RISC-V Instruction Set Architecture

Chun-Jen Tsai

NYCU

09/22/2023

RISC-V Instruction Set Architecture

❑ The 5th generation RISC ISA design from UC Berkeley[†]

  ■ Work begins in 2010 in Parallel Computing Lab, Berkeley

  ■ License-free, royalty-free

  ■ Designed for all, from 32-bit μP to 128-bit supercomputers

  ■ Standard maintained by RISC-V Foundation (http://riscv.org)

❑ Why RISC-V?

  ■ The industry is tired of the ISA dominations by Intel and ARM
    $\rightarrow$ Could RISC-V be the solution?

  ■ ISA is not the key point, we just need a common machine language

[†] https://riscv.org/about/history/

RISC-V Registers

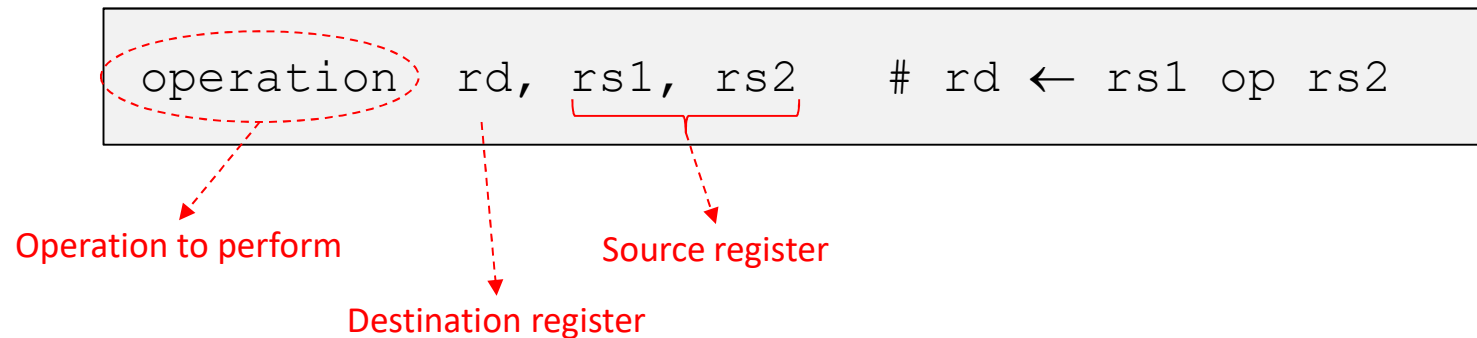❑ There are 32 general-purpose registers, `x0~x31`

- ■ Each register can be of XLEN bits, XLEN = 32, 64, or 128, depending on the ISA class: RV32, RV64, or RV128
- ■ Each register has an alias for programming convention:

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

†ABI stands for Application Binary Interface.

Instruction Format

❑ Classical three-address format are used:

```
operation  rd, rs1, rs2   # rd ← rs1 op rs2
```

Operation to perform

Destination register

Source register

❑ Little-endian format is adopted

❑ Minimal operations are defined in the ISA

- Only 47 instructions in the base instruction set
- For example, most ISA have a `mov` instruction to move data between registers. Here, you just use `addi` instead:

```
addi  rd, rs1, 0   # rd ← rs1
```

Instruction Classifications

❑ User-Level Base Integer Instructions (Base I)

- Load / Store
- Shift / Arithmetic
- Logical / Compare
- Branch / Jump-and-Link
- Synchronization / System
- Counter

❑ Multiplication and Division Instructions (Extension M)

❑ Atomic Instructions (Extension A)

❑ Privileged Instructions (for ISR and OS support)

❑ Aquila conforms to the RV32-IMA specification

Instruction Format

❑ The coding of an instruction is based on:
- Does it have a destination register?
- Does it have zero, one, or two source registers?
- Does it have immediate values?

