



NATIONAL RESEARCH
UNIVERSITY



Computer Architecture and Operating Systems

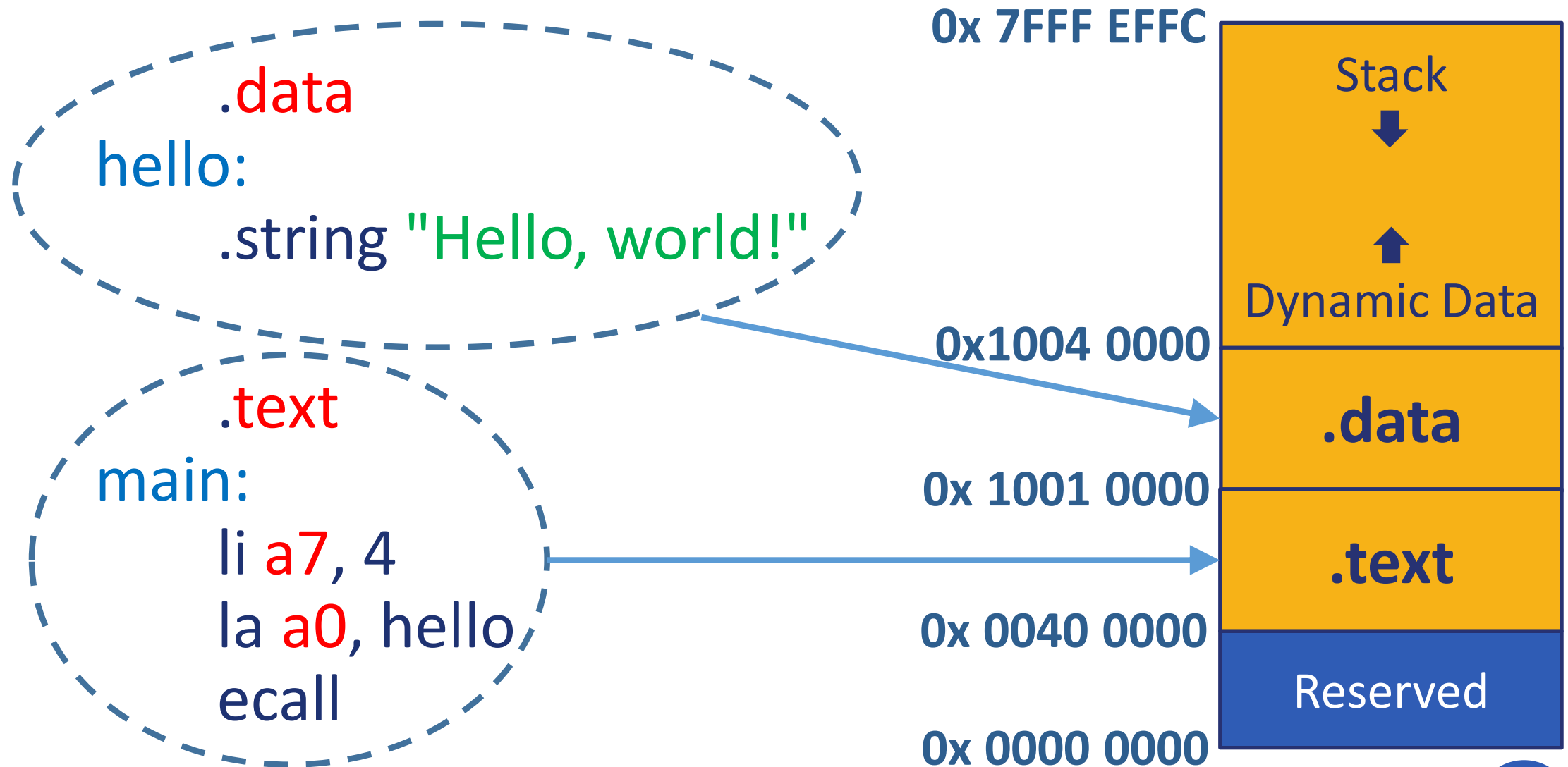
Lecture 5: Assembly Programming – Branches and Arrays

