

CS 110

Computer Architecture

Lecture 6:

RISC-V Instruction Formats

Instructors:

Sören Schwertfeger & Chundong Wang

<https://robotics.shanghaitech.edu.cn/courses/ca/20s/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C



Admin

- Add/ drop is over =>
 - Participation will be checked more rigorous now!
- Midterm I will be canceled.
- Adjustment of grading scaling
 - Still might change in the future!



Course Grading

- Projects: 30% (-3%)
- Homework: 15% (-2%)
- Lab: 5%
- Exams: 30%
 - ~~Midterm 1: 10% ?~~ (-10%)
 - Midterm 2: 10%
 - Final: 20%
- Participation: 10% (+5%)
- Quizzes: 10% (+10%)



Participation

- Participation: 10%
 - Maybe each item 2%:
 - Video Lecture Poll Participation
 - Online Lecture Participation (join zoom lecture)
 - Online Lecture Quiz Participation
 - Lab, Homework, Project Attendance
 - Piazza statistics:

Student Participation Report

Name, Email	days online	posts viewed*	contributions**
	37	160	20
	32	159	30
	25	159	55



Quiz

- Quizzes: 10%
 - Video Lecture piazza polls (maybe 3%)
 - Online Lecture Quizzes (maybe 7%)
- Online Lecture Quizzes Grading:
 - Graded generously
 - Test of paying attention & understanding basic concepts
 - Submitting (almost) empty solution 15 minutes before deadline will get 0 points.



Admin

- Head TA Yanjie Song keeps track of students with technical difficulties – contact him if there are such problems! We will find a solution.
- HW3 is published – due March 27 – start early!
- Project 1.1 will be published today!
- Never share your code with anybody!

RISC-V ISA so far...

- Registers we know so far (All of them!)
 - a0-a7 for function arguments, a0-a1 for return values
 - sp, stack pointer, ra return address
 - s0-s11 saved registers
 - t0-t6 temporaries
 - zero
- Instructions we know:
 - Arithmetic: add, addi, sub
 - Logical: sll, srl, slli, srli, slai, and, or, xor, andi, ori, xori
 - Decision: beq, bne, blt, bge
 - Unconditional branches (jumps): j, jr
 - Functions called with **jal**, return with **jr ra**.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!

12 Shift Instructions...

- Two versions of all shift instructions. Shift amount via:
 - Register
 - Immediate
- (On RV64: additional “word” version of instruction: only works on first 32bit of 64bit register)
- Shift Left
- Shift Right Arithmetic: Fill upper bits with **msb**
- Shift Right Logic: Fill upper bits with 0's

sll, sllw	R	Shift Left (Word)	$R[rd] = R[rs1] \ll R[rs2]$	1)
slli, slliw	I	Shift Left Immediate (Word)	$R[rd] = R[rs1] \ll imm$	1)
sra, sraw	R	Shift Right Arithmetic (Word)	$R[rd] = R[rs1] \gg R[rs2]$	1,5)
srai, srailw	I	Shift Right Arith Imm (Word)	$R[rd] = R[rs1] \gg imm$	1,5)
srl, srlw	R	Shift Right (Word)	$R[rd] = R[rs1] \gg R[rs2]$	1)
srli, srliw	I	Shift Right Immediate (Word)	$R[rd] = R[rs1] \gg imm$	1)

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift

Frame Pointer!?

- As a reminder, we shove all the C local variables etc. on the stack...
 - Combined with space for all the saved registers
 - This is called the "activation record" or "call frame" or "call record"
- But a naive compiler may cause the stack pointer to bounce up and down during a function call
 - Can be a lot simpler to have a compiler do a bunch of pushes and pops when it needs a bit of temporary space: more so on a CISC rather than a RISC however
- Plus: not all programming languages can store all activation records on the stack:
 - The use of lambda in Scheme, Python, Go, etc. requires that some call frames are allocated on the heap since variables may last beyond the function call!

Convention: Use **s0** as a Frame Pointer (**fp**)

- At the start, save **s0 (x8)** and then have the Frame pointer point to one below the **sp** when you were called...

```
addi sp sp -20 # Initially grabbing 5 words of space
sw ra 16(sp)   #
sw fp 12(sp)   # save fp/s0/x8
addi fp sp 20  # Points to the start of this call record
...
```

- Now we can address local variables off the frame pointer rather than the stack pointer
 - Simplifies the compiler
 - Since it can now move the stack up and down easily
 - Simplifies the *debugger*

But note...

- It isn't necessary in C...
 - Most C compilers has a `-f-omit-frame-pointer` option on most architectures
 - It just fubars debugging a bit
- So for our hand-written assembly, we will generally ignore the frame pointer
- The calling convention says it doesn't matter if you use a frame pointer or not!
 - It is just a callee saved register, so if you use it as a frame pointer...
It will be preserved just like any other saved register
But if you just use it as `$0`, that makes no difference!

The Stack Is Also For Local Variables...

- e.g. **char[20] foo;**
- Requires enough space on the stack
 - May need padding
- So then to pass **foo** to something in **a0...**

```
addi a0 sp offset-for-foo-off-sp
addi a0 fp offset-for-foo-off-fp
```

 - If you are using the frame pointer...

The Stack Is Also For Arguments

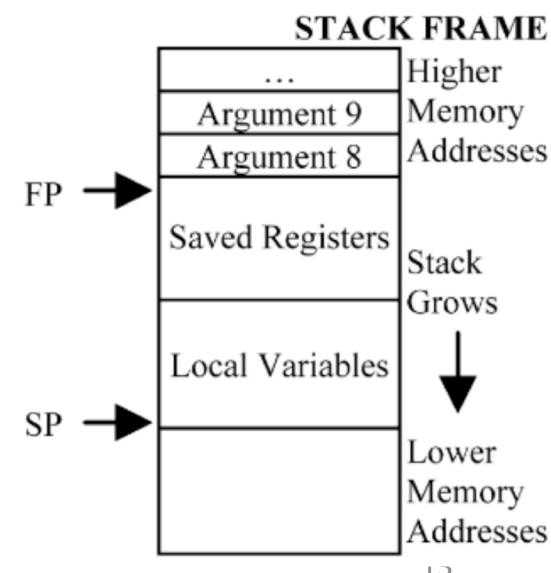
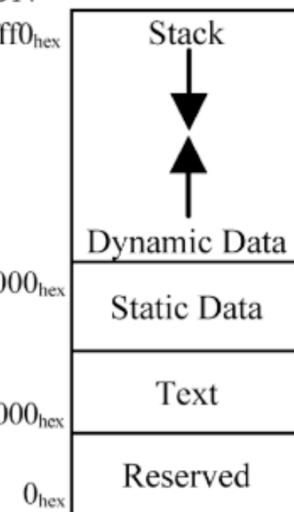
- Arguments 1-8 are passed in **a0-a7**
- But what about a 9th argument or more?
- But what about complex structs as arguments?
 - Pass those on the stack!
 - When the function is called,
 - **0(sp)** → arg #9
 - **4(sp)** → arg #10...
- **ALWAYS** keep sp the lowest address used!
 - Because: Interrupts may use your stack!
 - => Arguments are in the frame of the caller!
- Don't need to memorize this for exams