❑ The encodings make sure that:
- `rd`, `rs1`, `rs2`, `funct3`, and `opcode` are in the same positions
- Sign bits of the immediate values are aligned
- `imm[]` bits positioned such that no shifting and minimal fanouts are required to form the actual immediate value

	31 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	func3	rd	opcode
I	imm[11:0]		rs1	func3	rd	opcode
S	imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
U	imm[31:12]				rd	opcode
SB	[12] imm[10:5]	rs2	rs1	func3	imm[4:1] [11]	opcode
UJ	[20] imm[10:1] [11]		imm[19:12]		rd	opcode
UI	imm[11:0]		rs1	func3	rd	opcode

Load / Store

❑ Load:

Instruction	Format	Mnemonic
Load Byte	I	LB rd, rs1, imm
Load Half Word	I	LH rd, rs1, imm
Load Word	I	LW rd, rs1, imm
Load Byte Unsigned	I	LBU rd, rs1, imm
Load Half Word Unsigned	I	LHU rd, rs1, imm

❑ Store:

Instruction	Format	Mnemonic
Store Byte	S	SB rs1, rs2, imm
Store Halfword	S	SH rs1, rs2, imm
Store Word	S	SW rs1, rs2, imm

❑ Instruction format:

	31 25	24 20	19 15	14 12	11 7	6 0
I	imm[11:0]		rs1	func3	rd	opcode
S	imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode

Shift / Arithmetic

❏ Shift:

Instruction	Format	Mnemonic
Shift Left	R	SLL rd, rs1, rs2
Shift Left Immediate	I	SLLI rd, rs1, imm
Shift Right	R	SRL rd, rs1, rs2
Shift Right Immediate	I	SRLI rd, rs1, imm
Shift Right Arithmetic	R	SRA rd, rs1, rs2
Shift Right Arithmetic Immediate	I	SRAI rd, rs1, imm

❏ Arithmetic:

Instruction	Format	Mnemonic
Add	R	ADD rd, rs1, rs2
Add Immediate	I	ADDI rd, rs1, imm
Subtract	R	SUB rd, rs1, rs2
Load Upper Immediate	U	LUI rd, imm
Add Upper Immediate to PC	U	AUIPC rd, imm

❏ Instruction format:

	31 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	func3	rd	opcode
I	imm[11:0]		rs1	func3	rd	opcode
U	imm[31:12]				rd	opcode

Logical / Compare

❑ Logical:

Instruction	Format	Mnemonic
XOR	R	XOR rd, rs1, rs2
XOR Immediate	I	XORI rd, rs1, imm
OR	R	OR rd, rs1, rs2
OR Immediate	I	ORI rd, rs1, imm
AND	R	AND rd, rs1, rs2
AND Immediate	I	ANDI rd, rs1, imm

❑ Compare:

Instruction	Format	Mnemonic
Set <	R	SLT rd, rs1, rs2
Set < Immediate	I	SLTI rd, rs1, imm
Set < Unsigned	R	SLTU rd, rs1, rs2
Set < Immediate Unsigned	I	SLTIU rd, rs1, imm

❑ Instruction format:

	31	25 24	20 19	15 14	12 11	7 6	0
R	funct7	rs2	rs1	func3	rd	opcode	
I	imm[11:0]		rs1	func3	rd	opcode	

Branch / Jump-and-Link

❑ Branch:

Instruction	Format	Mnemonic
Branch $=$	SB	BEQ rs1, rs2, imm
Branch $\neq$	SB	BNE rs1, rs2, imm
Branch $<$	SB	BLT rs1, rs2, imm
Branch $\geq$	SB	BGE rs1, rs2, imm
Branch $<$ Unsigned	SB	BLTU rs1, rs2, imm
Branch $\geq$	SB	BGEU rs1, rs2, imm

❑ Jump-and-Link:

Instruction	Format	Mnemonic
Jump & Link	UJ	JAL rd, imm
Indirect Jump & Link with Register	UI	JALR rd, rs1, imm

❑ Instruction format:

	31	25 24	20 19	15 14	12 11	7 6	0
SB	[12] imm[10:5]	rs2	rs1	func3	imm[4:1][11]	opcode	
UJ	[20] imm[10:1]	[11]	imm[19:12]		rd	opcode	
UI	imm[11:0]		rs1	func3	rd	opcode	

Synchronization / System

❑ Fence (barrier):

Instruction	Format	Mnemonic
Synch Thread	I	FENCE
Synch Instruction & Data	I	FENCE.I

