计算机组成与系统结构
Computer Organization & System Architecture

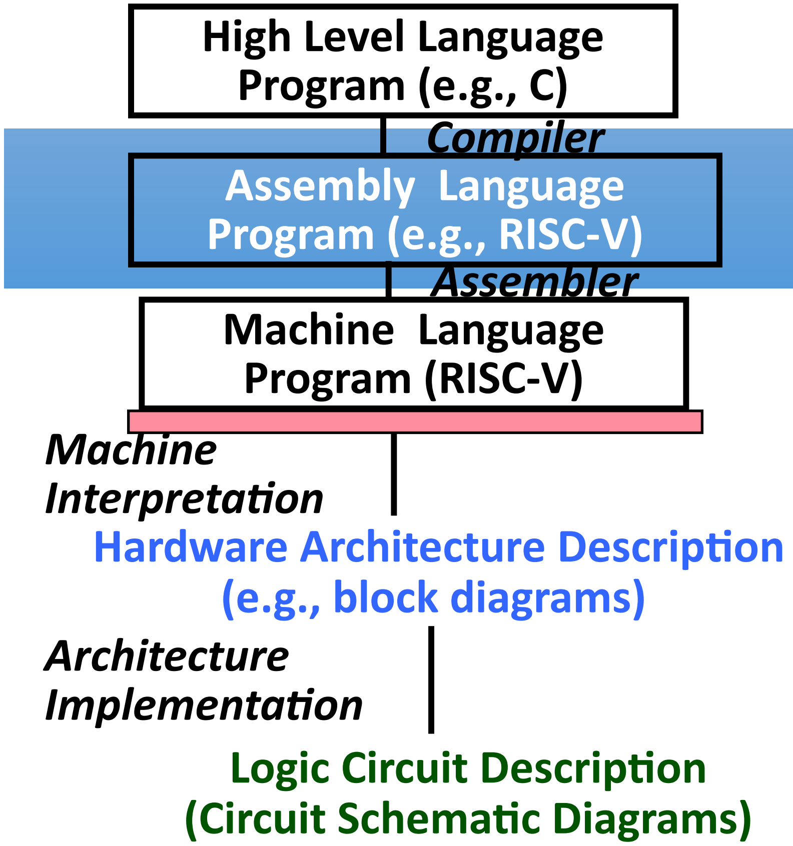
Huang Kejie (黄科杰) 百人计划研究员

Office: 玉泉校区老生仪楼 304

Email address: huangkejie@zju.edu.cn

HP: 17706443800

Levels of Representation/Interpretation



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

Compiler

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

Assembler

Machine Language Program (RISC-V)

Machine Interpretation

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

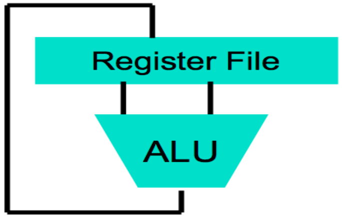
Architecture Implementation

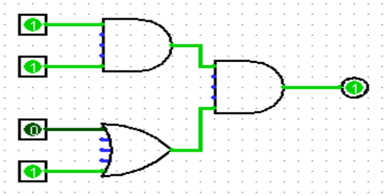
Logic Circuit Description (Circuit Schematic Diagrams)

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

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

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





Instruction Set Architecture (ISA)

- Job of a CPU (*Central Processing Unit*, aka *Core*): execute *instructions*

- Instructions: CPU's primitives operations
  - Like a sentence: operations (verbs) applied to operands (objects) processed in sequence …
  - With additional operations to change the sequence

- CPUs belong to "families," each implementing its own set of instructions

- CPU's particular set of instructions implements an *Instruction Set Architecture (ISA)*
  - Examples: ARM, Intel x86, MIPS, RISC-V, IBM/Motorola PowerPC (old Mac), Intel IA64, ...

