CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 8: RISC-V Instruction Format

Instructors: Rosalie Fang, **Charles Hong**, Jero Wang

Announcements - Logistics

- Labs back in Soda 271
- Topic-specific exam prep sections begin.
    - RISC-V: Thursday and Friday this week, 5-7pm in Cory 540AB (same content in both sections)
- Midterm next Friday 7/14: Logistics coming soon.

Announcements - Assignments

- Labs 3 and 4 released, due Tuesday, July 11th, 11:59 PM PT
    - Tomorrow's lab sections will cover lab 3, and next Tuesday's lab sections will cover lab 4
- Homework 2 due today (Wednesday), July 5th, 11:59 PM PT
- Homework 3 will be released this week and is due Wednesday, July 12th, 11:59 PM PT
    - Since this is a long homework, it will be worth twice as much as a normal homework (6 course points as opposed to 3).
- Project 2 was released and part A is due Friday, July 7th, 11:59 PM PT

Last Time

- Translating C concepts into RISC-V
  - Variables
  - Arrays
  - If-Else
  - Loops
  - Functions
- Calling Convention

Agenda

- Lecture 6 (Last Wednesday)
  - Intro to Assembly Languages
  - RISC-V Registers
  - RISC-V Instructions and Instruction Types
- Lecture 7 (Last Thursday)
  - Translating C concepts into RISC-V
    - Variables
    - Arrays
    - If-Else
    - Loops
    - Functions
  - Calling Convention
- Lecture 8 (Today)
  - Calling Convention (cont.)
  - Translating RISC-V to Binary

Calling Convention (cont.)

Calling Convention

- Saved registers:
  - **s0-s11**, **ra**
  - Should not be modified by functions (can be used, but must be restored).
- Temporary registers:
  - **t0-t6**, **a0-a7**
  - Can be modified by functions.
  - Careful—might be modified by functions you call

Stack

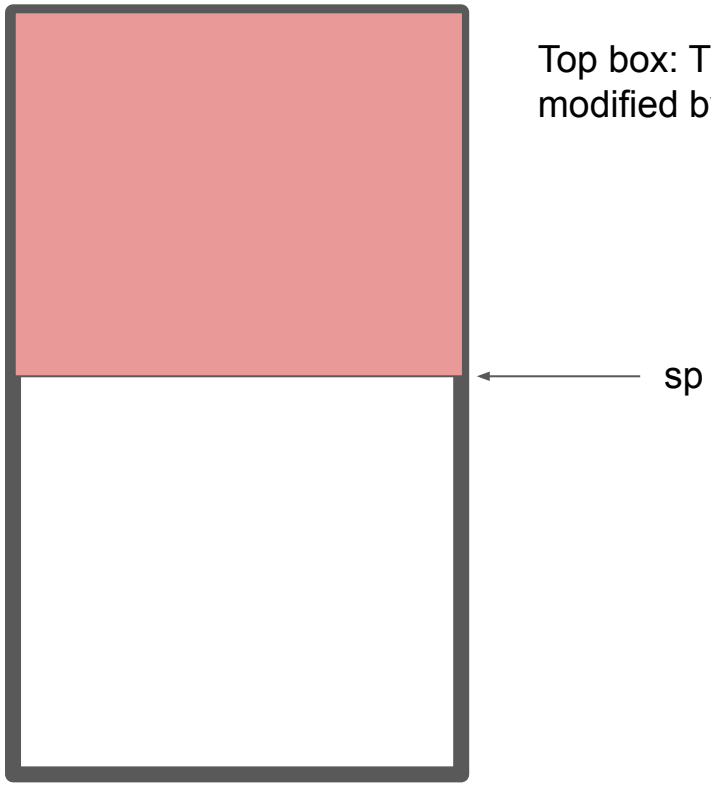
- A function should restore the stack before returning. To be precise:
  - Data **above** sp at function start is considered **immutable**, and must not be modified by the function.
  - Data **below** the sp at function start is **mutable**; the function may modify anything in that area.
  - The **value of sp** must be **restored** by the end of the function. Usually done by subtracting, then later adding the same amount from the sp (ex. `addi sp sp -8` at the beginning of a function, `addi sp sp 8` at the end)

Stack Example

PROLOGUE:
1: The function is called
2: Move the stack pointer down
3: Save any data we need to

BODY:
4: Run the function (which may call further functions)

EPILOGUE
5: Restore any registers we need to
6: Move the stack pointer back up
7: Return from the function



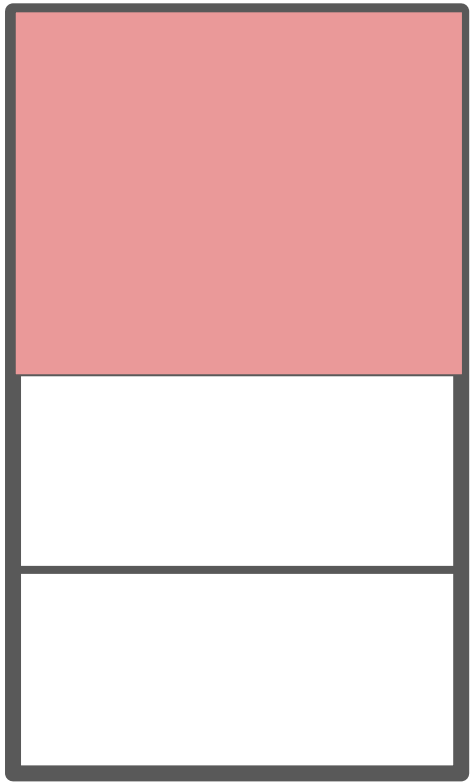
Top box: This memory cannot be modified by our function

Stack Example

PROLOGUE:
1: The function is called
2: Move the stack pointer down
3: Save any data we need to

BODY:
4: Run the function (which may call further functions)

EPILOGUE
5: Restore any registers we need to
6: Move the stack pointer back up
7: Return from the function



Top box: This memory cannot be modified by our function

This space is now useable by our function

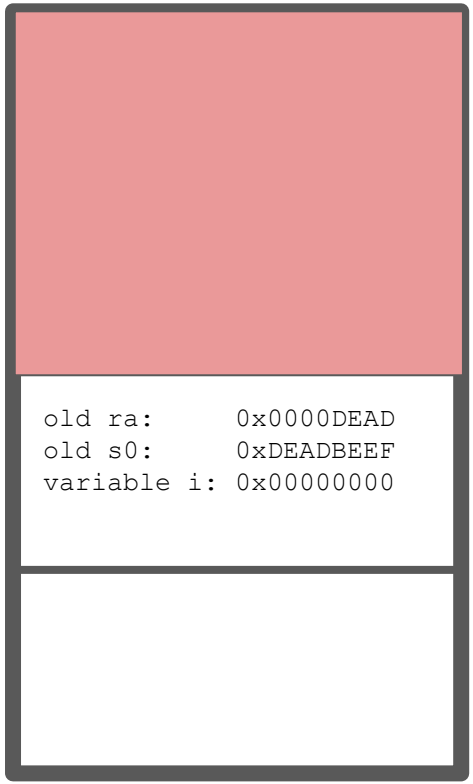
sp

Stack Example

PROLOGUE:
1: The function is called
2: Move the stack pointer down
3: Save any data we need to

BODY:
4: Run the function (which may call further functions)

EPILOGUE
5: Restore any registers we need to
6: Move the stack pointer back up
7: Return from the function



