Computer Architecture 1

Computer Organization and Design

T H E H A R D W A R E / S O F T W A R E I N T E R F A C E

[Adapted from Computer Organization and Design, RISC-V Edition, Patterson & Hennessy, © 2018, MK]
[Adapted from Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolíc, © 2020, UC Berkeley]

Instructions: Language of the Computer

- Introductions
- Operations of the Computer Hardware
- Operands of the Computer Hardware
- Logical Operations
- Instructions for Making Decisions

Introduction

- To demonstrate how easy it is to pick up other instruction sets, we will also take a quick look at two other popular instruction sets.
  - MIPS is an elegant example of the instruction sets designed since the 1980s. In several respects, RISC-V follows a similar design.
  - The Intel x86 originated in the 1970s, but still today powers both the PC and the Cloud of the post-PC era.

Assembly Language

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw   x3, 0(x10)
lw   x4, 4(x10)
sw   x4, 0(x10)
sw   x3, 4(x10)
```

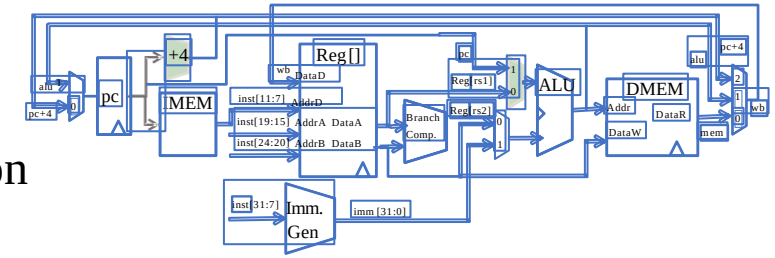
Anything can be represented as a number, i.e., data or instructions

Assembler

Machine Language Program (RISC-V)

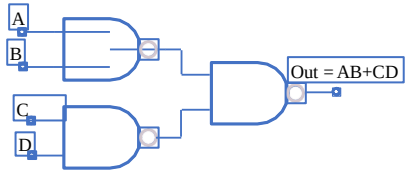
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



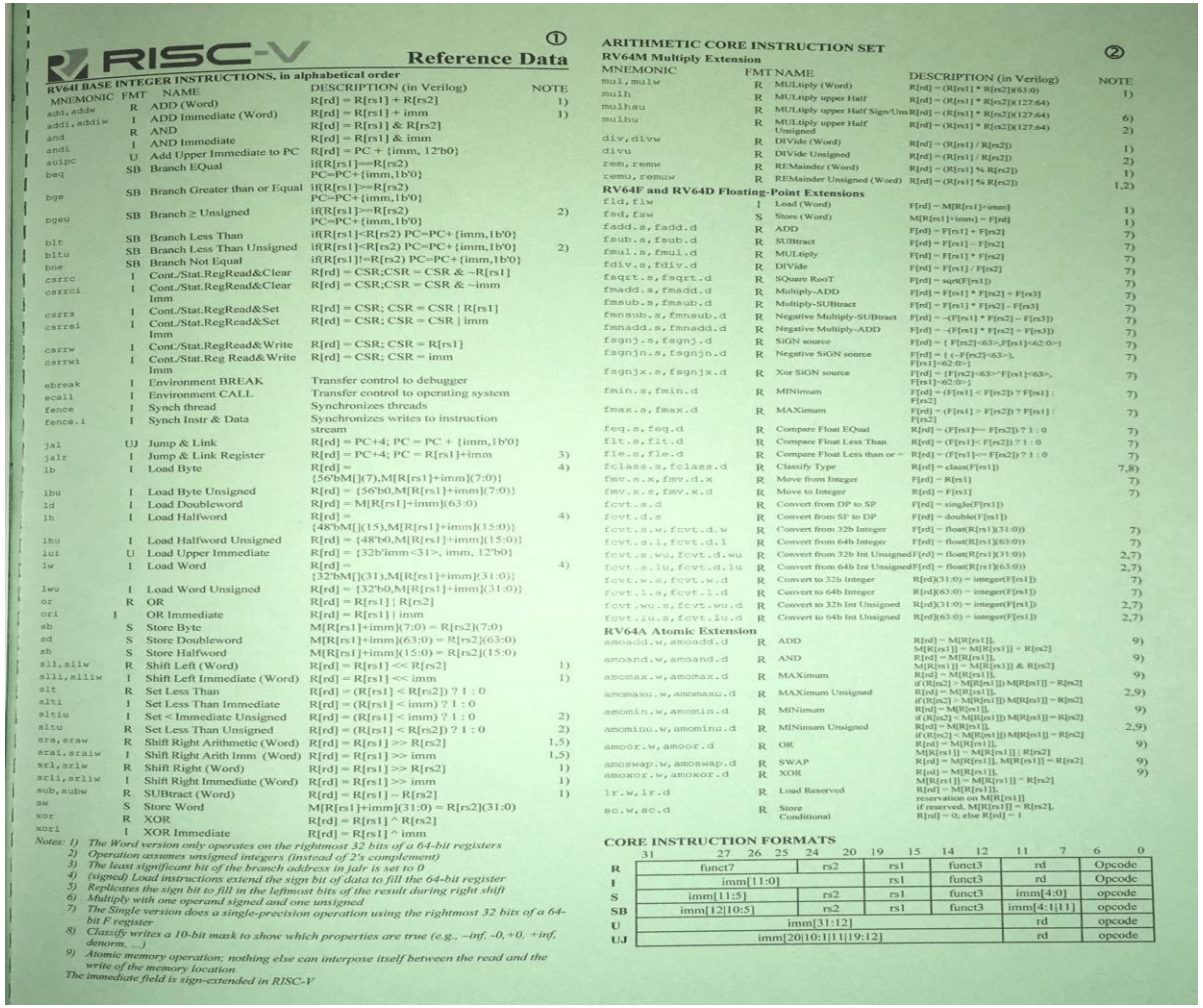
Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
  - Like a sentence: operations (verbs) applied to operands (objects) processed in sequence …
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an *Instruction Set Architecture* (*ISA*).
  - Examples: ARM (cell phones), Intel x86 (i9, i7, i5, i3), IBM Power, IBM/Motorola PowerPC (old Macs), MIPS, RISC-V, …

RISC-V Architecture

- New open-source, license-free ISA spec
  - Supported by growing shared software ecosystem
  - Appropriate for all levels of computing system, from microcontrollers to supercomputers
  - 32-bit, 64-bit, and 128-bit variants (we're using 32-bit in class, textbook uses 64-bit)