Andrei Tatarnikov

atatarnikov@hse.ru

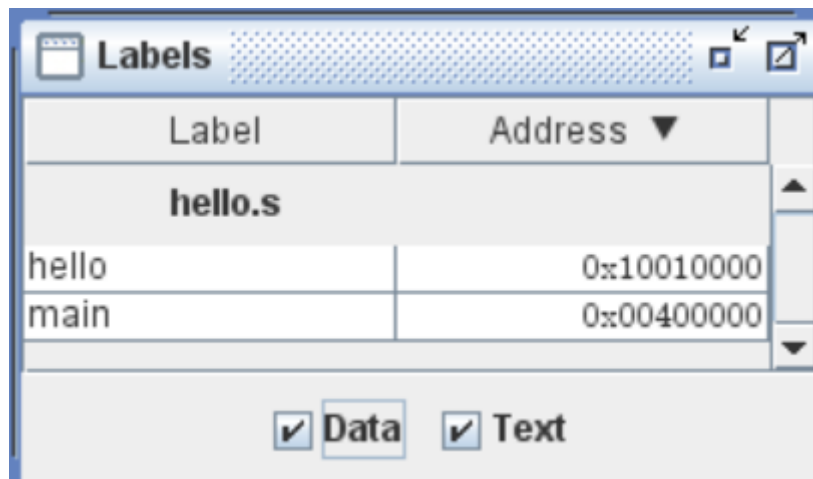
[@andrewt0301](https://twitter.com/andrewt0301)

Program Structure and Memory Layout



Labels

- **Labels** are symbolic names for addresses (in the .data or .text segment).
- **Labels** are used by control-flow instructions (branches and jumps).
- **Labels** are used by load and store instructions.



The screenshot shows a window titled 'Labels' with a table of labels. The table has two columns: 'Label' and 'Address'. The file 'hello.s' is selected. The table lists two labels: 'hello' at address 0x10010000 and 'main' at address 0x00400000. At the bottom, there are checkboxes for 'Data' and 'Text', both of which are checked.

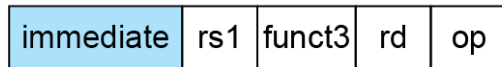
Label	Address ▼
hello.s	
hello	0x10010000
main	0x00400000

☒ Data ☒ Text

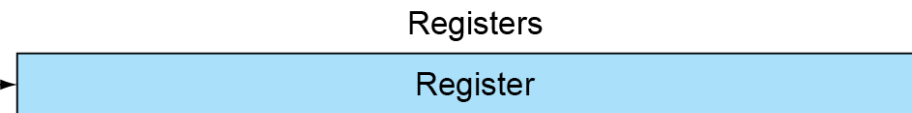
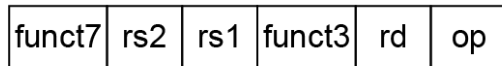
Addressing

Addresses can be represented in several ways

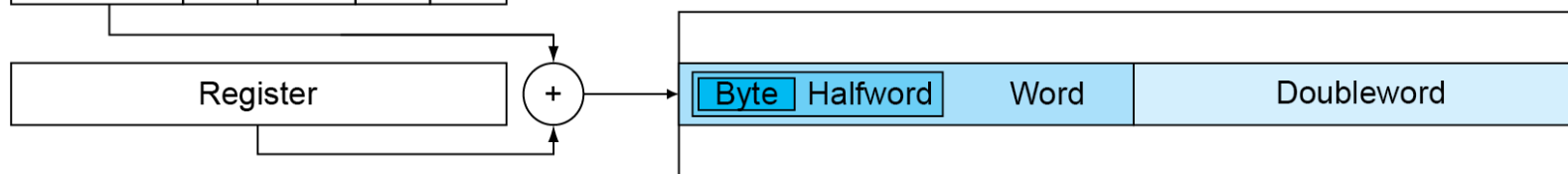
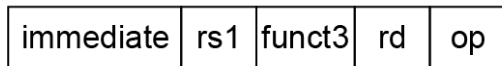
1. Immediate addressing



