

CS 61C:

Great Ideas in Computer Architecture

*Introduction to Assembly Language and
RISC-V Instruction Set Architecture*

Instructors:

Krste Asanović & Randy H. Katz

<http://inst.eecs.Berkeley.edu/~cs61c>

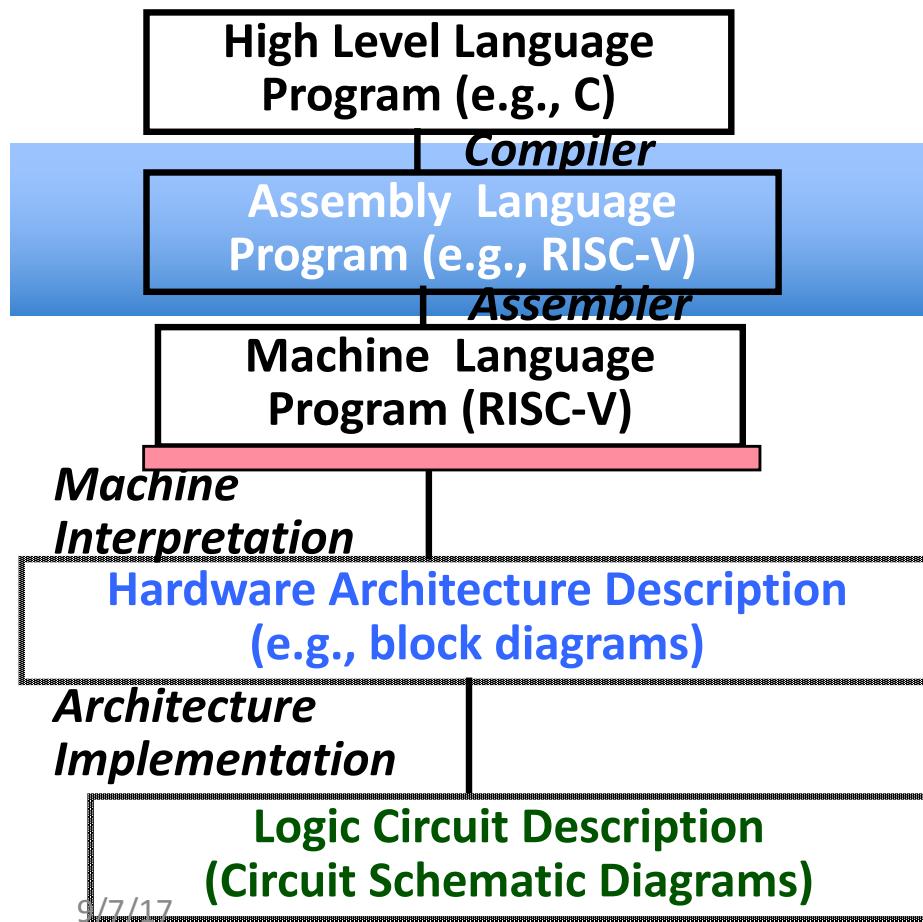
Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion ...

Outline

- **Assembly Language**
 - RISC-V Architecture
 - Registers vs. Variables
 - RISC-V Instructions
 - C-to-RISC-V Patterns
 - And in Conclusion ...

