



Programming with the RISC-V Instruction Set

Dr. Brian Mitchell



Programming with the RISC-V Instruction Set

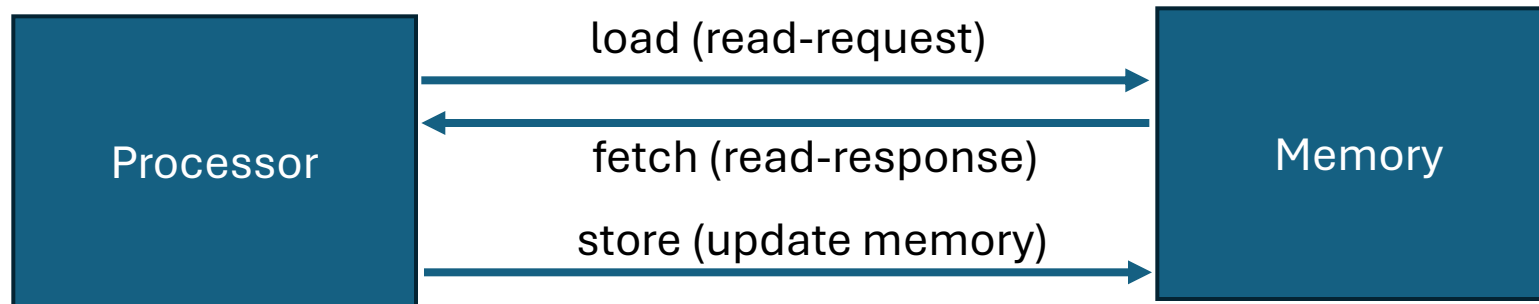
Part 1: Intro to Programming Directly on the RV Processor

Dr. Brian Mitchell

Introduction



- The stored-program concept:
 - Instructions are represented as 32-bit words (each word is 4 bytes).
 - Programs can be stored in memory to be read just like data.
- Fetch & Execute Cycle for instructions
 - Instruction is fetched and put into a special register
 - Bits in the instruction register determine the subsequent actions
 - When done, fetch the next instruction and continue execution



The RISC-V architecture follows a **load/store architectural model** – in other words the processor does not directly operate against memory, data from memory must first be fetched into the processor, executed, and any results must be saved back into memory.

RISC-V Naming Conventions



All RISC-V Processors are labeled RV Followed by 32, 64, 128 and then Labels To Define Capabilities

Character		Name	Description
What we will be using	I	RV32I	Base integer instruction set
	E	RV32E	Base integer for embedded (16-bit registers)
	M	RV32IM	Multiply/Divide extension
	A	RV32IMA	Atomic instructions
	F	RV32IMAF	Single-precision floating-point
	D	RV32IMAFD	Double-precision floating-point
	G	RV32G	Shorthand for the IMAFD
	Q	RV32Q	Quad-precision floating-point
	C	RV32C	Compressed instructions
	K	RV32K	Scalar cryptography
	H	RV32H	Hypervisor extension
	V	RV32V	Vector operations ¹
	B	RV32B	Bit manipulation operations ¹
	P	RV32P	DSP and packed SIMD instructions ¹

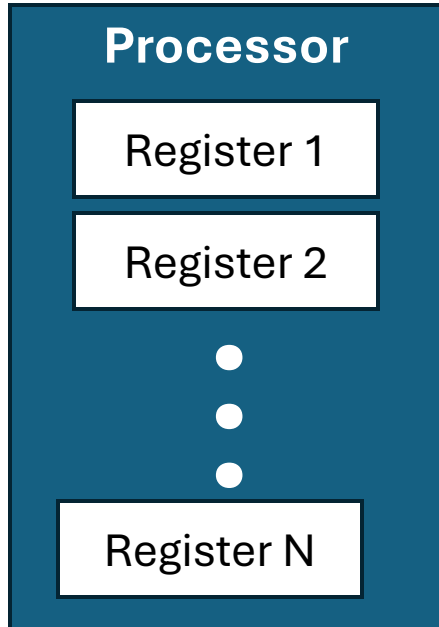
Memory Organization



- Viewed as a large, single-dimensional array, where a memory address is an index into the array
- RISC-V, like many other systems can address memory in byte chunks
- The memory address (= index) points to a byte in memory
- This is often referred to as “Byte Addressing”

Memory Address	Memory Contents
0x00000000	0x00
0x00000001	0x00
0x00000002	0x00
0x00000003	0x00
0x00000004	0x00
0x00000005	0x00
0x00000006	0x00
0x00000007	0x00

Memory Organization & Registers



- Other key “memory” is storage that the processor uses to facilitate the execution of instructions-**called registers.**
- Different processors have a different number of registers, and use different register sizes
- A few slides back we covered the RISC-V naming conventions **RV###[Capabilities]**
- Where **###=[32|64|128]** – this is the number of bits that the processor can operate on at any one time and defines the size of the register
- For this class we will be using the RV32IM processor (base integer + multiply/divide extensions)



The RV32I Processor Organization



Memory Address	Memory Contents
0x00000000	0x00000000
0x00000004	0x00000000
0x00000008	0x00000000
0x0000000C	0x00000000
0x00000010	0x00000000
0x00000014	0x00000000
0x00000018	0x00000000
0x0000001C	0x00000000
0x00000020	0x00000000

- The RV32I is a 32 bit processor
- There are 32 registers, named x0 ... x31
- Each register is 32 bits
- Memory is organized into “words” = 4 bytes = 32 bits
- The processor accesses memory at word boundaries only (alignment)
- You will be mastering counting in 4’s 😊 during this course.

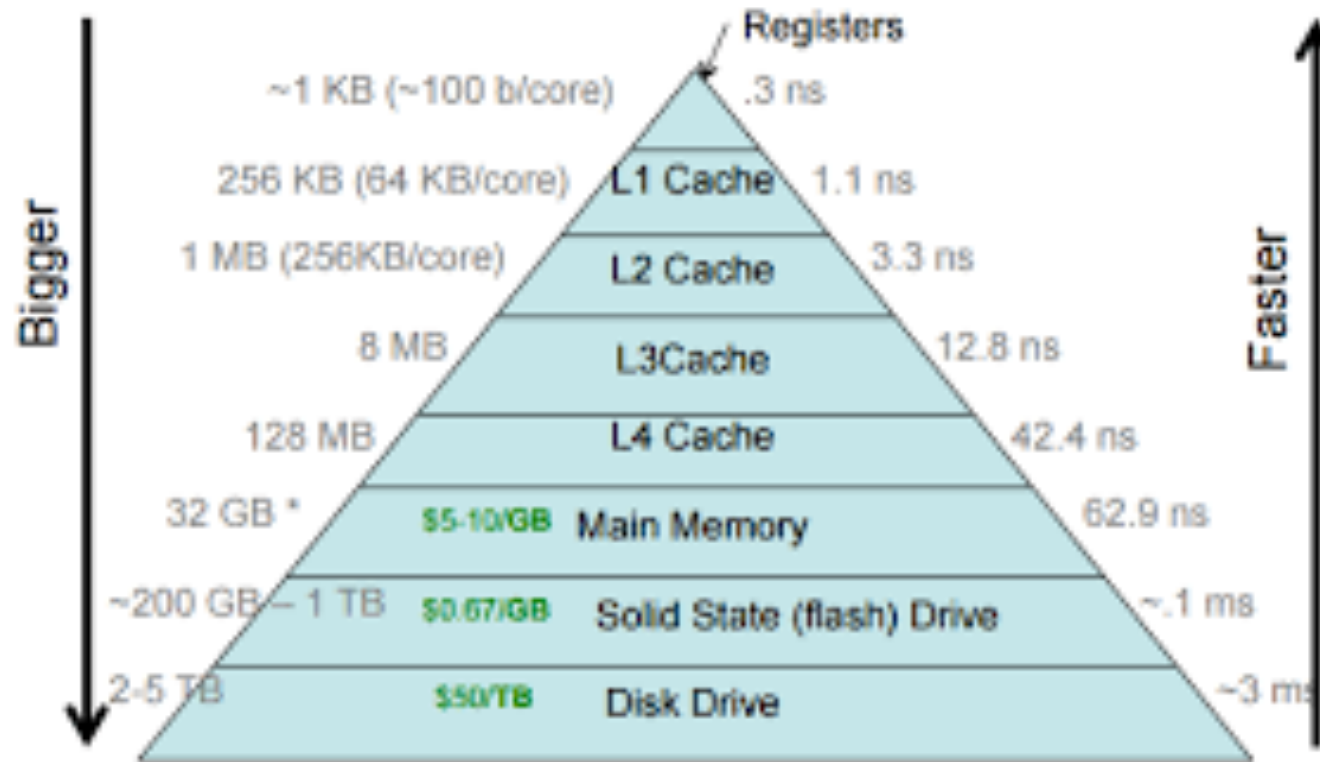
The RV32I Processor: Memory Organization



Memory Address	Memory Contents
0x00000000 to 0x0000FFFF	.text segment Program Instructions
0x00010000 to 0x0001FFFF	.data (constant) segment Read only constant values and variables
0x00020000 to 0x0002FFFF	.globl segment Read/write global variables
0x00030000 to 0xFFFFFFFFFC	DYNAMIC MEMORY HEAP (aka, malloc) ↓ ↑ STACK (aka, functions)

- Given a 32 bit address space the maximum size of memory is:
 - $0 \dots 2^{32}-1$ bytes
 - $0 \dots 2^{30}-4$ words
- Memory is divided into “regions” to manage the execution of programs
- In the dynamic region, the heap grows in an increasing direction, and the stack in a decreasing direction
 - Must be careful to manage that the stack and heap do not over-run each other

Memory Types & Memory Hierarchy



- RV is a load/store architecture
- All operations are performed in a constrained number of registers
- The entire storage hierarchy is used to execute modern programs
- Execution efficiency of your program depends on how and which storage is used

Most programmers do not think about this as efficiency at the hardware/software boundary is largely handled by optimizations in compiles, the operating system and the processor.



Big vs Little Endian

- Computer architectures must decide how to interpret byte-order for multi-byte values
- Specified in the ISA, there are 2 types – **big endian** and **little endian**.
RISV is a **little endian** architecture

Consider the following 32-bit value: 0x12345678, we can treat this value as four individual bytes, two 16-bit values, or one 32-bit value

VALUE IN MEMORY	BIG ENDIAN	LITTLE ENDIAN								
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr></table>	1	2	3	4	5	6	7	8	0x12, 0x34, 0x56, 0x78	0x12, 0x34, 0x56, 0x78
1	2	3	4	5	6	7	8			
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr></table>	1	2	3	4	5	6	7	8	0x1234, 0x5678	0x3412, 0x7856
1	2	3	4	5	6	7	8			
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr></table>	1	2	3	4	5	6	7	8	0x12345678	0x78563412
1	2	3	4	5	6	7	8			



Big vs Little Endian

Revisiting the following 32-bit value: 0x12345678

VALUE IN MEMORY	BIG ENDIAN	LITTLE ENDIAN								
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr></table>	1	2	3	4	5	6	7	8	0x12, 0x34, 0x56, 0x78	0x12, 0x34, 0x56, 0x78
1	2	3	4	5	6	7	8			
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr></table>	1	2	3	4	5	6	7	8	0x1234, 0x5678	0x3412, 0x7856
1	2	3	4	5	6	7	8			
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr></table>	1	2	3	4	5	6	7	8	0x12345678	0x78563412
1	2	3	4	5	6	7	8			

Think about it like reading a book from left to right, the bytes in memory are sequenced in the following order [0x12, 0x23, 0x56, 0x78] thus the question is:

Are you reading the bytes in from most to least significant bytes (big-endian) or from least to most significant bytes (little-endian)?

NOTE: Older ISA architectures tended to favor big-endian, most modern processors use little-endian, or can be configured to use either byte encoding. We can largely ignore the differences until we get into network programming.

The RV32I Processor: Registers



Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–7	t0–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

RISC-V registers are named x0-x31, but also have alternative ABI (application binary interface) names that make software development easier

The main register categories are the “arguments” in a0-a7, the “save” registers in s0-s11, the “temporary registers” in t0-t6, and some “special-purpose registers” in ra, sp, zero. We will discuss these registers later.

RISC-V Instructions



RISC-V instructions come in several different formats, which we will discuss shortly. The first type of instruction we will look at takes the form:

opcode rd, rs1, rs2 # where rd, rs1, and rs2 are registers

Example in C we code:

```
c = a + b;
```

The compiler decides to assign c to x5, a to x6 and b to x7:

```
add x5, x6, x7          # x5 <- x6 + x7
```

Which is equivalent to:

```
add t0, t1, t2         # t0 <- t1 + t2
```



RISC-V Instructions

The RISC-V card (posted on Blackboard) will be our go to reference for this course...

