CS 211: Intro to Computer Architecture

9.2: RISC-V Assembly

Minesh Patel

Spring 2025 – Thursday 27 March

Announcements

- Assignments
  - **PA3**: due **Friday @ 23:59**
  - **Extra Credit:** replaces WA6, due in **two weeks**

Agenda

- **Inside a Processor**
  - **Lots of Pictures**
  - Inside a Processor Core
  - Cache and Register Memory

- RISC-V Assembly
  - Instructions and Opcodes
  - Immediate Values
  - Function Arguments

Recap: Processor and Memory

Let's take a look inside

Program

(instructions for number manipulation)

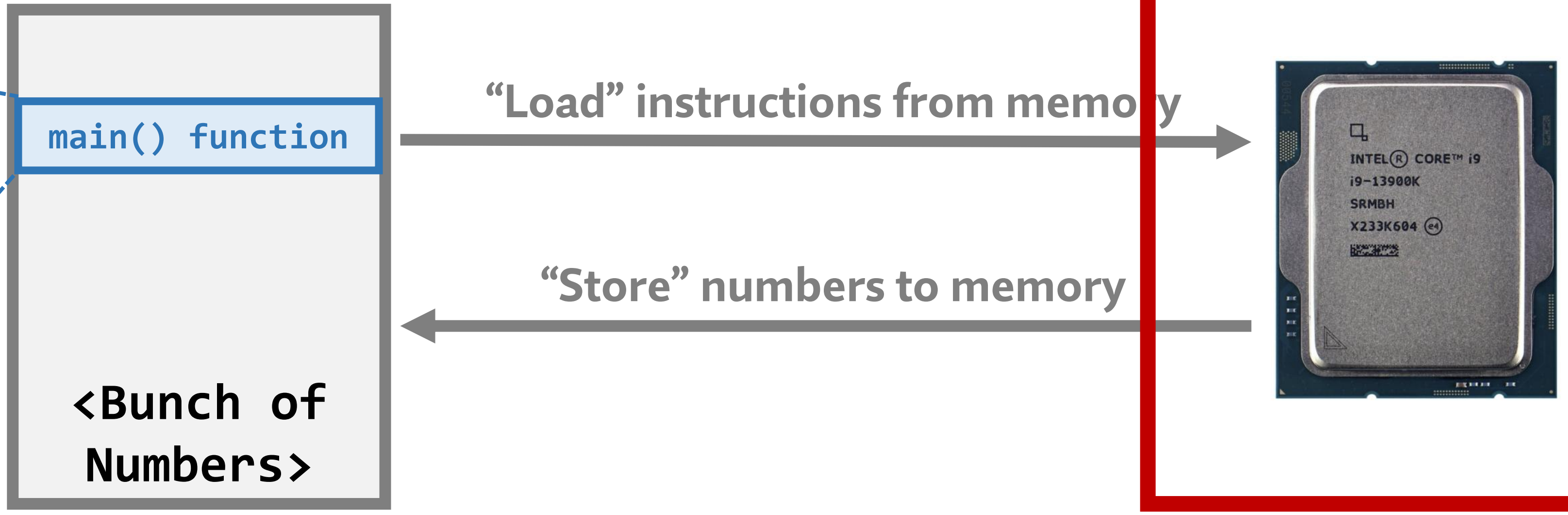
```
main:
        addi sp,sp,-32
        sd ra,24(sp)
        sd s0,16(sp)
        addi s0,sp,32
        mv a5,a0
        sd a1,-32(s0)
        sw a5,-20(s0)
        lla a0,.LC0
        call puts@plt
        li a5,0
        mv a0,a5
        ld ra,24(sp)
        ld s0,16(sp)
        addi sp,sp,32
        jr ra
```

Memory

(numbers that represent various things)

Processor

(executes instructions)



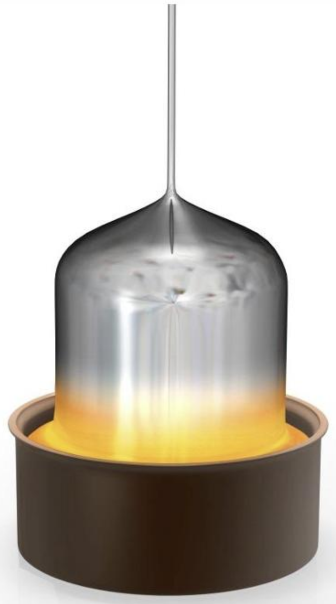
“From Sand to Silicon”

Intel, “Making of a Chip Illustrations,” January 2012.

Sand / Ingot

Sand

Silicon is the second most abundant element in the earth’s crust. Common sand has a high percentage of silicon. Silicon – the starting material for computer chips – is a semiconductor, meaning that it can be readily turned into an excellent conductor or an insulator of electricity, by the introduction of minor amounts of impurities.



Melted Silicon –

scale: wafer level (~300mm / 12 inch)

In order to be used for computer chips, silicon must be purified so there is less than one alien atom per billion. It is pulled from a melted state to form a solid which is a single, continuous and unbroken crystal lattice in the shape of a cylinder, known as an ingot.

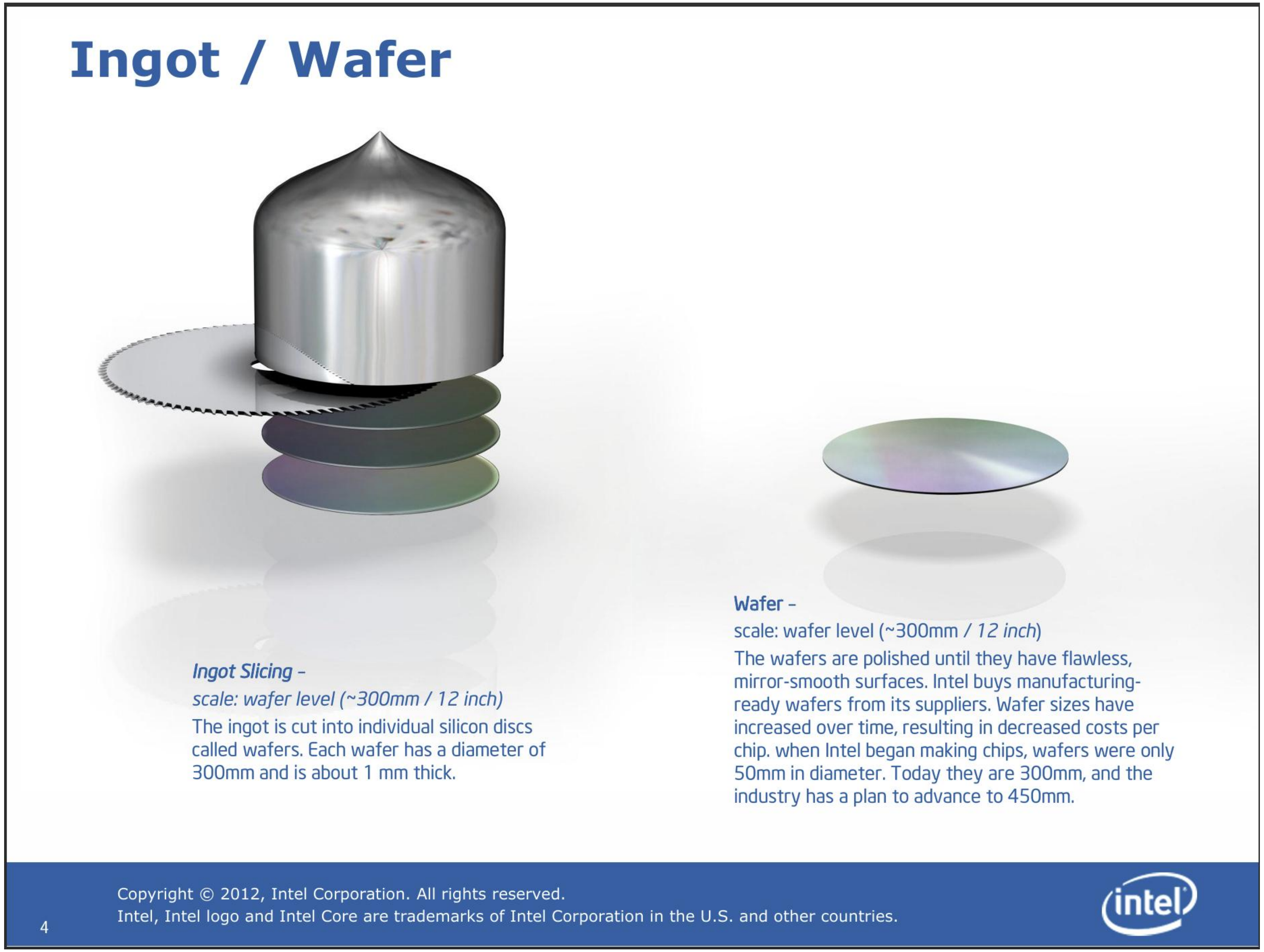
Monocrystalline Silicon Ingot –

scale: wafer level (~300mm / 12 inch)

The ingot has a diameter of 300mm and weighs about 100 kg.

"From Sand to Silicon"

Intel, "Making of a Chip Illustrations," January 2012.

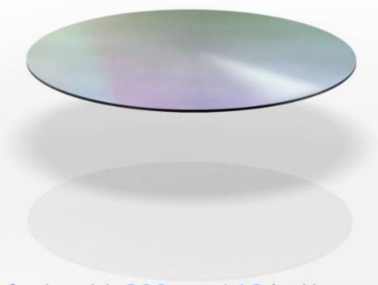


Ingot / Wafer

Ingot Slicing –

scale: wafer level (~300mm / 12 inch)

The ingot is cut into individual silicon discs called wafers. Each wafer has a diameter of 300mm and is about 1 mm thick.



Wafer –

scale: wafer level (~300mm / *12 inch*)

The wafers are polished until they have flawless, mirror-smooth surfaces. Intel buys manufacturing-ready wafers from its suppliers. Wafer sizes have increased over time, resulting in decreased costs per chip. when Intel began making chips, wafers were only 50mm in diameter. Today they are 300mm, and the industry has a plan to advance to 450mm.

