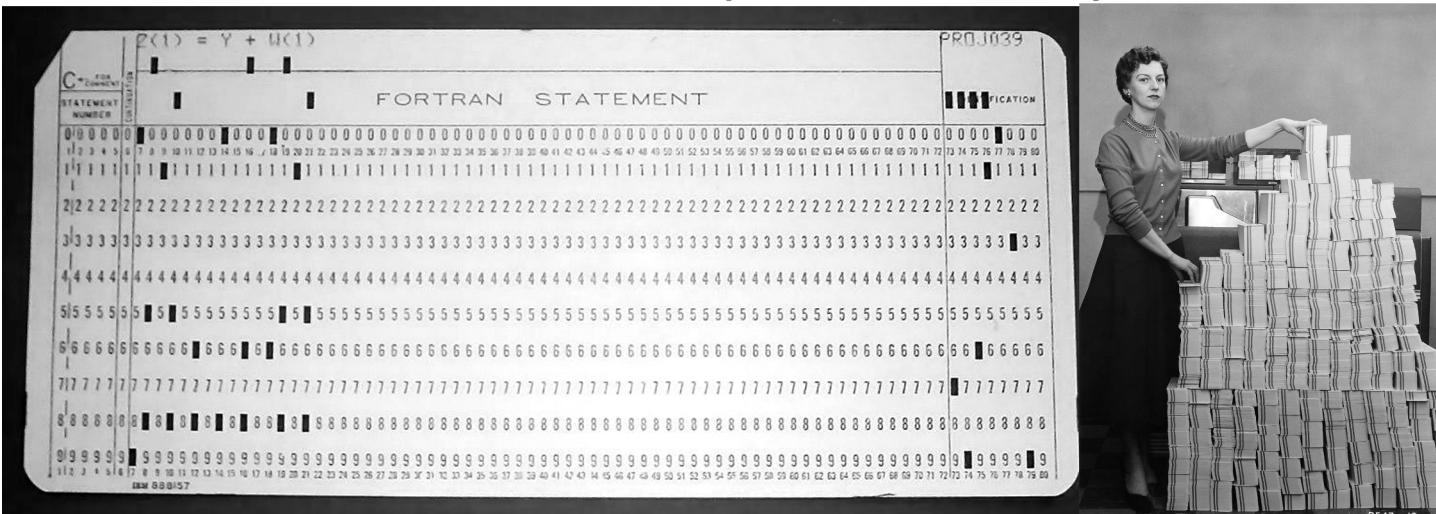


Great Ideas in Computer Architecture

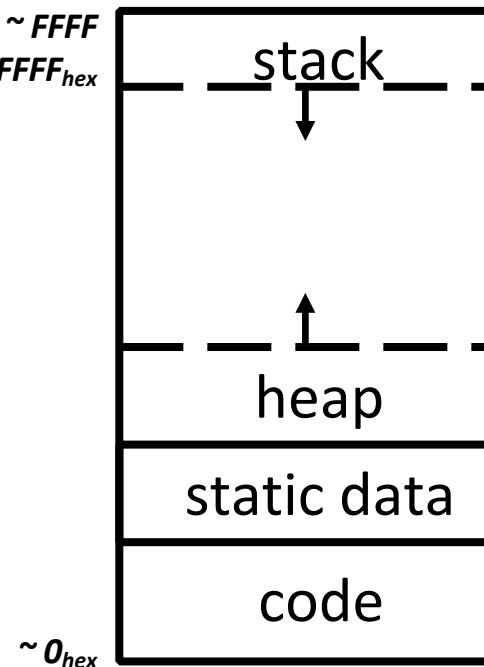
Introduction to Machine Language: RISC-V

