Very Large Scale Integration II - VLSI II

RISC-V

Prof. Dr. Berna Örs Yalçın

Res. Asst. Serdar Duran

ITU VLSI Laboratories

Istanbul Technical University

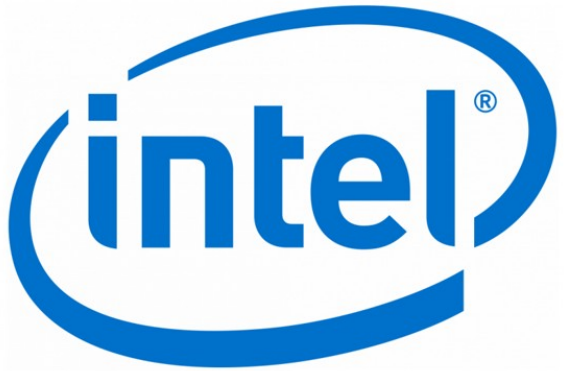
INTRODUCTION

Instruction Set Architecture

- Most personal computers, laptops and game consoles sold are based on the **x86 architecture (Intel, AMD)**.
- Mobile categories such as smartphones or tablets are dominated by **ARM architecture (Qualcomm, Apple)**.
- These architectures are licensed and manufacturers have to make payment for the use of them.

x86 Architecture

- **x86** is a family of ISAs initially developed by Intel based on the Intel 8086 microprocessor.
- Only a few companies hold x86 architectural licenses (Intel and AMD).
- It is primarily "CISC" design with emphasis on backward compatibility.



ARM Architecture

- **ARM** only creates and licenses its designs rather than manufacturing and selling its own physical CPUs..
- Characteristic feature of Arm processors is their low electric power consumption.
- It is primarily a "RISC" design.
- Companies using or manufacturing ARM processors:
  - Analog Devices, Apple, Atmel, Huawei, Mediatek, NXP Semiconductors, Qualcomm, Samsung, STM, Texas Instruments, Xilinx.

RISC-V

- **RISC-V** is an academia leading open-sourced ISA.
- It is usable in both academia and industry without copyright issues.
- Krste Asanović and David Patterson in UCLA are founders and developers of RISC-V.
- It is a "RISC" type design.

RISC-V Features

- Suitable for direct hardware implementation, not just simulation or binary translation.
- Avoids “over-architecting” for a particular microarchitecture style or implementation technology.
- Supports open source operating systems and software toolchains.
- Supports multicore or manycore implementations, including heterogeneous multiprocessors (accelerators, co-processors).

RV32I Base Integer Set

Base Integer Sets

- **RISC-V** has a modular design, consisting of alternative **base integer sets**, with added **optional extensions** (e.g. floating-point).
- Each base integer set is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers.

- There are currently four base integer ISAs: RV32I, RV64I, RV32E, RV128I.
  - **RV32I:** 32-bit integer set, 32-bit wide 32 registers.
  - **RV32E:** 32-bit integer set, 32-bit wide 16 registers, reduced version of RV32I.
  - **RV64I:** 64-bit integer set, 64-bit wide 64 registers, extension of RV32I.
  - **RV128I:** 128-bit integer set, 128-bit wide 128 registers, extension of RV32I and RV64I.