Levels of Representation/Interpretation

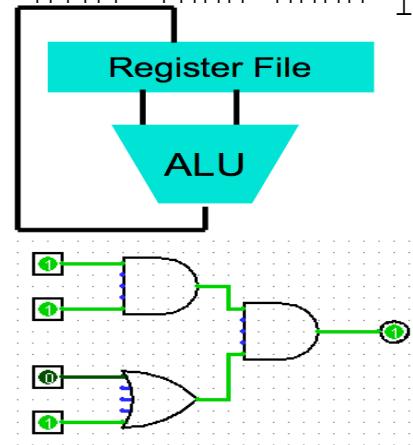


**temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;**

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

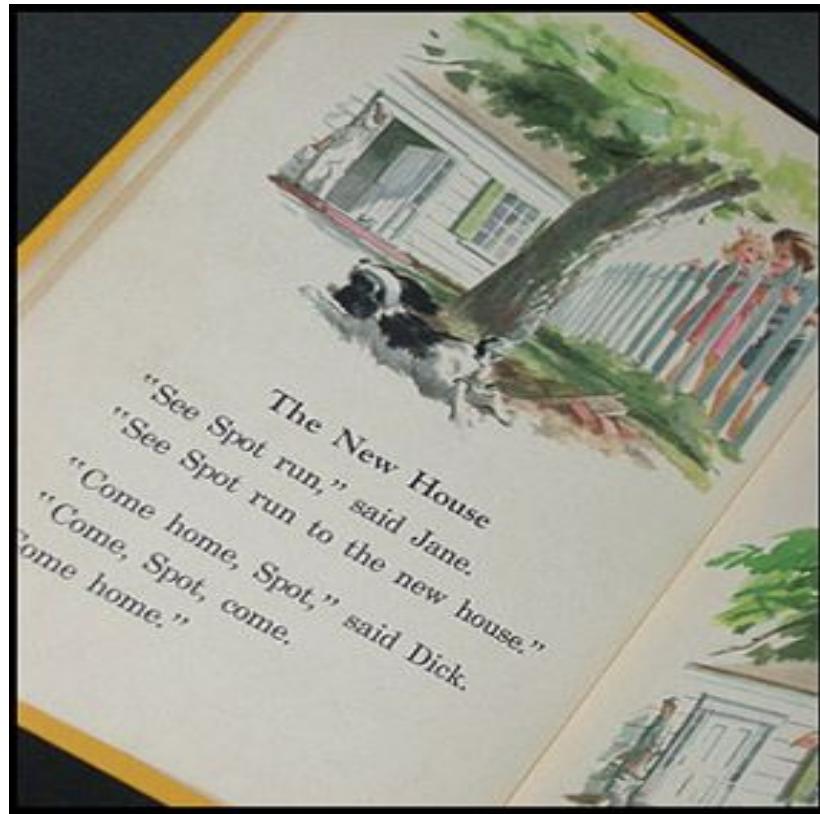
Anything can be represented as a *number*, i.e., data or instructions

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111



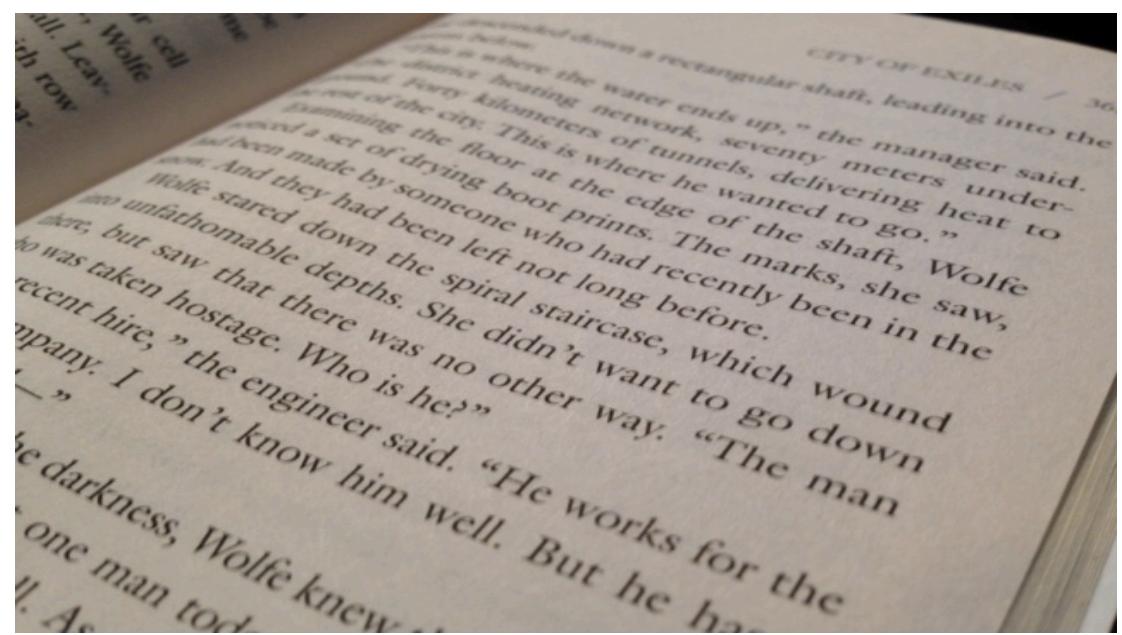
Instruction Set Architecture (ISA)

- 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, ...



Assembly Language

9/7/17



High-Level Language

6



Assembly Language

9/7/17

High-Level Language

7

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)