- Why RISC-V instead of Intel 80x86?
  - RISC-V is simple, elegant. Don't want to get bogged down in gritty details.
  - RISC-V has exponential adoption

RISC V – GREEN CARD



RISC-V Reference Data ①

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2]) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2]) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2]) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2] PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2] PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2] PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0, +0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply upper Half	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
mulhu	R	MULtiply upper Half Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] – F[rs3]	7)
fnmsub.s,fnmsub.d	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fnmadd.s,fnmadd.d	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] + F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = { (–F[rs2]<63>), F[rs1]<62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.s.d	R	Convert from DP to SP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert from SP to DP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

RV64A Atomic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if(R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if(R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31–27	26–25	24–20	19–15	14–12	11–7	6–0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

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

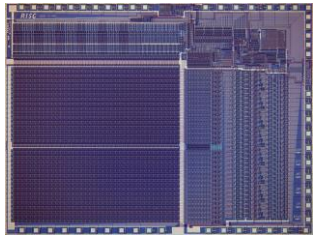
https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf

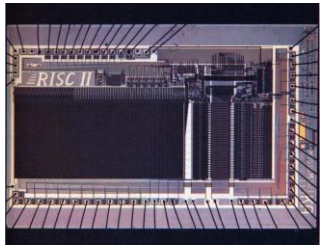
RISC V Origin https://cs61c.org/resources/

- Started in Summer 2010 to support open research and teaching at UC Berkeley
  - Lineage can be traced to RISC-I/II projects (1980s)
