specifier as the first operand

- For example, in

```
	addu	rd, rs, rt
```

rd is the destination
rs & rt are sources

Role of an Assembler

- An assembler accepts a file containing a program written in a low-level, but textual assembly language and ***produces a file containing that same program in a machine code*** (*i.e.,* numeric) representation
- Other roles of the assembler
  - It allows the programmer to use ***labels on instructions and data*** and to reference those label in the program
    - This means that – in most cases – the programmer does not need to use numeric addresses
  - It allows labels to be assigned ***numeric constant values*** and referenced
  - It converts ***character and string constants*** into their character code values
  - It accepts ***comments***
  - It may extend the instruction set with ***pseudo-instructions***
  - It may accept various assembler ***directives***

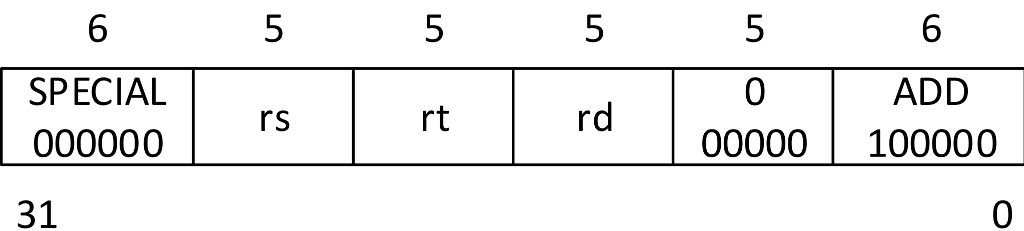
Instruction Set Architecture (ISA) Descriptive Information

- The description of each instruction will
  - Give its opcode name
  - Define the overall instruction format
  - Define the assembly language syntax
  - In English, concisely describe what the instruction does
  - In a mathematical notation, describe what the instruction does
  - Show precisely how the instruction is encoded in its machine representation

Add Instruction

- **ADD** Instruction, R-Type
- **Format:** ADD rd, rs, rt
- **Description:** The contents of general register *rs* and the contents of general register *rt* are added to form a 32-bit result. The result is placed in general register *rd*.
  An overflow exception occurs if the two highest order carry-out bits differ (2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.
- **Operation:** GPR[rd] ← GPR[rs] + GPR[rt]

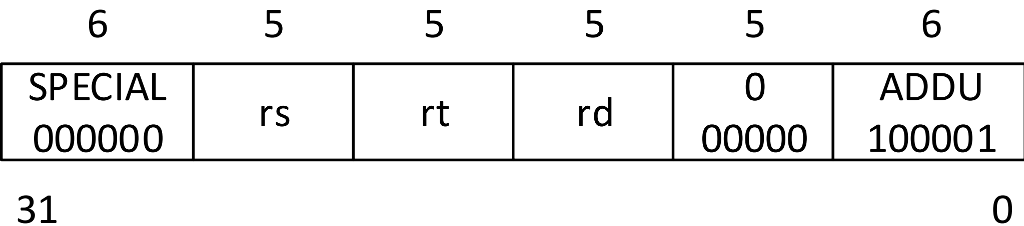
ADD Instruction Fields



Add Unsigned Instruction

- **ADDU** Instruction, R-Type
- **Format:** ADDU rd, rs, rt
- **Description:** The contents of general register *rs* and the contents of general register *rt* are added to form a 32-bit result. The result is placed in general register *rd*.
  No overflow exception occurs under any circumstances.
  The only difference between this instruction and the ADD instruction is that ADDU never causes an overflow exception.
- **Operation:** GPR[rd] ← GPR[rs] + GPR[rt]

