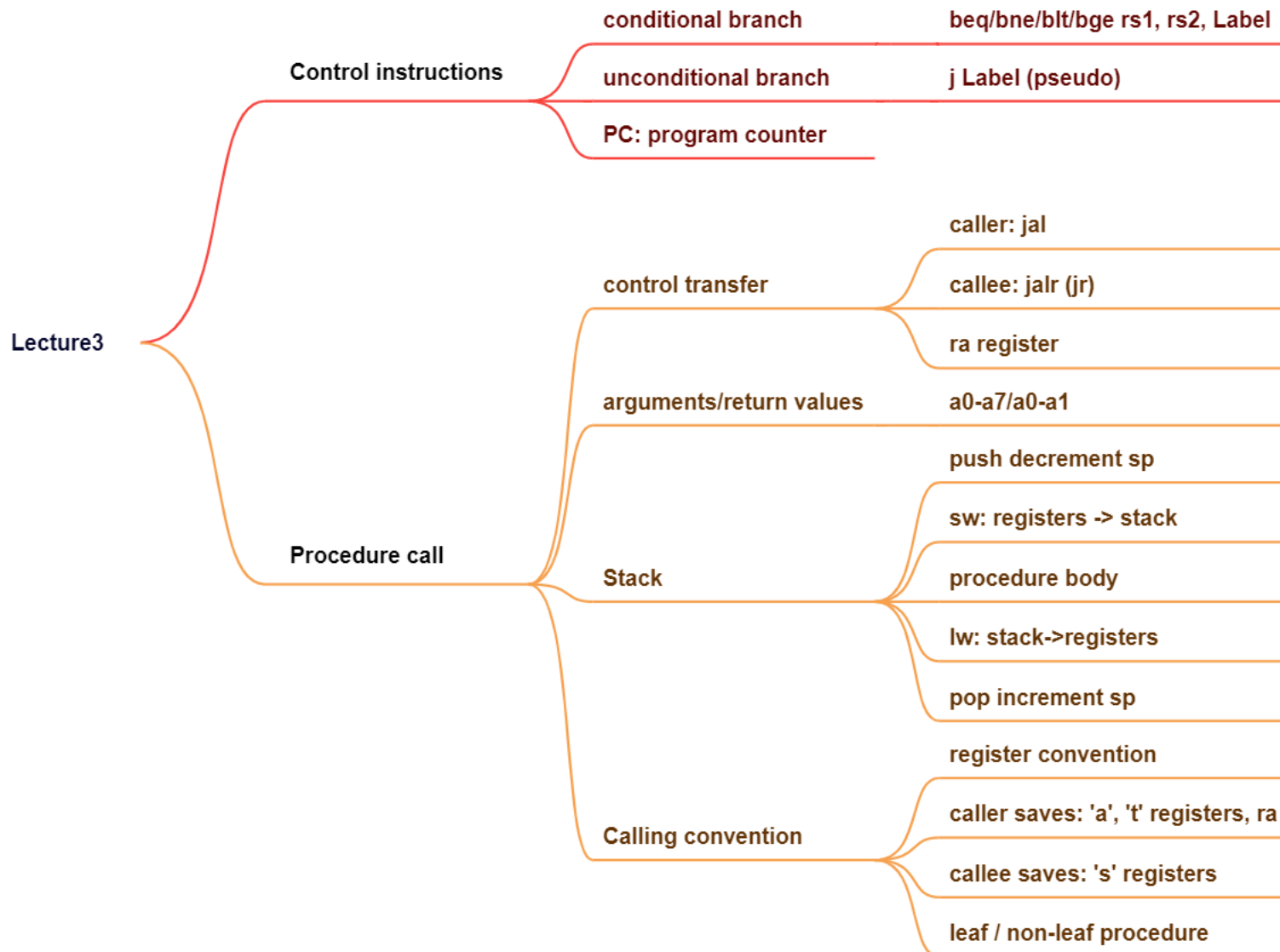


COMPUTER ORGANIZATION

Lecture 4 RISC-V Instruction Format

2024 Spring

Recap



Instructions as Numbers

- Most data we work with is in words (32-bit chunks):
 - Each register holds a word
 - lw and sw both access memory one word at a time
- So how do we represent instructions?
 - Remember: Computer only represents 1s and 0s, so assembly code “`add x10, x11, x0`” is meaningless to hardware
 - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
 - Same 32-bit instruction definitions used for RV32, RV64, RV128

Instructions in Binary

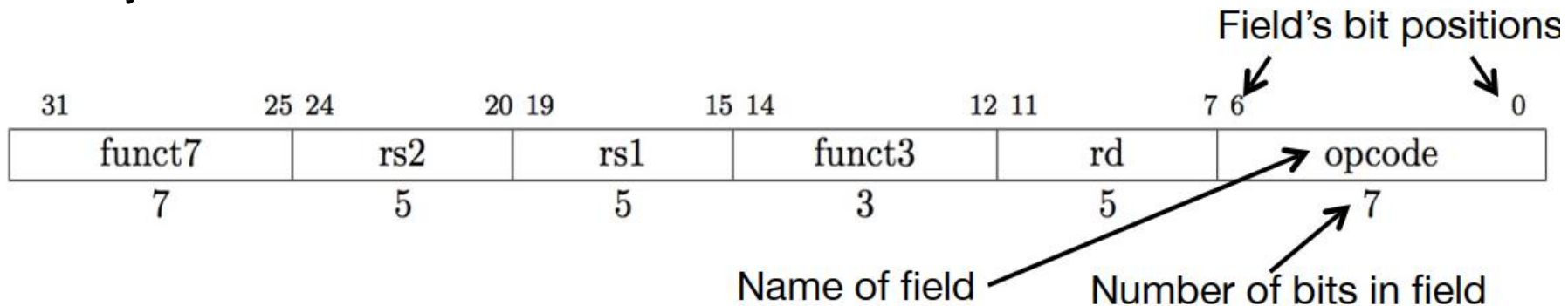
- Divide 32-bit instruction word into “fields”
- Each field tells processor something about instruction
- We could define different set of fields for each instruction, but for hardware simplicity, group possible instructions into six basic types of instruction formats:
 - **R-format** for register-register arithmetic/logical operations
 - **I-format** for register-immediate ALU operations and loads
 - **S-format** for stores
 - **B-format** for branches (SB in textbook)
 - **U-format** for 20-bit upper immediate instructions
 - **J-format** for jumps (UJ in textbook)

