

# Lecture 1 - 4<sup>th</sup> Sept 2024

## Instruction Set Architecture: how hardware executes software

### • What is a computer?

↳ just a machine that does whatever the software tells it to do.

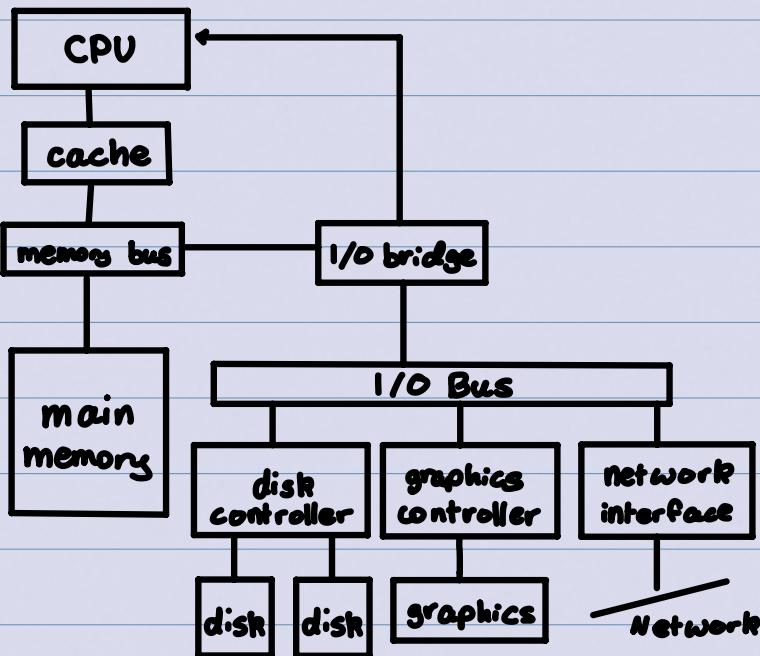
↳ computer > digital circuits > logic gates > transistors

↳ Software is just a series of instructions.

↳ The five classic components of a computer:

- CPU: Central Arithmetic (ALU) and Central Control
- Memory System
- Input and Output

↳ But, here's a more realistic view:



# Architecture vs Microarchitecture

## ↳ Processor architecture:

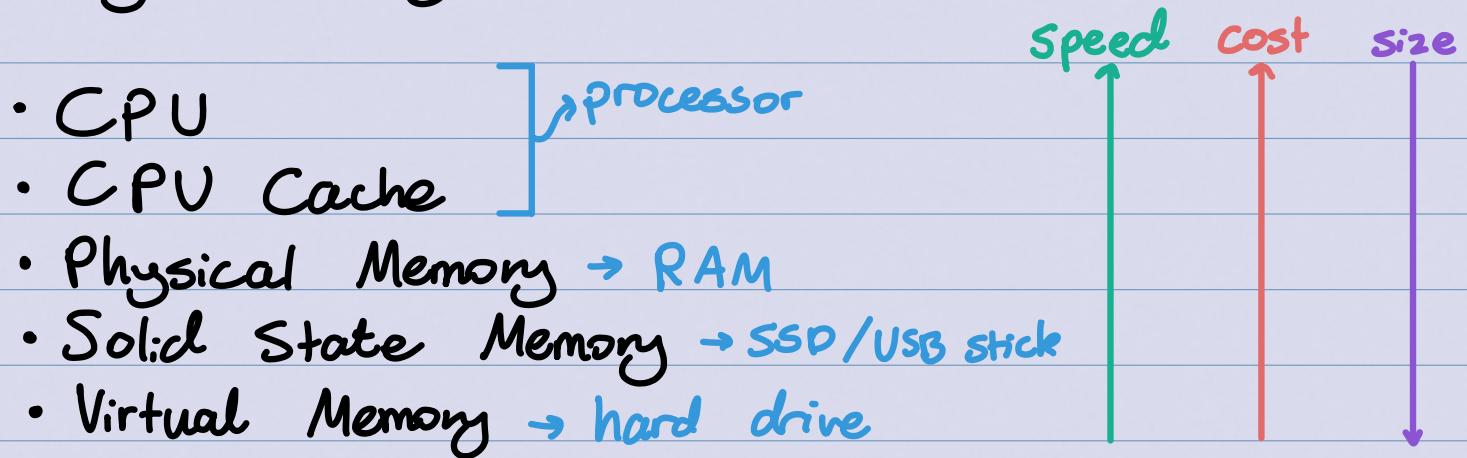
- Functional appearance to software (ISA)
  - Exactly what instructions does it have?
  - Number of memory/storage locations it has
  - Interface!

in this case, compilers!