  - As the project matured, it migrated to RISC-V foundation (www.riscv.org)
- Many commercial and research projects based on RISC-V, open-source and proprietary
- Widely used in education
- Read more:
  - https://riscv.org/risc-v-history/
  - https://riscv.org/risc-v-genealogy/

RISC-I



RISC-II



Operations of the Computer Hardware

- Instruction set for a particular architecture (e.g. RISC-V) is represented by the Assembly language
- Each line of assembly code represents one instruction for the computer

Preliminary discussion of the logical design of an electronic computing instrument[1]

Arthur W. Burks / Herman H. Goldstine / John von Neumann

Instruction sets

3.1. It is easy to see by formal-logical methods that there exist codes that are *in abstracto* adequate to control and cause the execution of any sequence of operations which are individually available in the machine and which are, in their entirety, conceivable by the problem planner. The really decisive considerations from the present point of view, in selecting a code, are more of a practical nature: simplicity of the equipment demanded by the code, and the clarity of its application to the actually important problems together with the speed of its handling of those problems. It would take us much too far afield to discuss these questions at all generally or from first principles. We will therefore restrict ourselves to analyzing only the type of code which we now envisage for our machine.

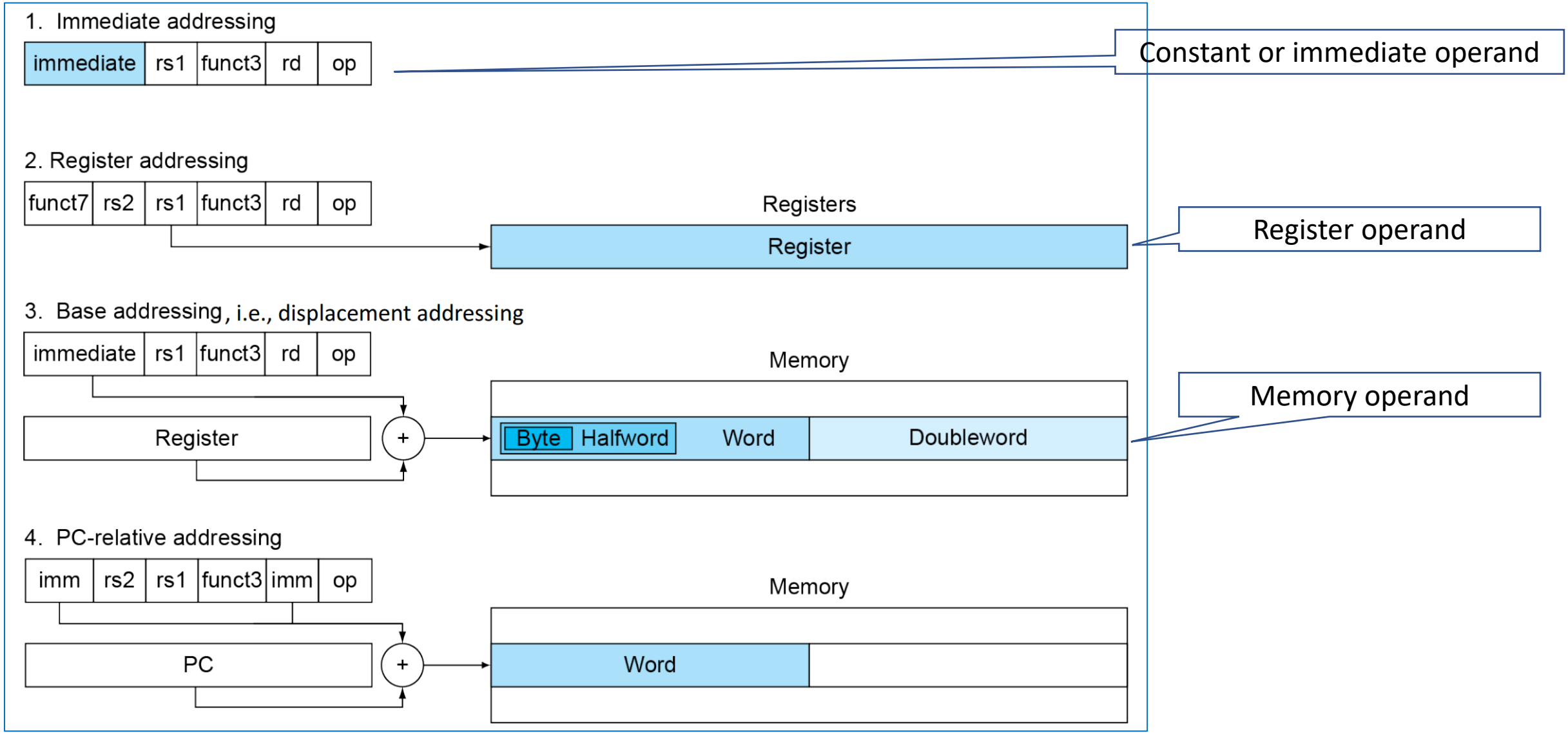
RISC V Instructions

Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

Arithmetic : + , / , * , -
Logical: AND, OR, XOR
Data transfer: Load, store
Shift : >>, <<
Condition branch: if … then
Unconditional branch: jump to loop

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load doubleword	`ld x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Doubleword from memory to register
	Store doubleword	`sd x5, 40(x6)`	`Memory[x6 + 40] = x5`	Doubleword from register to memory
