Imperial College
London

Lecture 6
Instruction Set Architecture
(RISC-V ISA)

Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

In this lecture, the instruction set architecture (ISA) of the RISC-V processor will be introduced. We will only consider the base instruction set for the 32-bit integer version of the ISA. Focus will be on the six different instruction types with emphasis each instruction's functionality and encoding of its machine code.

ISA of a processor does not dictate how the processor is implemented. It only defines how to the processor is programmed. The actual hardware architecture will be covered in the next lecture.

RISC-V

- Developed by Krste Asanovic, David Patterson and colleagues at UC Berkeley in 2010
- First widely accepted **open-source** computer architecture
- Underlying design principles:
  1. **Simplicity favours regularity**
  2. **Make the common case fast**
  3. **Smaller is faster**
  4. **Good design demands good compromises**

RISC-V is by no means the first or the only Reduced Instruction Set Computer that enjoys widespread adoption. The early RISC processor from Berkeley, the MIPS, was used for over four decades in industry. The UK's ARM processor (original stands for Acorn Risc Machine) is the most manufactured CPU in history with an estimated 200 billions being shipped to date. However, RISC-V is the first widely accepted open-source RISC processor. Opening the ISA to the public (royalty free) is a new business model and has captured much attention in the past 5 years. Together with the free toolchain for development and many open-source design freely available, RISC-V is expected to pose real competition to x86 and ARM architecture to become at least one of the dominant play in the sector.

RISC-V was developed by Krste Asanovic and Dave Patterson (and others) in Berkeley in early 2010's. The ISA was first published in 2011, and its future development and ratification are under the control of RISC-V Foudation and RISC-V International located in Switzerland.

According to Patterson and Hennessy's textbook, the underpinning design principles are shown on the slides.

Design Principles

Principle 1: Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware

Principle 2: Make the common case fast

- RISC-V includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- RISC-V is a ***reduced instruction set computer*** **(RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are ***complex instruction set computers*** **(CISC)**

Principle 3: Smaller is Faster

H&H p301-303

As shown in later slides, the instruction set has highly consistent format. The base instruction set is known as RV32I (RISC-V 32-bit integer only) only has 40 instructions. The ISA has two sources and one destination operands. The format of the instructions are divided into only six different types.

The second principle is the RISC-V only has commonly used instructions. Yet, it is Turing Complete, meaning that it can be used to implement any computer algorithms. This makes the RISC-V implementation both small and fast. Complex operations are achieved by stringing together multiple instructions.

Instructions: Addition & Subtraction

C Code	RISC-V assembly code
`a = b + c;`	`add a, b, c`
`a = b - c;`	`sub a, b, c`

- **`Add/sub:`** mnemonic indicates operation to perform
- **`b, c:`** source operands (on which the operation is performed)
- **`a:`** destination operand (to which the result is written)

More complex code is handled by multiple RISC-V instructions.

C Code	RISC-V assembly code
`a = b + c - d;`	`add t, b, c  # t = b + c` `sub a, t, d  # a = t - d`

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

H&H p303

To understand the RV32I ISA, we start with the add and subtract instructions as shown in this slide. The diea is very simple – it shows that for add instructions, we need two source operands (b and c) and one destimation (a).

RISC-V Operands

- **Operand location:** physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V called “32-bit architecture” because it operates on 32-bit data

H&H p304

Where do operands come from and where does the destination operand go?

There are three possibilities. The fastest and most often used operand is from or to **registers** on the CPU chip itself. RISC-V RV32I has 32 32-bit registers. They form an integral part of the CPU design and accessing them is easy and fast. 32 registers means the instruction must use 3 x 5-bit = 15 bit of the 32-bit instruction to specify a register-only (R-type) instruction.

The second possibility is from data memory. To access this, the instruction must specify the data memory address, using a register as a pointer. In RISC-V, the register content is often specified with an associated offset constant value as part of the instruction.

The third possibility is from instruction memory, i.e. the operand is a constant within the instruction itself. In ”RISC-V speak”, this is called an immediate.

32-bit RISC-V Instruction Types

Instruction Type	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate (I-type)	imm[11:0]												rs1					funct3			rd					opcode						
Upper (U-type)	imm[31:12]																				rd					opcode						
Store (S-type)	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch (B-type)	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump (J-type)	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):**: Destination register specifies register which will receive result of computation

RISC-V RV32I has six types of instructions.