## ↳ Processor microarchitecture:

- Logical Structure that implements the architecture
  - Number of functional units, interconnection, control
  - Size of the caches
  - Not visible to the software
  - Implementation!

## Memory Hierarchy:



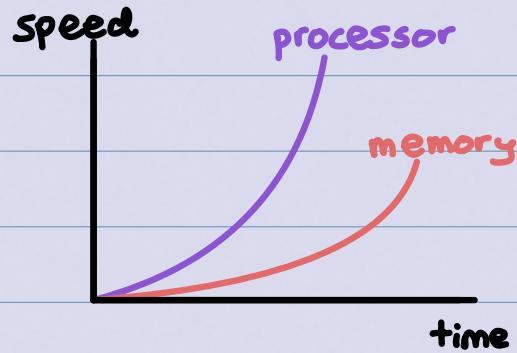
- Moore's Law: every 18-24 months,

- 2x transistors on same chip area
- 2x processor speed
- 2x memory capacity
- ½ energy/power consumption.

↳ technically not a law, but it's a self-fulfilling prophecy.

**Memory Wall:** every 2 years,

- 2x Instructions / second
- 2x memory capacity
- 1.1x memory latency



↳ growing disparity between processor and memory performance!  
 ∵ significant effort in reducing / hiding memory latency.

**Latency:** time from start to finish (aka response time).

**Throughput:** number of tasks completed per time unit  
 (aka bandwidth)

↳ throughput can exploit parallelism, but latency cannot!

• Improving the latency of a component always improves the overall system throughput.

$$\text{Speedup} = \frac{\text{Performance } (\alpha)}{\text{Performance } (y)} \quad \text{→ "}\alpha\text{ is "speedup" times faster than } y\text{"}$$

↳ can be broken down into  $\frac{\text{throughput } (\alpha)}{\text{throughput } (y)}$  or  
 $\frac{\text{latency } (y)}{\text{latency } (\alpha)}$ .  
 don't forget - latency will likely be  $< 1$ , so need to invert fraction!

**Iron Law of Performance:**

$$\text{CPU Time: } \frac{\text{instructions}}{\text{program}} \cdot \frac{\text{cycles}}{\text{instruction}} \cdot \frac{\text{Seconds}}{\text{cycle}}$$

aka CPI

IPC is inverse - instructions per cycle

Amdahl's Law : Speedup =  $\frac{1}{(1 - \text{Frac}_{\text{EHN}}) + \frac{\text{Frac}_{\text{EHN}}}{\text{Speedup}_{\text{EHN}}}}$

Example :

- Enhancement 1: speedup of 20 on 10% of time

$$\hookrightarrow \frac{1}{(1 - 0.1) + \frac{0.1}{20}} = \frac{1}{0.9 + 0.005} = 1.105.$$

- Enhancement 2: speedup of 1.6 on 80% of time.

$$\hookrightarrow \frac{1}{(1 - 0.8) + \frac{0.8}{1.6}} = \frac{1}{0.2 + 0.5} = \frac{1}{0.7} = 1.43.$$

$\therefore$ , the second enhancement is better  $\rightarrow$  even though the first enhancement has a speedup of 20, it's only helpful 10% of the time.

$\hookrightarrow$  "make the common case fast" !!

Power Consumption : transistors consume / dissipate power in two ways:

- Dynamic Power : Consumed by activities in a circuit (switching transistors)

$$\hookrightarrow P = \frac{1}{2} C \cdot V^2 \cdot f \cdot a$$

where:  $C$  = capacitance ( $\sim$  chip area)

$V$  = power supply voltage

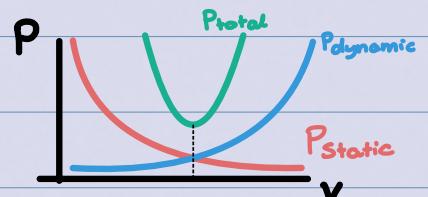
$f$  = clock frequency

$a$  = activity factor

- Static Power : consumed when powered on but idle (leaking current)

↳ as voltage decreases, leakage increases.

- we strive for the "sweet spot" between static and dynamic power:



- Power Wall: supply voltage (generally) decreasing over time.

↳ emphasis on power starting ~2000.

- Since ~2000, we've "hit a wall" and aren't making as much progress.

- What is an ISA?

↳ Functional and precise specification of a computer.

- An ISA is a "contract" between the software and the hardware.
  - Specifies what hardware promises to do when it sees certain instructions, but not how it does it.