	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

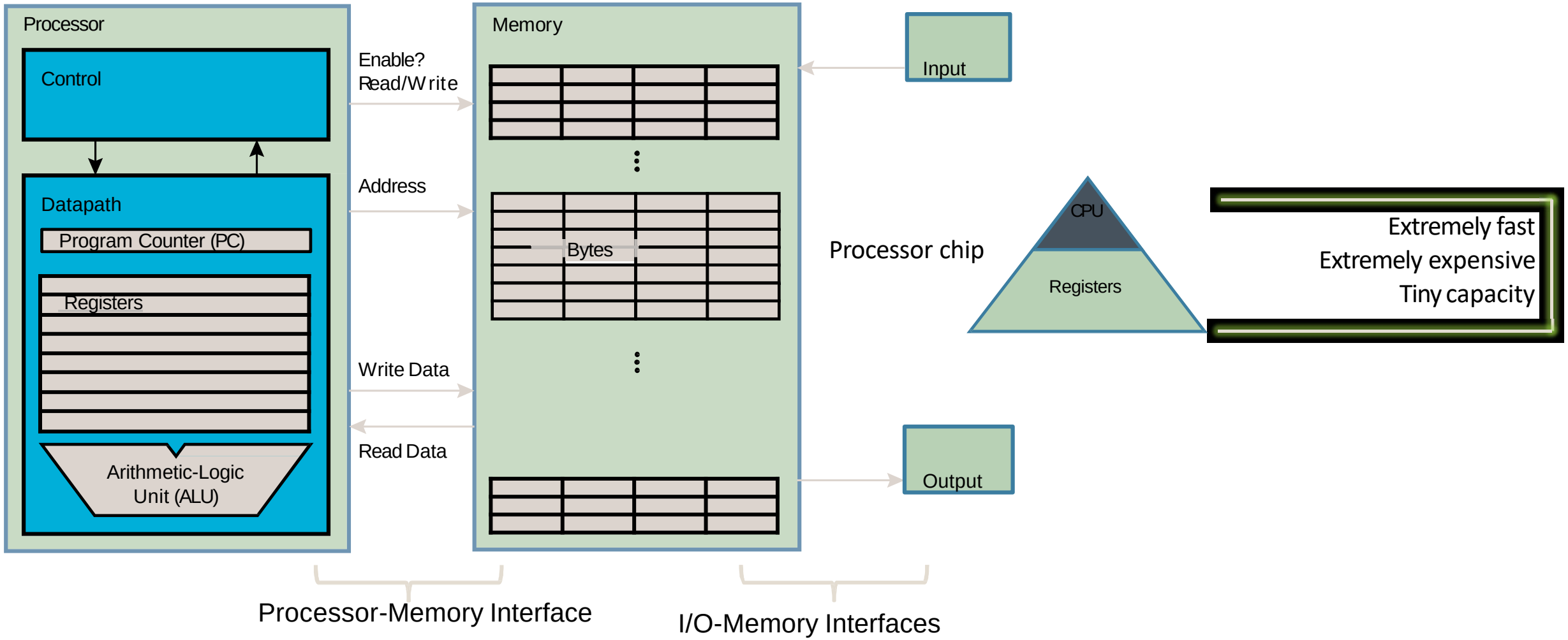
Operand of the Computer Hardware: RISC-V Addressing Summary



Assembly Variables: Registers (1/3)

- Elements of hardware: registers
- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly operands are registers
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they're very fast (faster than 0.25ns)
  - Recall light is $3x10^8$m/s = 0.3m/ns = 30cm/ns = 10cm/0.3ns!!!… where 0.3ns is the clock period of a 3.33GHz computer

Aside: Registers are Inside the Processor



Assembly Variables: Registers (2/3)

- Drawback: Since registers are in hardware, there is a predetermined number of them
  - Solution: RISC-V code must be very carefully put together to efficiently use registers
- 32 registers in RISC-V. Why 32?
  - Smaller is faster, but too small is bad. Goldilocks principle ("This porridge is too hot; This porridge is too cold; this porridge is just right")
- Each RISC-V register is 32 bits wide (in RV32 variant)
  - Groups of 32 bits called a word in RV32
  - P&H textbook uses the 64-bit variant RV64

Assembly Variables: Registers (3/3)

- Registers are numbered from 0 to 31
  - Referred to by number x0–x31
- X0 is special, always holds value zero
  - So only 31 registers able to hold variable values
- Each register can be referred to by number or name
  - Will add names later

Register	ABI Name	Description	Saver
x0	zero	hardwired zero	-
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global pointer	-
x4	tp	thread pointer	-
x5-7	t0-2	temporary registers	Caller
x8	s0 / fp	saved register / frame pointer	Callee
x9	s1	saved register	Callee
x10-11	a0-1	function arguments / return values	Caller
x12-17	a2-7	function arguments	Caller
x18-27	s2-11	saved registers	Callee
x28-31	t3-6	temporary registers	Caller

C, Java variables vs. registers

- In C (and most high-level languages) variables declared first and given a type. E.g.,

