

CS 61C:

Great Ideas in Computer Architecture

More Function Calls &

RISC-V Instruction Formats

Administrivia

- Project 1 in the books!
- Project 2 spec released: RISC-V Assembly Language Programming
 - Deadlines: part A - Mon 10/4, part B - Mon 10/11
- Assignments Due this Week:
 - Homework 3: 9/24 (Friday)
 - Lab 3, RISC-V Assembly Language, due 9/24
 - Reminder: lab checkoffs will end *promptly* at 4PM on Fridays!
- Upcoming Assignments:
 - Homework 4 released, due next Friday 10/1

Administrivia

- **Midterm Exam rescheduled:**
- Based on poll - new date/time:
- Tuesday, October 19th, 7:00 PM - 9:00 PM PT
 - reply to piazza poll on scope
 - in-person and remote option
 - more details later

Outline

- More Function Calls
- RISC-V Instruction Formats

Outline

- More Function Calls
- RISC-V Instruction Formats

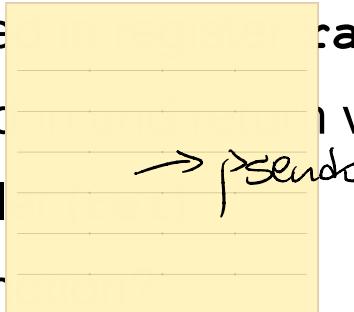
Review: To call a Function

- Use **jal** instruction to transfer control to function (callee)

- Register convention:

- return address is saved in **ra**
 - arguments get passed in values in **a0-a7**

- Use **jalr ra** to return to caller
- What If a Function Calls a Function?



jump and link

jal ra offset

Jump to PC + offset,
set ra to PC + 4

jalr ra rd (offset)

Jump to rd + offset
Set ra to PC + 4

- Note: this could mean a function calling itself - *recursion*.
 - Would clobber (overwrite) the values in **a0-a7** and **ra**
 - What is the solution?

Nested Procedures (1/2)

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

- Function called **sumSquare** is calling **mult**
- So there's a value in **ra** that **sumSquare** wants to jump back to, but this will be overwritten by the call to **mult**

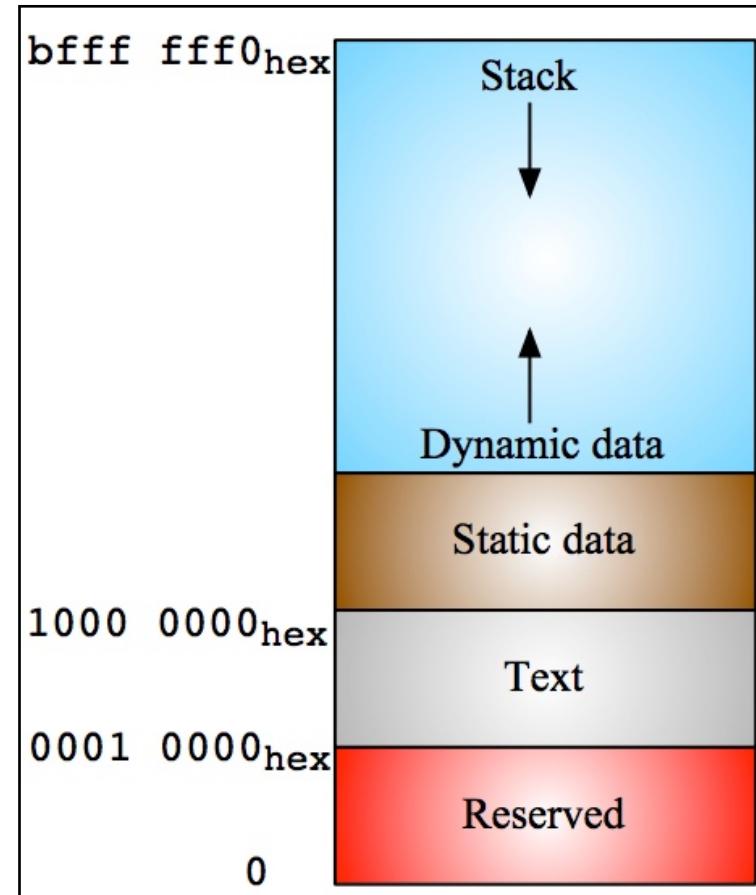
Need to save **sumSquare** return address before call to **mult**

Nested Procedures (2/2)

- In general, may need to save some registers in addition to `ra`.
- Again, use the *stack* for this.
- When a C program is run, there are three important memory areas allocated:
 - **Static**: Variables declared once per program, cease to exist only after execution completes - e.g., C **globals**
 - **Heap**: Variables declared dynamically via `malloc`
 - **Stack**: Space to be used by procedure during execution; this is where we can save register values AND local variables

