CS510 Computer Architecture

Lecture 03: MIPS CPU Design

Soontae Kim

Spring 2017

School of Computing, KAIST

MIPS64 (64-bit)

Instruction Categories:

- Load/Store.
- Computational.
- Jump and Branch.
- Floating Point (using coprocessor).
- Memory Management.
- Special.

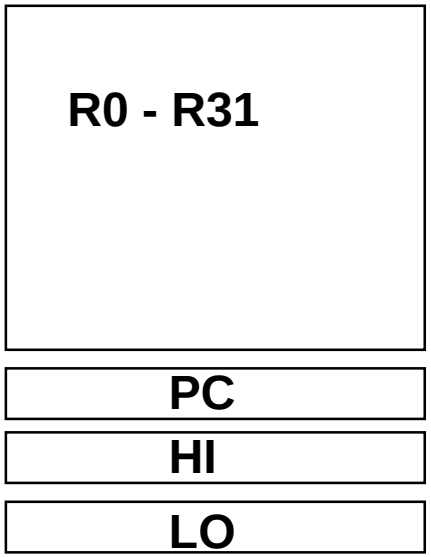
5 Addressing Modes:

- Register direct (arithmetic).
- Immedate (arithmetic).
- Base register + immediate offset (loads and stores).
- PC relative (branches).
- Pseudodirect (jumps)

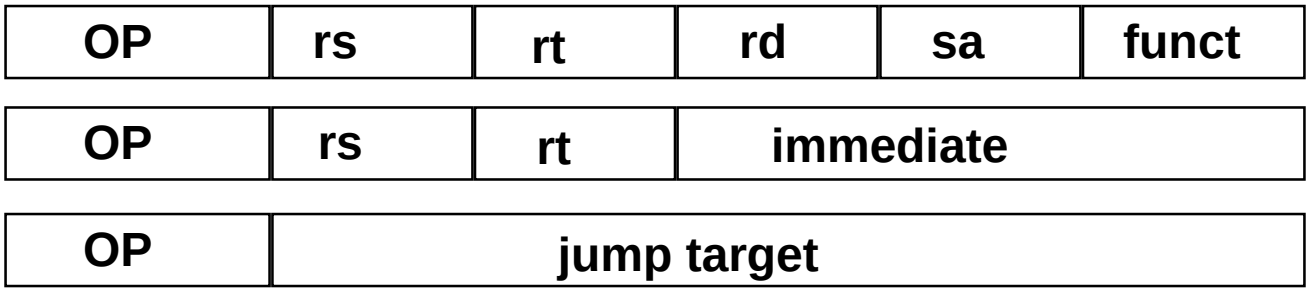
Operand Sizes:

- 1, 2, 4, 8 bytes.

Registers



Instruction Encoding: 3 Instruction Formats, all 32 bits wide.



*** MIPS (Microprocessor without interlocked pipeline stage)**

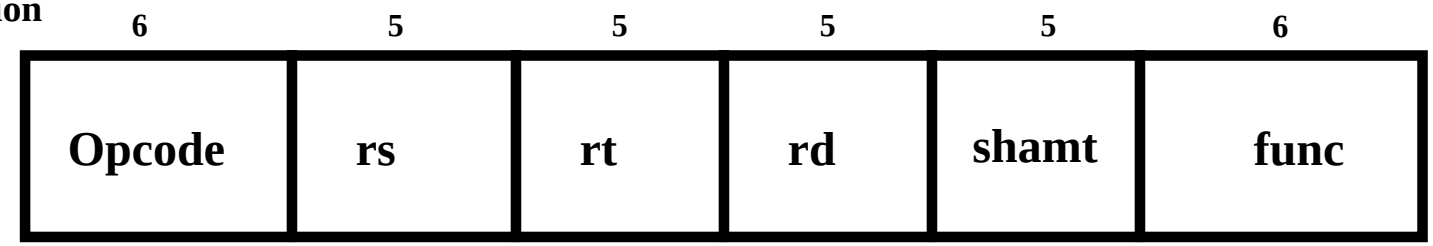
An Instruction Set Example: MIPS64

- **A RISC-type 64-bit instruction set architecture based on instruction set design considerations of Appendix A:**
  - **Use general-purpose registers with a load/store architecture to access memory.**
  - **Reduced number of addressing modes: displacement (offset size of 16 bits), immediate (16 bits).**
  - **Data sizes: 8 (byte), 16 (half word) , 32 (word), 64 (double word) bit integers and 32-bit or 64-bit IEEE 754 floating-point numbers.**
  - **Use fixed instruction encoding (32 bits) for performance.**
  - **32 64-bit general-purpose integer registers GPRs, R0, …., R31. R0 always has a value of zero.**
  - **Separate 32 64-bit floating point registers FPRs: F0, F1 … F31 When holding a 32-bit single-precision number, half of the FPR is not used.**

