بسم اللہ الرحمٰن الرحیم

EE213 COMPUTER ORGANIZATION AND ASSEMBLY LANGUAGE

Fall 2017

Instruction Set Architecture, MIPS

OUTLINES

- Instruction Set Architecture
- CISC VS RISC
- Introduction to MIPS

THE INSTRUCTION SET ARCHITECTURE (ISA)

- ISA is the set of instructions a computer can execute.
- All programs are combination of these instructions.
- An ISA defines everything a machine language programmer needs to know in order to program a computer.
- ISA defines a set of operations, their semantics, and rules for their use.

RISC / CISC

- Both are Instruction Set Architecture (ISA).
  - Emphasize on instructions; not the hardware.

- **CISC** = **C**omplex **I**nstruction **S**et **C**omputer
  - Slower
  - Fewer Instructions per program
  - E.g. x86

- **RISC = R**educed **I**nstruction **S**et **C**omputer.
  - Faster, simpler hardware.
  - More Instruction per program
  - E.g. MIPS

https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

CISC

- The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible.
  - This is achieved by building processor hardware that is capable of understanding and executing a series of operations.

- One of the primary advantages of CISC is that the compiler has to do very little work to translate a high-level language statement into assembly.

- Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

RISC

- RISC processors only use simple instructions that can be executed within one clock cycle.
- At first, this may seem like a much less efficient way of completing the operation.
  - Because there are more lines of code, more RAM is needed to store the assembly level instructions.
  - The compiler must also perform more work to convert a high-level language statement into code of this form.
- However, Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle command.
  - These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers.
- Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.

RISC

- RISC processors typically have a **load-store** architecture.
- This means there are two instructions for accessing memory:
  I. A load (l) instruction to load data from memory and
  II. a store (s) instruction to write data to memory.
- It also means that none of the other instructions can access memory directly.
- So, an instruction like "add this byte from memory to register 1" from a CISC instruction set would need two instructions in a load-store architecture: "load this byte from memory into register 2" and "add register 2 to register 1".

RISC VS CISC

- Reasons for CISC:
  - Small memory
  - ASM programmers take full advantage of more complex instructions.
- With CISC, processor design complexity is the issue.

- Advantages of RISC:
  - Shorter design time
  - More general purpose registers, caches, pipelining
  - Greater Speed
  - Assembly doesn't need to closely match with HLL.

MIPS

- **M**icroprocessor without **I**nterlock **P**ipelined **S**tages.
- Developed by Stanford University in early 80s.
    - Idea was to develop a processor whose architecture would represent the lowering of the compiler to the hardware level, as opposed to the raising of hardware to the software level.
- The early MIPS architectures were 32-bit, with 64-bit versions added later.

https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html

MIPS BASIC

- **INSTRUCTIONS**
  - 4 bytes (32 bits)
  - 4 bytes aligned (they start at the addresses that are multiple of 4)

- **MEMORY DATA TYPES**
  - BYTES: 8 bits
  - Half Words: 16 bits
  - Words: 32 Bits
  - Memory is denoted "M" (e.g. **M[000C]** is the byte at address 000C h)

- **REGISTERS**
- 32 4-byte registers in the **register file**: an array of processor registers..
- Denoted "R" (e.g. R[2] is register 2)

Name	number	use	Callee saved
$zero	0	zero	n/a
$at	1	Assemble Temp	no
$v0 - $v1	2 - 3	return value	no
$a0 - $a3	4 - 7	arguments	no
$t0 - $t7	8 - 15	temporaries	no
$s0 - $s7	16 - 23	saved temporaries	yes
$t8 - $t9	24 - 25	temporaries	no
$k0 - $k1	26 - 27	Res. for OS	yes
$gp	28	global ptr	yes
$sp	29	stack ptr	yes
$fp	30	frame ptr	yes
$ra	31	return address	yes

- All register are same.
  - Where a register is needed, any register will work

- By convention we use them for particular tasks.