http://inst.eecs.berkeley.edu/~cs61c/resources/RISCV_Green_Sheet.pdf

Reference Data					
RV64 BASE INTEGER INSTRUCTIONS, in alphabetical order					
Mnemonic	FMT	NAME	DESCRIPTION (in Verilog)	NOTE	
add, addw	R	ADD (Word)	$R[\text{rd}] = R[\text{rs1}] + R[\text{rs2}]$	1)	
addi, addiw	I	ADD Immediate (Word)	$R[\text{rd}] = R[\text{rs1}] + \text{imm}$	1)	
and	R	AND	$R[\text{rd}] = R[\text{rs1}] \& R[\text{rs2}]$		
andi	I	AND Immediate	$R[\text{rd}] = R[\text{rs1}] \& \text{imm}$		
auipc	U	Add Upper Immediate to PC	$R[\text{rd}] = \text{PC} + [\text{imm}, 12'b0]$		
beq	SB	Branch Equal	$\text{if } R[\text{rs1}] == R[\text{rs2}] \text{ then } \text{PC} = \text{PC} + [\text{imm}, 1b'0]$		
bge	SB	Branch Greater than or Equal	$\text{if } R[\text{rs1}] >= R[\text{rs2}] \text{ then } \text{PC} = \text{PC} + [\text{imm}, 1b'0]$		
bgeu	SB	Branch Greater than Unsigned	$\text{if } R[\text{rs1}] > R[\text{rs2}] \text{ then } \text{PC} = \text{PC} + [\text{imm}, 1b'0]$	2)	
blt	SB	Branch Less Than	$\text{if } R[\text{rs1}] < R[\text{rs2}] \text{ then } \text{PC} = \text{PC} + [\text{imm}, 1b'0]$		
bltu	SB	Branch Less Than Unsigned	$\text{if } R[\text{rs1}] < R[\text{rs2}] \text{ then } \text{PC} = \text{PC} + [\text{imm}, 1b'0]$	2)	
bne	SB	Branch Not Equal	$\text{if } R[\text{rs1}] != R[\text{rs2}] \text{ then } \text{PC} = \text{PC} + [\text{imm}, 1b'0]$		
carro	I	Cont./Stat.RegRead&Clear	$R[\text{rd}] = \text{CSR} \text{CSR} = \text{CSR} \& \sim \text{imm}$		
carrci	I	Cont./Stat.RegRead&Clear imm	$R[\text{rd}] = \text{CSR} \text{CSR} = \text{CSR} \& \sim \text{imm}$		
csrrs	I	Cont./Stat.RegRead&Set	$R[\text{rd}] = \text{CSR}; \text{CSR} = \text{CSR} R[\text{rs1}]$		
csrrsi	I	Cont./Stat.RegRead&Set imm	$R[\text{rd}] = \text{CSR}; \text{CSR} = \text{CSR} \text{imm}$		
carwr	I	Cont./Stat.RegRead&Write	$R[\text{rd}] = \text{CSR}; \text{CSR} = R[\text{rs1}]$		
carriw	I	Cont./Stat.RegRead&Write imm	$R[\text{rd}] = \text{CSR}; \text{CSR} = \text{imm}$		
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	$R[\text{rd}] = \text{PC} + 4; \text{PC} = \text{PC} + [\text{imm}, 1b'0]$		
jalr	UJ	Jump & Link Register	$R[\text{rd}] = \text{PC} + 4; \text{PC} = R[\text{rs1}] + \text{imm}$	3)	
lb	I	Load Byte	$R[\text{rd}] = [56'bM[7], M[R[\text{rs1}] + \text{imm}][7:0]]$	4)	
lbu	I	Load Byte Unsigned	$R[\text{rd}] = [56'b0, M[R[\text{rs1}] + \text{imm}][7:0]]$		
ld	I	Load Doubleword	$R[\text{rd}] = M[R[\text{rs1}] + \text{imm}][63:0]$		
lh	I	Load Halfword	$R[\text{rd}] = [48'bM[15], M[R[\text{rs1}] + \text{imm}][15:0]]$	4)	
lbu	I	Load Halfword Unsigned	$R[\text{rd}] = [48'b0, M[R[\text{rs1}] + \text{imm}][15:0]]$		
lui	U	Load Upper Immediate	$R[\text{rd}] = [32'bimm<31>, imm, 12'b0]$		
lw	I	Load Word	$R[\text{rd}] = [32'bM[31], M[R[\text{rs1}] + \text{imm}][31:0]]$	4)	
lwu	I	Load Word Unsigned	$R[\text{rd}] = [32'b0, M[R[\text{rs1}] + \text{imm}][31:0]]$		