    separate integer and floating register = to fix the instruciton encoding 32 bit

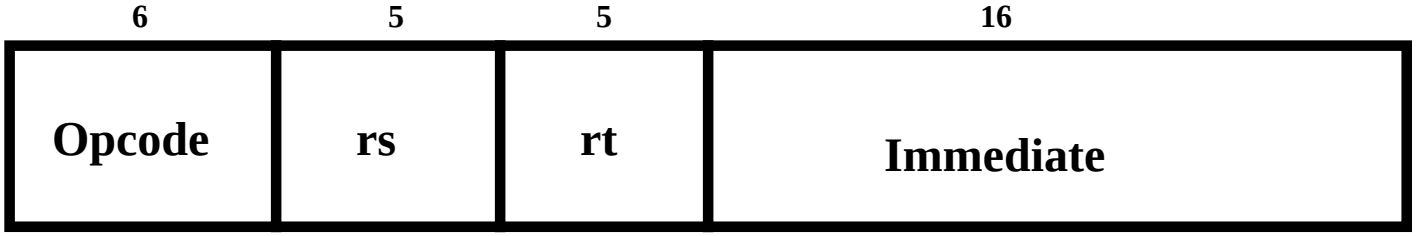
MIPS64 Instruction Format

R - type instruction
(register)



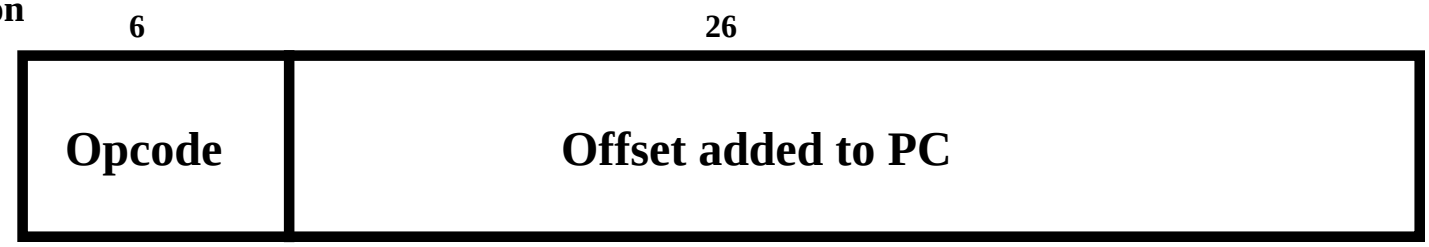
Register-register ALU operations: rd ← rs func rt Function encodes the data path operation: Add, Sub ...

I - type instruction



Encodes: Loads and stores of bytes, words, half words, double words.
All immediates (rt ← rs op immediate) Conditional branch instructions
Jump register, jump and link register (rs = destination, immediate = 0)