SiFive RISC-V Processor Wafer

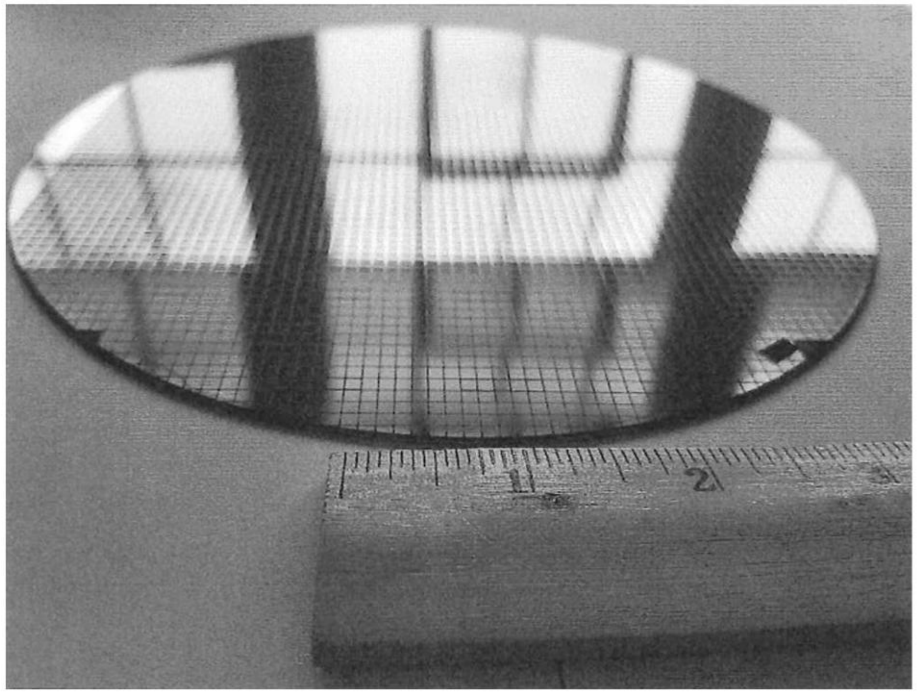
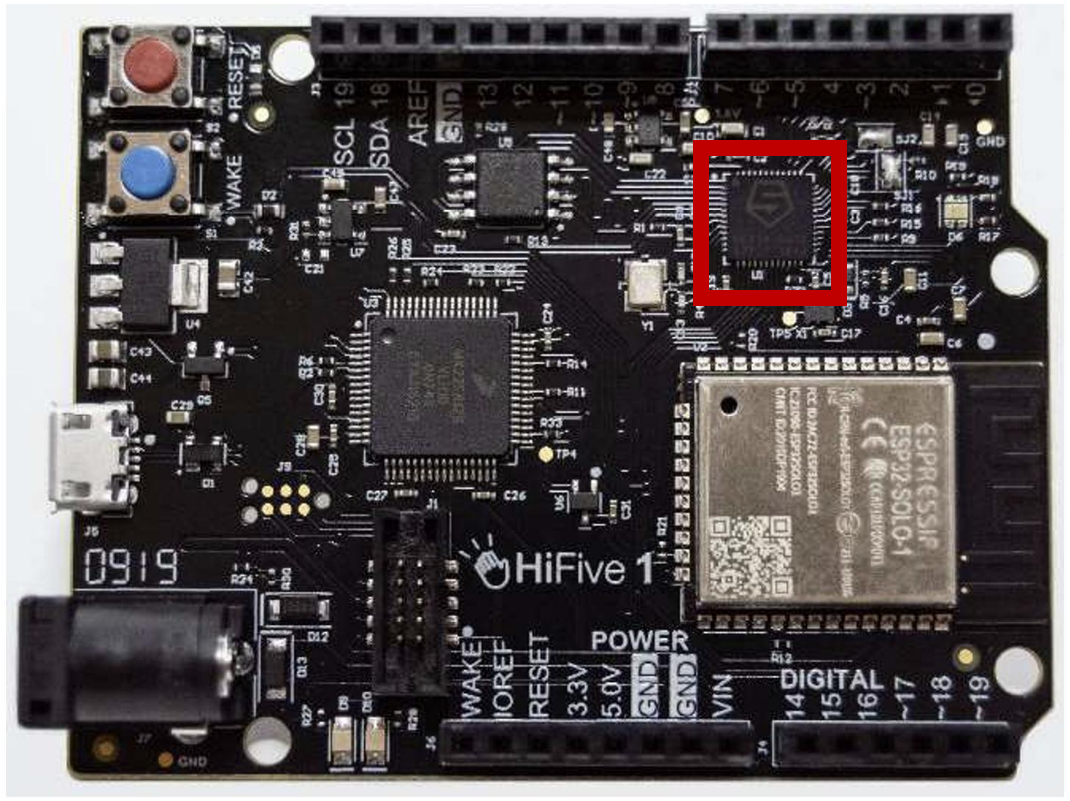


Figure 1.16 This 200 mm diameter wafer of RISC-V dies was designed by SiFive. It h
two types of RISC-V dies using an older, larger processing line. An FE310 die is 2.
mm × 2.72 mm and an SiFive test die that is 2.89 mm × 2.72 mm. The wafer contai
1846 of the former and 1866 of the latter, totaling 3712 chips.

Hennessy and Patterson, “Computer Architecture” 6/E.

