EE-222: Microprocessor Systems

RISC vs. CISC Architectures

Instructor: Dr. Arbab Latif

NUST

SCHOOL OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE (SEECS)

Below Your Program: Levels of Abstraction

Higher-Level Language Program (e.g. C)

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

Compiler

Assembly Language Program (e.g. RISCV)

```
lw    x5, 0(x2)
lw    x6, 4(x2)
sw    x6, 0(x2)
```

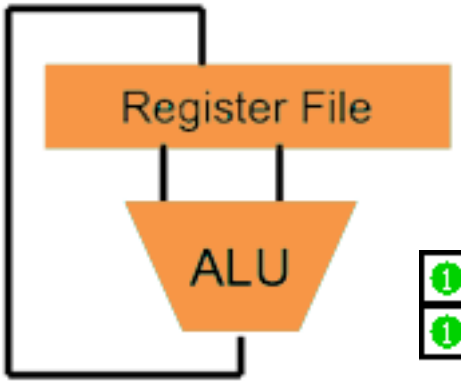
EVERY INSTRUCTION IS DIRECTLY IMPLEMENTED IN THE HARDWARE

Ma… Program (RISCV)

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

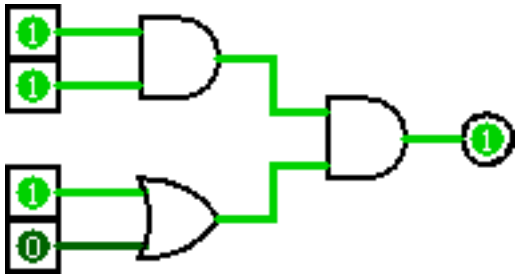
Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Think about It!

- Every Instruction is Directly Implemented in the Hardware:
  - Think in terms of CISC vs. RISC

CISC	RISC
MULT 2:3, 5:2	**LOAD A, 2:3** **LOAD B, 5:2** **PROD A, B** **STORE 2:3, A**

  - CISC:
    - Variable size instructions -> Complex instruction decoder -> More transistors
    - Multi-cycle instructions
  - RISC:
    - Simplified instructions -> simple instruction decoder
    - Single-cycle instructions (mostly)

Changing the Architecture CISC vs. RISC

- Early trend:
  - Add more and more instructions to do elaborate operations – *Complex Instruction Set Computing* (CISC)
  - difficult to learn and comprehend language
  - super-complicated (slow?) hardware

- Later on:
  - Opposite philosophy later began to dominate: *Reduced Instruction Set Computing* (RISC)
  - Simpler (and smaller) instruction set makes it easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones

RISC Architecture

- Feature 1: RISC processors have a fixed instruction size.
  - It makes the task of instruction decoder easier.
    - In AVR the instructions are 2 or 4 bytes.
  - In CISC processors instructions have different lengths
    - E.g. in 8051
      - CLR C ; a 1-byte instruction
      - ADD A, #20H ; a 2-byte instruction
      - LJMP HERE; a 3-byte instruction

RISC architecture

- Feature 2: Reduce the number of instructions
  - Pros: Reduces the number of used transistors
  - Cons:
    - Can make the assembly programming more difficult
    - Can lead to using more memory

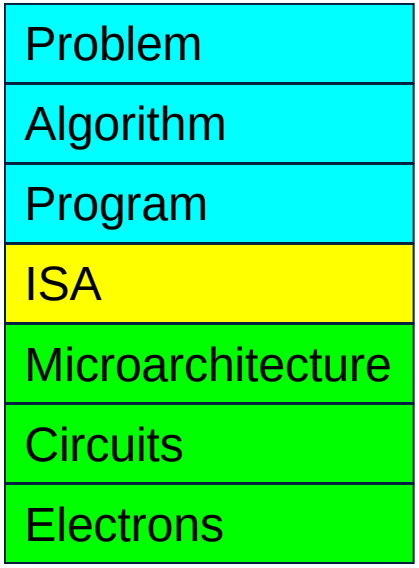
Feature 3: more than 95% of instructions are executed in 1 machine cycle

What is Instruction Set Architecture (ISA)?

The Instruction Set Architecture

- The ISA is the interface between what the software commands and what the hardware carries out
  - i.e the vocabulary of commands understood by a given architecture

- The ISA specifies
  - The memory organization
    - Instruction address space (ATmega16a: 2^{13})
    - Word- or Byte-addressable
  - The register set
    - 32x8 GPR registers in AVR
  - The instruction set
    - Opcodes
    - Data types
    - Addressing modes
    - Semantics of instructions



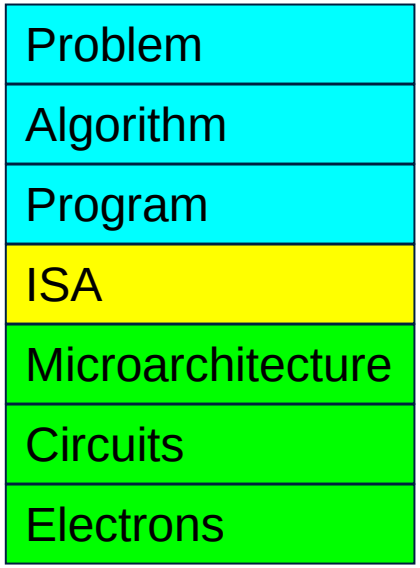
Microarchitecture

- An **implementation** of the ISA:
  - Under specific design constraints and goals

- Anything done in hardware without exposure to software
  - Pipelining
  - In-order versus out-of-order instruction execution
  - Memory access scheduling policy
  - Speculative execution
  - Superscalar processing (multiple instruction issue?)
  - Clock gating
  - Caching? Levels, size, associativity, replacement policy
  - Prefetching?
  - Voltage/frequency scaling?
  - Error correction?

Wrap-up: ISA vs. Microarchitecture

- ISA
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs
- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - **ISA, uarch**, circuits
  - "Architecture" = ISA + microarchitecture



Introduction to RISC-V

Money Makes the Mare Go!

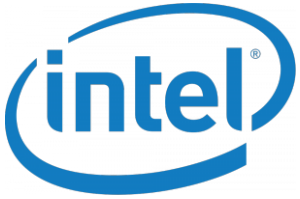
Choose a Licensing Plan That's Right for Your Business

	 Arm DesignStart	 Arm Flexible Access	 Standard Licensing
	Access a select mix of IP products and tools	Access IP products, support, tools and training	Full product portfolio, support and training
	Learn More >	Learn More >	Learn More >
Upfront Costs	**$0** for the Cortex-M0, Cortex-M1 and Cortex-M3 CPUs **$75K** for the Cortex-A5 CPU	**$75K** entry package annual access fee **$200K** standard package annual access fee	Upfront license fees based on license terms
Licensing	No additional license fee for the Cortex-M0, Cortex-M1, and Cortex-M3 CPUs $50K license fee for the Cortex-A5 CPU at tape out	Per product license fees due at tape out	Upfront license fees based on license terms

https://www.arm.com/why-arm/how-licensing-works

What if there were free and
open ISAs we could use for everything?