## CISC vs RISC :

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
 

↳ CISC (Complex Instruction Set Computing)

- Now, we keep the instruction set small and simple and let the software do the complicated operations by composing simpler ones.
  - This makes it easier to build fast hardware.
- ↳ RISC (Reduced Instruction Set Computing)

### Instruction Set:

- Arithmetic and logic: add, subtract, multiply, divide, AND, OR, XOR, ...
- Data movement: move, load, store
- Control flow: branch and jump
- System (privileged): used to manage processor state, handle exceptions, etc.
- Each instruction has a specific format and encoding

### Registers

- General-Purpose Registers: can be used for various purposes as determined by the programmer or compiler.
- Special-Purpose Registers: have specific roles such as:
  - ↳ Program Counter (PC): holds the address of the next instruction to be executed.
  - ↳ Stack Pointer (SP): points to the top of the stack in memory
  - ↳ Status Register: holds flags that indicate the results of operations (e.g., zero flag).

## Memory Model:

- this is virtual memory,  
not physical!
- **Address Space**: defines the range of memory locations that can be addressed
    - ↳ Eg: a 32-bit address space can address  $2^{32} = 4\text{ GB}$ .
  - **Byte Ordering**: specifies whether multi-byte values are stored with the most significant byte first (big-endian) or last (little-endian)
  - **Alignment Requirements**: some ISAs require data to be aligned in memory in specific ways.
    - ↳ Eg: 32-bit values may need to be stored at addresses that are multiples of four.

## Addressing Modes:

- **Immediate**: the operand is included directly in the instruction itself.
  - ↳ often used for small constants and quick arithmetic operations.
- **Register**: the operand is stored in a CPU register
  - ↳ often used for frequently-accessed variables and intermediate results.
- **Direct**: the instruction contains the memory address where the operand is located.
  - ↳ often used for accessing static variables or fixed memory locations.
- **Indirect**: the instruction specifies a register that contains the memory address of the operand
  - ↳ often used for accessing pointer-based structures.

## RISC-V

- New open-source and license-free ISA spec.
  - Appropriate for all levels of computing system, from microcontrollers to supercomputers.
  - Simple and elegant!
  - Designed with modularity and extensibility in mind
  - Many optional extensions targeting different use cases:
    - M: integer multiplication and division
    - A: atomic instructions
    - F: single-precision floating point
    - D: double-precision floating point
    - C: compressed instructions
    - V: vector operations
- ↳ Eg: "RV64MC" would be a RISC V 64-bit implementation with the extensions M and C.

- There are 32 registers in RISC-V, numbered from 0 to 31:  $x_0 - x_{31}$ 
  - ↳  $x_0$  is special, it always holds the value zero.
- Each register is 32-bits wide → note, 32 bits is called a "word"
- Some registers have conventional uses:
  - ↳  $x_1$ : return address (ra)
  - $x_2$ : stack pointer (sp)
  - $x_{10} - x_{17}$ : function arguments/return values ( $a_0 - a_7$ )
- Since registers are close to the processor, they're very fast (faster than 0.25 ns)!

## RISC-V arithmetic instruction syntax:

opname, rd, rs1, rs2

opname: operation, by name

rd: "destination", operand getting result

rs1: "source 1", 1<sup>st</sup> operand for operation

rs2: "source 2", 2<sup>nd</sup> operand for operation

Eg: add, x1, x2, x3

↳  $a = b + c$ , where  $a$  is stored at register 1,  $b$  in register 2, and  $c$  in register 3.

Example: how would we implement the following

Statement:  $a = b + c + d - e$ ?  $a$  is  $x10$ ,  $b$  is  $x1$ ,  $c$  is  $x2$ ,  $d$  is  $x3$ , and  $e$  is  $x4$ .

↳  
add x10, x1, x2 #  $a = b + c$   
add x10, x10, x3 #  $a = a + d$   
sub x10, x10, x4 #  $a = a - e$

Example:  $f = (g + h) - (i + j)$ ; , where  $f$  is  $x19$ ,

$g$  is  $x20$ ,  $h$  is  $x21$ ,  $i$  is  $x22$ , and  $j$  is  $x23$ .

↳ we'll have to use intermediate temporary registers!

↳  
add x5, x20, x21 # temp1 = g + h  
add x6, x22, x23 # temp2 = i + j

sub x19, x5, x6

# f = temp1 - temp2

## Immediates : numerical constants

↳ add immediate instruction: similar to add instruction, but the last operand must be a number, not a register:

addi rd, rs1, number

↳ Eg: add 6 to the value in register x3.

↳ addi x3, x3, 6 #  $x_3 = x_3 + 6$ .

• There is no subtract immediate instruction in RISC-V → just pass a negative number!

• Register zero ( $x_0$ ): RISC-V hardwires the register 0 ( $x_0$ ) to value 0.

