System I

RISCV ISA

Haifeng Liu

Zhejiang University

Overview

- RISC-V ISA
- RISC-V Assembly Language

What is RISC-V?

- RISC-V (pronounced "risk-five") is an ISA standard
  - An open-source implementation of a reduced instruction set computing (RISC) based instruction set architecture (ISA)
  - There was RISC-I, II, III, IV before
- Most ISAs: X86, ARM, Power, MIPS, SPARC
  - Commercially protected by patents
  - Preventing practical efforts to reproduce the computer systems.
- RISC-V is open
  - Permitting any person or group to construct compatible computers
  - Use associated software
- Originated in 2010 by researchers at UC Berkeley
  - Krste Asanović, David Patterson and students
- ISA Specifications
  - Unprivileged specification version 20191213 (v2.2)
  - Privileged specification version 20211203 (v1.11)
  - More on github: https://github.com/riscv/riscv-isa-manual

https://riscv.org/

Goals in Defining RISC-V

- A completely open ISA that is freely available to academia and industry
- A real ISA suitable for direct native hardware implementation, not just simulation nor binary translation
- An ISA that avoids "over-architecting" for
  - A particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of-order) or
  - Implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these
- RISC-V ISA includes
  - A small base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and
  - Optional standard extensions, to support general-purpose software development
  - Optional customer extensions
- Support for the revised 2008 IEEE-754 floating-point standard

RISC-V Principles

- Generally kept very simple and extendable
  - Whether short, long, or variable
- Separated into multiple specifications
  - User-level ISA spec (compute instructions)
  - Compressed ISA spec (16-bit instructions)
  - Privileged ISA spec (supervisor-mode instructions)
  - More…
- ISA support is given by RV + word-width + extensions supported
  - E.g., RV32I means 32-bit RISC-V with support for the I (integer) instruction set

User-Level ISA

- Defines the normal instructions needed for computation
  - A mandatory **base integer ISA**
    - **I: Integer instructions**:
      - ALU
      - Branches/jumps
      - Loads/stores
  - **Standard extensions**
    - M: Integer Multiplication and Division
    - A: Atomic Instructions
    - F: Single-Precision Floating-Point
    - D: Double-Precision Floating-Point
    - C: Compressed Instructions (16 bit)
    - **G = IMAFD: integer base + four standard extensions**
  - Optional extensions

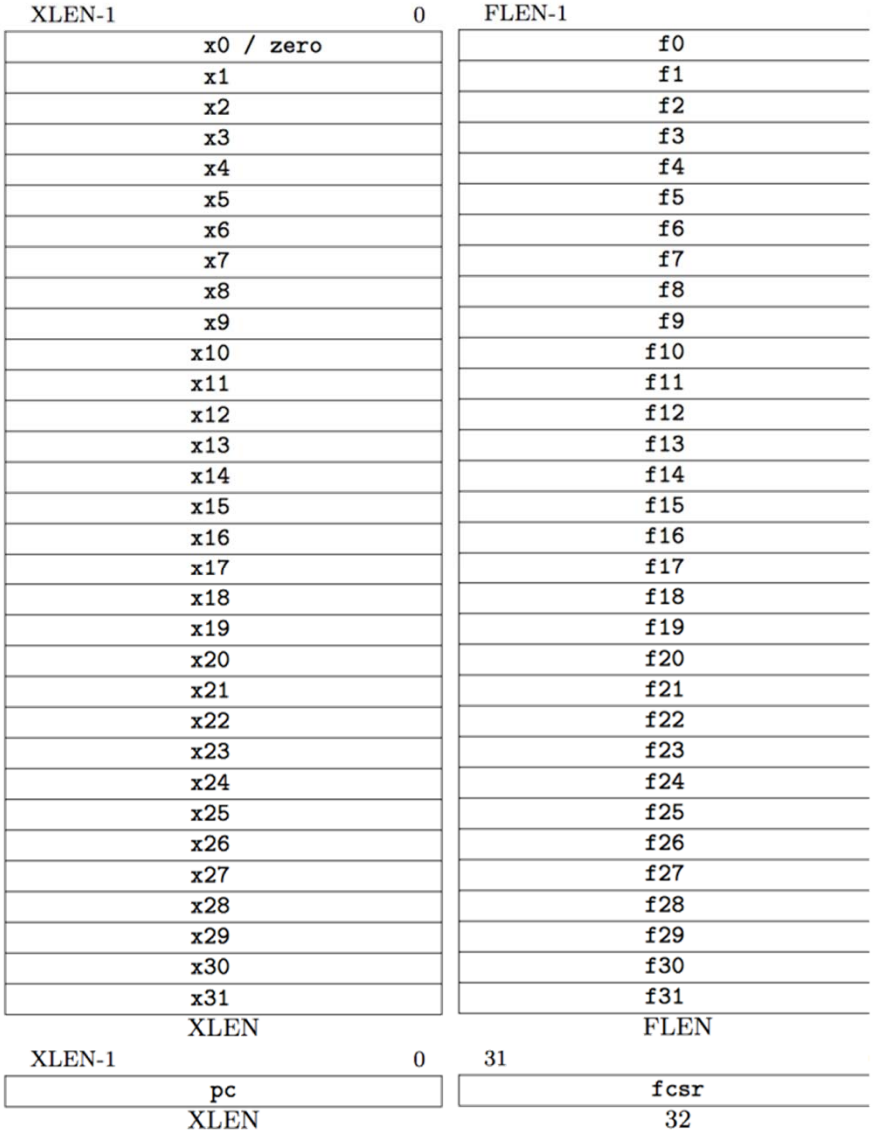
Basic RISC-V ISA

- Both 32-bit and 64-bit address space variants
  - RV32 and RV64
- Easy to subset/extend for education/research
  - RV32IM, RV32IMA, RV32IMAFD, RV32G
- SPEC on the website
  - www.riscv.org

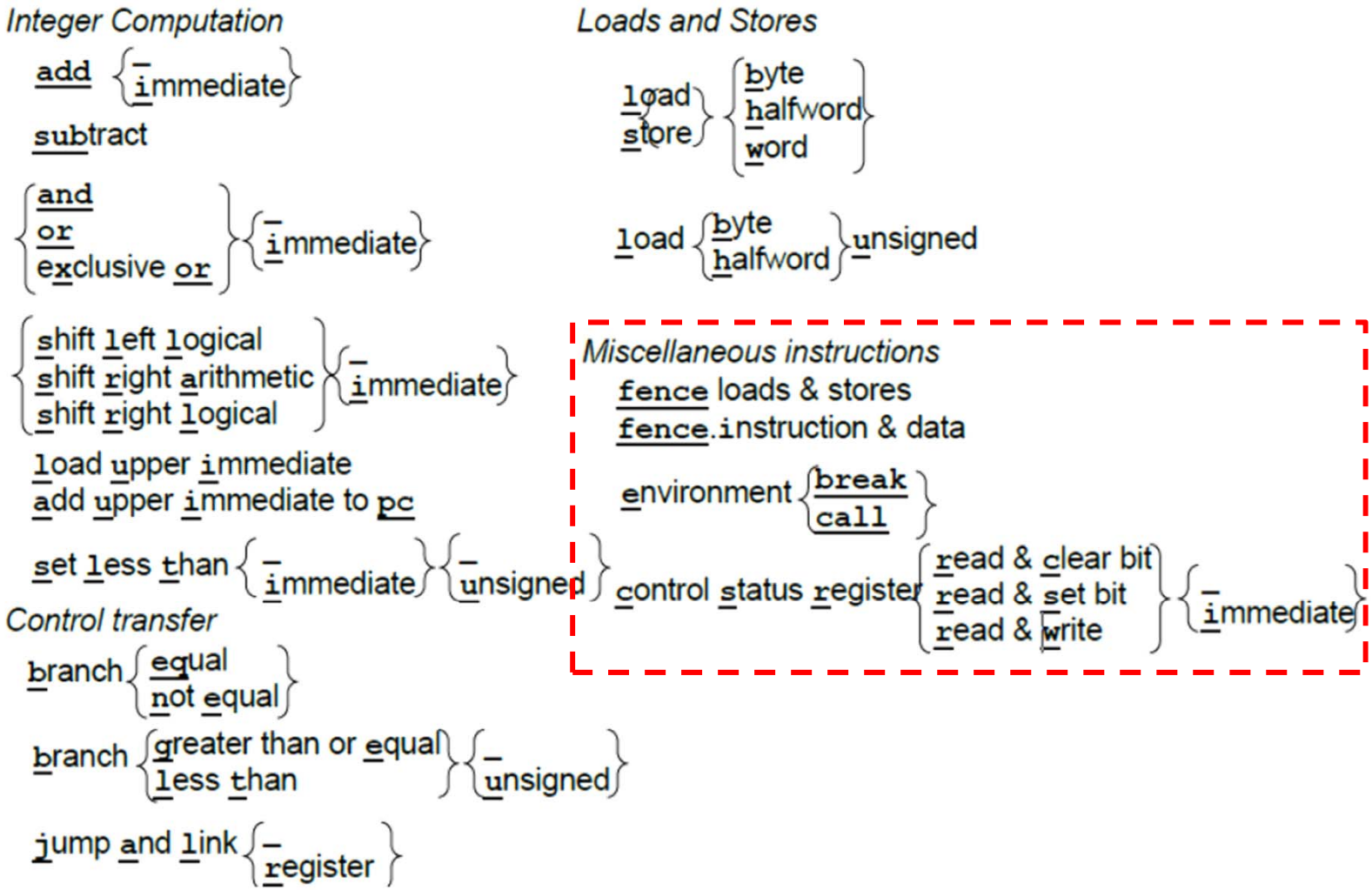
Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space

RISC-V Processor State

- Program counter (**PC**)
- 32 32/64-bit integer registers (**x0-x31**)
  - x0 always contains a 0
  - x1 to hold the return address on a call.
- 32 floating-point (FP) registers (**f0-f31**)
  - Each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)
- FP status register (**fsr**), used for FP rounding mode & exception reporting



RV32I


Integer Computation
add { _ immediate }
subtract
{ and / or / exclusive or } { _ immediate }
{ shift left logical / shift right arithmetic / shift right logical } { _ immediate }
load upper immediate
add upper immediate to pc
set less than { _ immediate } { _ unsigned }
Control transfer
branch { equal / not equal }
branch { greater than or equal / less than } { _ unsigned }
jump and link { _ register }
Loads and Stores
{ load / store } { byte / halfword / word }
load { byte / halfword } unsigned
Miscellaneous instructions
fence loads & stores
fence.instruction & data
environment { break / call }
control status register { read & clear bit / read & set bit / read & write } { _ immediate }

ALU Instructions

Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	`Regs[x1]←Regs[x2]+Regs[x3]`
`addi x1,x2,3`	Add immediate unsigned	`Regs[x1]←Regs[x2]+3`
`lui x1,42`	Load upper immediate	$\texttt{Regs[x1]} \leftarrow 0^{32} \#\# 42 \#\# 0^{12}$
`sll x1,x2,5`	Shift left logical	`Regs[x1]←Regs[x2]<<5`
`slt x1,x2,x3`	Set less than	`if (Regs[x2]<Regs[x3]) Regs[x1]←1 else Regs[x1]←0`

Figure A.26 The basic ALU instructions in RISC-V are available both with register-register operands and with one immediate operand. LUI uses the U-format that employs the rs1 field as part of the immediate, yielding a 20-bit immediate.

Load/Store Instructions

Example instruction	Instruction name	Meaning
`ld x1,80(x2)`	Load doubleword	Regs[x1] ← Mem[80+Regs[x2]]
`lw x1,60(x2)`	Load word	Regs[x1] $\leftarrow_{64}$ Mem[60+Regs[x2]]$_0$)^{32} ## Mem[60+Regs[x2]]
`lwu x1,60(x2)`	Load word unsigned	Regs[x1] $\leftarrow_{64}$ 0^{32} ## Mem[60+Regs[x2]]
`lb x1,40(x3)`	Load byte	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{56} ## Mem[40+Regs[x3]]
`lbu x1,40(x3)`	Load byte unsigned	Regs[x1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[x3]]
`lh x1,40(x3)`	Load half word	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{48} ## Mem[40+Regs[x3]]
`flw f0,50(x3)`	Load FP single	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x3]] ## 0^{32}
`fld f0,50(x2)`	Load FP double	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x2]]
`sd x2,400(x3)`	Store double	Mem[400+Regs[x3]] $\leftarrow_{64}$ Regs[x2]
`sw x3,500(x4)`	Store word	Mem[500+Regs[x4]] $\leftarrow_{32}$ Regs[x3]$_{32..63}$
`fsw f0,40(x3)`	Store FP single	Mem[40+Regs[x3]] $\leftarrow_{32}$ Regs[f0]$_{0..31}$
`fsd f0,40(x3)`	Store FP double	Mem[40+Regs[x3]] $\leftarrow_{64}$ Regs[f0]
`sh x3,502(x2)`	Store half	Mem[502+Regs[x2]] $\leftarrow_{16}$ Regs[x3]$_{48..63}$
`sb x2,41(x3)`	Store byte	Mem[41+Regs[x3]] $\leftarrow_{8}$ Regs[x2]$_{56..63}$

Figure A.25 The load and store instructions in RISC-V. Loads shorter than 64 bits are available in both sign-extended and zero-extended forms. All memory references use a single addressing mode. Of course, both loads and stores are available for all the data types shown. Because RV64G supports double precision floating point, all single precision floating point loads must be aligned in the FP register, which are 64-bits wide.

Control Transfer Instructions

Example instruction	Instruction name	Meaning
`jal x1,offset`	Jump and link	`Regs[x1]←PC+4; PC←PC + (offset<<1)`
`jalr x1,x2,offset`	Jump and link register	`Regs[x1]←PC+4; PC←Regs[x2]+offset`
`beq x3,x4,offset`	Branch equal zero	`if (Regs[x3]==Regs[x4]) PC←PC + (offset<<1)`
`bgt x3,x4,name`	Branch not equal zero	`if (Regs[x3]>Regs[x4]) PC←PC + (offset<<1)`

Figure A.27 Typical control flow instructions in RISC-V. All control instructions, except jumps to an address in a register, are PC-relative.

RISC-V Dynamic Instruction Mix for SPECint2006

Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%

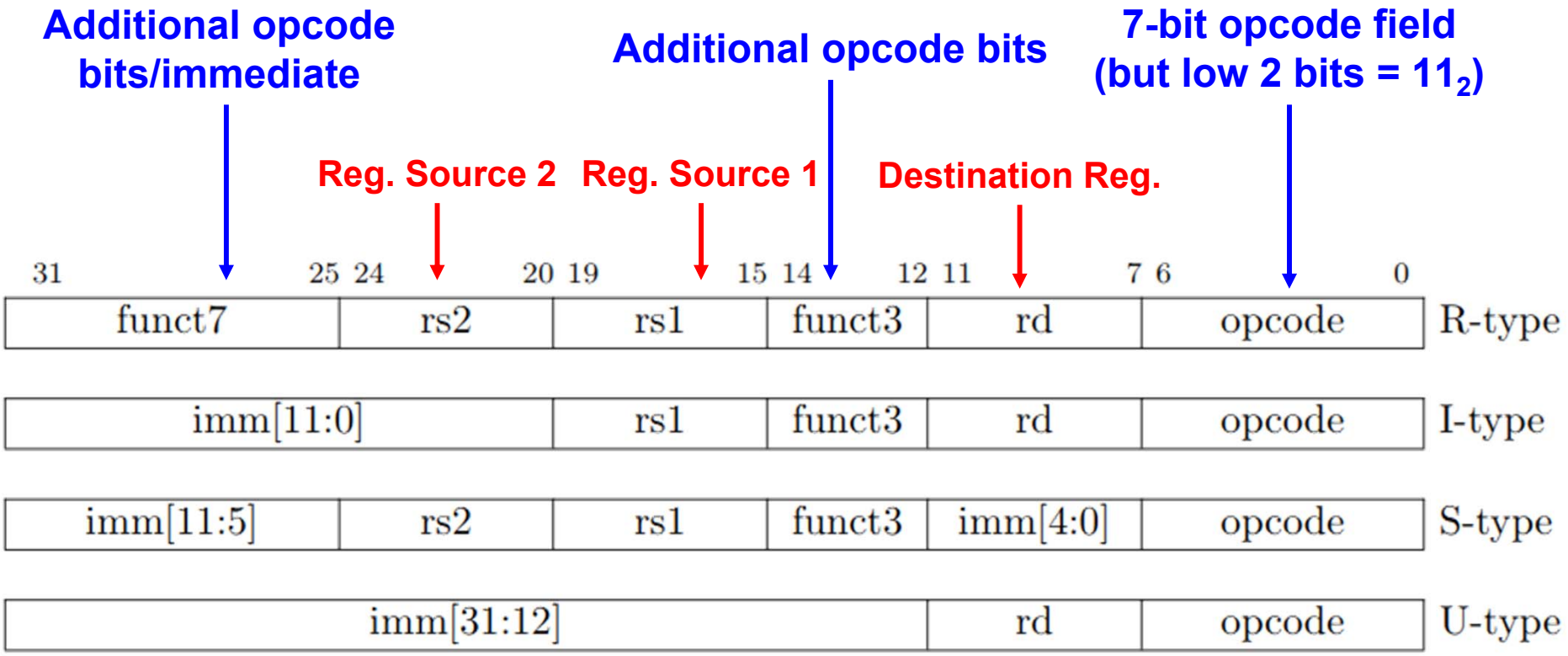
Figure A.29 RISC-V dynamic instruction mix for the SPECint2006 programs. Omnetpp includes 7% of the instructions that are floating point loads, stores, operations, or compares; no other program includes even 1% of other instruction types. A change in gcc in SPECint2006, creates an anomaly in behavior. Typical integer programs have load frequencies that are 1/5 to 3x the store frequency. In gcc, the store frequency is actually higher than the load frequency! This arises because a large fraction of the execution time is spent in a loop that clears memory by storing x0 (not where a compiler like gcc would usually spend most of its execution time!). A store instruction that stores a register pair, which some other RISC ISAs have included, would address this issue.

RISC-V Hybrid Instruction Encoding

- 16, 32, 48, 64, … bits length encoding
- Base instruction set (RV32) always has fixed 32-bit instructions with lowest two bits = 11_2
- All branches and jumps have targets at 16-bit granularity (even in base ISA where all instructions are fixed 32 bits)

Four Core RISC-V Instruction Formats

- https://github.com/riscv/riscv-opcodes/



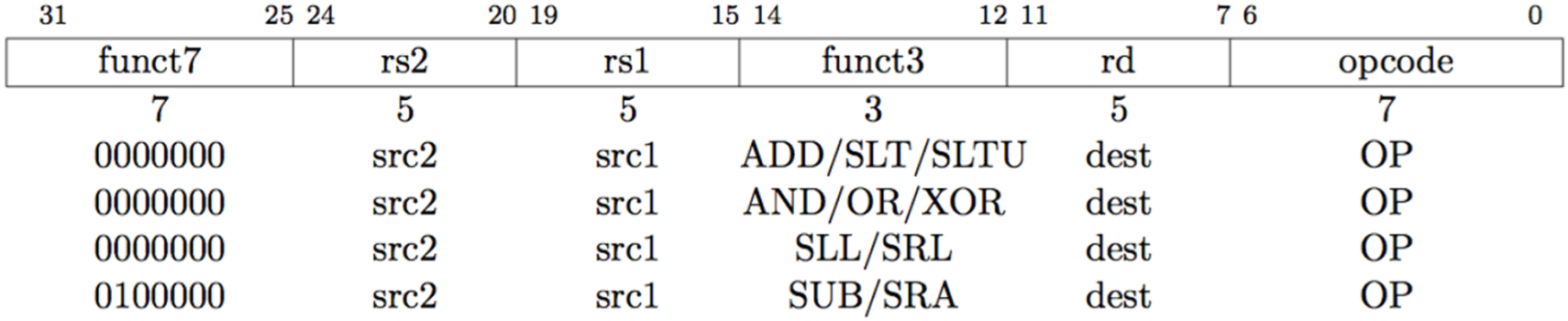
Aligned on a four-byte boundary in memory. There are variants!
Sign bit of immediates always on bit 31 of instruction. Register fields never move.

RISC-V Encoding Summary

Name	Field						Comments
Field size	**7 bits**	**5 bits**	**5 bits**	**3 bits**	**5 bits**	**7 bits**	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	imm[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	Stores
B-type	imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode	Conditional branch format
J-type	imm[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	imm[31:12]				rd	opcode	Upper immediate format

ALU Instructions: R-Type

- R-type (Register)
  - rs1 and rs2 are the source register, rd is the destination
  - ADD/SUB
  - SLT, SLTU: set less than
  - SRL, SLL, SRA: shift logic or arithmetic left or right



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

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

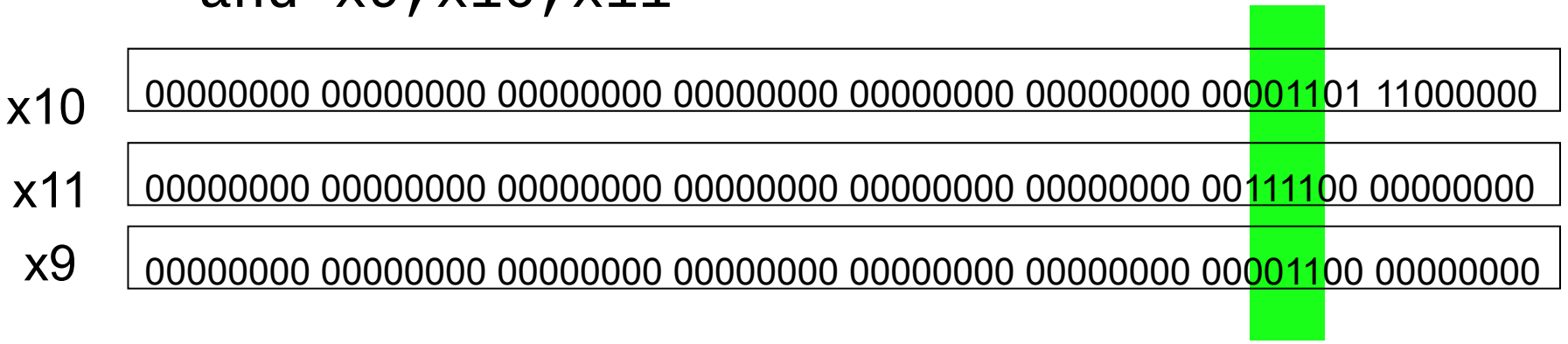
- RISC-V code
  - add x9, x20, x21
  - Decimal version of machine code
  - | 0 | 21 | 20 | 0 | 9 | 51 |
  - Binary version of machine code
  - | 0000000 | 10101 | 10100 | 000 | 01001 | 0110011 |
  - 7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B316$

And Operations

- AND operator
- Useful to mask bits in a word
- Select some bits, clear others to 0
- It is bit-by-bit (bitwise-AND)
- Result=1 : both bits of the operands are 1