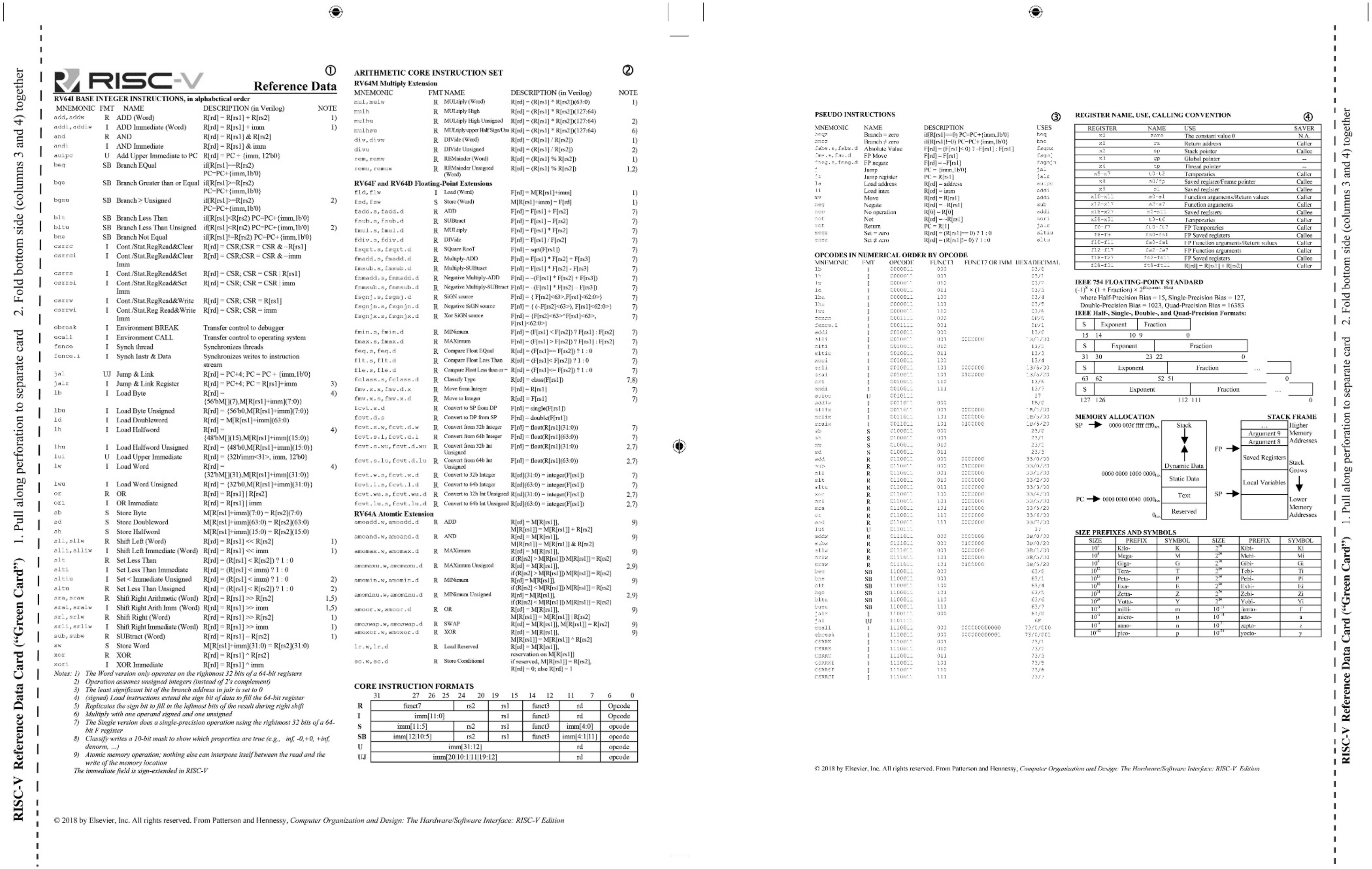
Instruction Set Architectures

- Early trend: add more instructions to new CPUs for elaborate operations
  - VAX architecture had an instruction to multiply polynomials!

- RISC philosophy (Cocke IBM, Patterson UCB, Hennessy Stanford, 1980s) – *Reduced Instruction Set Computing*
  - Keep the instruction set small and simple, in order to build fast hardware
  - Let software do complicated operations by composing simpler ones

RISC-V Green Card (in textbook)

- Inspired by the IBM 360 "Green Card"



RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

①

RISC-V **Reference Data**

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 High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
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

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
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)
fnmadd.s,fnmadd.d	R	Negative Multiply-ADD	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)
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 to SP from DP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert to DP from SP	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 Atomtic Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE

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 – 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

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]<0) ? -F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = -F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = -R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

IEEE Half-, Single-, Double-, and Quad-Precision Formats:

S	Exponent	Fraction
15	14 – 10	9 – 0

S	Exponent	Fraction
31	30 – 23	22 – 0

S	Exponent	Fraction ...
63	62 – 52	51 – 0

S	Exponent	Fraction ...
127	126 – 112	111 – 0

MEMORY ALLOCATION **STACK FRAME**

SP → 0000 003f ffff fff0_{hex} Stack Dynamic Data

0000 0000 1000 0000_{hex} Static Data

PC → 0000 0000 0040 0000_{hex} Text

0_{hex} Reserved

... Argument 9 Argument 8 Higher Memory Addresses

FP → Saved Registers Stack Grows

SP → Local Variables Lower Memory Addresses

SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

What is RISC-V?

- Fifth generation of RISC design from UC Berkeley
- A high-quality, license-free, royalty-free RISC ISA specification
- Experiencing rapid uptake in both industry and academia
- Both proprietary and open-source core implementations
- 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)
- Standard maintained by non-profit RISC-V Foundation

Foundation Members (60+)

Platinum:

Berkeley Architecture Research

bluespec

cortus

DRAPER

IBM

Google

SAMSUNG

Microsemi

Mellanox TECHNOLOGIES

NVIDIA

QUALCOMM

Microsoft

SiFive

Rambus Cryptography Research

WD Western Digital

HUAWEI

Gold, Silver, Auditors:

AMD

ANDES TECHNOLOGY

antmicro EMBEDDED SYSTEMS

Blockstream

ICT 中科院计算所 INSTITUTE OF COMPUTING TECHNOLOGY, CAS

codasip

Esperanto Technologies

ESPRESSIF

GRAY RESEARCH

MEDIATEK

IDT

INTRINSIX

ETH zürich

LATTICE SEMICONDUCTOR

lowRISC

MINIMA PROCESSOR

MIT CSAIL

ROA LOGIC

Rumble Development

runtime.io

Sur Technology

Syntacore Custom cores and tools

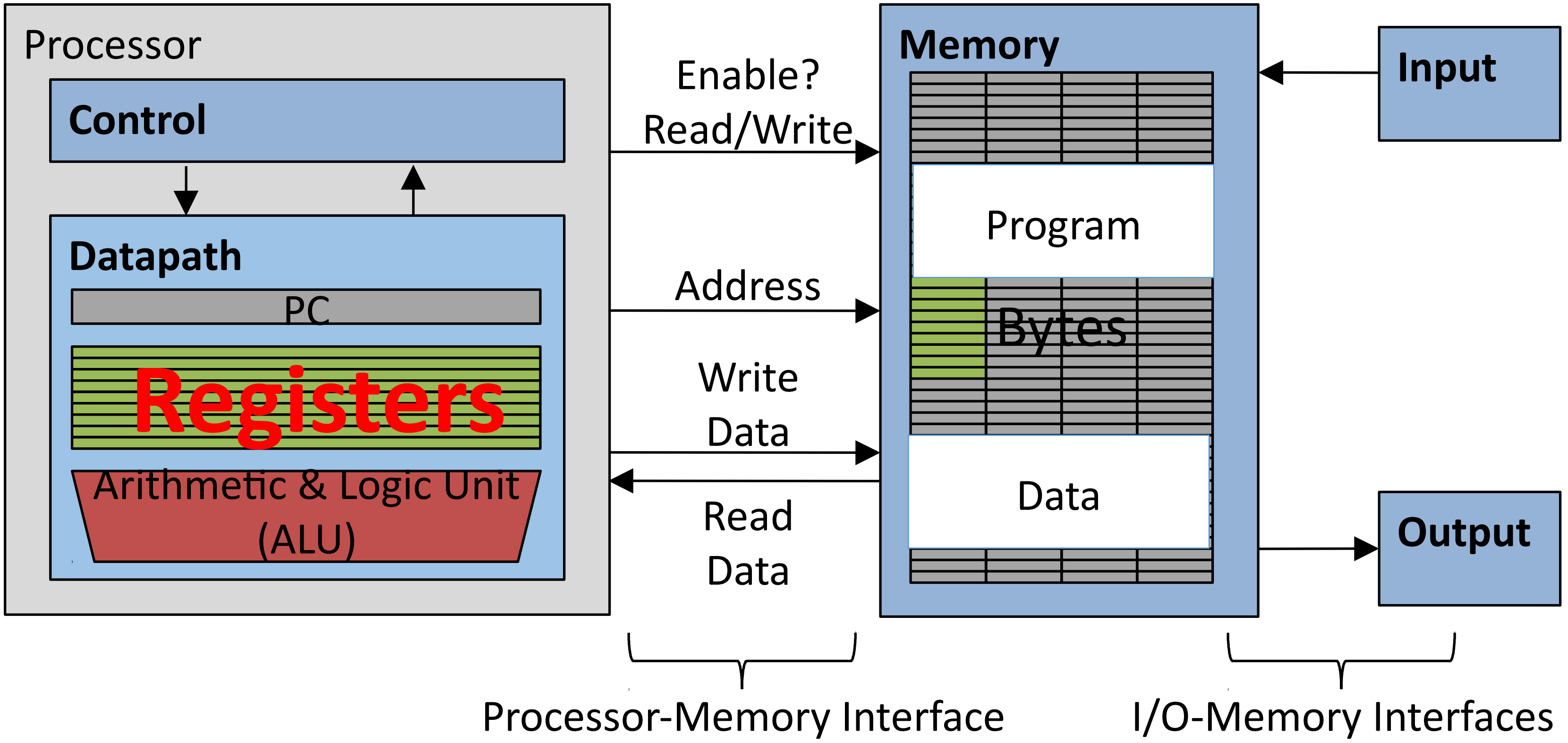
Technolution

VectorBlox embedded supercomputing

Assembly Variables: Registers

- Unlike HLL like C or Java, assembly does not have variables as you know and love them
  - More primitive, closer what simple hardware can directly support
- Assembly operands are objects called registers
  - Limited number of special places to hold values, built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 ns - light travels 1 foot in 1 ns!!!)

Registers live inside the Processor



Number of RISC-V Registers

- Drawback: Since registers are in hardware, there are a limited number of them
  - Solution: RISC-V code must be carefully written to efficiently use registers
- 32 registers in RISC-V, referred to by number **`x0–x31`**
  - Registers are also given symbolic names, described later
  - 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 (***RV32*** variant of RISC-V ISA)
  - Groups of 32 bits called a **word** in RISC-V ISA
  - P&H CoD textbook uses the 64-bit variant RV64 (explain differences later)
- **`x0`** is special, always holds value zero
  - So really only 31 registers able to hold variable values

C, Java Variables vs. Registers

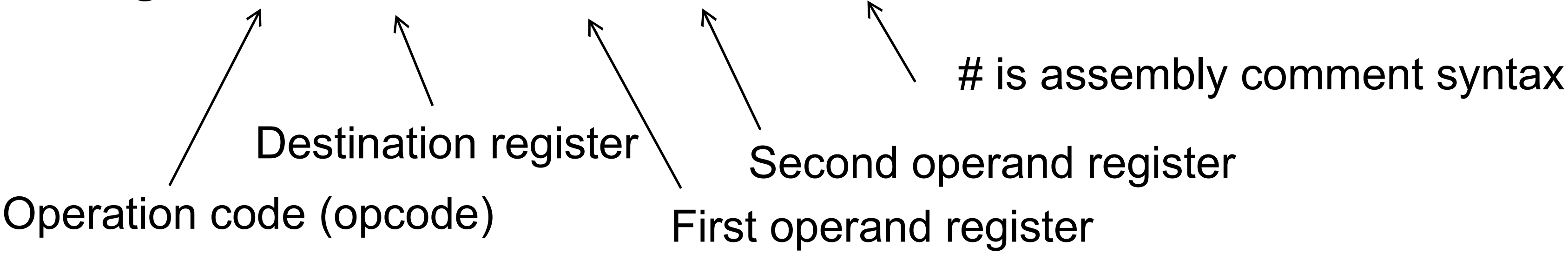
- In C (and most HLLs):
  - Variables declared and given a type
  - Example:

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

  - Each variable can ONLY represent a value of the type it was declared (e.g., cannot mix and match int and char variables)
- In Assembly Language:
  - Registers have no type;
  - Operation determines how register contents are interpreted

RISC-V Instruction Assembly Syntax

- Instructions have an opcode and operands
- E.g., **`add x1, x2, x3 # x1 = x2 + x3`**



# is assembly comment syntax

Destination register

Second operand register

Operation code (opcode)