Instructor: Stephan Kaminsky

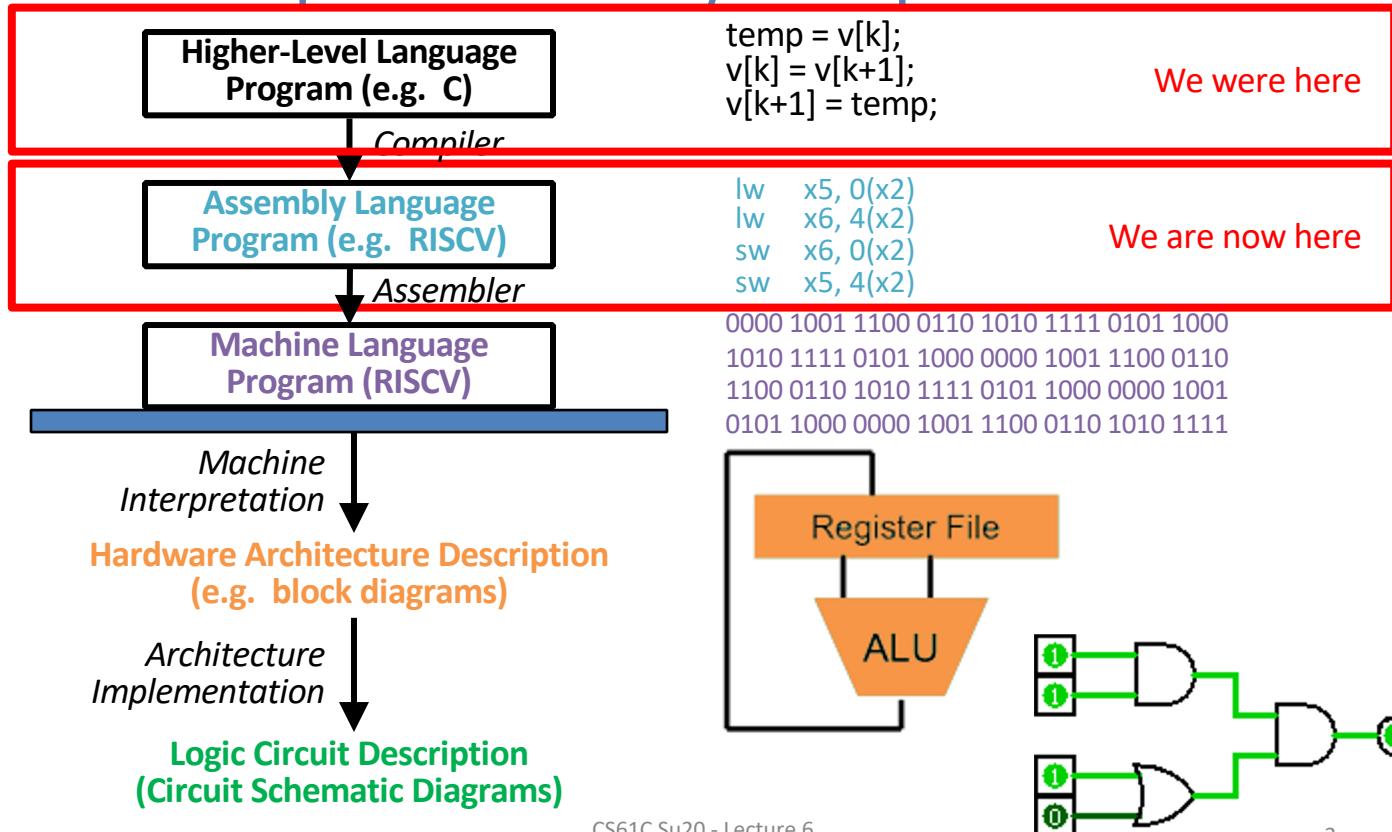


Review of Last Lecture

- C Memory Layout
 - Stack: local variables
 - Static data: global variables
 - Code: machine instructions
 - Heap: dynamic storage using malloc and free
 - must be used CAREFULLY



Great Idea #1: Levels of Representation/Interpretation



Assembly

(Also known as: Assembly Language, ASM)

- A low-level programming language where the program instructions match a particular architecture's operations
- Splits a program into many small instructions that each do one single part of the process

C program

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

Assembly program

`add t1, s3, s4`

`add t2, s1, s2`

`sub s0, t2, t1`

There are many assembly languages

- A low-level programming language where the program instructions match **a particular architecture's** operations
- Each architecture will have a different set of operations that it supports
(although there are many similarities)
- Assembly is not *portable* to other architectures (like C is)

Mainstream Instruction Set Architectures



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, Android), Raspberry
Pi, Embedded systems
[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, embedded
systems, academic use
[RISCV Instruction Set](#)

Which instructions should an assembly include?

There are some obviously useful instructions:

- add, subtract, and, bit shift
- read and write memory

But what about:

- only run the next instruction if these two values are equal
- perform four pairwise multiplications simultaneously
- add two ascii numbers together ('2' + '3' = 5)

Complex/Reduced Instruction Set Computing

- Early trend: add more and more instructions to do elaborate operations:
Complex Instruction Set Computing (CISC)
 - difficult to learn and comprehend language
 - less work for the compiler
 - complicated hardware runs more slowly
- 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 dominates modern computing



How important is this idea?

- RISC architectures dominate computing
 - ARM is RISC and all smartphones use ARM
- Old CISC architectures (x86) are RISC-like underneath these days
- 2017 Turing Award to Patterson and Hennessy

RISC-V is what we'll use in class

- Fifth generation of RISC design from UC Berkeley
 - Professor Krste Asanovic and the Adept Lab
- Open-source Instruction Set specification
- Experiencing rapid uptake in industry and academia
- Appropriate for all levels of computing
 - Embedded microcontrollers to supercomputers
 - 32-bit, 64-bit, and 128-bit variants
 - Designed with academic use in mind



RISC-V Resources

Full RISC-V Architecture

<http://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2016-1.pdf>

Everything we need for CS61C

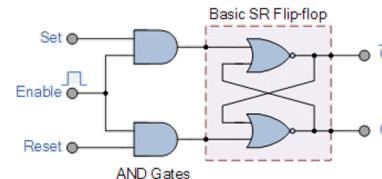
<https://cs61c.org/resources/pdf?file=riscvcard.pdf> (“Green Card”)

RISCV Agenda

- Background
- **Registers**
- Assembly Code
- Basic Arithmetic Instructions
- Immediate Instructions
- Data Transfer Instructions
- Control Flow Instructions
- Shifting Instructions
- Other useful Instructions
- Summary