Top box: This memory cannot be modified by our function

This space is now useable by our function

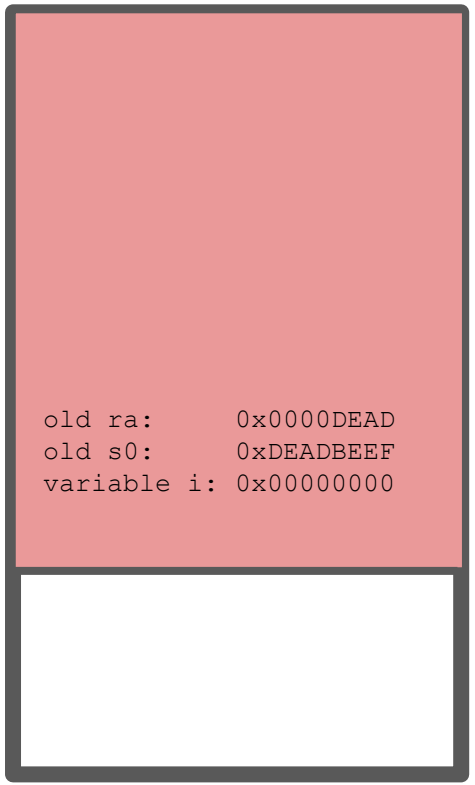
sp

Stack Example

PROLOGUE:
1: The function is called
2: Move the stack pointer down
3: Save any data we need to

BODY:
4: Run the function (which may call further functions)

EPILOGUE
5: Restore any registers we need to
6: Move the stack pointer back up
7: Return from the function



Top box: This memory cannot be modified by our function

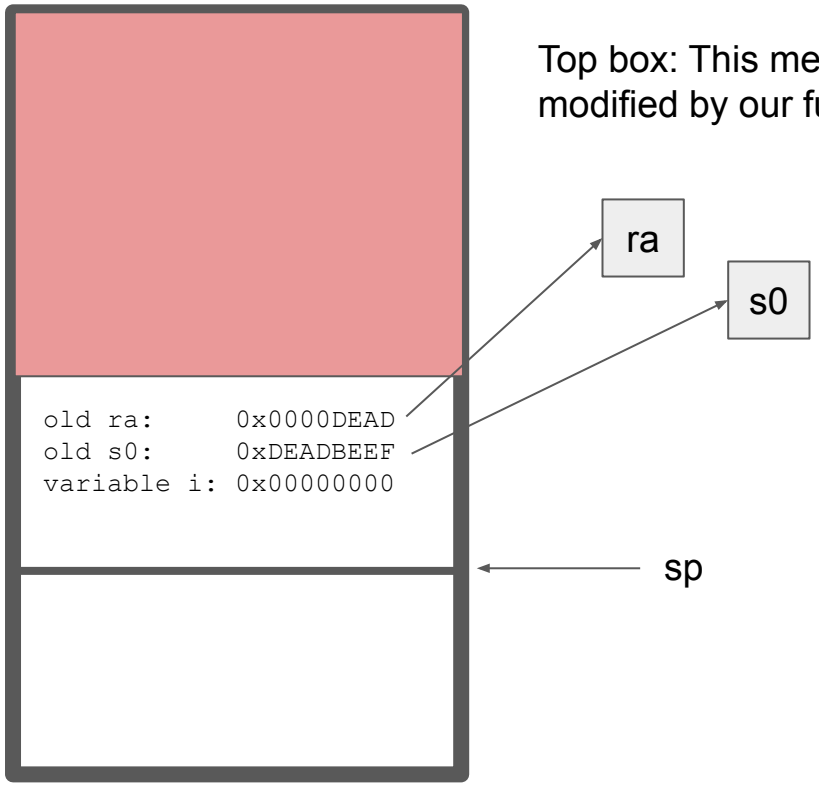
If we call another function then that function is not allowed to modify this block of memory

Stack Example

PROLOGUE:
1: The function is called
2: Move the stack pointer down
3: Save any data we need to

BODY:
4: Run the function (which may call further functions)

EPILOGUE
5: Restore any registers we need to
6: Move the stack pointer back up
7: Return from the function



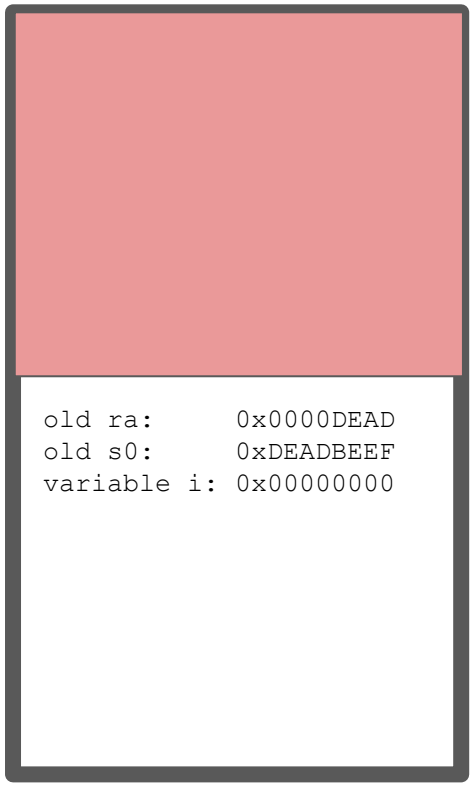
Top box: This memory cannot be modified by our function

Stack Example

PROLOGUE:
1: The function is called
2: Move the stack pointer down
3: Save any data we need to

BODY:
4: Run the function (which may call further functions)

EPILOGUE
5: Restore any registers we need to
6: Move the stack pointer back up
7: Return from the function



Top box: This memory cannot be modified by our function

sp

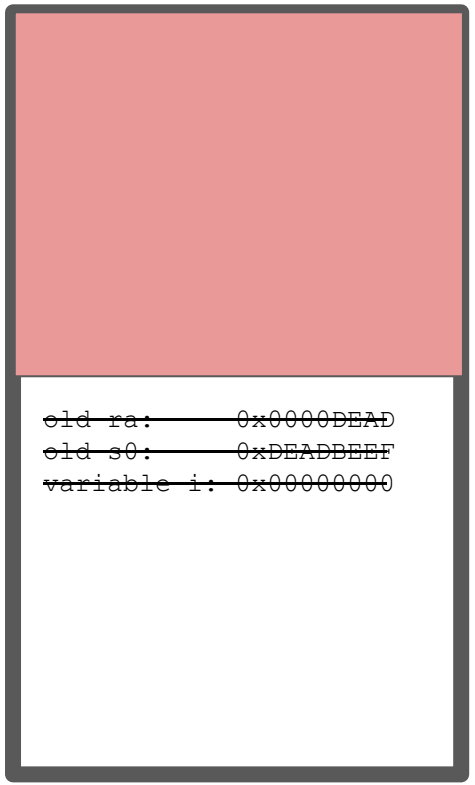
Move sp back up.

Stack Example

PROLOGUE:
1: The function is called
2: Move the stack pointer down
3: Save any data we need to

BODY:
4: Run the function (which may call further functions)

EPILOGUE
5: Restore any registers we need to
6: Move the stack pointer back up
7: Return from the function



Top box: This memory cannot be modified by our function

Stack is restored to the same state it was in before the function was called.