R-type (**Register/register**) instructions use only registers as source and destiantions. This instruction type is mostly used for arithmetic and logic operations involving the ALU.

I-type (**Immediate**) instructions has one of the two source operands specified within the 32-bit instruction word as a 12-bit constant (or immediate). This constant is regards as 12-bit signed 2's complement number, which is always sign extended to form a 32-bit operand.

S-type (**Store**) instructions are exclusively used for storing contents of a register to data memory.

B-type (**Branch**) instructions are used to control program flow. It compares two operands stored in registers and branch to a destination address relative to the current Program Counter value.

J-type (**Jump**) instructions are used for subroutine calls.

U-type (**Upper immediate**) instructions are used to specify the upper 20 bits immediate value of a register.

RISC-V Registers

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

H&H p305

RISC-V RV32 has 32 registers designated as **x0** to **x31**. They are "general purpose" registers in the sense that the ISA allows them to be used for any purpose with the exception of **x0**, which ALWAYS contain the value 32'b0. Writing to x0 does not change its content.

Having **x1** to **x31** for any general use can be confusing. Common good practice is included in a guideline where specific registers are used for special functions. For example x1 is used to store the **return address** (of a subroutine) and therefore **x1** is also called **ra**.

The table above shows the various aliases for all 32 registers. You are recommend o use the given name of these registers to make the program more readable. For example instead of using **x0**, you should always refer to it as **zero**.

RISC-V operand from Registers

Name	Register Number	Usage
`s0/fp`	x8	Saved register / Frame pointer
`s1`	x9	Saved register
`s2-11`	x18-27	Saved registers

C Code

```
a = b + c;

a = b + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b, s2 = c
add s0, s1, s2

# s0 = a, s1 = b
addi s0, s1, 6
```

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						

H&H p305

Consider again the add instructions. ADD is a typical ALU instruction in the class of arithematic and logic operations. It needs two source operands and one destination operands to store the results. Shown here is the instruction: `add s0, s1, s2` which uses three registers. Consider the encode of this instructions (slide 21).

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`

The operation is specified with the opcode, funct3 and funct7 fields of the instructions.
opcode = 7'h38 (51), funct3 = 3'b0, funct7 = 7'b0.
rd: s0 = x8 = 5'b01000, rs1 = s1 = x9 = 5'b01001, rs2 = s2 = x18 = 5'b10010
If we fill in the fields with these values according the diagram here, we get:

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
	0	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	1	1	0	0	1	1

Therefore this instruction has a machine code of 32'h01248433.

Similar, for: `addi s0, s1, 6`

op	funct3	funct7	Type	Instruction	Description	Operation
0010011 (19)	000	-	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`

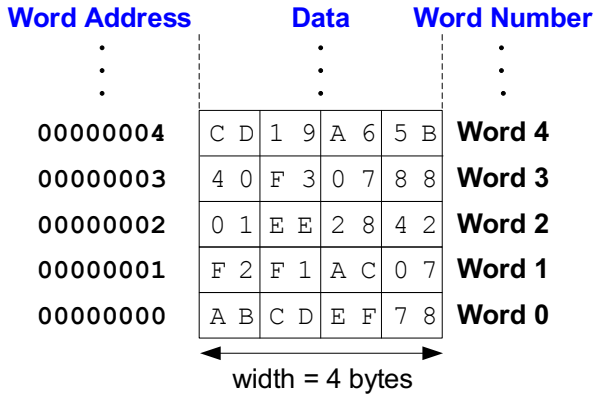
opcode = 7'h13 (19), funct3 = 3'b0.
rd: s0 = x8 = 5'b01000, rs1 = s1 = x9 = 5'b01001 as before.
Imm_{12} = 12'h6.

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
	0	0	0	0	0	0	0	0	0	0	0	6	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1

Therefore this instruction has a machine code of 32’h00648413.

