Programming Microprocessor – Instruction Set Architecture II

CS2053 Computer Architecture

Computer Science & Engineering

University of Moratuwa

Sulochana Sooriyaarachchi

Chathuranga Hettiarachchi

Slides adopted from Dr.Dilum Bandara

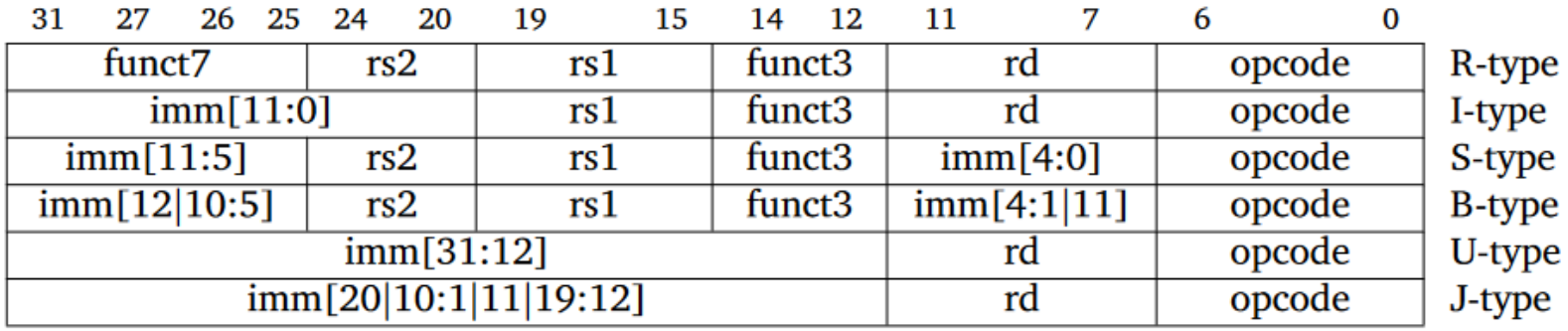
Encoding Instructions

- Various instruction types
- Limited word size for registers, addresses, and instructions
  - Consider 32bit words in RV32I
  - All the instructions are 32bits
  - Example : If we need to load an immediate value to 32-bit register, how to fit all opcode and operands within 32-bit instruction?
    - Work with small numbers
    - Make compromises

Types of RISC-V Instructions

- Register Type
  - Source Registers
  - Destination Register
- Immediate Type
  - Registers
  - Immediate Value
  - Load type
    - Base memory Address
    - Destination Register
    - Offset from base memory address
- Store Type
  - Memory address
  - Value to be stored
- Branching Type
  - Two Registers for comparison
  - Offset from current PC to branching destination
- Upper-Immediate Type
  - Destination register
  - Upper Immediate Value
- Jump Type
  - Register to backup the current PC+4
  - Offset to jump destination

Instruction Formats(Types)



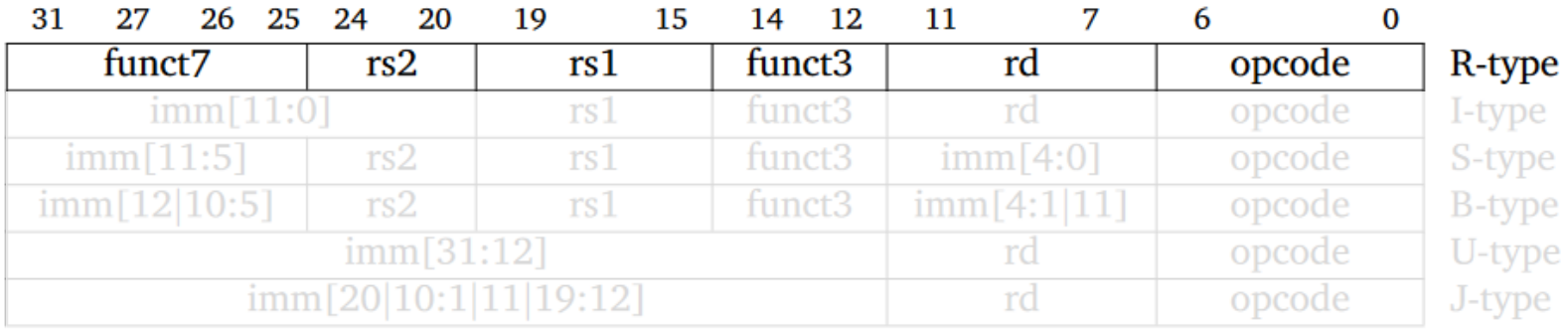
31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

- Fields
  - Opcode : 7bits
  - funct 3 : 3 bit function
  - funct 7 : 7 bit function
  - rs1, rs2 : two source registers (5 bits each)
  - rd : destination register (5 bits)

Instruction Formats (6 Types)

- 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: ***jump*** instructions: `jal`

