RISC-V ISA

Computer Systems Organization, Spring 2018

Lavanya Ramapantulu
Email: lavanya.r@iiit.ac.in
Office: Vindhya A5-304b

Learning Objectives

- Program abstractions and privilege modes
- Design principles of RISC-V ISA
- Instruction encoding formats
- Types of instructions

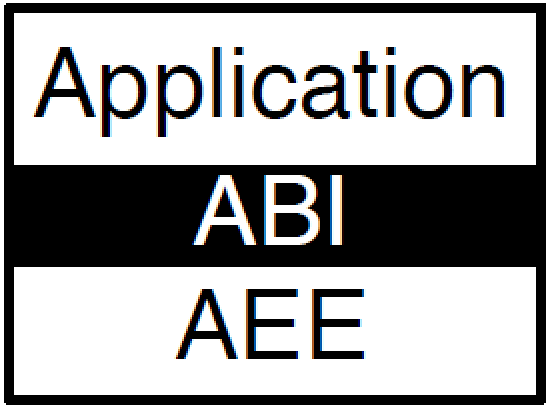
Why RISC-V

- Open and free
- Not domain-specific
- To keep things simple, flexible and extensible
- No baggage of legacy

RISC-V ISA manuals

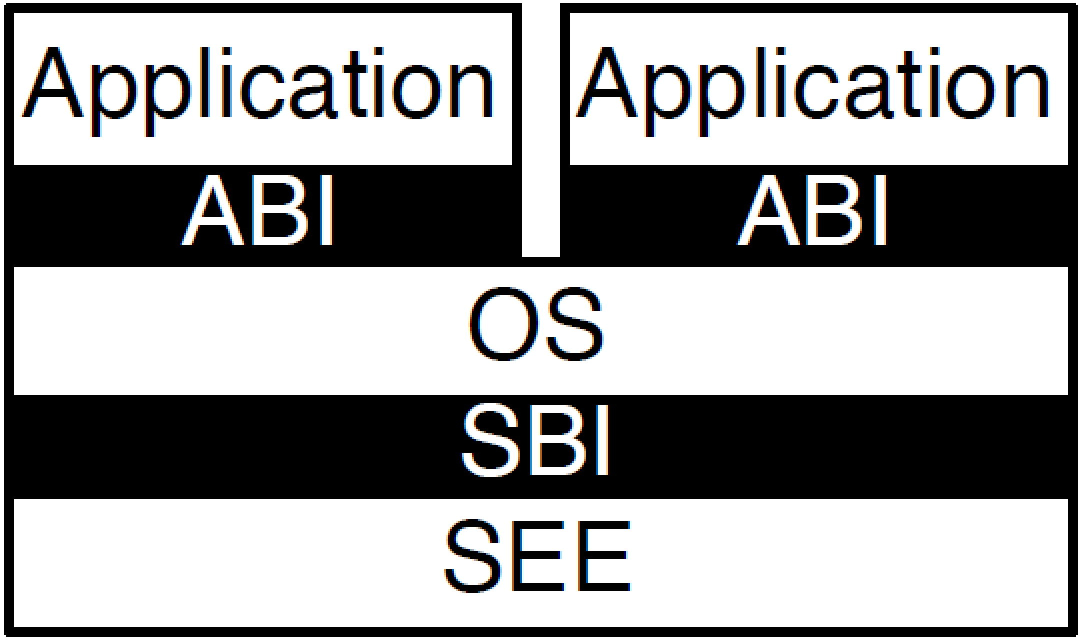
- User level – Volume 1
- Privileged level – Volume 2