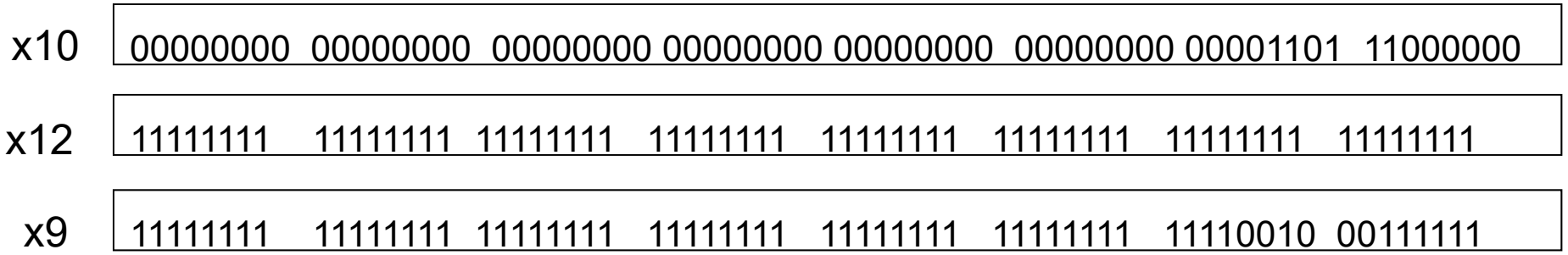
```
and x9,x10,x11
```



XOR Operations

- Differencing operation
  - Set some bits to 1, leave others unchanged

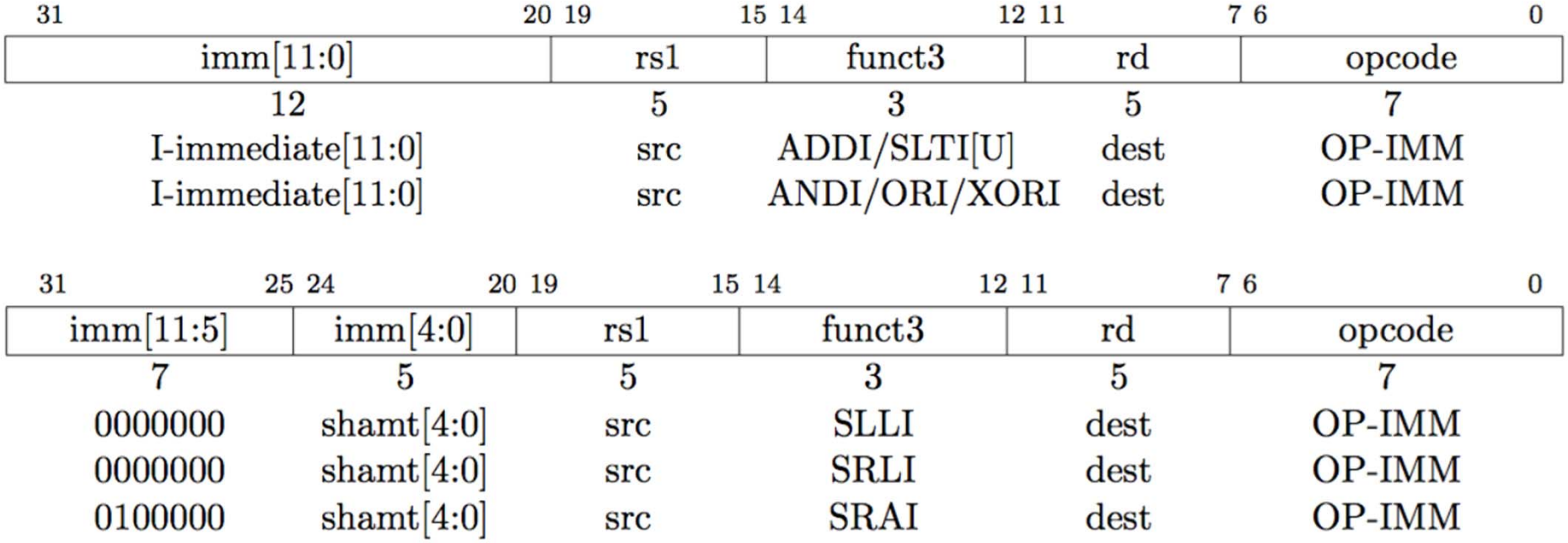
```
xor x9,x10,x12  // NOT operation
```



ALU Instructions: I-Type

- I-type (immediate), all immediates in all instructions are sign extended
  - ADDI: adds sign extended 12-bit immediate to rs1
  - SLTI(U): set less than immediate
  - ANDI/ORI/XORI: logical operations
  - SLLI/SRLI/SRAI: shifts by constants

I-type instructions end with I



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

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

Shift Operations

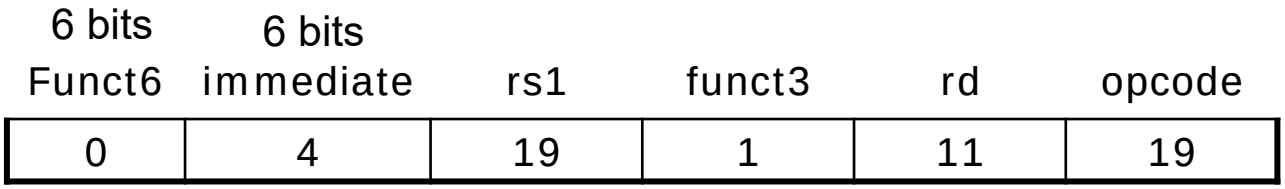
- *Shift* operator
  - Move all the bits in a word to left or right, filling emptied bits with 0
  - Shifting left by ***i*** is same result as multiplying by 2^{i}

0000 0000 0000 0000 0000 0000 0000 1001 $(9)_{10}$

Shift left 4

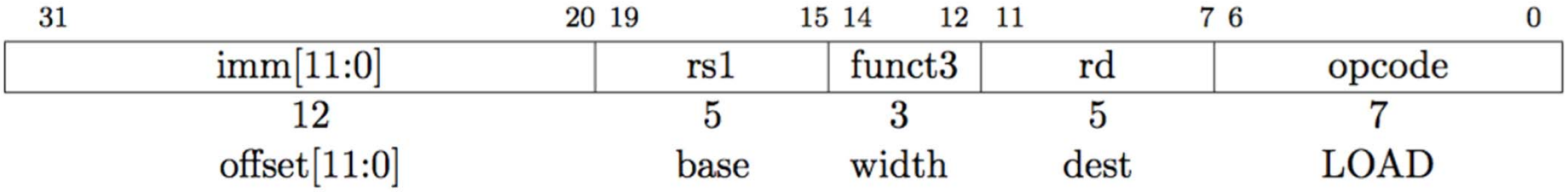
0000 0000 0000 0000 0000 0000 1001 0000 $(9\times16=144)_{10}$

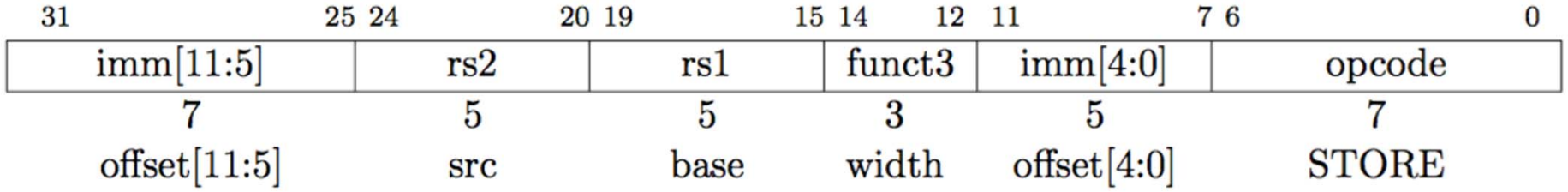
slli x11, x19, 4 // reg x11=reg x19 << 4 bit



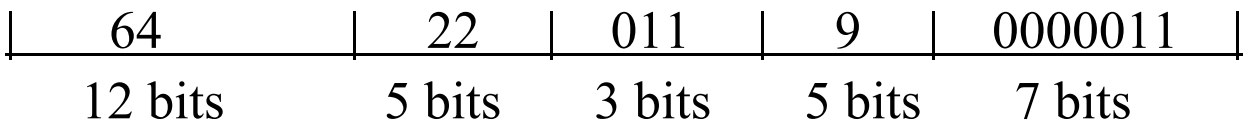
Load/Store Instructions: I/S-Type

- Load instruction (I-type)
  - rd = MEM (rs1 + imm)
- Store instruction (S-type)
  - MEM (rs1 + imm) = rs2





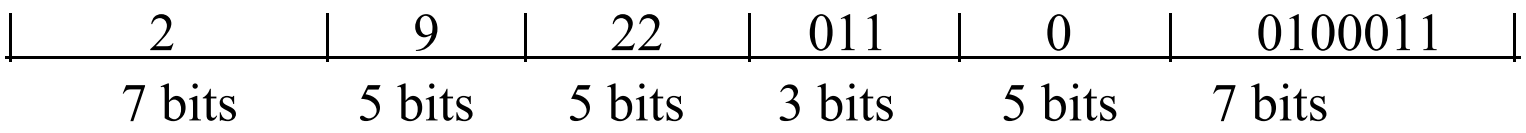
- I-format
  - ld x9, 64(x22)



- S-format

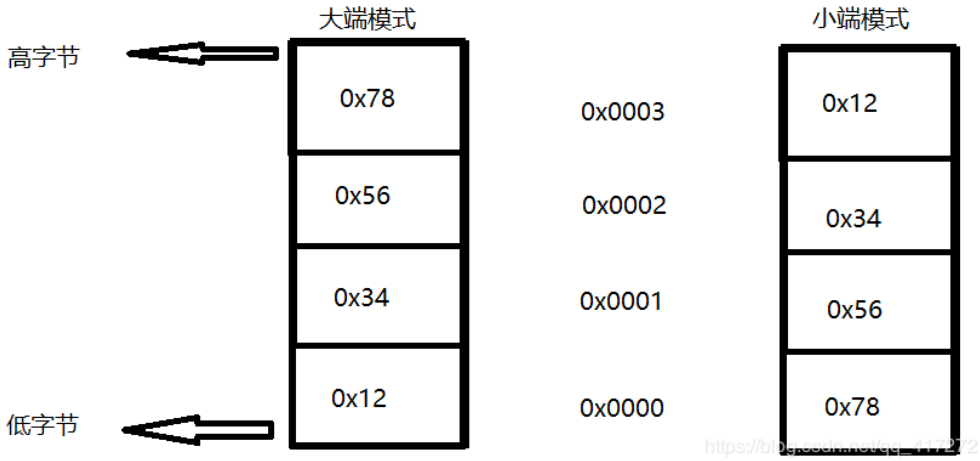
```
S_imm = {{20{inst[31]}}, Inst[31:25], inst[11:7]};
```

  - sd x9, 64(x22)



Endianness/byte order

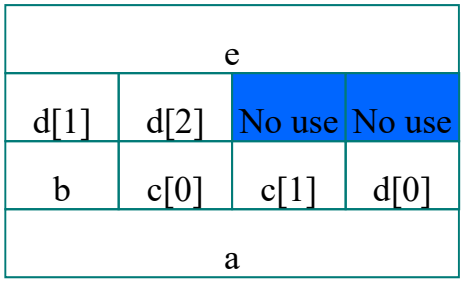
- Big endian：
  - 数据的高字节存放在低地址；
  - 数据的低字节存放在高地址
  - PowerPC
- Little endian：
  - 数据的高字节存放在高地址；
  - 数据的低字节存放在低地址
  - RISC-V
- E.g. : 32位机器上存放0x123456789，其大小端模式存储如下：



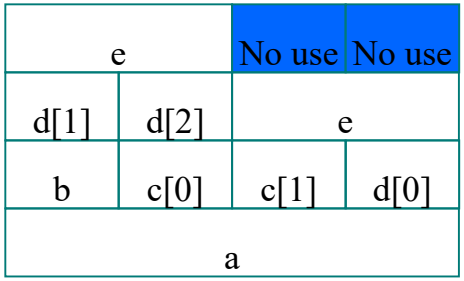
Memory Alignment

```
struct {
        int a;
        char b;
        char c[2];
        char d[3]
        float e;
}
```

Alignment restriction



No alignment restriction



因为一次只能读出4字节内存中的一行

这样布局，e变量不能一次读出

g = h + A[i]

- Compiling using a variable array index
  - C code:

    g = h + A[i] ; // A is an array of 100 doublewords

    (Assume: g, h, ***i*** -- x1, x2, ***x4*** base address of A -- x3)
  - RISC-V code:

```
add   x5, x4, x4     #  temp reg x5 = 2 * i
add   x5, x5, x5     #  temp reg x5 = 4 * i
add   x5, x5, x5     #  temp reg x5 = 8 * i
add   x5, x5, x3     #  x5 = address of A[i] (8 * i + x3)
ld    $x6, 0(x5)     #  temp reg x6 = A[i]
add   x1, x2, x6     #  g = h + A[i]
```

ALU Instructions: U-Type

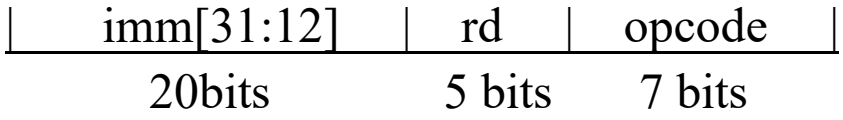
- LUI/AUIPC: load upper immediate/add upper immediate to PC

- Writes 20-bit immediate to top of destination register
- Used to build large immediates
- 12-bit immediates are signed, so have to account for sign when building 32-bit immediates in 2-instruction sequence (LUI high-20 bits, ADDI low-12 bits)

RISC-V Addressing for Wide Immediate and Addresses

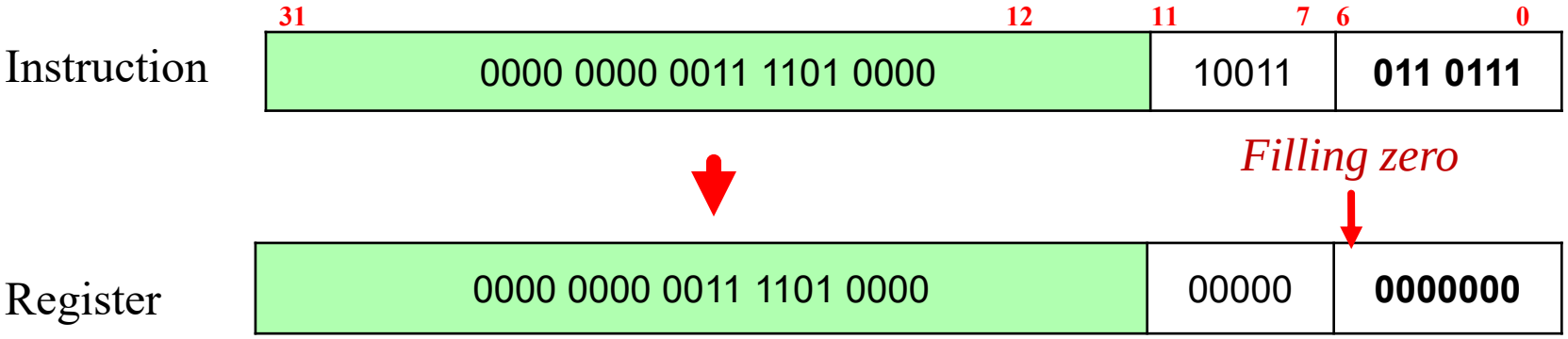
- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rd, constant
```



  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0

```
lui x19, 976  // 0x003D0
```



Example of Loading a 32-bit constant

- The 32-bit constant:

  0000 0000 0011 1101 0000 ***1***001 0000 0000 $(=976*16^3+2304)_{10}$

  RISC V code:

  ```
  lui   s3, 976
  addi  s3, s3, 2304
  ```

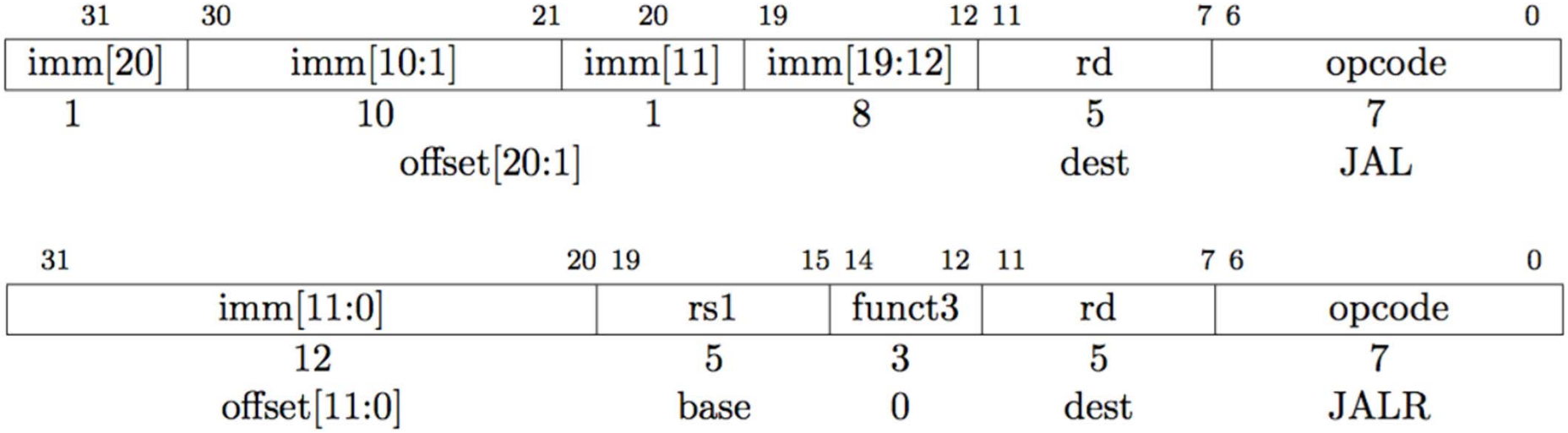
  The value of s3 afterward is:

00000000000000000000000000000000 **0000 0000 0011 1101 0000** ***1***001 0000 0000

- Note：Why does it need two steps?

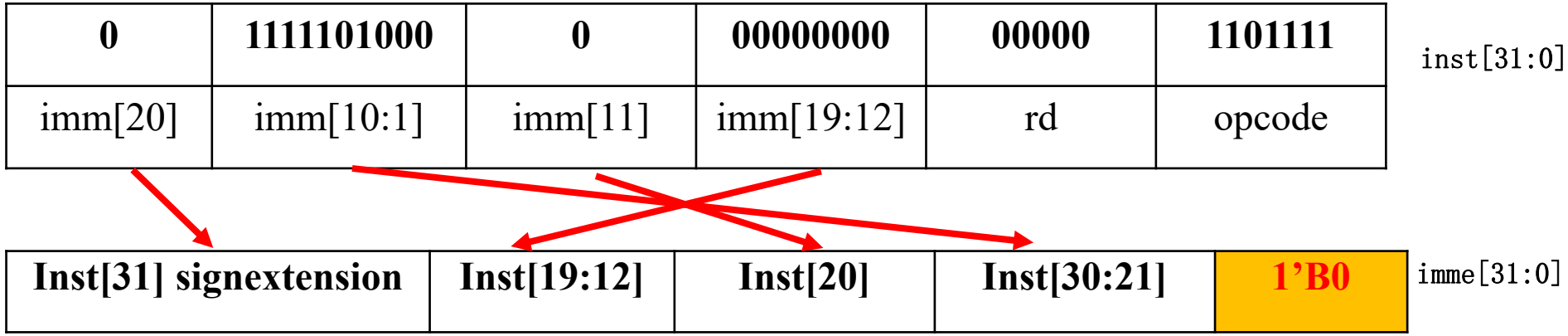
Control Transfer Instructions: J-Type

- No architecturally visible delay slots
- Unconditional jumps: PC + offset target
  - JAL: jump and link, also writes PC + 4 to x1, J-type
    - Offset scaled by 1-bit left shift – can jump to 16-bit instruction boundary (same for branches)
  - JLAR: jump and link register where imm (12bits) + rs1 = target



Jump Addressing

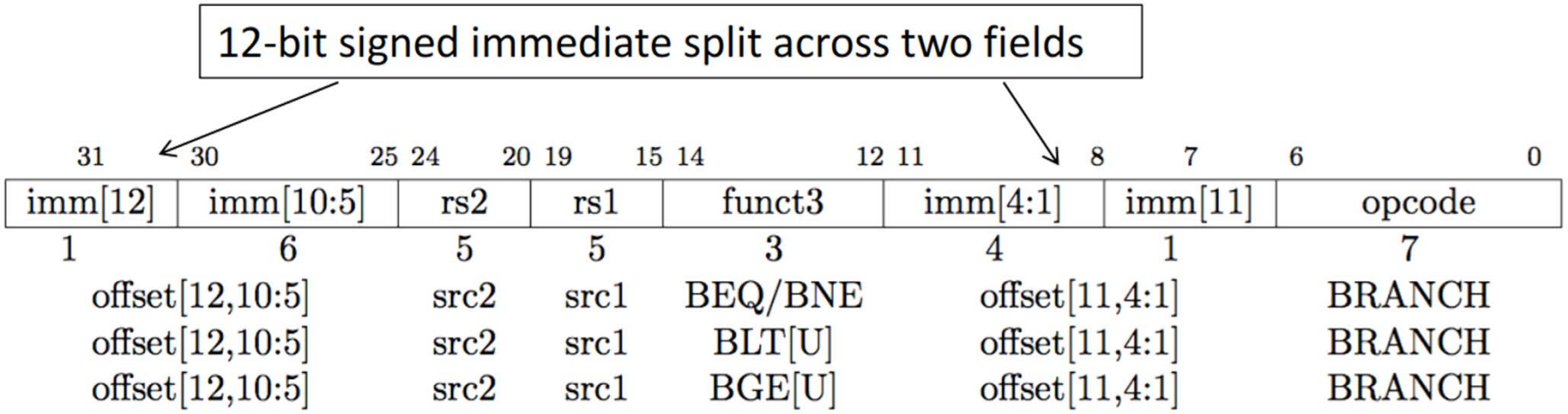
- Jump and link (`jal`) target uses 20-bit immediate for larger range
- J type: jal x0, 2000, //2000 = 0111 1101 0000



- For long jumps, eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

Control Transfer Instructions: B-Type

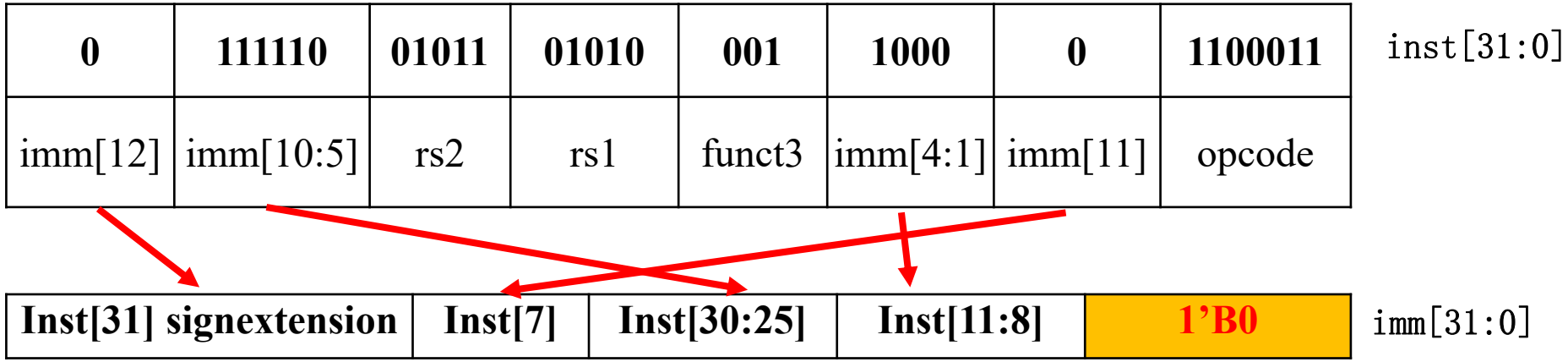
- No architecturally visible delay slots
- Conditional branches: B-type and PC + offset target



Branches, compare two registers, PC + (immediate << 1) target (signed offset in multiples of two). Branches do not have delay slot.

Branch Addressing

- Most branch targets are near branch
  - Forward or backward
- B-type: bne x10, x11, 2000 //2000 = 0111 1101 0000



- PC-relative addressing
  - Target address = PC + branch offset

Compiling an *if* statement to a branch

(Assume: f ~ j ---- x19 ~ x23)

- C code:

```
if ( i == j )  f = g + h ; else f = g – h;
```

- RISC-V assembly code:

```
        bne    x22, x23, ELSE    // go to ELSE if  i != j
        add    x19, x20, x21     // f = g + h  ( skipped if  i  not equals  j )
        beq    x0, x0,  EXIT
ELSE: sub      x19, x20, x21     // f = g - h  ( skipped if  i  equals  j )
EXIT:
```

More Conditional Operations

- `blt rs1, rs2, L1`
  - if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1`
  - if (rs1 >= rs2) branch to instruction labeled L1
- Example
  - if (a > b) a += 1; (a in x22, b in x23)

```
    bge  x23, x22, Exit       // branch if b >= a
    addi x22, x22, 1
Exit:
```

Bounds check Shortcut

- Reduce an index-out-of-bounds check
  - If (x20>=x11 & x20<0) goto IndexOutofBounds
  - RISC-V version:

    bgeu x20, x11, IndexOutofBounds

Compiling a *while* loop

- C language: (Assume: i and k---- x22 and x24 base of save ---- x25)

  while (save[i]==k) i=i+1;

- RISC-V assembler code