Instruction Format-Register Type

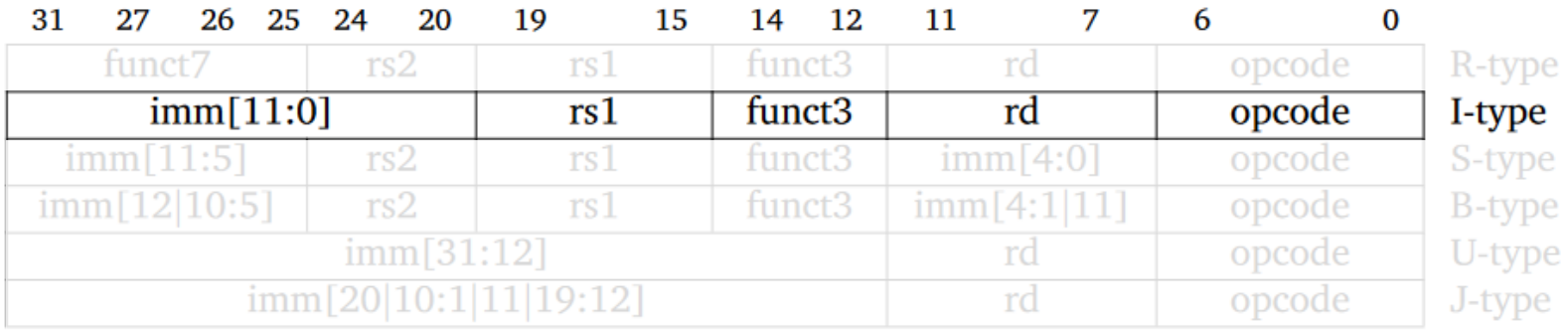


RV32I Base Integer Instructions (Register Type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

For the complete standard specifications:
https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

Instruction Format-Immediate Type



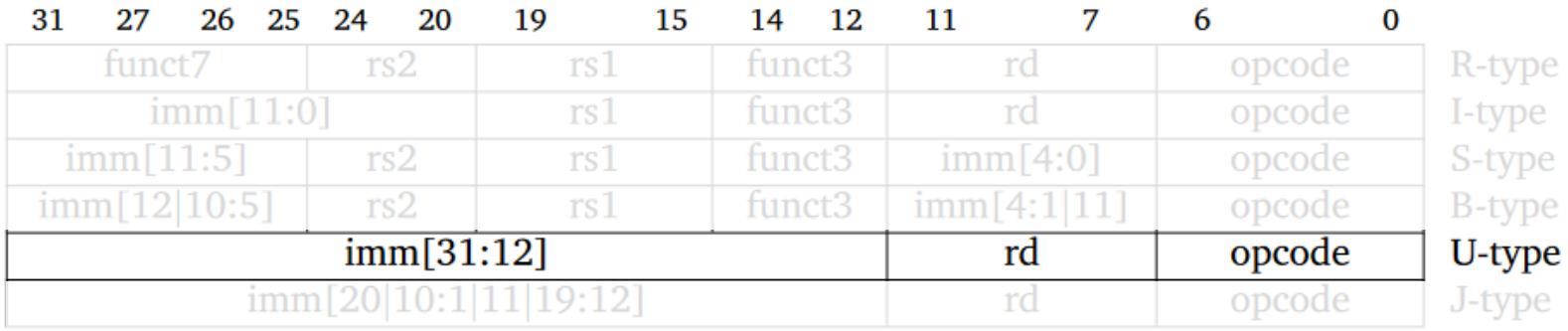
RV32I Base Integer Instructions (Immediate Type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends

Discuss later today

Upper Immediate Instructions

Instruction Format-Upper Immediate



lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	

Load Upper Immediate (lui)

lui t1,0x70070

Fill up the upper 20 bits of destination register with immediate value

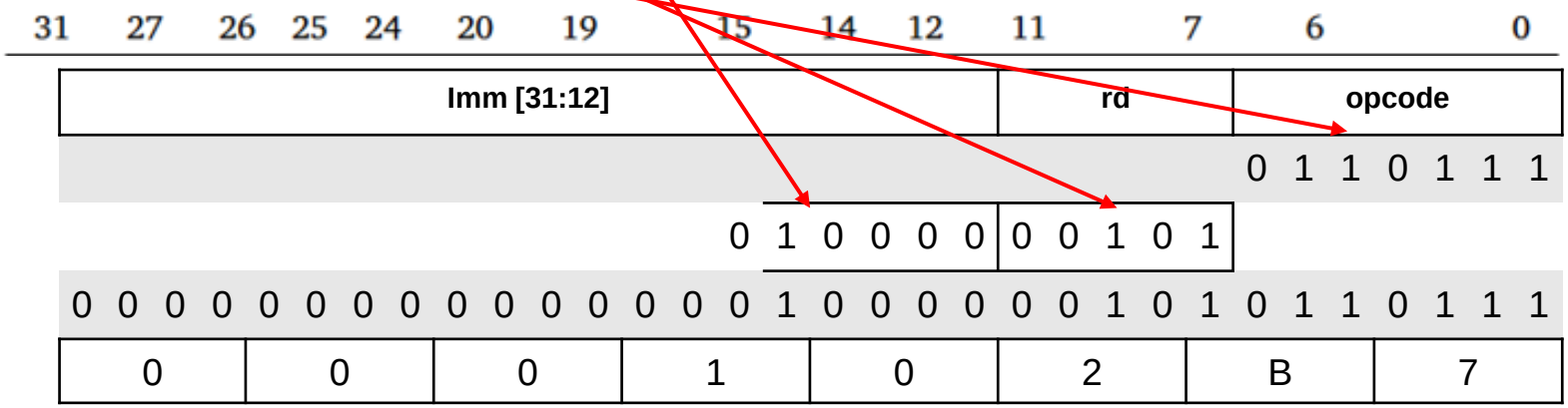
Add Upper Immediate value and Program Counter (auipc)

auipc a0,0x2

Fill the upper 20 bits of destination register with immediate value

U Type Instruction Example

RISCV Instruction: `lui x5,0x10`



Machine Instruction: **0x000102B7**

Result ?? Rd = imm << 12

Register x5 (t0) = 0x0010 <<12
= 0x00010000

We can work with 20bit immediate values!