Welcome to RISC-V



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

Macbooks & PCs
(Core i3, i5, i7, M)
x86 Instruction Set

ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

RISC-V

Designer	University of California, Berkeley
Bits	32, 64, 128
Introduced	2010
Version	2.2
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little

Versatile and open-source
Relatively new, designed for cloud computing, high-end phones, small embedded sys.
RISCV Instruction Set

What is RISC-V?

- RISC-V (pronounced "risk-five")
  - Is an open specification of an Instruction Set Architecture (ISA) based on reduced instruction set computing (RISC) 5th iteration:
    - Describes the way in which software talks to an underlying processor
  - There was RISC-I, II, III, IV before

- Most ISAs: X86, ARM, Power, MIPS, SPARC
  - Commercially protected by patents
  - Preventing practical efforts to reproduce the computer systems.

- RISC-V is open
  - Permitting any person or group to construct compatible computers
  - Use associated software

https://riscv.org/

What's Different About RISC-V?

- Simple:
  - Far smaller than proprietary ISAs
  - 2500 pages for x86, ARMv8 manual vs 200 for RISC-V manual
    - 47 RISCV instructions vs 3500+ 80x86 instructions

- Clean-slate design:
  - 25 years later, so can learn from mistakes of predecessors
  - Avoids μarchitecture or technology-dependent features

What's Different About RISC-V?

- Modular:
  - A mandatory **Base integer ISA**
    - **I: Integer instructions:**
      - **ALU**
      - **Branches/jumps**
      - **Loads/stores**
  - **Standard Extensions**
    - **M: Integer Multiplication and Division**
    - **A: Atomic Instructions**
    - **F: Single-Precision Floating-Point**
    - **D: Double-Precision Floating-Point**

What's Different About RISC-V?

- Modular:

Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space

What's Different About RISC-V?

- Supports specialization:
  - Vast opcode space reserved
- Community designed:
  - Base and standard extensions finished
  - Grow via optional extensions vs. incremental required features
- RISC-V Foundation extends ISA for technical reasons:
  - vs. private corporation for internal (marketing) reasons

Class ISA is RISC-V

- RISC-V is a new free, simple, clean, extensible ISA developed at Berkeley for education and research
  - Both of the dominant ISAs (x86 and ARM) are too complex to use for teaching or research
- RISC-V Foundation manages standard riscv.org
- RISC-V has taken off commercially
  - Nvidia is using RISC-V in all future GPUs
  - Western Digital is using RISC-V in all future products
- Now upstream support for many tools (gcc, Linux, FreeBSD, …)
- Govt. India selected RISC-V as national ISA


Berkeley Architecture Research
DRAPER
bluespec
C-SKY
Mentor
A Siemens Business
LATTICE SEMICONDUCTOR
Mellanox TECHNOLOGIES
NXP
cortus
Rambus
Cryptography Research
SiFive
Micron
SAMSUNG
Google
QUALCOMM
HUAWEI
IBM
Western Digital
Microsemi
NVIDIA
RISC-V
ESPRESSIF
MEDIATEK
ANDES TECHNOLOGY
antmicro
EMBEDDED SYSTEMS
runtime.io
DOVER MICROSYSTEMS
BAE SYSTEMS
Esperanto Technologies
VeriSilicon
inside secure
DRIVING TRUST
GRAY RESEARCH
IDT
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
ISTUARY
INNOVATION GROUP
SEGGER
It simply works!
NUS
National University of Singapore
谨峰电子科技
WWW.GLORYSPACE.CN
GOWIN
GRIDOOG
"Eyes on the Grid, Eyes on the Buck"
LAUTERBACH
DEVELOPMENT TOOLS
INTRINSIX
ETH zürich
MIT CSAIL
ICT
中科院计算所
ASHLING
THE DEVELOPMENT SYSTEMS COMPANY
codasip
GREENWAVES TECHNOLOGIES
Roa Logic
Silicon Proven IP for FPGA and ASIC
SHC
SH CONSULTING
MINIMA
PROCESSOR
Rumble
Development
TRINAMIC
imperas
XtremeEDA
VectorBlox
embedded supercomputing
UBILITE
Hello IoT World
SEAGATE
EMDALO TECHNOLOGIES
Embedded Software Solutions
EMBECOSM
SECURE RF
Securing the Internet of Things
BERKELEY LAB
Syntacore
Custom cores and tools
lowRISC
TechanaLye
SIEMENS
优矽科技
UC TECH IP
Technolution
ultraSoC
csem
PRINCETON UNIVERSITY
Blockstream
dxcorr
GLOBALFOUNDRIES
expresslogic

Why RISC-V

- Open and free
- Not domain-specific
- To keep things simple, flexible and extensible
- No baggage of legacy

RISC-V ISA

RISC-V ISA Design Principle-1

- Design Principle 1: Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

- e.g. All arithmetic operations have same form
  - Two sources and one destination

- add a, b, c // a gets b + c

RISC-V ISA Design Principle-2

- Design Principle 2: Smaller is faster
  - memory is larger than no. of registers, use register operands

- e.g. Arithmetic operations use register operands and not direct memory

- most implementations have decoding the operands on the critical path so only 32 registers

RISC-V ISA Design Principle-3

- Design Principle 3: Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

- support for immediate operands,

- e.g. addi x22, x22, 4

RISC-V ISA Design Principle-4

- Design Principle 4: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

- e.g. R-format, I-format, S-format

RISC-V ISA

RISC-V – How Many Registers?

- RISC-V has a 32 × 64-bit register file [RV64]
  - RV32 has 32x32 register file
  - Use for frequently accessed data
  - 64-bit data is called a "doubleword"
    - 32 x 64-bit general purpose registers x0 to x30
  - 32-bit data is called a "word"

REGISTER NAME, USE, CALLING CONVENTION

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

RISC-V – How Many Registers?

REGISTER NAME, USE, CALLING CONVENTION

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

RISC-V Processor State

- Program counter (PC)
- 32 32/64-bit integer registers (x0-x31)
  - x0 always contains a 0
  - x1 to hold the return address on a call.
- 32 floating-point (FP) registers (f0-f31)
  - Each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)

