VLSI SYSTEMS AND ARCHITECTURE

2021-22

Lecture 4

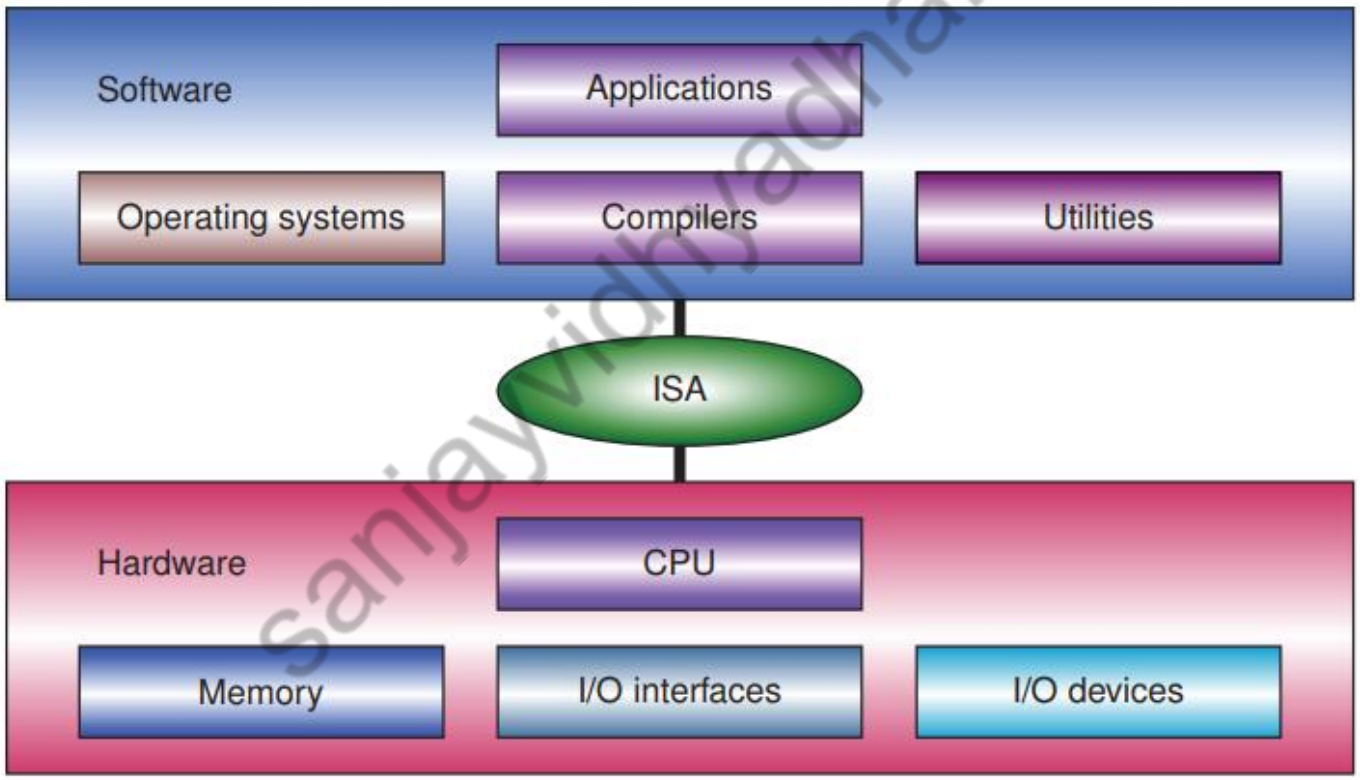
Instruction Set Architecture and MIPS Instructions

By Dr. Sanjay Vidhyadharan

Instruction Set Architecture

ISA is an important issue in hardware/software codesign.

- An ISA tells compiler developers “what a CPU can do,” and
- Tells CPU designers “what a CPU should do.”



Instruction Set Architecture

ISA is an important issue in hardware/software codesign.

- An ISA tells compiler developers "what a CPU can do," and
- Tells CPU designers "what a CPU should do."

An ISA defines the formats of instructions, the operations of instructions, the types of operands, the memory and registers the instructions can access, the byte ordering, and the addressing modes.