Recursive functions: C code

```
int fact(int x) {
    if(x == 0) return 1;
    int y = fact(x-1);
    return x * y;
}
int result = fact(7);

printf("%d\n", result); //Should output 5040
```

Recursive functions: RISC-V Starter

```
addi a0 x0 7
jal fact

#Print out the value stored in a0 as an integer, then exit.
#This should output 5040
mv a1 a0
li a0 1
ecall
li a0 10
ecall
#End print and exit

fact: #TODO
```

Recursive functions: Solution

```
fact:
#x: s0
#y: a0

beq a0 x0 TailCase      # If in the tail case, handle separately
addi sp sp -8           # Begin Prologue: set aside space for 8 bytes of data
sw s0 0(sp)             # Store s0 (since we plan on using it)
sw ra 4(sp)             # Store ra (since we plan on calling another function)

mv s0 a0                # x = input (Save in s register so we keep it through the recursive call)
addi a0 a0 -1           # Set argument to x-1
jal fact                # Recursively call fact. Expect return value in y=a0
mul a0 a0 s0            # Output = x * y

lw s0 0(sp)             # Begin Epilogue: Restore old value of s0
lw ra 4(sp)             # Restore old value of ra
addi sp sp 8            # Restore stack pointer to original value
jr ra                   # Return

TailCase:
addi a0 x0 1            # In tail case, set a0 to 1, and return
jr ra
```

Recursive functions: Analysis

- As long as you follow calling convention (abbreviated CC), it is possible to make any C function.
- If you fail to follow CC, then that leads to extremely difficult bugs, because the underlying assumptions we make about how function calls work are now broken
- If your function has multiple return commands (such as through branching code), make sure that the stack gets restored regardless of which branch gets taken.
    - An easy way to ensure this is to have an epilogue label that you always jump to – see examples of this in discussion!

RISC-V Instruction Format - Agenda

- **Intro**
- R-types
- I-types
- S-types
- U-types
- B-types
- J-types
- Concluding Notes

Great Idea #1: Abstraction (Layers of Representation/Interpretation)

CS61C

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

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

Compiler

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

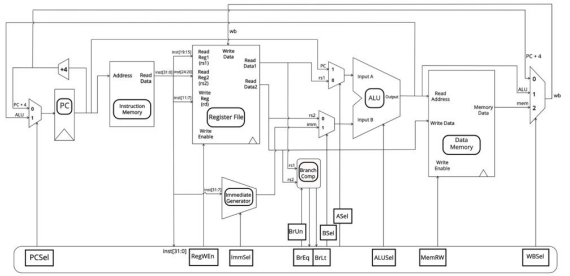
```
lw   x3, 0(x10)
lw   x4, 4(x10)
sw   x4, 0(x10)
sw   x3, 4(x10)
```

