

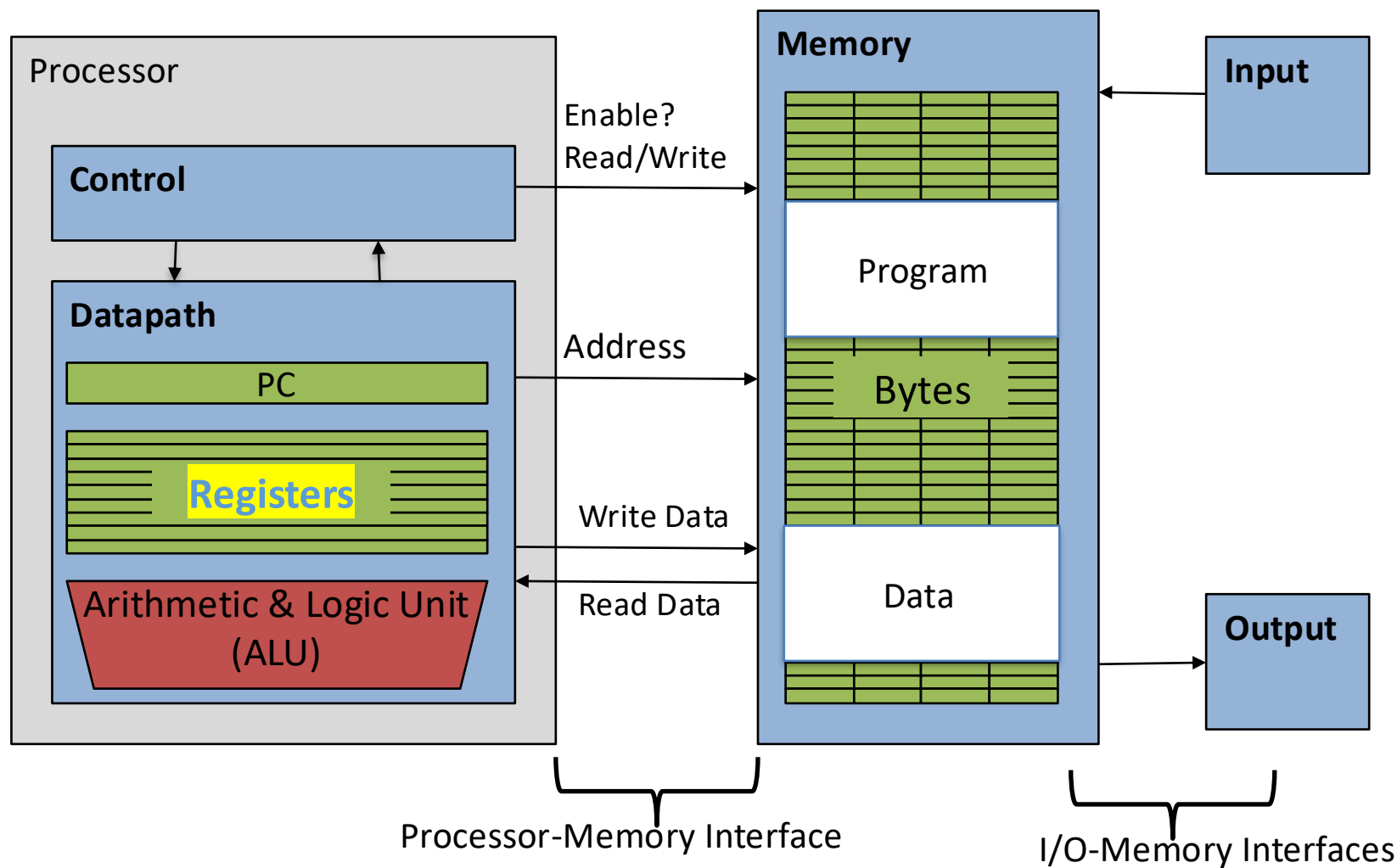


CS110A Introduction to Computer Systems

ISA and RISC-V

Chundong Wang

A High-level View of Computer





Introduction to ISA

- **Instruction Set Architecture**
- Part of the **abstract model** of a computer that defines how the CPU is controlled by the software; **interface** between the hardware and the software;
- Programmers' **manual** because it is the portion of the machine that is visible to the assembly language programmers, the compiler writers, and the application programmers;
- Defines the supported data types, the registers, how the hardware manages main memory, key features, instructions that can be executed (**instruction set**), and the input/output model of multiple ISA implementations;
- ISA can be extended by adding instructions or other capabilities (ARM).

ISA: Manual for building microarchitecture or processors

Microarchitecture: Implementation of an ISA

Popular ISAs

X86/AMD64

- ◆ Dominant architecture for personal computers and servers
- ◆ Name derived from Intel 8086/80186/80286...
- ◆ Multiple version: X86-16, X86-32 (IA-32), X86-64 (AMD64)
- ◆ Extensions such as MMX, SSE, etc.

Major vendors

- Intel, AMD, VIA, Zhaoxin, DM&P, RDC
- To OEM (original equipment manufacturer)

CISC (complex instruction set computer)

- Variable-length instructions
- Allow memory access with instructions other than load or store
- VAX architecture had an instruction to multiply polynomials!

add: X86 integer addition

Syntax

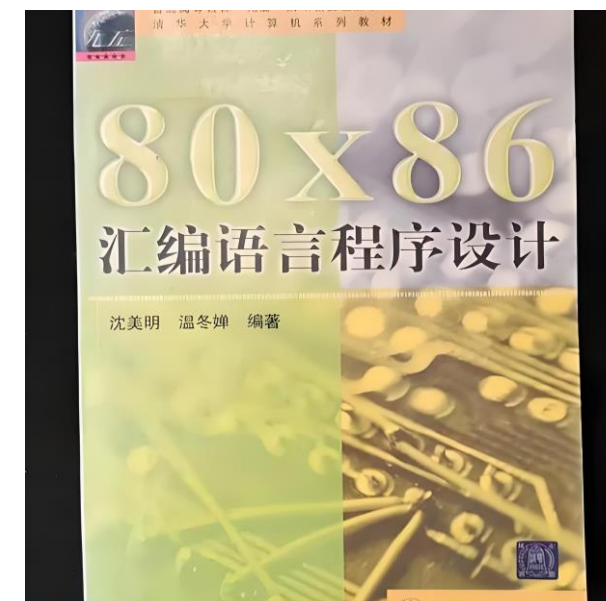
add <reg>,<reg>

add <reg>,<mem>

add <mem>,<reg>

add <mem>,imm*

... ..





Popular ISAs

ARM

- Dominant architecture for embedded devices
- Advanced RISC Machine
- Multiple version: ARMv1-ARMv9

add: ARM addition

Syntax

add(S) <reg>,<reg>,<reg or imm>

Major vendors

- Apple, Huawei, Qualcomm, Xilinx, etc.
- ARM sells IP cores to IC vendors (core licence)
- IC vendors sell MCU/CPU/SoC to OEM or for self use

RISC (reduced instruction set computer)

- 32-bit fixed-length instructions (not actually for Thumb-16)
- Allow memory access with only load or store instructions
- Simpler to design hardware. Generally generate smaller heat



RISC vs. CISC

Disassembly of section __TEXT,__text:

0000000000000000 <ltmp0>:

```

0: ff c3 00 d1 : sub sp, sp, #48
4: fd 7b 02 a9 : stp x29, x30, [sp, #32]
8: fd 83 00 91 : add x29, sp, #32
c: 08 00 80 52 : mov w8, #0
10: e8 0f 00 b9 : str w8, [sp, #12]
14: bf c3 1f b8 : stur wzr, [x29, #-4]
18: 48 9a 80 52 : mov w8, #1234
1c: a8 83 1f b8 : stur w8, [x29, #-8]
20: 28 1c 82 52 : mov w8, #4321
24: a8 43 1f b8 : stur w8, [x29, #-12]
28: a8 83 5f b8 : ldur w8, [x29, #-8]
2c: a9 43 5f b8 : ldur w9, [x29, #-12]
30: 08 01 09 0b : add w8, w8, w9
34: e8 13 00 b9 : str w8, [sp, #16]
38: e9 13 40 b9 : ldr w9, [sp, #16]
3c: e8 03 09 aa : mov x8, x9
40: e9 03 00 91 : mov x9, sp
44: 28 01 00 f9 : str x8, [x9]
48: 00 00 00 90 : adrp x0, 0x0 <ltmp0+0x48>
4c: 00 00 00 91 : add x0, x0, #0
50: 00 00 00 94 : bl 0x50 <ltmp0+0x50>
54: e0 0f 40 b9 : ldr w0, [sp, #12]
58: fd 7b 42 a9 : ldp x29, x30, [sp, #32]
5c: ff c3 00 91 : add sp, sp, #48
60: c0 03 5f d6 : ret

```

Assembly

Compiled on Mac machine using ARM CPU

0000000000000054 <main>:

```

54: 55 : push %rbp
55: 48 89 e5 : mov %rsp,%rbp
58: 48 83 ec 30 : sub $0x30,%rsp
5c: e8 00 00 00 00 : call 61 <main+0xd>
61: c7 45 fc d2 04 00 00 : movl $0x4d2,-0x4(%rbp)
68: c7 45 f8 e1 10 00 00 : movl $0x10e1,-0x8(%rbp)
6f: 8b 55 fc : mov -0x4(%rbp),%edx
72: 8b 45 f8 : mov -0x8(%rbp),%eax
75: 01 d0 : add %edx,%eax
77: 89 45 f4 : mov %eax,-0xc(%rbp)
7a: 8b 45 f4 : mov -0xc(%rbp),%eax
7d: 89 c2 : mov %eax,%edx
7f: 48 8d 05 00 00 00 00 : lea 0x0(%rip),%rax # 86 <main+0x32>
86: 48 89 c1 : mov %rax,%rcx
89: e8 72 ff ff ff : call 0 <printf>
8e: b8 00 00 00 00 : mov $0x0,%eax
93: 48 83 c4 30 : add $0x30,%rsp
97: 5d : pop %rbp
98: c3 : ret
99: 90 : nop
9a: 90 : nop
9b: 90 : nop
9c: 90 : nop
9d: 90 : nop
9e: 90 : nop
9f: 90 : nop

```

Assembly

Compiled on Windows machine using Intel CPU



Popular ISAs

RISC philosophy (John Cocke IBM, John Hennessy Stanford, David Patterson Berkeley, 1980s)

Hennessy & Patterson won ACM A.M. Turing Award (2017)

Reduced Instruction Set Computer (RISC)

- Keep the instruction set small and simple, makes it easier to build fast hardware.
- Let software do complicated operations by composing simpler ones.

