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

Lectures 11+12: RISC-V Instruction Formats

Instructors: Dan Garcia, Justin Yokota

Computer Architecture in the News

North Carolina State University (NC State) researchers proposed the addition of a “white light” to traffic signals to allow autonomous vehicles (AVs) to help control the flow of traffic. This “white phase” is based on the ability of AVs to communicate with other AVs and the computer controlling the traffic signal. The white light would be activated when multiple AVs approach the intersection, indicating the AVs are coordinating their movement to facilitate more efficient traffic flow.

NC State’s Ali Hajbabaie explained, “Red lights will still mean stop. Green lights will still mean go. And white lights will tell human drivers to simply follow the car in front of them.” The traffic signal would return to the traditional green-yellow-red signal pattern when the intersection is again controlled by human drivers. In simulations, the researchers found AVs and the white phase would improve traffic flow and reduce fuel consumption.

https://news.ncsu.edu/2023/02/traffic-light-for-autonomous-cars/

Agenda

- Lecture 11
  - Calling Convention Overview
  - Intro
  - R-types
  - I-types
  - S-types
- Lecture 12
  - U-types
  - B-types
  - J-types
  - Concluding Notes
- (Putting these two lectures together so it's easier to reference later)

Agenda

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

Calling Convention

- Technically RISC-V doesn't mandate CC. This means that there are NO indicators that you're breaking CC until you try using it with someone else's code
- After you call a function, you should assume (even if you're calling your own function!):
    - All the t registers, a registers, and ra were set to random values
    - All the s registers are the same
    - sp is the same as before, and everything above sp is the same
- Before you return from a function, make sure that:
    - The sp and all s registers are back to their original value
    - ra is set to the value it was when you called the function (remember that ret is just jr ra!)
    - Your output is in a0/a1 according to the spec
    - Your code doesn't use the value of t0 (and other registers) before setting it

Some Final Notes on RISC-V Coding

- When a program starts, all registers are set to 0, except sp, which is set to the top of the stack.
- x3 and x4 are the registers gp and tp, and are used for storing the global pointer (for use with the heap) and thread pointer (for use with threads), respectively. Ignore them for the purposes of this class.
- ecall *can* be used to run various environment calls (ex. prints, sbrk, exit), but you don't need to use them (we give helper functions in utils)
- Don't start a comment with "define", "if", or "ifdef"; it gets interpreted as a preprocessor command because local Venus is bad.

Agenda

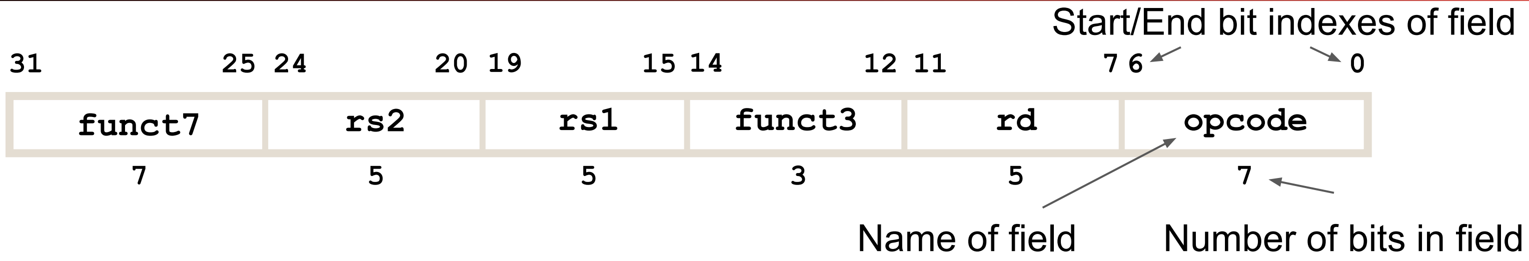
Overview

- Assembly languages are primarily useful because they can be directly translated into binary code that can be run by a CPU.
- RISC-V has a particularly simple structure: Each instruction is translated into instructions of the same length; for RV32 (the version we learn in this class), each instruction is 32 bits (4 bytes) long.
- Different instructions require different values
    - "add" specifies 3 register inputs
    - "addi" specifies 2 registers and 1 immediate
- As such, we define multiple formats, with each instruction getting encoded in its format.
- Overall design philosophy: Split the 32 bits into "chunks" for each component of an instruction, and try to overlap these chunks as much as possible to simplify the underlying circuit.
- Most of this information is presented in compressed form on our reference card, so there's no need to memorize the exact numbers associated with each instruction