REGISTER NAME, USE, CALLING CONVENTION

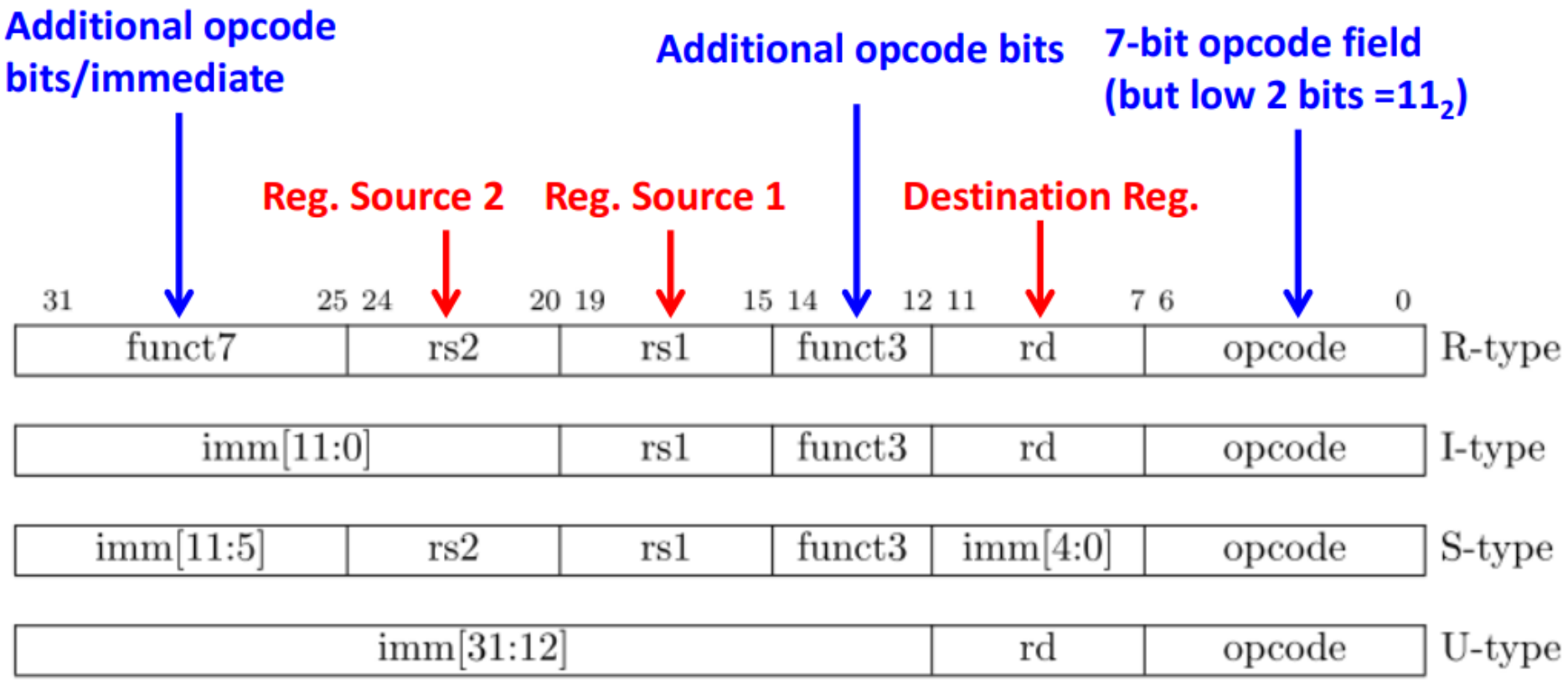
REGISTER	NAME	USE
x0	zero	The constant value 0
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporaries
x8	s0/fp	Saved register/Frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Function arguments/Return values
x12-x17	a2-a7	Function arguments
x18-x27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

Four Core RISC-V Instruction Formats

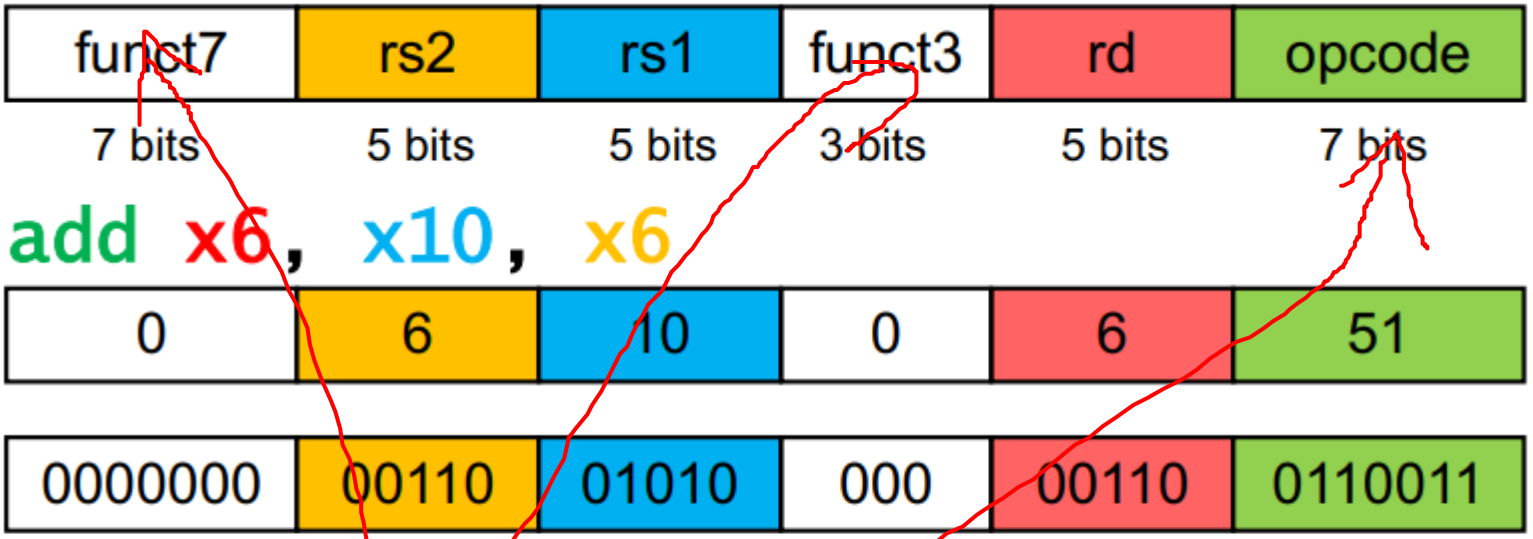
Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

Four Core RISC-V Instruction Formats

https://github.com/riscv/riscv-opcodes/blob/master/opcodes



R-Format Encoding Example



$0000\ 0000\ 0110\ 0101\ 0000\ 0011\ 0011\ 0011_{two} =$

00650333_{16}

Specifications and Software From riscv.org and github.com/riscv

- Specification from RISC-V website
  - https://riscv.org/specifications/
- RISC-V software includes
  - GNU Compiler Collection (GCC) toolchain (with GDB, the debugger)
    - https://github.com/riscv/riscv-tools
  - LLVM toolchain
  - A simulator ("Spike")
    - https://github.com/riscv/riscv-isa-sim
  - Standard simulator QEMU
    - https://github.com/riscv/riscv-qemu
- Operating systems support exists for Linux
  - https://github.com/riscv/riscv-linux
- A JavaScript ISA simulator to run a RISC-V Linux system on a web browser
  - https://github.com/riscv/riscv-angel

Class Tools

- Venus Simulator: https://venus.cs61c.org/

- 5-stage processor pipeline simulator and assembly code editor
  - https://github.com/mortbopet/Ripes

