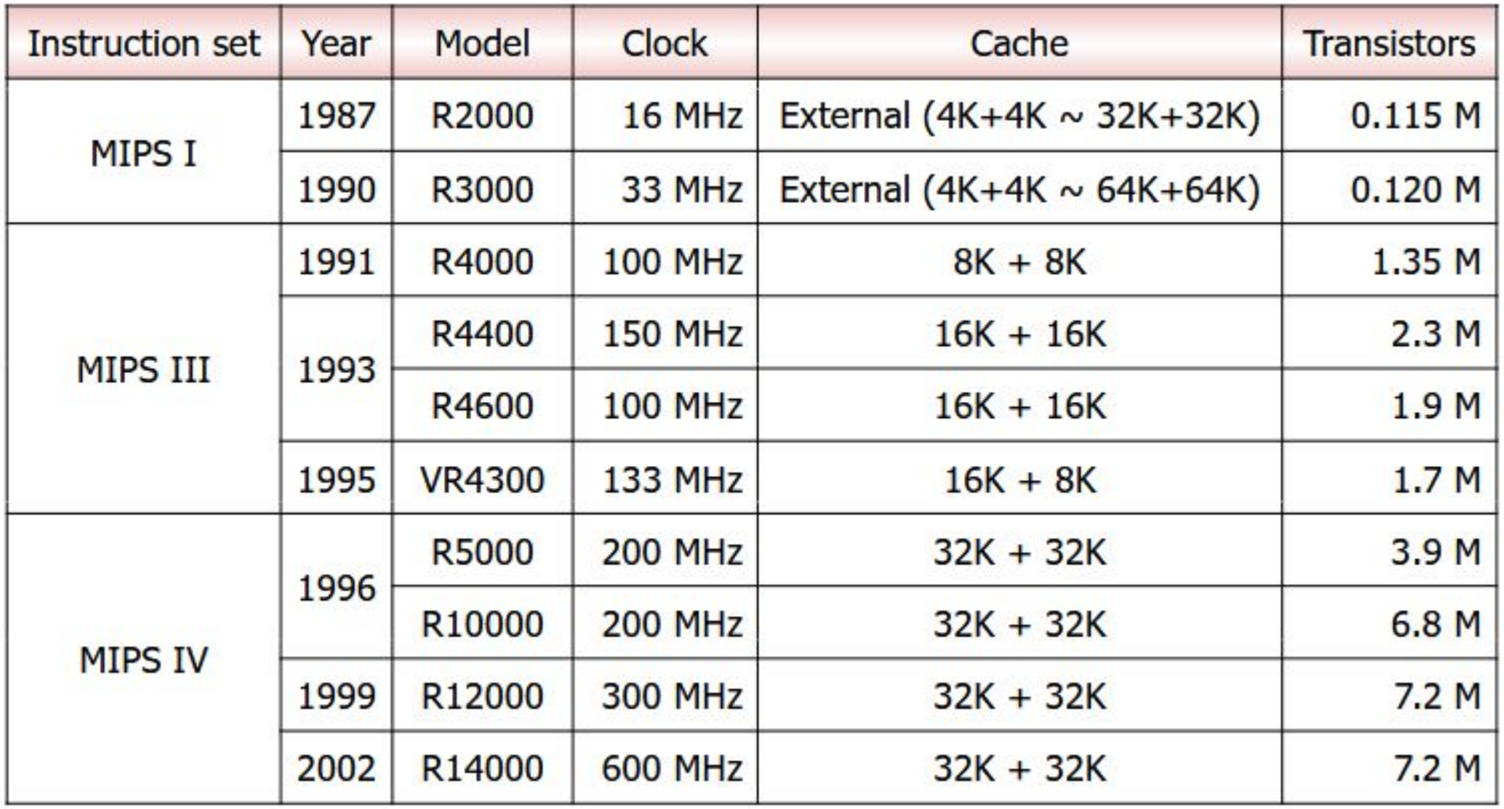
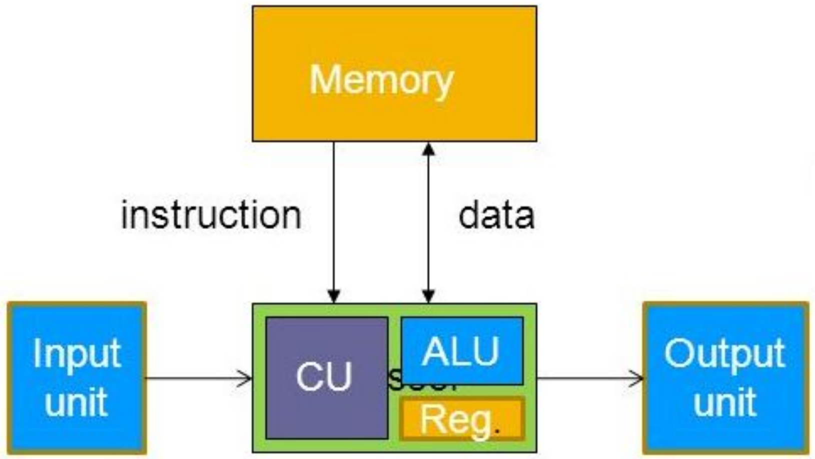
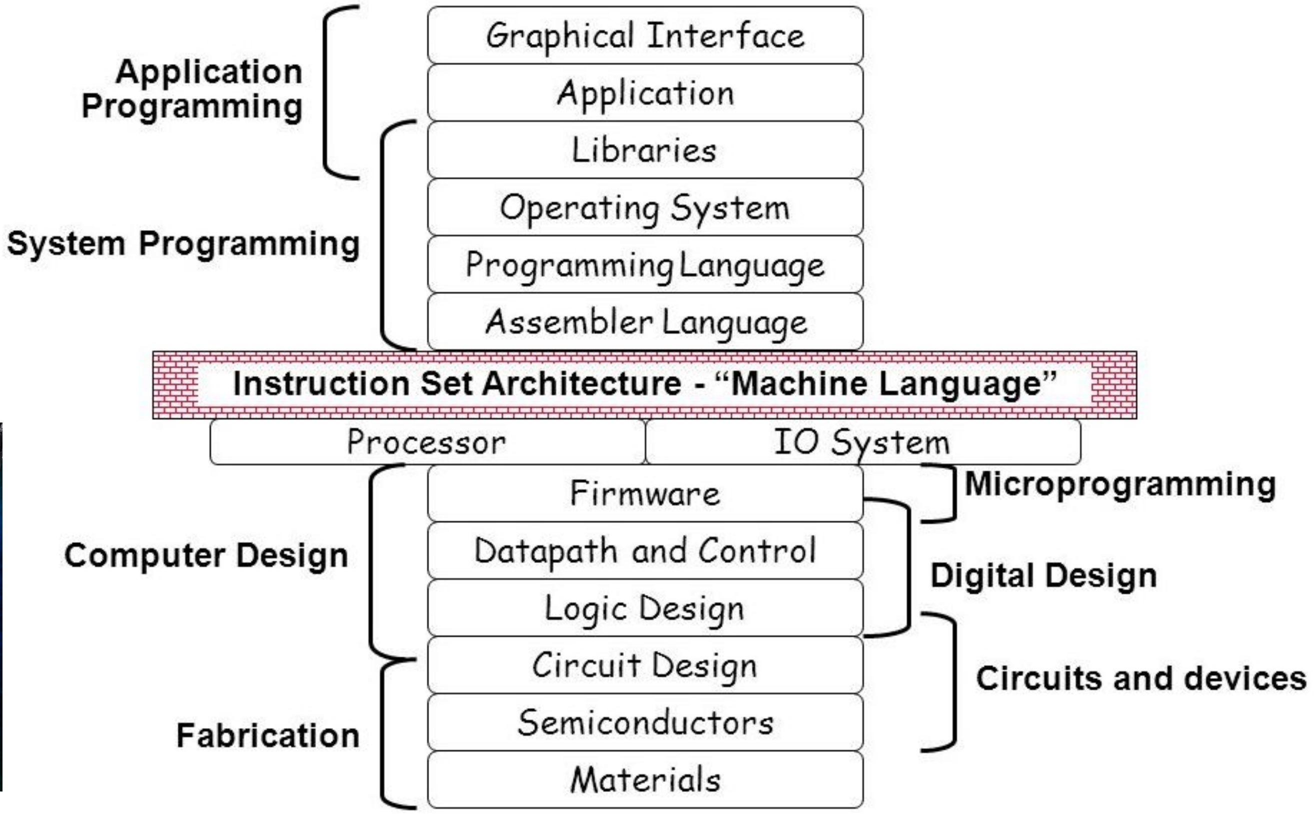
MIPS architecture
&
Instruction Set



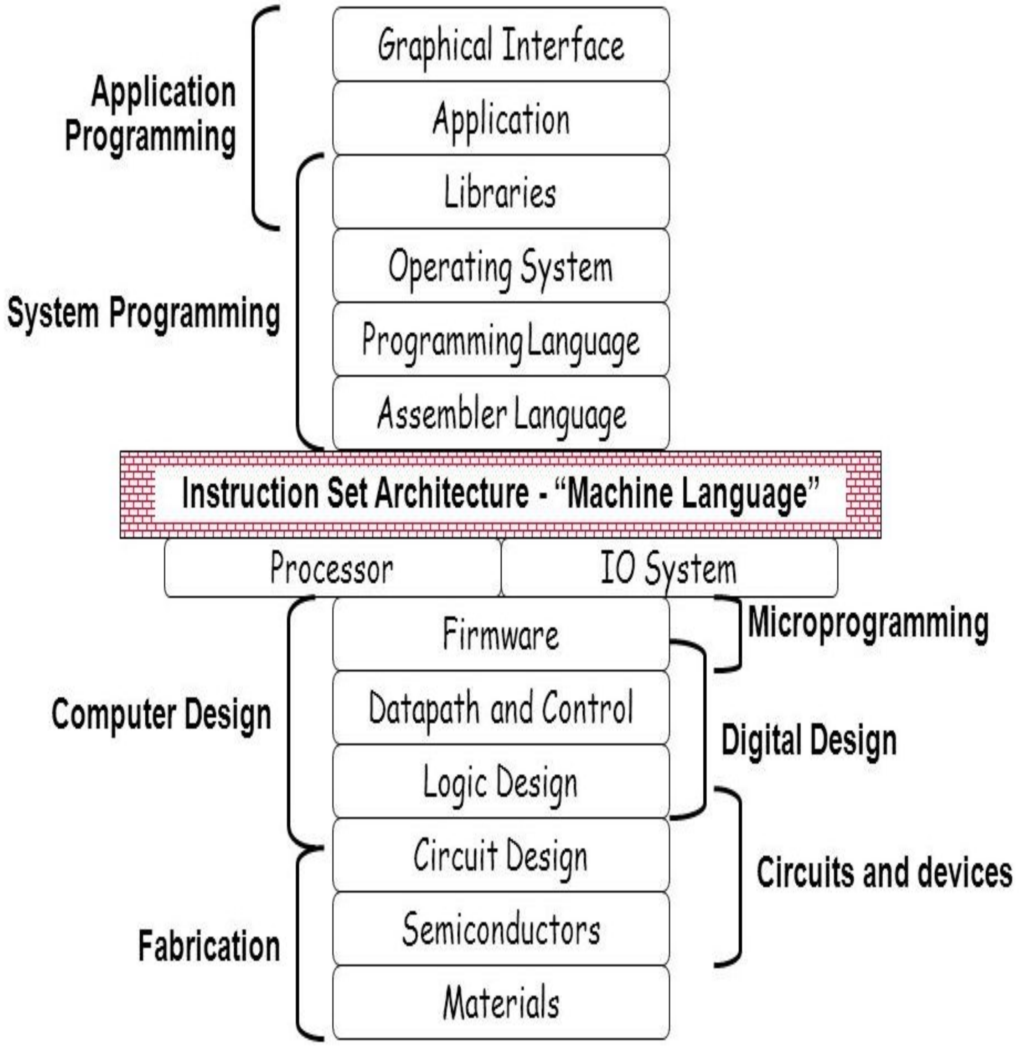
Instruction set	Year	Model	Clock	Cache	Transistors
MIPS I	1987	R2000	16 MHz	External (4K+4K ~ 32K+32K)	0.115 M
	1990	R3000	33 MHz	External (4K+4K ~ 64K+64K)	0.120 M
MIPS III	1991	R4000	100 MHz	8K + 8K	1.35 M
	1993	R4400	150 MHz	16K + 16K	2.3 M
		R4600	100 MHz	16K + 16K	1.9 M
	1995	VR4300	133 MHz	16K + 8K	1.7 M
MIPS IV	1996	R5000	200 MHz	32K + 32K	3.9 M
		R10000	200 MHz	32K + 32K	6.8 M
	1999	R12000	300 MHz	32K + 32K	7.2 M
	2002	R14000	600 MHz	32K + 32K	7.2 M


Memory
instruction
data
Input unit
CU
ALU
Reg.
Output unit


Application Programming
System Programming
Graphical Interface
Application
Libraries
Operating System
Programming Language
Assembler Language
Instruction Set Architecture - "Machine Language"
Processor
IO System
Firmware
Datapath and Control
Logic Design
Circuit Design
Semiconductors
Materials
Computer Design
Fabrication
Microprogramming
Digital Design
Circuits and devices

Instruction Set Architecture

- The instruction set provides commands to the processor, to tell it what it needs to do.
- The instruction set consists of addressing modes, instructions, data types, registers, memory architecture, I/O mechanisms etc.
- The processor's architecture and instruction set determine how many cycles, or ticks, are needed to execute a given instruction.
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction set.



Classification of ISA

- An ISA may be classified in a number of different ways. A common classification is by architectural *complexity*.
- Complex instruction set computer (CISC) has many specialized instructions, some of which may only be rarely used in practical programs.
- Reduced instruction set computer (RISC) simplifies the processor by efficiently implementing only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines, having their resulting additional processor execution time offset by infrequent use.
- Very long instruction word (VLIW) architectures, and the closely related *long instruction word* (LIW) and *explicitly parallel instruction computing* (EPIC) architectures. These architectures seek to exploit instruction-level parallelism with less hardware than RISC and CISC by making the compiler responsible for instruction issue and scheduling.

Mainstream Architecture Design Philosophy

- CISC — Complex Instruction Set Computers
  - IBM 360 (a legend)
  - Digital VAX
  - Intel x86
- RISC — Reduced Instruction Set Computers
  - IBM 801 project
  - Sun Sparc (from Berkeley)
  - MIPS (from Stanford)
- EPIC — Explicitly Parallel Instruction Computing
  - Intel/HP's answer to 64-bit ISA
  - Evolved from
    - VLIW (Very Long Instruction Word): Adopted by most DSP
    - Polycyclic architecture (TRW-ECL)
    - Cydra5 (Cydrome)
    - HP Playdoh
  - Challenged by AMD's Optaron (x86-64)

Mainstream ISAs

- Intel 80x86
  - Used in Macbooks and PCs
  - Found in Core i3, Core i5, Core i7, etc.
- Advanced RISC Machine (ARM)
  - Smart phone-like devices: iPhone, iPad, iPod, etc.
  - The most popular RISC (20x more common than 80x86)
- MIPS
  - Networking equipment, PS2, PSP
  - Very similar to ARM

Background of RISC

- IBM RISC technology originated in 1974 in a project to design a large telephone-switching network capable of handing an average of three hundred calls per second. With an approximate 20 000 instructions per call and Stringent real-time response requirements, the performance target was 12 million instructions per second (MIPS).
- This specialized application required a very fast processor, but did not have to perform computed instructions and had little demand for floating-point calculations. Other than moving data between registers and memory, the machine had to be able to add, combine fields extracted from several registers, perform branches, and carry out input/output operations.

Background of RISC

- When the telephone project was terminated in 1975, the machine itself had not been built, but the design had progressed to the point where it seemed to be an excellent basis for a general-purpose, high-performance miniprocessor. The attractiveness of the processor design stemmed from projections that it would be able to compute at high speed relative to its cost in a variety of application areas.
- The most important features of the telephone switching machine which contributed to its low cost/performance ratio were 1) separate instruction and data caches, allowing a much higher bandwidth between memory and CPU; 2) no arithmetic operations to storage, which greatly simplified the pipeline; and 3) uniform instruction length and simplicity of design, making possible a very short cycle time: ten levels of logic. (For example, all register-to-register operations executed in one cycle.)

Background of RISC

- John Cocke and his colleagues developed simpler ISAs and compilers for minicomputers. As an experiment, they retargeted their research compilers to use only the simple register-register operations and load-store data transfers of the IBM 360 ISA, avoiding the more complicated instructions. They found that programs ran up to three times faster using the simple subset.

- Emer and Clark found 20% of the VAX instructions needed 60% of the microcode and represented only 0.2% of the execution time.

RISC: *Reduced Instruction Set Computer*

A type of microprocessor architecture that utilizes a small, highly-optimized set of instructions

History: The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy.

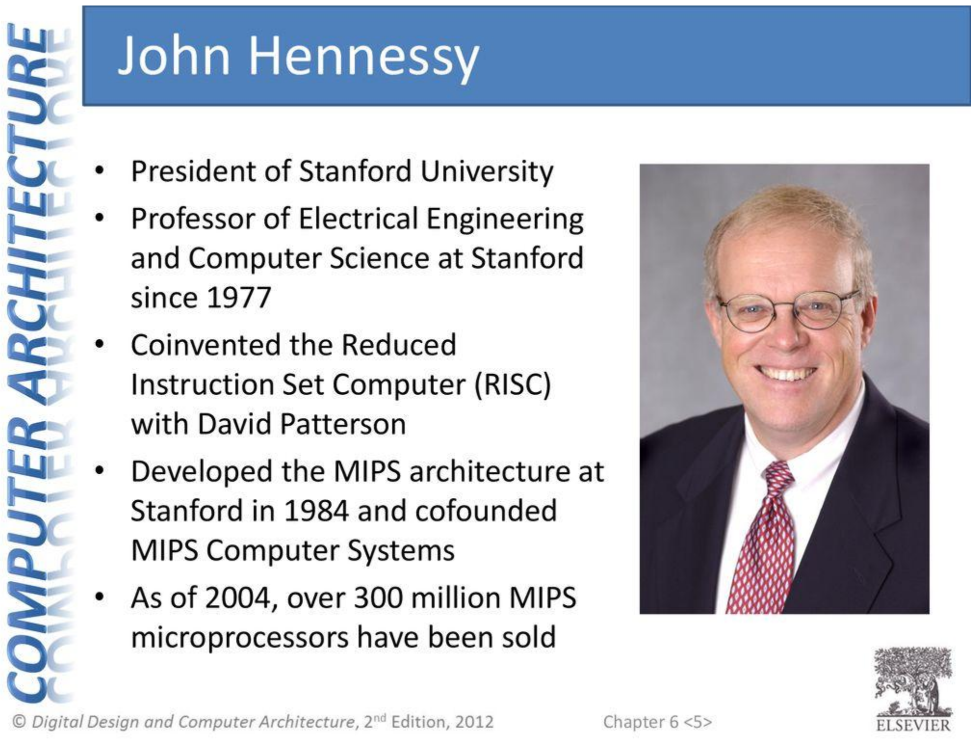
Design features of RISC processors:

- *one cycle execution time*: RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called pipelining;
- *pipelining*: a technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
- *large number of registers*: RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

MIPS

The MIPS processor was developed as part of a VLSI research program at Stanford University in the early 80s.