Popular ISAs

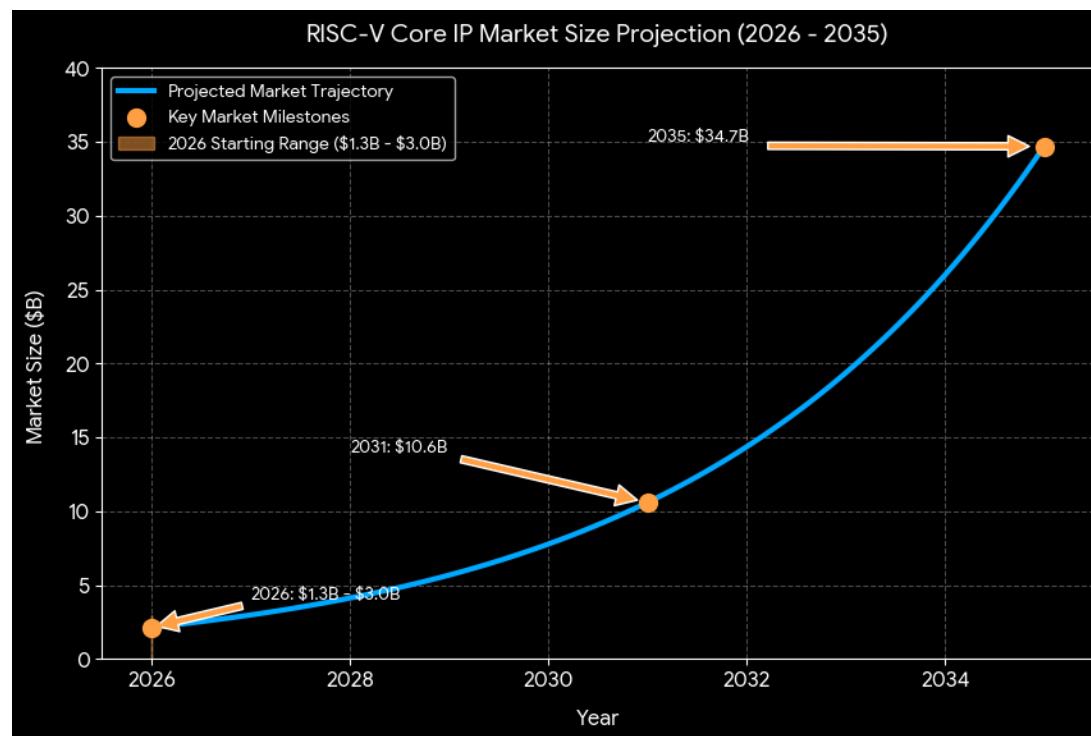
RISC-V (pronounced “risk-five”)

- Started as a summer project in UC Berkeley, 2010
- The ISA itself is published in 2011 as open source
- RISC-V foundation formed 2015 to own, maintain and publish IP related to RISC-V’s definition (a nonprofit business association)

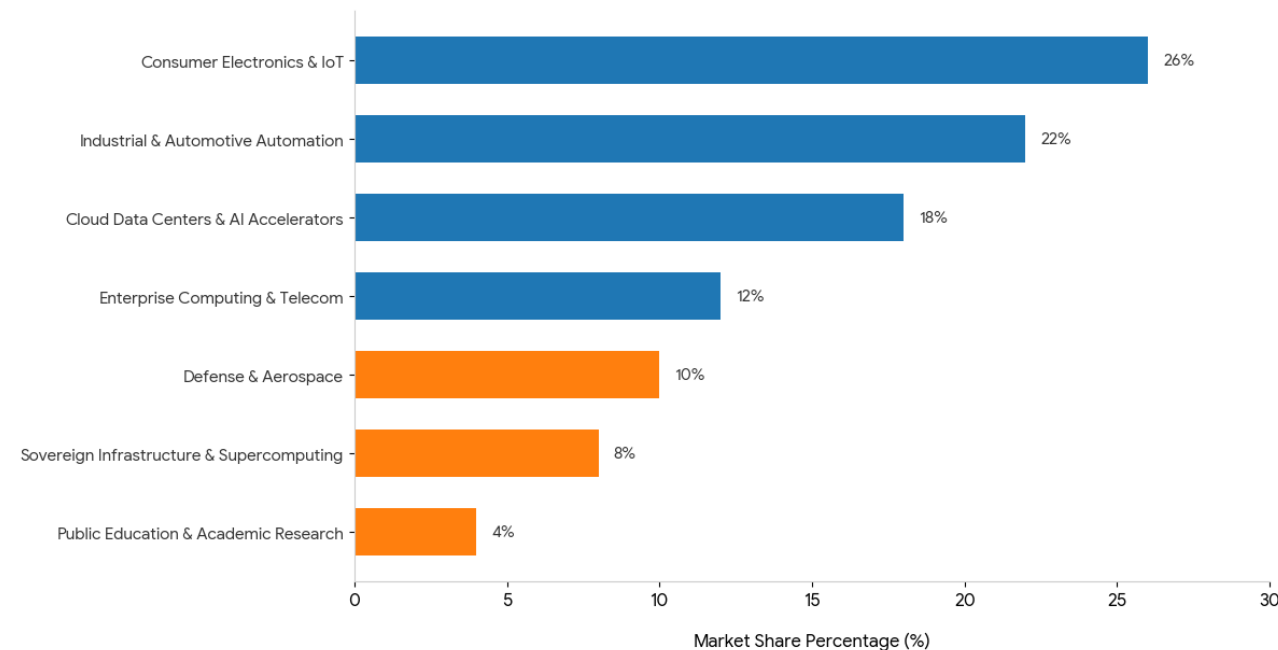
More than 3,100 members and still growing

- Alibaba Cloud: T-Head 玄铁 C series; E series, and R series
- Huawei: Hi3861V100 SoC for IoT/smart home
- Tencent: a premier member
- Intel, Google, Meta, SiFive, AMD/Xilinx, etc.
- ShanghaiTech hold several RISC-V Summits China recent years!
- Other ISA examples: MIPS, IBM/Motorola PowerPC (quite old Mac), Intel IA64, ...

The usage of RISC-V



An estimate of market size by Google Gemini



RISC-V chip utilization (market share by domain)

□ Why RISC-V instead of Intel x86?

- RISC-V is simple, elegant and open-source. Don't want to get bogged down in gritty details.

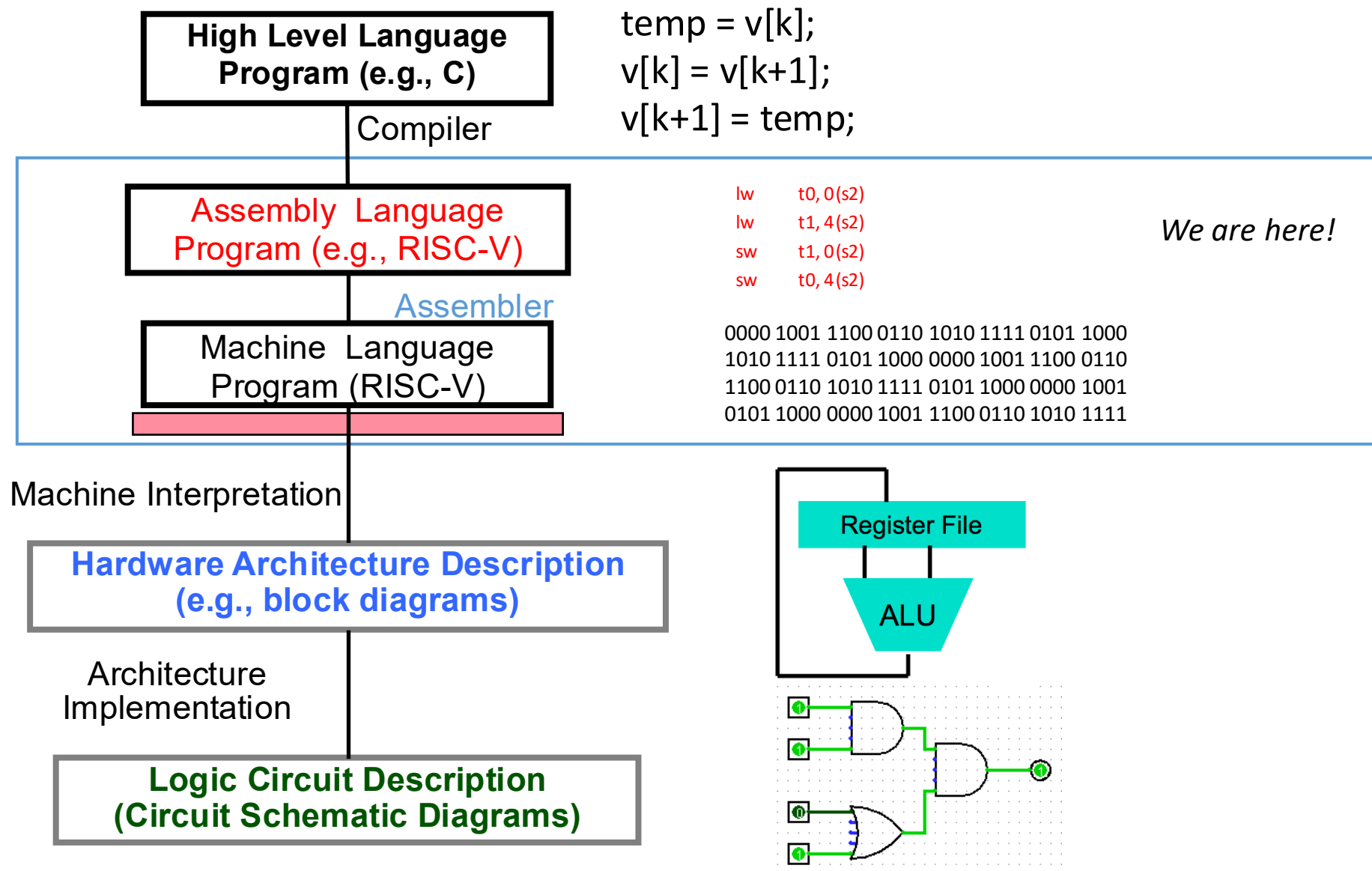
□ It is flexible

- Enabled by different extensions

Name	Description
Base	
RV32I	Base Integer Instruction Set, 32-bit
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 register
RV64I	Base Integer Instruction Set, 64-bit
RV64E	Base Integer Instruction Set (embedded), 64-bit
RV128I	Base Integer Instruction Set, 128-bit
Extension	
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point

$$RVG = RVI + M + A + F + D$$

Where are we?



Assembly Language

- Basic job of a CPU: execute a series of instructions.
- Instructions are the primitive operations that the CPU may execute.
- Basic job of a instruction: change the state of a computer.

Disassembly of section __TEXT,__text:

```
0000000000000000 <tmp0>:
 0: ff c3 00 d1  sub sp, sp, #48
 4: fd 7b 02 a9  stp x29, x30, [sp, #32]
 8: fd 83 00 91  add x29, sp, #32
 c: 08 00 80 52  mov w8, #0
10: e8 0f 00 b9  str w8, [sp, #12]
14: bf c3 1f b8  stur wzr, [x29, #-4]
18: 48 9a 80 52  mov w8, #1234
1c: a8 83 1f b8  stur w8, [x29, #-8]
20: 28 1c 82 52  mov w8, #4321
24: a8 43 1f b8  stur w8, [x29, #-12]
28: a8 83 5f b8  ldur w8, [x29, #-8]
2c: a9 43 5f b8  ldur w9, [x29, #-12]
30: 08 01 09 0b  add w8, w8, w9
34: e8 13 00 b9  str w8, [sp, #16]
38: e9 13 40 b9  ldr w9, [sp, #16]
3c: e8 03 09 aa  mov x8, x9
40: e9 03 00 91  mov x9, sp
44: 28 01 00 f9  str x8, [x9]
48: 00 00 00 90  adrp x0, 0x0 <tmp0+0x48>
4c: 00 00 00 91  add x0, x0, #0
50: 00 00 00 94  bl 0x50 <tmp0+0x50>
54: e0 0f 40 b9  ldr w0, [sp, #12]
58: fd 7b 42 a9  ldp x29, x30, [sp, #32]
5c: ff c3 00 91  add sp, sp, #48
60: c0 03 5f d6  ret
```

ARM Assembly

Compiled on Mac machine using ARM CPU

Exercise 1: Can we execute this assembly on an x86 CPU?