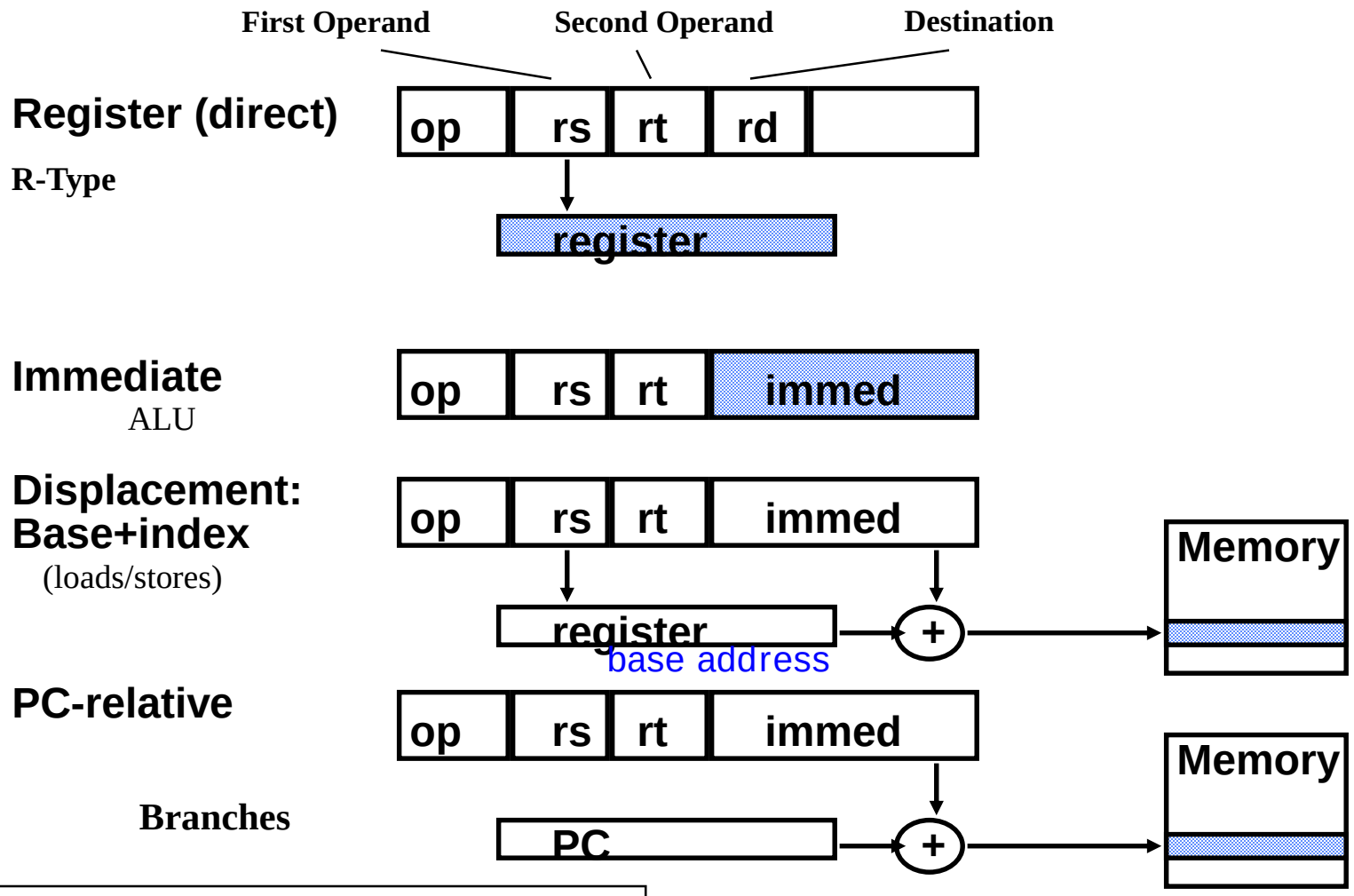
J - Type instruction



Jump, and jump and link

MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide



Pseudodirect Addressing for jumps (J-Type) not shown here

MIPS64 Instructions: Load and Store

LD R1,30(R2)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[30+Regs[R2]]
LW R1, 60(R2)	Load word	Regs[R1] $\leftarrow_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]
LB R1, 40(R3)	Load byte	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{56} ## Mem[40+Regs[R3]]
LBU R1, 40(R3)	Load byte unsigned	Regs[R1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1, 40(R3)	Load half word	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ## Mem[40 + Regs[R3]]
L.S F0, 50(R3)	Load FP single	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0, 50(R2)	Load FP double	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	Store double word	Mem [500+Regs[R4]] $\leftarrow_{64}$ Reg[R3]
SW R3,500(R4)	Store word	Mem [500+Regs[R4]] $\leftarrow_{32}$ Reg[R3]
S.S F0, 40(R3)	Store FP single	Mem [40, Regs[R3]] $\leftarrow_{32}$ Regs[F0] $_{0\ldots31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]] $\leftarrow$-$_{64}$ Regs[F0]
SH R3, 502(R2)	Store half	Mem[502+Regs[R2]] $\leftarrow_{16}$ Regs[R3]$_{48\ldots63}$
SB R2, 41(R3)	Store byte	Mem[41 + Regs[R3]] $\leftarrow_{8}$ Regs[R2] $_{56\ldots63}$

the lenghth of the data being transferred

mean 32 bit extend

selection of a bit from a field(most-signifiant bit starting at 0)