ADDU Instruction Fields



Three-Operand R-Type Instructions

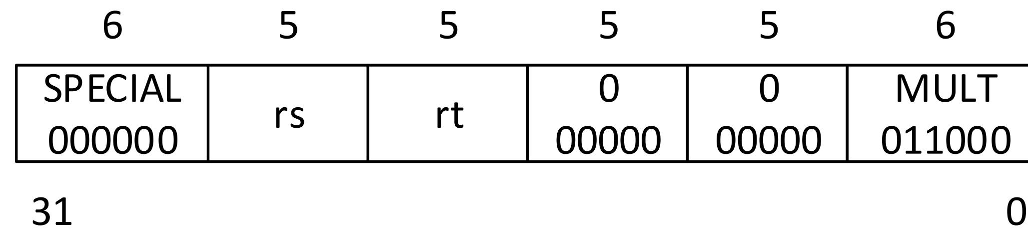
- ADD: Add
- ADDU: Add Unsigned
- SUB: Subtract
- SUBU: Subtract Unsigned
- SLT: Set on Less Than
  - Compare as signed 32-bit integers
  - Result is 1 if true, 0 if false
- SLTU: Set on Less Than Unsigned
  - Compare as unsigned 32-bit integers
  - Result is 1 if true, 0 if false
- AND: Bitwise Logical AND
- OR: Bitwise Logical OR
- XOR: Bitwise Logical Exclusive-OR
- NOR: Bitwise Logical NOR

Multiply Instruction

- **MULT** Instruction, Two-Operand R-Type
- **Format:** MULT rs, rt
- **Description:** The contents of general registers *rs* and *rt* are multiplied as 32-bit 2's complement (*i.e.,* signed) values. No integer overflow exception occurs under any circumstances. This instruction is only valid when *rd* = 0.
When the operation completes, the low order word of the double result is loaded into special register *LO*, and the high order word of the double result is loaded into special register *HI*.
If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two other instructions.
- **Operation:**

T-2:	LO is undefined HI is undefined
T-1:	LO is undefined HI is undefined
T:	t ← GPR[rs] * GPR[rt] LO ← $t_{31..0}$ HI ← $t_{63..32}$

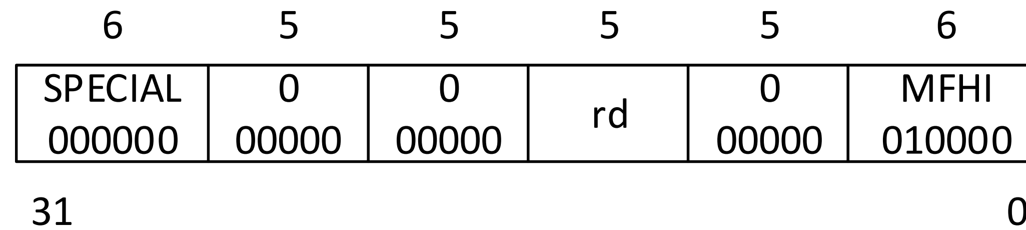
MULT Instruction Fields



Move From HI Instruction

- **MFHI** Instruction, One-Operand R-Type
- **Format:** MFHI rd
- **Description:** The contents of special register *HI* are placed in general register *rd*.
  To ensure proper operation in the event of interrupts, the two instructions which follow an MFHI instruction may not be any of the instructions which modify the *HI* register: MULT, MULTU, DIV, DIVU, or MTHI.
- **Operation:** GPR[rd] $\leftarrow$ HI

MFHI Instruction Fields



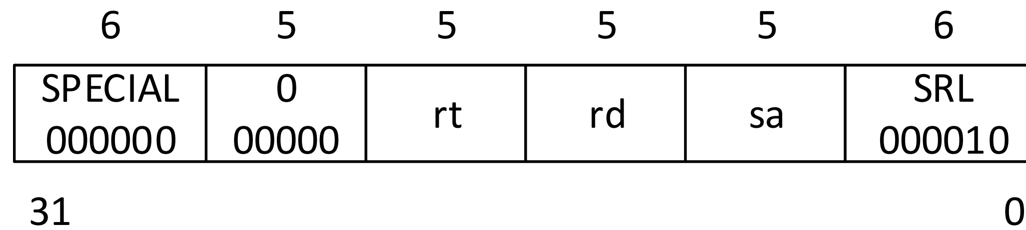
Multiply/Divide R-Type Instructions

- MULT: Multiply
- MULTU: Multiply Unsigned
  - Similar to MULT, but…
  - GPR[rs] & GPR[rt] are both treated as 32-bit unsigned values
- DIV: Divide
  - Divides GPR[rs] by GPR[rt], LO ← quotient, HI ← remainder
  - GPR[rs] & GPR[rt] are both treated as 32-bit 2's complement values
- DIVU: Divide Unsigned
  - Similar to DIV, but…
  - GPR[rs] & GPR[rt] are both treated as 32-bit unsigned values
- MFHI: Move From HI
- MFLO: Move From LO
- MTHI: Move To HI
- MTLO: Move To LO