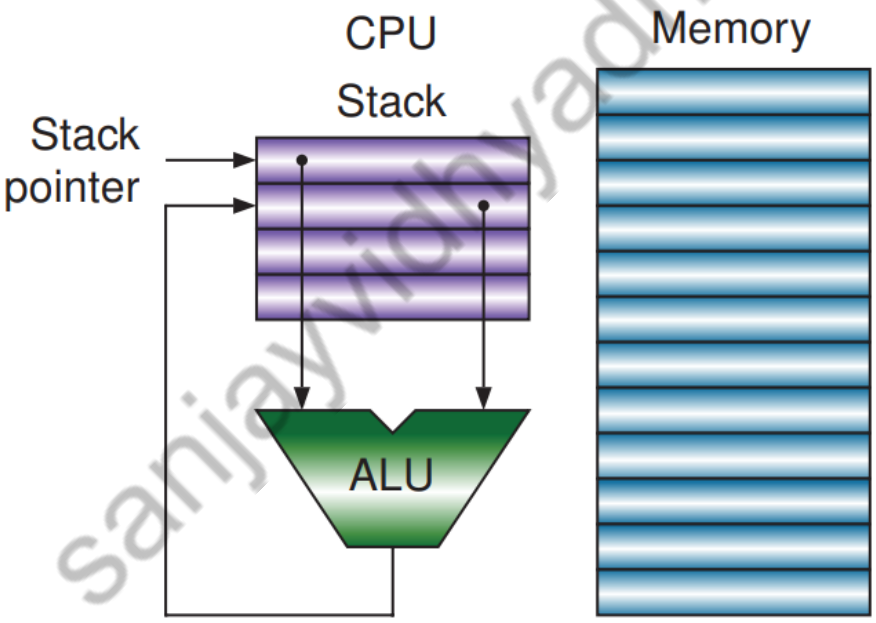
ISA Types:

- **Arithmetic Operation:** *Addition, subtraction, multiplication, division, and square root*
- **Logic Operation:** *Bitwise logical AND, OR, XOR, and NOT*
- **Shift Operation:** *Logical shift left, Logical shift right, Arithmetic shift right, rotate shift, and rotate shift with carry.*
- **Memory Access:** *Load, Store*
- **Input/Output Access:** *In , Out*
- **Control Transfer:** *Jumps, Returns, Call subroutines*
- **Floating-Point Calculation:** *ldf, addf*
- **System Control:** *Interrupts*

Instruction Set Architecture

Stack Architecture

Two source operands are popped from the top of the stack, and the result is pushed onto the stack. The stack has a feature of first-in last-out or last-in first-out; it cannot be accessed randomly. JVM (Java virtual machine) Bytecode uses this architecture.

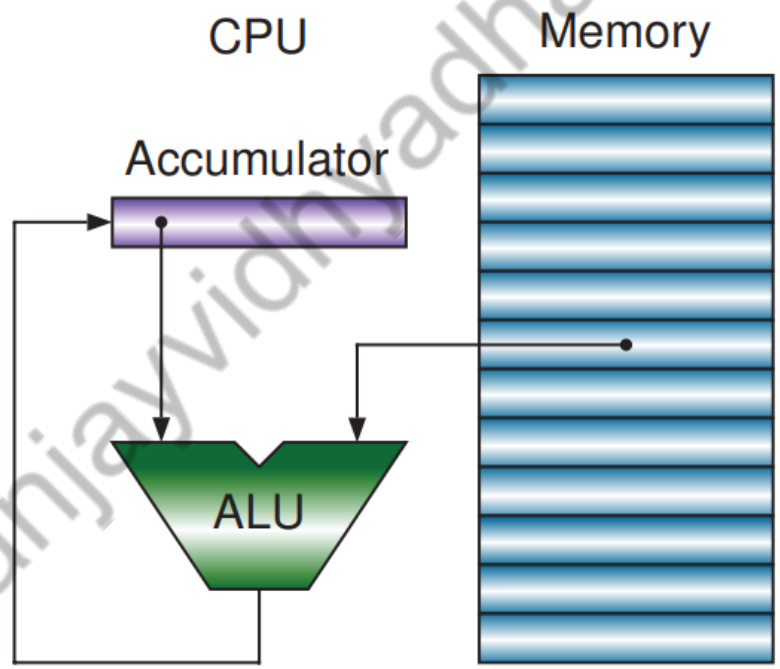


(a) Stack architecture

Instruction Set Architecture

Accumulator Architecture

In an accumulator architecture, one operand is implicitly in the accumulator and the other operand is in the memory or register. The operation result is stored in accumulator implicitly. Z80 and 6502 ISAs use this architecture.