Professor John Hennessy started the development of MIPS with a brainstorming class for graduate students. The readings and idea sessions helped launch the development of the processor which became one of the first RISC processors, with IBM and Berkeley developing processors at around the same time.



COMPUTER ARCHITECTURE

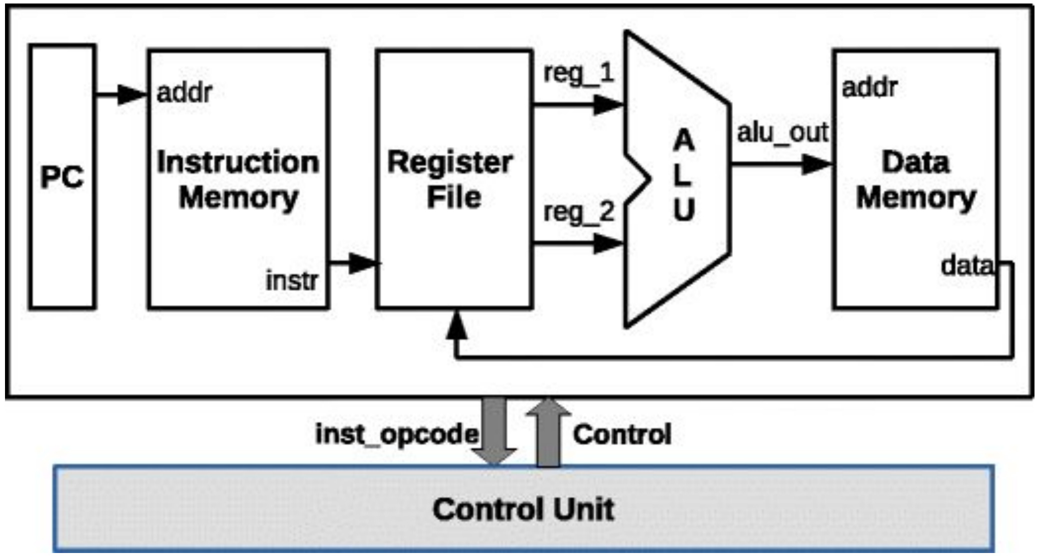
John Hennessy

- President of Stanford University
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Coinvented the Reduced Instruction Set Computer (RISC) with David Patterson
- Developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems
- As of 2004, over 300 million MIPS microprocessors have been sold

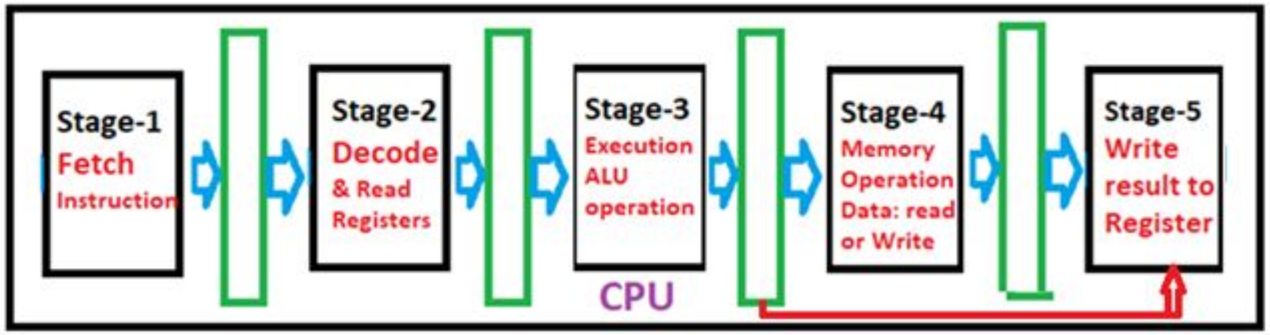
© *Digital Design and Computer Architecture*, 2nd Edition, 2012 Chapter 6 <5> ELSEVIER

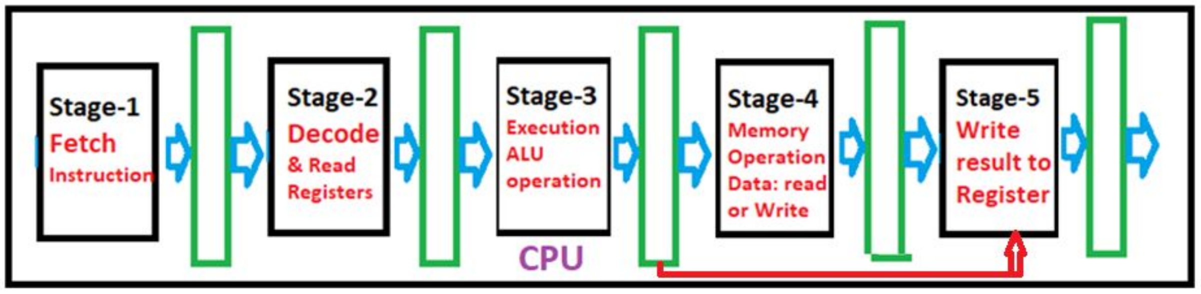
MIPS Architecture

- The Stanford research group had a strong background in compilers, which led them 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, which had been a long running design philosophy in the hardware industry.
- Thus, the MIPS processor implemented a smaller, simpler instruction set. Each of the instructions included in the chip design ran in a single clock cycle. The processor used a technique called pipelining to more efficiently process instructions.
- MIPS used 32 registers, each 32 bits wide (a bit pattern of this size is referred to as a *word*).



- Instruction cycle of MIPS processor was subdivided into **five stages**:
- **Instruction Fetch (IF)**
- **Instruction Decode (ID) and Register Read**
- **Execution (EXE)**
- **Memory read/write(MEM)**
- **Write Back** result **(WB)** to Registers





How Pipelining Works

Instruction\ clock cycles	1	2	3	4	5	6	7	8
Instruction-1	IF	ID	EX	MA	WB			
Instruction-2		IF	ID	EX	MA	WB		
Instruction-3			IF	ID	EX	MA	WB	
Instruction-4				IF	ID	EX	MA	
Instruction-5					IF	ID	EX	
Instruction-6						IF	ID	
Instruction-7							IF	

Pipelining, a standard feature in RISC processors, is much like an assembly line. Because the processor works on different steps of the instruction at the same time, more instructions can be executed in a shorter period of time.

CPI = 1

Non-pipelined processor (CPI = 5)

Instruction/Clock cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instruction-1	IF	ID	EX	MEM	WB										
Instruction-2						IF	ID	EX	MEM	WB					
Instruction-3											IF	ID	EX	MEM	WB

Common RISC Simplifications

- **Fixed instruction length:**
  Simplifies fetching instructions from memory
- **Simplified addressing modes:**
  Simplifies fetching operands from memory
- **Few and simple instructions in the instruction set:**
  Simplifies instruction execution
- **Minimize memory access instructions (load/store):**
  Simplifies necessary hardware for memory access
- **Let compiler do heavy lifting:**
  Breaks complex statements into multiple assembly instructions

MIPS

(Microprocessor without Interlocked Pipeline Stages)

- MIPS, an acronym for Microprocessor **without Interlocked Pipeline Stages**, was a research project conducted by John L. Hennessy at Stanford University between 1981 and 1984.
- MIPS is a reduced instruction set computer (RISC) instruction set *architecture* (ISA)
- In 1985, MIPS Computer Systems (now MIPS Technologies, based in the United States) announced a new instruction set architecture (ISA) also called MIPS, and its first implementation, the R2000 microprocessor.
- The **R2000** is a 32-bit microprocessor chip set that implemented the MIPS I instruction set architecture (ISA).
- A fixed-length, regularly encoded instruction set and uses a load/store data model –Used by NEC, Cisco, Silicon Graphics, Sony, Nintendo.
- **R2000** was used in the DEC station 2100 and DEC station 3100 as well as several others.

The instruction set consists of about 111 total instructions.
A variety of basic instructions, including:

21 arithmetic instructions (+, -, *, /, %)

8 logic instructions (&, |, ~)

8 bit manipulation instructions

12 comparison instructions (>, <, =, >=, <=, ¬)

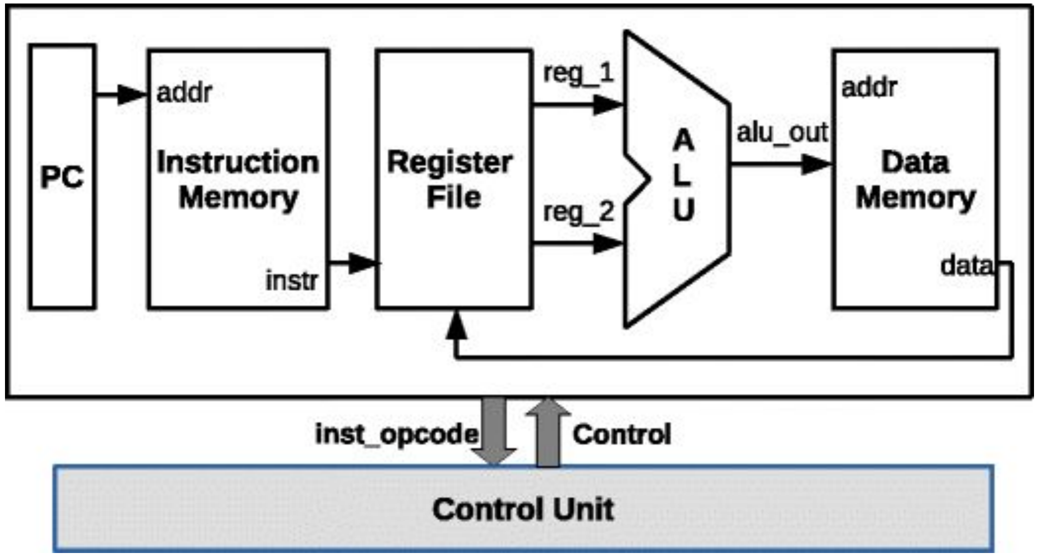
25 branch/jump instructions

15 load instructions

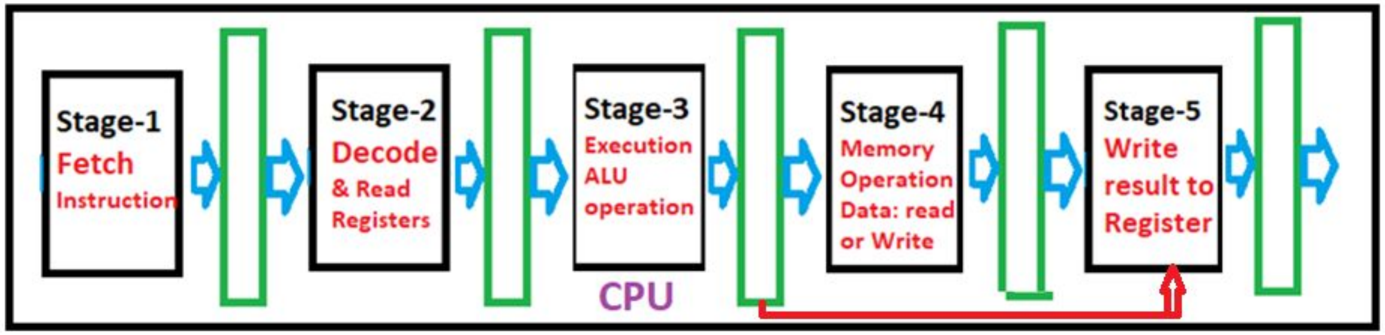
10 store instructions

8 move instructions

4 miscellaneous instructions



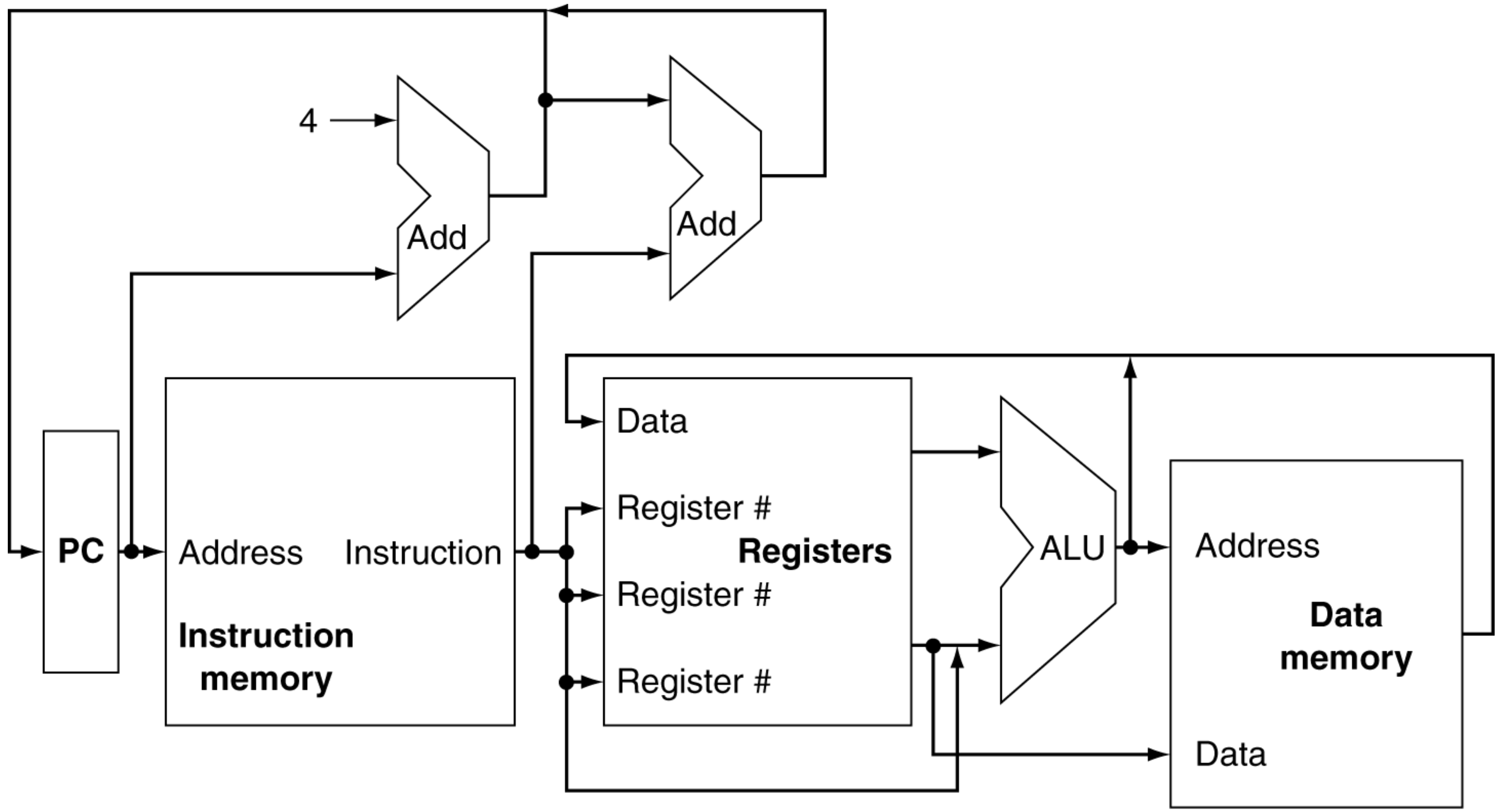
- Instruction cycle of MIPS processor was subdivided into **five stages**:
- **Instruction Fetch (IF)**
- **Instruction Decode (ID) and Register Read**
- **Execution (EXE)**
- **Memory read/write(MEM)**
- **Write Back** result **(WB)** to Registers

