CSE 4305
Computer Organization and Architecture

Lecture 3
Instruction Set Architecture

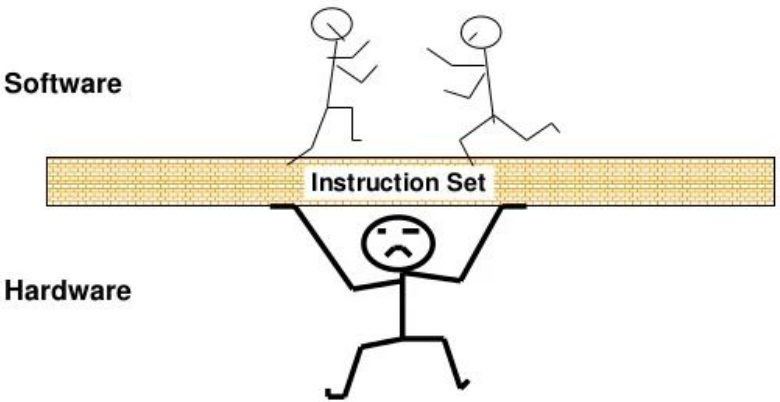
Sabrina Islam
Lecturer, CSE, IUT
Contact: +8801832239897
E-mail: sabrinaislam22@iut-dhaka.edu

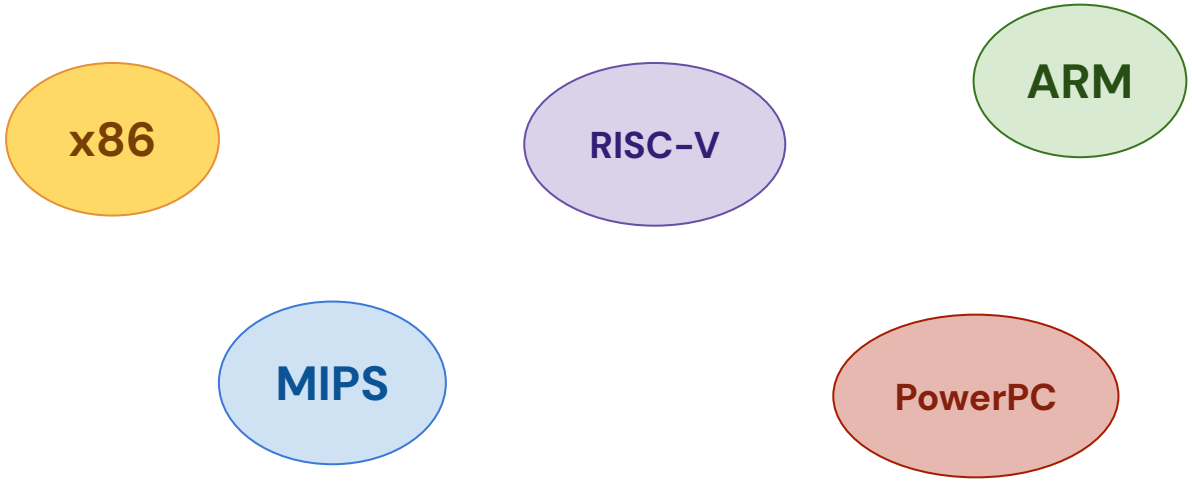
Instruction Set Architecture (ISA)

- Language of the computer.
- Interface between the hardware and the software.
- Abstract model of a computer.
- It specifies the operational capabilities as well as how the CPU performs various supported operations.

Why is ISA so important?

- Compatibility and Portability.
- High-level abstraction of the underlying software.
- Performance optimization.





ISA

They are usually quite similar in order to ensure the simplicity of the equipments.

CISC vs RISC

CISC:

- Complex Instruction Set Architecture
- Variable instruction length
- Much powerful instruction
- Instruction may take more than a single clock cycle to get executed

RISC:

- Reduced Instruction Set Architecture
- Fixed length instructions
- Simpler instruction
- Instruction takes a single clock cycle to get executed

RISC-V

- Originally developed at UC Berkeley starting in 2010.
- Open-source Instruction Set Architecture (ISA).
- RISC-V offers both 32-bit and 64-bit variants.
- One of its main goals is to keep the design of RISC-V based on simplicity and performance, as opposed to focusing on commercial interests.

RISC-V operands:

- Registers (32)
- Memory

RISC-V operation/instruction categories:

- Arithmetic
- Data Transfer
- Logical
- Conditional Branch
- Unconditional Branch

RISC-V *Add* Instruction:

- It is an arithmetic instruction.
- One instruction performs only one operation.
- Each instruction has **3 variables** exactly.

```
add a, b, c
```

add the two variables `b` and `c` and to put their sum in `a`.