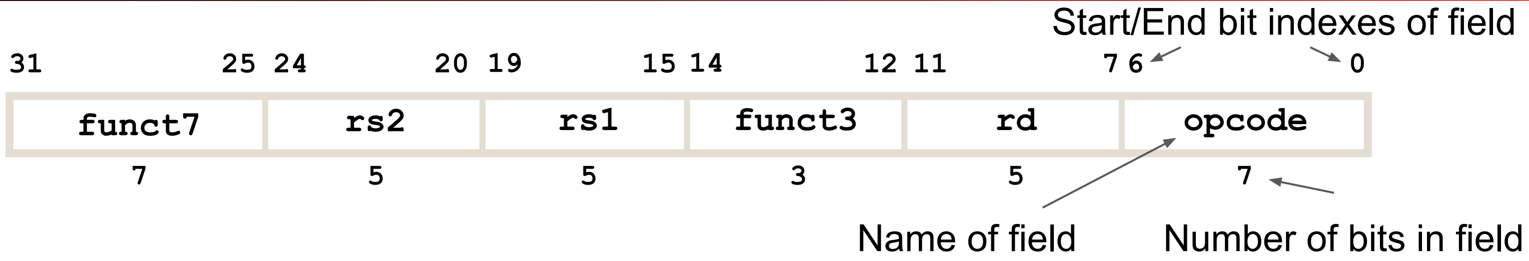
Agenda

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

R-Type

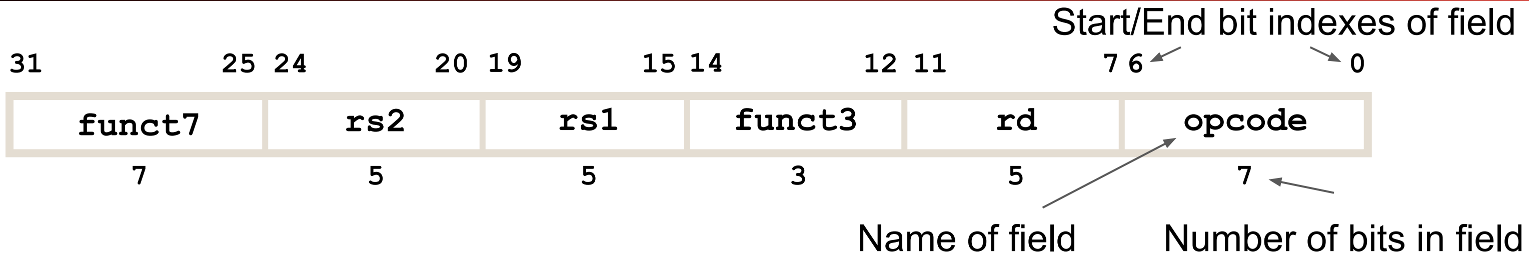

Start/End bit indexes of field
31
25 24
20 19
15 14
12 11
7 6
0
funct7
rs2
rs1
funct3
rd
opcode
7
5
5
3
5
7
Name of field
Number of bits in field

R-Type



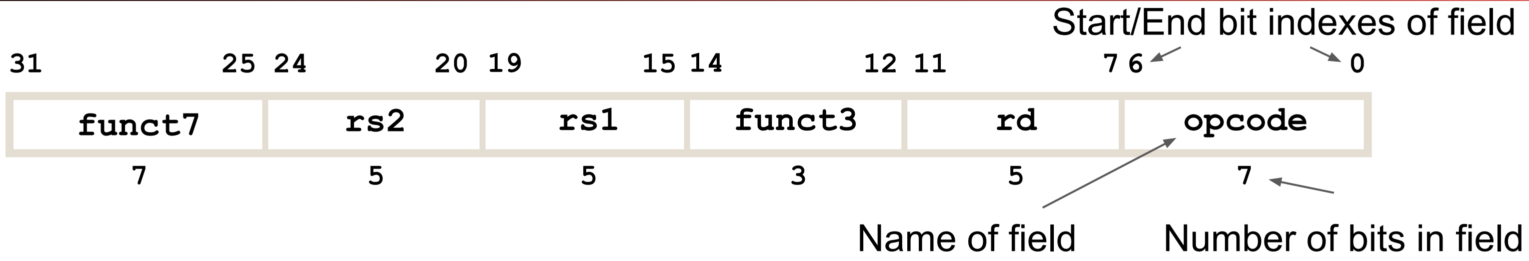
- Designed for instructions with 3 registers and no immediate
  - Arithmetic operators like add or sub
- Each register is identified by its number. 32 registers → 5 bits to identify one register uniquely
  - x0 → 0b00000
  - a0 → x10 → 0b01010
- rd: Destination register
- rs1: 1st source register
- rs2: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

