Computer Architecture
Computer Engineering Track
Tutorial 10

Processor Architecture Layout

MIPS Architecture and Instruction Set

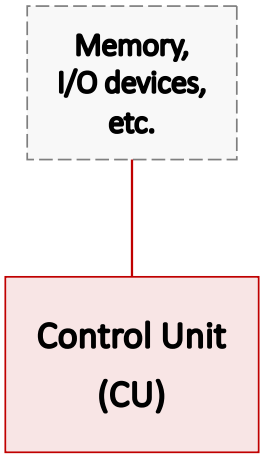
Artem Burmyakov, Muhammad Fahim, Alexander Tormasov

November 05, 2020

INNOPOLIS UNIVERSITY

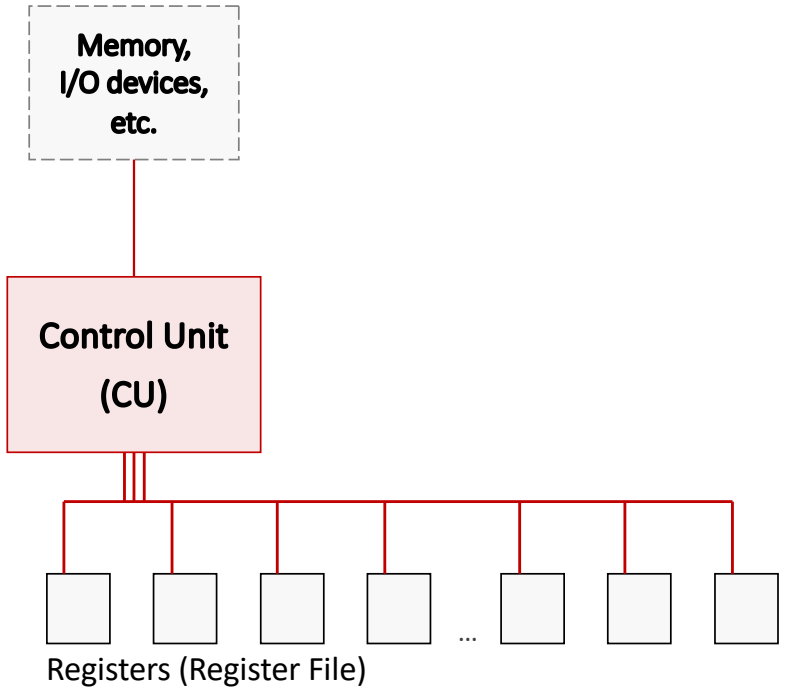
Processor Architecture Layout

- Fetches instructions from memory



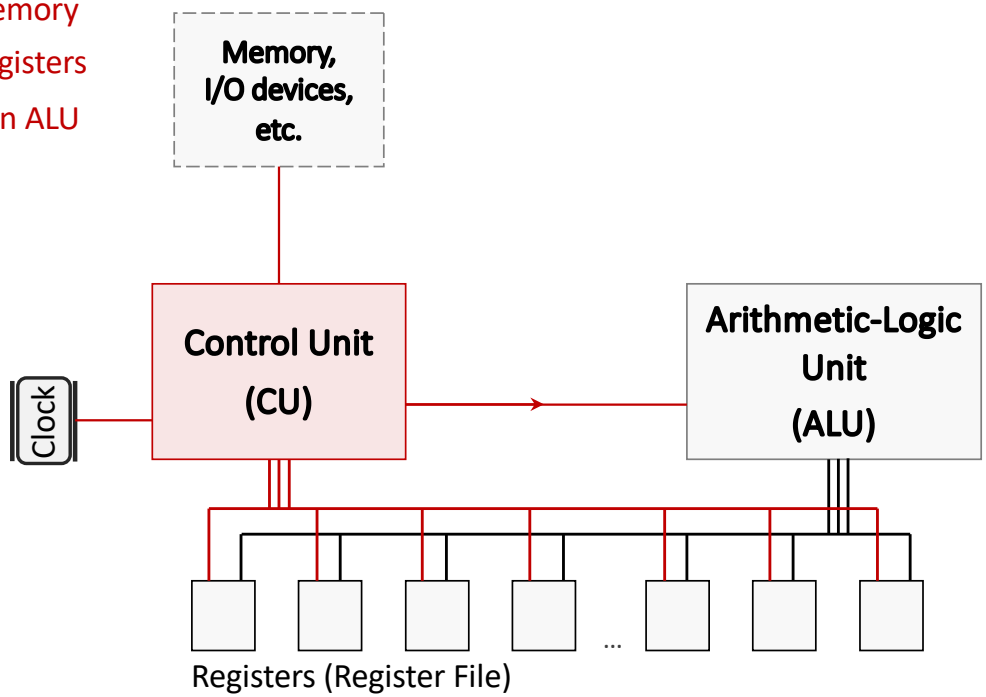
Processor Architecture Layout

- Fetches instructions from memory
- Decodes instructions over registers



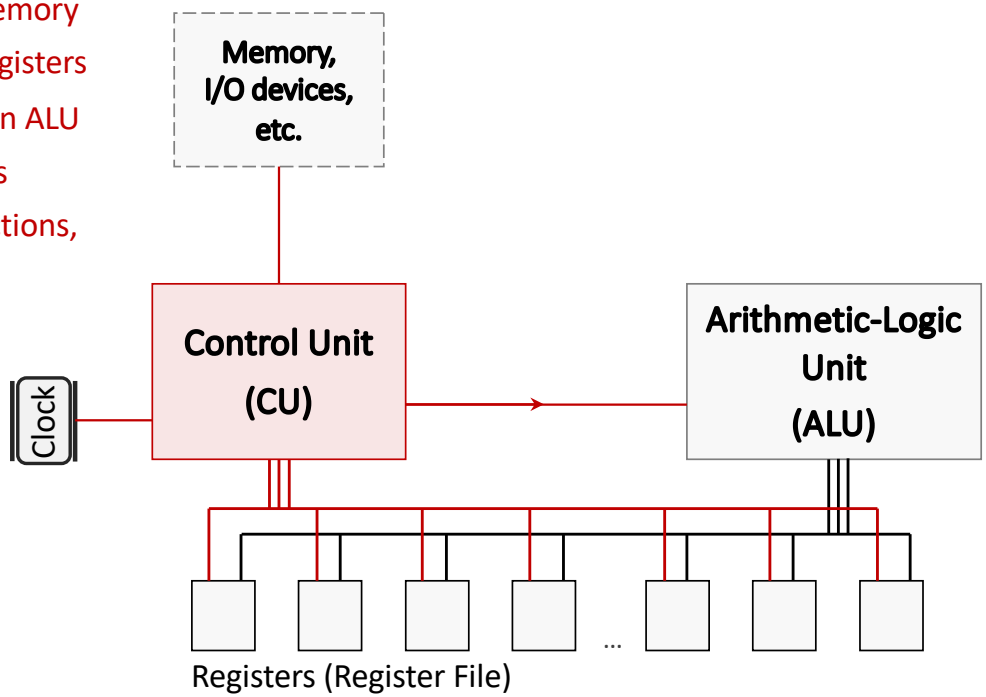
Processor Architecture Layout

- Fetches instructions from memory
- Decodes instructions over registers
- Controls the execution flow in ALU



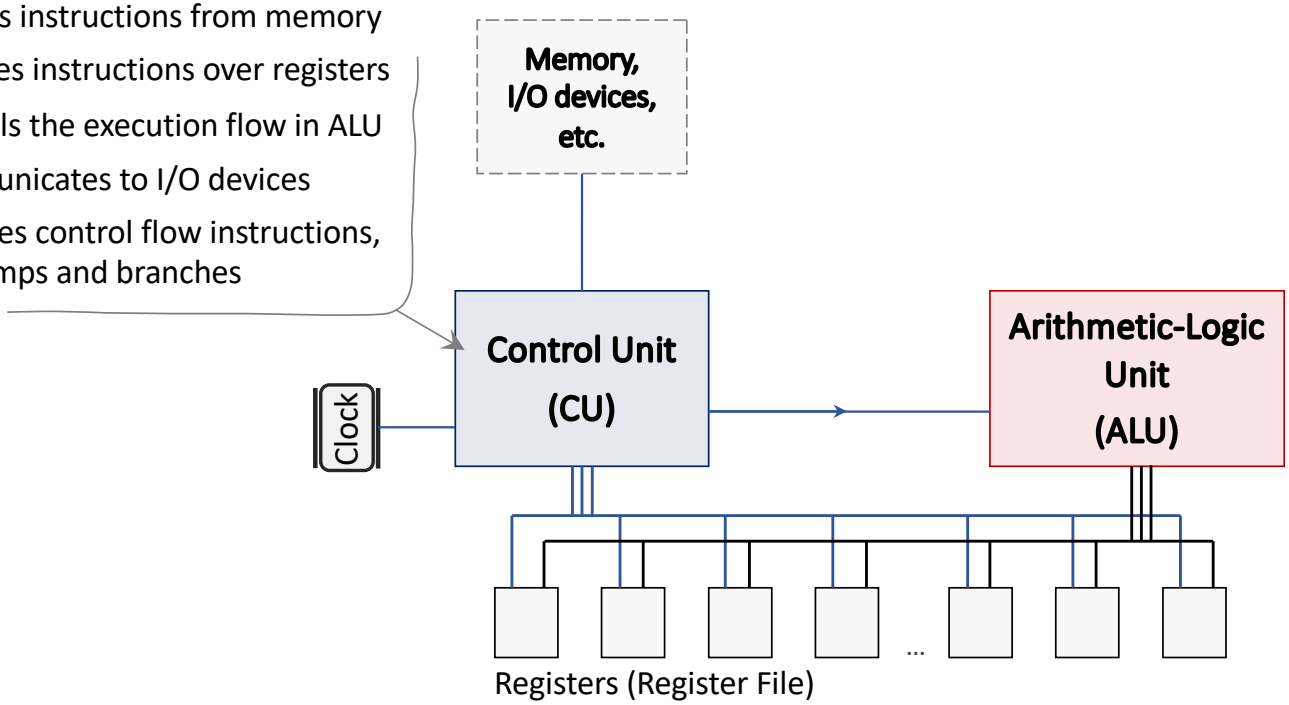
Processor Architecture Layout

- Fetches instructions from memory
- Decodes instructions over registers
- Controls the execution flow in ALU
- Communicates to I/O devices
- Executes control flow instructions, e.g. jumps and branches