RISC-V Instruction Formats

31	30	25	24	21	20	19	15	14	12	11	8	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]		imm[10:5]			rs2			rs1		funct3		imm[4:1]	imm[11]	opcode		B-type	
imm[31:12]										rd			opcode		U-type		
imm[20]		imm[10:1]				imm[11]		imm[19:12]				rd			opcode		J-type

R-Format Instructions

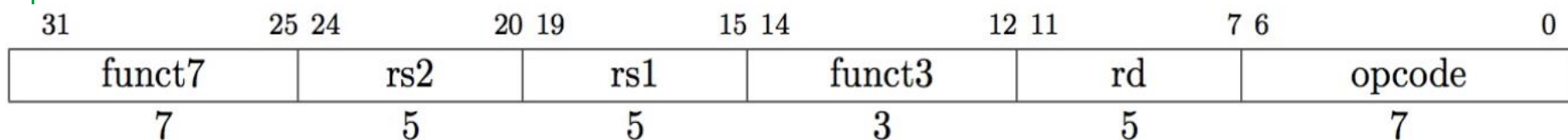
- Layout Annotation



- This example: 32-bit instruction word divided into six fields of differing numbers of bits each field: $7+5+5+3+5+7 = 32$
- In this case:
 - opcode is a 7-bit field that lives in bits 0-6 of the instruction
 - rs2 is a 5-bit field that lives in bits 20-24 of the instruction

R-Format Instructions

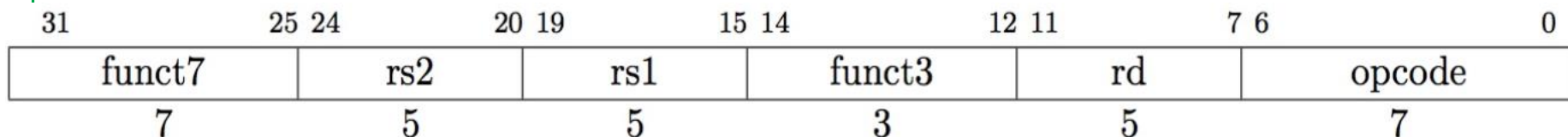
- opcode/funct fields



- opcode: partially specifies which instruction it is
 - Note: This field contains 0110011_{two} for all R-Format register-register arithmetic/logical instructions
- funct7+funct3: combined with opcode, these two fields describe what operation to perform
- Question: Why aren't opcode and funct7 and funct3 a single 17-bit field?
 - We'll answer this later

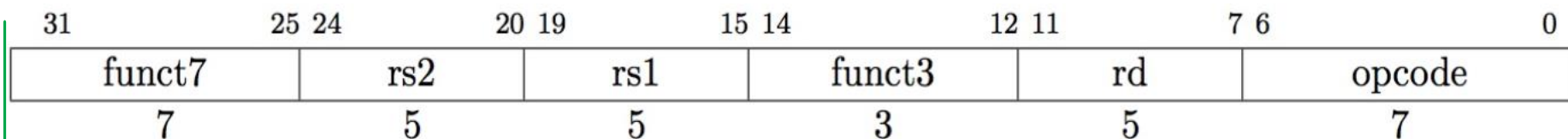
R-Format Instructions

- register specifiers



- Each register field (rs1, rs2, rd) holds a 5-bit unsigned integer [0-31] corresponding to a register number (x0-x31)
 - rs1 (Source Register #1): specifies register containing first operand
 - rs2 : specifies second register operand
 - rd (Destination Register): specifies register which will receive result of computation

R-Format Example



- Example: Convert RISC-V Assembly to Machine Code:

`add x18, x19, x10` (0x00a98933)

0000000	01010	10011	000	10010	0110011
---------	-------	-------	-----	-------	---------

ADD rs2=10 rs1=19 ADD rd=18 Reg-Reg OP

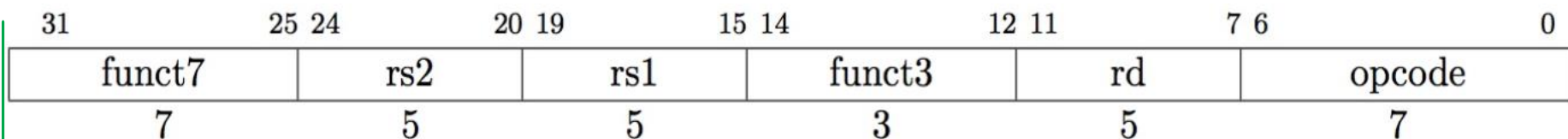
- Example

`sub x10, x11, x12` (0x40C58533)

0100000	01100	01011	000	01010	0110011
---------	-------	-------	-----	-------	---------

SUB rs2=12 rs1=11 SUB rd=10 Reg-Reg OP

R-Format Example



- Example: Convert Machine Code to RISC-V Assembly:

- 0x01B342B3
- 0000 0001 1011 0011 0100 0010 1011 0011
- 0000000_11011_00110_100_00101_0110011

0000000	11011	00110	100	00101	0110011
XOR	rs2=27	rs1=6	XOR	rd=5	Reg-Reg OP

• Answer: xor x5, x6, x27

All RV32 R-format instructions

- This information can be found in RISC-V reference card

funct7			funct3		opcode	
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Encoding in funct7 + funct3 selects particular operation

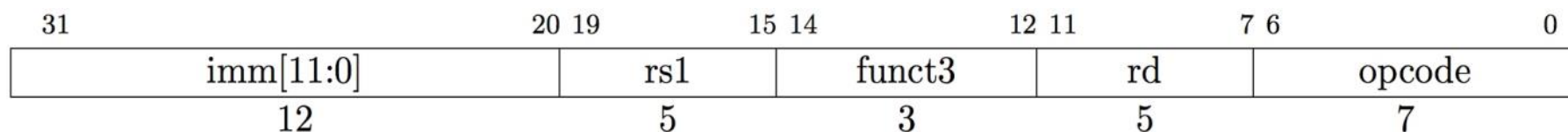
- Exercise: **or x5, x6, x7** in RISC-V machine code?