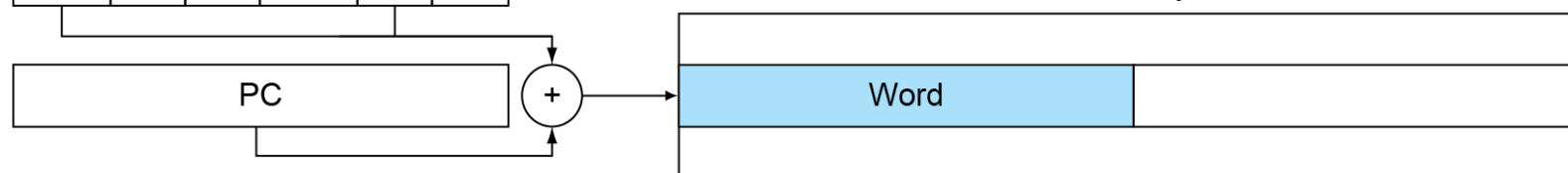
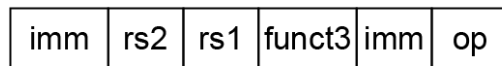
2. Register addressing



3. Base addressing



4. PC-relative addressing



Program Counter

- **Program Counter (PC)** is a special register that stores the address of the currently executed instruction.
- When an instruction is executed, the PC is incremented by the size of the instruction (4 bytes) to point to the next instruction.
- Branch and jump instructions assign to the PC new addresses to change the control flow.
- Branch instructions use PC-relative addresses (increment or decrement current value by an offset).

Branch Instructions

Branch Instructions

- Branch = `beq rs1, rs2, label`
- Branch \neq `bne rs1, rs2, label`
- Branch $<$ `blt rs1, rs2, label`
- Branch \geq `bge rs1, rs2, label`
- Branch $<$ Unsigned `bltu rs1, rs2, label`
- Branch \geq Unsigned `bgeu rs1, rs2, label`

