Representing Instructions in the Computer

- We have seen the syntax of various assembly language instructions
  - Ex. add t0, s1, s2
- We also know that inside the computer, everything must be in the form of a binary number (ones and zeros)
- The add instruction above becomes the following 32-bit binary number in RISC-V

00000001001001001000001010110011
(0x102452b3)

What Do All The Bits Mean?

- The 32-bit instruction can be divided into sections or fields of bits as follows

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
0000000	10010	01001	000	00101	0110011
1	2	3	4	5	6

- Each smaller field of bits serves a specific purpose as a result of its location in the instruction
    - fields 1, 4 & 6 = operation (add)
    - field 2 = 2nd source register (s2 = x18)
    - field 3 = 1st source register (s1 = x9)
    - field 5 = destination register (t0 = x5)

Instruction Field Names

- The six fields of the binary add instruction each relate to the operation and operands specified in the assembly language instruction

funct7	rs2	rs1	funct3	rd	opcode

  - opcode: specifies basic operation and instruction format
  - rd: destination register
  - funct3: additional opcode field
  - rs1: first source register
  - rs2: second source register
  - funct7: additional opcode field

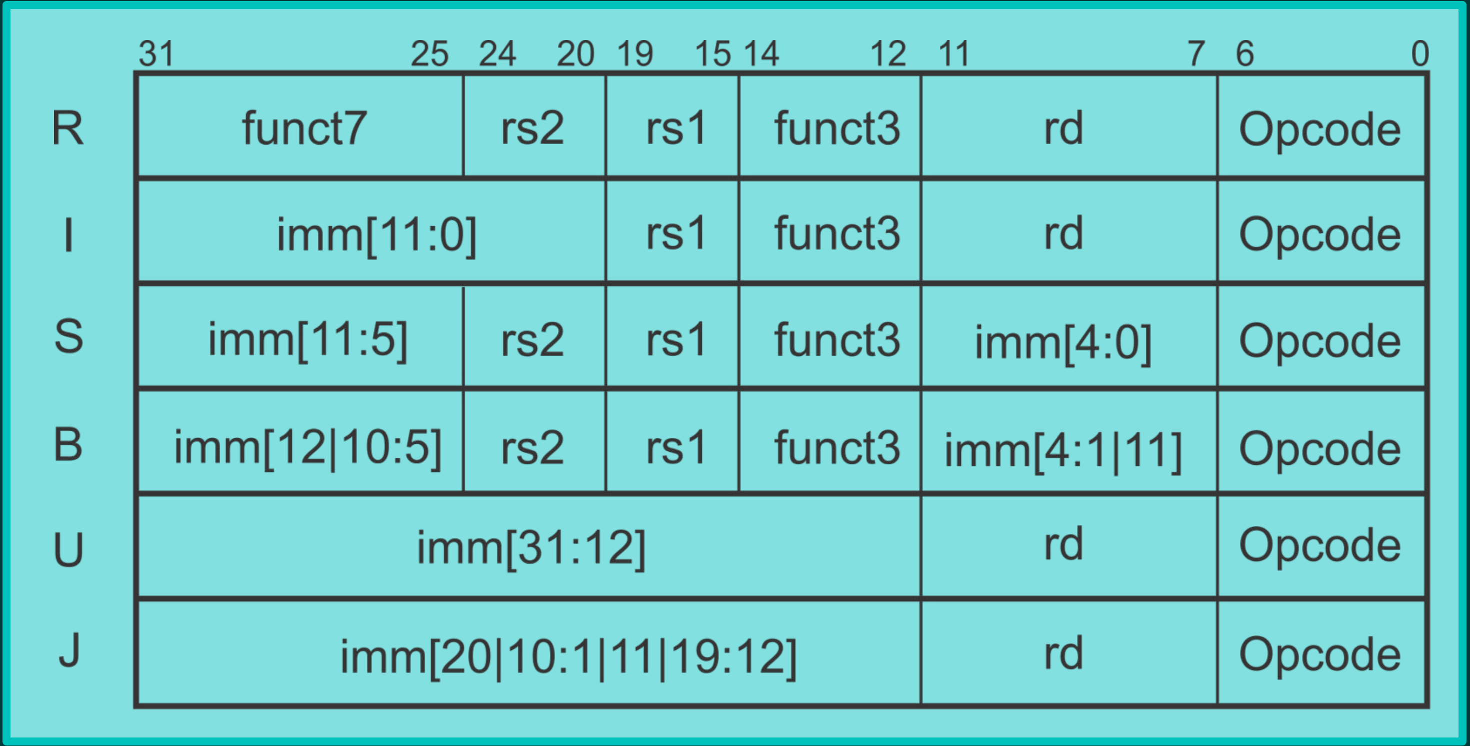
Instruction Formats

- The layout of an instruction is called the *instruction format*
- The simplest instruction set design would have
  - all instructions fixed length
  - only one instruction format (same number of fields in every instruction)
- But this is too restrictive
  - Different instructions will reference varying number of operands of various sizes
  - Ex.

