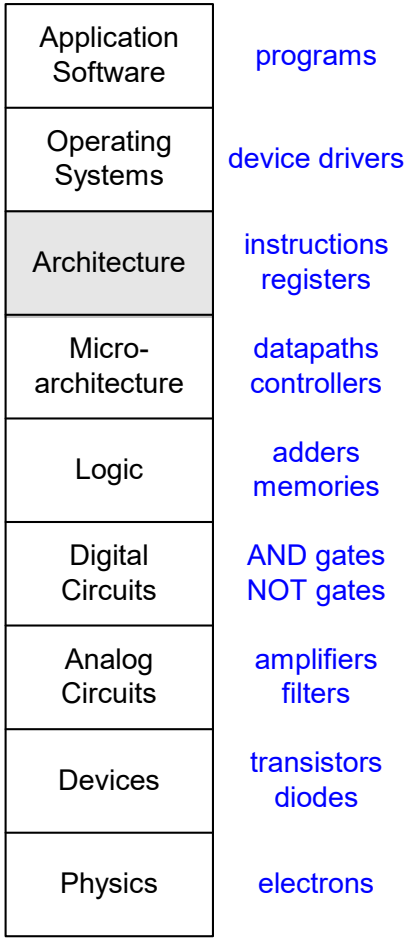
RISC-V Instruction Set Architecture (part 1)

Acknowledgment: Slides are adapted from Harris and Harris textbook instructor's material

Introduction

- Jumping up a few levels of abstraction
- **(Instruction Set) Architecture:** Programmer's view of computer
  - Defined by instructions & operand locations
- **Microarchitecture:** How to implement an architecture in hardware (covered in Chapter 7 of the book)



Assembly Language

- **Instructions:** Commands in a computer's language
  - **Assembly language:** Human-readable format of instructions
  - **Machine language:** Computer-readable format (1's and 0's)

RISC-V Instruction Set Architecture

- 5th generation of RISC design from UC Berkeley, pronounced as RISC-”five”, started in 2010
  - High-quality, license-free, royalty-free RISC ISA specification
  - Standard maintained by non-profit RISC-V foundation (>3000 members that contribute and collaborate on the standard)
  - Both proprietary and open-source core implementations (~$10 billion RISC-V cores in 2022, found in ¼ ASICs and FPGA designs, projected >$60 billion RISC-V cores by 2025)
- Appropriate for all levels of computing systems, from microcontrollers to supercomputers

Once you’ve learned one architecture, it’s easy to learn others (not likely that it will be needed though)!

Underlying Architecture Design Principles

1. **Simplicity favors regularity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

Instructions: Addition

C Code

```
a = b + c;
```

RISC-V assembly code

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

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

Instructions: Subtraction

- Similar to addition - only mnemonic changes

C Code

```
a = b - c;
```

RISC-V assembly code

```
sub a, b, c
```

- **sub:** mnemonic
- **b, c:** source operands
- **a:** destination operand

Design Principle 1: Simplicity Favors Regularity

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

Multiple Instructions

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

C Code

```
a = b + c - d;
```

RISC-V assembly code

```
add t, b, c  # t = b + c
sub a, t, d  # a = t - d
```

Design Principle 2: Make the Common Case Fast

- Base RISC-V ISAs include only simple, commonly used instructions
  - RV32I has 49 instructions (our focus)
  - There are multiple ISA extensions for specialized computing needs (integer multiplication and division, floating point, bit manipulation, multithreaded computing, vector processing, etc.)
- 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)**
  - x86 has > 3500 instruction variants

Instruction Operands

- Operands (based on their physical location)
  - Registers
  - Memory
  - Constants (also called *immediates*)

Operands: Registers

- RISC-V base integer ISA has 32 registers each 32-bit wide
  - RISC-V extensions with 16-bit, 64-bit, and 128-bit register width
- Registers are faster than memory
- We call such ISA a “32-bit architecture” because it operates on 32-bit data

Design Principle 3: Smaller is Faster

- RISC-V includes only a small number of registers
- Commonly used variables kept in registers