❑ System call:

Instruction	Format	Mnemonic
System CALL	I	ECALL
System BREAK	I	EBREAK

❑ Instruction format:

31	25 24	20 19	15 14	12 11	7 6	0
I	imm[11:0]	rs1	func3	rd	opcode	

Fence/Barrier Instructions

❑ For advanced computers, a "memory cell" may not contain a deterministic value, for example:

- Different values may be mapped to an address due to caching
- Memory operations may not be executed in a predictable order
- A value written into a memory cell may read back differently

❑ There are two instructions to help resolve the issues: `fence` and `fence.i`

- `fence` is used to make sure a memory operation is completed before its next usage
- `fence.i` is used to guarantee the coherence of the instruction memory

Memory Barrier with `fence`

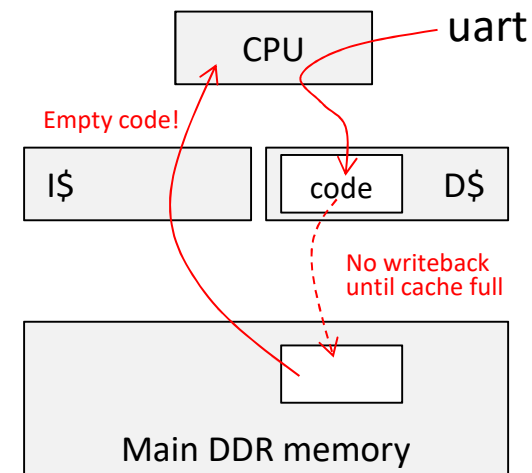❑ To ensure the ordering of memory operations, use the `fence` instruction:

```
sw   t0, 0(t2)
lw   t1, 4(t2)       v.s.
sw   zero, 8(t2)
```

```
sw   t0, 0(t2)          → predecessor
fence w, rw        ----→ memory barrier
lw   t1, 4(t2)
sw   zero, 8(t2)       → successors
```
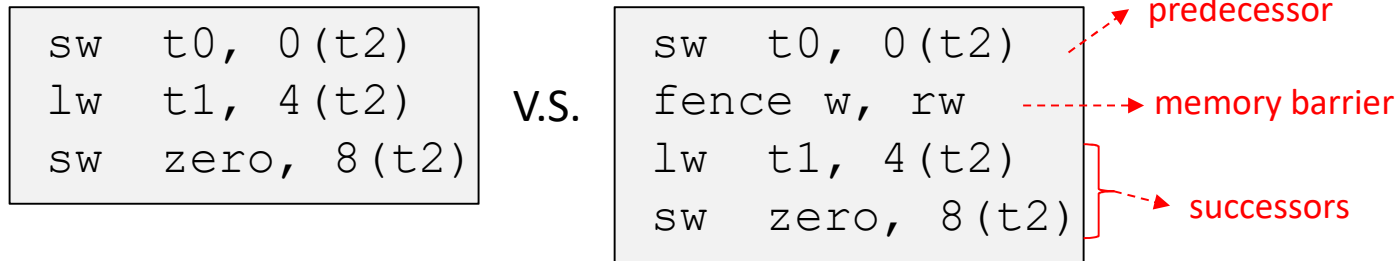
■ The first code does not impose any load/store ordering

– No apparent data dependency here

– The compiler can reorder the instructions freely? → big mistake!

■ The 2nd code force the store operation `sw t0,0(t2)` to be executed before the load/store operations after `fence`.

Cache Coherence with `fence.i`

❑ On systems without a coherent cache, to make sure I and D caches are consistent, use `fence.i`:

- Ensure cache coherence by programmer; low hardware cost
- Negative impact on performance
- Cannot guarantee coherence across multiple cores

❑ For example, in `uartboot.c`:

```
// load a program from UART to DDR DRAM
    . . .

// Call the entry point for execution.
asm volatile ("fence.i");
asm volatile ("lui t0, %hi(prog)");
asm volatile ("lw ra, %lo(prog)(t0)");
asm volatile ("jalr ra, ra, 0");
```

CPU

uart

Empty code!