    ```
    add   t2, t1, t0       jal   ra, gohere
    addi  t3, t4, 1        lw    s1, 4(s0)
    ```

Instruction Formats (continued)

- The RISC-V instruction set design was meant to be simple
  - It uses fixed size formats
  - 32 bits for all instructions
- However, RISC-V has six instruction formats



	31 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	Opcode
B	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	Opcode
U	imm[31:12]				rd	Opcode
J	imm[20|10:1|11|19:12]				rd	Opcode

Instruction Formats (continued)

- Why six?
  - Just one format is too restrictive for fixed sized instruction
    - Need to represent multiple operands that differ for different instructions
    - Registers (5 bits to access 32 registers)
      - some instructions specify 3, others only 2, some only 1
    - Immediates (12 bits for constants and memory offsets)
    - Some instructions need more bits for constant
      - some instructions specify 20 bit immediates

- Design Principle #3: *Good design demands good compromises*

RISC-V Instruction Formats

- R-format: for register-register arithmetic operations
  - Ex. add, sub, sll, srl, slt
- I-format: for register-immediate arithmetic operations and loads
  - Ex. addi, ori, lw, lh, lb
- S-format: for stores
  - Ex. sw, sh, sb
- B-format: for branches (minor variant of S-format, called SB initially)
  - Ex. beq, bne, blt
- U-format: for 20-bit upper immediate instructions
  - Ex. lui, auipc
- J-format: for jumps (minor variant of U-format, called UJ initially)
  - Ex. jal

R Instruction Format

- Register - register
- Format for arithmetic and logic instructions

funct7	rs2	rs1	funct3	rd	opcode

  - opcode: partially specifies what instruction it is
    - This field is equal to 0110011 for all R-Format register – register arithmetic instructions
  - funct7 + funct3: combined with opcode, these two fields describe what operation to perform
  - rd: destination register
  - rs1: first source register
  - rs2: second source register

Example R Format Instructions

add	0000000	rs2	rs1	000	rd	0110011
sub	0100000	rs2	rs1	000	rd	0110011
sll	0000000	rs2	rs1	001	rd	0110011
slt	0000000	rs2	rs1	010	rd	0110011
sltu	0000000	rs2	rs1	011	rd	0110011
xor	0000000	rs2	rs1	100	rd	0110011
srl	0100000	rs2	rs1	101	rd	0110011
sra	0000000	rs2	rs1	101	rd	0110011
or	0000000	rs2	rs1	110	rd	0110011
and	0000000	rs2	rs1	111	rd	0110011

Different encoding in funct7 + funct3 selects different operations

I Instruction Format

- Register – immediate
- Format for register-immediate arithmetic and loads; also for jump and link register (jalr)

imm[11:0]	rs1	funct3	rd	opcode

  - imm[11:0] = 12-bit signed constant
  - imm[11:0] can hold values in range -2048 to +2047
  - Immediate is always sign-extended to 64-bits before use in an arithmetic operation
  - For loads, 12-bit signed immediate is added to the base address in register rs1 to form the memory address
  - This is very similar to the add-immediate operation but used to create address not to create final result
  - The value loaded from memory is stored in register rd