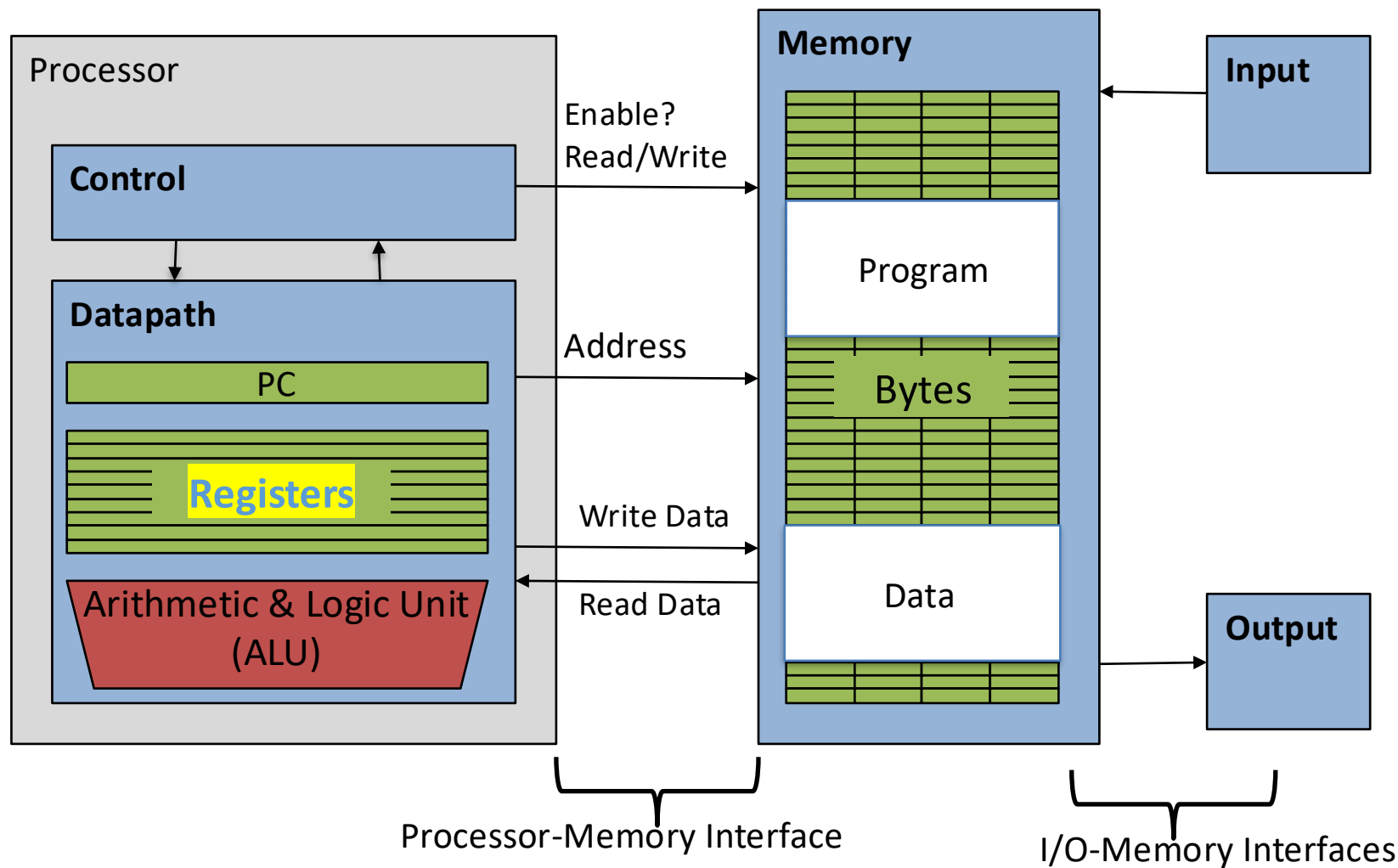
Exercise 2: Can we use a Linux OS to run ARM assembly?



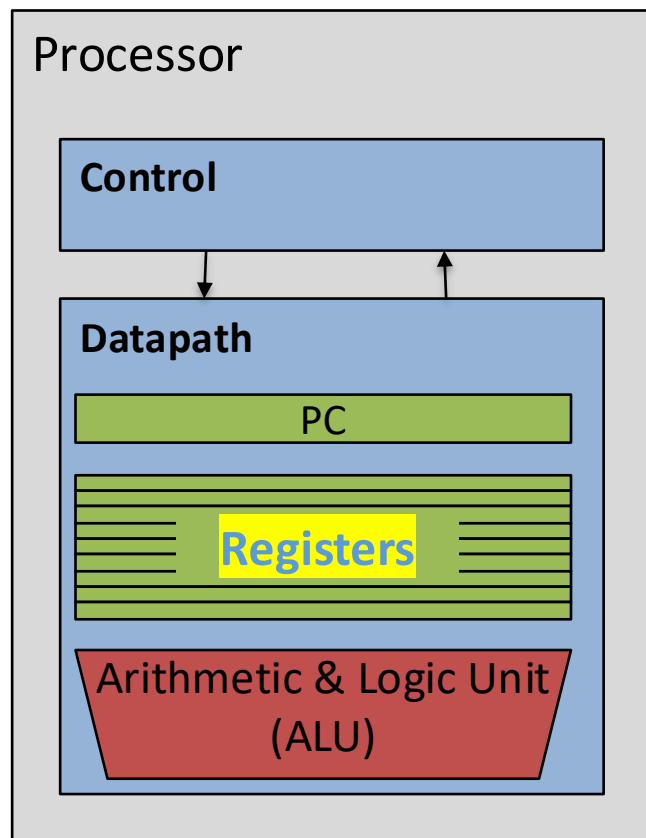
CPU State: Assembly Registers

- Unlike C or Java, assembly cannot use variables
 - Keep assembly/computer hardware abstract simple
- Assembly operands are registers
 - Limited number of special locations/memory built directly into the CPU
 - Operations can only be performed on these registers in RISC-V
- Benefit: Since registers are directly in hardware (CPU), they are very fast

Recap: A High-level View of Computer



RV32I Registers



- 32 registers in RISC-V
 - Why 32? Smaller is faster, but too small is bad.
- Each RV32 register is 32-bit wide
 - Groups of 32 bits called a word in RV32; P&H textbook uses 64-bit variant RV64 (doubleword)

Registers



Similar to memory, use “address” to refer to specific location



Hold address of the current instruction

```

1c: a8 83 1f b8  stur w8, [x29, #-8]
20: 28 1c 82 52  mov w8, #4321
24: a8 43 1f b8  stur w8, [x29, #-12]
28: a8 83 5f b8  ldur w8, [x29, #-8]
2c: a9 43 5f b8  ldur w9, [x29, #-12]
30: 08 01 09 0b  add w8, w8, w9
  
```



C, Java variables vs. registers

- In C (and most high level languages) variables declared first and given a type
 - Example: `int fahr, celsius;`
 `char a, b, c, d, e;`
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match int and char variables).
- In Assembly Language, registers have no type, **simply stores 0s and 1s**; operation determines how register contents are treated (think about the hardware)



Assembly Instructions

- In assembly language, each statement (called an instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other high level languages), each line of assembly code contains at most 1 instruction
- Another way to make your code more readable: comments!
- Hash (#) is used for RISC-V comments
 - anything from hash mark to end of line is a comment and will be ignored

Assembly Instructions

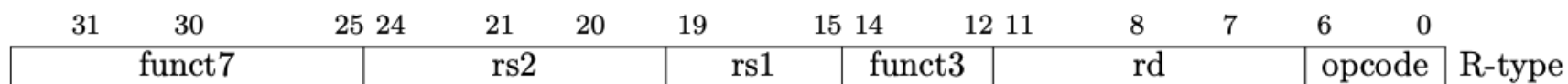
Different types of instructions (4 core types + B/J based on the handling of immediate)

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			

- Different types have different format but “rs1”, “rs2” and “rd” are at the same position (hardware friendly)
- As an ID number, the machine code of the instructions has different fields; format depends on their operands/type

Assembly Instructions

Different types of instructions (4 core types + B/J based on the handling of immediate)



- **R-type**
 - Register-register operation, mainly for arithmetic & logic
 - Has two operands (accessed from the source registers, rs1 & rs2) and one output (saved to the destination register, rd)
 - Cannot access main memory (instruction executed by CPU alone, no data exchange with main memory)

RV32I R-type Arithmetic

Syntax of instructions

- Addition: `add rd,rs1,rs2` (operation `rd,rs1,rs2`)
- Subtraction: `sub rd, rs1, rs2`

Adds the value stored in register `rs1` to that of `rs2` and stores the sum into register `rd`, similar to `a = b+c`, `a` $\Leftrightarrow$ `rd`, `b` $\Leftrightarrow$ `rs1`, `c` $\Leftrightarrow$ `rs2`

- Example: `add x5, x2, x1`
 `add x6, x0, x5`
 `sub x4, x1, x3`

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
	x4
	x5
	x6
	x7

RV32I R-type Logic Operation

Syntax of instructions:

- AND/OR/XOR: `and/or/xor rd, rs1, rs2`

Logically **bit-wise** and/or/xor the value stored in register rs1 and that of rs2 and stores the result into register rd, equivalent to $a = b \ (\&/\ ||/\wedge) \ c$, $a \Leftrightarrow \text{rd}$, $b \Leftrightarrow \text{rs1}$, $c \Leftrightarrow \text{rs2}$

- Example: `and x5, x2, x1`
`xor x6, x1, x5`

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
	x4
	x5
	x6
	x7

RV32I R-type Compare

Syntax of instructions

- SLT/SLTU: `slt/sltu rd, rs1, rs2`

Compare the value stored in register `rs1` and that of `rs2`, sets `rd=1`, if `rs1<rs2` otherwise `rd=0`, equivalent to `a = b < c ? 1 : 0`, `a` $\Leftrightarrow$ `rd`, `b` $\Leftrightarrow$ `rs1`, `c` $\Leftrightarrow$ `rs2`. Treat the numbers as signed/unsigned with `slt/sltu`.

- Example: `slt x5, x2, x1;` `slt x4, x3, x1;` `sltu x5, x3, x1`

- Overflow detection (unsigned)

```
add x5, x3, x3; sltu x6, x5, x3
```

- Overflow detection (signed)?

```
add t0, t1, t2
```

```
slti t3, t2, 0
```

```
slt t4, t0, t1
```

```
bne t3, t4, overflow
```

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
	x4
	x5
	x6
	x7

RV32I R-type Shift

Syntax of instructions:

- Shift left/right (arithmetic): `sll/srl/sra rd, rs1, rs2`

Left/Right shifts the value stored in register `rs1` by lower 5 bits of `rs2`, equivalent to $a = b \ll c$, $a \Leftarrow rd$, $b \Leftarrow rs1$, $c \Leftarrow rs2$.

arithmetic: sign extended.

- Example: `sll x5, x2, x4`

`srl x6, x1, x4`

`sra x7, x3, x4`

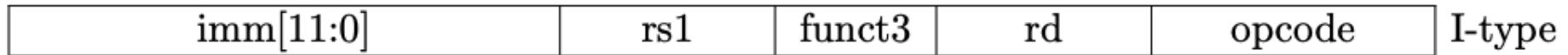
- What is the arithmetic effect by shifting?

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Assembly Instructions

Different types of instructions



- I-type
 - Register-Immediate type
 - Has two operands (one accessed from source register, another a constant/immediate, [sign-extended](#)) and one output (saved to destination register)
 - Can do arithmetic, logic and load from main memory

RV32I I-type Arithmetic

Syntax of instructions

- Addition: `addi rd, rs1, imm`

Adds `imm` to `rs1`, stores the result to `rd`, and `imm` is a signed number.