Extensions

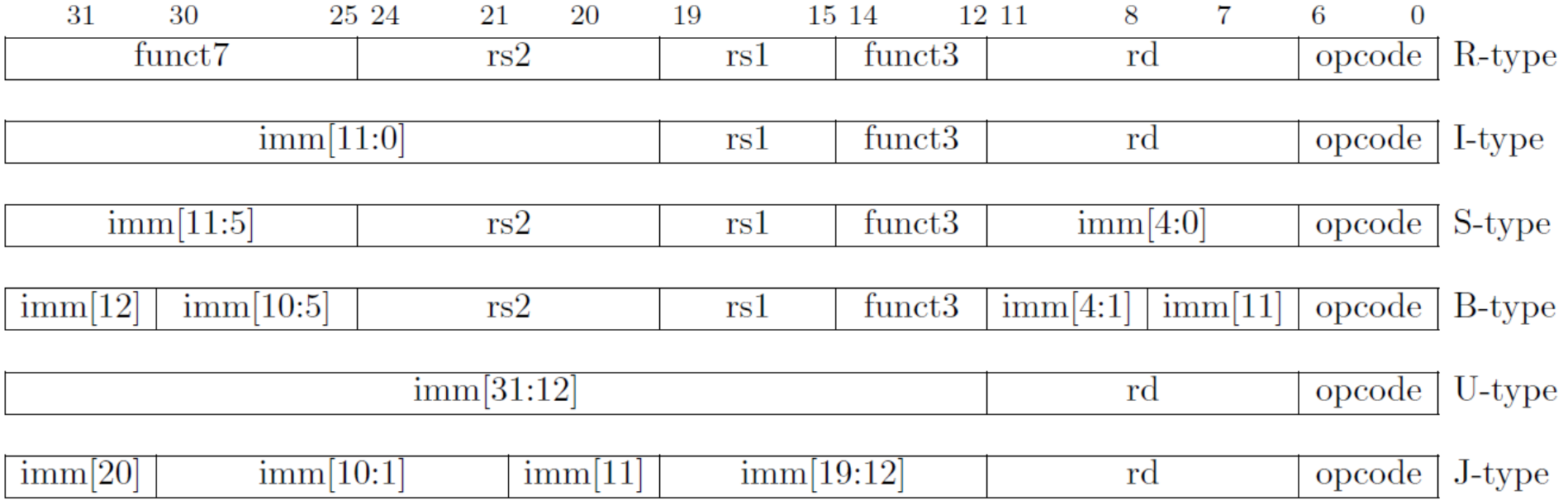
- Each base integer ISA are extended with one or more **optional instruction-set extensions**:
  - The standard integer multiplication and division extension is named “M”,
  - The standard atomic instruction extension, denoted by “A”, for inter-processor synchronization.
  - The standard single-precision floating-point extension, denoted by “F”.
  - The standard double-precision floating-point extension, denoted by “D”.
  - The standard quad-precision floating-point extension, denoted by “Q”.
  - The standard “C” compressed instruction extension provides narrower 16-bit forms of common instructions.

Extensions

- **“M”** contains instructions that multiply or divide values held in two integer registers.
- **“F”** contains single precision floating-point (32 bit) computational instructions.
- **“D”** contains double-precision floating-point (64 bit) computational instructions.
- **“Q”** contains quad-precision floating-point (128 bit) computational instructions.
- **“A”,** contains instructions to support synchronization between multiple RISC-V threads running in the same memory space.
- **“C”**, contains instructions which reduces static and dynamic code size by adding short 16-bit instruction encodings for common operations.

Instruction Formats

- **RV32I** has 6 different instruction formats (R/I/U/S/B/J) fixed **32 bits** in length.



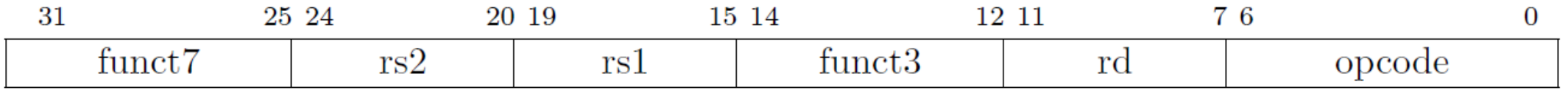
Instruction Formats

Format	Bit																													
	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						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- *opcode* (7 bits): Partially specifies which of the 6 types of *instruction formats*.
- *funct7*, and *funct3* (10 bits): These two fields, further than the *opcode* field, specify the operation to be performed.
- *rs1*, *rs2*, or *rd* (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.

Register-Register Operations

- **ADD** performs the addition of rs1 and rs2.
- **SUB** performs the subtraction of rs2 from rs1. Overflows are ignored.
- **AND**, **OR**, and **XOR** perform bitwise logical operations.