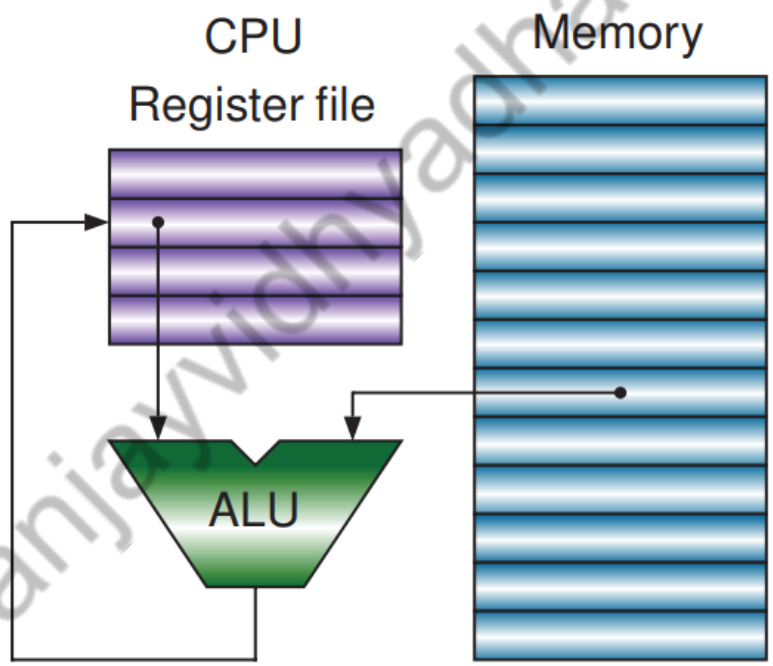


(b) Accumulator architecture

Instruction Set Architecture

Register–Memory Architecture

In a register–memory architecture, one operand is explicitly in a general-purpose register file and the other operand is in the memory. The operation result is stored in the register. The Intel x86 ISA uses this architecture.

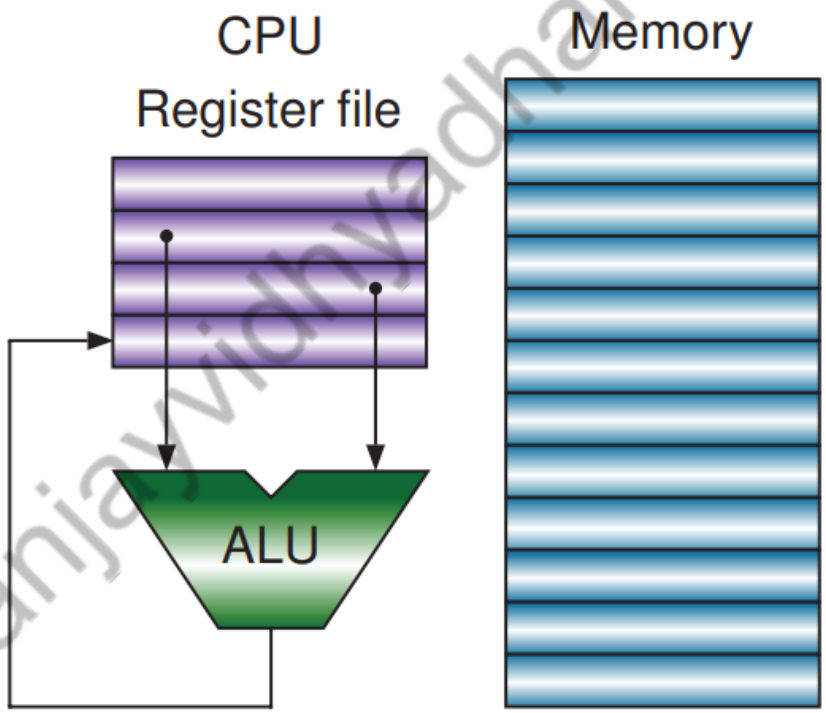


(c) Register–memory architecture

Instruction Set Architecture

Register–Register Architecture

In a register–register architecture, all operands are explicitly in a general-purpose register file. The operation result is also stored in the register. Almost all RISC type ISAs use this architecture.