- Example: `addi x5, x4, 10`

`addi x6, x4, -10`

- Similarly, `andi/ori/xori/slti/sltui`
 - All the `imm`'s are sign-extended (details will be covered in later lectures)
- `slli/srli/srai` shift `rs1` by the lower 5-bits of `imm`, `srai` is distinguished by using one of the higher bit of the `imm` (or `funct7` field)

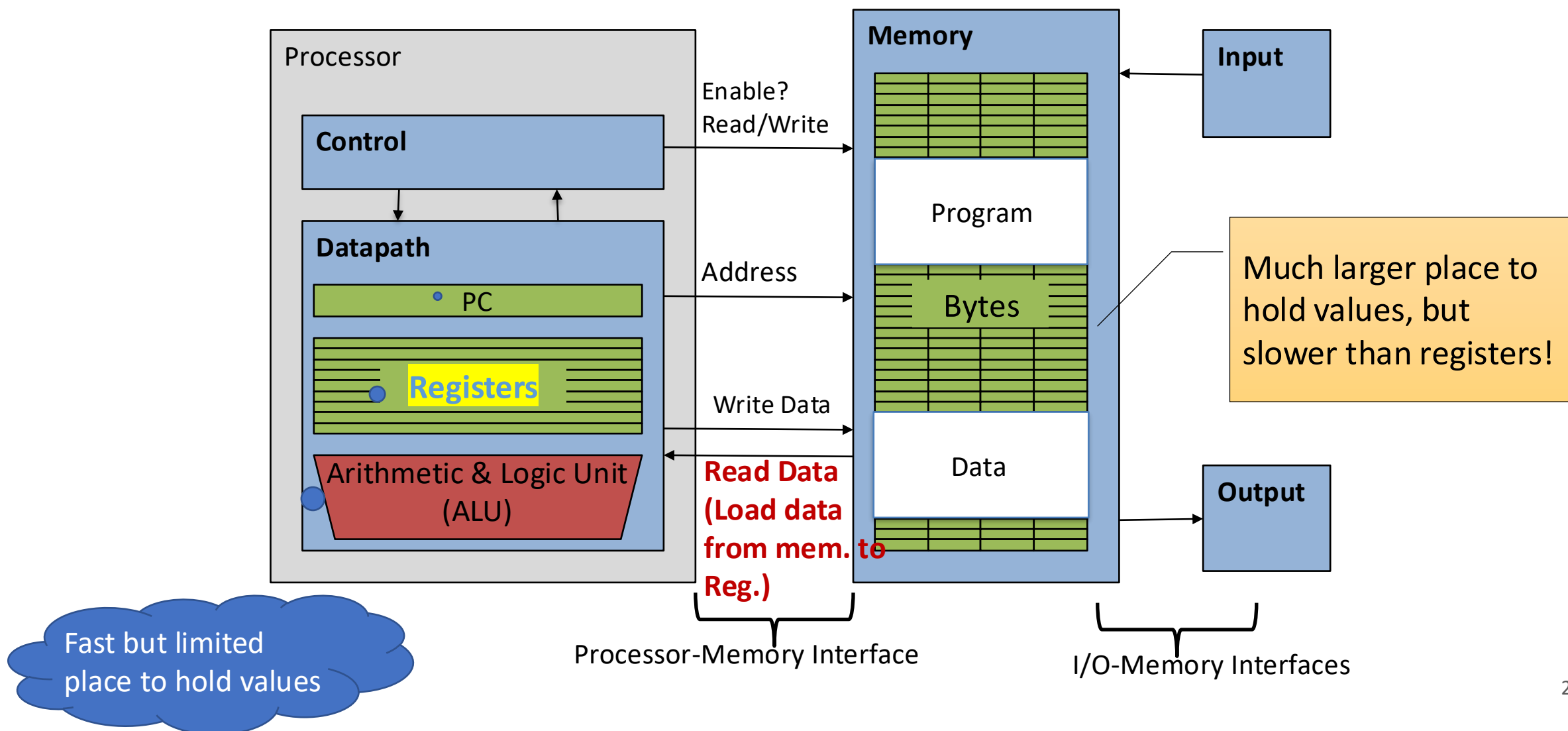
Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x3	x4
	x5
	x6
	x7



Load/store instructions

RV32I I-type Load





Assembly Instructions—Load

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:0]	rs1	funct3	rd	opcode	I-type
-----------	-----	--------	----	--------	--------

- `lw rd, imm(rs1)` : Load word at *address* to register *rd*, in which *address* = (number in *rs2*) + *imm*

- Example

`lw x1, 12(x4)`

address = 4 + 12 = 0x10

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Registers

Bytes				
56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	⋮
34	12	cd	ab	c
56	34	23	01	8
34	12	cd	ab	4
56	34	23	01	0

Main memory

Big Endian vs. Little Endian

- The order in which BYTES are stored in memory
- Bits always stored as usual. (E.g., 0xC2=0b 1100 0010)

Consider the number 1025 as we normally write it:

BYTE3 BYTE2 BYTE1 BYTE0
00000000 00000000 0000100 00000001

Big Endian

ADDR3 ADDR2 ADDR1 ADDR0
BYTE0 BYTE1 BYTE2 BYTE3
00000001 0000100 00000000 00000000

Examples

Names in the West (e.g. Chundong, Wang)

”Network Byte Order”: most network protocols

IBM z/Architecture; very old Macs

Little Endian

ADDR3 ADDR2 ADDR1 ADDR0
BYTE3 BYTE2 BYTE1 BYTE0
00000000 00000000 0000100 00000001

Examples

Names in China (e.g. WANG Chundong)

CANopen

Intel x86; RISC-V (can also support big-endian)

en.wikipedia.org/wiki/Big_endian

big-endian: MIPS, IA-64, PowerPC



Assembly Instructions—Load

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:0]	rs1	funct3	rd	opcode	I-type
-----------	-----	--------	----	--------	--------

- `lw rd, imm(rs1)` : Load word at *address* to register *rd*, in which *address* = (number in *rs2*) + *imm*

- Example

```
lw x1, 12(x4)
```

```
address = 4 + 12 = 0x10
```

- C code example

```
int A[100];
```

```
/*assume &A[0] = 4*/
```

```
G = A[3];
```

```
/*load G/A[3] to x1/
```

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Registers

Bytes				
56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	⋮
34	12	cd	ab	
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

Main memory



Assembly Instructions—Load

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:0]	rs1	funct3	rd	opcode	I-type
-----------	-----	--------	----	--------	--------

- lb/lbu rd, imm(rs1):** Load signed/unsigned byte at *address* to register rd

$address = (\text{number in rs1}) + imm$

- Example

lb x1, 12(x4)

$address = 4 + 12 = 0x10$

lbu x1, 12(x4)

- lh/lhu rd, imm(rs1):** Load signed/unsigned halfword at *address* to register rd

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Registers

Bytes				
56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	⋮
34	12	cd	ab	
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

Main memory



Assembly Instructions—Load

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:0]	rs1	funct3	rd	opcode	I-type
-----------	-----	--------	----	--------	--------

- `sw rs2, imm(rs1)`: Store word at rs2 to memory *address*

$address = (\text{number in rs1}) + imm$

- Example

`sw x1, 12(x4)`

$address = 4 + 12 = 0x10$

- C code example

```
int A[100];
```

```
/* &A[0] => x4 */
```

```
A[3] = h;
```

```
/* h in rs2 => A[3] */
```

- Similarly,

`sh`: Store lower 16 bits at rs2

`sb`: Store lower 8 bits at rs2

No shu/sbu?

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Registers

Bytes				
56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	⋮
34	12	cd	ab	c
56	34	23	01	8
34	12	cd	ab	4
56	34	23	01	0

Main memory



Memory Alignment

RISC-V **does not require** that integers be word aligned...

- But it can be very very bad if you don't make sure they are...

Consequences of unaligned integers

- Slowdown: The processor is allowed to be a lot slower when it happens
 - In fact, a RISC-V processor may natively only support aligned accesses, and do unaligned-access in software!
An unaligned load could take hundreds of times longer!
- Lack of atomicity: The whole thing doesn't happen at once... can introduce lots of very subtle bugs

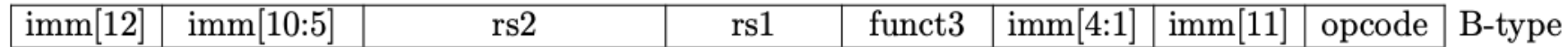
So in practice, RISC-V **recommends** integers to be aligned on 4- byte boundaries; halfword 2-byte boundaries



Decision-making instructions

Computer Decision Making—Branch

- Normal operation: execute instructions in sequence
- In C: if/while/for-statement; function call
- RISC-V provides conditional branch (B-type) & unconditional jump (J-type)



□ RISC-V: similar to if-statement instruction

`beq rs1, rs2, L(imm/label)`

meaning: go to statement labeled if (value in `rs1`) == (value in `rs2`); otherwise, go to next statement

- `beq` stands for branch if equal
- Similarly, `bne` for branch if not equal

Computer Decision Making—Branch

- Example: `beq rs1, rs2, L(imm/label)`

- C code

```
int main(void) {  
    int i=5;  
    if (i!=6){  
        i++;  
    }  
    else i--;  
    return 0;  
}
```

Compile



- Assembly

```
    ;      addi x2, x0, 5  
    ;      addi x3, x0, 6  
    ;      bne x2, x3, L1 #imm = 8  
    ;      beq x2, x3, L2  
L1: addi x2, x2, 1  
    ;      ret #kind of jump, psuedo-instruction  
L2: addi x2, x2, -1  
    ret
```

assembler

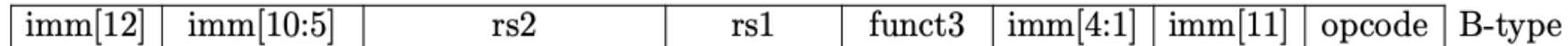
```
1c: a8 83 1f b8  
20: 28 1c 82 52  
24: a8 43 1f b8  
28: a8 83 5f b8  
2c: a9 43 5f b8  
30: 08 01 09 0b
```

- Label can also point to data (more in discussion)



Computer Decision Making—Branch

- Normal operation: execute instructions in sequence
- In programming languages: `if/while/for`-statement
- RISV-V provides conditional branch & unconditional jump



- RISC-V: if-statement instructions are

`blt/bltu/bge/bgeu rs1, rs2, L(imm/label)`

meaning: go to statement labeled L if (value in `rs1`) $</\geq$ (value in `rs2`) using signed/unsigned comparison; otherwise, go to the next statement



C Loop Mapped to RISC-V Assembly

```
int A[20];  
  
int sum = 0;  
  
for (int i=0; i < 20; i++)  
    sum += A[i];
```

Assume x8 holds pointer to A

Assign x10=sum

add x9, x8, x0 # x9=&A[0]

add x10, x0, x0 # sum=0

add x11, x0, x0 # i=0

addi x13, x0, 20 # x13=20

Loop:

bge x11, x13, Done

lw x12, 0(x9) # x12=A[i]

add x10, x10, x12 # sum+=

addi x9, x9, 4 # &A[i+1]

addi x11, x11, 1 # i++

j Loop

Done:

ret

Optimization

- The simple translation is sub-optimal!
 - Inner loop is now 4 instructions rather than 7
 - And only 1 branch/jump rather than two: Because first time through is always true so can move check to the end!
- The compiler will often do this automatically for optimization
 - See that i is only used as an index in a loop

Assume x8 holds pointer to A

Assign x10=sum

add x10, x0, x0 # sum=0

add x11, x8, x0 # ptr = A

addi x12, x11, 80 # end = A + 80

Loop:

lw x13, 0(x11) # x13 = *ptr

add x10, x10, x13 # sum += x13

addi x11, x11, 4 # ptr++

blt x11, x12, Loop # ptr < end

- This optimization is not required
- Line by line translation is good
- Correctness first, performance second

Arrays and Pointers

```
int i;  
int array[10];  
  
for (i = 0; i < 10; i++)  
{  
    array[i] = ...;  
}
```

```
int *p;  
int array[10];  
  
for (p = array; p < &array[10]; p++)  
{  
    *p = ...;  
}
```

These code sequences have the same effect!