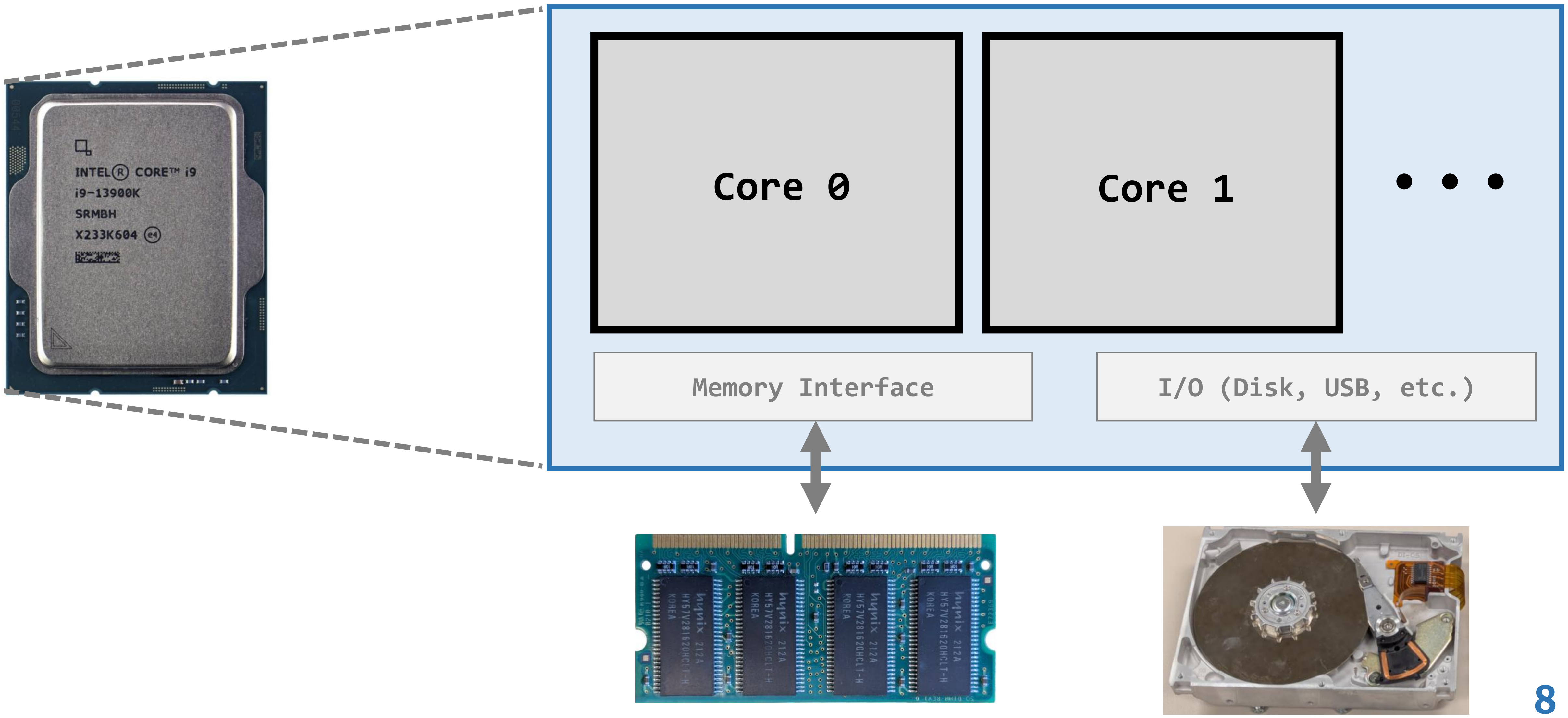
HiFive 1 Rev B01



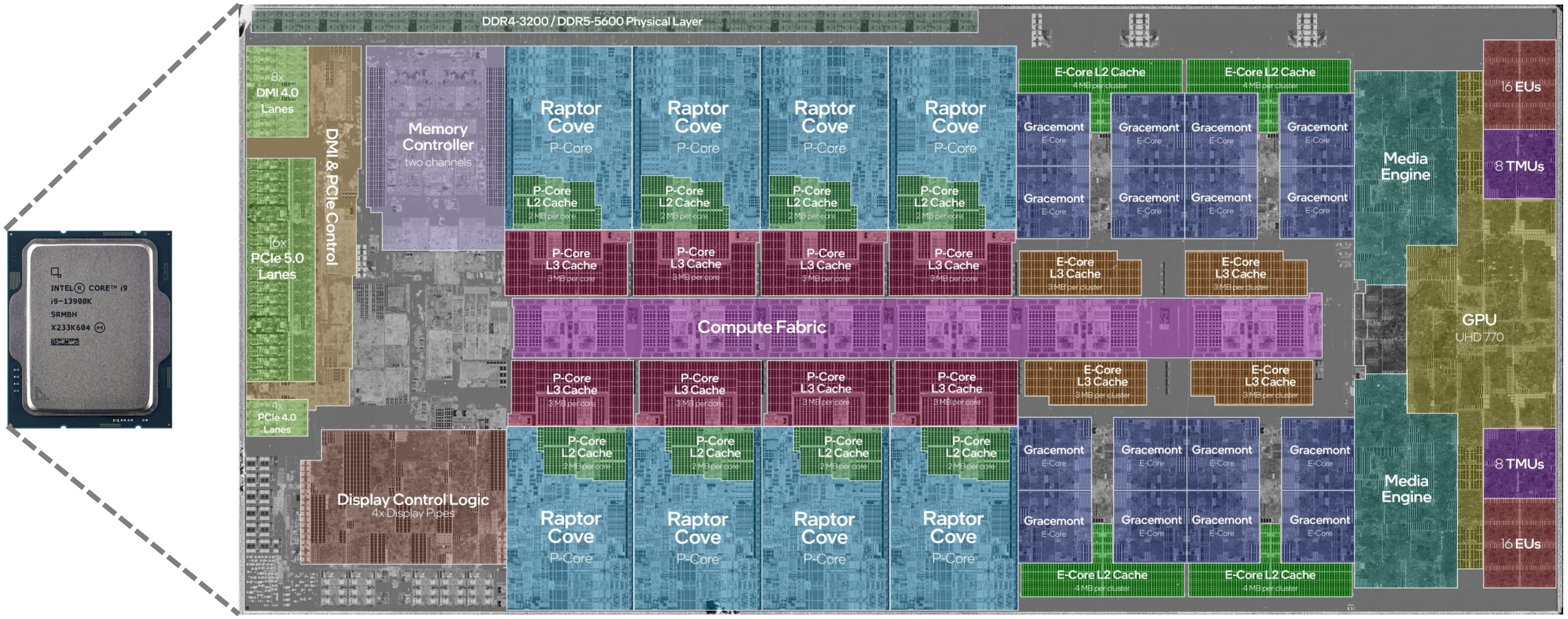
SiFive, “HiFive1 Rev B Schematics,” 2021.

Mental Model: What's Inside a Processor?

CS 211 Abstraction of a Processor

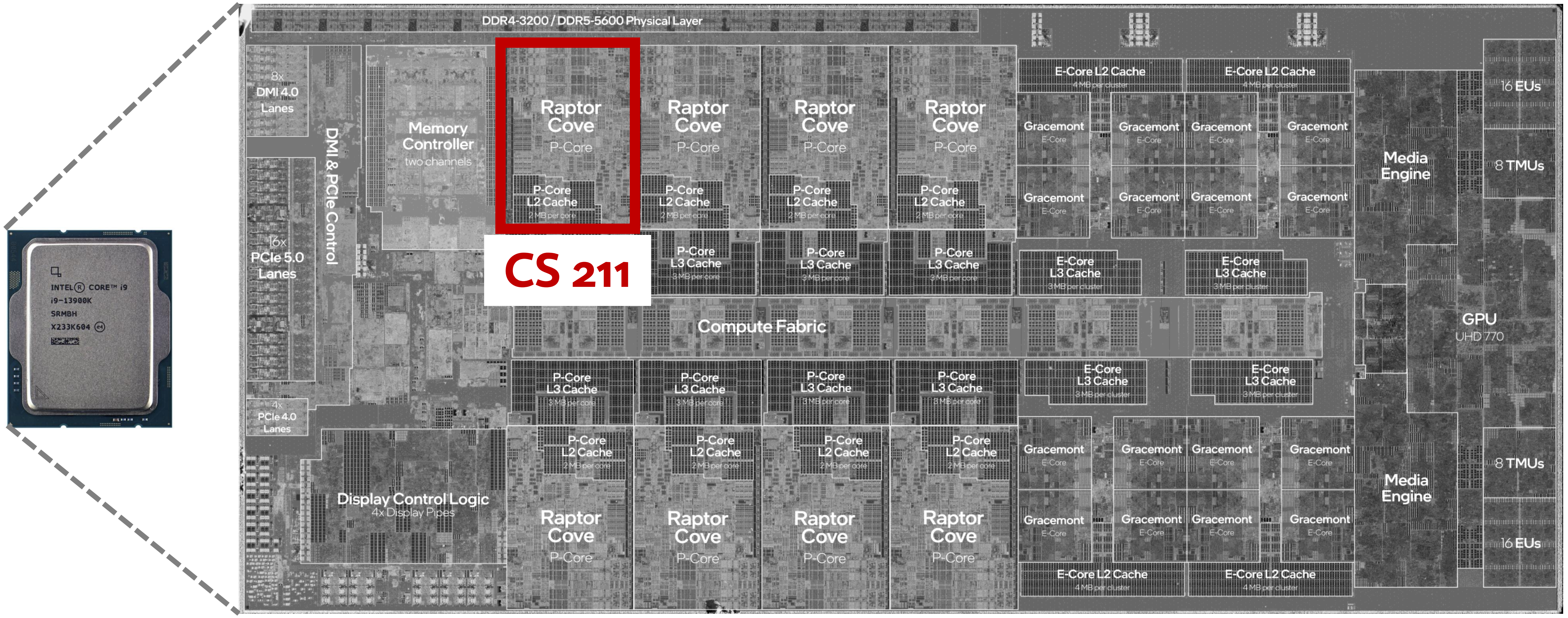


Inside an Intel i9-13900K



https://upload.wikimedia.org/wikipedia/commons/a/a4/Intel_Core_i9-13900K_Labelled_Die_Shot.jpg

Inside an Intel i9-13900K

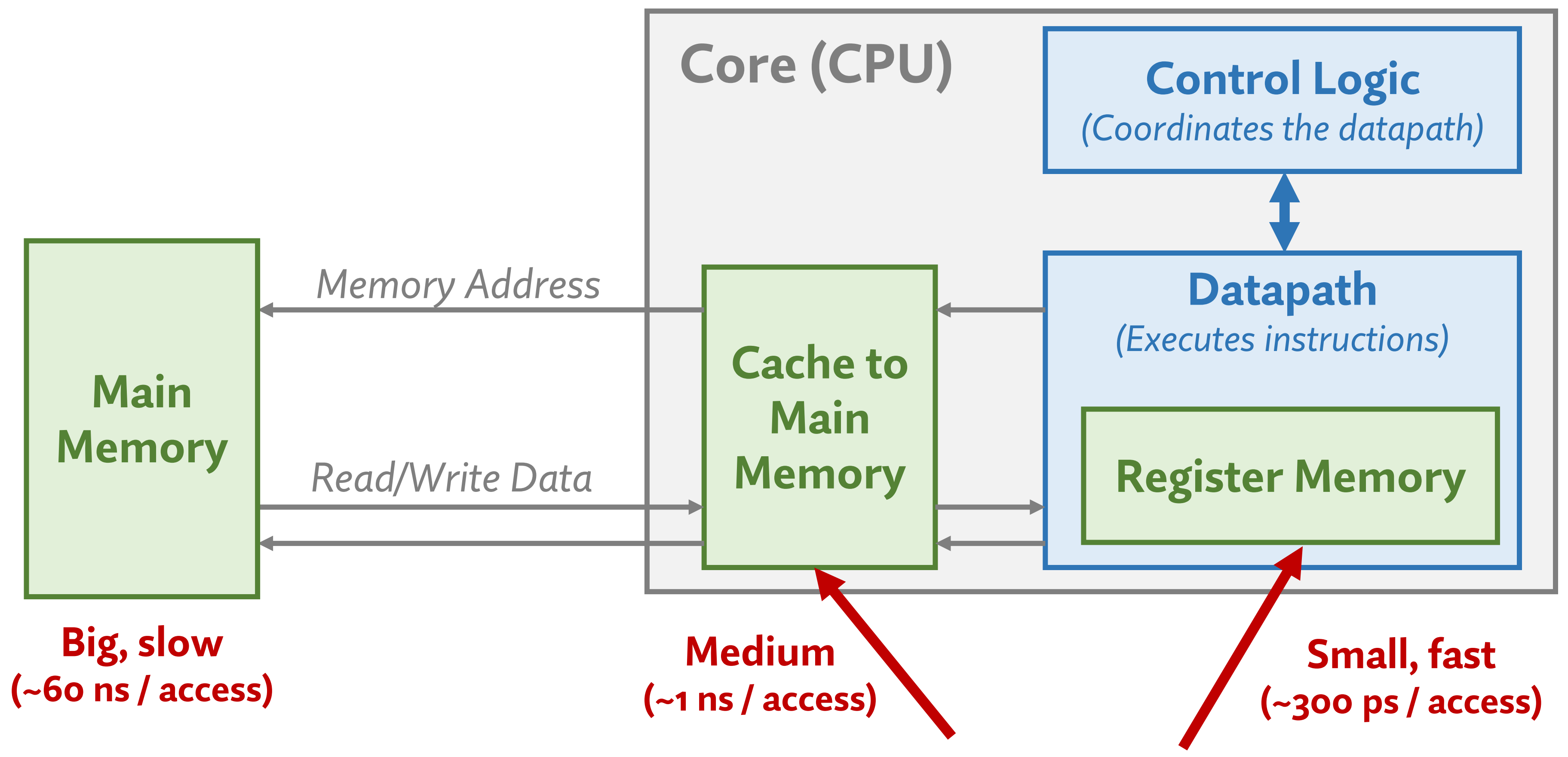


Agenda

- Inside a Processor
  - Lots of Pictures
  - **Inside a Processor Core**
  - Cache and Register Memory

- RISC-V Assembly
  - Instructions and Opcodes
  - Immediate Values
  - Function Arguments