RISC-V Implementations

- A list from
  - https://riscv.org/risc-v-cores/
- The Indian IIT-Madras is developing six RISC-V open-source CPU designs (SHAKTI) for six distinct usages
  - https://shaktiproject.bitbucket.io/index.html

- SiFive HiFive Unleashed
  - First Linux RISC-V Board
    - First shipment: June 2018
  - https://www.sifive.com/
  - https://github.com/sifive/freedom

Recommended Reading

- RISC-V Edition - Computer Organization and Design: The Hardware Software Interface **(COD)** - David A. Patterson, John L. Hennessy:
  - Chapter-2

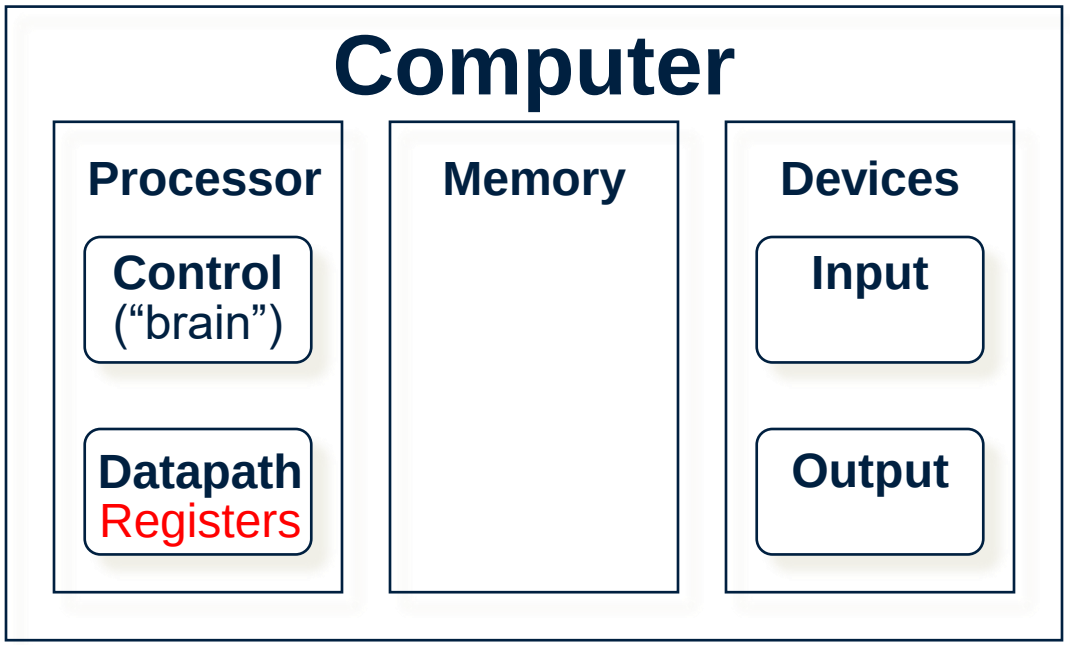
Variables in Hardware

- Already know instructions must be directly implemented in hardware

- What about variables?!?!
  - Live in Memory
  - Relative to your processor, memory is soooooo slooooooowwwwwwwwwwwwwww
  - *predetermined* small number of fast **registers** in processor to hold variables.

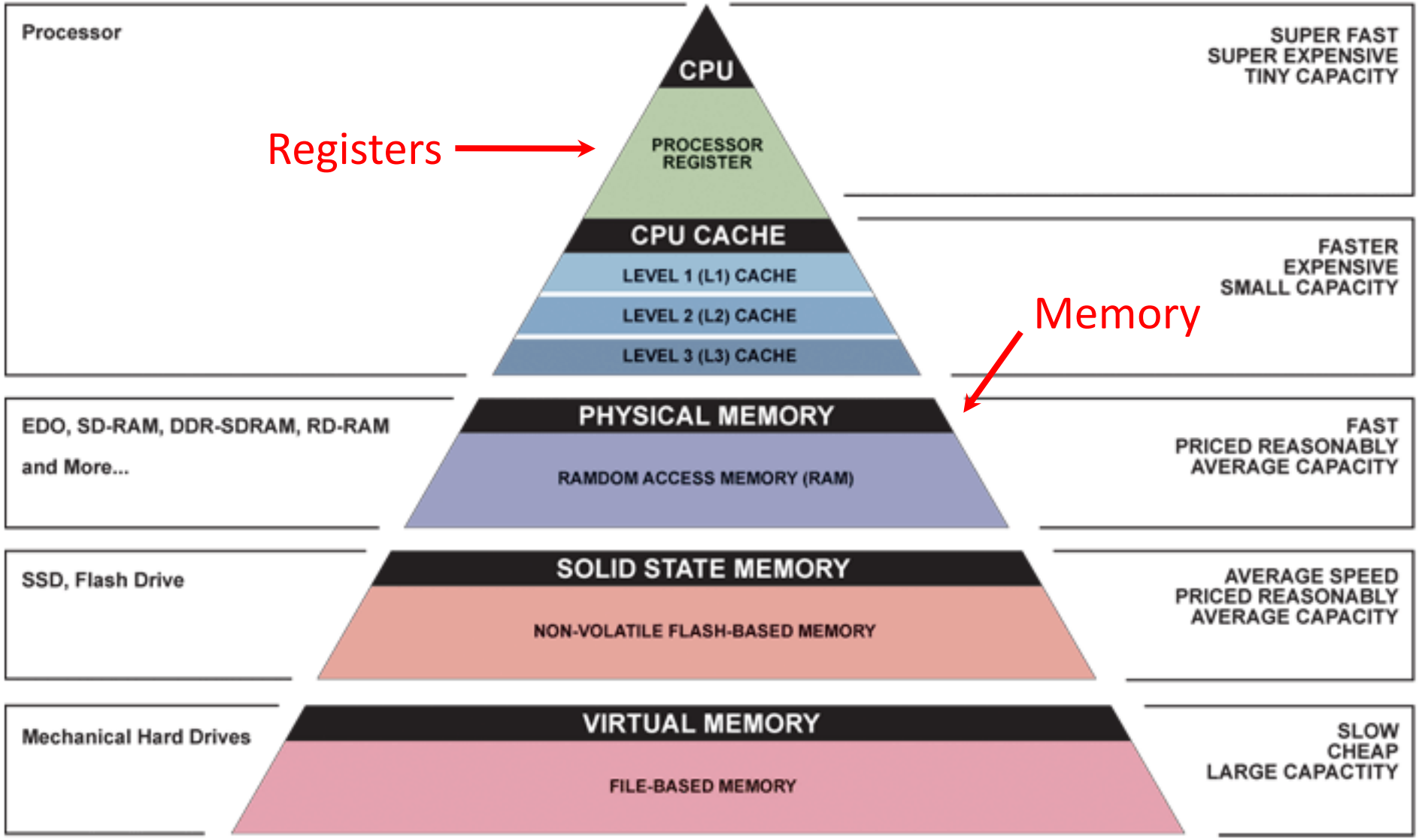
Registers in a Computer

- Five components of a computer!
  - Control
  - Datapath
  - Memory
  - Input
  - Output