signed extend 4 byte + original 4 byte

MIPS64 Instructions: Arithmetic/Logical

DADDU R1, R2, R3 **Add unsigned** **Regs[R1] ← Regs[R2] + Regs[R3]**

DADDI R1, R2, #3 **Add immediate** **Regs[R1] ← Regs[R2] + 3**

immediate

LUI R1, #42 **Load upper immediate** **Regs[R1] ← 0^{32} ##42 ## 0^{16}**

32bit의 upper 16bit를 immediate로 set

DSLL R1, R2, #5 **Shift left logical** **Regs[R1] ← Regs [R2] <<5**

DSLT R1, R2, R3 **Set less than** **if (regs[R2] < Regs[R3])**

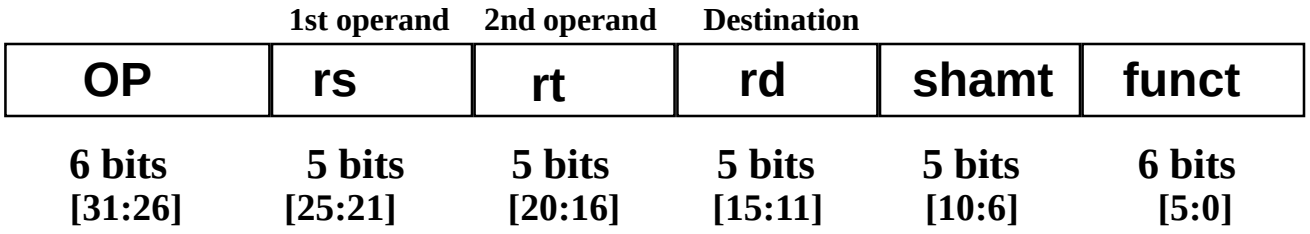
Regs [R1] ← 1 else Regs[R1] ← 0

MIPS64 Instructions: Control-Flow

J name	**Jump**	**$PC_{36..63} \leftarrow$ name**
JAL name	**Jump and link**	**Regs[31] $\leftarrow$ PC+4; $PC_{36..63} \leftarrow$ name;** **((PC+4)- 2^{27}) $\leq$ name < ((PC + 4) + 2^{27})**
JALR R2	**Jump and link register**	**Regs[R31] $\leftarrow$ PC+4; PC $\leftarrow$ Regs[R2]**
JR R3	**Jump register**	**PC $\leftarrow$ Regs[R3]**
BEQZ R4, name	**Branch equal zero**	**if (Regs[R4] ==0) $PC_{46..63} \leftarrow$ name;** **((PC+4) -2^{17}) $\leq$ name < ((PC+4) + 2^{17}**
BNEZ R4, Name	**Branch not equal zero**	**if (Regs[R4] != 0) $PC_{46..63} \leftarrow$ name** **((PC+4) - 2^{17}) $\leq$ name < ((PC +4) + 2^{17}**
MOVZ R1,R2,R3	**Conditional move if zero**	**if (Regs[R3] ==0) Regs[R1] $\leftarrow$ Regs[R2]**

MIPS R-Type (ALU) Instruction Fields

R-Type: All ALU instructions that use three registers



- **op: Opcode, basic operation of the instruction.**
  - For R-Type op = 0
- **rs: The first register source operand.**
- **rt: The second register source operand.**
- **rd: The register destination operand.**
- **shamt: Shift amount used in constant shift operations.**
- **funct: Function, selects the specific variant of operation in the op field.**

Rs, rt , rd
are register specifier fields

Independent RTN: register transfer notation

Instruction Word ← Mem[PC]
R[rd] ← R[rs] funct R[rt]
PC ← PC + 4

Funct field value examples:
Add = 32 Sub = 34 AND = 36 OR =37 NOR = 39

Operand register in rs

Destination register in rd

Operand register in rt

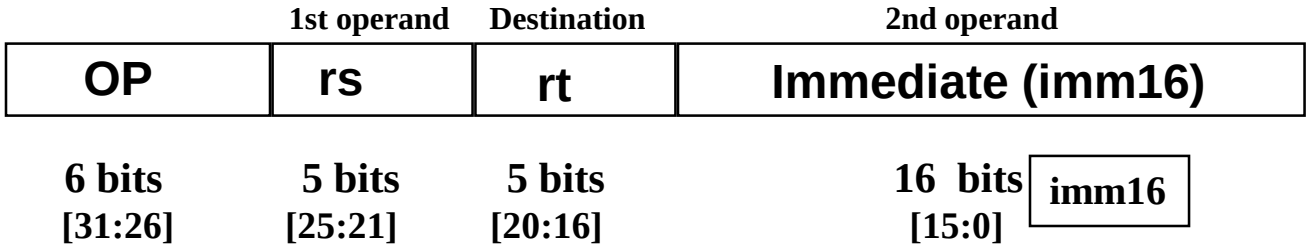
Examples:

add $1,$2,$3
sub $1,$2,$3

and $1,$2,$3
or $1,$2,$3

MIPS ALU I-Type Instruction Fields

I-Type ALU instructions that use two registers and an immediate value
Loads/stores, conditional branches.



- **op: Opcode, operation of the instruction.**
- **rs: The register source operand.**
- **rt: The result destination register.**
- **immediate: Constant second operand for ALU instruction.**

Independent RTN for addi:

Instruction Word ← Mem[PC]
R[rt] ← R[rs] + imm16
PC ← PC + 4

Examples:

add immediate: addi $1,$2,100

and immediate andi $1,$2,10

OP = 8

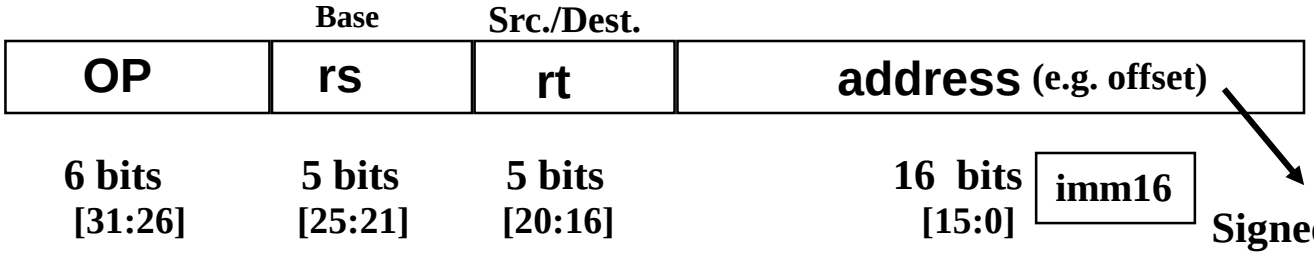
OP = 12

Result register in rt

Source operand register in rs

Constant operand in immediate

MIPS Load/Store I-Type Instruction Fields



- **op: Opcode, operation of the instruction.**
  - For load word op = 35, for store word op = 43.
- **rs: The register containing memory base address.**
- **rt: For loads, the destination register. For stores, the source register of value to be stored.**
- **address: 16-bit memory address offset in bytes added to base register.**