```
int fahr, celsius;
char a, b, c, d, e;
```

- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match intand char variables).
- In assembly language, the registers have no type
  - Operation determines how register contents are treated

Comments in Assembly

- Make your code more readable: comments!
  - Hash (#) is used for RISC-V comments anything from hash mark to end of line is a comment and will be ignored
- This is just like the C99 //

`Note: Different from C.`

- C comments have format **`/* comment */`** so they can span many lines
- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands
  - Unlike in C (and most other high-level languages), each line of assembly code contains at most 1 instruction
  - Instructions are related to operations (=, +, -, *, /) in C or Java

RISC-V Addition and Subtraction (1/4)

- Syntax of Instructions:

```
one two, three, four
add x1, x2, x3
```

- where:

```
one = operation by name

two = operand getting result (“destination,” x1)

three = 1st operand for operation (“source1,” x2)

four = 2nd operand for operation (“source2,” x3)
```

- Syntax is rigid: `1 operator, 3 operands`
- Why? Keep hardware simple via regularity

Addition and Subtraction of Integers (2/4)

- Addition in Assembly
  - Example: **`add x1,x2,x3 (in RISC-V)`**
  - Equivalent to: **`a = b + c (in C)`**
  - where C variables ⇔ RISC-V registers are: **`a ⇔ x1, b ⇔ x2, c ⇔ x3`**
- Subtraction in Assembly
  - Example: **`sub x3,x4,x5 (in RISC-V)`**
  - Equivalent to: **`d = e - f (in C)`**
  - where C variables ⇔ RISC-V registers are:

  `d ⇔ x3, e ⇔ x4, f ⇔ x5`

Addition and Subtraction of Integers (3/4)