Processor Architecture Layout

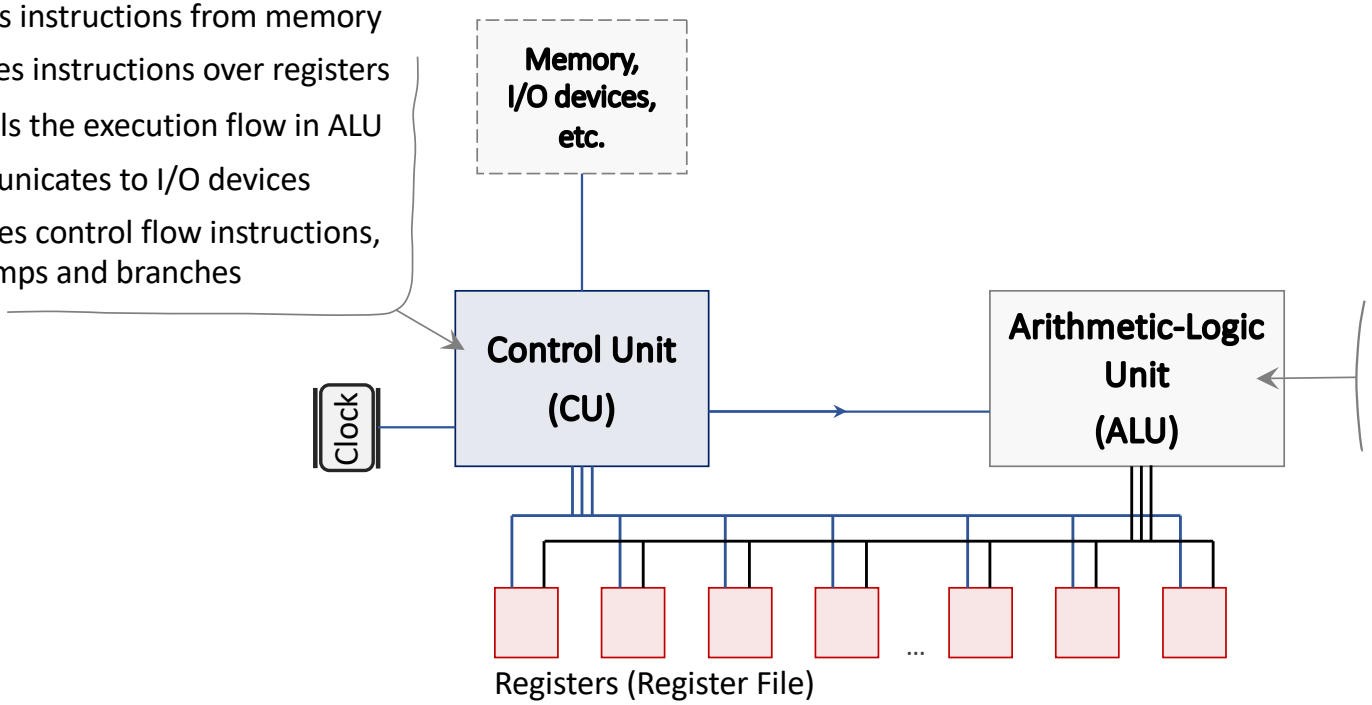
- Fetches instructions from memory
- Decodes instructions over registers
- Controls the execution flow in ALU
- Communicates to I/O devices
- Executes control flow instructions, e.g. jumps and branches



- Executes arithmetic and logic instructions
- Designed for the minimized propagation delay

Processor Architecture Layout

- Fetches instructions from memory
- Decodes instructions over registers
- Controls the execution flow in ALU
- Communicates to I/O devices
- Executes control flow instructions, e.g. jumps and branches



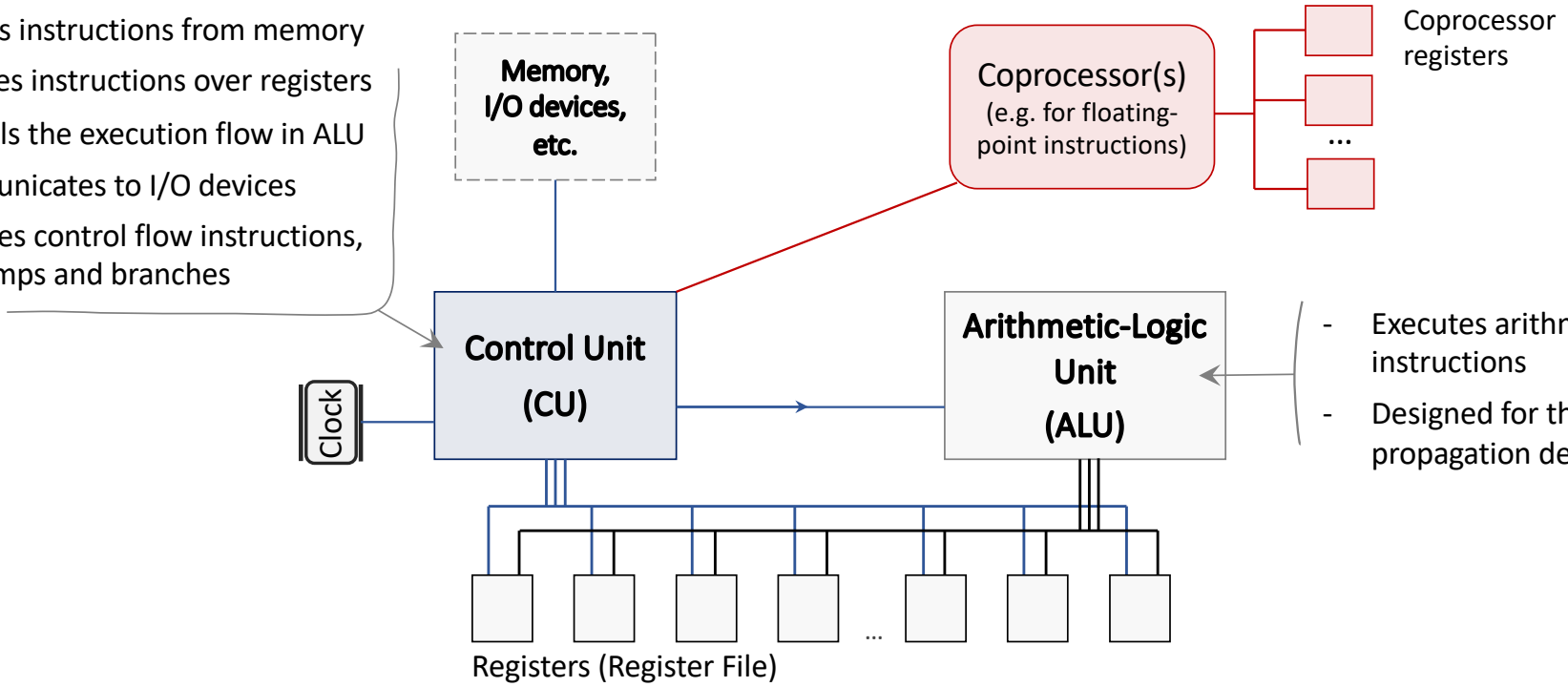
- Executes arithmetic and logic instructions
- Designed for the minimized propagation delay

- Fast memory elements, directly connected to CU and ALU
- Store data for instruction to execute (e.g. instruction code, its input arguments), as well as the result of its execution
- Every register is reserved for a specific purpose (e.g. to store input arguments, or the result of computation)

Processor Architecture Layout

- An optimized circuit for a limited set of specific instructions (e.g. floating-point)
- Has its own registers

- Fetches instructions from memory
- Decodes instructions over registers
- Controls the execution flow in ALU
- Communicates to I/O devices
- Executes control flow instructions, e.g. jumps and branches



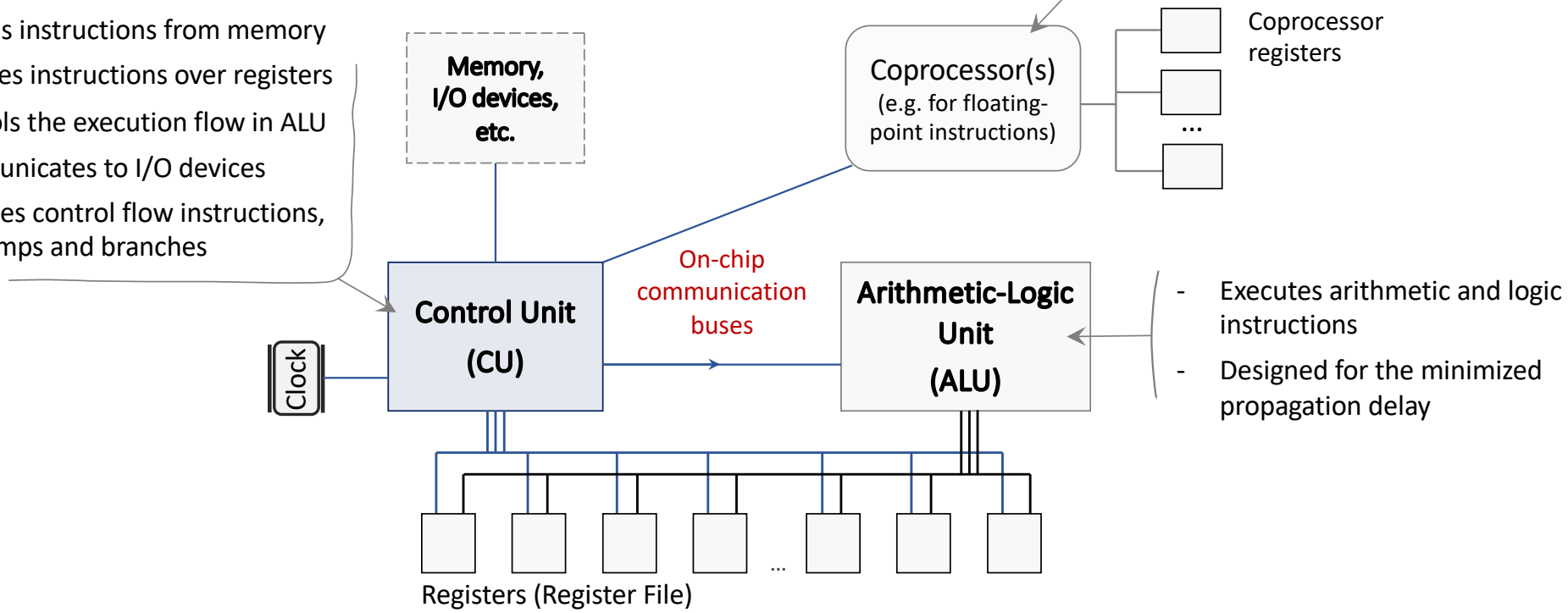
- Executes arithmetic and logic instructions
- Designed for the minimized propagation delay

- Fast memory elements, directly connected to CU and ALU
- Store data for instruction to execute (e.g. instruction code, its input arguments), as well as the result of its execution
- Every register is reserved for a specific purpose (e.g. to store input arguments, or the result of computation)

Processor Architecture Layout

- An optimized circuit for a limited set of specific instructions (e.g. floating-point)
- Has its own registers

- Fetches instructions from memory
- Decodes instructions over registers
- Controls the execution flow in ALU
- Communicates to I/O devices
- Executes control flow instructions, e.g. jumps and branches