I$

code   D$

No writeback until cache full

Main DDR memory

System / Counter (Pseudo Instr.)

❑ Counter:

Instruction	Format	Mnemonic
Read CYCLE	I	RDCYCLE rd
Read CYCLE Upper Half	I	RDCYCLEH rd
Read TIME	I	RDTIME rd
Read TIME Upper Half	I	RDTIMEH rd
Read INSTRET	I	RDINSTRET rd
Read INSTRET Upper Half	I	RDINSTRETH rd

❑ Instruction format:

31		20	15	12	7	0
I	imm[11:0]		rs1	func3	rd	Opcode

RISC-V Base Spec. and Extensions

❑ RISC-V Specification is divided into base ISA and extensions:

Base	Version	Status
RVWMO	2.0	Ratified
RV32I	2.1	Ratified
RV64I	2.1	Ratified
RV32E	1.9	Draft
RV128I	1.7	Draft

Extension	Version	Status
M	2.0	Ratified
A	2.1	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
Counters	2.0	Draft
L	0.0	Draft
B	0.0	Draft
J	0.0	Draft
T	0.0	Draft
P	0.2	Draft
V	0.7	Draft
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
Zam	0.1	Draft
Ztso	0.1	Frozen

Memory model specification

Base integer instructions

Optional instructions

OS and system state related instructions

Instruction Extension M

❑ Multiplication and Division

Instruction	Format	Mnemonic
Multiply	R	MUL rd, rs1, rs2
Multiply Upper Half	R	MULH rd, rs1, rs2
Multiply Half Signed-Unsigned	R	MULHSU rd, rs1, rs2
Multiply Upper Half Unsigned	R	MULHU rd, rs1, rs2
Divide	R	DIV rd, rs1, rs2
Divide Unsigned	R	DIVU rd, rs1, rs2
Remainder	R	REM rd, rs1, rs2
Remainder Unsigned	R	REMU rd, rs1, rs2

❑ Instruction format:

31		20	15	12	7	0
R	func7	Rs2	rs1	func3	rd	opcode

Instruction Extension A

❑ Atomic Instructions

Instruction	Format	Mnemonic
Load Reserved	R	LR.W rd, rs1, rs2
Store Conditional	R	SC.W rd, rs1, rs2
Atomic Swap	R	AMOSWAP.W rd, rs1, rs2
Atomic Add	R	AMOADD.W rd, rs1, rs2
Atomic XOR	R	AMOXOR.W rd, rs1, rs2
Atomic AND	R	AMOAND.W rd, rs1, rs2
Atomic OR	R	AMOOR.W rd, rs1, rs2
Atomic Minimum	R	AMOMIN.W rd, rs1, rs2
Atomic Maximum	R	AMOMAX.W rd, rs1, rs2
Atomic Minimum Unsigned	R	AMOMINU.W rd, rs1, rs2
Atomic Maximum Unsigned	R	AMOMAXU.W rd, rs1, rs2

Lock-free { Load Reserved, Store Conditional }

Lock-based { Atomic Swap … Atomic Maximum Unsigned }

❑ Instruction format:

31		20	15	12	7	0
R	func7	rs2	rs1	func3	rd	opcode

Lock-Based Atomic Instructions

❑ A lock-based atomic instruction, e.g. `AMOSWAP.W`, read-then-write a memory cell without any interruption

  ■ In concurrent programming, a mutex (or a critical section) is often implemented using an atomic operation

  ■ For a single-core system, interrupt-masking can be used to achieve the goal $\rightarrow$ clumsy but working

❑ Example:

```
    li t0, 1                    # Initialize swap value.

again:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez t0, again             # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
```

Issues with Lock-based Instructions

❑ Lock-base atomic instructions simply lock the "memory bus" for a couple of cycles until the operation is done

- Each instruction *usually* takes more than one cycles to execute, known as Read-Modify-Write (RMW) cycles
- Easy to implement in hardware, transparent to software

❑ Hurt performance, especially in a multicore system that uses a shared bus topology

- The shared bus will be locked frequently for unrelated operations when you have thousands of threads running concurrently