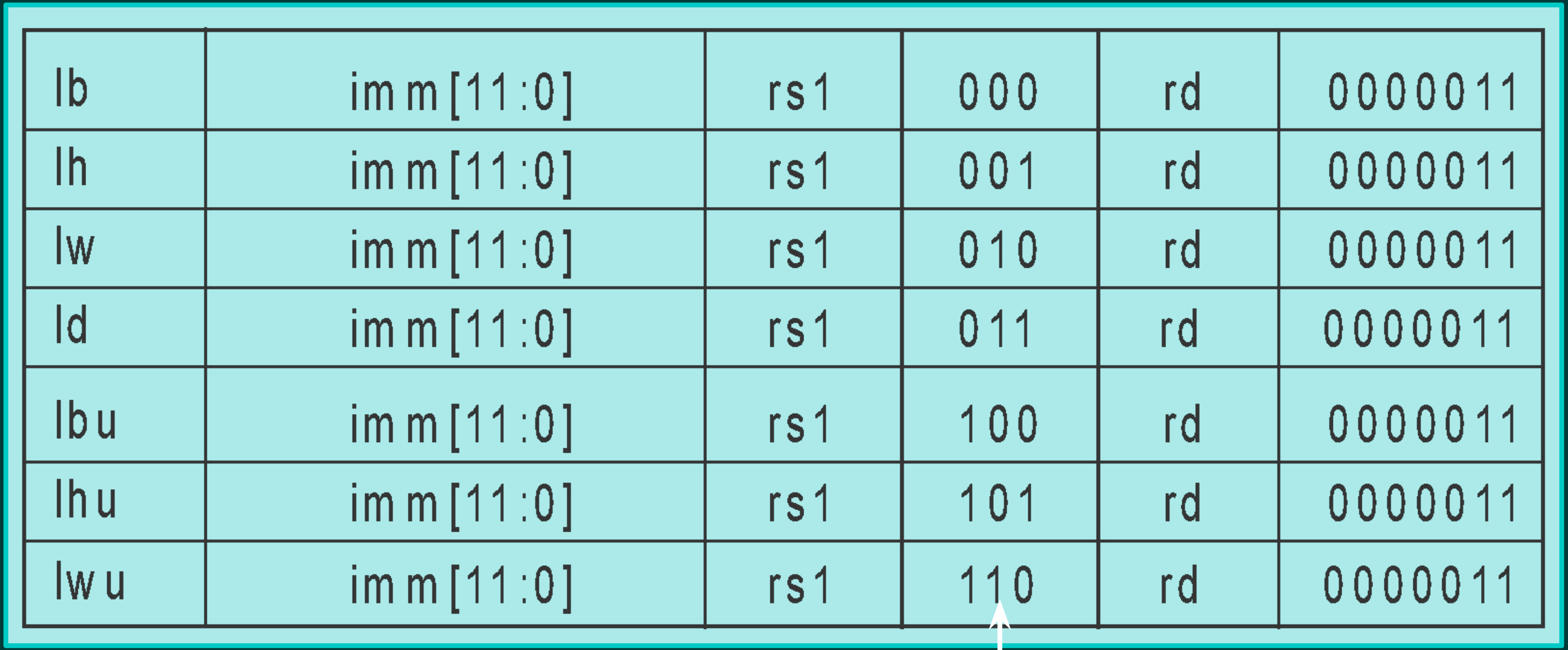
Example I Format Instructions

addi	imm[11:0]		rs1	000	rd	0010011
slti	imm[11:0]		rs1	010	rd	0010011
sltiu	imm[11:0]		rs1	011	rd	0010011
xori	imm[11:0]		rs1	100	rd	0010011
ori	imm[11:0]		rs1	110	rd	0010011
andi	imm[11:0]		rs1	111	rd	0010011
slli	0000000	shamt	rs1	001	rd	0010011
srli	0000000	shamt	rs1	101	rd	0010011
srai	0100000	shamt	rs1	101	rd	0010011

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 6 bits of the immediate value for shift amount (can only shift by 0-63 bit positions)

Load Instructions



lb	imm[11:0]	rs1	000	rd	0000011
lh	imm[11:0]	rs1	001	rd	0000011
lw	imm[11:0]	rs1	010	rd	0000011
ld	imm[11:0]	rs1	011	rd	0000011
lbu	imm[11:0]	rs1	100	rd	0000011
lhu	imm[11:0]	rs1	101	rd	0000011
lwu	imm[11:0]	rs1	110	rd	0000011

funct3 field encodes size and sign of load data

- LBU is “load unsigned byte”
- LH is “load halfword”, which loads 16 bits (2 bytes) and sign-extends to fill destination 64-bit register
- LHU is “load unsigned halfword”, which zero-extends 16 bits to fill destination 64-bit register
- LWU is “load word unsigned”, which zero-extends 32 bits to fill destination 64-bit register

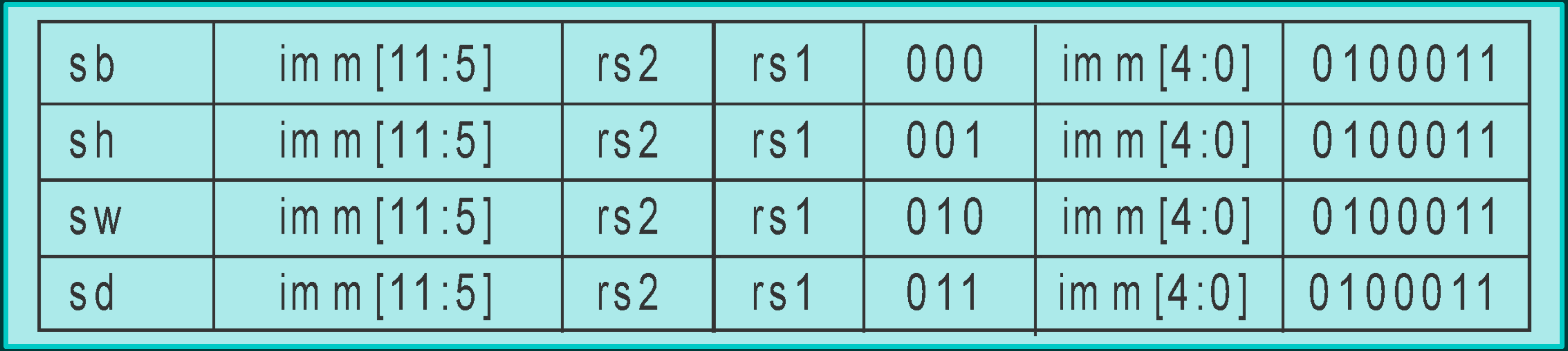
S Instruction Format

- Store
- Format for store instructions

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

  - Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well as need immediate offset!
  - Can’t have both rs2 and immediate in same place as other instructions!
  - Note that stores don’t write a value to the register file, no rd!
  - RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – keep rs1 & rs2 fields in same place
  - Register names more critical than immediate bits in hardware design

Store Instructions



sb	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011
sh	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011
sw	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
sd	imm[11:5]	rs2	rs1	011	imm[4:0]	0100011

Different opcode value from loads

B Instruction Format

- Branch
- Format for branch instructions

Imm[12]	Imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

  - Modification of the S instruction format
  - Opcode, funct3, rs1 and rs2 fields are located in the same fields as other instruction formats
  - Primary difference is the location of the bits of the immediate
  - Immediate represents the offset for the target instruction of the branch

B Format Immediate Bits

- In all instructions where there was an immediate field, it contained 12 bits numbered 0 – 11
- Careful examination of the 12 bits of the subdivided immediate field in the B format shows the numbering 1 – 12
- The 12 immediate bits encode ***even*** 13-bit signed byte offsets
  - Lowest bit of offset is always zero, so no need to store it
  - This arrangement accommodates the compressed instruction format defined in the C extension
    - 16-bit (2-byte) instructions are defined

RISC-V C Extension: 16-bit instructions

- Extension to RISC-V base ISA supports 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length
- To enable this, RISC-V scales the branch offset by 2 bytes even when there are no 16-bit instructions
- Reduces branch reach by half and means that ½ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach $\pm 2^{10} \times$ 32-bit instructions either side of PC