Hardware uses registers for variables

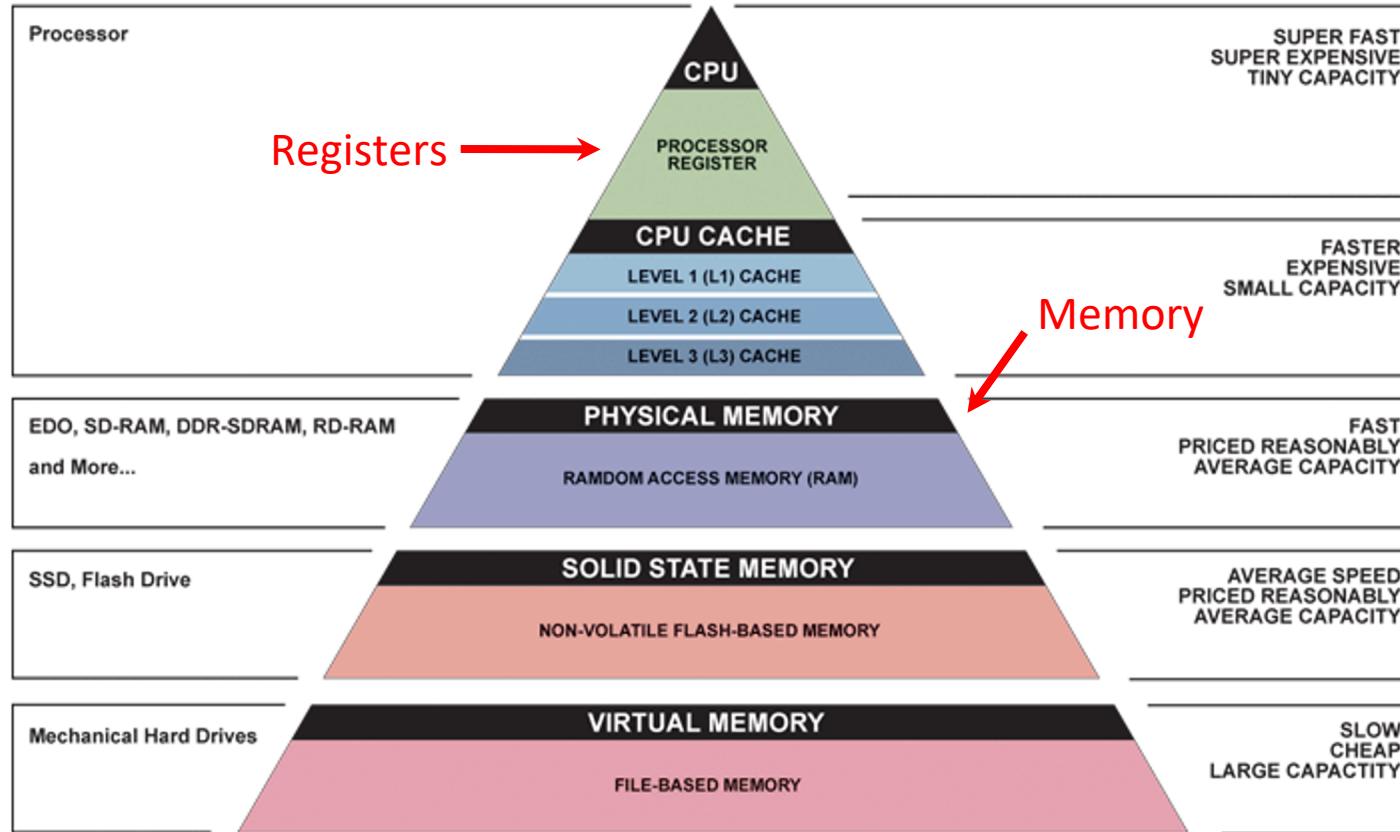
- Unlike C, assembly doesn't have variables as you know them
- Instead, assembly uses registers to store values
- Registers are:
 - Small memories of a fixed size (32-bit in our system)
 - Can be read or written
 - Limited in number (32 registers on our system)
 - Very fast and low power to access



Registers 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 are not all variables in memory?
 - Smaller is faster: registers 100-500 times faster
 - Memory Hierarchy
 - Registers: 32 registers * 32 bits = 128 Bytes
 - RAM: 4-32 GB
 - SSD: 100-1000 GB

Great Idea #3: Principle of Locality/ Memory Hierarchy



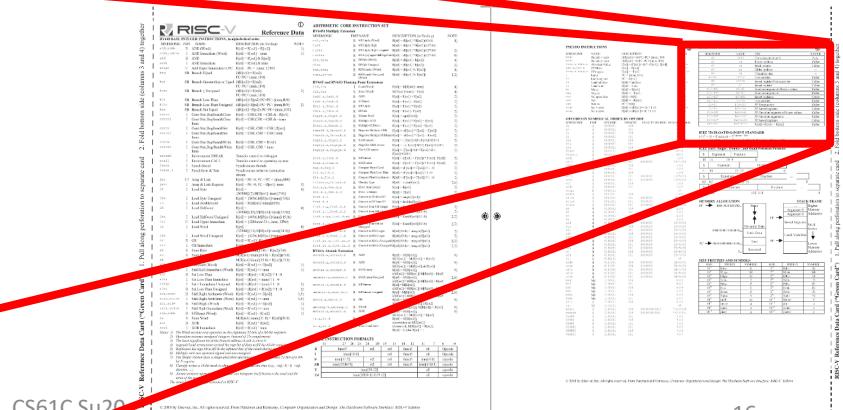
RISCV -- How Many Registers?

- 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**

Note: a word is a fixed-sized piece of data handled as a unit by the instruction set or hardware of the processor. Normally a word is defined as the size of a CPU's registers.