Inside a Processor Core: Overview



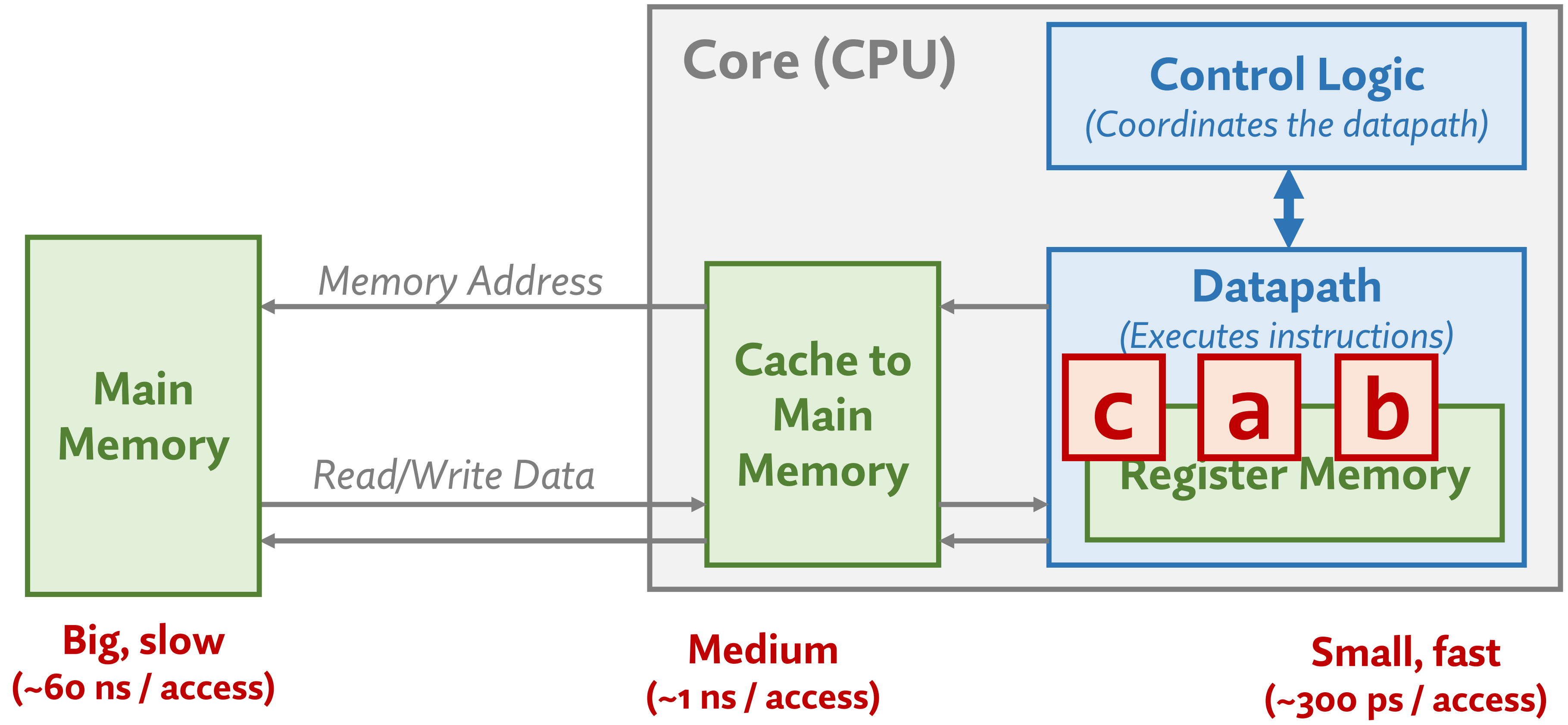
Two new memories!

- Both are performance optimizations
  - Beyond the C language abstraction
  - **Much** faster to access than main memory

Performance: “c = a + b” In Register Memory

C Statement

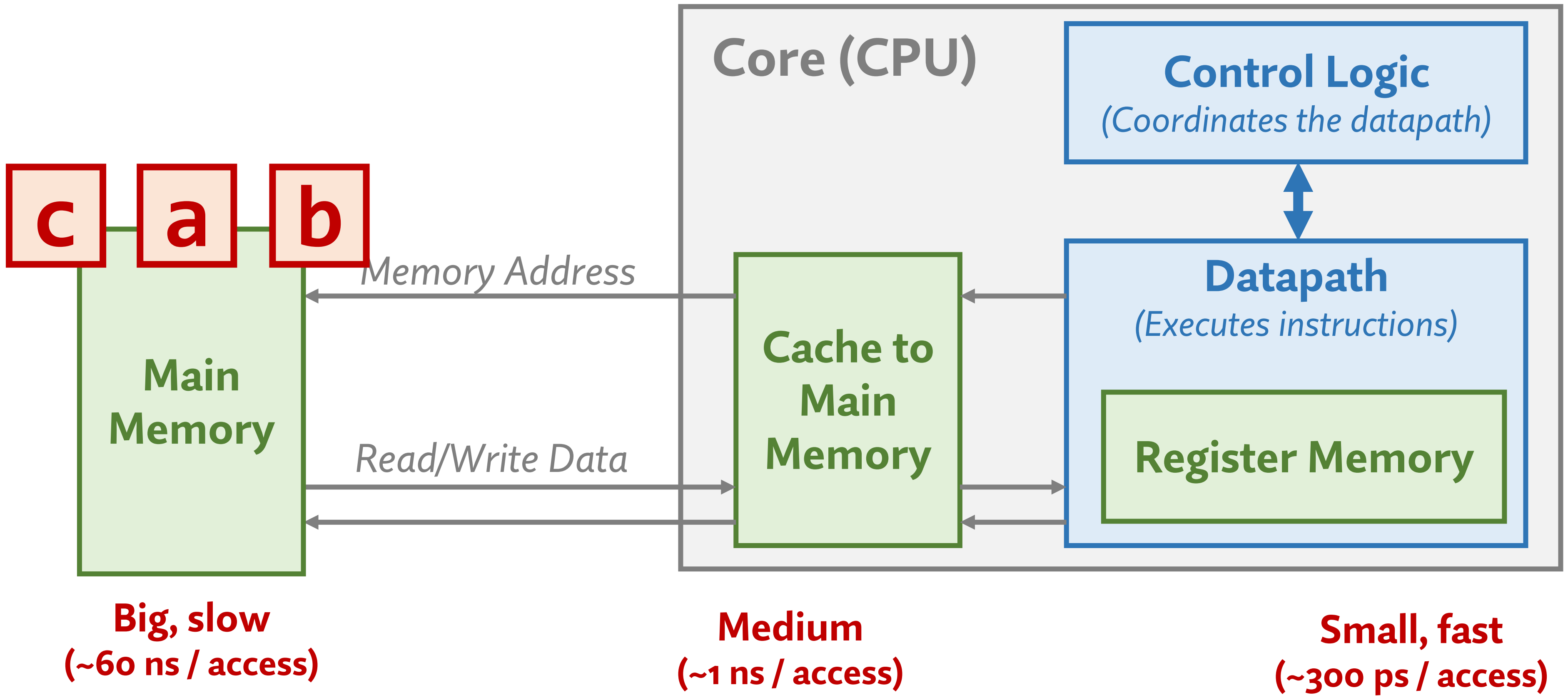
```
c = a + b;
```



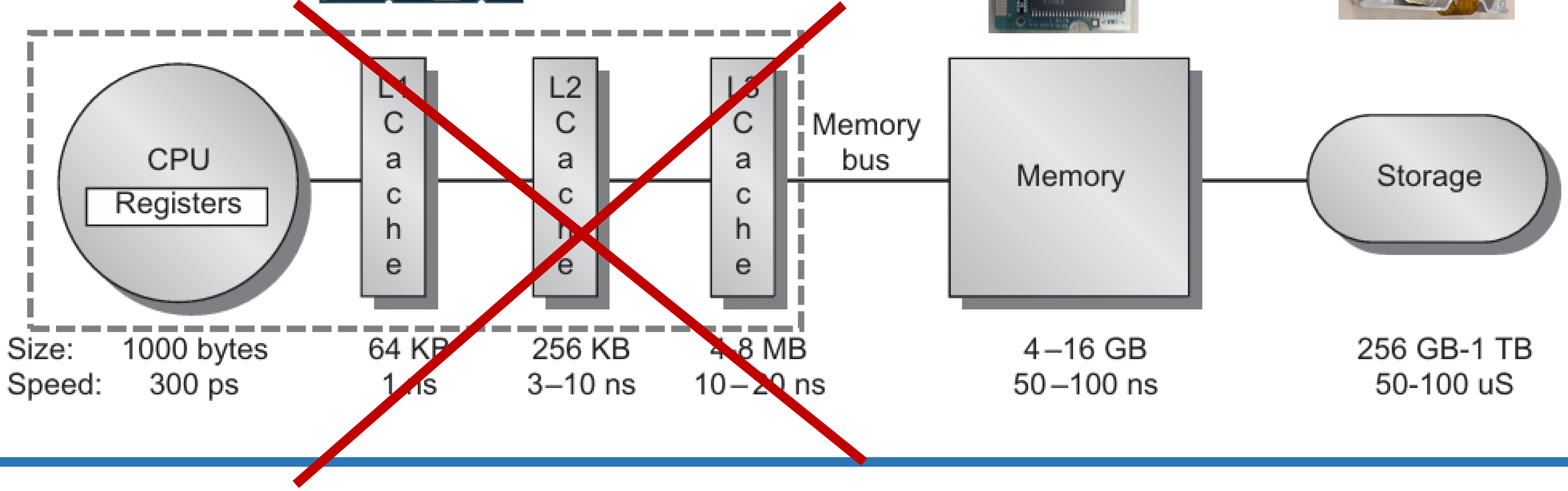
Performance: “c = a + b” In Main Memory

C Statement

```
c = a + b;
```



Aside: Memory Hierarchy



Smaller, fast

Larger, slower

We'll ignore caches for a few weeks

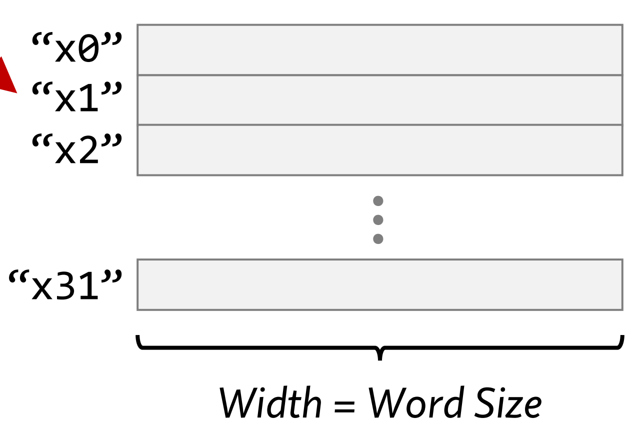
Hennessy and Patterson, "Computer Architecture" 6/E.

