Chapter 2 Instructions: Language of the Computer

Instructions

- Instruction
  - Basic command describing an operation of a processor
  - The interface between hardware and software

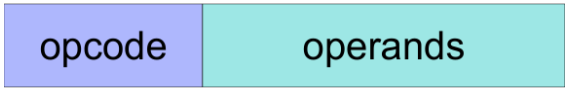
High-level language program (in C)

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

Assembly language program (for MIPS)

```
swap:
        muli $2, $5,4
        add  $2, $4,$2
        lw   $15, 0($2)
        lw   $16, 4($2)
        sw   $16, 0($2)
        sw   $15, 4($2)
        jr   $31
```

Opcode (operation code)
Operands: data for the operation



Assembly language representation

add $2, $4, $2

opcode operand

Binary language representation

00000000000110000001100000100001

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Instruction Set

- Instruction Set Architecture (ISA)
  - Set of instructions that can run on a given H/W (Processor)
  - Language of a computer
  - Interface between hardware and software
- Instruction sets are similar across different processors
  - Similar hardware technology
  - Similar underlying principles
  - Common set of functionalities all computers must provide
- Popular instruction sets
    - ARMv7, ARMv8, Intel x86, Intel IA-64, IA-32, Sun SPARC, IBM POWER, DEC Alpha, AMD64 …

Microarchitecture

- Implementation of the ISA under specific design constraints and goals
- Implementations (µarch) can be many as long as it satisfies the specification (ISA)
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core, …
- Anything done in hardware without exposure to software
  - Pipelining, Speculative execution, Memory access scheduling, ….

CISC vs. RISC

- CISC (Complex Instruction Set Computer)
  - Older design idea (x86 instruction set is CISC)
  - (Multi-clock) Complex and variable length instructions
  - Many (powerful) instructions supported within the ISA
    - Ex) LOAD and STORE incorporated into instructions
- Pros
  - Makes assembly programming much easier (lots of assembly programming in 60-70's)
  - Compiler is also simpler
  - Reduced instruction memory usage
- Cons
  - Designing CPU is much harder

- Approach in the early days
  - To add new instructions to meet new requirements/needs

→ **New Philosophy: Let's keep instructions small and simple. Let's handle complex tasks in software.**

CISC vs. RISC

- RISC (Reduced Instruction Set Computer)
  - Newer concept than CISC
  - Simple, standardized instructions
  - ARM, MIPS, POWER, RISC-V
  - Small instruction set, CISC type operation becomes a chain of RISC operations
- Pros
  - Easier to design CPU
  - Smaller instruction set => higher clock speed
- Cons
  - Assembly language typically longer (compiler design issue)
  - Heavy use of memory
- Most modern x86 processors are implemented using RISC techniques

CISC Approach: Expecting smaller inst count, but CPI increases.

RICS Approach: Trying to reduce CPI, but inst count increases.

$$\text{CPU time} = \frac{Instructions}{Program} \times \frac{Clock\ Cycles}{Instruction} \times \frac{Seconds}{Clock\ Cycle}$$

RISC-V

- RISC-V is an open standard ISA based on RISC principles
  - A completely open ISA that is freely available to academia and industry
  - RISC-V is simpler and more elegant than 80x86 ISA
  - Rapid adoption
- Fifth RISC ISA design developed at UC Berkeley
  - Now managed by the RISC-V foundation (riscv.org)

Arithmetic Operation

- Notation
  - **add a,b,c** # a = b + c
- RISC-V arithmetic instruction must have 3 operands. Why? Design simplicity.
- If we want to add 4 values; b, c, d and e into a
  - **add a, b, c**
  - **add a, a, d**
  - **add a, a, e**

  → Three instructions are required
- Design principle
  - Simplicity favors regularity.
    - Implementation becomes easier
    - Potential performance benefits

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled Assembly code:

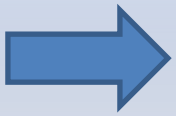
```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

Register Operands

- Operands of arithmetic instructions must be registers
  - Register: storage inside CPU
- Register size of RISC-V: 64 bits (*doubleword*)
  - Number of registers: 32
- RISC-V notation convention for expressing registers
  - (x + register number): x0, x1, x2, …

C code:

```
f = (g + h) - (i + j);
```



Compiled RISC-V code:

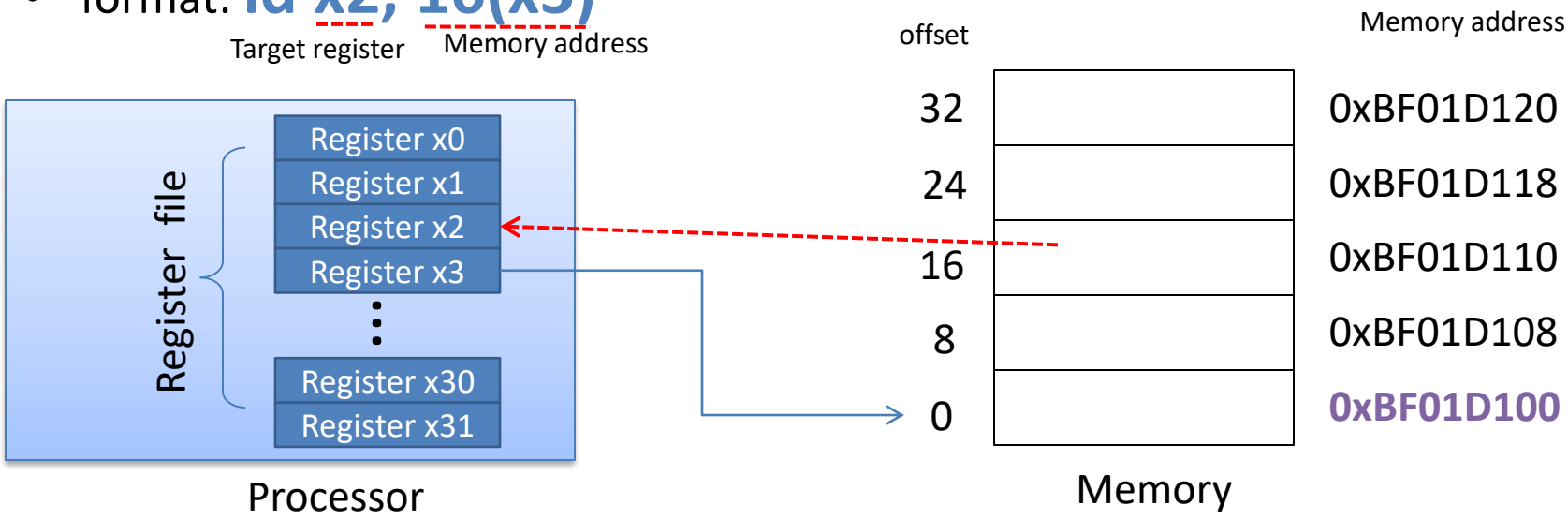
```
add x5, x20, x21   # temp x5 = g + h
add x6, x22, x23   # temp x6 = i + j
sub x19, x5, x6    # f = x5 – x6
```

f, g, h, i, j are assigned to the registers x19, x20, x21, x22, x23, respectively

Memory Operands

- Registers are limited, and data is usually larger.
  - Arithmetic must be done on registers.
  - Therefore, we need '*data transfer instructions*'
    - To/From memory and registers
- Copying data from memory to register: load
  - Address: Location of data element within a memory array
  - **ld**: *load doubleword*
  - format: **ld x2, 16(x3)**

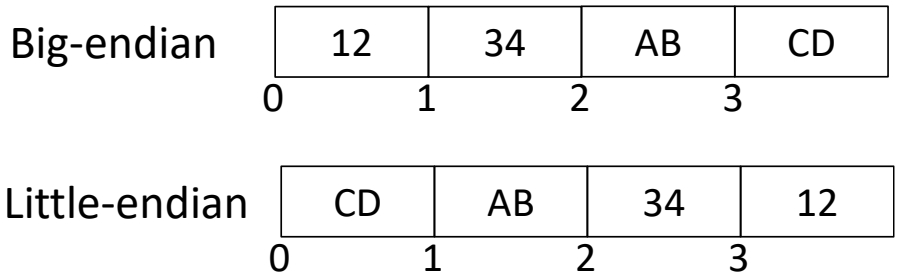
Target register: ld x2 — Memory address: 16(x3) — 0xBF01D100



Endian

- Big-endian vs. little-endian
  - In Big-endian, the most significant byte of data is placed at the lowest address.
  - In Little-endian, the least significant byte of data is placed at the lowest address.

  - Number: 0x1234ABCD



- RISC-V uses “little-endian”

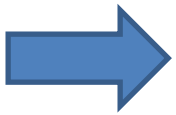
Memory Operands

- Store instruction
  - sd: store doubleword
  - format: sd x9, 96(x22)

- Base address of the array A is in x22
- h is associated with register x21

C code:

```
A[12] = h + A[8];
```



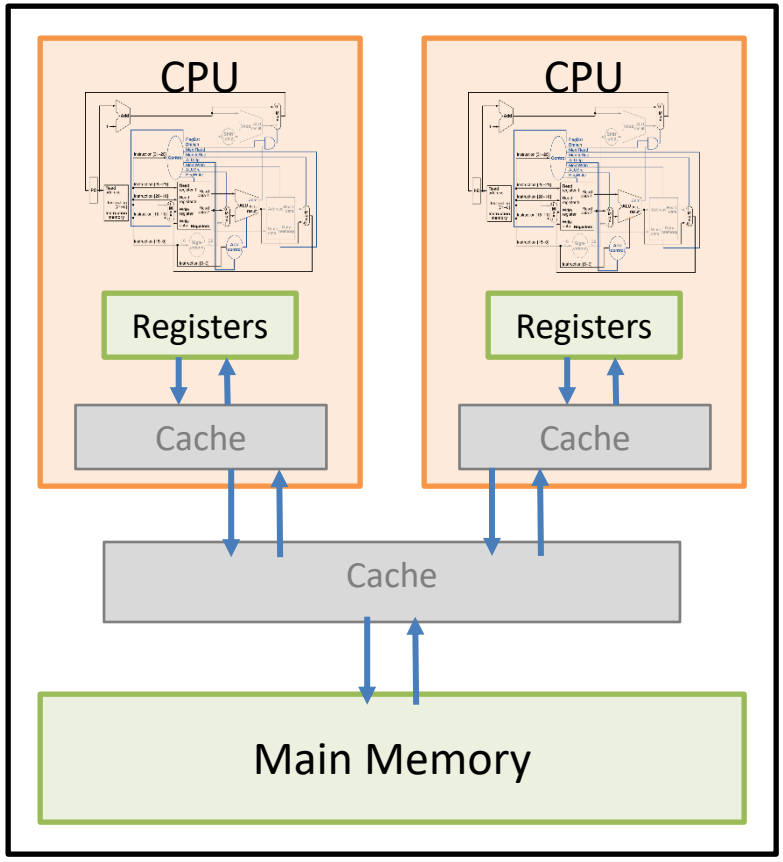
Compiled RISC-V code:

```
ld x9, 64(x22)
add x9, x21, x9
sd x9, 96(x22)
```

Register vs. Memory

- Access speed
  - Register > Memory
- Capacity
  - Register < Memory
- Energy consumption
  - Register < Memory

- Programs most likely have more variables than the number of registers.
- Frequently used variables:
  - Better to be in register than memory
    - Compilers try to optimize for this
  - *Spilling*: process of putting less commonly used variables into memory



Constant or Immediate Operands

- Arithmetic operation that uses a constant number

  Ex) Adding 4 to a variable

  - Option 1: Memory has 4 stored somewhere. It must be loaded first.