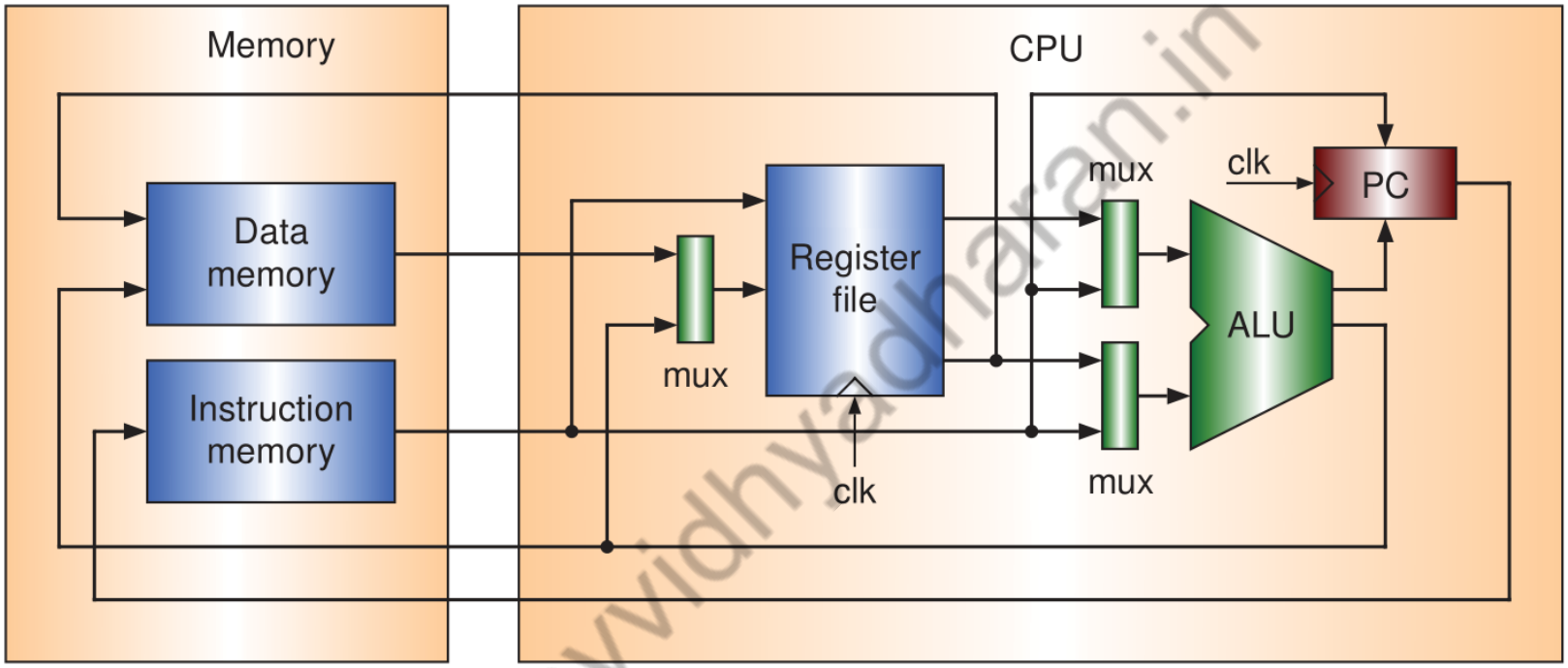


(d) Register–register architecture

MIPS Instructions

MIPS (Microprocessor without Interlocked Pipelined Stages is a family of reduced instruction set computer (RISC) instruction set architectures (ISA) developed by MIPS Computer Systems, now MIPS Technologies, based in the United States.