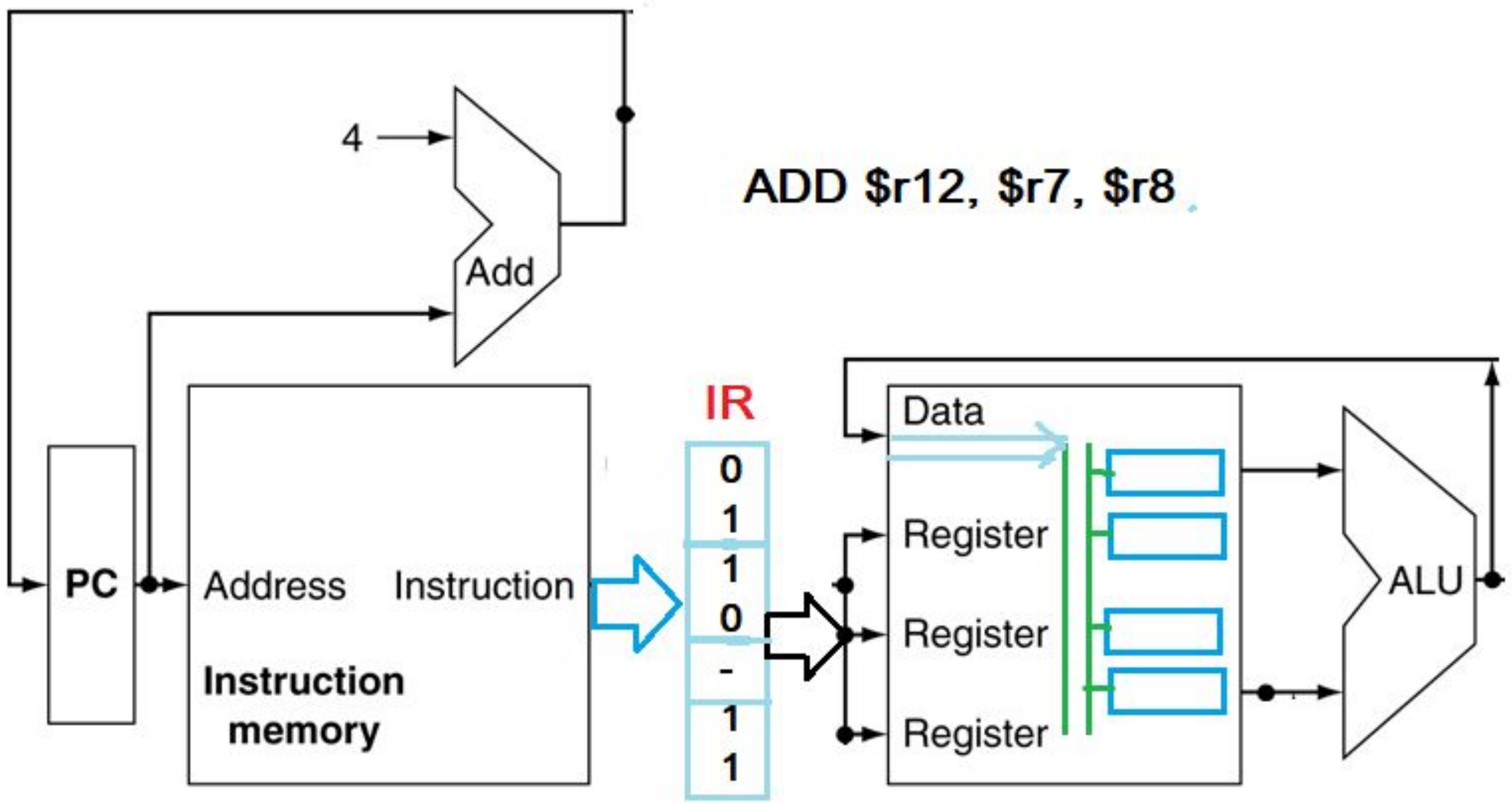


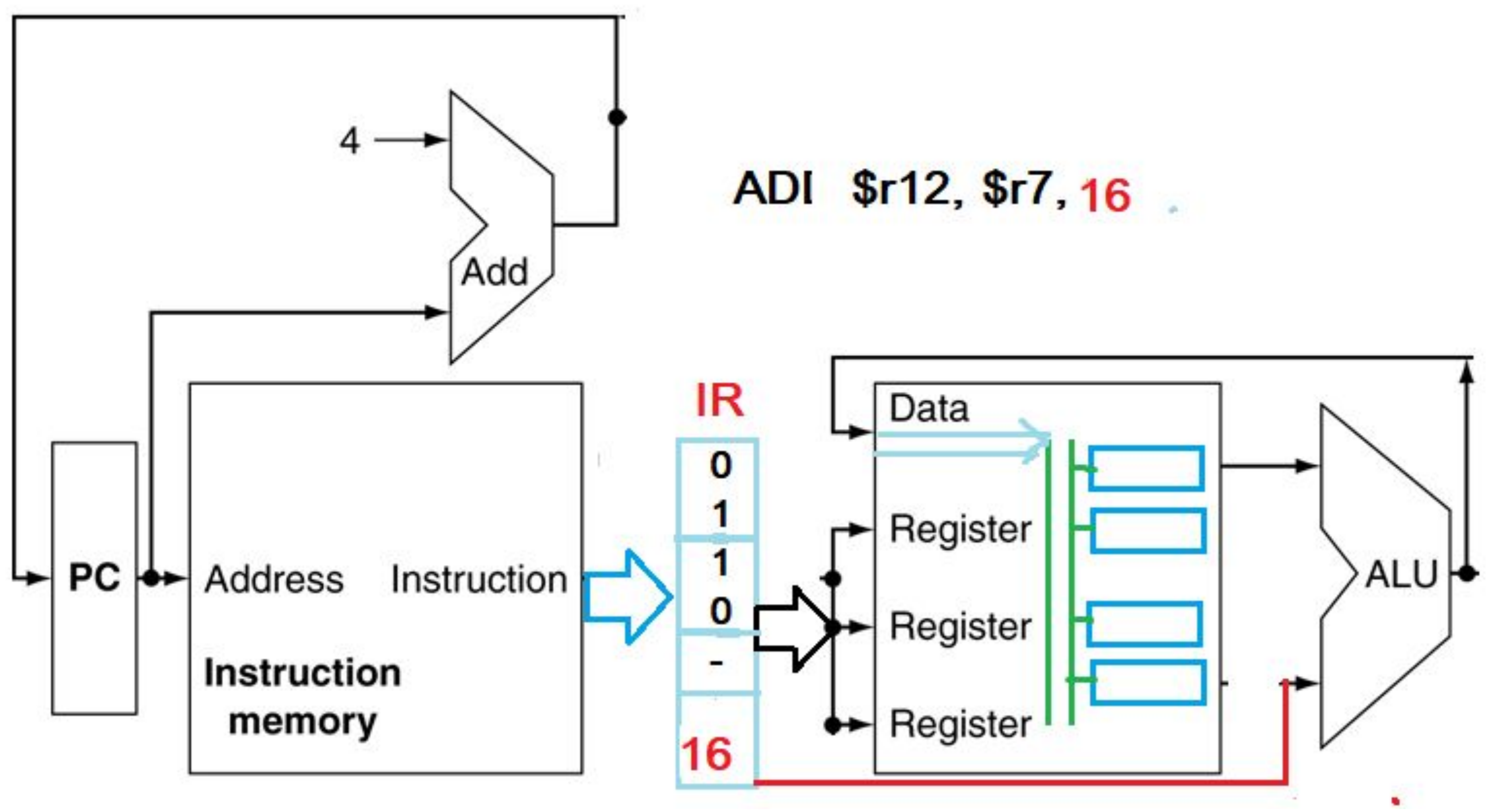
Instruction Execution

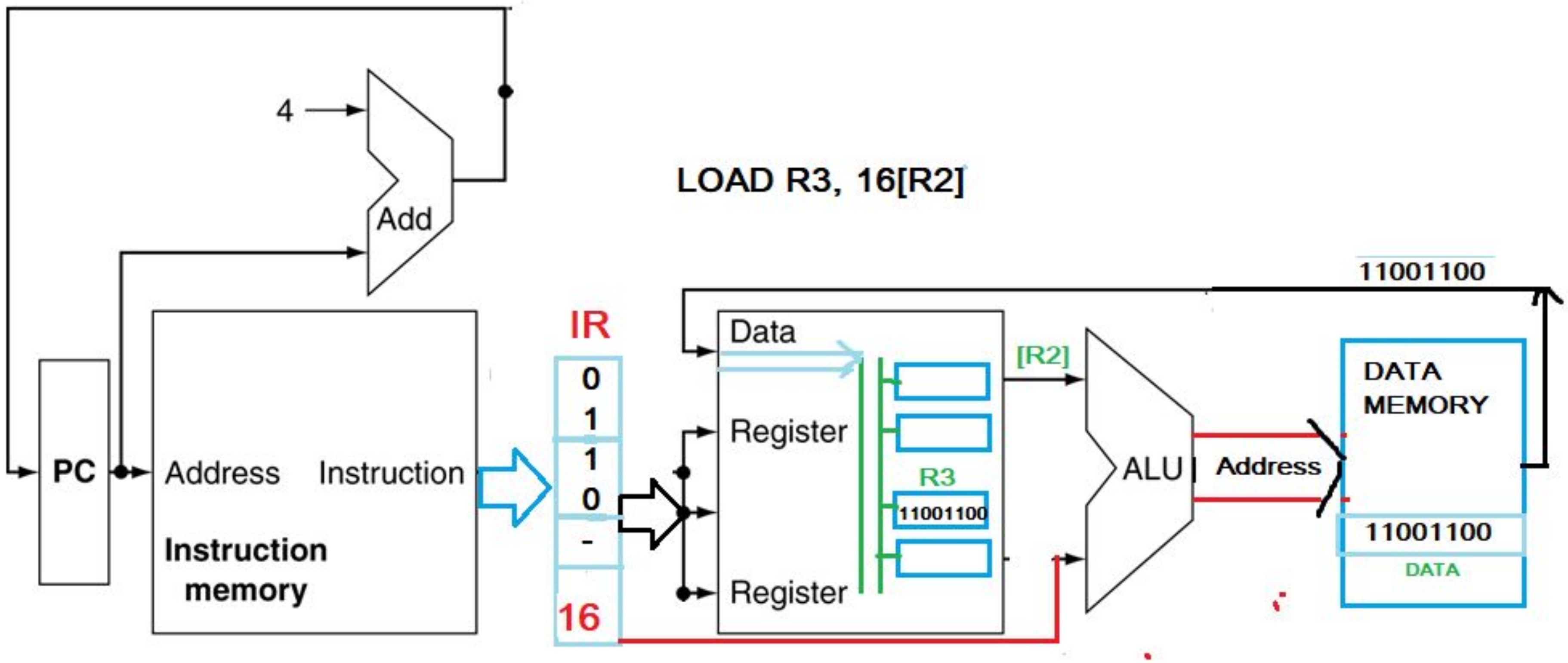
- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC $\leftarrow$ target address or PC + 4

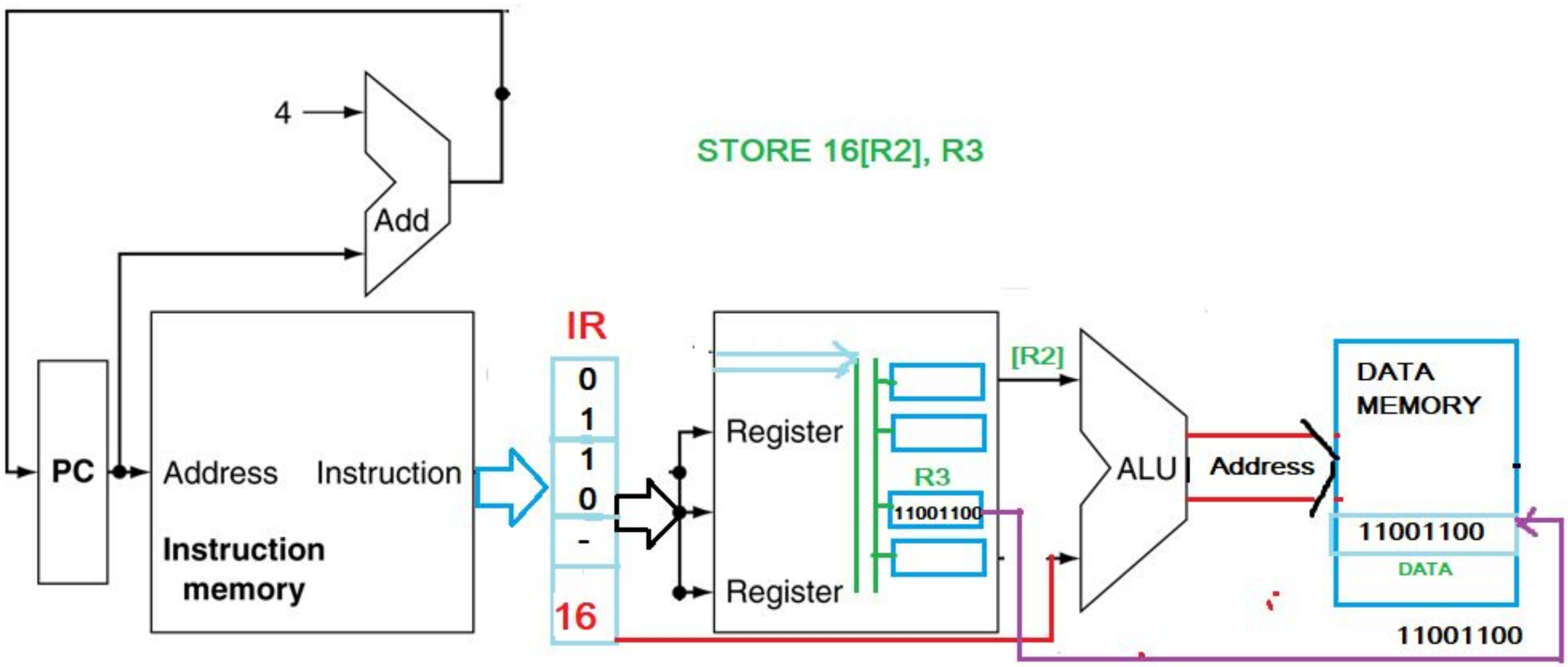
CPU Overview



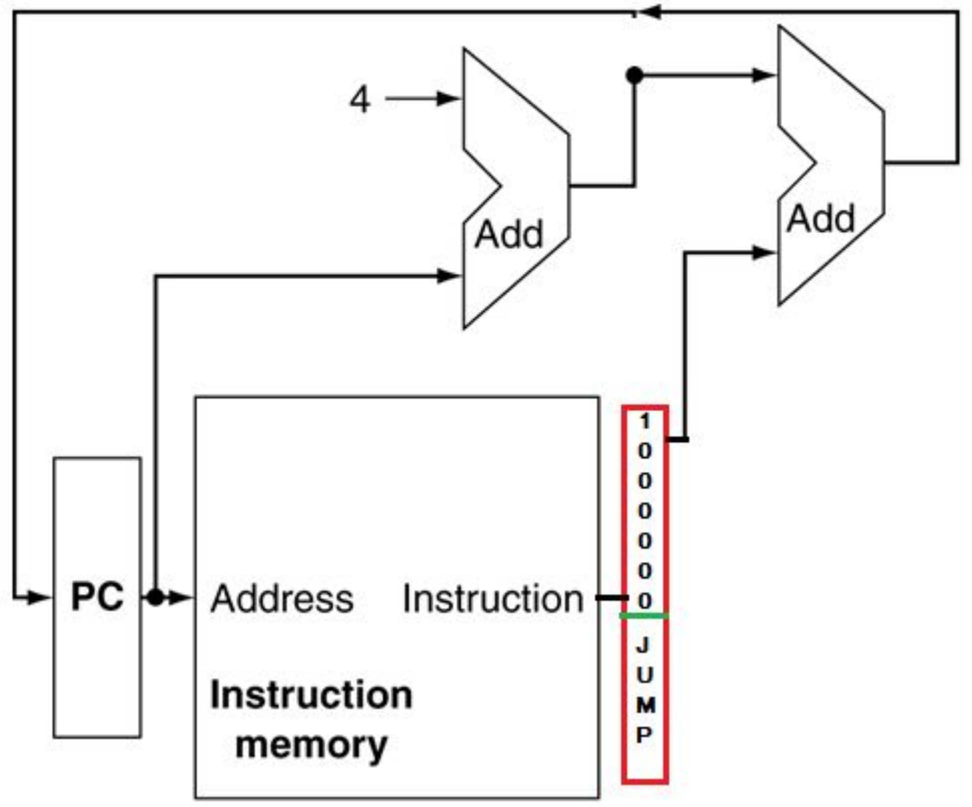

4
Add
ADD $r12, $r7, $r8
PC
Address
Instruction
Instruction memory
IR
0
1
1
0
-
1
1
Data
Register
Register
Register
ALU


4
Add
ADI $r12, $r7, 16
PC
Address
Instruction
Instruction memory
IR
0
1
1
0
-
16
Data
Register
Register
Register
ALU

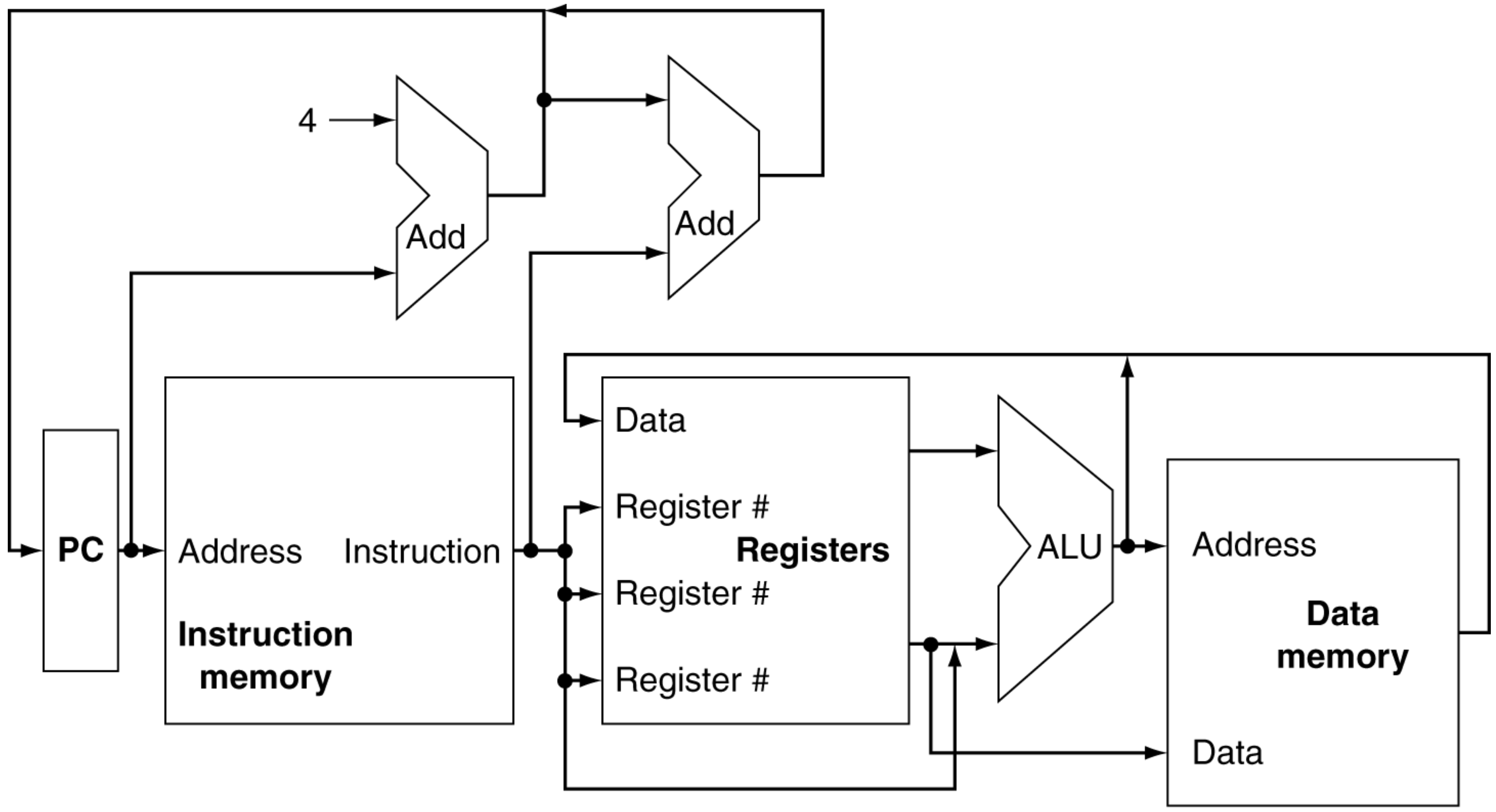

LOAD R3, 16[R2]
4
Add
PC
Address
Instruction
Instruction memory
IR
0
1
1
0
-
16
Data
Register
Register
R3
11001100
[R2]
ALU
Address
DATA MEMORY
11001100
DATA
11001100


4
Add
STORE 16[R2], R3
IR
0
1
1
0
-
16
PC
Address
Instruction
Instruction memory
Register
Register
R3
11001100
[R2]
ALU
Address
DATA MEMORY
11001100
DATA
11001100