```
var1: .dword 4
...
ld x9, var1(x3)    # x3 + var1's offset
add x22, x22, x9
```

  - Option 2: Use immediate operands
    - "add immediate", `addi`

```
addi x22, x22, 4
```

    - Immediate operand instructions are faster and use less energy
    - 4 is included inside the instruction bits

Constant Zero

- Register `x0`
  - Special register hard-wired to have value 0
  - Cannot be overwritten (modified)
  - Useful for common operations
    - E.g., move data between registers

```
add x22, x21, x0        # x22 ← x21
```

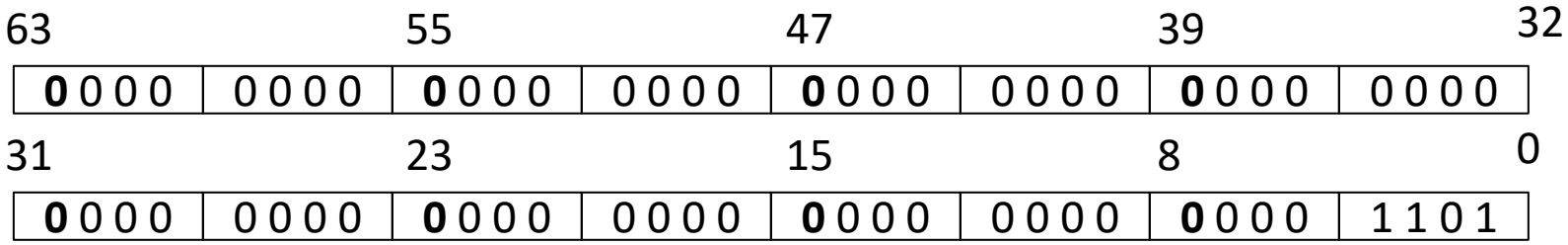
    - E.g., negate the value

```
sub x22, x0, x22        # x22 = -x22
```

Signed and Unsigned Numbers

- Numbers in base 2 (binary numbers)
  - ex) 123 base 10 = 1111011 base 2
- Value of *i*th digit d: d x Base^{i}
- $1101_{two} = (1x2^3)+(1x2^2)+(0x2^1)+(1x2^0)$

  $= 8 + 4 + 0 + 1 = 13_{ten}$
- 1101 in a doubleword



  - LSB (least significant bit) the right most bit, bit position 0
  - MSB (most significant bit) the left most bit, bit position 63

Unsigned Numbers

- RISC-V word can represent 2^{64} bit patterns

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$

. . . . . .

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 18,446,774,073,709,551,613_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 18,446,744,073,709,551,614_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 18,446,744,073,709,551,615_{ten}$

- Decimal values calculated using:

$$(x_{63} \times 2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

→ representation of the unsigned numbers

- Representation limit
  - Computer can express the numbers only within the given bit range
  - If arithmetic operation results in the larger number than representable, ***overflow*** happens.

Signed Numbers

- Expressing the sign of a number
  - One bit to indicate positive or negative sign
  - Where should the sign be? How does it affect the arithmetic operation?
- Naïve solution
  - Simply use the MSB as a sign → ***"sign and magnitude"*** representation

$0000_2 = 0_{10}$	$1000_2 = -0_{10}$
$0001_2 = 1_{10}$	$1001_2 = -1_{10}$
$0010_2 = 2_{10}$	$1010_2 = -2_{10}$
$0011_2 = 3_{10}$	$1011_2 = -3_{10}$
$0100_2 = 4_{10}$	$1100_2 = -4_{10}$
$0101_2 = 5_{10}$	$1101_2 = -5_{10}$
$0110_2 = 6_{10}$	$1110_2 = -6_{10}$
$0111_2 = 7_{10}$	$1111_2 = -7_{10}$

$6 - 3 = 0110 + 1011 = ???$

Signed Numbers

- 1's complement
  - Invert positive number bits to use as a negative number

$0000_2 = 0_{10}$	$1111_2 = -0_{10}$
$0001_2 = 1_{10}$	$1110_2 = -1_{10}$
$0010_2 = 2_{10}$	$1101_2 = -2_{10}$
$0011_2 = 3_{10}$	$1100_2 = -3_{10}$
$0100_2 = 4_{10}$	$1011_2 = -4_{10}$
$0101_2 = 5_{10}$	$1010_2 = -5_{10}$
$0110_2 = 6_{10}$	$1001_2 = -6_{10}$
$0111_2 = 7_{10}$	$1000_2 = -7_{10}$

$6 - 3 = 0110 + 1100 = 0010 = 2$???

$6 - 3 = 0110 + 1100 + 1 = 0011 = 3$

carry

- 2's complement

$0000_2 = 0_{10}$	$1111_2 = -1_{10}$
$0001_2 = 1_{10}$	$1110_2 = -2_{10}$
$0010_2 = 2_{10}$	$\mathbf{1101}_2 = -3_{10}$
$\mathbf{0011}_2 = 3_{10}$	$1100_2 = -4_{10}$
$0100_2 = 4_{10}$	$1011_2 = -5_{10}$
$0101_2 = 5_{10}$	$1010_2 = -6_{10}$
$0110_2 = 6_{10}$	$1001_2 = -7_{10}$
$0111_2 = 7_{10}$	$1000_2 = -8_{10}$

$6 - \mathbf{3} = 0110 + \mathbf{1101} = \mathbf{0011} = 3$

Signed and Unsigned Numbers

- 2's Complement
  - All negative numbers have 1 in MSB
    - To find out if number is positive or negative, H/W needs to check only one bit
  - Computing the decimal value

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

ex) 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111100

$= (-1x2^{63})+(1x2^{62})+(1x2^{61})+\ldots+(1x2^2)+(0x2^1)+(0x2^0)$

$= -9{,}223{,}372{,}036{,}854{,}775{,}808 + 9{,}223{,}372{,}036{,}854{,}775{,}804 = -4$

- Negating 2's complement binary number
  - Negate all bits and add 1

$0000_2 = 0_{10}$ $\quad$ $1111_2 = -1_{10}$
$0001_2 = 1_{10}$ $\quad$ $1110_2 = -2_{10}$
$0010_2 = 2_{10}$ $\quad$ $\mathbf{1101}_2 = -3_{10}$
$\mathbf{0011}_2 = 3_{10}$ $\quad$ $1100_2 = -4_{10}$

2→-2: 0010 → 1101 → add 1 → 1110

-2→2: 1110 → 0001 → add 1 → 0010

Signed and Unsigned Numbers

- Sign extension
  - Converting a binary number in *n* bits to a binary number in *m* bits, where m>n.
  - Take the MSB and replicate it to fill up all the bits
  - ex) 0000 0010 = 2
    - → 0000 0000 0000 0010 = 2
  - ex) 1111 1110 = -2
    - → 1111 1111 1111 1110 = -2
- Signed load performs sign extension to correctly load value
  - loading 16 bit number into 32 bit register
  - instructions: **`lb`** (load byte), **`lh`** (load halfword), **`lw`** (load word)
  - If you don't want the sign extension, use **`lbu`** (load byte unsigned)

Representing Instructions

- RISC-V **R-type** instruction format (32bit)

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

- opcode: Basic operation
- rd: destination register
- funct3: additional opcode
- rs1: 1st source register
- rs2: 2nd source register
- funct7: additional opcode

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

add x9, x20, x21

0	21	20	0	9	51

0000000	10101	10100	000	01001	0110011

Representing Instructions

- RISC-V **I-type** instruction format

Interpreted as 2's complement number

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

Instruction encoding information

Instruction	Format	immediate	rs1	funct3	rd	opcode
`addi` (add immediate)	I	constant	reg	000	reg	0010011
`ld` (load doubleword)	I	address	reg	011	reg	0000011