Shift Right Logical Instruction

- **SRL** Instruction, Shift R-Type
- **Format:** SRL rd, rt, sa
- **Description:** The contents of general register *rt* are shifted right by *sa* bits, inserting zeros into the high order bits. The result is placed in general register *rd*.
- **Operation:** $GPR[rd] \leftarrow 0^{sa} \mid\mid GPR[rt]_{31..sa}$

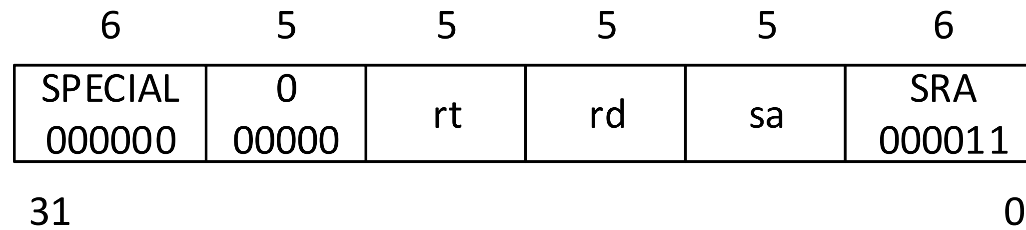
SRL Instruction Fields



Shift Right Arithmetic Instruction

- **SRA** Instruction, Shift R-Type
- **Format:** SRA rd, rt, sa
- **Description:** The contents of general register *rt* are shifted right by *sa* bits, sign-extending the high order bit. The 32-bit result is placed in general register *rd*.
- **Operation:** $GPR[rd] \leftarrow (GPR[rt]_{31})^{sa} \,||\, GPR[rt]_{31..sa}$

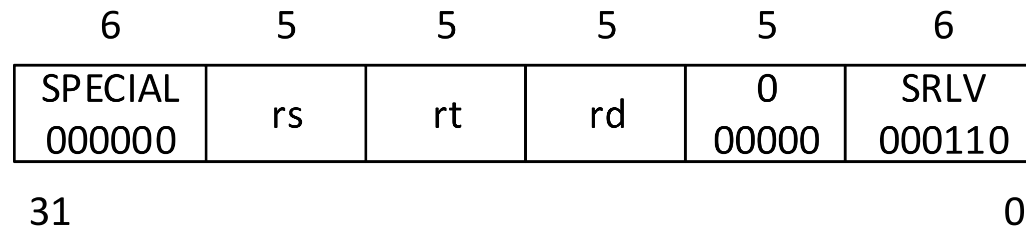
SRA Instruction Fields



Shift Right Logical Variable Instruction

- **SRLV** Instruction, Shift R-Type
- **Format:** SRLV rd, rt, rs
- **Description:** The contents of general register *rt* are shifted right by the number of bits specified by the low order five bits of general register *rs*, inserting zeros into the high order bits. The 32-bit result is placed in general register *rd*.
- **Operation:** $s \leftarrow GPR[rs]_{4..0}$
  $GPR[rd] \leftarrow 0^s \,||\, GPR[rt]_{31..s}$

SRLV Instruction Fields



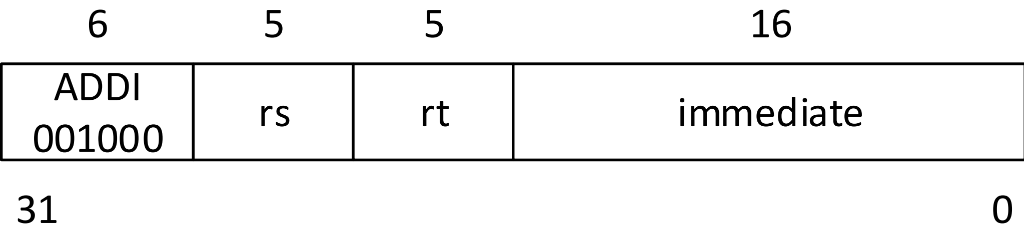
Shift R-Type Instructions

- SLL: Shift Left Logical
- SRL: Shift Right Logical
- SRA: Shift Right Arithmetic
- SLLV: Shift Left Logical Variable
- SRLV: Shift Right Logical Variable
- SRAV: Shift Right Arithmetic Variable