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	$rd = rs1 + rs2$	msb-extends
sub	SUB	R	0110011	0x0	0x20	$rd = rs1 - rs2$	
xor	XOR	R	0110011	0x4	0x00	$rd = rs1 \wedge rs2$	
or	OR	R	0110011	0x6	0x00	$rd = rs1 rs2$	
and	AND	R	0110011	0x7	0x00	$rd = rs1 \& rs2$	
sll	Shift Left Logical	R	0110011	0x1	0x00	$rd = rs1 \ll rs2$	
srl	Shift Right Logical	R	0110011	0x5	0x00	$rd = rs1 \gg rs2$	
sra	Shift Right Arith*	R	0110011	0x5	0x20	$rd = rs1 \gg rs2$	zero-extends
slt	Set Less Than	R	0110011	0x2	0x00	$rd = (rs1 < rs2)?1:0$	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	$rd = (rs1 < rs2)?1:0$	
addi	ADD Immediate	I	0010011	0x0	imm[5:11]=0x00 imm[5:11]=0x00 imm[5:11]=0x20	$rd = rs1 + imm$	msb-extends
xori	XOR Immediate	I	0010011	0x4		$rd = rs1 \wedge imm$	
ori	OR Immediate	I	0010011	0x6		$rd = rs1 imm$	
andi	AND Immediate	I	0010011	0x7		$rd = rs1 \& imm$	
slli	Shift Left Logical Imm	I	0010011	0x1		$rd = rs1 \ll imm[0:4]$	
srli	Shift Right Logical Imm	I	0010011	0x5		$rd = rs1 \gg imm[0:4]$	
srai	Shift Right Arith Imm	I	0010011	0x5		$rd = rs1 \gg imm[0:4]$	
slti	Set Less Than Imm	I	0010011	0x2		$rd = (rs1 < imm)?1:0$	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		$rd = (rs1 < imm)?1:0$	
lb	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$	zero-extends
lh	Load Half	I	0000011	0x1		$rd = M[rs1+imm][0:15]$	
lw	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$	
lbu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$	
lhu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$	

Register Usage



- Operations for arithmetic operations **must be placed** in registers
- There are a constrained number of registers in any processor, RISCV = 32 (5 specialized [**x0-x4**], 27 are general purpose [**x5-x31**])
- A compiler associates variables with registers:
 - There are no variables when we program in assembler – the compiler translates variable names and values to register names and values – **they are abstractions in programming languages**
 - The compiler tries to keep the most frequently used variables in registers
 - When no registers are available and one is needed, the compiler selects a register and saves its value out to memory (more on this later)

Note that one key attribute associated with the efficiency of a compiler is on how effective it is in allocating variables to registers. Frequent need to store values and then reload them can slow down program execution. Recall the memory hierarchy.



Not all registers can be used by our programs for managing data – Example the zero register

Register **x0** in RISC-V has the ABI name of **zero**

The **zero** register is **read-only** and is **hard wired** with the constant 0.

Why do this, and is it useful?

Consider one common operation that involves copying a value from one register to another:

add t0, t1, zero #t0 = 0+t1 => Copy value from t1 to t0

This may be non-intuitive at first, but it is very fast and efficient!

Note this is only one example of the usefulness of special purpose registers like the **zero register. We will see other examples soon with the **sp** and **ra** registers.**

Simulating a C Compiler



As we have seen RISC-V operations like `add` take 2 registers and produce a result, storing it in a third. Consider this example in C:

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

As of this point, let's assume that `f`, `g`, `h`, `i`, and `j` are in registers `s3-s7` respectively, the compiler would need to break this into multiple instructions:

```
add t0, s3, s4      # t0 = g + h
add t1, s5, s6      # t1 = i + j
sub s3, t0, t1      # f = t0 - t1 = (g + h) - (i + j)
```

Note the introduction of the `sub` instruction, unlike `add`, the order of the operands matters; obviously $(a - b)$ is not the same as $(b - a)$.

Memory Operations



Many useful data structures operate on buffers or blocks of memory and not individual words or bytes

- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte
- RISC-V is Little Endian
 - Least-significant byte at least address of a word
 - *c.f.* Big Endian: most-significant byte at least address
- RISC-V does not require words to be aligned in memory unlike other ISAs
 - However, for this class, and in most real implementations, RISC-V aligns to words
 - For RV32, 64, 128, alignment is to 32 bits, 64 bits, and 128 bits respectively



Array Example

Arrays are programming abstractions provided by higher level languages, they don't really exist at the processor level – it's all about calculating the memory address of each array element.

```
int a[3] = {1,2,3};    //assume a is loaded into
                        //memory location 0x1000
```

a is a pointer to this memory so $\&a = \&a[0] = 0x1000$

0x00001000	0x00000001	$a[0]=1, \&a[0] = 0x1000$
0x00001004	0x00000002	$a[1]=2, \&a[1] = 0x1004$
0x00001008	0x00000003	$a[2]=3, \&a[2] = 0x1008$

in general, $\&a[i]$, where $i = \{0, 1, \text{ or } 2\}$
 $= \underline{\&a + (i * 4)}$ ← **base_address + offset**

$a[0]=1000+(0*4)=1000; \quad a[1]=1000+(1*4)=1004; \quad a[2]=1000+(2*4)=1008$



Load and Store

- RISC-V uses a Load/Store architecture
- **All arithmetic operations must take place in registers**
- To manipulate memory, a value must:
 - Be first loaded in from memory to a register
 - Operated on within RISC-V registers
 - The result stored back into memory
- RISC-V provides load and store operations for these purposes

Example: We want to change the value of `a[1]` from 2 to 4

a =

0x00001000	0x00000001
0x00001004	0x00000002
0x00001008	0x00000003

STEPS

1. Calculate the address of `a[1]`
 $= 1000 + (1 * 4) = 1004$
2. Load memory address 1004 which is 2 into a register
3. Change the value in the loaded the register to 4
4. Save the value back to memory location 1004



Load and Store

The basic RISC-V load and store instructions are load-word **lw** and store-word **sw**. These instructions are formatted as follows:

```
lw R1, immediate(R2)  # R1 = &(R2 + immediate)
sw R1, immediate(R2)  # &(R2 + immediate) = R1
```

Example Revisit: We want to change the value of a[1] from 2 to 4

a =

0x00001000	0x00000001
0x00001004	0x00000002
0x00001008	0x00000003

```
#assume &a = 1000 in register t0
#assume new value of 4 is in register t1

lw  t2, 4(t0)      #t2 = &(1000+4) = &1004 = 2
add t2, zero, t1   #t2 = 0 + 4 = 4
sw  t2, 4(t0)      #&(1000+4) = &(1004) = 4
```



Revisiting Load and Store – Another Example

Consider the following C code:

```
a[12] = h + a[8];
```

Assume the address of `a` &`a` is in register `s0`, and the value of `h` is in register `s1`

```
lw    t0, 32(s0)      # t0 = a[8] = (&a)+(4*8)
add   t0, t0, s1      # t0 = h + a[8]
sw    t0, 48(s0)      # a[12] = (&a)+(4*12) = &a+48 = t0
```

Note that this would be invalid:

```
add   48(s0), s1, 32(s0) #would be nice, but not allowed
```

Why is the above not allowed?

Summary Part 1: Intro to Programming Directly on the RV Processor



- RV is a modern, open-source processor, with specifications to support 16-, 32-, 64-, and 128-bit hardware addressing
- We will be using the RV32IM specification in this class
- RV32IM (RV here after) addresses memory by default using 4-byte (32-bit) words
- RV is a load / store architecture – all operations that mutate values require the values to be in registers
- RV provides 5 special purpose and 27 general purpose registers
- All addressable memory is divided into 4 distinct areas - **.text**, **.data**, **.globl**, and a dynamic memory area used for the heap and stack

Summary Part 1: Intro to Programming Directly on the RV Processor



- The RV dynamic memory area is organized such that the heap grows with increasing memory addresses, and the stack grows with decreasing memory addresses
- Must be careful that the stack and heap do not intersect (out-of-memory)
- We use assembler instructions to write code directly on the processor, they take the form of an opcode and register references.
- When we program in higher level languages, constructs like variables and arrays are really abstractions, at the processor level they translate into address calculations and operations performed against values in registers.
- RV as a load-store architecture must optimize the memory hierarchy since different storage types have different performance characteristics



Programming with the RISC-V Instruction Set

Part 2: The Basics of Assembly Programming

Dr. Brian Mitchell



Bitwise Logical Operations

Lets start by adding more instructions to our programming vocabulary

- Instructions for bitwise manipulation
- Useful for extracting and inserting groups of bits in a word

Operation	C	Java	RISC-V
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise XOR	^	^	xor, xori
Bitwise NOT	~	~	Implemented with xor



(Logical) Shift Operations

- Useful for integer multiplication and division by powers of 2
 - Assume $t2 = 3$, and $t1 = 16$

`sll t0,t1,t2`

t2	0000 0000 0000 0000 0000 0000 0000 0011
t1	0000 0000 0000 0000 0000 0000 0001 0000
t0	0000 0000 0000 0000 0000 0001 1000 0000

`srl t0,t1,t2`

t2	0000 0000 0000 0000 0000 0000 0000 0011
t1	0000 0000 0000 0000 0000 0000 0001 0000
t0	0000 0000 0000 0000 0000 0000 0000 0010

Note we shifted by 3 bits and $2^3 = 8$

So $16 \ll 3 = 16 * 8 = 128$ and $16 \gg 3 = 16 / 8 = 2$

NOTE: Real integer multiplication and division is a relatively slow operation, so when we need to multiply or divide by powers of 2 we will use very fast shift operations instead. We already saw an example where we need to frequently multiply by 4, remember it?

AND Operations



- Useful to mask bits in a word
 - Select some bits, clear others to 0
 - When bit is 0, it will **clear** the other bit, when bit is 1 it will **copy** the other bit

and t0, t1, t2

t2	0000 0000 0000 0000 0000 1101 1100 0000
t1	0000 0000 0000 0000 0011 1100 0000 0000
t0	0000 0000 0000 0000 0000 1100 0000 0000



OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged
 - When bit is zero it will **copy** the other bit, when bit is 1 it will **set** the other bit regardless of its value

or t0, t1, t2

t2	0000 0000 0000 0000 0000 1101 1100 0000
t1	0000 0000 0000 0000 0011 1100 0000 0000
t0	0000 0000 0000 0000 0011 1101 1100 0000

XOR Operations



- Useful to determine bit equality in a word (or bits)
 - xor is zero when the bits are the same and one when different

xor truth table

A (Input 1)	B (Input 2)	$X = A'B + AB'$
0	0	0
0	1	1
1	0	1
1	1	0

`xor t0, t1, t2`

t2	0000 0000 0000 0000 0000 1101 1100 0000
t1	0000 0000 0000 0000 0011 1100 0000 0000
t0	0000 0000 0000 0000 0011 0001 1100 0000



NOT Operations

- Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
- RISC-V does NOT have a NOT operation, but one can be implemented using XOR.
- Note that $\text{NOT}(x) = \text{XOR}(x, 1)$

```
xor    t0, t1, t2 # assume t2= 0xFFFFFFFF (all 1's)
xori   t0, t1, 0xFFFFFFFF #alternative format
```

t2	1111 1111 1111 1111 1111 1111 1111 1111
t1	0000 0000 0000 0000 0011 1100 0000 0000
t0	1111 1111 1111 1111 1100 0011 1111 1111

Can you think of a reason why RISC-V does not have a not operation and instead requires an alternative implementation?

Sidebar: Why we don't have many nice things when directly programming the processor



Memory Address	Memory Contents
0x00000000 to 0x0000FFFF	.text segment Program Instructions
0x00010000 to 0x0001FFFF	.data (constant) segment Read only constant values and variables
0x00020000 to 0x0002FFFF	.globl segment Read/write global variables
0x00030000 to 0xFFFFFFFF	DYNAMIC MEMORY HEAP (aka, malloc) ↓ ↑ STACK (aka, functions)

All instructions in RV are encoded into a sequence of 32-bit values (words) and stored in the **.text** segment.

- Every RV instruction (aka **an opcode which is a number**) along with its operands must fit within 32 bits.
- The RV designers decided to allocate **7 bits** to the opcode, leaving **25 bits** to specify the instruction parameters
- $2^7 = 128$ represents the maximum number of opcodes supported.
- 128 is a hard limit, so if there are multiple ways to do something, only one will be supported on the processor directly.

Sidebar: We get some of the nice things back using a tool called the assembler



Your RV Assembly Code

The Assembler Tool
(Pass 1)

Translated and
Optimized RV
Assembly Code

The Assembler Tool
(Pass 2)

Binary Code that
can be loaded into
the .text segment

Examples - Consider:

```
xori t0,t0,0xFFFFFFFF = not t0,t0  
add  t0,zero,t1        = mv  t0,t1
```

There is **NO NEED** for a **not** or move (**not**) operation since they can be implemented using **xori** and **add** instead

This consequence **REDUCES** program readability, so the assembler supports a set of instructions not implemented on the processor and translates them for you.

Also recall registers **t0** and **t1** do not exist ; these names convey programmer intent to promote readability, the assembler also translates these into real register names:

```
xori x5,x5,0xFFFFFFFF = not t0,t0  
add  x5,zero,x6        = mv  t0,t1
```



Immediate Instructions

- So far - all instructions we examined required all data to be in registers
- Sometimes **we want to use constants** in instructions
- Instructions that support constant parameters are called immediate instructions
- We have already seen 2 immediate instructions `lw` and `sw`

```
lw R1, immediate(R2) # example lw t0, 0(t1)
sw R1, immediate(R2) # example sw t2, 8(t3)
```

Assume for each instruction *t#* is preloaded with a value where # is the register number, eg., *t4=4*, *t5=5*

Immediate formats for other instructions we have already seen

Instruction	Immediate Version	Example	Comments
add	addi Rd,Rs,immediate	addi t0,t1,4	$t0 \leftarrow t1 + 4 = 1 + 4 = 5$
sub	addi Rd,Rs,immediate	addi t0,t5,-3	Notice there is no subi, we use a negative immediate value. $t6 \leftarrow 5 - 3 = 2$
sll	slli Rd,Rs,immediate	slli t0,t1,2	$t0 \leftarrow 1 \ll 2 = 1 * 4 = 4$
srl	arli Rd,Rs,immediate	srli t0,t4,1	$t0 \leftarrow 4 \gg 1 = 4 / 2 = 2$
and	andi Rd,Rs,immediate	andi t0,t3,1	$t0 \leftarrow 0b011 \& 0b001 = 0b001 = 1$
or	ori Rd,Rs,immediate	ori t0,t4,1	$t0 \leftarrow 0b100 \mid 0b001 = 0b101 = 5$
xor	xori Rd,Rs,immediate	xori t0,t5,3	$t4 \leftarrow 0b101 \mid 0b010 = 0b110 = 6$

Abstract Processor Execution Overview



We have seen several instructions so far that can be executed by the RISC-V process, however, how does execution work?