Assembler

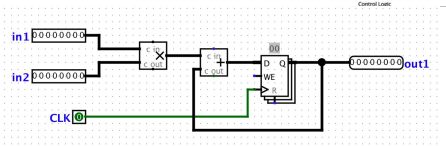
Machine Language Program (RISC-V)

```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

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



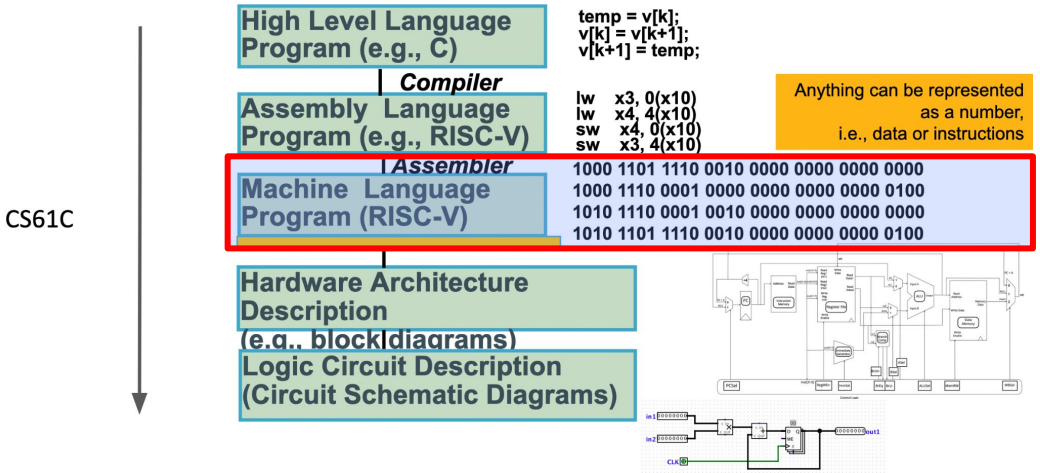
Logic Circuit Description (Circuit Schematic Diagrams)



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

Great Idea #1: Abstraction (Layers of Representation/Interpretation)

- Assembly languages should be able to be directly translated into binary code that can be run by a CPU.
- Processed by **assembler** into **machine language**



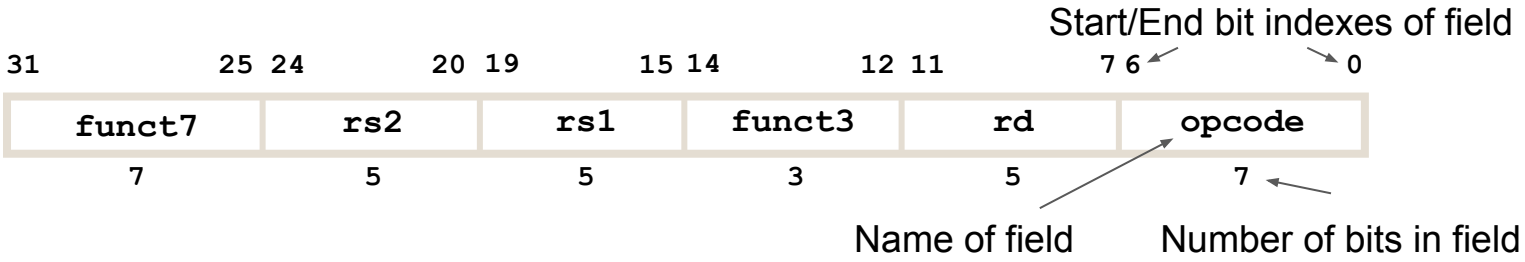
RISC-V Instruction Format: Intro

- RISC-V is particularly simple: Each instruction is translated into 32 bits (4 bytes)
  - For RV32 (the version in this class)
- Different instructions require different values
  - "add" specifies 3 register inputs
  - "addi" specifies 2 registers and 1 immediate
  - Idea: Encode different instructions, differently
- Overall design philosophy: Categorize similar instructions into a groups. For each group, define how the 32 bits are used to hold all needed values
  - Called **instruction formats**
  - Try to make different formats similar, where possible!
- Formats are on reference card—don’t need to memorize

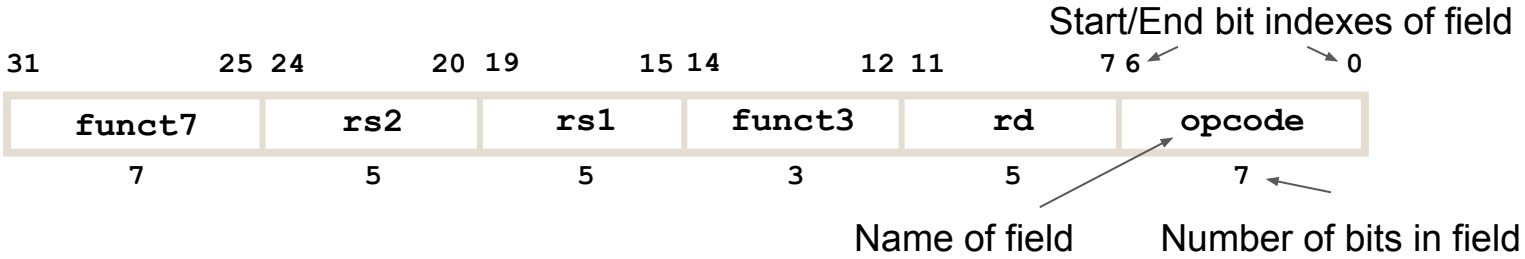
RISC-V Instruction Format - Agenda

- Intro
- **R-types**
- I-types
- S-types
- U-types
- B-types
- J-types
- Concluding Notes

R-Type

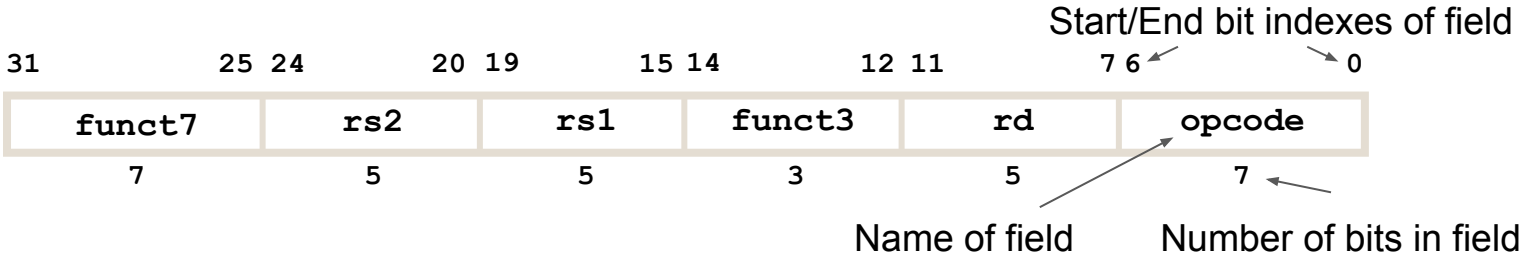

Start/End bit indexes of field
31
25 24
20 19
15 14
12 11
7 6
0
funct7
rs2
rs1
funct3
rd
opcode
7
5
5
3
5
7
Name of field
Number of bits in field

R-Type



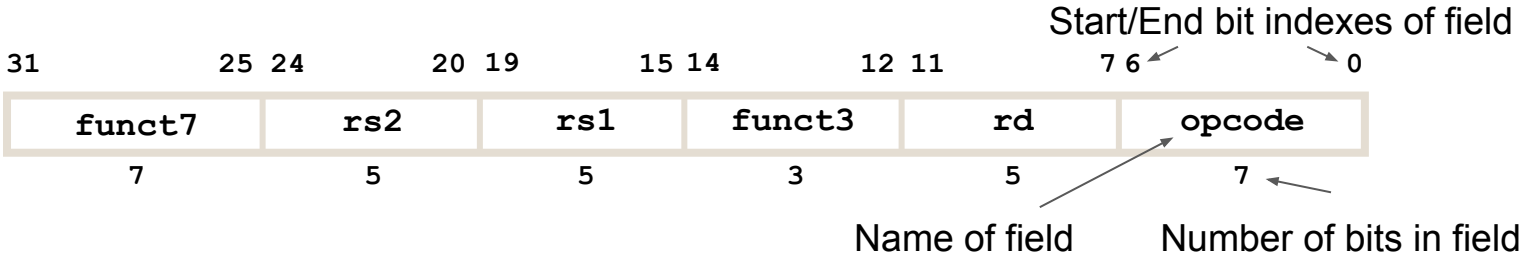
- Designed for instructions with 3 registers and no immediate
  - Arithmetic operators like add or sub
- Each register is identified by its number. 32 registers ⟶ 5 bits to identify one register uniquely
  - `x0` ⟶ `0b00000`
  - `a0` ⟶ `x10` ⟶ `0b01010`
- rd: Destination register
- rs1: 1st source register
- rs2: 2nd source register

R-Type



- opcode: Instruction identifier, the last 7 bits of the instruction in all instruction formats
- Some sets of similar instructions get assigned the same opcode
  - Ex. All arithmetic R-type instructions have the opcode 0x33
  - Instructions with the same format tend to have similar opcodes
- funct3: 3-bit identifier to differentiate instructions with the same opcode
- funct7: Extra 7-bit identifier for extremely similar instructions with the same opcode and funct3 (such as `sra` and `srl`)

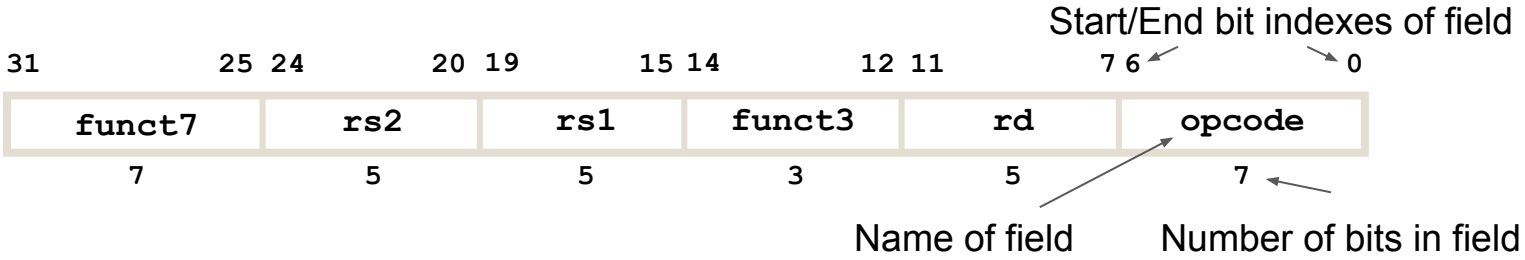
R-Type: Instruction to Hex Translation (1/5)



Translate "add s2 s3 s4" to hex

- Step 1: Determine opcode and instruction type from reference card
  - `Type:`
  - `Opcode:`
  - `funct3:`
  - `funct7:`
- Step 2: Write out format
  - `0b ??????? ????? ????? ??? ????? ???????`

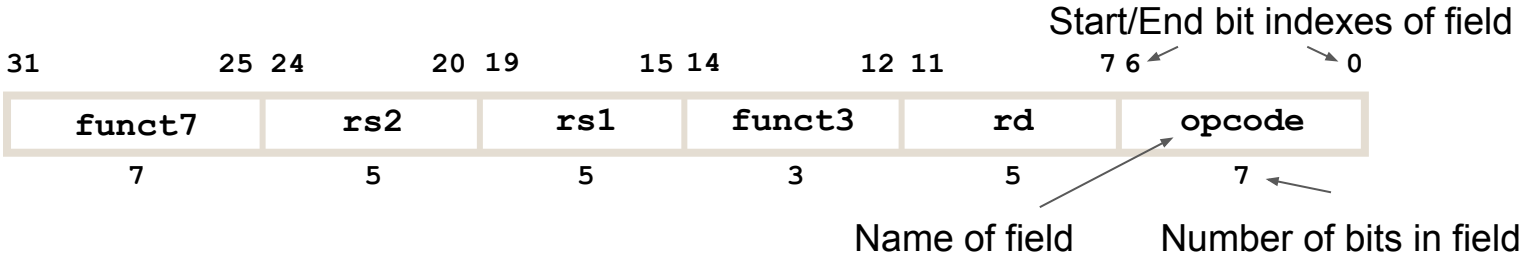
R-Type: Instruction to Hex Translation (2/5)



Translate "add s2 s3 s4" to hex

- Step 1: Determine opcode and instruction type from reference card
  - `Type: R`
  - `Opcode: 0b011 0011`
  - `funct3: 0b000`
  - `funct7: 0b000 0000`
- Step 2: Write out format
  - `0b ??????? ????? ????? ??? ????? ???????`

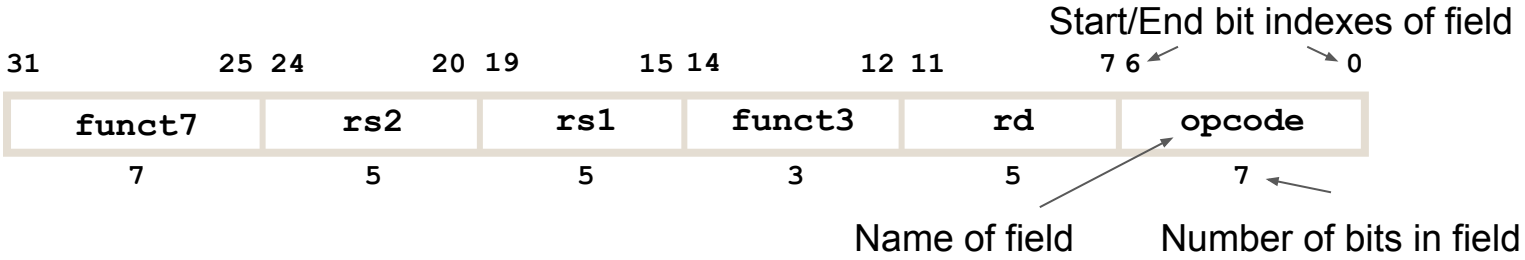
R-Type: Instruction to Hex Translation (3/5)



Translate "add s2 s3 s4" to hex

- Step 1: Determine opcode and instruction type from reference card
  - `Type: R`
  - `Opcode: 0b011 0011`
  - `funct3: 0b000`
  - `funct7: 0b000 0000`
- Step 2: Write out format
  - `0b 0000000 ????? ????? 000 ????? 0110011`

R-Type: Instruction to Hex Translation (4/5)