or	R	OR	$R[\text{rd}] = R[\text{rs1}] \mid R[\text{rs2}]$		
ori	I	OR Immediate	$R[\text{rd}] = R[\text{rs1}] \mid \text{imm}$		
sb	S	Store Byte	$M[R[\text{rs1}] + \text{imm}][7:0] = R[\text{rs2}][7:0]$		
sd	S	Store Doubleword	$M[R[\text{rs1}] + \text{imm}][47:32] = R[\text{rs2}][47:32]$		
sh	S	Store Halfword	$M[R[\text{rs1}] + \text{imm}][15:0] = R[\text{rs2}][15:0]$		
sl, sll	R	Shift Left (Word)	$R[\text{rd}] = R[\text{rs1}] << R[\text{rs2}]$	1)	
sll, sllw	I	Shift Left Immediate (Word)	$R[\text{rd}] = R[\text{rs1}] << \text{imm}$		
slt	R	Set Less Than	$R[\text{rd}] = R[\text{rs1}] < R[\text{rs2}] ? 1 : 0$		
slti	I	Set Less Than Immediate	$R[\text{rd}] = R[\text{rs1}] < \text{imm} ? 1 : 0$		
sltiu	I	Set Less Than Unsigned	$R[\text{rd}] = R[\text{rs1}] < R[\text{rs2}] ? 1 : 0$	2)	
sr, sraw	R	Shift Right Arithmetic (Word)	$R[\text{rd}] = R[\text{rs1}] >> R[\text{rs2}]$	1,5)	
sr1, srw1	I	Shift Right Arith Imm (Word)	$R[\text{rd}] = R[\text{rs1}] >> \text{imm}$	1,5)	
sr1, srwl	R	Shift Right (Word)	$R[\text{rd}] = R[\text{rs1}] >> R[\text{rs2}]$	1)	
sr1, srwl	I	Shift Right Immediate (Word)	$R[\text{rd}] = R[\text{rs1}] >> \text{imm}$	1)	
sub, subw	R	SUBtract (Word)	$R[\text{rd}] = R[\text{rs1}] - R[\text{rs2}]$		
sw	S	Store Word	$M[R[\text{rs1}] + \text{imm}[31:0]] = R[\text{rs2}][31:0]$		
xor	R	XOR	$R[\text{rd}] = R[\text{rs1}] \wedge R[\text{rs2}]$		
xori	I	XOR Immediate	$R[\text{rd}] = R[\text{rs1}] \wedge \text{imm}$		
Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit register 2) Operation assumes unsigned integers (instead of 2's complement) 3) The least significant bit of the branch address in jal 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) Replicates with one operand signed and one unsigned 7) The word version does a single-precision operation using the rightmost 32 bits of a 64-bit register 8) Classify writes a 10-bit mask to show which properties are true (e.g., $-inf$, 0 , $+inf$, $denom$, ...) 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					
RV64M Multiply Extension					
Mnemonic	FMT	Name	FMT NAME	DESCRIPTION (in Verilog)	NOTE
mul, mulw	R	MULtiply (Word)	R[rd] = (R[n1]*1) & R[n2](32'b0)		1)
mulh	R	MULtiply upper Half	R[rd] = (R[n1]*1) & R[n2](32'b127:64)		
mulhu	R	MULtiply upper Half Sign/Um	R[rd] = (R[n1]*1) & R[n2](32'b127:64)		6)
mulhsu	R	MULtiply upper Half Sign/Um	R[rd] = (R[n1]*1) & R[n2](32'b127:64)		2)
div, divw	R	Divide (Word)	R[rd] = (R[n1]*1) / R[n2]		
divu	R	Divide Unsigned	R[rd] = (R[n1]*1) / R[n2]		1)
rem, remw	R	REmainder (Word)	R[rd] = (R[n1]*1) % R[n2]		2)
remu, remuw	R	REmainder Unsigned (Word)	R[rd] = (R[n1]*1) % R[n2]		1)
RV64F and RV64D Floating-Point Extensions					
fld, fhw	I	Load (Word)	F[rd] = M[R[n1]*1] & imm		
fsd, fsw	S	Store (Word)	M[R[n1]*1] = imm & F[rd]		1)
fadd, s, fadd.d	R	ADD	F[rd] = F[n1]*1 + F[n2]		
fsub, s, fsub.d	R	SUBtract	F[rd] = (F[n1]*1) - F[n2]		7)
fmul, s, fmul.d	R	MULtiply	F[rd] = F[n1]*1 * F[n2]		