MEMORY ALLOCATION
SP → 0000 003f ffff fff0_{hex}

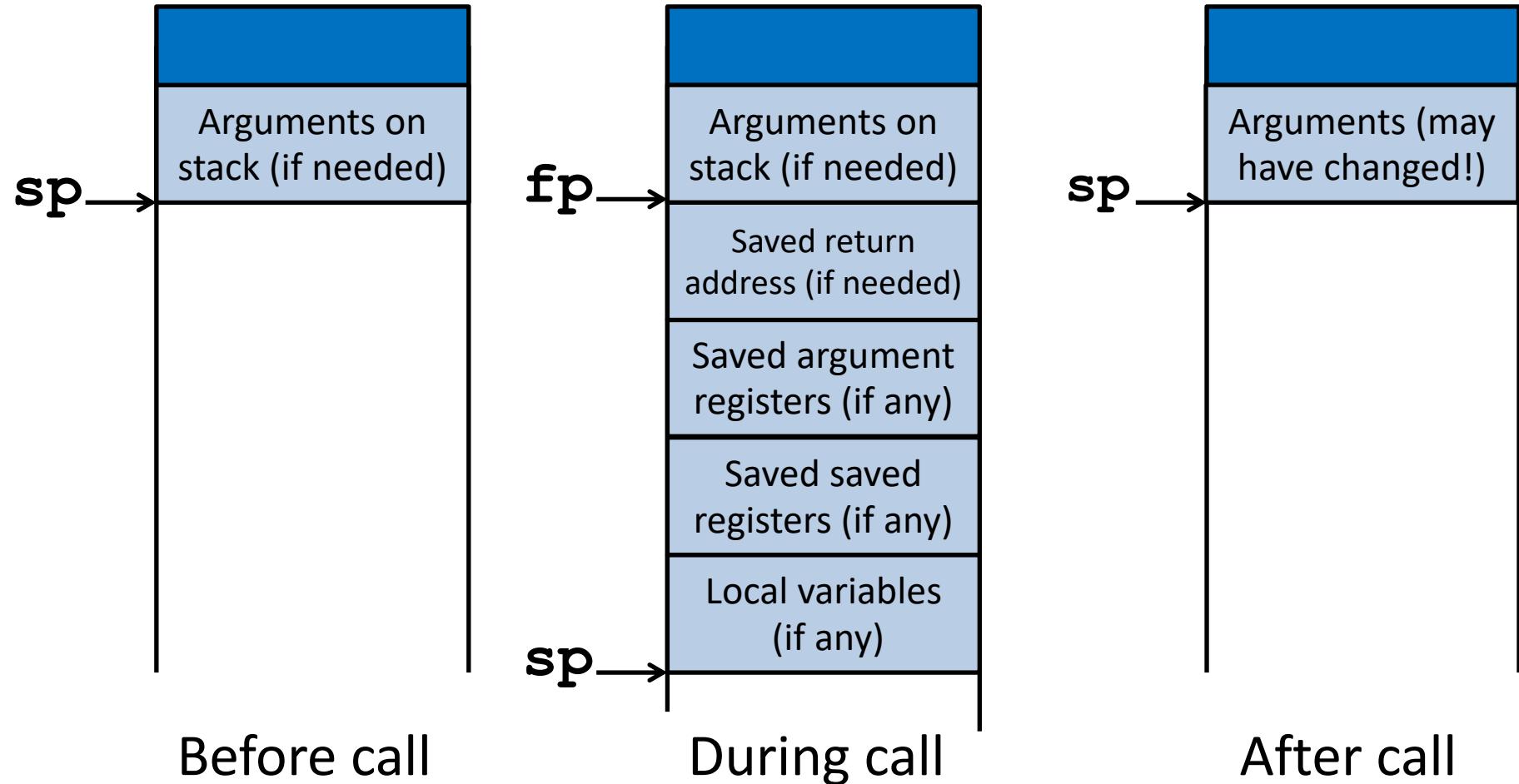
0000 0000 1000 0000_{hex}

PC → 0000 0000 0040 0000_{hex}

0_{hex}



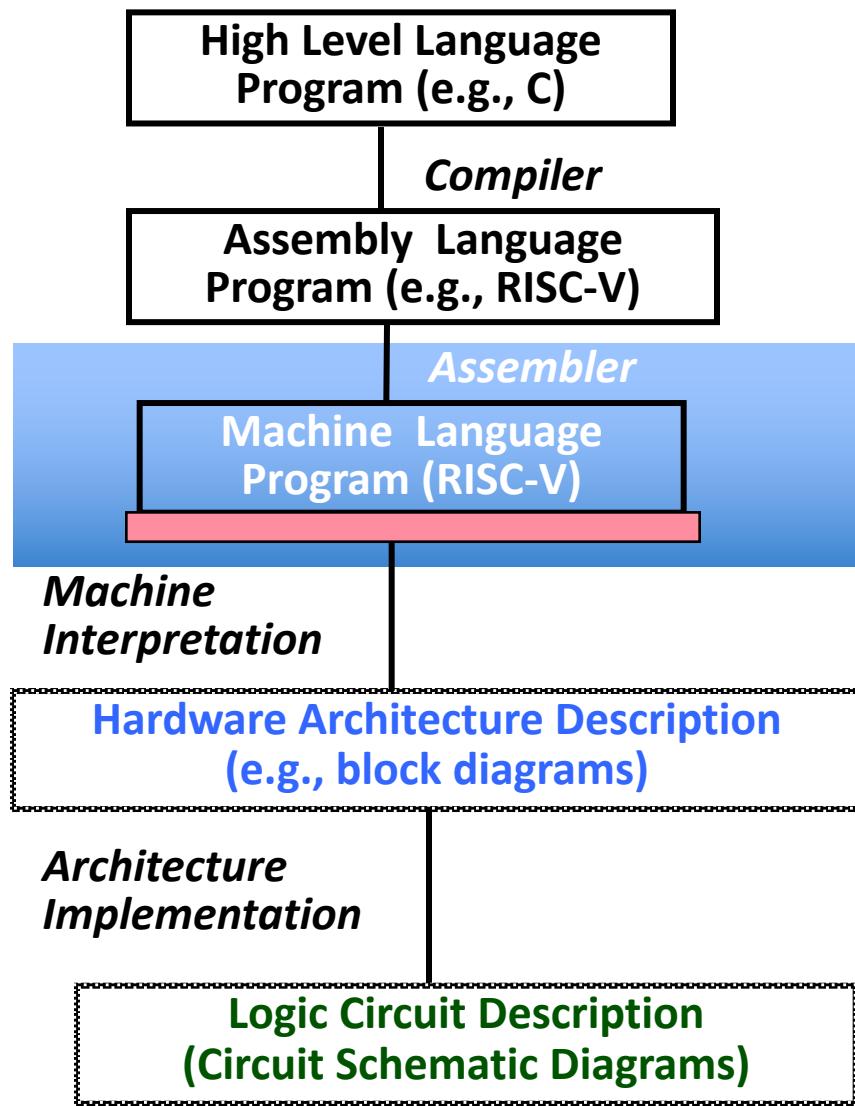
Stack Before, During, After Function



Register Allocation

- We have some set of registers that are useful for local variables, temporaries that last across function calls, etc...
- We have some other set of registers that are just for temporary use
- Which ones do we use? What do we instead save on the stack?
- This is the "Register Allocation" problem
 - Experience it in great detail in CS 131 Compilers ...
- Can either be trivial or NP-complete!

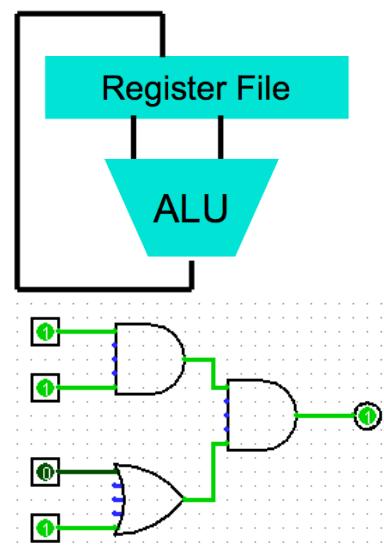
Levels of Representation/Interpretation



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

`lw xt0, 0(x2)
lw xt1, 4(x2)
sw xt1, 0(x2)
sw xt0, 4(x2)`

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111



Big Idea: Stored-Program Computer

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

- 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

Consequence #1: Everything Addressed

- 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
 - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limited in Java by language design
- One 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 **ARM** (phone) and **PCs**
- New machines 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 8086 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)