Program Abstractions



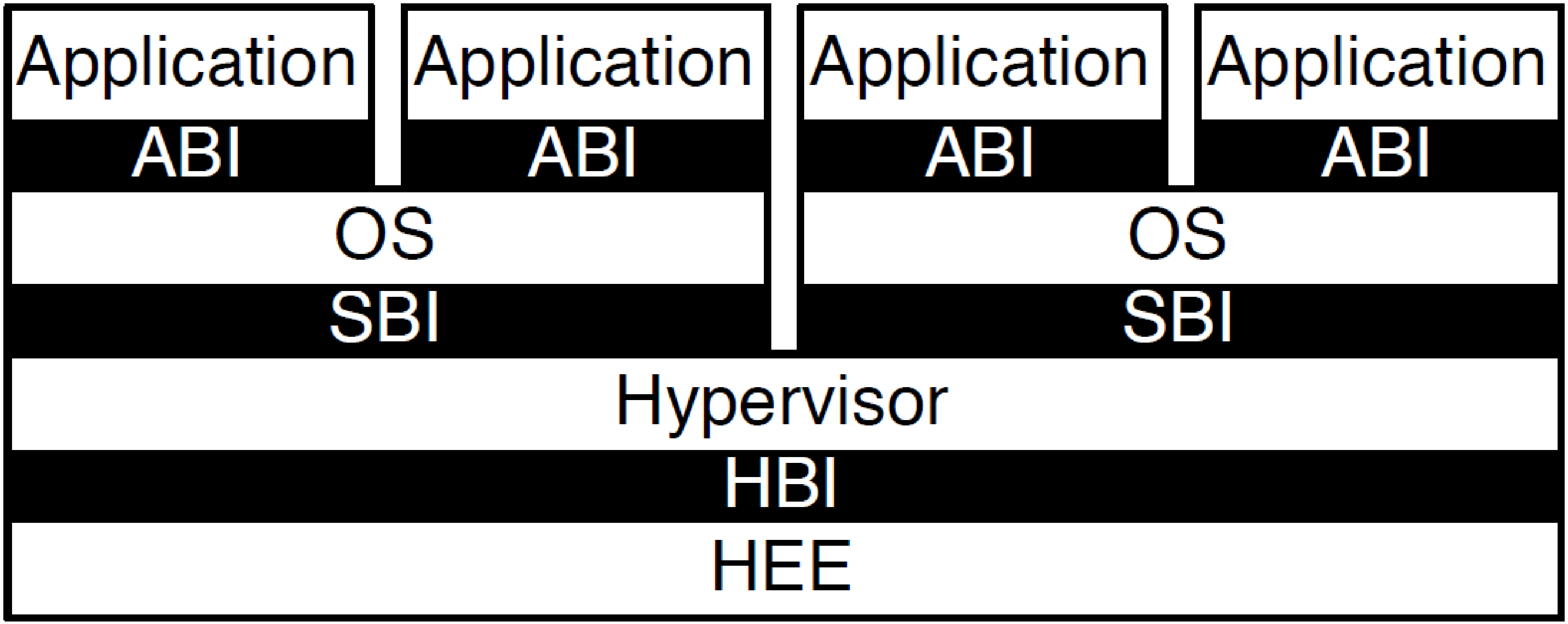
Source: https://riscv.org/specifications/privileged-isa

Program Abstractions



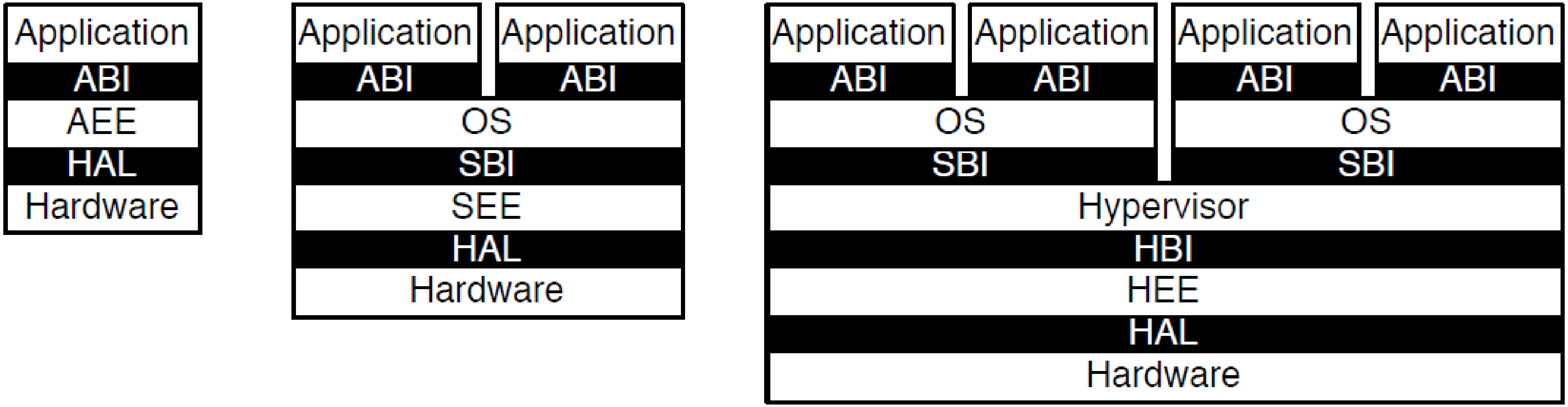
Source: https://riscv.org/specifications/privileged-isa

Program Abstractions



Source: https://riscv.org/specifications/privileged-isa

Hardware Abstraction



Source: https://riscv.org/specifications/privileged-isa

RISC-V privilege levels

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

- control and status register bits indicate the level

Source: https://riscv.org/specifications/privileged-isa

RISC-V versions

- Which version does the implementation support ?