R-Type



- opcode: Instruction identifier: Always the last 7 bits of the instruction over all instruction formats
  - Can therefore be used to determine which instruction format is currently in use.
- Some sets of similar instructions get assigned the same opcode
  - Ex. All arithmetic R-type instructions have the opcode 0x33
- 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 funct7 (such as sra and srl)

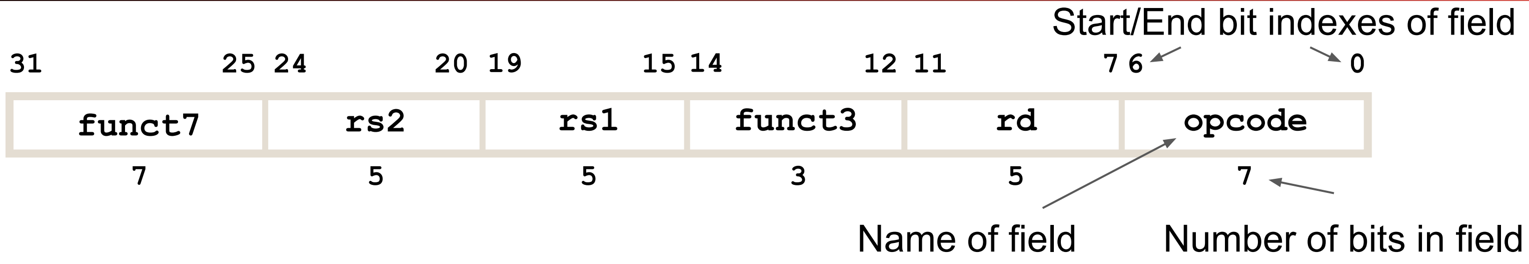
R-Type: Example Translation



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 ??????? ????? ????? ??? ????? ???????`