I-Format Instructions

- What about instructions with immediates?
 - Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise
 - 5-bit field only represents numbers up to the value 31: would like immediates to be much larger
- Define another instruction format that is mostly consistent with R-format
 - Note: if instruction has immediate, then uses at most 2 registers (one source, one destination)

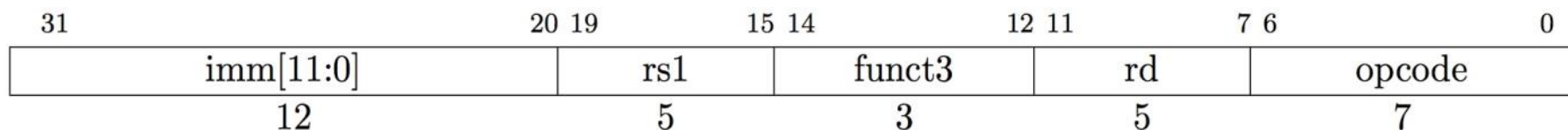
I-Format Instructions

- Layout Annotation



- Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, imm[11:0]
- Remaining field format (rs1, funct3, rd, opcode) same as before
- imm[11:0] can hold values in range $[-2048_{\text{ten}}, +2047_{\text{ten}}]$
- Immediate is always sign-extended to 32-bits before use in an arithmetic/logic operation
- We'll later see how to handle immediates > 12 bits

I-Format Instructions Example



- Example: Convert RISC-V Assembly to Machine Code:

`addi x15,x1,-50`

111111001110	00001	000	01111	0010011
imm=-50	rs1=1	ADD	rd=15	OP-Imm

- Example:

`slli x20, x8, 5`

0000000 00101	01000	001	10100	0010011
imm = 0000000_shmnt(5)	rs1= 8	SLLI	rd=20	OP-Imm

All RV32 I-format Arithmetic/Logical Instructions

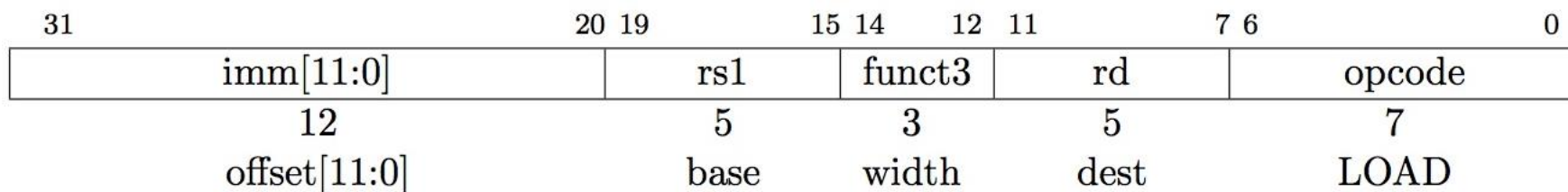
imm		funct3		opcode		
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

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 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

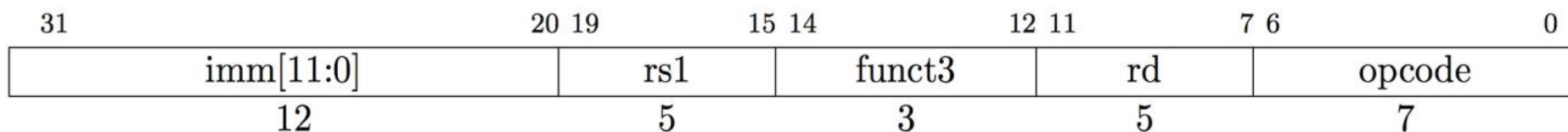
Load Instructions are also I-Type

- `lw rd, immediate(rs1) # rd = Mem[rs1+imm]`



- The 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

I-Format Load Example



- Convert RISC-V Assembly to Machine Code:

`lw x14, 8(x2)` (0x00812703)

000000001000	00010	010	01110	0000011
imm=+8	rs1=2	LW	rd=14	LOAD

- Exercise

- `lb x6, 4(x5)`

- Machine code:

- 000000000100 00101 100 00110 0000011

- 0x0042C303

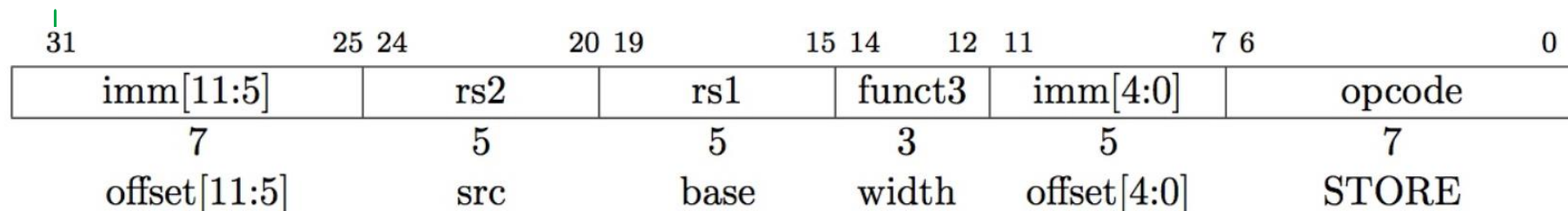
All RV32 Load Instructions

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

↑
funct3 field encodes size and
signedness of load data

- LBU is “load unsigned byte”
- LH is “load halfword”, which loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register
- LHU is “load unsigned halfword”, which zero-extends 16 bits to fill destination 32-bit register
- There is no LWU in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register