RISC-V Register Set

Name	Register Number	Use
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	Temporary registers
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	Temporary registers

- Registers used for specific purposes:
  - `x0` always holds the constant value 0.
  - the *saved registers*, `s0-s11`, used to hold variables
  - the *temporary registers*, `t0 - t6`, used to hold intermediate values during a larger computation
  - Discuss others later

Instructions with Registers

- Revisit add instruction

C Code

```
a = b + c
```

RISC-V assembly code

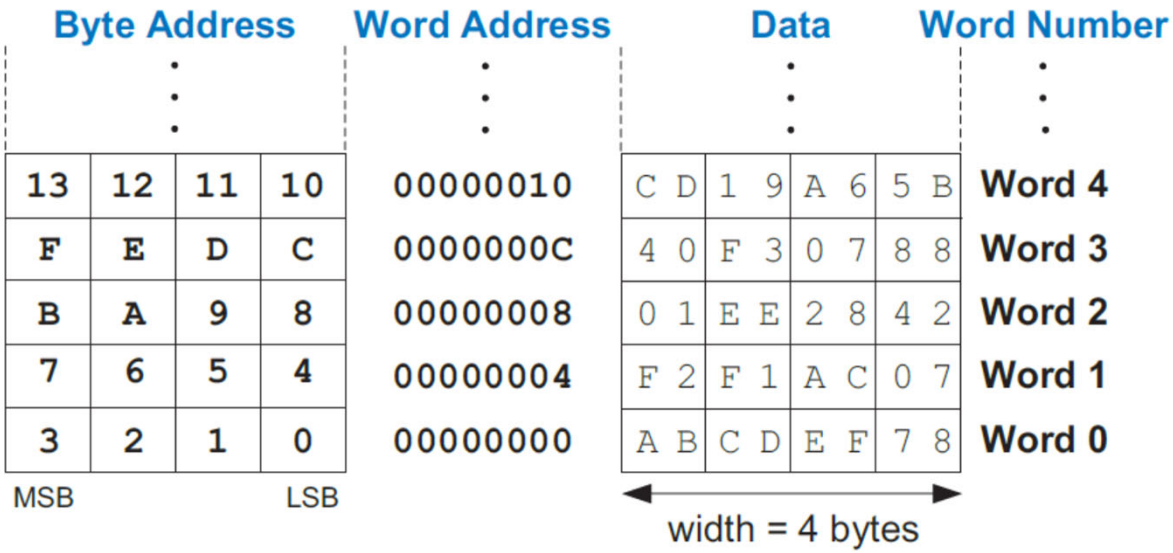
```
# s0 = a, s1 = b, s2 = c
add s0, s1, s2
```

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers

Byte-Addressable Memory

- Note that each 8-bit chuck of data, i.e. a byte, has a unique address, while practical data are typically in chunks of words (i.e., 4 byte wide in 32 bit architecture)
- Word addresses are typically aligned (divisible by 4)
- RISC-V is typically little-endian (as opposed to big-endian order of bytes, discussed in Section 6.6.1 & asked in hw2)



Reading from Memory

- Memory read called ***load***
- **Mnemonic** for reading word of data from memory: *load word* (`lw`)
- **Format:**

```
lw s0, -4(t1)
```

- **Address calculation:**
  - add *base address* (`t1`) to *offset* (- 4)
  - address = `(t1 - 4)`
- **Result:**
  - `s0` holds the value at address (`t1 - 4`)