Add Immediate Instruction

- **ADDI** Instruction, I-Type
- **Format:** ADDI rt, rs, immediate
- **Description:** The 16-bit immediate is sign-extended and added to the contents of general register *rs* to form a 32-bit result. The result is placed in general register *rt*.
  An overflow exception occurs if the two highest order carry-out bits differ (2's-complement overflow). The destination register *rt* is not modified when an integer overflow exception occurs.
- **Operation:** $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} || immediate_{15..0}$

ADDI Instruction Fields



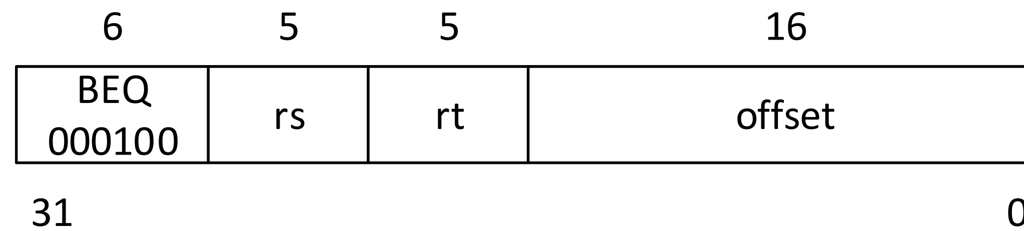
ALU I-Type Instructions

- ADDI: Add Immediate
- ADDIU: Add Immediate Unsigned
    - The 16-bit immediate is sign-extended
    - No overflow exception occurs under any circumstances
- SLTI: Set on Less Than Immediate
    - The 16-bit immediate is sign-extended
    - Compare as signed integers
- SLTIU: Set on Less Than Immediate Unsigned
    - The 16-bit immediate is sign-extended
    - Compare as unsigned integers
- ANDI: Bitwise Logical AND Immediate
    - The 16-bit immediate is zero-extended
- ORI: Bitwise Logical OR Immediate
    - The 16-bit immediate is zero-extended
- XORI: Bitwise Logical Exclusive-OR Immediate
    - The 16-bit immediate is zero-extended
- LUI: Load Upper Immediate
    - LUI rt, immediate
    - Field *rs* should be zero (SBZ)
    - The 16-bit immediate is shifted left 16 bits; the low order 16 bits are set to zeros; result is stored in general register *rt*

Branch on Equal Instruction

- **BEQ** Instruction, I-Type
- **Format:** BEQ rs, rt, offset
- **Description:** A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended to 32 bits. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are equal, then the program branches to the target address, with a delay of one instruction.
  Operation: T: targetOffset ← $(\text{offset}_{15})^{14}$ || offset || 0^2
  condition ← (GPR[rs] = GPR[rt])
  T+1: if condition then
  PC ← PC + targetOffset
  endif

BEQ Instruction Fields



Branch I-Type Instructions

- BEQ: Branch on Equal
- BNE: Branch on Not Equal
- For all of the following instructions:
  - Only one register is specified
  - The assembly language has the form: *OPCODE* rs, offset
  - Field *rt* should be zero (SBZ)
  - Treat GPR[rs] as a signed integer
- BLEZ: Branch on Less Than or Equal to Zero
- BGTZ: Branch on Greater Than Zero
- BLTZ: Branch on Less Than Zero
- BGEZ: Branch on Greater Than or Equal to Zero
- BLTZAL: Branch on Less Than Zero and Link
- BGEZAL: Branch on Greater Than or Equal to Zero and Link

The *And Link* Instructions

- The ***And Link*** instructions place the address of the instruction following the delay slot into the link (return address) register (*$ra* or *$31*) unconditionally (*i.e.,* whether or not the instruction branches, register *$ra* is modified with the potential return address)
- Register *rs* may not be general register 31