RISC



1. The instruction is fetched from the instruction memory. The content of PC (program counter) is used as the address of the instruction memory.
2. Each of the two input operands of ALU can be **either a register datum or an immediate provided in the instruction**. The multiplexers are used for selecting an input from two inputs.

MIPS Instructions

Three Types of MIPS Instructions

1. Memory reference: lw, sw
2. ALU : add, sub, and, or, ….
3. Branching : beq

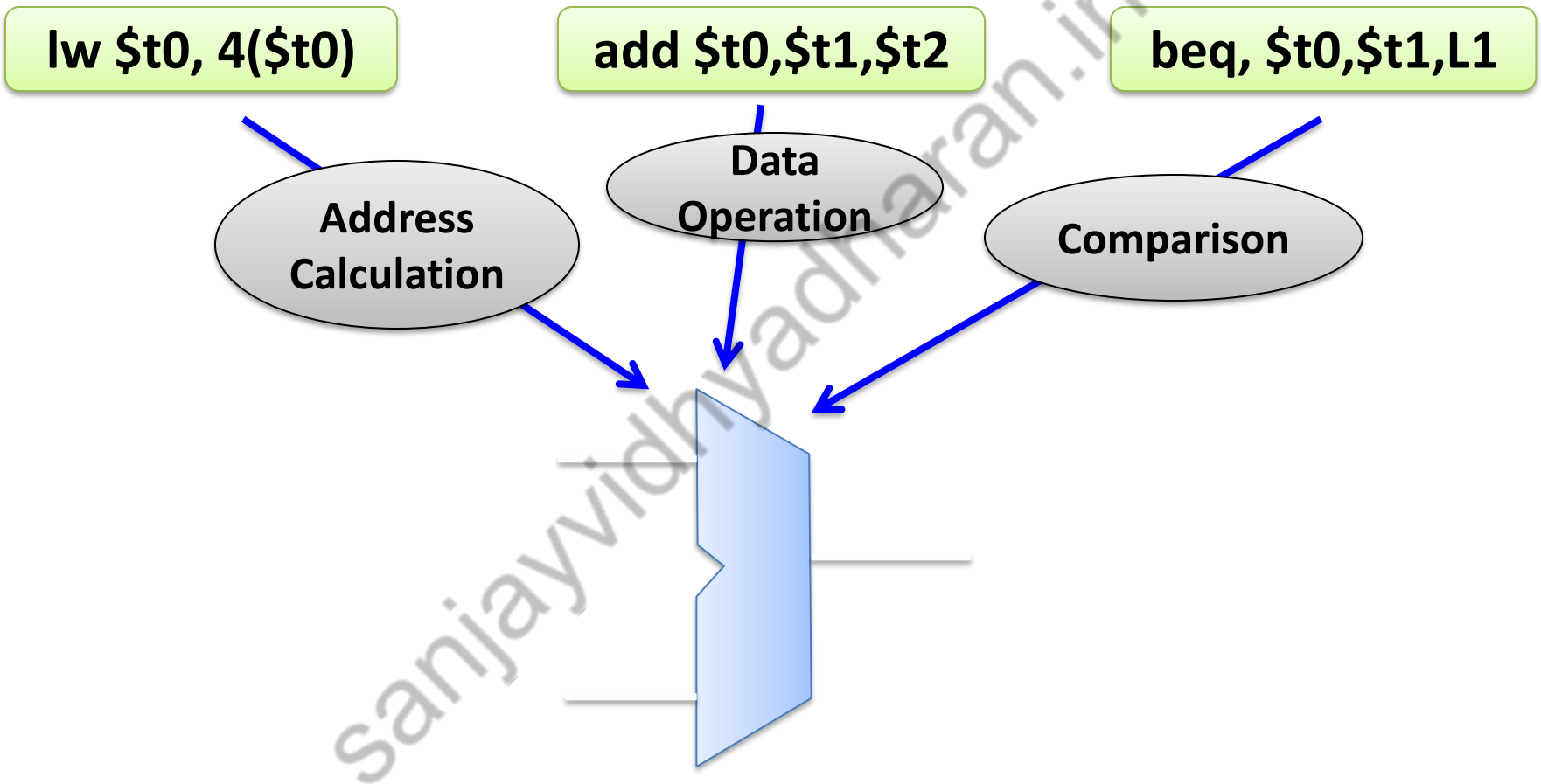
Three instruction encoding formats:

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

31-26	25-21	20-16	15-0
opcode	*rs*	*rt*	*imm*

31-26	25-0
opcode	*pseudodirect jump address*

MIPS Instructions



MIPS Instructions

MIPS general-purpose registers

Register name	Register number	Use
$zero	0	Constant 0
$at	1	Assembler temporary
$v0 to $v1	2 to 3	Function return value
$a0 to $a3	4 to 7	Function parameters
$t0 to $t7	8 to 15	Temporaries
$s0 to $s7	16 to 23	Saved temporaries
$t8 to $t9	24 to 25	Temporaries
$k0 to $k1	26 to 27	Reserved for OS kernel
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

Note that the register 0 is not a register; its content is always a constant 0. Although Table 4.3 gives the convention on the usage of the registers, there are no differences among the remaining 31 registers from the hardware point of view

MIPS Instructions

ALU Instructions

```
add/sub/and/or/xor rd, rs, rt # rd <-- rs op rt
```

These fve instructions have the same format and perform addition, subtraction, AND, OR, and XOR, respectively. `rs` and `rt` are the two source register numbers, and `rd` is the destination register number.

```
sll/srl/sra rd, rt, sa # rd <-- rt shift sa
```

These are three shift instructions: shift left logical (`sll`), shift right logical (`srl`), and shift right arithmetic (`sra`). The shift amount is given by the 5-bit `sa`

```
addi rt, rs, immediate # rt <-- rs + (sign)immediate
```

The `addi` (add immediate) instruction adds a 16-bit signed `immediate` to the 32-bit value in register `rs`. The result of the addition is saved to register `rt`. The 16-bit `immediate` is sign-extended into 32 bits

MIPS Instructions

ALU Instructions

```
andi/ori/xori rt,rs,immediate #rt <--rs op (zero)immediate
```

The `andi` (and immediate), `ori` (or immediate), and `xori` (exclusive or immediate) instructions perform bitwise logical AND, OR, and XOR, respectively, on the value in register `rs` and a 16-bit unsigned `immediate`. The result is written to register `rt`. The 16-bit `immediate` is zero-extended into 32 bits.

```
lui rt, immediate # rt <-- immediate << 16
```

The `lui` (load upper immediate) instruction shifts `immediate` to the left by 16 bit. By using `lui` and `ori` instructions, we can set a register to an any 32-bit constant (`lui` sets high 16 bits and `ori` sets low 16 bits).

MIPS Instructions

Memory reference

```
lw rt, offset(rs) # rt <-- memory[rs + offset]
```

The `lw` (load word) instruction loads a 32-bit word from memory. The memory address is calculated by adding a 16-bit signed immediate (`offset`) to the 32-bit value in register `rs`. The loaded word is saved to register `rt`.

```
sw rt, offset(rs) # memory[rs + offset] <-- rt
```

The `sw` (store word) instruction stores a 32-bit word to memory. The memory address is calculated by adding a 16-bit signed `offset` to the 32-bit value in register `rs`. The word to be stored is in register `rt`.

MIPS Instructions

Branching

```
beq rs, rt, label # if (rs == rt) PC <-- label
```

The `beq` (branch on equal) instruction transfers control to a PC-relative target address (`label`) if the values in registers `rs` and `rt` are equal. The branch target address is calculated by adding an 18-bit signed constant (the 16-bit `offset` shifted to the left by 2 bits) to the address of the instruction following `beq`.

```
bne rs, rt, label # if (rs != rt) PC <-- label
```

The `bne` (branch on not equal) instruction transfers control to a PC-relative target address (`label`) if the values in registers `rs` and `rt` are not equal. The branch target address is calculated by adding an 18-bit signed constant (the 16-bit `offset` shifted to the left by 2 bits) to the address of the instruction following `bne`.