- Registers are part of the Datapath

Memory Hierarchy



Register vs. Memory

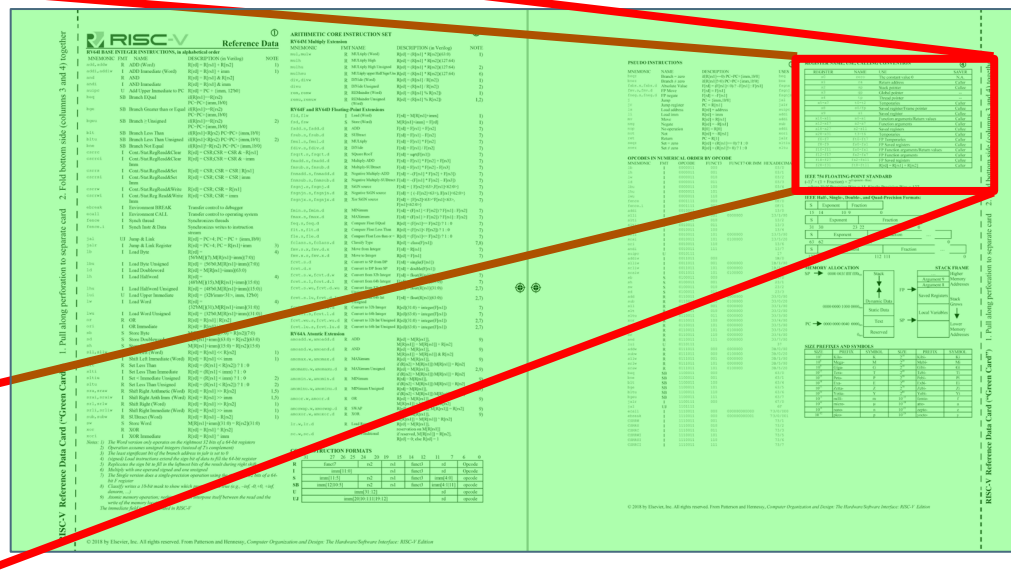
- What if more variables than registers?
  - Keep most frequently used in registers and move the rest to memory (called *spilling* to memory)
- Why not all variables in memory?
  - Smaller is faster: registers 100-500 times faster
  - Registers more versatile
    - In 1 arithmetic instruction: read 2 operands, perform 1 operation, and 1 write
    - In 1 data transfer instruction: 1 memory read/write, no operation

RISC-V – How Many Registers?

- Tradeoff between speed and availability
  - more registers → can house more variables simultaneously; all registers are slower.
- RISCV has 32 registers (x0-x31)
  - Each register is 32 bits wide and holds a word

REGISTER NAME, USE, CALLING CONVENTION

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



RISC-V – How Many Registers?

REGISTER NAME, USE, CALLING CONVENTION

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

RISC-V Registers

- Register denoted by '`x`' can be referenced by number (`x0-x31`) or name:
  - Registers that hold programmer variables:

    `s0-s1` ↔ `x8-x9`

    `s2-s11` ↔ `x18-x27`
  - Registers that hold temporary variables:

    `t0-t2` ↔ `x5-x7`

    `t3-t6` ↔ `x28-x31`
  - You'll learn about the other 13 registers later

- *Registers have no type* (C concept); the operation being performed determines how register contents are treated

RISC-V Processor State

- Program counter (PC)
- 32 32/64-bit integer registers (x0-x31)
  - x0 always contains a 0
  - x1 to hold the return address on a call.
- 32 floating-point (FP) registers (f0-f31)
  - Each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)

REGISTER NAME, USE, CALLING CONVENTION

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

Side Note: RV64

- RISC-V has a 32 × 64-bit register file [RV64]
  - RV32 has 32x32 register file
  - Use for frequently accessed data
  - 64-bit data is called a "doubleword"
    - 32 x 64-bit general purpose registers x0 to x30
  - 32-bit data is called a "word"

REGISTER NAME, USE, CALLING CONVENTION

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

Registers: Summary

- In high-level languages, number of variables limited only by available memory

- ISAs have a fixed, small number of operands called registers
  - Special locations built directly into hardware
  - **Benefit:** Registers are EXTREMELY FAST (faster than 1 billionth of a second)
  - **Drawback:** Operations can only be performed on these predetermined number of registers

Getting to Know RISC-V Assembly

RISC-V Manual Should Be Your Friend

http://inst.eecs.berkeley.edu/~cs61c/su18/img/riscvcard.pdf (Instruction Set)

http://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2016-1.pdf (Architecture)

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

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'b M[](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'b M[](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] = {32'b imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'b M[](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,sllw	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

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

flw,fld	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsw,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.o.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)

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

PSEUDO INSTRUCTIONS

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]	fsgnjx
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
jal	UJ	0000011	000		03/0
lb	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

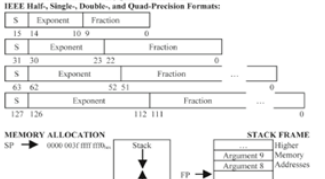
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5–x7	t0–t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10–x11	a0–a1	Function arguments/Return 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

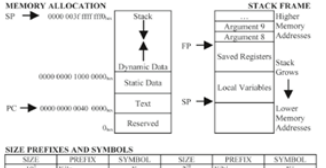
$(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

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:



MEMORY ALLOCATION



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

RISC-V Instructions

- Instruction Syntax

```
op dst, src1, src2
```

  - 1 operator, 3 operands
    - `op` = operation name (“operator”)
    - `dst` = register getting result (“destination”)
    - `src1` = first register for operation (“source 1”)
    - `src2` = second register for operation (“source 2”)

RISC-V Instructions

- One operation per instruction,
  at most one instruction per line

- Assembly instructions are related to C operations
  (=, +, –, *, /, &, |, etc.)
  - Must be, since C code decomposes into assembly!
  - A single line of C may break up into several lines of RISCV

Basic Arithmetic Instructions

RISCV Instructions Example

- Your very first instructions!
  (assume here that the variables `a`, `b`, and `c` are assigned to registers `s1`, `s2`, and `s3`, respectively)

- Integer Addition (`add`)
  - C: `a = b + c`
  - RISCV: `add s1, s2, s3`
- Integer Subtraction (`sub`)
  - C: `a = b - c`
  - RISCV: `sub s1, s2, s3`

RISCV Instructions Example

- Suppose `a` → `s0`, `b` → `s1`, `c` → `s2`, `d` → `s3` and `e` → `s4`. Convert the following C statement to RISCV:

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

```
add t1, s3, s4
add t2, s1, s2
sub s0, t2, t1
```

Ordering of instructions matters (must follow order of operations)