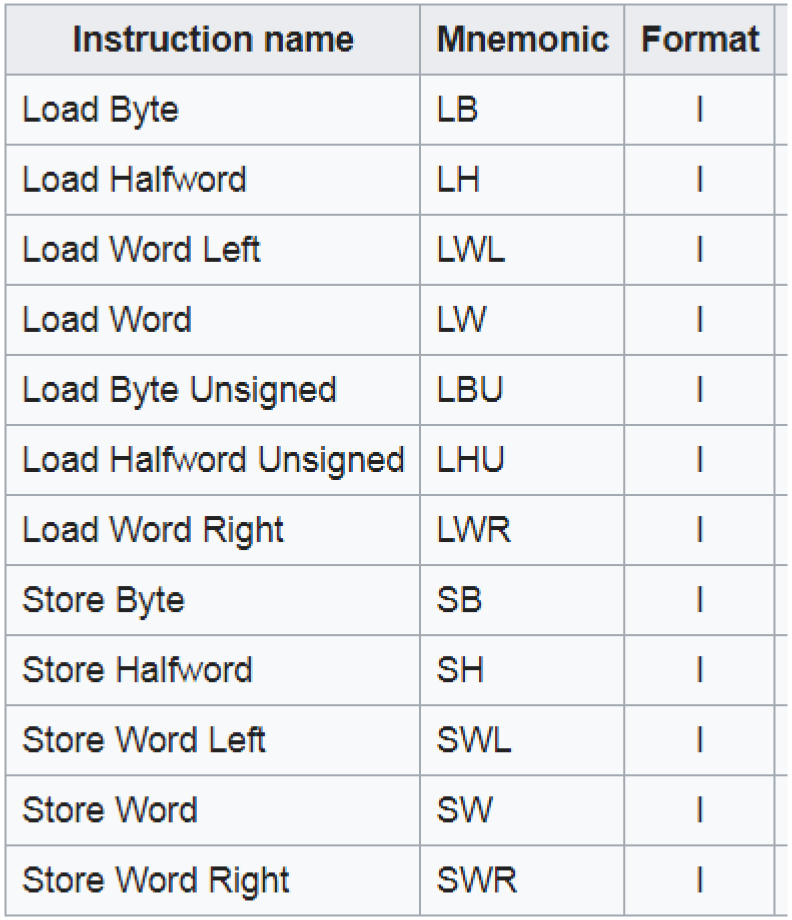
- $zero is the “zero register” which is always zero; writes to it have no effect.

MIPS INSTRUCTION SET

- The instruction set consists of a variety of basic instructions, including:
- 21 arithmetic instructions (+, -, *, /, %)
- 8 logic instructions (&, |, ~, XOR)
- 8 bit manipulation instructions
- 12 comparison instructions (>, <, =, >=, <=, ¬)
- 25 branch/jump instructions
- 15 load instructions
- 10 store instructions
- 8 move instructions
- 4 miscellaneous instructions

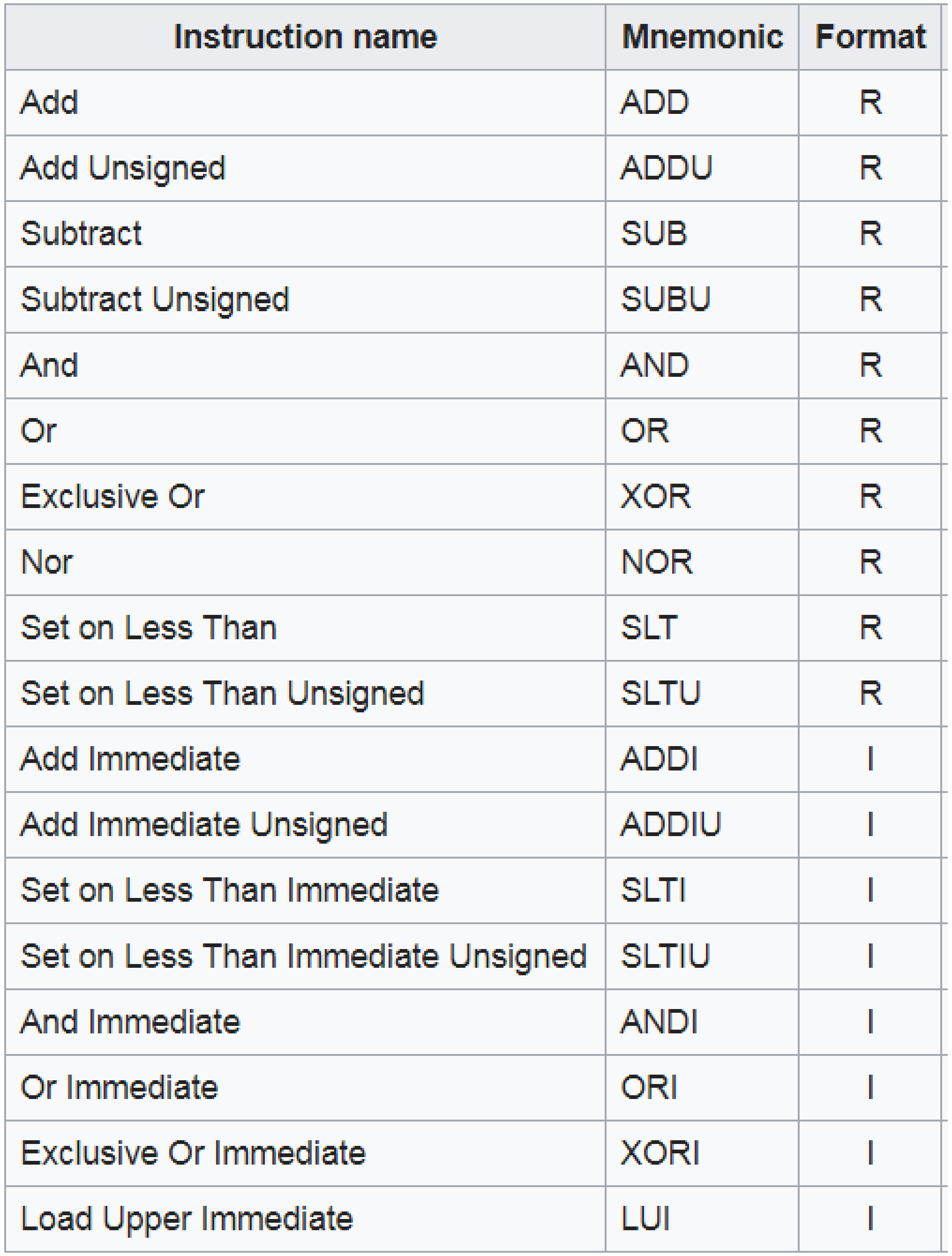
https://en.wikipedia.org/wiki/MIPS_architecture