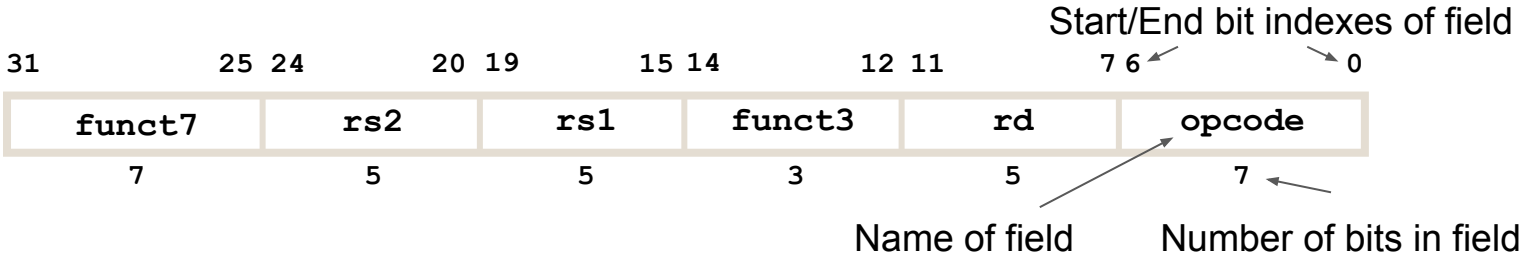


Translate "add s2 s3 s4" to hex

- Step 3: Registers
  - `s2 -> x18 ->0b10010 (rd)`
  - `s3 -> x19 ->0b10011 (rs1)`
  - `s4 -> x20 ->0b10100 (rs2)`

  - `0b 0000000 10100 10011 000 10010 0110011`

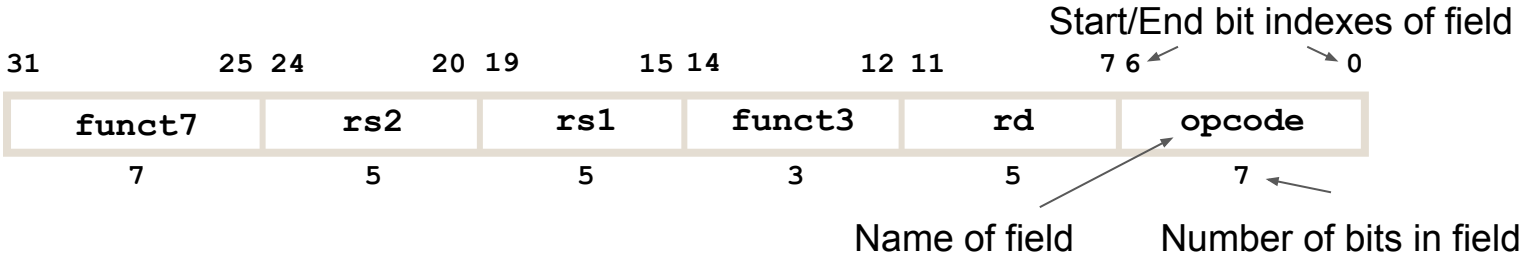
R-Type: Instruction to Hex Translation (5/5)



Translate "add s2 s3 s4" to hex

- Step 4: Convert to hex
  - `0b 0000000 10100 10011 000 10010 0110011`
  - `0b 0000 0001 0100 1001 1000 1001 0011 0011`
  - `0x01498933`

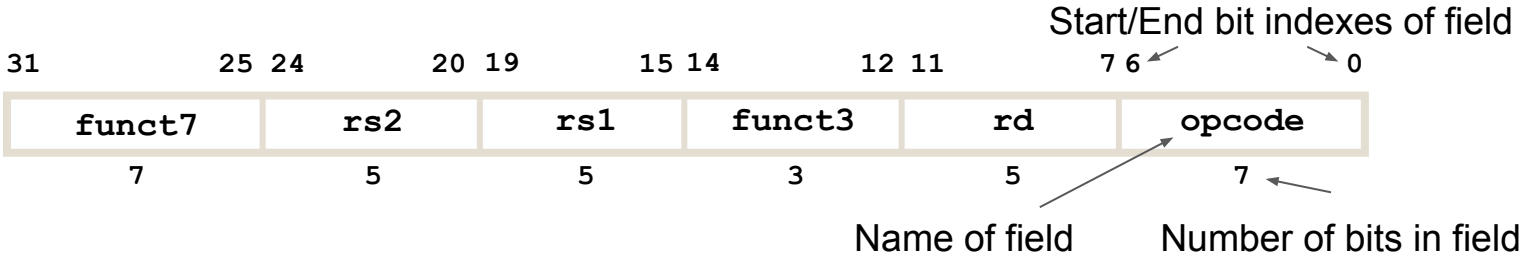
R-Type: Hex to Instruction Translation (1/3)



Translate "0x01B3 42B3" to instruction

- Step 1: Convert to binary and determine opcode and instruction type from reference card
  - `Binary: 0b0000 0001 1011 0011 0100 0010 1011 0011`
  - `Opcode: last 7 bits = 0b011 0011`
  - `Conclusion: R-type instruction`

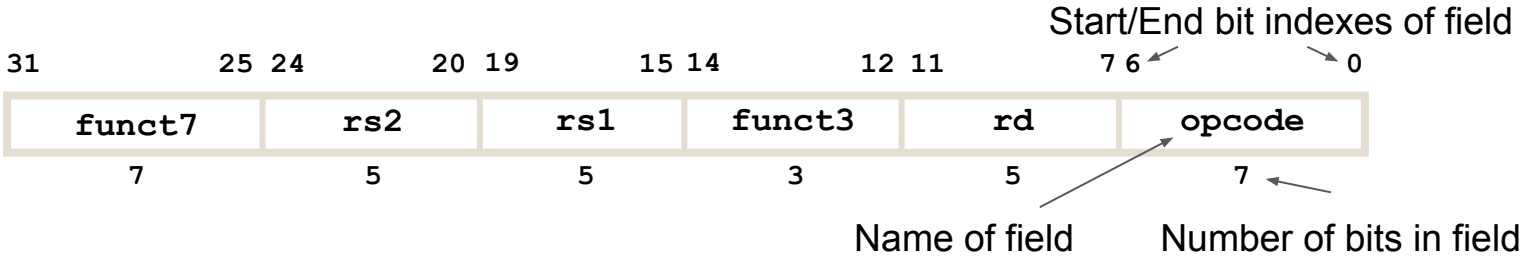
R-Type: Hex to Instruction Translation (2/3)



Translate "0x01B3 42B3" to instruction

- Step 2: Split according to R-type format
  - `Binary: 0b0000000 11011 00110 100 00101 0110011`
- Step 3: Determine funct3/funct7 for instruction
  - `funct3: 0b100`
  - `funct7: 0b000 0000`
  - `Conclusion: xor operation`

R-Type: Hex to Instruction Translation (3/3)



Translate "0x01B3 42B3" to instruction

- Step 2: Split according to R-type format
  - `Binary: 0b0000000 11011 00110 100 00101 0110011`
- Step 4: Determine registers
  - `rd:  0b00101 -> x5 -> t0`
  - `rs1: 0b00110 -> x6 -> t1`
  - `rs2: 0b11011 -> x27-> s11`
- Conclusion: `xor t0 t1 s11`

R-Type: All Instructions

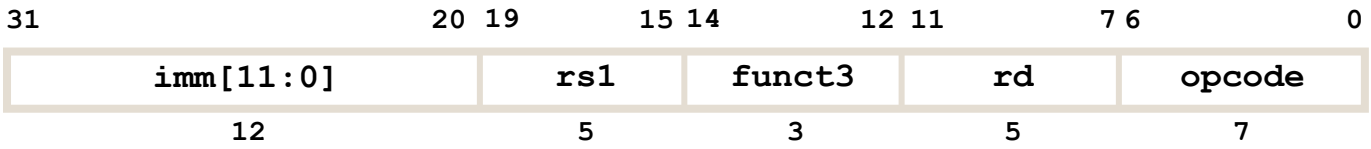
Same opcode

Instruction	Name	Description	Type	Opcode	Funct3	Funct7
`add rd rs1 rs2`	ADD	`rd = rs1 + rs2`	R	`011 0011`	`000`	`000 0000`
`sub rd rs1 rs2`	SUBtract	`rd = rs1 - rs2`	R	`011 0011`	`000`	`010 0000`
`and rd rs1 rs2`	bitwise AND	`rd = rs1 & rs2`	R	`011 0011`	`111`	`000 0000`
`or rd rs1 rs2`	bitwise OR	`rd = rs1 | rs2`	R	`011 0011`	`110`	`000 0000`
`xor rd rs1 rs2`	bitwise XOR	`rd = rs1 ^ rs2`	R	`011 0011`	`100`	`000 0000`
`sll rd rs1 rs2`	Shift Left Logical	`rd = rs1 << rs2`	R	`011 0011`	`001`	`000 0000`
`srl rd rs1 rs2`	Shift Right Logical	`rd = rs1 >> rs2` (Zero-extend)	R	`011 0011`	`101`	`000 0000`
`sra rd rs1 rs2`	Shift Right Arithmetic	`rd = rs1 >> rs2` (Sign-extend)	R	`011 0011`	`101`	`010 0000`
`slt rd rs1 rs2`	Set Less Than (signed)	`rd = (rs1 < rs2) ? 1 : 0`	R	`011 0011`	`010`	`000 0000`
`sltu rd rs1 rs2`	Set Less Than (Unsigned)		R	`011 0011`	`011`	`000 0000`

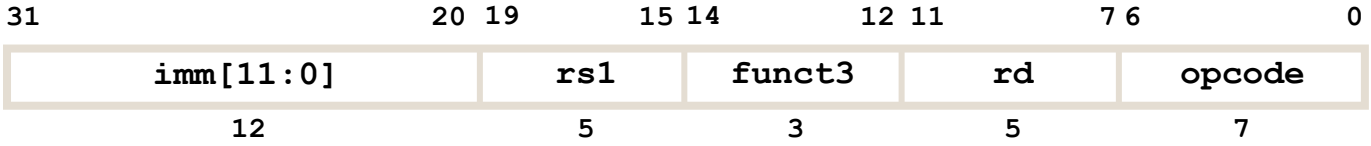
RISC-V Instruction Format - Agenda

- Intro
- R-types
- **I-types**
- S-types
- U-types
- B-types
- J-types
- Concluding Notes

I-Type

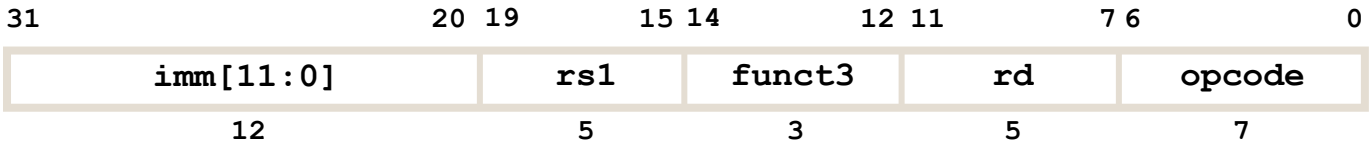


I-Type



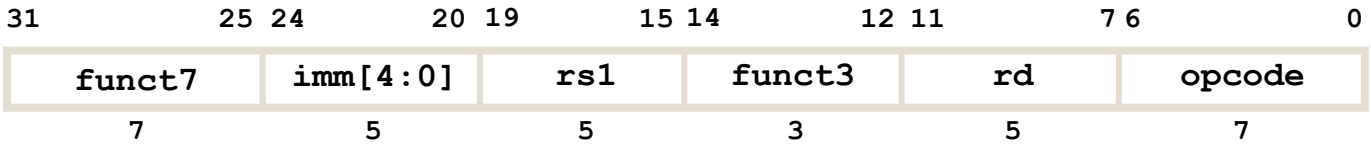
- Designed for instructions with 2 registers (rs1 and rd) and 1 immediate
  - Arithmetic operations with immediates (`addi, andi, slti, …`)
  - Loads (`lw, lh, lhu, lb, lbu`)
  - `jalr`
  - `ecall` and `ebreak` are also technically I-types, but they ignore the rd, rs1, and immediate, and their value isn't really in scope.
  - Stores use rs1 and rs2, so we have a separate instruction format for them.
- Most components are stored the same way as before, with the addition of the imm component

I-Type



- Immediate is stored in the component imm
  - Note the [11:0], which indicates that we store the 11th bit of the immediate at position 31, the 10th bit of the immediate at position 30, …, the 0th bit of the immediate at position 20
- I-type immediates are 12 bits
  - Therefore, we can only store a 12-bit integer as an immediate