↳ Eg: add x3, x4, x0 : f = g

Eg: addi x3, x0, 0x ff : f = 0x ff

∴, the instruction add x0, x1, x2 won't do anything!

Also, x0 is useful for clearing a register or negating a number.

RISC-V is a load-store architecture!

↳ Load-store (aka register-register) architecture: memory access is limited to load and store instructions. All other instructions (arithmetic, logical, etc) operate only on registers.

RISC-V Load Instructions Syntax:

$l[\text{size}] \text{ rd, imm(rs1)}$

$l$ : Stands for load

[size]: specifies the size of the data to load

↳ w for word, h for halfword, b for byte

rd: destination register

imm: immediate offset value

rs1: the base register containing a memory address

Example: translate the following code to RISC-V:

int A[100];

$g = h + A[3];$

$lw \ x10, \ 12(x15)$

# g gets A[3]

$add \ x10, \ x12, \ x10$

#  $g = h + g = h + A[3]$

RISC-V Store Instructions Syntax:

$s[\text{size}] \text{ rs2, imm(rs1)}$

s: Stands for store

[size]: specifies the size of the data to store

↳ w for word, h for halfword, b for byte

rs2: the source register containing the data to store

imm: an immediate offset value

rs1: the base register containing a memory address

Example: translate the following code to RISC-V:

int A[100];

A[10] = h + A[3];

lw x10, 12(x15)  
add x10, x12, x10  
sw x10, 40(x15)

temp A[3]  
register of A  
temp h  
temp  
temp 40 bits into A aka A[10]

# temp gets A[3]

# temp = h + temp = h + A[3]

# store temp in A[10]

Loading and Storing Bytes:

- Using lb and sb.
- the most significant bit of the byte is extended to fill the rest of the word → aka "sign-extend" the byte.

↳ Example: what does lb x10, 3(x11) do?

↳ contents of memory location with address = 3 + contents of register x11 is copied to the low byte position of register x10.

Example: what will be in x12 after these instructions?

addi x11, x0, 0x3F5

sw x11, 0(x5)

lb x12, 1(x5)

1)  $x11$  becomes  $1001001031F5$  *this is in hex for simplicity*  
↳ each 2 digits represents a byte  
↳ 4 bytes is a word!

2)  $x5$  becomes  $x11 = 1001001031F5$

*sign-extended* ↳ skip first byte, because  $1(x5)$

3)  $x12$  becomes  $1001001001031$

**Pseudoinstructions:** instructions that are not actually implemented in hardware but are recognized by the assembler and translated into one or more real hardware instructions.

**Decision making:** based on computation, do (or don't do) something different. Eg: if - else

↳ Assembly instructions support decision making called branch.

- **Branch:** change of control flow

- **Conditional Branch:** change control flow based on outcome of comparison.

↳ beq, bne, blt(u), bgt (u)

- **Unconditional Branch:** always branch

↳ j, jal, jr

**RISC-V Conditional Branch Instructions Syntax:**

b[cond] rs1, rs2, L

b: stands for branch

[cond]: specifies the condition of the branch.

↳ eq for equal, ne for not equal, etc.

rs1, rs2: registers containing the 2 operands for comparison.

L: a label (symbolic address for another position in the code)

Example: translate the following code to RISC-V instructions:

if (i==j) {  
    f = g+h; }

, assume i → x13, j → x14, f → x10,

g → x11, h → x12.

bne x13, x14, Exit

# if (i ≠ j), branch to Exit

add x10, x11, x12

# ∵ i=j, so f = g+h

Exit:

# Exit label with nothing after it.

↳ note: it's common to need to negate the if statement.

Example: translate the following code to RISC-V instructions:

if (i==j) { f = g+h; }  
else { f = g-h }

, assume i → x13, j → x14, f → x10,

g → x11, h → x12.

bne x13, x14, Else

# if (i ≠ j), branch to Else

add x10, x11, x12

# ∵ i=j, so f = g+h

j Exit

# end if i=j

Else: sub x10, x11, x12

# ∵ i ≠ j, so f = g-h

Exit:

# end

• General programs need to test < and > as well

# RISC-V Magnitude Comparison Instruction Syntax:

blt reg1, reg2, L

blt: Stands for branch on less than

if (reg1 < reg2)

reg1, reg2: registers containing the operands to compare

L: Label to branch to

- There is also bltu, which acts exactly like blt, but does not consider negative numbers.

↳ bltu: branch on less than unsigned.