- Having exactly 3 variables simplifies the hardware design.
- Conforms to the Design Principle:
  - ***"Simplicity favors regularity"***

Example:

Compile the following C program into RISC-V assembly language:

a = b + c;
d = a –e;

Ans:

```
add a, b, c
sub d, a, e
```

Practice:

Compile the following C program into RISC-V assembly language:

f = (g + h) -(i + j);

Ans:

```
add t0, g, h //temporary variable t0 contains g + h
add t1, i, j // temporary variable t1 contains i + j
sub f, t0, t1 // f gets t0 –t1, which is (g + h) –(i + j)
```

RISC-V operand: Registers

- 32 registers: x0-x31
- The size of a register in the RISC-V architecture is 64 bits.
- the three operands of RISC-V arithmetic instructions must each be chosen from one of the 32 64-bit registers.
- The limited number of registers follow the design principle:
    - ***"Smaller is faster"***
- There must be a balance between increasing the number of registers and keeping the clock cycle fast.

Now let's solve the same problem using registers:

f = (g + h) -(i + j);

The variables f, g, h, i, and j are assigned to the registers x19, x20, x21, x22, and x23, respectively. What is the compiled RISC-V code?

Ans:

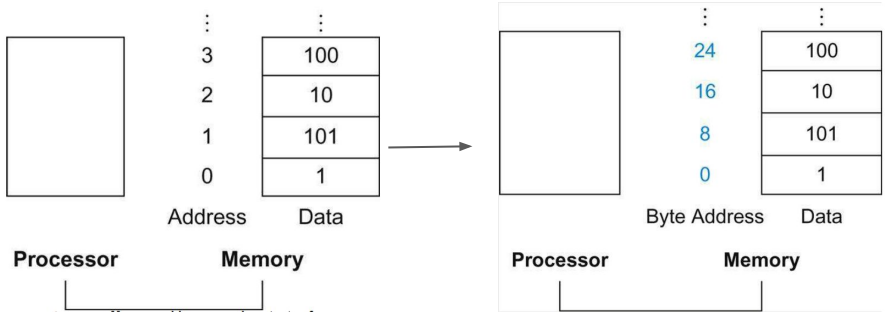
```
add x5, x20, x21 // register x5 contains g + h

add x6, x22, x23 // register x6 contains i + j

sub x19, x5, x6 // f gets x5 –x6, which is (g + h)–(i + j)
```

But where do we store complex data structures like arrays or structures?

Memory

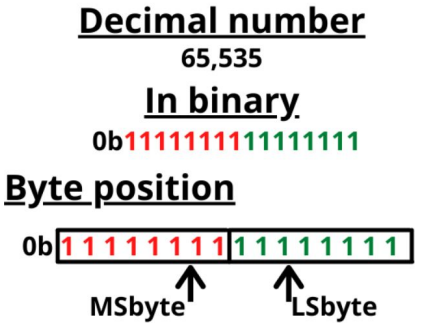
RISC-V operand: Memory

- Used for composite data
  - Arrays, Structures, Dynamic data
- To perform arithmetic operations
  - Load values from memory into registers
  - Stores values from registers to memory.
- RISC-V is **Little Endian.**
- RISC-V does not require words to be aligned in memory.



Endianness:

Endianness is a term that describes the order in which a sequence of bytes is stored in computer memory.



1. Big Endian:
   a. the "big end" (most significant value in the sequence) is stored first, at the lowest storage address.
2. Little Endian:
   a. the "little end" (least significant value in the sequence) is stored first.

Endianness:

Which one is better?

- Depends on the context (i.e. interpreting binary data formats like network packets or file formats, etc situations)

Some processors and systems use little-endian byte order (e.g., x86, x86-64, ARM), while others use big-endian byte order (e.g., PowerPC, some networking protocols).
Bi-endian format: allows switching between little-endian and big-endian as needed.

Memory Operand Example:

Let's assume that A is an array of 100 doublewords and that the compiler has associated the variables g and h with the registers x20 and x21 as before. Let's also assume that the starting address, or base address, of the array is in x22. Compile this C assignment statement:

g = h + A[8];

Ans:

```
ld x9, 8(x22)        // Temporary reg x9 gets A[8]
add x20, x21, x9     // g = h + A[8]
```

Data Transfer Instruction (Load/Store):

Assume variable h is associated with register x21 and the base address of the array A is in x22. What is the RISC-V assembly code for the C assignment statement below?

A[12] = h + A[8];

Ans:

```
ld x9, 64(x22)       // Temporary reg x9 gets A[8]
add x9, x21, x9      // Temporary reg x9 gets h + A[8]
sd x9, 96(x22)       // Stores h + A[8] back into A[12]
```

Registers vs Memory

- Registers are faster to access than memory.
- Registers have higher throughput than memory.
- Operating on memory data requires Load and Store.
  - More instructions to execute.
- Compilers must use registers for variables as much as possible.
  - Only spill to memory for less frequently used variables.
- Cache helps

Immediate Operands

Using quick add instruction:

```
addi x22, x22, 4 // x22 = x22 + 4
```

Make the common case fast:

- Small constants are common
- Immediate operands avoid a load instruction.