7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

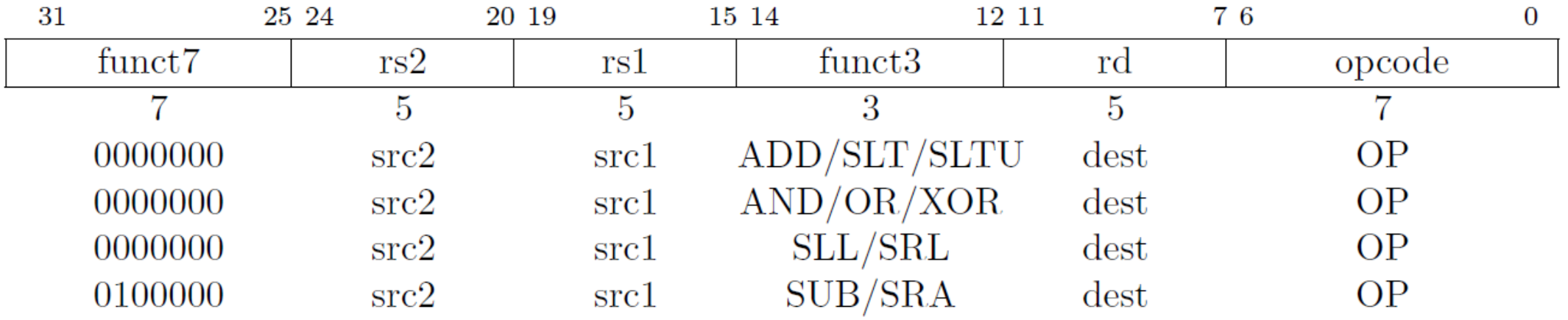
Register-Register Operations

- **SLT** and **SLTU** perform signed and unsigned compares respectively,
  - rs2-rs1 > 0 => rd = 1
  - in SLT when both are treated as signed numbers comparison is made, else 0 is written to rd.
  - in SLTU the values are treated as unsigned numbers.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Register-Register Operations

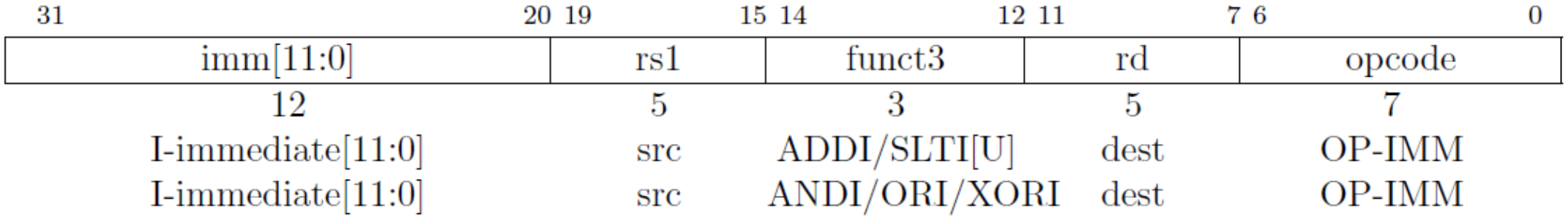
- **SLL**, **SRL** perform logical left and logical right
  - rs1 is the operand to be shifted, lower 5 bits of rs2 is the shift amount (shamt)
- **SRA** arithmetic right shifts
  - the original sign bit is copied into the vacated upper bits.



31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Register-Immediate Operations

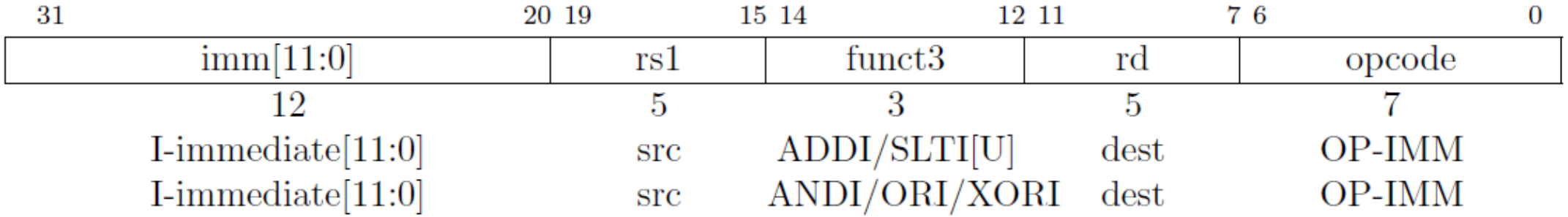
- **ADDI** adds the sign-extended 12-bit immediate to register rs1.
- **SLTI** compares register rs1 with sign extended immediate
  - when both are treated as signed numbers, else 0 is written to rd.



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

Register-Immediate Operations

- **SLTIU** compares the values as unsigned numbers
  - the immediate is first sign-extended to 32 bits then treated as an unsigned number.
- **ANDI**, **ORI**, **XORI** are logical bitwise operations.



31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

Register-Immediate Operations

- **SLLI** is a logical left shift, **SRLI** is a logical right shift.
  - rs1 is the operand to be shifted, lower 5 bits of the I-immediate field is the shift amount (shamt)
- **SRAI** is an arithmetic right shift
  - the original sign bit is copied into the vacated upper bits.