MIPS Instructions

Branching

```
j target # PC <-- target
```

The `j` (jump) instruction transfers control to a target address whose low 28 bits are the 26-bit `target` shifted to the left by 2 bits; the remaining upper bits are the corresponding bits of the address of the instruction following the `j` instruction.

```
jal target # $31 <-- PC + 8; PC <-- target
```

The `jal` (jump and link) is a subroutine call instruction that does the same job as the `j` instruction and meanwhile saves the return address to register `$31`. The return address is the location at which execution continues after returning from the subroutine. MIPS uses the delayed branch technique; the return address is `PC + 8`.

```
jr rs # PC <-- rs
```

The `jr` (jump register) instruction transfers control to a target address given in register `rs`. The return from a subroutine can be done by letting `rs` to be 31

MIPS Instructions

Twenty MIPS integer instructions

Inst.	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	Meaning
add	000000	rs	rt	rd	00000	100000	Register add
sub	000000	rs	rt	rd	00000	100010	Register subtract
and	000000	rs	rt	rd	00000	100100	Register AND
or	000000	rs	rt	rd	00000	100101	Register OR
xor	000000	rs	rt	rd	00000	100110	Register XOR
sll	000000	00000	rt	rd	sa	000000	Shift left
srl	000000	00000	rt	rd	sa	000010	Logical shift right
sra	000000	00000	rt	rd	sa	000011	Arithmetic shift right
jr	000000	rs	00000	00000	00000	001000	Register jump
addi	001000	rs	rt		Immediate		Immediate add
andi	001100	rs	rt		Immediate		Immediate AND
ori	001101	rs	rt		Immediate		Immediate OR
xori	001110	rs	rt		Immediate		Immediate XOR
lw	100011	rs	rt		offset		Load memory word
sw	101011	rs	rt		offset		Store memory word
beq	000100	rs	rt		offset		Branch on equal
bne	000101	rs	rt		offset		Branch on not equal
lui	001111	00000	rt		immediate		Load upper immediate
j	000010			address			Jump
jal	000011			address			Call

MIPS Instructions

MIPS assembler

MIPS assembler supports several pseudo-instructions. A pseudo-instruction will be translated into an actual instruction or a sequence of instructions by the assembler.

Examples

1. A nop (no operation) pseudo-instruction is translated into sll $0, $0, 0, which shifts the content of register $0 to the left by 0 bit and writes the result to register $0. The 32-bit encoding of this instruction is 0.

2. Another pseudo-instruction is `li` (load immediate), which can load a 32-bit immediate to a register. For example, `li $4, 0x1234abcd` loads the 32-bit immediate `0x1234abcd` to register `$4`. The assembler translates `li $4, 0x1234abcd` into a sequence of two instructions: `lui $4, 0x1234` and `ori $4, $4, 0xabcd`. The first instruction writes `0x12340000` to register `$4`, and the second instruction writes the result of `0x12340000` OR `0x0000abcd`, or `0x1234abcd`, to register `$4`.

0x- indicates hexadecimal number

MIPS Instructions

Some MIPS pseudo-instruction examples

Pseudo-instruction	Actual instruction(s)	Meaning
`nop`	`sll $0, $0, 0`	No operation
`clear $4`	`sll $4, $0, 0`	Clear
`move $4, $5`	`sll $4, $5, 0`	Copy content of `$5` to `$4`
`not $4, $5`	`nor $4, $5, $0`	Write inverse of `$5` to `$4`
`subi $4, $5, 1`	`addi $4, $5, -1`	Subtract a 16-bit immediate
`li $4, 0x1234abcd`	`lui $4, 0x1234` `ori $4, $4, 0xabcd`	Load a 32-bit immediate
`la $4, label`	`lui $4, %hi(label)` `ori $4, $4, %lo(label)`	Load a label address
`b label`	`beq $0, $0, label`	Unconditional branch
`bz $4, label`	`beq $4, $0, label`	Branch on zero
`bnz $4, label`	`bne $4, $0, label`	Branch on not zero

References

COMPUTER PRINCIPLES AND DESIGN IN VERILOG HDL
By Yamin Li
Wiley Publications

Thank you