fdiv, s, fdiv.d	R	DIVide	F[rd] = F[n1]*1 / F[n2]		
fsqrt, s, fsqrt.d	R	SQure Root	F[rd] = sqrt(F[n1]*1)		
fmadd, s, fmadd.d	R	Mul+Add	F[rd] = F[n1]*1 * F[n2] + F[n3]		
fmsub, s, fmsub.d	R	Mul-SUBtract	F[rd] = F[n1]*1 * F[n2] - F[n3]		
fmadd.s, fmadd.d	R	Negative Mul+Subtract	F[rd] = -(F[n1]*1) * F[n2] - F[n3]		
fmadd.d, s, fmadd.d	R	Negative Mul+Add	F[rd] = -(F[n1]*1) * F[n2] + F[n3]		
fsgnj, s, fsgnj.d	R	SIGN source	F[rd] = (-1)^F[n1]*1		
fsgnjn, s, fsgnjn.d	R	Negative SIGN source	F[rd] = (-1)^F[n1]*1		
fsgnjx, s, fsgnjx.d	R	Xor SIGN source	F[rd] = F[rd] & (F[n1]*1 & F[n2])		
fmin, s, fmin.d	R	MINimum	F[rd] = (F[n1]*1 < F[n2]) ? F[n1] : F[n2]		7)
fmax, s, fmax.d	R	MAXimum	F[rd] = (F[n1]*1 > F[n2]) ? F[n1] : F[n2]		
fcpq, s, fcpq.d	R	Compare Float Equal	R[rd] = (F[n1]*1 == F[n2])*1		
fclt, s, fclt.d	R	Compare Float Less Than	R[rd] = (F[n1]*1 < F[n2])*1		
fclt.e, fclt.d	R	Compare Float Less than or =	R[rd] = (F[n1]*1 <= F[n2])*1		
fclass, s, fclass.d	R	Classify Type	R[rd] = class(F[n1]*1)		7,8)
fmv.x, s, fmv.x, d	R	Move from Integer	R[rd] = F[n1]		
fcvt.w, s, fcvt.w, d	R	Move to Integer	R[rd] = F[n1]		
fcvt.d, s, fcvt.d, w	R	Convert from DP to SP	F[rd] = single(F[n1]*1)		
fcvt.s, w, fcvt.d, w	R	Convert from SP to DP	F[rd] = double(F[n1]*1)		
fcvt.t, s, fcvt.d, w	R	Convert from 32b Integer	F[rd] = float(R[n1]*1 31:0)		
fcvt.t, s, fcvt.d, l	R	Convert from 64b Integer	F[rd] = float(R[n1]*1 63:0)		
fcvt.s, w, fcvt.d, dw	R	Convert from 32b Int (Unsigned)	F[rd] = float(R[n1]*1 31:0)		2,7)
fcvt.t, s, w, fcvt.d, lu	R	Convert from 64b Int (Unsigned)	F[rd] = float(R[n1]*1 63:0)		2,7)
fcvt.t, s, w, fcvt.w, d	R	Convert to 32b Integer	R[rd] = F[n1]*1 & integer(F[n1]*1)		
fcvt.t, s, w, fcvt.w, u	R	Convert to 64b Integer	R[rd] = F[n1]*1 & integer(F[n1]*1)		2,7)
fcvt.t, s, w, fcvt.u, ld	R	Convert to 32b Int Unsigned	R[rd] = F[n1]*1 & integer(F[n1]*1)		2,7)
fcvt.t, s, w, fcvt.u, lu	R	Convert to 64b Int Unsigned	R[rd] = F[n1]*1 & integer(F[n1]*1)		2,7)
RV64A Atomic Extension					
amadd, w, amadd.d	R	ADD	R[rd] = M[R[n1]*1]		9)
amand, w, amand.d	R	AND	M[R[n1]*1] & M[R[n2]*1] = R[rd]		9)
ammax, w, amax.d	R	MAXimum	M[R[n1]*1] & M[R[n2]*1] = R[rd]		9)
ammaxu, w, amaxu.d	R	MAXimum Unsigned	M[R[n1]*1] & M[R[n2]*1] = R[rd]		2,9)
amomin, w, amomin.d	R	MINimum	R[rd] = M[R[n1]*1]		9)
amominu, w, amominu.d	R	MINimum Unsigned	R[rd] = M[R[n1]*1]		2,9)
amoor, w, amoord	R	OR	R[rd] = M[R[n1]*1]		9)
amowap, w, amowap.d	R	SWAP	M[R[n1]*1] & M[R[n2]*1] = R[rd]		9)
amoxor, w, amoxor.d	R	XOR	R[rd] = M[R[n1]*1] ^ M[R[n2]*1]		9)
lr, w, lr.d	R	Load Reserved	R[rd] = M[R[n1]*1], reservation = R[n2]		
sc, w, sc.d	R	Store Conditional	R[rd] = M[R[n1]*1] + R[n2]		
CORE INSTRUCTION FORMATS					
31	27	26	25	20	19
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[12:0:5]		rs2	rs1	funct3 imm[4:1][11] opcode
U	imm[31:12]				rd opcode
	imm[20:10][11:1][9:12]				