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

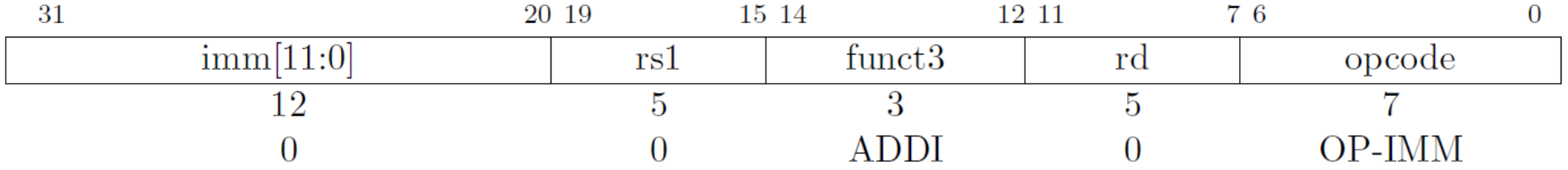
Register-Immediate Operations

- **LUI** (load upper immediate) is used to build 32-bit constants and uses the U-type format.
  - LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros
- **AUIPC** (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format.
  - **AUIPC** adds 20-bit U-immediate, filling in the lowest 12 bits with zeros, to the address in the rd.

31 — 12	11 — 7	6 — 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

NOP Instruction

- **NOP** instruction does not change anything, except for advancing the pc or counters.
  - NOP is encoded as ADDI x0, x0, 0.
- **NOP** can be used for:
  - keep some memory space void
  - create small-time delay



Register File

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

→ Always zero!

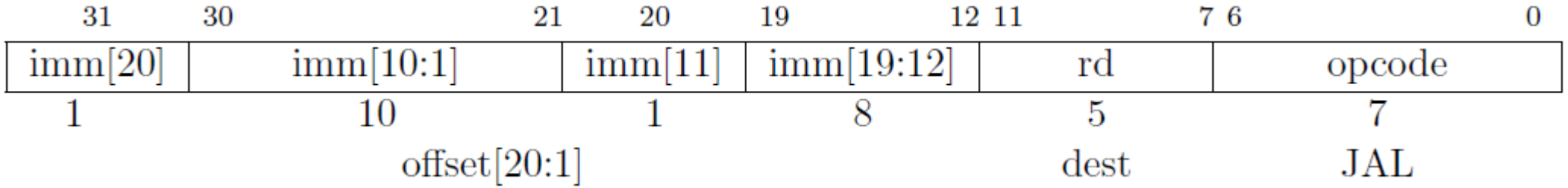
Control Transfer Instructions

- Unconditional Jumps:
  - **JAL**: Direct jump and link
  - **JALR**: Indirect jump and link

- Conditional Branches:
  - **BEQ**: Branch if equal
  - **BNE**: Branch if not equal
  - **BLT/BLTU**: Branch if less than
  - **BGE/BGEU**: Branch if greater than

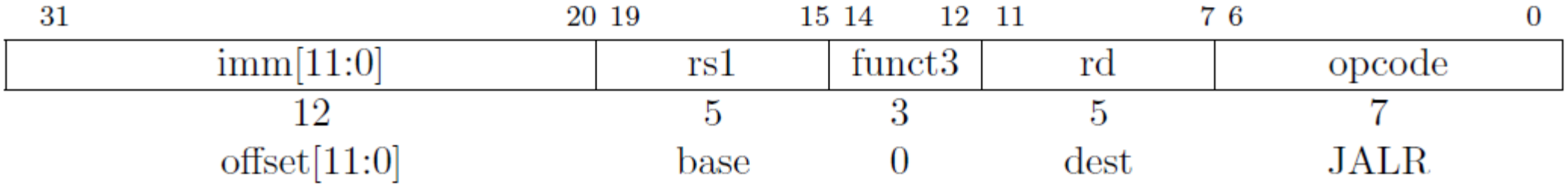
Unconditional Jumps

- **JAL** (jump and link) uses the J-type format,
  - 1 bit offset is added to the 20 bits sign-extended immediate, so jumps multiple of 2 bytes,
  - 2^{20} = ±1 MiB range,
  - stores the next address (pc+4) into register rd.



Unconditional Jumps

- **JALR** (jump and link register) uses the I-type format,
  - the target address is obtained by adding the sign-extended 12-bit I-immediate to the register rs1, then setting the least-significant bit of the result to zero (offset).
  - 2^{12} = ±4 KiB or 2^{32} = ±4 GiB
  - stores the next address (pc+4) into register rd.