- ISA Name format: **RV[###][abc.....xyz]**
  - RV: RISCV
  - [###]: 32, 64 or 128 (register width and user address space)
  - [abc…xyz] extensions supported
- Base integer ISA + extensions

RISC-V ISA Design Principle-1

- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

- E.g. All arithmetic operations have same form
  - Two sources and one destination

```
add a, b, c  // a gets b + c
```

RISC-V ISA Design Principle-2

- *Design Principle 2:* Smaller is faster
  - memory is larger than no. of registers, use register operands

- E.g. Arithmetic operations use register operands and not direct memory
- most implementations have decoding the operands on the critical path so only 32 registers

RISC-V ISA Design Principle-3

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

- support for immediate operands,
- e.g. `addi x22, x22, 4`

RISC-V ISA Design Principle-4

- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

- E.g. R-format and I-format, I-format versus S-format

Instruction Encoding

- Variable length encoding supported
- Base-ISA: 32-bits

		xxxxxxxxxxxxxxaa	16-bit (aa $\neq$ 11)
	xxxxxxxxxxxxxxxx	xxxxxxxxxxxbbb11	32-bit (bbb $\neq$ 111)
$\cdots$xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	48-bit
$\cdots$xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxx0111111	64-bit
$\cdots$xxxx	xxxxxxxxxxxxxxxx	xxxxxnnnn1111111	(80+16*nnnn)-bit, nnnn$\neq$1111
$\cdots$xxxx	xxxxxxxxxxxxxxxx	xxxxx11111111111	Reserved for $\geq$320-bits
Byte Address: base+4	base+2	base	

Base Instruction Formats

31 — 25	24 — 20	19 — 15	14 — 12	11 — 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[31:12]				rd	opcode	U-type

- Register
- Immediate
- Stores
- Loads with immediate

R-format Instruction

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

R-format Example

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

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

$015A04B3_{16}$

R-format Instructions

- Shift operations (logical and arithmetic)
  - SLL, SRL, SRA (why no SLA ?)
- Arithmetic operations
  - ADD, SUB
- Logical operations
  - XOR, OR, AND (missing NOT ?)
- Compare operations
  - SLT, SLTU (what is a good implementation of SLTU?)

I-format Instruction

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended

I-format Instructions

- Loads: LB, LH, LW, LBU, LHU (why not stores ?)
- Shifts: SLLI
- Arithmetic: ADDI (why not sub ?)
- Logical: XORI, ORI, ANDI
- Compare: SLTI, SLTIU
- System call and break , Sync threads, Counters

S-format Instruction

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

- Stores: SB, SH, SW

U-format Instruction

immediate	rd	opcode
20 bits	5 bits	7 bits

- Why is this separate format needed

- How to load a 32 bit constant into a register ?
  - Rd [31:12] == immediate[19:0]
  - Rd [11:0] == 12’b0

- Load upper immediate (LUI)
- Add upper immediate to PC (AUIPC)

Other instruction formats

- What is missing ?
- NOP ?
- Is the above list complete ?
- Control flow instructions

SB-format Instruction

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1 bit	6 bits	5 bits	5 bits	3 bits	4 bits	1 bit	7 bits

- Why different immediate format for branch instructions

- What about imm[0] ?

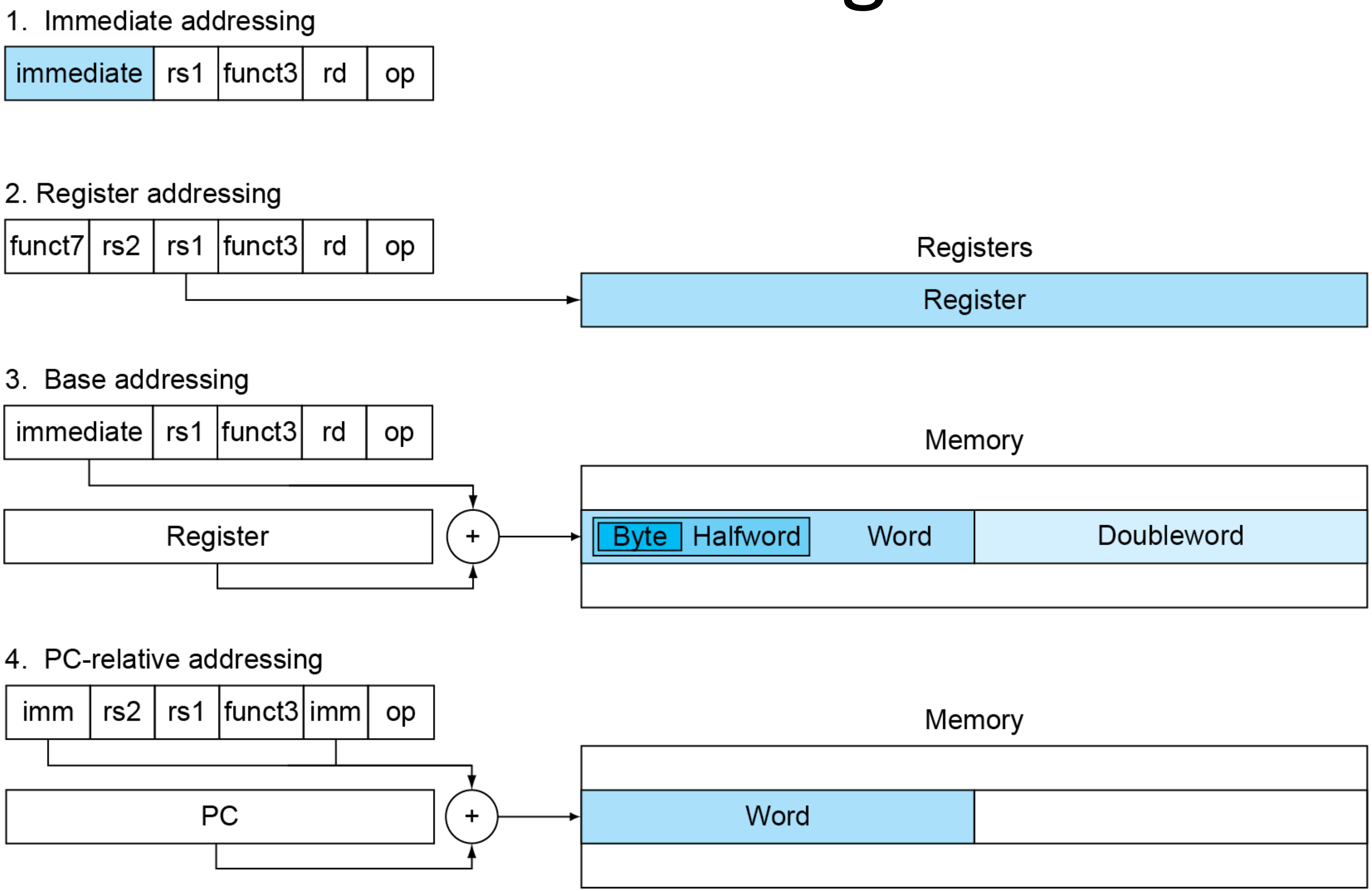
- Branches: BEQ, BNE, BLT, BGE, BLTU, BGEU
- What about overflows ?