Recall this example:

```
# in C f = (g + h) - (i + j);  
add t0, s3, s4      # t0 = g + h  
add t1, s5, s6      # t1 = i + j  
sub s3, t0, t1      # f = t0 - t1 = (g + h) - (i + j)
```

The first thing that happens is that the instructions shown above are assembled into their binary equivalent. Note, each instruction in RISC-V is encoded as a 32 bit word (aka it has a binary value):

```
0: 014982b3 # add t0,s3,s4  
4: 016a8333 # add t1,s5,s6  
8: 406289b3 # sub s3,t0,t1
```

Online RISC-V Assembler
<https://riscvasm.lucasteske.dev/#>

We will learn how to do this ourselves a little later

Abstract Processor Execution Overview – the **pc**

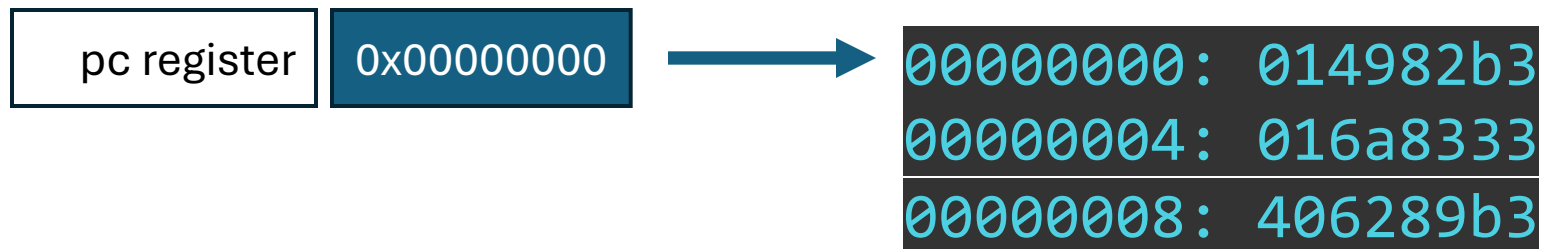


Since the instructions are really just 32 bit binary values they can be loaded into memory, lets say we load them into memory starting at address 0x00000000:

```
00000000: 014982b3
00000004: 016a8333
00000008: 406289b3
```

We have already discussed RISC-V registers (x0-x31), which are used to write instructions that run on the processor. **There is another register called the program counter (pc) that the processor uses to execute instructions.**

To start executing the program, the **pc** register is set to the address of the first instruction:

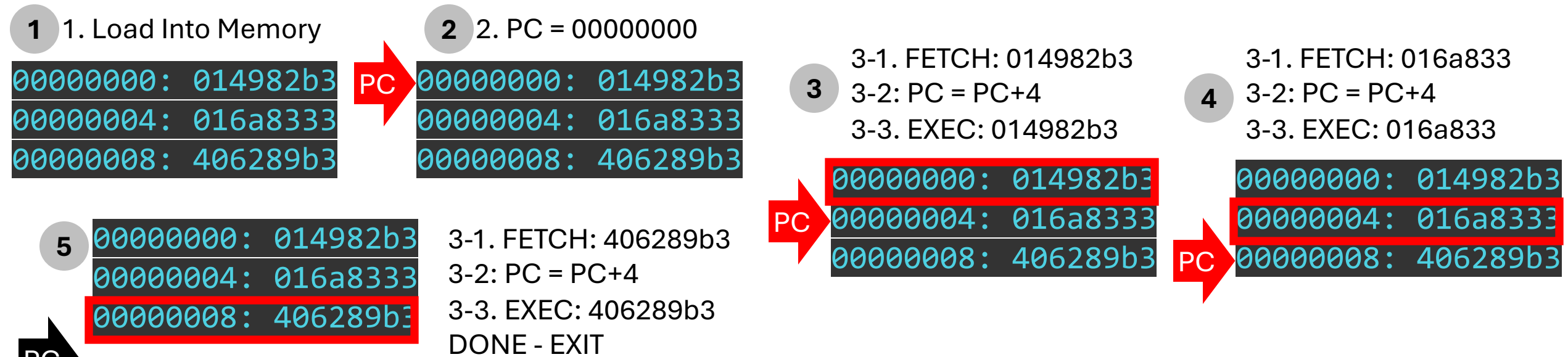




Abstract Processor Execution Overview

With the pc is set, the execution process uses the following algorithm:

1. Load program into memory
2. Set pc to address of first instruction
3. While there are more instructions to execute (loop):
 - 3-1: Fetch instruction from memory at (address = pc) into the processor
 - 3-2: $PC \leftarrow PC + 4$ # now pc points to the next instruction
 - 3-3: Execute the instruction that was fetched in step 3-1





More on the program counter (**pc**) register

From below we see that RV has 32 registers we can use, and none of them are labeled **pc**.

Register	ABI Name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5–7	t0–2
x8	s0/fp
x9	s1
x10–11	a0–1
x12–17	a2–7
x18–27	s2–11
x28–31	t3–6

- The **pc** register is an **INTERNAL** register used by the processor – we are unable to manipulate it directly
- The processor uses the **pc** register to **blindly** fetch the next instruction to be executed – it wants to be as fast as possible.
- Allowing the programmer to directly modify the **pc** register could lead to many nasty software bugs and security issues.
- We will be digging into other instructions that allow us to change flow control via jumping and branching. Executing these instructions will result in indirectly controlling the **pc** register.



A First Example

Lets see if we can write the code for this “C” program:

```
void swap(int v[], int k){  
    int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

```
int main(){  
    int v[] = {1,2,3,4,5};  
    swap(v,2);  
}
```

Unfortunately, we still don't have all of the information that we need to write the entire program, but we do know how to write the body of the swap() function.

What do you think we are missing?



A First Example – The body of `swap()`

Assume register `a0 = &v`, `a1 = k`

```
void swap(int v[], int k){  
    int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

swap:

```
slli a1,a1,2      #a1 = 4*a1  
add t0,a0,a1      #a0 = &v[k] = &v + 4*k  
lw t1,0(t0)       #t1 = v[k]  
lw t2,4(t0)       #t2 = v[k+1]  
sw t1,4(t0)       #v[k] = v[k+1]  
sw t2,0(t0)       #v[k+1] = temp  
jr ra             #return
```

Note that **swap:** is a label. We can name it anything we want. At runtime it is an alias for the address of the instruction on the same line. **Thus swap will be the address of the `slli a1,a1,2` instruction when we use it in our code.**



A First Example

What don't we know how to do yet?

```
void swap(int v[], int k){  
    int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

4

```
1 int main(){  
    int v[] = {1,2,3,4,5}; 2  
    swap(v,2); 3  
} 5
```

1. We don't know how to set the pc register to start execution at the main() function
2. We don't know how to setup and initialize an array.
3. We don't know how to call the swap function()
4. We don't know how to return from the swap function back to main()
5. We don't know how to stop execution when main() ends



A First Example

Recall that RISC-V has a well-defined memory layout

We need to clearly define where our code goes and where our data goes in our program

Memory Address	Memory Contents
0x00000000 to 0x0000FFFF	.text segment Program Instructions
0x00010000 to 0x0001FFFF	.data (constant) segment Read only constant values and variables
0x00020000 to 0x0002FFFF	.globl segment Read/write global variables
0x00030000 to 0xFFFFFFFF	DYNAMIC MEMORY HEAP (aka, malloc) ↓ ↑ STACK (aka, functions)

```
void swap(int v[], int k){  
    int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

```
int main(){  
    int v[] = {1,2,3,4,5};  
    swap(v,2);  
}
```



A First Example

Recall that RISC-V has a well-defined memory layout

We need to examine how to interact with the **.data** section load and store information

Memory Address	Memory Contents
0x00000000 to 0x0000FFFF	.text segment Program Instructions
0x00010000 to 0x0001FFFF	.data (constant) segment Read only constant values and variables
0x00020000 to 0x0002FFFF	.globl segment Read/write global variables
0x00030000 to 0xFFFFFFFF	DYNAMIC MEMORY HEAP (aka, malloc) ↓ ↑ STACK (aka, functions)

.data

```
buff: .word 1,2,3,4,5      # array of words
wrd:  .word 0              # single word
msg:  .asciiz "Hello, World!\n" # a string
char: .byte '\n'          # a 8-bit char
sbyt: .byte 55             # a byte value
byta: .byte 10,20,30,40    # a byte array
zbuf: .space 5             # a 5 word buffer
```

New Helper Instructions

```
la t0, wrd      #loads address of "wrd" into t0
li t1, 5        #loads the value of 5 into t1
```

Note the **li** instruction is a helper to place an integer value directly into a register. The **la** instruction gets the address of a label from the **.data** section. From there we can use **lw** and **sw** to load and save values to memory locations

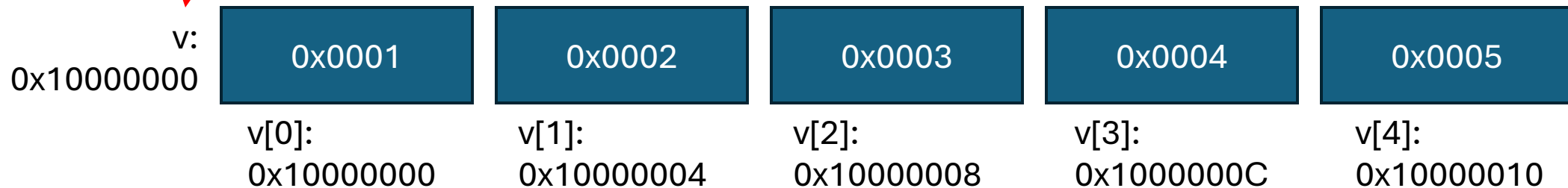


A First Example - "Arrays"

Constructs like arrays really don't exist at the machine level, its just allocated space that we manage directly

```
int main(){  
    int v[] = {1,2,3,4,5};  
    swap(v,2);  
}
```

```
.data  
v:      .word 1,2,3,4,5 #assume loaded at 0x10000000  
.text  
la t0, v      # t0 <- &v == v[0]  
lw t1, 0(t0)  #loads v[0] into t1  
lw t2, 4(t0)  #value in v[1]  
li s0, 10  
sw s0, 8(t0)  #store 10 in v[2]
```



For a word array the address of of $v[i]$ is translated to $\&v + (i * 4)$

$v[0] = \&v + 0*4$	$v[1] = \&v + 1*4$	$v[2] = \&v + 2*4$	$v[3] = \&v + 3*4$	$v[4] = \&v + 4*4$
$v[0] = \&v + 0$	$v[1] = \&v + 4$	$v[2] = \&v + 8$	$v[3] = \&v + 12$	$v[4] = \&v + 16$



A First Example

```
1 .data
2 v: .word 1,2,3,4,5

1 .text
3 .globl main

4 main:
    #code goes here

4 swap:
    #code goes here
```

1. Setup the data and text sections using the .data and .text assembler directives
2. Allocate space for the v[] in the .data segment, initialize it as well
3. Define the entry point of the program as a global variable
4. Define the labels for where main and swap will go

```
.data
v: .word 1,2,3,4,5
```

```
.text
.globl main
```

```
main:
    la    a0, v           #a0 = &v
    li    a1, 3           #a1 = k = 3
5    jal  swap

    # Exit the program
    li    a0, 10          # Exit code for ecalls
6    ecalls

swap:
    #code goes here
```

5. We use the jal instruction to make the function call to swap. jal is the jump and link instruction
6. Notice the swap: label would be right after the jal call if we didn't have the code in 6. You need to tell the runtime that you are exiting. That is done via a system call using the ecalls instruction



A First Example

```
.text
.globl main
main:
    la  a0, v           #a0 = &v
    li  a1, 3           #a1 = k = 3
    7  jal swap

    # Exit the program
    9  li a0, 10         # Exit code for ecalls
    ecall

#swap(v,k)
swap:
    slli a1,a1,2         #a1 = 4*a1
    add  t0,a0,a1        #a0 = &v[k] = &v + 4*k
    lw   t1,0(t0)        #t1 = v[k]
    lw   t2,4(t0)        #t2 = v[k+1]
    sw   t1,4(t0)        #v[k] = v[k+1]
    sw   t2,0(t0)        #v[k+1] = temp
    8  jr  ra            #return
```

7. The jal function has a side effect – it does 2 things
FIRST, it sets the ra register to PC+4, then sets
PC to the address of swap.
8. Since the jal instruction set the ra register to the
line of code after the call, jr ra jumps to the
address stored in the ra register, therefore it
returns back to the next instruction after the
jal call.
9. We need a way to tell the runtime environment that
our program is done executing. This is done via
executing a system call (syscall). This is done by
putting the syscall number (10 in this case) in
register a0, and then executing the ecall
instruction.