Unconditional Jumps

- **JALR** instruction allows jump anywhere in a 32-bit absolute address range.
  - LUI instruction can first load rs1 with the upper 20 bits of a target address, then JALR can add in the 12 lower bits.
- **JALR** instruction does not treat the 12-bit immediate as multiples of 2 bytes like **JAL!**
- JAL and JALR uses PC-relative addressing:
  - Target Address = PC + jump address
  - Jump address in JAL = ± immediate << 1
  - Jump address in JALR = ± immediate ± register

Return/Address Stack

- A JAL instruction should push the return address onto a return-address stack only when rd=x1 or x5 (link registers)

- JALR instructions should push/pop a return-address stack:

rd	*rs1*	*rs1=rd*	RAS action
!*link*	!*link*	-	none
!*link*	*link*	-	pop
link	!*link*	-	push
link	*link*	0	pop, then push
link	*link*	1	push

- **link** is true when the register is either x1 or x5

Conditional Branches

- All branch instructions use the B-type instruction format,
  - 1-bit offset is added to the 12 bits sign-extended immediate, so jumps multiples of 2 bytes,
  - 2^{12} = ±4 KiB
- **BEQ** and **BNE** take the branch if registers rs1 and rs2 are equal or unequal respectively.

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

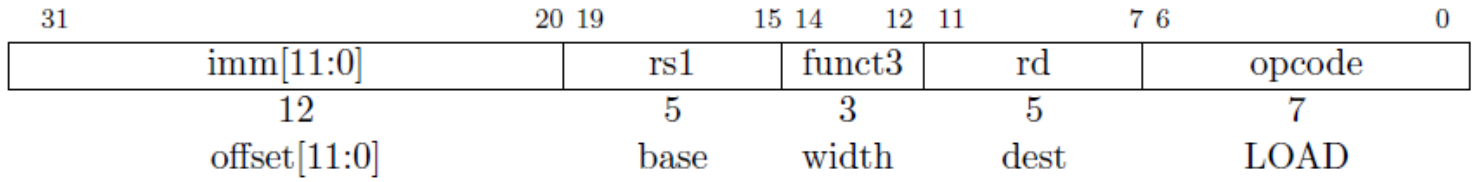
Conditional Branches

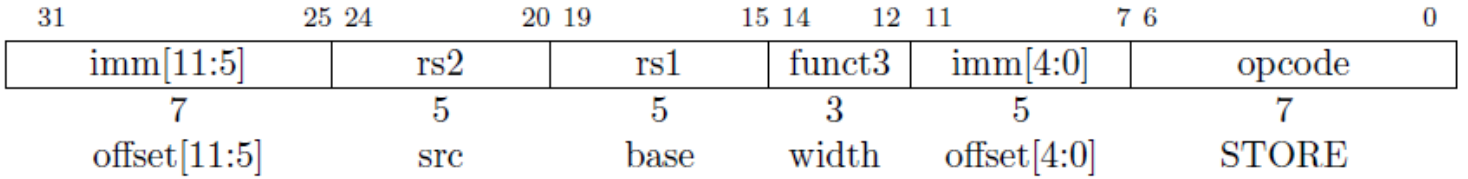
- **BLT** and **BLTU** take the branch if rs1 is less than rs2, using signed and unsigned comparison respectively.
- **BGE** and **BGEU** take the branch if rs1 is greater than or equal to rs2, using signed and unsigned comparison respectively.

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

Load and Store Instructions

- Only **load** and **store** instructions access memory and arithmetic instructions only operate on registers.
- **Loads** are encoded in the I-type format and **stores** are S-type,
  - Memory address is obtained by adding register rs1 to the sign-extended 12-bit offset.
  - **Loads** copy a value from memory to register rd.
  - **Stores** copy the value in register rs2 to memory.





Load and Store Instructions

- **LW** loads a 32-bit value from memory into rd.
- **LH** loads a 16-bit value from memory,
  - sign-extends to 32-bits before storing in rd,
- **LHU** loads a 16-bit value from memory,
  - zero extends to 32-bits before storing in rd,

imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

Load and Store Instructions

- **LB** and **LBU** are defined analogously for 8-bit values.
- **SW**, **SH**, and **SB** instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory.

imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

Pseudo-Instructions

- There are some assembly language instructions that do not have a direct machine language equivalent. (Tables 25.2, 25.3)

`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 ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero

Some Key Points

- **Address space** should be byte-addressable.
- Little-endian configuration should be used for order of bytes.
- Be careful about the exceptions,
  - e.g. in jumps and branching target addresses should be in four-byte boundary,
- Addressing mode is PC-relative for program memory.

References

- The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191213
- https://en.wikipedia.org/wiki/Arm_Ltd
- https://en.wikipedia.org/wiki/X86
- https://en.wikipedia.org/wiki/RISC-V