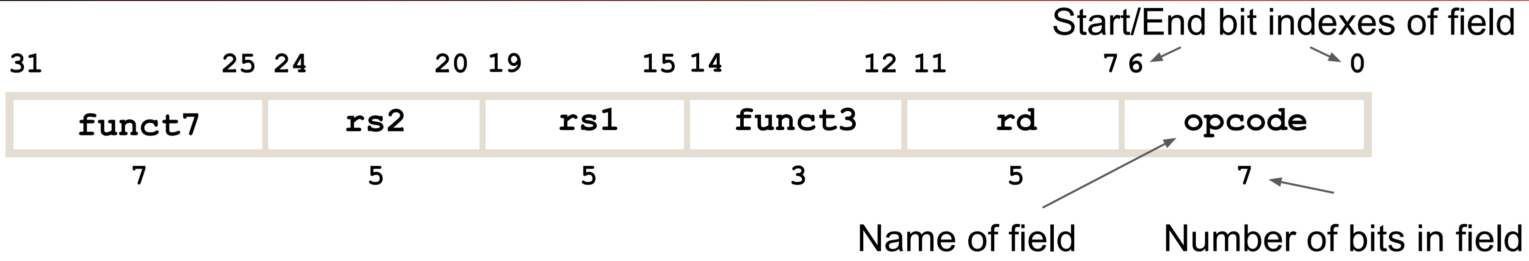
R-Type: Example Translation



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: Example Translation

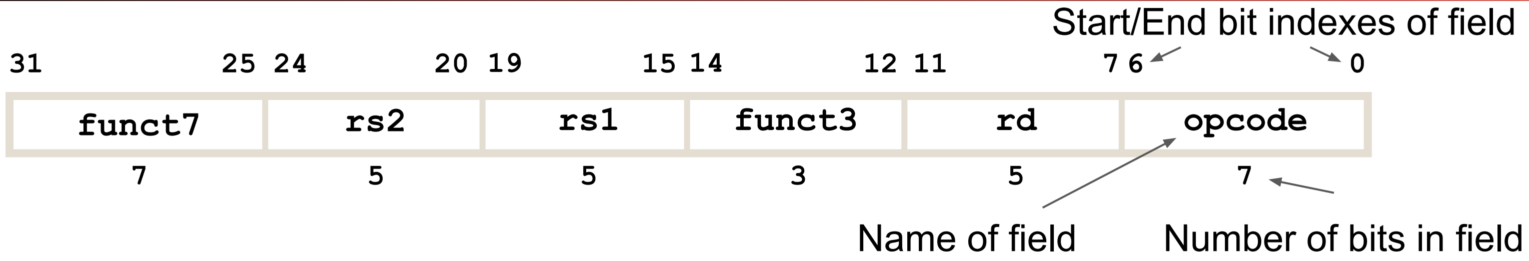


Translate "add s2 s3 s4" to hex

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

  - `0b 0000000 10100 10011 000 10010 0110011`

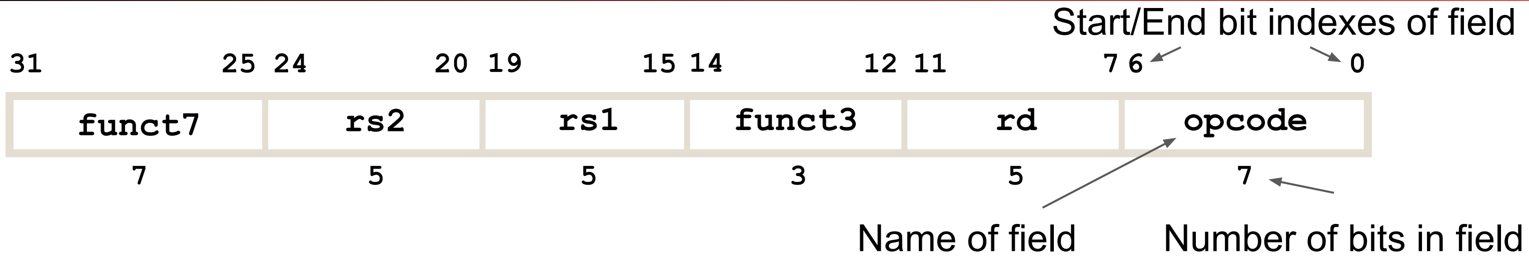
R-Type: Example Translation



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`

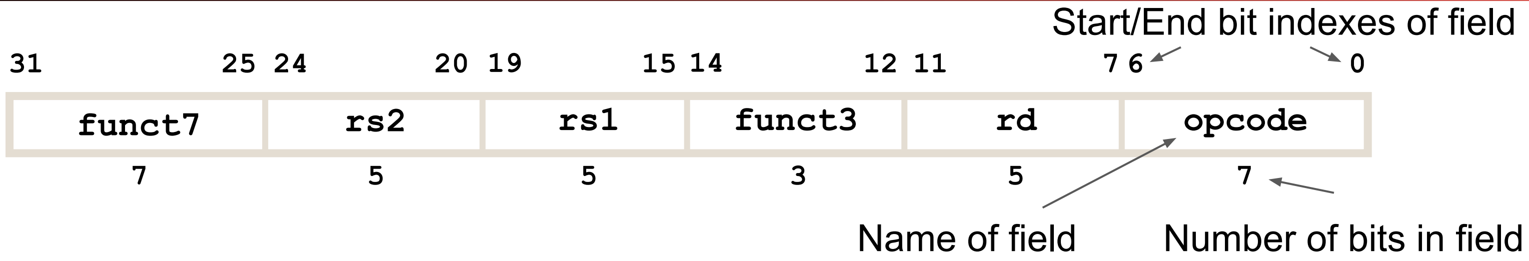
R-Type: Example Translation



Translate "0x01B3 42B3" to hex

- 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`

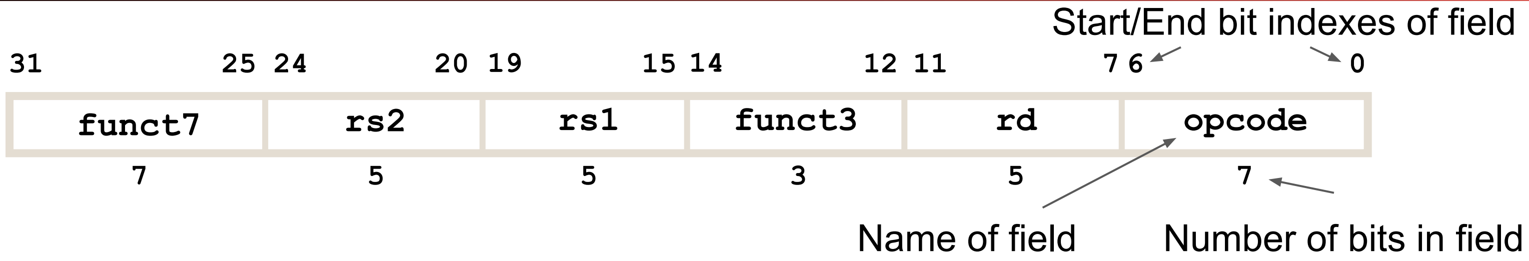
R-Type: Example Translation