Branch Target Calculation Example

- Code sequence defining a loop

```
loop:  beq   s3, a0, end
       add   s2, s2, a0
       addi  s3, s3, -1
       beq   zero, zero, loop
end:   # target instruction
```

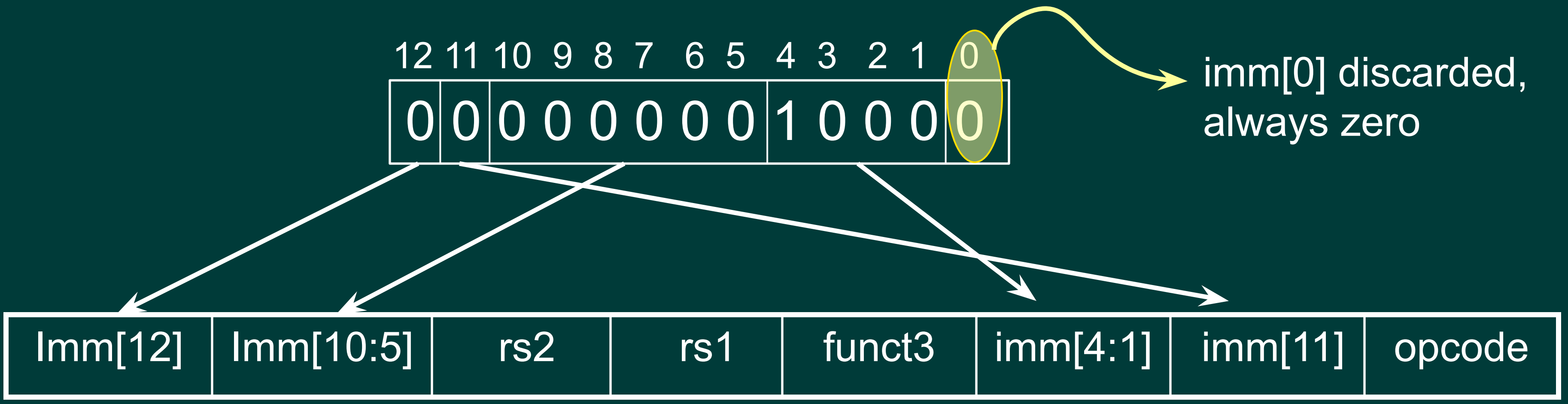
1
2
3
4

count instructions from branch

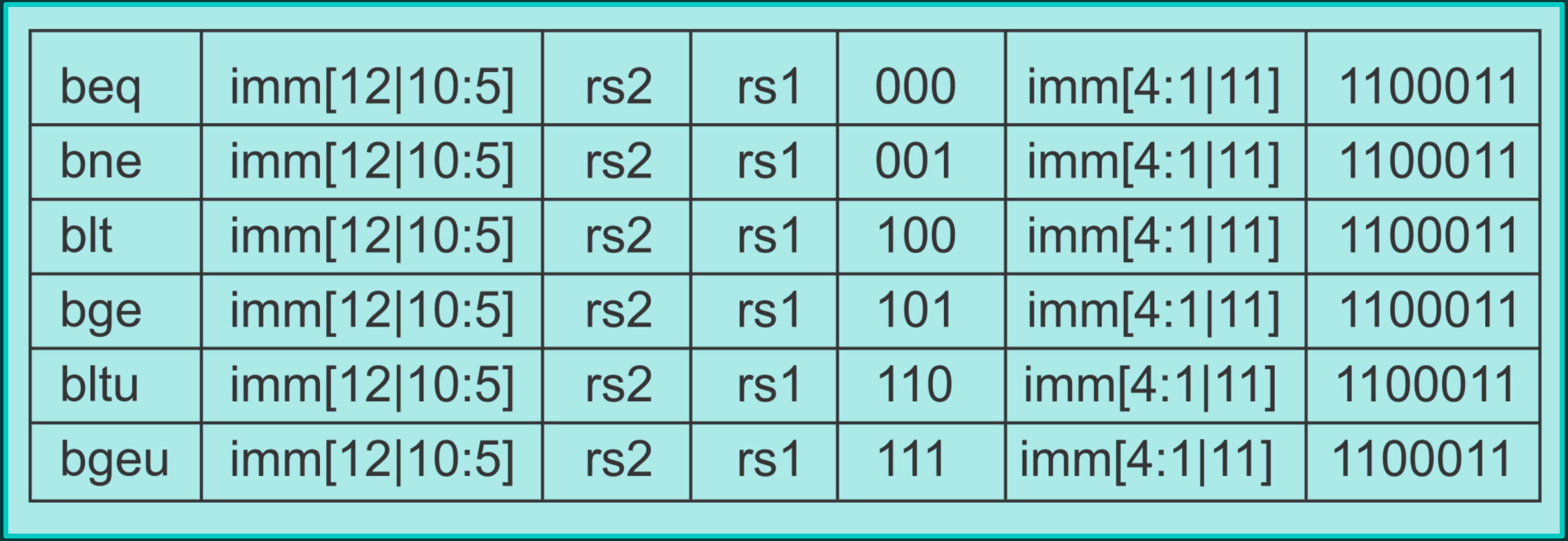
- Branch offset = 4 × 32-bit instructions = 16 bytes
  - 32 bits = 4 bytes; 4 x 4 = 16
  - Branch with offset of 0, branches to itself

Branch Target Calculation Example

- Offset value = 16 encoded in 13 bits
  - 13-bit immediate, imm[12:0], with value 16



Branch Instructions



beq	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011
bne	imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011
blt	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011
bge	imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011
bltu	imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011
bgeu	imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011

PC-Relative Addressing

- After the branch target offset has been calculated, it must be transformed into the memory address of the target instruction.
- The offset is added to the value of the program counter which holds the address of the next instruction after the branch.
- This makes the range of addresses that are reachable from the branch instruction relative to the current location in the program.
- This defines a type of addressing called PC-relative.

