CSC 3210 Computer organization and programming

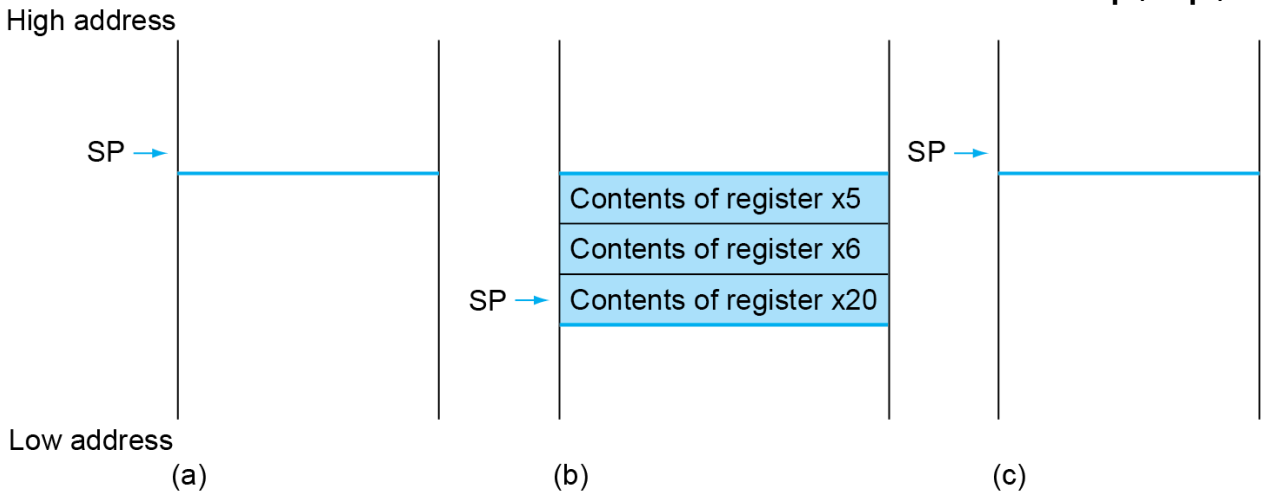
Chunlan Gao

Georgia State University

Data on the Stack

```
addi sp, sp, -12  # Allocate 12 bytes stack space
 sw   x5, 8(sp)   # Save x5 to stack (offset 8)
 sw   x6, 4(sp)   # Save x6 to stack (offset 4)
 sw  x20, 0(sp)   # Save x20 to stack (offset 0)
```

```
addi sp, sp, 12 # Free stack space
```



Non-Leaf Procedures

- Procedures that call other procedures (caller)
- 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:

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

  - Argument n in x10
  - Result in x10

Non-Leaf Procedure Example

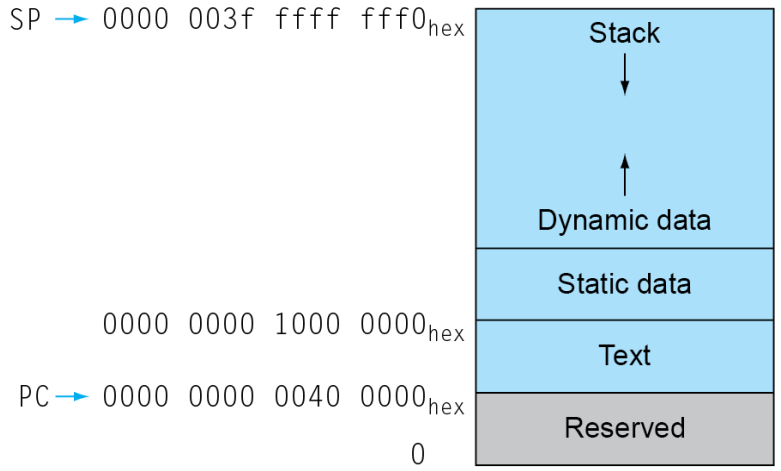
- RISC-V code:

```
fact:
    addi sp,sp,-8        \\ adjust stack for 2 items
    sw   x1,4(sp)        \\save the return address
    sw   x10,0(sp)       \\save the argument n
    addi x5,x10,-1       \\ x5 = n - 1
    bge  x5,x0,L1        \\    if n >= 1, go to L1
    addi x10,x0,1        \\    Else, set return value to 1
    addi sp,sp,8         \\    free the stack
    jalr x0,0(x1)        \\ return to the caller
L1: addi x10,x10,-1      \\    n >= 1: n = n - 1
    jal  x1,fact         \\    call fact with (n-1)
    addi x6,x10,0        \\ return from jal: move result of fact (n - 1) to x6:
    lw   x10,0(sp)       \\    Restore caller's n
    lw  x1,4(sp)         \\    Restore caller's return address
    addi sp,sp,8         \\ free the stack
    mul  x10,x10,x6      \\   n * fact(n-1)
    jalr x0,0(x1)        \\ return to the caller
```

```
int fact (int n)
{
if (n < 1) return (1);
else return (n * fact(n – 1));
}
```