RV32 Memory Allocation

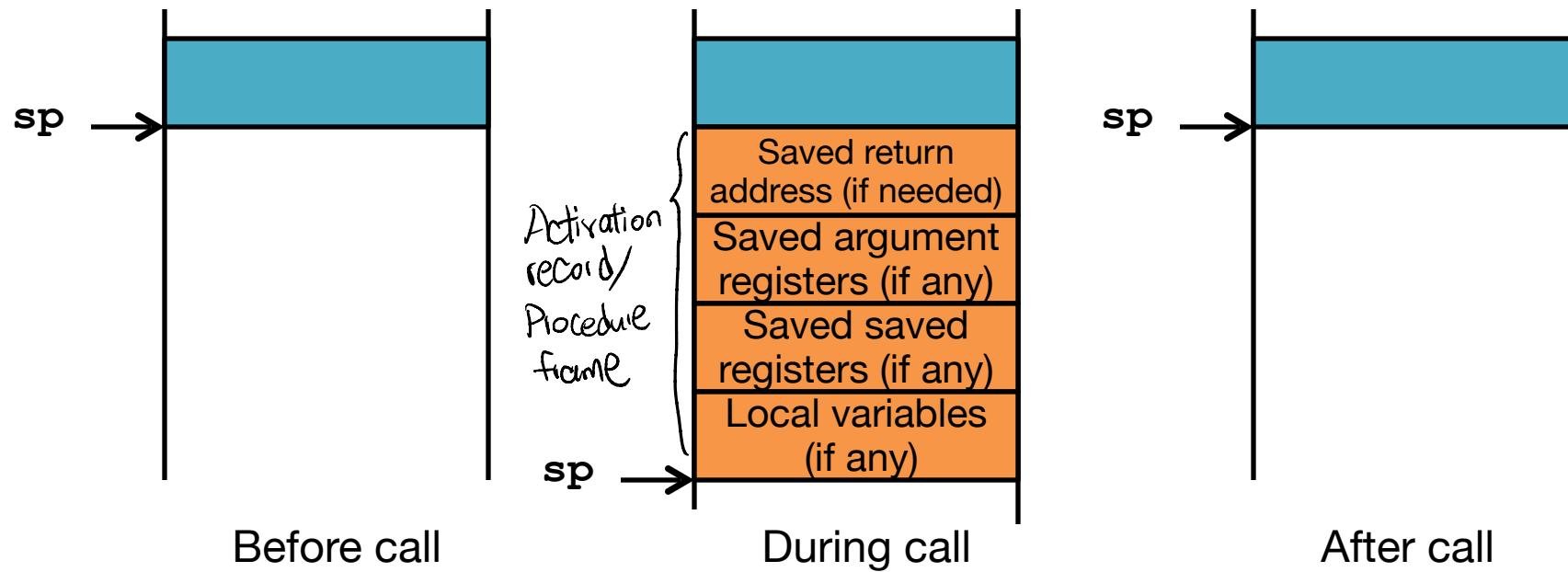
- RV32 convention (RV64 and RV128 have different memory layouts)
- Stack starts in high memory and grows down
 - Hexadecimal (base 16) : $bfff_fff0_{hex}$
- RV32 programs (*text segment*) in low end
 - 0001_0000_{hex}
- *static data segment* (constants and other static variables) above text for static variables
 - RISC-V convention *global pointer* (*gp*) points to static
 - RV32 *gp* = 1000_0000_{hex}
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses



Allocating Space on Stack

- C has two storage classes: automatic and static
 - *Automatic* variables are local to a function and discarded when function exits
 - *Static* variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that aren't in registers
- *Procedure frame* or *activation record*: segment of stack with saved registers and local variables

Stack Before, During, After Function



Using the Stack (1/2)

- So we have a register **sp** which always points to the last used space in the stack
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info
- So, how do we compile this?

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

Using the Stack (2/2)

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y; }
```

sumSquare:

“push”

```
    addi sp,sp,-8    # reserve space on stack  
    sw ra, 4(sp)    # save ret addr  
    sw a1, 0(sp)    # save y  
    mv a1,a0  
    jal mult  
    lw a1, 0(sp)
```

~~mult(x,x)~~ Set second parameter to x

call mult Now a0 is result of x^*x

restore y Restore original second arg
mult() + y returnVal = $x^*x + y$

“pop”

```
    add a0,a0,a1  
    lw ra, 4(sp)  
    addi sp,sp,8  
    jr ra
```

get ret addr

restore stack

mult: ...

A Richer Translation Example...

- **struct node {unsigned char c, struct node *next};**
 - c will be at 0, next will be at 4 because of alignment
 - `sizeof(struct node) == 8`
- **struct node * foo(char c) {**
 struct node *n;
 if (c < 0) return 0;
 n = malloc(sizeof(struct node));
 n->next = foo(c - 1);
 n->c = c;
 return n;
}

So What Will We Need?

- We'll need to save **ra**
 - Because we are calling other functions
- We'll need a local variable for **c**
 - Because we are calling other functions, lets put this in **s0**
- We'll need a local variable for **n**
 - Lets put this in **s1**
- So lets form the “preamble” and “postamble”
 - What we always do on entering and leaving the function
 - So we need to save **ra**, and the old versions of **s0** and **s1**

Preamble and Postamble

```
foo:  
    addi sp,sp,-12    # Get stack space for 3 registers  
    sw s0,0(sp)        # Save s0 (it is callee saved)  
    sw s1,4(sp)        # Save s1 (it is callee saved)  
    sw ra,8(sp)        # Save ra (it will get overwritten)  
  
{body goes here}    # whole function stuff...  
  
foo_exit:            # Assume return value already in a0  
    lw s0,0(sp)        # Restore Registers  
    lw s1,4(sp)  
    lw ra,8(sp)  
    add sp,sp,12        # Restore stack pointer  
    ret                # aka.. jalr x0 ra
```

And now the body...