Branch Pseudoinstructions

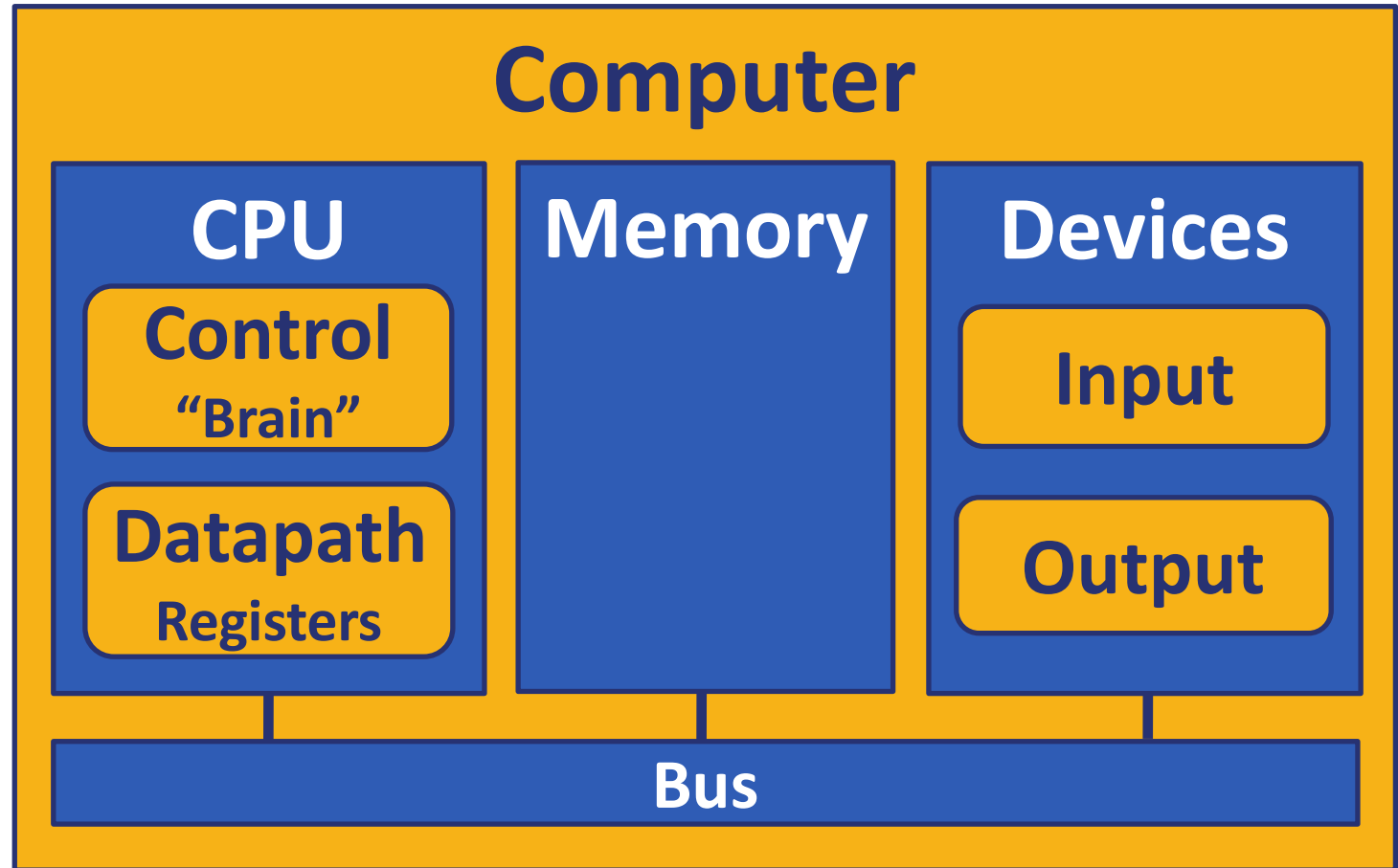
Branch Pseudoinstructions

- TODO

How Computer Works

Main Parts:

- Control
- Datapath
- Memory
- Input
- Output



Stored Program Concept

- 32-bit instructions and data stored in memory
- Program is a sequence of instructions
- To run a new program:
 - Simply load the new program into memory
- Program Execution:
 - CPU fetches (reads) instructions from memory in sequence
 - CPU performs the specified operations

Stored Program Representation

Assembly Code

```
lw  t0, 32 (t1)
add s1, s0, s2
addi t0, s3, -12
sub  t0, t3, t5
```

Machine Code

```
0x02032283
0x012404B3
0xFF498293
0x41EE02B3
```

Stored Program

Program Counter (PC):
keeps track of current
instruction

Memory

Address	Instructions
...	...
0x00400000	0x02032283
0x00400004	0x012404B3
0x00400008	0xFF498293
0x0040000C	0x41EE02B3
...	...

RISC-V Instructions

Name	Description	Version	Status
Base			
RVWMO	Weak Memory Ordering	2.0	Ratified
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open
Extensions			
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified
A	Standard Extension for Atomic Instructions	2.1	Ratified
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified
G	Shorthand for the base integer set (I) and above extensions (MAFD)	N/A	N/A
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified
C	Standard Extension for Compressed Instructions	2.0	Ratified
ZiCSR	Control and Status Register (CSR)	2.0	Ratified
Zifencei	Instruction-Fetch Fence	2.0	Ratified
And more standard and custom extensions...			

Design Principles

- **Design Principle 1:** Simplicity favors regularity
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost
- **Design Principle 2:** Make the common case fast
 - Most common cases affect the performance the most
- **Design Principle 3:** Smaller is faster
 - 32 registers, fewer instructions
- **Design Principle 4:** Good design demands good compromises
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

Six Instruction Formats

- **R-format:** instructions using 3 register inputs
 - add, xor, mul - arithmetic/logical ops
- **I-format:** instructions with immediates, loads
 - addi, lw, jalr, slli
- **S-format:** store instructions
 - sw, sb
- **SB-format:** branch instructions
 - beq, bge
- **U-format:** instructions with upper immediates
 - lui, auipc - upper immediate is 20-bits
- **UJ-format:** the jump instruction
 - jal

R-format Instructions

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

add x9, x20, x21

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

0000 0001 0101 1010 0000 0100 1011 0011_{two} = 015A04B3₁₆

■ Arithmetic Instructions

- opcode: operation code
- rd: destination register number
- funct3: 3-bit function code (additional opcode)
- rs1 and rs2: first and second source register 5-bit numbers
- funct7: 7-bit function code (additional opcode)

I-format Instructions

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

addi t0, t1, 123

0x123	6	0	5	19
000100100011	00110	000	00101	0010011

0001 0010 0011 0011 0000 0010 1001 0011_{two} = 0x12330293₁₆

- Immediate arithmetic and load instructions
 - rs1: source or base address register number
 - immediate: constant operand, or offset added to base address
 - 2s-complement, sign extended

S-format Instructions

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
-----------	-----	-----	--------	----------	--------

7 bits

5 bits

5 bits

3 bits

5 bits

7 bits

sw t0, 4(t1)