REGISTER NAME, USE, CALLING CONVENTION

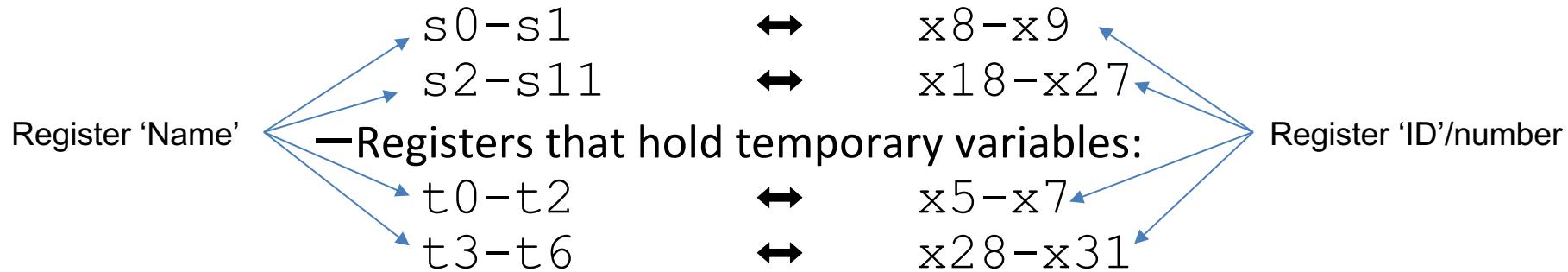
REGISTER	NAME	USE
x0	zero	The constant value 0
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporaries
x8	s0/fp	Saved register/Frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Function arguments/Return values
x12-x17	a2-a7	Function arguments
x18-x27	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 val
f12-f17	fa2-fa7	FP Function arguments
f18-f27	fs2-fs11	FP Saved registers
f28-f31	ft8-ft11	$R[rd] = R[rs1] + R[rs2]$



RISCV Registers

- Register denoted by 'x' can be referenced by number (x_0-x_{31}) or name:

- Registers that hold programmer variables:



- Other registers have special purposes we'll discuss later

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

A Special Register

What's the most important number?

Disclaimer: in programming

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!
 - any instruction writing to `x0` has no effect

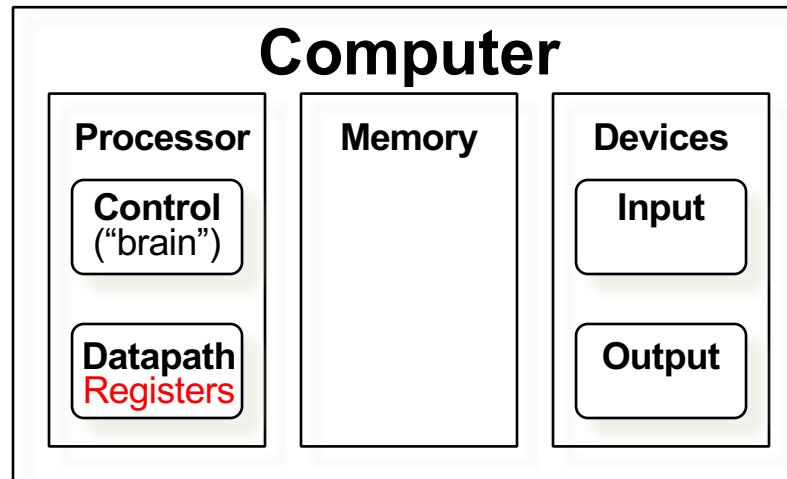
Registers -- Summary

- 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

Registers in a Computer

- We begin our study of how a computer works!

- Control
- Datapath
- Memory
- Input
- Output



- Registers are part of the Datapath

RISCV Agenda

- Background
- Registers
- **Assembly Code**
- Basic Arithmetic Instructions
- Immediate Instructions
- Data Transfer Instructions
- Control Flow Instructions
- Shifting Instructions
- Other useful Instructions
- Summary

RISCV Instructions (1/2)

- Instruction Syntax is rigid:

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”)

- Keep hardware simple via regularity

RISCV Instructions (2/2)

- 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 RISC-V

Example RISC-V Assembly

```
# Fibonacci Sequence
main:    add      t0, x0, x0
          addi     t1, x0, 1
          la       t3, n
          lw       t3, 0(t3)
fib:     beq     t3, x0, finish
          add     t2, t1, t0
          mv      t0, t1
          mv      t1, t2
          addi   t3, t3, -1
          j       fib
finish:  addi   a0, x0, 1
          addi   a1, t0, 0
          ecall # print integer ecall
          addi   a0, x0, 10
          ecall # terminate ecall
```

Example RISC-V Assembly

```
# Fibonacci Sequence
main:    add      t0, x0, x0
          addi     t1, x0, 1
          la       t3, n
          lw       t3, 0(t3)
fib:     beq     t3, x0, finish
          add     t2, t1, t0
          mv      t0, t1
          mv      t1, t2
          addi   t3, t3, -1
          j      fib
finish: addi   a0, x0, 1
          addi   a1, t0, 0
          ecall # print integer ecall
          addi   a0, x0, 10
          ecall # terminate ecall
```

Various assembly
instructions

Example RISC-V Assembly

```
# Fibonacci Sequence
main:    add      t0, x0, x0
          addi     t1, x0, 1
          la       t3, n
          lw       t3, 0(t3)
fib:     beq     t3, x0, finish
          add     t2, t1, t0
          mv      t0, t1
          mv      t1, t2
          addi   t3, t3, -1
          j      fib
finish: addi   a0, x0, 1
          addi   a1, t0, 0
          ecall # print integer ecall
          addi   a0, x0, 10
          ecall # terminate ecall
```

Comments use the
symbol

Example RISC-V Assembly

```
# Fibonacci Sequence
main:    add    t0, x0, x0
          addi   t1, x0, 1
          la     t3, n
          lw     t3, 0(t3)
fib:     beq   t3, x0, finish
          add   t2, t1, t0
          mv    t0, t1
          mv    t1, t2
          addi  t3, t3, -1
          j     fib
finish: addi  a0, x0, 1
         addi  a1, t0, 0
         ecall # print integer ecall
         addi  a0, x0, 10
         ecall # terminate ecall
```

Labels are arbitrary names that mark a section of code

We'll get back to these later

Example RISC-V Assembly

```
# Fibonacci Sequence  
main:    add      t0, x0, x0
```

The diagram illustrates the mapping of RISC-V assembly instruction fields to memory locations. It shows four blue arrows pointing from labels below the assembly code to specific fields in the instruction:

- A vertical arrow points from the label "op" to the first field of the instruction, which contains "add".
- A vertical arrow points from the label "dst" to the second field, which contains "t0".
- A diagonal arrow points from the label "src1" to the third field, which contains "x0".
- A diagonal arrow points from the label "src2" to the fourth field, which contains "x0".