RISC-V U-type instructions usage

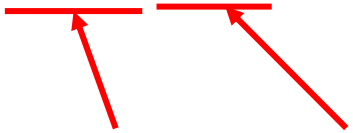
Some instructions are difficult to get done with a single instruction.

Example:

Load immediate **value 0x700707FF** (or any 32 bits long number) to x6 register

Since we must encode the instruction opcode in 32 bits as well, it is impossible to do in a single instruction.

Split the 32bit value to two parts. [31:12][11:0]



Upper Part (20 bits)

Lower Part (12 bits)

Need two instruction to do this:

lui x6, 0x70070 (U type instruction)

addi x6, 0x7FF

Example 1:Load 0x000707FF to t0

- Split to two numbers 0x00070000 + 0x000007FF
  - 0x0070 to t0 upper bits
    - lui t0,0x70

    0x10 = 0b01110000 << 12

    t0 = 0b 0000 0000 0000 0111 0000 0000 0000 0000

    0x7FF = 0b 0000 0000 0000 0000 0000 0111 1111 1111
    - addi t0,t0, 0x7FF

      = 0b 0000 0000 0000 0111 0000 0111 1111 1111

      = 0x000707FF

    So , we loaded 0x000707FF to t0 32bit register

Example 2:Load 0x0000FFFF to t0

- 0x7FF is the max value for addi instruction
- Split to two numbers 0x00010000 + (-1)
  - 0x0010 to t0 upper bits
    - lui t0,0x10

    0x10 = 0b00010000 << 12

    t0 = 0b 0000 0000 0000 0001 0000 0000 0000 0000

    -1 = 0b 1111 1111 1111 1111 1111 1111 1111 1111
    - addi t0,t0, -1

      = 0b 0000 0000 0000 0000 1111 1111 1111 1111

      = 0x0000FFFF

    So , we loaded 0x0000FFFF to t0 32bit register

    How about loading 0x0000F800 ???

Meh..

- Why so complicated? Difficult to write programs
- Who designed these??
- Can’t we make it easier???

- Early days, the instruction sets used to be relatively easy to understand and write.
  - Microchip PIC instruction set.
  - Z80 instructions.

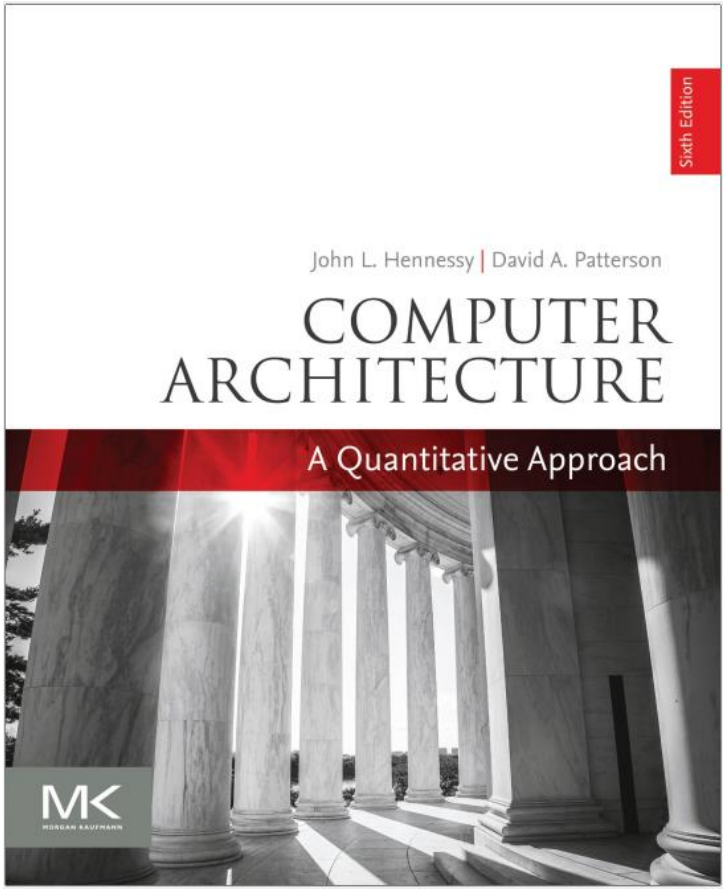
- We need to understand real world demands

How to identify a good ISA?

- Meet functional requirements

- Are the real-world computations cheaper, faster and energy efficient?
- Have less “code density”, so the memory for storing and transferring programs will be less
- The base instruction set is selected based on the real-world performance measures.
  - Run a collection of real-world programs with and without a specific instruction
    - “Benchmark” Example: SPEC benchmark

- Good time to start reading the book.
  - Chapter 1



Assembly Programmers View

- Writing code ?
  - Usually, we write codes in C or Python etc.
  - Compilers and Assemblers can take care of converting high-level codes to machine codes.
- Let us define some ‘high-level assembly’ instructions as well to make it easier to write assembly codes

Assembly Programmers View

- Example:
  - Loading 32bit values to registers using base instructions is complicated.
  - Let our assembler handle the complexity
  - We shall write intuitive instructions

Pseudo Instructions

```
li t1, 0x7FFFFFFF
```

- Load immediate value to t1 register (assembler will convert this to following two “base instructions”)

```
80000337   lui  t1,0x80000
fff30313   addi t1,t1,-1 # 7fffffff
```

- List of pseudo instructions can be found in riscv-card.