Registers vs. Main Memory

Registers

Array of "words"

'x' is just a name: does NOT refer to hex values ☹

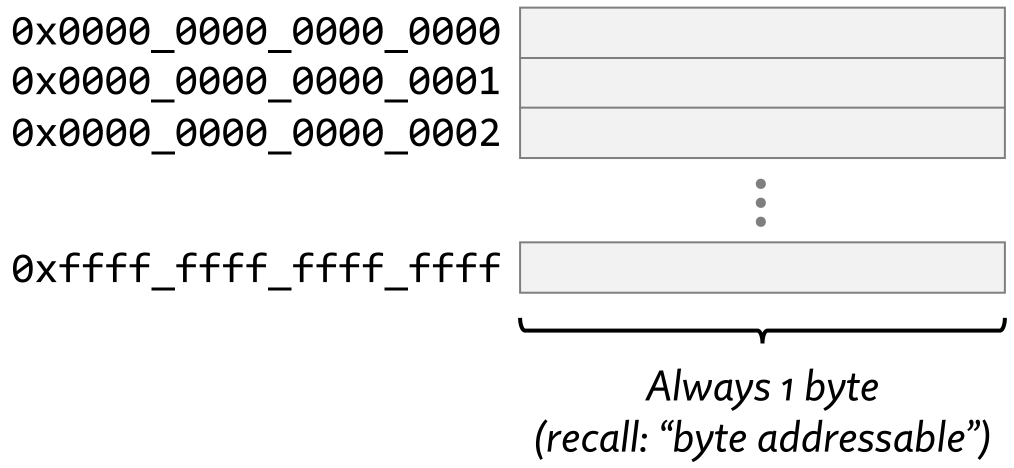


RISC-V Version	Word Size
RV32 (32-bit)	32-bits
RV64 (64-bit)	64-bits
RV128 (128-bit)	128-bits

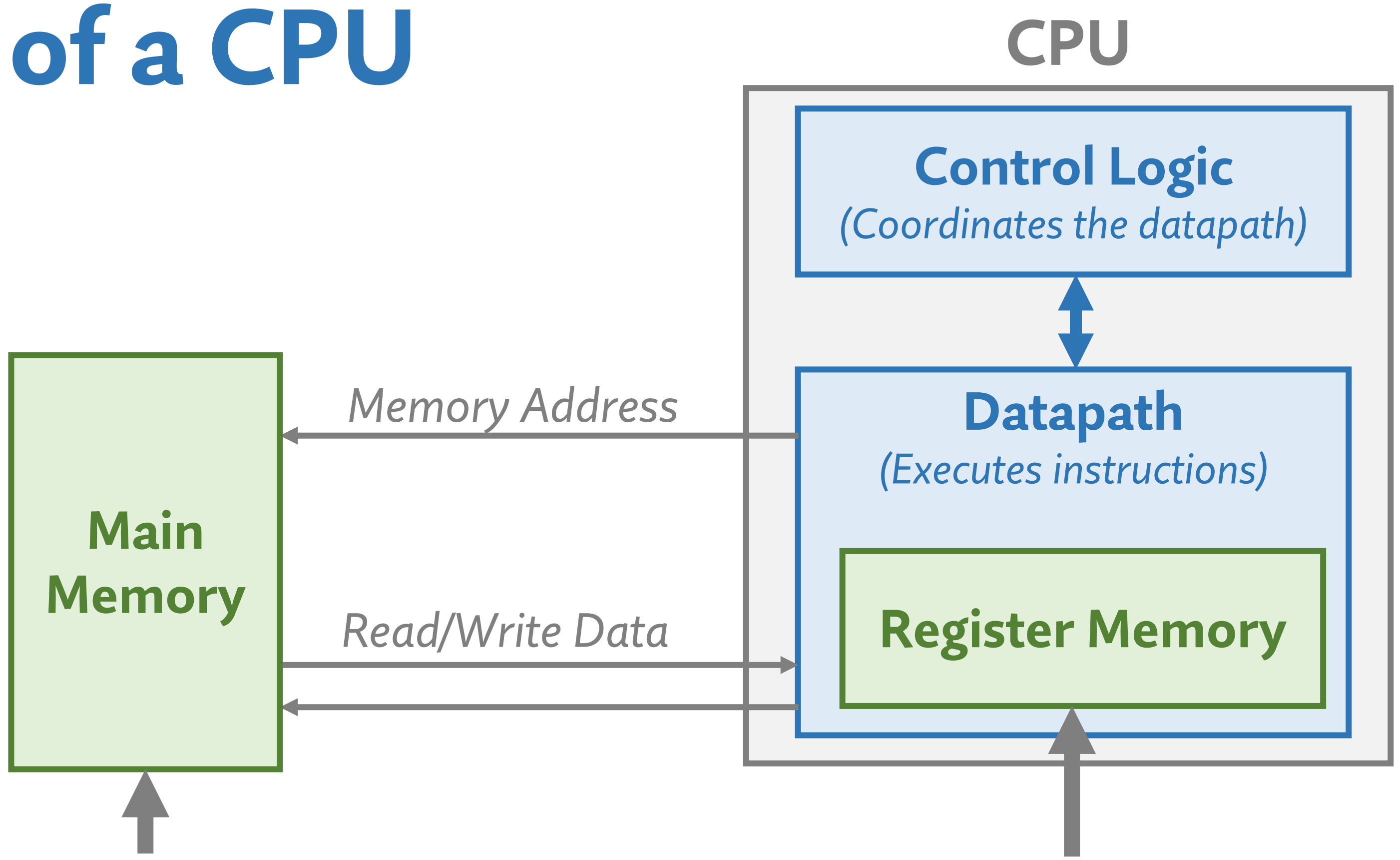
This Course

Main Memory

Array of bytes



Final Model of a CPU



Special “**load**” and “**store**” instructions access main memory

Assembly instructions operate on **registers**

```
.func
    lw x10, 0(x11) # x10 = *(x11 + 0)
    sw x10, 8(x11) # *(x11 + 8) = x10
```

```
.func
    add x10, x11, x12  # x10 = x11 + x12
```

Agenda

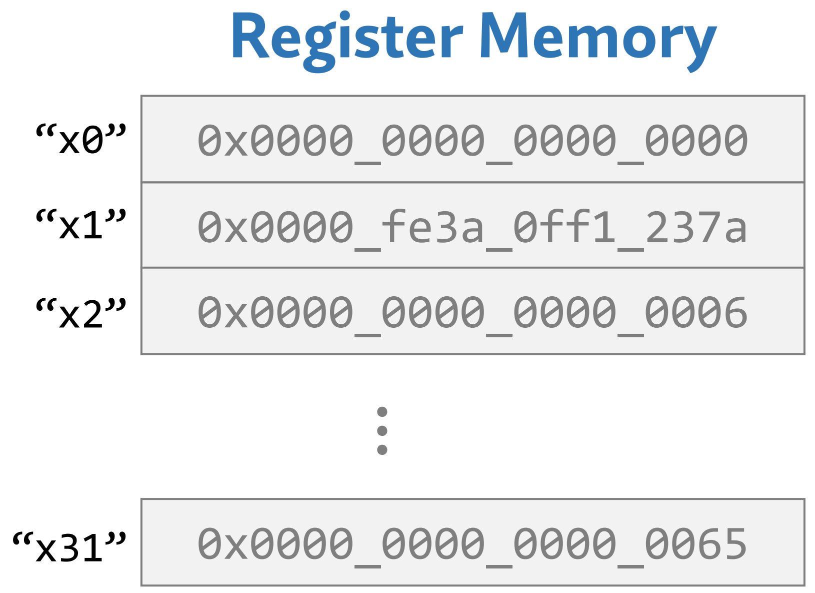
- Inside a Processor
  - Lots of Pictures
  - Inside a Processor Core
  - Cache and Register Memory
- **RISC-V Assembly**
  - Instructions and Opcodes
  - Immediate Values
  - Function Arguments

Assembly Has No Type System

- Every register is just a **collection of bits**

- Could represent….
  - A memory address
  - A register "address" (e.g., 6 = x6)
  - A two's complement integer
  - An unsigned integer
  - A character
  - …

- **Your job to decide + keep track**



Special Registers

- By convention, some registers are **reserved for specific uses**

Register	ABI Name	Description
`x0`	`zero`	Hard-wired zero
`x1`	`ra`	Return address
`x2`	`sp`	Stack pointer
`x3`	`gp`	Global pointer
`x4`	`tp`	Thread pointer
`x5–7`	`t0–2`	Temporaries
`x8`	`s0/fp`	Saved register/frame pointer
`x9`	`s1`	Saved register
`x10–11`	`a0–1`	Function arguments/return values
`x12–17`	`a2–7`	Function arguments
`x18–27`	`s2–11`	Saved registers
`x28–31`	`t3–6`	Temporaries

We will mostly talk about these

"RISC-V ABIs Specification," Version 1.1.

Agenda

- Inside a Processor
  - Lots of Pictures
  - Inside a Processor Core
  - Cache and Register Memory

- RISC-V Assembly
  - **Instructions and Opcodes**
  - Immediate Values
  - Function Arguments

Instruction Examples

- Assembly programs have **one instruction** per line of source code

```
opcode rd, rs1, rs2
```

The operation to perform
(e.g., add, bit shift, load/store)

Destination register

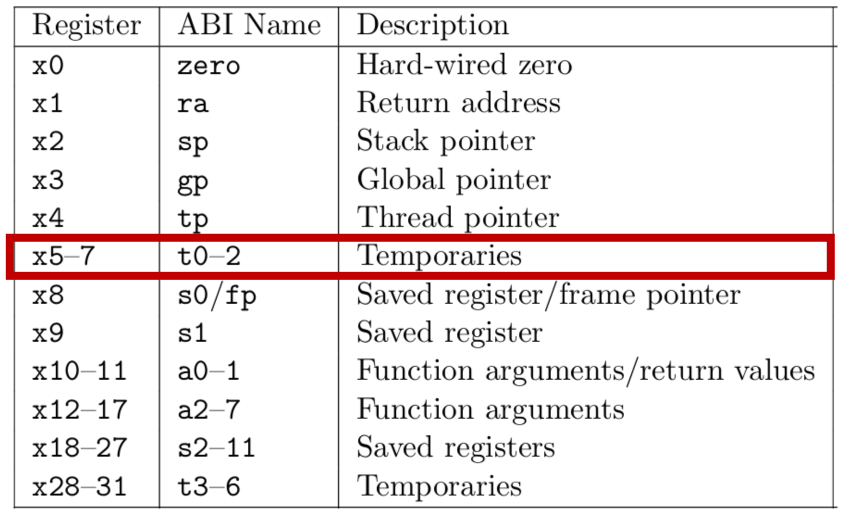
Source register(s)

```
add x5, x6, x7
```

x5 = x6 + x7

```
sub x5, x6, x7
```

x5 = x6 - x7



Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–7	t0–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