Loops in Assembly: while / do... while / for

Example: translate the C code to RISC-V instructions:

```
int A[20];           Assume x8 is A[0], x10 is sum, x11 is i  
int sum = 0;  
for (int i = 0; i < 20; i++) { sum += A[i]; }
```

add x9, x8, x0	# copy A[0] to x9
add x10, x0, x0	# Sum (x10) = 0
add x11, x0, x0	# int i = 0
addi x13, x0, 20	# x13 = 20 (loop bound)

Loop:

bge x11, x13, Done	# end if i > 20
lw x12, 0(x9)	# load A[i] into x12
add x10, x10, x12	# sum += A[i]

addi  $x9, x9, 4$  #  $A[i+1]$  in  $x9$   
addi  $x11, x11, 1$  #  $i++$   
j Loop # loop again  
Done: # end!

## Logical Operators:

### • Bit-by-bit AND

↳ C: &

RISC-V: and

### • Bit-by-bit OR

↳ C: |

RISC-V: or

### • Bit-by-bit XOR

↳ C: ^

RISC-V: xor

### • Bit-by-bit shift left

↳ C: <<

RISC-V: sll

### • Bit-by-bit shift right

↳ C: >>

RISC-V: srl

↳ useful for moving,  
extracting, and inserting  
groups of bits!

### ↳ Bitwise and used for 'masks'

↳ andi with  $0x000000FF$  isolates the least significant byte

↳ andi with  $0xFF000000$  isolates the most significant byte

↳ XOR gate with  $x$  and 1 gives  $\bar{x}$ !

↳ there is no logical NOT in RISC-V, because it can be implemented using XOR.

## Shift Left Logical (SLL) and immediate (SLLI):

Slli: rd, rs1, number

Slli: stands for shift left logical immediate

rd: destination register

rs1: source register

number: amount to shift

Example: Slli x11, x12, 2

↳ Store in x11 the value from x12 shifted by 2 bits to the left, inserting 0s on the right. the bits on the left disappear (not wrap).

↳ before: 0000 ... 0010

after: 0000 ... 1000

Srl and SRLI do the exact opposite shift.

## Arithmetic Shifting:

Shift right arithmetic: Sra, Srai

↳ moves n bits to the right

• inserts high-order sign bit into empty bits

Srai rd, rs1, number

srai: Stands for shift right arithmetic immediate

rd: destination register

rs1: source register

number: amount to shift

Example: srai x10, x10, 4

↳ replace the contents of register x10 by bit shifting 4 bits to the right with the same sign.

before: 1111 1111 1111 1111 1111 1111 1110 0111 = -25

after: 1111 1111 1111 1111 1111 1111 1111 1110 = -2

↳ srai is NOT the same as dividing by  $2^n$ .

• Six Fundamental Steps in Calling a Function:

- 1) Put arguments in a place where the function can access them
- 2) Transfer control to function
- 3) Acquire (local) storage resources needed for the function
- 4) Perform desired task of the function
- 5) Put return value in a place where calling code can access it and restore any registers you used; release local storage.
- 6) Return control to point of origin, since a function can be called from several points in a program.

- Registers are faster than memory, so use them!
- ↳  $a0 - a7$  ( $\times 10 - \times 17$ ): 8 argument registers to pass parameters and two return values ( $a0 - a1$ )
- ↳  $ra$ : one return address register to return the point of origin ( $\times 1$ )
- ↳  $s0 - s1$  ( $\times 8 - \times 9$ ),  $s2 - s11$  ( $\times 18 - \times 27$ ): Saved registers
- Jump and Link (jal) Instruction:
  - ↳ "link" means form an address or link that points to calling site to allow function to return to proper address.
  - ↳ jumps to address and simultaneously saves the address of the following instruction in register  $ra$ .
  - ↳ Eg:  $jal rd, \text{FunctionLabel}$
- ↳ Return from function: jump register (jr) instruction:
  - ↳ unconditional jump to address specified in register:  $jr ra$
  - ↳ Eg:  $jalr rd, rs, imm$

**Stack:** last-in-first-out (LIFO) queue

↳ it's in memory, so need register to point to it.

This is called the stack pointer (sp - x2).

- Convention is to grow stack down from high to low addresses.
- push: places data onto stack; decrements sp.
- pop: removes data from stack; increments sp.

## Stack Frame:

↳ includes:

↳ return "instruction" address

parameters (arguments)

space for other local variables.

- Stack frames are contiguous blocks of memory; SP tells where bottom of stack frame
- When a procedure is called, a new stack frame opens
- When a procedure returns, the stack frame collapses.

Example: translate this C code to RISC-V:

```
int Leaf(int g, int h, int i, int j) {  
    int f;  
    f = (g+h) - (i+j);  
    return f;
```

}

↳ g, h, i, and j in a0, a1, a2, and a3. f in s0.