U Instruction Format

- Upper immediate
- Format for load upper immediate and add upper immediate to PC

imm[31:12]	rd	opcode

  - Has 20-bit immediate in upper 20 bits of 32-bit instruction word
  - One destination register, rd
  - lui writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
  - auipc adds upper immediate value to PC and places result in destination register
    - used for programmer controlled PC-Relative addressing
  - Used with a second instruction (addi, ori), you can generate any 32-bit memory address

Upper Immediate Instructions

lui	imm[31:12]	rd	0110111
auipc	imm[31:12]	rd	0010111

AUIPC Instruction

- Add upper immediate to PC
- This instruction adds a 20-bit immediate value to the upper 20 bits of the program counter
- This instruction enables PC-relative addressing in RISC-V
- To form a complete 32-bit PC-relative address, auipc forms a partial result
- A subsequent addi instruction adds in the lower 12 bits to complete the 32-bit address

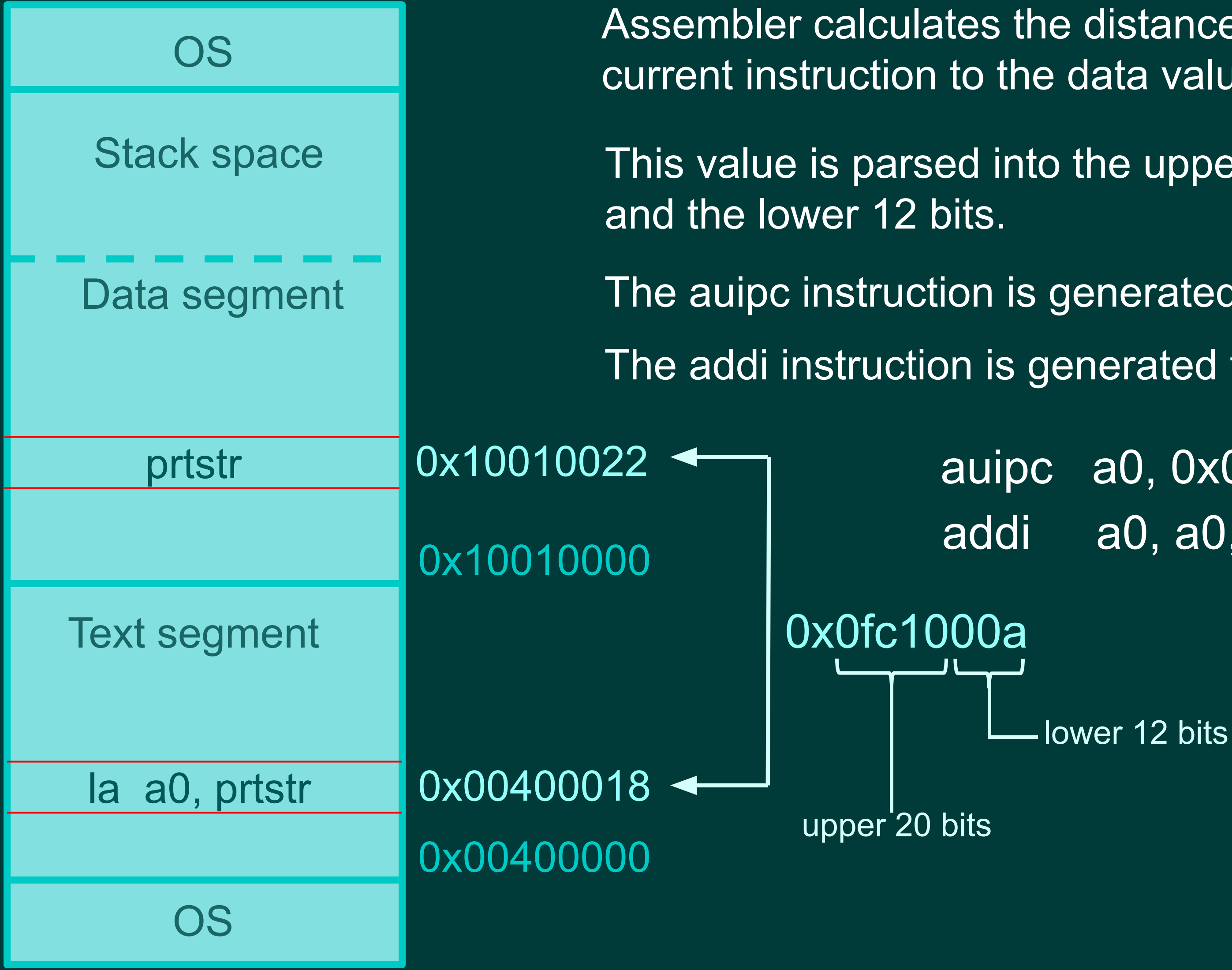
Load Address Conversion

Assembler calculates the distance from the current instruction to the data value.