Signed and Unsigned numbers

- Base 2 representation of numbers. Why?
- We need a representation that also distinguishes the positive from the negative.
- 2's complement binary representation for signed numbers.
- When signed numbers and when unsigned?

Unsigned Binary Integers

- The numbers having n-bits can range from 0 to 2^n
- Using 64 bits: 0 to +18,446,774,073,709,551,615(10)
- Example:

0000 0000 ….001011(base-2)

$$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{ten}$$
$$= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{ten}$$
$$= 8+0+2+1_{ten}$$
$$= 11_{ten}$$

Signed Binary Integers

- 2's complement is used for signed numbers
- n-bits numbers range from -2^{n-1} $to + 2^{n-1}$
- Using 64 bits: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- Example:

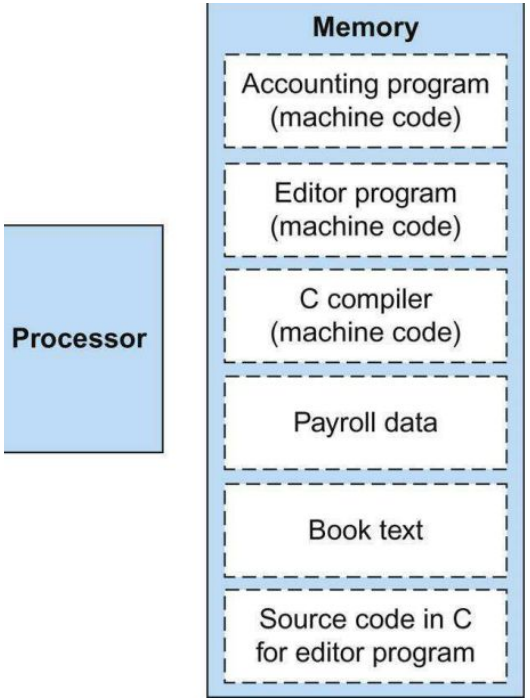
  1111 1111….111100(base-2)

$$(1 \times -2^{63}) + (1 \times 2^{62}) + (1 \times 2^{61}) + \cdots + (1 \times 2^{1}) + (0 \times 2^{1}) + (0 \times 2^{0})$$
$$= -2^{63} + 2^{62} + 2^{61} + \cdots + 2^{2} + 0 + 0$$
$$= -9,223,372,036,854,775,808_{ten} + 9,223,372,036,854,775,804_{ten}$$
$$= -4_{ten}$$

Stored-Program Concept

Instructions are data!

- Programs can be stored in memory as numbers
- Before: A number can mean anything
- Now: Make convention for interpreting numbers as instructions



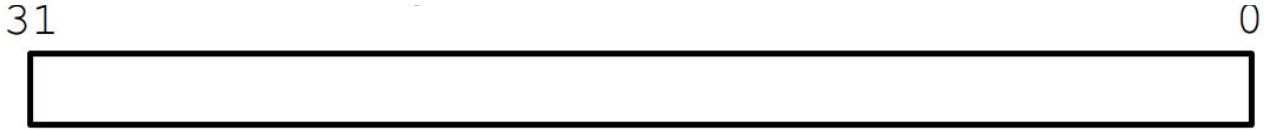
Representing Instructions

Instructions are encoded in binary

- Called machine code

RISC-V instructions

- Encoded as 32-bit instruction words
- Small number of formats encoding operation code (opcode), register numbers, etc.
- Regularity!



Representing Instructions

Let's see an instruction example:

add x9, x20, x22

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

The 32 bits of an instruction are divided into different "fields".

But…

Is this format suitable/enough
for all sorts of operations?

Representing Instructions

- Same length instruction → simpler hardware
- Same format for all instructions → easier for compilation
- Decision:
  - keep all instructions the same length,
  - but define different instruction formats
  - conforms to the final design principle:

"Good design demands good compromises."

Representing Instructions

There are 6 types of instruction formats:

- R-Format - registers
- I-Format - immediates, loads
- S-Format - store instructions
- U-Format - branch instructions
- SB-Format - instructions with upper immediates
- UJ-Format - jump instructions

R-Format Instruction

- Define “fields” of the following number of bits
  - each section: 7 + 5 + 5 + 3 + 5 + 7 = 32
  - each field has a name
- Each field is viewed as its own unsigned int.