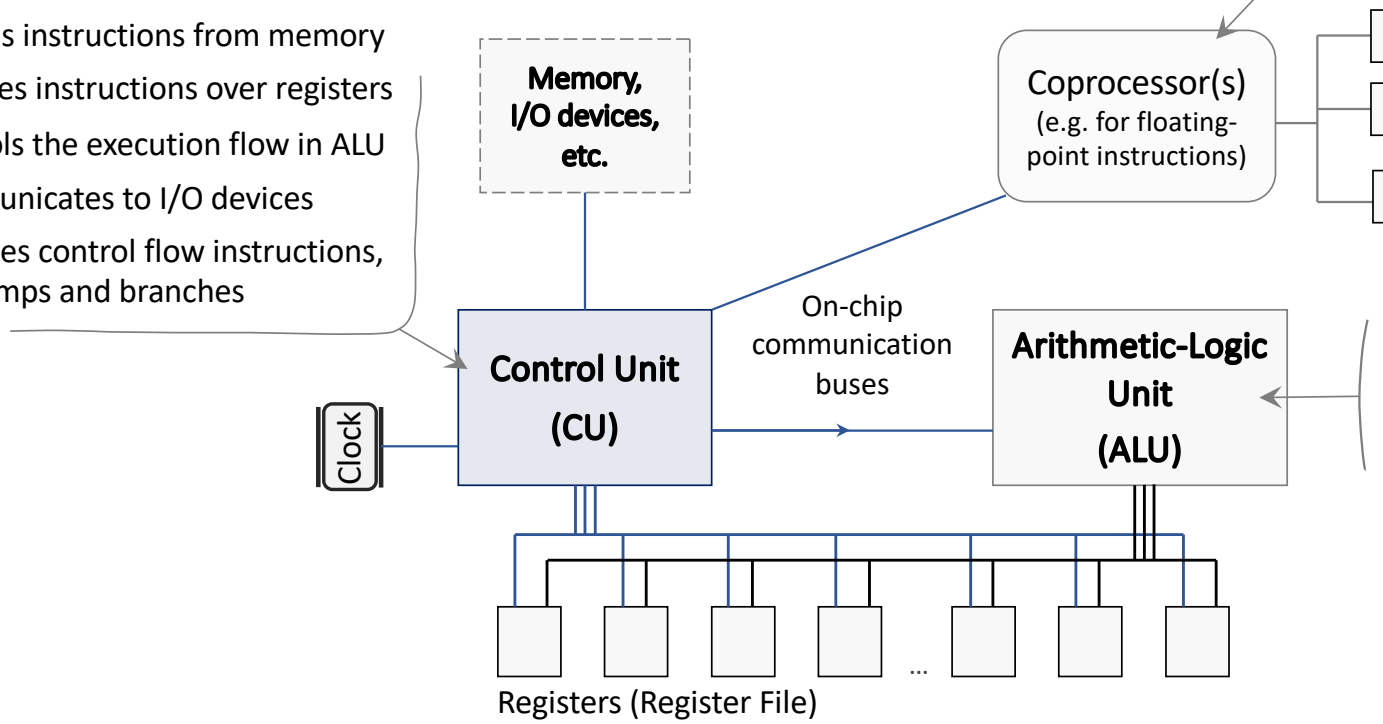
- Executes arithmetic and logic instructions
- Designed for the minimized propagation delay

- Fast memory elements, directly connected to CU and ALU
- Store data for instruction to execute (e.g. instruction code, its input arguments), as well as the result of its execution
- Every register is reserved for a specific purpose (e.g. to store input arguments, or the result of computation)

Some Aspects Affecting the CPU Clock Frequency

- An optimized circuit for a limited set of specific instructions (e.g. floating-point)
- Has its own registers

- Fetches instructions from memory
- Decodes instructions over registers
- Controls the execution flow in ALU
- Communicates to I/O devices
- Executes control flow instructions, e.g. jumps and branches



- Executes arithmetic and logic instructions
- Designed for the minimized propagation delay

- Fast memory elements, directly connected to CU and ALU
- Store data for instruction to execute (e.g. instruction code, its input arguments), as well as the result of its execution
- Every register is reserved for a specific purpose (e.g. to store input arguments, or the result of computation)

A supported Instruction Set (IS):

The worst-case propagation delay of the slowest instruction matters most

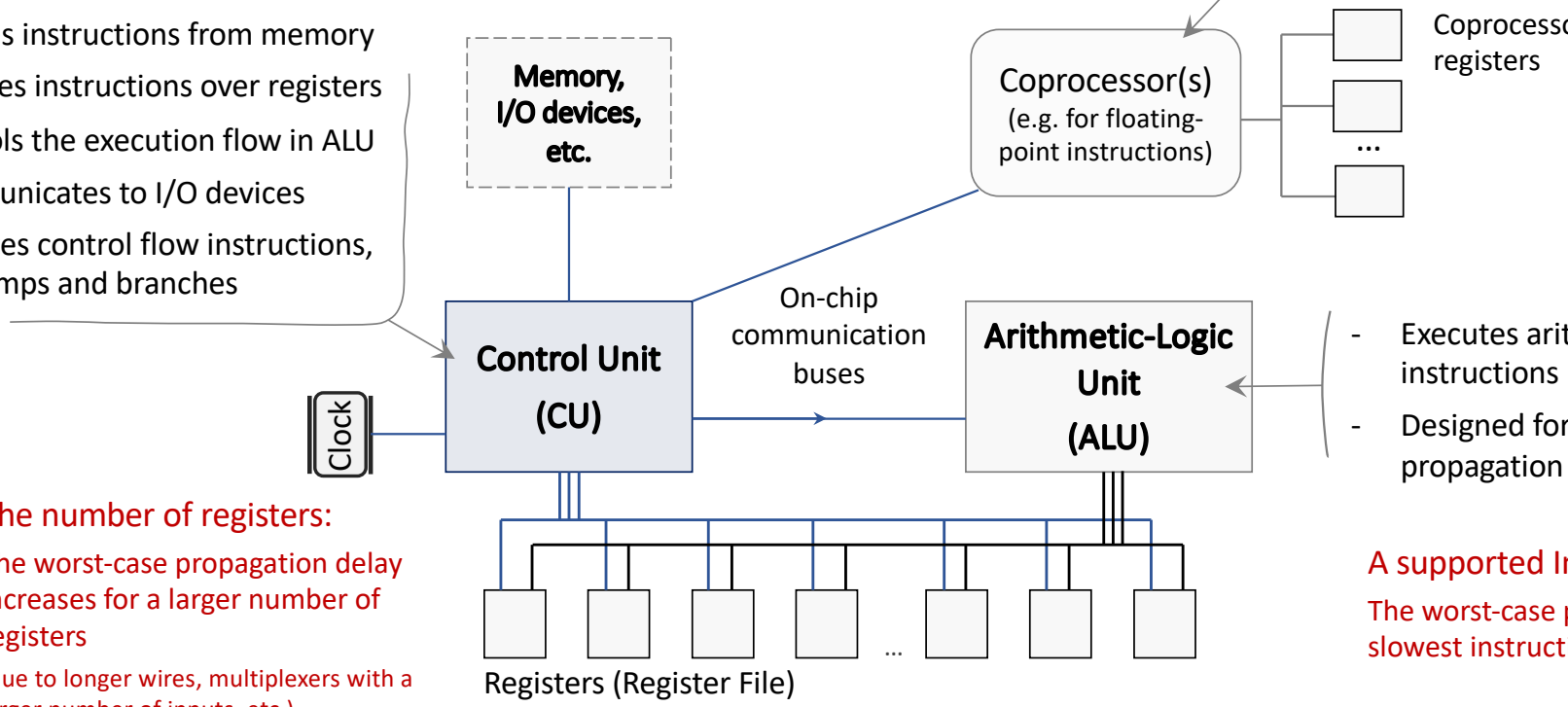
Note: The design objective is to have an optimized performance in the average use case, and not to have a higher number of instructions supported

(recall the difference between RISC and CISC)

Some Aspects Affecting the CPU Clock Frequency

- An optimized circuit for a limited set of specific instructions (e.g. floating-point)
- Has its own registers

- Fetches instructions from memory
- Decodes instructions over registers
- Controls the execution flow in ALU
- Communicates to I/O devices
- Executes control flow instructions, e.g. jumps and branches



- Executes arithmetic and logic instructions
- Designed for the minimized propagation delay

The number of registers:

The worst-case propagation delay increases for a larger number of registers

(due to longer wires, multiplexers with a larger number of inputs, etc.)

A supported Instruction Set (IS):

The worst-case propagation delay of the slowest instruction matters most

Note: The design objective is to have an optimized performance in the average use case, and not to have a higher number of instructions supported

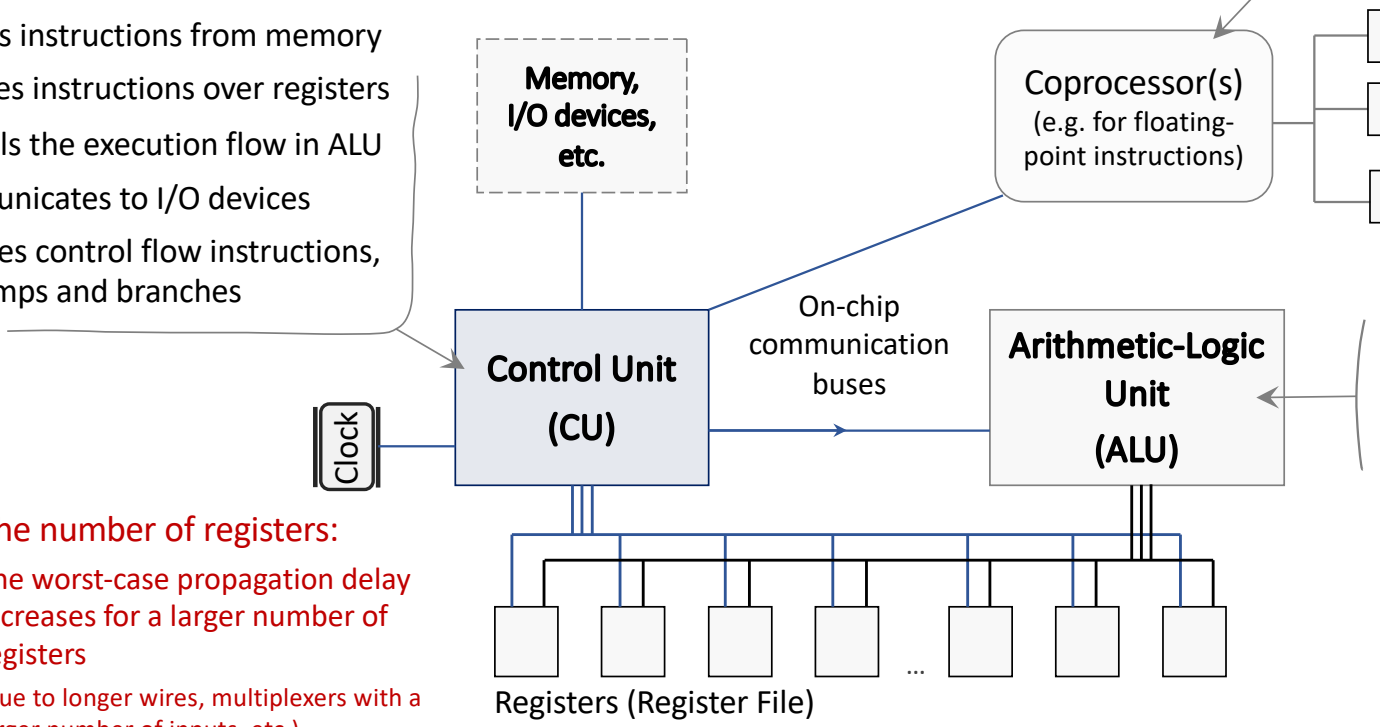
(recall the difference between RISC and CISC)

- Fast memory elements, directly connected to CU and ALU
- Store data for instruction to execute (e.g. instruction code, its input arguments), as well as the result of its execution
- Every register is reserved for a specific purpose (e.g. to store input arguments, or the result of computation)

Some Aspects Affecting the CPU Clock Frequency

- An optimized circuit for a limited set of specific instructions (e.g. floating-point)
- Has its own registers

- Fetches instructions from memory
- Decodes instructions over registers
- Controls the execution flow in ALU
- Communicates to I/O devices
- Executes control flow instructions, e.g. jumps and branches



- Executes arithmetic and logic instructions
- Designed for the minimized propagation delay

The number of registers:

The worst-case propagation delay increases for a larger number of registers

(due to longer wires, multiplexers with a larger number of inputs, etc.)