- Currently most data we work with is in words (32-bit chunks):
 - Each register is a word.
 - **lw** and **sw** both access memory one word at a time.
- So how do we represent instructions?
 - Remember: Computer only understands 1s and 0s, so “**add x10, x11, x0**” is meaningless.
 - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words, too
 - Same 32-bit instructions used for RV32, RV64, RV128

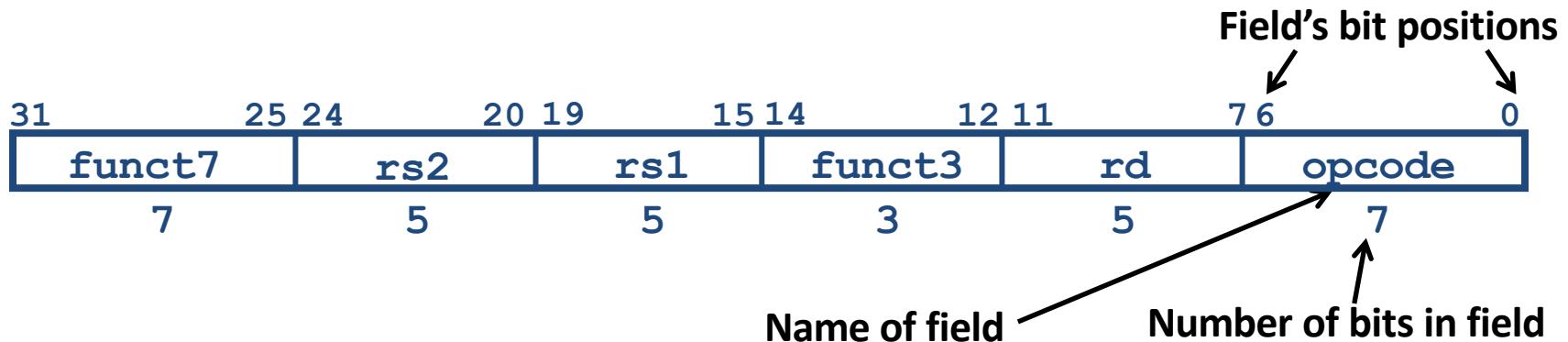
Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into “**fields**”.
- Each field tells processor something about instruction.
- We could define different fields for each instruction, but RISC-V seeks simplicity, so define 6 basic types of instruction formats:
 - R-format for register-register arithmetic operations
 - I-format for register-immediate arithmetic operations and loads
 - S-format for stores
 - B-format for branches (minor variant of S-format, called SB before)
 - U-format for 20-bit upper immediate instructions
 - J-format for jumps (minor variant of U-format, called UJ before)

Summary of RISC-V Instruction Formats

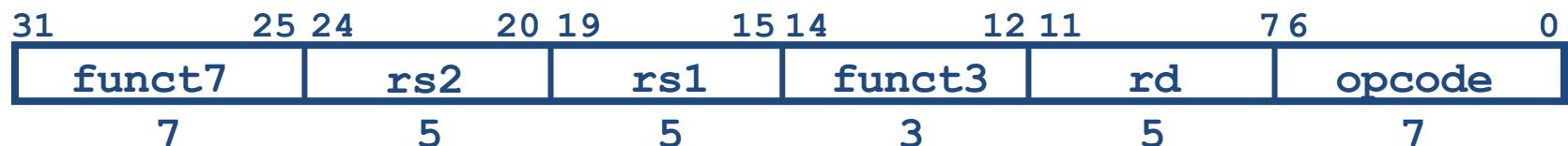
31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
															R-type
funct7		rs2		rs1		funct3			rd		opcode				I-type
imm[11:0]			rs1		funct3			rd		opcode					S-type
imm[11:5]		rs2		rs1		funct3	imm[4:0]			opcode					B-type
imm[12 10:5]		rs2		rs1		funct3	imm[4:1 11]		imm[11:12]		opcode				U-type
imm[20 10:1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