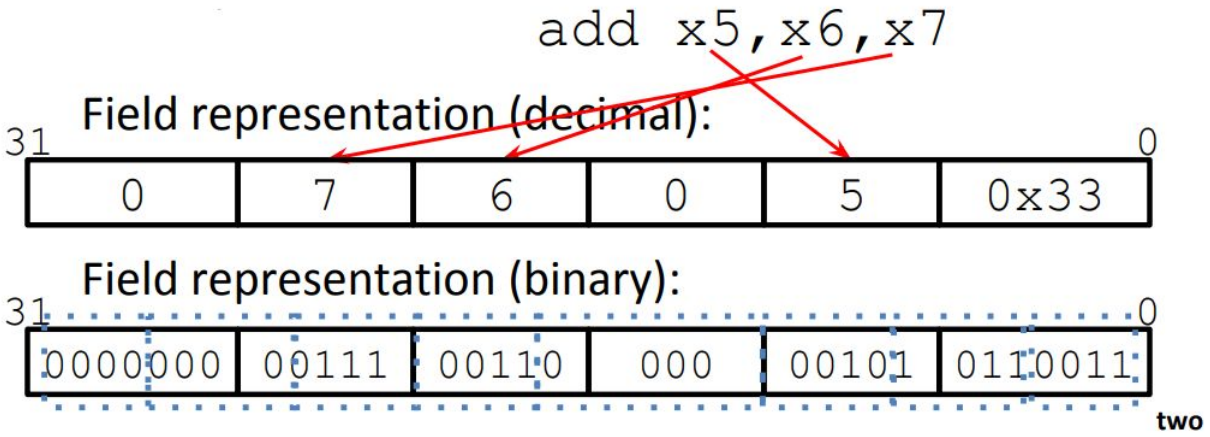
funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

R-Format Instruction

The meaning of each field:

- **opcode (7):** partially specifies operation
- **funct7+funct3 (10):**
  - combined with opcode, these two fields describe what operation to perform. $(2^7) \times (2^3) = (2^{10}) = 1024$
- **rs1 (5):** 1st operand ("source register 1").
- **rs2 (5):** 2nd operand (second source register)
- **rd (5):** "destination register" — receives the result of computation

R-Format Example:



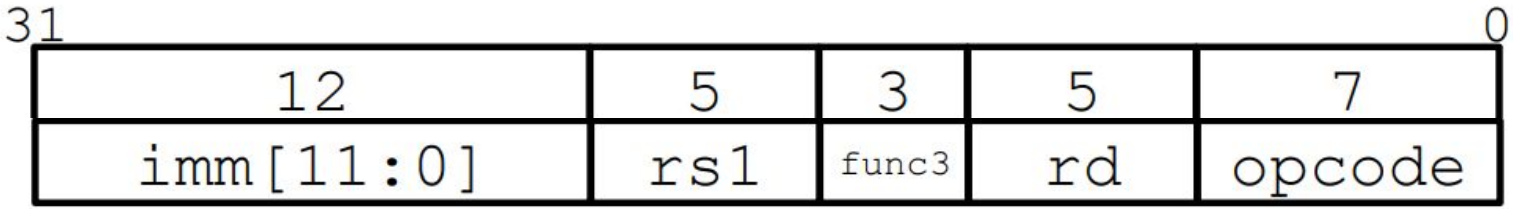
All RV32 R-Format Instructions:

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

I-Format Instruction

- Instruction used for Load and Immediate values
- Define a new instruction format that is mostly consistent with R-Format
  - If instruction has immediate, then it uses at most 2 registers (1 src, 1 dst)
- "fields" of the following number of bits
  - each: 12 + 5 + 3 + 5 + 7 = 32 bits
- rs2 and funct7 replaced by 12-bit signed immediate, imm[11:0]



I-Format 'immediate' Instruction

The fields:

- **opcode (7):** uniquely specifies the instruction
- **rs1 (5):** specifies a register operand
- **rd (5):** destination register
- **immediate (12):**
  - 12 bit number
  - Can represent 2^{12} different immediates
  - imm[11:0] can hold values in range $[-2^{11}, +2^{11})$