RISC-V operands from memory

- Each 32-bit data word has a unique address



RISC-V uses **byte-addressable** memory (i.e. byte has a unique address), so each 32-bit word uses 4 byte addresses

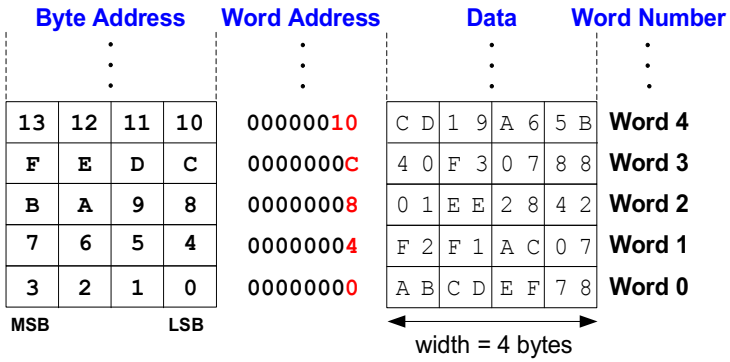
H&H p307

32-bit operands in memory occupies 4 bytes. Some processor uses one unique address for each 32-bit words. MIPS processor is one such example. Everything instruction or data word has a unique address – it is "word addressable" processor.

RISC-V uses byte-addressable to access memory, where EVERY BYTE has a unique address.

RISC-V Byte-addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- 32-bit word = 4 bytes, so word address **increments by 4**



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Therefore, in RISC-V, every 32-bit occupies four unique addresses. If the least significant byte has an address of base = 4, then the most significant byte has an address of base + 3 = 7 as shown above.

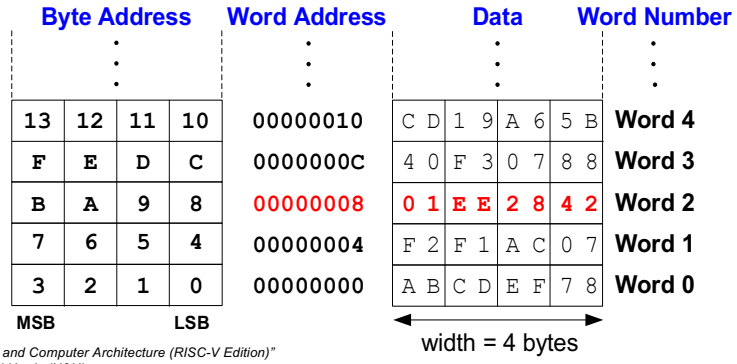
Since all RISC-V instructions are 32-bit, addresses of the instruction memory are all aligned to an increment of 4.

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 8 into s3.
- s3 holds the value 0x1EE2842 after load

RISC-V assembly code

```
lw s3, 8(zero)  # read word at address 8 into s3
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

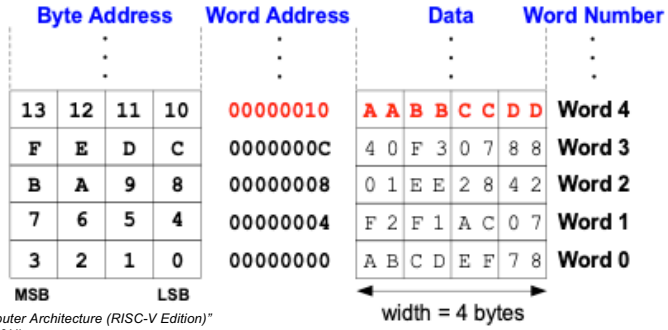
This shows an example of how reading from data memory into register s3 at data memory address 32'h8.

Writing Byte-Addressable Memory

- **Example:** store the value held in t7 into memory address 0x10 (16)
  - if t7 holds the value 0xAABBCCDD, then after the sw completes, word 4 (at address 0x10) in memory will contain that value

RISC-V assembly code

```
SW t7, 0x10(zero)  # write t7 into address 16
```



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

This is an example of storing from register t7 to memory address 32'h10.

RISC-V: Operands from Constants

- 12-bit signed constants (immediates) using addi:

C Code

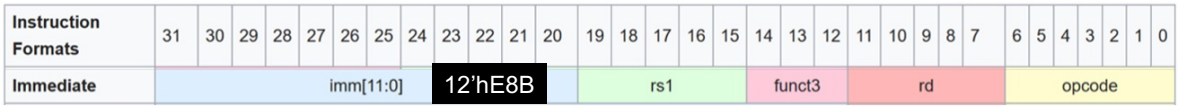
```
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

372 = 12'h174 = 12'b0001_0111_0100
-372 = 12'b1110_1000_1100 = 12'hE8B

- Form 32-bit constant using **sign extension**



Any immediate that needs **more than 12 bits** cannot use this method.