```
Loop:   slli   x10, x22, 3      // temp reg x10 = 8 * i
        add    x10, x10, x25    // x10 = address of save[i]
        ld     x9, 0(x10)       // temp reg x9 = save[i]
        bne    x9, x24, Exit    // go to Exit if save[i] != k
        addi   x22, x22, 1      // i = i + 1
        beq    x0,  x0,  Loop   // go to Loop
Exit:
```

Show branch offset in machine language

	Address	fun7	rs2	rs1	fun3	rd/offset	OP	Hex
		instructions Code with Binary						
Loop: slli	80000	0000000	00011	10110	001	01010	0010011	003B1513
add	80004	0000000	11001	01010	000	01010	0110011	01950533
ld	80008	0000000	00000	01010	011	01001	0000011	00053483
bne	***80012***	0000000	11000	01001	001	01100	1100011	01849663
addi	80016	0000000	00001	10110	000	10110	0010011	001B0B13
beq	80020	1111111	00000	00000	000	10101	1100011	FE0006E3
Exit:	80024	……						

-10

6 =0110

-20 = 80000 - 80020

PC + offset : 12 = 80024 - 80012

- Modification:
  - All RISC-V instructions are 4 bytes long
  - PC-relative addressing refers to the number of halfwords
    - The address field at 80012 above should be 6 instead of 12

- **While branch target is far away**
  - Inserts an unconditional jump to target
  - Invert the condition so that the branch decides whether to skip the jump
- **Example**
  - Given a branch:

```
beq   x10, x0, L1
```

  - Rewrite it to offer a much greater branching distance:

```
        bne    x10, x0, L2
        jal    x0,   L1
L2:
```

Case/Switch

Compiling a switch using ***jump address table***

(Assume: f ~ k ---- x20 ~ x25 x5 contains 4)

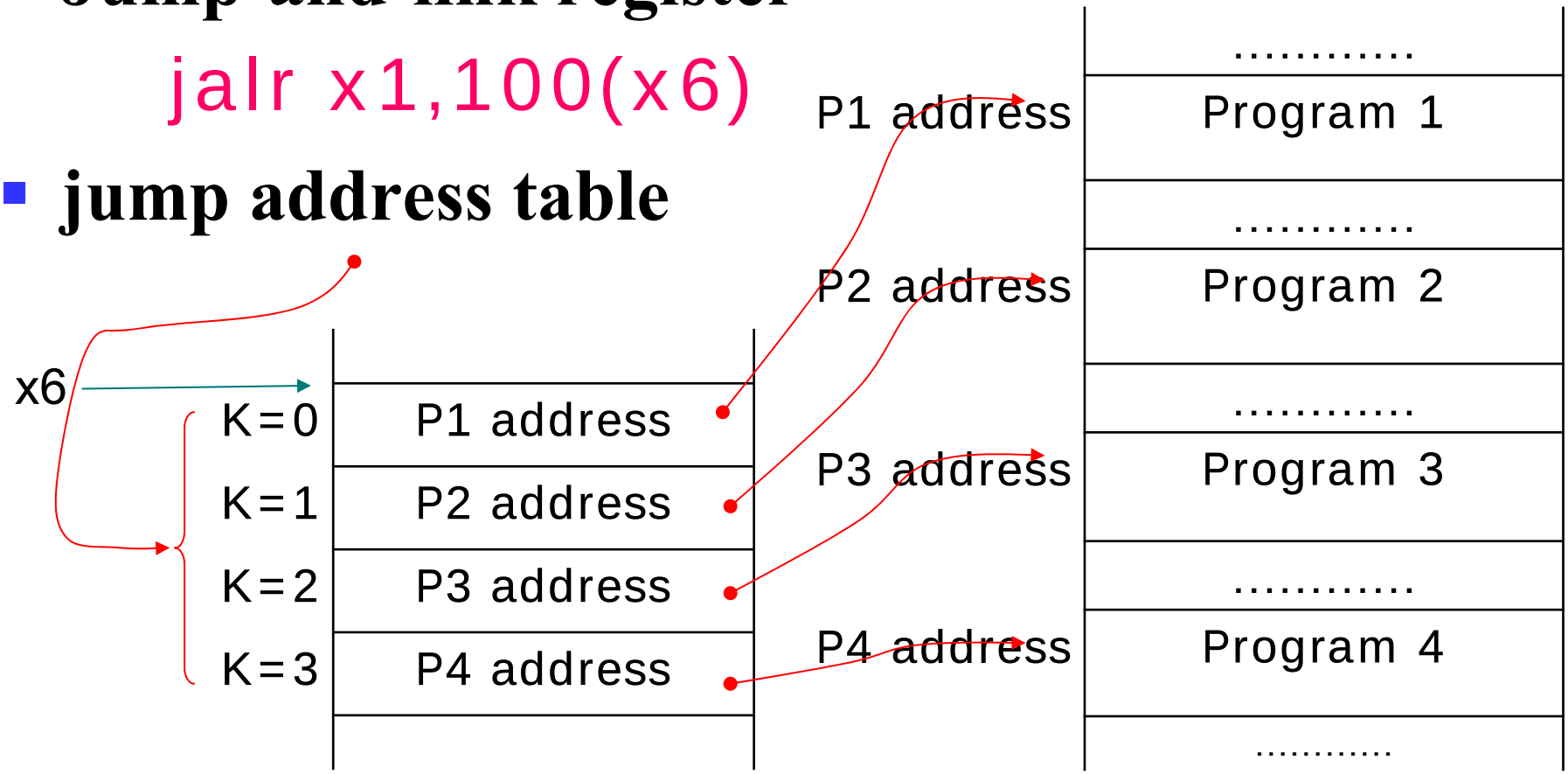
- C code:

```
switch ( k )  {
    case  0 :   f  =  i  +  j ;  break ;   /*  k  =  0  */
    case  1 :   f  =  g +  h ;  break ;   /*  k  =  1  */
    case  2 :   f  =  g  -  h ;  break ;   /*  k  =  2  */
    case  3 :   f  =  i  -  j ;  break ;   /*  k  =  3  */
}
```

- **Jump-and-link register**

  jalr x1,100(x6)

- **jump address table**



RISC-V assembly code:

```
        blt     x25, x0, Exit        // test if  k  <  0
        bge     x25, x5, Exit        // if  k  >=  4,  go to Exit
        slli    x7, x25, 3           // temp reg x7  =  8  *  k
        add     x7, x7, x6           // x7  =  address of JumpTable[k]
        ld      x7, 0(x7)            // x7 gets JumpTable[k]
        jalr    x1, 0(x7)            // jump  entrance
Exit:
```

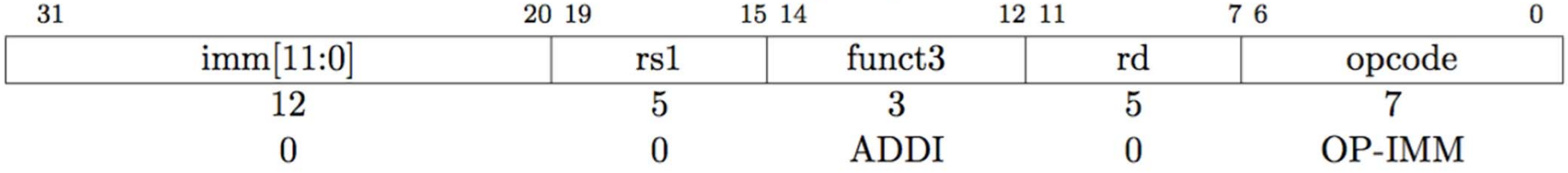
jump address table

L0:address
L1:address
L2: address
L3:address

```
L0:   add    x20, x23, x24      // k  =  0  so  f  gets  i  +  j
      jalr   x0, 0(x1)          // end of this case so go to Exit
L1:   add    x20, x21, x22      // k  =  1  so  f  gets  g  +  h
      jalr   x0, 0(x1)          // end of this case so go to Exit
L2:   sub    x20, x21, x22      // k  =  2  so  f  gets  g  -  h
      jalr   x0, 0(x1)          // end of this case so go to Exit
L3:   sub    x20, x23, x24      // k  =  3  so  f  gets  i  -  j
      jalr   x0, 0(x1)          //end of  switch statement
```

Where is NOP?

ADDI x0, x0, 0



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

Decoding Machine Language

- Machine instruction (0x00578833)

  0000 0000 0101 0111 1000 1000 0011 0011

  - Decoding
    - Determine the operation from opcode

      opcode: 0110011 → **R-type arithmetic instruction**

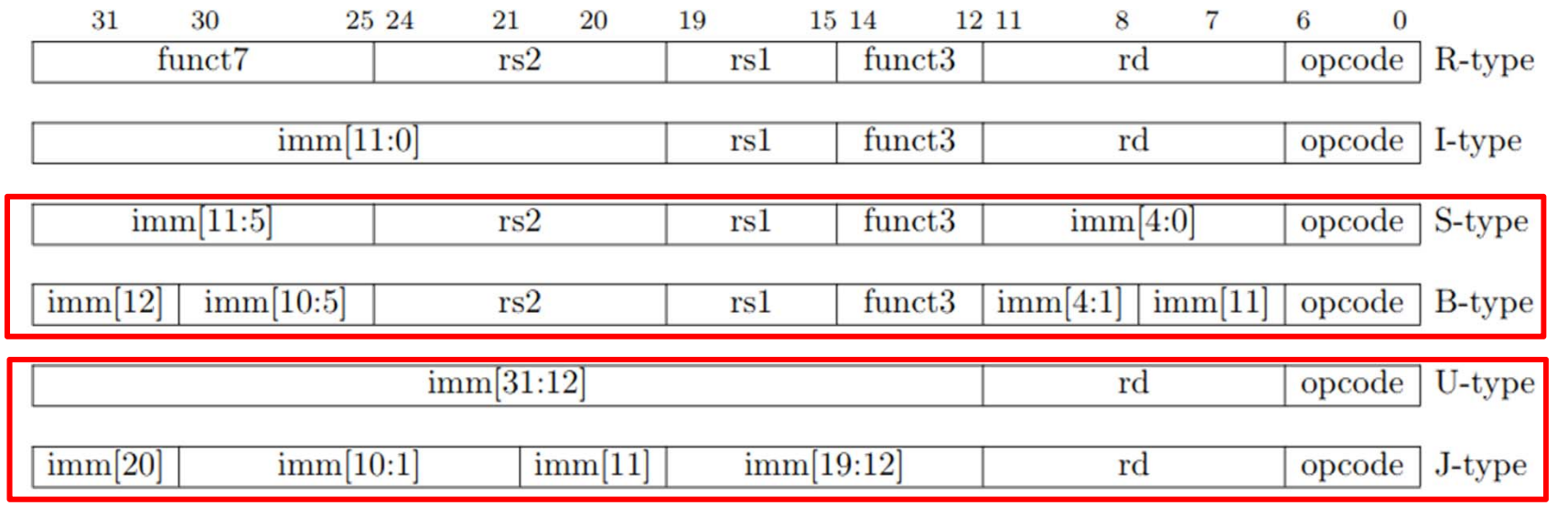
funct7	rs2	rs1	funct3	rd	opcode
0000000	00101	01111	000	10000	0110011

      funct7 and funct3 are all 0 → **add instruction**
    - Determine other fields

      rs2: x5; rs1: x15; rd: x16
    - Show the assembly instruction

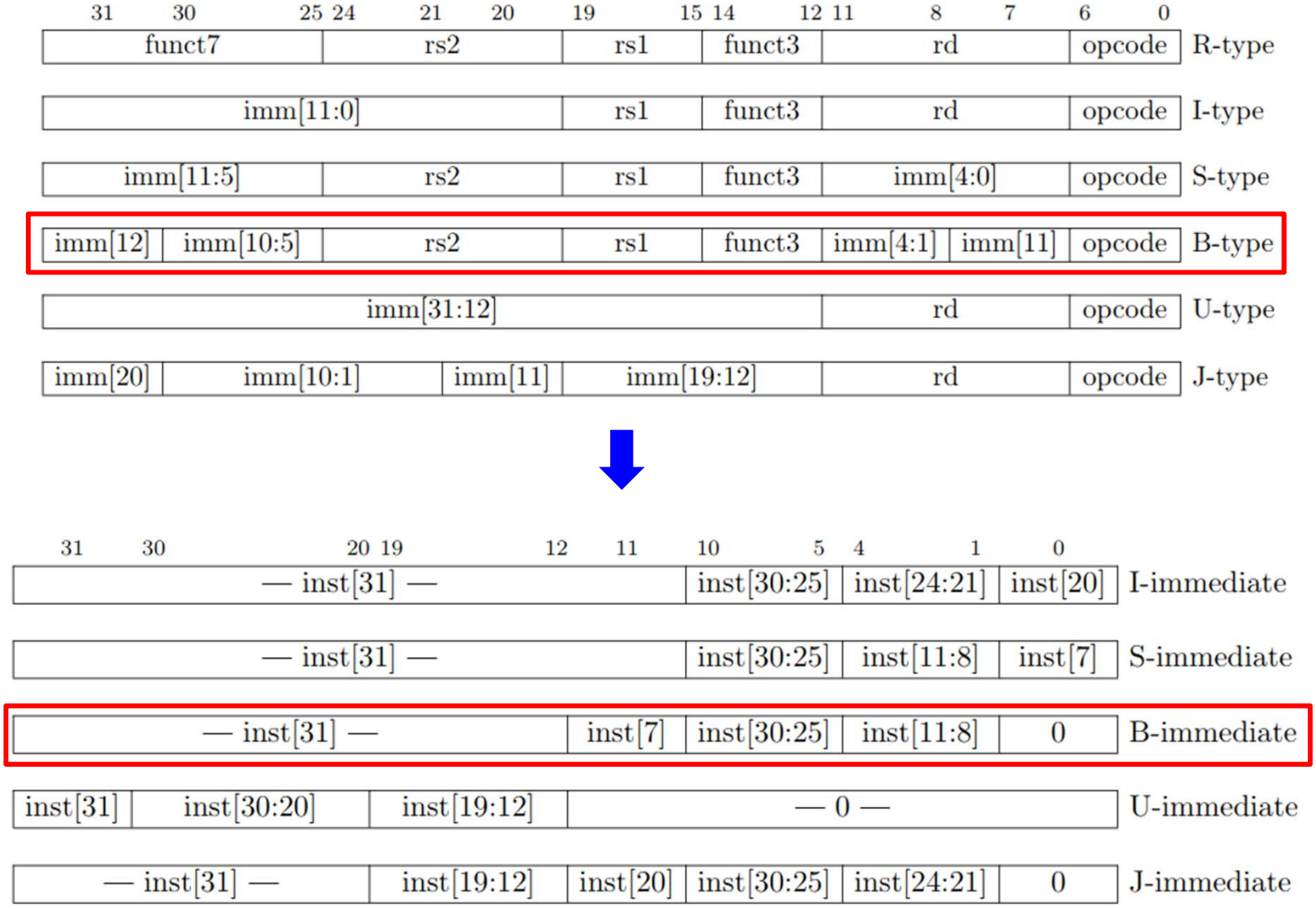
      add x16, x15, x5 (Note: add rd,rs1,rs2)

Immediate Encoding Variants

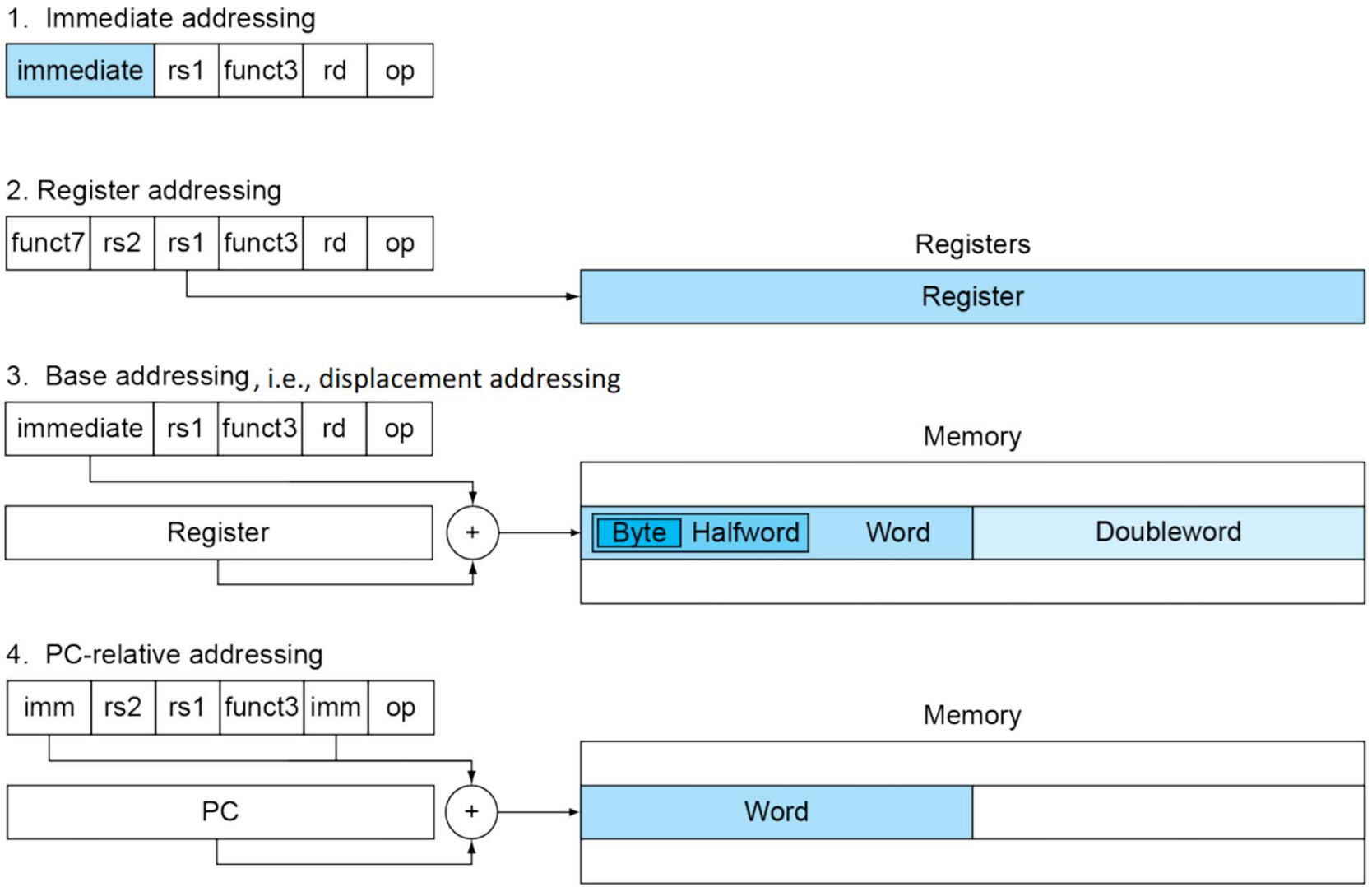


- S-type vs. B-type
  - The 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.
- U-type vs. J-type
  - Similarly, the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

Immediate Encoding Variants



RISC-V Addressing Modes



Privileged ISA: Modes

- RISC-V privileged spec defines 3 levels of privilege, called modes
  - Machine mode is the highest privileged mode and the only required mode

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

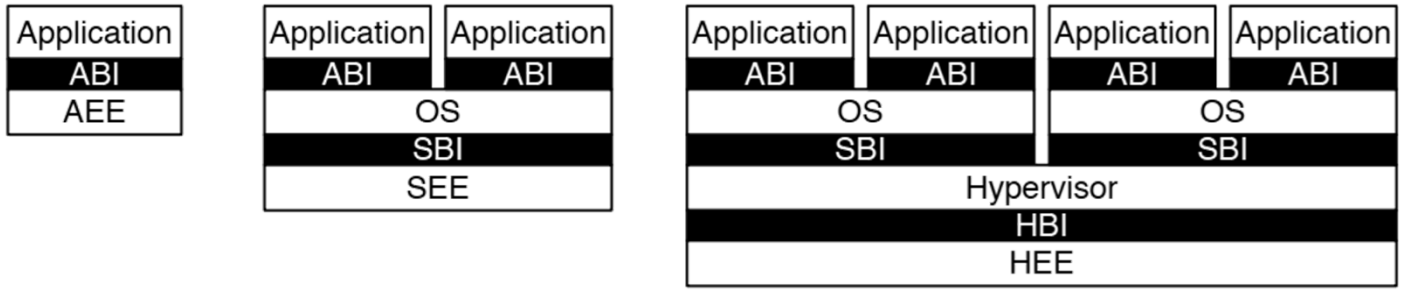
- More-privileged modes generally have access to all of the features of less-privileged modes, and they add additional functionality not available to less-privileged modes, such as the ability to handle interrupts and perform I/O. Processors typically spend most of their execution time in their least-privileged mode; interrupts and exceptions transfer control to more-privileged modes.

Software Stack and Instructions

- Implementations might provide anywhere from 1 to 4 privilege modes trading off reduced isolation for lower implementation cost

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

- RISC-V privileged software stack



- RV32/64 privileged instructions

machine-mode
supervisor-mode
} trap return

supervisor-mode fence.virtual memory address

wait for interrupt

RISC-V Reference Card

Open RISC-V Reference Card ①

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base		+RV64I	
Shifts	Shift Left Logical	R	SLL	rd,rs1,rs2	SLLW	rd,rs1,rs2
	Shift Left Log. Imm.	I	SLLI	rd,rs1,shamt	SLLIW	rd,rs1,shamt
	Shift Right Logical	R	SRL	rd,rs1,rs2	SRLW	rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI	rd,rs1,shamt	SRLIW	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2	SRAW	rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI	rd,rs1,shamt	SRAIW	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2	ADDW	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm	ADDIW	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2	SUBW	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm		
	Add Upper Imm to PC	U	AUIPC	rd,imm		
Logical	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	Set <	R	SLT	rd,rs1,rs2		
	Set < Immediate	I	SLTI	rd,rs1,imm		
	Set < Unsigned	R	SLTU	rd,rs1,rs2		
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm		
Branches	Branch =	B	BEQ	rs1,rs2,imm		
	Branch ≠	B	BNE	rs1,rs2,imm		
	Branch <	B	BLT	rs1,rs2,imm		
	Branch ≥	B	BGE	rs1,rs2,imm		
	Branch < Unsigned	B	BLTU	rs1,rs2,imm		
	Branch ≥ Unsigned	B	BGEU	rs1,rs2,imm		
Jump & Link	J&L	J	JAL	rd,imm		
	Jump & Link Register	I	JALR	rd,rs1,imm		
Synch	Synch thread	I	FENCE			
	Synch Instr & Data	I	FENCE.I			
Environment	CALL	I	ECALL			
	BREAK	I	EBREAK			

Control Status Register (CSR)

Read/Write	I	CSRRW	rd,csr,rs1
Read & Set Bit	I	CSRRS	rd,csr,rs1
Read & Clear Bit	I	CSRRC	rd,csr,rs1
Read/Write Imm	I	CSRRWI	rd,csr,imm
Read & Set Bit Imm	I	CSRRSI	rd,csr,imm
Read & Clear Bit Imm	I	CSRRCI	rd,csr,imm

Loads	Load Byte	I	LB	rd,rs1,imm	
	Load Halfword	I	LH	rd,rs1,imm	
	Load Byte Unsigned	I	LBU	rd,rs1,imm	
	Load Half Unsigned	I	LHU	rd,rs1,imm	
	Load Word	I	LW	rd,rs1,imm	
Stores	Store Byte	S	SB	rs1,rs2,imm	
	Store Halfword	S	SH	rs1,rs2,imm	
	Store Word	S	SW	rs1,rs2,imm	SD rs1,rs2,imm

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	MRET
	Supervisor-mode trap return	R	SRET
Interrupt	Wait for Interrupt	R	WFI
MMU	Virtual Memory FENCE	R	SFENCE.VMA rs1,rs2

Examples of the 60 RV Pseudoinstructions

Branch = 0 (BEQ rs,x0,imm)	J	BEQZ rs,imm
Jump (uses JAL x0,imm)	J	J imm
MoVe (uses ADDI rd,rs,0)	R	MV rd,rs
RETurn (uses JALR x0,0,ra)	I	RET

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC		RISC-V equivalent	
Loads	Load Word	CL	C.LW	rd',rs1',imm	LW	rd',rs1',imm*4
	Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4
	Float Load Word SP	CL	C.FLW	rd',rs1',imm	FLW	rd',rs1',imm*8
	Float Load Word	CI	C.FLWSP	rd,imm	FLW	rd,sp,imm*8
	Float Load Double	CL	C.FLD	rd',rs1',imm	FLD	rd',rs1',imm*16
	Float Load Double SP	CI	C.FLDSP	rd,imm	FLD	rd,sp,imm*16
Stores	Store Word	CS	C.SW	rs1',rs2',imm	SW	rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4
	Float Store Word	CS	C.FSW	rs1',rs2',imm	FSW	rs1',rs2',imm*8
	Float Store Word SP	CSS	C.FSWSP	rs2,imm	FSW	rs2,sp,imm*8
	Float Store Double	CS	C.FSD	rs1',rs2',imm	FSD	rs1',rs2',imm*16
	Float Store Double SP	CSS	C.FSDSP	rs2,imm	FSD	rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1
	ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd',imm	ADDI	rd',sp,imm*4
	SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1
	AND	CR	C.AND	rd,rs1	AND	rd,rd,rs1
	AND Immediate	CI	C.ANDI	rd,imm	ANDI	rd,rd,imm
	OR	CR	C.OR	rd,rs1	OR	rd,rd,rs1
	eXclusive OR	CR	C.XOR	rd,rs1	AND	rd,rd,rs1
	MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0
	Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm
	Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
	Shift Right Ari. Imm.	CI	C.SRAI	rd,imm	SRAI	rd,rd,imm
	Shift Right Log. Imm.	CI	C.SRLI	rd,imm	SRLI	rd,rd,imm
Branches	Branch=0	CB	C.BEQZ	rs1',imm	BEQ	rs1',x0,imm
	Branch≠0	CB	C.BNEZ	rs1',imm	BNE	rs1',x0,imm
Jump	Jump	CJ	C.J	imm	JAL	x0,imm
	Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0
Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
	Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0
System	Env. BREAK	CI	C.EBREAK		EBREAK	

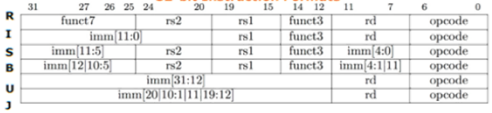
+RV64I	
LWU	rd,rs1,imm
LD	rd,rs1,imm
SD	rs1,rs2,imm

Optional Compressed Extention: RV64C

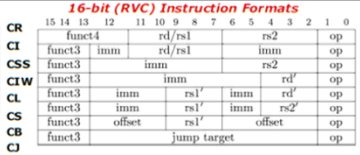
All RV32C (except C.JAL*, 4 word loads, 4 word strores) plus:*

ADD Word (C.ADDW)	Load Doubleword (C.LD)
ADD Imm. Word (C.ADDIW)	Load Doubleword SP (C.LDSP)
SUBtract Word (C.SUBW)	Store Doubleword (C.SD)
	Store Doubleword SP (C.SDSP)

32-bit Instruction Formats



16-bit (RVC) Instruction Formats



RISC-V Integer Base (RV32I/64I), privileged, and optional RV32/64C. Registers `x1-x31` and the `PC` are 32 bits wide in RV32I and 64 in RV64I (`x0=0`). RV64I adds 12 instructions for the wider data. Every 16-bit RVC instruction maps to an existing 32-bit RISC-V instruction.

Open RISC-V Reference Card ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV64M	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MULW	rd,rs1,rs2
	MULtiply High	R	MULH	rd,rs1,rs2		
	MULtiply High Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply High Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIVW	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REMW	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMUW	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV64A	
Load	Load Reserved	R	LR.W	rd,rs1	LR.D	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.D	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.D	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.D	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.D	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.D	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.D	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.D	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.D	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.D	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.D	rd,rs1,rs2

Two Optional Floating-Point Instruction Extensions: RVF & RVD

Category	Name	Fmt	RV32{F|D} (SP,DP Fl. Pt.)		+RV64{F|D}	
Move	Move from Integer	R	FMV.W.X	rd,rs1	FMV.D.X	rd,rs1
	Move to Integer	R	FMV.X.W	rd,rs1	FMV.X.D	rd,rs1
Convert	ConVerT from Int	R	FCVT.{S|D}.W	rd,rs1	FCVT.{S|D}.L	rd,rs1
	ConVerT from Int Unsigned	R	FCVT.{S|D}.WU	rd,rs1	FCVT.{S|D}.LU	rd,rs1
	ConVerT to Int	R	FCVT.W.{S|D}	rd,rs1	FCVT.L.{S|D}	rd,rs1
	ConVerT to Int Unsigned	R	FCVT.WU.{S|D}	rd,rs1	FCVT.LU.{S|D}	rd,rs1
Load	Load	I	FL{W,D}	rd,rs1,imm		
Store	Store	S	FS{W,D}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D}	rd,rs1,rs2		
Compare	compare Float =	R	FEQ.{S|D}	rd,rs1,rs2		
	compare Float <	R	FLT.{S|D}	rd,rs1,rs2		
	compare Float ≤	R	FLE.{S|D}	rd,rs1,rs2		
Categorize	CLASSify type	R	FCLASS.{S|D}	rd,rs1		
Configure	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

Calling Convention

Register	ABI Name	Saver
x0	zero	---
x1	ra	Caller
x2	sp	Callee
x3	gp	---
x4	tp	---
x5-7	t0-2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-11	a0-1	Caller
x12-17	a2-7	Caller
x18-27	s2-11	Callee
x28-31	t3-t6	Caller
f0-7	ft0-7	Caller
f8-9	fs0-1	Callee
f10-11	fa0-1	Caller
f12-17	fa2-7	Caller
f18-27	fs2-11	Callee
f28-31	ft8-11	Caller
zero	Hardwired zero	
ra	Return address	
sp	Stack pointer	
gp	Global pointer	
tp	Thread pointer	
t0-0,ft0-7	Temporaries	
s0-11,fs0-11	Saved registers	
a0-7,fa0-7	Function args	

Optional Vector Extension: RVV

	Name	Fmt	RV32V/R64V
SET Vector Len.	R	VSETVL	rd,rs1
MULtiply High	R	VMULH	rd,rs1,rs2
REMainder	R	VREM	rd,rs1,rs2
Shift Left Log.	R	VSLL	rd,rs1,rs2
Shift Right Log.	R	VSRL	rd,rs1,rs2
Shift R. Arith.	R	VSRA	rd,rs1,rs2
LoaD	I	VLD	rd,rs1,imm
LoaD Strided	R	VLDS	rd,rs1,rs2
LoaD indeXed	R	VLDX	rd,rs1,rs2
STore	S	VST	rd,rs1,imm
STore Strided	R	VSTS	rd,rs1,rs2
STore indeXed	R	VSTX	rd,rs1,rs2
AMO SWAP	R	AMOSWAP	rd,rs1,rs2
AMO ADD	R	AMOADD	rd,rs1,rs2
AMO XOR	R	AMOXOR	rd,rs1,rs2
AMO AND	R	AMOAND	rd,rs1,rs2
AMO OR	R	AMOOR	rd,rs1,rs2
AMO MINimum	R	AMOMIN	rd,rs1,rs2
AMO MAXimum	R	AMOMAX	rd,rs1,rs2
Predicate =	R	VPEQ	rd,rs1,rs2
Predicate ≠	R	VPNE	rd,rs1,rs2
Predicate <	R	VPLT	rd,rs1,rs2
Predicate ≥	R	VPGE	rd,rs1,rs2
Predicate AND	R	VPAND	rd,rs1,rs2
Pred. AND NOT	R	VPANDN	rd,rs1,rs2
Predicate OR	R	VPOR	rd,rs1,rs2
Predicate XOR	R	VPXOR	rd,rs1,rs2
Predicate NOT	R	VPNOT	rd,rs1
Pred. SWAP	R	VPSWAP	rd,rs1
MOVe	R	VMOV	rd,rs1
ConVerT	R	VCVT	rd,rs1
ADD	R	VADD	rd,rs1,rs2
SUBtract	R	VSUB	rd,rs1,rs2
MULtiply	R	VMUL	rd,rs1,rs2
DIVide	R	VDIV	rd,rs1,rs2
SQuare RooT	R	VSQRT	rd,rs1,rs2
Multiply-ADD	R	VFMADD	rd,rs1,rs2,rs3
Multiply-SUBtract	R	VFMSUB	rd,rs1,rs2,rs3
Neg. Mul.-SUB	R	VFNMSUB	rd,rs1,rs2,rs3
Neg. Mul.-ADD	R	VFNMADD	rd,rs1,rs2,rs3
SiGN inJect	R	VSGNJ	rd,rs1,rs2
Neg SiGN inJect	R	VSGNJN	rd,rs1,rs2
Xor SiGN inJect	R	VSGNJX	rd,rs1,rs2
MINimum	R	VMIN	rd,rs1,rs2
MAXimum	R	VMAX	rd,rs1,rs2
XOR	R	VXOR	rd,rs1,rs2
OR	R	VOR	rd,rs1,rs2
AND	R	VAND	rd,rs1,rs2
CLASS	R	VCLASS	rd,rs1
SET Data Conf.	R	VSETDCFG	rd,rs1
EXTRACT	R	VEXTRACT	rd,rs1,rs2
MERGE	R	VMERGE	rd,rs1,rs2
SELECT	R	VSELECT	rd,rs1,rs2

RISC-V calling convention and five optional extensions: 8 RV32M; 11 RV32A; 34 floating-point instructions each for 32- and 64-bit data (RV32F, RV32D); and 53 RV32V. Using regex notation, {} means set, so `FADD.{F|D}` is both `FADD.F` and `FADD.D`. RV32{F|D} adds registers `f0-f31`, whose width matches the widest precision, and a floating-point control and status register `fcsr`. RV32V adds vector registers `v0-v31`, vector predicate registers `vp0-vp7`, and vector length register `vl`. RV64 adds a few instructions: RVM gets 4, RVA 11, RVF 6, RVD 6, and RVV 0.

RISC-V operands

Name	Example	Comments
32 registers	*x*0-*x*31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{61} memory words	Memory[0], Memory[8], …, Memory[18,446,744,073,709,551,608]]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential double word accesses differ by 8. Memory holds data structures, arrays, and spilled registers.

Name	Register no.	Usage	Preserved on call
***x*0(zero)**	**0**	**The constant value 0**	n.a.
*x*1(**ra**)	1	Return address(link register)	yes
*x*2(**sp**)	2	Stack pointer	yes
*x*3(gp)	3	Global pointer	yes
*x*4(tp)	4	Thread pointer	yes
*x*5-*x*7(t0-t2)	5-7	Temporaries	no
*x*8(s0/fp)	8	Saved/frame point	Yes
*x*9(s1)	9	Saved	Yes
*x*10-*x*17(a0-a7)	10-17	Arguments/results	no
*x*18-*x*27(s2-s11)	18-27	Saved	yes
*x*28-*x*31(t3-t6)	28-31	Temporaries	No
PC	-	Auipc(Add Upper Immediate to PC)	Yes

Summary of RISC-V instruction encoding

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100

Format	Instruction	Opcode	Funct3	Funct6/7
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	ld	0000011	011	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	lwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.

Format	Instruction	Opcode	Funct3	Funct6/7
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
B-type	beq	1100011	000	n.a.
	bne	1100011	001	n.a.
	blt	1100011	100	n.a.
	bge	1100011	101	n.a.
	bltu	1100011	110	n.a.
	bgeu	1100011	111	n.a.
U-type	lui	0110111	n.a.	n.a.
J-type	jal	1101111	n.a.	n.a.

Specifications and Software

- Specification from RISC-V website
  - https://riscv.org/specifications/
- RISC-V software includes
  - Toolchain projects
    - https://wiki.riscv.org/display/HOME/Toolchain+Projects
  - A simulator (“spike”)
    - https://github.com/riscv-software-src/riscv-isa-sim
  - Standard simulator QEMU (Upstream now)
    - https://github.com/riscv/riscv-qemu
- Operating systems support exists for Linux (Upstream now)
  - https://github.com/riscv/riscv-linux
- A javascript ISA simulator to run a RISC-V Linux system on a web browser
  - https://github.com/riscv/riscv-angel

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Some basic concepts
  - From source code to a running program

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Some basic concepts
  - From source code to a running program

Function calls

- Caller: calling function
  - ***main*** in this case

C Code