LET'S SEE THIS CODE IN ACTION

Constant Substitutions



Keeping code readable, especially with assembly can be a complex undertaking since this environment is somewhat minimalistic.

Remember the code to exit our program using `ecall 10`:

```
li a0, 10
ecall
```

I don't know about you, but I always need to look these up. Programming languages like C allow us to define constants using `#define`. We can do the same thing with RISC-V assembler by using the `.equ` directive:

```
.equ EXIT_SYSCALL_NUMBER, 10 #new constant name
...
li a0, EXIT_SYSCALL_NUMBER
ecall
```



Constant Substitutions (Limitations)

Using constants in your code improves readability. Lets revisit the code below:

```
.equ EXIT_SYSCALL_NUMBER, 10 #new constant name
...
li a0, EXIT_SYSCALL_NUMBER
ecall
```

We will get into this later, but there really is no such thing as the **li** instruction – it gets translated into **addi a0,a0,EXIT_SYSCALL_NUMBER = addi a0,a0,10**.

The **addi** instruction only allows for 2^{12} as the maximum value for the immediate field. Thus, the range of values you can use must be between -2048 and +2047.

Jump Instructions



The 3 basic Jump instructions in RISC-V

- | | |
|----------------------------|---|
| <code>j address</code> | Jump to address, where address is a label
example: <code>j swap</code> |
| <code>jal address</code> | Jump to address, where address is a label and also set the ra register to pc+4 . example: <code>jal swap</code> |
| <code>jr [register]</code> | Jump to the address in the specified register.
example: <code>jr ra, jr t0</code> |

Jump instructions alter the value of the PC register to change program flow control

Jump Instructions - Advanced



Jump and Link Register - JALR

```
jalr [register-1], immediate(register-2)
```

This is a more complex, but flexible version of `jal`. In this version you jump to the address of `immediate+register-1`, and store `pc+4` in `register-2`

```
jal swap #jump to swap,      la t0, swap    #put &swap in t0
    #ra <- PC+4              jalr ra,0(t0) #jump to swap,
                               #ra <- PC+4
```

Provides flexibility to use alternative registers for return address, also flexibility in jumping to addresses by offsets. Think function pointers. We will not be using it in this class, but you should be aware it exists.

Jump Instructions and the Program Counter



Recall the abstract RISC-V Execution Algorithm

1. Load program into memory
2. Set pc to address of first instruction
3. While there are more instructions to execute:
 - 3-1: Fetch instruction from memory at (address = pc) into the processor
 - 3-2: $PC \leftarrow PC + 4$ # now pc points to the next instruction
 - 3-3: Execute the instruction that was fetched in step 3-1

How does this work with jump instructions?

For the **j** and **jr** instructions 3-3 simply changes the **pc** register to the address specified

For the **jal** and **jalr** instructions 3-3 changes the **pc** register to the address specified and also saves **pc+4**, which you should be able to see has already been calculated in step 3-3 (aka – it's the current value of **pc** at the time of execution in 3-3).



RISC Architectures

- The RISC-V processor implements a RISC architecture
- RISC = Reduced Instruction Set Computer
- The other type of architecture is called CISC or Complex Instruction Set Architecture

RISC Architectures

- Minimal number of highly optimized instructions
- All instructions execute in a single clock cycle, for now, just think that they all take the same amount of time
- Examples: RISC-V, MIPS, ARM

CISC Architectures

- Provides a robust and flexible set of instructions
- Instructions take different numbers of clock cycles, for now, just think that not all instructions execute in the same amount of time
- Examples: Intel



Machine Language

- Machine Language is the binary representation of instructions
- Computers only understand 1's and 0's
- Since RISC-V is a RISC architecture many things are standardized:
 - 32-bit data
 - 32-bit instructions
 - 32-bit addresses
- All instructions can be directly translated into their equivalent machine language representation

Machine Language Example



[add Rd, Rs1, Rs2]

add t0,t1,t2 #t0 = t1+t2

Register	ABI Name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5–7	t0–2
x8	s0/fp
x9	s1
x10–11	a0–1
x12–17	a2–7
x18–27	s2–11
x28–31	t3–6

First, translate from ABI register names to the physical RISC-V register number. From above **t0=x5, t1=x6, t2=x7**

Instruction Format for add

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Next, use the RISC-V reference card to look up the funct7, funct3 and opcode for the add instruction. **opcode=51, funct7=funct3=0**

0	7	6	0	5	51		
0000000	00111	00110	000	00101	0110011		
0000	0000	0111	0011	0000	0010	1011	0011
0	0	7	3	0	2	b	3

add t0,t1,t2 #= 0x00730b3

Machine Language Example



```
[sll Rd, Rs1, Rs2]
sll s0, s1, s2      #s0 = s1 << s2
```

Register	ABI Name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5–7	t0–2
x8	s0/fp
x9	s1
x10–11	a0–1
x12–17	a2–7
x18–27	s2–11
x28–31	t3–6

ABI register names **s0=x8, s1=x9, s2=x18**

Instruction Format for sll

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Next, use the RISC-V reference card to look up the funct7, funct3 and opcode for the sll instruction. **opcode=51, funct3=1, funct7=0**

0	18	9	1	8	51		
0000000	10010	01001	001	00101	0110011		
0000	0001	0010	0100	1001	0010	1011	0011
0	1	2	4	9	2	b	3

```
sll s0, s1, s2      #= 0x012492b3
```



Machine Language Format Observations

Instruction Format for `and` and `sll`

	funct7	rs2	rs1	funct3	rd	opcode
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
add	0000000	00111	00110	000	00101	0110011
sll	0000000	10010	01001	001	00101	0110011

Notice the opcode field is 7 bits – thus the total number of opcodes is 2^7 or 128

Notice all register fields are 5 bits – thus we can support 2^5 or 32 registers

Thus for instructions that take 3 registers, we need 7 bits for the opcode and 3×5 bits for the 3 registers. Thus a total of 22 bits are required.

The remaining 10 bits are broken down into the funct7 and funct3 fields.

In this case the opcode + funct3 field is used to determine the exact instruction (add or sll)

Machine Language Example



```
[slli Rd, Rs1, immediate]
slli s0, s1, 7      #s0 = s1 << s2
```

Register	ABI Name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5–7	t0–2
x8	s0/fp
x9	s1
x10–11	a0–1
x12–17	a2–7
x18–27	s2–11
x28–31	t3–6

ABI register names **s0=x8, s1=x9, immed = 7**

Instruction Format for slli

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

Next, use the RISC-V reference card to look up the funct3 and opcode for the slli instruction. **opcode=19, funct3=1**

7	9	1	8	19
000000000111	01001	001	01000	0010011
0000 0000 0111 0100 1001 0100 0001 0011				
0 0 7 4 9 4 1 3				

```
slli s0, s1, 7      #= 0x0074913
```



RISC-V Instruction Formats

Not sure if you noticed, but the `add` and `sll` instructions were encoded the same way, but the `slli` instruction was encoded differently

RISC-V has multiple encoding formats based on the type of instruction.
`add` and `sll` are R-type instructions and `slli` is an example of an I-type instruction.

<https://github.com/jameslzhu/riscv-card/blob/master/riscv-card.pdf>

Core Instruction Formats

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

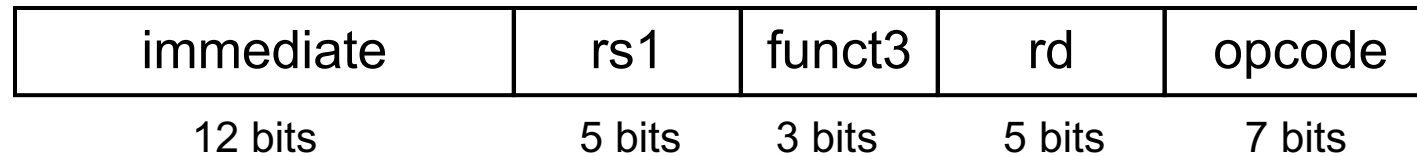


RISC Architecture Machine Language Tradeoffs

As an example, let's look at the load word and store word instructions

```
[lw/sw Rd, immediate(Rs1)]  
# lw t0, 4(t1), sw t0,4(t1)
```

I-Type Format for lw and sw



From the data sheet: **lw**: opcode=3, funct3=2
sw: opcode=35, funct3=2

Where is the design compromise?
Good design often requires compromises

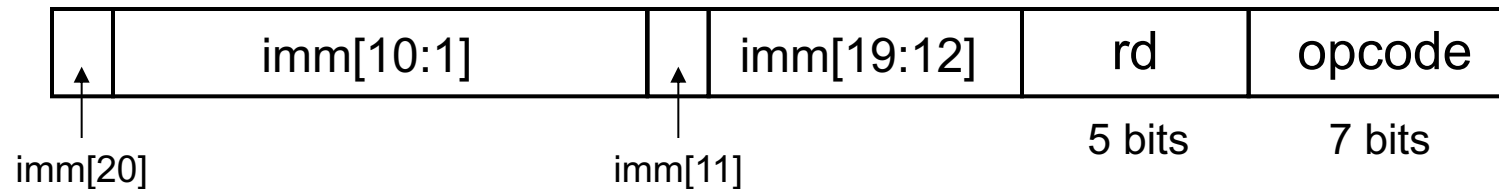


RISC Architecture Machine Language Tradeoffs

As another example, let's look at the jump and jump and link instructions

j swap
jal swap

J-Type Format for j and jal



Anything seem strange about how the 20 bits for the address are encoded?

In the second part of this course, we will see why this encoding makes sense for efficiency – for now just accept it as part of the specification 😊



RISC Architecture Machine Language Design makes efficiency a **first-class** concern

```
addi t0,t1,100
```

```
srli t0,t1,4
```

```
j swap
```



```
li t2,100  
add t0,t1,t2
```

```
li t2,4  
srli t0,t1,t2
```

```
la t0, swap  
jr t0
```

We have a constraint on the number of opcodes, and things like the size of the immediate values and address sizes in the jump instruction. The goal is to be as efficient as possible. One instruction is twice as fast as two so the designers tried to optimize around fast common operations.

Summary Part 2: The basics of assembly programming



- We examined several different types of RISC-V instructions – many with basic and immediate versions
- How memory is structured and how programs and data are put into memory
- Assembler directives to put code in the **.text** area and data in the .data area. The assembler directive **.globl** to indicate where execution should start (we have to export that address as a global variable)
- How to use jump instructions to implement simulate function calls where the flow of control is changed requiring a way back to where we came from
- How to do array processing translating array addressing such as **a[5]** into base and offset addressing **a[5]=&a[0]+(5*4)**



Summary Part 2: The basics of assembly programming



- RV has several different instruction formats – we looked at the R,I and J formats so far
- How to use the instruction formats to manually generate machine code using the RV data card
- Optimizations made by the designers of RV including encoding all instructions as 7 bit fields resulting in a hard cap of 128 overall instructions
- How the assembler helps us to overcome these tradeoffs via doing instruction and register translation
- What system calls are, and why we need them, also how they are invoked via the **ecall** instruction.
- How the abstract execution model works by managing the **pc** register – **pc+4** is the default, but we can directly manipulate **pc** the by using **j**, **jal**, and **jr** instructions



Programming with the RISC-V Instruction Set

Part 3: Conditional Programming, Looping and Branching

Dr. Brian Mitchell

Conditional Branch Instructions



- We have already learned that processors (including RISC-V) execute instructions that are loaded into memory in order by default.
 - Instructions are 32 bits (4 bytes) in the RV32 architecture
 - By default, after fetching the current instruction from the address in the **pc** register, the **pc** is advanced to **pc+4**
- We have also seen how we can alter the default control flow behavior by changing the **pc** directly using jump instructions – **jr** and **jal**
- We now want to examine how we can branch to a labeled instruction based on if a condition is true or not.

```
if (a == b) {  
    // body of the true  
    // if condition  
}  
//next instruction to be  
//executed
```

```
if (a == b) {  
    // body of the true  
    // if condition  
} else {  
    // body of the not  
    // true condition  
}
```

```
while (a < b) {  
    // body of the loop  
    // if condition  
}  
//next instruction to be  
//executed
```

and so on...

Conditional Branch Instructions



RISCV has 6 built-in branch instructions

B-Type RISC-V Instruction Format

immed[12 10:5]	rs2	rs1	funct3	immed[4:1 11]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Instruction	Example	Interpretation
beq	beq t0, t1, label	If t0 == t1 jump to label
bne	bne t0, t1, label	If t0 != t1 jump to label
blt	blt t0, t1, label	If t0 < t1 jump to label
bge	bge t0, t1, label	If t0 > t1 jump to label
bltu	bltu t0, t1, label	If t0 < t1 jump to label
bgeu	bgeu t0, t1, label	If t0 > t1 jump to label

Note that beq, bne, blt, bge treat the data inside registers as signed values, and that bltu and bgeu treat the values as unsigned integers

Sidebar: Signed and Unsigned Integers



In base-10 we treat numbers as positive by default, and add the – sign for negative numbers.