Some RISC V Pseudo Instructions

`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if $=$ zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero

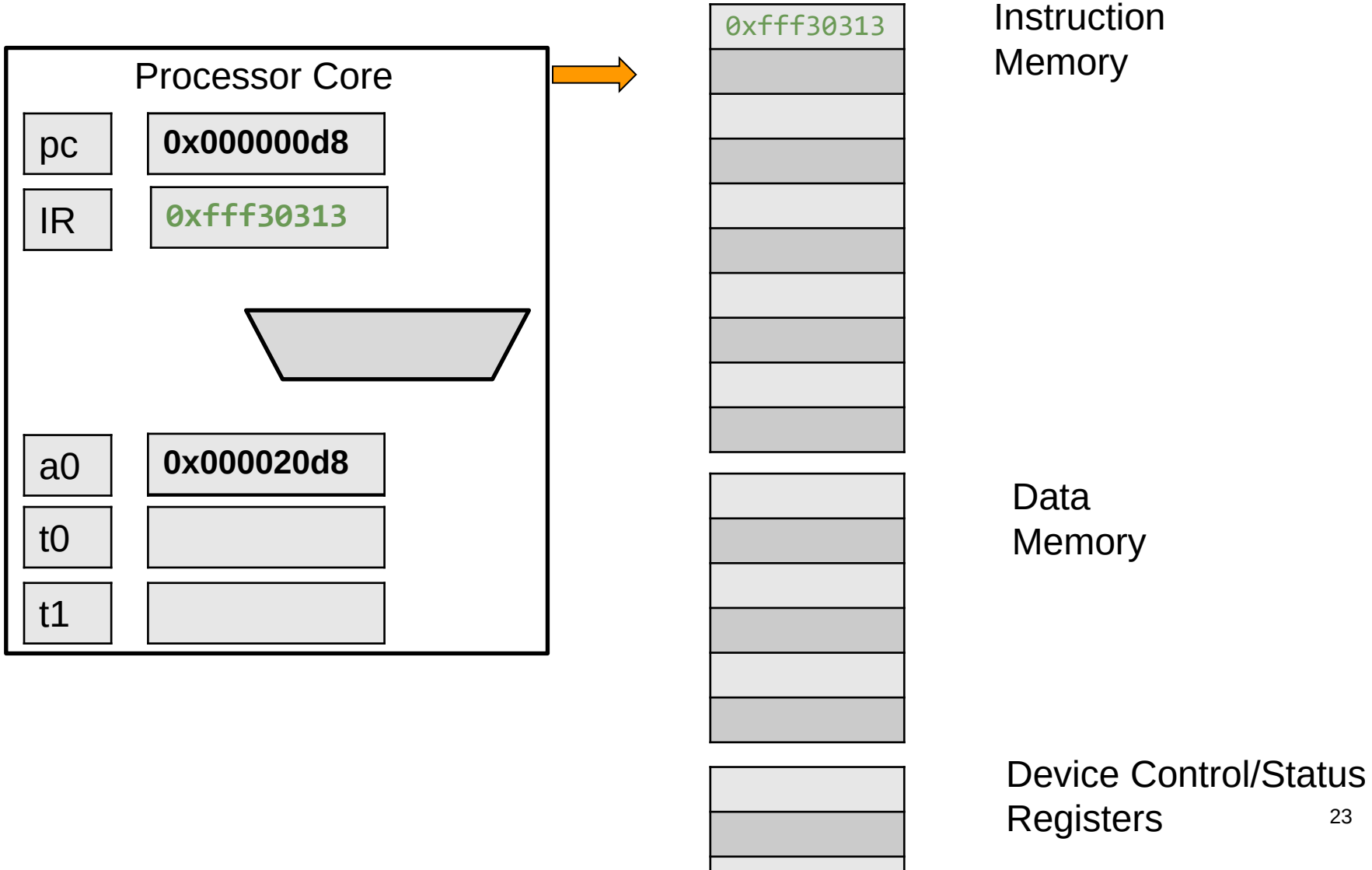
Some more are available…

Memory Related Instructions

Registers vs Memory (Norms)

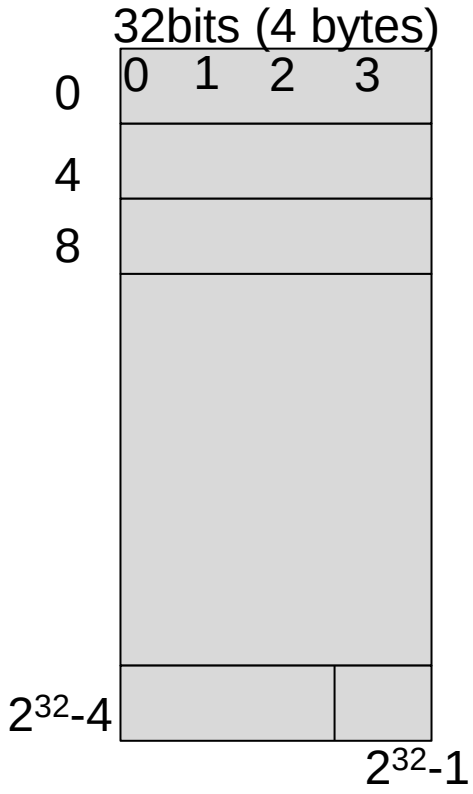
- General Purpose Registers
  - Named (norm)
  - Limited
  - Located very close to the ALU
  - Expensive
  - Fast
  - Locations are not related
- Memory
  - Addressed
  - Larger/Extended
  - Away from ALU
  - Cheaper compared to registers
  - Slow compared to registers
  - Memory “map”: relative locations (Viewed as an array or a contiguous space)