H&H p306

An operand can also be a constant encoded within the instruction itself. Here comes a problem: since RISC-V instructions are all **single** 32-bit words, and an operand is also 32-bit wide, how can an immediate constant operand be embedded in a 32-bit instruction?

If the constant operand has a value of -2048 to -2047 (12'hFFF to 12'h8FF), the operand can be fully specified with a 12-bit binary number in 2's complement form. As it turns out, most constants in computer programs are small. For example, to refer to an offset index of an array, the index often falls within this range of numbers.

In RV32I, I-type instructions have 12 bits reserved for such a constant operand as shown in the slide here. The constant is always **sign exterended** before being used as an operand.

RISC-V: Operand with 32-bit Constants

- Use load upper immediate (lui) and addi
- lui: puts an immediate in the upper 20 bits of destination register and 0's in lower 12 bits

C Code

```
int a = 0xFEDC8765;
```

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

Remember that addi **sign-extends** its 12-bit immediate constant

H&H p306

Using a 12-bit immediate constant works most of the time. However, there are times when a program requires to load a register (say) with a 32-bit constant value.

In RV32I, this is achieved by splitting the constants into two parts – the upper 20 bit, which can be loaded into a register, using the instruction "load upper immediate" `lui`.

For example, the instruction: `lui  s0, 0xFEDC8` load into s0 the value 32'hFEDC8000. This is then added to the bottom 12 bits of the constant with the "add immediate" `addi` instruction:

```
addi  s0, s0 0x765.
```

This works perfectly if the MSB of the 12-bit immediate operand is 0. Unfortunately, if the MSB of the 12-bit constant (i.e. bit 11) is a 1, the constant is then sign extended. When added to the upper 20-bits previously loaded value in s0, the answer will be wrong because the upper 20-bit will be modified. This is because in 2's complement representation, a 20-bit value of 20'hFFFFF is equivalent to -1. Therefore the upper 20-bit, after the addi instruction with be 1 lower than what it should be.

RISC-V: 32-bit Constants (bit 11 is 1)

- If **bit 11** of the constant is **1**, increment upper 20 bits by **1** in `lui`

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC9     # s0 = 0xFEDC9000
addi s0, s0, -341    # s0 = 0xFEDC9000 + 0xFFFFFEAB
                     #    = 0xFEDC8EAB
```

Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

Therefore, if bit 11 of the 32-bit constant is 1, we load the upper 20-bit with a constant that is 1 larger than the constant.

In this example, the constant is 32'hFEDC8EAB. Bit 11 is 1. Upper 20-bit is 20'hFEDC8, and lower 12-bit is 12'hEAB, which is -341 in 2's complement representation after sign extension.

We first load s0 with 0xFEDC9 (1 larger than the upper value). After the addi instruction, s0 will have the correct 32-bit constant value.

Fortunately the assembly and compiler for RISC-V take care of this automatically.

RISC-V: Psuedoinstruction

- Load immediate 32-bit word is tedious.
- Pseudoinstruction – Assembler program translate "Load Immediate" instruction "li" to two real RISC-V instructions: "lui" and "addi"

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V pseudoinstructions

```
# s0 = a
li  s0, 0xFEDC8EAB
```

RISC-V real instructions

```
# s0 = a
lui  s0, 0xFEDC9
addi s0, s0, 0xEAB
```

- RISC-V has many pseudoinstructions (see later lectures)

RISC-V has many instructions missing deliberately to make is small and fast. More complex operations are accomplished by multiple instructions or by an instruction that result in the same operation.

For example there is no instruction to load a register with a constant value. To load s0 with the small constant 6, we use the instruction:

```
addi  s0, zero, 6
```

To load s0 with a large constant 0xFEDC8EAB, we use the two instructions:

```
lui   s0, 0xFEDC9
addi  s0, s0, 0xEAB
```

This makes the assembly language program of RISC-V much harder to read and understand. Fortunately, RISC-V assembler understand a number of pseudo instructions. These instructions do not exist in the RISC-V ISA, but are translated into equivalent RV32I instructions.

To load a register with a constant of any size constant (up to 32 bits), one can use the "load immediate" `li` pseudoinstruction.