```
ld x14, 8(x2)
```

000000001000	00010	010	01111	0000011

← Find the error!

Representing Instructions

- RISC-V **S-type** instruction format
  - Two registers and one immediate field
  - Two immediate fields: 7bits+5bits
  - rs1 and rs2 remain in the same position for all instructions

immediate	rs2	rs1	funct3	immediate	opcode
7 bits	5 bits	5 bits	3bits	5bits	7bits

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

Compare with R-type

`sd x14, 8(x2)`

0000000		00010	011	01000	

← Fill in the blank

Instruction encoding information

Instruction	Format	immed-iate	rs2	rs1	funct3	immed-iate	opcode
sd (store doubleword)	S	address	reg	reg	011	address	0100011

Logical Operations

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

- Shift operation

  00000000 00001001 = 9

  shift left 5 00000001 00100000 = ?

- Shift instructions (I-type)
  - `slli`, `srli` – shift left (right) logical immediate
  - `sra`, `srai` – shift right arithmetic (immediate): fill with sign-bit

    1100 0011 → 1110 0001

AND/OR Instructions

- and, andi

```
and x9, x10, x11      // reg x9 = reg x10 & reg x11
```

x10	00000000	00000000	00000000	00000000	00000000	00000000	00001101	11000000
x11	00000000	00000000	00000000	00000000	00000000	00000000	00111100	00000000
x9	00000000	00000000	00000000	00000000	00000000	00000000	00001100	00000000

- or, ori

```
or x9, x10, x11 // reg x9 = reg x10 | reg x11
```

x10	00000000	00000000	00000000	00000000	00000000	00000000	00001101	11000000
x11	00000000	00000000	00000000	00000000	00000000	00000000	00111100	00000000
x9	00000000	00000000	00000000	00000000	00000000	00000000	00111101	11000000

XOR Instructions

- `xor`

```
xor x9, x10, x12 // reg x9 = reg x10 ^ reg x12
```

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x12	11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
x9	11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

- `not`
  - `xor` with 111…1111

Input 1	Input 2	output
0	**1**	**1**
1	**1**	**0**
0	0	0
1	0	1

Instructions for Making Decisions

- Branch instructions
  - *if* statement

```
beq rs1, rs2, L1
```

  - beq: "branch if equal"
  - Go to the statement labeled L1 if the value in register rs1 equals the value in register rs2

```
bne rs1, rs2, L1
```

  - bne: "branch if not equal"

- `beq, bne`: conditional branches

Compiling **if-then-else** into Conditional Branches

- C code segment

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

- Variable to register assignments
  - f:x19, g:x20, h:x21, i:x22, j:x23

- RISC-V assembly code

Text after # is comment.

```
        bne x22, x23, Else   # go to Else if i ≠ j
        add x19, x20, x21    # f = g + h (skipped if i ≠ j)
        beq x0, x0, Exit     # if 0 == 0, go to Exit
Else: sub x19, x20, x21      # f = g − h (skipped if i = j)
Exit:
```

Compiling a while loop

- C code segment

```
While (save[i]==k)
    i += 1;
```

- Variable to register assignments
  - i:x22, k:x24, base of save: x25

- RISC-V assembly code

Text after # is comment.