Registers vs Memory



Instructions for memory access

- Load values from memory to registers
- Store values to memory from registers



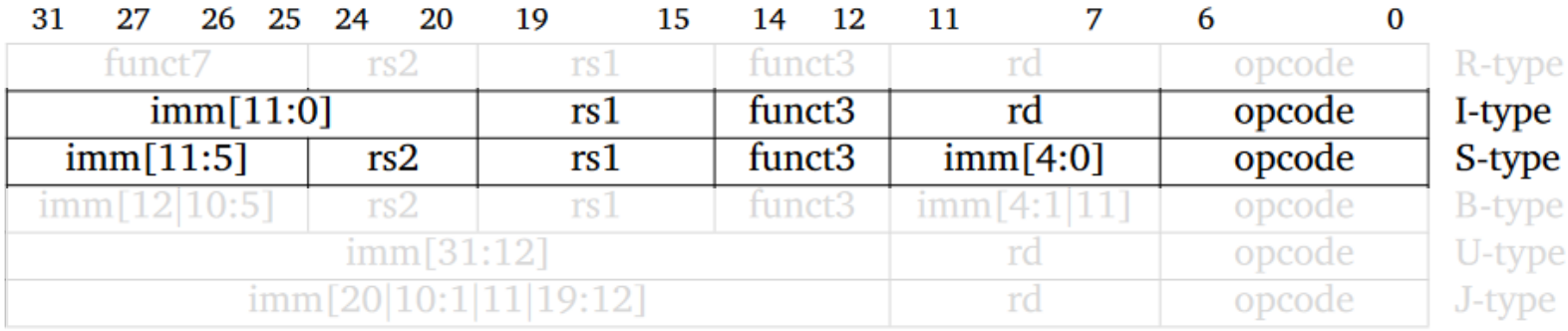
- Address? (Byte addresses)
  - How many addresses?
    - RV32I => Address bus is 32bits

      Maximum Memory Space: 2^{32}

      4294967296

      2^32 /1024/1024/1024 = 4GB
    - RV64I => Address bus is 64bits
- Values?
  - 32 bits

Load and Store Instructions



Load: copy a value from memory to register *rd* : Immediate type
Source is a memory address

Store: copy the value in register *rs2* to memory : Store type

Need the proper 32bit address

RV32I Base Integer Instructions (Immediate type and Store type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	

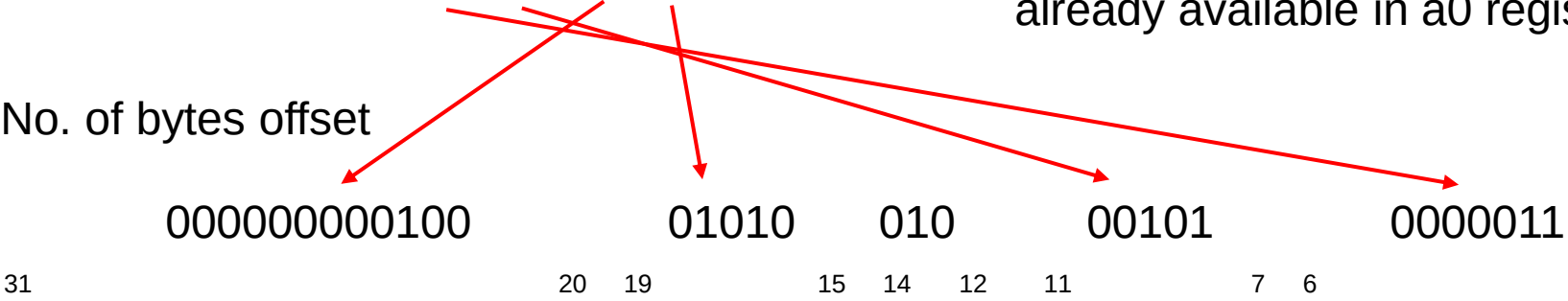
Load Word Instruction (I type)

```
lw t0, 4(a0)
```

RISCV Instruction: `lw x5, 4(x10)`

Assume we have a ‘base address’ already available in a0 register

No. of bytes offset



Imm[11:0]	rs1	funct3	rd	opcode
Byte offset (immediate)	Base address register	funct3 lw	Destination register	I type load
		0 1 0		0 0 0 0 0 1 1
0 0 1 0 0	0 1 0 1 0		0 0 1 0 1	
0 0 0 0 0 0 0 0 0 1 0 0	0 1 0 1 0	0 1 0	0 0 1 0 1	0 0 0 0 0 1 1
0	0 4	5	2	2	8	3