First operand register

Addition and Subtraction of Integers

- 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 Example 1

- 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
```

- A single line of C may turn into several RISC-V instructions

Immediates

- Immediates are numerical constants

- They appear often in code, so there are special instructions for them

- Add Immediate:

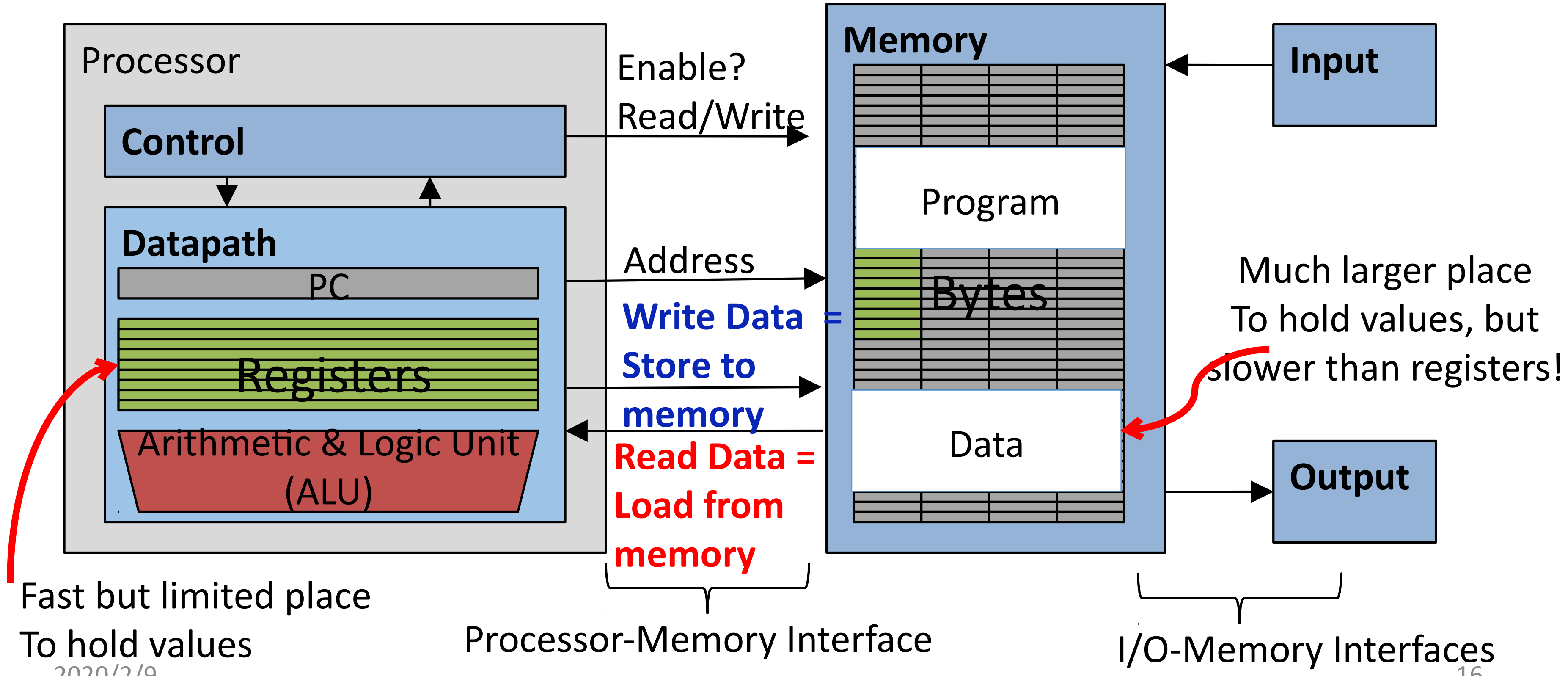
```
addi x3,x4,-10          (in RISC-V)
f = g - 10              (in C)
```

  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

```
add x3,x4,x0            (in RISC-V)
f = g                   (in C)
```

Data Transfer: Load from and Store to memory



Memory Addresses are in Bytes

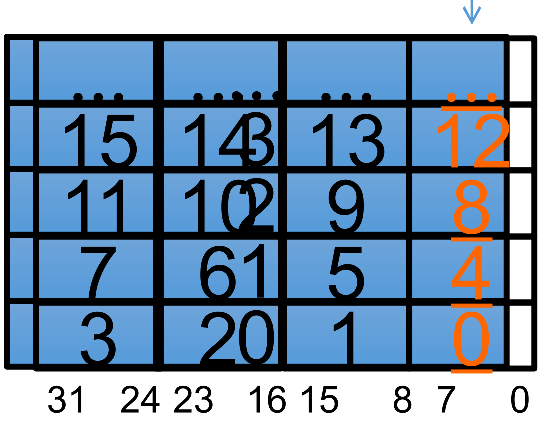
- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)–works fine if everything is a multiple of 8 bits

- 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

Transfer from Memory to Register (load)

- C code

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

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

```
lw  x10,12(x13) # Reg x10 gets A[3]
add x11,x12,x10 # g = h + A[3]
```

Note: **x13** – base register (pointer to A[0])

12 – offset in bytes

Offset must be a constant known at assembly time

Transfer from Register to Memory (store)

- C code

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

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

```
lw  x10,12(x13)   # Reg x10 gets A[3]
add x10,x12,x10   # g = h + A[3]
sw  x10,40(x13)   # A[10] = h + A[3]
```

Note: **x13** – base register (pointer to A[0])

12, 40 – offset in bytes

x13+12 and **x13+40** must be multiples of 4

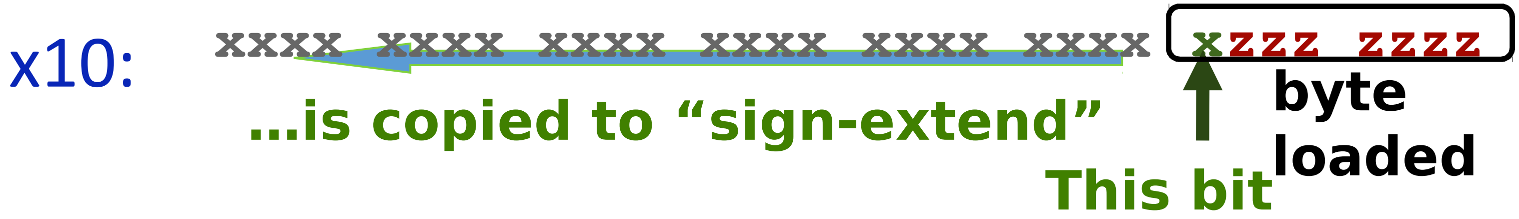
Loading and Storing Bytes

- In addition to word data transfers (lw, sw), RISC-V has byte data transfers:
  - load byte: **`lb`**
  - store byte: **`sb`**

RISC-V also has “unsigned byte” loads (**`lbu`**) which zero extend to fill register. Why no unsigned store byte **`sbu`**?

- 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`**.



x10:
xxxx xxxx xxxx xxxx xxxx xxxx xzzz zzzz
…is copied to “sign-extend”
This bit
byte loaded

Your turn

```
addi x11,x0,0x3f5
sw x11,0(x5)
lb x12,1(x5)
```

Answer	x12
RED	0x5
GREEN	0xf
ORANGE	0x3
YELLOW	0xffffffff

Your turn