A supported Instruction Set (IS):

The worst-case propagation delay of the slowest instruction matters most

Note: The design objective is to have an optimized performance in the average use case, and not to have a higher number of instructions supported

(recall the difference between RISC and CISC)

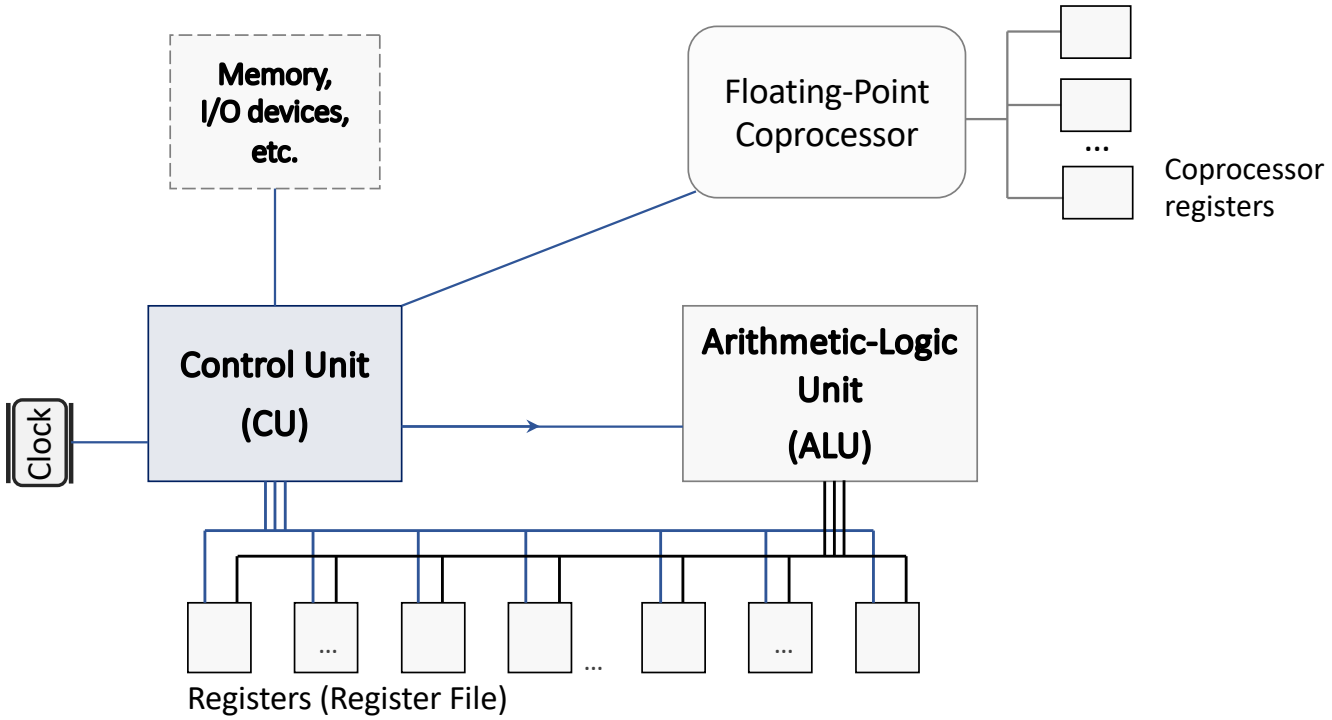
- Fast memory elements, directly connected to CU and ALU
- Store data for instruction to execute (e.g. instruction code, its input arguments), as well as the result of its execution
- Every register is reserved for a specific purpose (e.g. to store input arguments, or the result of computation)

Some other aspects:

- The length of the binary representation for instructions;
- Storage capacity of registers

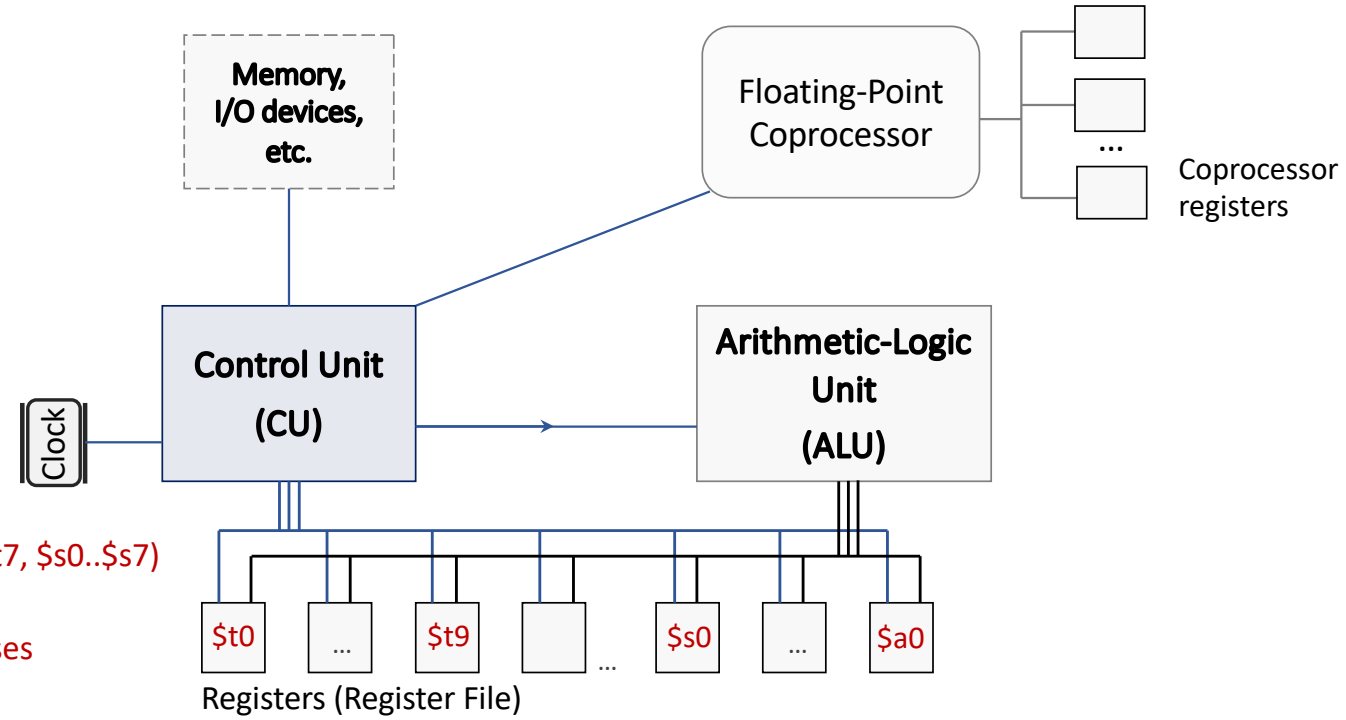
MIPS Processor Architecture

MIPS is a RISC architecture



MIPS Processor Architecture

MIPS is a RISC architecture

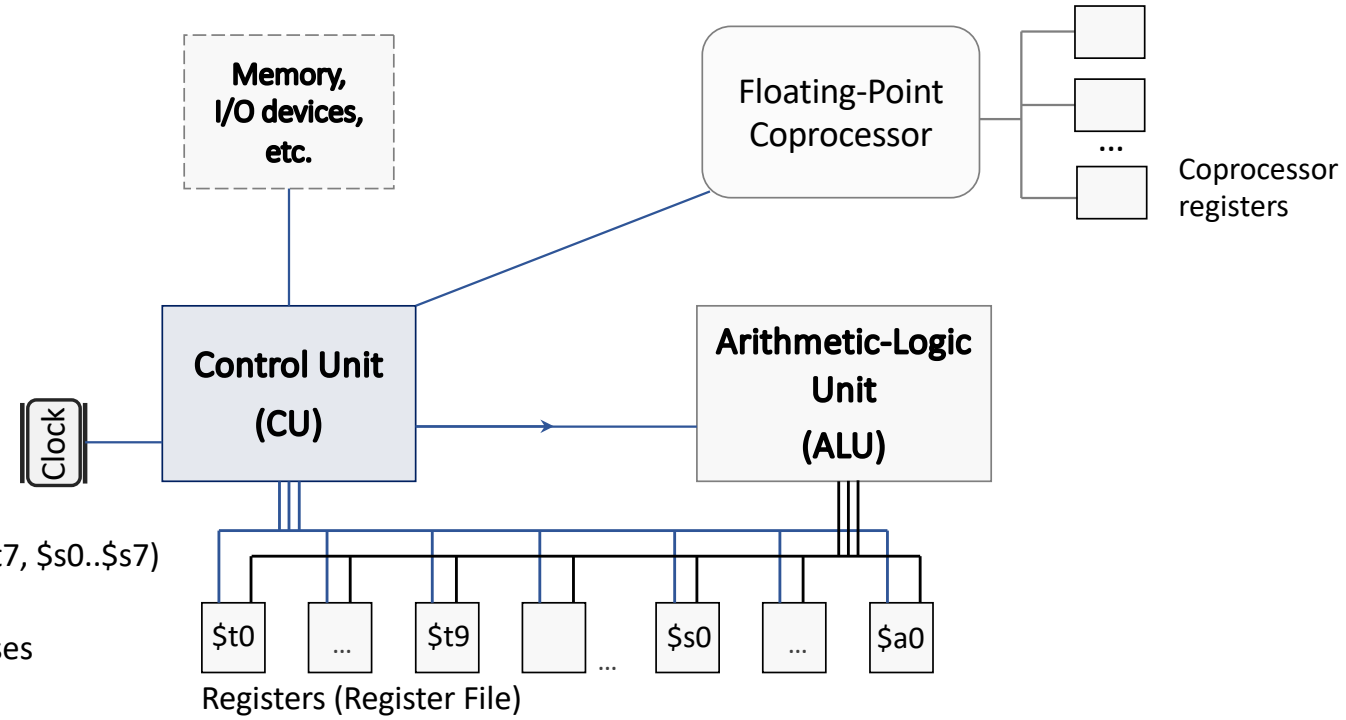


32 general-purpose registers:

- Have their names (like $t0..$t7, $s0..$s7)
- Directly addressable
- Reserved for different purposes

MIPS Processor Architecture

MIPS is a RISC architecture



32 general-purpose registers:

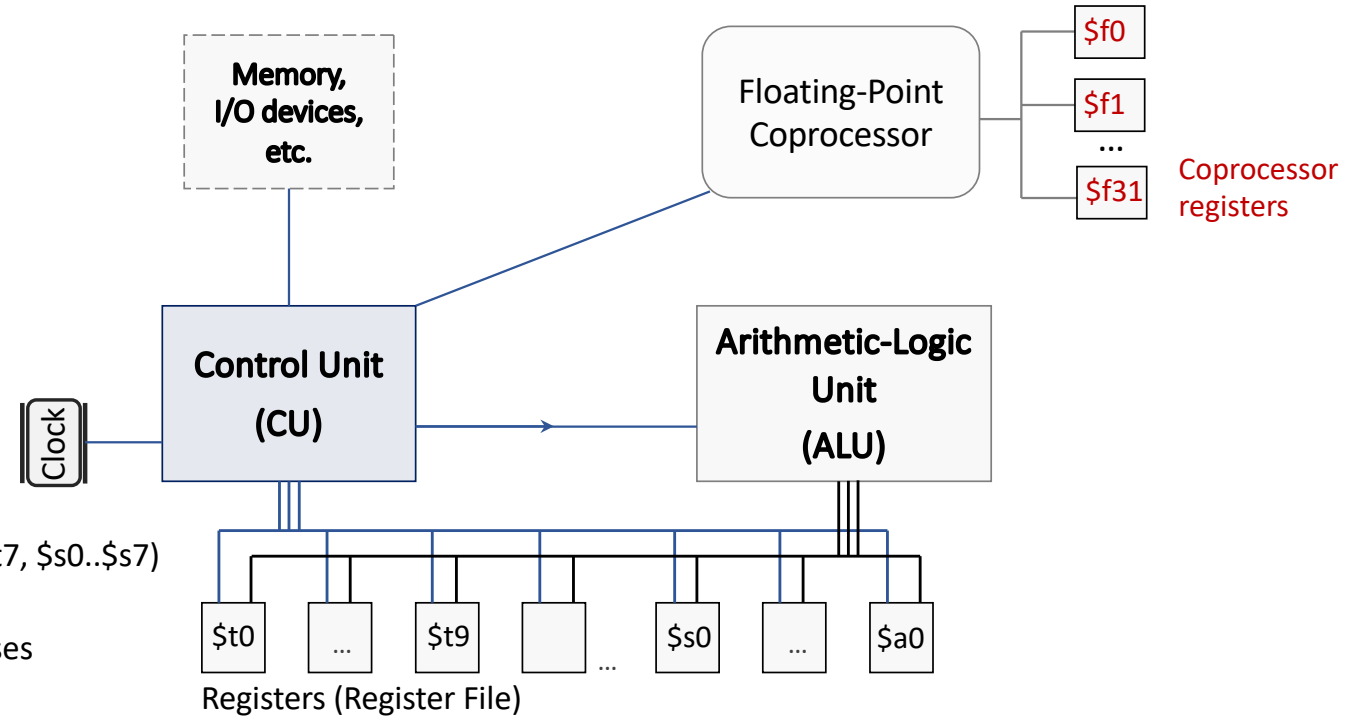
- Have their names (like $t0..$t7, $s0..$s7)
- Directly addressable
- Reserved for different purposes