```
blt a0,x0,foo_true    # if c < 0, jump to foo_true
foo_false:
                    # this label ends up being ignored but
                    # it is useful documentation
mv s0,a0              # save c in s0
li a0,8                # sizeof(struct node) (pseudoinst)
jal malloc              # call malloc
mv s1,a0              # save n in s1
addi a0,s0,-1          # c-1 in a0
jal foo                # call foo recursively
sw a0,4(s1)             # write the return value into n->next
sb s0,0(s1)             # write c into n->c (just a byte)
mv a0,s1              # return n in a0
j foo_exit
foo_true:
add a0,x0,x0          # return 0 in a0
```

We skipped some possible optimizations ...

- On the leaf node ($c < 0$) we didn't need to save **ra** (or even **s0** & **s1** since we don't need to use them)
- We could get away with only one saved register..
 - Save **c** into **s0**
 - call **malloc**
 - save **c** into **n[0]**
 - calc **c-1**
 - save **n** in **s0**
 - recursive call
- For us, our version is good enough.

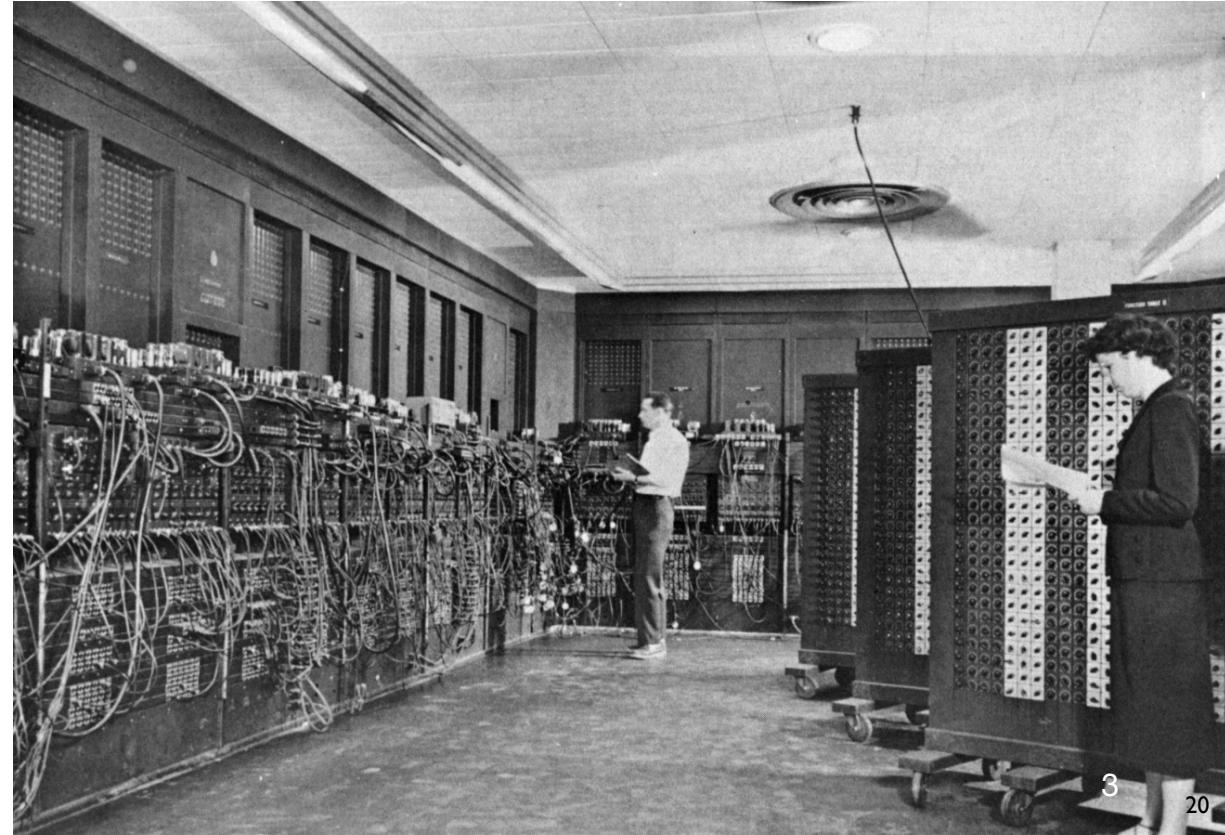
Outline

- More Function Calls
- RISC-V Instruction Formats

ENIAC (U.Penn., 1946)

First Electronic General-Purpose Computer

- Blazingly fast (multiply in 2.8ms!)
 - 10 decimal digits \times 10 decimal digits
- But needed 2-3 days to setup new program, as programmed with patch cords and switches



Big Idea: Stored-Program Computer

- Instructions are represented as bit patterns - can think of these as numbers
- Therefore, entire programs can be stored in memory to be read or written just like data
- Can reprogram quickly (seconds), don't have to rewire computer (days)
- Known as the “von Neumann” computers after widely distributed tech report on EDVAC project
 - Wrote-up discussions of Eckert and Mauchly
 - Anticipated earlier by Turing and Zuse

First Draft of a Report on the EDVAC

by

John von Neumann

Contract No. W-670-ORD-4926

Between the

United States Army Ordnance Department

and the

University of Pennsylvania

Moore School of Electrical Engineering

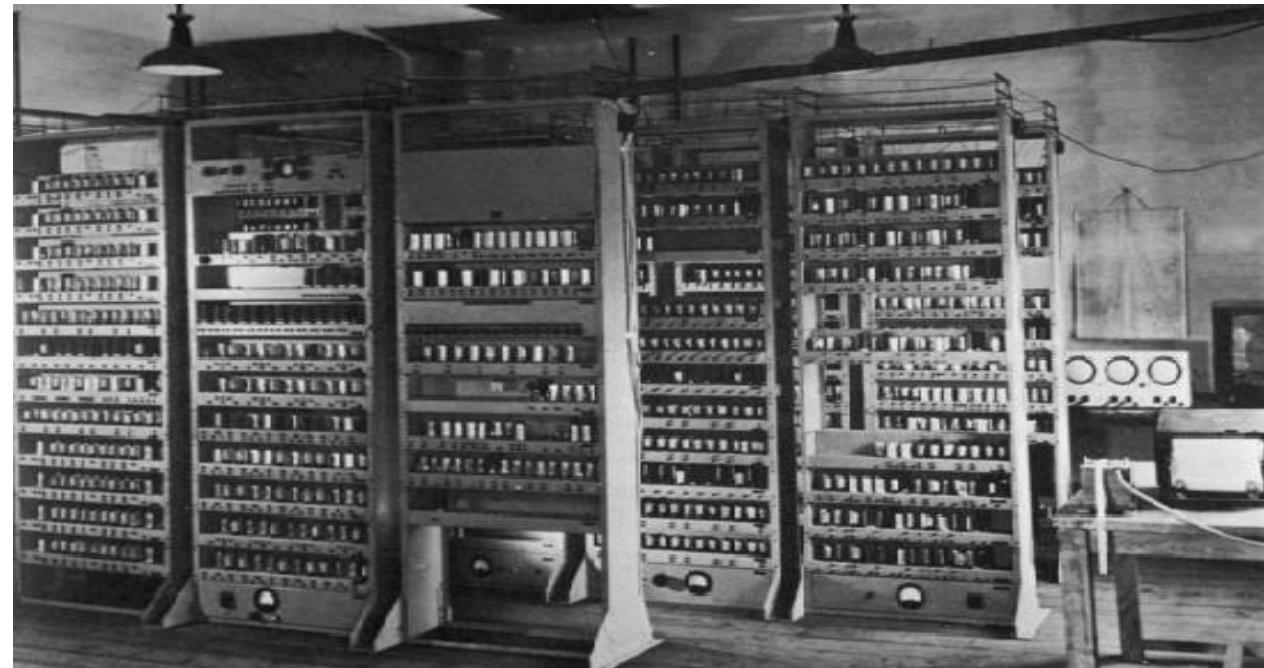
University of Pennsylvania

June 30, 1945

EDSAC (Cambridge, 1949)

First General Stored-Program Computer

- Programs held as “numbers” in memory
- 35-bit binary 2’s complement words



Consequence #1: Everything Has a Memory Address

- Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
 - Both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
- One special register keeps address of instruction being executed: “Program Counter” (PC)
 - Basically a pointer to memory
 - Intel calls it Instruction Pointer (a better name)

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
 - Programs bound to specific instruction set
 - Different version for phones and PCs, etc.
- New machines in the same family want to run old programs (“binaries”) as well as programs compiled to new instructions
- Leads to “backward-compatible” instruction set evolving over time
 - Selection of Intel 8088 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set; could still run program from 1981 PC today

Instructions as Numbers (1/2)

- Most data we work with is in words (32-bit chunks):
 - Each register holds a word
 - `lw` and `sw` both access memory one word at a time
- So how do we represent instructions?
 - Remember: Computer only represents 1s and 0s, so assembler string “`add x10, x11, x0`” is meaningless to hardware
 - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
 - Same 32-bit instruction definitions used for RV32, RV64, RV128

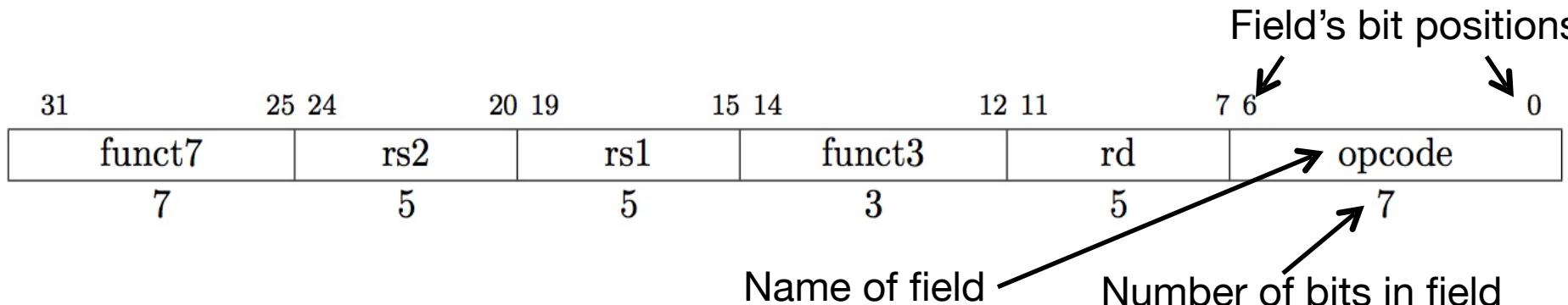
Instructions as Numbers (2/2)

- Divide 32-bit instruction word into “**fields**”
- Each field tells processor something about instruction
- We could define different set of fields for each instruction, but for hardware simplicity, group possible instructions into six basic types of instruction formats:
 - **R-format** for register-register arithmetic/logical operations
 - **I-format** for register-immediate ALU operations and loads
 - **S-format** for stores
 - **B-format** for branches
 - **U-format** for 20-bit upper immediate instructions
 - **J-format** for jumps

Summary of RISC-V Instruction Formats

| 31 | 30 | 25 24 | 21 | 20 | 19 | 15 14 | 12 11 | 8 | 7 | 6 | 0 | |
|----|----|-------|----|----|----|-------|-------|---|---|---|---|--------|
| | | | | | | | | | | | | R-type |
| | | | | | | | | | | | | I-type |
| | | | | | | | | | | | | S-type |
| | | | | | | | | | | | | B-type |
| | | | | | | | | | | | | U-type |