Translate Assembly to C

```
addi x10, x0, 0x7
```

```
add x12, x0, x0
```

```
label_a:
```

```
andi x14, x10, 1
```

```
beq x14, x0, label_b
```

```
add x12, x10, x12
```

```
label_b:
```

```
addi x10, x10, -1
```

```
bne x10, x0, label_a
```

```
x10 = 7
```

```
x12 = 0
```

```
label_a: x14 = x10 & 1
```

```
if (x14!=0)
```

```
{x12 = x10+x12;}
```

```
label_b: x10 = x10-1;
```

```
if (x10!=0)
```

```
{go to label_a;}
```



Function call

Call a Function—Unconditional Jump

```

0000000100003f40 <_main>:
100003f40: ff c3 00 d1  sub sp, sp, #48
... ..
100003f58: 48 9a 80 52    mov w8, #1234
100003f5c: a8 83 1f b8  stur w8, [x29, #-8]
100003f60: 28 1c 82 52    mov w8, #4321
100003f64: a8 43 1f b8  stur w8, [x29, #-12]
100003f68: a8 83 5f b8  ldur w8, [x29, #-8]
100003f6c: a9 43 5f b8  ldur w9, [x29, #-12]
100003f70: 08 01 09 0b    add w8, w8, w9
... ..
100003f90: 05 00 00 94    bl  0x100003fa4 <_printf+0x100003fa4>
... ..
Disassembly of section __TEXT,__stubs:
0000000100003fa4 <__stubs>:
100003fa4: 10 00 00 b0    adrp
100003fa8: 10 02 40 f9  ldr
100003fac: 00 02 1f d6  br

```

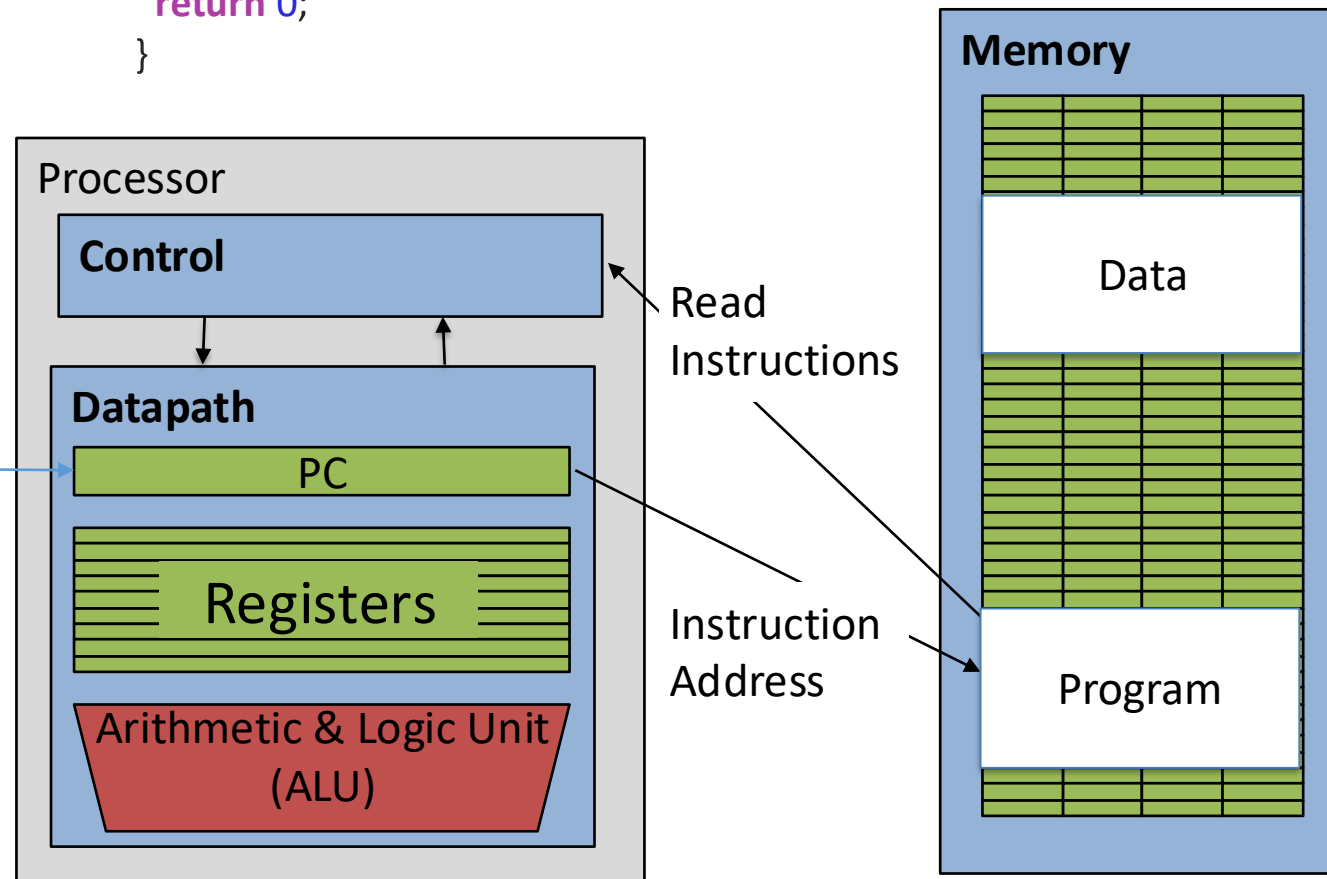
Increase by 4
each time an
instruction is
executed

Except for branch/jump/function call

```

#include <stdio.h>
int main() { //compute 1234 + 4321
    int x = 1234, y = 4321;
    int z = x+y;
    printf("z=%d/n",z);
    return 0;
}

```





Call a Function

```
#include <stdio.h>
int sum_two_number(int a, int b)
{
    int y;
    return y=a+b;
}
int main(int argc, const char * argv[])
{
    int x=4321, y=1234;
    int a=1,b=2,c=3,d=4,e=5,f=6,g=0;
    y = sum_two_number(x,y);
    c = sum_two_number(a,b);
    f = sum_two_number(e,d);
    g = sum_two_number(c,f);
    printf("Sum is %d.\n",y);
    return 0;
}
```

1. Put parameters in a place where function can access them
2. Transfer control to function (PC jump to sum_two_number)
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling code can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

RISC-V Function Call Conventions

- Registers faster than memory, so use them as much as possible
- Give names to registers and conventions on how to use them

REGISTER NAME, USE, CALLING CONVENTION

④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

Older version: <https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf>

Latest draft: <https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/tag/draft-20230220-87f4a72d5aeaf048b35a230e0ba5accd1bfcf072>

Part of application binary interface (ABI)

RISC-V Function Call Conventions

- a0–a7 (x10-x17): eight argument registers to pass parameters and return values (a0-a1)
- ra: one return address register to return to the point of origin (x1)
- Also s0-s1 (x8-x9) and s2-s11 (x18-x27): saved registers

REGISTER NAME, USE, CALLING CONVENTION

④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

Older version: <https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf>

Latest draft: <https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/tag/draft-20230220-87f4a72d5aeaf048b35a230e0ba5accd1bfcf072>

Part of application binary interface (ABI)

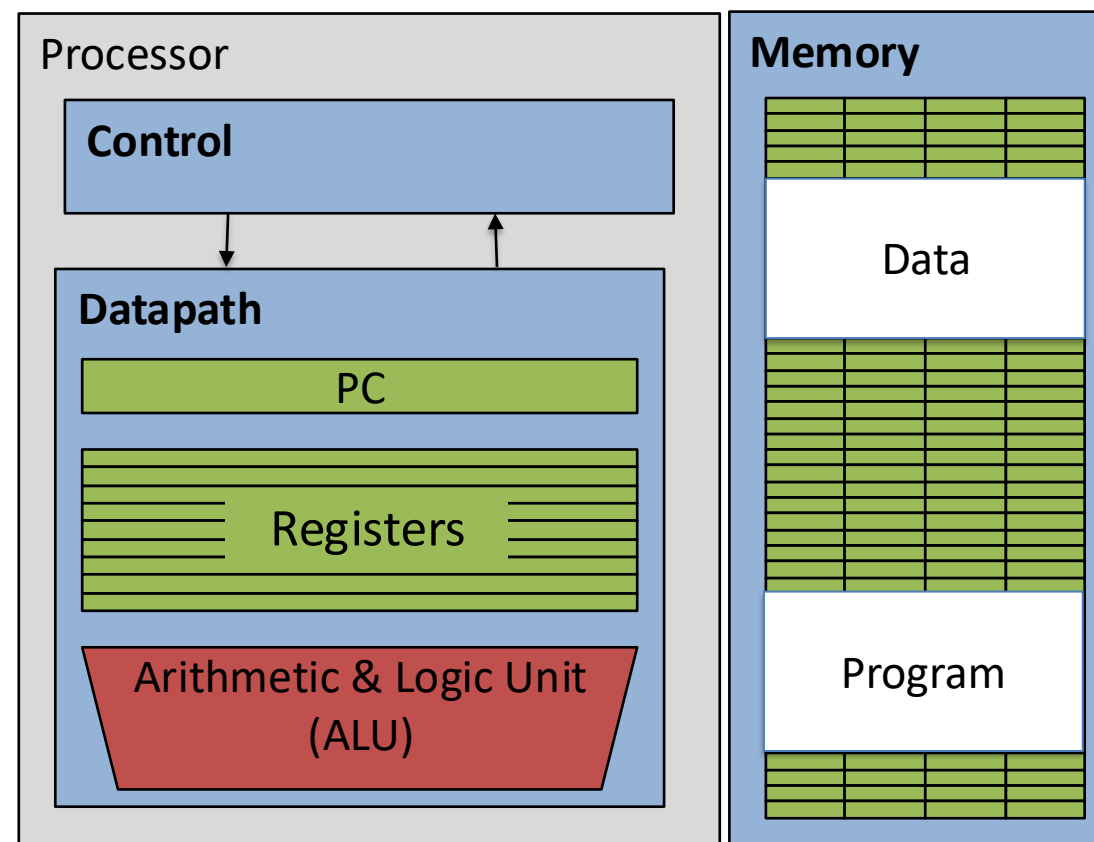
Call a Function

a0–a7 (x10-x17): eight argument registers to pass parameters and return values (a0-a1)

```
#include <stdio.h>
int sum_two_number(int a, int b)
{
    int y;
    return y=a+b;
}
int main(int argc, const char * argv[]) {
    int x=4321, y=1234;
    int a=1,b=2,c=3,d=4,e=5,f=6,g=0;
    y = sum_two_number(x,y);
    c = sum_two_number(a,b);
    f = sum_two_number(e,d);
    g = sum_two_number(c,f);
    printf("Sum is %d.\n",y);
    return 0;
}
```

x and y are function arguments;
Can be put in registers a0-a7

y is returned function argument;
Can be put in registers a0-a1



Call a Function

- ra: one return address register to return to the point of origin (x1)

```
#include <stdio.h>
int sum_two_number(int a, int b)
{
    int y;
    return y=a+b;
}
int main(int argc, const char * argv[]) {
    int x=4321, y=1234;
    int a=1,b=2,c=3,d=4,e=5,f=6,g=0;
    y = sum_two_number(x,y);
    c = sum_two_number(a,b);
    f = sum_two_number(e,d);
    g = sum_two_number(c,f);
    printf("Sum is %d.\n",y);
    return 0;
}
```