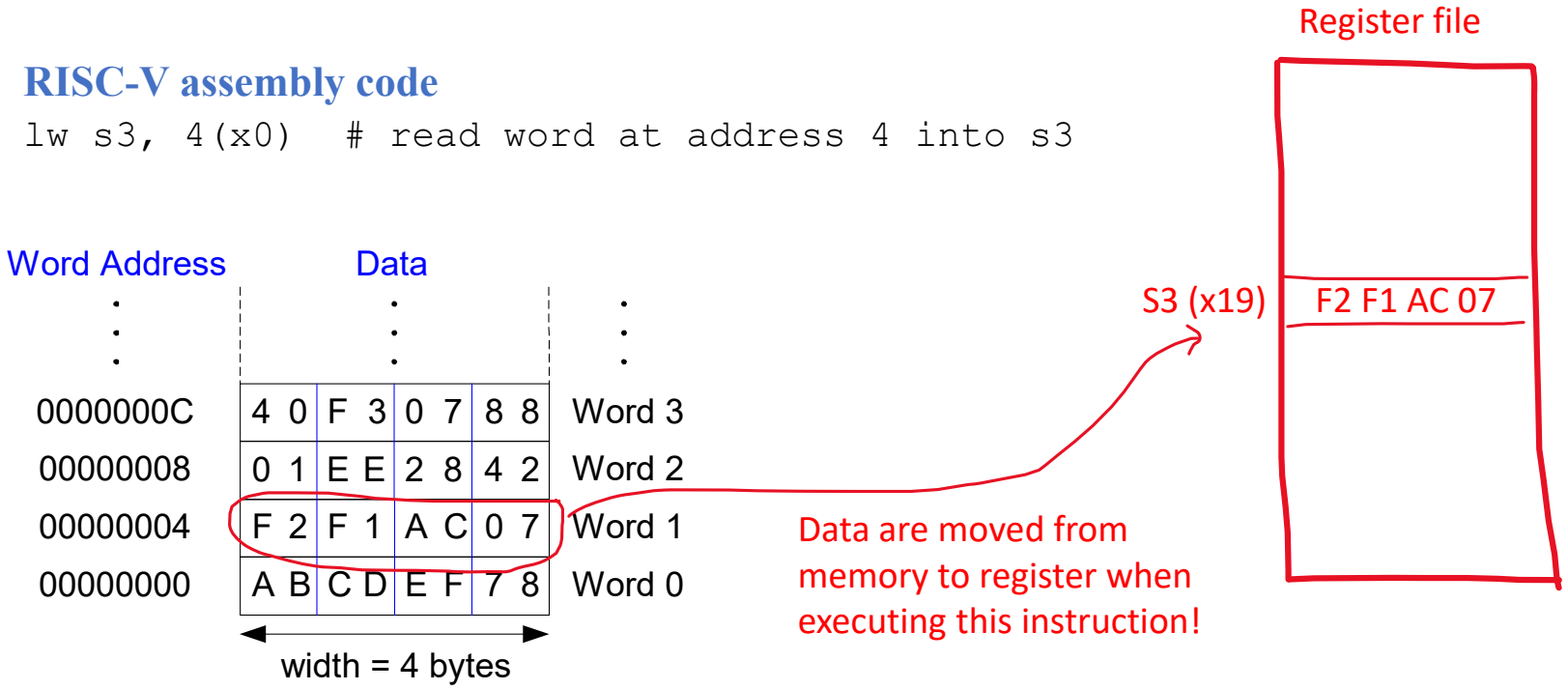
Any register may be used as base address

Reading from Memory Example

- **Example:** Load a word of data at memory address 4 into `s3`
- `s3` holds the value 0xF2F1AC07 after load

RISC-V assembly code

```
lw s3, 4(x0)  # read word at address 4 into s3
```



Writing to Memory

- Memory write are called ***store***
- **Mnemonic** for writing a word of data to memory : *store word* (`sw`)
- **Format:**

```
sw s1, +8(t0)
```

- **Address calculation:**
  - add *base address* (`t0`) to *offset* (+8)
  - address = `(t0 + 8)`
- **Result:**
  - Memory location specified by address (`t0 + 8`) will hold a value of register `s1`