| | | | | | | | | | | | | J-type |

R-Format Instruction Layout Annotation



- This example: 32-bit instruction word divided into six fields of differing numbers of bits each field: $7+5+5+3+5+7 = 32$
- In this case:
 - **opcode** is a 7-bit field that lives in bits 0-6 of the instruction
 - **rs2** is a 5-bit field that lives in bits 20-24 of the instruction
 - etc.

R-Format Instructions opcode/func7 fields

| 31 | 25 24 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|--------|-------|-------|--------|-------|--------|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode | |
| 7 | 5 | 5 | 3 | 5 | 7 | |

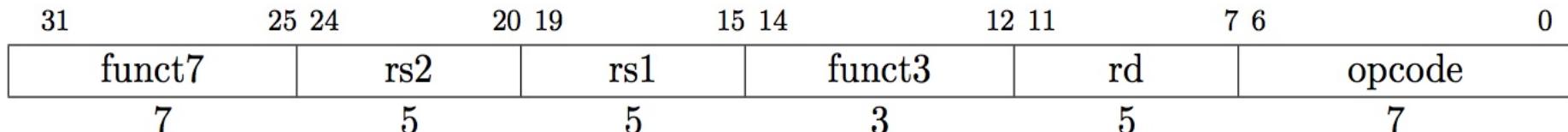
- **opcode**: partially specifies which instruction it is
 - Note: This field is contains 0110011 _{two} for all R-Format register-register arithmetic/logical instructions
- **funct7+funct3**: combined with **opcode**, these two fields describe what operation to perform
- Question: Why aren't **opcode** and **funct7** and **funct3** a single 17-bit field?
 - We'll answer this later

R-Format Instructions register specifiers

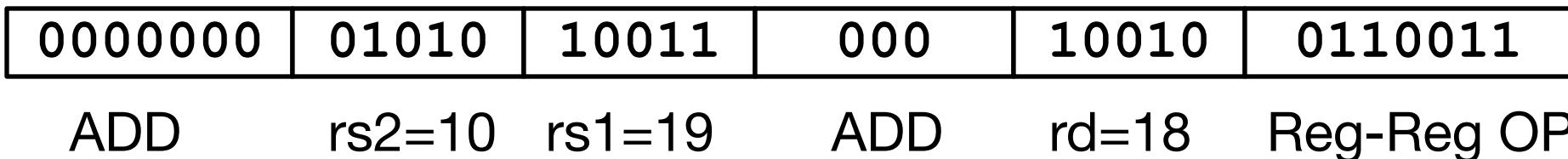
| 31 | 25 24 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|--------|-------|-------|--------|-------|--------|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode | |
| 7 | 5 | 5 | 3 | 5 | 7 | |

- Each register field (rs1, rs2, rd) holds a 5-bit unsigned integer [0-31] corresponding to a register number (x0-x31)
- rs1 (Source Register #1): specifies register containing first operand
- rs2: specifies second register operand
- rd (Destination Register): specifies register which will receive result of computation

R-Format Example



- RISC-V Assembly Instruction:
add x18, x19, x10



All RV32 R-format instructions

| funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---------|-----|-----|--------|----|---------|
| 0000000 | | | 000 | rd | 0110011 |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 |

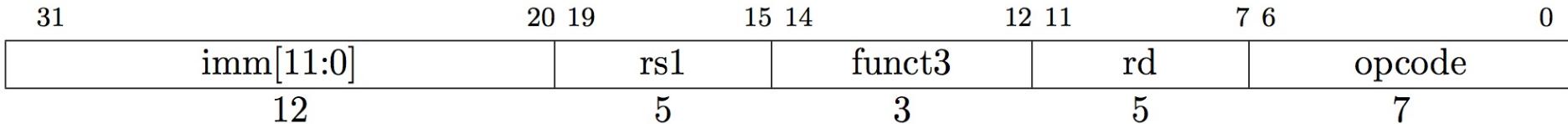


Encoding in funct7 + funct3 selects particular operation

I-Format Instructions

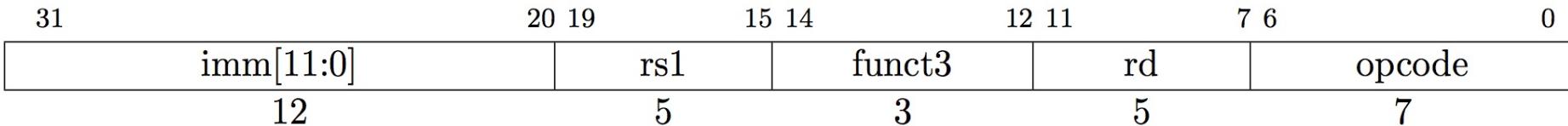
- What about instructions with immediates?
 - Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise
 - 5-bit field only represents numbers up to the value 31: would like immediates to be much larger
- Define another instruction format that is mostly consistent with R-format
 - Note: if instruction has immediate, then uses at most 2 registers (one source, one destination)

I-Format Instruction Layout

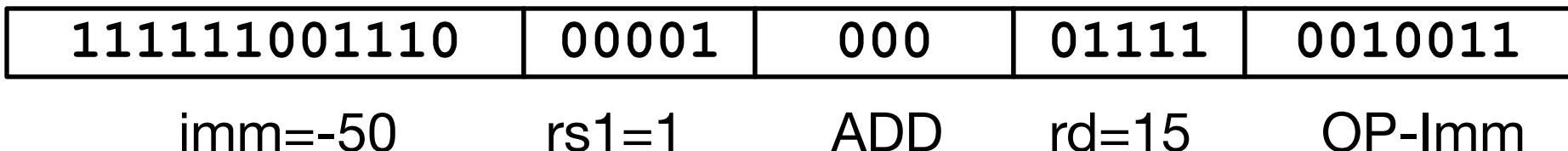


- Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, `imm[11:0]`
- Remaining field format (`rs1`, `funct3`, `rd`, `opcode`) same as before
- `imm[11:0]` can hold values in range $[-2048_{\text{ten}}, +2047_{\text{ten}}]$
- Immediate is **always** sign-extended to 32-bits before use in an arithmetic/logic operation
- We'll later see how to handle immediates > 12 bits

I-Format Example



- RISC-V Assembly Instruction:
addi x15, x1, -50



All RV32 I-format Arithmetic/Logical Instructions

| imm | funct3 | opcode |
|-----------|--------|--------|
| imm[11:0] | rs1 | 000 |
| imm[11:0] | rs1 | 010 |
| imm[11:0] | rs1 | 011 |
| imm[11:0] | rs1 | 100 |
| imm[11:0] | rs1 | 110 |
| imm[11:0] | rs1 | 111 |
| 0000000 | shamt | 001 |
| 0000000 | shamt | 101 |
| 0100000 | shamt | 101 |

One of the higher-order immediate bits is used to distinguish “shift right logical” (SRLI) from “shift right arithmetic” (SRAI)



“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

Load Instructions are also I-Type

$$rd = M[rs1+imm][0:31]$$

| 31 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|--------------------|-----------|------------|-----------|-----------|---|
| imm[11:0] | rs1 | funct3 | rd | opcode | |
| 12 offset[11:0] | 5 base | 3 width | 5 dest | 7 LOAD | |

- The 12-bit signed immediate is added to the base address in register **rs1** to form the memory address
 - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register **rd**

I-Format Load Example

- RISC-V Assembly Instruction:

lw x14, 8(x2)

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

| | | | | |
|--------------|-------|-----|-------|---------|
| 000000001000 | 00010 | 010 | 01110 | 0000011 |
|--------------|-------|-----|-------|---------|

imm=+8

rs1=2

LW

rd=14

LOAD

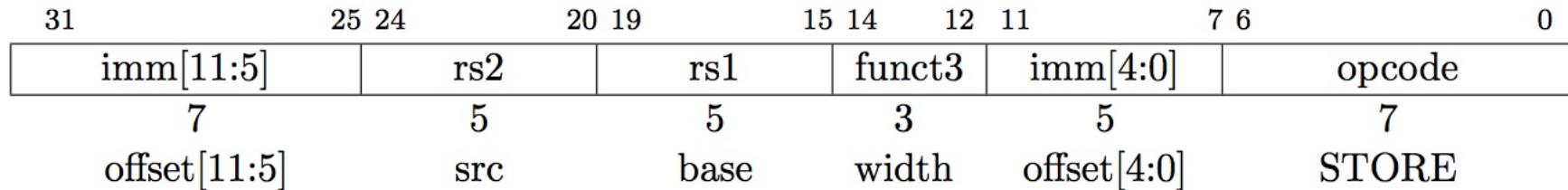
All RV32 Load Instructions

| | | | | | |
|-----------|-----|-----|----|---------|-----|
| imm[11:0] | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | rs1 | 001 | rd | 0000011 | LH |