Func_called:

```
0x2000 //one instruction
0x2004 //another instruction
... .. //need jump back to main()
```

Start:

```
0x1000 //one instruction
0x1004 //another instruction
0x1008 //a third instruction
0x100c //PC jump to 0x2000 (call function
sum_two_number)
0x1010 //next instruction... ..
... ..
```

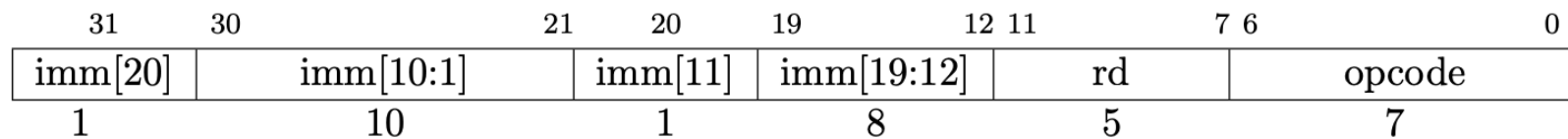


Save this value to
register ra

Call a Function—J-type

JAL: Jump & Link, jump to function

Unconditional jump (J-type)



`jal rd label`

Jump to label ($\text{imm} + \text{PC}$, explain later) and save return address ($\text{PC} + 4$) to `rd`;

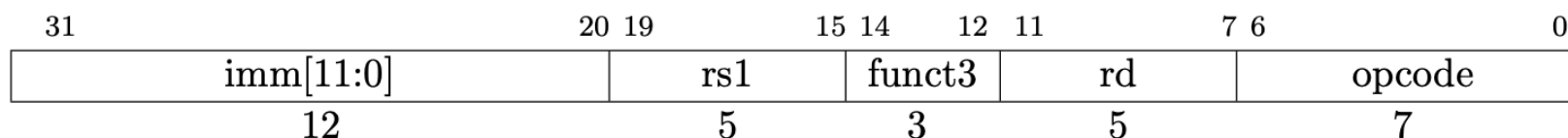
`rd` is `x1 (ra)` by convention; sometimes can be `x5`.

When `rd` is `x0`, it is simply unconditional jump (`j`) without recording $\text{PC} + 4$.

Return—JALR

JALR: Jump & Link Register

Unconditional jump (*l-type encoding*)



jalr rd rs1 label

Jump to label $(imm+rs1)\&\sim 1$ and save return address ($PC+4$) to *rd*

rs1 can be the return address we just saved to *ra*

When *rd* is *x0*, it is simply unconditional jump (*j*) without recording $PC+4$.



Unconditional Jump

—jal rd offset —jalr rd rs offset

- Jump and Link (jal)

- Add the immediate value to the current address in the program (the “program counter”), go to that location
 - The offset is 20 bits, sign extended and left-shifted one (not two)
- At the same time, store into rd the value of PC+4
 - So we know where it came from (need to return to)
- `jal offset == jal x1 offset` (pseudo-instruction; x1 = ra = return address)
- `j offset == jal x0 offset` (jump is a pseudo-instruction in RISC-V)

- Two uses:

- Unconditional jumps in loops and the like
- Calling other functions



Jump and Link Register

The same except the destination

- Instead of PC + immediate it is [value in rs] + immediate
 - Same immediate format as I-type: 12 bits, sign extended

Again, if you don't want to record where you jump to...

- `jr rs == jalr x0 rs`

Two main uses

- Returning from functions (which were called using Jump and Link)
- Calling pointers to function

Call a Function

```
#include <stdio.h>
int sum_two_number(int a, int b)
{
    int y;
    return y=a+b;
}
int main(int argc, const char * argv[])
{
    int x=4321, y=1234;
    int a=1,b=2,c=3,d=4,e=5,f=6,g=0;
    y = sum_two_number(x,y);
    c = sum_two_number(a,b);
    f = sum_two_number(e,d);
    g = sum_two_number(c,f);
    printf("Sum is %d.\n",y);
    return 0;
}

int (*p)(void);
```

1. Put parameters in a place where function can access them ([addi/add](#) etc. to copy)
2. Transfer control to function (PC jump to [sum_two_number](#), [jal/jalr](#))
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function ([we have learned this](#))
5. Put result value in a place where calling code can access it and restore any registers you used ([addi/add](#) etc. to copy)
6. Return control to point of origin, since a function can be called from several points in a program ([jalr](#))

REGISTER NAME, USE, CALLING CONVENTION

④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

Notes on Functions

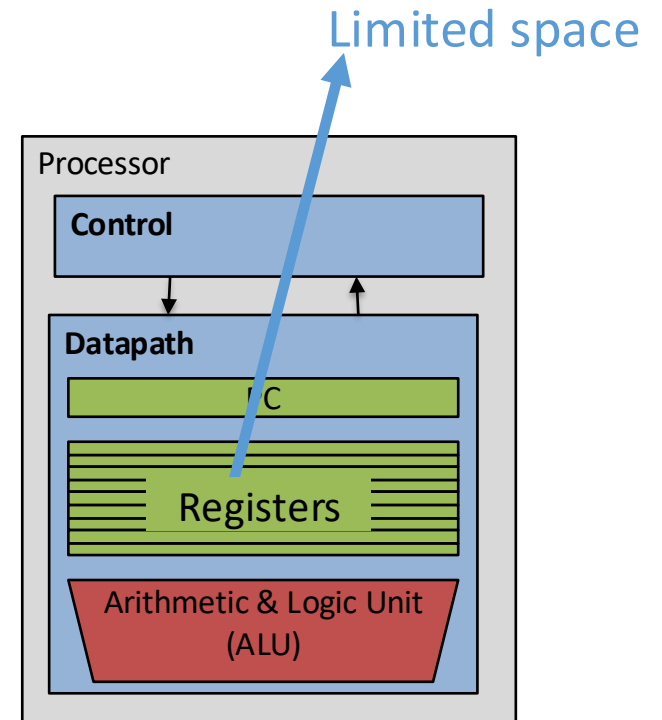
- Calling program (**caller**) puts parameters into registers **a0-a7** and uses **jal X** to invoke (**callee**) at address labeled **X**
- PC provides the returning point (PC+4 stored in ra)
- What value does **jal X** place into **ra**?
- **jr ra** puts address inside **ra** back into PC
- **New problem, small # of GPRs**

```
#include <stdio.h>
int sum_two_number(int a, int b)
{
    int y;
    return y=a+b;
}
int main(int argc, const char * argv[])
{
    int a=1,b=2,c=3,d=4,e=5,f=6,g=0;
    int h,i,j,k,l,m,n,cs110,cs110p,...;
    y = sum_two_number(x,y);
    ...
}
```

3. Acquire (local) storage resources needed for function

Where are Old Register Values Saved to Restore Them after Function Call?

- Need a place to save old values before call function, restore them when return, and delete; use the **big main memory**
- Ideal is stack: last-in-first-out queue (e.g., stack of plates)
 - Push: placing data onto stack
 - Pop: removing data from stack
- Stack in memory, so need register to point to it
- **sp** is the stack pointer in RISC-V (**x2**)
- Convention is grow from high to low addresses
 - Push decrements **sp**, Pop increments **sp**

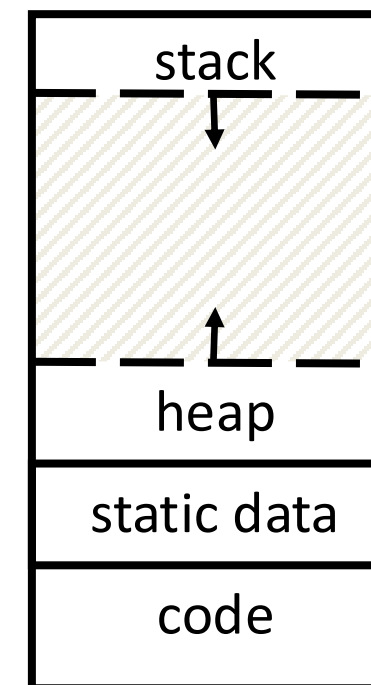


Stack

- Stack frame may include:
 - Return “instruction” address
 - Parameters (spill)
 - Space for other local variables
- Stack frames contiguous; stack pointer (**sp**/x2) tells where bottom of stack frame is
- When procedure/function begins, **sp** is decreased to create space; data are stored in the stack frame (**push**)
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames; **sp** restores (**pop**)

FFFF FFFF_{hex}

Memory Address
(32 bits assumed here)



~ 0000 0000_{hex}

Calling Convention

- ❑ Could have saved all variables to main memory
- ❑ Calling convention to make things easier
- ❑ Callee-saved registers: callee clean the mess
 - Callee needs to save old value of s^* (and any other callee saved registers), and makes sure they are not changed after return, and also sp

REGISTER NAME, USE, CALLING CONVENTION

④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

Example

- Leaf function: a function that calls no function

```
int Leaf (int g, int h, int i, int j)
{
    int f; f = (g + h) - (i + j);
    return f;
}

int main (void){
    int a=1, b=2, c=3, d=4, e, x;
    e = Leaf(a,b,d,c); //call with jal
    // return address in ra(x1)
    ... ..
    return e;
} /*a function called by OS*/
```

0	x0/zero
ra	x1
sp	x2
... ..	
s1	x9
a0	x10
a1	x11
a2	x12
a3	x13
a4	x14
... ..	

Parameter variables g,h,i and j in argument registers **a1**,**a2**, **a3**, and **a4**, and f in **a0** when returned, and assume **x** in **s1**

Assume function **Leaf** use **s1** for intermediate results

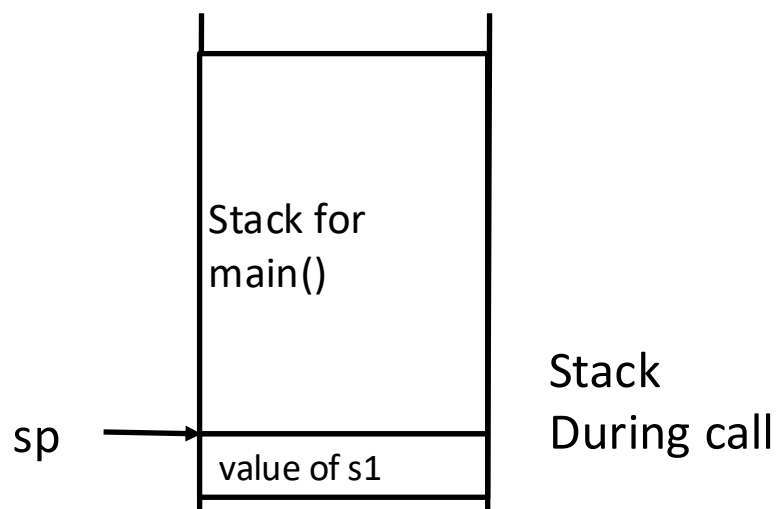
Register **ra** consideration



RISC-V Code for Leaf()

Leaf:

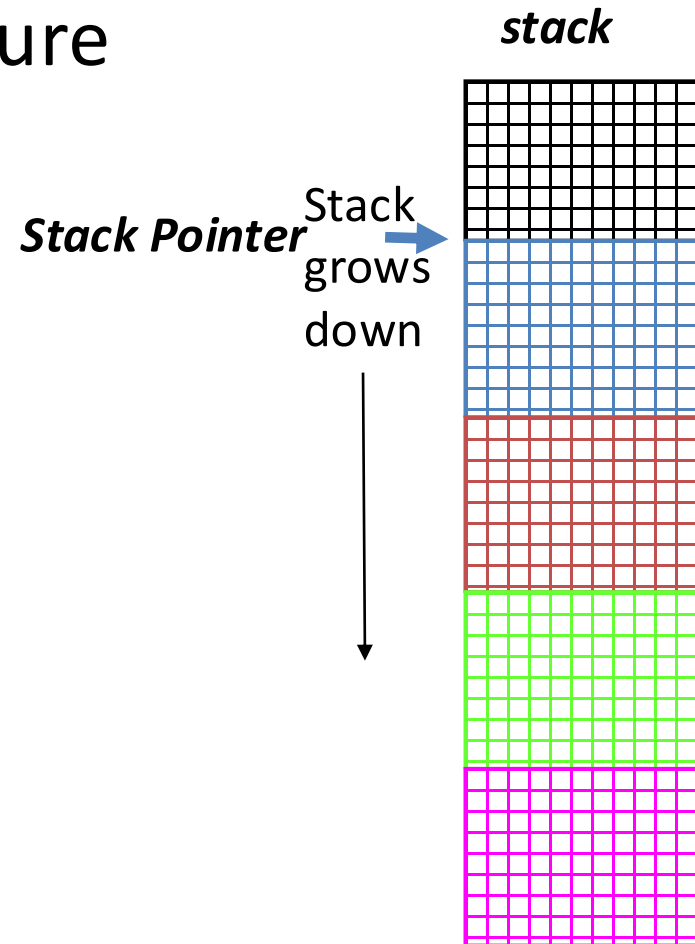
```
addi sp, sp, -4 # adjust stack for 1 items, callee saved s1
sw  s1, 0(sp) # save callee saved s1 to stack
add  s1, a0, a1 # s1 = g + h
add  a2, a2, a3 # j = i + j
sub  a0, s1, a2 # calculate result (g + h) - (i + j)
      # return value (g + h) - (i + j)
lw   s1, 0(sp) # restore register s1 for caller
addi sp, sp, 4 # adjust stack to delete 1 items
jr   ra # jump back to caller (pseudo-assembly: ret)
```



Stack Animation

- Last In, First Out (LIFO) data structure

```
main ()  
{ a(0);  
}  
  void a (int m)  
  { b(1);  
  }  
    void b (int n)  
    { c(2);  
    }  
      void c (int o)  
      { d(3);  
      }  
        void d (int p)  
        {  
        }  
        }
```



Optimization for Leaf()

Caller-saved registers: caller clean the mess

- Caller needs to save old value of caller-saved registers, and makes sure they are not changed for further use

REGISTER NAME, USE, CALLING CONVENTION

④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

Optimization for Leaf()

Caller-saved registers: caller clean the mess

- Caller needs to save old value of caller-saved registers, and makes sure they are not changed for further use

Leaf:

```
addi sp, sp, -4 # adjust stack for 1 items, callee saved s1  
sw s1, 0(sp) # save callee saved s1 to stack  
add s1, a0, a1 # s1 = g + h  
add a t1 2, a3 # j = i + j  
sub a0, s1, a2 # calculate result (g + h) - (i + j)  
          # re t1 value (g + h) - (i + j)  
lw s1, 0(sp) # restore register s1 for caller  
addi sp, sp, 4 # adjust stack to delete 1 items  
jr ra # jump back to caller (pseudo-assembly: ret)
```

Nested Call

```
int bar(int g,int h,int i,int j)
{
    int f = (g + h) - (i + j);
    return f;
}
int foo(int x)
{
    //do stuff
    int x = bar(g, h, i, j);
    //g, h, i, j for other use
    return (x/2);
}
int main()
{
    // do stuff 1
    foo(x); // call with jal
    // do stuff 2
}
```

In `foo`, `g`, `h`, `i`, and `j`
are in `s0-s3`

```
foo:
# do stuff (code omitted)
# save ra
addi sp, sp, -4           (Prologue)
sw ra, 0(sp)
# set up argument registers
add a0, s0, x0
add a1, s1, x0
add a2, s2, x0
add a3, s3, x0
jal bar
# restore ra
srai a0, a0, 1
lw ra, 0(sp)              (Epilogue)
addi sp, sp, 4
jr ra
```

Nested Call

```
int bar(int g,int h,int i,int j)
{
    int f = (g + h) - (i + j);
    return f;
}
int foo(int x)
{
    //do stuff
    int x = bar(g, h, i, j);
    //g, h, i, j for other use
    return (x/2);
}
int main()
{
    // do stuff 1
    foo(x); // call with jal
    // do stuff 2
}
```

In foo, g, h, i, and j
are in t0-t3

```
foo:
# do stuff (code omitted)
# save ra
addi sp, sp, -4           (Prologue)
sw ra, 0(sp)
# set up argument registers
add a0, s0, x0
add a1, s1, x0
add a2, s2, x0
add a3, s3, x0
jal bar
# restore ra
srai a0, a0, 1
lw ra, 0(sp)              (Epilogue)
addi sp, sp, 4
jr ra
```

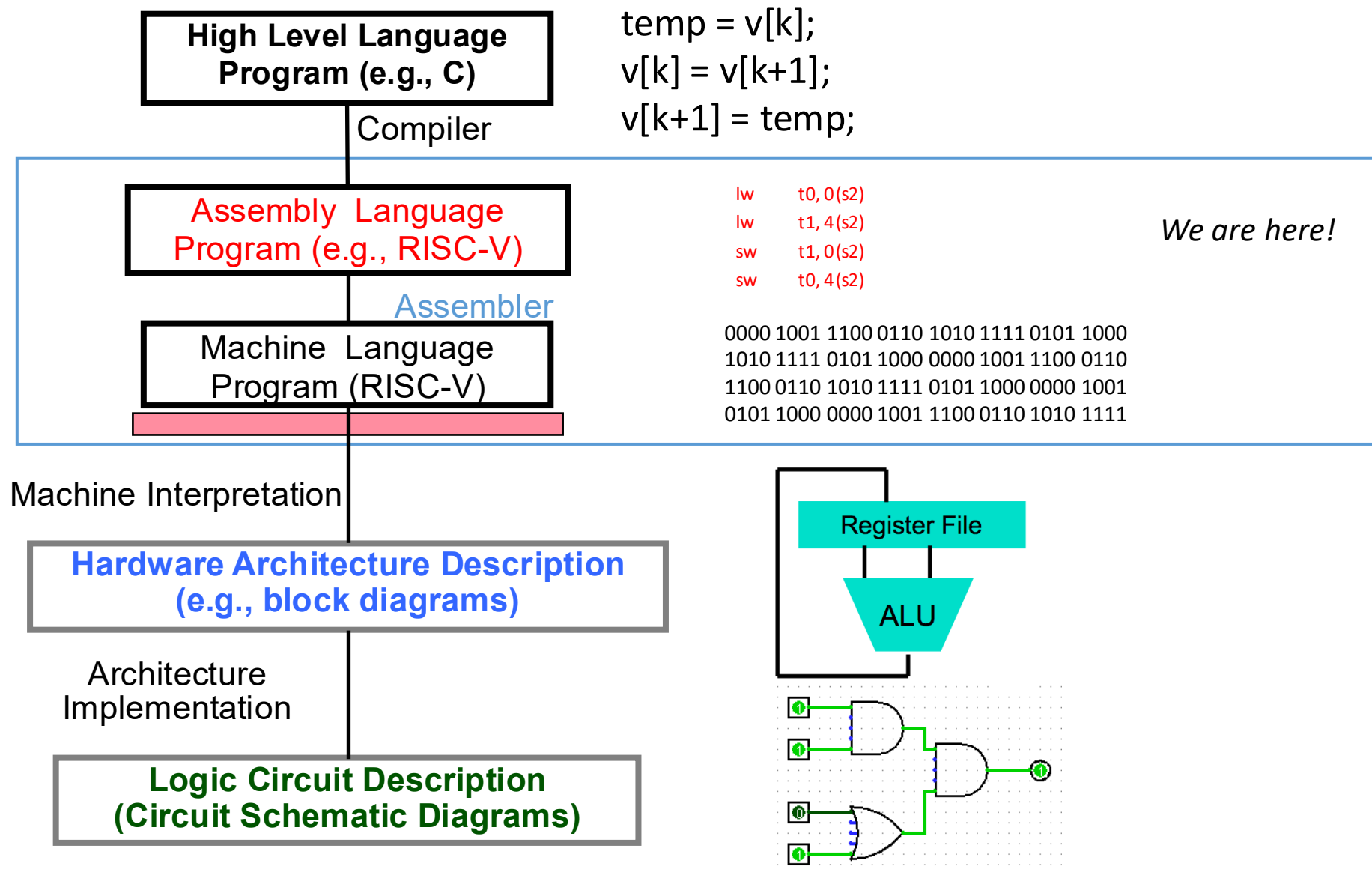
Call a Function

1. Caller put parameters in a place where function can access them (a0-a7, or stack when registers not avail.), and then save caller-saved registers to stack (e.g. ra, a0-a7,t0-t11) **if necessary**
2. Transfer control to callee function (PC jump to function): jal/jalr, ra is changed to where caller left
3. Acquire (local) storage resources needed for function: change sp (size decided when compiling); Callee push callee-saved registers to stack (e.g., s0-s11) **if necessary**
4. Perform desired task of the function
5. Put result value in a place where calling code can access it (a0, a1), and restore/pop callee-saved registers (s0-s11, sp)
6. Return control to point of origin, since a function can be called from several points in a program (jr ra); caller restores caller-saved registers



Instruction Encoding and Decoding

Recap: interpretation



RV32I Instruction Encoding

~ High Bit

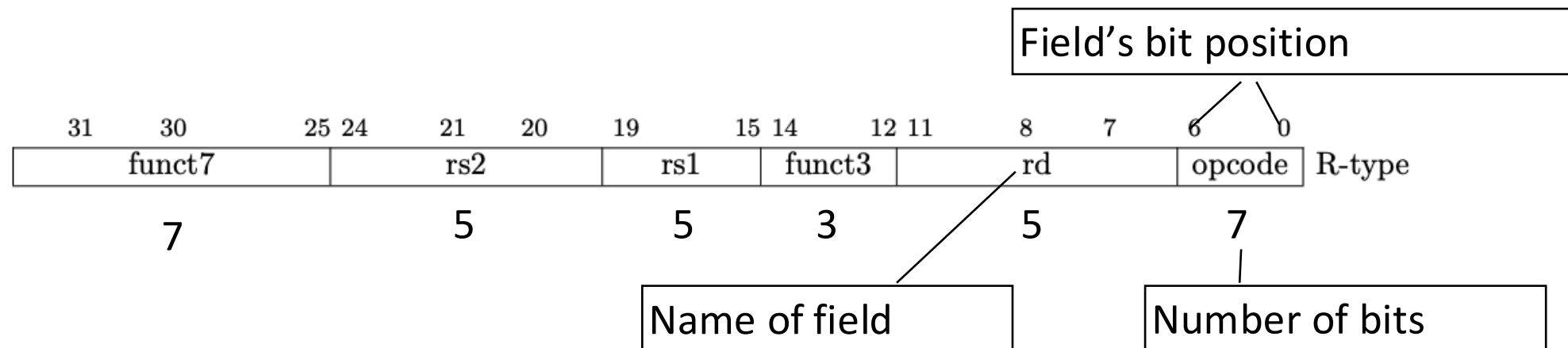
~ Low Bit

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		

- All 32-bit in length (not the case in RVC)
- Generally, fields are aligned if present (rs1, rs2, rd, funct3, funct7, opcode)
- Different number/type of operands/result