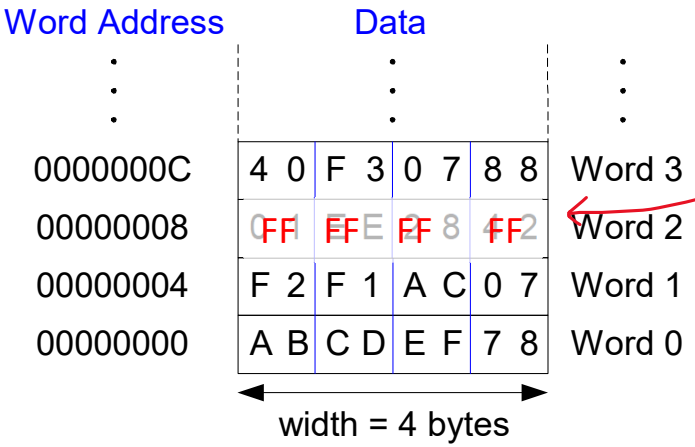
Any register may be used as base address

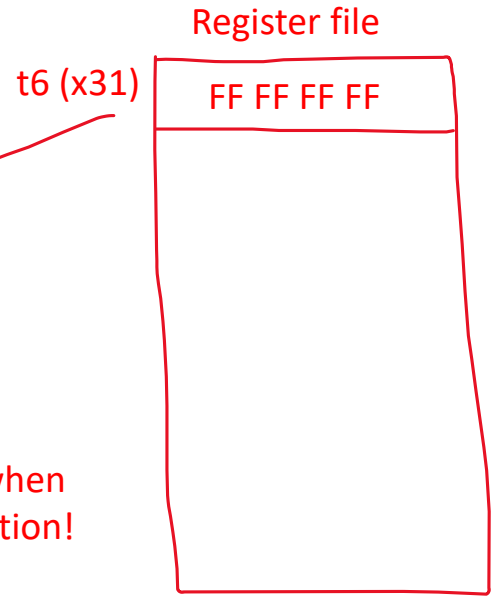
Writing to Memory Example

- **Example:** stores the value held in `t6` into memory address 8

RISC-V assembly code

```
sw t6, 8(x0)   # write t6 into address 8
```





Data are moved from register to memory when executing this instruction!

Design Principle 4: Good Design Demands Good Compromises

- Multiple instruction formats allow flexibility
  - `add`, `sub`: use 3 register operands
  - `lw`, `sw`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster)

Operands: Constants/Immediates

- `lw` and `sw` use constants or *immediates*
- *immediate*ly available from instruction
- 12-bit two's complement number

- New instruction `addi`: Add immediate
- Subtract immediate (`subi`) necessary?

C Code

```
a = a + 4;
b = a - 12;
```

RISC-V assembly code

```
# s0 = a, s1 = b
addi s0, s0, 4
addi s1, s0, -12
```

Control Flow Instructions

- Instructions are typically executed in the sequential order specified in the assembly program
- It is extremely useful to be able to execute instructions out of sequence. This is performed using control flow (branch) instructions.

- Types of branches:
  - **Conditional**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
    - branch less then (`blt`), branch greater then (`bge`) and their unsigned versions
  - **Unconditional**
    - jump (`j`) – pseudo instruction, converted to `jal`
    - jump register (`jalr`) – will be introduced later
    - jump and link (`jal`) – will be introduced later

Conditional Branching (**beq**)

```
# RISC-V assembly
 addi  s0, x0, 0          # s0 = 0
 addi  s1, x0, 0          # s1 = 0
 beq   s0, s1, target     # branch is taken
 addi  s1, s1, 1          # not executed

target:                   # label
 add   s1, s1, s0         # s1 = 0 + 0 = 0
```

Labels indicate instruction location. They can't be reserved words and must be followed by colon (:)

The Branch Not Equal (bne)

```
# RISC-V assembly
 addi  s0, x0, 0          # s0 = 0
 addi  s1, x0, 1          # s1 = 1
 bne   s0, s1, target     # branch is taken
 addi  s1, s1, 1          # not executed

target:                   # label
 add   s1, s1, s0         # s1 = 0 + 1 = 1
```