| imm[11:0] | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:0] | rs1 | 101 | rd | 0000011 | LHU |

funct3 field encodes size and signedness of load data

- LBU is “load unsigned byte”
- LH is “load halfword”, which loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register
- LHU is “load unsigned halfword”, which zero-extends 16 bits to fill destination 32-bit register
- There is no LWU in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register

S-Format Used for Stores



- Store needs to read two registers, **rs1** for base memory address, and **rs2** for data to be stored, as well as need immediate offset!
- Can't have both **rs2** and immediate in same place as other instructions!
- Note that stores don't write a value to the register file, **no rd!**
- RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – *keep rs1/rs2 fields in same place.*

S-Format Example

- RISC-V Assembly Instruction:

sw x14, 8(x2)

| 31 | 25 24 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|-------------------|----------|-----------|------------|------------------|------------|---|
| imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | |
| 7 offset[11:5] | 5 src | 5 base | 3 width | 5 offset[4:0] | 7 STORE | |

| | | | | | |
|---------|-------|-------|-----|-------|---------|
| 0000000 | 01110 | 00010 | 010 | 01000 | 0100011 |
|---------|-------|-------|-----|-------|---------|

offset[11:5] rs2=14 rs1=2 SW offset[4:0] STORE

=0

=8

| | |
|---------|-------|
| 0000000 | 01000 |
|---------|-------|

combined 12-bit offset = 8

All RV32 Store Instructions

| | | | | | | |
|-----------|-----|-----|-----|----------|---------|----|
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |

↑ funct3 field encodes size

RISC-V Conditional Branches

→ jump to relative address of PC + immediate

- E.g., **BEQ x1, x2, Label**
- Branches read two registers but don't write a register (similar to stores)
- How to encode the label, i.e., where to branch to?
- We use an immediate to encode *PC relative offset*
 - If we **don't** take the branch:
 $PC = PC + 4$ (i.e., next instruction)
 - If we **do** take the branch:
 $PC = PC + \text{immediate}$

Branching Instruction Usage

- Branches typically used for loops (if-else, while, for)
 - Loops are generally small (< 50 instructions)
 - Function calls and unconditional jumps handled with jump instructions (J-Format)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
 - Largest branch distance limited by size of code
 - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing

- **PC-Relative Addressing:** Use the **immediate** field as a two's-complement *offset* relative to PC
 - Branches generally change the PC by a small amount
 - With the 12-bit immediate, could specify $\pm 2^{11}$ byte address offset from the PC
- To improve the reach of a single branch instruction, *in principle*, could multiply the byte immediate by 4 before adding to PC (*instructions are 4 bytes and word aligned*).
 - This would allow one branch instruction to reach $\pm 2^{11} \times 32$ -bit instructions either side of PC
 - Four times greater reach than using byte offset
 - However ...

RISC-V Feature, nx16-bit instructions

- However, extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 2-Bytes in length
- To enable this, RISC-V always scales the branch immediate by 2 bytes - even when there are no 16-bit instructions
- This means for us,
 1. the low bit of the stored immediate value will always be 0)
 2. The immediate is left-shifted by 1 before adding to PC
- RISC-V conditional branches can only reach $\pm 2^{10} \times 32$ -bit instructions either side of PC

RISC-V B-Format for Branches

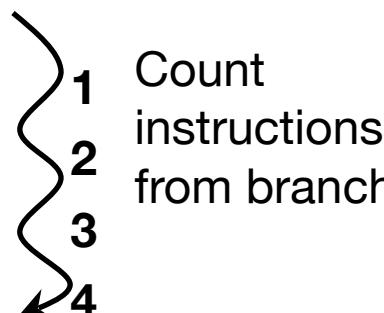
| 31 | 30 | 25 24 | 20 19 | 15 14 | 12 11 | 8 | 7 | 6 | 0 |
|---------|-----------|-------|-------|--------|----------|---------|---|--------|---|
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | | opcode | |
| 1 | 6 | 5 | 5 | 3 | 4 | 1 | | 7 | |

- B-format is mostly same as S-Format, with two register sources (**rs1/rs2**) and a 12-bit immediate
- But now immediate represents the **branch offset in units of half-words**. To convert to units of Bytes, left-shift by 1.
- The 12 immediate bits encode **even 13-bit signed byte offsets** (**lowest bit of offset is always zero, so no need to store it**)
 - Thus the **imm[12:1]** in the total encoding, compared with **imm[11:0]** in the I-type encodings

Branch Example, determine offset

- RISC-V Code:

```
Loop: beq x19,x10,End
      add x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:  # target instruction
```



1 Count
2 instructions
3 from branch
4

- Branch offset = **4x32-bit instructions = 16 bytes**
- (Branch with offset of 0, branches to itself)

Branch Example, encode offset

- RISC-V Code:

```
Loop: beq x19,x10,End
      add x18,x18,x10
      addi x19,x19,-1
      j Loop
End: # target instruction
```

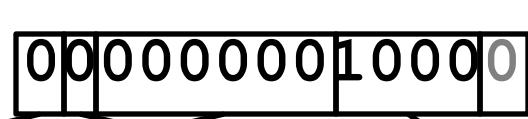
offset = 16 bytes =
8x2

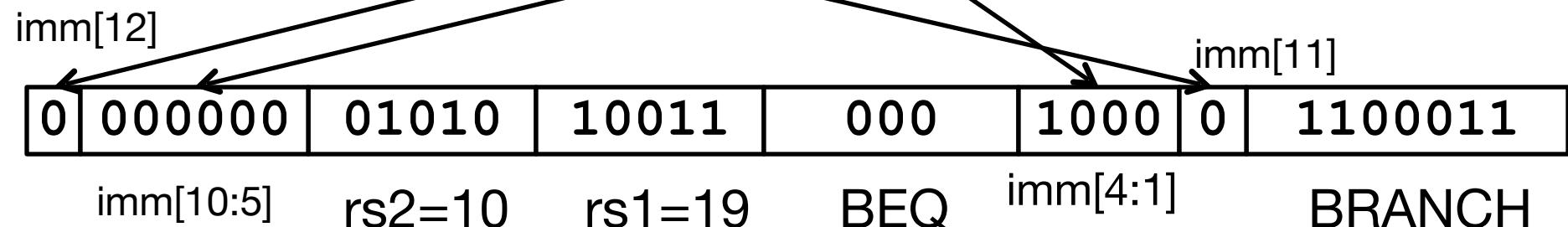
| | | | | | |
|---------|--------|--------|-----|-------|---------|
| ??????? | 01010 | 10011 | 000 | ????? | 1100011 |
| imm | rs2=10 | rs1=19 | BEQ | imm | BRANCH |

Branch Example, complete encoding

beq x19, x10, offset = 16 bytes

13-bit immediate, imm[12:0], with value 16

 imm[0] discarded,
always zero



All RISC-V Branch Instructions

| | | | | | | |
|--------------|-----|-----|-----|-------------|---------|------|
| imm[12 10:5] | rs2 | rs1 | 000 | imm[4:1 11] | 1100011 | BEQ |
| imm[12 10:5] | rs2 | rs1 | 001 | imm[4:1 11] | 1100011 | BNE |
| imm[12 10:5] | rs2 | rs1 | 100 | imm[4:1 11] | 1100011 | BLT |
| imm[12 10:5] | rs2 | rs1 | 101 | imm[4:1 11] | 1100011 | BGE |
| imm[12 10:5] | rs2 | rs1 | 110 | imm[4:1 11] | 1100011 | BLTU |
| imm[12 10:5] | rs2 | rs1 | 111 | imm[4:1 11] | 1100011 | BGEU |

Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
 - If moving individual lines of code, then yes
 - If moving all of code, then no (because PC-relative offsets)
 - What do we do if destination is $> 2^{10}$ instructions away from branch?

U-Format for “Upper Immediate” instructions

| 31 | 12 11 | 7 6 | 0 |
|--------------------|-------|--------|---|
| imm[31:12] | rd | opcode | |
| 20 | 5 | 7 | |
| U-immediate[31:12] | dest | LUI | |
| U-immediate[31:12] | dest | AUIPC | |

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
 - LUI – Load Upper Immediate, $rd = imm \ll 12$
 - AUIPC – Add Upper Immediate to PC, $rd = PC + (imm \ll 12)$

LUI to create long immediates

- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