Inefficiency of Lock-based Scheme

❑ Lock-based atomic access to one memory cell stops all accesses to all other memory cells
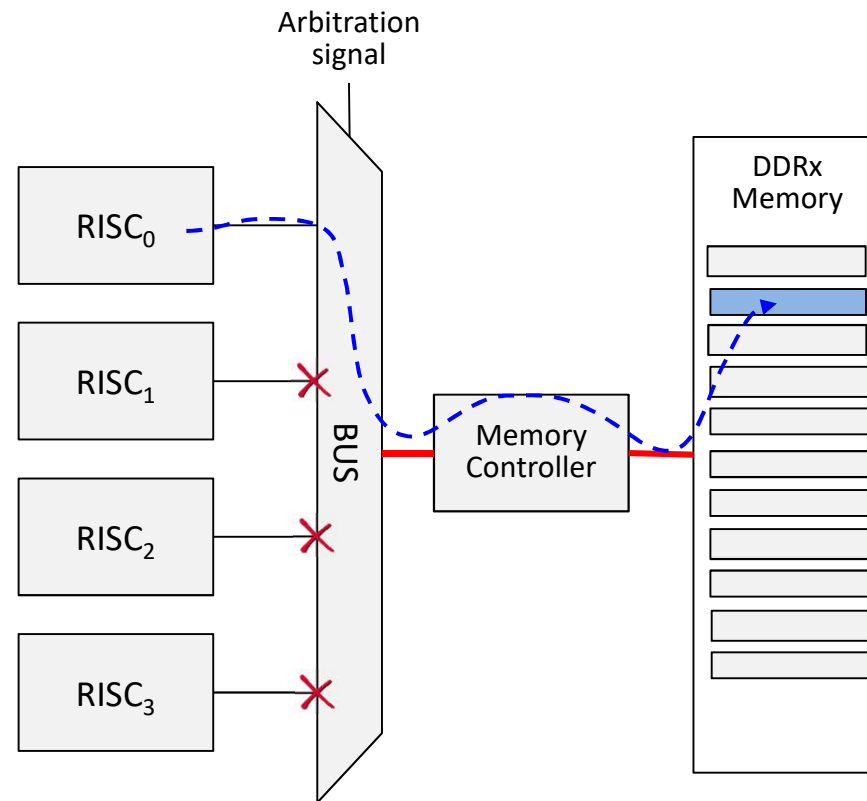
Lock-Free Atomic Instructions

❑ Load-Reserved (LR) and Store-Conditional (SC) instructions were proposed back in the 1970's:

   ■ A pair of LR and SC instructions are used to implement accesses to mutex variables

   ■ LR returns the current value of a memory location

   ■ SC to the same memory location stores a new value only if no updates have occurred to that location since the LR

❑ With the LR/SC scheme, it is guaranteed that only one of the threads that try to acquire a mutex can success

LR/SC Usage Example

❑ We can use LR/SC to implement a compare-and-swap
function without locking the bus:

```
        # a0 holds address of memory location
        # a1 holds expected value
        # a2 holds desired value
        # a0 holds return value, 0 if successful, !0 otherwise
cas:
    lr.w t0, (a0)       # Load original value.
    bne t0, a1, fail    # Doesn't match, so fail.
    sc.w a0, a2, (a0)   # Try to update.
    jr ra               # Return.
fail:
    li a0, 1            # Set return to failure.
    jr ra               # Return
```

Pseudo Assembly Instructions

❑ Some pseudo instructions[†] are defined to facilitate assembly coding for programmers

  ■ These instructions will be mapped to some base instructions by the assembler

| Pseudo instruction	Base instruction	meaning
j offset	jal x0, offset	Jump
jal offset	jal x1, offset	Jump and link
jr rs	jalr x0, 0(rs)	Jump register
jalr rs	jalr x1, 0(rs)	Jump and link register
Ret	jalr x0, 0(x1)	Return from subroutine
call offset	auipc x1, offset[31:12]+offset[11]	Call far-away subroutine
	jalr x1, offset[11:0](x1)	
nop	addi x0, x0, 0	No operation
mv rd, rs	addi rd, rs, 0	Copy register
li rd, immediate	*Myriad sequences*	Load a large immediate

† For all pseudo instructions, see Chap 25 in *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, July 2020.