```
li    s0, 6
li    s0, 0xFEDC9
```

Slide 29 shows all the pseudo instructions that RISC-V assembler accepts.

RISC-V: Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative

Register Only

- Operands found in registers
  - **Example:** add s0, t2, t3
  - **Example:** sub t6, s1, 0

Immediate

- 12-bit signed immediate used as an operand
  - **Example:** addi s4, t5, -73
  - **Example:** ori t3, t7, 0xFF

H&H p340

Specifying where the operand comes from is called “addressing modes” of an ISA. We have already discussed the two of the four addressing modes found in RISC-V ISA: Register addressing and Immediate addressing. We will now consider the remain two other addressing modes: Base addressing (with offset) and Program Counter Relative addressing.

RISC-V: Base + Offset Addressing

Base Addressing

- Loads and Stores
- Address of operand is:

  base address + immediate

  – **Example:** lw s4, 72(zero)
    - address = 0 + 72

  – **Example:** sw t2, -25(t1)
    - address = t1 - 25

Base addressing mode uses one of the registers content as the address into memory. What stored in the register is not the actual operand, but it stores the address of the operand. In C++, we call this a **pointer** - it points to the place where the operand is stored.

In RISC-V, Base addressing is always used with an offset value which must be a 12-bit 2's complement immediate constant. The "load" and "store" instrutions use this mode of addressing.

RISC-V: PC-relative Addressing

PC-Relative Addressing: branches and jal

Example:

Address		**Instruction**
0x354	L1:	addi s1, s1, 1
0x358		sub t0, t1, s7
...		...
0xEB0		bne s8, s9, L1

The label is (0xEB0-0x354) = 0xB5C (**2908**) instructions **before** bne

The final addressing mode is the Program Counter, or PC-relative addressing. The operand is derived from the PC value by adding a 13-bit (not 12-bit) 2's complement offset. This type of addressing is ONLY used by the branch and jump instructions.

For example, the above "branch if not equal" instruction compares s8 and s9 contents. If they are NOT the same, then the PC counter is load with the address of L1, which is 0x354.

How is the value 0x354 encoded in the instruction? The immediate constant is calculated with the value of PC for the bne instruction, which ix 0xEB0. The offset is calculated by 0xEB0 – 0x354 = 0xB5C. Therefore the stored immediate value is therefore the value -2908.

RISC-V: Instruction coding for Branch offset

Assembly