Translate "0x01B3 42B3" to hex

- 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`

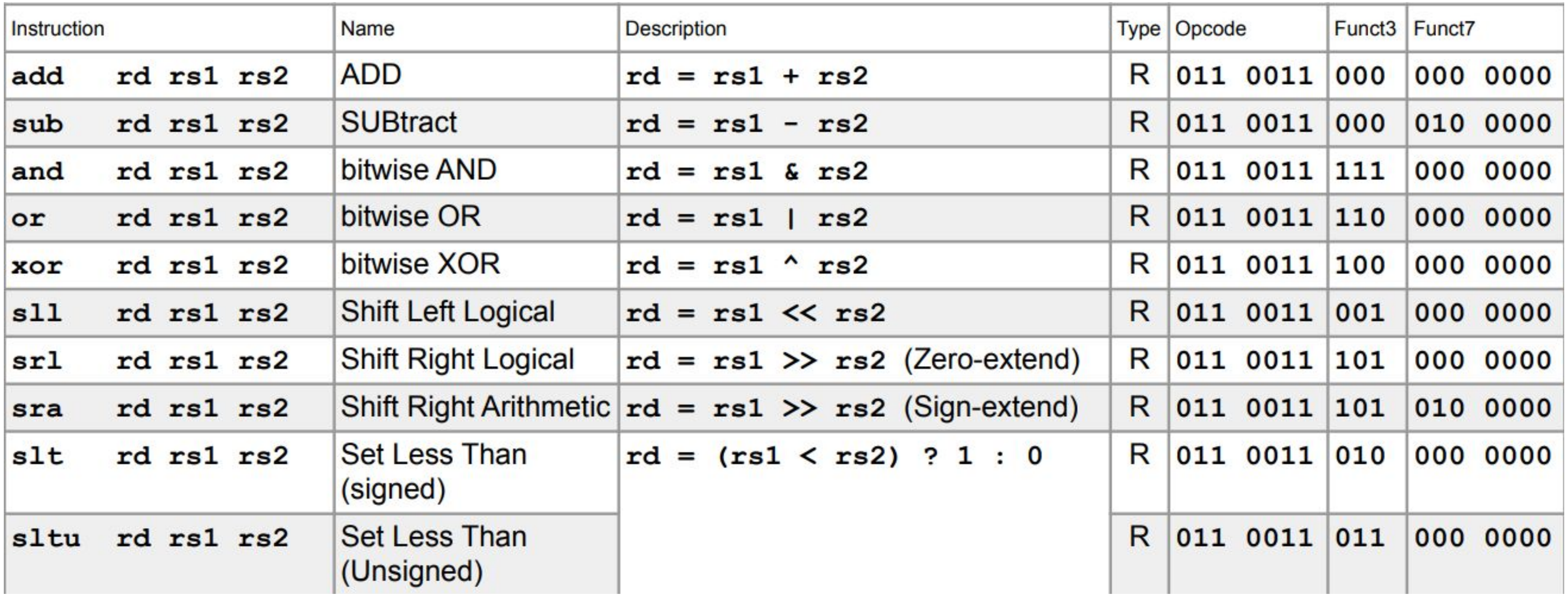
R-Type: Example Translation



Translate "0x01B3 42B3" to hex

- 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



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

Agenda

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

I-Type

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7

I-Type

31 20	19 15	14 12	11 7	6 0
`imm[11:0]`	`rs1`	`funct3`	`rd`	`opcode`
12	5	3	5	7

- Designed for instructions with 2 registers (rs1 and rd) and 1 immediate
  - Arithmetic operations with immediates
  - Loads
  - 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