This value is parsed into the upper 20 bits and the lower 12 bits.

The auipc instruction is generated.

The addi instruction is generated to add 10.



OS

Stack space

Data segment

prtstr

Text segment

la a0, prtstr

OS

0x10010022

0x10010000

0x00400018

0x00400000

0x0fc1000a

lower 12 bits

upper 20 bits

```
auipc   a0, 0x0fc10
addi    a0, a0, 0x10
```

J Instruction Format

- Unconditional jump
- Format for jump and link (jal) instruction

Imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

  - jal saves PC+4 in register rd (the return address)
  - Set PC = PC + offset (PC-relative branch)
  - Target somewhere within ±2^{19} locations, 2 bytes apart
    - ±2^{18} 32-bit instructions
  - Immediate encoding optimized similar to branch instruction to reduce hardware cost
  - jal instruction used for subroutine calls

JALR

- Format for jump and link register (jalr) instruction

imm[11:0]	rs1	000	rd	opcode

- The jump and link register actually uses the I-format
  - The target address is obtained by adding the sign-extended 12-bit immediate to the register rs1, then setting the least-significant bit of the result to zero
  - This instruction is classified as an indirect jump

Jump Instructions

<table>
<tr><td>jal</td><td colspan="3">imm[20|10:1|11|19:12]</td><td>rd</td><td>1101111</td></tr>
<tr><td>jalr</td><td>imm[11:0]</td><td>rs1</td><td>000</td><td>rd</td><td>1100111</td></tr>
</table>

The Big Picture

- Computers are designed based on two key principles
  - Instructions are represented internally as numbers
  - Programs can be stored in memory, read and written as numbers
- This is called the stored-program concept
  - It was the first evolutionary change from which subsequent computer systems were designed
- John von Neumann at Princeton Institute for Advanced Studies first implemented this idea in 1952 on a computer called IAS
  - von Neumann architecture is the name often cited

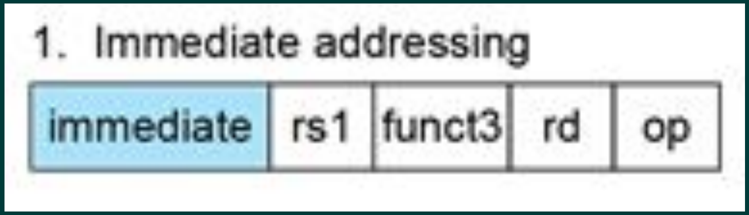
The Big Picture (continued)

- The concept allows programs and the data they need to be stored in memory at the same time
  - A program is executed simply by telling the system where in memory to start reading binary machine code vs binary data
- This simplifies the memory hardware, processor, and software
  - Memory technology can be the same for both programs and data
  - Lowers cost / improves performance

RISC-V Addressing Mode Summary (1)

- Addressing mode: one of several addressing regimes delimited by their varied use of operands and/or addresses
- Addressing modes define how the operands needed by instructions are determined

1. Immediate addressing: an operand is a constant in the instruction itself

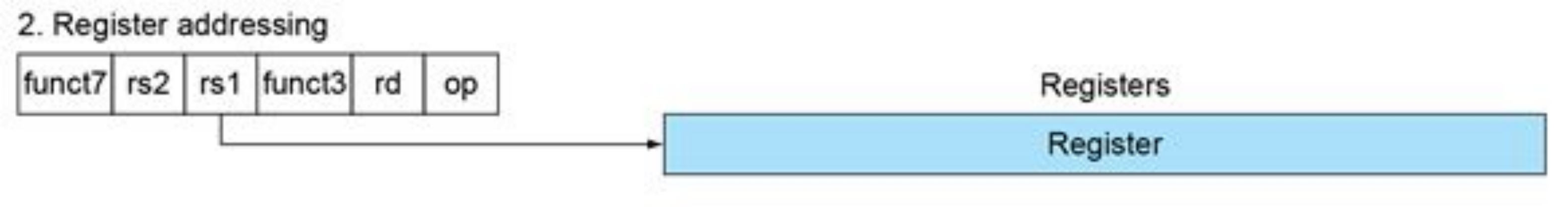


1. Immediate addressing

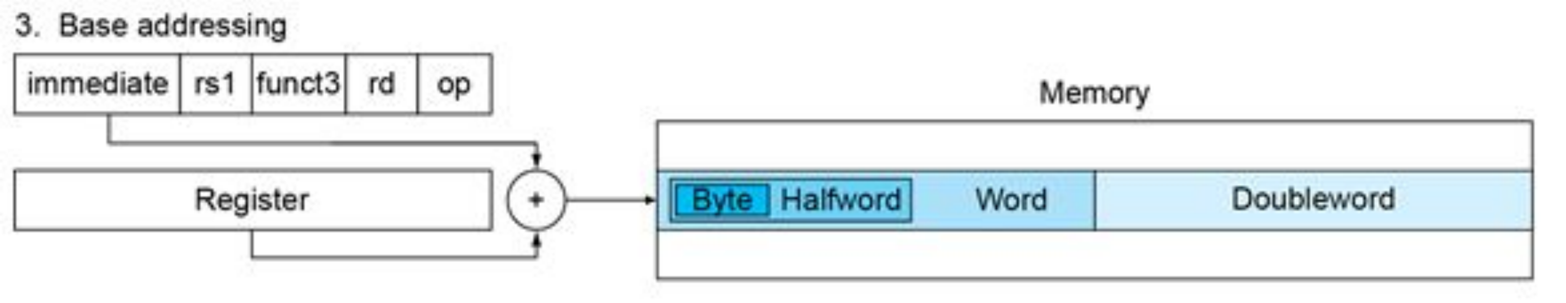
immediate	rs1	funct3	rd	op

RISC-V Addressing Mode Summary (2)