↓

Leaf:

```
addi sp, sp, - 8      # adjust stack for 2 items
sw s1, 4(sp)          # save s1 for later
sw s0, 0(sp)          # save s0 for later
add s0, a0, a1        # f = g + h
add s1, a2, a3        # s1 = i + j
sub a0, s0, s1        # f = f - s1
lw s0, 0(sp)          # restore s0 for caller
lw s1, 4(sp)          # restore s1 for caller
addi sp, sp, 8         # adjust stack to remove 2 items
jr ra                 # jump back to caller
```

## • Register Conventions:

- **Caller:** the calling function
- **Callee:** the function being called
- when the callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- Register Conventions are a set of generally accepted rules as to which registers will be unchanged after a procedure call (jal).
- Preserved across function call (callee-saved):
  - ↳ caller can rely on values being unchanged

↳ sp, gp, tp, s0-s11, (s0 == fp)

- Not Preserved across function call (caller-saved):
  - ↳ caller cannot rely on values being unchanged.
  - ↳ a0-a7, ra, t0-t6.

## Non-Leaf Procedures:

- Procedures that call other procedures (nested)
  - ↳ both a caller and a callee!
- For nested call, caller needs to save on the stack:
  - 1) it's return address
  - 2) any arguments and temporary registers needed after the call
- Restore from the stack after the call.

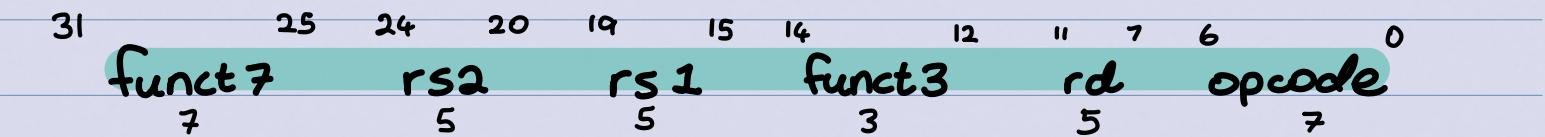
RISC-V's 32-bit instruction words are divided

into fields

- ↳ each field tells the processor something about the instruction

- Six basic types of instruction formats:
  - ↳ R-Format: register-register arithmetic operations
  - ↳ I-Format: register-immediate arithmetic operations; loads
  - ↳ S-Format: Stores
  - ↳ B-Format: branches (minor variant of S-Format)
  - ↳ U-Format: 20-bit upper immediate instructions
  - ↳ J-Format: jumps (minor variant of U-Format)

## R-Format Instruction Layout: opname, rd, rs1, rs2



- funct7, funct3, and opcode describes what operation to perform.
- all R-Format instructions have opcode 0110011.
- register fields (rs1, rs2, rd) hold 5-bit unsigned integers [0-31] corresponding to a register (x0 - x31)
- all 10 RV32 R-Format Instructions:

funct7			funct3		opcode	
2's comp negation of rs2	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

Eight funct3 fields for ten instructions

set less than (see lab manual)

Different funct7 & funct3 encodings select different operations.

- But, immediates need to be wider.

## ∴ the I-Format:



- imm[11:0] holds 12-bit wide immediate values:
  - ↳ Values in range - 2048 : 2047
  - ↳ CPU sign extends to 32 bits before use in an arithmetic operation.

## · all 9 RV32 I-Format Arithmetic Instructions:

Same funct3 fields as corresponding R-format operation (remember, no subi)

		funct3	opcode		
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	0010011
0000000	shamt	rs1	101	rd	0010011
0100000	shamt	rs1	101	rd	0010011



"Shift by Immediate" instructions encode the shift amount in the lower-order 5 bits of the imm.

- We can only (meaningfully) shift 32-b word by 0-31 positions.
- One higher-order immediate bit used for **sign extend** (srlti vs. srai). Same bit position as in R-Format!

## · load instructions are also I-Format!

### ↳ All 5 RV32 Load Instructions:

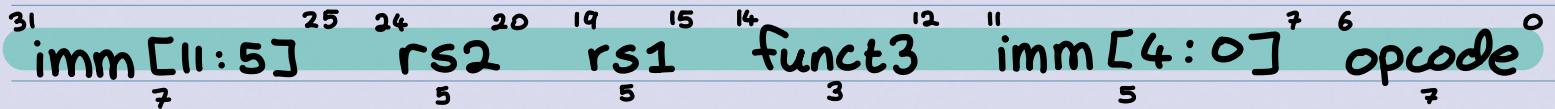
Encodes data size and "signedness" of load operation

		funct3	opcode		
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

- lb : "load byte", lh : "load halfword (16 bits)"
  - ↳ sign extend to fill upper bits of 32-bit register
- lbu / lhu : same as previous, but **unsigned**!
  - ↳ 0 extend to fill upper bits of 32-bit register.