UJ-format Instruction

imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
20 bits				5 bits	7 bits

- Why different immediate format for jump ?
- What about imm[0] ?
- JAL – jump and link
- What about JALR (jump and link return ?)
  - I-type format, Why ?

Addressing Modes



Types of Immediate

31	30 20	19 12	11	10 5	4 1	0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

32-bit Constants

- 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`

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

`addi x19,x19,128  // 0x500`

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

Branch Addressing

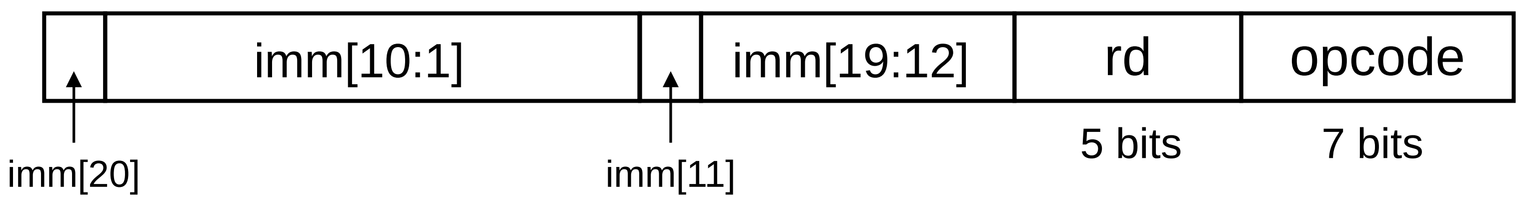
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:

imm[12]	imm [10:5]	rs2	rs1	funct3	imm [4:1]	imm[11]	opcode

- PC-relative addressing
  - Target address = PC + immediate × 2

Jump Addressing

- Jump and link (`jal`) target uses 20-bit immediate for larger range
- UJ format:



- 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

Registers

- 32 integer registers
  – Floating point optional
- Width is flexible
- ABI is open and standardized
  – Software interoperability
  – e.g. gcc assembler accepts both register names X## or ABI names

List of Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register to ABI mapping

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Source: Page 109 of RISCV Base ISA Manual

Procedure calling

- Steps required
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call (address in x1)