```
beq s8,  s9,  L1
(beq x24, x25, L1)
```

Relative offset = -2908

$imm_{12:0}$ = -2908 1 0 1 0 0 1 0 1 0 0 1 0 0

bit number 12 11 10 9 8 7 6 5 4 3 2 1 0

Instruction Formats	31	30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0
Branch	[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	[11] opcode

Field Values

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
1100 101	24	25	1	0010 0	99
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Machine Code

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	
1100 101	11000	11001	001	0010 0	110 0011	(0xCB8C9263)
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

The way that RISC-V encodes the relative offset of -2908 is complicated and appears illogical. In fact the design decision for this instruction is very clever and is aimed at making the hardware implementation as simple as possible.

Here are the design constraints that determine how the instruction is encoded:

1. It uses the same fields for opcode (7 bits), funct3 (3 bits), rs1 and rs2 (5 bits) as other instructions. This means that 20 bits of the 32-bit instructions are already used. So there are 12 bits left for encoding the offset.
2. Since the branch destination is ALWAYS an instruction address, and that RISC-V uses byte-addressable memory, the instructions for RV32I is ALWAYS aligned to 4. In other words, there is no need to store the bottom 2 bits of the offset – they are always zero. However, there is a variant of RISC-V ISA which targets microcontroller, where the among of program memory is limited. The "Compressed" extension of RISC-V ISA includes 16-bit instructions (i.e. packing two instructions into a 32-bit word). Therefore, the instruction address can be an increment of 2 instead of 4, meaning that only bit 0 is always 0.
3. It is convenient in hardware that the bits used for encoding B-type immediate values should be similar to that used for I-type and S-type instructions. Therefore the locations of bits are the same for imm[4:1], imm[10:5]. However, the branch immediate is 13 bits instead of 12 bits, therefore imm[12] now takes the place of imm[11] in other case. They are both sign bits.
4. Since imm[0] is always 0, there is no need to store it. Instead imm[11] is

stored here!

R-type Instructions: 3 register instructions

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 − rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << $rs2_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> $rs2_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> $rs2_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2

This an the next few slides are summary of ALL the 40 instructions in RISC-V RV32I ISA.

Here is the R-type instructions that perform arithmetic and logical operations using three registers. They all share the opcode of 51 decimal (or 0x33). The funct3 and funct7 fields defines the specific operation.

I & S-type Instructions: All involve imm constants

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$ = rs2

This group includes two instruction types which both require TWO register operands and one 12-bit immediate operands.

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						

The I-type instructions specify either a load instruction or a ALU instructions. Here we specify a destination register rd to store the result of an memory read of the ALU operation, and a source register rs1 to specify an operand for the ALU operation or the address of the data to fetch.

Two opcodes are used for I-type instructions: 3 for load instructions and 19 for ALU immediate instruction. Note that some I-type instructions (shift instructions) do not use sign-extension to the immediate values.

The S-type instructions does not require a destination register because the destination is data memory. However they require two source registers, one contain the value to write to memory, and a second has the base address of the destination. The 12-bit immediate offset is split into two parts, using the funct7 field of instr[31:25] and the rd field of instr[11:7], combined to form imm[11:0].

B-type Instructions: PC-relative Branches

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						

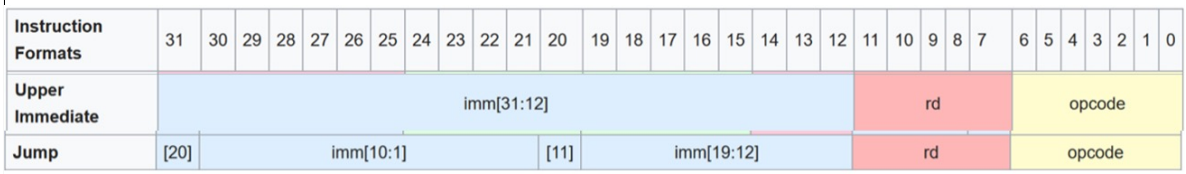
op	funct3	funct7	Type	Instruction	Description	Operation
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA

H&H p311

We have discussed the encoding of branch instructions in details in slide 19 & 20. Note that the opcode for B-type instructions is 99 or 0x63. funct3 defines the conditions under which branch takes place.

When implementing B-type instruction in hardware, one could use the ALU to perform the comparison, or create special branch unit which provides performs ONLY the comparison and no other operations and generates all the required conditions. The second option makes the design cleaner.