Examples:

Store word: **sw $3, 500($4)**

source register in rt — Offset — base register in rs

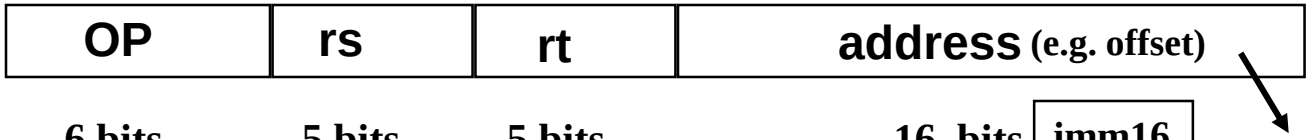
Instruction Word ← Mem[PC]
Mem[R[rs] + imm16] ← R[rt]
PC ← PC + 4

Load word: **lw $1, 32($2)**

Destination register in rt — Offset — base register in rs

Instruction Word ← Mem[PC]
R[rt] ← Mem[R[rs] + imm16]
PC ← PC + 4

MIPS Branch I-Type Instruction Fields



- **op: Opcode, operation of the instruction.**
- **rs: The first register being compared**
- **rt: The second register being compared.**
- **address: 16-bit memory address branch target offset in words added to PC to form branch address.**

Examples:

OP = 4 **Branch on equal** **beq $1,$2,100**

OP = 5 **Branch on not equal** **bne $1,$2,100**

Register in rs

Register in rt

offset in bytes equal to instruction address field x 4

Added to PC+4 to form branch target

Independent RTN for beq:

Instruction Word ← Mem[PC]
R[rs] = R[rt] : PC ← PC + 4 + imm16 x 4
R[rs] ≠ R[rt] : PC ← PC + 4

MIPS J-Type Instruction Fields

J-Type: Include jump j, jump and link jal

6 bits
[31:26]

26 bits
[25:0]

Jump target in words

Word = 4 bytes

- **op: Opcode, operation of the instruction.**
  - Jump j op = 2
  - Jump and link jal op = 3
- **jump target: jump memory address in words.**

Jump memory address in bytes equal to instruction field jump target x 4

Examples:

Jump **j 10000**

Jump and link **jal 10000**

Effective 32-bit jump address: **PC(31-28)##jump_target##00**

PC(31-28)

From PC+4

4 bits **26 bits** **2 bits**

Independent RTN for j:

Instruction Word ← Mem[PC]
PC ← PC + 4
PC ← PC(31-28)##jump_target##00

Overview of MIPS Instruction Micro-operations

- **All instructions go through these common steps:**
  - Send program counter to instruction memory and fetch the instruction. (*fetch)* Instruction ← Mem[PC]
  - Update the program counter to point to next instruction PC ← PC + 4
  - Read one or two registers, using instruction fields. (*decode*)
- **Additional instruction execution actions (*execution*) depend on the instruction in question, but similarities exist:**
  - All instruction classes (except J type) use the ALU after reading the registers:
    - **Memory reference instructions use it for address calculation.**
    - **Arithmetic and logic instructions (R-Type), use it for the specified operation.**
    - **Branches use it for comparison.**
- **Additional execution steps where instruction classes differ:**
  - Memory reference instructions: Access memory for a load or store and write load data to register.
  - Arithmetic and logic instructions: Write ALU result back in register.
  - Branch instructions: Change next instruction address based on comparison.