In binary define the value of an **unsigned** number as follows:

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

Example: 7 in 8 bits = 00000111
 $= 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 4 + 2 + 1 = 7$

In binary define the value of an **signed** number as follows:

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

Example: -7 in 8 bits = 11111001
 $= -1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3$
 $+ 1 * 2^1 = -128 + 64 + 32 + 16 + 8 + 1$
 $= -7$

Signed numbers in binary are represented using 2's complement (1) Invert all bits; (2) add 1

7 = 00000111 (invert) 11111000 (add 1) 11111001 = -7

Sidebar: Sign Extension in Binary

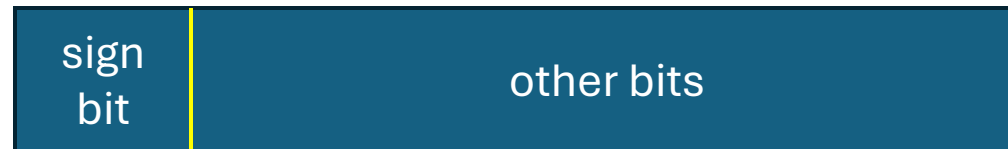


Consider the number +7 in binary = 0b111. It takes 3 bits. What would +7 be in 5 bits? 0b00111. What about 10 bits 0b0000000111

Just like decimal numbers when using signed numbers we can sign extend by adding zeros to the front of the number

Consider the number -7 in binary = 0b1001. It takes 4 bits. What would -7 be in 5 bits? 0b11001. What about 10 bits 0b1111111001

In binary sign extending a negative value means adding a sequence of ones.



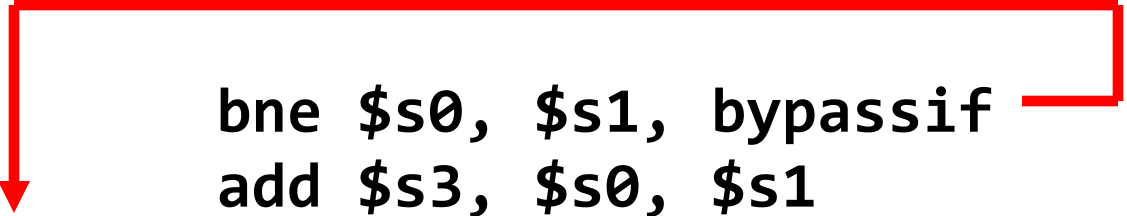
In general the sign bit is the most-significant-bit, and will be 0 for + numbers and 1 for negative numbers. To sign extend we take the sign bit and repeat it

Control



- **Decision making instructions**
 - **Alter the control flow – change the "next" instruction to be executed**
- **Example: if (i==j) h = i + j;**

```
        bne $s0, $s1, bypassif  
        add $s3, $s0, $s1  
bypassif: # handle the next instruction
```

A red arrow originates from the right side of the `bne $s0, $s1, bypassif` instruction and points down to the `bypassif:` label, illustrating the jump in control flow.

Notice that by default the pc advances to pc+4 so if we don't want to take the if condition we must jump around it.

Normally we think of an if statement around what to do if some condition is true, when working with the machine this is the default, think instead about where to jump if the condition is NOT true.



If-Then-Else Structure

- Example:

if (i!=j)	beq s4, s5, Equal	#Handle else condition
h=i+j;	add s3, s4, s5	#body of if
else	j GoOn	#jump over else condition
h=i-j;	Equal: sub s3, s4, s5	#body of else
j++;	GoOn: addi s5, s5, 1	#first instruction after if - j

Way to think about If-Then-Else:

1. Setup 2 labels (1) at beginning of else block; and (2) at first statement after else block.
2. Setup branch condition from if expression so that if true you jump to the else label
3. Setup unconditional jump at end of if block to "jump around" the else block

Looping



Example: Given an array of data and a constant, count how many times in a row that constant appears starting from the beginning of the array

.data

```
loop_buff: .word 2,2,1,5
```

.text

loop-demo:

```
#while (loop_buf[i] == k) i += 1;
# i in s3
# k in s5
# address of loop_buff in s6
##### INIT #####
addi s3, x0, 0      # i = 0
addi s5, x0, 2      # k = 2
la s6, loop_buff    # s6 = &loop_buff
##### END INIT #####
```

loop:

```
slli t0, s3, 2      # t0 = 4*i
add t0, t0, s6       # t0 = &loop_buff[i]
lw t1, 0(t0)        # t1 = loop_buff[i]
bne t1, s5, end-loop # (loop_buff[i] != k)
                        # goto end-loop
addi s3, s3, 1       # i += 1
j loop              # do the loop
end-loop:
#result is in s3
```

Looping



Notice the similarity of coding loops with coding if statements:

```
if (condition) {  
    //if block  
}  
//next statement
```

```
branch-statement condition-not-met-label  
    #if block code  
    #if block code
```

```
condition-not-met-label:  
    #next statement
```

```
loop (condition) {  
    //loop block while  
    //condition is true  
}  
//next statement
```

```
loop-label: branch-statement condition-not-met-label  
    #loop block code  
    #loop block code  
j loop-label #jump to the loop
```

```
condition-not-met-label:  
    #next statement
```




Sidebar: Enabling Programmer Readability via assembler translation (aka pseudo-instructions)

Recall the different instruction formats supported by the RISC-V processor:

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

Notice the one thing in common is the opcode field has 7 bits, meaning there are a maximum of 2^7 or 128 instructions that can be supported.

To provide programmer convenience sometimes certain instructions are accepted by the developer tooling and translated into real instructions automatically that can be executed by the processor.

This expands the number of instructions the programmer can use without taking up valuable opcodes, which have a fixed limit

Sidebar: Enabling Programmer Readability via assembler translation (aka pseudo-instructions)



I have already fooled you a few times with this....

```
jal swap      # first save the value of pc+4 in ra, then jump to swap
j  loop       # unconditionally jump to the loop label
```

These are **NOT REAL** RISC-V instructions, the processor does not support them

The **REAL** RISC-V instruction for jal is as follows:

```
jal dest-register, jump-label    # FIRST put pc+4 in dest-register
                                   # SECOND set pc = jump-label-address
```

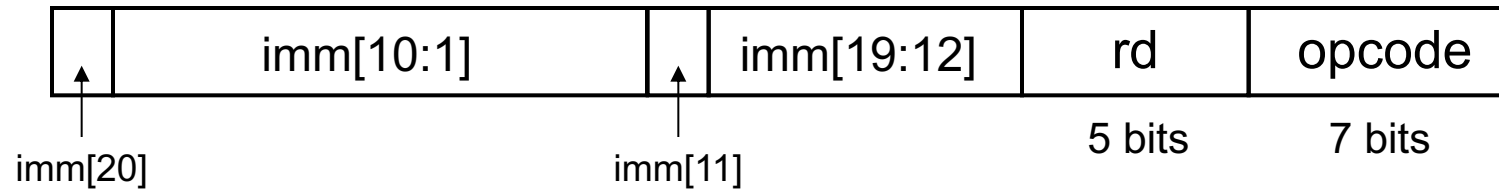
The developer tooling for RISC-V helps you out by allowing special types of instructions, called pseudo-instructions, and translates them to real instructions automatically

```
jal swap      # gets automatically translated to jal ra, swap
j  loop       # gets automatically translated to jal zero, loop
               # the zero register is read-only so this basically drops pc+4
```

Addressing in Jumps



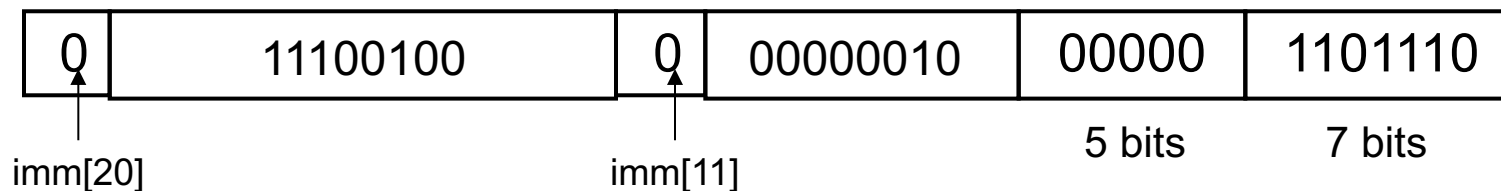
Recall the J-Format Instruction layout:



We use this for instructions like **jal** and **j**

- The opcode is **7 bits** - 1101110
- The Rd register where pc+4 will go is 5 bits, use 00000 if we don't care
- The remaining **20 bits** are treated as an immediate value representing the “address” we want to jump to
- **Ignore the unusual ordering of the 20 bits for now, just know that we have 20 bits to work with**

Example: **j 10000** is really **jal zero,10000**; in machine code the instruction is 0x7100206F



Sidebar: Creative Engineering – How Jumps and Branches really work with immediate values



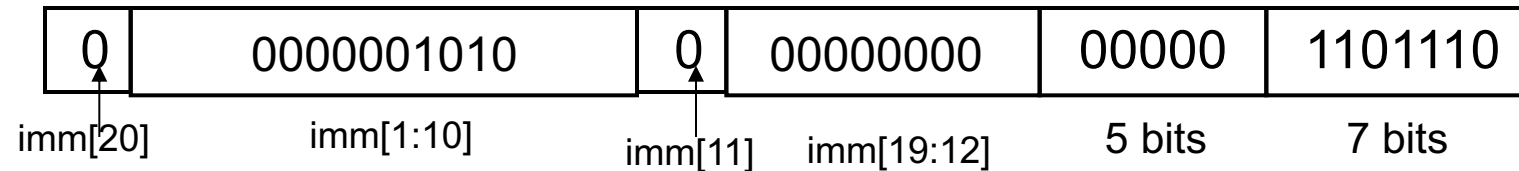
j stupid_demo

```
addi s0, s0, 1
addi s1, s1, 1
addi s2, s2, 1
addi s3, s3, 1
stupid_demo:
jr ra
```

MEMORY	INSTRUCTION
0x00000000	0x0140006F jal x0 20
0x00000004	0x00140413 addi x8 x8 1
0x00000008	0x00148493 addi x9 x9 1
0x0000000C	0x00190913 addi x18 x18 1
0x00000010	0x00198993 addi x19 x19 1
0x00000014	0x00008067 jalr x0 x1 0

The machine code for **j stupid_demo** is 0x0140006F

0x0140006F in binary is 0000 0001 0100 0000 0000 0000 0110 1111



Now lets collect the immediate value in proper bit order: 0000 0000 0000 0000 1010 = 10 = 0x0A in hex

But from above there is no instruction at memory address 0x0000000A
What's wrong? Any Ideas?

Sidebar: Creative Engineering – How Jumps and Branches really work with immediate values



	MEMORY	INSTRUCTION	
j stupid_demo	0x00000000	0x0140006F jal x0 20	The machine code for j stupid_demo is 0x0140006F
addi s0, s0, 1	0x00000004	0x00140413 addi x8 x8 1	
addi s1, s1, 1	0x00000008	0x00148493 addi x9 x9 1	
addi s2, s2, 1	0x0000000C	0x00190913 addi x18 x18 1	
addi s3, s3, 1	0x00000010	0x00198993 addi x19 x19 1	
stupid_demo:	0x00000014	0x00008067 jalr x0 x1 0	
jr ra			

According to the disassembly, the **j** instruction is at memory address 0 (0x00000000) and the immediate value is 0x0A

RISC-V uses PC-Relative addressing. So the immediate value is not an address it gets translated to PC + immediate

So the real jump address is pc+immediate = 0x00000000 + 0x0A = 0x0A

That is not correct either – UGH – Lets Keep Going

Sidebar: Creative Engineering – How Jumps and Branches really work with immediate values



	MEMORY	INSTRUCTION	
j stupid_demo	0x00000000	0x0140006F jal x0 20	The machine code for j stupid_demo is 0x0140006F
addi s0, s0, 1	0x00000004	0x00140413 addi x8 x8 1	
addi s1, s1, 1	0x00000008	0x00148493 addi x9 x9 1	
addi s2, s2, 1	0x0000000C	0x00190913 addi x18 x18 1	
addi s3, s3, 1	0x00000010	0x00198993 addi x19 x19 1	
stupid_demo:	0x00000014	0x00008067 jalr x0 x1 0	
jr ra			

- Recall the j format has can hold a 20 bit immediate, and that number is treated as a signed relative address to the current PC.
- RISC-V uses a 32 bit address space, so 20-bits cannot cover the entire address space (TRADEOFF)
- RISC-V does not like odd memory addresses, so we can get one extra bit for free by shifting left by 1

Sidebar: Creative Engineering – How Jumps and Branches really work with immediate values



	MEMORY	INSTRUCTION
j stupid_demo	0x00000000	0x0140006F jal x0 20
addi s0, s0, 1	0x00000004	0x00140413 addi x8 x8 1
addi s1, s1, 1	0x00000008	0x00148493 addi x9 x9 1
addi s2, s2, 1	0x0000000C	0x00190913 addi x18 x18 1
addi s3, s3, 1	0x00000010	0x00198993 addi x19 x19 1
stupid_demo:	0x00000014	0x00008067 jalr x0 x1 0
jr ra		

The machine code for **j stupid_demo** is **0x0140006F**

The 20-bit immediate value for **j stupid_demo**

000000000000000000001010

Process to convert the 20-bit immediate to a 32 bit immediate value

000000000000000000001010	# Original 20 bit immediate
0000000000000000000010100	# Shift left by 1 bit, ensuring LSB is 0
000000000000000000000000000010100	# Sign extend to 32 bits, this now equals 20

Jump address is now calculated $pc + \text{immediate} = 0 + 20 = 20 = 0x14$

Sidebar: Creative Engineering – How Jumps and Branches really work with immediate values



	MEMORY	INSTRUCTION	
j stupid_demo	0x00000000	0x0140006F jal x0 20	The machine code for j stupid_demo is 0x0140006F
addi s0, s0, 1	0x00000004	0x00140413 addi x8 x8 1	
addi s1, s1, 1	0x00000008	0x00148493 addi x9 x9 1	
addi s2, s2, 1	0x0000000C	0x00190913 addi x18 x18 1	
addi s3, s3, 1	0x00000010	0x00198993 addi x19 x19 1	
stupid_demo:	0x00000014	0x00008067 jalr x0 x1 0	
jr ra			

- Recall the j format has can hold a 20 bit immediate, but we get a 21st bit for free by shifting once to the left – **WHY CAN WE DO THIS SAFELY?**
- That is still only 21 bits but we need 32 bits -> We convert this to 32 bits via sign extension, but this still cant cover the entire address space. **WHY IS THIS NOT A PROBLEM MOST OF THE TIME?**
- RISC-V uses PC relative addressing. **WHY IS THIS A GOOD IDEA?**
- If we needed to jump far away, AKA 21 bits is not enough. **HOW CAN WE DO THIS?**

Target Addressing Example



- Loop code example, Assume Loop at location 80000

```
Loop: slli  t1, s3, 2
      addi  t1, t1, 6
      lw    t0, 0(t1)
      bne   t0, s5, Exit
      addi  s3, s3, 1
      j     Loop
```

80000	00299313
80004	00630313
80008	00032283
80012	01529663
80016	00198993
80020	????????