R-Format Instruction Layout



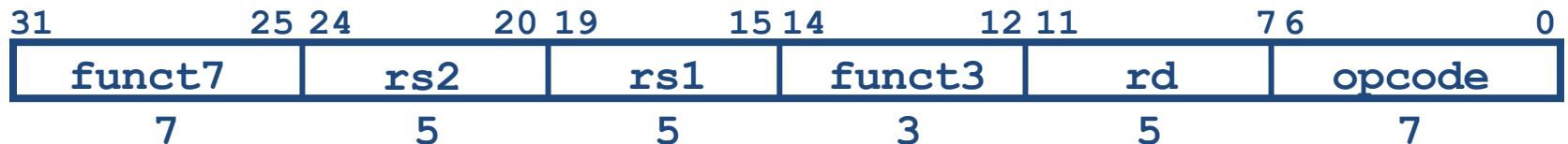
- 32-bit instruction word divided into six fields of varying numbers of bits each: $7+5+5+3+5+7 = 32$
- Examples
 - **opcode** is a 7-bit field that lives in bits 6-0 of the instruction
 - **rs2** is a 5-bit field that lives in bits 24-20 of the instruction

R-Format Instructions opcode/funct fields



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

R-Format Instructions register specifiers

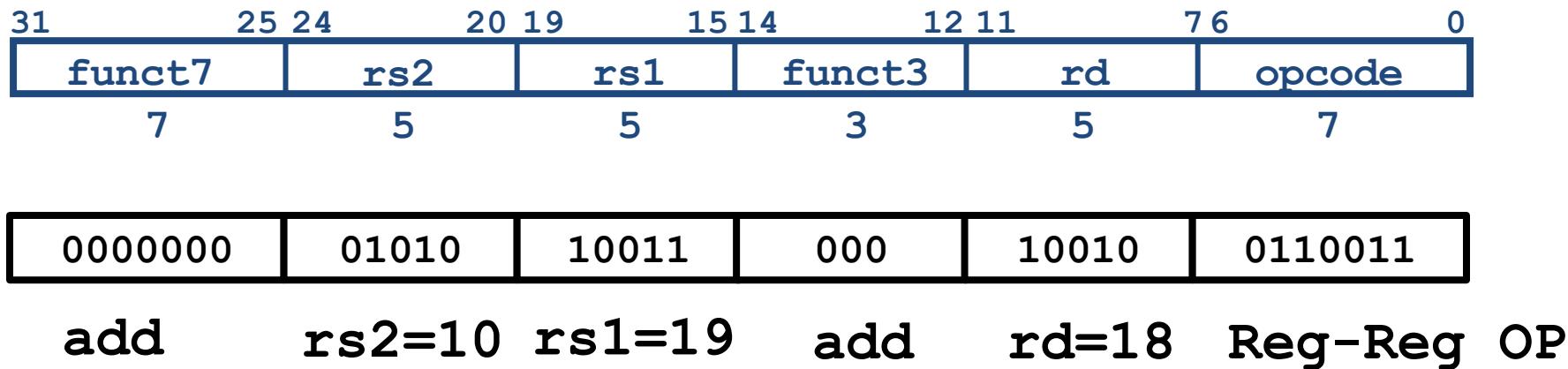


- **rs1** (Source Register #1): specifies register containing first operand
- **rs2**: specifies second register operand
- **rd** (Destination Register): specifies register which will receive result of computation
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register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

R-Format Example

- RISC-V Assembly Instruction:

add x18, x19, x10



All RV32 R-format instructions

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

Different encoding in funct7 + funct3 selects different operations

Question

- What is correct encoding of add x4, x3, x2 ?

A: 4021 8233_{hex}

B: 0021 82b3_{hex}

C: 4021 82b3_{hex}

D: 0021 8233_{hex}

E: 0021 8234_{hex}

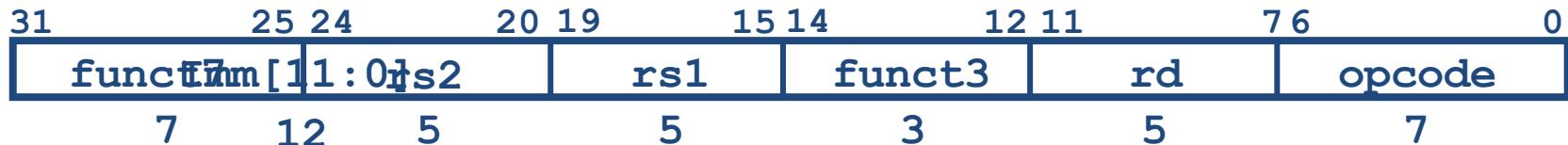


31	25 24	20 19	15 14	12 11	7 6	0	
0000000	rs2	rs1	000	rd	0110011		add
0100000	rs2	rs1	000	rd	0110011		sub
0000000	rs2	rs1	100	rd	0110011		xor
0000000	rs2	rs1	110	rd	0110011		or
0000000	rs2	rs1	111	rd	0110011		and