```
addi x11,x0,0x3f5
sw x11,0(x5)
lb x12,1(x5)
```

Answer	x12
RED	0x5
GREEN	0xf
ORANGE	0x3
YELLOW	0xffffffff

Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory (DRAM): Billions of bytes (2 GB to 8 GB on laptop)

- and physics dictates…
  - Smaller is faster

- How much faster are registers than DRAM??

- About 100-500 times faster!
  - in terms of latency of one access

RISC-V Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called *logical operations*

Logical operations	C operators	Java operators	RISC-V instructions
Bit-by-bit AND	&	&	**`and`**
Bit-by-bit OR	|	|	**`or`**
Bit-by-bit XOR	^	^	**`xor`**
Shift left logical	<<	<<	**`sll`**
Shift right logical	>>	>>	**`srl`**

Logical Shifting

- *Shift Left Logical*: `slli x11,x12,2 #x11=x12<<2`
  - Store in `x11` the value from `x12` shifted 2 bits to the left (they fall off end), **inserting 0's** on right; << in C

Before: `0000 0002`$_{\text{hex}}$

`0000 0000 0000 0000 0000 0000 0000 0010`$_{\text{two}}$

After: `0000 0008`$_{\text{hex}}$

`0000 0000 0000 0000 0000 0000 0000 1000`$_{\text{two}}$

- What arithmetic effect does shift left have?

- *Shift Right Logical*: `srli` is opposite shift; >>
  - Zero bits inserted at left of word, right bits shifted off end

Arithmetic Shifting

- *Shift Right Arithmetic* (**`srai`**) moves n bits to the right (insert high-order sign bit into empty bits)

- For example, if register **`x10`** contained

  `1111 1111 1111 1111 1111 1111 1110 0111`$_{\text{two}}$**`= -25`**$_{\text{ten}}$

- If execute **`sra x10, x10, 4`**, result is:

  `1111 1111 1111 1111 1111 1111 1111 1110`$_{\text{two}}$**`= -2`**$_{\text{ten}}$

- Unfortunately, this is NOT same as dividing by 2^n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

Computer Decision Making

- Based on computation, do something different
- In programming languages: *if*-statement

- RISC-V: *if*-statement instruction is

```
beq register1,register2,L1
```

  means: go to statement labeled L1
  if (value in register1) == (value in register2)
  ….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`**)

- **Unconditional Branch** – always branch
  - a RISC-V instruction for this: *jump (j)*

Example *if* Statement

- Assuming translations below, compile if block

```
f → x10      g → x11     h → x12
i → x13      j → x14

if (i == j)     bne x13,x14,Exit
   f = g + h;      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)    bne x13,x14,Else
   f = g + h;     add x10,x11,x12
else        j Exit
   f = g - h;     Else: sub x10,x11,x12
            Exit:
```

Magnitude Compares in RISC-V

- Until now, we’ve only tested equalities (== and != in C); 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)` // treat registers as signed integers

  `goto label;`

- “Branch on Less Than Unsigned”

  Syntax: `bltu reg1,reg2, label`

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

  `goto label;`

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

Peer Instruction

Which of the following is TRUE?

- RED: **`add x10,x11,4(x12)`** is valid in RV32
- GREEN: can byte address 8GB of memory with an RV32 word
- ORANGE: **`imm`** must be multiple of 4 for **`lw x10,imm(x10)`** to be valid
- YELLOW: None of the above

Peer Instruction

Which of the following is TRUE?

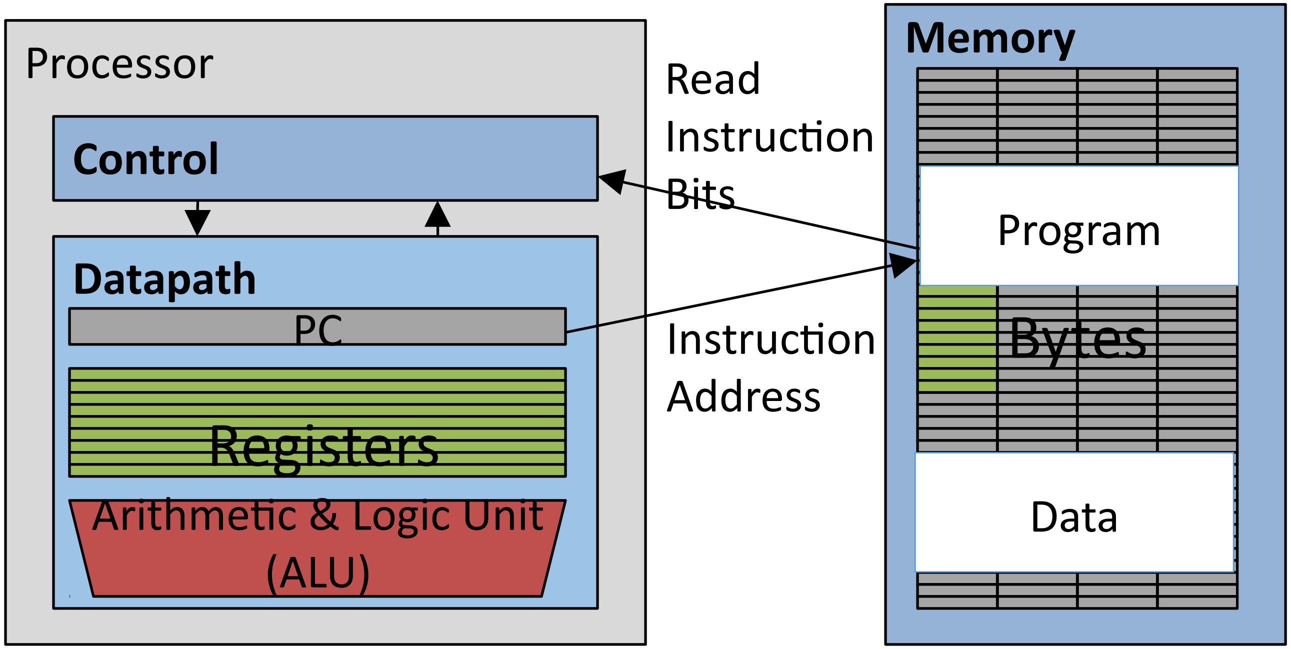
RED: **`add x10,x11,4(x12)`** is valid in RV32

GREEN: can byte address 8GB of memory with an RV32 word

ORANGE: **`imm`** must be multiple of 4 for **`lw x10,imm(x10)`** to be valid

YELLOW: None of the above

Program Execution



- **PC** (program counter) is internal register inside processor holding byte address of next instruction to be executed

- Instruction is fetched from memory, then control unit executes instruction using datapath and memory system, and updates program counter (default is add +4 bytes to PC, to move to next sequential instruction)

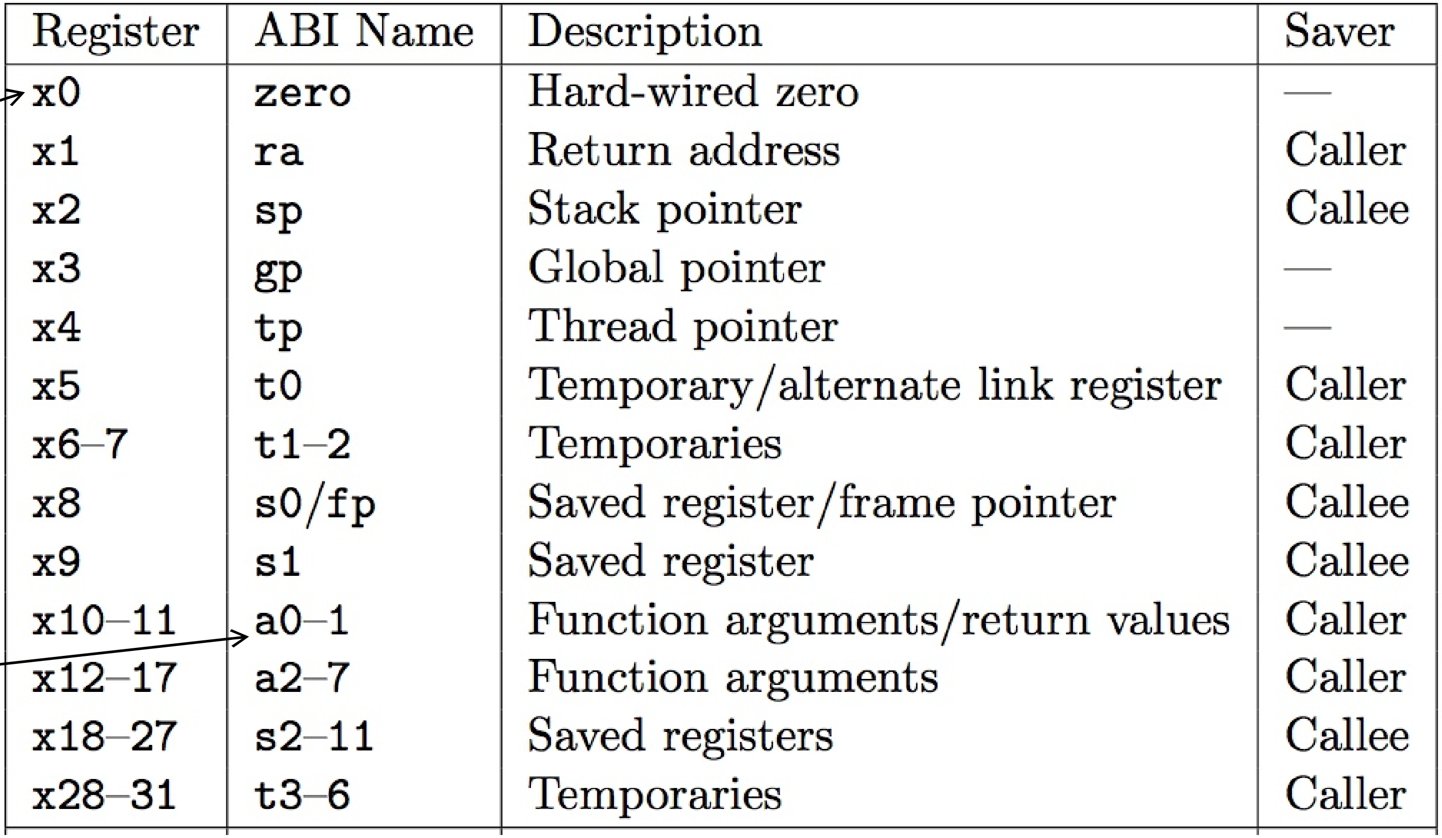
Helpful RISC-V Assembler Features

- Symbolic register names
  - E.g., **`a0-a7`** for argument registers (**`x10-x17`**)
  - E.g., zero for **`x0`**

- Pseudo-instructions
  - Shorthand syntax for common assembly idioms
  - E.g., **`mv rd, rs = addi rd, rs, 0`**
  - E.g.2, **`li rd, 13 = addi rd, x0, 13`**

RISC-V Symbolic Register Names

Numbers hardware understands



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	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	Temporaries	Caller

Human-friendly symbolic names in assembly code

Six Fundamental Steps in Calling a Function

1. Put parameters in a place where function can access them

2. Transfer control to function

3. Acquire (local) storage resources needed for function

4. Perform desired task of the function

5. Put result value in a place where calling code can access it and restore any registers you used

6. Return control to point of origin, since a function can be called from several points in a program

RISC-V Function Call Conventions

- Registers faster than memory, so use them
- **`a0–a7`** (**`x10-x17`**): eight *argument* registers to pass parameters and two return values (**`a0-a1`**)
- **`ra`**: one *return address* register to return to the point of origin (**`x1`**)

Instruction Support for Functions (1/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
}
int sum(int x, int y) {
return x+y;
}
```