```
Loop:  slli x10, x22, 3        # x10 = i x 8
       add x10, x10, x25       # x10 = address of save[i]
       ld x9, 0(x10)           # 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:
```

Other Conditional Branches

- blt: “branch if less than”

```
blt rs1, rs2, L1
```

  - Compare the values (2’s complement number) in rs1 and rs2
  - Branch if rs1 is smaller

- bge: “branch if greater than or equal”

```
bge rs1, rs2, L1
```

  - Branch if rs2 is smaller

- bltu, bgeu
  - Numbers in registers are treated as unsigned

Array Index Out-of-bound check

- Array index check

```
int[] age = new int[5];
age[-2] = 12;
age[2] = 4;
age[12] = 4;
```

  - Need to check if index>0 and index<=max_length

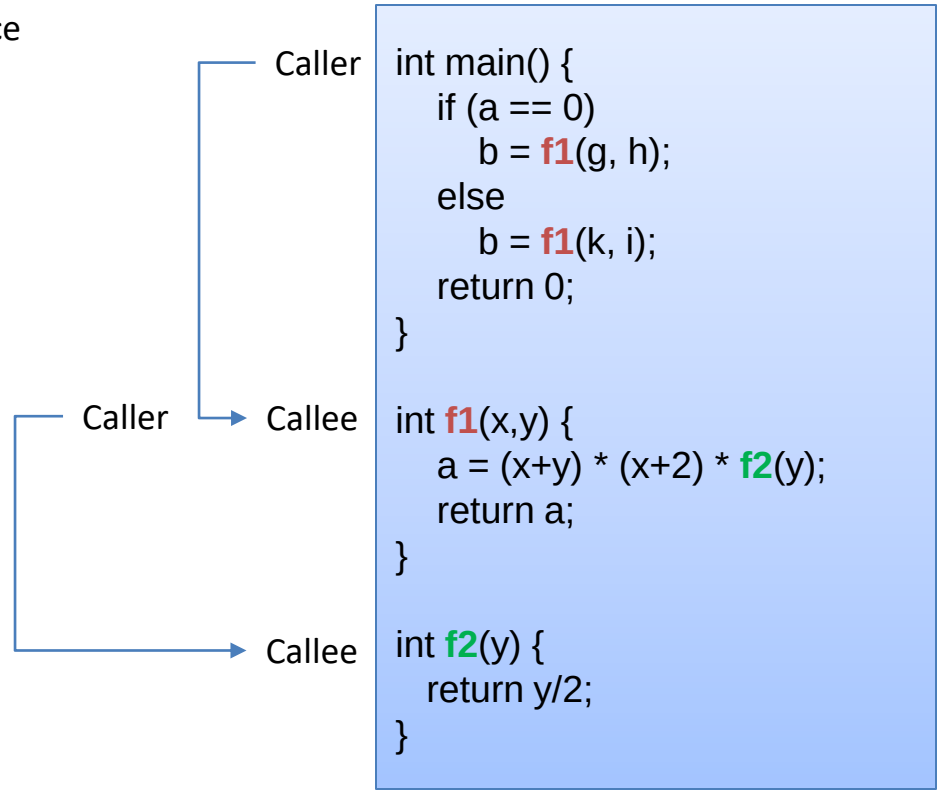
- bgeu can be used to check both conditions

Text after # is comment.

```
# if x20 >= x11 or x20 < 0, goto IndexOutOfBounds
bgeu x20, x11, IndexOutOfBounds
```

Supporting Procedures

- Procedure (function) calls
  - Pass function parameters
  - Transfer control
    - Acquire necessary memory space
  - Run procedure
  - Store return value (result) to predefined location
  - Return control



Supporting Procedures

- RISC-V conventions
  - x10~x17: eight parameter registers in which to pass parameters or return values
  - x1: one return address register
- RISC-V instruction for procedure handling

```
jal x1, ProcedureAddress
```

  - jal: “jump-and-link”
  - Branch unconditionally and save the address of the next instruction (return address) to the designated register
- Program counter (PC) register
  - Contains the address of current instruction being executed
  - jal saves PC+4 to the rd (usually x1)
- Unconditional branch without saving the return address

```
jal x0, Label
```

Supporting Procedures

- Returning from a procedure
  - jalr: “jump and link register” instruction (I-type)

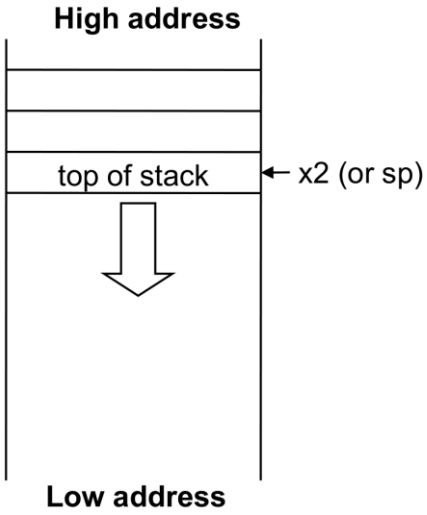
```
jalr x0, 0(x1)
```

  - Branches to the address stored in register x1 (i.e., 0 + address in x1)

** x0 is hard-wired to zero, writing to x0 will have the effect of discarding the value (e.g., return address)

Using More Registers

- Procedure may need more than 8 registers for execution
  - Compiler chooses to use other additional registers
    - Save the value of registers to memory → spill
    - Restore the value after procedure finishes



- Ideal place for spilling registers: stack
  - Last-in-first-out queue in memory
  - Memory area for spilling registers
  - Stack pointer, x2 (sp)
  - Stack grows from high address to low address
  - stack pointer is adjusted by one doubleword for each register that is saved (i.e., push) and restored (i.e., pop)

Compiling a Leaf Procedure

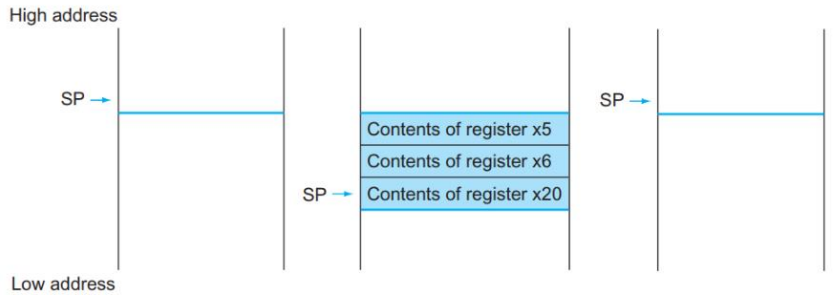
- Leaf procedure
  - Procedure that does not call other procedure

```
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;
}

// g → x10, h → x11
// i → x12, j → x13
// f → x20
```

Text after // is comment.