2. Register addressing: the operands are registers

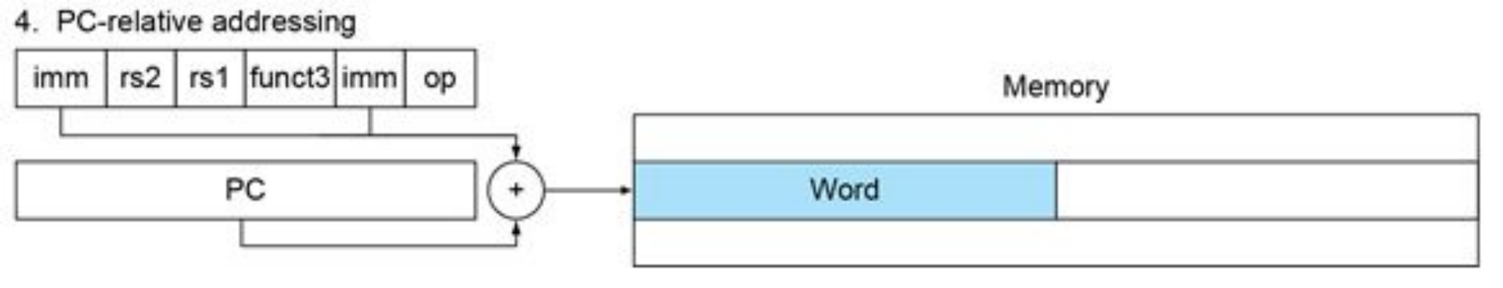


3. Base addressing: the operand is at the memory location whose address is the sum of a register and a constant in the instruction



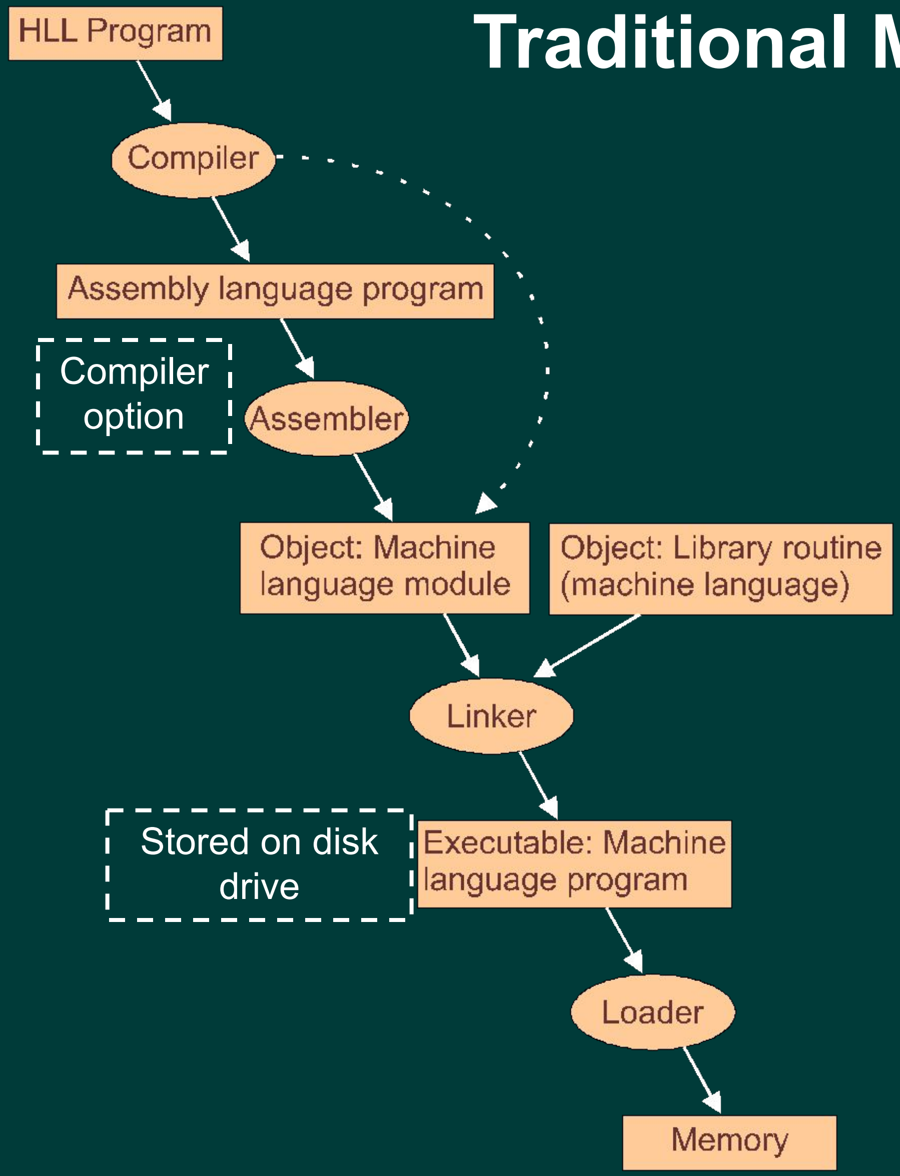
RISC-V Addressing Mode Summary (3)