31 20	19 15	14 12	11 7	6 0
`imm[11:0]`	`rs1`	`funct3`	`rd`	`opcode`
12	5	3	5	7

- 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

31 25	24 20	19 15	14 12	11 7	6 0
funct7	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7

- Special note: For shift instructions (slli, srli, srai), we only have a max shift of 31
    - Any larger shift will shift all our data off the number
- As such, these instructions use a modified I-type that specifies a funct7

I-Type: Arithmetic Instructions

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	

I-Type: Load and Jump Instructions

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`

Agenda

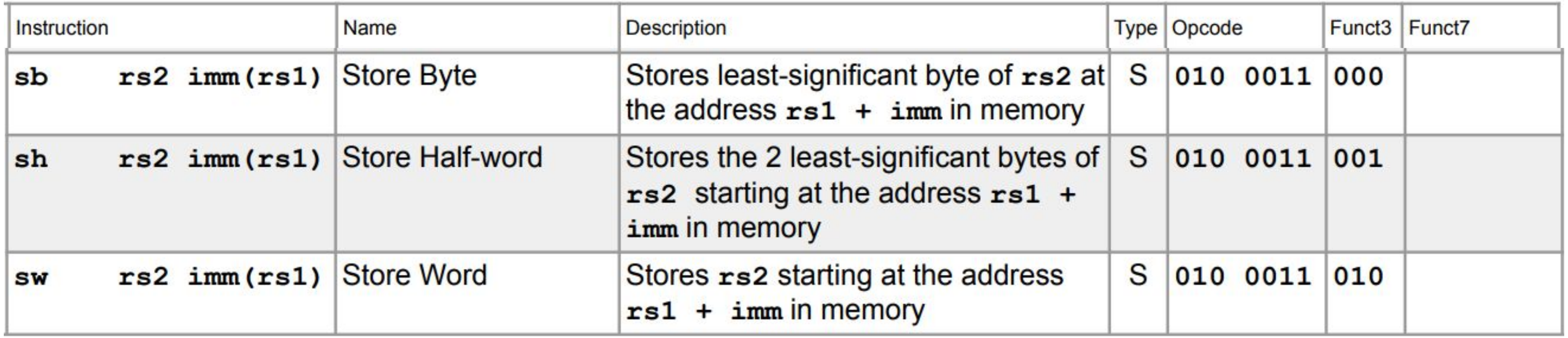
- Calling Convention Overview
- Intro
- R-types
- I-types
- **S-types**
- U-types
- B-types
- J-types
- Concluding Notes

S-Type

31 25	24 20	19 15	14 12	11 7	6 0
`imm[11:5]`	`rs2`	`rs1`	`funct3`	`imm[4:0]`	`opcode`
7	5	5	3	5	7

- Designed for instructions with 2 source registers and an immediate
  - Store instructions
- Note that we put rs1 and rs2 in the same spots as in R-type instructions, so we need to split the immediate bits to "fill in" the remaining gaps.
- Immediate is similar, but now we 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!
  - This is because we want rs1 to always be the register that gets added to immediates, to simplify our circuitry

Practice: Translate an instruction!

The code for this is embedded in HW 4.7

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

Lectures 11+12: RISC-V Instruction Formats

Instructors: Dan Garcia, Justin Yokota

Computer Architecture in the News

New artificial intelligence technology being developed by engineers at Brigham Young University could significantly cut down on the time and cost that goes into film study for Super Bowl-bound teams (and all NFL and college football teams), while also enhancing game strategy by harnessing the power of big data.

BYU professor D.J. Lee, master's student Jacob Newman and Ph.D. students Andrew Sumsion and Shad Torrie are using AI to automate the time-consuming process of analyzing and annotating game footage manually. Using deep learning and computer vision, the researchers have created an algorithm that can consistently locate and label players from game film and determine the formation of the offensive team — a process that currently requires a slew of video assistants.

"We obtain greater than 90% accuracy on both player detection and labeling, and 84.8% accuracy on formation identification. These results prove the feasibility of building a complete American football strategy analysis system using artificial intelligence."