What does this instruction do?

There are many instructions

We'll be covering the **types** of instructions in class.

You should look through what specific commands exist on the green card.

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)
add, addw	R	ADD (Word)	$R[rd] = R[rs1] + R[rs2]$
addi, addiw	I	ADD Immediate (Word)	$R[rd] = R[rs1] + imm$
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	$\begin{aligned} &\text{if}(R[rs1] == R[rs2]) \\ &\quad PC = PC + \{imm, 1b'0\} \end{aligned}$
bge	SB	Branch Greater than Equal	$\begin{aligned} &\text{if}(R[rs1] >= R[rs2]) \\ &\quad PC = PC + \{imm, 1b'0\} \end{aligned}$
bgeu	SB	Branch \geq Unsigned	$\begin{aligned} &\text{if}(R[rs1] >= R[rs2]) \\ &\quad PC = PC + \{imm, 1b'0\} \end{aligned}$
blt	SB	Branch Less Than	$\begin{aligned} &\text{if}(R[rs1] < R[rs2]) \\ &\quad PC = PC + \{imm, 1b'0\} \end{aligned}$
bitu	SB	Branch Less Than Unsigned	$\begin{aligned} &\text{if}(R[rs1] < R[rs2]) \\ &\quad PC = PC + \{imm, 1b'0\} \end{aligned}$
bne	SB	Branch Not Equal	$\begin{aligned} &\text{if}(R[rs1] != R[rs2]) \\ &\quad PC = PC + \{imm, 1b'0\} \end{aligned}$
ebreak	I	Environment BREAK	Transfer control to debugger
ecall	I	Environment CALL	Transfer control to operating system
jal	UJ	Jump & Link	$R[rd] = PC+4; PC = PC + \{imm, 1b'0\}$
jalr	I	Jump & Link Register	$R[rd] = PC+4; PC = R[rs1]+imm$
lb	I	Load Byte	$R[rd] = \{56'bM[](7), M[R[rs1]]+imm\}(7:0)$
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	$R[rd] = \{48'bM[](15), M[R[rs1]]+imm\}(15:0)$
lhu	I	Load Halfword Unsigned	$R[rd] = \{48'b0, M[R[rs1]]+imm\}(15:0)$
lui	U	Load Upper Immediate	$R[rd] = \{32'bimm<31>, imm, 12'b0\}$
lw	I	Load Word	$R[rd] = \{32'bM[](31), M[R[rs1]]+imm\}(31:0)$
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, slliw	R	Shift Left (Word)	$R[rd] = R[rs1] << R[rs2]$
slli, slliw	I	Shift Left Immediate (Word)	$R[rd] = R[rs1] << imm$
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	$R[rd] = (R[rs1] < imm) ? 1 : 0$
sltu	R	Set Less Than Unsigned	$R[rd] = (R[rs1] < R[rs2]) ? 1 : 0$
sra, sraw	R	Shift Right Arithmetic (Word)	$R[rd] = R[rs1] >> R[rs2]$
srai, srailw	I	Shift Right Arith Imm (Word)	$R[rd] = R[rs1] >> imm$
srl, srliw	R	Shift Right (Word)	$R[rd] = R[rs1] >> R[rs2]$
srl, srliw	I	Shift Right Immediate (Word)	$R[rd] = R[rs1] >> imm$
sub, subw	R	SUBtract (Word)	$R[rd] = R[rs1] - R[rs2]$
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	$R[rd] = R[rs1] ^ imm$

RISCV Agenda

- Background
- Registers
- Assembly Code
- **Basic Arithmetic Instructions**
- Immediate Instructions
- Data Transfer Instructions
- Control Flow Instructions
- Shifting Instructions
- Other useful Instructions
- Summary

RISCV Instructions Example

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 $a \rightarrow s0, b \rightarrow s1, c \rightarrow s2, d \rightarrow s3$ and $e \rightarrow s4$. 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

Greencard Explanation

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)
add, addw	R	ADD (Word)	$R[rd] = R[rs1] + R[rs2]$

Op in assembly
(operations with w at the end work with 32bits (a word in RISC-V) and are meant for systems larger than 32 bits)



Verilog (hardware description)
R: array of registers

Read as:
Read register at rs1 and register at rs2, Add them, and write to register at rd



RISCV Agenda

- Background
- Registers
- Assembly Code
- Basic Arithmetic Instructions
- **Immediate Instructions**
- Data Transfer Instructions
- Control Flow Instructions
- Shifting Instructions
- Other useful Instructions
- Summary

Immediates

- Numerical constants are called **immediates**
- Separate instruction syntax for immediates:

opi dst, src, imm

- Operation names end with ‘i’, replace 2nd source register with an immediate
 - Immediates can be up to 12-bits in size
 - Interpreted as sign-extended two’s complement
- Example Uses:

- addi s1, s2, 5 # a=b+5
- addi s3, s3, 1 # c++

RISC-V Immediate Example

- Suppose $a \rightarrow s0, b \rightarrow s1$.