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



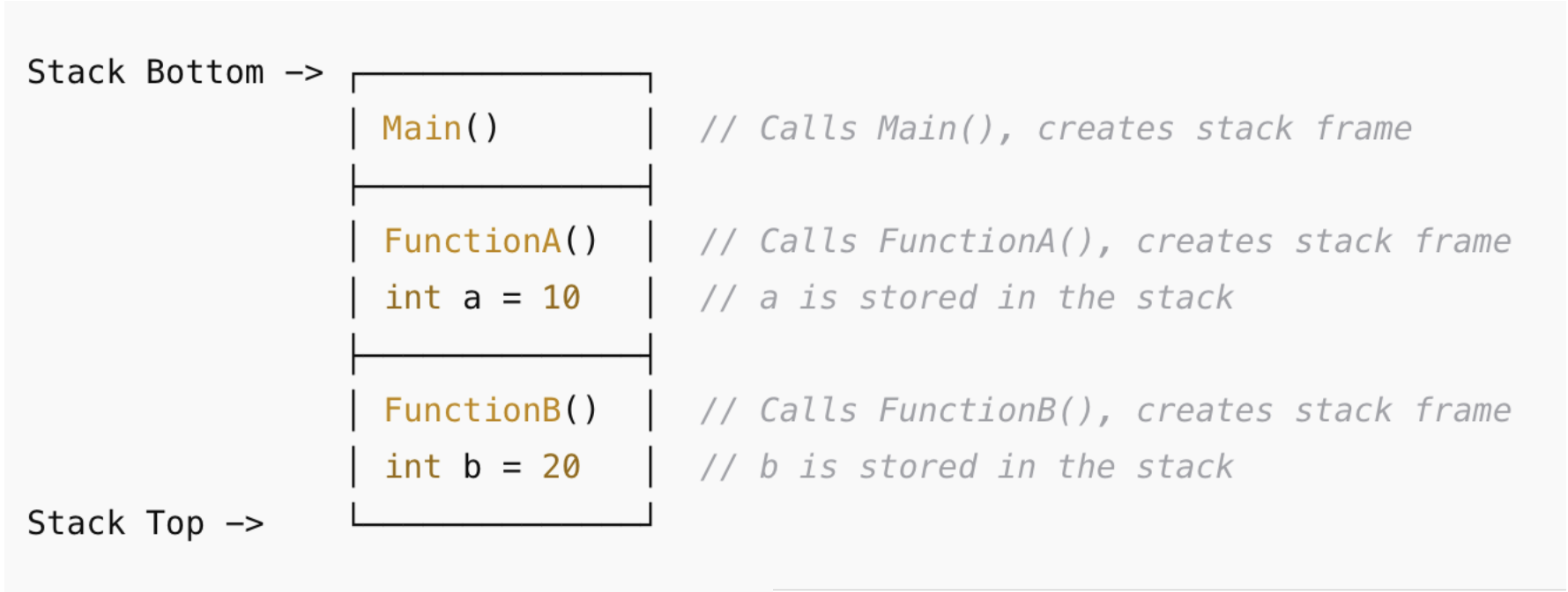
Example of stack

```
void FunctionA()
{
   int a = 10;
   FunctionB();
}

void FunctionB()
{
   int b = 20;
}

void Main()
{
   FunctionA();
}
```

Execution Process & Stack Changes:

1. Main() starts execution, and its **stack frame** is pushed onto the stack.
2. Main() calls FunctionA(), pushing FunctionA()'s **stack frame**.
3. FunctionA() declares int a = 10;, storing a on the stack.
4. FunctionA() calls FunctionB(), pushing FunctionB()'s **stack frame**.
5. FunctionB() declares int b = 20;, storing b on the stack.
6. FunctionB() finishes, and its stack frame is popped.
7. FunctionA() finishes, and its stack frame is popped.
8. Main() finishes, and its stack frame is popped, clearing the stack.