```
leaf_example:
addi sp, sp, -24  // adjust stack to make room for 3 items
sd   x5, 16(sp)   // save register x5 for use afterwards
sd   x6, 8(sp)    // save register x6 for use afterwards
sd   x20, 0(sp)   // save register x20 for use afterwards
add x5, x10, x11  // register x5 contains g + h
add x6, x12, x13  // register x6 contains i + j
sub x20, x5, x6   // f = x5 – x6, which is (g + h) – (i + j)
addi x10, x20, 0  // returns f (x10 = x20 + 0)
ld x20, 0(sp)     // restore register x20 for caller
ld x6, 8(sp)      // restore register x6 for caller
ld x5, 16(sp)     // restore register x5 for caller
addi sp, sp, 24   // adjust stack to delete 3 items
jalr x0, 0(x1)    // branch back to calling routine
```



Register Saving Convention

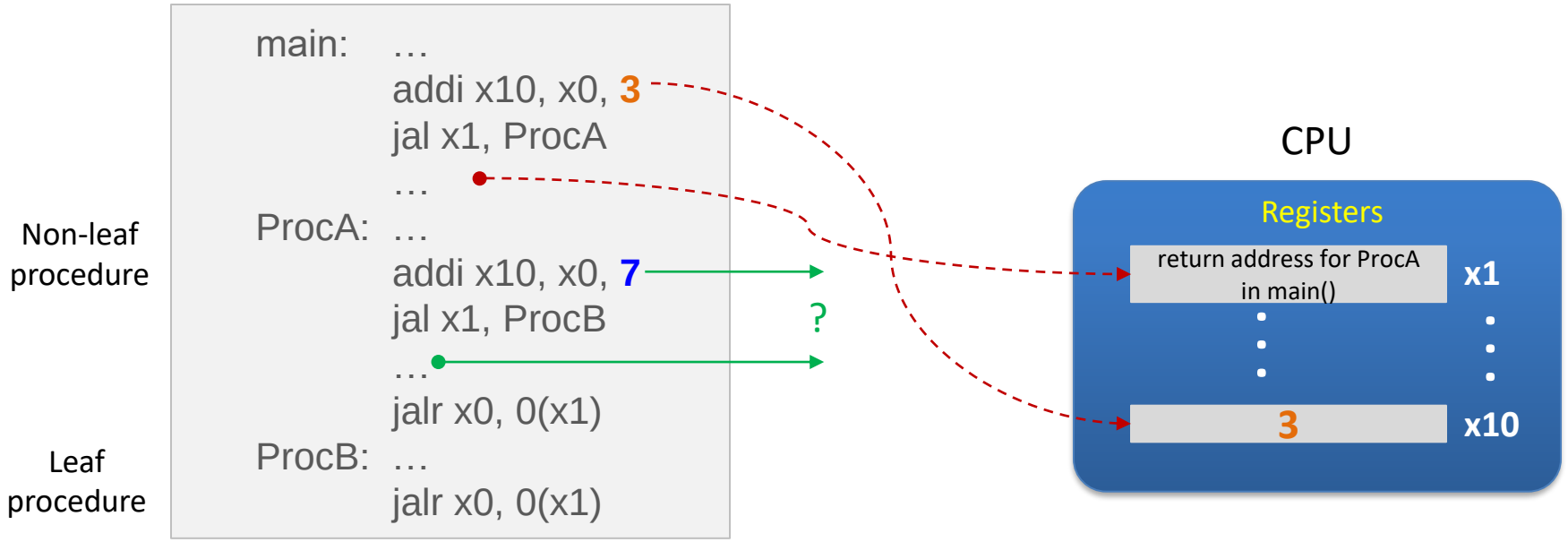
- Two groups of registers in RISC-V
  - No need for callee to save
    - x5-x7, x28-x31
    - registers for temporary use
  - Callee must preserve (save, use and restore) if used
    - x8,x9, x18-x27

Text after // is comment.