Source:https://news.byu.edu/intellect/new-byu-developed-ai-tecg-could-benefit-future-super-bowl-opponents
https://www.mdpi.com/2079-9292/12/3/726

Agenda

U-type instructions: lui and auipc

- Up until now, we haven't talked about what these two instructions actually do
- 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

- Consider li:
    - How would you translate "li t0 0x12345678" to instructions?
        - Can't just do "addi t0 x0 0x12345678" because that's way too big
    - Multiple addis or addis with sllis would work, but require many instructions for some numbers.
    - Ideally, we want to be able to do this in exactly 2 instructions
        - 1 instruction is impossible since no 32-bit object can encode all 2^{32} possible immediates AND all 32 possible destination registers
- Solution: lui instruction
    - In the above example, we can do:

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

- This works, as long as we give U-type instructions 20 bits of immediate

LUI: Corner case

- How would you translate "li t0 0xABCDEFFF" to instructions?
- Initial idea:

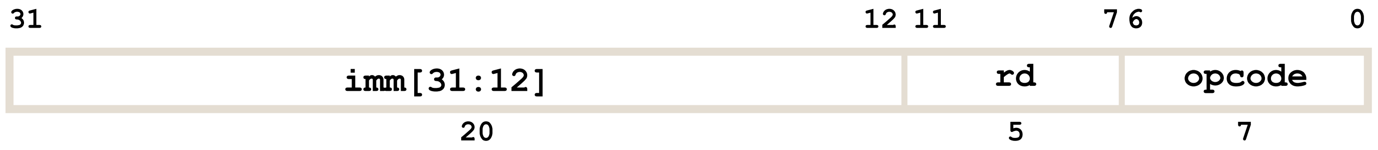
  -
    ```
    lui t0 0xABCDE
    addi t0 t0 0xFFF
    ```
- Problem: 0xFFF isn't 4095; it's -1
  - After the first line, we get t0 = 0xABCDE000
  - After the second line, we get t0 = 0xABCDDFFF instead!
- As such, we need to be careful in this case and do lui t0 0xABCDF instead

  -
    ```
    lui t0 0xABCDF #t0 stores 0xABCDF000
    addi t0 t0 0xFFF #t0 stores 0xABCDEFFF
    ```
- This ends up affecting li instructions only when the offset has its 11th bit set to 1, so it's an easy case to forget about.

AUIPC and Relative Addressing

- auipc similarly gets used primarily as a way to save an arbitrary value when used with an addi
- The main 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), but that would change the addresses of all the labels in our code
- 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 we move our code 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
- Note that there's a slight inconsistency between how the immediate is treated in the instruction format compared to the instruction itself
    - Note that the instruction format listed here doesn't store the bottom 12 bits of the value imm
    - If we write "lui t0 0x12345", we should treat imm as 0x12345000, and thus store 0x12345 in those 20 bits, instead of 0x00123.
    - This is due to RISC-V not actually having a formal syntax (RISC-V only specifies the binary encoding), so the syntax that got adopted was a variation of x86 syntax

U-Type: All Instructions

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

Agenda

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

Labels

- Recall: Labels don't actually exist. 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 writing code using 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