Inspired by the IBM 360 “Green Card”

IBM System/360 Reference Data					
MACHINE INSTRUCTIONS					
NAME	MNEMONIC	CODE	OP	RD	OPERANDS
Add (c)	AR	1A	RR	R1,R2	
Add (c)	A	5A	RX	R1,D20K2,B21	
Add Decimal (c,d)	AP	FA	SS	D11L,B11,D21L2,B21	
Add Halfword (c)	AH	4A	RX	R1,D20K2,B21	
Add Logical (c)	ALR	1E	RR	R1,R2	
Add Logical (c)	AL	5E	RX	R1,D20K2,B21	
AND (c)	NR	14	RR	R1,R2	
AND (c)	N	54	RX	R1,D20K2,B21	
AND (c)	NI	94	SI	D11B11,I2	
AND (c)	NC	04	SS	D11L,B10,D21B21	
Branch and Link	BALN	05	RR	R1,R2	
Branch and Link	BAL	45	RX	R1,D20K2,B21	
Branch and Store (c)	BASR	0D	RR	R1,R2	
Branch and Store (c)	BAS	4D	RX	R1,D20K2,B21	
Branch on Condition	BCR	07	RR	M1,R2	
Branch on Condition	BC	47	RX	M1,D20K2,B21	
Branch on Count	BCTR	06	RR	R1,R2	
Branch on Count	BCT	46	RX	R1,D20K2,B21	
Branch on Index High	BXH	86	RS	R1,R3,D21B21	
Branch on Index Low or Equal	BXL	87	RS	R1,R3,D21B21	
Compare (c)	CR	19	RR	R1,R2	
Compare (c)	C	59	RX	R1,D20K2,B21	
Compare Decimal (c,d)	CP	F9	SS	D11L,B11,D21L2,B21	
Compare Halfword (c)	CH	49	RX	R1,D20K2,B21	
Compare Logical (c)	CLR	15	RR	R1,R2	
Compare Logical (c)	CL	55	RX	R1,D20K2,B21	
Compare Logical (c)	CLC	05	SS	D11L,B10,D21B21	
Compare Logical (c)	CLI	95	SI	D11B11,I2	
Convert to Binary	CVB	4F	RX	R1,D20K2,B21	
Convert to Decimal	CVD	4E	RX	R1,D20K2,B21	
Diagnose (c)	BD	SI			
Divide	DR	1D	RR	R1,R2	
Divide	D	5D	RX	R1,D20K2,B21	
Divide Decimal (c)	DP	FD	SS	D11L,B11,D21L2,B21	
Edit (c)	ED	0E	SS	D11L,B10,D21B21	
Edit and Mark (c,d)	EDMK	DF	SS	D11L,B10,D21B21	
Exclusive OR (c)	XR	17	RR	R1,R2	
Exclusive OR (c)	X	57	RX	R1,D20K2,B21	
Exclusive OR (c)	XI	97	SI	D11B11,I2	
Exclusive OR (c)	XC	07	SS	D11L,B10,D21B21	
Execute	EX	44	RX	R1,D20K2,B21	
Halt I/O (c)	HIO	9E	SI	D11B10	
Insert String Key (c,p)	ISK	42	RX	R1,D20K2,B21	
Load	LR	18	RR	R1,R2	
Load	L	58	RX	R1,D20K2,B21	
Load Address	LA	41	RX	R1,D20K2,B21	
Load and Test (c)	LTR	12	RR	R1,R2	
Load Complement (c)	LCR	13	RR	R1,R2	
Load Halfword	LH	46	RX	R1,D20K2,B21	
Load Multiple	LM	96	RS	R1,R3,D21B21	
Load Multiple Control (c,p)	LMC	86	RS	R1,R3,D21B21	
Load Negative (c)	LNR	11	RR	R1,R2	
Load Positive (c)	LPR	56	RR	R1,R2	
Load PSW In,p0	LPSW	82	SI	D11B10	
Load Real Address In,p1	LRA	81	RX	R1,D20K2,B21	
Move	MVI	92	SI	D11B11,I2	
Move	MVC	D2	SS	D11L,B11,D21B21	
Move Numerics	MVN	D1	SS	D11L,B11,D21B21	
Move with Offset	MVO	F1	SS	D11L,B11,D21L2,B21	
Move Zones	MVZ	D3	SS	D11L,B11,D21B21	
Multiply	M	5C	RX	R1,D20K2,B21	
Multiply	MP	FC	SS	D11L,B11,D21L2,B21	
Multiply Decimal (c)	MH	4C	RX	R1,D20K2,B21	
Multiply Halfword	DR	16	RR	R1,R2	
DR (c)	D	56	RX	R1,D20K2,B21	
DR (c)	DI	96	SI	D11B11,I2	

Outline

- Assembly Language
- **RISC-V Architecture**
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion ...



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



Platinum:



Berkeley
Architecture
Research



Rambus

Cryptography Research



Gold, Silver, Auditors:



Esperanto
Technologies



MEDIATEK



Rumble

Development
Technolution



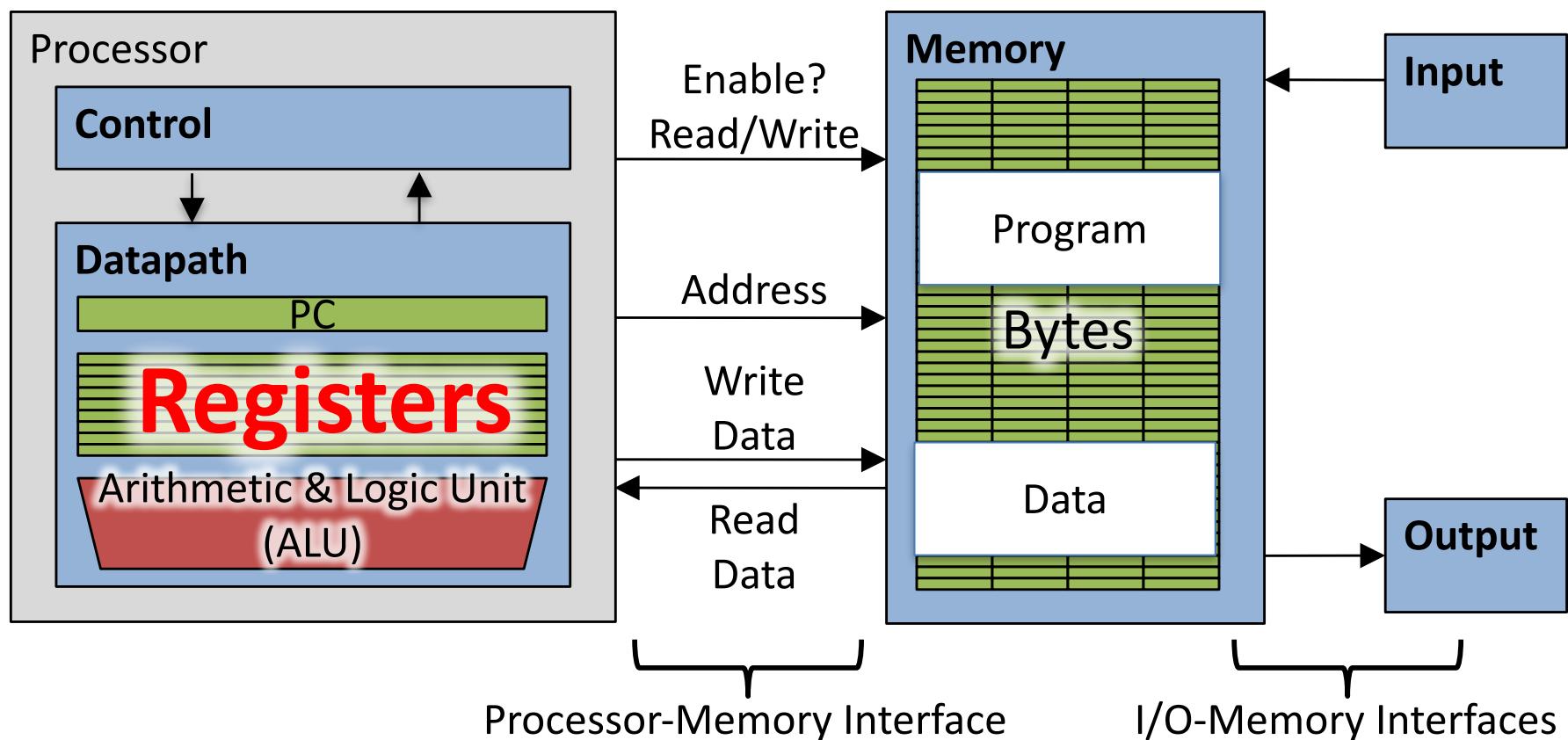
Outline

- Assembly Language
- RISC-V Architecture
- **Registers vs. Variables**
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion ...

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

Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- **RISC-V Instructions**
- C-to-RISC-V Patterns
- And in Conclusion ...

RISC-V Instruction Assembly Syntax

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

Operation code (opcode)

Destination register

First operand register

Second operand register

is assembly comment syntax

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

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)

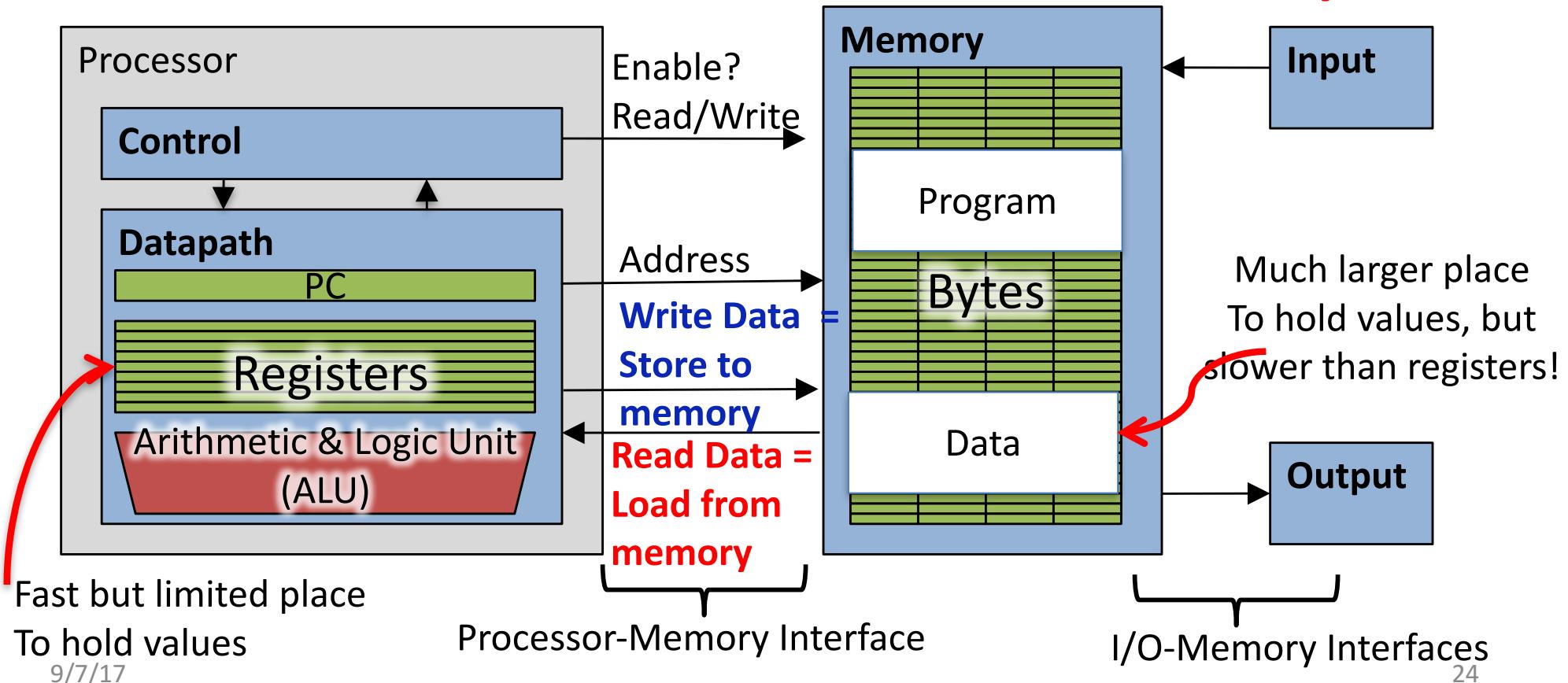
where RISC-V registers `x3, x4` are associated with C variables `f, g`