I-Format Instructions

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

I-Format Instruction Layout

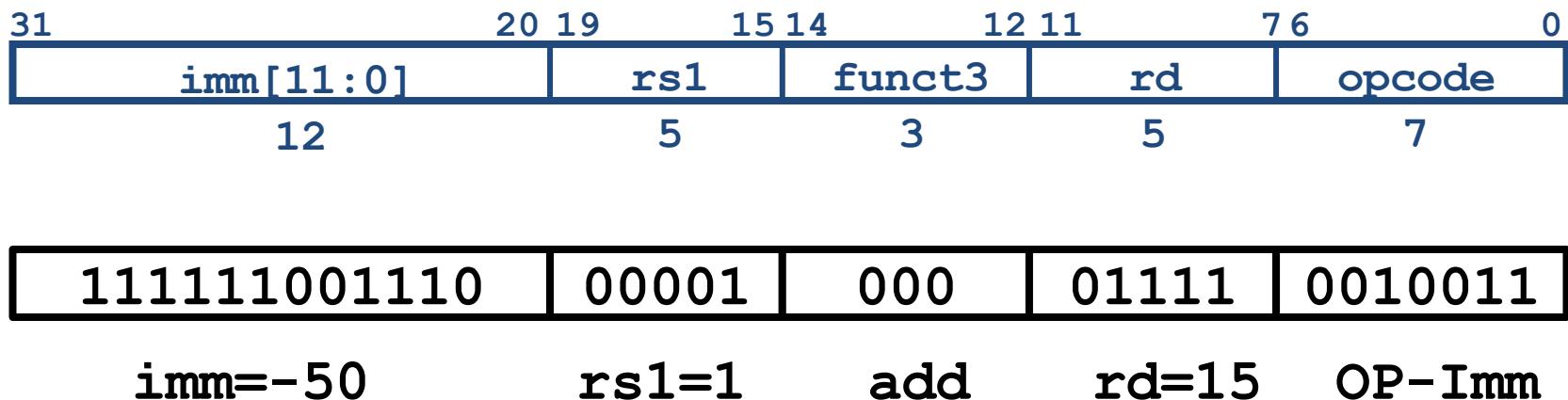


- Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, **imm[11:0]**
- Remaining fields (rs1, funct3, rd, opcode) same as before
- imm[11:0] can hold values in range $[-2048_{\text{ten}}, +2047_{\text{ten}}]$
- Immediate is always sign-extended to 32-bits before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

I-Format Example

- RISC-V Assembly Instruction:

addi x15, x1, -50



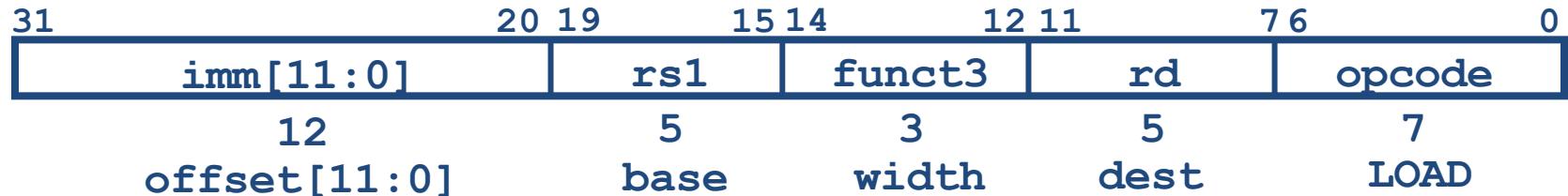
All RV32 I-format Arithmetic Instructions

imm[11:0]	rs1	000	rd	0010011	addi
imm[11:0]	rs1	010	rd	0010011	slti
imm[11:0]	rs1	011	rd	0010011	sltiu
imm[11:0]	rs1	100	rd	0010011	xori
imm[11:0]	rs1	110	rd	0010011	ori
imm[11:0]	rs1	111	rd	0010011	andi
0000000	shamt	rs1	001	rd	slli
0000000	shamt	rs1	101	rd	srl
0100000	shamt	rs1	101	rd	srai

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

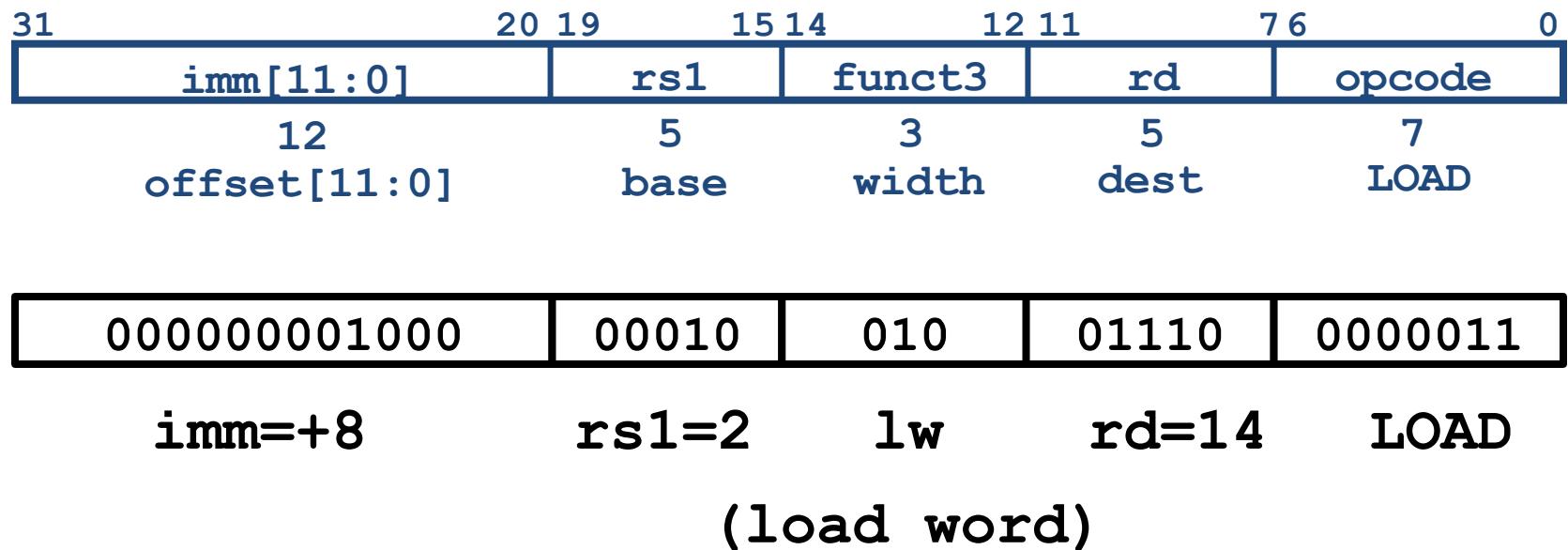


- The 12-bit signed immediate is added to the base address in register **rs1** to form the memory address
 - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register rd

I-Format Load Example

- RISC-V Assembly Instruction:

lw x14, 8(x2)



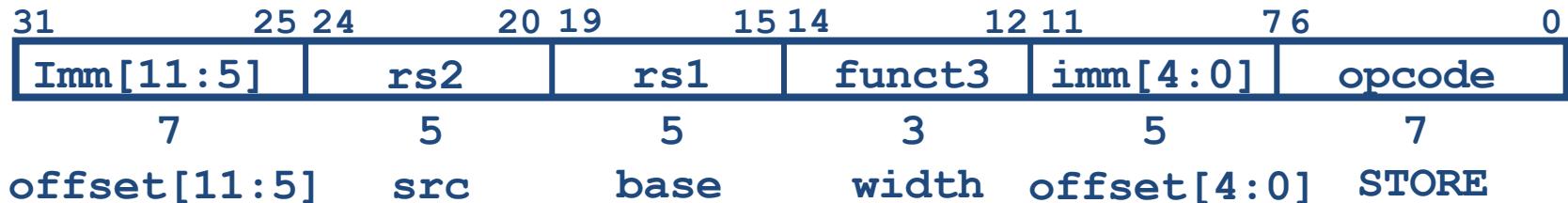
All RV32 Load Instructions

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 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
 - register names more critical than immediate bits in hardware design