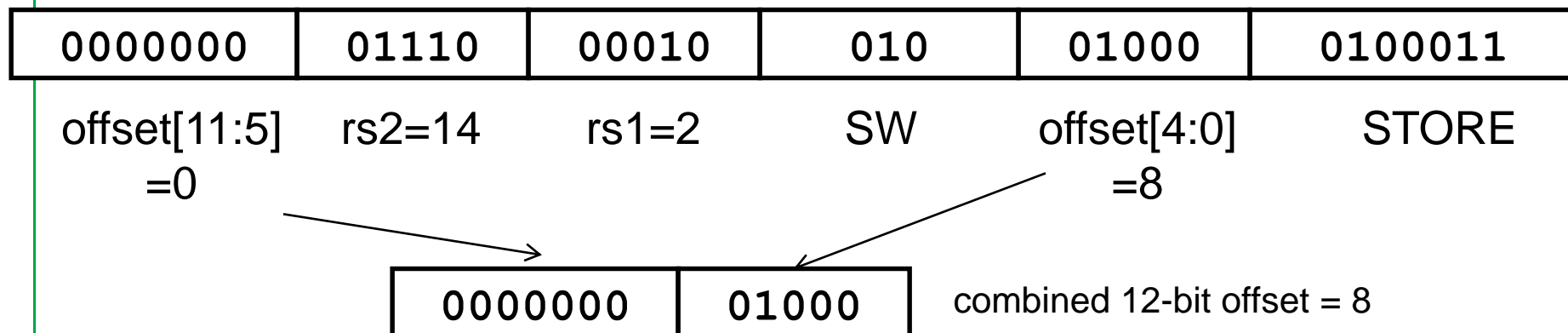
S-Format Used for Stores



- Convert RISC-V Assembly to Machine Code:

`sw x14, 8(x2)`

offset == immediate



All RV32 Store Instructions

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

RISC-V Conditional Branches

- E.g., `BEQ x1, x2, Label`
- Branches typically used for if-else, while, for, etc
 - Generally small (< 50 instructions)
 - Function calls and unconditional jumps handled with jump instructions (J-Format)
- Branches read two registers but don't write any register (similar to stores)
- How to encode the label, i.e., where to branch to?
- We use an immediate to encode **PC relative offset**
 - If we **don't** take the branch:
 - $PC = PC + 4$ (i.e., next instruction)
 - If we **do** take the branch:
 - $PC = PC + \text{immediate}$

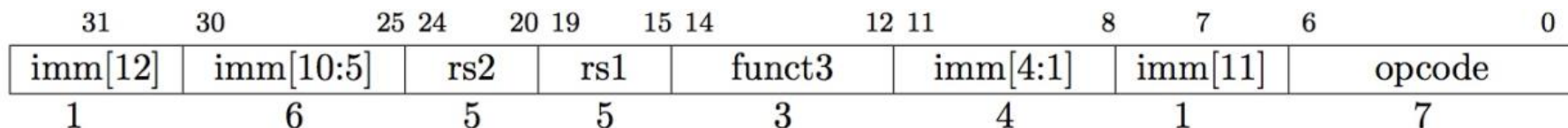
PC-Relative Addressing

- PC-Relative Addressing: Use the immediate field as a two's complement offset relative to PC
 - Branches generally change the PC by a small amount
 - With the 12-bit immediate, could specify $\pm 2^{11}$ **byte address** offset from the PC
- For relative offset, We don't use byte address offset from PC as immediate
 - RISC-V instructions are 32-bit “word-aligned”: Address of instruction is always a multiple of 4 (in bytes), meaning lowest 2 bits are always 0
 - PC ALWAYS points to an instruction
- Should we use word address instead of byte address for PC offset?
 - **Still NO!**
 - Because extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 2-Bytes in length
 - So what's the appropriate solution?

RISC-V Feature, $n \times 16$ -bit instructions

- To enable this, RISC-V always scales the binary instruction's branch immediate by 2 bytes - even when there are no 16-bit instructions
 - thus: Instead of specifying $\pm 2^{11}$ bytes from the PC, we will now specify $\pm 2^{11}$ half words, i.e. $\pm 2^{12}$ bytes from PC
- This means: RISC-V conditional branches can reach $\pm 2^{10} \times 32\text{bit}$ instructions either side of PC
- Thus we have:
- PC-relative addressing
 - Target address = PC + immediate $\times 2$

RISC-V B-Format for Branches



- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents the branch offset in units of half-words. To convert to units of Bytes, left-shift by 1.
- The 12 immediate bits encode actually 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)
 - Thus the imm[12:1] in the total encoding, compared with imm[11:0] in the I-type encodings

Branch Example

- RISC-V Assembly:

```

Loop:  beq x19,x10,End
        add x18,x18,x10
        addi x19,x19,-1
        j  Loop
End:

```

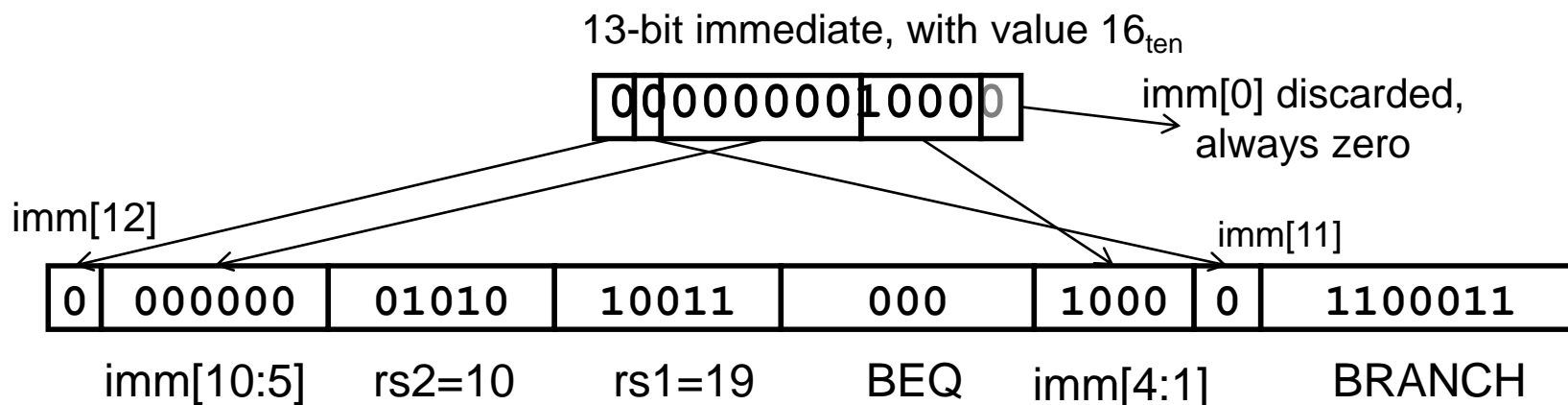
Count instructions from branch

1
2
3
4

- Branch offset = $4 \times 32\text{-bit instructions} = 16 \text{ bytes} = 8 \times 2$


beq x19,x10, 16 # 16 is offset in bytes

- 16_{ten} is 13-bit immediate, high 12 bits are encoded
- beq target address = $\text{PC} + 8 \times 2 = \text{PC} + 16$



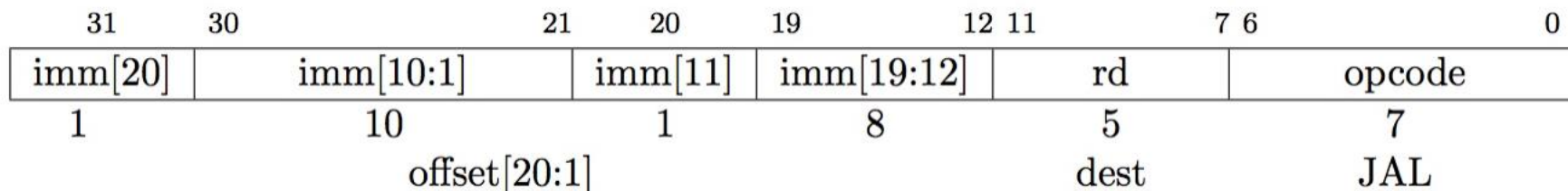
Branching Far Away

- Does the value in branch immediate field change if we move the code?
 - If moving individual lines of code, then yes
 - If moving all of code, then no (because PC-relative offsets)
- What do we do if destination is $> 2^{10}$ instructions away from branch?
 - replace conditional jump to unconditional jump

<code>beq x10,x0, far</code>		<code>bne x10,x0, next</code>
<code># next instr</code>		<code>j far</code>
<code>...</code>		<code>next: # next instr</code>
<code>...</code>		<code>...</code>
<code>far:</code>		<code>...</code>
		<code>far:</code>

- What exactly is “j far” in basic instruction?

J-Format for Jump Instructions



- `jal rd, Label`
 - Example: `jal, x0, Label # j Label(pseudo)`
- JAL saves PC+4 in register rd (the return address)
 - “j Label” is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ half-word offset
 - $\pm 2^{18}$ 32-bit instructions from PC
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Branch & Jump Example

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0			
funct7				rs2				rs1		funct3		rd		opcode		R-type	
imm[11:0]						rs1		funct3		rd		opcode		I-type			
imm[12]		imm[10:5]		rs2				rs1		funct3		imm[4:1]		imm[11]		opcode	B-type
imm[20]		imm[10:1]				imm[11]		imm[19:12]				rd		opcode		J-type	

Loop: slli x6, x9, 2	#I	80000 _{ten}	000000000010_01001_001_00110_0010011
add x6, x6, x11	#R	80004	0000000 01011_00110_000_00110_0110011
lw x5, 0(x6)	#I	80008	0000000000000_00110_010_00101_0000011
bne x5, x10, Exit	#B	80012	0_000000_01010_00101_001_0110_0_1100011
addi x9, x9, 1	#I	80016	0000000000001_01001_000_01001_0010011
j Loop	#J	80020	1_111110110_1_1111111_00000_1101111
Exit: ...		80024	...

- bne x5,x10,12

- immediate = 12 (13 bits decimal but encoded high 12 bits in instruction) →

0	0	0	0	0	0	0	0	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---
- bne target = PC+{inst[31],inst[7],inst[30:25],inst[11:8]}x2=80012 + 6*2 = 80024

- jal x0,-20

- immediate = -20 (21 bits decimal but encoded high 20 bits in instruction)
- j target = PC+{inst[31],inst[19:12],inst[20],inst[30:21]}x2=80024+(-10*2)=80000

32-bit Constants

- Most constants are small, 12-bit immediate is sufficient
- For the occasional 32-bit constant
 - LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
 - Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (lui/addi)

- Example, set 0x87654321

```
lui x10, 0x87654 # x10 = 0x87654000
```

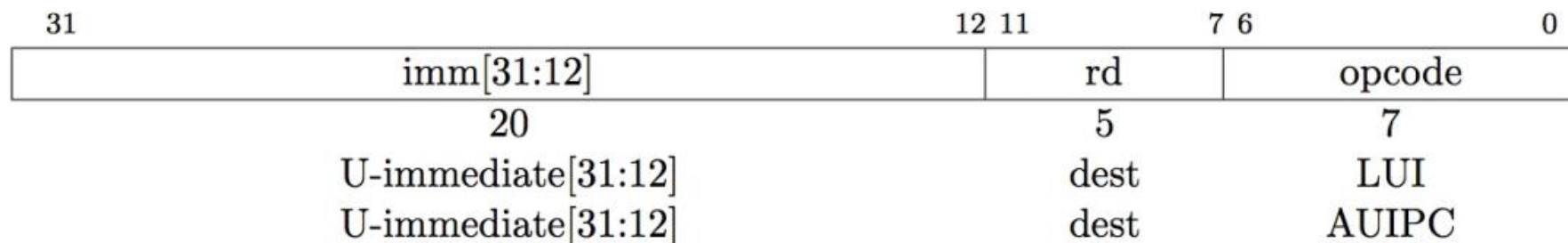
```
addi x10, x10, 0x321 # x10 = 0x87654321
```