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

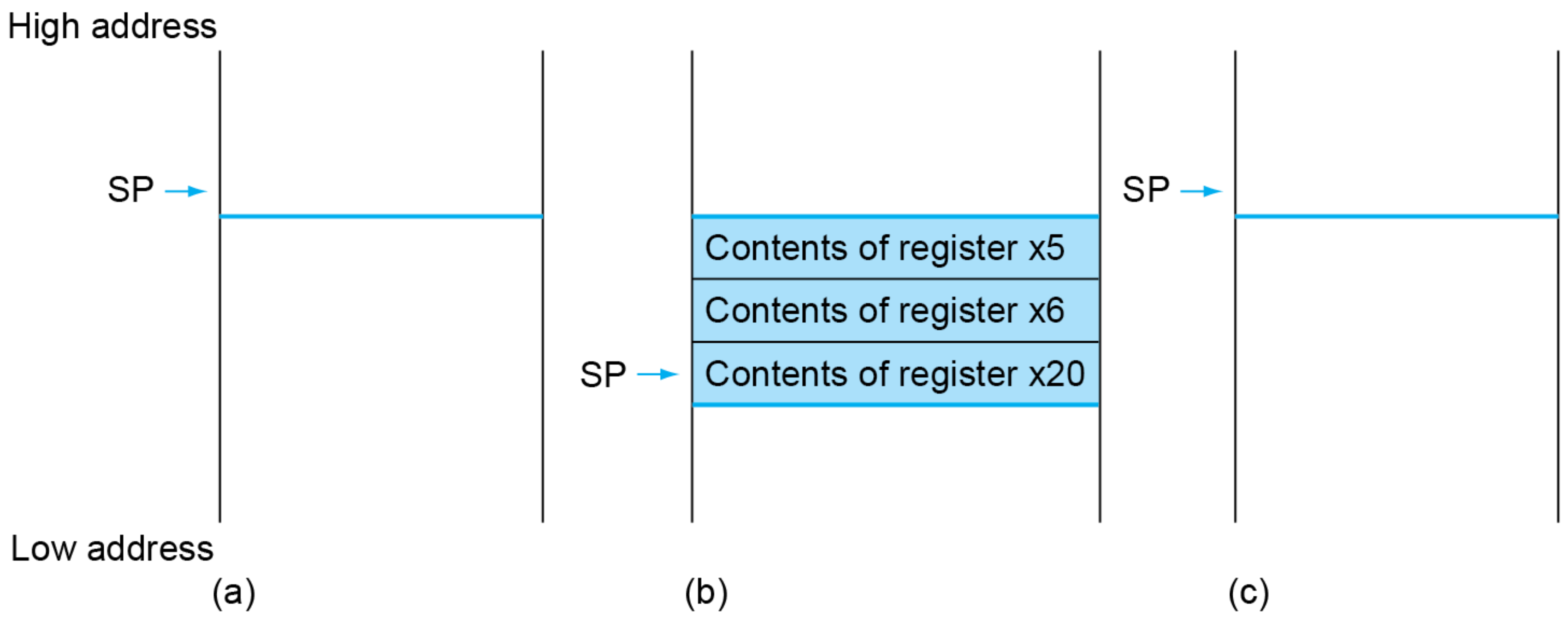
Leaf Procedure Example

```
leaf_example:
  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)        Restore x5, x6, x20 from stack
  ld   x6,8(sp)
  ld   x5,16(sp)
  addi sp,sp,24
  jalr x0,0(x1)         Return to caller
```

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

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

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

Non-Leaf Procedure Example

- C code:

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

  - Argument n in x10
  - Result in x10

Leaf Procedure Example

- RISC-V code:

```
fact:
    addi sp,sp,-16
    sd   x1,8(sp)
    sd   x10,0(sp)
    addi x5,x10,-1
    bge  x5,x0,L1
    addi x10,x0,1
    addi sp,sp,16
    jalr x0,0(x1)
L1: addi x10,x10,-1
    jal  x1,fact
    addi x6,x10,0
    ld   x10,0(sp)
    ld   x1,8(sp)
    addi sp,sp,16
    mul  x10,x10,x6
    jalr x0,0(x1)
```

Save return address and n on stack

x5 = n - 1

if n >= 1, go to L1

Else, set return value to 1

Pop stack, don't bother restoring values

Return

n = n - 1

call fact(n-1)

move result of fact(n - 1) to x6

Restore caller's n

Restore caller's return address

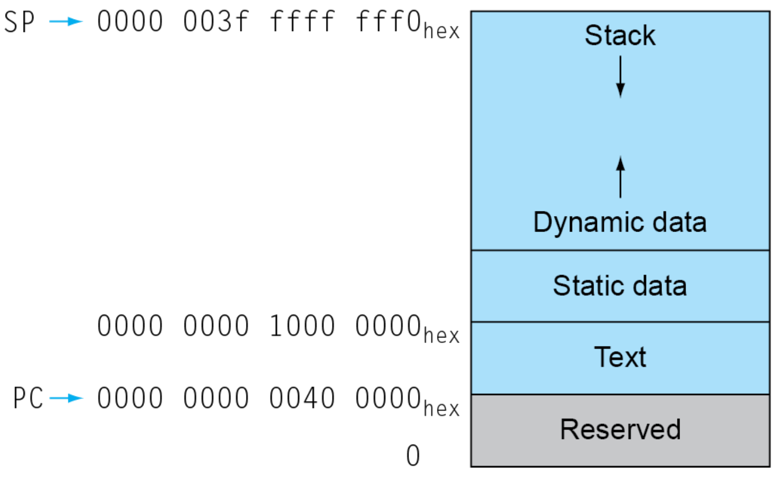
Pop stack

return n * fact(n-1)