```
void main()
{
   int y;
   y = sum(42, 7);
   ...
}
```

- Callee: called function
  - ***sum*** in this case

```
int sum(int a, int b)
{
   return (a + b);
}
```

Function calls

- Caller: calling function
  - ***main*** in this case
  - Passes arguments to callee
  - Jumps to callee
- Callee: called function
  - ***sum*** in this case
  - Performs the function
  - Returns result to caller
  - Returns to point of call

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Calling Convention

- Six general steps in calling a function
  - Place the arguments where the function can access them.
  - Jump to the function (using RV32I's jal).
  - Acquire local storage resources the function needs, saving registers as required.
  - Perform the desired task of the function.
  - Place the function result value where the calling program can access it, restore any registers, and release any local storage resources.
  - Since a function can be called from several points in a program, return control to the point of origin (using ret).

Function Entry and Exit

- The function **prologue**
  - If there are too many function arguments and variables to fit in the registers, the prologue allocates space on **the stack** for the function **frame**, as it is called.

```
entry_label:
  addi sp,sp,-framesize    # Allocate space for stack frame
                           # by adjusting stack pointer (sp register)
  sw   ra,framesize-4(sp)  # Save return address (ra register)
  # save other registers to stack if needed
  ... # body of the function
```

- The function **epilogue**
  - Undoes the stack frame and returns to the point of origin.

```
  # restore registers from stack if needed
  lw   ra,framesize-4(sp)  # Restore return address register
  addi sp,sp, framesize    # De-allocate space for stack frame
  ret                      # Return to calling point
```

Procedure Call Instructions

- Procedure call: jump and link

```
jal x1, ProcedureLabel
```

  - Address of following instruction put in x1
  - Jumps to target address
- Procedure return: jump and link register

```
jalr x0, 0(x1)
```

  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Using More Registers

- Registers for procedure calling
  - x10~ x17: eight argument registers to pass parameters or return values
  - x1: one return address register to return to origin point
- Stack： Ideal data structure for spilling registers
  - Push, pop
  - Stack pointer (sp): x2
- Stack grow from higher address to lower address
  - Push: sp= sp-8
  - Pop: sp = sp+8

Leaf Procedure Example

- C code:

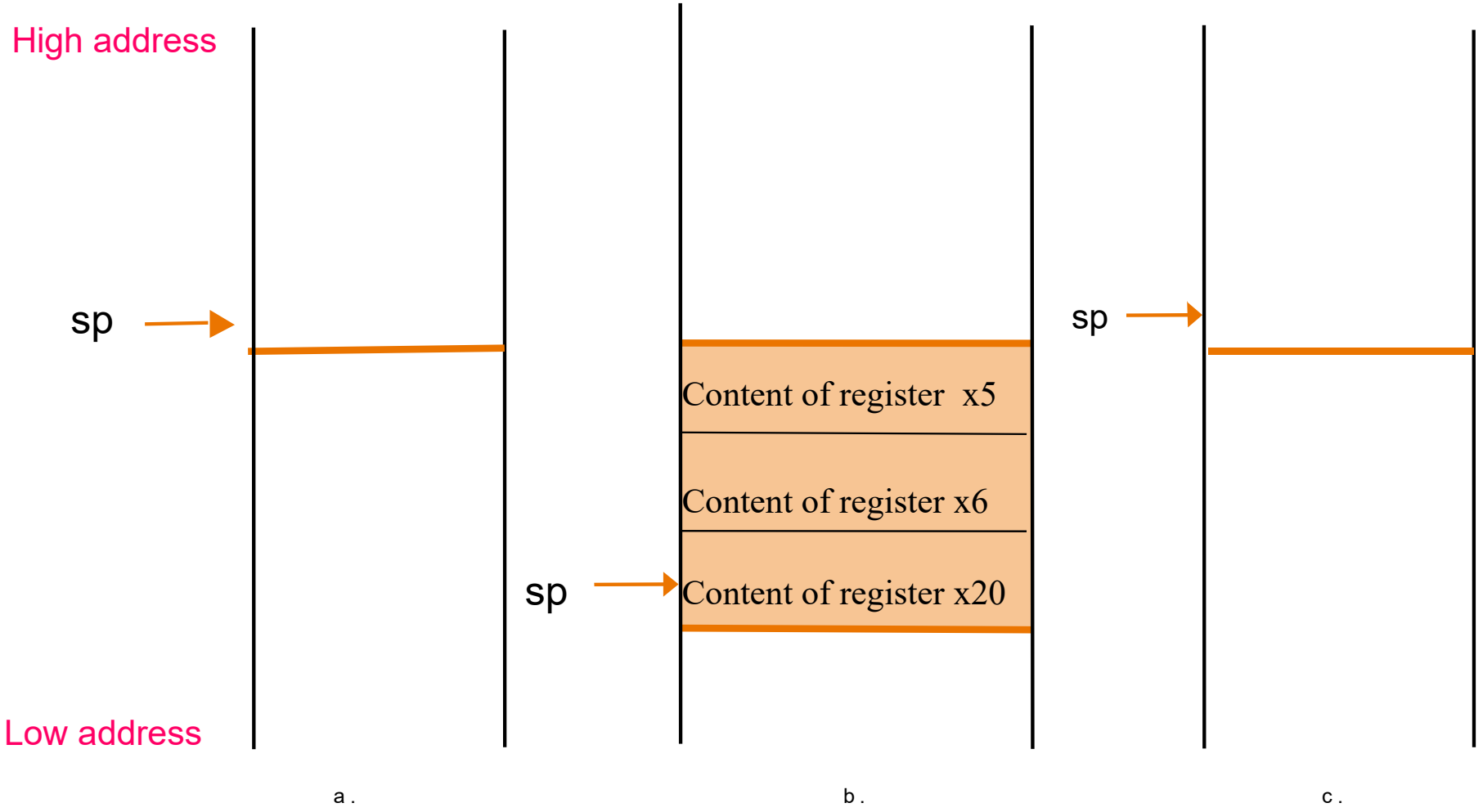
```
long long int leaf_example (
   long long int g, long long int h,
   long long int i, long long int j) {
      long long int f;
      f = (g + h) - (i + j);
      return f;
   }
```

  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Need to save x5, x6, x20 on stack

RISC-V code:

```
addi sp,sp,-24          //Save x5, x6, x20 on stack
sd   x5,16(sp)
sd   x6,8(sp)
sd   x20,0(sp)
add  x5,x10,x11         // x5 = g + h
add  x6,x12,x1          // x6 = i + j
sub  x20,x5,x6          // f = x5 – x6
addi x10,x20,0          // copy f to return register
ld   x20,0(sp)          // Resore x5, x6, x20 from stack
ld   x6,8(sp)
ld   x5,16(sp)
addi sp,sp,24
jalr x0,0(x1)           // Return to caller
```

Local Data on the Stack



Register Usage

- x5 – x7, x28 – x31: temporary registers
  - Not preserved by the callee
- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them

Saved/temporary registers

- 父函数保证：子函数能随便使用temporary registers (x5-x7, x28-x31)，返回给父函数的时候，x5-x7, x28-x31的值可以被改变。
- 子函数保证：返回给父函数的时候，saved registers(x18-x27) 保持父函数调用子函数前的值。

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Nested Procedure

- Example Compiling a recursive procedure that computes n!, suppose argument n is in x10, and results in x10

```
long long   fact ( long long   n )
{
      if ( n  <  1 )   return  ( 1 ) ;
          else  return   ( n  *  fact ( n  -  1 ) ) ;
}
```

- RISC-V assembly code

```
fact:   addi    sp, sp, 16          // adjust stack for 2 items
        sd      x1, 8(sp)           // save the return address
        sd      x10, 0(sp)          // save the argument  n
        addi    x5, x10, -1         // x5 = n  -  1
        bge     x5, x0, L1          // if  n  >=  1, go to L1(else)
        addi    x10, x0, 1          // return 1 if n <1
        addi    sp, sp, 16          // Recover sp (Why not recover x1and x10 ?)
        jalr    x0, 0(x1)           // return to caller
```