```
addi sp, sp, -24          // adjust stack to make room for 3 items
sd x5, 16(sp)             // save register x5 for use afterwards
sd x6, 8(sp)              // save register x6 for use afterwards
sd x20, 0(sp)             // save register x20 for use afterwards
add x5, x10, x11          // register x5 contains g + h
add x6, x12, x13          // register x6 contains i + j
sub x20, x5, x6           // f = x5 − x6, which is (g + h) − (i + j)
addi x10, x20, 0          // returns f (x10 = x20 + 0)
ld x20, 0(sp)             // restore register x20 for caller
ld x6, 8(sp)              // restore register x6 for caller
ld x5, 16(sp)             // restore register x5 for caller
addi sp, sp, 24           // adjust stack to delete 3 items
jalr x0, 0(x1)            // branch back to calling routine
```

Nested Procedure

- Issues with nested procedures



- Register Preservation Responsibility

Callee saves these	Caller saves these
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

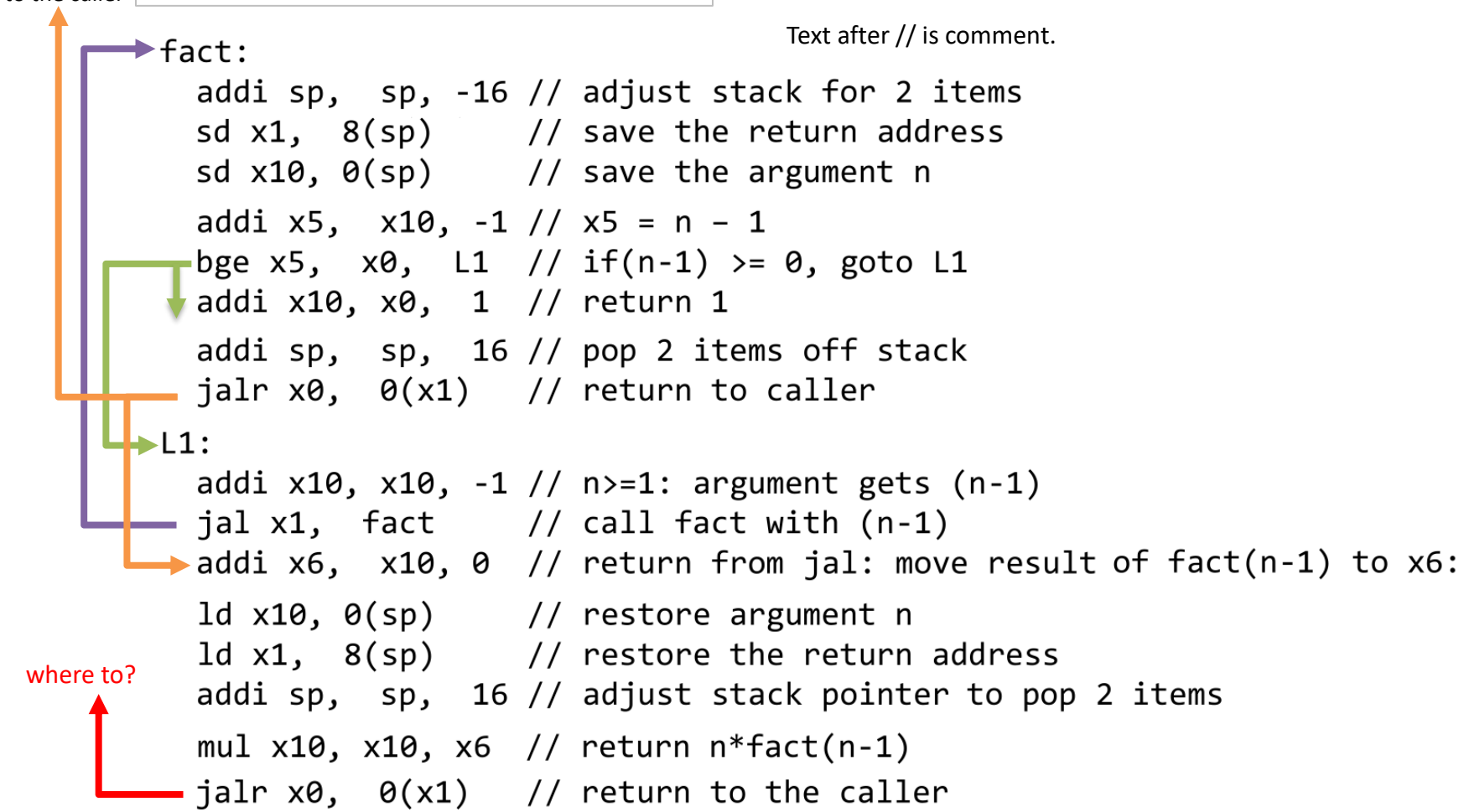
Recursive Procedure

- Factorial

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

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

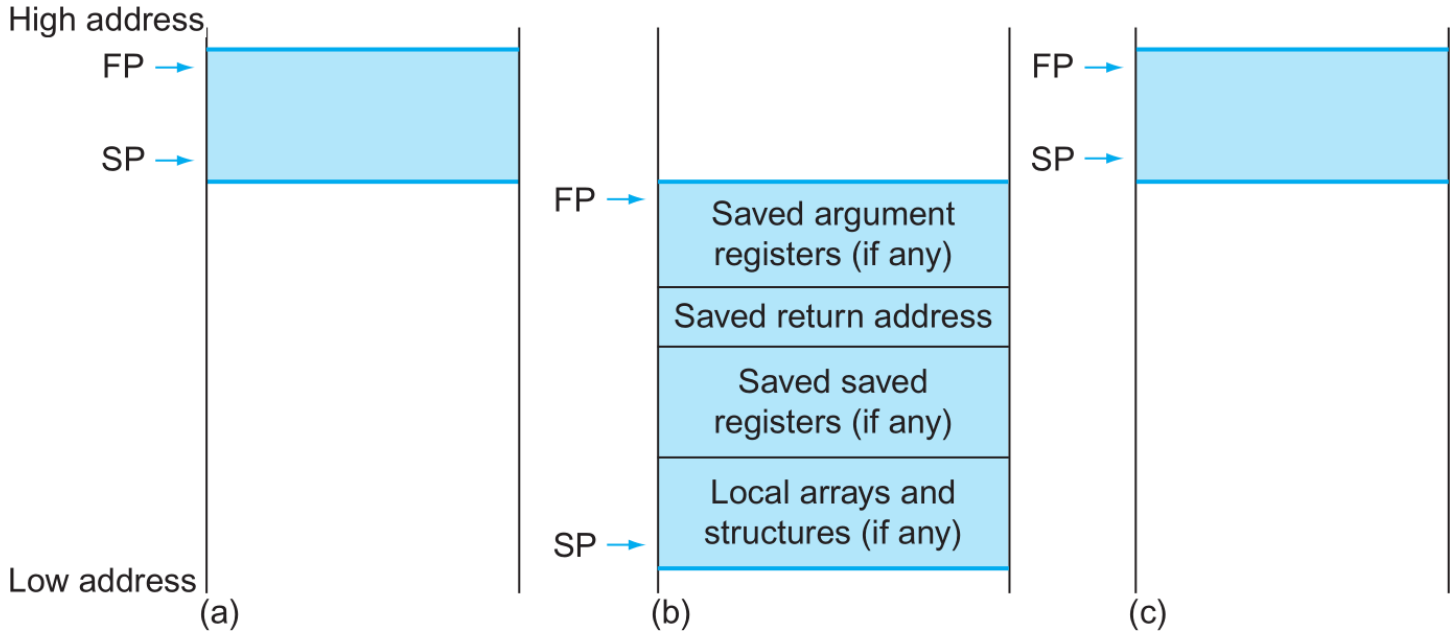
Text after // is comment.



Procedure's Local Data on the Stack

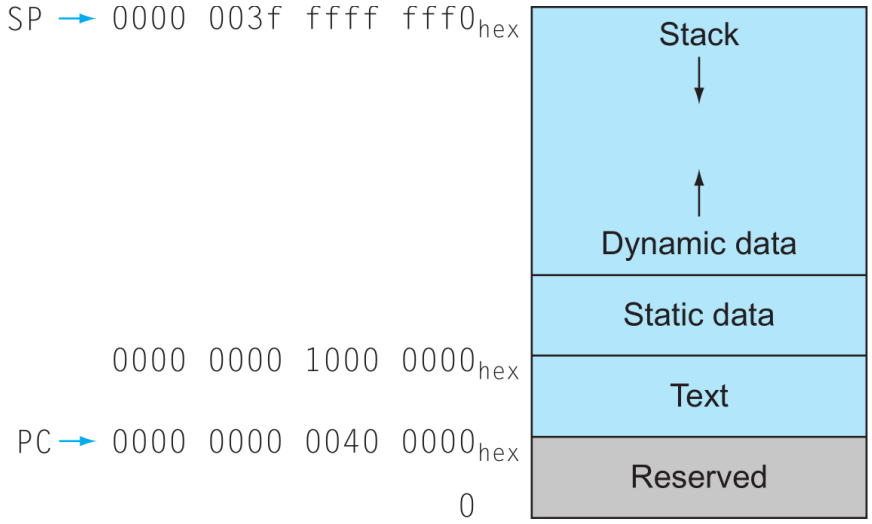
```
int fact (int n) {
   int m = 3;
   if (n<1) return (1);
   else return (m*n*fact(n-1));
}
```

- Local variables within the procedure
  - Placed into stack
- Procedure frame (= activation record)
  - Segment of stack that contains *saved registers, return address and local variables*
- Frame pointer, FP (x8) – point to the first word of the procedure frame
- SP may change during execution



Allocating Memory on the Heap

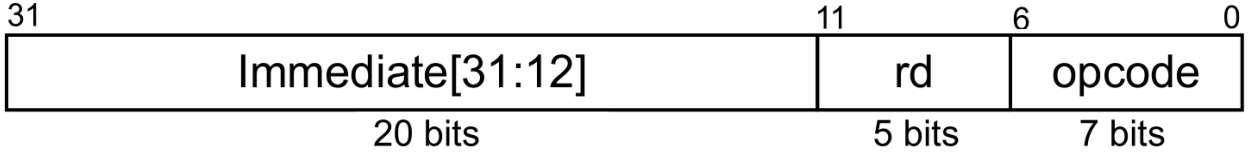
- Address space convention
  - Address space: range of addresses that can be used/allocated/referred
    - It has nothing to do with the real physical memory



- Text segment
  - text = instructions, program binaries
- Static data segment
  - constants and static variables
- Heap (for dynamic data)
  - Place where dynamically allocated memory goes to
    - linked list, tree …
  - Grows upward
  - malloc() and free() in C

Wide Immediate Operands

- There are times when constants are too big to fit into 12 bits
- RISC-V provides `lui` (*Load Upper Immediate*) instruction
  - **U-type** instruction



  - Load 20 bit constant into bit position 31~12 from a register
  - Left 32 bits are extended with 31th bit value
  - Rightmost 12 bits are filled with zeros

- How do we make this value in a register?