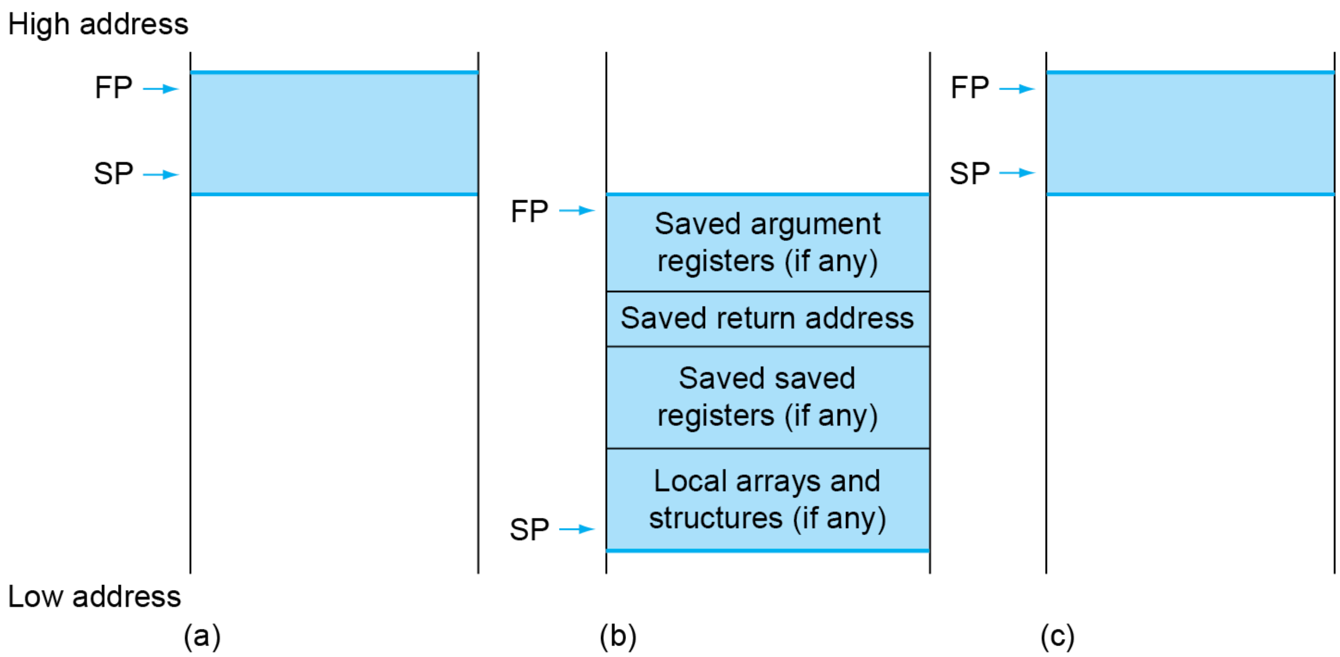
return

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



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

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ size_t i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

String Copy Example

- RISC-V code:

```
strcpy:
    addi sp,sp,-8          // adjust stack for 1
doubleword
    sd   x19,0(sp)         // push x19
    add  x19,x0,x0         // i=0
L1: add  x5,x19,x10        // x5 = addr of y[i]
    lbu  x6,0(x5)          // x6 = y[i]
    add  x7,x19,x10        // x7 = addr of x[i]
    sb   x6,0(x7)          // x[i] = y[i]
    beq  x6,x0,L2          // if y[i] == 0 then exit
    addi x19,x19,1         // i = i + 1
    jal  x0,L1             // next iteration of loop
L2: ld   x19,0(sp)         // restore saved x19
    addi sp,sp,8           // pop 1 doubleword from stack
    jalr x0,0(x1)          // and return
```

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in RISC-V

- Load reserved: `lr.d rd,(rs1)`
  - Load from address in rs1 to rd
  - Place reservation on memory address
- Store conditional: `sc.d rd,(rs1),rs2`
  - Store from rs2 to address in rs1
  - Succeeds if location not changed since the `lr.d`
    - Returns 0 in rd
  - Fails if location is changed
    - Returns non-zero value in rd

Synchronization in RISC-V

- Example 1: atomic swap (to test/set lock variable)

```
again:  lr.d x10,(x20)
        sc.d x11,(x20),x23 // X11 = status
        bne  x11,x0,again  // branch if store failed
        addi x23,x10,0     // X23 = loaded value
```