```
L1:  addi   x10, x10, -1        // n >= 1: argument gets ( n - 1 )
      jal   x1, fact             // call fact with ( n - 1 )
     add    x6, x10, x0
      ld    x10, 0(sp)           // restore argument n
      ld    x1, 8(sp)            // restore the return address
      addi  sp, sp, 16           // adjust stack pointer to pop 2 items
      mul   x10, x10, x6         // return n*fact ( n - 1 )
      jalr  x0,  0(x1)           // return to the caller
```

- Why x10 is saved? Why x1 is saved?
- Preserved things across a procedure call
  Saved registers, stack pointer register(sp),
  return address register(x1), stack above the stack pointer
- Not preserved things across a procedure call
  Temporary registers, argument registers(x10 ~ x17),
  return value registers (x10 ~ x17), stack below the stack pointer

Disadvantages of recursion

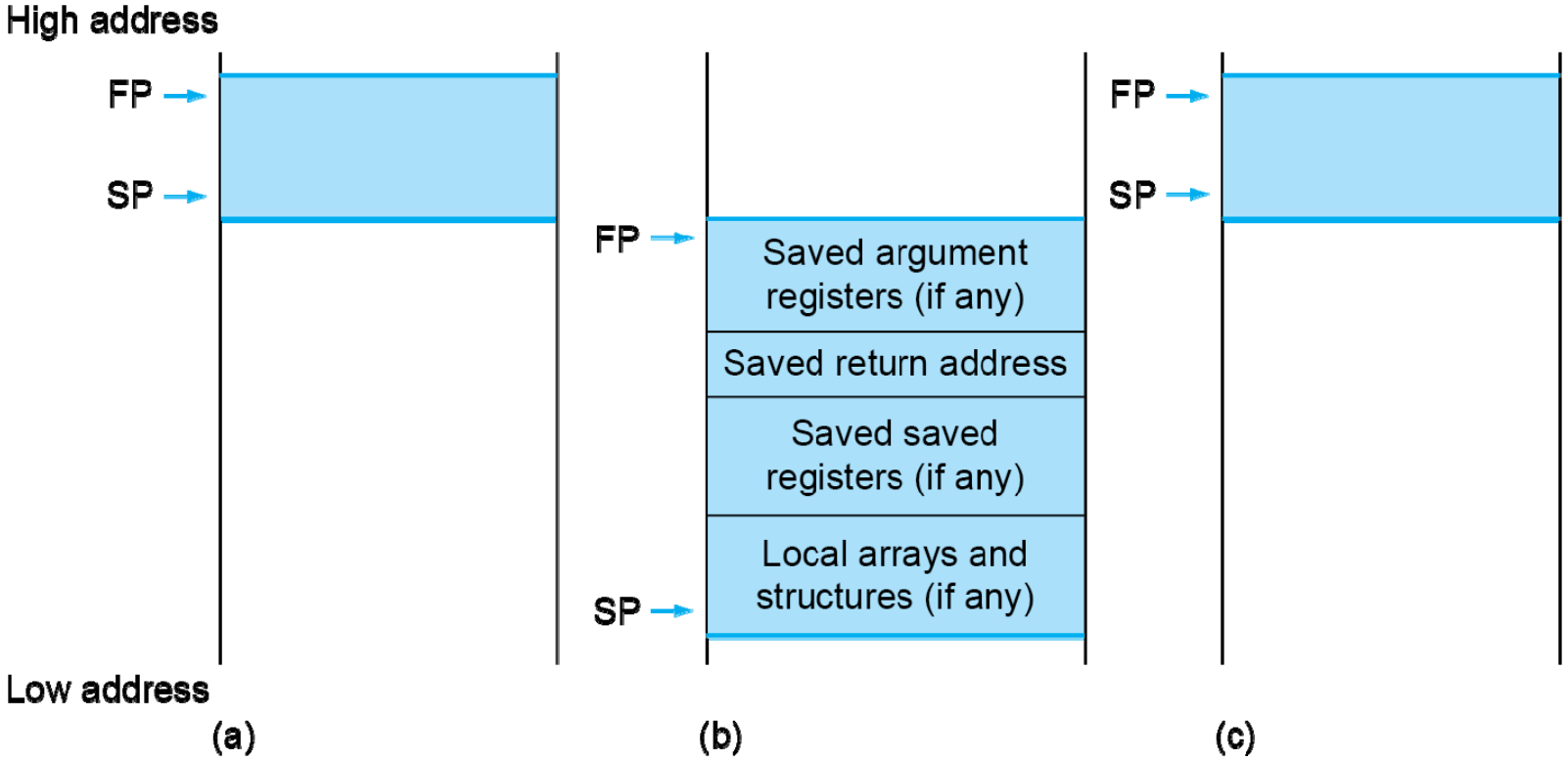
- Use too much resource，to protect the processor status，recursion may result in stack overflow.
- Need push and pop，takes a lot of memory space leading to inefficient usage of memory.
- How to avoid？ use loop instead of recursion （tail call）.

What is and what is not preserved across a procedure call

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

- Storage class of C variables
  - *automatic*
  - *static*
- Procedure frame and frame pointer (x8 or fp)
  - The importance of fp
  - *automatic*
- Global pointer (x3 or gp)
  - *static*

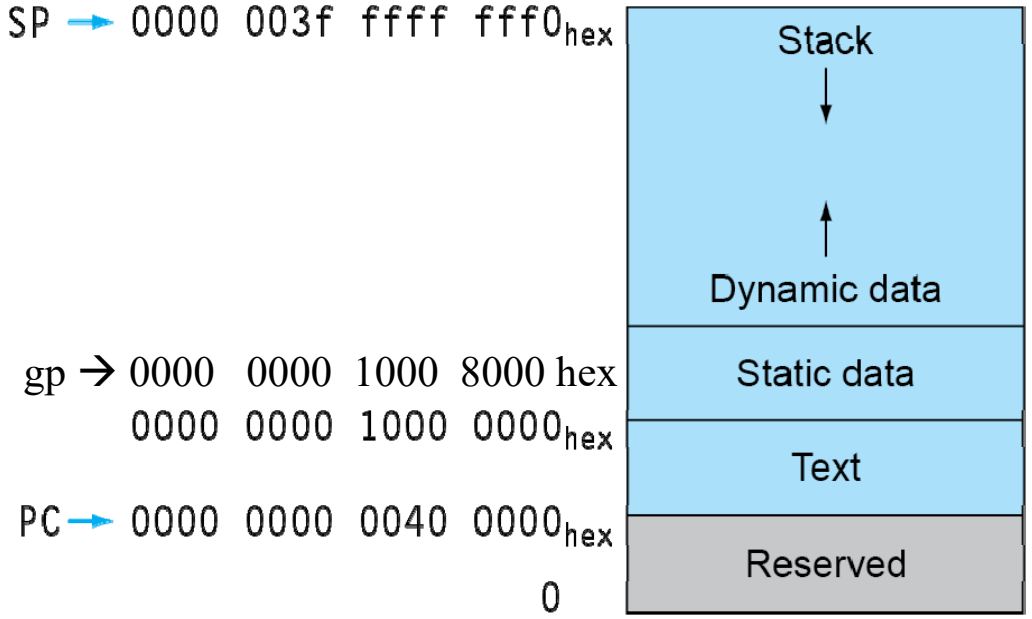
Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - x3 (global pointer) initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



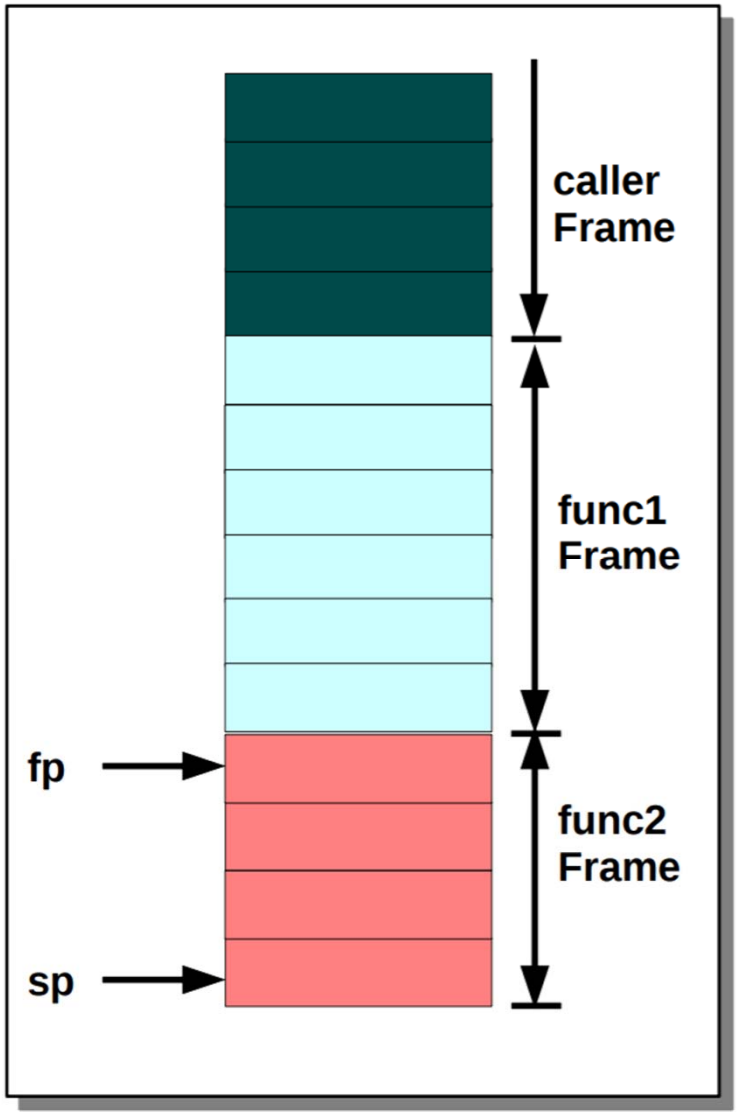
RISC-V register conventions

Name	Register name	Usage	Preserved On call?
x0	0	The constant value 0	n.a.
x1(ra)	1	Return address(link register)	yes
x2(sp)	2	Stack pointer	yes
x3(gp)	3	Global pointer	yes
x4(tp)	4	Thread pointer	yes
x5-x7	5-7	Temporaries	no
x8-x9	8-9	Saved	yes
x10-x17	10-17	Arguments/results	no
x18-x27	18-27	Saved	yes
x28-x31	28-31	Temporaries	no

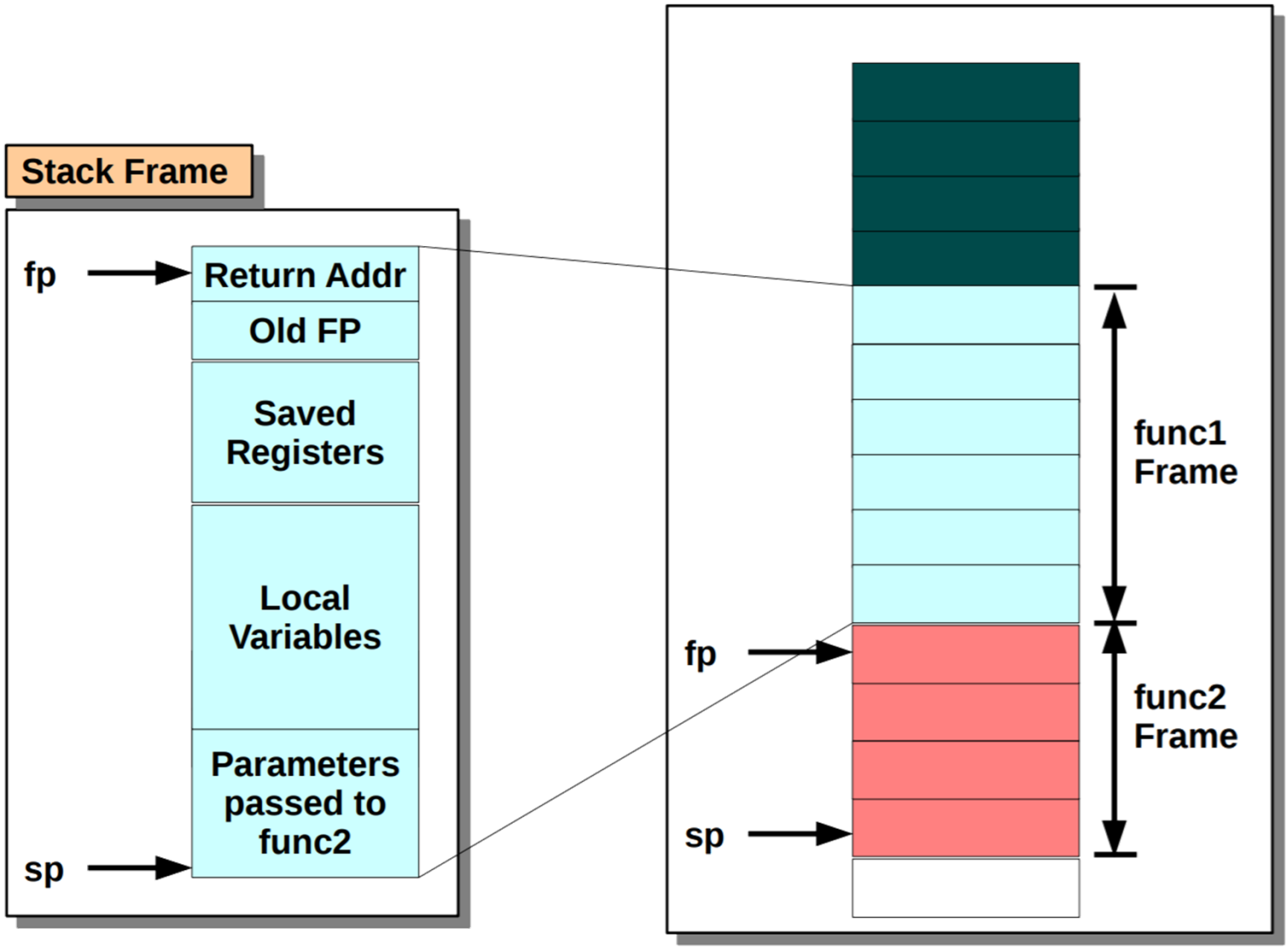
Stack Frame

- Private space for a subroutine allocated on entry and deallocated on exit

- Identified by a ***Frame Pointer*** (fp (x8))



Stack Frame


Stack Frame
fp
Return Addr
Old FP
Saved Registers
Local Variables
Parameters passed to func2
sp
fp
sp
func1 Frame
func2 Frame

Stack Overflow

- Smashing The Stack For Fun And Profit
  - http://phrack.org/issues/49/14.html

Title : Smashing The Stack For Fun And Profit

Author : Aleph1

.o0 Phrack 49 0o.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

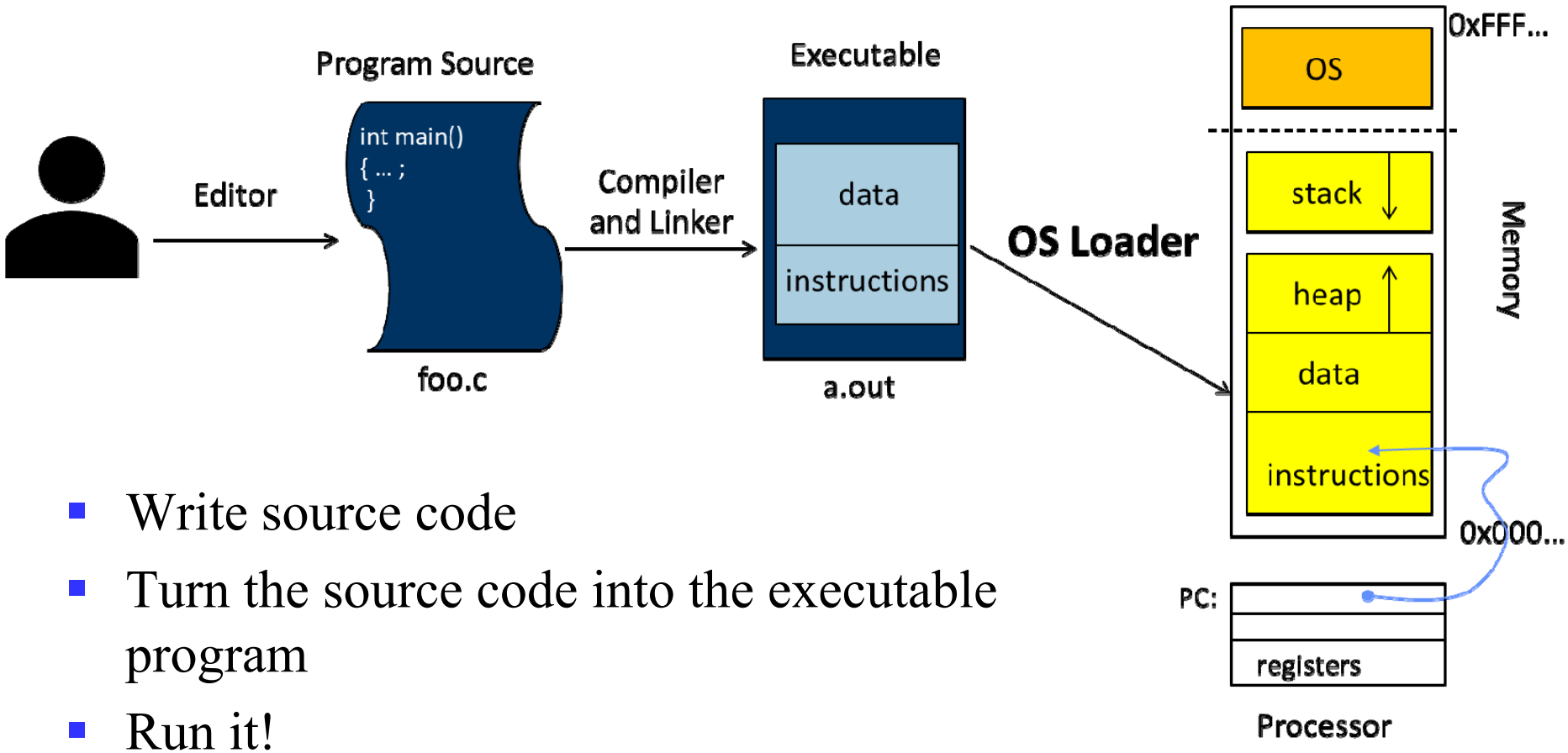
by Aleph One
aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations
it is possible to corrupt the execution stack by writing past
the end of an array declared auto in a routine. Code that does
this is said to smash the stack, and can cause return from the
routine to jump to a random address. This can produce some of
the most insidious data-dependent bugs known to mankind.
Variants include trash the stack, scribble the stack, mangle
the stack; the term mung the stack is not used, as this is
never done intentionally. See spam; see also alias bug,
fandango on core, memory leak, precedence lossage, overrun screw.

Overview

- RISC-V ISA
- RISC-V Assembly Language
  - Some basic concepts
  - From source code to a running program

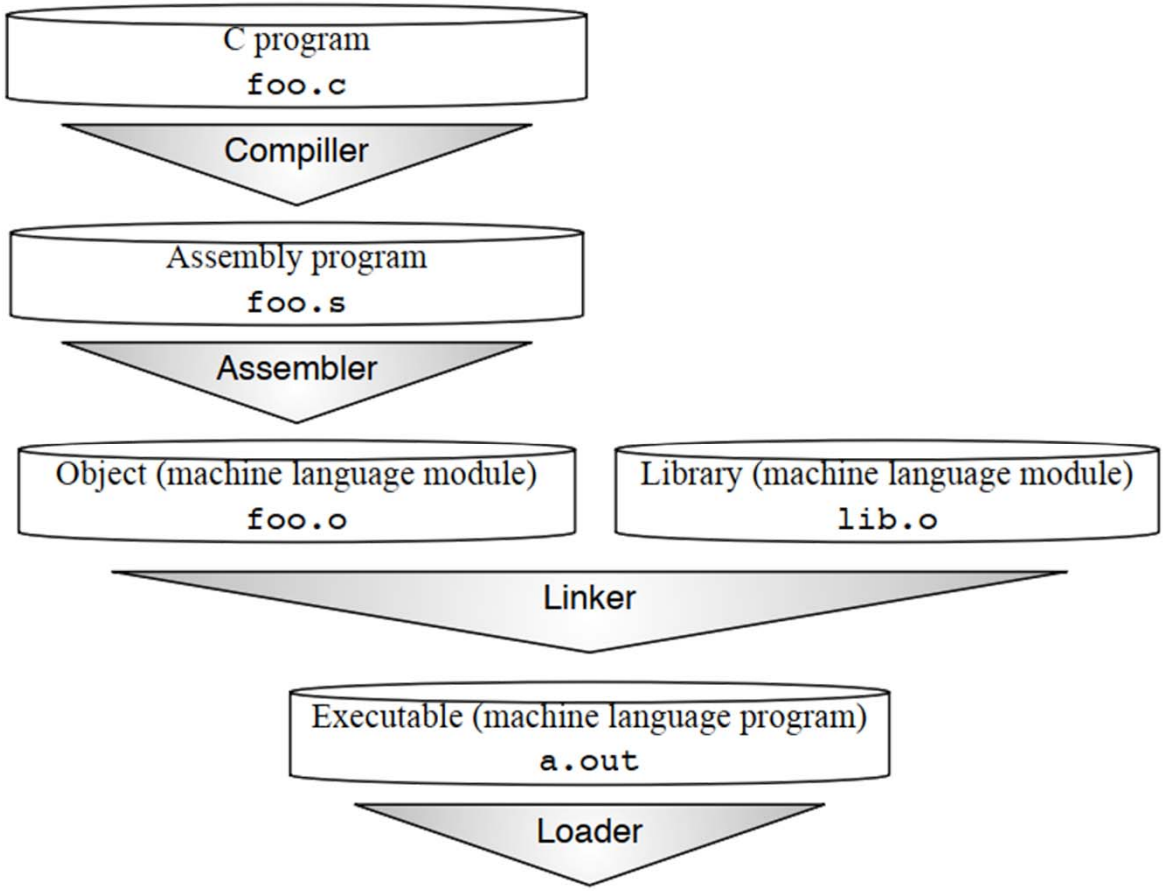
From Source Code to A Running Program



- Write source code
- Turn the source code into the executable program
- Run it!

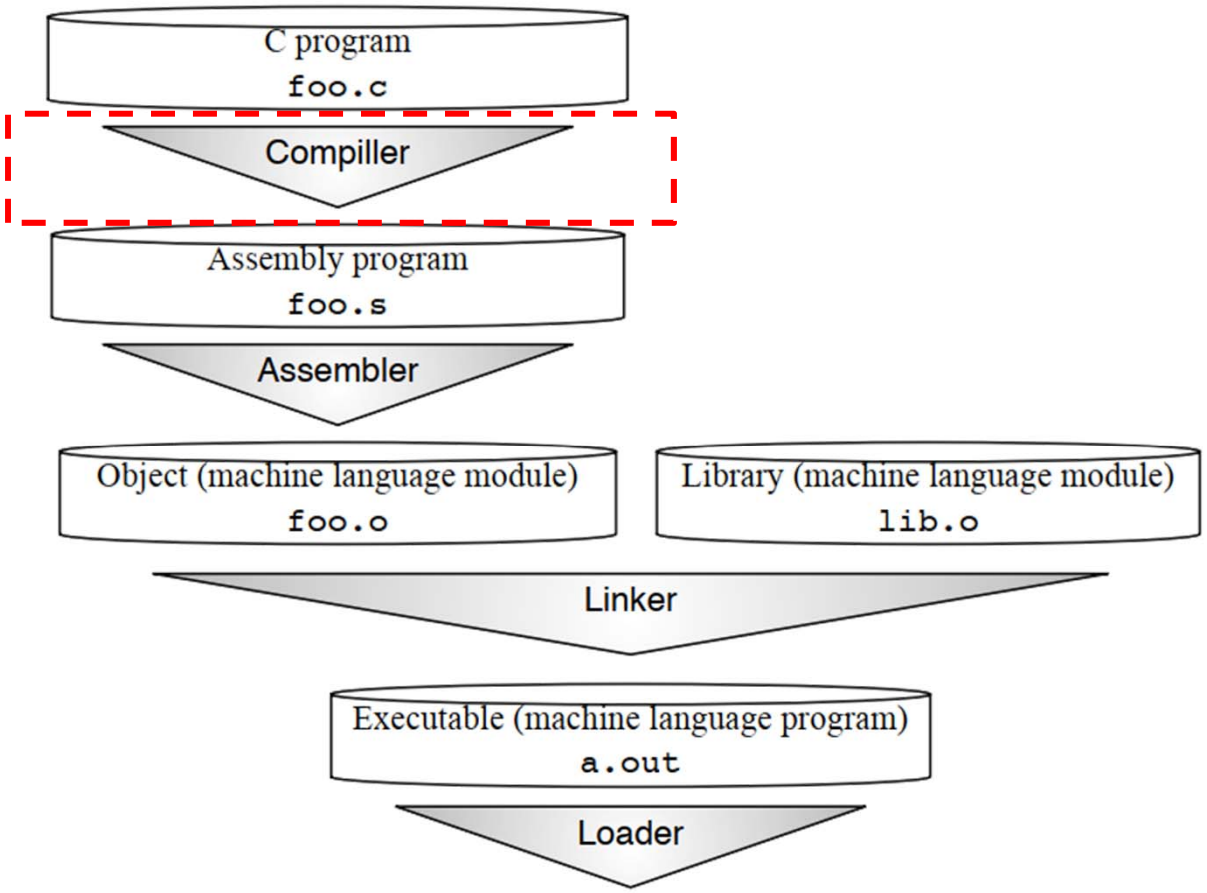
From C Source Code to A Running Program

- Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation.



Compiler: *.c -> *.s

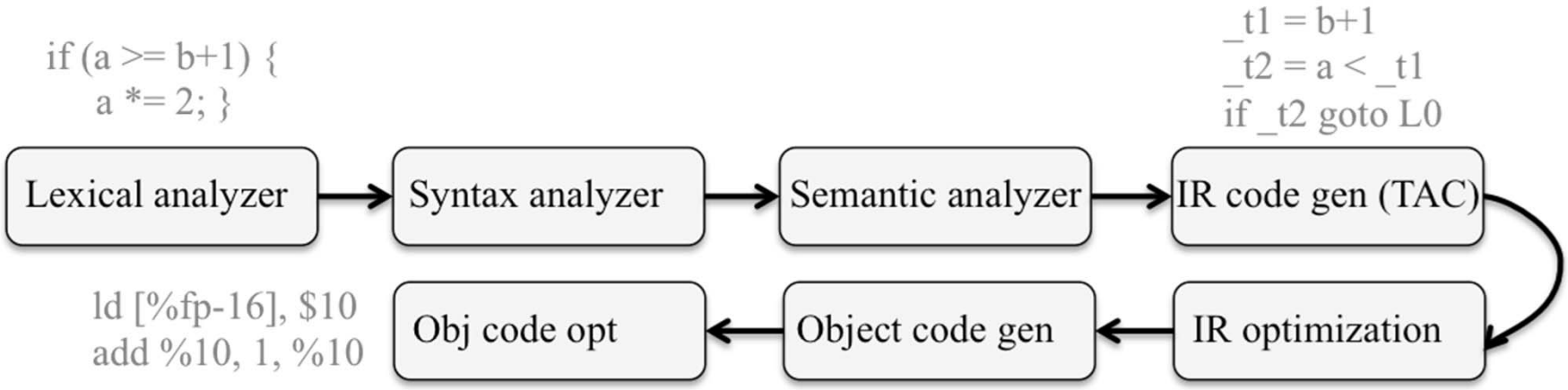
- Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation.



Compiler

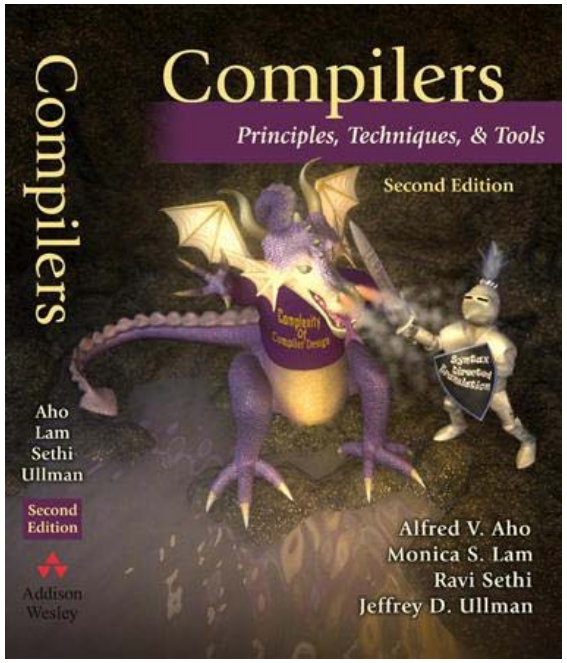
- Preprocessor (*.c -> *i)
  - Expands all macro definitions and include statements (and anything else starting with a #) and passes the result to the actual compiler.

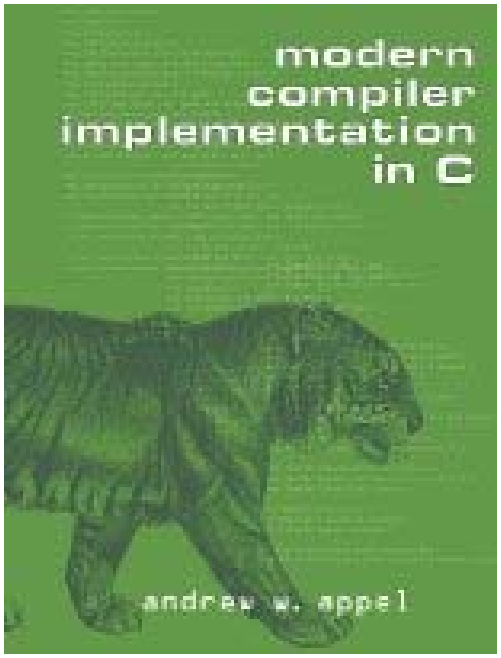
- Compiler (*.i -> *.s)

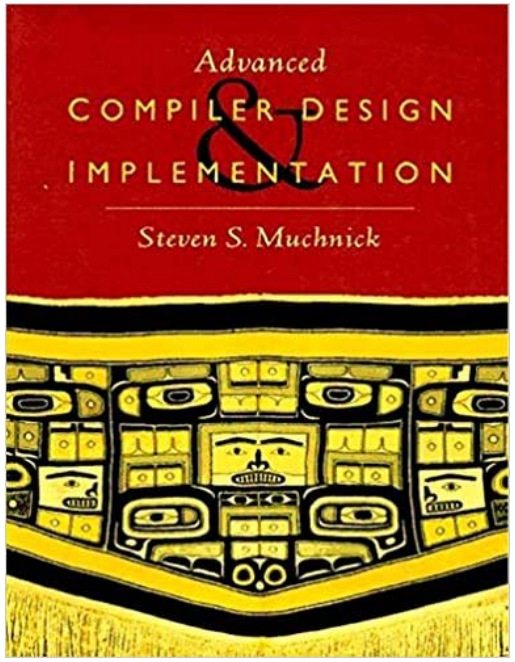


https://people.cs.pitt.edu/~xianeizhang/notes/Linking.html

NOT Be Detailed in This Class…



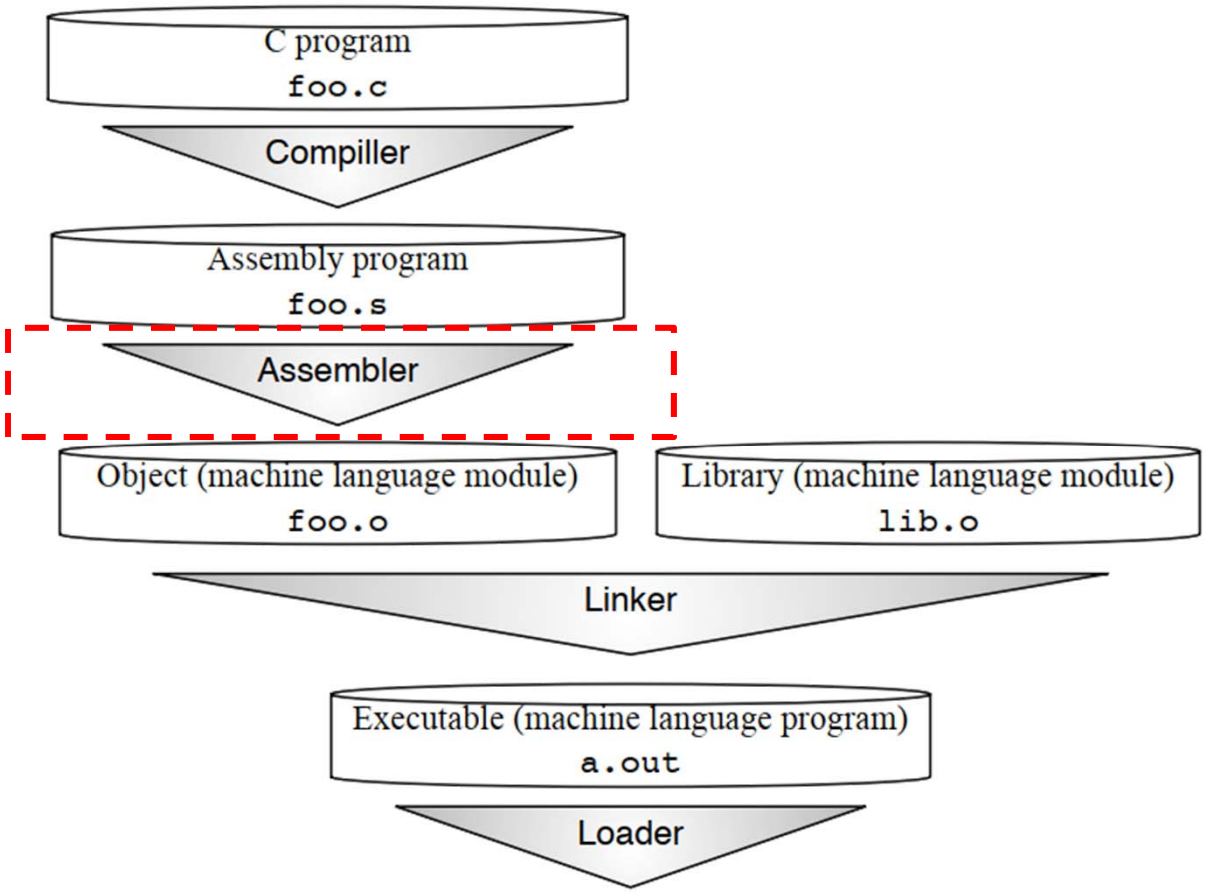




Because You Deserve More!

Assembler: *.s -> *.o

- Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation.



Assembler

- Not simply to produce object code from the instructions that the processor understands, but to extend them to include operations useful for the assembly language programmer or the compiler writer. This category, based on clever configurations of regular instructions, is called ***pseudo-instructions***.

Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi x0, x0, 0`	No operation
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`j offset`	`jal x0, offset`	Jump
`jr rs`	`jalr x0, rs, 0`	Jump register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine

Assembler

Pseudoinstruction	Base Instruction(s)	Meaning
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`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
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`jal offset`	`jal x1, offset`	Jump and link
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine

“Hello World” in C