- Most instructions use signed immediates, so our range for I-type immediates is [-2048, 2047].
  - Ex. "addi sp sp -2048" is valid, but "addi sp sp -2052" is NOT valid RISC-V code

I*-Type



- Special note: For shift instructions (`slli, srli, srai`), max shift of 31
  - Any larger shift will shift all our data off the number
- These instructions I*, a modified I-type that specifies a funct7
  - Why? Let's take a look at the instructions →

I-Type: Arithmetic Instructions

Same opcode
(different from R-types)

Instruction	Name	Description	Type	Opcode	Funct3	Funct7
`addi rd rs1 imm`	ADD Immediate	`rd = rs1 + imm`	I	`001 0011`	`000`	
`andi rd rs1 imm`	bitwise AND Immediate	`rd = rs1 & imm`	I	`001 0011`	`111`	
`ori rd rs1 imm`	bitwise OR Immediate	`rd = rs1 | imm`	I	`001 0011`	`110`	
`xori rd rs1 imm`	bitwise XOR Immediate	`rd = rs1 ^ imm`	I	`001 0011`	`100`	
`slli rd rs1 imm`	Shift Left Logical Immediate	`rd = rs1 << imm`	I*	`001 0011`	`001`	`000 0000`
`srli rd rs1 imm`	Shift Right Logical Immediate	`rd = rs1 >> imm` (Zero-extend)	I*	`001 0011`	`101`	`000 0000`
`srai rd rs1 imm`	Shift Right Arithmetic Immediate	`rd = rs1 >> imm` (Sign-extend)	I*	`001 0011`	`101`	`010 0000`
`slti rd rs1 imm`	Set Less Than Immediate (signed)	`rd = (rs1 < imm) ? 1 : 0`	I	`001 0011`	`010`	
`sltiu rd rs1 imm`	Set Less Than Immediate (Unsigned)		I	`001 0011`	`011`	

9 instructions
3 `funct3` bits

I-Type: Load and Jump Instructions

Same opcode
(different from arithmetic insts)

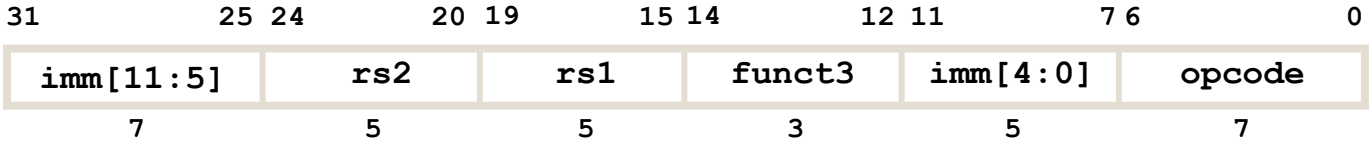
Instruction	Name	Description	Type	Opcode	Funct3	Funct7
`lb rd imm(rs1)`	Load Byte	`rd` = 1 byte of memory at address `rs1 + imm`, sign-extended	I	`000 0011`	`000`	
`lbu rd imm(rs1)`	Load Byte (Unsigned)	`rd` = 1 byte of memory at address `rs1 + imm`, zero-extended	I	`000 0011`	`100`	
`lh rd imm(rs1)`	Load Half-word	`rd` = 2 bytes of memory starting at address `rs1 + imm`, sign-extended	I	`000 0011`	`001`	
`lhu rd imm(rs1)`	Load Half-word (Unsigned)	`rd` = 2 bytes of memory starting at address `rs1 + imm`, zero-extended	I	`000 0011`	`101`	
`lw rd imm(rs1)`	Load Word	`rd` = 4 bytes of memory starting at address `rs1 + imm`	I	`000 0011`	`010`	

`jalr rd rs1 imm`	Jump And Link Register	`rd = PC + 4` `PC = rs1 + imm`	I	`110 0111`	`000`	

RISC-V Instruction Format - Agenda

- Intro
- R-types
- I-types
- **S-types**
- U-types
- B-types
- J-types
- Concluding Notes

S-Type



- Designed for instructions with 2 source registers and an immediate
  - **Store instructions**
- Why is immediate split up?
  - To keep rs1 and rs2 in same spots as in R-type instructions
- Immediate similar to I-types, but now need to "piece together the immediate"
  - Ex. If we had immediate 0b1101 0101 0001, then we would put 0b110 1010 in the first immediate box, and 0b10001 in the second immediate box.

S-Type: All Instructions

Instruction	Name	Description	Type	Opcode	Funct3	Funct7
`sb rs2 imm(rs1)`	Store Byte	Stores least-significant byte of `rs2` at the address `rs1 + imm` in memory	S	`010 0011`	`000`	
`sh rs2 imm(rs1)`	Store Half-word	Stores the 2 least-significant bytes of `rs2` starting at the address `rs1 + imm` in memory	S	`010 0011`	`001`	
`sw rs2 imm(rs1)`	Store Word	Stores `rs2` starting at the address `rs1 + imm` in memory	S	`010 0011`	`010`	

- Warning: rs2 comes before rs1 in store instructions!
  - rs1 is always the register added to the immediate

RISC-V Instruction Format - Agenda

- Intro
- R-types
- I-types
- S-types
- **U-types**
- B-types
- J-types
- Concluding Notes

U-type instructions: lui and auipc

- Load Upper Immediate:

```
lui rd imm
```

  - Sets rd to imm << 12

- Add Upper Immediate to Program Counter:

```
auipc rd imm
```

  - Sets rd to (imm << 12) + PC

- Recall: primarily used in two pseudoinstructions:
  - `li rd imm`: Set rd to imm
  - `la rd Label`: Set rd to the address of Label

LUI

- How to translate "`li t0 0x12345678`" to instructions?
  - "`addi t0 x0 0x12345678`"?
    - Immediate too big!
  - Multiple addis or addis with sllis would work
    - Need at least 2 slli+addi, plus an addi (5 instructions)
  - 1 instruction is is 32 bits → need at least 2 instructions to load a 32 bit immediate
- Solution: lui instruction
  - In the above example, we can do:

```
lui t0 0x12345
addi t0 t0 0x678
```

- This works, assuming lui (U-type) has 20 bits of immediate

LUI: Corner case

- How would you translate “`li t0 0xABCDEFFF`” to instructions?
- Initial idea:
  - ```
    lui t0 0xABCDE
    addi t0 t0 0xFFF
    ```
- Problem: 0xFFF isn't 4095; it's **-1**
  - After 1st line: t0 = 0xABCDE000
  - After 2nd line: t0 = 0xABCD**D**FFF
- How to fix?
  - “`lui t0 0xABCD`**`F`**” instead
  - `0xABCDF000 - 1 = 0xABCDEFFF`
- Affects li instructions when **offset’s 11th bit is 1**
    ```

AUIPC and Relative Addressing

- auipc used similarly with addi
- Difference is that it adds its result to PC
- Often when writing code, we want to allow multiple programs to be combined (like with libraries)
  - Might change addresses of instructions
- To avoid this issue, many instructions involving labels use relative addressing instead of absolute addressing.
  - Absolute address: "This label is at location 0x000000FC". This fails if our label moves to a different place in memory
  - Relative address: "This label is 48 bytes after the current line of code". This still works if we move both the line of code and the label the same distance.
- As such, auipc often gets used with la instructions.

U-Type

- Designed for instructions which need 20 immediate bits
  - lui and auipc
- Where is imm[11:0]?
  - Mismatch between how you write instruction, and its encoding
  - If we write "lui t0 0x12345", store 0x12345000 to rd

U-Type: All Instructions

Instruction	Name	Description	Type	Opcode	Funct3
`auipc rd immu`	Add Upper Immediate to PC	`imm = immu << 12` `rd = PC + imm`	U	`001 0111`	
`lui rd immu`	Load Upper Immediate	`imm = immu << 12` `rd = imm`	U	`011 0111`	

RISC-V Instruction Format - Agenda

- Intro
- R-types
- I-types
- S-types
- U-types
- **B-types**
- J-types
- Concluding Notes

Labels

- Labels **don’t exist in machine code**. When translating RISC-V to binary, we need to convert all labels into explicit references to a particular line of code
- Recall: Since we want to be able to move around code blocks in memory, we prefer to use **relative addressing** instead of **absolute addresses**.
- Solution: When assembling code that uses labels, first convert the label into an ***offset***, which specifies how many bytes off we would need to jump to get to that label.