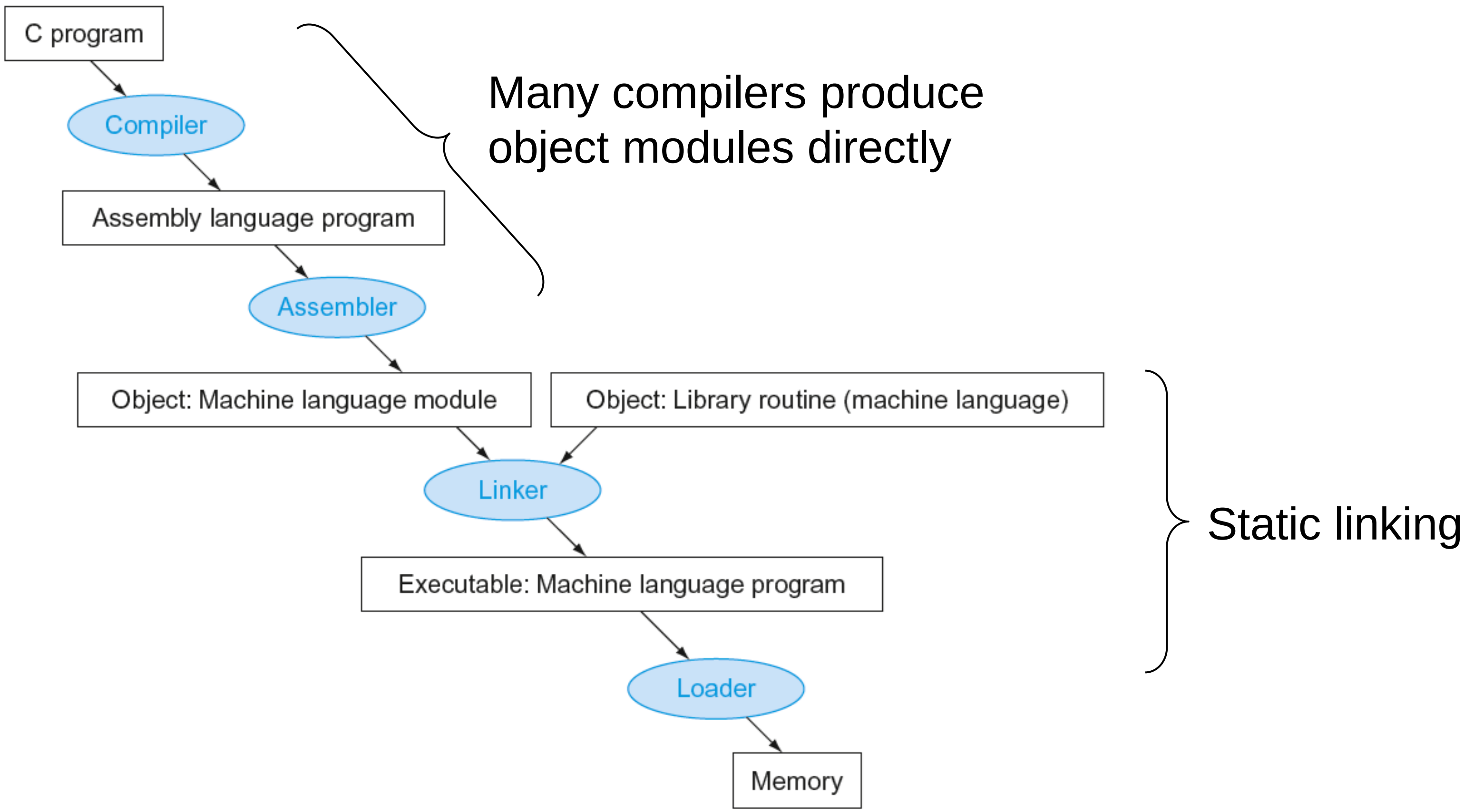
- Example 2: lock

```
        addi x12,x0,1          // copy locked value
again:  lr.d x10,(x20)         // read lock
        bne  x10,x0,again      // check if it is 0 yet
        sc.d x11,(x20),x12     // attempt to store
        bne  x11,x0,again      // branch if fails
```

- Unlock:

```
        sd   x0,0(x20)         // free lock
```

Translation and Startup



Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- 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

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including sp, fp, gp)
  6. Jump to startup routine
     - Copies arguments to x10, … and calls main
     - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