Comparing C and Assembly Code

“Register Allocation”

C Object	Register
`uint64_t a`	`x5`
`uint64_t b`	`x6`
`uint64_t c`	`x7`

C Statement

```
a = b + c;
```

Equivalent Assembly Code

```
add x5, x6, x7 # a = b + c
```

Python-style comments

Example Assembly Code

C Object	Register
`uint64_t a`	`x5`
`uint64_t b`	`x6`
`uint64_t c`	`x7`
`uint64_t d`	`x8`

C Statement(s)

Multiple instructions needed

Equivalent Assembly Code

```
a = b + c - d;
```

```
add x5, x6, x7 # a = b + c
sub x5, x5, x8 # a -= d
```

```
a += b + c - d;
```

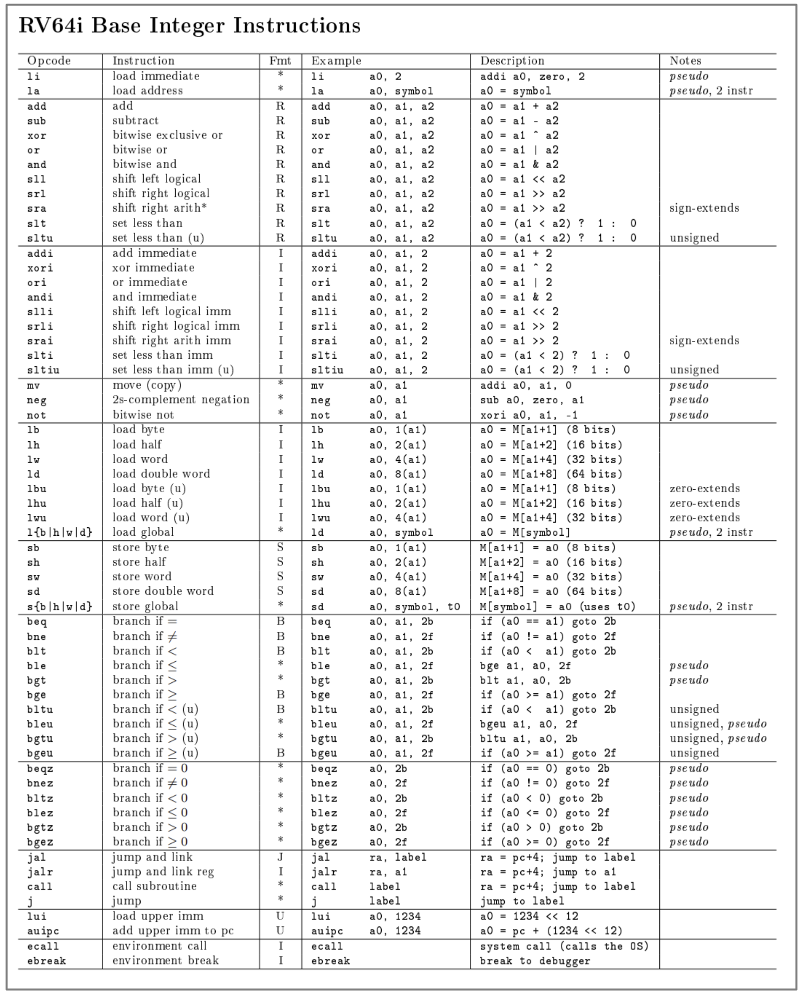
```
add x5, x5, x6 # a += b
add x5, x5, x7 # a += c
sub x5, x5, x8 # a -= d
```

Not necessarily unique

RV64i Reference Sheets

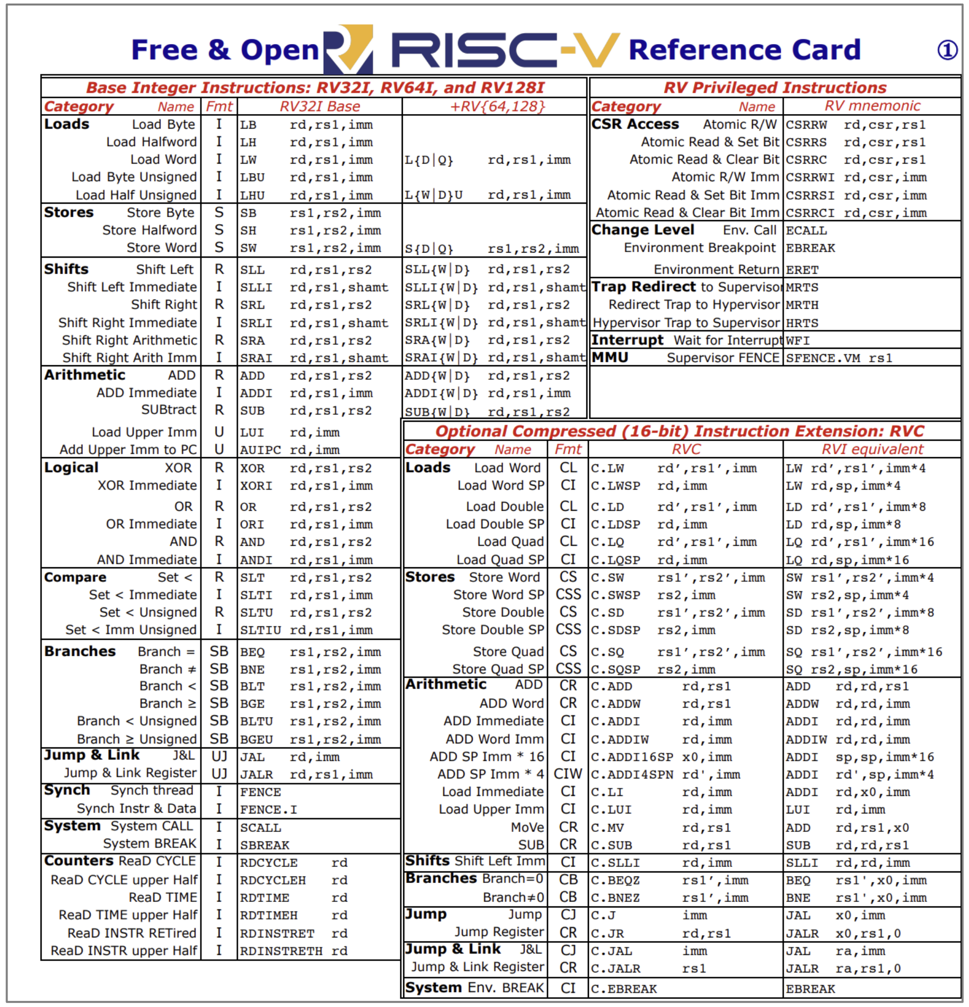
- We are using a **subset** of the RISC-V ISA called "**RV64i**"

~57 Total Instructions



RV64i Base Integer Instructions

Opcode	Instruction	Fmt	Example	Description	Notes
li	load immediate	*	li a0, 2	addi a0, zero, 2	*pseudo*
la	load address	*	la a0, symbol	a0 = symbol	*pseudo*, 2 instr
add	add	R	add a0, a1, a2	a0 = a1 + a2	
sub	subtract	R	sub a0, a1, a2	a0 = a1 - a2	
xor	bitwise exclusive or	R	xor a0, a1, a2	a0 = a1 ^ a2	
or	bitwise or	R	or a0, a1, a2	a0 = a1 | a2	
and	bitwise and	R	and a0, a1, a2	a0 = a1 & a2	
sll	shift left logical	R	sll a0, a1, a2	a0 = a1 << a2	
srl	shift right logical	R	srl a0, a1, a2	a0 = a1 >> a2	
sra	shift right arith*	R	sra a0, a1, a2	a0 = a1 >> a2	sign-extends
slt	set less than	R	slt a0, a1, a2	a0 = (a1 < a2) ? 1 : 0	
sltu	set less than (u)	R	sltu a0, a1, a2	a0 = (a1 < a2) ? 1 : 0	unsigned
addi	add immediate	I	addi a0, a1, 2	a0 = a1 + 2	
xori	xor immediate	I	xori a0, a1, 2	a0 = a1 ^ 2	
ori	or immediate	I	ori a0, a1, 2	a0 = a1 | 2	
andi	and immediate	I	andi a0, a1, 2	a0 = a1 & 2	
slli	shift left logical imm	I	slli a0, a1, 2	a0 = a1 << 2	
srli	shift right logical imm	I	srli a0, a1, 2	a0 = a1 >> 2	
srai	shift right arith imm	I	srai a0, a1, 2	a0 = a1 >> 2	sign-extends
slti	set less than imm	I	slti a0, a1, 2	a0 = (a1 < 2) ? 1 : 0	
sltiu	set less than imm (u)	I	sltiu a0, a1, 2	a0 = (a1 < 2) ? 1 : 0	unsigned
mv	move (copy)	*	mv a0, a1	addi a0, a1, 0	*pseudo*
neg	2s-complement negation	*	neg a0, a1	sub a0, zero, a1	*pseudo*
not	bitwise not	*	not a0, a1	xori a0, a1, -1	*pseudo*
lb	load byte	I	lb a0, 1(a1)	a0 = M[a1+1] (8 bits)	
lh	load half	I	lh a0, 2(a1)	a0 = M[a1+2] (16 bits)	
lw	load word	I	lw a0, 4(a1)	a0 = M[a1+4] (32 bits)	
ld	load double word	I	ld a0, 8(a1)	a0 = M[a1+8] (64 bits)	
lbu	load byte (u)	I	lbu a0, 1(a1)	a0 = M[a1+1] (8 bits)	zero-extends
lhu	load half (u)	I	lhu a0, 2(a1)	a0 = M[a1+2] (16 bits)	zero-extends
lwu	load word (u)	I	lwu a0, 4(a1)	a0 = M[a1+4] (32 bits)	zero-extends
l{b|h|w|d}	load global	*	ld a0, symbol	a0 = M[symbol]	*pseudo*, 2 instr
sb	store byte	S	sb a0, 1(a1)	M[a1+1] = a0 (8 bits)	
sh	store half	S	sh a0, 2(a1)	M[a1+2] = a0 (16 bits)	
sw	store word	S	sw a0, 4(a1)	M[a1+4] = a0 (32 bits)	
sd	store double word	S	sd a0, 8(a1)	M[a1+8] = a0 (64 bits)	
s{b|h|w|d}	store global	*	sd a0, symbol, t0	M[symbol] = a0 (uses t0)	*pseudo*, 2 instr
beq	branch if =	B	beq a0, a1, 2b	if (a0 == a1) goto 2b	
bne	branch if ≠	B	bne a0, a1, 2f	if (a0 != a1) goto 2f	
blt	branch if <	B	blt a0, a1, 2b	if (a0 < a1) goto 2b	
ble	branch if ≤	*	ble a0, a1, 2f	bge a1, a0, 2f	*pseudo*
bgt	branch if >	*	bgt a0, a1, 2b	blt a1, a0, 2b	*pseudo*
bge	branch if ≥	B	bge a0, a1, 2f	if (a0 >= a1) goto 2f	
bltu	branch if < (u)	B	bltu a0, a1, 2b	if (a0 < a1) goto 2b	unsigned
bleu	branch if ≤ (u)	*	bleu a0, a1, 2f	bgeu a1, a0, 2f	unsigned, *pseudo*
bgtu	branch if > (u)	*	bgtu a0, a1, 2b	bltu a1, a0, 2b	unsigned, *pseudo*
bgeu	branch if ≥ (u)	B	bgeu a0, a1, 2f	if (a0 >= a1) goto 2f	unsigned
beqz	branch if = 0	*	beqz a0, 2b	if (a0 == 0) goto 2b	*pseudo*
bnez	branch if ≠ 0	*	bnez a0, 2f	if (a0 != 0) goto 2f	*pseudo*
bltz	branch if < 0	*	bltz a0, 2b	if (a0 < 0) goto 2b	*pseudo*
blez	branch if ≤ 0	*	blez a0, 2f	if (a0 <= 0) goto 2f	*pseudo*
bgtz	branch if > 0	*	bgtz a0, 2b	if (a0 > 0) goto 2b	*pseudo*
bgez	branch if ≥ 0	*	bgez a0, 2f	if (a0 >= 0) goto 2f	*pseudo*
jal	jump and link	J	jal ra, label	ra = pc+4; jump to label	
jalr	jump and link reg	I	jalr ra, a1	ra = pc+4; jump to a1	
call	call subroutine	*	call label	ra = pc+4; jump to label	
j	jump	*	j label	jump to label	
lui	load upper imm	U	lui a0, 1234	a0 = 1234 << 12	
auipc	add upper imm to pc	U	auipc a0, 1234	a0 = pc + (1234 << 12)	
ecall	environment call	I	ecall	system call (calls the OS)	
ebreak	environment break	I	ebreak	break to debugger	



Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link	J&L	UJ	JAL rd,imm	
	Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE rd	
	ReaD CYCLE upper Half	I	RDCYCLEH rd	
	ReaD TIME	I	RDTIME rd	
	ReaD TIME upper Half	I	RDTIMEH rd	
	ReaD INSTR RETired	I	RDINSTRET rd	
	ReaD INSTR upper Half	I	RDINSTRETH rd	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Atomic R/W Imm	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level	Env. Call	ECALL
	Environment Breakpoint	EBREAK
	Environment Return	ERET
Trap Redirect	to Supervisor	MRTS
	Redirect Trap to Hypervisor	MRTH
	Hypervisor Trap to Supervisor	HRTS
Interrupt	Wait for Interrupt	WFI
MMU	Supervisor FENCE	SFENCE.VM rs1

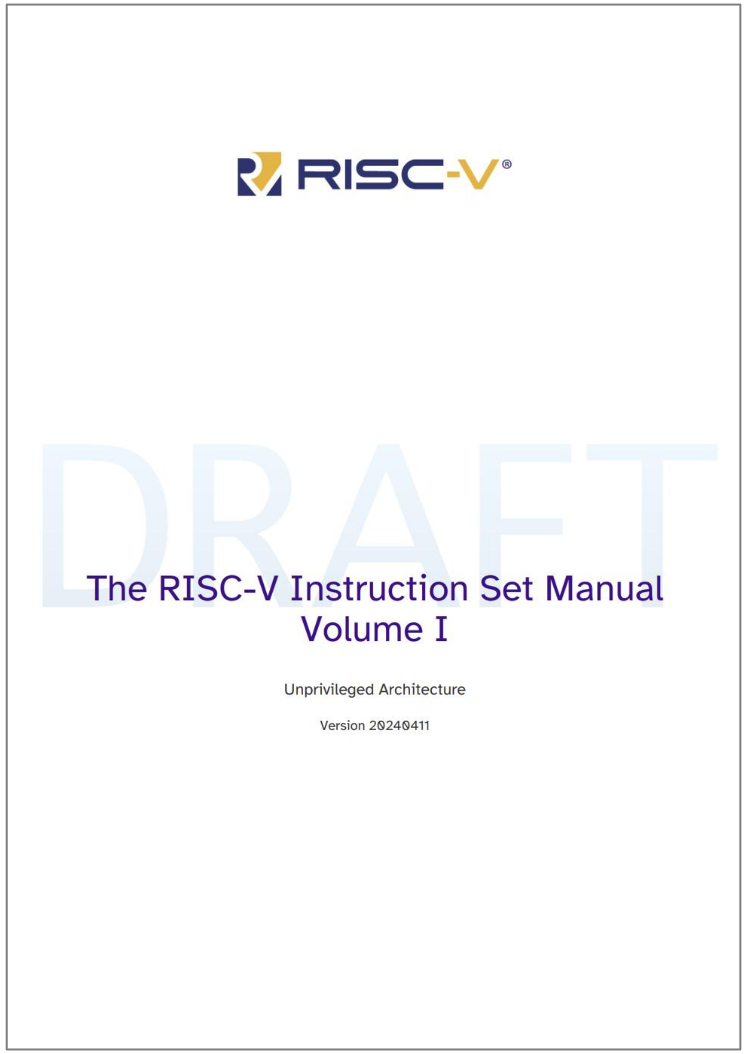
Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	C.LW rd′,rs1′,imm	LW rd′,rs1′,imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Load Double	CL	C.LD rd′,rs1′,imm	LD rd′,rs1′,imm*8
	Load Double SP	CI	C.LDSP rd,imm	LD rd,sp,imm*8
	Load Quad	CL	C.LQ rd′,rs1′,imm	LQ rd′,rs1′,imm*16
	Load Quad SP	CI	C.LQSP rd,imm	LQ rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1′,rs2′,imm	SW rs1′,rs2′,imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Store Double	CS	C.SD rs1′,rs2′,imm	SD rs1′,rs2′,imm*8
	Store Double SP	CSS	C.SDSP rs2,imm	SD rs2,sp,imm*8
	Store Quad	CS	C.SQ rs1′,rs2′,imm	SQ rs1′,rs2′,imm*16
	Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Word	CR	C.ADDW rd,rs1	ADDW rd,rd,imm
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD Word Imm	CI	C.ADDIW rd,imm	ADDIW rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1′,imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1′,imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

https://www.cs.utahtech.edu/cs/2810/riscv-card.pdf

https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf

RISC-V Instruction Set Manual

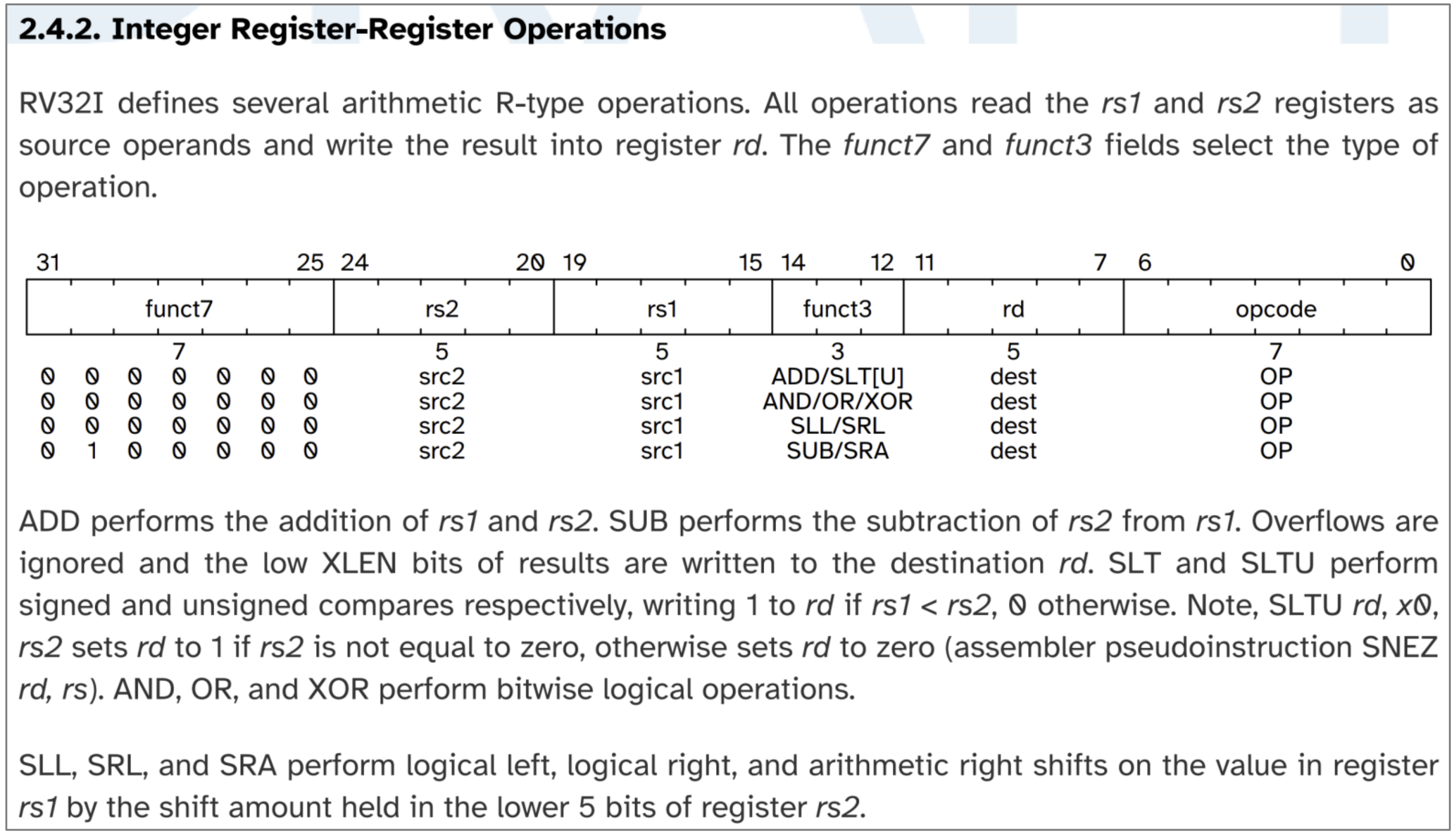

RISC-V®

The RISC-V Instruction Set Manual
Volume I

Unprivileged Architecture

Version 20240411

Example: ADD/SUB/AND/OR/XOR/SHIFT


2.4.2. Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0 0 0 0 0 0 0	src2	src1	ADD/SLT[U]	dest	OP
0 0 0 0 0 0 0	src2	src1	AND/OR/XOR	dest	OP
0 0 0 0 0 0 0	src2	src1	SLL/SRL	dest	OP
0 1 0 0 0 0 0	src2	src1	SUB/SRA	dest	OP

ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, *x0*, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd, rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

https://github.com/riscv/riscv-isa-manual/

Bitwise Instructions

"Register Allocation"

C Object	Register
`uint64_t a`	x5
`uint64_t b`	x6
`uint64_t c`	x7

C Statement	Equivalent Assembly Code
`a = b & c;`	`add x5, x6, x7`
`a = b | c;`	`or x5, x6, x7`
`a = b ^ c;`	`xor x5, x6, x7`

Logical (Unsigned) Bit Shifts

“Register Allocation”

C Object	Register
`uint64_t a`	`x5`
`uint64_t b`	`x6`
`uint64_t c`	`x7`

C Statement	Equivalent Assembly Code
`a = b << c;`	`sll x5, x6, x7`
`a = b >> c;`	`srl x5, x6, x7`

Arithmetic (Signed) Bit Shifts

"Register Allocation"

C Object	Register
`int64_t a`	`x5`
`int64_t b`	`x6`
`int64_t c`	`x7`

C Statement	Equivalent Assembly Code
`a = b << c;`	*<does not exist>*
`a = b >> c;`	`sra x5, x6, x7`

Register Copy

“Register Allocation”

C Object	Register
`uint64_t a`	`x5`
`uint64_t b`	`x6`

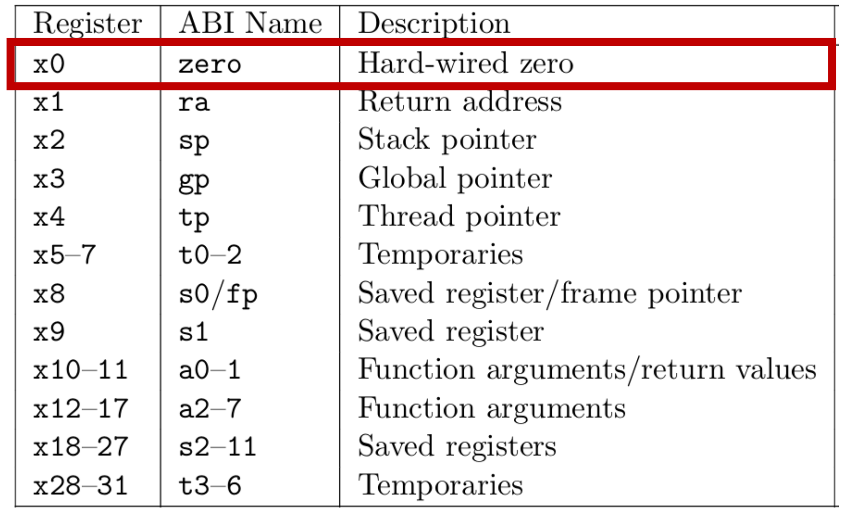
C Statement	Equivalent Assembly Code
`a = b;`	`mv x5, x6`

Pseudo-Instruction

Syntactic sugar to the assembler

(not a machine code instruction)

```
add x5, x0, x6
```



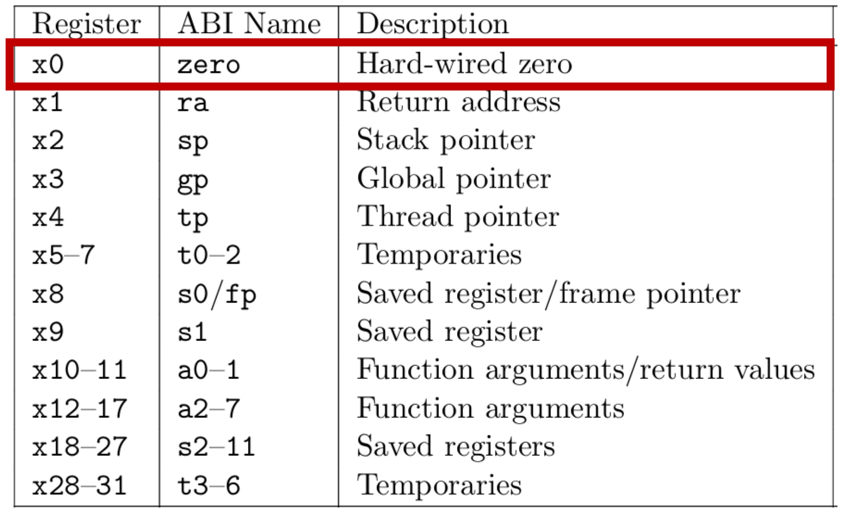
Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–7	t0–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

Agenda

- Inside a Processor
  - Lots of Pictures
  - Inside a Processor Core
  - Cache and Register Memory

- RISC-V Assembly
  - Instructions and Opcodes
  - **Immediate Values**
  - Function Arguments

Immediate Values



Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–7	t0–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

C Object	**Register**
`uint64_t a`	`x5`

C Statement(s)	**Equivalent Assembly Code**
`a = 1;`	`addi x5, x0, 0x1 # a = 0 + 1`
`a = -1;`	`addi x5, x0, -0x1 # a = 0 - 1`

Numeric constants

String Constants

C Object	Register
`char *a`	`x5`

C Statement(s)

```
void func(void)
{
        const char *a = "cs211";
}
```

Equivalent Assembly Code

```
string:
   .asciz "cs211"

func:
   la x5, string
```

Label

Defines a pointer to a location in the code

Pseudo-Instruction

Syntactic sugar to the assembler *(not a machine code instruction)*

RV64i So Far

Instructions

Register-Register Arithmetic

```
add rd, rs1, rs2
sub rd, rs1, rs2
and rd, rs1, rs2
or rd, rs1, rs2
xor rd, rs1, rs2
sll rd, rs1, rs2
srl rd, rs1, rs2
sra rd, rs1, rs2
```

Register-Immediate Arithmetic

```
addi rd, rs1, imm
```

Pseudo-Instructions

```
.asciz <C-style string>
```

```
mv rd, rs1
```

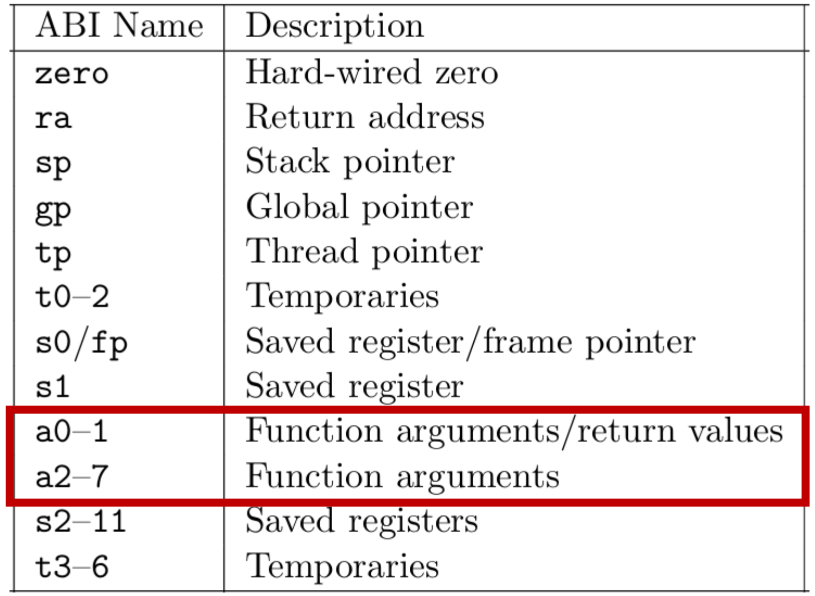
Agenda

- Inside a Processor
  - Lots of Pictures
  - Inside a Processor Core
  - Cache and Register Memory

- RISC-V Assembly
  - Instructions and Opcodes
  - Immediate Values
  - **Function Arguments**

C Function Calling Conventions

- **Function arguments** are in `a0-a7`
- **Return values** are in `a0` (+`a1`, if >64 bits)
- Narrower C types are **sign/zero extended**



ABI Name	Description
zero	Hard-wired zero
ra	Return address
sp	Stack pointer
gp	Global pointer
tp	Thread pointer
t0–2	Temporaries
s0/fp	Saved register/frame pointer
s1	Saved register
a0–1	Function arguments/return values
a2–7	Function arguments
s2–11	Saved registers
t3–6	Temporaries

func.c

```
int                     // return: a0
func(uint8_t a,         // a: a0
    uint16_t b,        // b: a1
    char *c,           // c: a2
    uint64_t d,        // d: a3
    int e,             // e: a4
    char f,            // f: a5
    unsigned int g,    // g: a6
    void *****h)       // h: a7
{
    return a + b;
}
```

func.S

```
func:
    add a0, a0, a1
    ret
```

CS 211: Intro to Computer Architecture

9.2: RISC-V Assembly

Minesh Patel

Spring 2025 – Thursday 27 March