Note: no lwe instruction in RISC-V!

## S-Format Instruction Layout:



- immediate is split up because RISC-V prioritizes keeping register fields in the same place.
- Store address = (Base Register) + (Immediate Offset)
- Store needs immediate and two read registers, but doesn't need a destination register!

## All 3 RV Store Instructions:

Encodes data size of store operation						
			funct3	opcode		
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

No sign/zero extending for store!  
We only write to memory  
the data width specified.

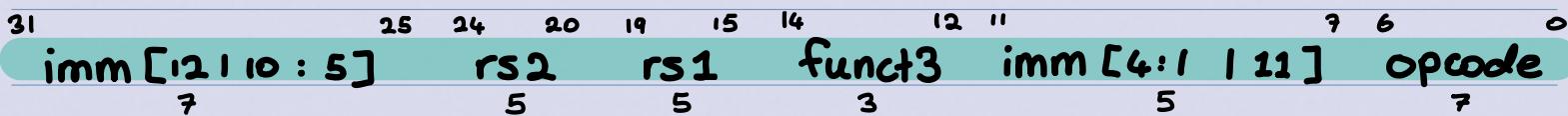
Same data width fields as  
load instructions.

- PC-Relative Addressing: supply a signed offset to update the program counter (PC)
- Position-Independent Code: if all of the code moves, relative offsets don't change!

Branches generally change the PC by a small

amount, therefore, we encode relative offsets as signed immediates.

- Contrast with Absolute Addressing: supply new address to overwrite PC
  - ↳ use sparingly - brittle to code movement
- RISC-V scales the branch offset by 2 bytes!
- B-Format Instruction Layout:



- all conditional branch instructions have opcode 1100011.
- immediate represents relative offset in increments of 2 bytes (half-words).
  - ↳ new-PC = PC + byte\_offset!
- 12 immediate bits imply  $\pm 2^{10}$  32 bit instructions:
  - 1 bit: 2's complement (allow +/- offset)
  - 1 bit: half-word / 16-bit instruction support .
- if imm[12:10:5] =  $\underline{z}xxxxxx$  and imm[4:1|11] =  $wwwwy$ , then the byte-offset is  $\underline{zy}xxxxxxwwww0$ .  
↳ lowest bit of offset is always zero!
- Instruction bit 31 is always the sign bit (highest bit to sign extend in immediate).
- All 6 RV32 B-Format Instructions :

	funct3				Opcode	
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

- B-Format has limited range:  $\pm 2^{10}$  32-bit instructions from current instruction.
  - ↳ if the destination is further away, use unconditional jump!
- J-Format Instruction Layout: jal rd, label



- ↳ Immediate represents relative offset in increments of 2 bytes.
- ↳ PC = PC + byte-offset
- ↳ 20 immediate bits imply  $\pm 2^{18}$  32-bit instructions reachable!
- ↳ 1 bit for 2's complement, and 1 for hw/16-bit instruction support.
- ↳ rd gets return address ( $rd = PC + 4$ )

but what if we still want to jump further?

- U-Format Instruction Layout: opname rd, immed
  - ↳ "Upper Immediate"



- ↳ immediate represents upper 20 bits of a 32-bit immediate
- ↳  $imm = immed \ll 12$

- lui instruction (Load Upper Immediate): lui rd, immed
  - Write a 20-bit immediate value into the upper 20 bits of register rd, and clear the lower 12 bits.
  - ↳  $rd = immed \ll 12$ .

↳ lui together with addi: (to set lower 12 bits)

can create any 32-bit value in a register:



Example:

lui x10, 0x87654 #  $x10 = 0x87654000$

addi x10, x10, 0x321 #  $x10 = 0x87654321$

- The li (load immediate) pseudoinstruction resolves to lui + addi as needed, eg: li x10, 87654321.

- However, there's an edge case because addi sign extends!

↳ Solution: if the 12-bit immediate is negative, just add one to the upper 20-bit load.

↳ just use li pseudoinstruction, it automatically handles this!

- auipc (Add Upper Immediate to PC): auipc rd, immed
  - ↳  $rd = PC + (immed \ll 12)$

↳ Example: auipc x5, 0xABCD E #  $x5 = PC + 0xABCD E000$

↳ In practice: Label: auipc x5, 0 #  $x5 = \text{address of label}$

• Note: unlike `jal` (relative to PC), `jalr` addresses are relative to `rs1`, which is modifiable by arithmetic instructions. ∴, `jalr` lets us do bigger jumps!

→ that's the end of the RISC-V ISA!!