- Thus, the ***And Link*** instructions are used to call subroutines

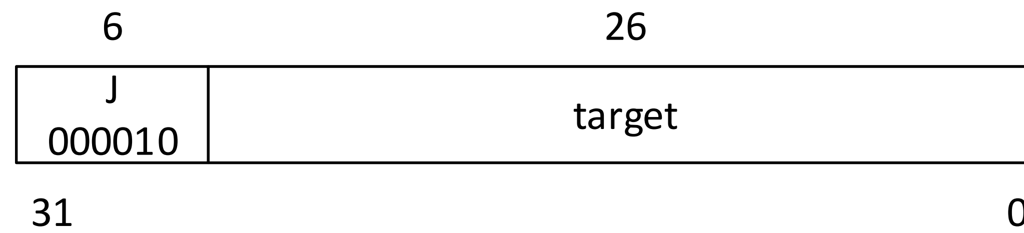
The *offset* in Branch Instructions

- Instructions are required to be on word-aligned addresses
  - Therefore, the low-order two bits of the address of an instruction are always zeros
- The target address of a branch (or jump) instruction must be word aligned
- There is no benefit to being able to produce a non-word aligned target address
- Therefore, the offset field is left shifted two bits to force word-alignment and allow a branch range which is four times larger than the non-shifted offset
- The ***offset*** field is 16 bits wide (and, therefore, can express a value from -32768 to 32767)
- Before being possibly added to the PC, the offset is left shifted two bits (and, therefore, can express a value from -32768*4 to 32767*4)
  - This allows a maximum branch range of from 32,768 instructions before the instruction in the delay slot to 32,767 instructions after the instruction in the delay slot

Jump Instruction

- **J** Instruction, J-Type
- **Format:** J target
- **Description:** The 26-bit target address is shifted left two bits and combined with the high order four bits of the address of the instruction in the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction.
  Operation: T: $temp \leftarrow target$
  T+1: $PC \leftarrow PC_{31..28} \,||\, temp \,||\, 0^2$

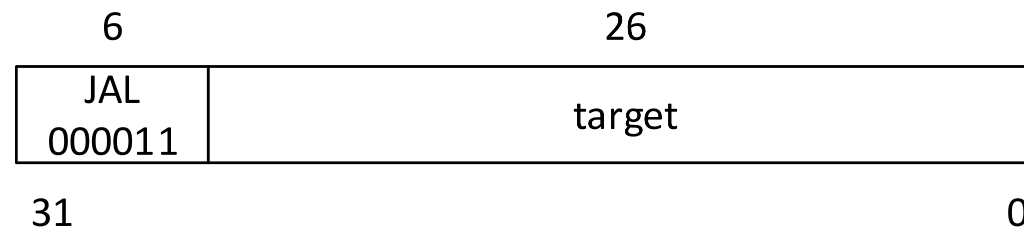
J Instruction Fields



Jump And Link Instruction

- **JAL** Instruction, J-Type
- **Format:** JAL target
- **Description:** The 26-bit target address is shifted left two bits and combined with the high order four bits of the address of the instruction in the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register, *r31*.
  Operation: T: temp ← target
  GPR[31] ← PC + 8
  T+1: $PC \leftarrow PC_{31..28} \mid\mid temp \mid\mid 0^2$

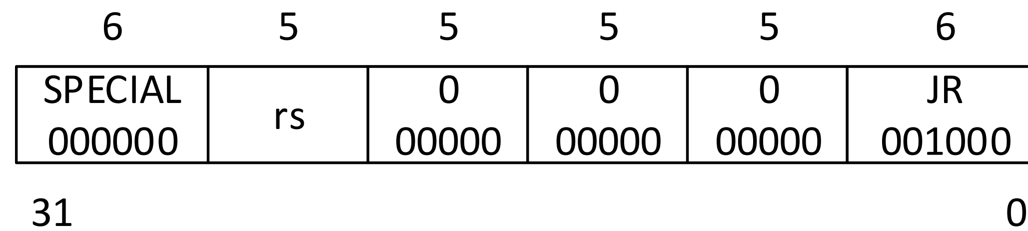
JAL Instruction Fields



Jump Register Instruction

- **JR** Instruction, R-Type
- **Format:** JR rs
- **Description:** The program unconditionally jumps to the address contained in general register *rs*, with a delay of one instruction. This instruction is only valid when *rd* = 0.
  The low-order two bits of the target address in register *rs* must be zeros because instructions must be word-aligned.
  Operation: T: temp ← GPR[rs]
  T+1: PC ← temp