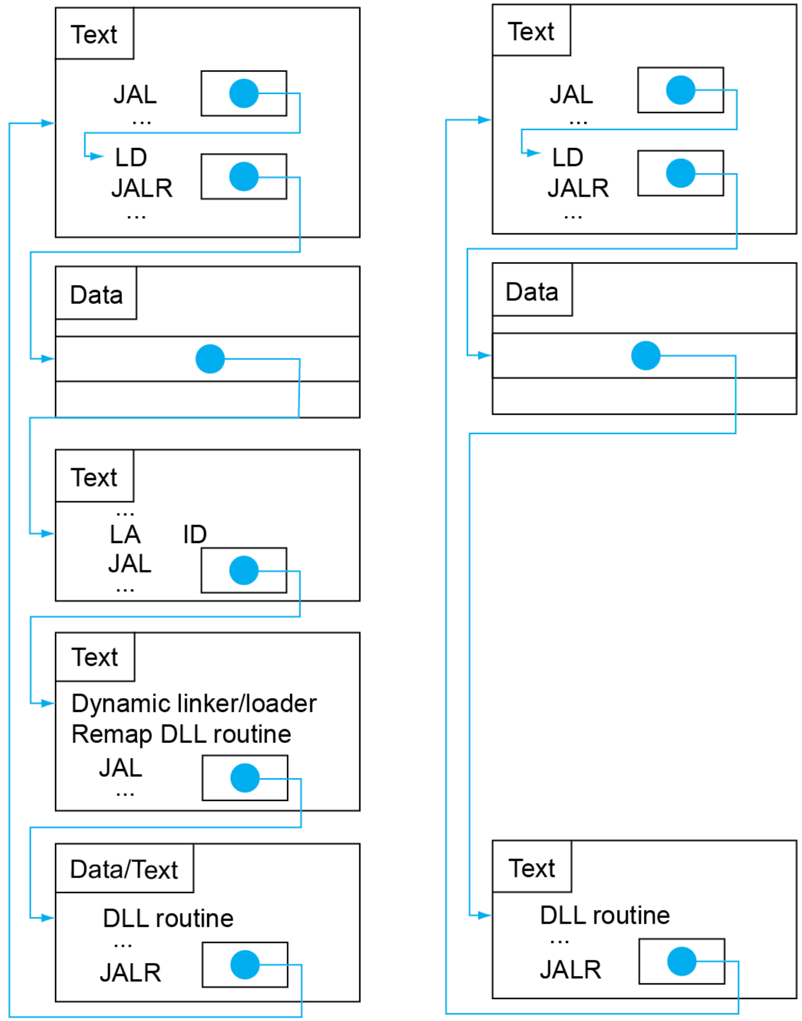
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code



(a) First call to DLL routine (b) Subsequent calls to DLL routine

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(long long int v[],
          long long int k)
{
  long long int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v in x10, k in x11, temp in x5

§2.13 A C Sort Example to Put It All Together

The Procedure Swap

```
swap:
  slli x6,x11,3     // reg x6 = k * 8
  add  x6,x10,x6    // reg x6 = v + (k * 8)
  ld   x5,0(x6)     // reg x5 (temp) = v[k]
  ld   x7,8(x6)     // reg x7 = v[k + 1]
  sd   x7,0(x6)     // v[k] = reg x7
  sd   x5,8(x6)     // v[k+1] = reg x5 (temp)
  jalr x0,0(x1)     // return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (long long int v[], size_t n)
{
  size_t i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  – v in x10, n in x11, i in x19, j in x20

The Outer Loop

- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
   li   x19,0              // i = 0
 for1tst:
   bge  x19,x11,exit1   // go to exit1 if x19 ≥ x11 (i≥n)

   (body of outer for-loop)

   addi x19,x19,1        // i += 1
   j    for1tst          // branch to test of outer loop
 exit1:
```

The Inner Loop

- Skeleton of inner loop:
  - for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j – = 1) {

```
    addi x20,x19,-1     // j = i –1
 for2tst:
    blt  x20,x0,exit2   // go to exit2 if X20 < 0 (j < 0)
    slli x5,x20,3       // reg x5 = j * 8
    add  x5,x10,x5      // reg x5 = v + (j * 8)
    ld   x6,0(x5)       // reg x6 = v[j]
    ld   x7,8(x5)       // reg x7 = v[j + 1]
    ble  x6,x7,exit2    // go to exit2 if x6 ≤ x7
    mv   x21, x10       // copy parameter x10 into x21
    mv   x22, x11       // copy parameter x11 into x22
    mv   x10, x21       // first swap parameter is v
    mv   x11, x20       // second swap parameter is j
    jal  x1,swap        // call swap
    addi x20,x20,-1     // j -= 1
    j    for2tst        // branch to test of inner loop
 exit2:
```

Preserving Registers

- Preserve saved registers:

```
  addi sp,sp,-40  // make room on stack for 5 regs
  sd   x1,32(sp)  // save x1 on stack
  sd   x22,24(sp) // save x22 on stack
  sd   x21,16(sp) // save x21 on stack
  sd   x20,8(sp)  // save x20 on stack
  sd   x19,0(sp)  // save x19 on stack
```

- Restore saved registers:

```
exit1:
  sd   x19,0(sp)  // restore x19 from stack
  sd   x20,8(sp)  // restore x20 from stack
  sd   x21,16(sp) // restore x21 from stack
  sd   x22,24(sp) // restore x22 from stack
  sd   x1,32(sp)  // restore x1 from stack
  addi sp,sp, 40  // restore stack pointer
  jalr x0,0(x1)
```

References

- RISCV Resources in Moodle, ISA Manual
- RISCV Resources in Moodle, Reference Card
- https://riscv.org/specifications/privileged-isa, Chapter 1