• Translator: converts a program from the source language to an equivalent program in another language.  
↳ note: translating to lower-level languages almost always means higher efficiency and performance.

• Interpreter: directly executes the program in the source language.

↳ note: easier to debug and port to different platforms.

• Compiler:

↳ Input: high-level language code (eg, `foo.c`)

Output: assembly language code (eg, `foo.s`)

↳ note: output may include pseudoinstructions!

• Assembler:

↳ Input: assembly language code (`foo.s`)

Output: machine language module, object file (eg, `foo.o`)

• Reads and uses directives

• Replaces pseudoinstructions with true assembly.

• Directives: give directions to the assembler

• often generated by the compiler

• directives do not produce machine instructions! Rather, they inform how to build different parts of the object file.

↳ .text

Subsequent items put in user Text segment (machine code)

↳ .data

Subsequent items put in user Data segment (source file data in binary)

↳ .globl sym

declares sym global and can be referenced from other files.

↳ .String str

store the string str in memory and null-terminate it

↳ end it will a '\0'

↳ .Word w<sub>1</sub> ... w<sub>n</sub>

store the n 32-bit quantities in successive memory words

## Object File Format:

1) Header

2) Text Segment

3) Data Segment

4) Symbol Table

5) Relocation Information

6) Debugging Information

### 1- object file header

↳ Size and position of other pieces of the object file

### 2- text segment

↳ machine code. all necessary info already in the instruction!

• Simple case: arithmetic / logical / shifts / etc

• PC-relative branches / jumps

↳ once pseudoinstructions are replaced, all PC-relative

addressing can be computed.

↪ take Two-Passes over the program:

Pass 1: remember positions of labels (store in symbol table)

Pass 2: Use label positions to generate machine code

### 3 - data segment

↪ however, some references, such as to other files or to static data, cannot yet be determined. ∴, the assembler jots them down in the Relocation Information and the Symbol Table!

### 4 - Symbol table

↪ list of "items" in this file

- Instruction Labels

↪ used to compute machine code for PC-Relative addressing in branches, function calling, etc

- global directive: labels can be referenced by other files

- Data: anything in the .data section

### 5 - relocation table

↪ lines of code to fix later (by linker)

- list of "items" whose address this file needs
- any external label jumped to
- any piece of data in static section

## • Linker

↳ Input: object files (eg, foo.o)

Output: executable machine code (eg, a.out)

• enables separate compilation of files.

↳ changes of one file does not require recompilation of the entire program!

↳ • puts together text segments from each .o file

• puts together data segments from each .o file, and concatenates to the end of the text segment.

• resolves references.

↳ ie, go through the relocation table and fill in all absolute addresses.

• Note: don't forget that B-type instructions don't need editing, as PC-Relative addressing is preserved, even if text is relocated!

so far, we have described the traditional method of statically linked libraries.

The alternative: dynamically-linked libraries (DLL):

↳ less disk space, but needs extra time to link!

↳ uses machine code as "lowest common denominator"

• Overall, makes compiler, linker, and OS more complex, but provides many benefits that often outweigh these complexities.

## Loader:

↳ Input: executable code (e.g., `a.out`)

Output: runs program

- stored on disk

- when an executable is run, loader loads it into memory and starts it.

- basically, loader is the operating system (OS)!

↓

in more detail:

- Loads program into a newly created address space

- ↳ reads executable's file header for sizes of text and data segments

- ↳ creates new address space for program large enough to hold text and data segments, along with a stack segment.

- copy instructions and data from executable file into new address space.

- copy arguments passed to the program onto the stack

- Initialize machine registers

- ↳ most registers cleared; stack pointer assigned address of first free stack location.

- Jump to start-up routine, which does the following:

- ↳ copy program arguments from stack to registers, set PC

- ↳ If main routine returns, terminate program with exit system call.

C program

object - machine language  
mode foo.o

Compiler → assembler → linker → loader → memory

assembly program  
foo.s

executable - machine  
language program a.out

Example: when are the machine code bits determined for the following assembly instructions?

a) add x6, x7, x8

↳ after assembly, as add is a simple instruction with all necessary info built-in!

b) jal x1, printf

↳ after linking, as the linker must resolve the absolute address of printf.

## Conclusion:

- Compiler converts a single high-level language file into a single assembly language file.
- Assembler removes pseudoinstructions, converts what it can to machine language, and creates a "checklist" for the linker (relocation table).
  - ↳ does 2 passes to resolve addresses, handling internal forward references.
- Linker combines several .o files and resolves absolute addresses.
  - ↳ enables separate compilation
- Loader loads executable into memory and begins execution of the program.

→ end of midterm content!