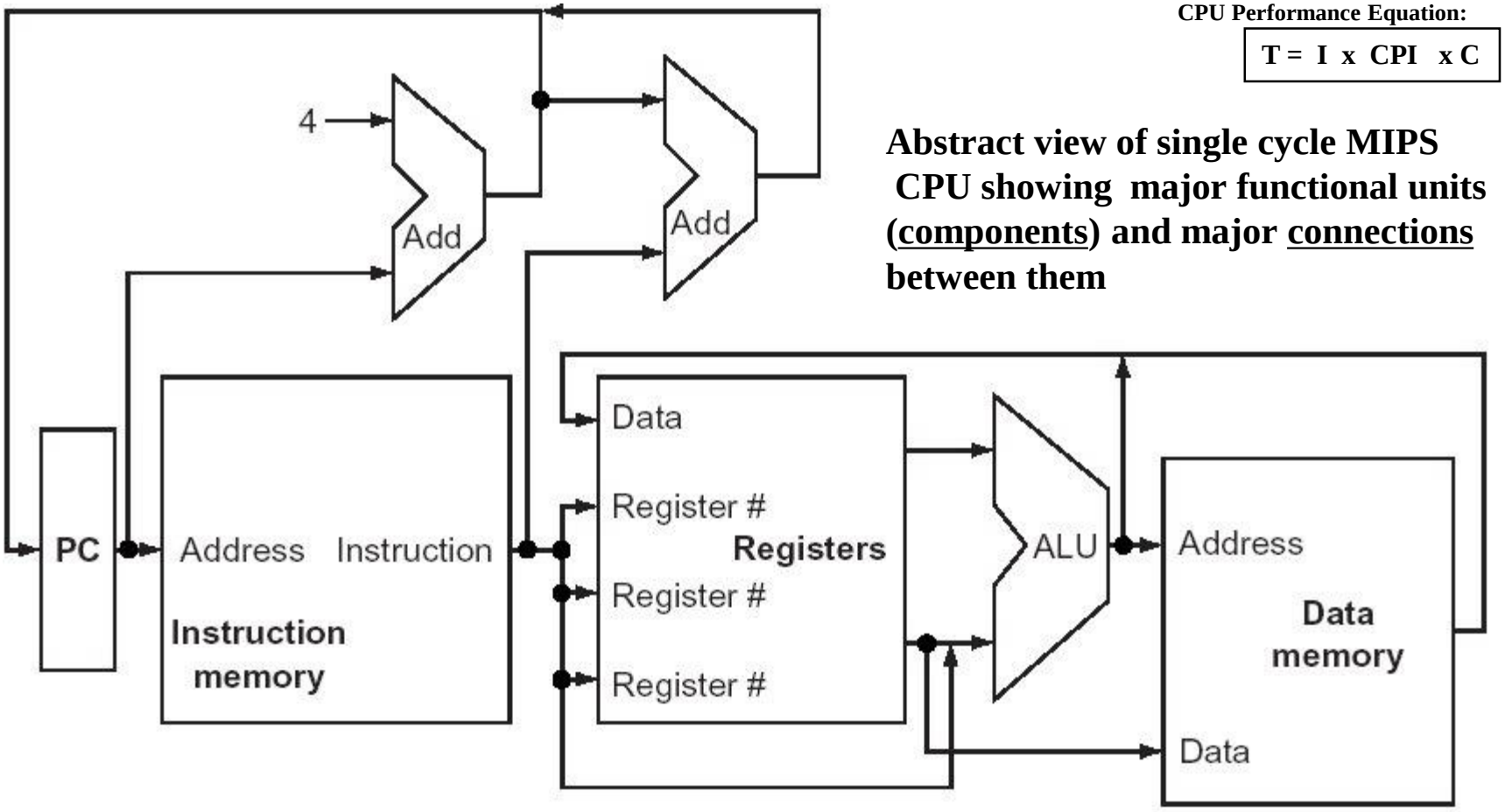
A Single Cycle MIPS CPU Design

Design target: **A single-cycle per instruction MIPS CPU design**

All micro-operations of an instruction are to be carried out in a single CPU clock cycle. Cycles Per Instruction = CPI = 1

CPU Performance Equation:

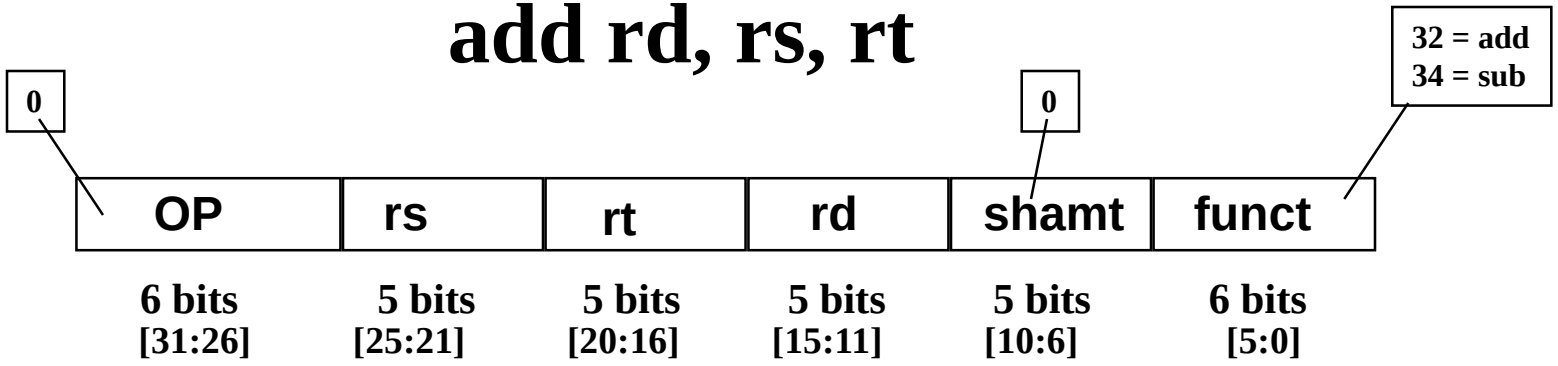
$$T = I \times CPI \times C$$



Abstract view of single cycle MIPS CPU showing major functional units (components) and major connections between them