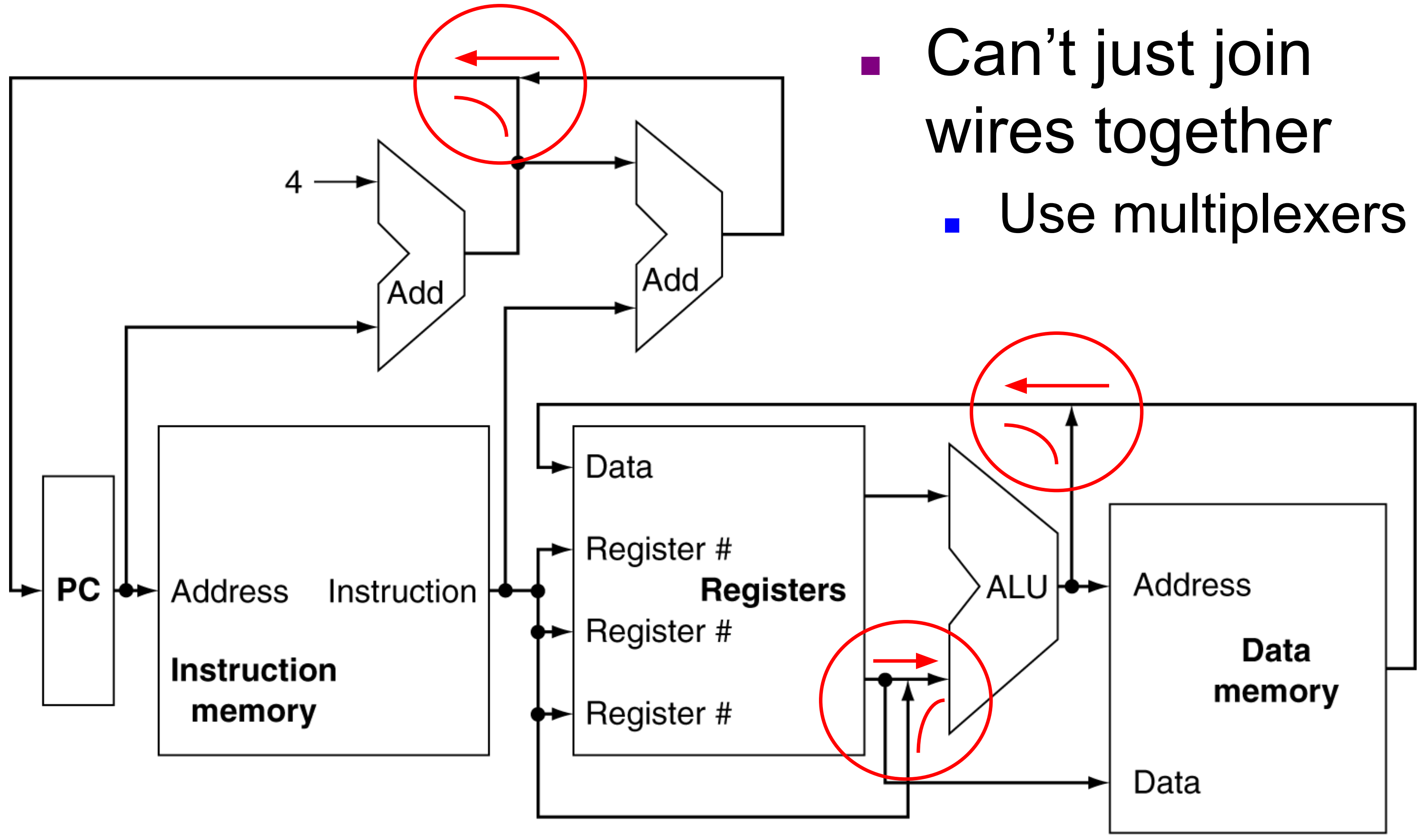
JUMP 64


4
Add
Add
PC
Address
Instruction
Instruction memory
1
0
0
0
0
0
0
J
U
M
P

CPU Overview

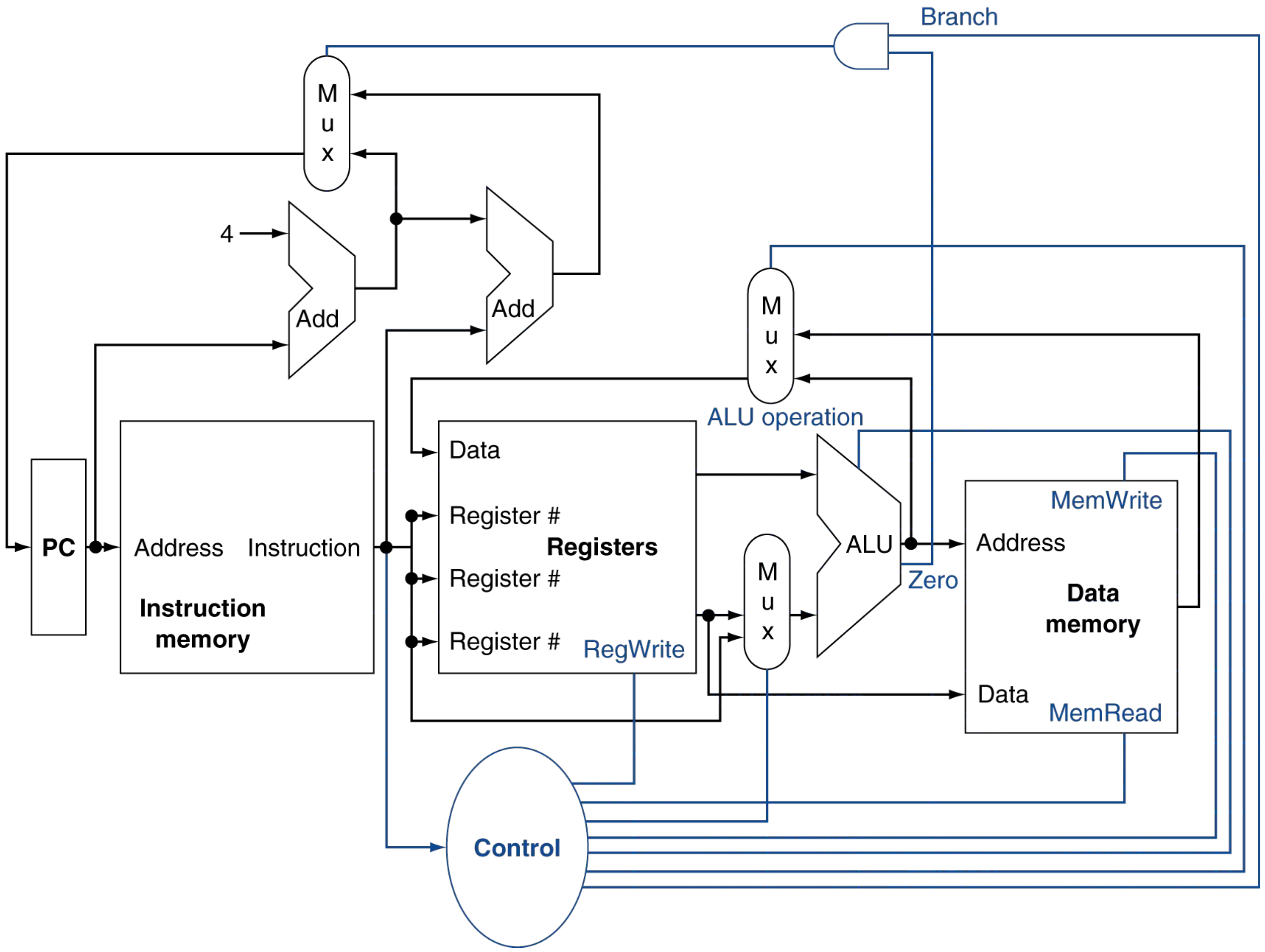


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control


Branch
M u x
4
Add
Add
M u x
ALU operation
PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
RegWrite
M u x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
Control

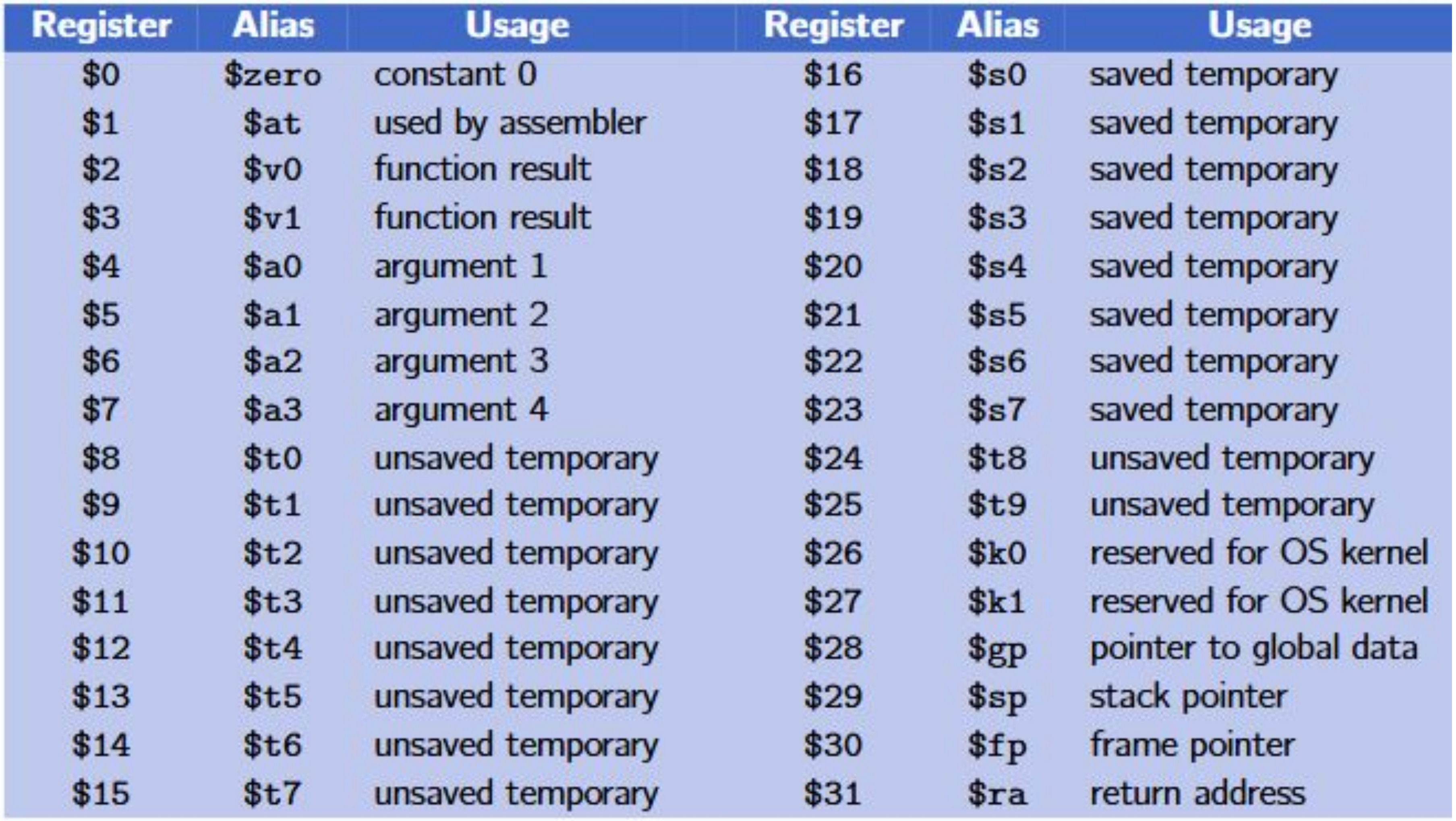
Computer Hardware Operands

- In high-level languages, number of variables limited only by available memory
- ISAs have a fixed, small number of operands called registers
  - Special locations built directly into hardware
  - **Benefit:** Registers are EXTREMELY FAST (faster than 1 billionth of a second)
  - **Drawback:** Operations can only be performed on these predetermined number of registers

MIPS Registers

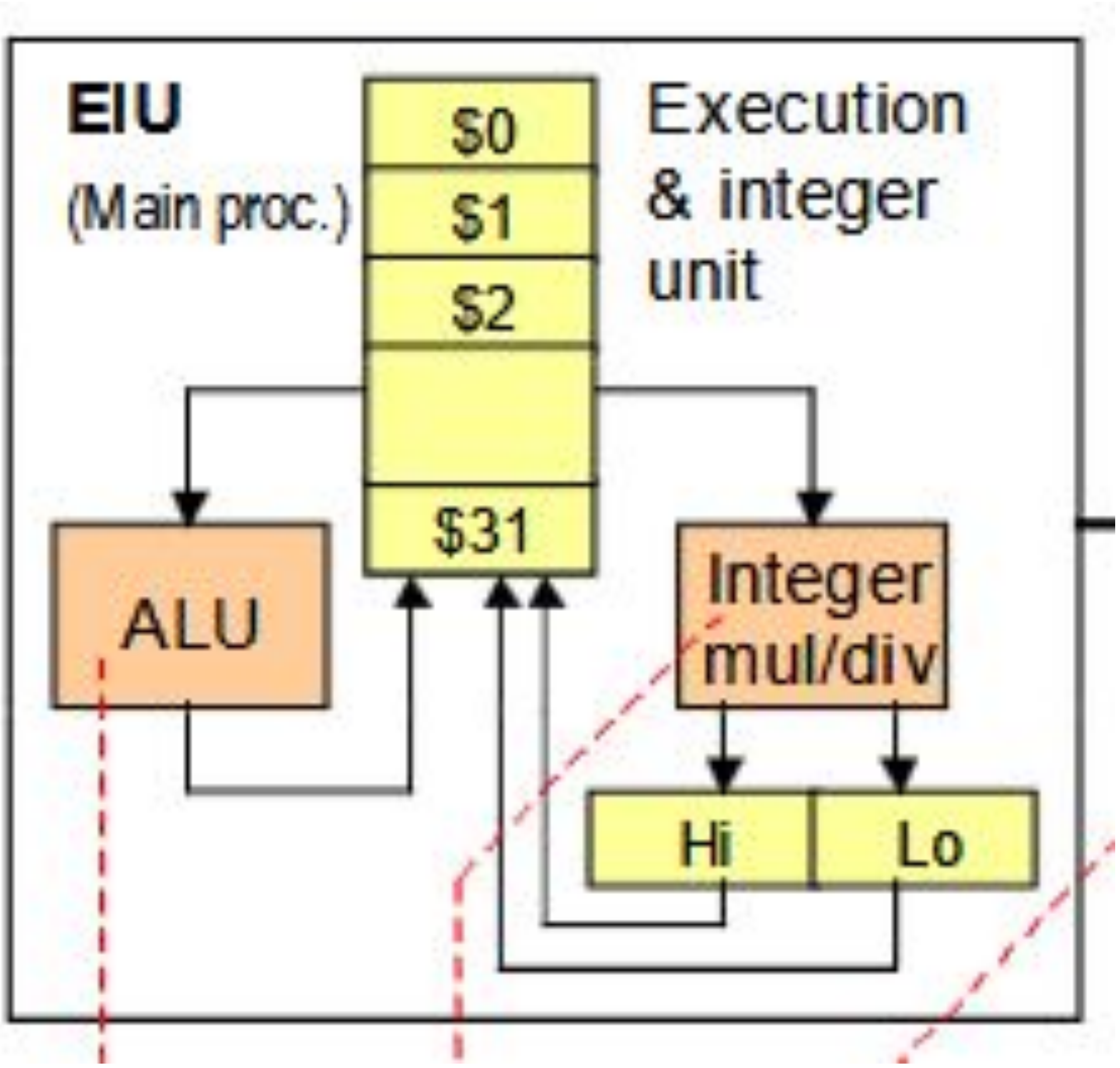
- MIPS has 32 registers
  - Each register is 32 bits wide and holds a word
- Tradeoff between speed and availability
  - Smaller number means faster hardware but insufficient to hold data for typical C programs
- *Registers have no type* (C concept); the operation being performed determines how register contents are treated

MIPS Registers



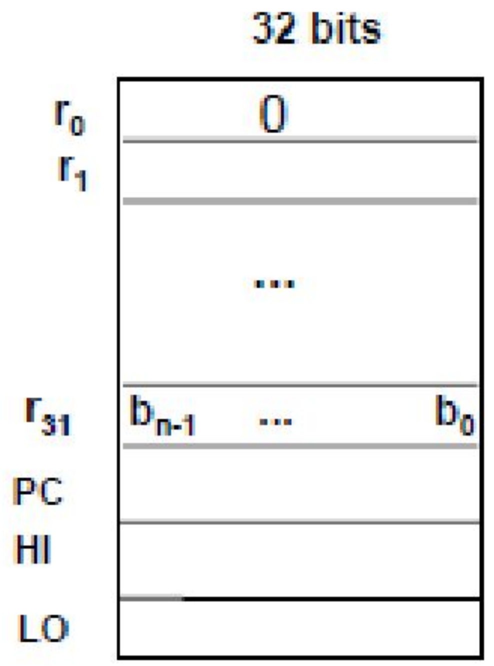
Register	Alias	Usage	Register	Alias	Usage
$0	$zero	constant 0	$16	$s0	saved temporary
$1	$at	used by assembler	$17	$s1	saved temporary
$2	$v0	function result	$18	$s2	saved temporary
$3	$v1	function result	$19	$s3	saved temporary
$4	$a0	argument 1	$20	$s4	saved temporary
$5	$a1	argument 2	$21	$s5	saved temporary
$6	$a2	argument 3	$22	$s6	saved temporary
$7	$a3	argument 4	$23	$s7	saved temporary
$8	$t0	unsaved temporary	$24	$t8	unsaved temporary
$9	$t1	unsaved temporary	$25	$t9	unsaved temporary
$10	$t2	unsaved temporary	$26	$k0	reserved for OS kernel
$11	$t3	unsaved temporary	$27	$k1	reserved for OS kernel
$12	$t4	unsaved temporary	$28	$gp	pointer to global data
$13	$t5	unsaved temporary	$29	$sp	stack pointer
$14	$t6	unsaved temporary	$30	$fp	frame pointer
$15	$t7	unsaved temporary	$31	$ra	return address

MIPS


EIU
(Main proc.)
$0
$1
$2
$31
Execution & integer unit
ALU
Integer mul/div
Hi
Lo

MIPS Registers (R2000/R3000)

- 32x32-bit GPRs (General purpose registers)
  - \$0 = \$zero (therefore only 31 GPRs)
  - \$1 = \$at (reserved for assembler)
  - \$2 - \$3 = \$v0 - \$v1 (return values)
  - \$4 - \$7 = \$a0 - \$a3 (arguments)
  - \$8 - \$15 = \$t0 - \$t7 (temporaries)
  - \$16 - \$23 = \$s0 - \$s7 (saved)
  - \$24 - \$25 = \$t8 - \$t9 (more temporaries)
  - \$26 - \$27 = \$k0 - \$k1 (reserved for OS)
  - \$28 = \$gp (global pointer)
  - \$29 = \$sp (stack pointer)
  - \$30 = \$fp (frame pointer)
  - \$31 = \$ra (return address)



- 32x32-bit floating point registers (paired double precision)
- HI, LO, PC
- Status, Cause, BadVAddr, EPC

MIPS Registers

- Register denoted by ‘`$`’ can be referenced by number (`$0-$31`) or name:
  - Registers that hold programmer variables:

    `$s0-$s7` ⟷ `$16-$23`
  - Registers that hold temporary variables:

    `$t0-$t7` ⟷ `$8-$15`

    `$t8-$t9` ⟷ `$24-$25`
  - You’ll learn about the other 14 registers later
- In general, using register names makes code more readable

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, $t1, …, $t7 for temporary values
  - $s0, $s1, …, $s7 for saved variables

- In MIPS assembly language
- $s0 - $s7 maps onto register 16 to 23
- $t0 - $t7   maps onto register 8 to 15

- $s0   means register 16
- $s1   means register 17
- …….
- $t0   means register 8
- $t1   means register 9
- ……..