Machine Instruction: **0x00452283**

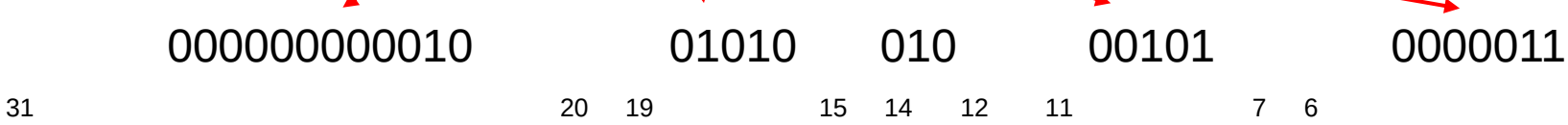
Load Half Instruction (I type)

```
lh t0, 2(a0)
```

RISCV Instruction: `lh x5, 2(x10)`

Assume we have a ‘base address’ already available in a0 register
Only half of the word will be loaded

No. of bytes offset

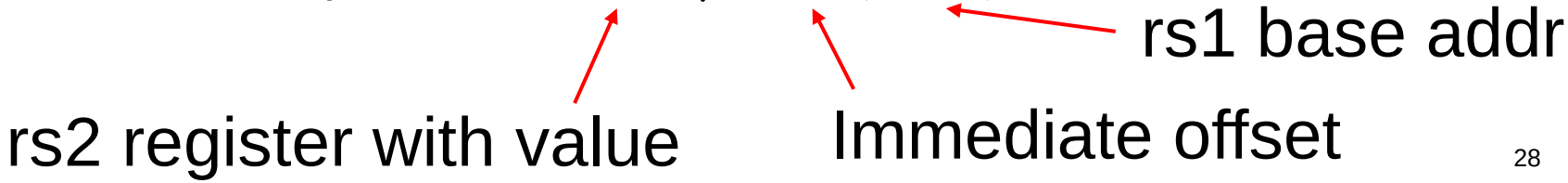


Imm[11:0]	rs1	funct3	rd	opcode
Byte offset (immediate)	Base address register	funct3 lh	Destination register	I type load
		0 0 1		0 0 0 0 0 1 1
0 0 0 1 0	0 1 0 1 0		0 0 1 0 1	
0 0 0 0 0 0 0 0 0 0 1 0	0 1 0 1	0 0 0 1	0 0 1 0 1	0 0 0 0 0 1 1
0	0 2	5	1	2	8	3

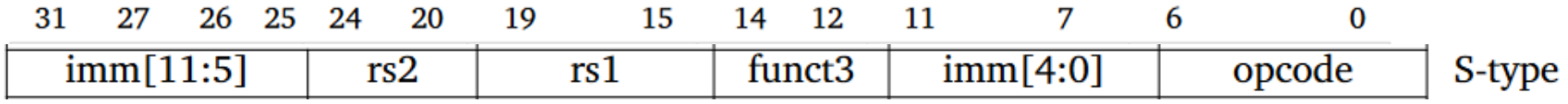
Machine Instruction: **0x00251283**

Store Instructions (S type)

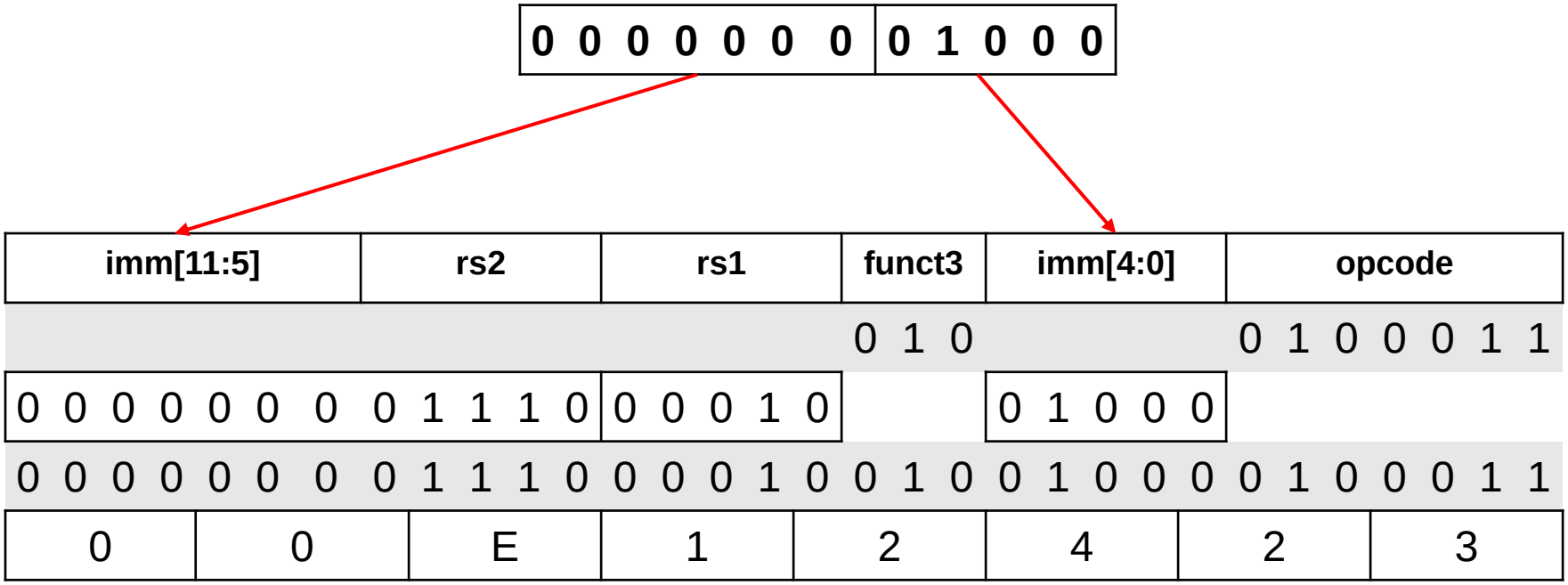
- To store a value in memory, you need three things
  - Memory location to save to
    - rs1: base memory address
    - Immediate memory location offset
  - Value to save
    - rs2: contains data to be stored
  - Choice: Move rs2 to different place or *split immediate offset*
    - Example `sw x14, 8(x2)`