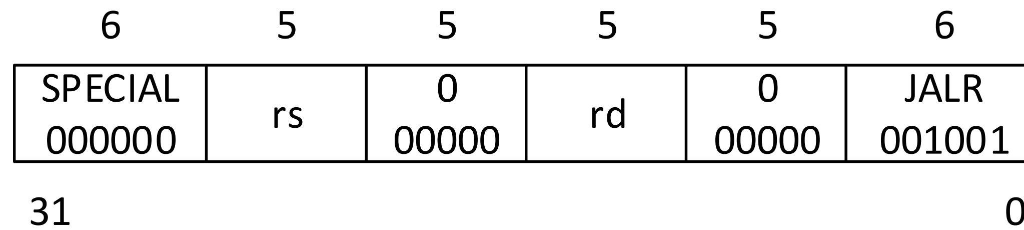
JR Instruction Fields



Jump And Link Register Instruction

- **JALR** Instruction, R-Type
- **Format:** JALR rs, rd
- **Description:** The program unconditionally jumps to the address contained in general register *rs*, with a delay of one instruction. The address of the instruction after the delay slot is placed in general register *rd*. Register specifiers *rs* and *rd* may not be equal.
  The low-order two bits of the target address in register *rs* must be zeros because instructions must be word-aligned.
  Operation: T: temp ← GPR[rs]
  GPR[rd] ← PC + 8
  T+1: PC ← temp

JALR Instruction Fields



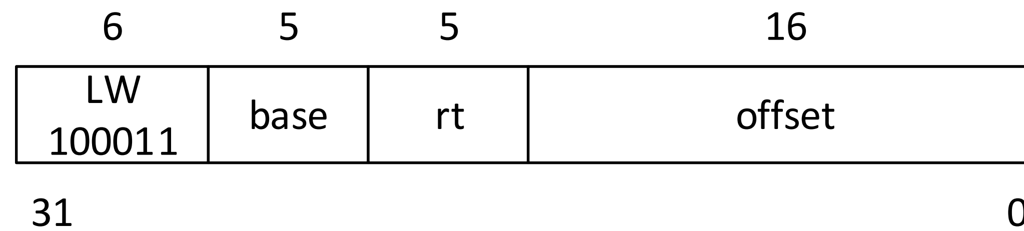
Jump Instructions

- J: Jump
- JAL: Jump And Link
- JR: Jump Register
- JALR: Jump And Link Register

Load Word Instruction

- **LW** Instruction, I-Type
- **Format:** LW rt, offset(base)
- **Description:** The 16-bit offset is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*.
  The two least significant bits of the effective address must be zeros.
  Operation: T: vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15..0}) + \text{GPR[base]}$
  mem ← LoadMemory(WORD, vAddr)
  GPR[rt] is undefined
  T+1: GPR[rt] ← mem

LW Instruction Fields



Load and Store Instructions

- LB: Load Byte
- LBU: Load Byte Unsigned
- LH: Load Halfword
- LHU: Load Halfword Unsigned
- LW: Load Word
- For the following Store instructions:
  - The assembly language has the same form as for the Load instructions: *OPCODE* rt, offset(base)
  - But, unlike other instructions, the leftmost operand, rt, is the source – not the destination; that is, GPR[rt] is stored in memory at the address given by GPR[base]+(sign-extension(offset))
- SB: Store Byte
- SH: Store Halfword
- SW: Store Word

Load and Store Instructions ***offset*** Field

- For all of the Load and Store instructions, the offset field is not shifted left before being address to the contents of the general register *base*
- This limits the range of halfwords and words that can be accessed via Load and Store instructions, but makes all these instructions uniform
  - A different design might allow twice the range when halfwords are accessed and four times the range when words are accessed

Push Implementation

- There are no dedicated stack manipulation instructions
- Pushing register $reg onto the stack is implemented by
  - addiu $sp, $sp, -4 # decrement the $sp for one word
  - sw $reg, 0($sp) # store $reg on the stack
- Similarly, pushing three registers onto the stack is implemented by
  - addiu $sp, $sp, -12 # decrement the $sp for three words
  - sw $firstReg, 8($sp) # store $firstReg on the stack
  - sw $secondReg, 4($sp) # store $secondReg on the stack
  - sw $thirdReg, 0($sp) # store $thirdReg on the stack

Pop Implementation

- There are no dedicated stack manipulation instructions
- Popping register $reg off of the stack is implemented by
  - lw $reg, 0($sp)   # load $reg from the stack
  - addiu $sp, $sp, 4   # increment the $sp for one word
- Similarly, popping three registers off of the stack is implemented by
  - lw $thirdReg, 0($sp)   # load $thirdReg from the stack
  - lw $secondReg, 4($sp)   # load $secondReg from the stack
  - lw $firstReg, 8($sp)   # load $firstReg from the stack
  - addiu $sp, $sp, 12   # increment the $sp for three words