Using Immediate Values

❑ An immediate value is a numerical constant that stores inside the instruction codeword:

```
addi  x3, x5, -10    # x3 ← x5 - 10
```

Stored in bits 31 to 20 in the instruction!

31	20	15	11	7	0
I	imm[11:0]	rs1	func3	rd	opcode
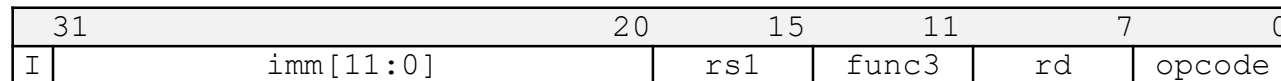
❑ Since for type-I instruction code, 12 bits are used for an immediate value, the range is only –2048 ~ 2047.

32-bit Number in 32-bit Instruction?

- ❑ For a 32-bit RISC processor, each instruction is usually coded in 32-bit (or less)
  - Facilitate single-cycle instruction fetch
  - On the contrary, a CISC processor typically uses variable-length coding of instructions → large immediate value can be encoded directly in the instruction codeword

- ❑ For RISC-V, how do you encode a 32-bit number in a 32-bit instruction as an immediate value?

Load a Large Constant

❑ The best way to load a large constant number into a register depends on the number itself:

- Universal way:

```
lui  x3, 0xABCDE      # x3 ← 0xABCDE000
addi x3, x3, 0x123    # x3 ← 0xABCDE123
```

Beware of sign-extension when MSB = 1!

- Case-by-case tricks:

```
addi x3, x0, -1       # x3 ← 0xFFFFFFFF
```

❑ A no-brainer pseudo instruction does the trick for you:

```
li x3, 0xABCDE123     # x3 ← 0xABCDE123
```

Accessing Data in Main Memory

❑ Indirect references (e.g. pointers) are often used to access data in main memory:

```
# To compute A[5] = A[3]+1, where A[] is an int array.
 li  x3,  _A_addr  # x3   ← &A[0]
 lw  x10, 12(x3)   # x10  ← A[3]
 add x10, X10, 1   # x10  ← x10 + 1
 sw  x10, 20(x3)   # A[5] ← x10
```

❑ Note that the offsets (12 and 20 in the above example) in indirect references are byte-offset. You must multiply a word address by 4 to get the correct offset.

Loading/Storing a Byte

❑ For a 32-bit CPU, loading 16-bit or 8-bit data from the main memory requires handling of sign-extension and address alignment:

- Load byte (`lb`) or load half word (`lh`) must handle sign extension. No need to do sign-extension for store instructions since main memory contains 8-bit cells.
- Address alignment problem comes from the bus protocol

❑ In reality, the bus width from the processor core to the main memory is 32 (for RV32).

- The transmission is always 32-bit even for `lb`/`sb`.
- A byte-select signal tells the CPU and the memory which bytes contains valid data

Word-Alignment Bus Issues

❑ Memory controller takes only word addresses and transmits word streams instead of byte streams

 ■ If the μP needs a data that is not aligned to word addresses, two requests must be made to the memory controller

 ■ Some μPs do not handle this due to inefficiency. As a result, an misaligned exception will be triggered to alert the programmer

❑ A compiler flag can be used to avoid generation of unaligned data accesses

 ■ Be careful when you link a library that is compiled without this "strict-alignment" flag

PC-Relative Addressing

❑ Accessing an absolute address cause trouble for program relocation

■ Virtual memory mapping is a good solution, but expensive

■ PC-relative addressing partially solve this problem with little extra cost

❑ In RISC-V, `AUIPC` is used for PC-relative addressing:

```
        j     Label        # Branch to Label
        .word 0xABCDE123 # Raw data (constant pool)
Label: auipc x3, -4       # x3 ← PC-4
        lw    x5, 0[x3]  # x5 ← 0xABCDE123
```

Function Call and Return

❑ For near function calls, Jump-and-Link is used:

```
jal ra, <Function>  # Call <Function>
```