Utilize temporary registers

Comments

Comments in RISCV

- Comments in RISCV follow hash mark (#) until the end of line
  - Improves readability and helps you keep track of variables/registers!

```
add t1, s3, s4     # t1=d+e
add t2, s1, s2     # t2=b+c
sub s0, t2, t1     # a=(b+c)-(d+e)
```

x0 [ZERO]

The Zero Register

- Zero appears so often in code and is so useful that it has its own register!
- Register zero (`x0` or `zero`) always has the value 0 and cannot be changed!
  - i.e. any instruction with `x0` as `dst` has no effect
- Example Uses:
  - `add  s3, x0, x0  # c=0`
  - `add  s1, s2, x0  # a=b`

Immediates

Immediates

- Numerical constants are called immediates
- Why Immediate Instructions?
  - Immediate operand avoids a Load Instruction
- Separate instruction syntax for immediates:

```
opi dst, src, imm
```

  - Operation names end with 'i', replace 2nd source register with an immediate
- Example Uses:
  - `addi s1, s2, 5  # a=b+5`
  - `addi s3, s3, 1  # c++`
- Why no `subi` instruction?
  - subi dst, src, imm = addi dst, src, -imm.

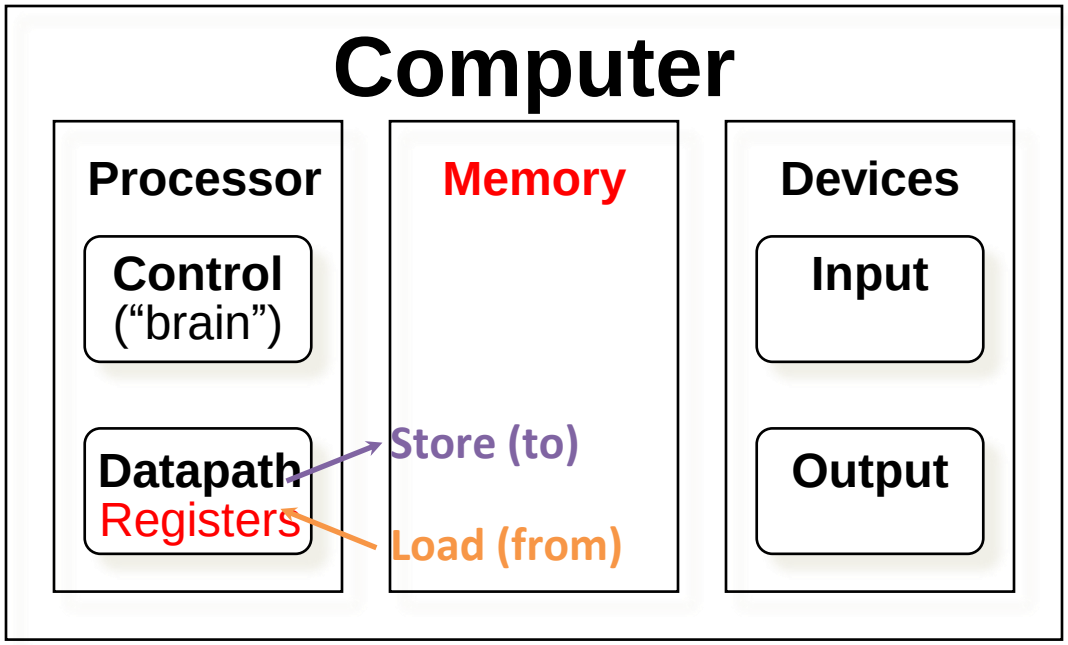
Recommended Reading

- RISC-V Edition - Computer Organization and Design_ The Hardware Software Interface - David A. Patterson, John L. Hennessy:
  - Chapter-2

Data Transfer Instructions

Data Transfer Instructions

- Data Transfer instructions are between registers (Datapath) and Memory
  - Allow us to fetch and store operands in memory



Data Transfer

- C variables map onto registers;
  What about large data structures like arrays?
  - Don't forget *memory*, our one-dimensional array indexed by addresses starting at 0

- RISCV instructions only operate on registers!

- Specialized data transfer instructions move data between registers and memory
  - Store (sw): register TO memory
  - Load (lw): register FROM memory

Data Transfer

- Instruction syntax for data transfer:

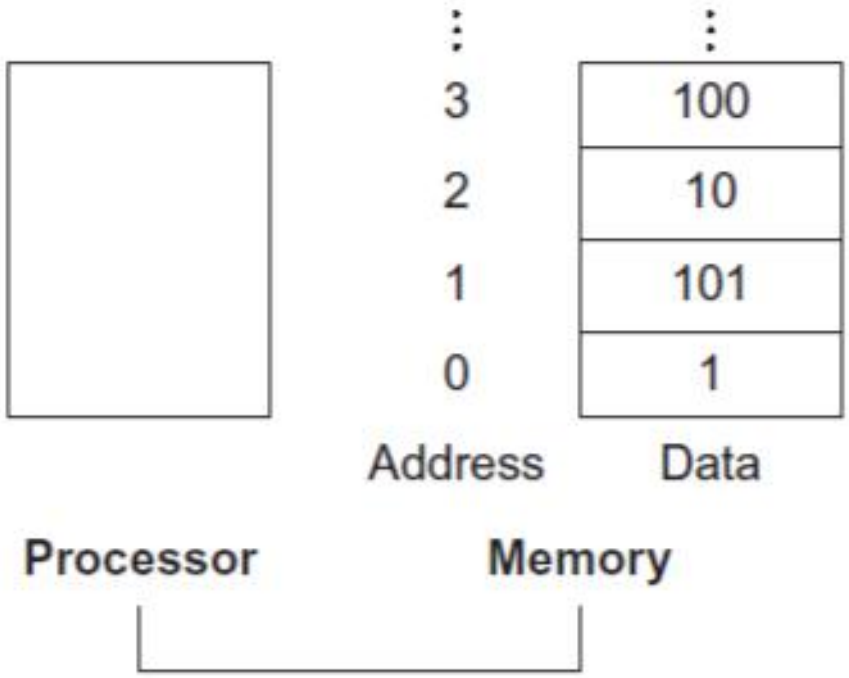
```
memop reg, off(bAddr)
```

  - `memop` = operation name ("operator")
  - `reg` = register for operation source or destination
  - `bAddr` = register with pointer to memory ("base address")
  - `off` = address offset (immediate) in bytes ("offset")

- Accesses memory at address `bAddr+off`
- **Reminder:** A register holds a word of raw data (no type) – make sure to use a register (and offset) that point to a valid memory address

Memory

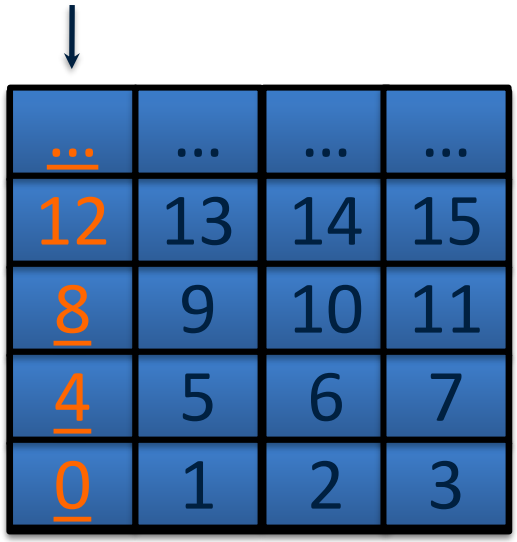
- Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0.