R-Format

Assembly: **Operation rd, rs1, rs2** (rd,rs1,rs2 unsigned numbers, represent No. of regs)

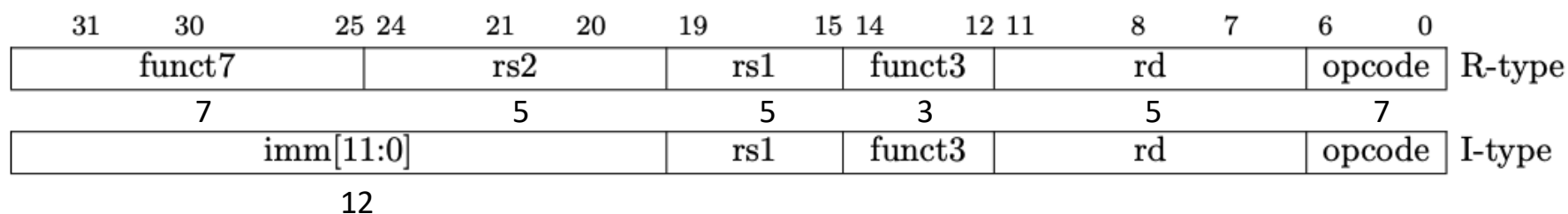


funct7/funct3/opcode fields:

- opcode: 0b0110011 for RV32I R-format arithmetic/logic operations
- funct7/funct3 together decide the type of operation

I-Format Arithmetic & Logic

Assembly: **Operation rd, rs1, rs2**



Register-immediate type

- imm: 12 bits, hold values for [-2048,2047]
- imm sign-extended before operations (sign-extension done in hardware)
- Opcode 0b0010011 for I-type arithmetic/logic operations

I-Format

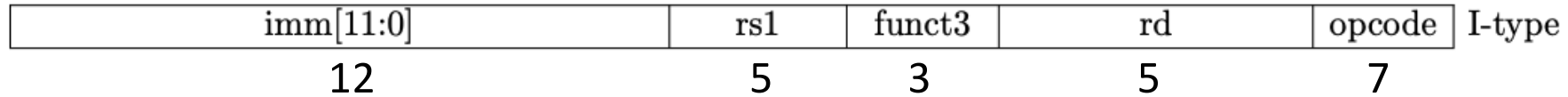
imm[11:0]	rs1	funct3	rd	opcode	I-type
12	5	3	5	7	

0000000	shamt[4:0]	src	001	dest	0010011	SLLI
0000000	shamt[4:0]	src	101	dest	0010011	SRLI
0100000	shamt[4:0]	src	101	dest	0010011	SRAI

Register-immediate type

- imm: 12 bits, hold values for [-2048,2047] **Not for shift operations**
- shamt not sign-extended before operations for shifts
- Opcode 0b0010011 for I-type arithmetic/logic operations
- Shift is specialized, since shift more than 31 bits is meaningless

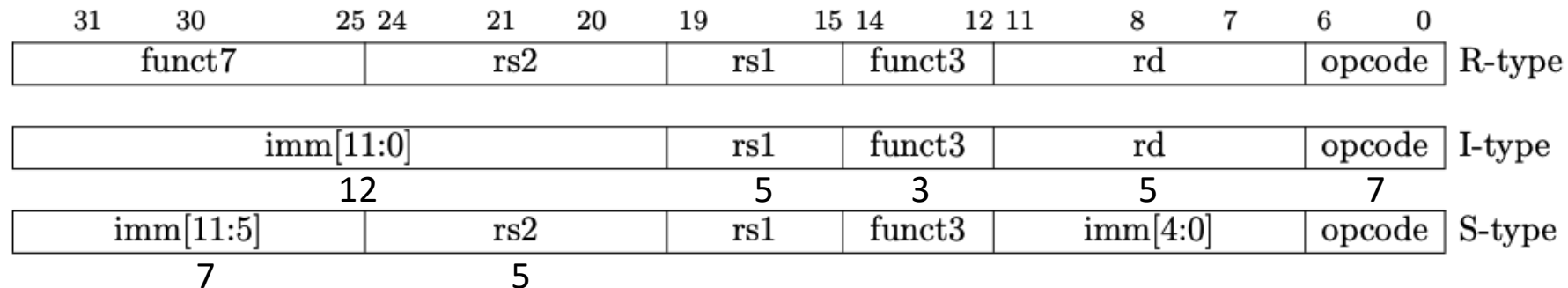
I-Format Load



Assembly: lw/lhu/lh/lb/lbu rd, (imm)rs1

- Opcode: 0b0000011 for RV32I I-format load operations
- funct3:
 - First bit indicates signed(0)/unsigned(1)
 - Last 2 bits indicates w(10)/h(01)/b(00)

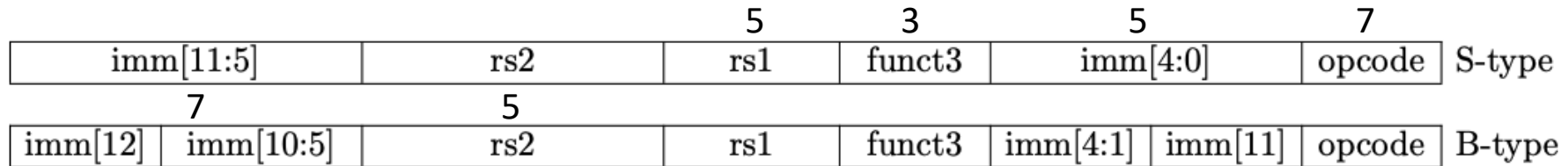
S-Format Store



Assembly: sw/sh/sb rs2,imm(rs1)

- Opcode: 0b0100011 for RV32I S-format store operations
- funct3:
 - Last 2 bits indicates w(10)/h(01)/b(00)
 - First bit 0

B-Format Conditional Branch



Assembly: bne/beq/blt/bltu/beg/begu rs1,rs2,label

- Opcode: 0b1100011 for RV32I B-format branch operations
- How to represent label?



Branching Instruction Usage

- Branches typically used for loops (if-else, while, for)
 - Loops are generally small (< 50 instructions)
- Recall: Instructions stored in a localized area of memory (Code/Text)
 - Largest branch distance limited by size of code
 - Address of current instruction stored in the program counter (PC)

C Loop Mapped to RISC-V Assembly

```
int A[20];  
int sum = 0;  
for (int i=0; i < 20; i++)  
    sum += A[i];
```

Assume x8 holds pointer to A

Assign x10=sum

add x9, x8, x0 # x9=&A[0]

add x10, x0, x0 # sum=0

add x11, x0, x0 # i=0

addi x13, x0, 20 # x13=20

Loop:

bge x11, x13, Done

lw x12, 0(x9) # x12=A[i]

add x10, x10, x12 # sum+=

addi x9, x9, 4 # &A[i+1]

addi x11, x11, 1 # i++

j Loop

Done:

PC-Relative Addressing

- PC-relative addressing: use the immediate field as a two's-complement offset to PC

- Branches generally change the PC by a small amount
- Can specify $\pm 2^{11}$ 'unit' addresses from the PC

- Recall

- Each instruction is 4-byte wide (4-byte aligned)
- Address is multiple of 4, least significant 2 bits "00"
- Can use bits [13:2] for imm
- Can specify $\pm 2^{13}$ 'unit' addresses from the PC
- **But**, to support RVC (16-bit/2-byte instruction) extension, [12:1] for imm/offset, can specify $\pm 2^{12}$ 'unit' addresses from the PC

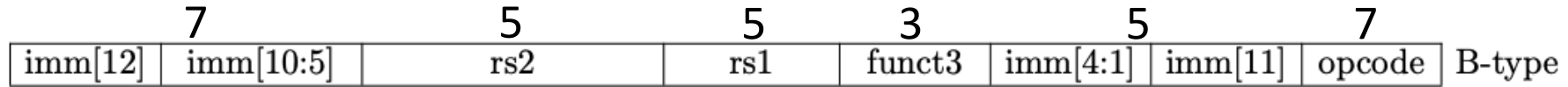
Disassembly of section .

0000000000000000 <ltmp0:

0:	ff	c3	00	d1
4:	fd	7b	02	a9
8:	fd	83	00	91
c:	08	00	80	52
10:	e8	0f	00	b9
14:	bf	c3	1f	b8
18:	48	9a	80	52
1c:	a8	83	1f	b8
20:	22	41	00	52

- Opposite to it, absolute addressing (use full address)

B-Format Conditional Branch



Assembly: bne/beq/blt/bltu/beg/begu rs1,rs2,label

- Opcode: 0b1100011 for RV32I B-format branch operations
- Label: PC-relative addressing, concatenate imm[12], imm[11], then imm[10:5] and imm[4:1] as offset (sign-extended)

Further on Conditional Branch

- To support RVC (16-bit/2-byte instruction) extension, [12:1] for imm/offset, can specify $\pm 2^{12}$ 'unit' addresses from the PC
- Equivalent to $\pm 2^{10}$ 32-bit instructions
- What if jump to farther away?

beq x10, x0, far

next instruction

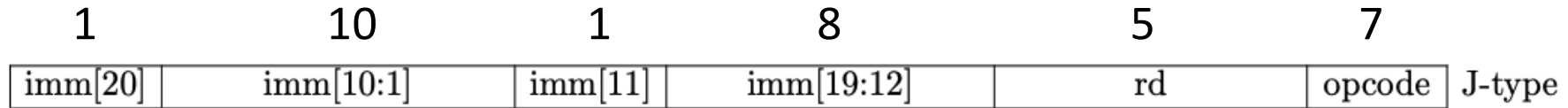
bne x10, x0, next

j far

next: # next instruction

j gets 20-bit imm

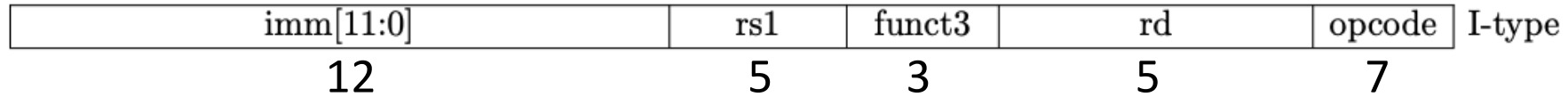
J-Format Jump Instruction



Assembly: jal rd, label

- Recall jal does 2 things:
 - Store PC+4 to rd as return address
 - Jump to label = $PC + \text{offset}(\text{imm})$
- Label translated by assembler to a 20-bit offset (encoded similar to Branch offset)
- Can access $\pm 2^{20}$ ‘unit’ addresses from the PC
- $\pm 2^{18}$ 32-bit instructions

I-Format Jump Instruction



Assembly: jalr rd,rs1,imm

- Recall jal does 2 things:
 - Store PC+4 to rd as return address
 - Jump to label = $rs + \text{offset}(\text{imm})$
- imm can hold values between [-2048,2047]
- Unlike JAL, include the last 0 using I-format

LUI and AUIPC

- Call function with 32-bit absolute address

lui x5, <hi20bits>

jalr ra, x5, <lo12bits>

- Jump PC-relative with 32-bit offset

auipc x5, <hi20bits>

jalr ra, x5, <lo12bits>

- Obtain PC value

auipc x5, 0

- Store/load with PC-relative 32-bit offset/32-bit absolute address

auipc x5, <hi20bits>/lui x5, <hi20bits>

sw/lw rd, (<lo12bits>)x5



Instruction Decoding

- Given an instruction in binary, how to interpret
- Reverse the procedure translating an instruction to machine code
 - Look up opcode/funct3/funct7 to identify type & operation
 - Find out rs1/rs2/rd/imm value, whichever presents
 - More in hardware design