- Corner case: set 0xDEADBEEF (addi 12-bit immediate is always sign-extended, if 12-bit immediate's sign bit is 1, will subtract from upper 20 bits)

```
lui x10, 0xDEADC # x10 = 0xDEADC000
```

```
addi x10, x10, 0xEEF # x10 = 0xDEADBEEF
```

U-Format for “Upper Immediate”



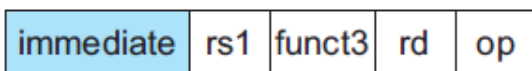
- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
 - LUI – Load Upper Immediate, $rd = imm \ll 12$
 - AUIPC – Add Upper Immediate to PC, $rd = PC + (imm \ll 12)$

Jumps to Absolute

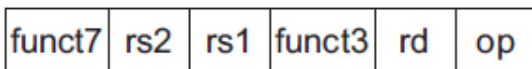
- For even farer jumps, we use `jalr`
- `jalr rd, rs, immediate / jalr rd, immediate(rs)`
 - Writes PC+4 to rd (return address)
 - Sets PC = rs + immediate
 - **no** multiplication by 2 (as imm represents absolute address now)
- Note: `jalr` is I-Format Instruction
- Example: jr psuedo-instructions
`jr ra # jalr x0, 0(ra)`
- Example: Call function at any 32-bit absolute address
`lui x1, <hi20bits>`
`jalr ra, x1, <lo12bits>`
- Example: Jump PC-relative with 32-bit offset
`auipc x1, <hi20bits>`
`jalr x0, x1, <lo12bits>`

RISC-V Addressing Summary

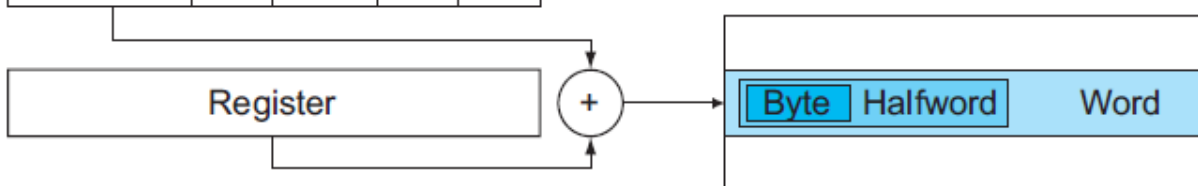
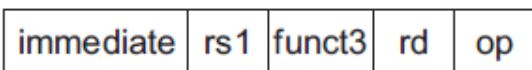
1. Immediate addressing



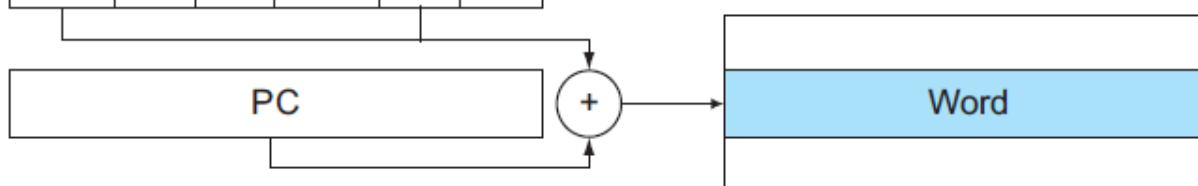
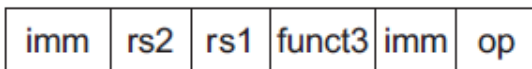
2. Register addressing