- How to do the following C statement?

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

- Break into multiple instructions

```
add x10, x1, x2 # a_temp = b + c
add x10, x10, x3 # a_temp = a_temp + d
sub x10, x10, x4 # a = a_temp - e
```

- Notice: A single line of C may break up into several lines of RISC-V.
- Notice: Everything after the hash mark on each line is ignored (comments).

Addition and Subtraction of Integers (4/4)

- How do we do this?

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

- Use intermediate temporary register

```
add x5, x20, x21 # a_temp = g + h
add x6, x22, x23 # b_temp = i + j
sub x19, x5, x6 # f = (g + h)- (i + j)
```

Immediates

- Immediates are numerical constants. They appear often in code, so there are special instructions for them.
- Add Immediate:

```
addi x3,x4,10 (in RISC-V)
f = g + 10 (in C)
```

- where RISC-V registers x3, x4 are associated with C variables f, g
- Syntax similar to **add** instruction, except that last argument is a number instead of a register.

Immediates

- There is no Subtract Immediate in RISC-V: Why? There are **add** and **sub**, but no **addi** counterpart
- Limit types of operations that can be done to absolute minimum if an operation can be decomposed into a simpler
- where RISC-V registers **`x3,x4`** are associated with C variables f, g, respectively

```
addi x3,x4,-10 (in RISC-V)
f = g - 10 in C
```

Register Zero

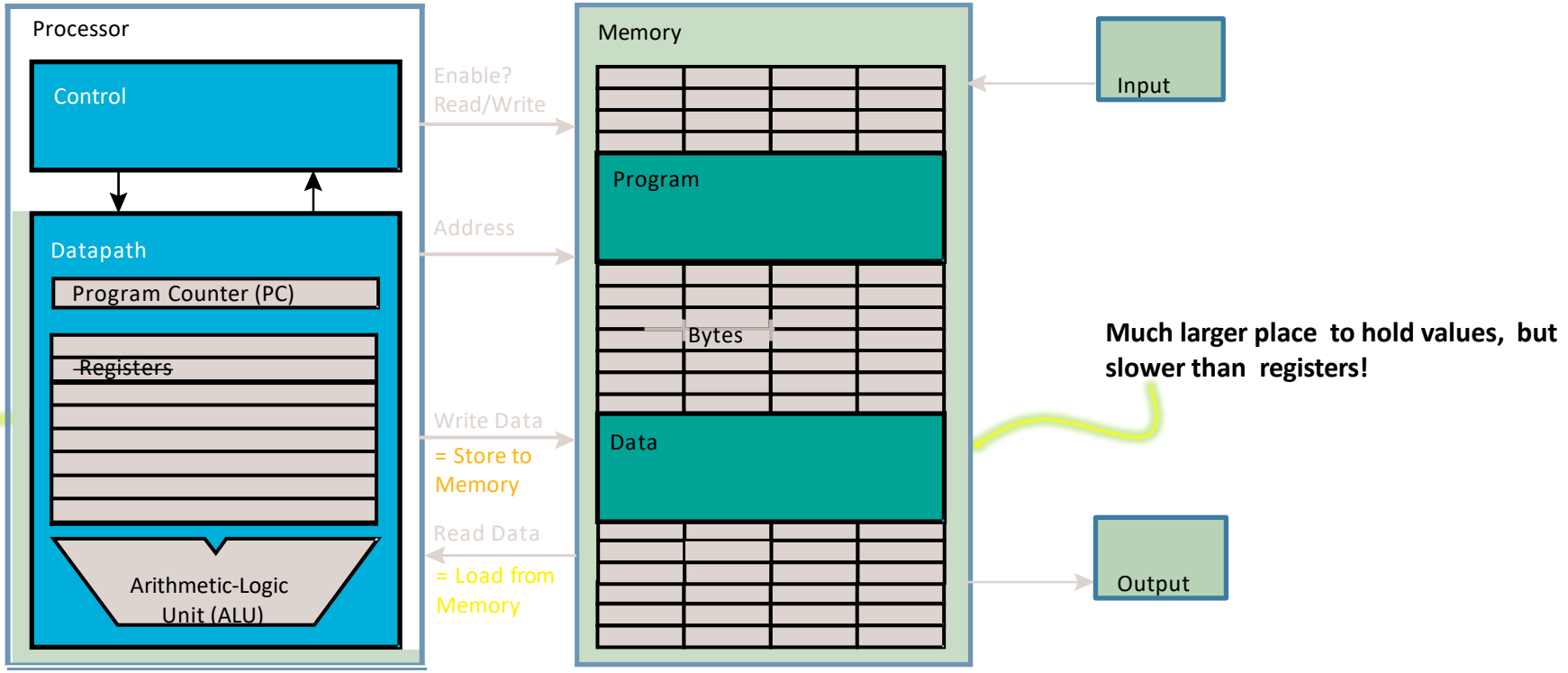
- One particular immediate, the number zero (0), appears very often in code.
- So the register zero (x0) is ‘hard-wired’ to value 0; e.g.