"Register spilling":
If the number of live variables exceeds the number of available registers, then the compiler spills some variables from registers into memory

MIPS Processor Architecture

MIPS is a RISC architecture

32 registers for floating-point instructions:
- Named by $f0..$f31
- Directly addressable
- Reserved for different purposes



32 general-purpose registers:
- Have their names (like $t0..$t7, $s0..$s7)
- Directly addressable
- Reserved for different purposes

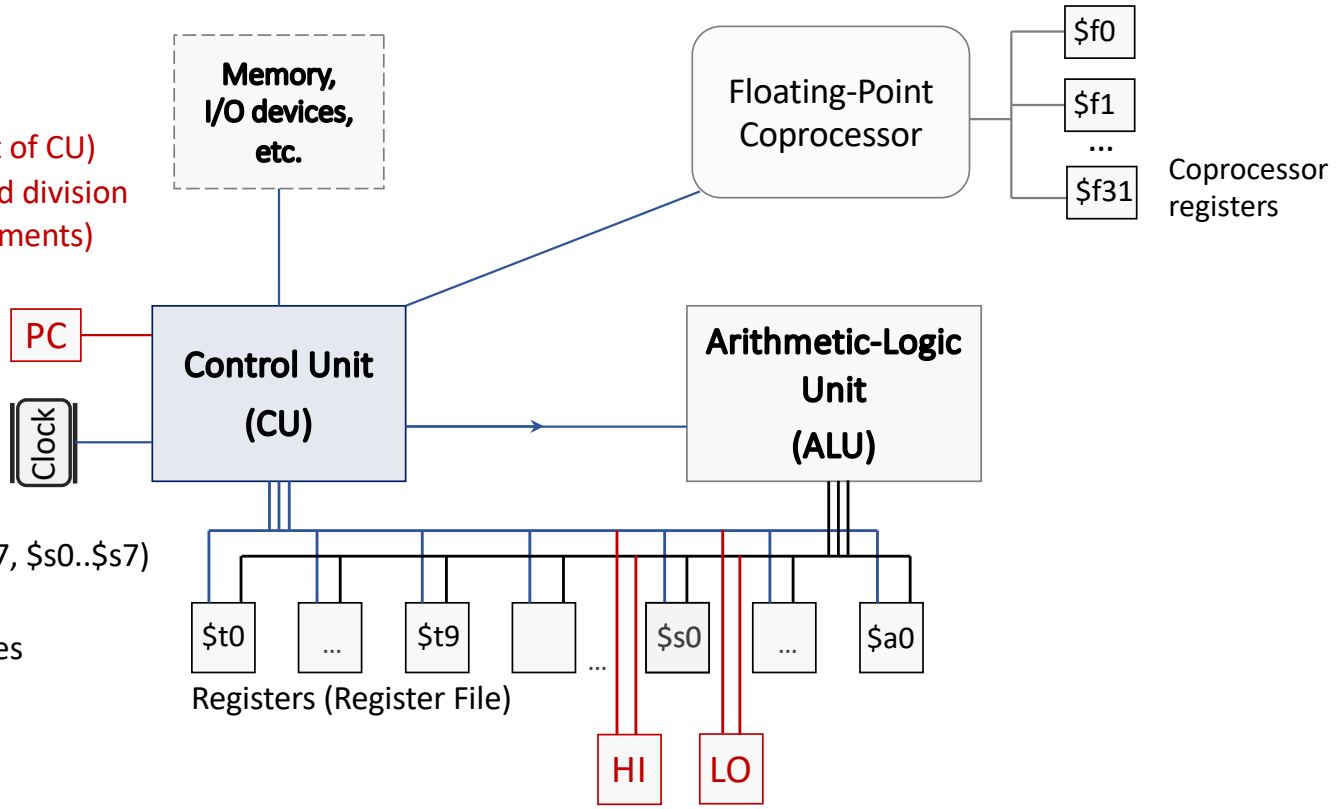
"Register spilling":
If the number of live variables exceeds the number of available registers, then the compiler spills some variables from registers into memory

MIPS Processor Architecture

MIPS is a RISC architecture

32 registers for floating-point instructions:
- Named by $f0..$f31
- Directly addressable
- Reserved for different purposes

3 special-purpose registers:
- PC – program counter (a part of CU)
- HI, LO – for multiplication and division instructions (for integer arguments)



32 general-purpose registers:
- Have their names (like $t0..$t7, $s0..$s7)
- Directly addressable
- Reserved for different purposes

Data is retrived from these registers by using special functions mfhi and mflo

"Register spilling":
If the number of live variables exceeds the number of available registers, then the compiler spills some variables from registers into memory

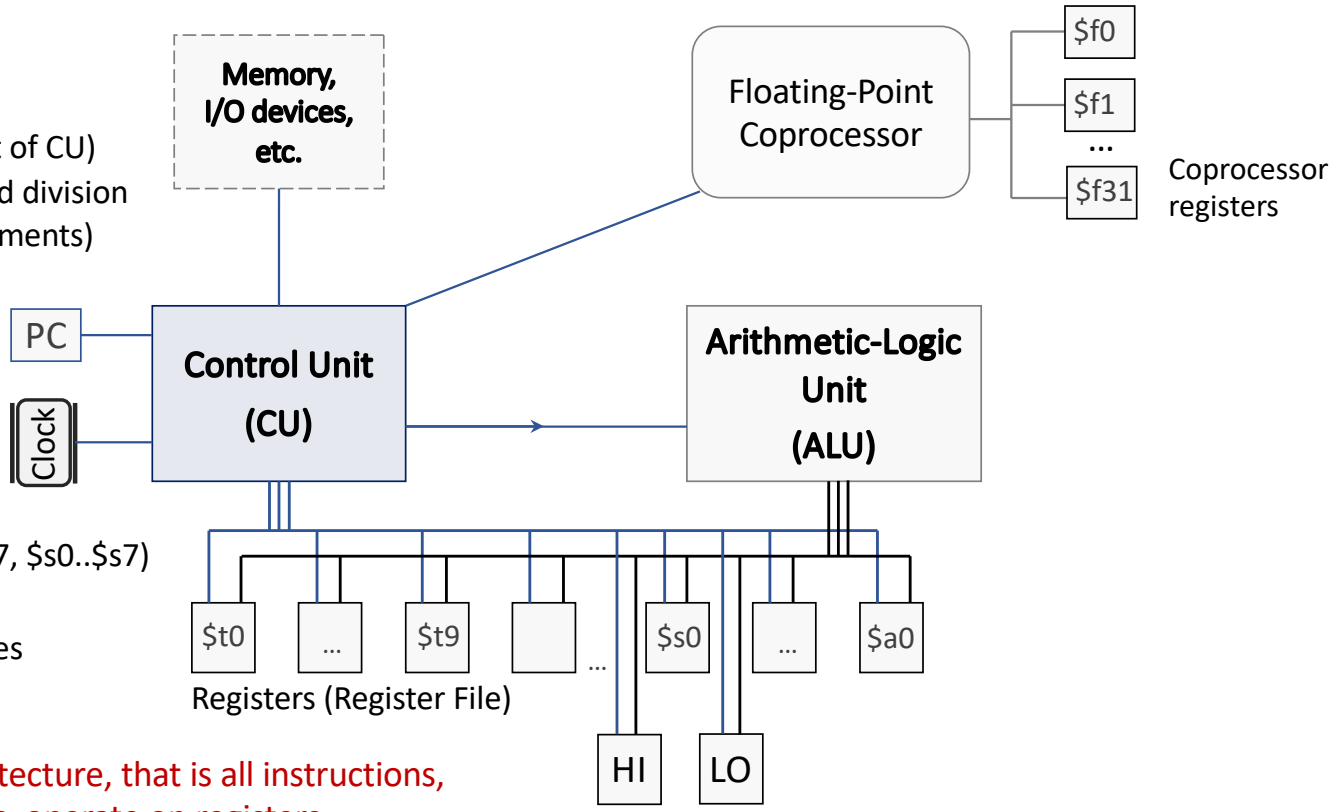
MIPS Processor Architecture

MIPS is a RISC architecture

3 special-purpose registers:
- PC – program counter (a part of CU)
- HI, LO – for multiplication and division instructions (for integer arguments)

32 general-purpose registers:
- Have their names (like $t0..$t7, $s0..$s7)
- Directly addressable
- Reserved for different purposes

32 registers for floating-point instructions:
- Named by $f0..$f31
- Directly addressable
- Reserved for different purposes



MIPS is a load/store architecture, that is all instructions, except for memory access, operate on registers

Data is retrived from these registers by using special functions mfhi and mflo

"Register spilling":
If the number of live variables exceeds the number of available registers, then the compiler spills some variables from registers into memory