```
LUI x10, 0x87654      # x10 = 0x87654000
```

```
ADDI x10, x10, 0x321 # x10 = 0x87654321
```

One Corner Case

How to set 0xDEADBEEF?

LUI x10, 0xDEADB # x10 = 0xDEADB000

ADDI x10, x10, 0xEEF # x10 = 0xDEADAEEF

1101 1110 1010 1101 1011
↓
10 1110 1110 1111

ADDI 12-bit immediate is always sign-extended, if top bit is set,
will subtract -1 from upper 20 bits

↓

Preserves sign in two's complements

Basically extends MSB to all
new spaces created.

Solution

How to set 0xDEADBEEF?

```
LUI x10, 0xDEADC      # x10 = 0xDEADC000
ADDI x10, x10, 0xEEF # x10 = 0xDEADBEEF
```

Pre-increment the value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

Assembler pseudo-op handles all of this:

```
li x10, 0xDEADBEEF # Creates two instructions
```

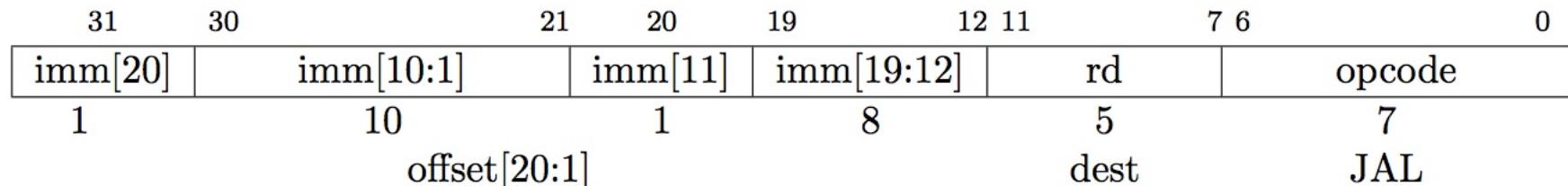
Uses of JAL

```
# j pseudo-instruction
j Label = jal x0, Label # Discard return address

# Call function within 218 instructions of PC
jal ra, FuncName
```

J-Format for Jump Instructions

- **JAL saves PC+4 in register rd (the return address)**
 - Assembler “j” jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
 - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost



Uses of JALR