Memory is Byte-Addressed

- What was the smallest data type we saw in C?
  - A char, which was a *byte* (8 bits)
  - Everything in multiples of 8 bits (e.g. 1 word = 4 bytes)

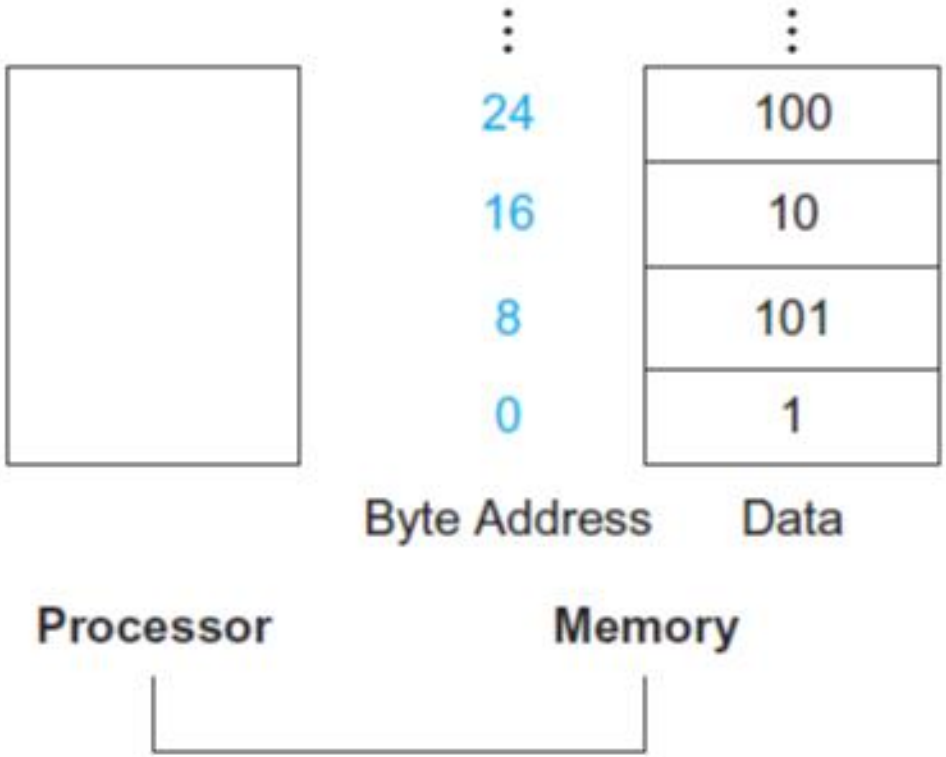
Assume here addr of lowest byte in word is addr of word



- Memory addresses are indexed by *bytes*, not words
- Word addresses are 4 bytes apart
  - Word addr is same as left-most byte
  - Addrs must be multiples of 4 to be "word-aligned"
- Pointer arithmetic not done for you in assembly
  - Must take data size into account yourself

Double-word Memory Address

- Double word addresses are 8 bytes



Data Transfer Instructions

- Load Word (`lw`)
  - Takes data at address `bAddr+off` FROM memory and places it into `reg`
- Store Word (`sw`)
  - Takes data in `reg` and stores it TO memory at address `bAddr+off`
- Example Usage:

```
# addr of int A[] -> s3, a -> s2
lw   t0,12(s3) # $t0=A[3]
add  t0,s2,t0 # $t0=A[3]+a
sw   t0,40(s3) # A[10]=A[3]+a
```

Remember! Pointer arithmetic not done for you in assembly
—Must take data size into account yourself

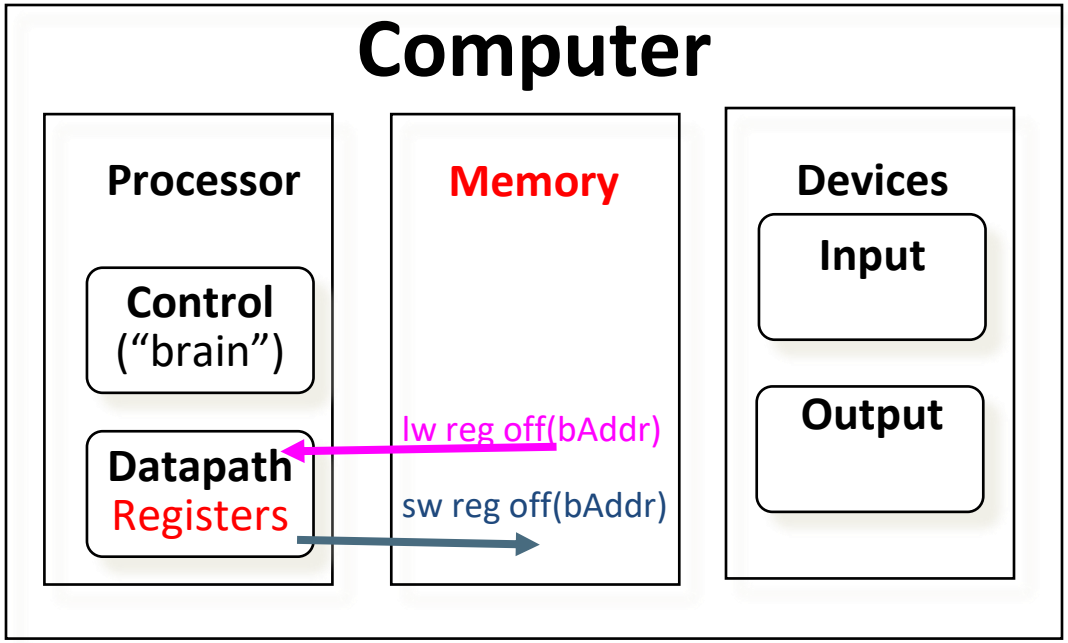
Recommended Reading

- RISC-V Edition - Computer Organization and Design_ The Hardware Software Interface - David A. Patterson, John L. Hennessy:
  - Chapter-2

Data Transfer Instructions Cont.…

Memory and Variable Size

- So Far:
  - lw reg, off(bAddr)
  - sw reg, off(bAddr)



- What about characters (1B) and shorts (sometimes 2B), etc?
  - Want to be able to use interact with memory values smaller than a word.