U & I -type Instructions: Upper & Jump/Link



Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Upper Immediate	imm[31:12]																				rd					opcode							
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode							

op	funct3	funct7	Type	Instruction	Description	Operation
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

- We will discuss auipc, jalr and jal instructions in another lecture

Finally there are four special instructions that are not in the other category. We have already discussed the lui instruction previously.

The U-type instructions are used to manipulate the upper 20-bit of a register to handle 32-bit immediate constants.

The J-type instructions are for function or subroutine calls. They will be discussed in a later lecture.

RISC-V Arithmetic instructions

Mnemonic	Instruction	Type	Description
ADD rd, rs1, rs2	Add	R	rd ← rs1 + rs2
SUB rd, rs1, rs2	Subtract	R	rd ← rs1 - rs2
ADDI rd, rs1, imm12	Add immediate	I	rd ← rs1 + imm12
SLT rd, rs1, rs2	Set less than	R	rd ← rs1 < rs2 ? 1 : 0
SLTI rd, rs1, imm12	Set less than immediate	I	rd ← rs1 < imm12 ? 1 : 0
SLTU rd, rs1, rs2	Set less than unsigned	R	rd ← rs1 < rs2 ? 1 : 0
SLTIU rd, rs1, imm12	Set less than immediate unsigned	I	rd ← rs1 < imm12 ? 1 : 0
LUI rd, imm20	Load upper immediate	U	rd ← imm20 << 12
AUIP rd, imm20	Add upper immediate to PC	U	rd ← PC + imm20 << 12

The next few slides provide a catalogue of all the RISC-V RV32I instructions in various groups.

All these instructions involve arithmetic operation.

RISC-V Logic instructions

Mnemonic	Instruction	Type	Description
AND rd, rs1, rs2	AND	R	rd ← rs1 & rs2
OR rd, rs1, rs2	OR	R	rd ← rs1 | rs2
XOR rd, rs1, rs2	XOR	R	rd ← rs1 ^ rs2
ANDI rd, rs1, imm12	AND immediate	I	rd ← rs1 & imm12
ORI rd, rs1, imm12	OR immediate	I	rd ← rs1 | imm12
XORI rd, rs1, imm12	XOR immediate	I	rd ← rs1 ^ imm12
SLL rd, rs1, rs2	Shift left logical	R	rd ← rs1 << rs2
SRL rd, rs1, rs2	Shift right logical	R	rd ← rs1 >> rs2
SRA rd, rs1, rs2	Shift right arithmetic	R	rd ← rs1 >> rs2
SLLI rd, rs1, shamt	Shift left logical immediate	I	rd ← rs1 << shamt
SRLI rd, rs1, shamt	Shift right logical imm.	I	rd ← rs1 >> shamt
SRAI rd, rs1, shamt	Shift right arithmetic immediate	I	rd ← rs1 >> shamt

These instructions perform logical operations.

RISC-V Load/Store instructions

Mnemonic	Instruction	Type	Description
LW rd, imm12(rs1)	Load word	I	rd ← mem[rs1 + imm12]
LH rd, imm12(rs1)	Load halfword	I	rd ← mem[rs1 + imm12]
LB rd, imm12(rs1)	Load byte	I	rd ← mem[rs1 + imm12]
LWU rd, imm12(rs1)	Load word unsigned	I	rd ← mem[rs1 + imm12]
LHU rd, imm12(rs1)	Load halfword unsigned	I	rd ← mem[rs1 + imm12]
LBU rd, imm12(rs1)	Load byte unsigned	I	rd ← mem[rs1 + imm12]
SW rs2, imm12(rs1)	Store word	S	rs2(31:0) → mem[rs1 + imm12]
SH rs2, imm12(rs1)	Store halfword	S	rs2(15:0) → mem[rs1 + imm12]
SB rs2, imm12(rs1)	Store byte	S	rs2(7:0) → em[rs1 + imm12]

These instructions perform data memory read and write operations using pointer address in register and an immediate offset.

RISC-V Branch & Jump instructions

Mnemonic	Instruction	Type	Description
`BEQ rs1, rs2, imm12`	Branch equal	SB	`if rs1 == rs2 pc ← pc + imm12`
`BNE rs1, rs2, imm12`	Branch not equal	SB	`if rs1 != rs2 pc ← pc + imm12`
`BGE rs1, rs2, imm12`	Branch greater than or equal	SB	`if rs1 >= rs2 pc ← pc + imm12`
`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	SB	`if rs1 >= rs2 pc ← pc + imm12`
`BLT rs1, rs2, imm12`	Branch less than	SB	`if rs1 < rs2 pc ← pc + imm12`
`BLTU rs1, rs2, imm12`	Branch less than unsigned	SB	`if rs1 < rs2 pc ← pc + imm12 << 1`
`JAL rd, imm20`	Jump and link	UJ	`rd ← pc + 4 pc ← pc + imm20`
`JALR rd, imm12(rs1)`	Jump and link register	I	`rd ← pc + 4 pc ← rs1 + imm12`

These are the branch and jump instructions involving offset to the Program Counter.

RISC-V Psuedoinstructions

Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`
`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]` `ADDI rd, rd, imm[11:0]`
`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]` `ADDI rd, rd, sym[11:0]`
`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`

Mnemonic	Instruction	Base instruction(s)
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`
`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]` `JALR ra, ra, offset[11:0]`
`RET`	Return from subroutine	`JALR zero, 0(ra)`
`NOP`	No operation	`ADDI zero, zero, 0`

These are all the pseudo instructions accepted by the RISC-V assembler but are not really RISC-V instructions in the ISA. They are translated by the RISC-V assembler to one or more RISC-V instructions to make the program more readable.