Lets encode
this instruction

Exit: ...

We want to jump from memory 0x80020 to address 0x8000. Immediate value for jump is -20.
(signed) 20 in binary is 010100 -> Convert to -20 = 101011 + 1 -> 101100. Shift right one to reverse the shift logic for decoding the j instruction -> 10110 which is -10 in decimal. Now sign extend to 20 bits:

11111111111111111110110

Target Addressing Example



```
Loop: slli  t1, s3, 2
      addi  t1, t1, 6
      lw    t0, 0(t1)
      bne   t0, s5, Exit
      addi  s3, s3, 1
      j     Loop
Exit: ...
```

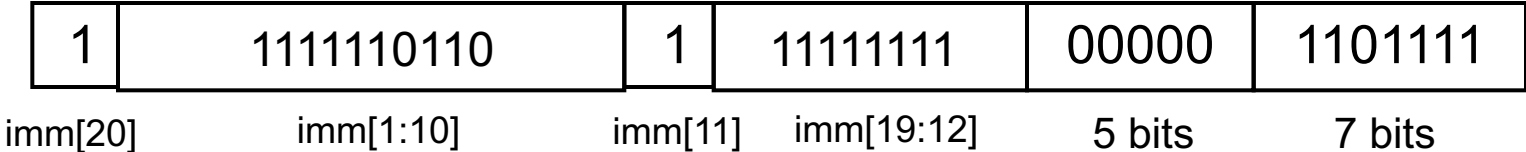
80000	00299313
80004	00630313
80008	00032283
80012	01529663
80016	00198993
80020	???????

Lets encode this instruction

From previous page immediate value:
11111111111111110110

Recall that **j <label>** is a pseudo instruction for
jal zero, label
zero register jal opcode

J-Instruction Format



From previous page immediate value:

1	1111110110	1	11111111	00000	1101111		
1111	1110	1101	1111	1111	0000	0110	1111
F	E	D	F	F	0	6	F

Machine Code
for **j loop** at 0x8020 is:
0xFEDFF06F

Our Next Instruction



```
slt  Rd, Rs1, Rs2    #Set if less then
                        #Rd = (Rs1 < Rs2) ? 1 : 0
slti  Rd, Rs1, const  #Set if less then
                        #Rd = (Rs1 < const) ? 1 : 0
```

slt / **slti** (aka set if less then) is useful in some branching situations. It checks **Rs1** and (**Rs2** / **const**) and sets **Rd** to 1 if **Rs1** is less than (**Rs2** / **const**), otherwise it sets **Rd** to 0.

Example: assume **s2=2**, **s3=3**, **s4=4** :

```
slt  s0, s3, s4  # s0 <- 1 because s3 < s4
slt  s1, s4, s3  # s0 <- 0 because s3 >= s4
slti s0, s3, 4   # s0 <- 1 because s3 < 4
slti s1, s4, 3   # s0 <- 0 because s3 >= 3
```



Branch Instruction Design

RISC-V has a robust set of branching instructions:

Instruction	Example	Interpretation
beq	beq t0, t1, label	If $t0 == t1$ jump to label
bne	bne t0, t1, label	If $t0 != t1$ jump to label
blt	blt t0, t1, label	If $t0 < t1$ jump to label
bge	bge t0, t1, label	If $t0 > t1$ jump to label
bltu	bltu t0, t1, label	If $t0 < t1$ jump to label
bgeu	bgeu t0, t1, label	If $t0 > t1$ jump to label

Question: Could we get by with just **slt** and **beq** and **bne**?



Branch Instruction Design

RISC-V has a robust set of branching instructions:

Instruction	Example	Interpretation
beq	beq t0, t1, label	If $t0 == t1$ jump to label
bne	bne t0, t1, label	If $t0 != t1$ jump to label
blt	blt t0, t1, label	If $t0 < t1$ jump to label
bge	bge t0, t1, label	If $t0 > t1$ jump to label
bltu	bltu t0, t1, label	If $t0 < t1$ jump to label
bgeu	bgeu t0, t1, label	If $t0 > t1$ jump to label

Question: Could we get by with just **slt** and **beq** and **bne**?

Question: Why do we have unsigned versions for **bgeu** and **bltu** but not the others?



Branch Instruction Design

Question: Could we get by with just **slt** and **beq** and **bne**?

```
bge rd, rs, label
```

```
slt t0, rd, rs
```

```
beq t0, zero, label
```

```
bgt rd, rs, label
```

```
slt t0, rs, rd
```

```
bne t0, zero, label
```

```
ble rd, rs, label
```

```
slt t0, rs, rd
```

```
beq t0, zero, label
```

```
blt $ rd, rs, label
```

```
slt t0, rd, rs
```

```
bne t0, zero, label
```

A: Not needed, but it will require 2 instructions and the use of a temporary register. Other processors like MIPS did not directly support these more flexible branch instructions, the RISC-V designers thought that it was worth it to implement these in a single instruction instead of 2.



Pseudoinstructions

- Pseudoinstructions do not correspond to real RISC-V instructions.
- The assembler translates pseudoinstructions to real instructions, usually requiring at least one or more instructions.
- Pseudoinstructions not only **make it easier to program**, it can also **add clarity** to the program, by making the intention of the programmer **more clear**.

See here for list: <https://github.com/jameslzhu/riscv-card/blob/master/riscv-card.pdf>

Examples, do you think it helps with programming clarity?

Use Case	Pseudo-Instruction	Real Instruction
Moving/Copying value from one register to another	<code>mv rd, rs</code>	<code>addi rd, rs, 0</code>
Calculating logical not	<code>not rd, rs</code>	<code>xori rd, rs, -1</code>
Call a function at a label	<code>jal label</code>	<code>jal rd, label</code>
Jump to a label directly	<code>j label</code>	<code>jal zero, label</code>
Load an immediate value directly into a register (example showing 2 instructions)	<code>li rd, const</code>	<code>addi rd, zero, zero</code> <code>addi rd, rd, const</code>

An extended RISC-V Instruction



The base RISC-V instruction set does not include a multiply (or divide) instruction. The “**M**” **extension** defines these instructions:

<code>mul Rd, Rs1, Rs2</code>	<code>#Multiply</code> <code>#Rd = Rs1 X Rs2 : bits 0 ... 31</code>
<code>mulh Rd, Rs1, Rs2</code>	<code>#Multiply high</code> <code>#Rd = Rs1 X Rs2 : bits 32 ... 63</code>

One complication with multiply is that multiplication of two 32-bit numbers results in a 64-bit value, but we only have 32-bit registers. This is why we have both the `mul`, and `mulh` instructions.

Note checking if multiplying two 32-bit numbers overflows into a 64 bit number is on the programmer. How do you think we do this?

SIDEBAR: The “M” extension also defines the `div`, and `rem` instructions to support integer division. This works like 'C' integer division and modulus.

Summary Part 3: Conditional Programming, Looping and Branching



- Binary number representation: signed numbers, unsigned numbers, 2's complement and sign extension
- How we support flow control in RV programs to implement common abstractions like branching (if/then and if/then/else statements) and loops
- The robust set of branching instructions supported by RV – **beq**, **bnr**, **blt**, **bge**, **bltu**, **bgeu** to support loops and branches
- Pseudo instructions, we have seen them in Part 2 but now we know what they are called and why they help with program readability.
- Implications associated with the space in the **b** and **j** instruction formats for the immediate value (they are not 32 bits) and how we use the principal of locality (using **pc +/- immediate value**) to solve for many common use cases.



Programming with the RISC-V Instruction Set

Part 4: Functions / Procedures / Subroutines, the role of the OS and linkers for libraries

Dr. Brian Mitchell

Design Principle Recap



- Simplicity favors regularity
 - All instructions 32 bits
 - All instructions have 1,2, or 3 operands (but no more)
- Smaller is faster
 - Only 32 registers
- Good design demands good compromises
 - All instructions are the same length
 - Limited number of instruction formats: R, I, J
 - Opcode constraints (7 bits) complemented by pseudo-instructions to provide program readability without defining new instructions
- Make common cases fast
 - Encode immediate constants in instructions even though they cannot cover the entire 32 bit address space “principal of locality”

The Anatomy of a Procedure / Function



You have created functions in other programming languages but probably have not given thought to what is really going on, **since the concept of a function does not exist at the processor level.**

```
int global_var = 0;

int main(){
    global_var = example_fun(1,2,3,4);
}

int example_fun (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}

int another_fun(int i){
    return i+1;
}
```

There is a lot going on here:

1. The storage strategy to hold the `int global_var` and the local variable `int f` are different.
2. We need to store the constants `1,2,3,4` that are passed as arguments to `example_fun()` somehow.
3. How do we manage passing function parameters and handle the return value?
4. Since functions do not exist, how do we simulate calling them and safely returning?
5. Its likely not obvious to you at this point, but the functions `example_fun()` and `another_fun()` need to be treated differently.



Lets start by reviewing the RV32I registers

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5–7	t0–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

We have seen these before, recall that they have real names and aliases (ABI Names) – we can use either format in our code.

The only purpose of the ABI names is to make our code easier to read.

Other than the read-only **x1** or **zero** register, we can use all of the other registers **for anything that we want**

Given everything going on with supporting functions, we establish **conventions** using ABI register names to make our code easier to read and maintain.

At a conceptual level functions are not that complex



In the execution of a procedure, the program must follow these steps:

- Place parameters in a place where the procedure can access them
- Transfer control to the procedure
- Acquire the storage resources needed for the procedure
- Perform the desired task
- Place the result where the calling program can access it
- Return control to the point of origin

Now we will dig into how we accomplish this in a consistent way with readable and maintainable code.



Conventions and Instructions used for simulating making a Procedure/Function Call

- **a0–a7**: provides eight argument registers used to pass parameters –
Q: What if you need to pass more than 8?
- **a0–a7**: provides eight registers to return values parameters –
Q: But wait, C, only allows you to return one value, why 8?
- **sp**: provides one register to manage the stack used to store local variables –
Q: What is a local variable?
- **ra**: provides one register to manage the return address–
Q: Why does this need to be carefully managed?
- **jal label**: provides the instruction to simulate calling a procedure, sets the PC to the address of the label, and saves PC+4 in register **ra**
Q: What side-effect does this instruction cause that requires careful attention?
- **jr ra**: provides the instruction to return from the call to the next instruction after the procedure call
Q: What if any house keeping needs to be done before making this call?

We will revisit shortly and answer the questions later



Conventions and Instructions used for simulating making a Procedure/Function Call

Argument Passing and Return Values Use the Same Registers – Be careful!

a0–a7: provides eight argument registers used to pass parameters

a0–a7: provides eight registers to return values parameters

```
main:
    li a0, 1
    jal add_one
    jal add_two
    #more instructions
    ret
add_one:
    addi a0, a0, 1
    jr ra
add_two:
    addi a0, a0, 2
    jr ra
```

versus

```
main:
    li s0, 1
    mv a0, s0
    jal add_one
    mv a0, s0
    jal add_two
    #more instructions
    ret
add_one:
    addi a0, a0, 1
    jr ra
add_two:
    addi a0, a0, 2
    jr ra
```

A nasty side effect / bug that can happen is if you forget that the value of an a register can change after a jal call – you must **save** the value yourself if you want to preserve it!