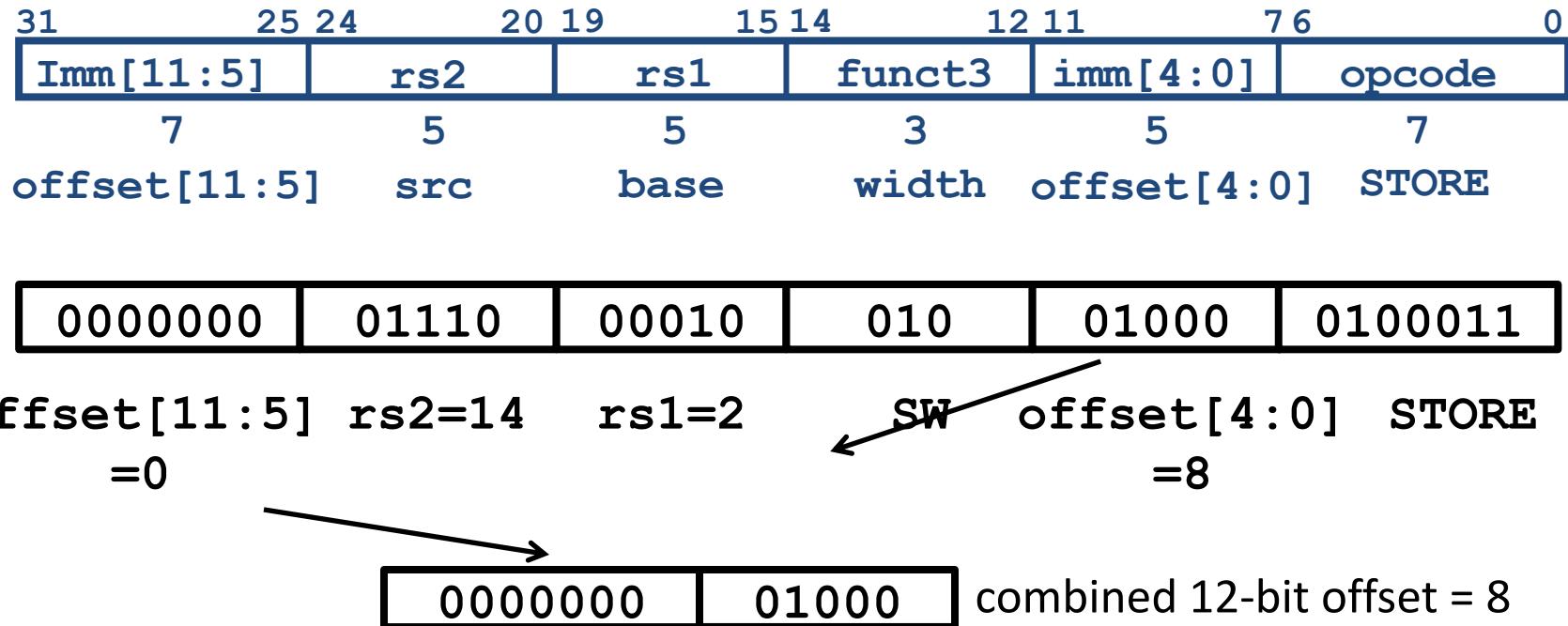
Keeping Registers always in the Same Place...

- The critical path for *all operations* includes fetching values from the registers
- By always placing the read sources in the same place, the register file can read without hesitation
 - If the data ends up being unnecessary (e.g. I-Type), it can be ignored
- Other RISCs have had slightly different encodings
 - Necessitating the logic to look at the instruction to determine which registers to read
- Example of one of the (many) little tweaks done in RISC-V to make things work better

S-Format Example

- RISC-V Assembly Instruction:

sw x14, 8(x2)



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

width

- Store byte, halfword, word



Q & A



Very nice Venus Tutorial

Ze Song

Will be available shortly after the lecture.



Quiz

Prepare for another Programming
PDF Quiz on Thursday!

Translate Machine Instruction to RISC-V Assembly



Piazza: "Online Lecture 6 Quiz"

Select ALL Assembly instructions that produce this machine instruction!

• **1,074,332,851_{ten}**

1. add s4 x18 zero
2. sub s4 s2 x0
3. add s1 x18 zero
4. sub s1 s2 x0
5. neg s4 x18
6. neg s1 s2
7. mv s4 x18
8. mv s1 s2

• **0x FF F3 43 13**

9. For 13-18 : instead of t0 use t1
10. For 13-18 : instead of t0 use t2
11. For 13-18 : instead of t0 use s0
12. For 13-18 : instead of t0 use s1
13. xor t0 t0 -1
14. xori t0 t0 -1
15. xor t0 t0 0xFFFF
16. xori t0 t0 0xFFFF
17. neg t0 t0
18. not t0 t0

CS 110

Computer Architecture

Lecture 6:

Branch Formats

Video 2

Instructors:

Sören Schwertfeger & Chundong Wang

<https://robotics.shanghaitech.edu.cn/courses/ca/20s/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

RISC-V Conditional Branches

- E.g., **BEQ x1, x2, Label**
- Branches read two registers but don't write a register (similar to stores)
- How to encode label, i.e., where to branch to?