- The function (callee) must be within $\pm 2^{18}$ instructions of PC
  - the `imm` value of `jal` is of 21-bit (bit 0 always zero), but you must exclude the sign bit and take into account word-alignment
- The return address (PC + 4) is stored in `ra` (i.e. `x1`)

❑ For function return, `jalr` is used :

```
jalr x0, 0(ra)         # Return to caller
```

- JALR is also used for far-away function calls.

```
jalr ra, 0(x3)         # Call function pointer x3
```

Calling Convention

❑ Each CPU ISA has to defines its Application Binary Interface (ABI) or calling convention

- This way different compilers can produce interchangeable object files for the linkers and loaders

❑ ABI must address the following issues:

- How to pass function parameters
- How to return function values
- How to manage local variables on the stacks
- How to save/restore the return address and stack pointers

Parameter/Return Value Passing (1/2)

❑ The RISC-V calling convention passes arguments in registers whenever possible

- Up to eight integer registers, `a0-a7` are used for this purpose.
- Arguments smaller than a pointer-word are passed in the least-significant bytes of the argument registers.
- Parameters that cannot be passed in registers will be passed using the stack

C type	Description	Bytes in RV32	Bytes in RV64
char	Character value/byte	1	1
short	Short integer	2	2
int	Integer	4	4
long	Long integer	4	8
long long	Long long integer	8	8
void*	Pointer	4	8
float	Single-precision float	4	4
double	Double-precision float	8	8
long double	Extended-precision float	16	16

Parameter/Return Value Passing (2/2)

❑ A `struct` parameter can be passed partially in registers and partially on stack.

❑ Values are returned in integer registers `a0` and `a1`.

- Larger return values are passed entirely in memory; the caller allocates this memory (from the stack) and passes its pointer as an implicit first parameter to the callee.

❑ The stack pointer must be 16-byte aligned for each increment/decrement operation

Call Sequence

1. Caller puts arguments in `a0-a7`
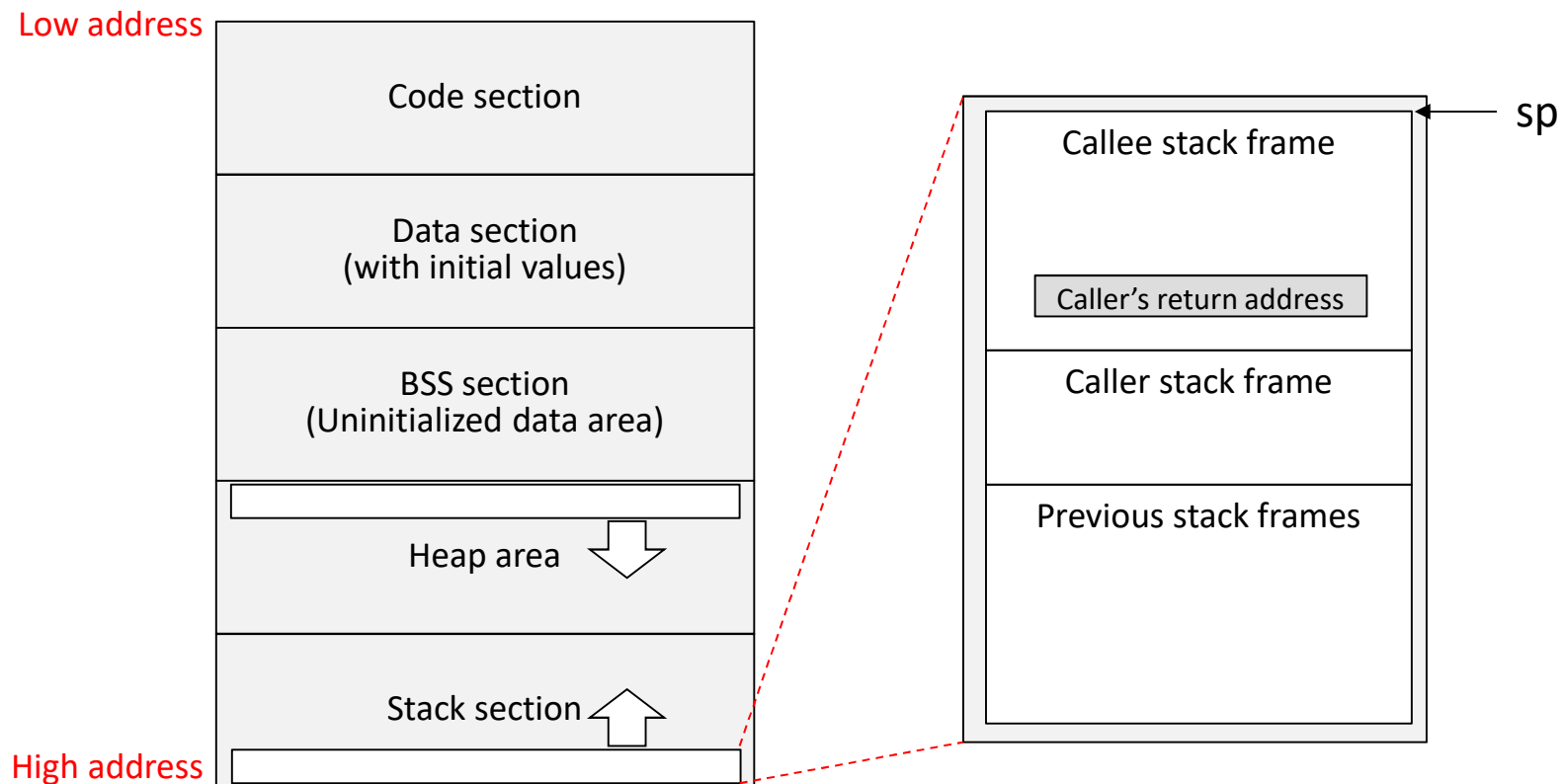2. Caller jumps to the callee by:

```
jal ra, <Function>  # <Function> is the callee.
```