Convert the following C statement to RISCV:

$a = (5+b) - 3;$

addi t1, s1, 5

addi s0, t1, -3

- Why no subi instruction?

RISCV Agenda

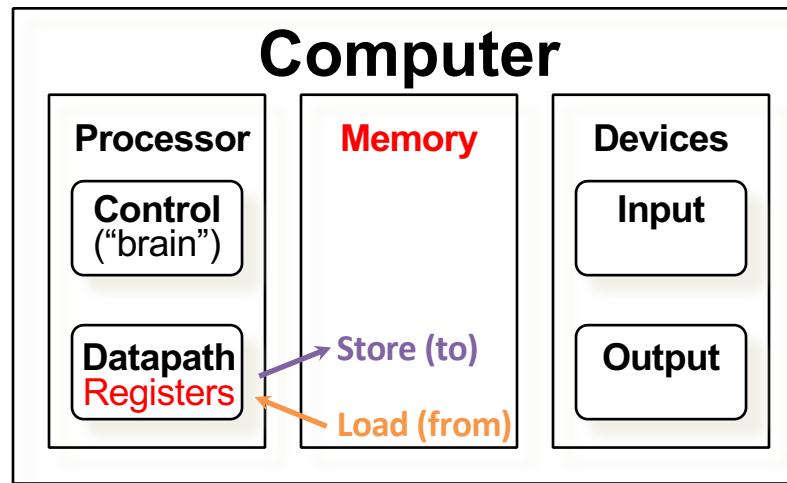
- Background
- Registers
- Assembly Code
- Basic Arithmetic Instructions
- Immediate Instructions
- **Data Transfer Instructions**
- Control Flow Instructions
- Shifting Instructions
- Other useful Instructions
- Summary

Data Transfer

- C variables map onto registers;
What about large data structures like arrays?
 - Assembly can also access *memory*
- But RISCV instructions only operate on registers!
- Specialized **data transfer instructions** move data between registers and memory
 - Store: register TO memory
 - Load: register FROM memory

Five Components of a Computer

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



Data Transfer

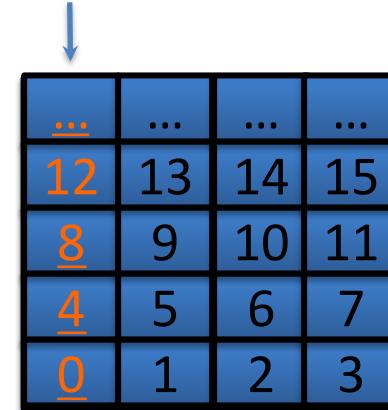
- Instruction syntax for data transfer:

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 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)
- Memory addresses are indexed by *bytes*, not words
- Word addresses are 4 bytes apart
 - Word addr is same as first 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



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: (address of int array[] -> s3, value of b-> s2)

C:

```
array[10] = array[3]+b;
```

Assembly:

```
lw      t0,12($3) # t0=A[3]
add    t0,$2,t0   # t0=A[3]+b
sw      t0,40($3) # A[10]=A[3]+b
```

Values can start off in memory

```
.data
```

```
source:
```

```
    .word 3
```

```
    .word 1
```

```
    .word 4
```

```
.text
```

```
main:
```

```
    la t1, source
```

```
    lw t2, 0(t1)
```

```
    lw t3, 4(t1)
```

- .data denotes data storage
 - .word, .byte, etc

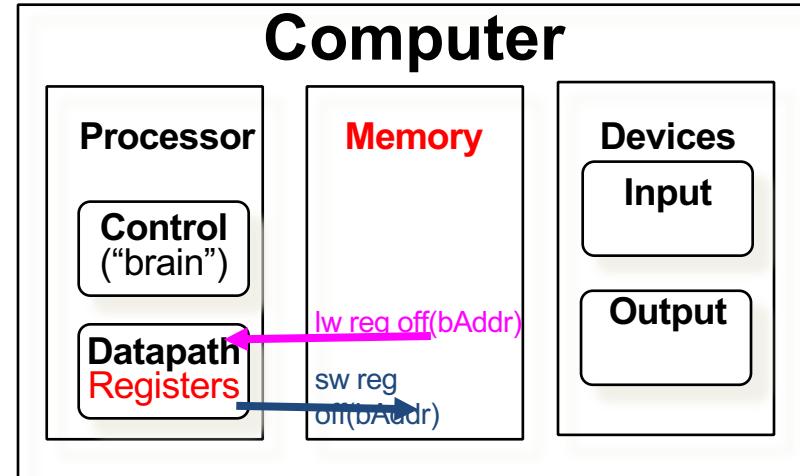
- labels for pointing to data

- .text denotes code storage

List of Venus assembler directives
<https://github.com/ThaumicMekanism/venus/wiki/Assembler-Directives>

Memory and Variable Size

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



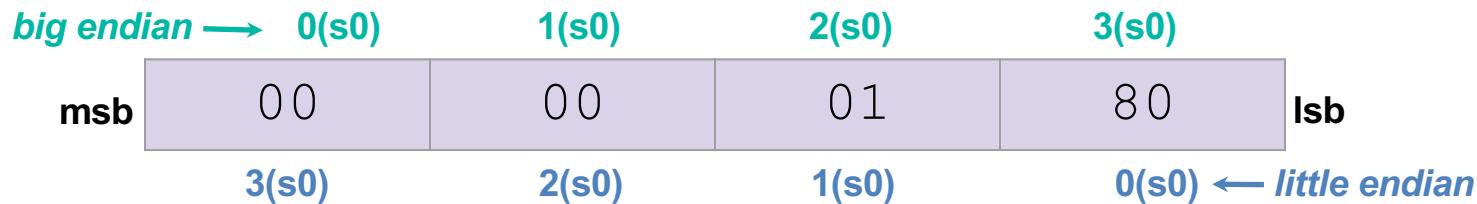
- What about characters (1 Byte) and shorts (sometimes 2 Bytes), etc?