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
    - Big here means the largest address, the least significant byte at the largest address. Byte address 0 1 2 3. The MSB at address 0 the LSB at address 3

- Little Endian:
  - least-significant byte at least address

    Little here means the least address. The least significant byte at the least address. Byte Address 0 1 2 3. MSB at address 3, the LSB at address 0

MIPS Instructions

- One operation per instruction, at most one instruction per line
- Assembly instructions are related to C operations (`=`, `+`, `-`, `*`, `/`, `&`, `|`, etc.)
  - Must be, since C code decomposes into assembly!
  - A single line of C may break up into several lines of MIPS

MIPS Instructions

Instruction Syntax is rigid:

```
        add  $s1, $s2, $s3
    op dst, src1, src2
```

1 operator, 3 operands

- `op` = operation name ("operator")
- `dst` = register getting result ("destination")
- `src1` = first register for operation ("source 1")
- `src2` = second register for operation ("source 2")

MIPS Instructions

- Instruction Syntax :

  `op dst, src1, src2`

  - `op` = operation name ("operator")
  - `dst` = register getting result ("destination")
  - `src1` = first register for operation ("source 1")
  - `src2` = second register for operation ("source 2")

- Integer Addition (`add`)

  C: `a = b + c`

  MIPS: **`add $s1, $s2, $s3`**

- Integer Subtraction (`sub`)

  C: `a = b - c`

  MIPS: **`sub $s1, $s2, $s3`**

MIPS Instructions Example

- Suppose `a=$s0`,
  `b=$s1`,
  `c=$s2`,
  `d=$s3`, and
  `e=$s4`.
- Convert the following C statement to MIPS:

```
a = (b + c) - (d + e);
```

```
add $t1, $s3, $s4
add $t2, $s1, $s2
sub $s0, $t2, $t1
```

$\downarrow$ Ordering of instructions matters (must follow order of operations)

Comments in MIPS

- Comments in MIPS follow hash mark (#) until the end of line
  - Improves readability and helps you keep track of variables/registers!

```
add $t1, $s3, $s4 # $t1=d+e
add $t2, $s1, $s2 # $t2=b+c
sub $s0, $t2, $t1 # a=(b+c)-(d+e)
```

The Zero Register

- Zero appears so often in code and is so useful that it has its own register!
- Register zero (`$0` or `$zero`) always has the value 0 and cannot be changed!
  - i.e. any instruction with `$0` as `dst` has no effect
- Example Uses:
  - `add   $s3,   $0,   $0   # c=0`
  - `add   $s1, $s2,   $0   # a=b`

Immediates

- Numerical constants are called immediates
- Separate instruction syntax for immediates:

```
opi dst, src, imm
```

Operation names end with ‘`i`’, replace 2nd source register with an immediate

Example Uses:

```
addi $s1, $s2, 5   # a=b+5
addi $s3, $s3, 1   # c++
```

Data Transfer

Store: register to memory

Load: register from memory

- Load Word

Reads data FROM memory and places it into `reg`

- Store Word

Saves contents of a `reg` and stores it to memory

Data Transfer

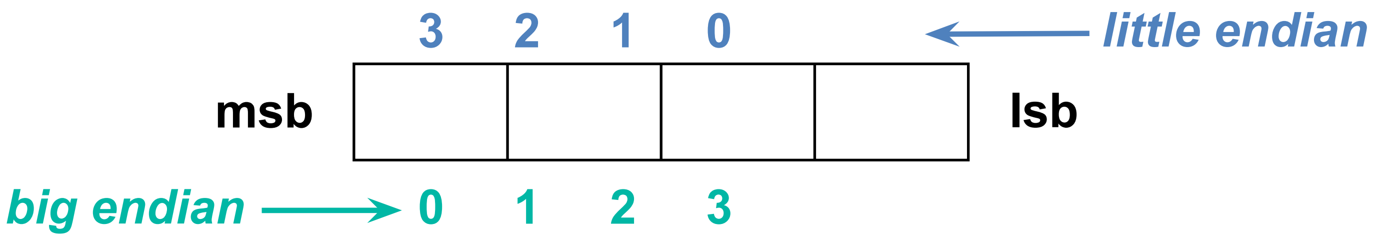
- Instruction syntax for data transfer between
  Register and Memory

  `op reg, off(bAddr)`

  - `op` = operation name ("operator")
  - `reg` = register for operation source or destination
  - `bAddr` = register with pointer to memory ("base address")
  - `off` = address offset (immediate) in bytes ("offset")
- Accesses memory at address `bAddr+off`
- Load Word: `lw  $t0,12($s3)`
  - Takes data at address `bAddr+off` FROM memory and places it into `reg`
- Store Word: `sw  $t0,40($s3)`
  - Takes data in `reg` and stores it TO memory at address `bAddr+off`

Endianness

- Big Endian: Most-significant byte at least address of word
  - word address = address of most significant byte
- Little Endian: Least-significant byte at least address of word
  - word address = address of least significant byte



- MIPS is bi-endian (can go either way)
  - Using MARS simulator in lab, which is little endian
- Why is this confusing?
  - Data stored in reverse order than you write it out!
  - Data `0x01020304` stored as `04 03 02 01` in memory

Increasing address

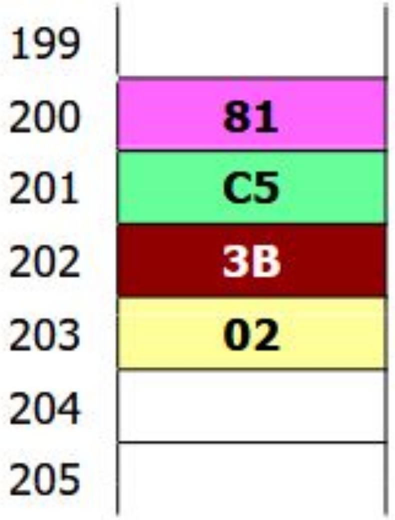
Data Transfer Instructions

- Load Word (`lw`)
  - Takes data at address `bAddr+off` FROM memory and places it into `reg`
- Store Word (`sw`)
  - Takes data in `reg` and stores it TO memory at address `bAddr+off`
- Example Usage:

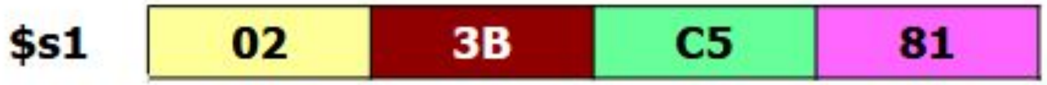
```
# addr of int A[] -> $s3, a -> $s0
lw  $t0,12($s3) # $t0=A[3]
add $t0,$s2,$t0 # $t0=A[3]+a
sw  $t0,40($s3) # A[10]=A[3]+a
```

Endianness and Load

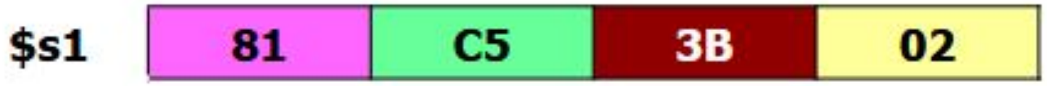
- **Load $s1 from M[200]**
  - `lw $s1,200($zero)`



- **Little endian**

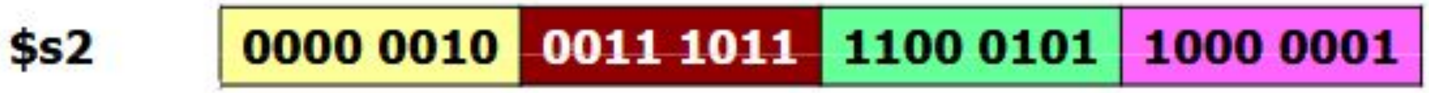


- **Big endian**

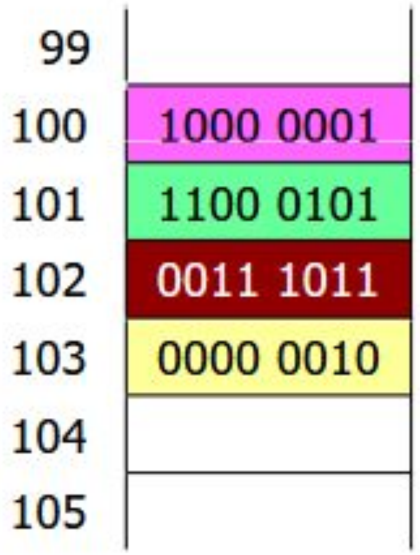


Endianness and Store

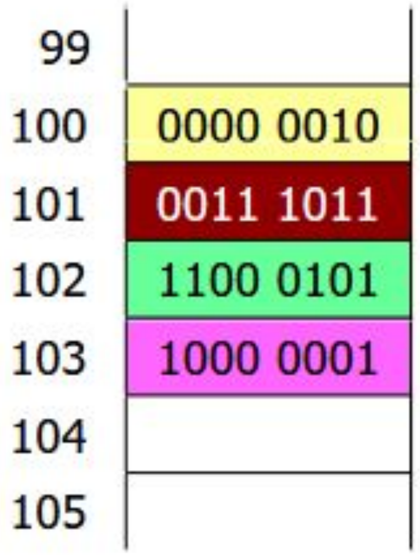
- **Store $s2 into M[100]**
  - `sw $s2,100($zero)`



- **Little endian**



- **Big endian**



Decision Making Instructions

- Branch If Equal (`beq`)
  - `beq reg1,reg2,label`
  - If value in `reg1` = value in `reg2`, go to `label`
- Branch If Not Equal (`bne`)
  - `bne reg1,reg2,label`
  - If value in `reg1` ≠ value in `reg2`, go to `label`
- Jump (`j`)
  - `j label`
  - Unconditional jump to `label`

branch on equal	`beq $1,$2,100`	if($1==$2) go to PC+4+100	Test if registers are equal
branch on not equal	`bne $1,$2,100`	if($1!=$2) go to PC+4+100	Test if registers are not equal
branch on greater than	`bgt $1,$2,100`	if($1>$2) go to PC+4+100	*Pseduo-instruction*
branch on greater than or equal	`bge $1,$2,100`	if($1>=$2) go to PC+4+100	*Pseduo-instruction*
branch on less than	`blt $1,$2,100`	if($1<$2) go to PC+4+100	*Pseduo-instruction*
branch on less than or equal	`ble $1,$2,100`	if($1<=$2) go to PC+4+100	*Pseduo-instruction*

```
beq reg1,reg2,label
j label
```

```
Loop: lb   $t0,0($s0)
      sb   $t0,0($s1)
      addi $s0,$s0,1
      addi $s1,$s1,1
      beq  $t0,$0,Exit
      j Loop
Exit:
```

Summary

- Computers understand the *instructions* of their *ISA*
- RISC Design Principles
  - Smaller is faster, keep it simple
- MIPS Registers: `$s0-$s7`, `$t0-$t9`, `$0`
- MIPS Instructions
  - Arithmetic: `add`, `sub`, `addi`
  - Data Transfer: `lw`, `sw`, `lb`, `sb`, `lbu`
  - Branching: `beq`, `bne`, `j`
- Memory is byte-addressed

C to MIPS Practice

- Let’s put our all of our new MIPS knowledge to use in an example: “Fast String Copy”
- C code is as follows:

```
/* Copy string from p to q */
char *p, *q;
while((*q++ = *p++) != ‘\0’) ;
```

- What do we know about its structure?
  - Single `while` loop
  - Exit condition is an equality test

C to MIPS Practice

- Start with code skeleton:

```
# copy String p to q
# p□$s0, q□$s1 (pointers)
Loop:                  # $t0 = *p
                       # *q = $t0
                       # p = p + 1
                       # q = q + 1
                       # if *p==0, go to Exit
      j Loop           # go to Loop
Exit:
```

C to MIPS Practice

- Fill in lines:

```
# copy String p to q
# p$s0, q$s1 (pointers)
Loop: lb   $t0,0($s0)   # $t0 = *p
      sb   $t0,0($s1)   # *q = $t0
      addi $s0,$s0,1      # p = p + 1
      addi $s1,$s1,1    # q = q + 1
      beq  $t0,$0,Exit # if *p==0, go to Exit
      j Loop            # go to Loop
Exit:
```

C to MIPS Practice

- Finished code:

```
# copy String p to q
# p$s0, q$s1 (pointers)
Loop: lb   $t0,0($s0)   # $t0 = *p
      sb   $t0,0($s1)   # *q = $t0
      addi $s0,$s0,1      # p = p + 1
      addi $s1,$s1,1    # q = q + 1
      beq  $t0,$0,Exit # if *p==0, go to Exit
      j Loop            # go to Loop
Exit: # N chars in p => N*6 instructions
```

C to MIPS Practice

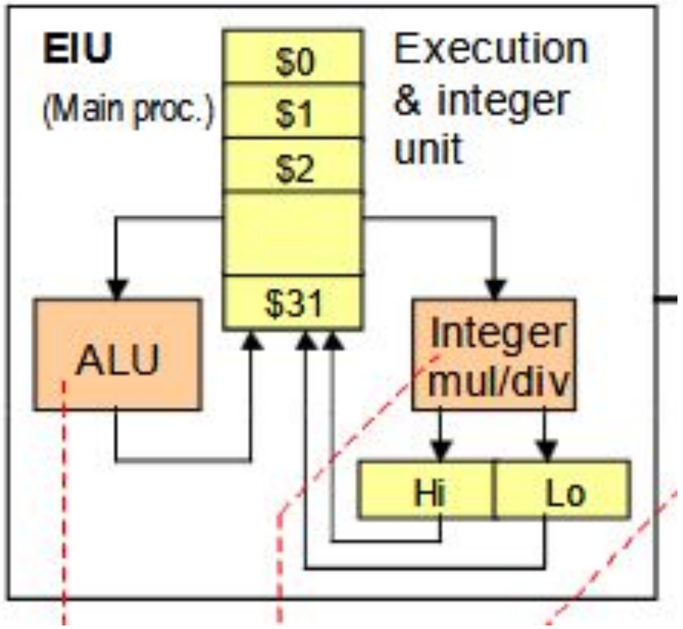
- Alternate code using bne:

```
# copy String p to q
# p□$s0, q□$s1 (pointers)
Loop: lb   $t0,0($s0)   # $t0 = *p
      sb   $t0,0($s1)   # *q = $t0
      addi $s0,$s0,1      # p = p + 1
      addi $s1,$s1,1    # q = q + 1
      bne  $t0,$0,Loop # if *p!=0, go to Loop
# N chars in p => N*5 instructions
```

Multiply	`mult $2,$3`	\$hi,\$low=\$2*\$3	Upper 32 bits stored in special register `hi` Lower 32 bits stored in special register `lo`
Divide	`div $2,$3`	\$hi,\$low=\$2/\$3	Remainder stored in special register `hi` Quotient stored in special register`lo`

MIPS Arithmetic Instructions: Multiplication

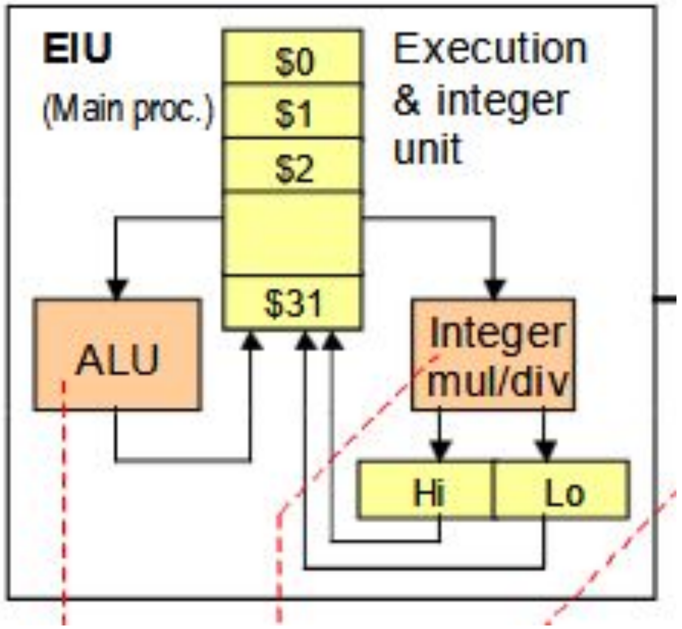
```
mult   $s0, $s1

mfhi   $t0
mflo   $t1
```



```
#  Multiply the numbers stored in these registers.
#  This yields a 64 bit number, which is stored in two
#  32 bits parts:  "hi" and "lo"
# loads the upper 32 bits from the product register
# loads the lower 32 bits from the product register
```

MIPS Arithmetic Instructions: Division

```
div   $s0, $s1
mfhi  $t0
mflo  $t1
```



```
#  Hi contains the remainder,  Lo contains quotient
#  remainder moved into $t0
#  quotient moved into $t1
```

MIPS Bitwise Instructions

Note: a□$s1, b□$s2, c□$s3

Instruction	C	MIPS
And	`a = b & c;`	`and  $s1,$s2,$s3`
And Immediate	`a = b & 0x1;`	`andi $s1,$s2,0x1`
Or	`a = b | c;`	`or   $s1,$s2,$s3`
Or Immediate	`a = b | 0x5;`	`ori  $s1,$s2,0x5`
Not Or	`a = ~(b | c);`	`nor  $s1,$s2,$s3`
Exclusive Or	`a = b ^ c;`	`xor  $s1,$s2,$s3`
Exclusive Or Immediate	`a = b ^ 0xF;`	`xori $s1,$s2,0xF`

Shifting Instructions

- In binary, shifting an unsigned number left is the same as multiplying by the corresponding power of 2
  - Shifting operations are faster
  - Does not work with shifting right/division
- *Logical shift*: Add zeros as you shift
- *Arithmetic shift*: Sign-extend as you shift
  - Only applies when you shift right (preserves sign)
- Can shift by immediate or value in a register

Shifting Instructions

Instruction Name	MIPS
Shift Left Logical	`sll  $s1,$s2,1`
Shift Left Logical Variable	`sllv $s1,$s2,$s3`
Shift Right Logical	`srl  $s1,$s2,2`
Shift Right Logical Variable	`srlv $s1,$s2,$s3`
Shift Right Arithmetic	`sra  $s1,$s2,3`
Shift Right Arithmetic Variable	`srav $s1,$s2,$s3`

- When using immediate, only values 0-31 are accepted
- When using variable, only lowest 5 bits are used (read as unsigned)

Machine Code

Instructions in Binary bits

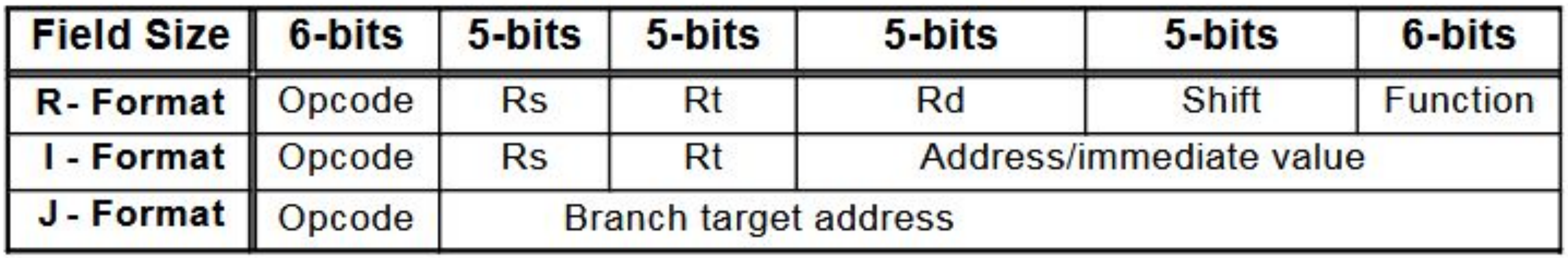
Instructions in Numbers

32 bits in binary

8-digits in hexcode

0x12A6012C

Table 14.1 MIPS 32-bit Instruction Formats.

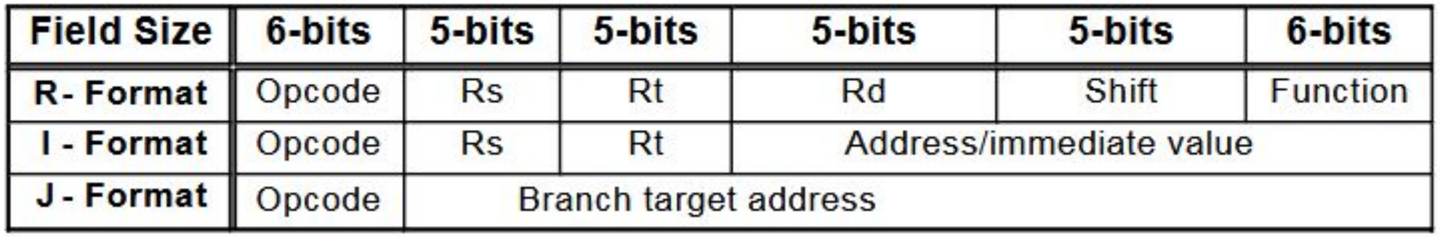


Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
R- Format	Opcode	Rs	Rt	Rd	Shift	Function
I - Format	Opcode	Rs	Rt	Address/immediate value		
J - Format	Opcode	Branch target address				

Instruction Formats

- I-Format: instructions with immediates, `lw`/`sw` (offset is immediate), and `beq`/`bne`
- J-Format: `j` and `jal`
- R-Format: all other instructions

Table 14.1 MIPS 32-bit Instruction Formats.



Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
R- Format	Opcode	Rs	Rt	Rd	Shift	Function
I - Format	Opcode	Rs	Rt	Address/immediate value		
J - Format	Opcode	Branch target address				

Instruction fields

op: operation code (opcode)

Rs: first source register number

Rt: second source register number

Rd: destination register number

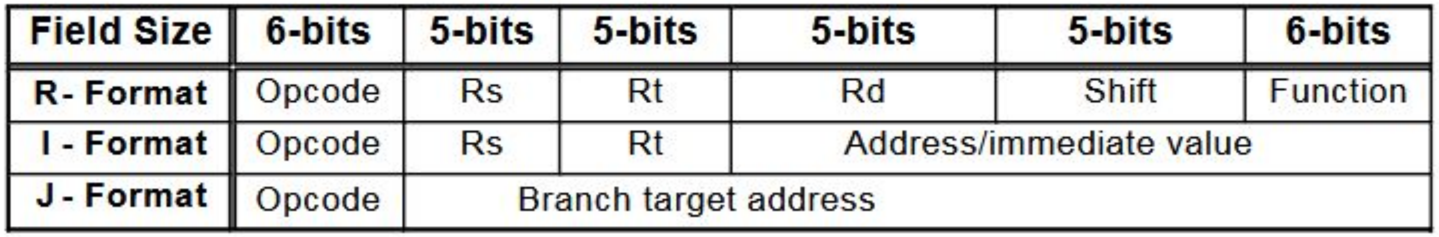
Shift: shift amount (00000 for now)

Function: function code (extends opcode)

Instructions as Numbers

- Divide the 32 bits of an instruction into "fields"
  - Each field tells the processor something about the instruction
  - Could use different fields for every instruction, but regularity leads to simplicity
- Define 3 types of *instruction formats:*
- R-Format
- I-Format
- J-Format

Table 14.1 MIPS 32-bit Instruction Formats.

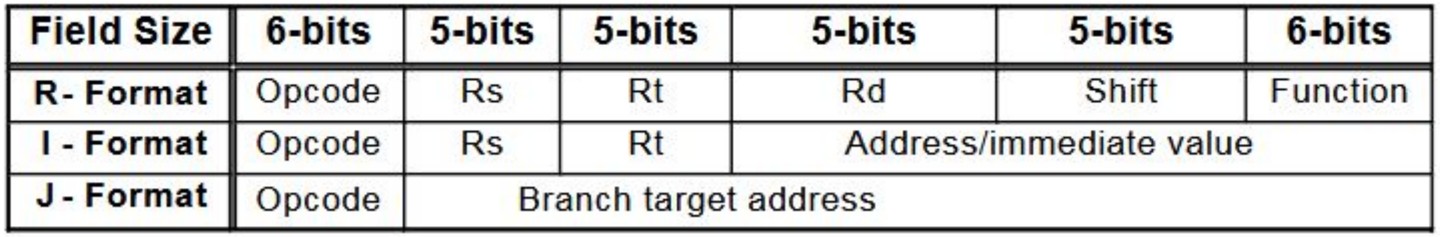


Field Size	**6-bits**	**5-bits**	**5-bits**	**5-bits**	**5-bits**	**6-bits**
R- Format	Opcode	Rs	Rt	Rd	Shift	Function
I - Format	Opcode	Rs	Rt	Address/immediate value		
J - Format	Opcode	Branch target address				

Instruction Formats

- I-Format: instructions with immediates, `lw/sw` (offset is immediate), and `beq/bne`
- J-Format: `j` and `jal`
- R-Format: all other instructions

Table 14.1 MIPS 32-bit Instruction Formats.



Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
R- Format	Opcode	Rs	Rt	Rd	Shift	Function
I - Format	Opcode	Rs	Rt	Address/immediate value		
J - Format	Opcode	Branch target address				

Instruction fields

op: operation code (opcode)

Rs: first source register number

Rt: second source register number

Rd: destination register number

Shift: shift amount (00000 for now)

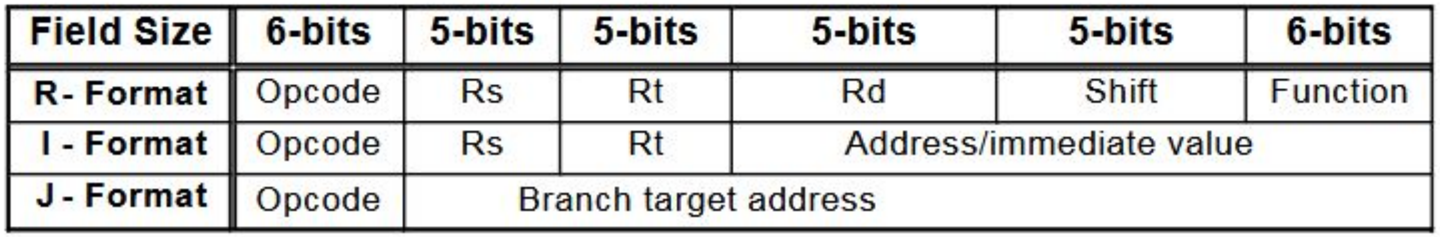
Function: function code (extends opcode)

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

Table 14.1 MIPS 32-bit Instruction Formats.



Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
R- Format	Opcode	Rs	Rt	Rd	Shift	Function
I - Format	Opcode	Rs	Rt	Address/immediate value		
J - Format	Opcode	Branch target address				

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add

0	17	18	8	0	32

000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2$ = 0x02324020

R-Format Instructions

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 5 + 5 + 6 = 32

31					0
6	5	5	5	5	6

- Each field has a name:

31					0
`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`

- Each field is viewed as its own unsigned int
  - 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63

R-Format Instructions

- `opcode` (6): partially specifies operation
  - Set at 0b000000 for all R-Format instructions
- `funct` (6): combined with `opcode`, this number exactly specifies the instruction
- `rs` (5): specifies register containing 1st operand ("source register")
- `rt` (5): specifies register containing 2nd operand ("target register" – name is misleading)
- `rd` (5): specifies register that receives the result of the computation ("destination register")
- `shamt` (5): The amount a shift instruction will shift by
  - Shifting a 32-bit word by more than 31 is useless
  - This field is set to `0` in all but the shift instructions

R-Format Example

31 0

opcode	rs	rt	rd	shamt	funct

- MIPS Instruction: `add $8,$9,$10`

Field representation (decimal):

31 0

0	9	10	8	0	32

Field representation (binary):

31 0

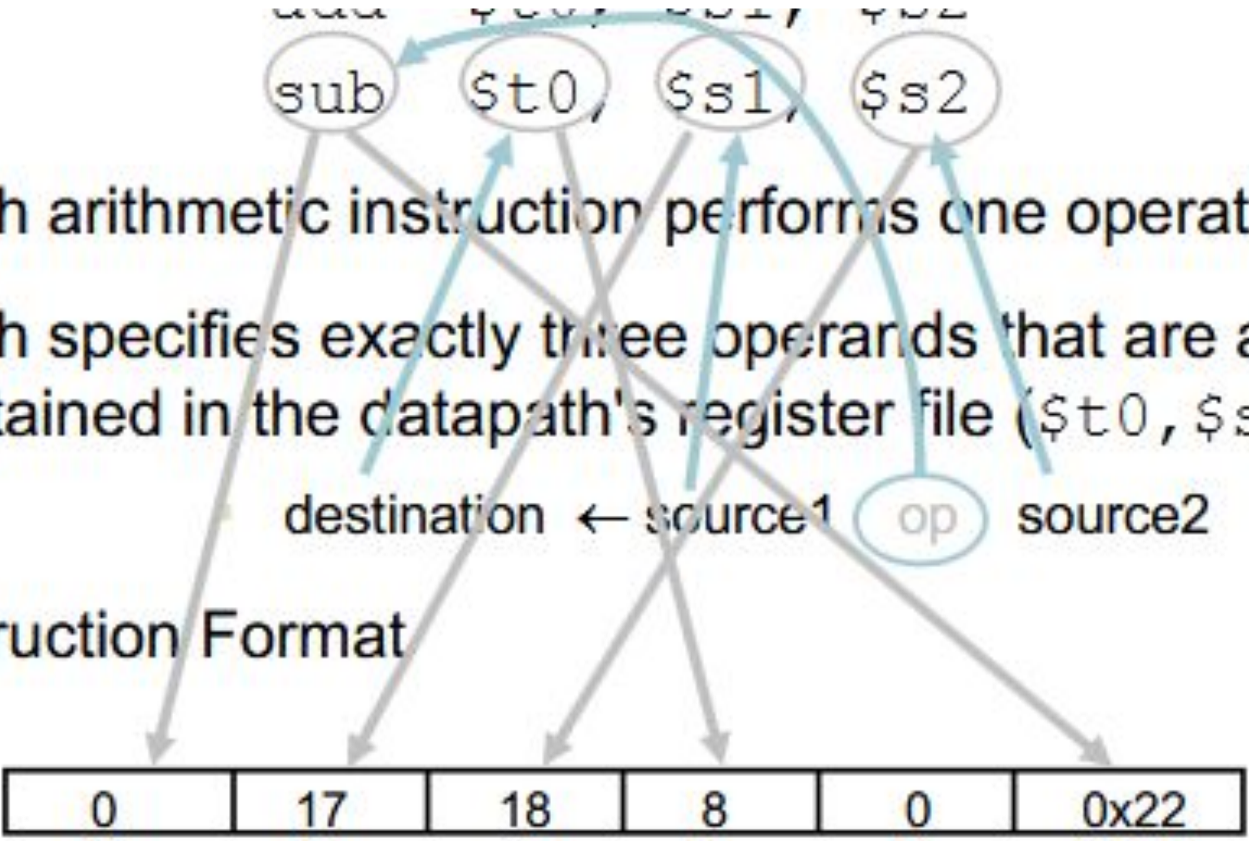
000000	01001	01010	01000	00000	100000

two

hex representation: 0x 012A 4020

decimal representation: 19,546,144

Called a Machine Language Instruction



Each arithmetic instruction performs one operation.

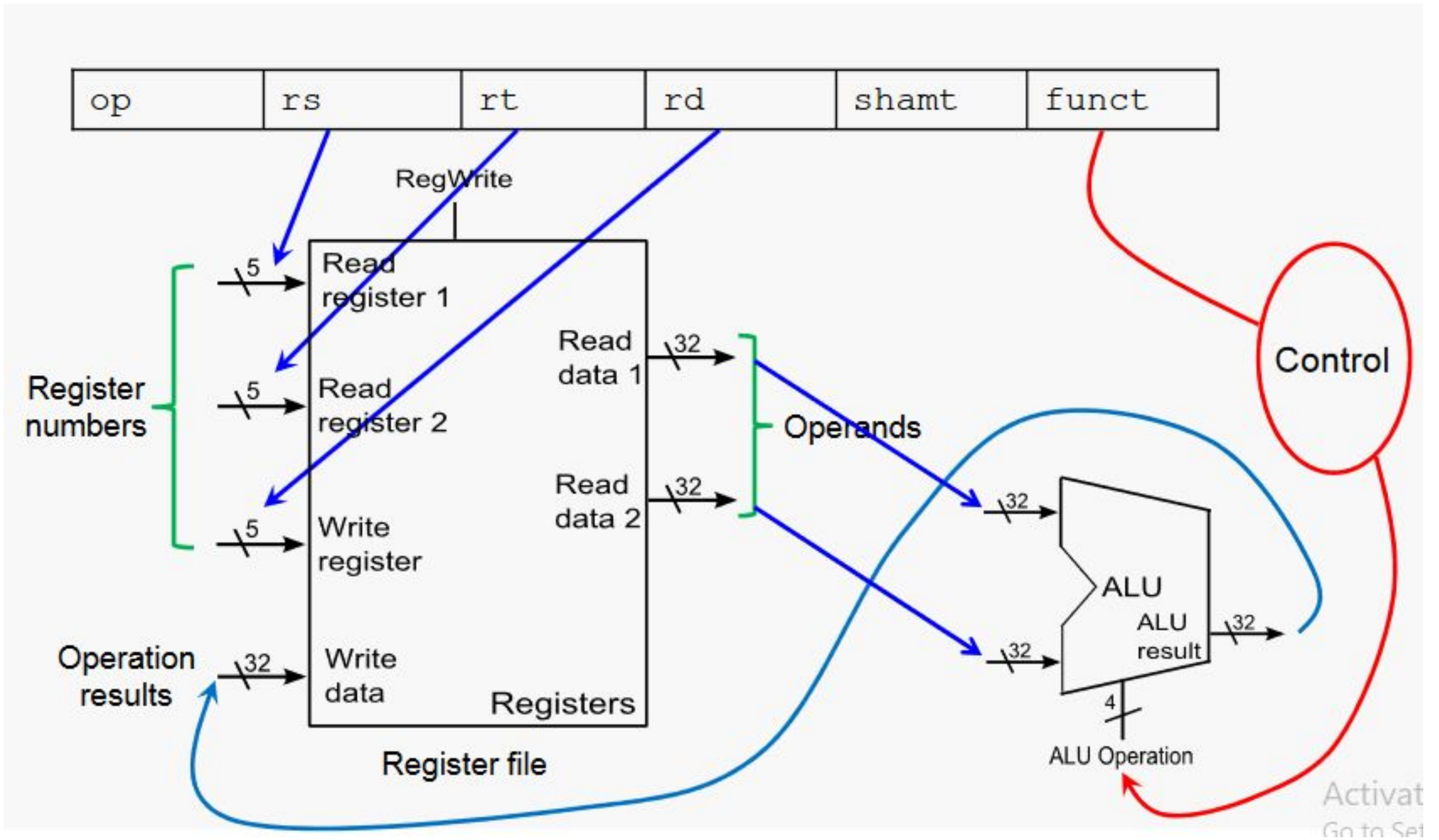
Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)

- destination ← source1 op source2

Instruction Format

0	17	18	8	0	0x22

R-format Instruction



MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic: `addi $s1,$s2,16`
- Load: `lw $t0,12($s3)`
- Store: `sw $t0,40($s3)`

rt: destination or source register number

Constant/offset: -2^{15} to $+2^{15}-1$

rs: first source register in ALU instruction

Base register in load/store instruction

Address: offset added to base address in rs

I-Format Instructions

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 16 = 32 bits

31			0
6	5	5	16

- Field names:

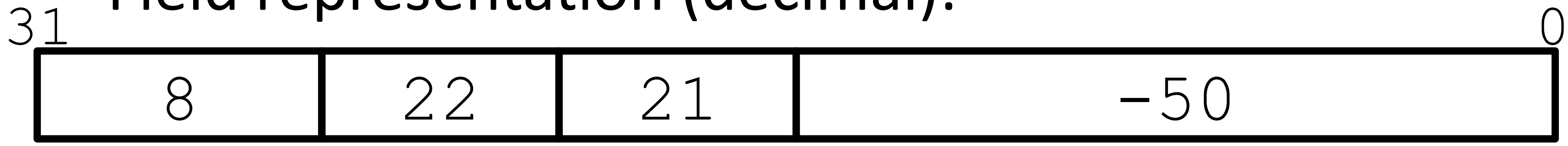
31			0
`opcode`	`rs`	`rt`	`immediate`

- **Key Concept:** Three fields are consistent with R-Format instructions
  - Most importantly, `opcode` is still in same location

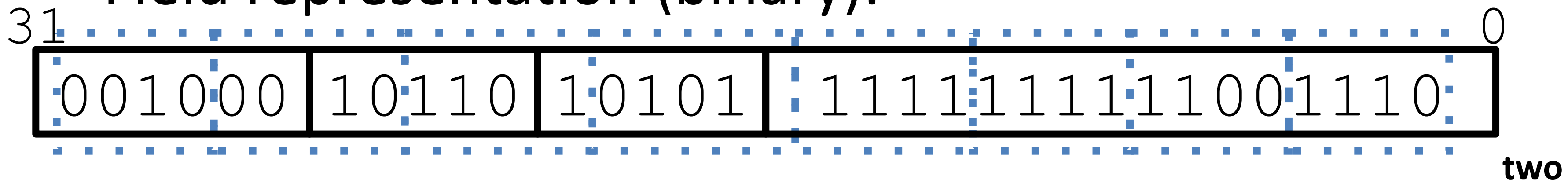
I-Format Example

- MIPS Instruction: `addi $21,$22,-50`

Field representation (decimal):



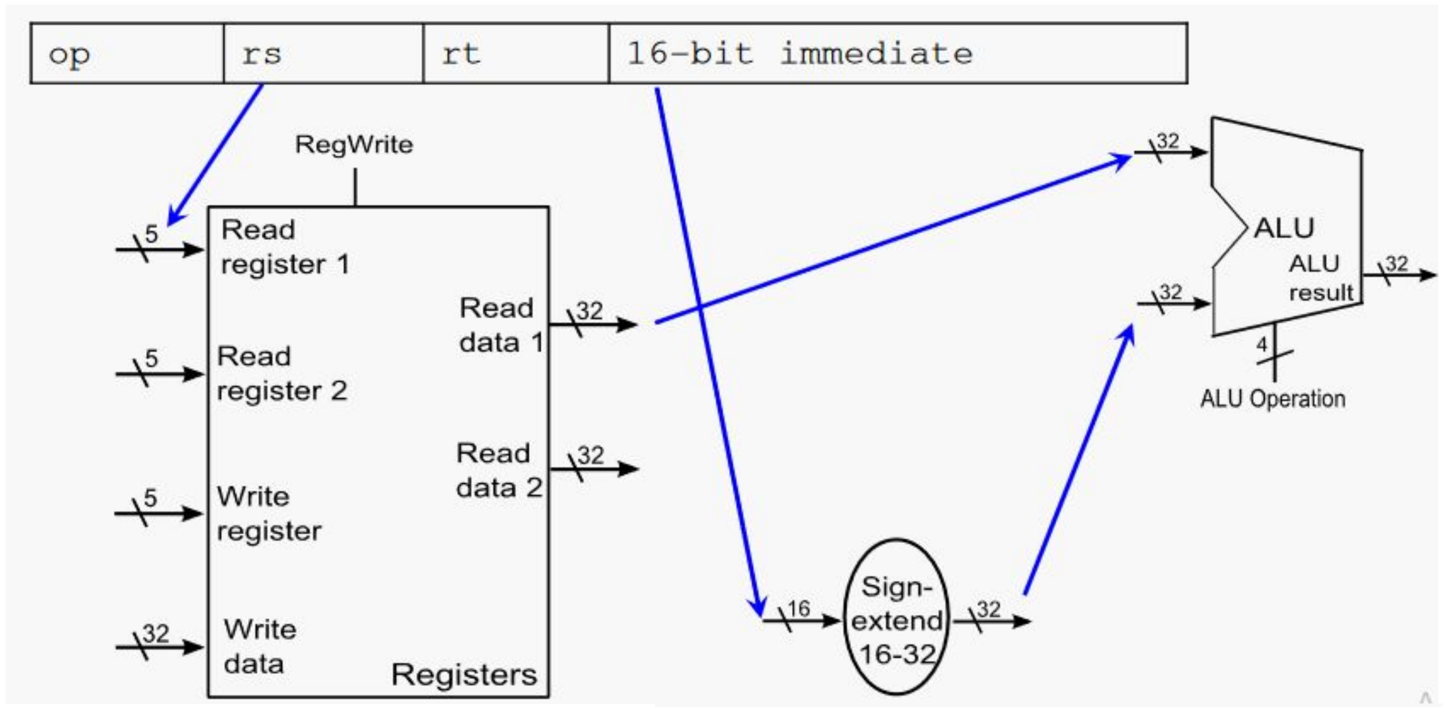
Field representation (binary):