```
add x3,x4,x0  # in RISC V
f = g   # in C
```

- where RISC-V registers **x3,x4** are associated with C variables **f, g**
- Defined in hardware, so an instruction

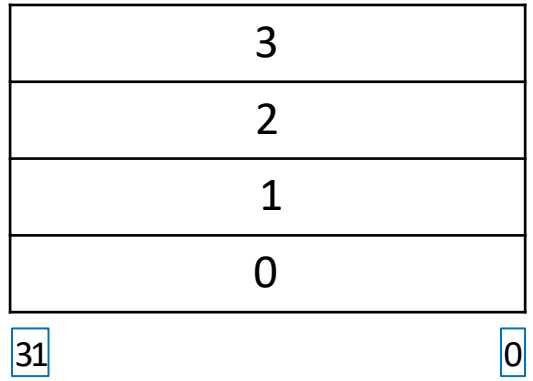
```
add x0,x3,x4 # will not do anything!
```

Data Transfer: Load from and Store to memory

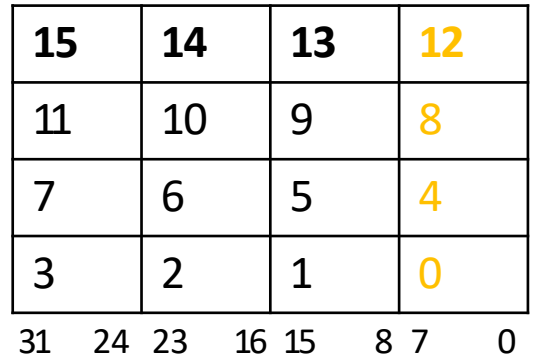


Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)–works fine if everything is a multiple of 8 bits
  - 8 bit chunk is called a byte (1 word = 4 bytes)
  - Memory addresses are really in bytes, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of rightmost byte – least-significant byte (i.e. Little-endian convention)



Least-significant byte in a word



Least-significant byte gets the smallest address

Big Endian vs. Little Endian

The adjective endian has its origin in the writings of 18th century writer Jonathan Swift. In the 1726 novel Gulliver's Travels, he portrays the conflict between sects of Lilliputians divided into those breaking the shell of a boiled egg from the big end or from the little end. He called them the "Big-Endians" and the "Little-Endians".

The order in which BYTES are stored in memory

Bits always stored as usual (E.g., 0xC2=0b 1100 0010)

Consider the number 1025 as we typically write it:

BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Big Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE0	BYTE1	BYTE2	BYTE3
00000001	00000100	00000000	00000000

Examples

Names in China or Hungary (e.g., Nikolić Bora)

Java Packages: (e.g., org.mypackage.HelloWorld)

Dates in ISO 8601 YYYY-MM-DD (e.g., 2020-09-07)

Eating Pizza crust first

Little Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Examples

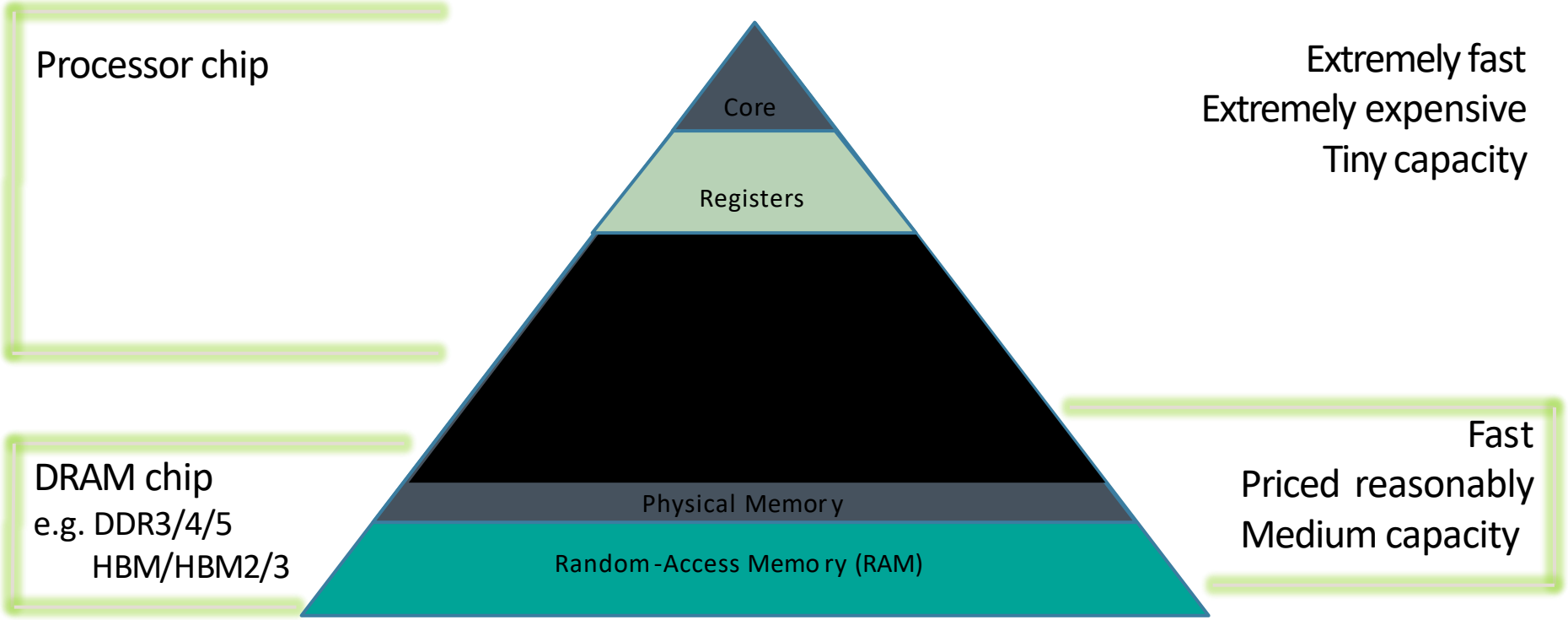
Names in the US (e.g., Bora Nikolić)

Internet names (e.g., cs.berkeley.edu)

Dates written in Europe DD/MM/YYYY (e.g., 07/09/2020)

Eating Pizza skinny part first

Speed of Registers vs. Memory



- Registers: 32 words (128 Bytes)
- Memory (DRAM): Billions of bytes (2 GB to 64 GB on laptop) and physics dictates that **Smaller is Faster**
- How much faster are registers than DRAM??
  - About 50-500 times faster! (in terms of latency of one access - tens of ns)
  - But subsequent words come every few ns

Load from Memory to Register

- C code