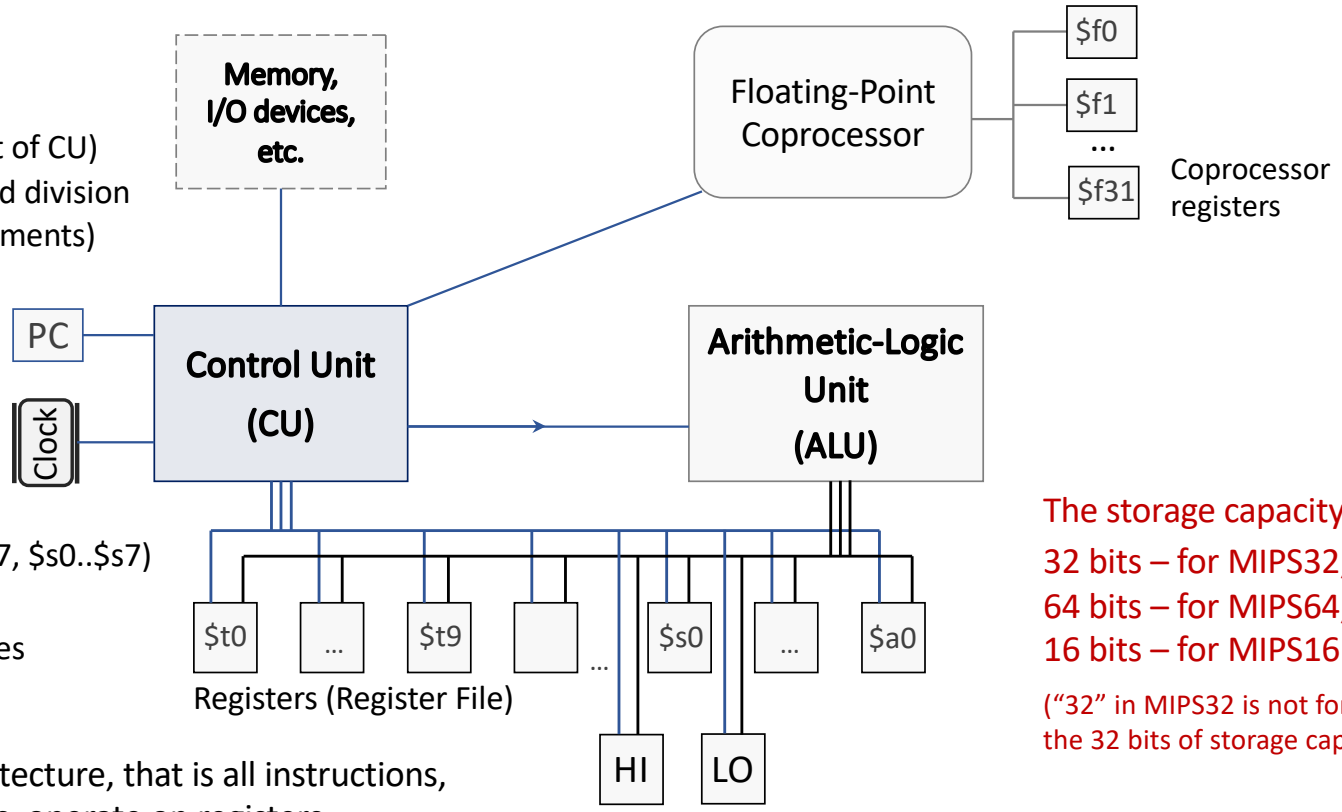
MIPS Processor Architecture

MIPS is a RISC architecture

3 special-purpose registers:
- PC – program counter (a part of CU)
- HI, LO – for multiplication and division instructions (for integer arguments)

32 general-purpose registers:
- Have their names (like $t0..$t7, $s0..$s7)
- Directly addressable
- Reserved for different purposes

32 registers for floating-point instructions:
- Named by $f0..$f31
- Directly addressable
- Reserved for different purposes



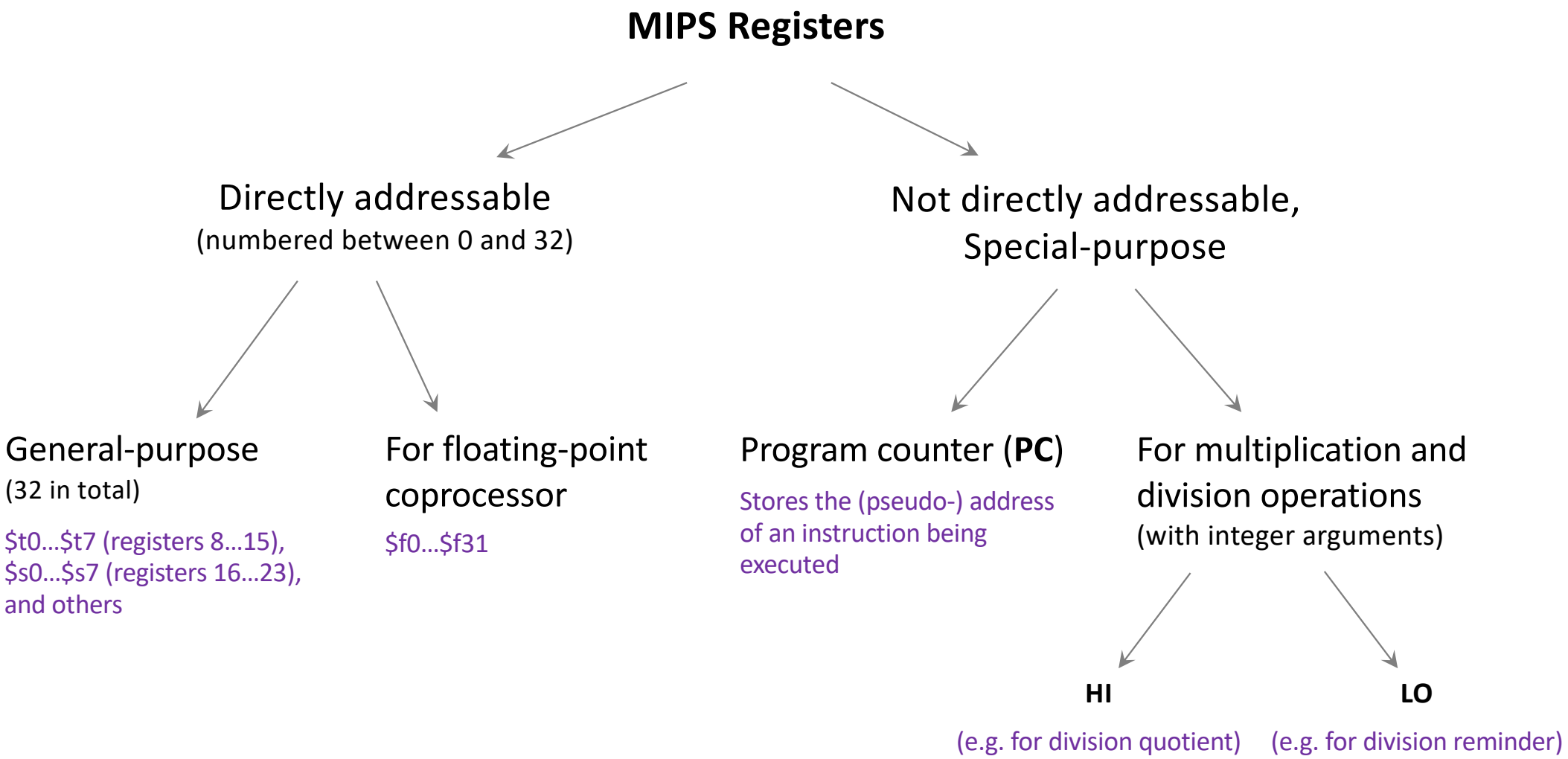
MIPS is a load/store architecture, that is all instructions, except for memory access, operate on registers

Data is retrived from these registers by using special functions mfhi and mflo

The storage capacity of a register varies:
32 bits – for MIPS32;
64 bits – for MIPS64;
16 bits – for MIPS16

("32" in MIPS32 is not for 32 registers, but for the 32 bits of storage capacity for every register)

"Register spilling":
If the number of live variables exceeds the number of available registers, then the compiler spills some variables from registers into memory


MIPS Registers
Directly addressable
(numbered between 0 and 32)
Not directly addressable,
Special-purpose
General-purpose
(32 in total)
$t0…$t7 (registers 8…15),
$s0…$s7 (registers 16…23),
and others
For floating-point
coprocessor
$f0…$f31
Program counter (PC)
Stores the (pseudo-) address
of an instruction being
executed
For multiplication and
division operations
(with integer arguments)
HI
(e.g. for division quotient)
LO
(e.g. for division reminder)

MIPS Instruction Set

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1 = $s2 + 20`	Used to add constants

The second argument is a constant value,
No need to load its value from the register (can be used "immediately")

Advantage: significantly faster and consumes less power, as compared to "*add*"

MIPS Instruction Set

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory

Units of data used:

1 word = 32 bits (equals to a MIPS instruction length)
1 halfword = 16 bits
1 byte = 8 bits

MIPS Instruction Set

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits

MIPS Instruction Set

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant

A fast way to multiply or divide by 2^n,
where n is the number of positions for left or right shifting