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 to PC
 - Branches generally change the PC by a small amount
 - Can specify $\pm 2^{11}$ ‘unit’ addresses from the PC
- Why not use byte as a unit of offset from PC?
 - Because instructions are 32-bits (4-bytes)
 - We don’t branch into middle of instruction

Scaling Branch Offset

- One idea: To improve the reach of a single branch instruction, multiply the offset by four bytes before adding to PC
- 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

RISC-V Feature, n×16-bit instructions

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length
- To enable this, RISC-V scales the branch offset by 2 bytes even when there are no 16-bit instructions
- Reduces branch reach by half and means that $\frac{1}{2}$ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach $\pm 2^{10} \times 32\text{-bit instructions}$ on either side of PC

Branch Calculation

- If we **don't** take the branch:

$PC = PC + 4$ (i.e., next instruction)

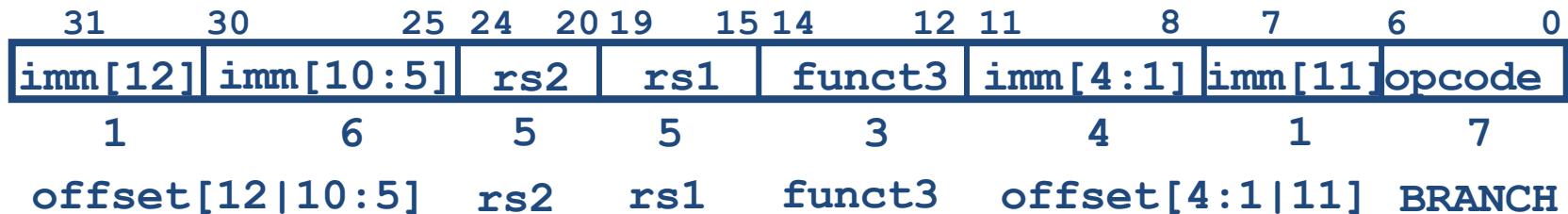
- If we **do** take the branch:

$PC = PC + \text{immediate} * 2$

- **Observations:**

- immediate is number of instructions to jump (remember, specifies words) either forward (+) or backwards (-)

RISC-V B-Format for Branches



- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate imm[12:1]
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode *even* 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

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
```

A vertical curly brace on the right side of the assembly code groups the four instructions following the branch target ('Loop') up to the 'End' label. To the right of this brace is a vertical column of numbers from 0 to 4, with an arrow pointing from the number 4 to the brace. To the right of the numbers is the text 'Count instructions from branch'.

0	Count
1	instructions
2	from branch
3	
4	

- Branch offset = **4×32-bit instructions = 16 bytes**
- (Branch with offset of 0, branches to itself)

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

0 Count
1 instructions
2 from branch
3
4



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

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 bytes

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

RISC-V Immediate Encoding

Instruction encodings, inst[31:0]

31	30	25 24	20 19	15 14	12 11	8 7 6	0	
		funct7	rs2	rs1	funct3	rd	opcode	R-type
		imm[11:0]		rs1	funct3	rd	opcode	I-type
		imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
		imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]	opcode	B-type

32-bit immediates produced, imm[31:0]

31	25 24	12	11	10	5	4	1	0	
-inst[31]-				inst[30:25]	inst[24:21]	inst[20]			I-imm.
-inst[31]-				inst[30:25]	inst[11:8]	inst[7]			S-imm.
-inst[31]-		inst[7]	inst[30:25]	inst[11:8]			0		B-imm.

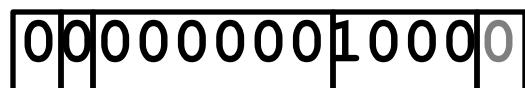
Upper bits sign-extended from inst[31] always

Only bit 7 of instruction changes role in immediate between S and B

Branch Example, complete encoding

beq x19,x10, offset = 16 bytes

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



imm[0] discarded,
always zero

imm[12]



imm[10:5] rs2=10

rs1=19

BEQ

imm[4:1]

BRANCH

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

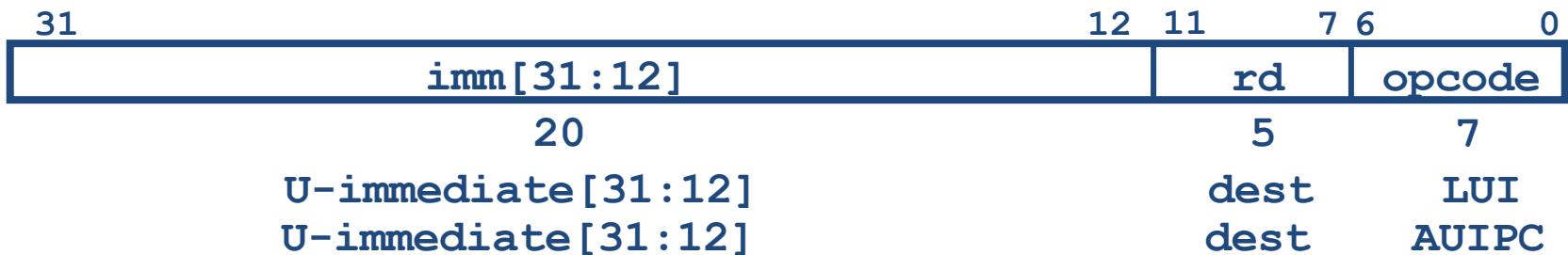
Questions on PC-addressing

- 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 ('position-independent code')
- What do we do if destination is $> 2^{10}$ instructions away from branch?
 - Other instructions save us

```
beq x10,x0,far  
# next instr
```

```
bne x10,x0,next  
j far  
next: # next instr
```

U-Format for “Upper Immediate” Instructions



- 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
 - AUIPC – Add Upper Immediate to PC

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
```

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

Solution

How to set 0xDEADBEEF?

```
LUI x10, 0xDEADC          # x10 = 0xDEADC000
```

```
ADDI x10, x10, 0xEEF    # x10 = 0xDEADBEEF
```