```
int  A[100];  g = h + A[3];
```

- Using Load Word (lw) in RISC-V:

```
lw  x10,12(x15) # Reg x10 gets A[3]
add x11,x12,x10 # g = h + A[3]
# x15 - base register (pointer to A[0])  12 - offset in bytes
# Offset must be a constant known at assembly time
```

Store from Register to Memory

- C code

```
int    A[100];  A[10] = h + A[3];
```

- Using Store Word (sw) in RISC-V:

```
lw     x10,12(x15) # Temp reg x10 gets A[3]
add x10,x12,x10    # Temp reg x10 gets h + A[3]
sw     x10,40(x15) # A[10] = h + A[3]
```

- Note:

```
x15 – base register (pointer) 12,40 – offsets in bytes
x15+12 and x15+40 must be multiples of 4
```

Loading and Storing Bytes

- In addition to word data transfers (lw, sw), RISC-V has byte data transfers:

  load byte: lb
  store byte: sb

- Same format as lw, sw. E.g.,

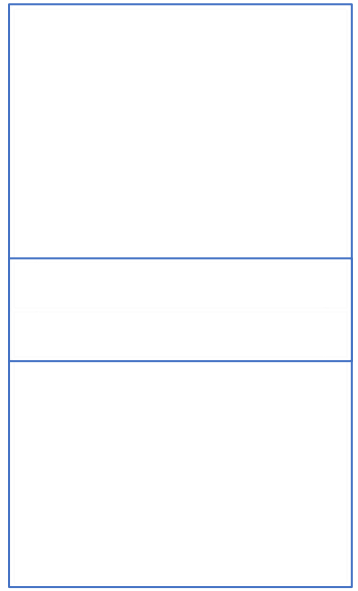
```
lb x10,3(x11)
```

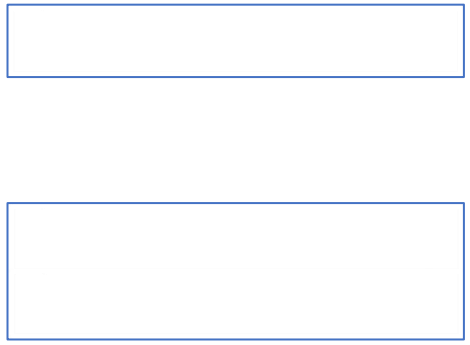
-> contents of memory location with address = sum of "3" + contents of register x11 is copied to the low byte position of register x10.

RISC-V also has "unsigned byte" loads (`lbu`) which zero extends to fill register. Why no unsigned store byte '`sbu`'?

Example: What is in x12 ?

```
addi x11,x0,0x3F5
sw x11,0(x5)
lb x12,1(x5)
```





Substituting addi

- The following two instructions:

```
lw   x10,12(x15) # Temp reg x10 gets A[3]
add x12,x12,x10  # reg x12 = reg x12 + A[3]
```

- Replace addi:

```
addi x12, value # value in A[3]
```

- But involve a load from memory!
- Add immediate is so common that it deserves its own instruction!

RV32 So Far…

- Addition/subtraction

```
add rd, rs1, rs2   sub rd, rs1, rs2
```

- Add immediate

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

- Load/store

```
lw rd, rs1, imm
lb rd,rs1, imm
lbu rd,rs1, imm
sw rs1, rs2, imm
sb rs1, rs2, imm
```

Computer Decision Making

- Based on computation, do something different
- In programming languages: if-statement
- RISC-V: if-statement instruction is

```
beq reg1,reg2,L1
```

- means: go to statement labeled L1 if (value in reg1) == (value in reg2) ….otherwise, go to next statement
- beq stands for branch if equal
- Other instruction: bne for branch if not equal

Types of Branches

- Branch – change of control flow
- Conditional Branch – change control flow depending on outcome of comparison
  - branch *if* equal (**`beq`**) or branch *if not* equal (**`bne`**)
  - Also branch if less than (**`blt`**) and branch if greater than or equal (**`bge`**)
  - And unsigned versions (**`bltu`**, **`bgeu`**)
- Unconditional Branch – always branch
  - a RISC-V instruction for this*: jump* (**`j`**), as in
    `j label`

Example *if* Statement

- Assuming translations below, compile if block

f → **x10** g → **x11** h → **x12**

i → **x13** j → **x14**

```
if (i == j)
  f = g + h;
```

```
bne x13,x14,Exit
add x10,x11,x12
Exit:
```

- May need to negate branch condition

Example *if-else* Statement

- Assuming translations below, compile

f → **x10** g → **x11** h → **x12**

i → **x13** j → **x14**

```
if (i == j)
  f = g + h;
Else
f = g - h;
```

```
bne x13,x14,Else
add x10,x11,x12
j Exit
Else:sub x10,x11,x12
Exit:
```

Magnitude Compares in RISC-V

- General programs need to test < and > as well.
- RISC-V magnitude-compare branches:

"Branch on Less Than"

Syntax: **`blt reg1,reg2, Label`**

Meaning: **`if (reg1 < reg2) goto Label;`**

"Branch on Less Than Unsigned"

Syntax: **`bltu reg1,reg2, Label`**

Meaning:
```
if (reg1 < reg2)// treat registers
                as unsigned integers
    goto label;
```

Also "Branch on Greater or Equal" **`bge`** and **`bgeu`**

Note: No '**`bgt`**' or '**`ble`**' instructions

Loops in C/Assembly

- There are three types of loops in C:
- while
- do … while
- for
- Each can be rewritten as either of the other two, so the same branching method can be applied to these loops as well.
- Key concept: Though there are multiple ways of writing a loop in RISC-V, the key to decision- making is conditional branch

C Loop Mapped to RISC-V Assembly

```
int A[20];  int sum = 0;
for (int i=0; i < 20; i++)
sum += A[i];
```

C Loop Mapped to RISC-V Assembly

```
int A[20];
int sum = 0;
for (int i=0; i < 20; i++)
    sum +=  A[i];
```

```
    add x9, x8, x0 # x9=&A[0]
    add x10, x0, x0 # sum
    add x11, x0, x0 # i
    addi x13,x0, 20 # x13
Loop:
    bge x11,x13,Done
    lw x12, 0(x9)    # x12 A[i]
    add x10,x10,x12  # sum
    addi x9, x9,4    # &A[i+1]
    addi x11,x11,1   # i++
    j Loop
Done:
```