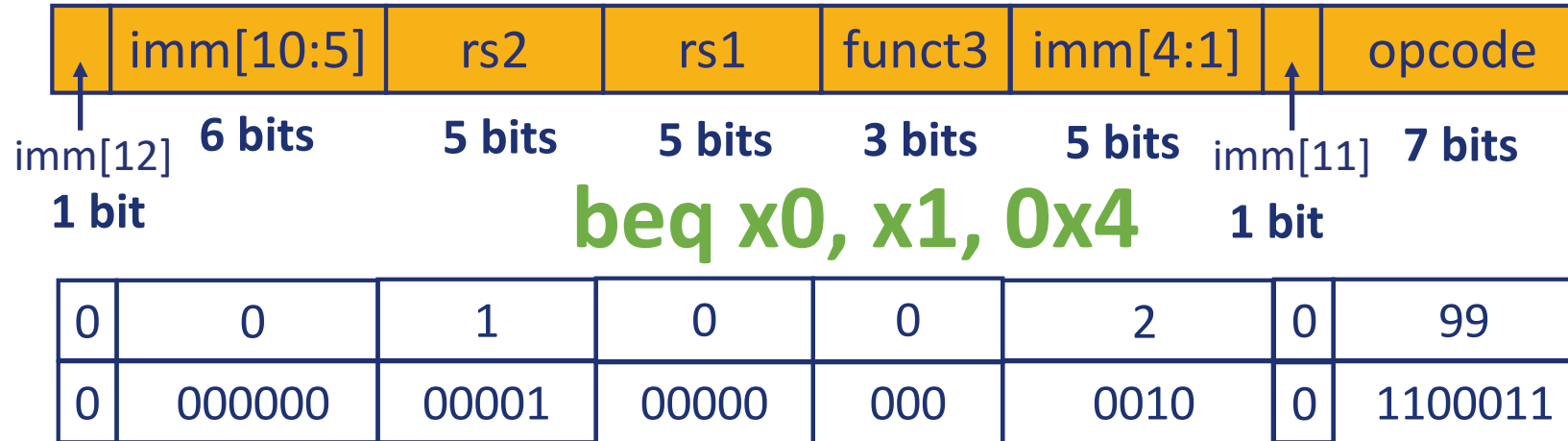
0	5	6	2	4	35
0000000	00101	00110	010	00100	100011

0000 0000 0101 0011 0010 0010 0010 0011_{two} = 0x00532223₁₆

■ Different immediate format for store instructions

- rs1: base address register number
- rs2: source operand register number
- immediate: offset added to base address
 - Split so that rs1 and rs2 fields always in the same place

SB-format Instructions



0000 0000 0001 0000 0000 0010 0110 0011_{two} = **00100263**₁₆

- Branch instructions specify
 - Opcode, two registers, target address
- Most branch targets are near branch
 - Branch range is +/- 4KB
- PC-relative addressing
 - Target address = PC + immediate × 2