RISC-V

```
address (shown in decimal)
1000
1004
1008
1012
1016
…
2000
2004
```

In RISC-V, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
}
int sum(int x, int y) {
return x+y;
}
```

RISC-V

```
address (shown in decimal)
1000 mv a0,s0   # x = a
1004 mv a1,s1   # y = b
1008 addi ra,zero,1016 #ra=1016
1012 j     sum  #jump to sum
1016 … # next instruction
…
2000 sum: add a0,a0,a1
2004 jr   ra      # new instr. "jump register"
```

Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
}
int sum(int x, int y) {
return x+y;
}
```

RISC-V

- Question: Why use jr here? Why not use j?
- Answer: **sum** might be called by many places, so we can't return to a fixed place. The calling proc to **sum** must be able to say "return here" somehow.

```
2000 sum: add a0,a0,a1
2004 jr   ra # new instr. "jump register"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address: jump and link (**jal**)

- Before:

```
1008 addi ra,zero,1016  #ra=1016
1012 j sum      #goto sum
```

- After:

```
1008 jal sum         # ra=1012,goto sum
```

- Why have a jal?
  - Make the common case fast: function calls very common
  - Reduce program size
  - Don’t have to know where code is in memory with **jal**!

Instruction Support for Functions (4/4)

- Invoke function: *jump and link* instruction (**`jal`**) (really should be **`laj`** “link and jump”)
  - “link” means form an *address or link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of the following instruction in register **`ra`**

```
jal FunctionLabel
```

- Return from function: jump register instruction (**`jr`**)
  - Unconditional jump to address specified in register: **`jr ra`**
  - Assembler shorthand: **`ret = jr ra`**

Example