- Syntax similar to add instruction, except that last argument is a number instead of a register

`add x3, x4, x0` (in RISC-V)

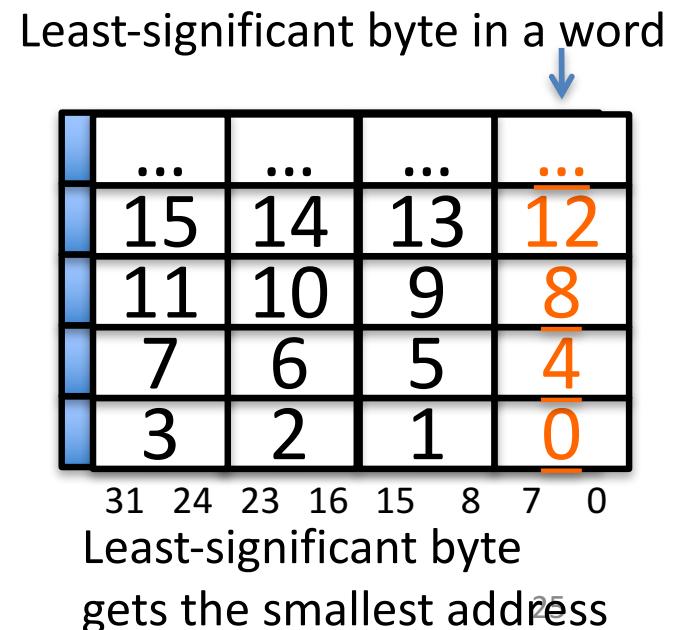
$f = g$ (in C)

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)



Transfer from Memory to Register

- 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

- C code

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

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

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

Note: x13 – base register (pointer)
 12, 40 – offsets 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`
- 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 extend to fill register. Why no unsigned store byte `sbu`?



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

Administrivia

- HW #0 due tomorrow night!
- HW #1 will be published soon
 - Two-part C programming assignment
- Small Group Tutoring sign ups will be out right after today's lecture
- Three weeks to Midterm #1!

Break!



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 x_{11} the value from x_{12} shifted 2 bits to the left (they fall off end), inserting 0's on right; $<<$ in C

Before: 0000 0002_{hex}

0000 0000 0000 0000 0000 0000 0000 0010_{two}

After: 0000 0008_{hex}

0000 0000 0000 0000 0000 0000 0000 1000_{two}

What arithmetic effect does shift left have?

- Shift Right Logical: `srlt` 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
 $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_2 = -25_{10}$
- If execute sra x10, x10, 4, result is:
 $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$
- 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* (**b_{eq}**) or branch *if not equal* (**b_{ne}**)
 - Also branch if less than (**b_{lt}**) and branch if greater than or equal (**b_{ge}**)
- **Unconditional Branch** – always branch
 - a RISC-V instruction for this: *jump* (**j**)

Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion ...

Example *if* Statement

- Assuming translations below, compile *if* block

$f \rightarrow x10$ $g \rightarrow x11$ $h \rightarrow x12$
 $i \rightarrow x13$ $j \rightarrow x14$

```
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

$f \rightarrow x10$ $g \rightarrow x11$ $h \rightarrow x12$

$i \rightarrow x13$ $j \rightarrow x14$

```
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
```

Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion ...

In Conclusion,...

- Instruction set architecture (ISA) specifies the set of commands (instructions) a computer can execute
- Hardware registers provide a few very fast variables for instructions to operate on
- RISC-V ISA requires software to break complex operations into a string of simple instructions, but enables faster, simple hardware
- Assembly code is human-readable version of computer's native machine code, converted to binary by an *assembler*