```
#include <stdio.h>
int main()
{
    printf("Hello, %s\n", "world");
    return 0;
}
```

Input: in Assembly Language (*.s)

```
  .text                     # Directive: enter text section
  .align 2                  # Directive: align code to 2^2 bytes
  .globl main               # Directive: declare global symbol main
main:                       # label for start of main
  addi sp,sp,-16            # allocate stack frame
  sw   ra,12(sp)            # save return address
  lui  a0,%hi(string1)      # compute address of
  addi a0,a0,%lo(string1)   #   string1
  lui  a1,%hi(string2)      # compute address of
  addi a1,a1,%lo(string2)   #   string2
  call printf               # call function printf
  lw   ra,12(sp)            # restore return address
  addi sp,sp,16             # deallocate stack frame
  li   a0,0                 # load return value 0
  ret                       # return
  .section .rodata          # Directive: enter read-only data section
  .balign 4                 # Directive: align data section to 4 bytes
string1:                    # label for first string
  .string "Hello, %s!\n"    # Directive: null-terminated string
string2:                    # label for second string
  .string "world"           # Directive: null-terminated string
```

Assembler Directives

- `.text`—Enter text section.
- `.align 2`—Align following code to 2^2 bytes.
- `.globl main`—Declare global symbol "main".
- `.section .rodata`—Enter read-only data section.
- `.balign 4`—Align data section to 4 bytes.
- `.string "Hello, %s!\n"`—Create this null-terminated string.
- `.string "world"`—Create this null-terminated string.

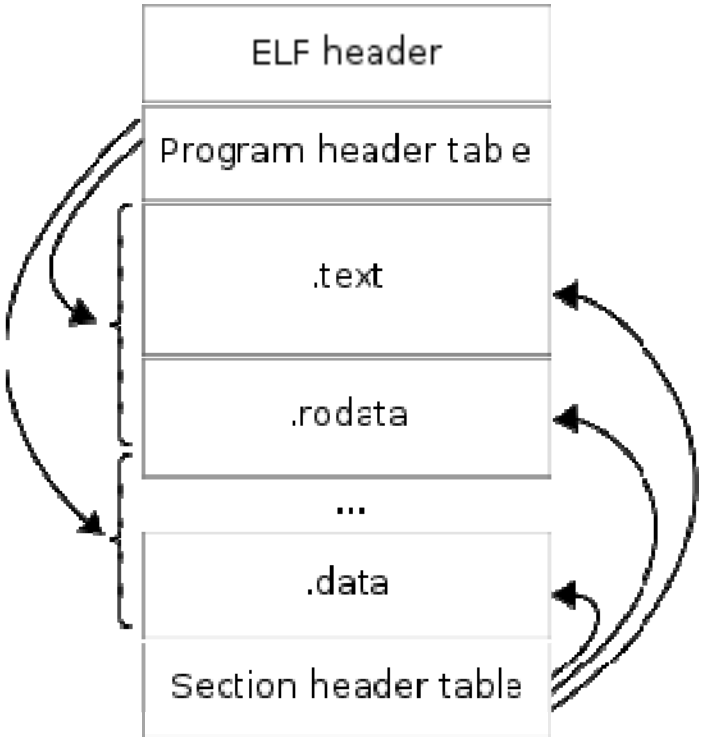
Output: in RISC-V Machine Language (*.o)

- The assembler produces the object using the **Executable and Linkable Format** (ELF, formerly named ***Extensible Linking Format***) standard format.

```
00000000 <main>:
   0: ff010113   addi   sp,sp,-16
   4: 00112623   sw     ra,12(sp)
   8: 00000537   lui    a0,0x0
   c: 00050513   mv     a0,a0
  10: 000005b7   lui    a1,0x0
  14: 00058593   mv     a1,a1
  18: 00000097   auipc  ra,0x0
  1c: 000080e7   jalr   ra
  20: 00c12083   lw     ra,12(sp)
  24: 01010113   addi   sp,sp,16
  28: 00000513   li     a0,0
  2c: 00008067   ret
```

Executable and Linkable Format (ELF)

- ELF is a common standard file format for executable files, object code, shared libraries, and core dumps.
- The standard binary file format for Unix and Unix-like systems on x86 processors by the 86open project.
- By design, the ELF format is flexible, extensible, and cross-platform. This has allowed it to be adopted by many different operating systems on many different hardware platforms.

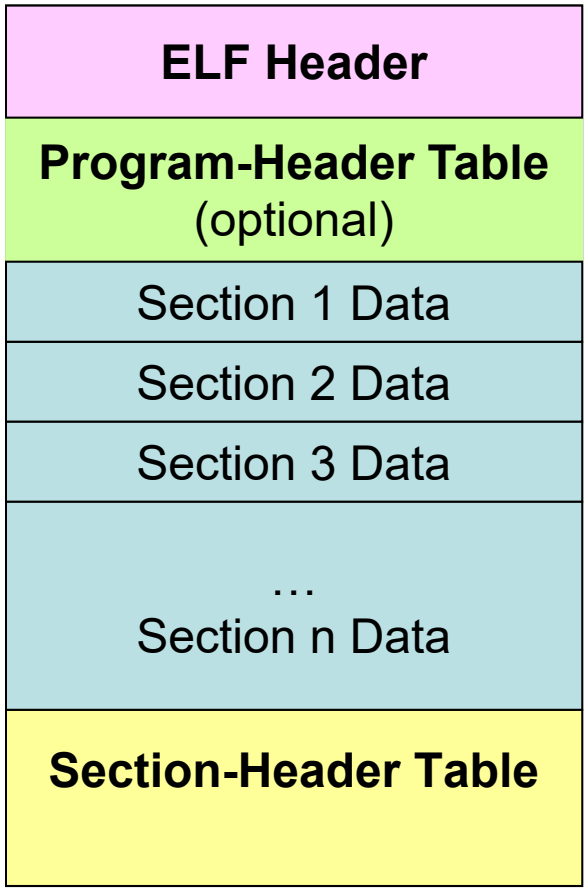


An ELF file has two views: the program header shows the *segments* used at run time, whereas the section header lists the set of *sections*.

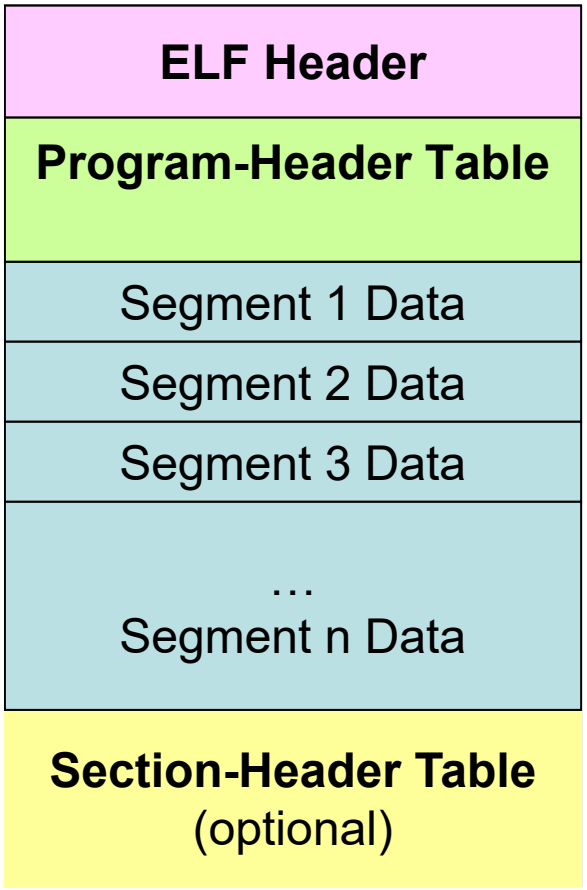
ELF Object Files

- Created by the assembler and the linker, object files are binary representations of programs intended to be executed directly on a processor.
  - Programs that require other abstract machines, such as shell scripts, are excluded.
- There are three main types of object files:
  - A **relocatable** file holds code and data suitable for **linking** with other object files to create an executable or a shared object file.
  - An **executable** file holds a program suitable for **execution**; the file specifies how ***exec*** creates a program's process image.
  - A **shared object** file holds code and data suitable for linking in two contexts. First, the linker processes the shared object file with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

Relocatable vs. Executable



Relocatable File



Executable File

Relocatable Object File

- As assembler's output
  - Binary machine code, but **NOT executable**
    - E.g., .o for Linux, while .obj for Windows
  - May refer to external symbols
    - Need a symbol table
  - Each object file has its own address space
    - Addresses will need to be fixed later

Format of Relocatable Object File

- Header
  - Size and position of pieces of file
- Text Section
  - Instructions
- Data Section
  - Static data (local/global vars, strings, constants)
- Debugging Information
  - Line number -> code address map, etc.
- Symbol Table
  - External (exported) references
  - Unresolved (imported) references

Commonly Used Sections

- Text (.section .text)
  - Contain code (instructions)
- Read-Only Data (.section .rodata)
  - Contains pre-initialized constants
- Read-Write Data (.section .data)
  - Contains pre-initialized variables
- BSS (.section .bss)
  - Contains un-initialized data
  - http://www.faqs.org/faqs/unix-faq/faq/part1/section-3.html

- Useful Tools: **Objdump** & **Readelf**

Object file

- object file header—size and position of the other pieces
- Text segment
- static data segment and dynamic data
- The relocation information ---- identifies absolute addresses of instruction and data words when the program is loaded into memory
- symbol table
- Debugging information

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	instruction	
	0	ld x10, 0(gp)	
	4	jal x1, 0	
	……	--	
Data segment	0	(X)	
	……	……	
Relocation information	Address	Instruction type	Dependency
	0	ld	X
	4	jal	B
Symbol table	label	Address	
	X	--	
	B	--	

Symbols and References

- Global labels: Externally visible “exported” symbols
  - Can be referenced from other object files
  - Exported functions, global variables

- Local labels: Internal visible only symbols
  - Only used within this object file
  - Static functions, static variables, loop labels, …

Issues Need to be Resolved

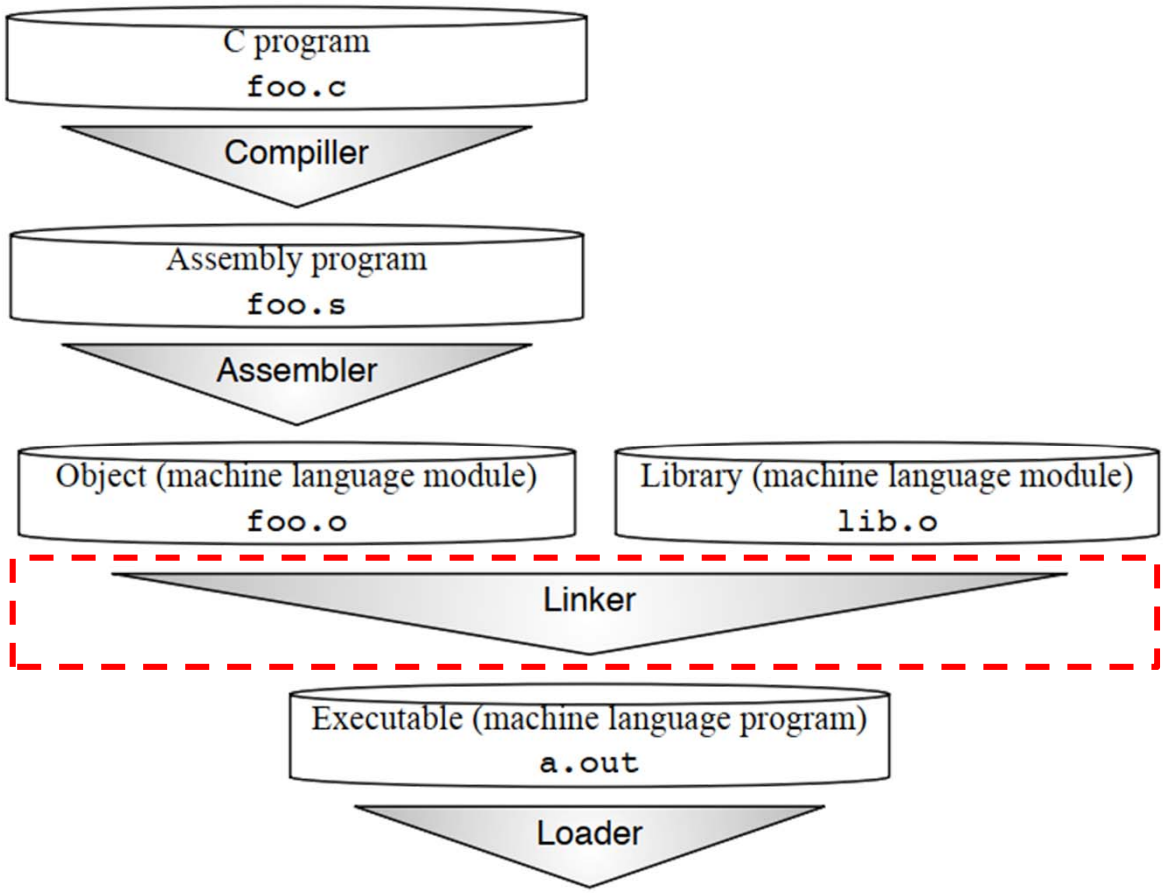
- How does the Assembler resolve local references?
- How does the Assembler resolve external references?

How to Resolve Local References?

- Handle forward references
  - Two-pass assembly
    - Do a pass through the whole program, allocate instructions and lay out data, thus determining addresses
    - Do a second pass, emitting instructions and data, with the correct label offsets now determined
  - One-pass (or backpatch) assembly
    - Do a pass through the whole program, emitting instructions, emit a 0 for jumps to labels not yet determined, keep track of where these instructions are
    - Backpatch, fill in 0 offsets as labels are defined

Linker: (multiple) *.o -> a.out

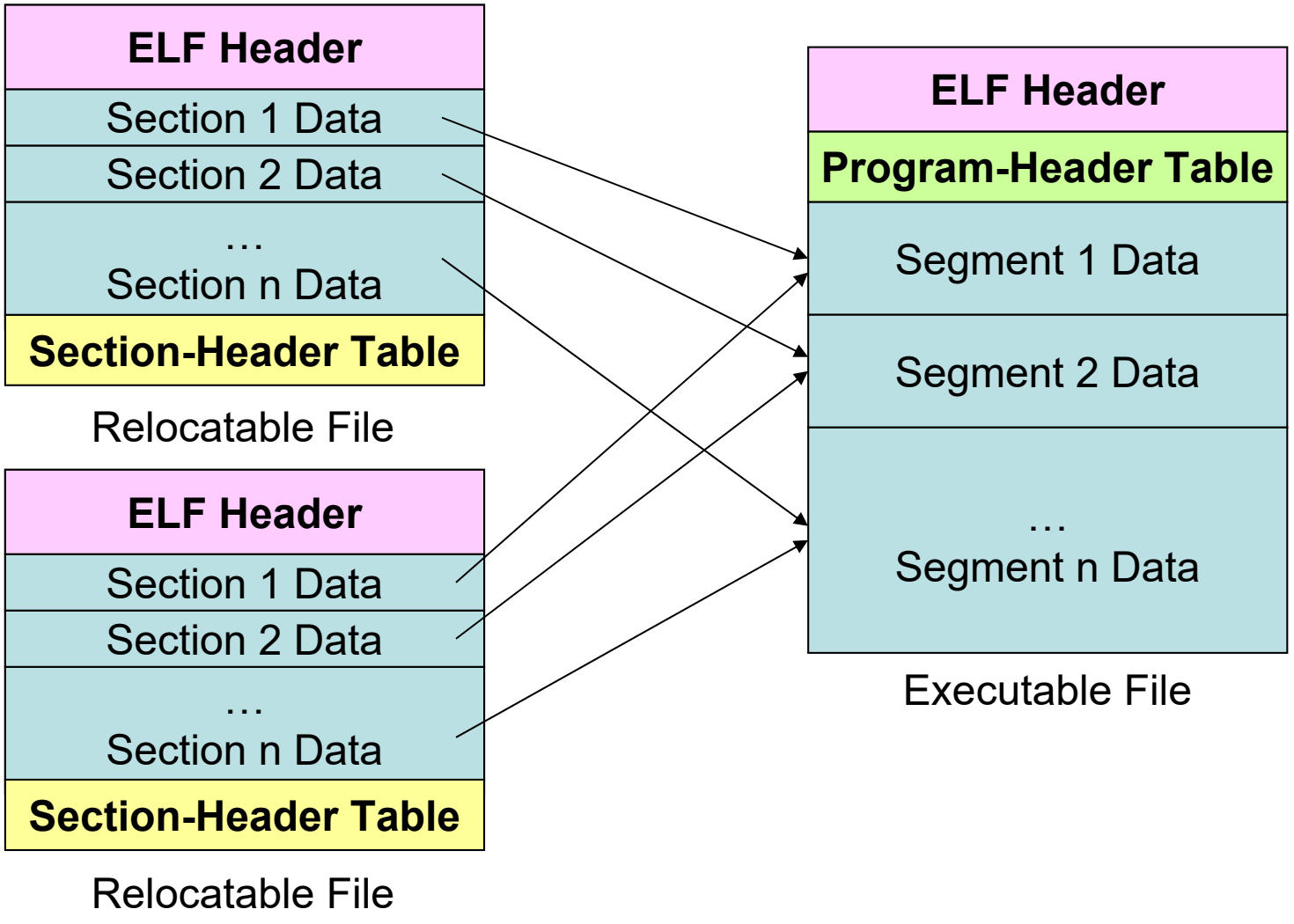
- Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation.



Linker

- Rather than compile all the source code every time one file changes, the linker allows individual files to be compiled and assembled separately.
- Why separate compile/assemble and linking steps?
  - Separately compiling modules and linking them together obviates the need to recompile the whole program every time something changes
    - Need to just recompile a small module
    - A linker coalesces object files together to create a complete program
- The linker “stitches” the new object code together with existing machine language modules, such as libraries.

Role of Linker



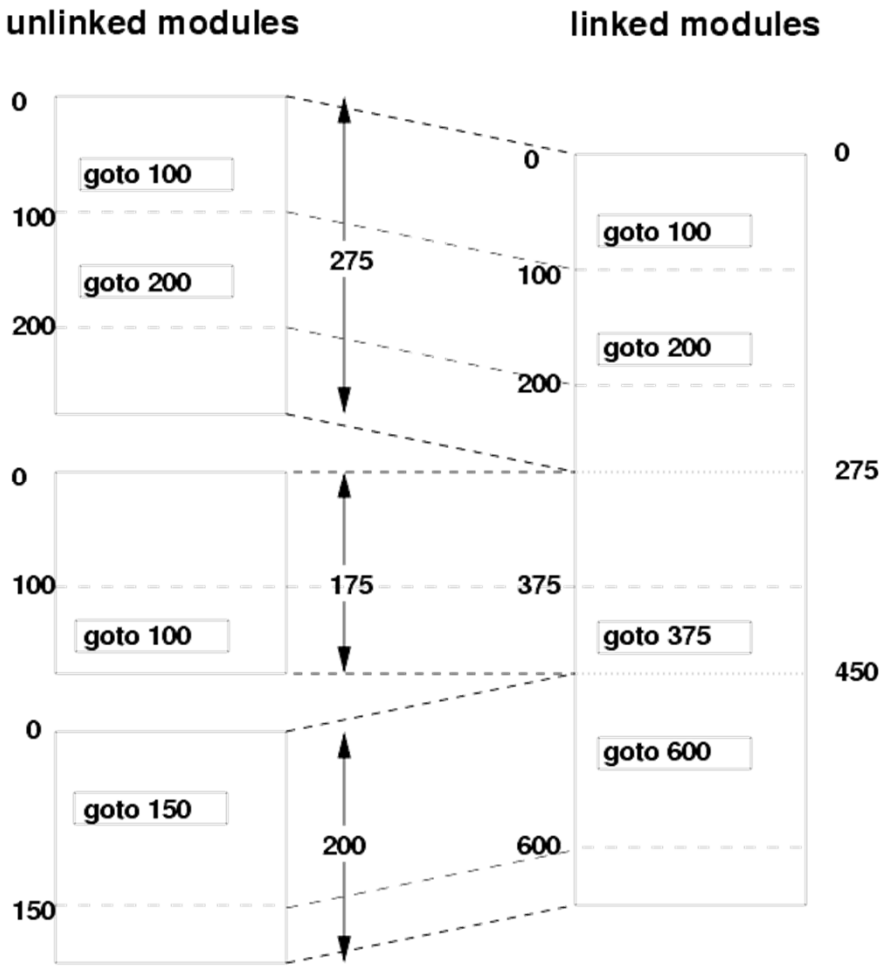
Issues Need to be Resolved

- How does the linker combine separately compiled files?
- How does linker resolve unresolved references?
- How does linker relocate data and code segments

- To combine object files into an executable file
  - Relocate each object's text and data segments
  - Resolve as-yet-unresolved symbols
  - Record top-level entry point in executable file

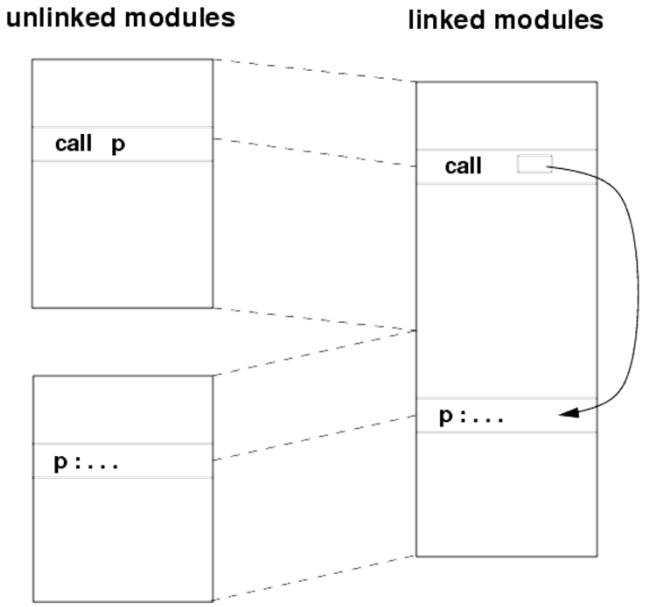
Linker Functions 1: Fixing Addresses

- Addresses in an object file are usually relative to the start of the code or data segment in that file.
- When different object files are combined:
  - The same kind of segments (text, data, read-only data, etc.) from the different object files get merged.
  - Addresses have to be “fixed up” to account for this merging.
  - The fixing up is done by the linker, using information embedded in the executable for this purpose (“relocations”).



Linker Function 2: Symbol Resolution

- Suppose:
  - Module B defines a symbol x;
  - Module A refers to x.
- The linker must:
  - Determine the location of x in the object module obtained from merging A and B; and
  - Modify references to x (in both A and B) to refer to this location.



- Each linkable module contains a symbol table, whose contents include:
  - Global symbols defined (maybe referenced) in the module.
  - Global symbols referenced but not defined in the module (these are generally called externals).
  - Segment names (e.g., text, data, rodata).
  - These are usually considered to be global symbols defined to be at the beginning of the segment.
  - Non-global symbols and line number information (optional), for debuggers.

Linking Object modules

- Object modules(including library routine) → **executable program**
- 3 step of Link
  - Place code and data modules symbolically in memory
  - Determine the addresses of data and instruction labels
  - Patch both the internal and external references (**Address of invoke**)
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`ld x10, 0(x3)`	
	4	`jal x1, 0`	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`ld`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sd x11, 0(x3)`	
	4	`jal x1, 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`sd`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0000\ 0000\ 0040\ 0000_{hex}$	`ld x10, 0(x3)`
	$0000\ 0000\ 0040\ 0004_{hex}$	`jal x1,` 252_{ten}
	...	...
	$0000\ 0000\ 0040\ 0100_{hex}$	`sd x11, 32(x3)`
	$0000\ 0000\ 0040\ 0104_{hex}$	`jal x1,` -260_{ten}
	...	...
Data segment	Address	
	$0000\ 0000\ 1000\ 0000_{hex}$	(X)
	...	...
	$0000\ 0000\ 1000\ 0020_{hex}$	(Y)
	...	...

Format of Executable Object File

- Header
  - **Location of main entry point**
- Text Segment
  - Instructions
- Data Segment
  - Static data (local/global vars, strings, constants)
- **Relocation Information**
  - Instructions and data that depend on actual addresses
  - Linker patches these bits after relocating segments
- Symbol Table
  - Exported and imported references
- Debugging Information

Various Executable Object Files

- Unix/Linux
  - a.out
  - COFF: Common Object File Format
  - ELF: Executable and Linking Format
  - …
- Windows
  - PE: Portable Executable

- All support both executable and other object files

Output: a.out

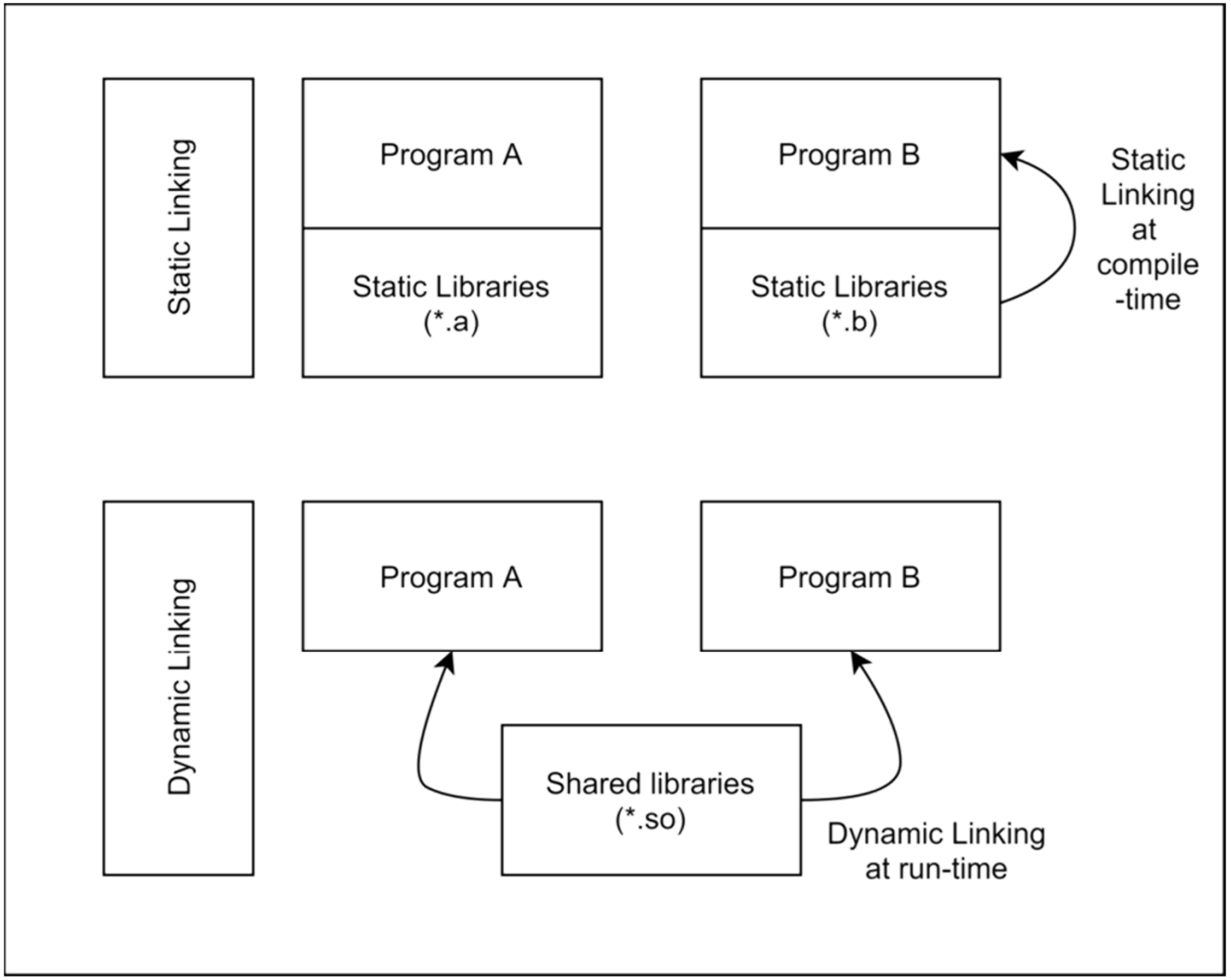
- In addition to the instructions, each object file contains a symbol table that includes all the labels in the program that must be given addresses as part of the linking process. This list includes labels to data as well as to code.