I-Format ‘immediate’ Example

RISC-V Instruction:

```
addi x15,x1,-50
```

| 111111001110 | 00001 | 000 | 01111 | 0010011 |

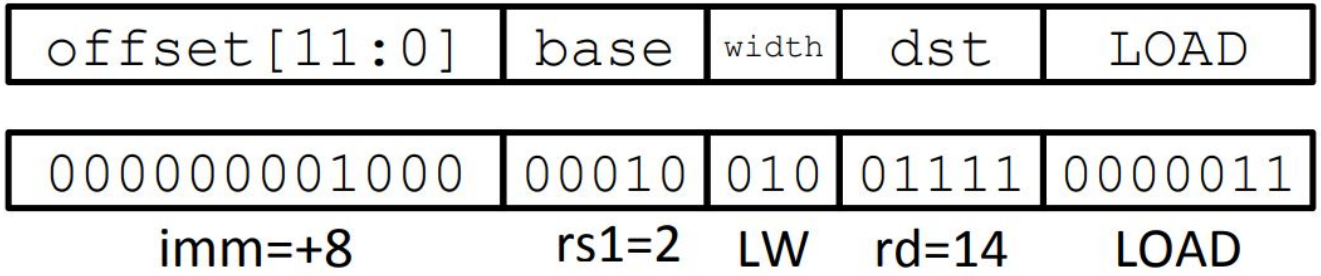
I-Format 'Load' Instruction

- **Offset:** The 12-bit signed offset is added to the base address in register rs1 to form the memory address
- Value loaded from memory is stored in rd

I-Format 'Load' Instruction Example

RISC-V Instruction:

```
lw x14, 8(x2)
```



All RV32 I-Format 'immediate' 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	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	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)

All RV32 I-Format 'Load' Instructions

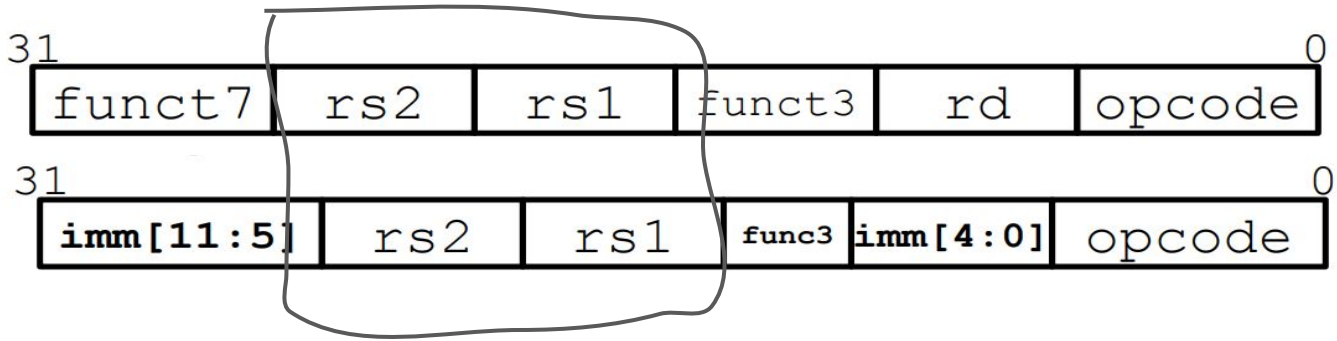
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

funct3 field encodes size and signedness of load data

S-Format Instruction

- Two registers:
  - **rs1** for base memory address
  - **rs2** for data to be stored
- No **rd**!
- The 12-bit **immediate** (offset) in the S-type format is split into two fields
  - upper 7 bits
  - lower 5 bits

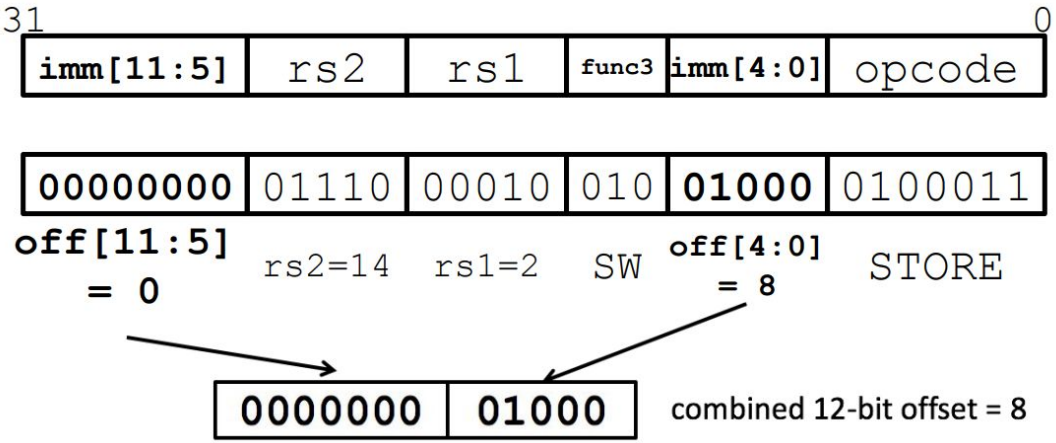
Why?