```
int Leaf
(int g, int h, int i, int j)
{
   int f;
   f = (g + h) – (i + j);
   return f;
}
```

- Parameter variables **g, h, i,** and **j** in argument registers **a0, a1, a2,** and **a3**, and **f** in **s0**

- Assume need one temporary register **s1**

Where Are Old Register Values Saved to Restore Them After Function Call?

- Need a place to save old values before call function, restore them when return, and delete

- Ideal is *stack*: last-in-first-out queue (e.g., stack of plates)
  - Push: placing data onto stack
  - Pop: removing data from stack

- Stack in memory, so need register to point to it

- **`sp`** is the *stack pointer* in RISC-V (**`x2`**)

- Convention is grow stack down from high to low addresses

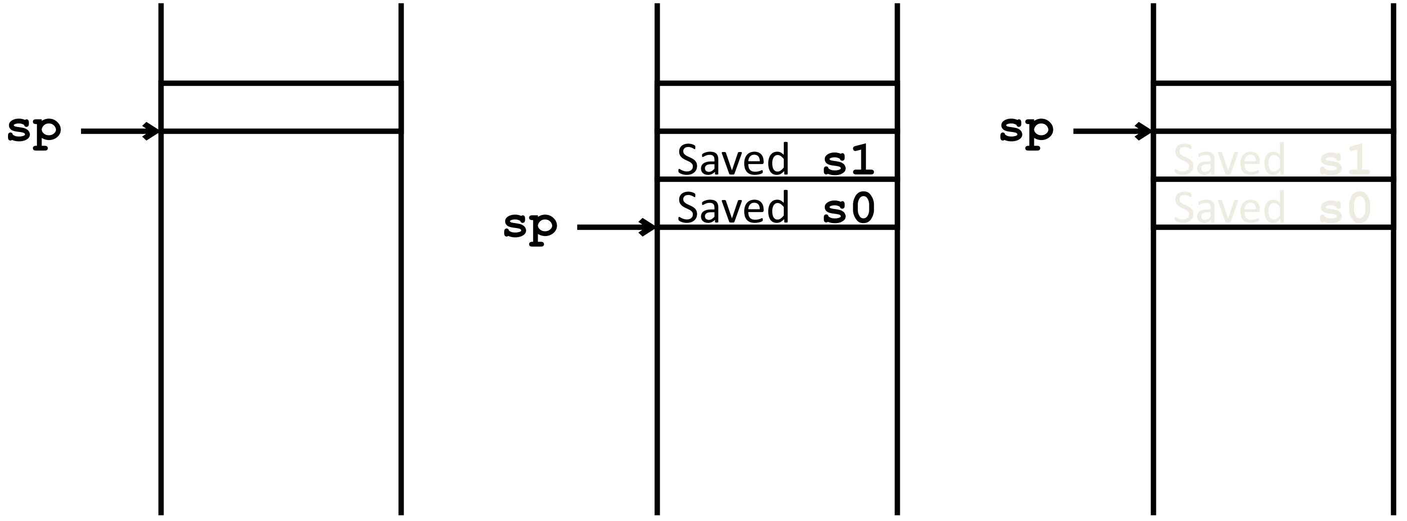
- Push decrements **`sp`**, Pop increments **`sp`**

RISC-V Code for Leaf()

```
Leaf: addi sp,sp,-8 # adjust stack for 2 items
   sw s1, 4(sp)  # save s1 for use afterwards
      sw s0, 0(sp)  # save s0 for use afterwards

      add s0,a0,a1  # f = g + h
      add s1,a2,a3  # s1 = i + j
      sub a0,s0,s1  # return value (g + h) – (i + j)

      lw s0, 0(sp)  # restore register s0 for caller
      lw s1, 4(sp)  # restore register s1 for caller
      addi sp,sp,8  # adjust stack to delete 2 items
      jr ra      # jump back to calling routine
```

Stack Before, During, After Function

- Need to save old values of **`s0`** and **`s1`**



Before call

During call

After call

What If a Function Calls a Function? Recursive Function Calls?

- Would clobber values in **`a0-a7`** and **`ra`**

- What is the solution?

Nested Procedures (1/2)

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y;
}
```

- Something called **sumSquare**, now **sumSquare** is calling **mult**

- So there's a value in **ra** that **sumSquare** wants to jump back to, but this will be overwritten by the call to mult

Need to save **sumSquare** return address before call to **mult**

Nested Procedures (2/2)

- In general, may need to save some other info in addition to **`ra`**.

- When a C program is run, there are three important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes - e.g., C globals
  - Heap: Variables declared dynamically via **`malloc`**
  - Stack: Space to be used by procedure during execution; this is where we can save register values

Optimized Function Convention

To reduce expensive loads and stores from spilling and restoring registers, RISC-V function-calling convention divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - **`sp, gp, tp,`** “saved registers” **`s0-s11`** (**`s0`** is also **`fp`**)

2. Not preserved across function call
   - Caller *cannot* rely on values being unchanged
   - Argument/return registers **`a0-a7,ra,`** “temporary registers” **`t0-t6`**

Peer Instruction

- Which statement is FALSE?
- RED: RISC-V uses **`jal`** to invoke a function and **`jr`** to return from a function
- GREEN: **`jal`** saves PC+1 in **`ra`**
- ORANGE: The callee can use temporary registers (**`t`***`i`*) without saving and restoring them
- YELLOW: The caller can rely on save registers (**`s`***`i`*) without fear of callee changing them

Peer Instruction

- Which statement is FALSE?
- RED: RISC-V uses **`jal`** to invoke a function and **`jr`** to return from a function
- GREEN: **`jal`** saves PC+1 in **`ra`**
- ORANGE: The callee can use temporary registers (**`t`***`i`*) without saving and restoring them
- YELLOW: The caller can rely on save registers (**`s`***`i`*) without fear of callee changing them