4. PC-relative addressing: where the branch address is the sum of the program counter and a constant in the instruction



Steps Leading to Program Execution

Traditional Model


HLL Program
Compiler
Assembly language program
Compiler option
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Stored on disk drive
Executable: Machine language program
Loader
Memory

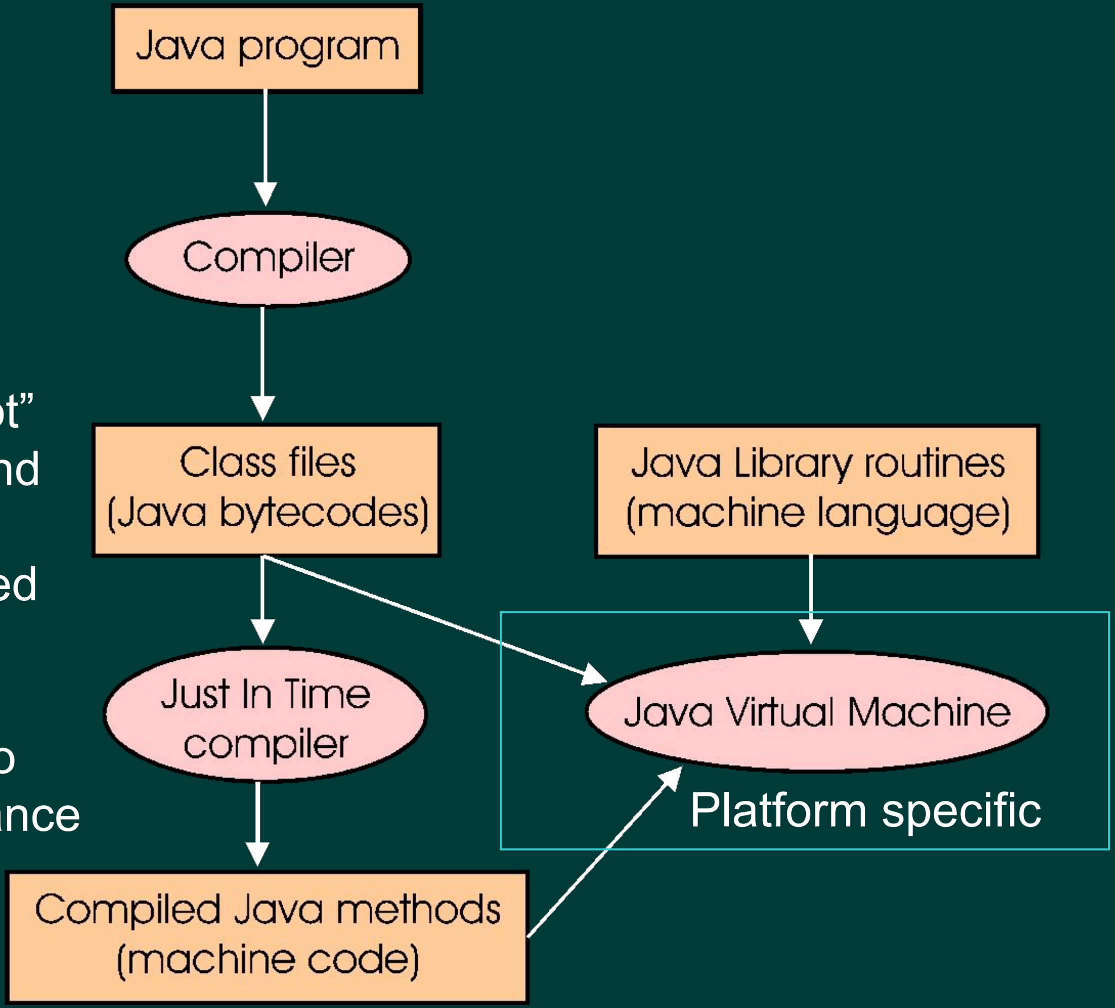
- Emphasis on fast execution time
- Targeted to specific architecture
- Historically large .exe programs due to entire library of routines combined into program (static linking).

DLLs (dynamically linked libraries) are now used in which routines are not loaded until the program executes and calls them.

Interpreted languages are translated and executed one statement at a time.

Java

- Different compilation from the traditional model
- Design goals of portability, not so much performance



JIT compiler, called “Hotspot” profiles running program and compiles “hot” methods to machine code which is saved for the next execution.

JIT compilers were added to help improve slow performance by Java interpreter.

Ch. 2: Concepts / Terminology / Techniques

- High Level Language (HLL)
- Machine language (ML)
- HW/SW Interface
  - Dynamic/static interface
- Compilation
- Instruction
- Instruction set
- Microarchitecture
- Assembly language
- Assembler directives
- Assembler labels
- Memory views
- Data types & sizes
- Registers
- Operands
- Types of operations
  - Arithmetic/logic, memory access, control, system calls
- Design principles (3 of them)
- Assembly language comments
- Memory addressing
- Base address
- Offsets
- Byte-addressable
- Word-addressable
- Alignment restriction
- Byte ordering
- Constants
- Immediate instructions
- Arithmetic vs. logic operations
- Conditional vs. unconditional branching
- Loops
- Instruction format
- Stored program concept
- Subroutines
- Stacks
- Addressing modes