```
# ret and jr psuedo-instructions
ret  =  jr ra  =  jalr x0, ra, 0
      ↓      ↓      ↗ offset
      rd     ra
# Call function at any 32-bit absolute address
lui  x1, <hi20bits>
jalr ra, x1, <lo12bits>

# Jump PC-relative with 32-bit offset
auipc x1, <hi20bits>      # Adds upper immediate value to
                             # and places result in x1
jalr x0, x1, <lo12bits>    # Same sign extension trick needed
                             # as LUI
```

JALR Instruction (I-Format)

| 31 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|--------------|-------|--------|-------|--------|---|
| imm[11:0] | rs1 | funct3 | rd | opcode | |
| 12 | 5 | 3 | 5 | 7 | |
| offset[11:0] | base | 0 | dest | JALR | |

- **JALR rd, rs, immediate**
 - Writes PC+4 to **rd** (return address)
 - Sets PC = **rs + immediate**
 - Uses same immediates as arithmetic and loads
 - **no** multiplication by 2 bytes

Summary of RISC-V Instruction Formats

| 31 | 30 | 25 24 | 21 | 20 | 19 | 15 14 | 12 11 | 8 | 7 | 6 | 0 | |
|----|---------|-----------|------------|-----|---------|-------|------------|--------|----------|----------|--------|--------|
| | funct7 | | | rs2 | | rs1 | | funct3 | | rd | | opcode |
| | | | | | | rs1 | | funct3 | | rd | | opcode |
| | | imm[11:0] | | | | rs1 | | funct3 | | rd | | opcode |
| | | imm[11:5] | | rs2 | | rs1 | | funct3 | | imm[4:0] | | opcode |
| | imm[12] | imm[10:5] | | rs2 | | rs1 | | funct3 | imm[4:1] | imm[11] | | opcode |
| | | | imm[31:12] | | | | | | rd | | opcode | U-type |
| | imm[20] | | imm[10:1] | | imm[11] | | imm[19:12] | | | rd | | opcode |
| | | | | | | | | | | | | J-type |

Complete RV32I ISA

| | | | | |
|-----------------------|-----|-----|-----|-------------|
| imm[31:12] | | | rd | 0110111 |
| imm[31:12] | | | rd | 0010111 |
| imm[20:10:1 11 19:12] | | | rd | 1101111 |
| imm[11:0] | | rs1 | 000 | rd |
| imm[12:10:5] | rs2 | rs1 | 000 | imm[4:1 11] |
| imm[12:10:5] | rs2 | rs1 | 001 | imm[4:1 11] |
| imm[12:10:5] | rs2 | rs1 | 100 | imm[4:1 11] |
| imm[12:10:5] | rs2 | rs1 | 101 | imm[4:1 11] |
| imm[12:10:5] | rs2 | rs1 | 110 | imm[4:1 11] |
| imm[12:10:5] | rs2 | rs1 | 111 | imm[4:1 11] |
| imm[11:0] | | rs1 | 000 | rd |
| imm[11:0] | | rs1 | 001 | rd |
| imm[11:0] | | rs1 | 010 | rd |
| imm[11:0] | | rs1 | 100 | rd |
| imm[11:0] | | rs1 | 101 | rd |
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] |
| imm[11:0] | | rs1 | 000 | rd |
| imm[11:0] | | rs1 | 010 | rd |
| imm[11:0] | | rs1 | 011 | rd |
| imm[11:0] | | rs1 | 100 | rd |
| imm[11:0] | | rs1 | 110 | rd |
| imm[11:0] | | rs1 | 111 | rd |

LUI
 AUIPC
 JAL
 JALR
 BEQ
 BNE
 BLT
 BGE
 BLTU
 BGEU
 LB
 LH
 LW
 LBU
 LHU
 SB
 SH
 SW
 ADDI
 SLTI
 SLTIU
 XORI
 ORI
 ANDI

| | | | | | |
|------------------|-------|-------|-------|---------|---------|
| 0000000 | shamt | rs1 | 001 | rd | 0010011 |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 |
| 0100000 | shamt | rs1 | 101 | rd | 0010011 |
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 |
| 0000 | pred | succ | 00000 | 000 | 00000 |
| 0000 | 0000 | 0000 | 00000 | 001 | 00000 |
| 0000000000000000 | | 00000 | 000 | 00000 | 1110011 |
| 00000000000001 | | 00000 | 000 | 00000 | 1110011 |
| csr | rs1 | 001 | rd | 1110011 | |
| csr | rs1 | 010 | rd | 1110011 | |
| csr | rs1 | 011 | rd | 1110011 | |
| csr | zimm | 101 | rd | 1110011 | |
| csr | zimm | 110 | rd | 1110011 | |
| csr | zimm | 111 | rd | 1110011 | |

Not in CS61C

| 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 |
| f0–7 | ft0–7 | FP temporaries | Caller |
| f8–9 | fs0–1 | FP saved registers | Callee |
| f10–11 | fa0–1 | FP arguments/return values | Caller |
| f12–17 | fa2–7 | FP arguments | Caller |
| f18–27 | fs2–11 | FP saved registers | Callee |
| f28–31 | ft8–11 | FP temporaries | Caller |

Table 18.2: RISC-V calling convention register usage.

Immediate storage.

| | | | | | | | | | | |
|----------|-------------|-------------|-------------|----------|-------------|-------------|----------|--------------|---|--|
| 31 | 30 | 20 19 | 12 | 11 | 10 | 5 | 4 | 1 | 0 | |
| — | inst[31] | — | | | inst[30:25] | inst[24:21] | inst[20] | I-immediate | | |
| — | inst[31] | — | | | inst[30:25] | inst[11:8] | inst[7] | S-immediate | | |
| — | inst[31] | — | | inst[7] | inst[30:25] | inst[11:8] | 0 | SB-immediate | | |
| inst[31] | inst[30:20] | inst[19:12] | | — 0 — | | | | U-immediate | | |
| — | inst[31] | — | inst[19:12] | inst[20] | inst[30:25] | inst[24:21] | 0 | UJ-immediate | | |