Pre-increment 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
```

Actually: Important!

The assembler treats the provided number for ADDI as signed number. So in order to get 0xEEF, we have to provide the according negative number! So actually, only this works:

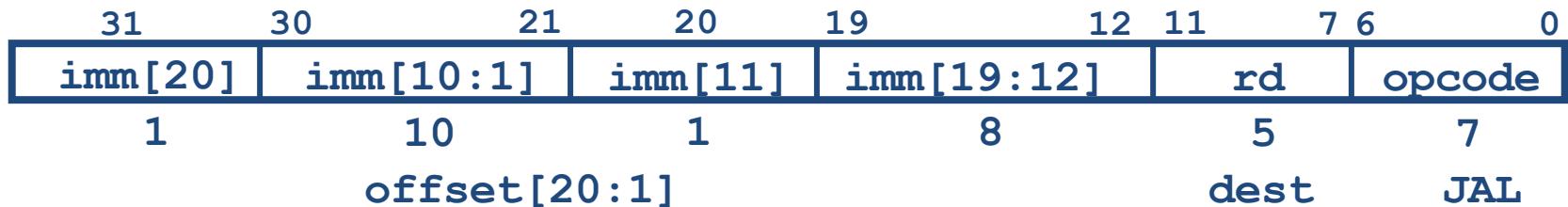
```
ADDI x10, x10, -273     # -273 = 0xFFFFFFFFEEF
```

AUIPC

- Adds upper immediate value to PC and places result in destination register
- Used for PC-relative addressing

Label: AUIPC x10, 0 # Puts address of label in x10

J-Format for Jump Instructions

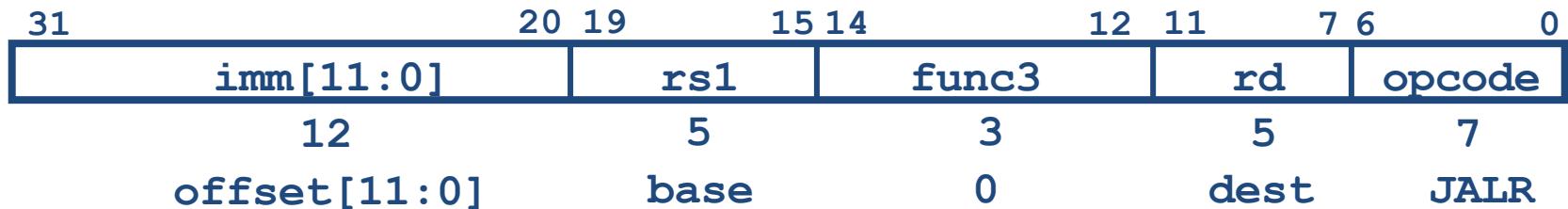


- JAL saves PC+4 in register rd (the return address)
 - Assembler “j” jump is pseudo-instruction, uses JAL but sets `rd=x0` to discard return address
- Set $\text{PC} = \text{PC} + \text{offset}$ (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
 - $\pm 2^{18}$ 32-bit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

Uses of JAL

```
# j pseudo-instruction  
j Label = jal x0, Label # Discard return address  
  
# Call function within 218 instructions of PC  
jal ra, FuncName
```

JALR Instruction (I-Format)



- 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
 - In contrast to branches and JAL

Uses of JALR

```
# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0

# 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>
jalr x0, x1, <lo12bits>
```

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			R-type
		imm[11:0]		rs1		funct3			rd		opcode			I-type
imm[11:5]		rs2		rs1		funct3		imm[4:0]			opcode			S-type
imm[12 10:5]		rs2		rs1		funct3	imm[4:1 11]				opcode			B-type
		imm[31:12]							rd		opcode			U-type
		imm[20 10:1 11]		imm[19:12]					rd		opcode			J-type

CORE INSTRUCTION FORMATS

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

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] 1100011
imm[12:10:5]	rs2	rs1	001	imm[4:1 11] 1100011
imm[12:10:5]	rs2	rs1	100	imm[4:1 11] 1100011
imm[12:10:5]	rs2	rs1	101	imm[4:1 11] 1100011
imm[12:10:5]	rs2	rs1	110	imm[4:1 11] 1100011
imm[12:10:5]	rs2	rs1	111	imm[4:1 11] 1100011
imm[11:0]		rs1	000	rd 0000011
imm[11:0]		rs1	001	rd 0000011
imm[11:0]		rs1	010	rd 0000011
imm[11:0]		rs1	100	rd 0000011
imm[11:0]		rs1	101	rd 0000011
imm[11:5]	rs2	rs1	000	imm[4:0] 0100011
imm[11:5]	rs2	rs1	001	imm[4:0] 0100011
imm[11:5]	rs2	rs1	010	imm[4:0] 0100011
imm[11:0]		rs1	000	rd 0010011
imm[11:0]		rs1	010	rd 0010011
imm[11:0]		rs1	011	rd 0010011
imm[11:0]		rs1	100	rd 0010011
imm[11:0]		rs1	110	rd 0010011
imm[11:0]		rs1	111	rd 0010011

LUI	0000000	shamt	rs1	001	rd	0010011	SLLI
AUIPC	0000000	shamt	rs1	101	rd	0010011	SRLI
JAL	0100000	shamt	rs1	101	rd	0010011	SRAI
JALR	0000000	rs2	rs1	000	rd	0110011	ADD
BEQ	0100000	rs2	rs1	000	rd	0110011	SUB
BNE	0000000	rs2	rs1	001	rd	0110011	SLL
BLT	0000000	rs2	rs1	010	rd	0110011	SLT
BGE	0000000	rs2	rs1	011	rd	0110011	SLTU
BLTU	0000000	rs2	rs1	100	rd	0110011	XOR
BGEU	0000000	rs2	rs1	101	rd	0110011	SRL
LB	0100000	rs2	rs1	101	rd	0110011	SRA
LH	0000000	rs2	rs1	110	rd	0110011	OR
LW	0000000	rs2	rs1	111	rd	0110011	AND
LBU	0000 pred	succ	00000	000	00000	0001111	FENCE
LHU	0000	0000	0000	001	00000	0001111	FENCE.I
SB	0000000000000000			00000	000	00000	1110011
SH	00000000000001			00000	000	00000	1110011
SW	csr	rs1	001	rd	1110011	ECALL	
ADDI	csr	rs1	010	rd	1110011	EBREAK	
SLTI	csr	rs1	011	rd	1110011	CSRRW	
SLTIU	csr	zimm	101	rd	1110011	CSRRS	
XORI	csr	zimm	110	rd	1110011	CSRRC	
ORI	csr	zimm	111	rd	1110011	CSRRWI	
ANDI	csr	zimm	111	rd	1110011	CSRRSI	
							CSRRCI

Not in CA lectures

“And in Conclusion...”

- Simplification works for RISC-V: Instructions are same size as data word (one word) so that they can use the same memory.
- Computer actually stores programs as a series of these 32-bit numbers.
- We have covered all RISC-V instructions and registers
 - R-type, I-type, S-type, B-type, U-type and J-type instructions
 - Practice assembling and disassembling

Question:

Piazza: "Lecture 6 RISC-v poll"

- Select (check) the machine instructions that correctly correspond to the Assembly instruction
- Venus is your friend!
 - But solve at least **I** (`jal x1 -44`) by hand!

A	0x FF 01 01 13	addi sp sp -16
B	0x 00 05 04 13	mv t0 a0
C	0x 00 00 05 13	mv a0 x0
D	0x 00 11 26 23	sw x1 16(x2)
E	0x 00 81 24 23	sw s0 8(sp)
F	0x 01 21 20 23	sw x18 0(x4)
G	0x 00 03 16 63	bne x10 x0 12
H	0x 03 00 00 6F	jal x1 48
I	0x FD 5F F0 EF	jal x1 -44
J	0x 00 01 02 B7	li t0 65536
K	0x 00 00 80 67	ret