S-Format Example

RISC-V Instruction:

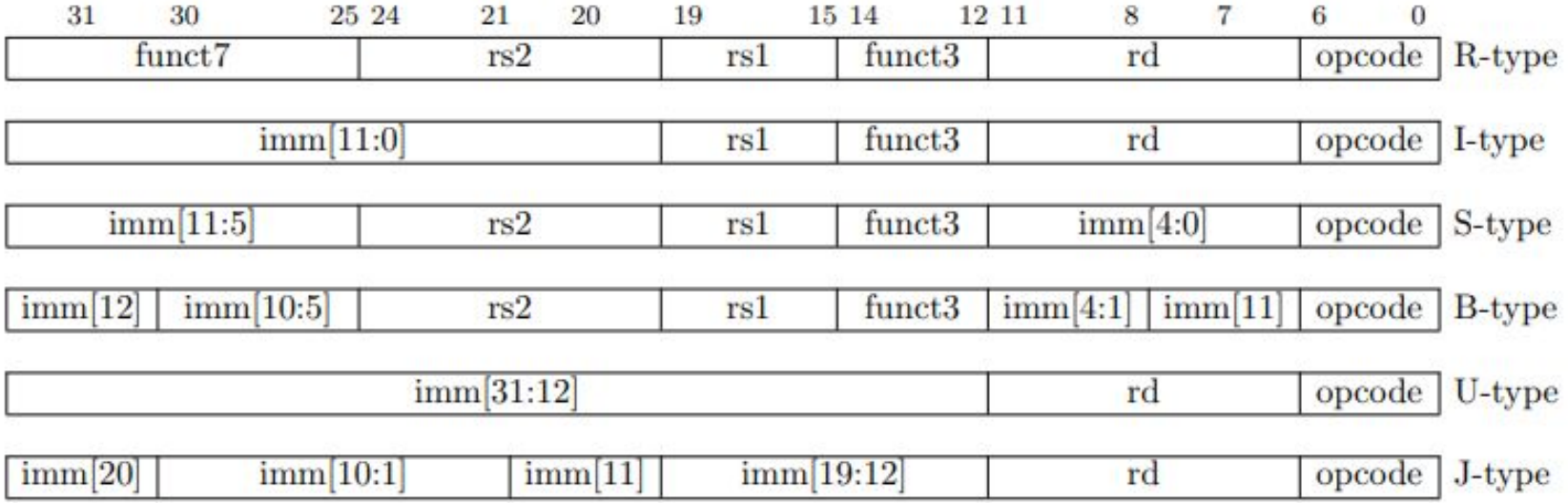
```
sw x14, 8(x2)
```



All RV32 S-Format 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

RV32 Instruction Formats



Problem:

x10 has the base of the array A and x21 corresponds to h.
The assignment statement:

```
A[30] = h + A[30] + 1;
```

After compilation:

```
ld x9, 240(x10) // Temporary reg x9 gets A[30]
add x9, x21, x9 // Temporary reg x9 gets h+A[30]
addi x9, x9, 1 // Temporary reg x9 gets h+A[30]+1
sd x9, 240(x10) // Stores h+A[30]+1 back into A[30]
```

What is the RISC-V machine language code for these instructions?

RISC-V Operations:

- Arithmetic ✓
- Data Transfer ✓
- Logical
- Conditional Branch
- Unconditional Branch

Logical Operations:

- Operating on fields of bits within a word or on individual bits.

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

Logical Operations (Shift):

- Moving all the bits in a doubleword to the left or right, filling the emptied bits with 0s.
- Two types:
  - Shift Left (slli - shift left logical immediate)
  - Shift Right (srli - shift right logical immediate)
- Uses the I-type format.
- Variants:
  - sll, srl, and sra
  - take the shift amount from a register, rather than from an immediate.

Logical Operations (Shift):

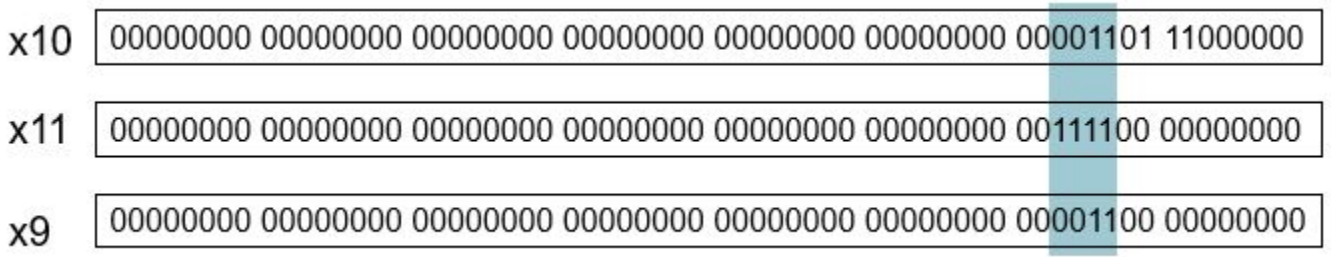
- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - slli by *i* bits multiplies the number by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - srli by *i* bits divides the number by 2^i (unsigned only)

- `slli x11, x19, 4 // reg x11 = reg x19 << 4 bits`

Logical Operations (AND):