Store Type Example 1



RISCV Instruction: `sw x14, 8(x2)`



Machine Instruction: **0x00E12423**

Store Instructions (S type)

```
sw t0, 4(a0)
```

RISCV Instruction: `sw x5, 4(x10)`

Assume we have a ‘base address’ available in a0 register

No use unless offset is large

00101 01010 010 00100 0100011

31 20 19 15 14 12 11 7 6 0

Imm[11:5]	rs2	rs1	funct3	Imm[4:0]	opcode
Immediate offset [4:0]	Source register	Base address register	funct3 sw	Immediate offset [4:0]	S type store
			0 1 0		0 1 0 0 0 1 1
	0 0 1 0 1	0 1 0 1 0		0 0 1 0 0	
0 0 0 0 0 0 0	0 0 1 0 1	0 1 0 1 0	0 1 0	0 0 1 0 0	0 1 0 0 0 1 1
0	0	5	5	2	2	2	3

Machine Instruction: **0x00552223**

Programmers Perspective

- We don’t know memory addresses.
- Let us write memory position independent code
  - Ask the assembler to declare a data section along with the instruction sequence
  - `auipc instruction allows to address relative to program counter.`

```
.data
    A: .word 0x1F2F3F4F
.text
.globl main

main:
    la a0, A
    # Pseudo instruction la (How it works? will discuss in next slide)
    # Load the address of symbol to the register

    li t1, 0x1E2E3E4E

    lw t0, 0(a0)
    # Get the value in a0 address, byte offset 0 and load it to t0 register

    sw t1, 0(a0)
    # Store value in t1 to A, at byte offset of 0

    ret
.end
```

auipc instruction and la pseudo instruction

lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	

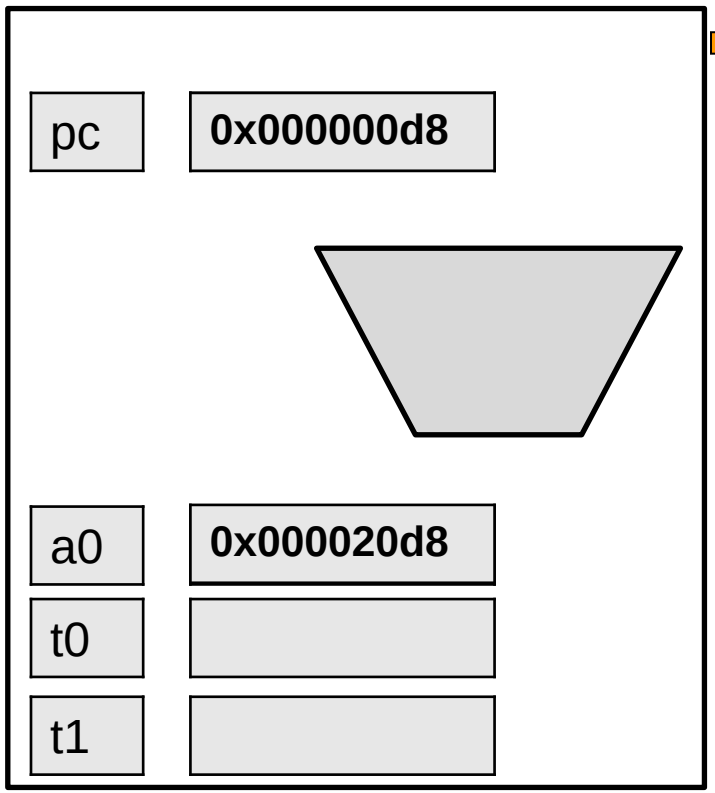
Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address

```
.data
    A: .word 0x1F2F3F4F
.text

.globl main
main:
    la a0, A
    la a0, A
    la a0, A
    ret
.end
```

```
000000d8 <main>:
  d8:  00002517    auipc   a0,0x2
  dc:  0b050513    addi    a0,a0,176 # 2188 <A>
  e0:  00002517    auipc   a0,0x2
  e4:  0a850513    addi    a0,a0,168 # 2188 <A>
  e8:  00002517    auipc   a0,0x2
  ec:  0a050513    addi    a0,a0,160 # 2188 <A>
  f0:  00008067    ret
```

auipc instruction `rd = PC + (imm << 12)`



000000d8 <main>:

d8:	00002517	auipc	a0,0x2
dc:	0b050513	addi	a0,a0,176 # 2188 <A>
e0:	00002517	auipc	a0,0x2
e4:	0a850513	addi	a0,a0,168 # 2188 <A>
e8:	00002517	auipc	a0,0x2
ec:	0a050513	addi	a0,a0,160 # 2188 <A>
f0:	00008067	ret	