Want to be able to use interact with memory values smaller than a word.

Endianness

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

$s0 = 0x000000180$



- RISC-V is **LittleEndian**

Sign Extension

- We want to represent the same number using more bits than before
 - **Sign Extend:** Take the most-significant bit and copy it to the new bits
 - $0b\ 11 = 0b\ 1111$
 - **Zero/One Pad:** Set the new bits with one/zero.
 - Zero Pad: $0b\ 11 = 0b\ 0011$
 - One Pad: $0b\ 11 = 0b\ 1111$
- All base RISC-V instructions sign extend when needed
 - auipc & lui technically would but they are filling the upper bits so there is nothing to fill.

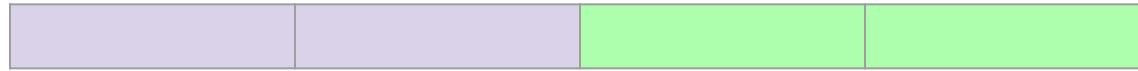
Byte Instructions



- `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=0xFFFFFFF80  
sb s2,2(s0) # *(s0)=0x00800180
```

Half-Word Instructions



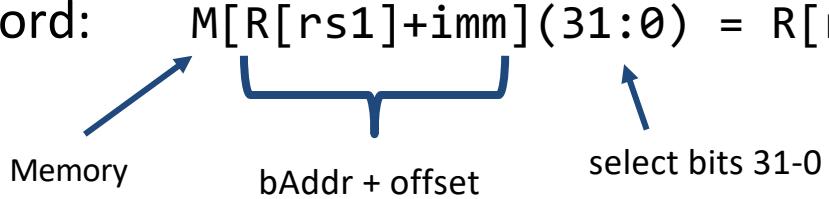
- lh reg, off(bAddr) “load half”
- sh reg, off(bAddr) “store half”
 - On sh, upper 16 bits are ignored
 - On lh, upper 16 bits are filled by sign-extension

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?

Data Transfer Greencard Explanation

Store word: $M[R[rs1]+imm](31:0) = R[rs2](31:0)$



Store byte: $M[R[rs1]+imm](7:0) = R[rs2](7:0)$

Data Transfer Greencard Explanation

Load byte: $R[rd] = \{24'bM[](7), M[R[Rs1]+imm](7:0)\}$

The expression $R[rd] = \{24'bM[](7), M[R[Rs1]+imm](7:0)\}$ is shown with curly braces. Two blue brackets are placed under the expression to indicate its components: one bracket groups the first part $24'bM[](7)$, and another bracket groups the second part $M[R[Rs1]+imm](7:0)$. Below the first bracket, the text "24 bits sign extended based on most-significant bit" is centered. Below the second bracket, the text "Byte from memory" is centered.

24 bits sign extended
based on most-
significant bit

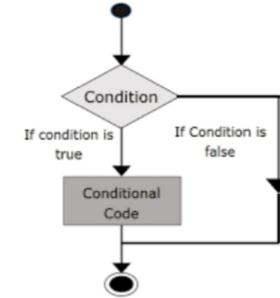
Byte from memory

RISCV Agenda

- Background
- Registers
- Assembly Code
- Basic Arithmetic Instructions
- Immediate Instructions
- Data Transfer Instructions
- **Control Flow Instructions**
- Shifting Instructions
- Other useful Instructions
- Summary

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 labels too, but they are considered bad style.
Don't use goto!



Decision Making Instructions

- Branch If Equal (beq)
 - beq reg1, reg2, label
 - If value in reg1 = value in reg2, go to label
 - Otherwise go to the next instruction
- 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) {  
    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  
???:  
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,else  
then:  
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

RISC Philosophy:

- Why create a “branch greater than” if you could swap the arguments and use “branch less than”?

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 (???):