U-format Instructions

imm[31:12]	rd	opcode
------------	----	--------

20 bits

5 bits

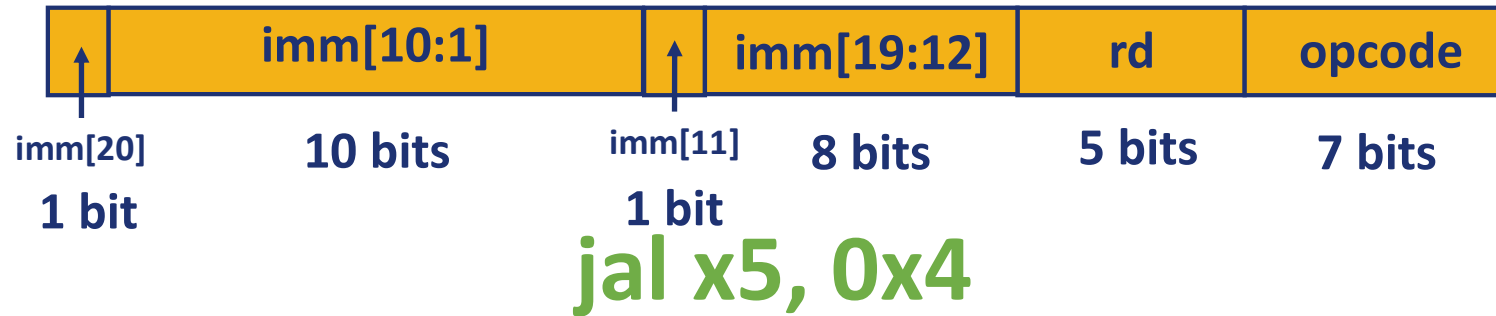
lui x5, 0x12345

0x12345	5	55
0001 0010 0011 0100 0101	00101	0110111

0001 0010 0011 0100 0101 0010 1011 0111_{two} = 123452b7₁₆

- Upper-immediate values: 20-bit values shifted left by 12 bits
 - opcode: operation code
 - rd: destination register number
 - imm: 20-bit immediate value

UI-format Instructions



0	2	0	0	5	111
0	0000000010	0	00000000	00101	1101111

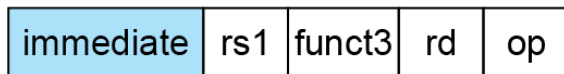
0000 0000 0100 0000 0000 0010 1110 1111_{two} = 004002ef₁₆

■ Jump instructions

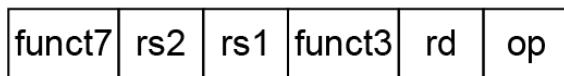
- opcode: operation code
- rd: destination link register
- imm: signed offset in multiples of 2 bytes added to PC
- Target address = PC + immediate × 2
- Jump range is +/- 1MB

RISC-V Addressing Summary

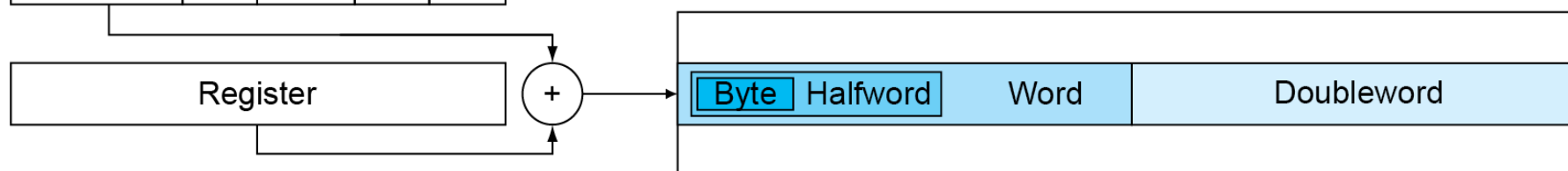
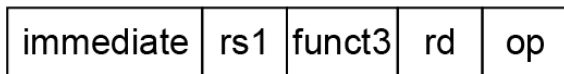
1. Immediate addressing



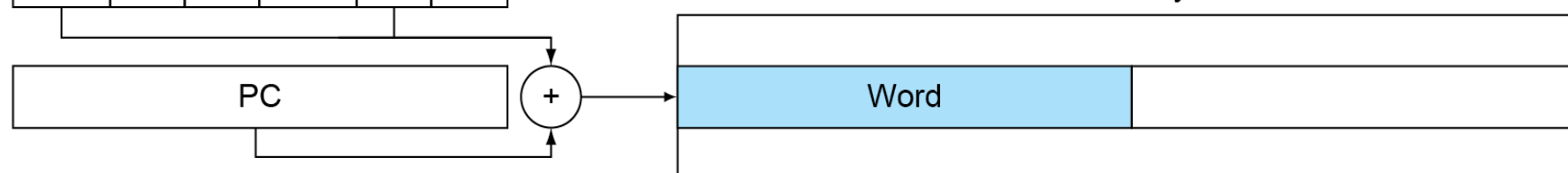
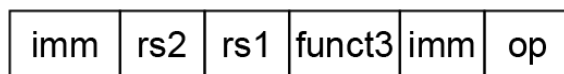
2. Register addressing



3. Base addressing



4. PC-relative addressing



Macros

Name (Field Size)	Field					Comments	
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Any Questions?

```
        .text
__start:  addi t1, zero, 0x18
          addi t2, zero, 0x21
cycle:    beq t1, t2, done
          slt t0, t1, t2
          bne t0, zero, if_less
          nop
          sub t1, t1, t2
          j cycle
          nop
if_less:  sub t2, t2, t1
          j cycle
done:     add t3, t1, zero
```