LOADS AND STORES



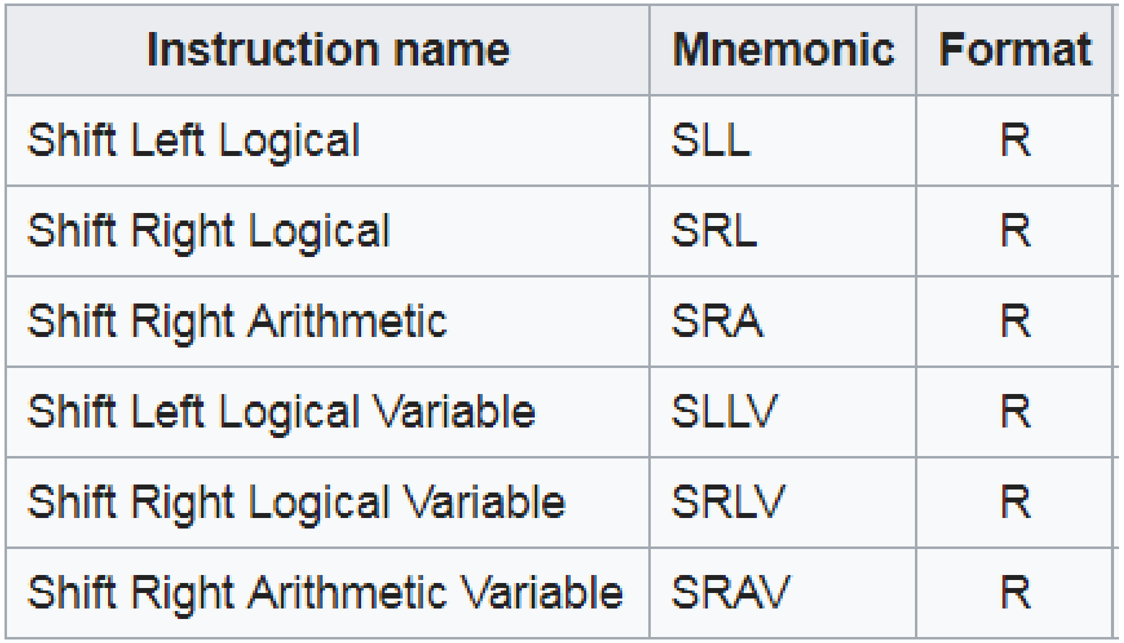
Instruction name	Mnemonic	Format
Load Byte	LB	I
Load Halfword	LH	I
Load Word Left	LWL	I
Load Word	LW	I
Load Byte Unsigned	LBU	I
Load Halfword Unsigned	LHU	I
Load Word Right	LWR	I
Store Byte	SB	I
Store Halfword	SH	I
Store Word Left	SWL	I
Store Word	SW	I
Store Word Right	SWR	I