3. Base addressing



4. PC-relative addressing

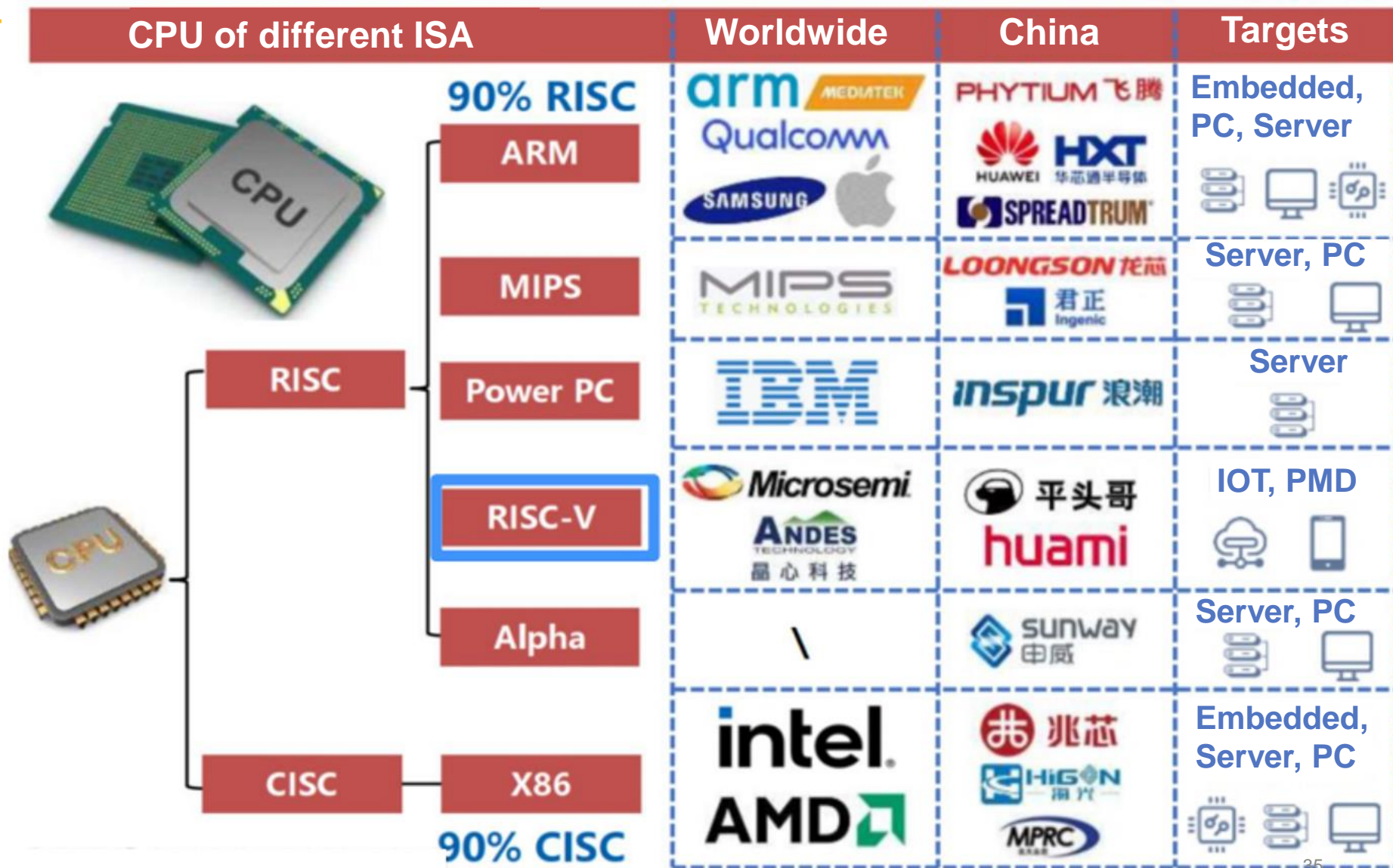




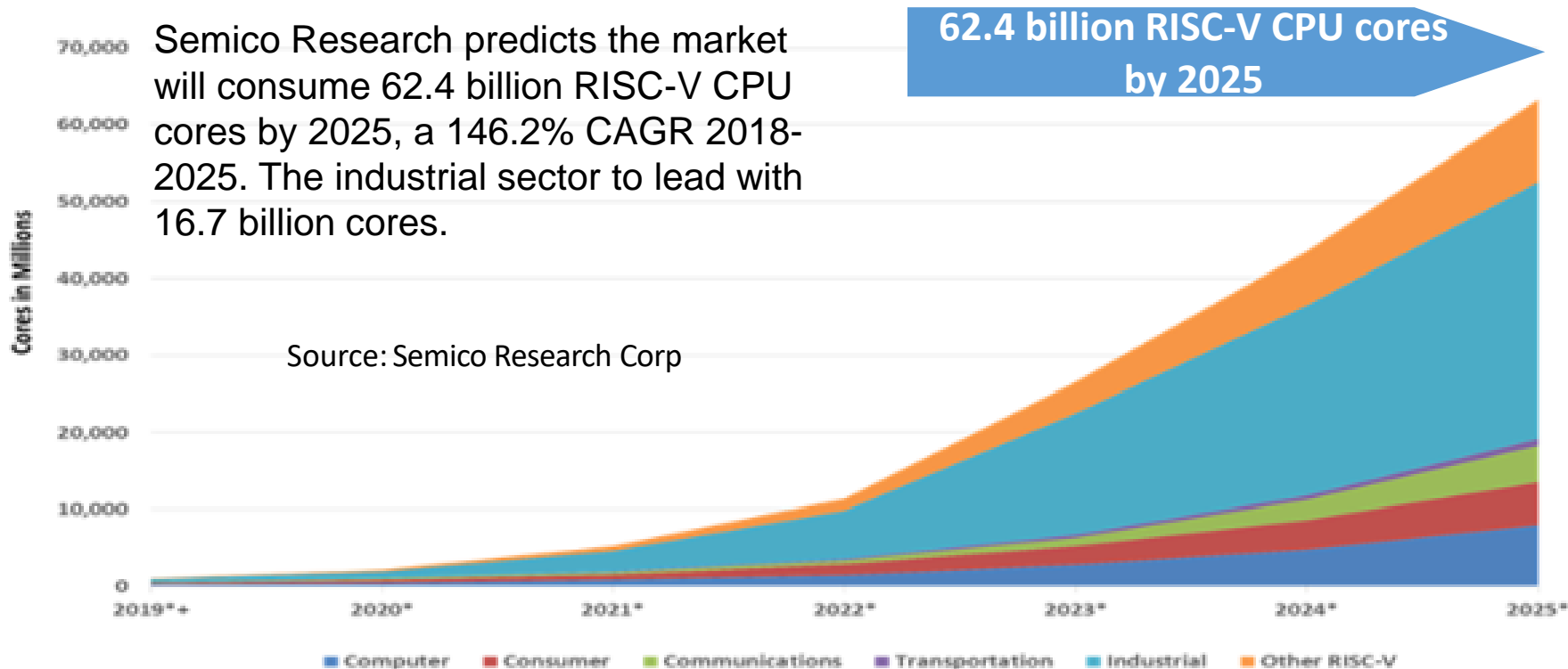
ISAs

- ARM
- x86
- RISC-V
- MIPS
- etc

RISC-V



RISC-V Market Share



MIPS Instructions

- MIPS: commercial predecessor to RISC-V
- Similar basic set of instructions
 - 32-bit instructions
 - 32 general purpose registers, register 0 is always 0
 - 32 floating-point registers
 - Memory accessed only by load/store instructions
 - Consistent use of addressing modes for all data sizes
- Different conditional branches
 - For $<$, $<=$, $>$, $>=$
 - RISC-V: blt, bge, bltu, bgeu
 - MIPS: slt, sltu (set less than, result is 0 or 1)
 - Then use beq, bne to complete the branch

MIPS vs. RISC-V

- Similar basic set of instructions

	MIPS32	RISC-V (RV32)
Date announced	1985	2010
License	Proprietary	Open-Source
Instruction size	32 bits	32 bits
Endianness	Big-endian	Little-endian
Addressing modes	5	4
Registers	32 × 32-bit	32 × 32-bit
Pipeline Stages	5 stages	5 stages
ISA type	Load-store	Load-store