Example of stack



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,-4              // adjust stack for 1 word
     sw   x19,0(sp)             // save x19
     add  x19,x0,x0             // i=0+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](write value to Rd)
     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:  lw   x19,0(sp)    // restore saved x19
     addi sp,sp,4               // pop 1 word from stack
     jalr x0,0(x1)              // and return
```

32-bit Constants

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

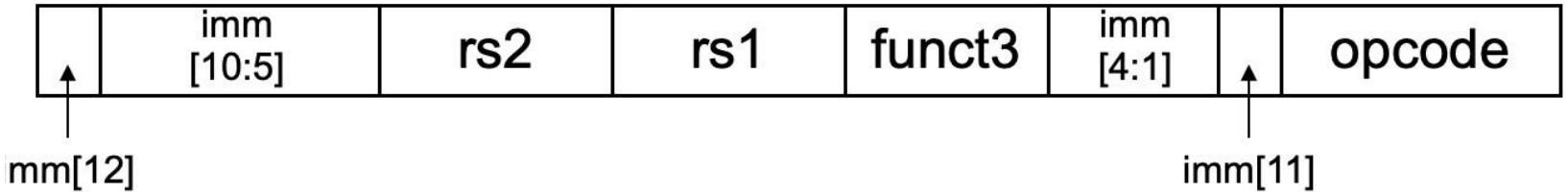
  `lui rd, constant （Load upper immediate）`
  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
- Clears bits [11:0] of rd to 0

Loading 32-niy constant into register X19, the constant is 0000 0000 0011 1101 0000 0101 0000 0000

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000
	`lui x19, 976 // 0x003D0 , [31-12]`		
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000
	`addi x19,x19,1280 // 0x500 [11-0]`		

Branch Addressing

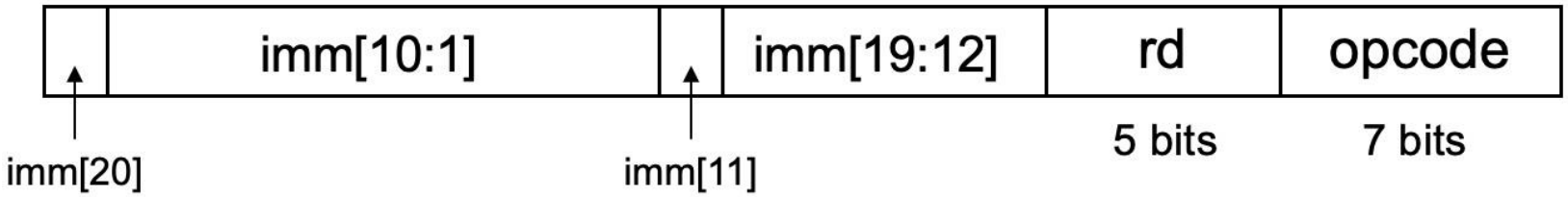
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB-type format



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

Jump Addressing

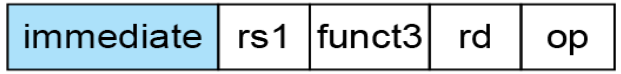
- Jump and link (`jal`) target uses 20-bit immediate for larger range
- UJ-type format:



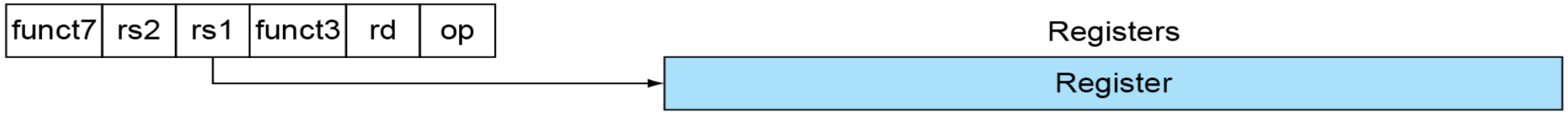
- For long jumps, eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

RISC-V Addressing Summary

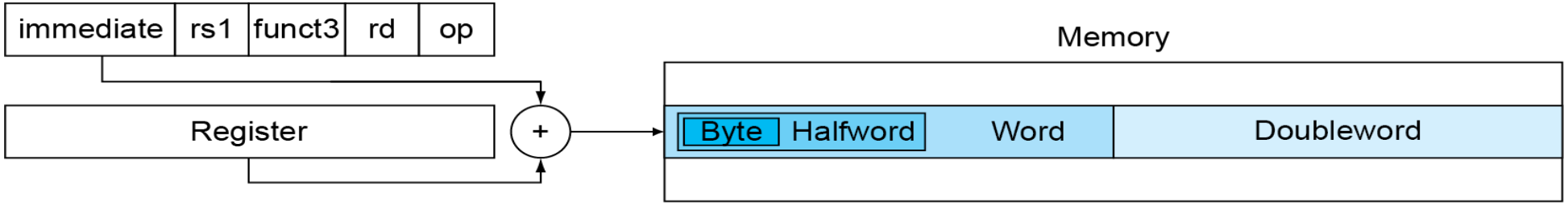
1. Immediate addressing



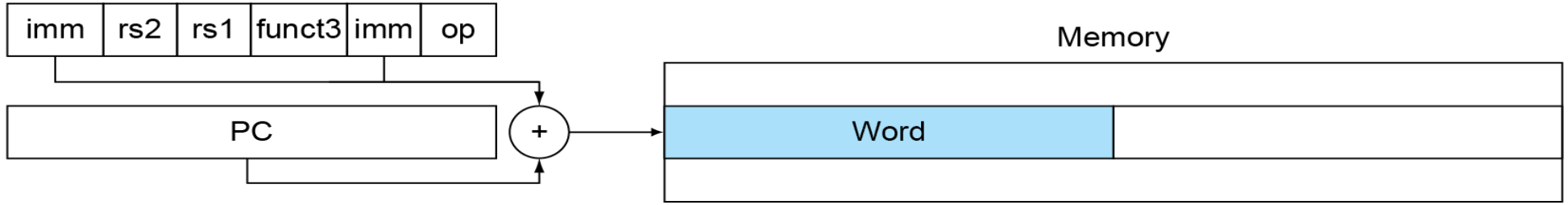
2. Register addressing



3. Base addressing



4. PC-relative addressing



RISC-V Encoding Summary