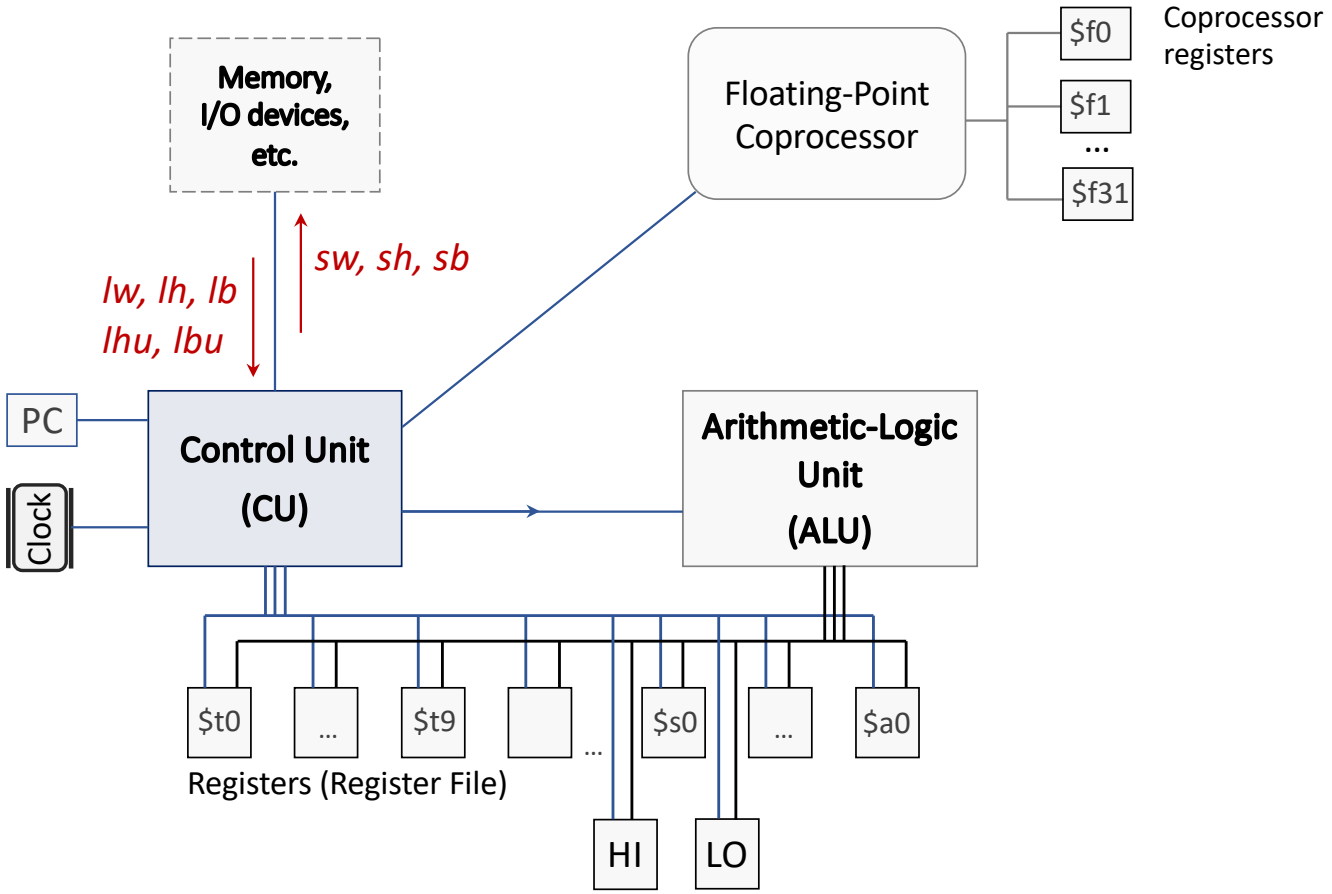
MIPS Instruction Set

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned

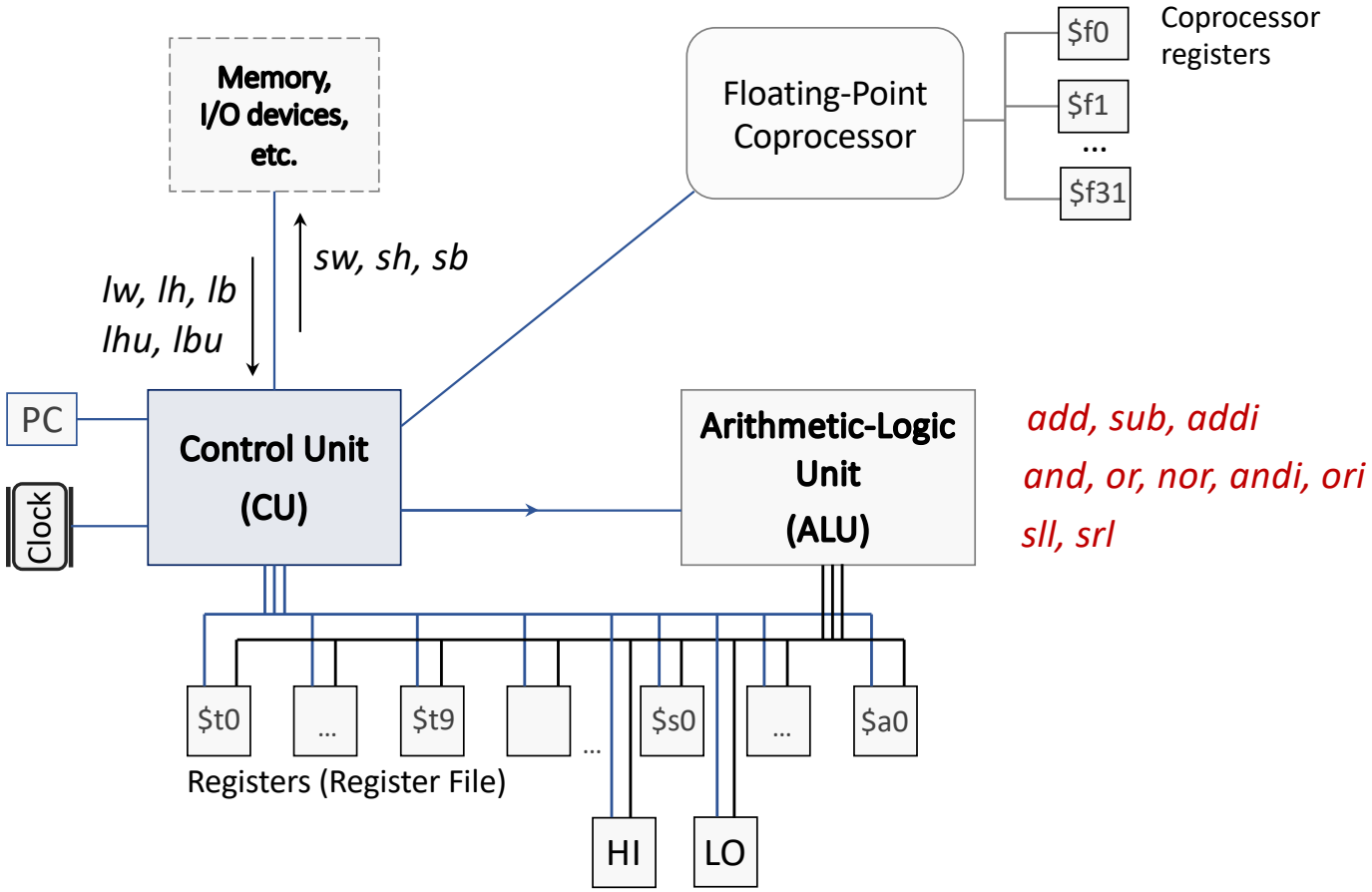
MIPS Instruction Set

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

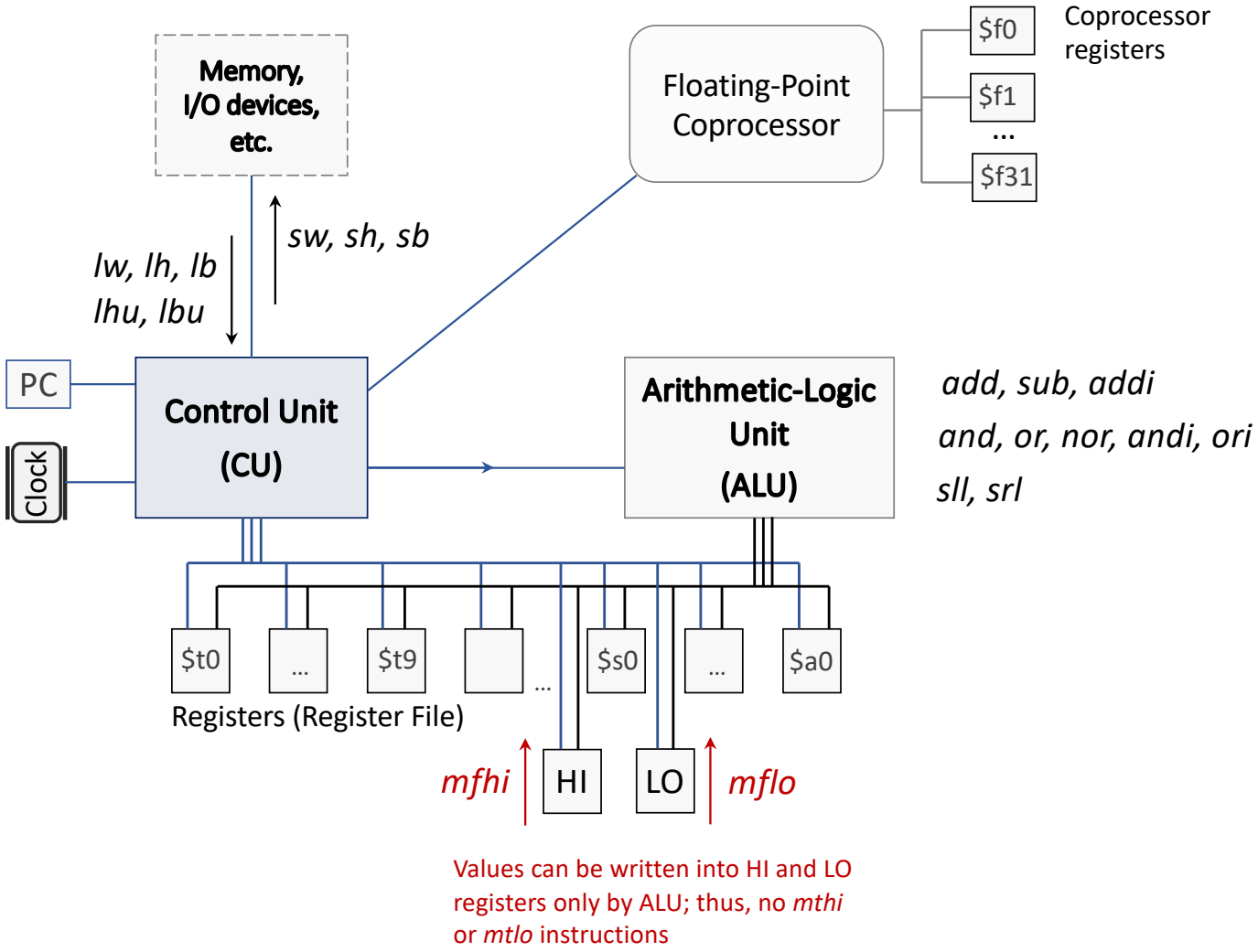
Sample MIPS Instructions



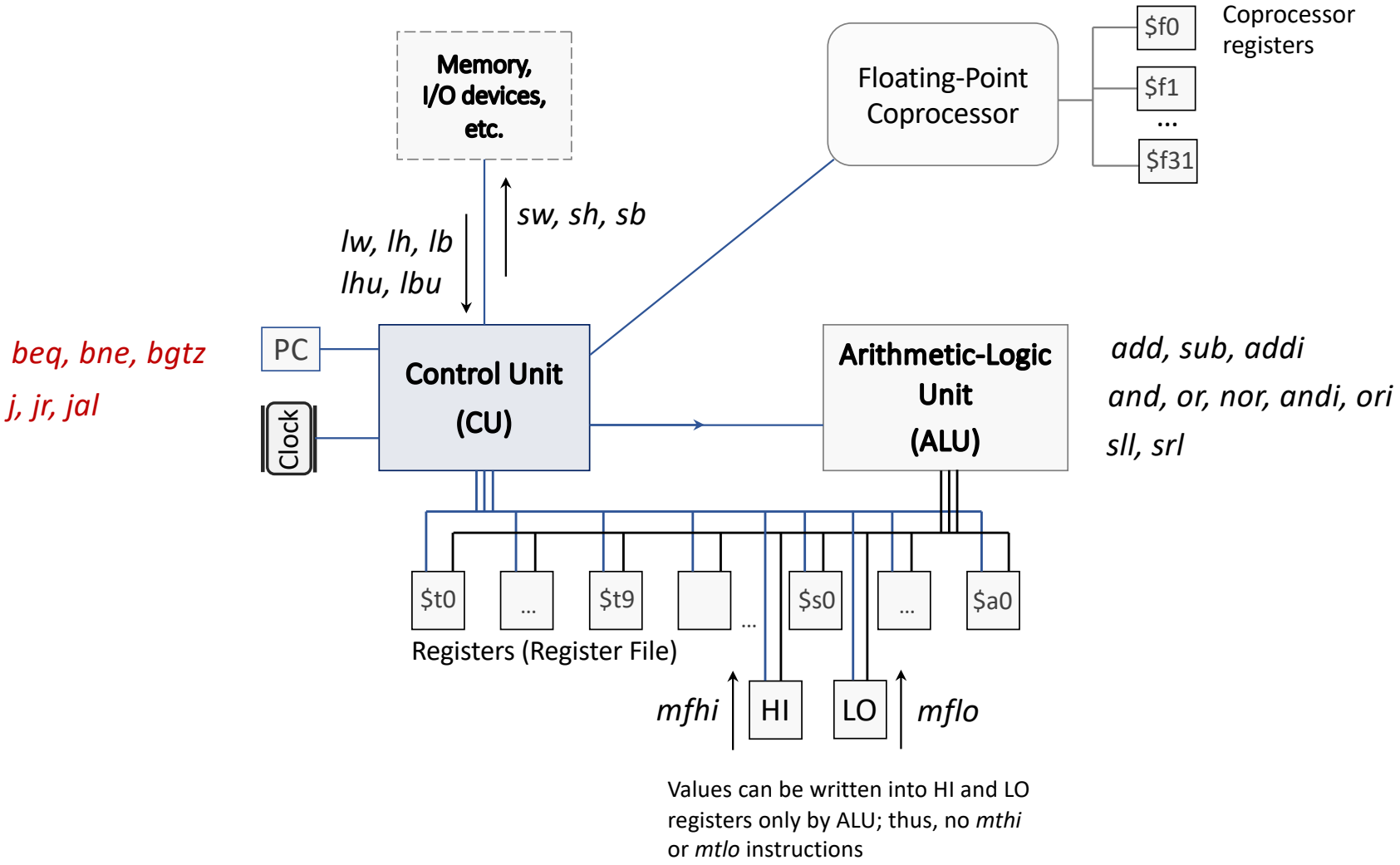