Conventions and Instructions used for simulating making a Procedure/Function Call

Using the stack and **sp** register for local variables and temporary storage

```
int main()
{
    int f;
    f = add1(5);
    f = add2(f);
}
```

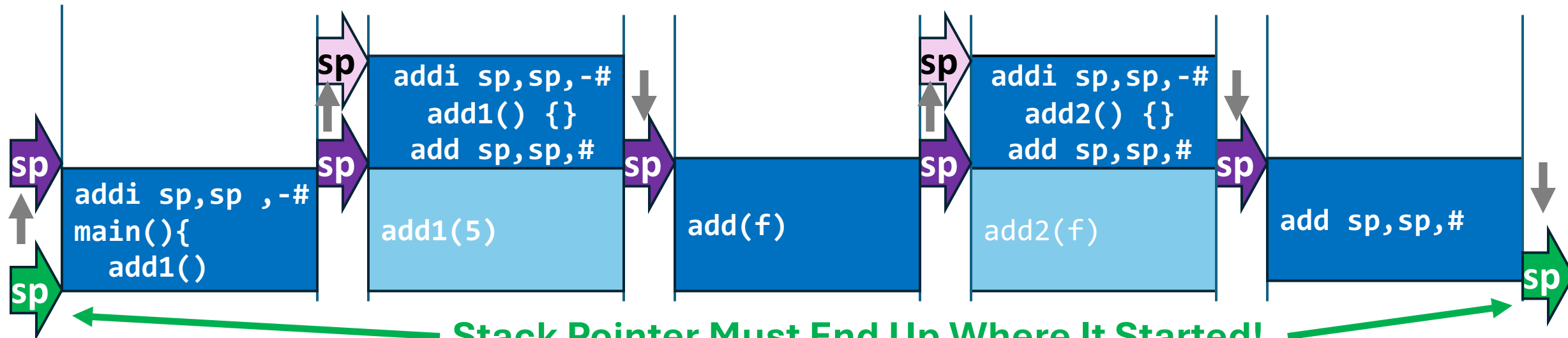
```
int add1(int x){
    int y;
    y = x + 1;
    return y;
}
```

```
int add2(int x){
    int y;
    y = x + 1;
    return y;
}
```

The initial value of **sp** is initialized by the runtime

Push onto the stack to **allocate** space:
addi sp,sp,-#

Pop off the stack to **free** space:
addi sp,sp,#



Stack Pointer Must End Up Where It Started!



Conventions and Instructions used for simulating making a Procedure/Function Call

Using the stack and **sp** register for local variables and temporary storage

Memory Address	Memory Contents
0x00000000 to 0x0000FFFF	.text segment Program Instructions
0x00010000 to 0x0001FFFF	.data (constant) segment Read only constant values and variables
0x00020000 to 0x0002FFFF	.data segment Read/write global variables
0x00030000 to 0xFFFFFFF0	DYNAMIC MEMORY HEAP (aka, malloc) ↓ X ↑ STACK (aka, functions)

1. The stack pointer register is initialized by the runtime
2. starts at a high memory address and grows upward (aka we subtract from **sp** to allocate)
3. As we have also seen we subtract from **sp** to free space
4. We reach an out of memory condition if the stack and heap grow to the point where they overlap

The programmer is responsible for managing exactly how much stack space they need to consume on each function call – each word saved requires 4 bytes



Conventions and Instructions used for simulating making a Procedure/Function Call

Leaf vs non-leaf functions require careful handling of the `ra` register.

`ra`: by convention is used to save the return address from a function call

- When we program and use functions, the overall shape of program **execution is a tree**, as functions can call 1 or more other functions.
- Any function that calls at least one other function is called a **non-leaf function**
- Any function that does **not call any other functions** is called a **leaf-function**

The `ra` register is used to track where execution returns after a function call (we call `jr ra` to return). **Non-leaf functions must save `ra`**, and typically `ra` is saved on the stack.

```
main:
    li a0, 1
    jal add_one
    jal add_two
    #more instructions
    ret
add_one:
    addi a0, a0, 1
    jal add2
    jr ra
add_two:
    addi a0, a0, 2
    jr ra
```

non-leaf function

non-leaf function

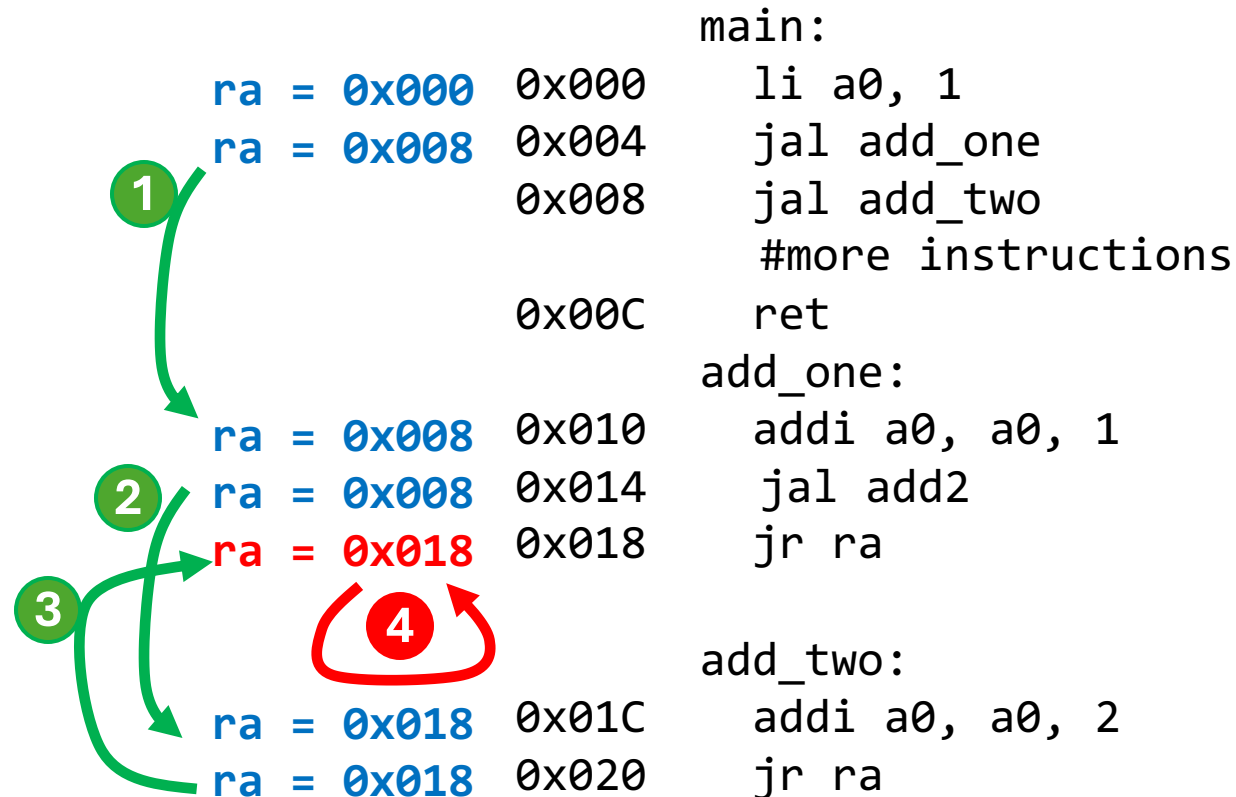
leaf function



Conventions and Instructions used for simulating making a Procedure/Function Call

Leaf vs non-leaf functions require careful handling of the **ra** register.

jal label: requires careful attention, because it mutates the ra register. If its not saved prior to this instruction the current function will not be able to return.



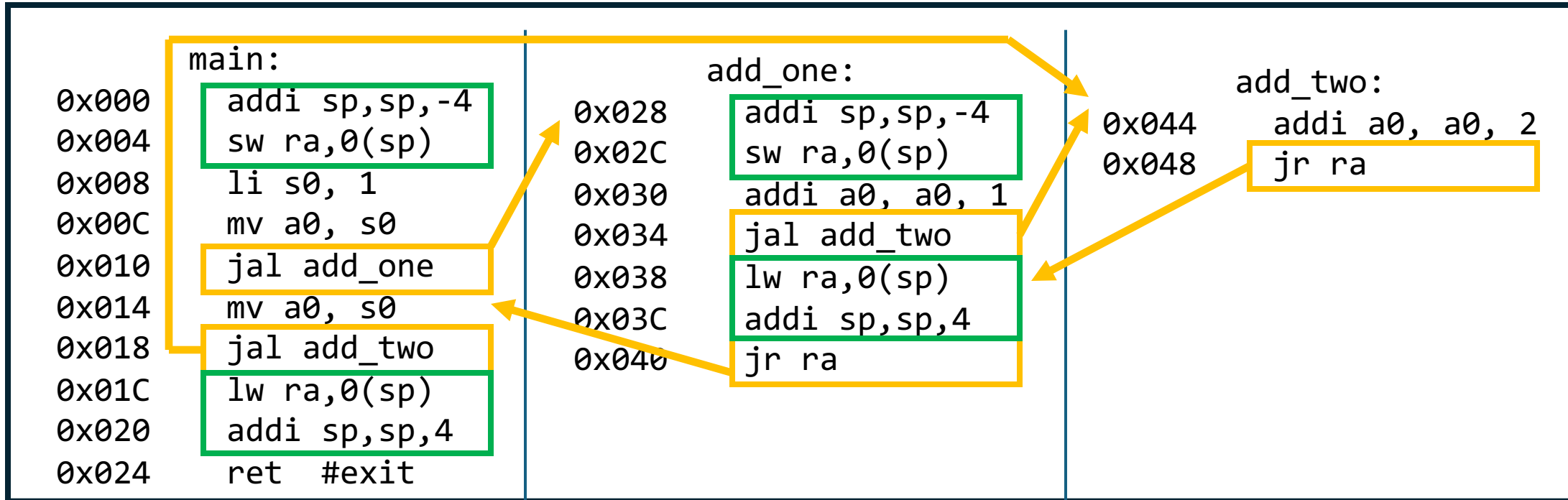
1. The call to **add_one** changes **ra** from 0x000 to 0x008
2. The call from **add_one** to **add_two** changes **ra** from 0x008 to 0x018
3. The call to **jr ra** from **add_two** changes the **pc** from 0x020 to the value of **ra** which is 0x018
4. The call to **jr ra** from **add_one** changes the **pc** from 0x018 to the value of **ra** which is 0x018.

Result: INFINITE LOOP



Conventions and Instructions used for simulating making a Procedure/Function Call

Properly using the stack pointer **sp** register to manage the **ra** register.



```
@ 0x000 ra      = 0x000
@ 0x004 0(sp)   = 0x000
@ 0x010 ra      = 0x014
@ 0x018 ra      = 0x01C
@ 0x01C ra      = 0x000
@ 0x024 EXIT / DONE
```

```
@ 0x028 ra      = 0x014
@ 0x02C 0(sp)   = 0x014
@ 0x034 ra      = 0x038
@ 0x038 ra      = 0x014
@ 0x040 pc=ra   = 0x014
```

```
@ 0x044 ra      = 0x038
@ 0x048 pc=ra   = 0x038 } 1st call from add_one

@ 0x044 ra      = 0x01C
@ 0x048 pc=ra   = 0x01C } 2nd call from main
```

Saving Registers using a Stack



- Additional registers used by the called procedure **must be saved** prior to use, or the values used by the calling procedure **will be corrupted**.
- The old values can be saved on a stack (call stack). After the called procedure completes, the old values can be popped off the stack and restored.
- **sp**: stack pointer register contains the address of the top of the stack. By convention, address on the stack grows from higher addresses to lower address, which implies that a push subtracts from **sp** and a pop adds to **sp**.
- When we obtain stack space by subtracting from **sp**, we are creating what is called a **stack frame**



Conventions and Instructions used for simulating making a Procedure/Function Call

- **a0–a7**: provides eight argument registers used to pass parameters –
Q: What if you need to pass more than 8?
- **a0–a7**: provides eight registers to return values parameters –
Q: But wait, C, only allows you to return one value, why 8?
- **sp**: provides one register to manage the stack used to store local variables –
Q: What is a local variable?
- **ra**: provides one register to manage the return address–
Q: Why does this need to be carefully managed?
- **jal label**: provides the instruction to simulate calling a procedure, sets the PC to the address of the label, and saves PC+4 in register **ra**
Q: What side-effect does this instruction cause that requires careful attention?
- **jr ra**: provides the instruction to return from the call to the next instruction after the procedure call
Q: What if any house keeping needs to be done before making this call?

Review: Lets answer these questions!



Other considerations for functions

Consider the following code:

main:

```
li a0, 1
li t0, 2
li s0, 3
```

jal call_function

```
# at this point what
# can we say about
# the value of the
# a0,t0,s0 registers?
```

```
#next instruction, eg:
```

```
addi t1, t0, 5
```

```
# can we assume here
# that t1 = 7 because
# isn't t0=2?
```

- We know RISC-V provides 32 registers **x0-x31**
- Out of the total 32 registers, we only have 29 to work with – the **zero** register is read only and we use the **ra** and **sp** registers for managing function calls and the stack
- **All functions share the same set of registers**

We need to establish register usage conventions to ensure programs are easier to write and maintain

Without conventions we must assume any register with values we care about could have been changed by the body of a function call



Register usage convention rules

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

The role of who is the “caller” and who is the “callee” is relative to where the function call is being made:

```
Caller  int main(){
        int j = call_fn();
    }

Callee int call_fn(){
        return 5;
    }
```

1. Before making a function call, the **caller** is responsible for saving any values (likely on the stack) that they want preserved from registers **t0–t2**, **a0–a8**, and **ra**. The caller can assume that the **s0–s11**, and **sp** registers will not be modified
2. Before mutating any of the **s0–s11** or **sp** registers, the **callee** is responsible for saving any values (likely on the stack) and then restoring them prior to returning from the function call.

How do these rules/conventions simplify programming and reduce bugs?