RISC-V Encoding refers to the process of converting **assembly instructions into machine code** in the RISC-V architecture.

Core Concepts of RISC-V Encoding

1. **Instruction Format**
   RISC-V follows a **fixed 32-bit instruction length** (with extensions supporting 16-bit and 64-bit instructions). Each instruction is broken down into several fields:
   1. **Opcode**: Defines the instruction category, e.g., LOAD, STORE, ALU, BRANCH, etc.
   2. **rd (Destination Register)**: Specifies the destination register.
   3. **funct3 (Function Code 3)**: Differentiates instruction variants, such as ADD vs. SUB.
   4. **rs1, rs2 (Source Registers)**: Hold operands for operations.
   5. **funct7 (Function Code 7)**: Further distinguishes operations like ADD and SUB.
   6. **Immediate**: An embedded value in the instruction.

RISC-V Encoding Summary

2. Instruction Formats (Types) RISC-V classifies instructions into the following formats:

- **R-Type (Register-to-Register Operations)**
  Example: ADD rd, rs1, rs2
- **I-Type (Immediate & Load Instructions)**
  Example: ADDI rd, rs1, imm
- **S-Type (Store Instructions)**
  Example: SW rs2, imm(rs1)
- **B-Type (Branch Instructions)**
  Example: BEQ rs1, rs2, offset
- **U-Type (Upper Immediate Load Instructions)**
  Example: LUI rd, imm
- **J-Type (Jump Instructions)**
  Example: JAL rd, offset

Example:

- addi x5, x0, 10 # x5 = 0 + 10

opcode = 0010011 (I-Type instruction)

rd = 00101 (Destination register x5)

funct3 = 000 (ADDI instruction)

rs1 = 00000 (Source register x0)

imm = 000000000001010 (Immediate value 10)

- Binary:000000000001010 00000 000 00101 0010011
- The final **32-bit machine code** in **hexadecimal**:

0x00A00293

RISC-V Encoding Summary

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