```
# i→s0, j→s1  
# a→s2, b→s3  
??? s0,s1,else  
then:  
add s2, s3, x0  
j end  
else:  
sub s2, x0, s3  
end:
```

Loops in RISCV

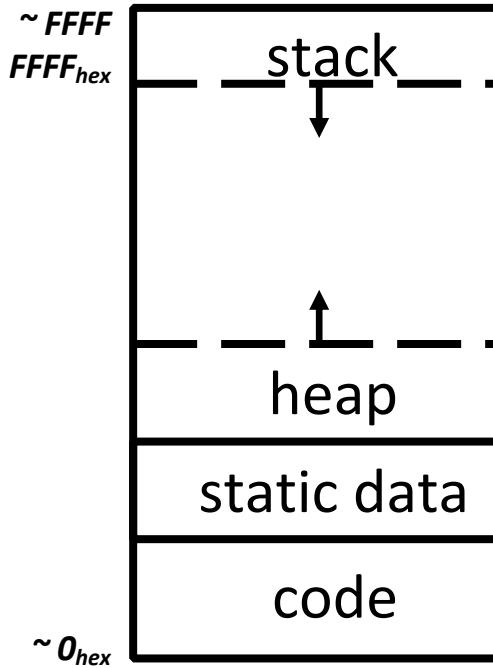
- There are three types of loops in C:
 - while, do...while, and for
 - Each can be rewritten as either of the other two, so the same concepts of decision-making apply
- These too can be created with branch instructions
- **Key Concept:** Though there are multiple ways to write a loop in RISCV, the key to decision-making is the conditional branch

Program Counter

- Branches and Jumps change the flow of execution by modifying the Program Counter (PC)
- The PC is a special register that contains the current address of the code that is being executed
 - Not accessible as part of the normal 32 registers

Instruction Addresses

- Instructions are stored as data in memory and have addresses!
 - Recall: Code section
 - Labels get converted to instruction addresses
 - More on this later this week
- The PC tracks where in memory the current instruction is



Control Flow Greencard Explanation

Branch Equal (beq): `if (R[rs1] == R[rs2])
 PC = PC+{imm,1b'0}`



if registers are equal:
jump to the immediate with one
bit of 0 prepended to it

(makes the PC an even number)

Immediate is calculated as the
offset to the label

RISCV Agenda

- Background
- Registers
- Assembly Code
- Basic Arithmetic Instructions
- Immediate Instructions
- Data Transfer Instructions
- Control Flow Instructions
- **Shifting Instructions**
- Other useful Instructions
- Summary

Shifting Instructions

- In binary, shifting an unsigned number left is the same as multiplying by the corresponding power of 2
 - Shifting operations are faster
 - Does not work with shifting right/division
- *Logical shift*: Add zeros as you shift
- *Arithmetic shift*: Sign-extend as you shift
 - Only applies when you shift right (preserves sign)
- Shift by immediate or value in a register

Shifting Instructions

Instruction Name	RISCV
Shift Left Logical	sll s1,s2,s3
Shift Left Logical Imm	slli s1,s2,imm
Shift Right Logical	srl s1,s2,s3
Shift Right Logical Imm	srlti s1,s2,imm
Shift Right Arithmetic	sra s1,s2,s3
Shift Right Arithmetic Imm	srai s1,s2,imm

- When using immediate, only values 0-31 are practical
- When using variable, only lowest 5 bits are used (read as unsigned)

Shifting Instructions

```
# sample calls to shift instructions
addi t0,x0,-256 # t0=0xFFFFFFF00
slli s0,t0,3      # s0=0xFFFFF800
srli s1,t0,8      # s1=0x00FFFFFF
srai s2,t0,8      # s2=0xFFFFFFFF

addi t1,x0,-22    # t1=0xFFFFFFFFEA
                   # low 5: 0b01010
sll s3,t0,t1      # s3=0xFFFFC0000
# same as slli s3,t0,10
```

Shifting Instructions

- Example 1:

```
# lbu using lw:    lbu s1,1($0)
lw    s1,0($0)          # get word
li    t0,0x0000FF00    # load bitmask
and   s1,s1,t0          # get 2nd byte
srlt  s1,s1,8           # shift into lowest
```

Shifting Instructions

- Example 2:

```
# sb using sw:    sb s1,3($0)
lw    t0,0($0)          # get current word
li    t1, 0xFFFFFFFF # load bitmask
and   t0,t0,t1          # zero top byte
slli  t1,s1,24          # shift into highest
or    t0,t0,t1          # combine
sw    t0,0($0)          # store back
```

RISCV Agenda

- Background
- Registers
- Assembly Code
- Basic Arithmetic Instructions
- Immediate Instructions
- Data Transfer Instructions
- Control Flow Instructions
- Shifting Instructions
- **Other useful Instructions**
- Summary

RISCV Arithmetic Instructions Multiply Extension

- **Multiplication** (mul and mulh)
 - mul dst, src1, src2
 - mulh dst, src1, src2
 - src1*src2: lower 32-bits through mul, upper 32-bits in mulh
- **Division** (div)
 - div dst, src1, src2
 - rem dst, src1, src2
 - src1/src2: quotient via div, remainder via rem

RISCV Bitwise Instructions

Note: $a \rightarrow s1$, $b \rightarrow s2$, $c \rightarrow s3$

Instruction	C	RISCV
And	$a = b \& c;$	and s1,s2,s3
And Immediate	$a = b \& 0x1;$	andi s1,s2,0x1
Or	$a = b c;$	or s1,s2,s3
Or Immediate	$a = b 0x5;$	ori s1,s2,0x5
Exclusive Or	$a = b ^ c;$	xor s1,s2,s3
Exclusive Or Immediate	$a = b ^ 0xF;$	xori s1,s2,0xF

Compare Instructions

- 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`

Another If Else Design

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:

```
# i→s0, j→s1  
# a→s2, b→s3  
slt t0 s0 s1  
beq t0, x0, else  
then:  
add s2, s3, x0  
j end  
else:  
sub s2, x0, s3  
end:
```

Environment Call

- `ecall` is a way for an application to interact with the operating system
- The value in register `a0` is given to the OS which performs special functions
 - Printing values
 - Exiting the program
 - Allocating more memory for the program

A list of the `ecall` values supported by venus:

<https://github.com/ThaumicMekanism/venus/wiki/Environmental-Calls>

RISCV Agenda

- Background
- Registers
- Assembly Code
- Basic Arithmetic Instructions
- Immediate Instructions
- Data Transfer Instructions
- Control Flow Instructions
- Shifting Instructions
- Other useful Instructions
- **Summary**

Summary

- Computers understand the *instructions* of their *Instruction Set Architecture (ISA)*
- RISC Design Principles
 - Smaller is faster, keep it simple
- RISC-V Registers: s0-s11, t0-t6, x0
- RISC-V Instructions
 - Arithmetic: add, sub, addi
 - Data Transfer: lw, sw
 - Memory is byte-addressed
 - Control Flow: beq, bne, j