Register Convention Example



```
.data
v: .word 1,2,3,4,5,0

.text
.globl main

main:
    la s0, v      #a0 = &v
    mv a0, s0
    li a1, 3      #a1 = k = 3
    jal swap

    # What can we say about a0,a1
    # and s0 here?

    # more code likely using &v
    # that is in s0
```

```
#swap(v,k)
swap:
    addi sp, sp -4  #setup stack frame
    sw s0, 0(sp)    #save s0 so it can be recovered

    slli a1,a1,2     #a1 = 4*a1
    add s0,a0,a1     #a0 = &v[k] = &v + 4*k
    lw t1,0(s0)      #t1 = v[k]
    lw t2,4(s0)      #t2 = v[k+1]
    sw t1,4(s0)      #v[k] = v[k+1]
    sw t2,0(s0)      #v[k+1] = temp
    li a0, 1         # return 1 to indicate success

    lw s0, 0(sp)     #restore s0 before returning
    addi sp, sp, 4   #restore stack pointer

    jr ra            #return
```

Register Convention Example



```
.data
```

```
v: .word 1,2,3,4,5,0
```

```
.text
```

```
.globl main
```

```
main:
```

```
la s0, v    #a0 = &v
```

```
mv a0, s0    #arg setup a0=&v
```

```
li a1, 3     #a1 = k = 3
```

```
jal swap
```

```
# print result
```

```
mv a1, a0    #put result in a1
```

```
li a0, 1     #print integer
```

```
ecall        #system call
```

```
ret          #end program
```

```
#swap(v,k)
```

```
swap:
```

```
addi sp, sp, -4    #setup stack frame
```

```
sw s0, 0(sp)       #save s0 so it can be recovered
```

```
slli a1, a1, 2     #a1 = 4*a1
```

```
add s0, a0, a1     #a0 = &v[k] = &v + 4*k
```

```
lw t1, 0(s0)       #t1 = v[k]
```

```
lw t2, 4(s0)       #t2 = v[k+1]
```

```
sw t1, 4(s0)       #v[k] = v[k+1]
```

```
sw t2, 0(s0)       #v[k+1] = temp
```

```
li a0, 1           # return 1 to indicate success
```

```
lw s0, 0(sp)       #restore s0 before returning
```

```
addi sp, sp, 4     #restore stack pointer
```

```
jr ra             #return
```



Managing function arguments as temporary variables in **a** registers



Preserving land using **long lived** variables in **s** registers



Using **short lived intermediate** values in **t** registers



Using the stack to manage the **sp** register to ensure **s** register values are preserved

Character Data



Up until this point we have only been working with word-sized (32 bit) data values, sometimes you want to work with character data (8 bits) and half-word values (16 bits).

- Byte-encoded character sets
 - ASCII: 128 characters
 - 95 graphic, 33 control
 - Latin-1: 256 characters
 - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
 - Used in Java, C++ wide characters, ...
 - Most of the world's alphabets, plus symbols
 - UTF-8, UTF-16: variable-length encodings



Byte/Half-word Operations

- RISC-V byte (8 bits) /halfword (16 bits) load/store
- String processing is a common case

```
lb Rd, offset(Rs1)
lbu Rd, offset(Rs1)
sb Rs2,offset(Rs1)
```

```
lh Rd, offset(Rs1)
lhu Rd, offset(Rs1)
sh Rs2,offset(Rs1)
```

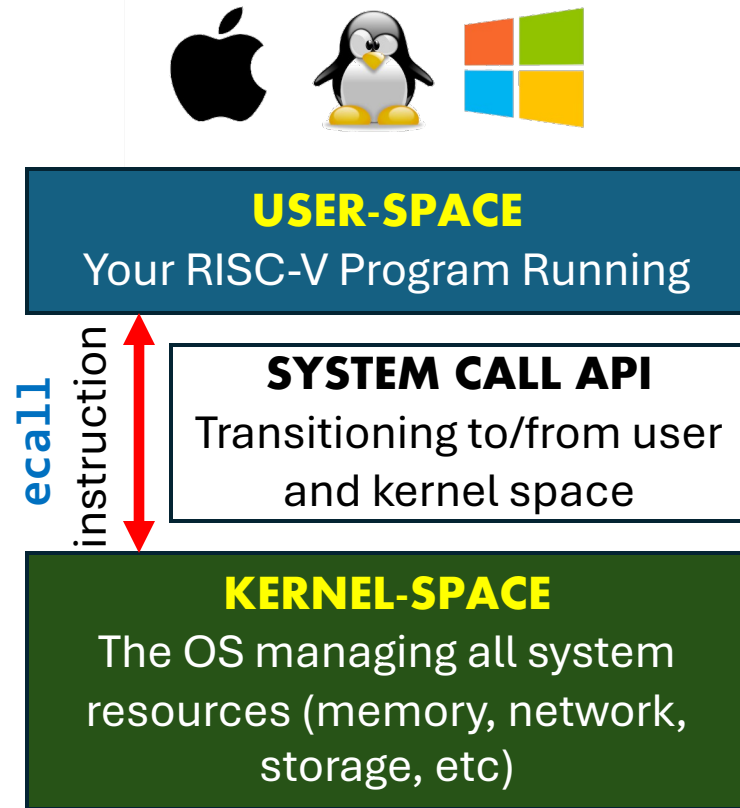
lb fetches the 32 bit value at **offset(Rs1)** and grabs bits [0:7] – it then sign extends using bit 7
lh fetches the 32 bit value at **offset(Rs1)** and grabs bits [0:15] – it then sign extends using bit 15
lbu and **lhu** work exactly the same as **lb** and **lh** but extend zeros from bits 8 and 16 respectively
sb stores bits [0:7] in **Rs2** at the address **offset(Rs1)**
sh stores bits [0:15] in **Rs2** at the address **offset(Rs1)**

This is how different data types in C are handled – char, short, int, and long. Note for long variables (64-bit) you will need to use 2 registers.

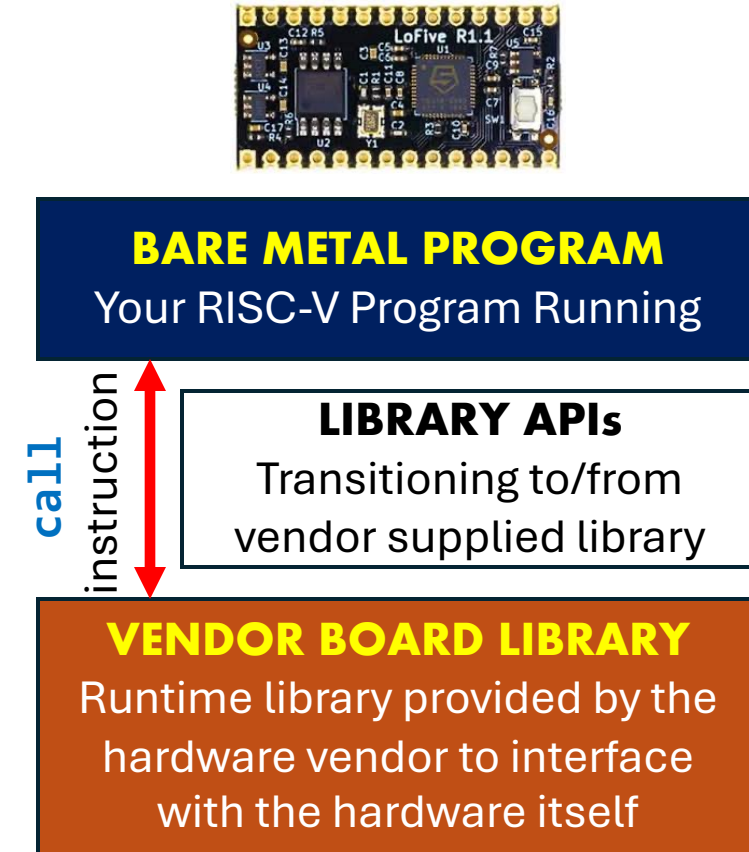


Final Thoughts: The “Runtime” Environment

All of your programming experience has probably been in an environment where you are using a traditional operating system such as Windows, MacOS, or Linux.



The OS provides a standard library to transition between user and kernel space, e.g., libc in linux. Consider what happens when you call printf()



The manufacturer of the board you are using provides technical documentation and libraries to interface with the board’s capabilities, e.g, make a wifi connection.



Example: RISC-V Microcontroller



DATA SHEET

https://sifive.cdn.prismic.io/sifive%2F3d777659-a0dd-49ed-a011-5bebbba17aecf_fe310-g002-ds.pdf

DEVELOPER LIBRARIES

<https://github.com/mwelling/freedom-e-sdk.git>

EXAMPLE

<https://github.com/sifive/example-sifive-welcome/blob/52434ebc25f427d5058aab82d3af3431b0d953e0/sifive-welcome.c>

New Instruction – **call** vs **ecall**

The OS provides a standard library to transition between user and kernel space, e.g., libc in linux. Consider what happens when you call printf()

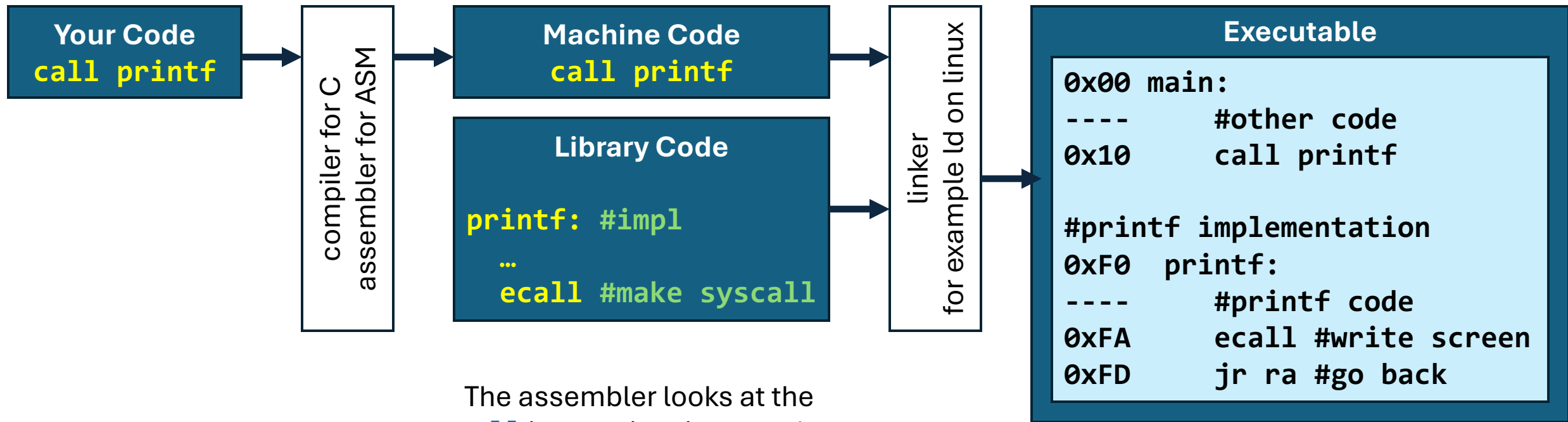
The manufacturer of the board you are using provides technical documentation and libraries to interface with the board's capabilities, e.g, make a wifi connection.

Working with libraries, versus making direct system calls



New Instruction – `call` vs `ecall`

The `call` instruction is used to call library code that will be linked into the final executable. The `ecall` instruction is used to make a system call to the operating system



The assembler looks at the `call` instruction, ignores the label, deferring to the linker to resolve

Example using Compiler Explorer

A screenshot of the Compiler Explorer web application. The browser address bar shows "godbolt.org". The interface has a top navigation bar with "COMPILER EXPLORER" and various tool logos (Intel, Solid Sands, JetBrains). Below this, there are tabs for "C++ source #1" and "RISC-V rv32gc clang (trunk) (Editor #1)". The left pane shows C++ code:

```
1 // Type your code here, or load an example.
2 #include <stdio.h>
3
4 int main(){
5     printf("hello cs281 \n");
6 }
7
8
9
10
11
```

 A red rectangle highlights the C++ code, and a red arrow points from it to the assembly code in the right pane. The right pane shows the generated RISC-V assembly:

```
1 main:
2     addi    sp, sp, -16
3     sw      ra, 12(sp)
4     sw      s0, 8(sp)
5     addi    s0, sp, 16
6     lui     a0, %hi(.L.str)
7     addi    a0, a0, %lo(.L.str)
8     call    printf
9     li      a0, 0
10    lw      ra, 12(sp)
11    lw      s0, 8(sp)
12    addi    sp, sp, 16
13    ret
14
15 .L.str:
16     .asciz  "hello cs281 \n"
```

Summary Part 4: Functions/Procedures/Subroutines, the role of the OS and linkers for libraries



- How to simulate functions via to setup, execute, and return.
- Defining function types: leaf- and non-leaf
- RV register conventions – the role of the callee and caller – and how it promotes programming efficiency and readability
- Properly managing flow control for functions via using jump instructions in conjunction with the **ra** register
- How the **a** registers are used for passing arguments and returning values
- What is a stack frame, why we need it, and how its allocated and freed using the **sp** register.
- Traditional (on the operating system) vs bare metal (no operating system) programming – interacting with the OS and/or libraries via the **ecall** and **call** instructions.



Done With:

Programming with the RISC-V Instruction Set

Next: RV Hardware

Dr. Brian Mitchell

Part 2 of course: Moving onto hardware



We have only scratched the surface on the robust RV assembly programming environment, but you should know have the knowledge to continue to learn additional features on your own using the open source resources provided by the RV designers

We are now transitioning onto how the hardware of the RV works in the second part of the course. We will be covering:

- Review of digital logic
- The core of the processor – the ALU and how its designed
- The remainder of the processor, how instructions are decoded and executed, how the ALU interfaces with registers, and how load/store operations are handled interfacing with memory/storage.
- The purpose of pipelining
- How pipelining improves the performance of processors, and some of the issues it can cause
- How some of the pipelining issues are handled/solved via using techniques such as forwarding and branch prediction