```
000101b0 <main>:
    101b0: ff010113 addi sp,sp,-16
    101b4: 00112623 sw   ra,12(sp)
    101b8: 00021537 lui  a0,0x21
    101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
    101c0: 000215b7 lui  a1,0x21
    101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
    101c8: 288000ef jal  ra,10450 <printf>
    101cc: 00c12083 lw   ra,12(sp)
    101d0: 01010113 addi sp,sp,16
    101d4: 00000513 li   a0,0
    101d8: 00008067 ret
```

Static vs. Dynamic Linking

- Static linking
  - All potential library code is linked and then loaded together ***before execution***.
  - Such libraries can be relatively large, so linking a popular library into multiple programs wastes memory. Moreover, the libraries are bound when linked—even when they are updated later to fix bugs—forcing the statically-linked code to use the old, buggy version.
- Dynamic linking
  - The desired external function is ***loaded and linked to the program only after it is first called***; if it's never called, it's never loaded and linked. Every call after the first one uses a fast link, so the dynamic overhead is only paid once.
  - Each time a program starts it links in the current version of the library functions it needs, which is how it can get the newest version. Furthermore, if multiple programs use the same dynamically linked library, the code in the library need appear only once in memory.
  - Instead of jumping to the real function, it jumps to a short (three-instruction) ***stub function***.

Static vs. Dynamic Linking



Static Libraries

- Static Library: Collection of object files
  (think: like a zip archive)

- Q: Every program contains the entire library?
- A: No, the linker picks only object files needed to resolve undefined references at link time

- E.g., libc.a contains many objects:
  - printf.o, fprintf.o, vprintf.o, sprintf.o, snprintf.o, …
  - read.o, write.o, open.o, close.o, mkdir.o, readdir.o, …
  - rand.o, exit.o, sleep.o, time.o, ….

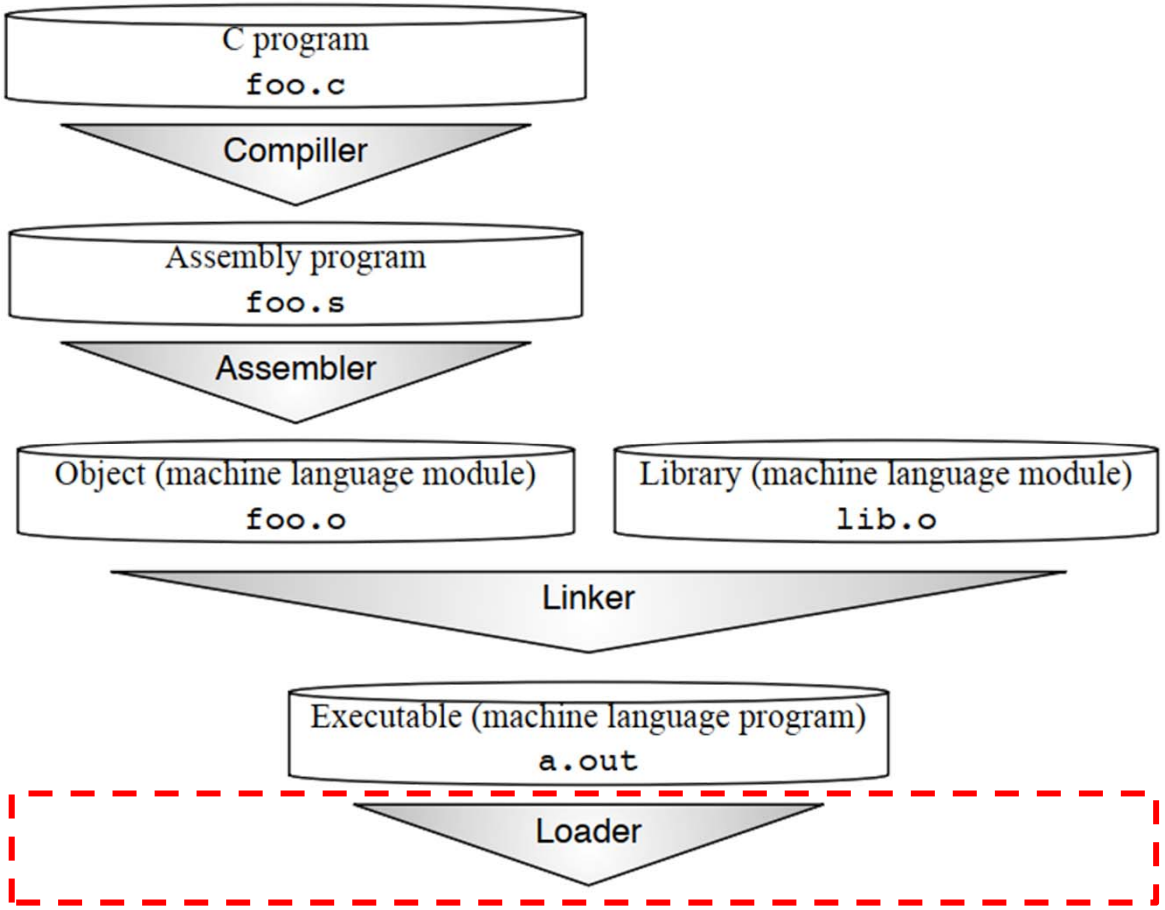
Shared Libraries

- Q: Every program contains parts of same library?
- A: No, they can use shared libraries
  - Executables all point to single shared library on disk
  - **Final linking (and relocations) done by the loader**

- Optimizations:
  - Library compiled at fixed non-zero address
  - Jump table in each program instead of relocations
  - Can even patch jumps *on-the-fly*

Loader: Run a Program

- Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation.

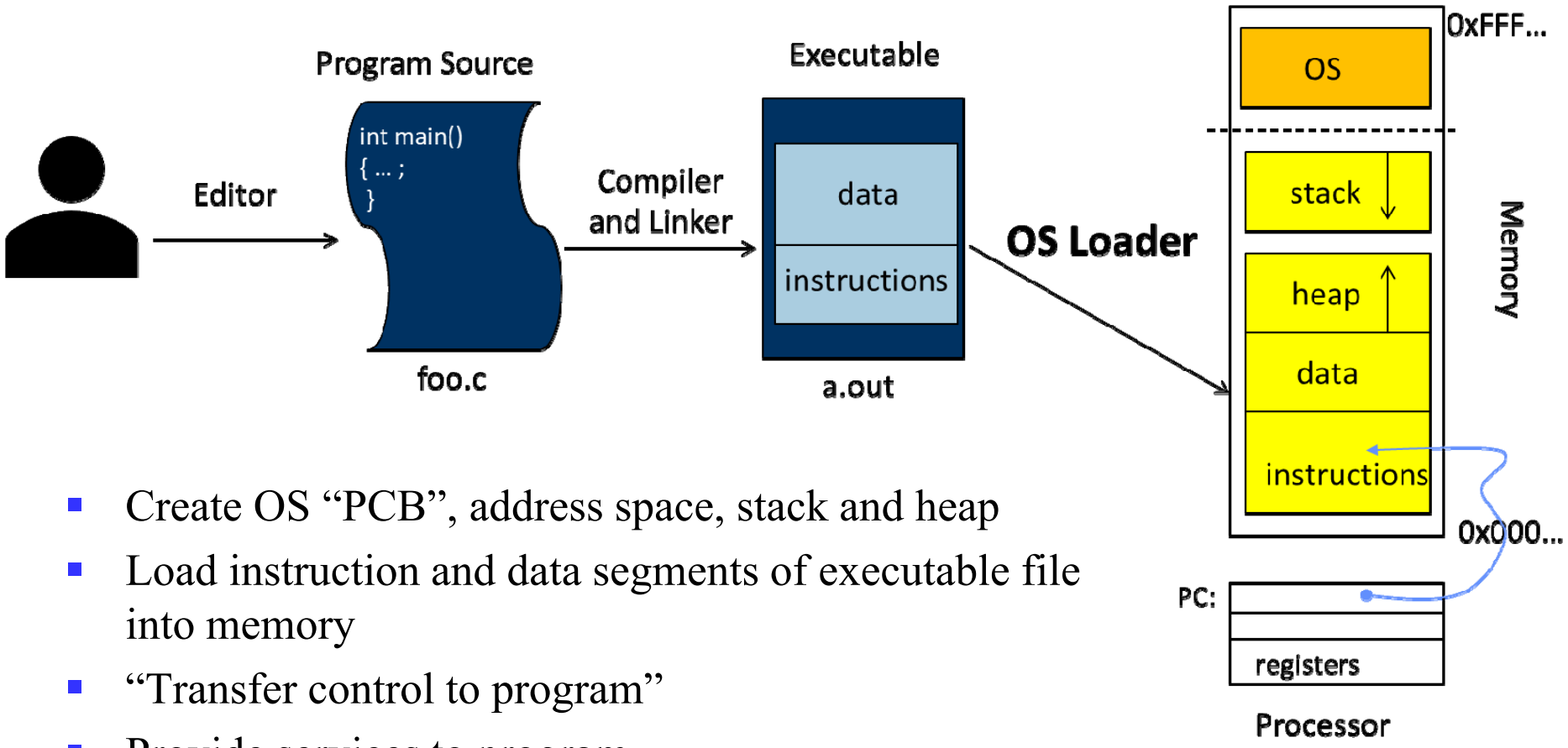


Loader

- A program like the one in a.out is an executable file kept in the computer's storage. When one is to be run, the loader's job is to load it into memory and jump to the starting address.
  - Initializes registers, stack, arguments to first function
  - Jumps to entry-point

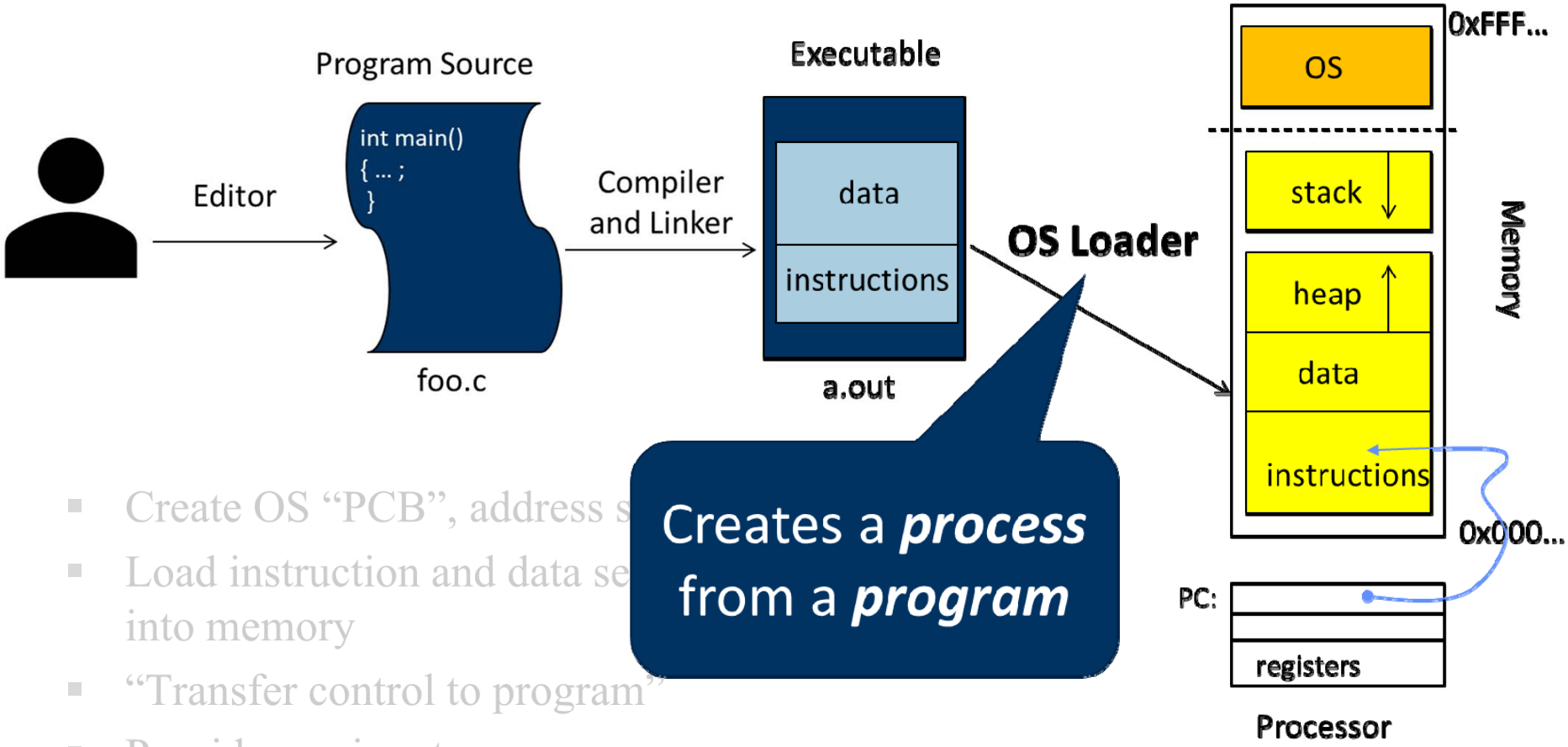
- The "loader" today is the operating system; stated alternatively, loading a.out is one of many tasks of an operating system.

How to Run a Program?



- Create OS "PCB", address space, stack and heap
- Load instruction and data segments of executable file into memory
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

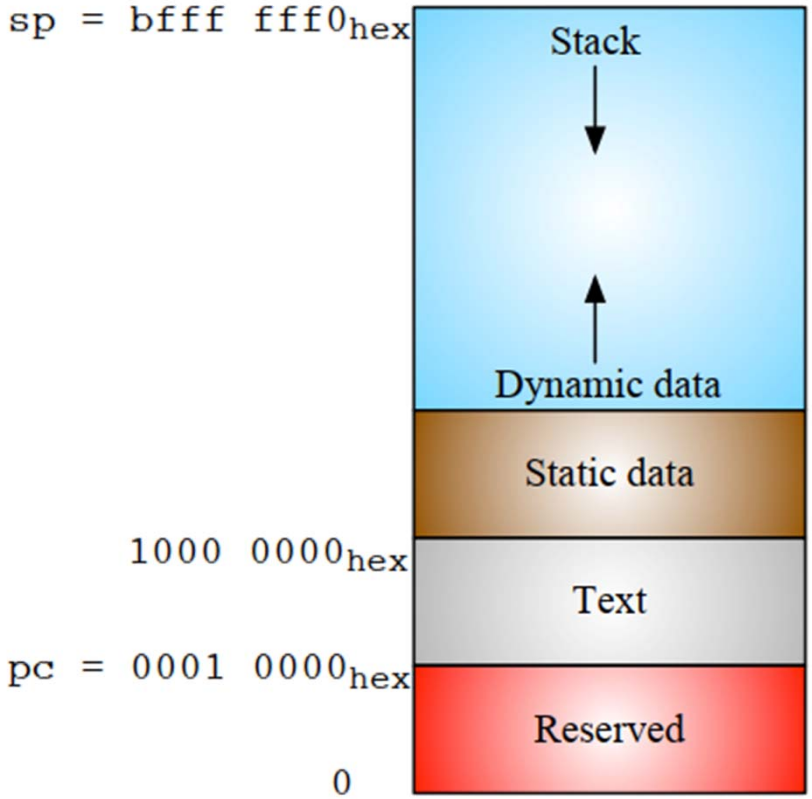
How to Run a Program?



- Create OS "PCB", address s
- Load instruction and data se
  into memory
- "Transfer control to program"
- Provide services to program
- While protecting OS and program

Memory Layout

- RV32I allocation of memory to program and data. The high addresses are the top of the figure, and the low addresses are the bottom.
- In this RISC-V software convention, the stack pointer (sp) starts at 0xbfffffff0 and grows down toward the Static data.
- The text (program code) starts at 0x00010000 and includes the statically-linked libraries.
- The Static data starts immediately above the text region; in this example, we assume that address is 0x10000000.
- Dynamic data, allocated in C by malloc(), is just above the Static data. Called the heap, it grows upward toward the stack. It includes the dynamically-linked libraries.



Loading Statically-linked Programs

- Programs are usually loaded at a fixed address in a fresh address space (so can be linked for that address).
- In such systems, loading involves the following actions:
  - Determine how much address space is needed from the object file header;
  - Allocate that address space;
  - Read the program into the segments in the address space;
  - Zero out any uninitialized data (“.bss” segment) if not done automatically by the virtual memory system.
  - Create a stack segment;
  - Set up any runtime information, e.g., program arguments or environment variables.
  - Start the program executing.

Loading Dynamically-linked Programs

- Loading is a little trickier for dynamically-linked programs. Instead of simply starting the program, the operating system starts ***the dynamic linker***. It in turn starts the desired program, and then handles all first-time external calls, copies the functions into memory, and edits the program after each call to point it to the correct function.
  - GOT & PLT

Position-Independent Code (PIC)

- If the load address for a program is not fixed (e.g., shared libraries), we use position independent code.
- Basic idea: separate code from data; generate code that doesn't depend on where it is loaded.
- PC-relative addressing can give position-independent code references.

PIC (cont'd): ELF Files

- ELF executable file characteristics:
  - Data pages follow code pages;
  - The offset from the code to the data does not depend on where the program is loaded.
- The linker creates a ***global offset table*** (**GOT**) that contains offsets to all global data used.
- If a program can load its own address into a register, it can then use a fixed offset to access the GOT, and thence the data.

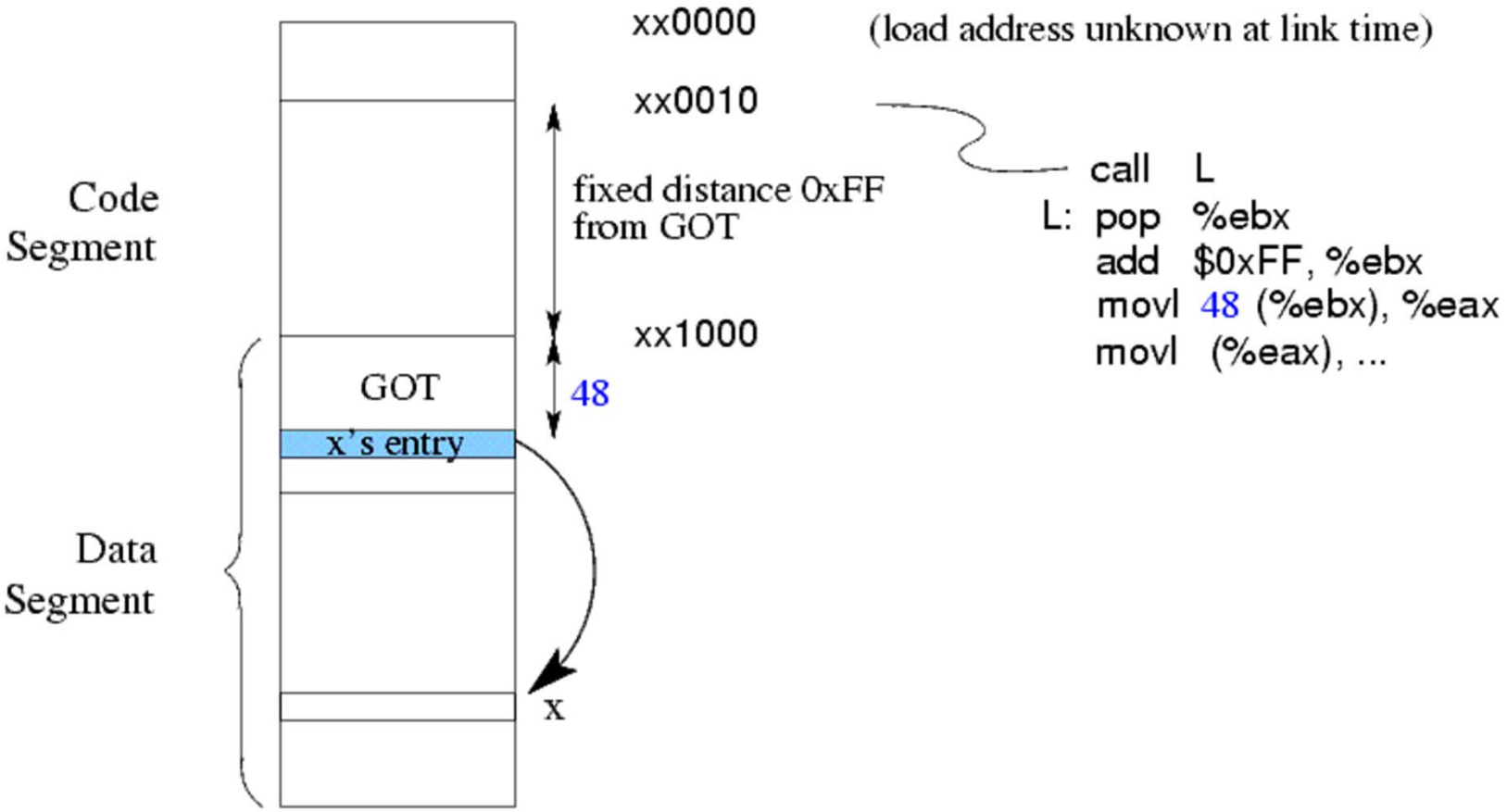
PIC (cont'd): Code on ELF

- Code to figure out its own address (x86):