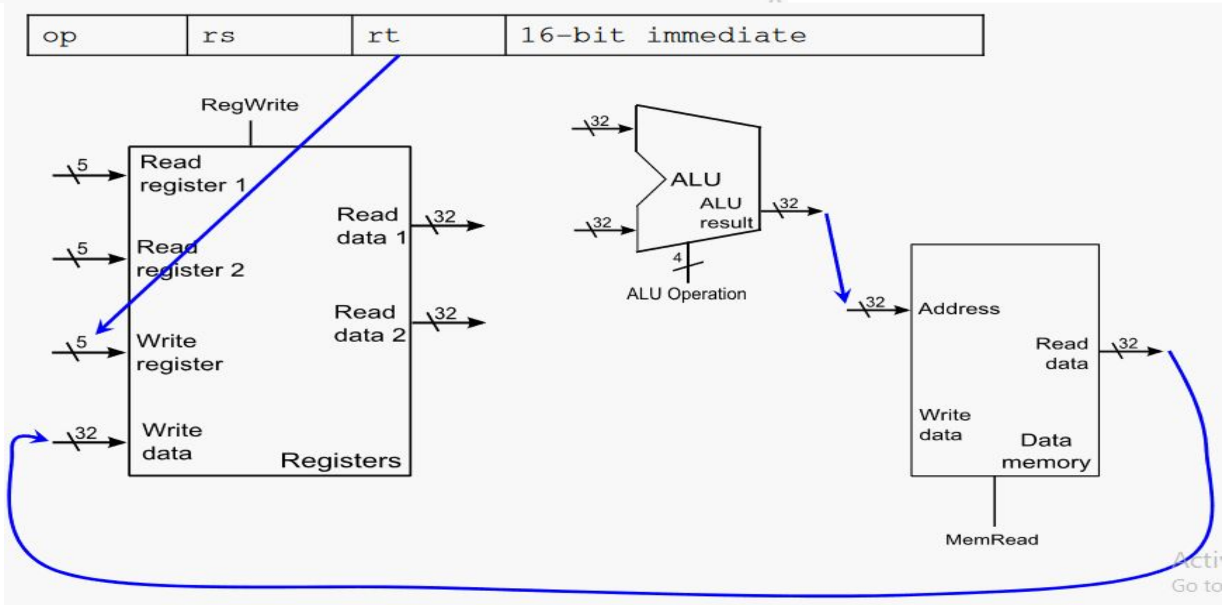


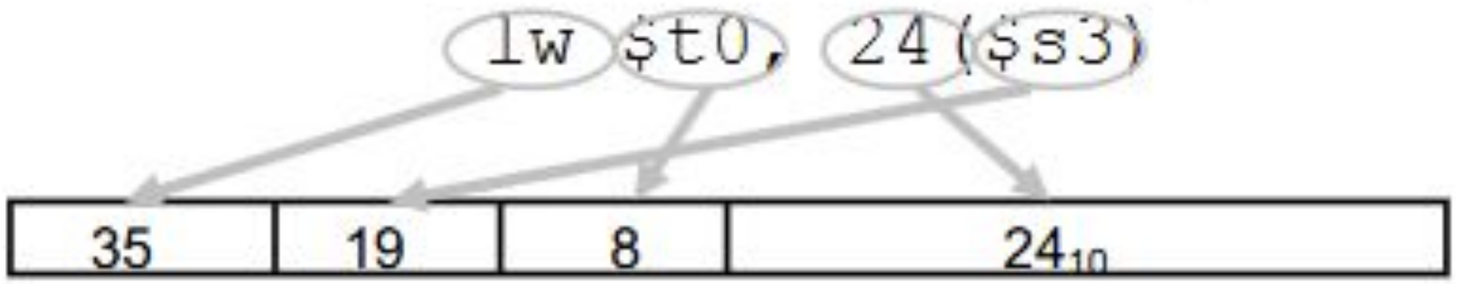
hex representation: `0x 22D5 FFCE`

decimal representation: `584,449,998`

Load Instruction `lw $t0,0($t1)`

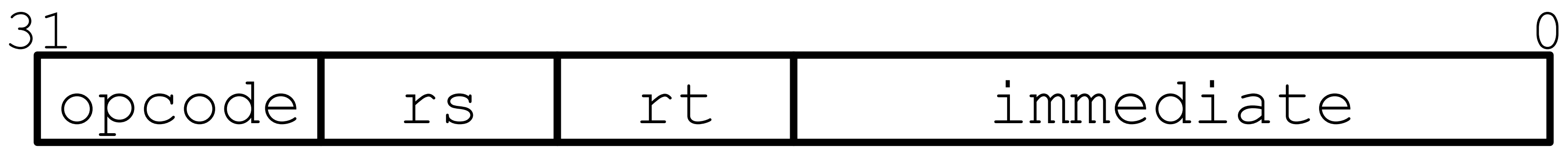





lw $t0, 24($s3)
35
19
8
24₁₀

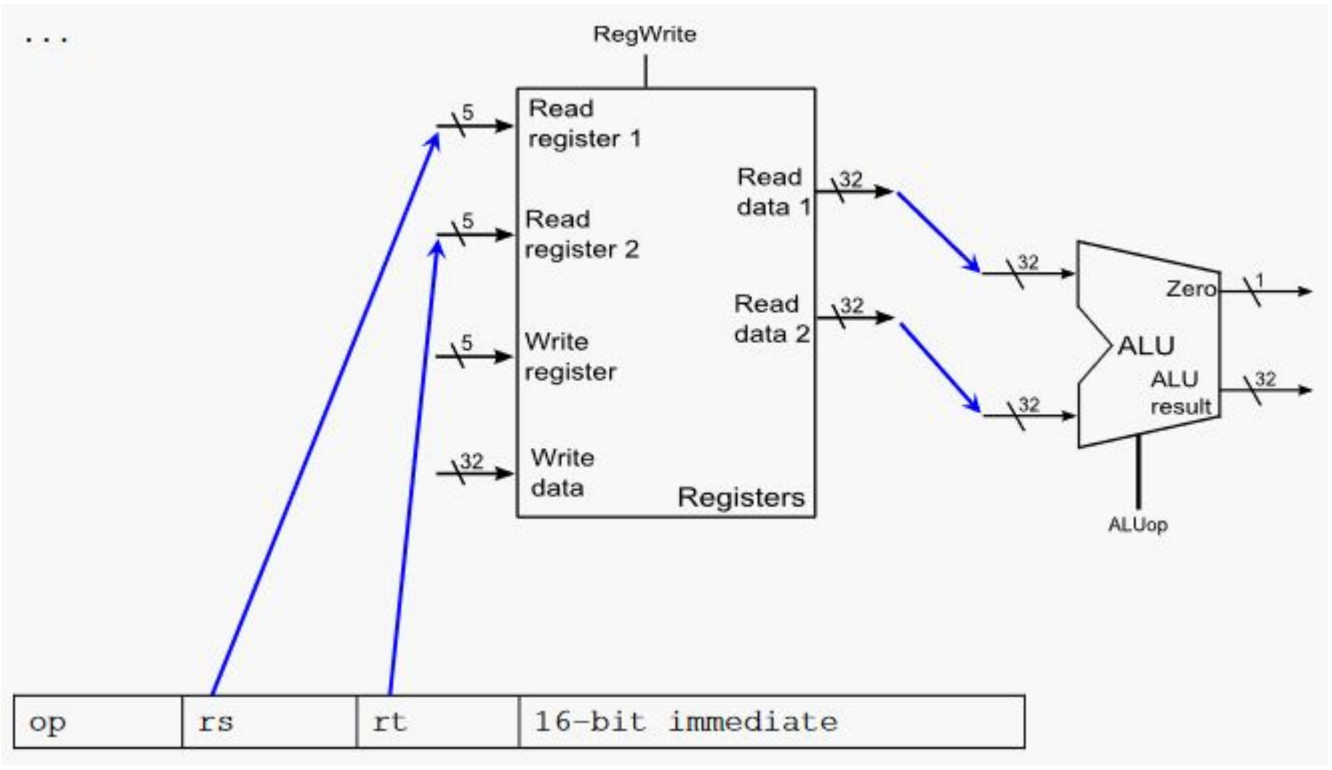
Branching Instructions

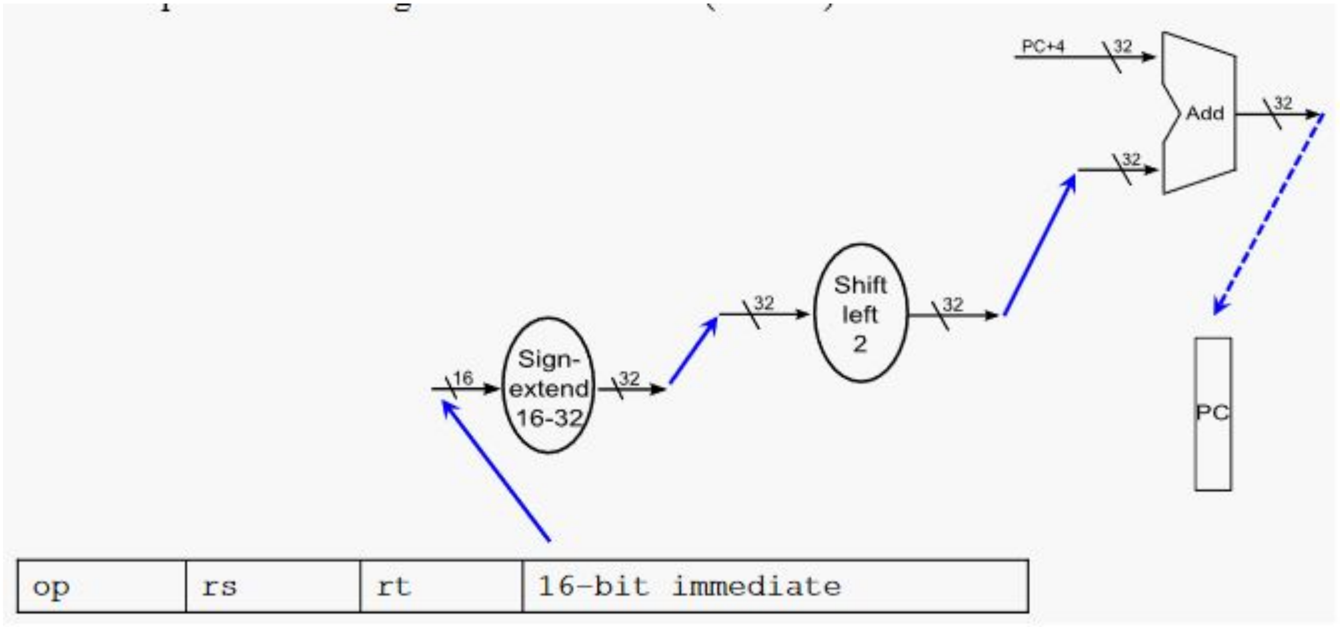
- `beq` and `bne`
  - Need to specify an address to go to
  - Also take two registers to compare
- Use I-Format:



  - `opcode` specifies `beq` (4) vs. `bne` (5)
  - `rs` and `rt` specify registers
  - How to best use `immediate` to specify addresses?

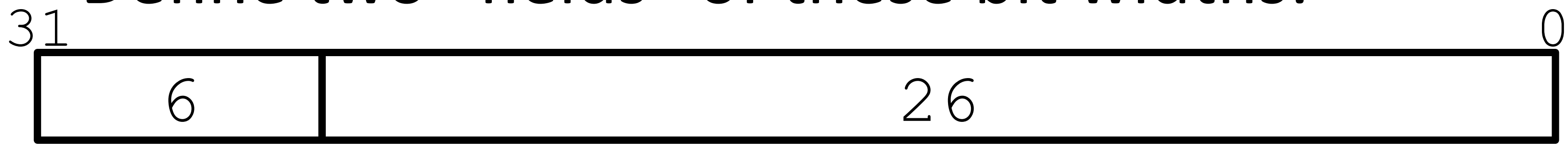
. . .


RegWrite
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
5
5
5
32
32
32
32
32
ALU
Zero
1
ALU result
32
ALUop
op
rs
rt
16-bit immediate

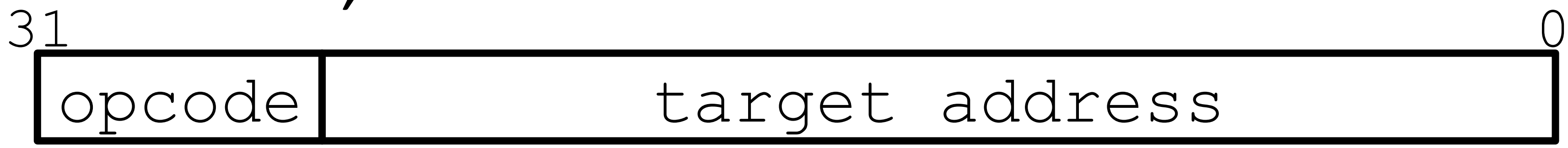

PC+4
32
Add
32
32
Shift left 2
32
32
Sign-extend 16-32
16
32
PC
op
rs
rt
16-bit immediate

J-Format Instructions

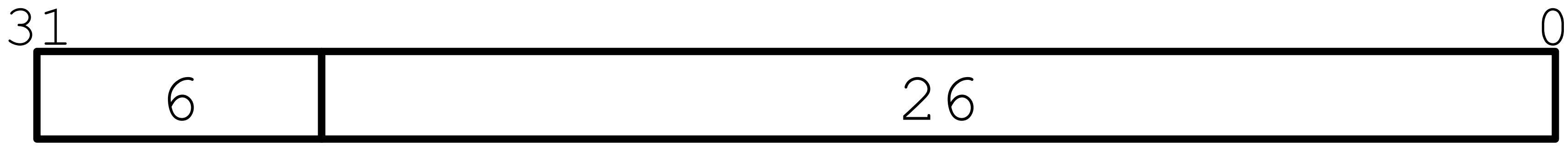
- Define two "fields" of these bit widths:

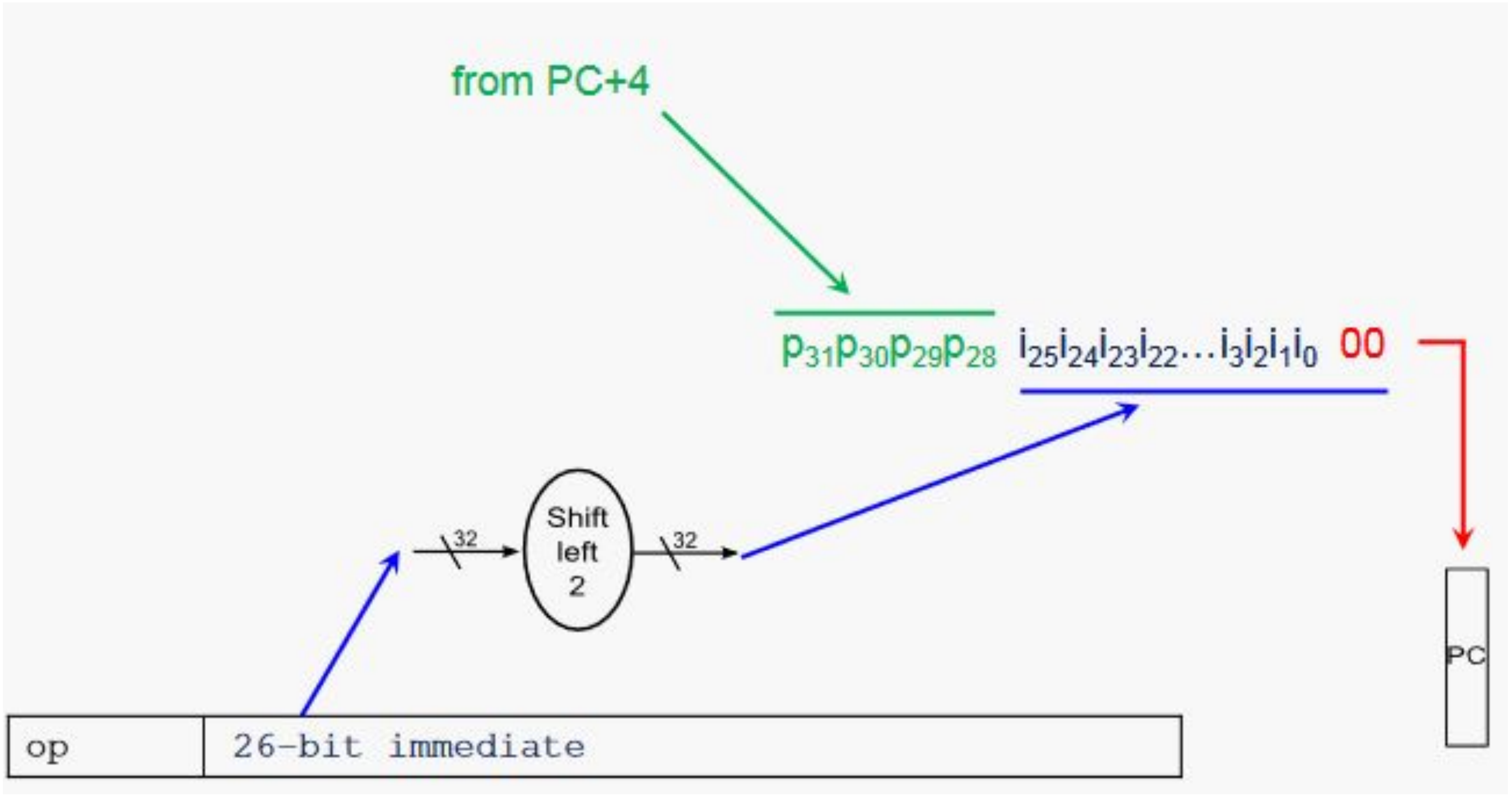


- As usual, each field has a name:



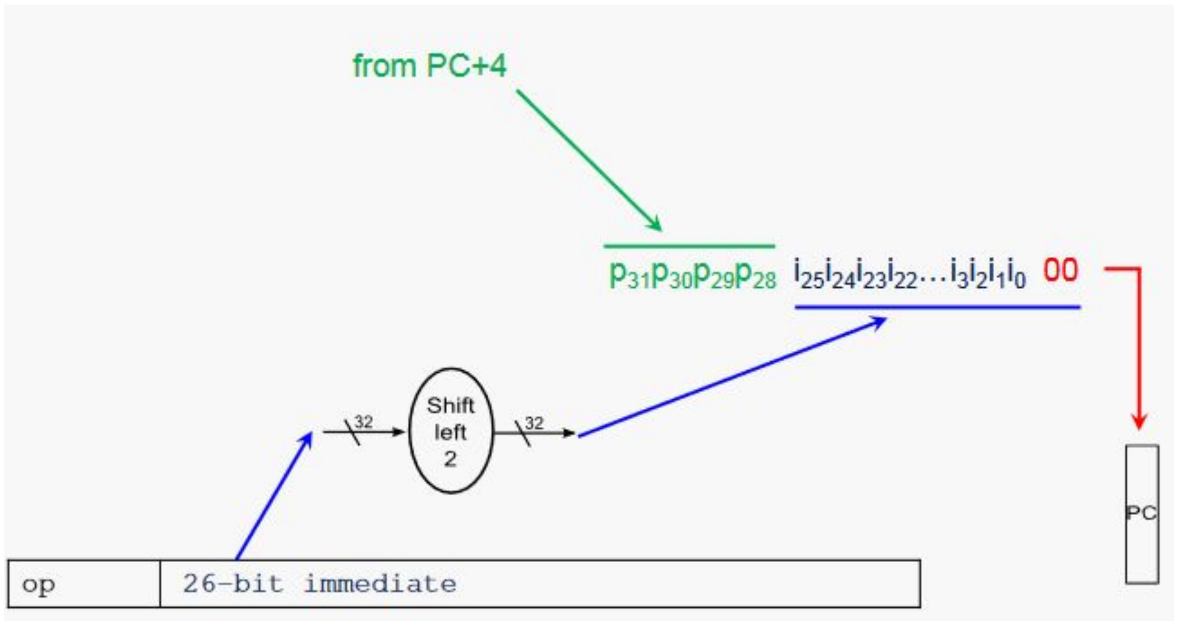
- **Key Concepts:**
  - Keep `opcode` field identical to R-Format and I-Format for consistency
  - Collapse all other fields to make room for large target address


31
0
6
26


from PC+4
$p_{31}p_{30}p_{29}p_{28}$
$i_{25}i_{24}i_{23}i_{22}\ldots i_3i_2i_1i_0$
00
32
Shift left 2
32
PC
op
26-bit immediate

J-Format Instructions

- Jump instruction:
  - New PC = { (PC+4)[31..28], target address, 0b00 }
- Notes:
  - { , , } means concatenation
    { 4 bits , 26 bits , 2 bits } = 32 bit address
  - Array indexing: [31..28] means highest 4 bits
  - For hardware reasons, use PC+4 instead of PC


from PC+4
$p_{31}p_{30}p_{29}p_{28}$
$i_{25}i_{24}i_{23}i_{22}\ldots i_3i_2i_1i_0$
00
32
Shift left 2
32
PC
op
26-bit immediate

Summary

- The Stored Program concept is very powerful
  - Instructions can be treated and manipulated the same way as data in both hardware and software
- MIPS Machine Language Instructions:

R:	opcode	rs	rt	rd	shamt	funct
I:	opcode	rs	rt	immediate		
J:	opcode	target address				

- Branches use PC-relative addressing, Jumps use absolute addressing

Assembly Practice

- Assembly is the process of converting assembly instructions into machine code
- On the following slides, there are 6-lines of assembly code, along with space for the machine code
- For each instruction,
  1) Identify the instruction type (R/I/J)
  2) Break the space into the proper fields
  3) Write field values in decimal
  4) Convert fields to binary
  5) Write out the machine code in hex
- Use your Green Sheet; answers follow

Code Questions

```
Addr   Instruction
 800   Loop: sll $t1,$s3,2

 804   addu  $t1,$t1,$s6

 808   lw    $t0,0($t1)

 812   beq   $t0,$s5, Exit

 816   addiu $s3,$s3,1

 820   j     Loop

       Exit:
```

Material from past lectures:

What type of C variable is probably stored in $s6?

Write an equivalent C loop using a□$s3, b□$s5, c□$s6. Define variable types (assume they are initialized somewhere) and feel free to introduce other variables as you like.

In English, what does this loop do?

Code Questions

```
Addr   Instruction
 800   Loop: sll $t1,$s3,2

 804   addu  $t1,$t1,$s6

 808   lw    $t0,0($t1)

 812   beq   $t0,$s5, Exit

 816   addiu $s3,$s3,1

 820   j     Loop

       Exit:
```

Material from past lectures:

What type of C variable is probably stored in $s6?

int * (or any pointer)

Write an equivalent C loop using a□$s3, b□$s5, c□$s6. Define variable types (assume they are initialized somewhere) and feel free to introduce other variables as you like.

```
int a,b,*c;
/* values initialized */
while(c[a] != b)  a++;
```

In English, what does this loop do?

Finds an entry in array c that matches b.

Assembly Practice Question

```
Addr  Instruction
 800  Loop: sll $t1,$s3,2
__:
 804  addu  $t1,$t1,$s6
__:
 808  lw    $t0,0($t1)
__:
 812  beq   $t0,$s5, Exit
__:
 816  addiu $s3,$s3,1
__:
 820  j     Loop
__:
      Exit:
```

Assembly Practice Answer (1/4)

Addr Instruction

```
 800   Loop: sll $t1,$s3,2
```

R:

opcode	rs	rt	rd	shamt	funct

```
 804   addu  $t1,$t1,$s6
```

R:

opcode	rs	rt	rd	shamt	funct

```
 808   lw    $t0,0($t1)
```

I:

opcode	rs	rt	immediate

```
 812   beq   $t0,$s5, Exit
```

I:

opcode	rs	rt	immediate

```
 816   addiu $s3,$s3,1
```

I:

opcode	rs	rt	immediate

```
 820   j     Loop
```

J:

opcode	target address

```
       Exit:
```

Assembly Practice Answer (2/4)

```
Addr   Instruction
800    Loop: sll $t1,$s3,2
```

R:

0	0	19	9	2	0

```
804    addu  $t1,$t1,$s6
```

R:

0	9	22	9	0	33

```
808    lw    $t0,0($t1)
```

I:

35	9	8	0

```
812    beq   $t0,$s5, Exit
```

I:

4	8	21	2

```
816    addiu $s3,$s3,1
```

I:

8	19	19	1

```
820    j     Loop
```

J:

2	200

```
       Exit:
```

Assembly Practice Answer (3/4)

```
Addr  Instruction
 800  Loop: sll $t1,$s3,2
R: | 000000 | 00000 | 10011 | 01001 | 00010 | 000000 |
 804  addu  $t1,$t1,$s6
R: | 000000 | 01001 | 10110 | 01001 | 00000 | 100001 |
 808  lw    $t0,0($t1)
I: | 100011 | 01001 | 01000 | 0000 0000 0000 0000 |
 812  beq   $t0,$s5, Exit
I: | 000100 | 01000 | 10101 | 0000 0000 0000 0010 |
 816  addiu $s3,$s3,1
I: | 001000 | 10011 | 10011 | 0000 0000 0000 0001 |
 820  j     Loop
J: | 000010 | 00 0000 0000 0000 0000 1100 1000 |
      Exit:
```

Assembly Practice Answer

```
Addr  Instruction
 800  Loop: sll $t1,$s3,2
R:    0x 0013 4880
 804  addu  $t1,$t1,$s6
R:    0x 0136 4821
 808  lw    $t0,0($t1)
I:    0x 8D28 0000
 812  beq   $t0,$s5, Exit
I:    0x 1115 0002
 816  addiu $s3,$s3,1
I:    0x 2273 0001
 820  j     Loop
J:    0x 0800 00C8
      Exit:
```

Example: `f=(g+h)-(i+j);`

- `int f, g, h, i, j;`
- `f, g, h, i, j` → `$s0, $s1, $s2, $s3, $s4`

[Answer]

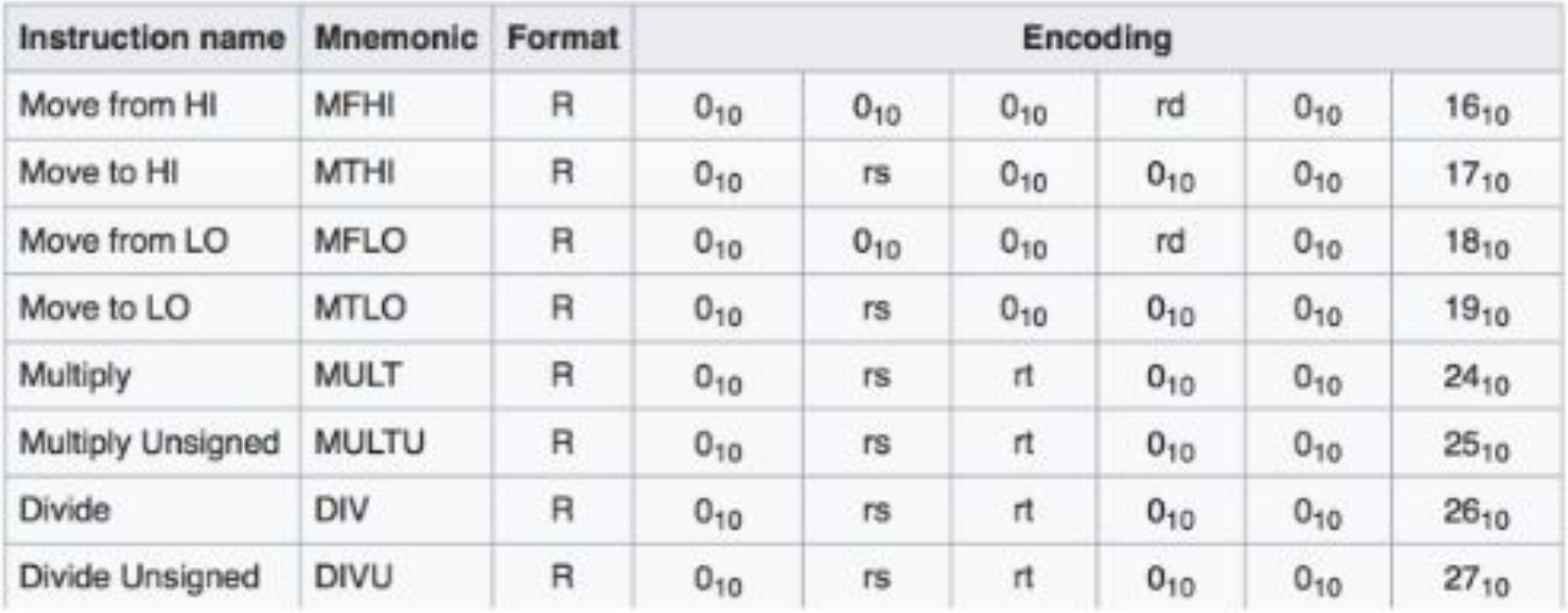
```
add   $t0, $s1, $s2
add   $t1, $s3, $s4
sub   $s0, $t0, $t1
```

MIPS R-type Instructions

Instruction name	Mnemonic	Format	Encoding					
Add	ADD	R	0_{10}	rs	rt	rd	0_{10}	32_{10}
Add Unsigned	ADDU	R	0_{10}	rs	rt	rd	0_{10}	33_{10}
Subtract	SUB	R	0_{10}	rs	rt	rd	0_{10}	34_{10}
Subtract Unsigned	SUBU	R	0_{10}	rs	rt	rd	0_{10}	35_{10}
And	AND	R	0_{10}	rs	rt	rd	0_{10}	36_{10}
Or	OR	R	0_{10}	rs	rt	rd	0_{10}	37_{10}
Exclusive Or	XOR	R	0_{10}	rs	rt	rd	0_{10}	38_{10}
Nor	NOR	R	0_{10}	rs	rt	rd	0_{10}	39_{10}
Set on Less Than	SLT	R	0_{10}	rs	rt	rd	0_{10}	42_{10}
Set on Less Than Unsigned	SLTU	R	0_{10}	rs	rt	rd	0_{10}	43_{10}

Instruction name	Mnemonic	Format	Encoding					
Shift Left Logical	SLL	R	0_{10}	0_{10}	rt	rd	ra	0_{10}
Shift Right Logical	SRL	R	0_{10}	0_{10}	rt	rd	sa	2_{10}
Shift Right Arithmetic	SRA	R	0_{10}	0_{10}	rt	rd	sa	3_{10}
Shift Left Logical Variable	SLLV	R	0_{10}	rs	rt	rd	0_{10}	4_{10}
Shift Right Logical Variable	SRLV	R	0_{10}	rs	rt	rd	0_{10}	6_{10}
Shift Right Arithmetic Variable	SRAV	R	0_{10}	rs	rt	rd	0_{10}	7_{10}

MIPS R-type Instructions



Instruction name	Mnemonic	Format	Encoding					
Move from HI	MFHI	R	0_{10}	0_{10}	0_{10}	rd	0_{10}	16_{10}
Move to HI	MTHI	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	17_{10}
Move from LO	MFLO	R	0_{10}	0_{10}	0_{10}	rd	0_{10}	18_{10}
Move to LO	MTLO	R	0_{10}	rs	0_{10}	0_{10}	0_{10}	19_{10}
Multiply	MULT	R	0_{10}	rs	rt	0_{10}	0_{10}	24_{10}
Multiply Unsigned	MULTU	R	0_{10}	rs	rt	0_{10}	0_{10}	25_{10}
Divide	DIV	R	0_{10}	rs	rt	0_{10}	0_{10}	26_{10}
Divide Unsigned	DIVU	R	0_{10}	rs	rt	0_{10}	0_{10}	27_{10}

op	rs	rt	immediate

I format

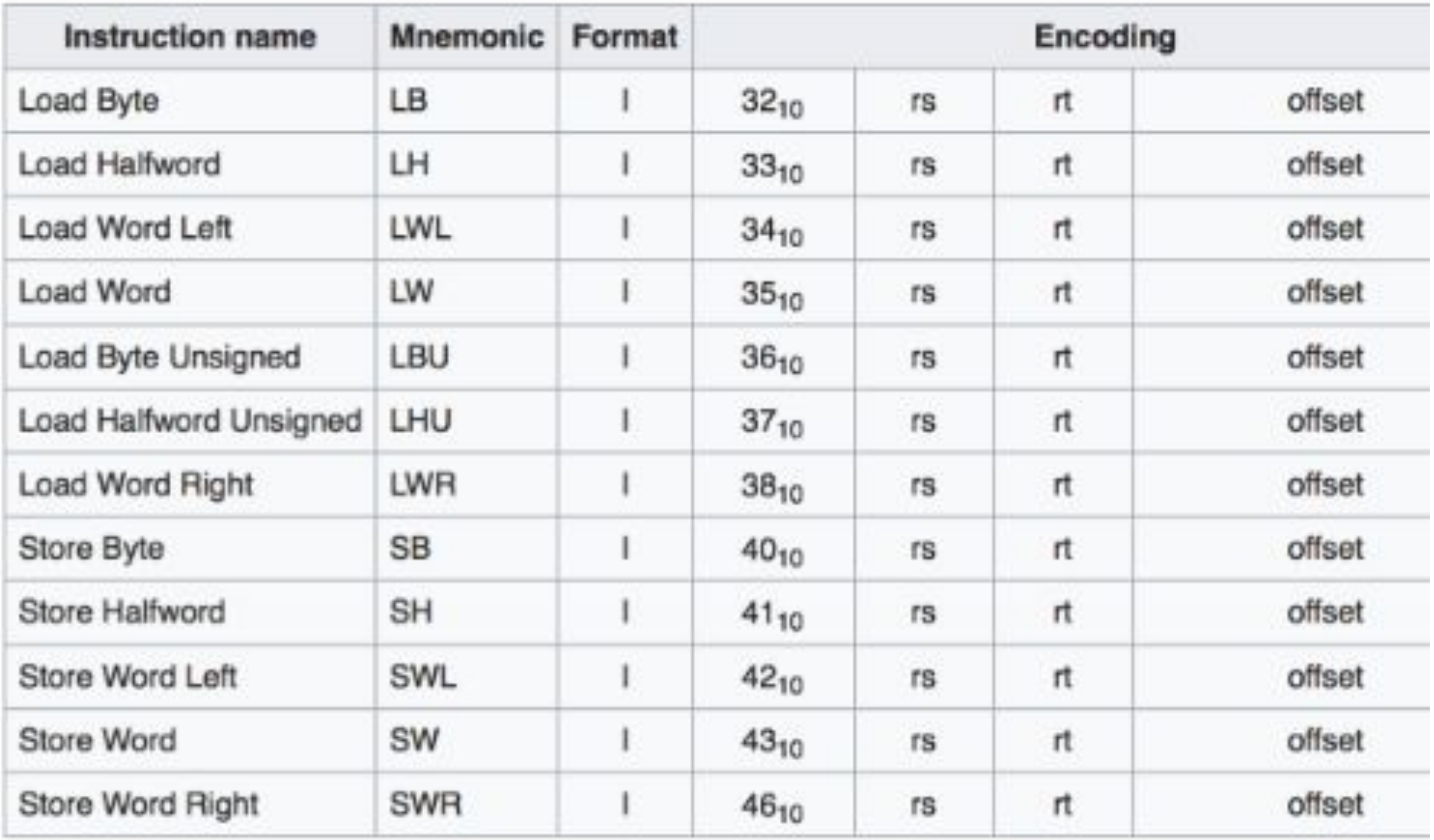
MIPS I-type Instructions

Add Immediate	ADDI	I	8_{10}	rs	rd	immediate
Add Immediate Unsigned	ADDIU	I	9_{10}	\$s	\$d	immediate
Set on Less Than Immediate	SLTI	I	10_{10}	\$s	\$d	immediate
Set on Less Than Immediate Unsigned	SLTIU	I	11_{10}	\$s	\$d	immediate
And Immediate	ANDI	I	12_{10}	\$s	\$d	immediate
Or Immediate	ORI	I	13_{10}	\$s	\$d	immediate
Exclusive Or Immediate	XORI	I	14_{10}	\$s	\$d	immediate
Load Upper Immediate	LUI	I	15_{10}	0_{10}	\$d	immediate

Branch on Equal	BEQ	I	4_{10}	rs	rt	offset
Branch on Not Equal	BNE	I	5_{10}	rs	rt	offset
Branch on Less Than or Equal to Zero	BLEZ	I	6_{10}	rs	0_{10}	offset
Branch on Greater Than Zero	BGTZ	I	7_{10}	rs	0_{10}	offset

Branch on Less Than Zero	BLTZ	I	1_{10}	rs	0_{10}	offset
Branch on Greater Than or Equal to Zero	BGEZ	I	1_{10}	rs	1_{10}	offset
Branch on Less Than Zero and Link	BLTZAL	I	1_{10}	rs	16	offset
Branch on Greater Than or Equal to Zero and Link	BGEZAL	I	1_{10}	rs	17	offset

MIPS I-type Instructions



Instruction name	Mnemonic	Format	Encoding			
Load Byte	LB	I	32_{10}	rs	rt	offset
Load Halfword	LH	I	33_{10}	rs	rt	offset
Load Word Left	LWL	I	34_{10}	rs	rt	offset
Load Word	LW	I	35_{10}	rs	rt	offset
Load Byte Unsigned	LBU	I	36_{10}	rs	rt	offset
Load Halfword Unsigned	LHU	I	37_{10}	rs	rt	offset
Load Word Right	LWR	I	38_{10}	rs	rt	offset
Store Byte	SB	I	40_{10}	rs	rt	offset
Store Halfword	SH	I	41_{10}	rs	rt	offset
Store Word Left	SWL	I	42_{10}	rs	rt	offset
Store Word	SW	I	43_{10}	rs	rt	offset
Store Word Right	SWR	I	46_{10}	rs	rt	offset