Sample MIPS Instructions



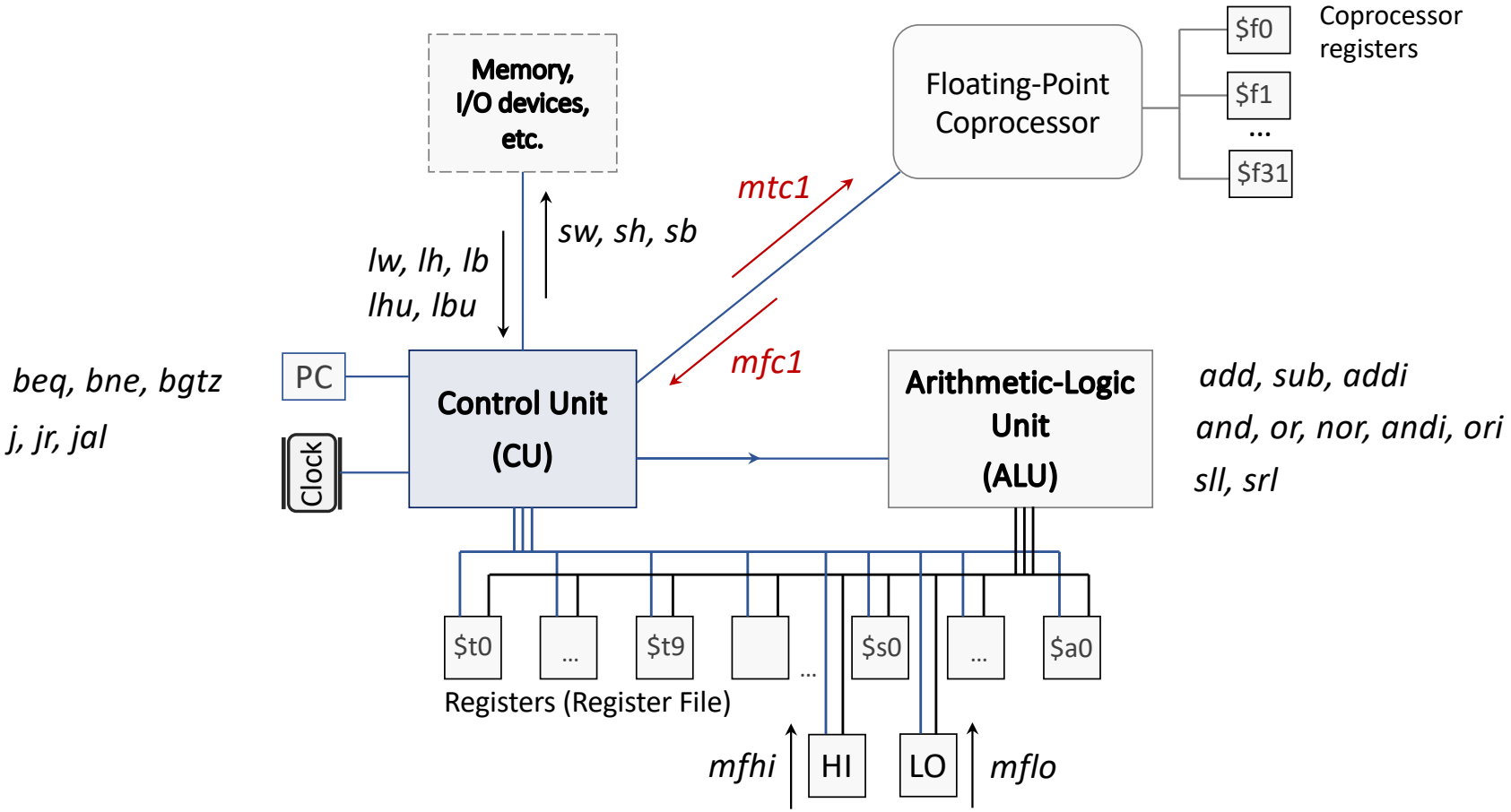
Sample MIPS Instructions



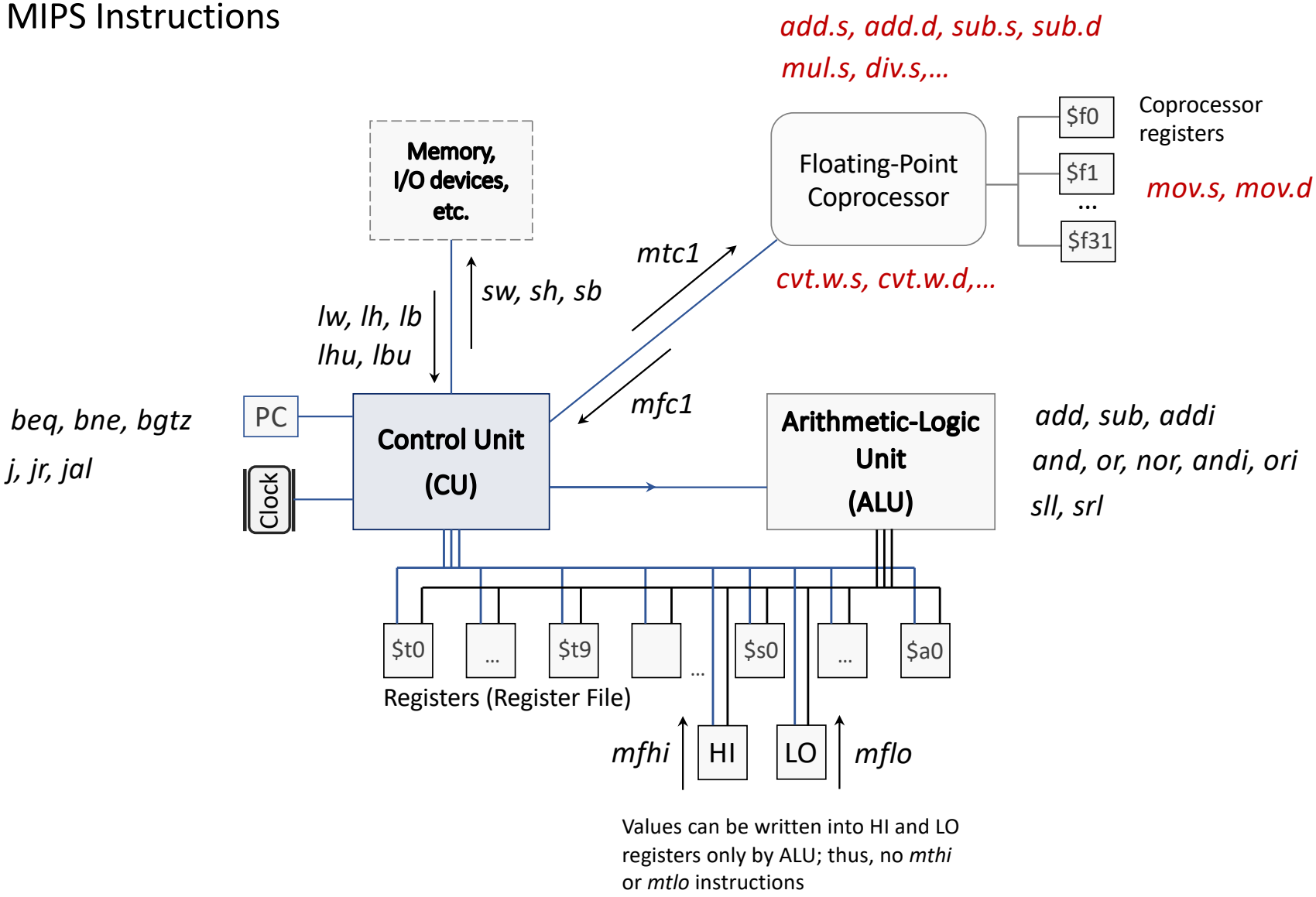
Sample MIPS Instructions



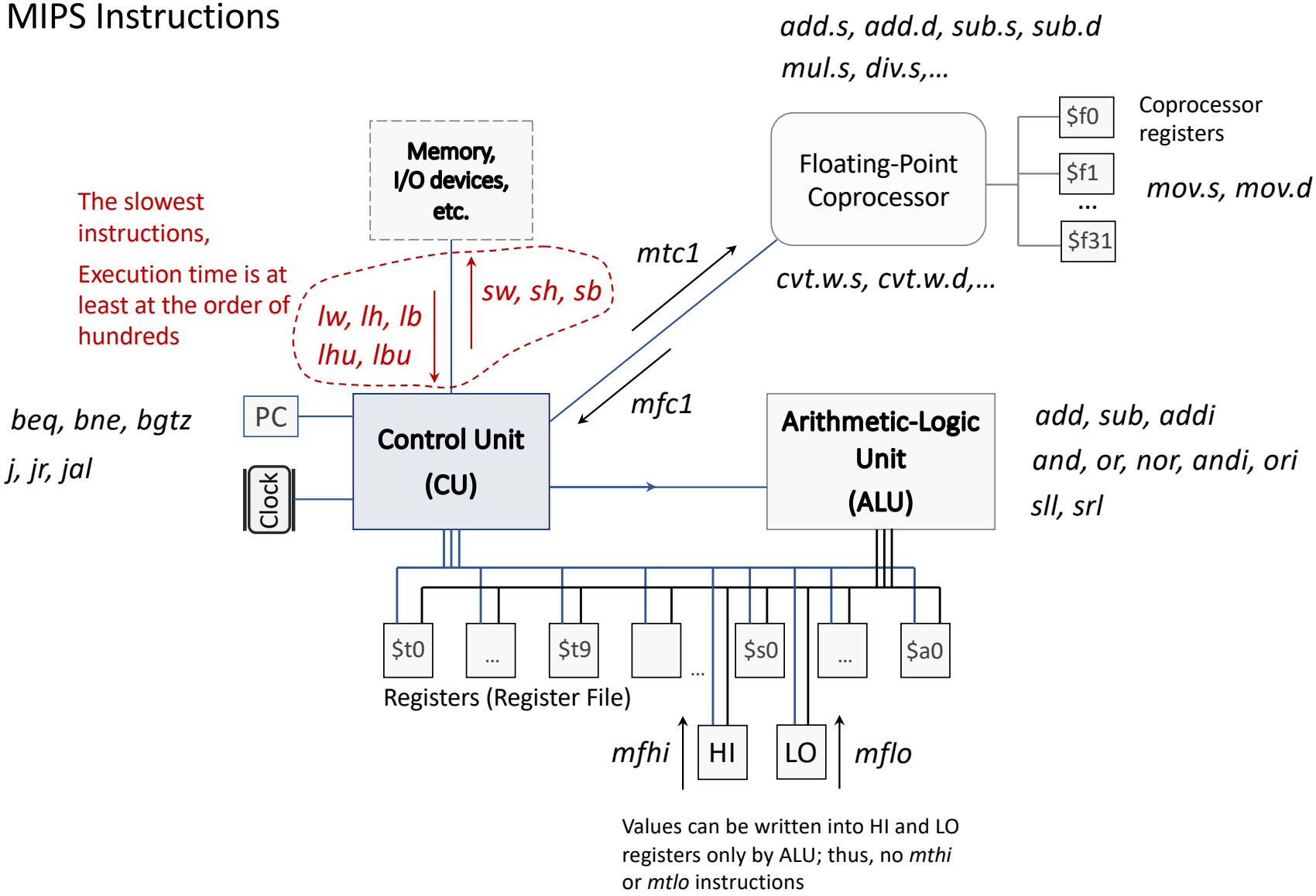
Sample MIPS Instructions



Sample MIPS Instructions



Sample MIPS Instructions



Sample MIPS Instructions