3. Callee decrements `sp` to allocate stack frame for special registers, parameters, and local variables
4. Callee performs the calculations, put result in `a0-a1`
5. Callee restores special registers, increments `sp` to free its stack frame
6. Callee returns control to the caller by:

```
jalr x0, 0(ra)
```

Runtime Memory Map

❑ The runtime memory organization for ABI

Low address

| Code section |
| Data section (with initial values) |
| BSS section (Uninitialized data area) |
| |
| Heap area ⬇ |
| Stack section ⬆ |
| |

High address

sp →

| Callee stack frame |
| Caller's return address |
| Caller stack frame |
| Previous stack frames |

A Simple Example

❑ The assembly code for `getchar()` in the Aquila library is as follows (compiler optimization is `-Os`):

```
inline int getchar(void) { return (int) inbyte(); }
```

```
000000d8 <getchar>:
  d8:    ff010113            addi    sp,sp,-16
  dc:    00112623            sw      ra,12(sp)
  e0:    f89ff0ef            jal     ra,68 <inbyte>
  e4:    00c12083            lw      ra,12(sp)
  e8:    01010113            addi    sp,sp,16
  ec:    00008067            ret
```

RISC-V calling convention

```
00000068 <inbyte>:
  68:    000017b7            lui     a5,0x1
  6c:    8a87a703            lw      a4,-1880(a5) # 8a8 <uart_status>
            . . . . . . . . . .

  80:    8b07a783            lw      a5,-1872(a5) # 8b0 <uart_rxfifo>
  84:    0007a503            lw      a0,0(a5)
  88:    0ff57513            andi    a0,a0,255
  8c:    00008067            ret
```

No "sp" adjustment because this is a bottom-level function.

Return value of unsigned char in a0.

Conclusions

❑ Today, ISA design is no rocket science

- The simpler the ISA design, the better

❑ The three mainstream ISAs may converge in the future

- X86/X86-64 is obsolete, even Intel is trying to move away from the legacy design
- ARM is the dominating ISA for mobile/embedded devices, and is gaining momentum for the server market; But, it may not have a bright future in the long run
- RISC-V may become the dominating ISA for instruction-based processors ten years from now

References

- ❑ RISC-V ISA Specifications:
  https://riscv.org/technical/specifications/
- ❑ RISC-V Assembly Language Manual:
  https://github.com/riscv/riscv-asm-manual