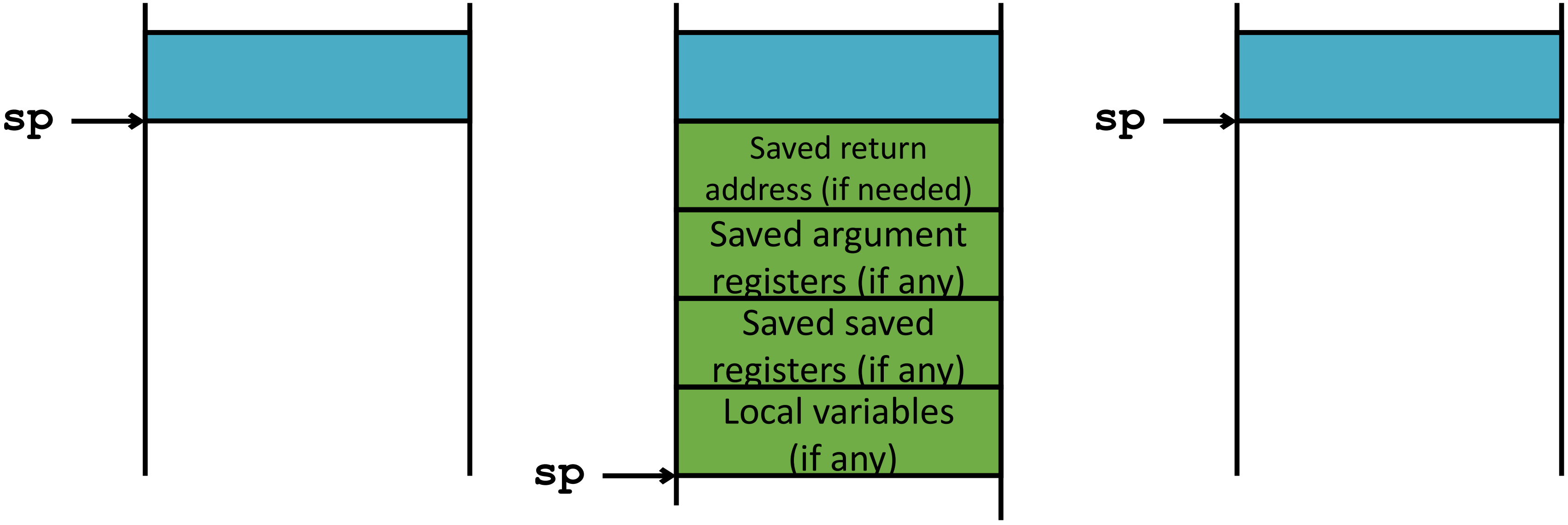
Allocating Space on Stack

- C has two storage classes: automatic and static
  - *Automatic* variables are local to function and discarded when function exits
  - *Static* variables exist across exits from and entries to procedures

- Use stack for automatic (local) variables that don't fit in registers

- *Procedure frame* or *activation record*: segment of stack with saved registers and local variables

Stack Before, During, After Function


sp
Saved return address (if needed)
Saved argument registers (if any)
Saved saved registers (if any)
Local variables (if any)
sp
sp

Before call

During call

After call

Using the Stack (1/2)

- So we have a register **sp** which always points to the last used space in the stack

- To use stack, we decrement this pointer by the amount of space we need and then fill it with info

- So, how do we compile this?

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y;
}
```

Using the Stack (2/2)

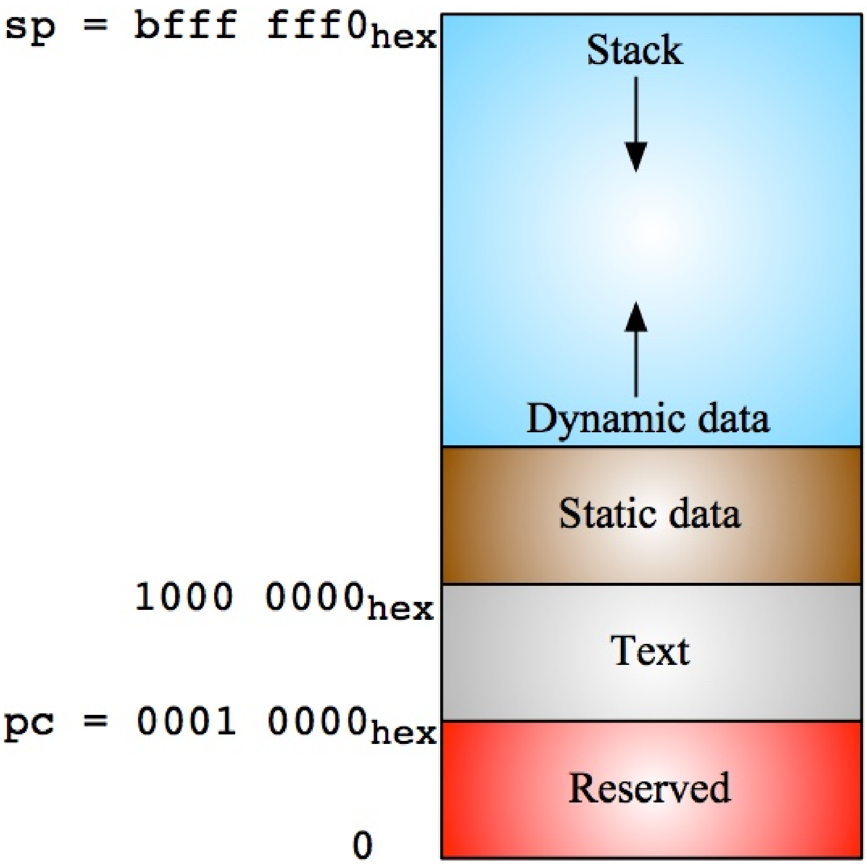
```
int sumSquare(int x, int y) {
   return mult(x,x)+ y; }


sumSquare:
      addi sp,sp,-8  # space on stack
      sw ra, 4(sp)   # save ret addr
      sw a1, 0(sp)   # save y
      mv a1,a0    # mult(x,x)
      jal mult    # call mult
      lw a1, 0(sp)   # restore y
      add a0,a0,a1   # mult()+y
      lw ra, 4(sp)   # get ret addr
      addi sp,sp,8   # restore stack
      jr ra
mult: ...
```

Where is the Stack in Memory?

- RV32 convention (RV64 and RV128 have different memory layouts)
- Stack starts in high memory and grows down
  - Hexadecimal (base 16) : **`bfff_fff0`**$_{hex}$
  - Stack must be aligned on 16-byte boundary (not true in examples above)
- RV32 programs (text segment) in low end

  `0001_0000`$_{hex}$
- *static data segment* (constants and other static variables) above text for static variables
  - RISC-V convention *global pointer* (**`gp`**) points to static
  - RV32 **`gp = 1000_0000`**$_{hex}$
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

RV32 Memory Allocation


sp = bfff fff0$_{hex}$
Stack
Dynamic data
Static data
1000 0000$_{hex}$
Text
pc = 0001 0000$_{hex}$
Reserved
0