ARM Market share

Markets for ARM in 2018

	Devices Shipped (Million of Units)	2018 Devices	Device CAGR	Chips/ Device	2018 Chips	Chip CAGR	Key Growth Areas for ARM
Mobile	Smart Phone	1,900	12%	3-5	6,500	11%	←
	Feature & Voice Phones	400	-13%	1-2	500	-14%	
	Mobile Computing* (apps only)	800	15%	1	800	15%	
Home	DTV & STB	560	4%	1-4	750	5%	←
	Home Networking	1,500	11%	1-3	2,250	18%	
	Consumer Entertainment	250	-7%	1-2	300	-10%	
Enterprise	PCs & Servers (apps only)	220	2%	1	220	2%	←
	Enterprise Networking	1,300	3%	1-2	1,400	3%	
	Computer Peripherals	800	15%	1-2	880	12%	
	Hard Disk & Solid State Drives	850	6%	1	850	6%	
Embedded	Automotive	3,800	5%	1	3,800	5%	←
	SmartCard	8,500	4%	1	8,500	4%	
	Microcontroller	15,000	11%	1	15,000	11%	
	Embedded Connectivity	3,000	-	1	3,000	n/a	
	Others **	1,500	8%	1	1,500	8%	
Total		40,500	10%		46,000	10%	

Source:
Gartner, IDC, SIA, and
ARM estimates

ARM Applications

Cortex[®]-M processors

MCU + DSP



RTOS

Smallest footprint / lowest power

Cortex[®]-R processors



Highest performance / real-time

Cortex[®]-A processors

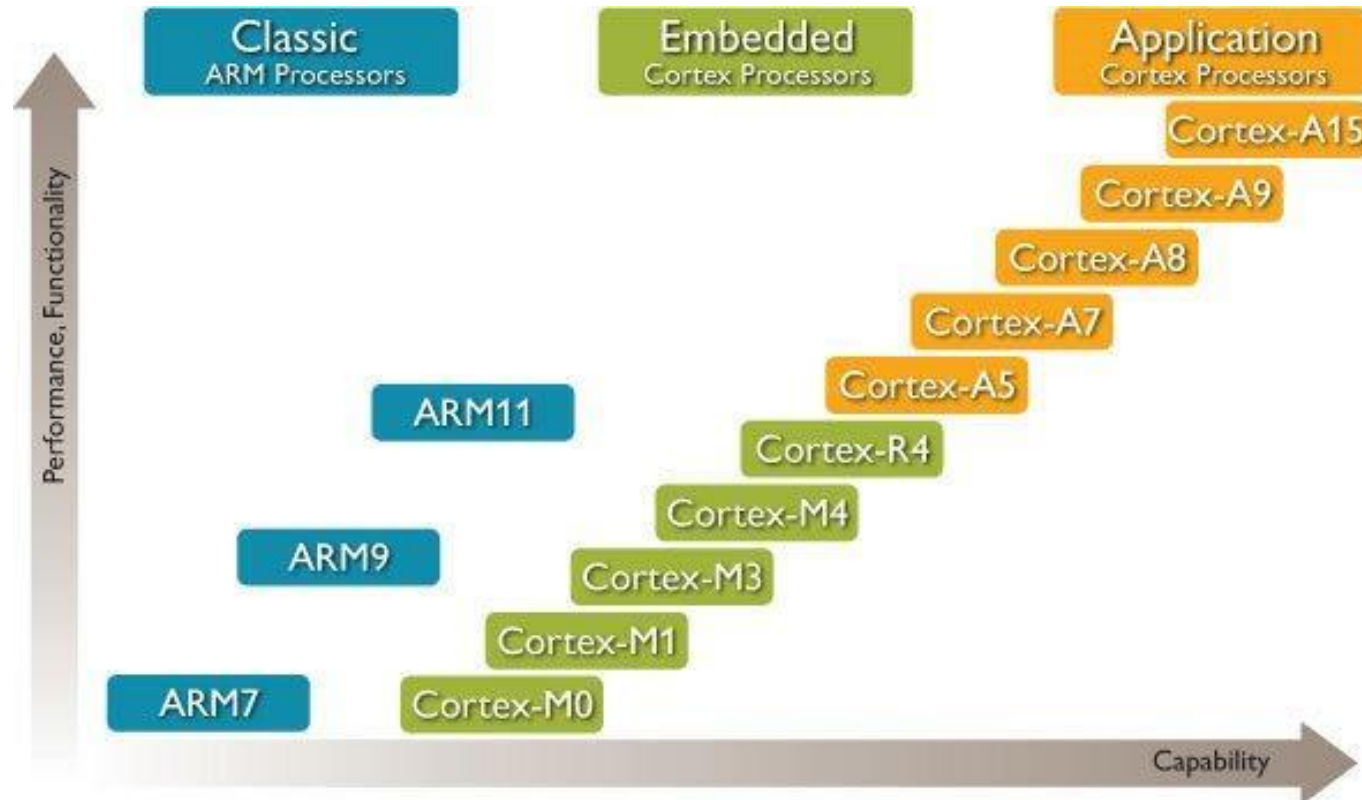


Rich OS

Highest performance



ARM CPU Series



Intel x86 Evolution: Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
• 8086	1978	29K	5-10
<ul style="list-style-type: none">• First 16-bit Intel processor. Basis for IBM PC & DOS• 1MB address space			
• 386	1985	275K	16-33
<ul style="list-style-type: none">• First 32 bit Intel processor , referred to as IA32• Added “flat addressing”, capable of running Unix			
• Pentium 4F	2004	125M	2800-3800
<ul style="list-style-type: none">• First 64-bit Intel processor, referred to as x86-64			
• Core 2	2006	291M	1060-3500
<ul style="list-style-type: none">• First multi-core Intel processor			
• Core i7	2008	731M	1700-3900
<ul style="list-style-type: none">• Four cores (our shark machines)			

Concluding Remarks

- Design principles
 1. Simplicity favors regularity
 2. Smaller is faster
 3. Make the common case fast
 4. Good design demands good compromises
- Layers of software/hardware
 - Compiler, assembler, hardware
- RISC-V: typical of RISC ISAs
 - c.f. x86