R-Type Example: Micro-Operation Sequence For ADD

add rd, rs, rt



Instruction Word ← Mem[PC]

PC ← PC + 4

R[rd] ← R[rs] + R[rt]

Program Memory

i.e Funct =add

Fetch the instruction

Increment PC

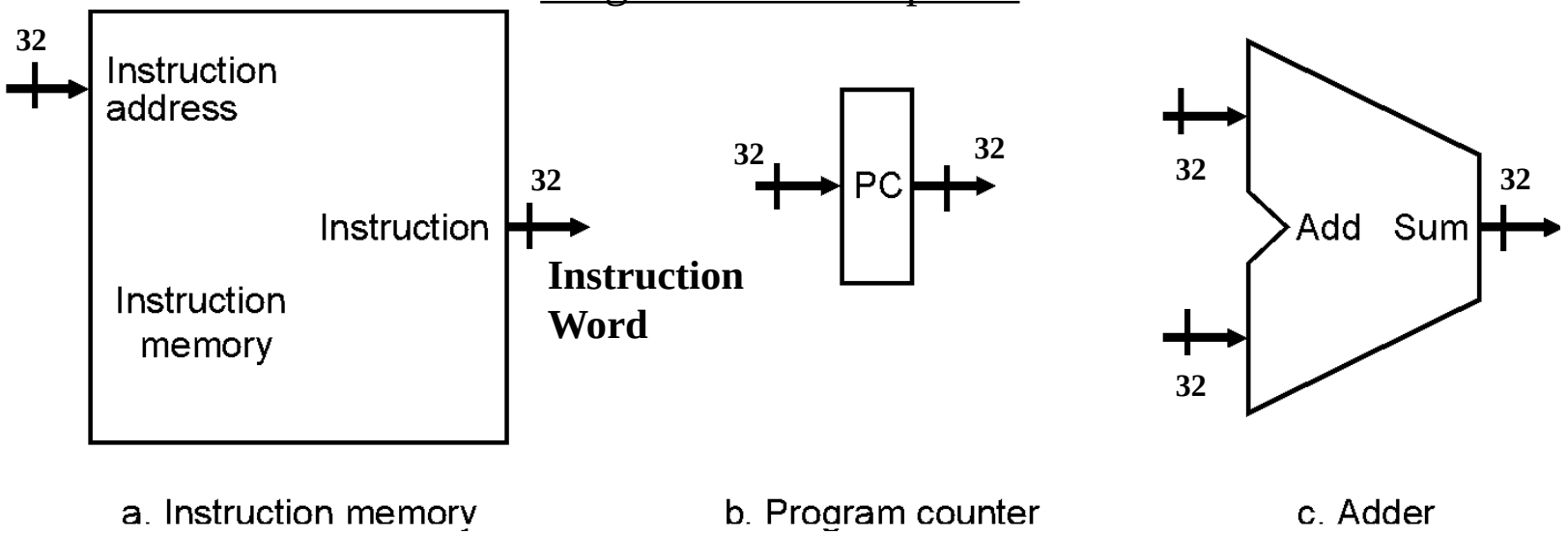
Common Steps

Add register rs to register rt, result in register rd

Initial Datapath Components

Three components needed by: Instruction Fetch: **Instruction ← Mem[PC]**

Program Counter Update: **PC ← PC + 4**



Two state elements (memory) needed to store and access instructions:

1 **Instruction memory:**

- **Only read access (by user code). No read control signal needed.**

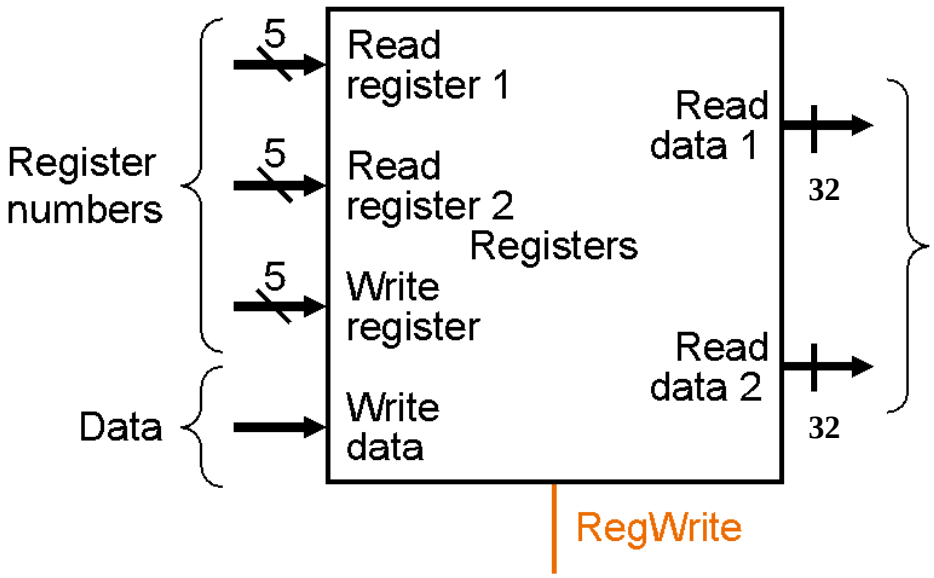
2 **Program counter (PC): 32-bit register.**