```
        call  L   /* push address of next instruction on stack */
L:      pop %ebx  /* pop address of this instruction into %ebx */
```

- Accessing a global variable x in PIC:
  - GOT has an entry, say at position k, for x. The dynamic linker fills in the address of x into this entry at load time.
  - Compute "my address" into a register, say %ebx (above);
  - %ebx += offset_to_GOT; /* fixed for a given program */
  - %eax = contents of location k(%ebx) /* %eax = addr. of x */
  - access memory location pointed at by %eax;

PIC on ELF: Example



Based on Linkers and Loaders, by J. R. Levine (Morgan Kaufman, 2000)

Shared Libraries

- Have a single copy of the library that is used by all running programs.
- Saves (disk and memory) space by avoiding replication of library code.
- Virtual memory management in the OS allows different processes to share “read-only” pages, e.g., text and read-only data.
  - This lets us get by with a single physical-memory copy of shared library code.

Shared Libraries: cont'd

- At link time, the linker:
  - Searches a (specified) set of libraries, in some fixed order, to find modules that resolve any undefined external symbols.
  - Puts a list of libraries containing such modules into the executable.
- At load time, the startup code:
  - Finds these libraries;
  - Maps them into the program's address space;
  - Carries out library-specific initialization.
- Startup code may be in the OS, in the executable, or in a special dynamic linker.

Dynamic Linking

- Defers much of the linking process until the program starts running.
- Easier to create, update than statically linked libraries.
- Has higher runtime performance cost than statically linked libraries:
  - Much of the linking process has to be redone each time a program runs.
  - Every dynamically linked symbol has to be looked up in the symbol table and resolved at runtime.

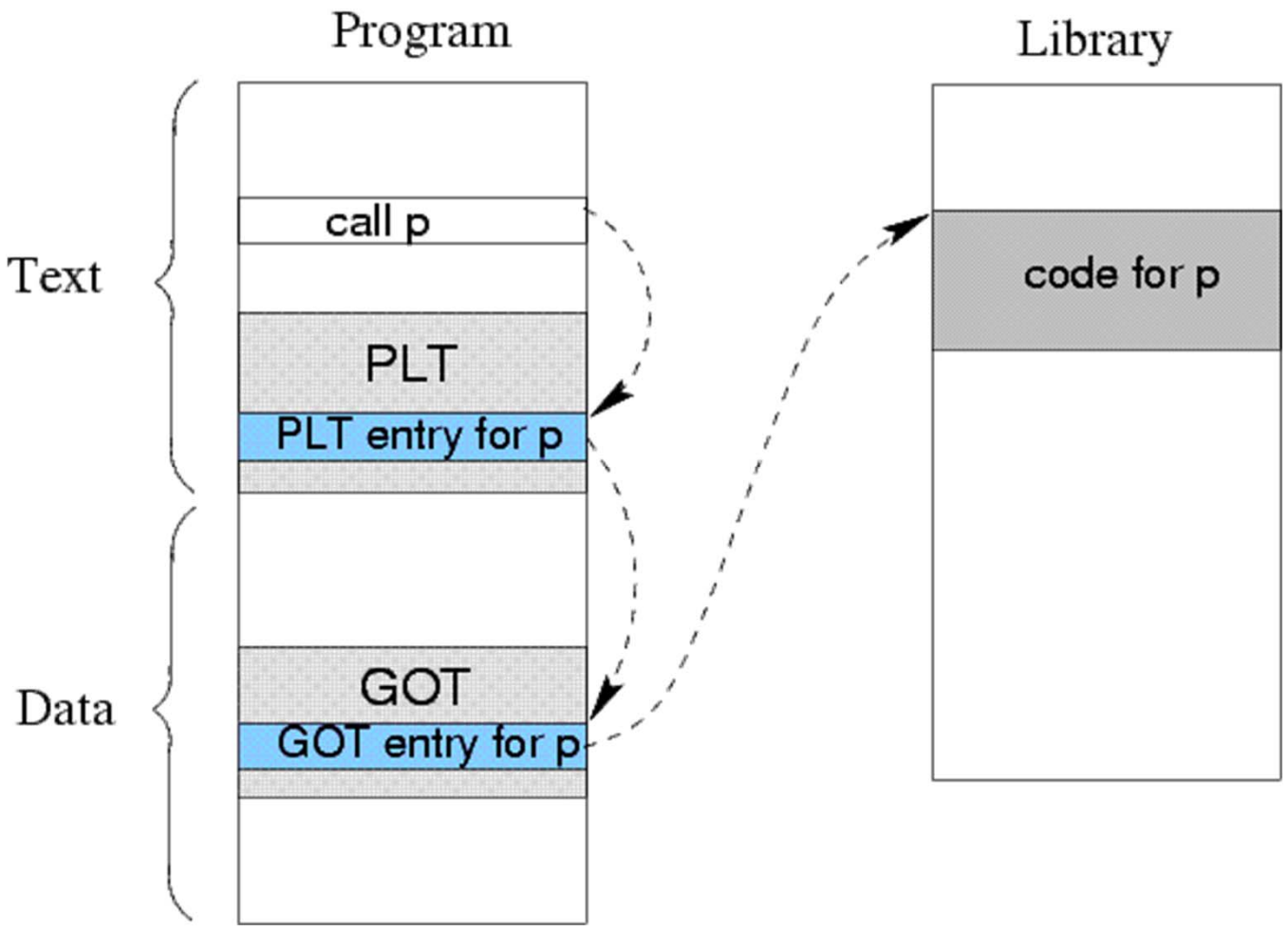
Dynamic Linking: Basic Mechanism

- A reference to a dynamically linked procedure *p* is mapped to code that invokes a *handler*.
- At runtime, when *p* is called, the handler gets executed:
  - The handler checks to see whether p has been loaded already (due to some other reference);
  - If so, the current reference is linked in, and execution continues normally.
  - Otherwise, the code for *p* is loaded and linked in.

Dynamic Linking: ELF Files

- ELF shared libraries use PIC (position independent code), so text sections do not need relocation.
- Data references use a GOT:
  - Each global symbol has a relocatable pointer to it in the GOT;
  - The dynamic linker relocates these pointers.
- We still need to invoke the dynamic linker on the first reference to a dynamically linked procedure.
  - Done using a ***procedure linkage table*** (**PLT**);
  - PLT adds a level of indirection for function calls (analogous to the GOT for data references).

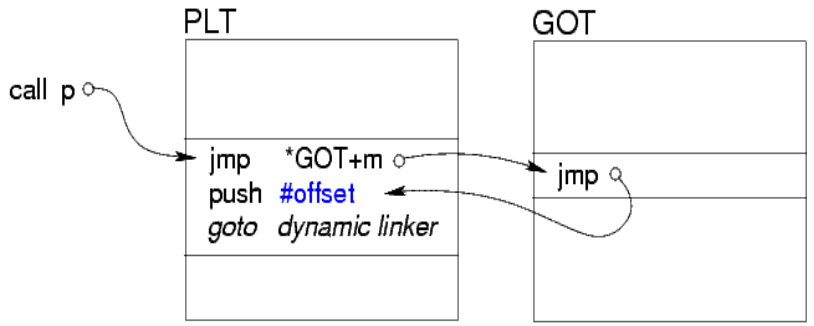
ELF Dynamic Linking: PLT and GOT



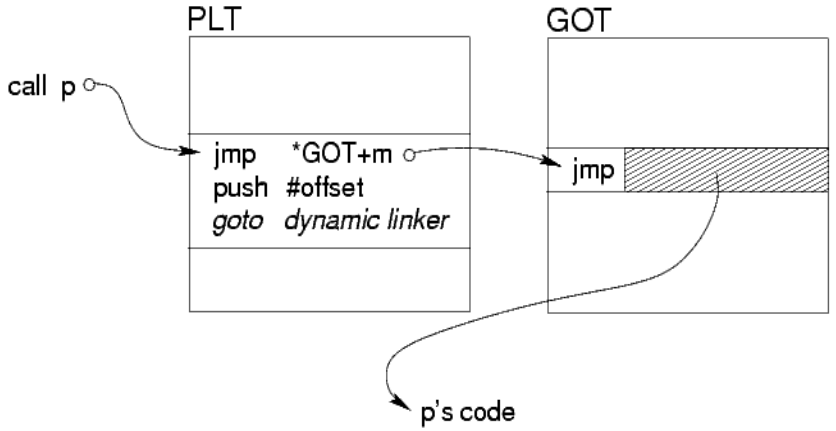
ELF Dynamic Linking: Lazy Linkage

- Initially, GOT entry points to PLT code that invokes the dynamic linker.
  - **Offset** identifies both the symbol being resolved and the corresponding GOT entry.
- The dynamic linker looks up the symbol value and updates the GOT entry.
- Subsequent calls bypass dynamic linker, go directly to callee.
- This reduces program startup time. Also, routines that are never called are not resolved.

Before:

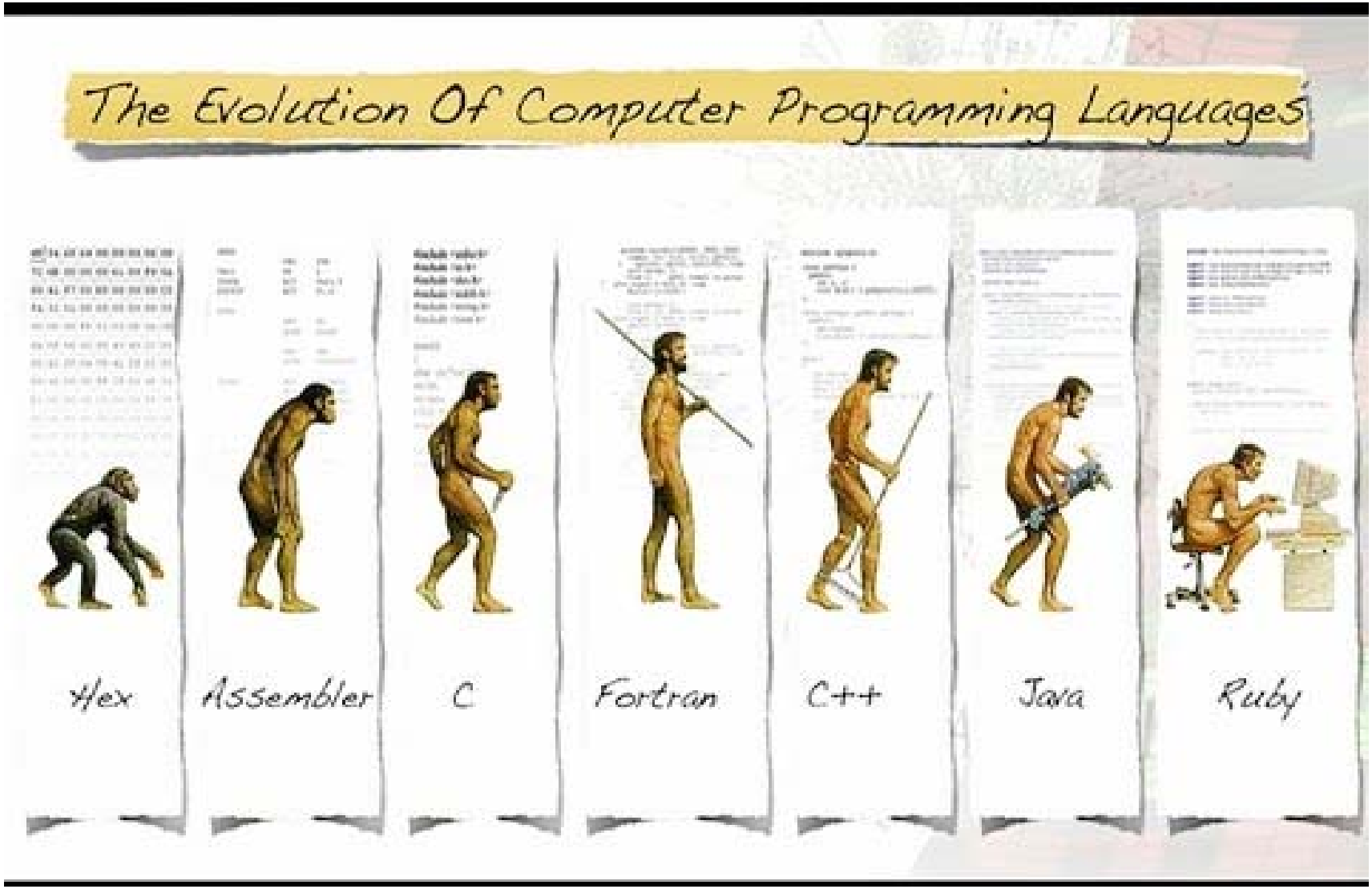


After:



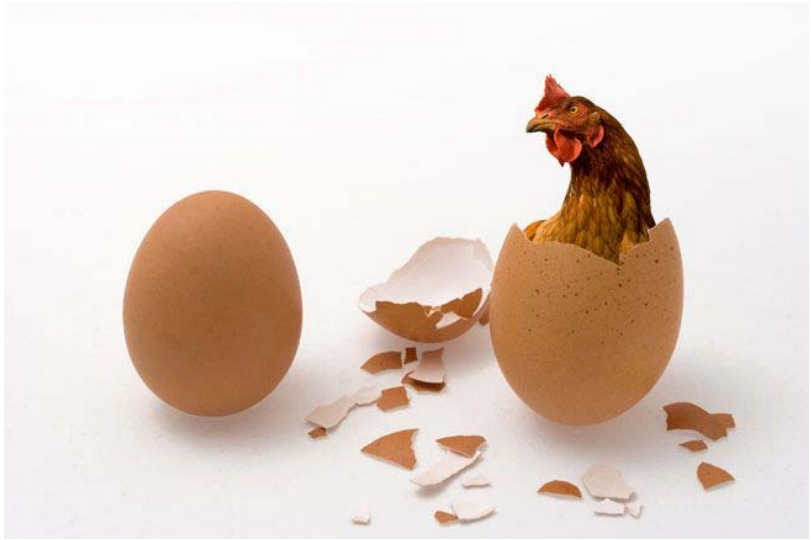
END

Evolution of Programming Languages



Bootstraping: Self-Compiling Compilers

- A compiler (or assembler) written in the source programming language that it intends to compile.
- An initial core version of the compiler (the bootstrap compiler) is generated in a different language (which could be assembly language); successive expanded versions of the compiler are developed using this minimal subset of the language.
- The problem of compiling a self-compiling compiler has been called the chicken-or-egg problem in compiler design, and bootstrapping is a solution to this problem



Reflections on Trusting Trust

Actual Entry Point: _start and crt0

- For most C and C++ programs, the true entry point is not ***main***, it's the ***_start*** function, which initializes the program runtime and invokes the program's *main* function.

- Conventionally, it is implemented as ***crt0*** (also known as ***c0***), a set of execution startup routines linked into a C program that performs any initialization work required before calling the program's *main* function.

- *crt0* generally takes the form of an object file called ***crt0.o***, often written in assembly language (***crt0.s***), which is automatically included by the linker into every executable file it builds. "crt" stands for "C runtime", and the zero stands for "the very beginning".

An Example of crt0.s

```
18      .text
19      .global _start
20      .type   _start, @function
21    _start:
22      # Initialize global pointer
23    .option push
24    .option norelax
25    1:auipc gp, %pcrel_hi(__global_pointer$)
26      addi  gp, gp, %pcrel_lo(1b)
27    .option pop
28
29      # Clear the bss segment
30      la      a0, _edata
31      la      a2, _end
32      sub     a2, a2, a0
33      li      a1, 0
34      call    memset
35    #ifdef _LITE_EXIT
36      # Make reference to atexit weak to avoid unconditionally pulling in
37      # support code.  Refer to comments in __atexit.c for more details.
38      .weak   atexit
39      la      a0, atexit
40      beqz    a0, .Lweak_atexit
41      .weak   __libc_fini_array
42    #endif
43
44      la      a0, __libc_fini_array   # Register global termination functions
45      call    atexit                  #  to be called upon exit
46    #ifdef _LITE_EXIT
47    .Lweak_atexit:
48    #endif
49      call    __libc_init_array       # Run global initialization functions
50
51      lw      a0, 0(sp)                 # a0 = argc
52      addi    a1, sp, __SIZEOF_POINTER__ # a1 = argv
53      li      a2, 0                     # a2 = envp = NULL
54      call    main
55      tail    exit
56      .size  _start, .-_start
```

Quiz

```
#include <stdio.h>
#include <stdlib.h>


#define ITEM_NUM 16
static int ITEM_SIZE = 4;


int main() {
    size_t buf_size = ITEM_NUM * ITEM_SIZE * sizeof(char);
    char* heap_buf = (char *) malloc(buf_size);
    if (heap_buf) {
        printf("Succeed to allocate: %d!\n", buf_size);
    }
    free(heap_buf);
    return 0;
}
```

Where does the assembler place the following symbols in the object file that it creates?

A. Text Section
B. Data Section
C. Exported reference in symbol table
D. Imported reference in symbol table
E. None of the above

Q1: ITEM_NUM
Q2: ITEM_SIZE
Q3: malloc