- Calculates a 1 only if there is a 1 in both operands.
- Useful to mask bits in a word
  - Select some bits, clear others to 0

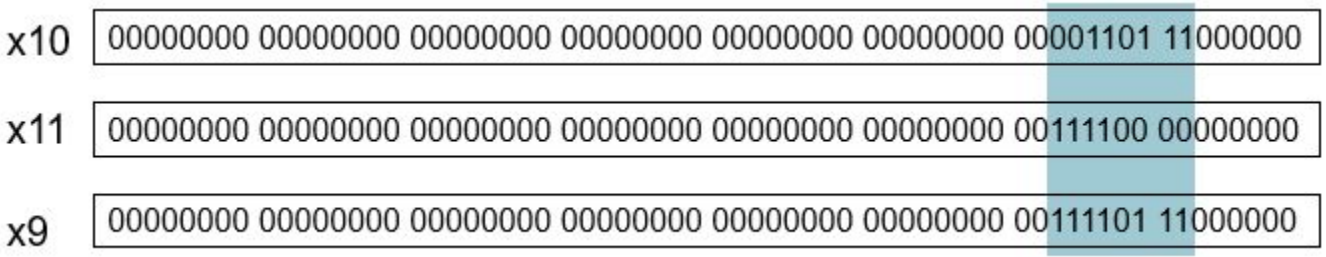
- `and x9, x10, x11        // reg x9 = reg x10 & reg x11`



Logical Operations (OR):

- Calculates a 1 only if there is a 1 in either operands.
- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

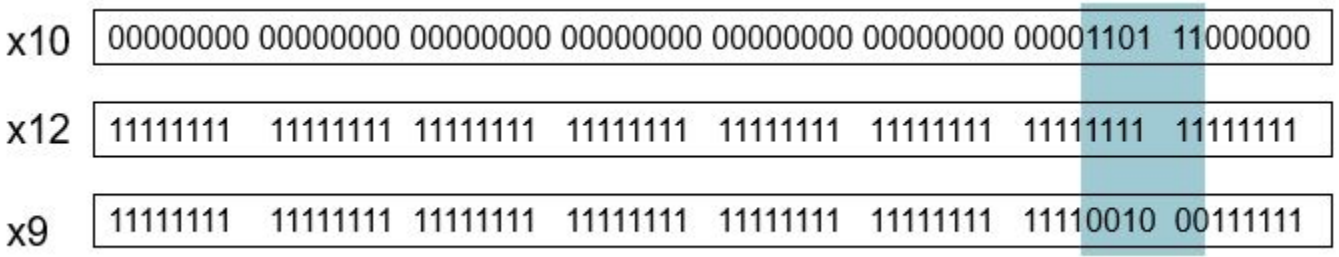
- `or x9, x10, x11 // reg x9 = reg x10 | reg x11`



Logical Operations (XOR):

- Calculates a 1 only if the values are different in the two operands.
- Equivalent to the NOT operation

- `xor x9, x10, x12 // reg x9 = reg x10 ^ reg x12`



RISC-V Operations:

- Arithmetic ✓
- Data Transfer ✓
- Logical ✓
- Conditional Branch
- Unconditional Branch

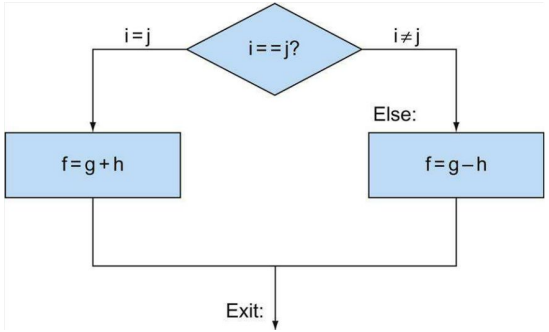
Conditional Operations:

- Instructions to make decision.
- These instructions branches to a labeled instruction if a condition is true. Otherwise, continues sequentially
- Two instructions:
  - `beq rs1, rs2, L1`
    – if (rs1 == rs2) branch to instruction labeled L1
  - `bne rs1, rs2, L1`
    – if (rs1 != rs2) branch to instruction labeled L1
- These instructions are called **Conditional Branch.**

Conditional Operations (IF):

Example:
In the following code segment, f, g, h, i, and j are variables. If the five variables f through j correspond to the five registers x19 through x23, what is the compiled RISC-V code for this C if statement?

```
if (i == j) f = g + h; else f = g –h;
```



Compiled RISC-V code:

```
bne x22, x23, Else
add x19, x20, x21
beq x0,x0,Exit // unconditional
Else: sub x19, x20, x21
Exit: …
```

Conditional Operations (Loop):

Example:

If the five variables i is in x22 and j is in x24, what is the compiled RISC-V code for this C if statement?

```
while (i != j){
    i+=1;
}
```

Compiled RISC-V code:

```
Loop: beq x22, x24, Exit
      addi x22, x22, 1
      beq x0, x0, Loop
Exit: …
```

Conditional Operations:

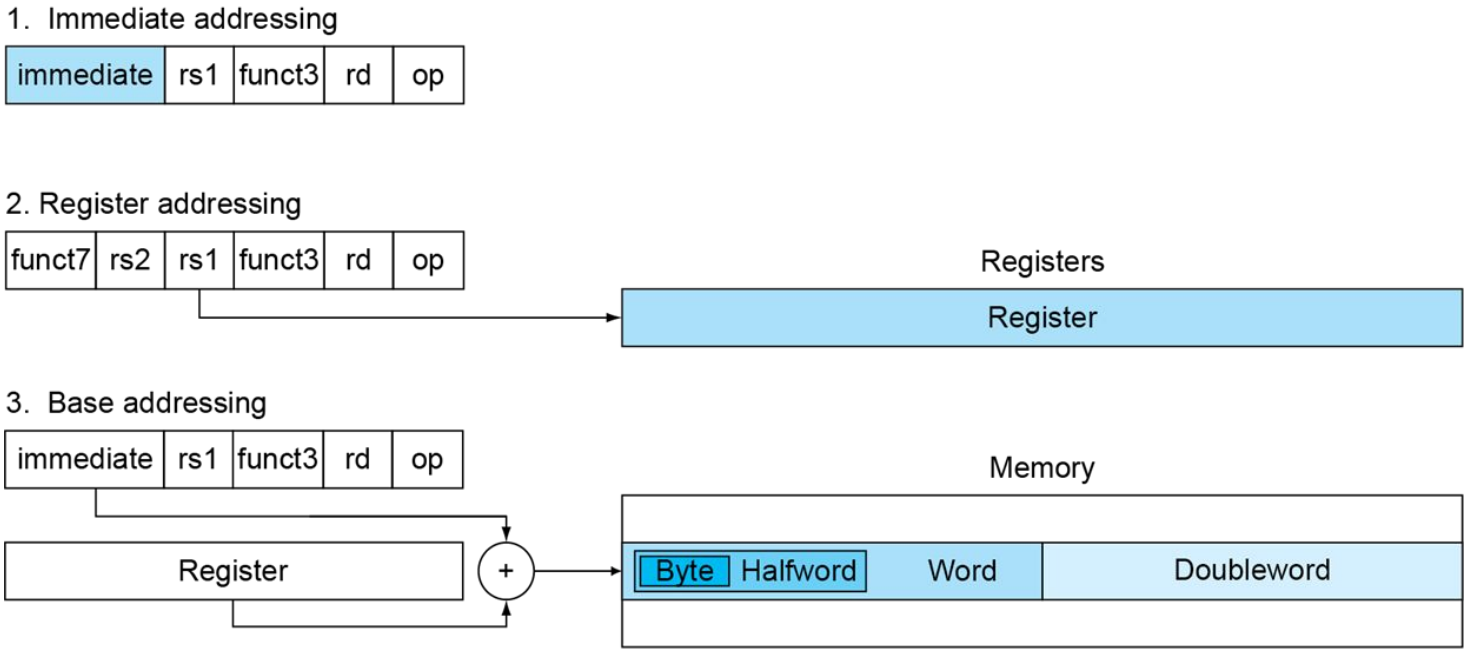
Other instructions:

- Branch if less than - `blt`
- Branch if greater than or equal - `bge`
- Branch if less than, unsigned - `bltu`
- Branch if greater than or equal, unsigned - `bgeu`

Case/Switch Statement :

- The simplest way to implement - converting it into a chain of *`if-then-else`* statements.

RISC-V Addressing summary:



Comparison with x86 (80386):

- Registers
  - 80386 contains only 8 general-purpose registers.
  - RISC-V contains four times more.
- Src/dst
  - The x86 arithmetic and logical instructions must have one operand act as both a source and a destination.
  - RISC-V allows separate registers for source and destination.
- Operand in memory
  - In x86, any instruction may have one operand in memory.
  - Isn't the case for RISC-V.

Fallacies:

1. ***More powerful instructions mean higher performance.***
    a. Fewer instructions required
    b. But complex instructions are hard to implement
        i. May slow down all instructions, including simple ones
    c. Compilers are good at making fast code from simple instructions
2. ***Write in assembly language to obtain the highest performance.***
    a. But modern compilers are better at dealing with modern processors.
    b. More lines of code – more errors and less productivity.

Reference:

1. Computer Architecture and Design RISC-V Edition, by David A Patterson and John L. Hennessy.
2. Documents from CS61C Su18, UC Berkeley.

Thank You!