- **Written at end of every clock cycle (edge-triggered) : No write control signal.**

3 **32-bit Adder: To compute the the next instruction address (PC + 4).**

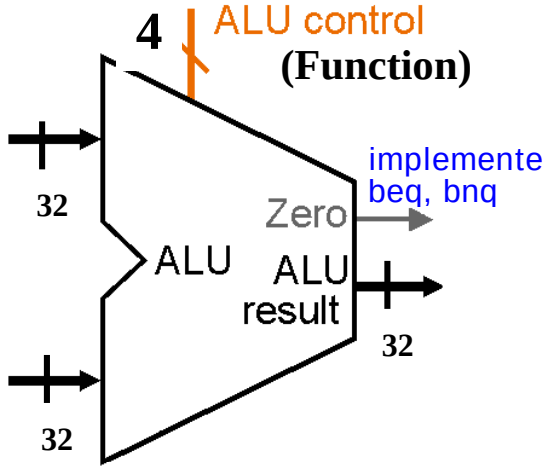
More Datapath Components

ISA Register File



a. Registers

Main 32-bit ALU



32-bit Arithmetic and Logic Unit (ALU)

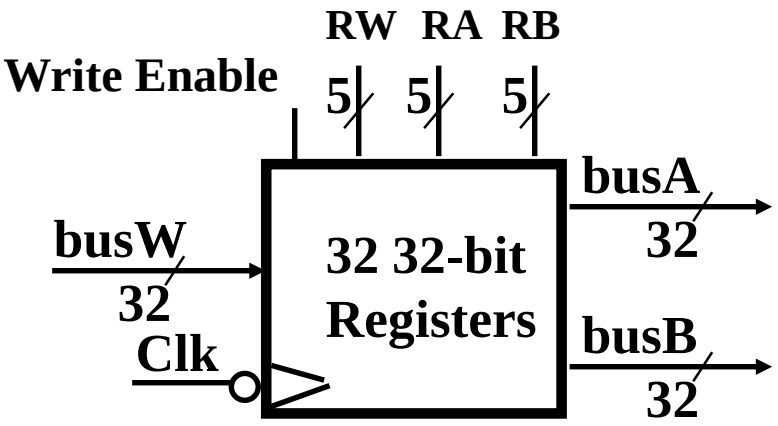
b. ALU

Register File:

- **Contains all ISA registers.**
- **Two read ports and one write port.**
- **Register writes by asserting write control signal**
- **<u>Clocking Methodology:</u> <u>Writes are edge-triggered</u>.**
  - **Thus can read and write to the same register in the same clock cycle.**

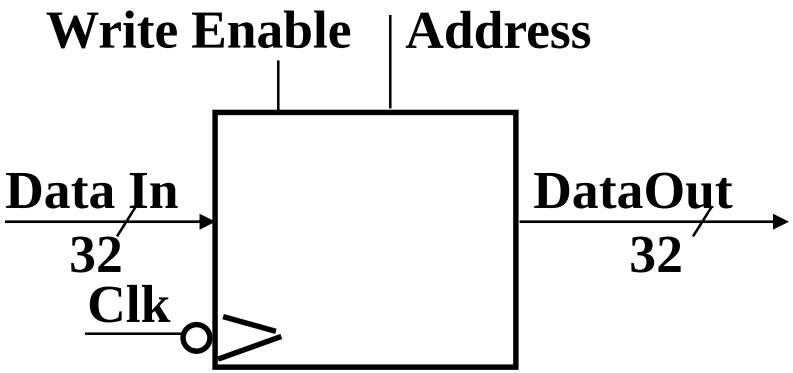
Zero = Zero flag = 1
When ALU result equals zero

Register File Details



- **Register File consists of 32 registers:**
  - Two 32-bit output busses:
    busA and busB
  - One 32-bit input bus: busW
- **Register is selected by:**
  - RA (number) selects the register to put on busA (data):
    busA = R[RA]
  - RB (number) selects the register to put on busB (data):
    busB = R[RB]
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
    Write Enable: R[RW] ← busW
- **Clock input (CLK)**
  - The CLK input is a factor ONLY during write operations.
  - During read operation, it behaves as a combinational logic block:
    - **RA or RB valid => busA or busB valid after "access time."**

Idealized Memory



- **Memory (idealized)**
  - One input bus: Data In.
  - One output bus: Data Out.
- **Memory word is selected by:**
  - Address selects the word to put on Data Out bus.
  - Write Enable = 1: address selects the memory word to be written via the Data In bus.
- **Clock input (CLK):**
  - The CLK input is a factor ONLY during write operation,
  - During read operation, this memory behaves as a combinational logic block:
    - **Address valid => Data Out valid after "access time."**
- **Ideal Memory = Short access time.**

Compared to other components