ALU



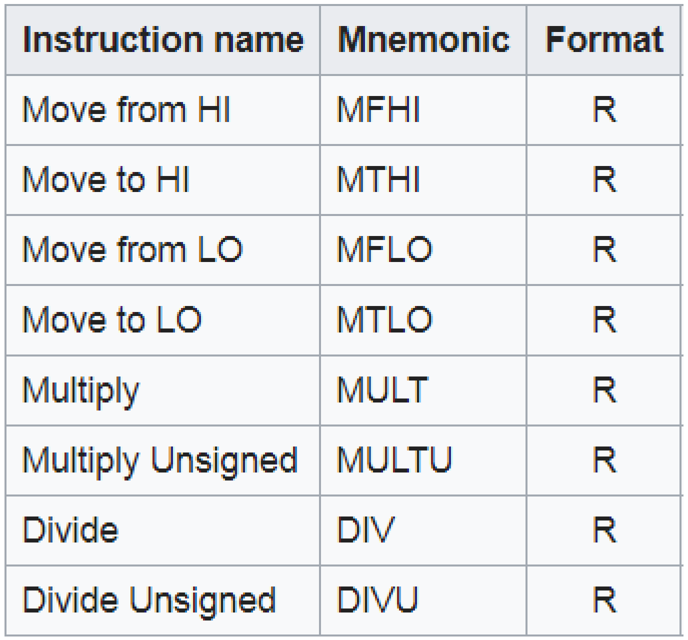
Instruction name	Mnemonic	Format
Add	ADD	R
Add Unsigned	ADDU	R
Subtract	SUB	R
Subtract Unsigned	SUBU	R
And	AND	R
Or	OR	R
Exclusive Or	XOR	R
Nor	NOR	R
Set on Less Than	SLT	R
Set on Less Than Unsigned	SLTU	R
Add Immediate	ADDI	I
Add Immediate Unsigned	ADDIU	I
Set on Less Than Immediate	SLTI	I
Set on Less Than Immediate Unsigned	SLTIU	I
And Immediate	ANDI	I
Or Immediate	ORI	I
Exclusive Or Immediate	XORI	I
Load Upper Immediate	LUI	I

SHIFTS



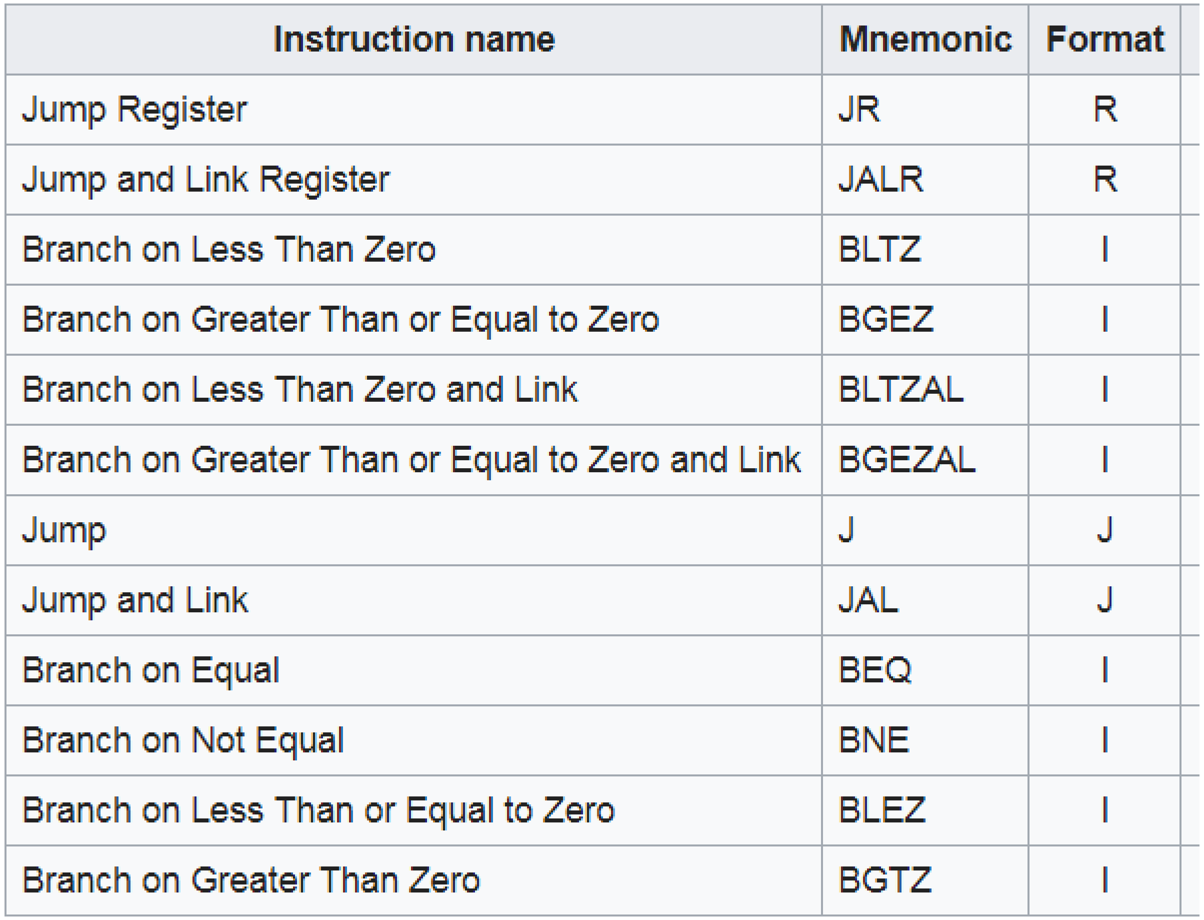
Instruction name	Mnemonic	Format
Shift Left Logical	SLL	R
Shift Right Logical	SRL	R
Shift Right Arithmetic	SRA	R
Shift Left Logical Variable	SLLV	R
Shift Right Logical Variable	SRLV	R
Shift Right Arithmetic Variable	SRAV	R