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

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

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

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

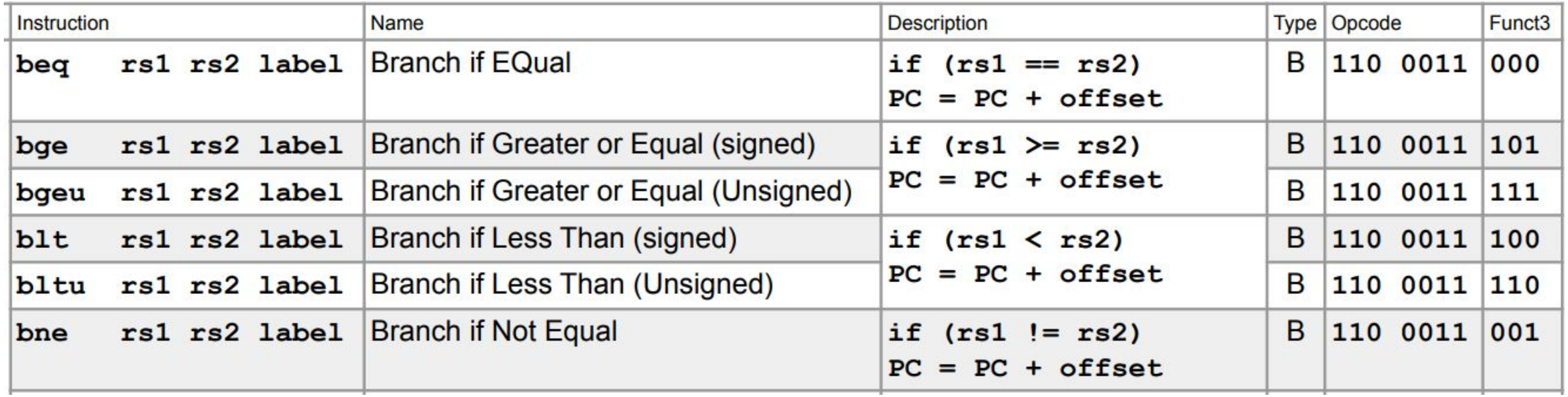
- Note that all the previous offsets were multiples of 4
    - Each instruction is always going to take 4 bytes of memory, so all offsets should be multiples of 4
- If we stored the immediate directly as a signed number, we'd always have the last two bits 0s.
    - At the same time, we have only a limited number of bits to store an immediate, which limits the total distance we can jump
    - If we can decide not to store those 0 bits, we can extend our immediate and allow for longer jumps
- Therefore, we don't store the lowest bit of an offset immediate
    - Some RISC-V extensions use 16-bit instructions, so we can't choose not to store the bottom two bits

B-Type

31 25	24 20	19 15	14 12	11 7	6 0
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
7	5	5	3	5	7

- Branch instructions also use 2 source registers and an immediate, so the format is similar to S-Type
  - This format is sometimes referred to as SB-type for that reason
- Note that the immediate is stored in a strange pattern
  - If we had the binary `0bA BCDE FGHI JKLM` (where each letter was a bit), the first box would store `0bACD EFGH` and the second box would store `0bI JKLB.` Bit M isn't stored.
  - This is also to simplify the underlying circuit; note that we put the MSB of our immediate in the MSB of our instruction (to simplify sign-extension), and other than that put 10 of the remaining 11 bits in the same position as S-type instructions
- Branch instructions have 13-bit immediates = [-4096, 4094] range, which is up to 2^{10} instructions up/down.

B-Type: All Instructions

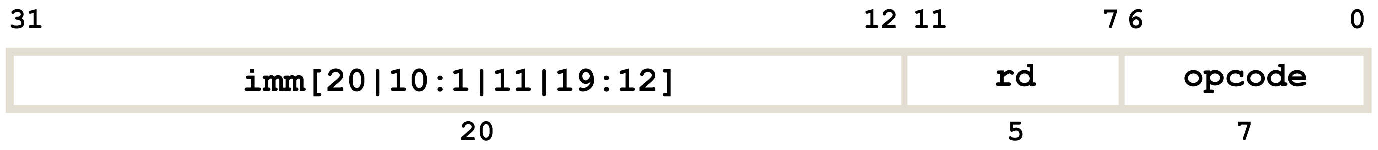


Instruction	Name	Description	Type	Opcode	Funct3
`beq rs1 rs2 label`	Branch if EQual	`if (rs1 == rs2)` `PC = PC + offset`	B	110 0011	000
`bge rs1 rs2 label`	Branch if Greater or Equal (signed)	`if (rs1 >= rs2)` `PC = PC + offset`	B	110 0011	101
`bgeu rs1 rs2 label`	Branch if Greater or Equal (Unsigned)		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

Agenda

- Calling Convention Overview
- 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 the U-type format for extra immediate bits
    - This format is sometimes referred to as UJ-type for that reason
- Note that the immediate is stored in an even stranger pattern
    - If we had the binary `0bA BCDE FGHI JKLM NOPQ RSTU` (where each letter was a bit), the data would be 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.
- Jumps have 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`	

Practice: Translate an instruction!

The code for this is embedded in HW 4.7

Agenda

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

How to handle immediates larger than you can store

- R-type and U-type instructions
  - Unneeded, since they either don't have immediates or have very specific use cases that never need to exceed the given immediate length
- 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

- 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

- Information on instruction formats and specific opcode/funct values are provided on the reference card here: https://cs61c.org/su22/pdfs/resources/reference-card.pdf