Trading Bytes with Memory (2 approaches)

- Method 1: Move words in and out of memory using bit-masking and shifting

```
lw   s0,0(s1)
andi s0,s0,0xFF # lowest byte
```

- Method 2: Load/store byte instructions

```
lb   s1,1(s0)
sb   s1,0(s0)
```

```
*(s0) = 0x00000180
```

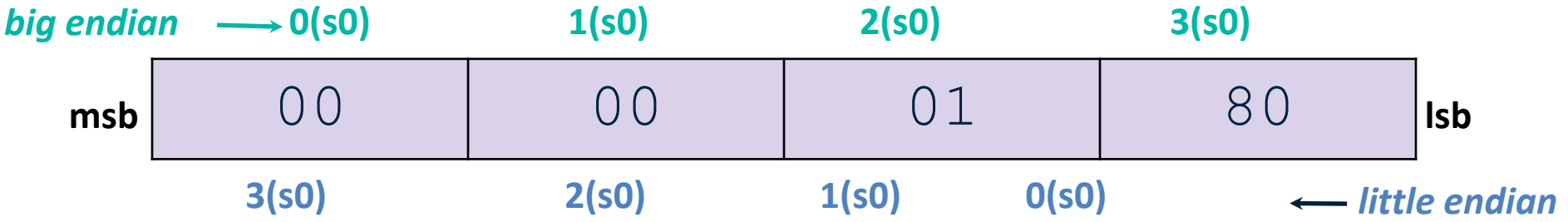
00	00	01	80

Endianness

- Big Endian: Most-significant byte at least address of word
  - word address = address of most significant byte

- Little Endian: Least-significant byte at least address of word
  - word address = address of least significant byte

```
*(s0) = 0x00000180
```



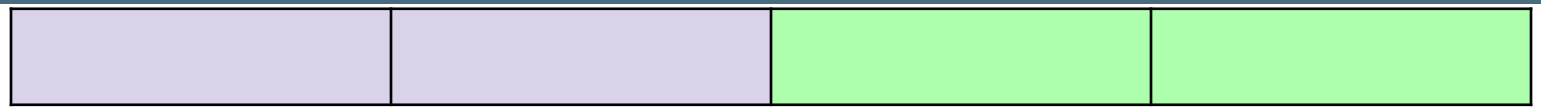
- RISC-V is Little Endian

Byte Instructions

- `lb`/`sb` utilize the least significant byte of the register
  - On `sb`, upper 24 bits are ignored
  - On `lb`, upper 24 bits are filled by sign-extension
- For example, let `*(s0) = 0x00000180`:

```
lb s1,1(s0)   # s1=0x00000001
lb s2,0(s0)   # s2=0xFFFFFF80
sb s2,2(s0)   # *(s0)=0x00800180
```

Half-Word Instructions



- `lh reg, off(bAddr)` "load half"
- `sh reg, off(bAddr)` "store half"
  - On `sh`, upper 16 bits are ignored
  - On `lh`, upper 16 bits are filled by sign-extension

Unsigned Instructions

- `lhu reg, off(bAddr)` "load half unsigned"
- `lbu reg, off(bAddr)` "load byte unsigned"
  - On `l(b/h)u`, upper bits are filled by zero-extension
- Why no s(h/b)u? Why no lwu?

Decision Making Instructions

Computer Decision Making

- In C, we had *control flow*
  - Outcomes of comparative/logical statements determined which blocks of code to execute

- In RISCV, we can't define blocks of code; all we have are labels
  - Defined by text followed by a colon (e.g. `main:`) and refers to the instruction that follows
  - Generate control flow by jumping to labels
  - C has these too, but they are considered bad style

Decision Making Instructions

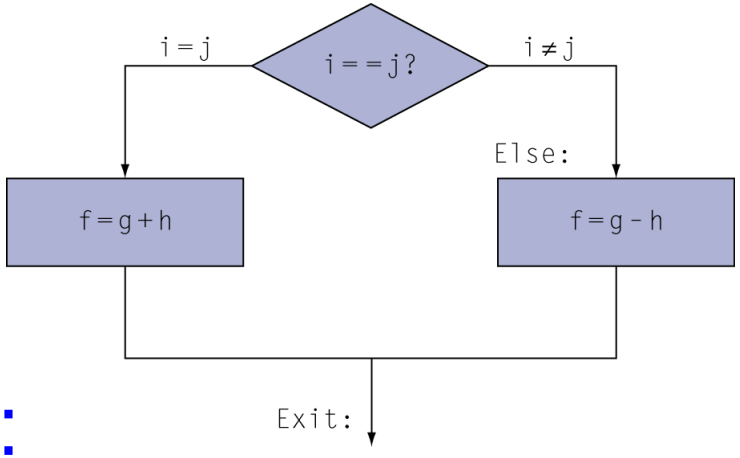
- Branch If Equal (`beq`)
  - `beq reg1,reg2,label`
  - If value in `reg1` = value in `reg2`, go to `label`
- Branch If Not Equal (`bne`)
  - `bne reg1,reg2,label`
  - If value in `reg1` ≠ value in `reg2`, go to `label`
- Jump (`j`)
  - `j label`
  - Unconditional jump to `label`

Breaking Down the If Else

- C code:

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

  - f, g, … in x19, x20, …



- Compiled RISC-V code:

```
        bne x22, x23, Else
        add x19, x20, x21
        beq x0,x0,Exit // unconditional
Else:   sub x19, x20, x21
Exit:   …
```

Assembler calculates addresses

Breaking Down the If Else

C Code:

```
if(i==j) {
   a = b  /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

RISCV (beq):

```
# i→s0, j→s1
# a→s2, b→s3

beq s0,s1,???
???    ——— This label unnecessary
sub s2, x0, s3
j   end
then:
add s2, s3, x0
end:
```

Breaking Down the If Else

C Code:

```
if(i==j) {
   a = b  /* then */
} else {
   a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

RISCV (bne):

```
# i→s0, j→s1
# a→s2, b→s3

bne s0,s1,???
???
add s2, s3, x0
j   end
else:
sub s2, x0, s3
end:
```

Branching on Conditions other than (Not) Equal

- Branch Less Than (blt)
  - `blt reg1,reg2, label`
  - If value in `reg1` < value in `reg2`, go to `label`
- Branch Greater Than or Equal (bge)
  - `bge reg1,reg2, label`
  - If value in `reg1` >= value in `reg2`, go to `label`

- Example
  - if (a > b) a += 1;
  - a in x22, b in x23

```
    bge  x23, x22, Exit       // branch if b >= a
    addi x22, x22, 1
Exit:
```

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu

- Example
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - x22 < x23 // signed
    - –1 < +1
  - x22 > x23 // unsigned
    - +4,294,967,295 > +1

Branching on Conditions other than (Not) Equal

- Set Less Than (slt)
  - `slt dst, reg1,reg2`
  - If value in `reg1` < value in `reg2`, `dst` = 1, else 0

- Set Less Than Immediate (slti)
  - `slti dst, reg1,imm`
  - If value in `reg1` < `imm`, `dst` = 1, else 0

Recommended Reading

- RISC-V Edition - Computer Organization and Design_ The Hardware Software Interface - David A. Patterson, John L. Hennessy:
  - Chapter-2