Unconditional Branching (`j`)

```
# RISC-V assembly
      addi   s0, x0, 4          # s0 = 4
      addi   s1, x0, 0          # s1 = 0
      j      target             # jump to target
      addi   s1, s1, 2          # not executed

  target:
      add    s1, s1, s0         # s1 = 0 + 4 = 4
```

Shift Instructions

- `slli:` shift left logical
  - **Example:** `slli t0, t1, 5  # t0 <= t1 << 5`
- `srli:` shift right logical
  - **Example:** `srli t0, t1, 5  # t0 <= t1 >> 5`
- `srai:` shift right arithmetic
  - **Example:** `srai t0, t1, 5  # t0 <= t1 >>> 5`

- Similar set of instructions (`sll, srl, sra`) in which shift amount comes from register rather than immediate

Logical Instructions

- **`and`, `or`, `xor`**
  - `and`: useful for **masking** bits
    - Masking all but the least significant byte of a value:
      0xF234012F AND 0x000000FF = 0x0000002F
  - `or`: useful for **combining** bit fields
    - Combine 0xF2340000 with 0x000012BC:
      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `xor`: e.g., useful for parity calculations

```
and  rd, rs1, rs2
or   rd, rs1, rs2
xor  rd, rs1, rs2
```

- **`andi`, `ori`, `xori`**
  - Similar instructions in which on of the sources is immediate value
  - Need to convert 12-bit immediate to 32 bit value. 12–bit immediate is zero-extended (*not* sign-extended)

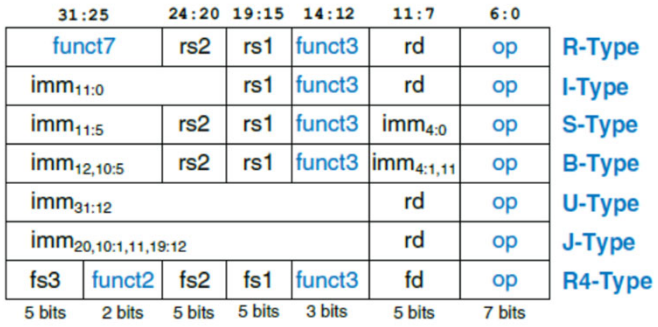
Reading / Storing Byte and Half Words

Instruction	Description	Operation
`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]`$_{7:0}$`)`
`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]`$_{15:0}$`)`
`lw rd, imm(rs1)`	load word	`rd = [Address]`$_{31:0}$
`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]`$_{7:0}$`)`
`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]`$_{15:0}$`)`
`sb rs2, imm(rs1)`	store byte	`[Address]`$_{7:0}$ `= rs2`$_{7:0}$
`sh rs2, imm(rs1)`	store half	`[Address]`$_{15:0}$`= rs2`$_{15:0}$
`sw rs2, imm(rs1)`	store word	`[Address]`$_{31:0}$`= rs2`

- `Address:` memory address: **rs1** + SignExt(**imm**$_{11:0}$)
- `[Address]:` data at memory location Address

RISC-V Base Instruction Reference Sheet

- 2nd page of Harris & Harris book
- More instructions:
  - LUI, SLT, SLTI, SLTU, SLTUI, AUIPC



- `imm`: signed immediate in $imm_{11:0}$
- `uimm`: 5-bit unsigned immediate in $imm_{4:0}$
- `upimm`: 20 upper bits of a 32-bit immediate, in $imm_{31:12}$
- `Address`: memory address: rs1 + SignExt($imm_{11:0}$)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({$imm_{12:1}$, 1'b0})
- `JTA`: jump target address: PC + SignExt({$imm_{20:1}$, 1'b0})
- `label`: text indicating instruction address
- `SignExt`: value sign-extended to 32 bits
- `ZeroExt`: value zero-extended to 32 bits
- `csr`: control and status register

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)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	`[Address]7:0 = rs27:0`
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	`[Address]15:0 = rs215:0`
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	`[Address]31:0 = rs2`
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 << rs24:0`
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 >> rs24:0`
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>> rs24:0`
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`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
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`
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`

*Encoded in $instr_{31:25}$, the upper seven bits of the immediate field