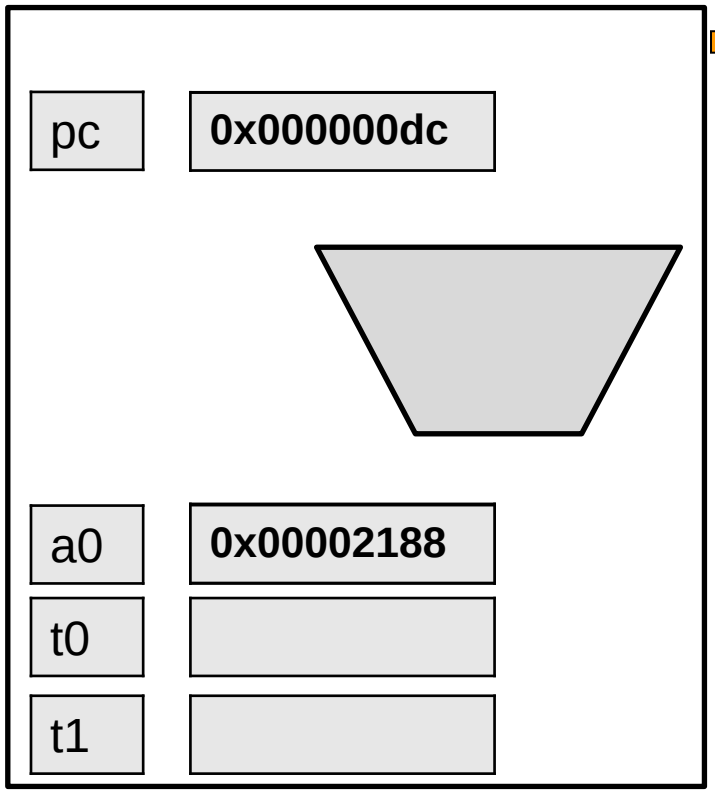
A: 0x00002188:

0x00002000

0x000000d8

0x000020d8

auipc instruction `rd = PC + (imm << 12)`



000000d8 <main>:

d8:	00002517	auipc	a0,0x2
dc:	0b050513	addi	a0,a0,176 # 2188 <A>
e0:	00002517	auipc	a0,0x2
e4:	0a850513	addi	a0,a0,168 # 2188 <A>
e8:	00002517	auipc	a0,0x2
ec:	0a050513	addi	a0,a0,160 # 2188 <A>
f0:	00008067	ret	

A: 0x00002188:

176_2 = 0x000000b0

0x000020dc
+ 0x000000b0
0x00002188

la pseudo instruction

- We wrote the same instruction, but it is converted to different immediate values by assembler

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address

```
.data
    A: .word 0x1F2F3F4F
.text

.globl main
main:
    la a0, A
    la a0, A
    la a0, A
    ret
.end
```

```
000000d8 <main>:
  d8:  00002517     auipc   a0,0x2
  dc:  0b050513     addi    a0,a0,176 # 2188 <A>
  e0:  00002517     auipc   a0,0x2
  e4:  0a850513     addi    a0,a0,168 # 2188 <A>
  e8:  00002517     auipc   a0,0x2
  ec:  0a050513     addi    a0,a0,160 # 2188 <A>
  f0:  00008067     ret
```

Branching Instructions

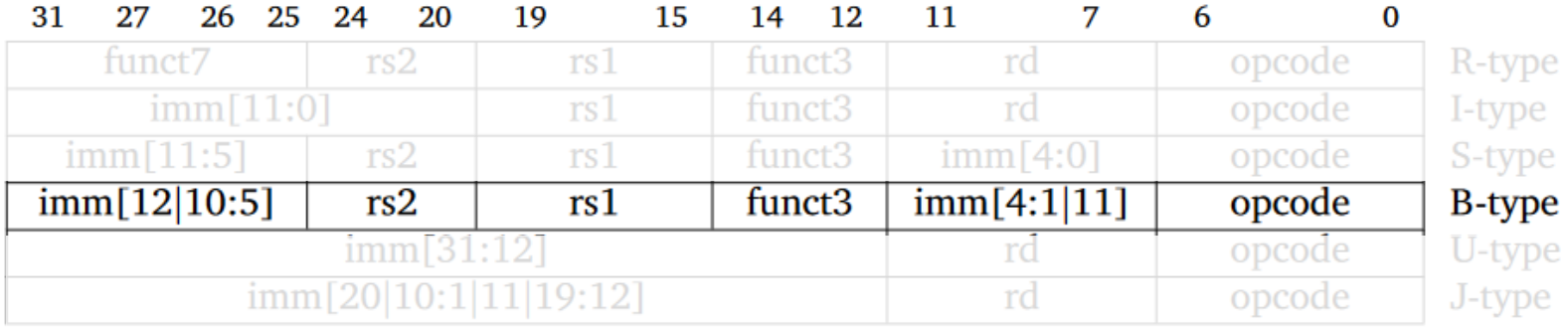
Branching Instructions

RV32I Base Integer Instructions (Branch type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends

- Change the Program Counter

  ⇔ Change the Program Flow



Thank you!