Example: Converting Labels into offsets (1/4)

Translate the labels in the following code into their corresponding offsets:

```
            beq x0 x0 target
            addi x0 x0 100
target:     addi x0 x0 100
            j target
            li t0 0x5F3759DF
            beq t0 t0 target
```

Example: Converting Labels into offsets (2/4)

Translate the labels in the following code into their corresponding offsets:

```
            beq x0 x0 target #+2 instructions = 8 bytes, so offset=8
            addi x0 x0 100
target:     addi x0 x0 100
            j target
            li t0 0x5F3759DF
            beq t0 t0 target
```

Example: Converting Labels into offsets (3/4)

Translate the labels in the following code into their corresponding offsets:

```
            beq x0 x0 target #+2 instructions = 8 bytes, so offset=8
            addi x0 x0 100
target:     addi x0 x0 100
            j target #-1 instruction = -4 bytes, so offset=-4
            li t0 0x5F3759DF
            beq t0 t0 target
```

Example: Converting Labels into offsets (4/4)

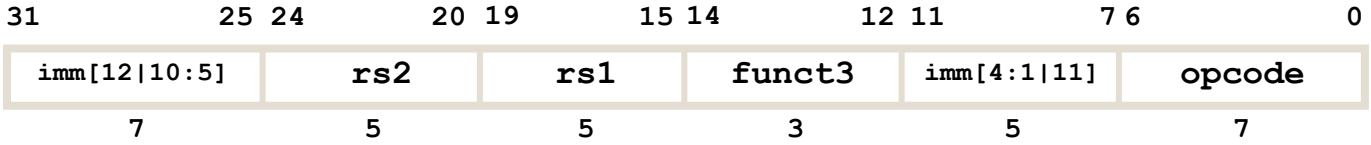
Translate the labels in the following code into their corresponding offsets:

```
            beq x0 x0 target #+2 instructions = 8 bytes, so offset=8
            addi x0 x0 100
target:     addi x0 x0 100
            j target #-1 instruction = -4 bytes, so offset=-4
            li t0 0x5F3759DF #The li here is actually 2 instructions
            beq t0 t0 target #-4 instructions, so offset=-16
```

Storing offsets

- Note that all the previous offsets were multiples of 4
  - One instruction always takes 4 bytes of memory (in RV32)
- If we stored the immediate directly as a signed number, we'd always have the last two bits 0s.
  - Limited number of immediate bits limits jump distance
  - Don’t store constant 0 bits → we can extend our immediate and allow for longer jumps
- In RISC-V: don’t store the lowest bit of an offset immediate
  - Some RISC-V extensions use 16-bit instructions, so keep things consistent and only cut off 1 bit

B-Type



- 2 source registers and an immediate (similar to S-Type)
  - B-type sometimes referred to as SB-type
- Note that the immediate is stored in a strange pattern
  - If immediate is binary **`0bABCDEFGHIJKLM`** (each letter 1 bit), components are:

    `0bACDEFGH` | `rs2` | `rs1` | `funct3` | **`0bIJKLB`** | `opcode` (Bit M not stored)

  - Why?
    - MSB goes in instruction MSB, to simplify sign-extension
    - Bits 10:1 have same position as in S-type instructions
- How far can we jump?
  - 12+1 = 13-bit immediates = [-4096, 4094] range, or up to 2^{10} instructions up/down.

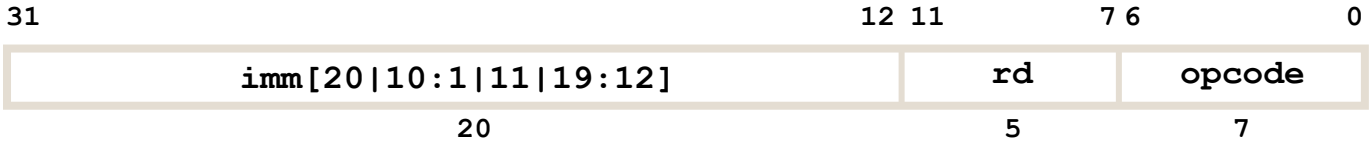
B-Type: All Instructions

Instruction	Name	Description	Type	Opcode	Funct3
`beq rs1 rs2 label`	Branch if EQual	`if (rs1 == rs2) PC = PC + offset`	B	`110 0011`	`000`
`bge rs1 rs2 label`	Branch if Greater or Equal (signed)	`if (rs1 >= rs2) PC = PC + offset`	B	`110 0011`	`101`
`bgeu rs1 rs2 label`	Branch if Greater or Equal (Unsigned)		B	`110 0011`	`111`
`blt rs1 rs2 label`	Branch if Less Than (signed)	`if (rs1 < rs2) PC = PC + offset`	B	`110 0011`	`100`
`bltu rs1 rs2 label`	Branch if Less Than (Unsigned)		B	`110 0011`	`110`
`bne rs1 rs2 label`	Branch if Not Equal	`if (rs1 != rs2) PC = PC + offset`	B	`110 0011`	`001`

RISC-V Instruction Format - Agenda

- Intro
- R-types
- I-types
- S-types
- U-types
- B-types
- **J-types**
- Concluding Notes

J-Type



- jal instructions use only 1 destination and an immediate, so we can use similar format to U-type for extra immediate bits
  - J-type sometimes referred to as UJ-type
- Note that the immediate is stored in an even stranger pattern
  - Binary `0bA BCDE FGHI JKLM NOPQ RSTU` (each letter is a bit) → gets stored as `0b AKLM NOPQ RSTJ BCDE FGHI`. As before, the last bit isn't stored.
  - Note that we put the MSB of our immediate in the MSB of our instruction, bits 19-12 in the same spot as U-types, and bits 10-1 in the same spot as I-types.
- 20+1 = 21-bit immediates, so up to 2^{18} instructions up/down

J-Type: All Instructions

Instruction	Name	Description	Type	Opcode	Funct3
`jal rd label`	Jump And Link	`rd = PC + 4` `PC = PC + offset`	J	`110 1111`	

RISC-V Instruction Format - Agenda

- Intro
- R-types
- I-types
- S-types
- U-types
- B-types
- J-types
- **Concluding Notes**

How to handle immediates larger than you can store (1/2)

- R-type instructions
  - Don’t have immediates
- U-type instructions
  - Used to handle these immediates
- I-type and S-type instructions
  - For arithmetic instructions, it's generally possible to store the immediate in a temporary first
    - Ex. if we want to do "`xor t0 t1 0xDEADBEEF`", we can do:

      ```
      li t2 0xDEADBEEF
      xor t0 t1 t2
      ```
  - For loads and stores, we can add the offset first, then do a 0-offset load (as with variable offset loads)

How to handle immediates larger than you can store (2/2)

- B-type and J-type instructions:
  - If a branch is:
    - Within 1024 instructions?
      - Branch normally (ex. `beq t0 t1 Label`)
    - Greater than 1024 instructions?
      - Invert the branch condition, and do a j instruction instead:

```
bne t0 t1 Next
j Label
Next:
```

  - If a jump is:
    - Within 2^{18} instructions?
      - Jump normally (ex. `j Label`)
    - Greater than 2^{18} instructions?
      - Do an auipc, then use jalr's immediate to offset the rest:

```
auipc t0 0x12345
jalr ra t0 0x678
```

Summary

- Each RISC-V instruction gets translate into a 32 bit encoding by the assembler.
- There are 7 formats for these encodings: R, I, I*, S, B, U, J
- They are designed to minimize the complexity of the circuits needed to run these binary-encoded instructions.
- You can translate between instructions and hex by following a standard procedure.
- Specific encodings can be found on the reference card (will be provided in exams).