MULT AND DIV



Instruction name	Mnemonic	Format
Move from HI	MFHI	R
Move to HI	MTHI	R
Move from LO	MFLO	R
Move to LO	MTLO	R
Multiply	MULT	R
Multiply Unsigned	MULTU	R
Divide	DIV	R
Divide Unsigned	DIVU	R

JUMP AND BRANCH



Instruction name	Mnemonic	Format
Jump Register	JR	R
Jump and Link Register	JALR	R
Branch on Less Than Zero	BLTZ	I
Branch on Greater Than or Equal to Zero	BGEZ	I
Branch on Less Than Zero and Link	BLTZAL	I
Branch on Greater Than or Equal to Zero and Link	BGEZAL	I
Jump	J	J
Jump and Link	JAL	J
Branch on Equal	BEQ	I
Branch on Not Equal	BNE	I
Branch on Less Than or Equal to Zero	BLEZ	I
Branch on Greater Than Zero	BGTZ	I

Instruction	**Example**	**Meaning**
Add	Add \$1,\$2,\$3	\$1 = \$2 + \$3
Subtract	Sub \$1,\$2,\$3	\$1 = \$2 - \$3
Add immediate	Addi \$1,\$2,100	\$1 = \$2 + 100
Add unsigned	Addu \$1,\$2,\$3	\$1 = \$2 + \$3
Subtract unsigned	Subu \$1,\$2,\$3	\$1 = \$2 - \$3
Add immediate unsigned	Addiu \$1,\$2,100	\$1 = \$2 + 100
Multiply	mult \$2,\$3	Hi, lo = \$2 * \$3
Multiply unsigned	Multu \$2,\$3	Hi, lo = \$2 * \$3
Divide	Div \$2,\$3	Lo = \$2 / \$3; hi remainder
Divide unsigned	Divu \$2,\$3	Lo = \$2 / \$3; hi remainder

MIPS INSTRUCTION TYPES

- **R-type instructions**, which perform arithmetic and logical operations on registers.
- **I-type instructions**, which deal with load/stores and immediate literal values, as well as branches.
- **J-type instructions**, which are used for jumps and function calls.

.DATA, .TEXT, .GLOBE DIRECTIVES

.DATA directive

- Defines the data segment of a program containing data
- The program's variables should be defined under this directive
- Assembler will allocate and initialize the storage of variables

.TEXT directive

- Defines the code segment of a program containing instructions

.GLOBL directive

- Declares a symbol as global
- Global symbols can be referenced from other files
- We use this directive to declare *main* procedure of a program