```
00000000 00000000 00000000 00000000 00000000 00111101 00000101 00000000
```

- lui x19, 976 // 976 = 0000 0000 0011 1101 0000

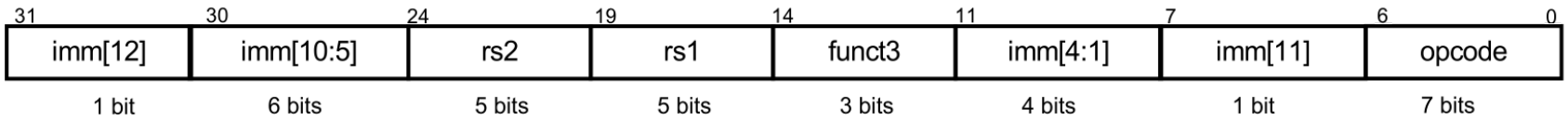
```
0000000 00000000 00000000 00000000 00000000 00111101 00000000 00000000
```

- addi x19, x19, 1280 // 1280 = 0101 00000000

```
00000000 00000000 00000000 00000000 00000000 00111101 00000101 00000000
```

Branch Instruction Format

- SB-type



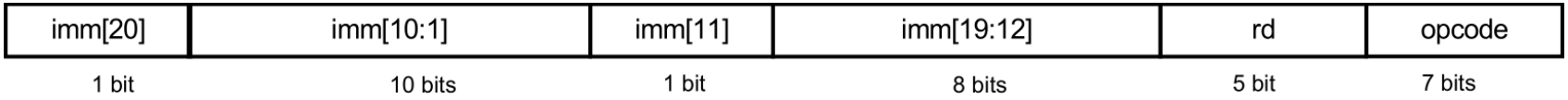
  - Can represent branch addresses from -4096 to 4094 in multiple of 2 (i.e. even address)

```
bne x10, x11, 2000    // if x10 != x11, go to location 2000 ten = 0111 1101 0000 0
```

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

  - imm[0] is assumed to be always 0, and thus not encoded
  - immediate field is 13 bits

- UJ-type



  - For unconditional branch instruction **jal** (the only instruction in this type)
  - imm[20:1] : 20-bit address immediate (signed number)
  - jal uses 21-bit address
  - imm[0] is assumed to be always zero

PC-relative Addressing

- Implication of immediate field size
  - Conditional branch: 13 bits
  - Unconditional branch: 21 bits

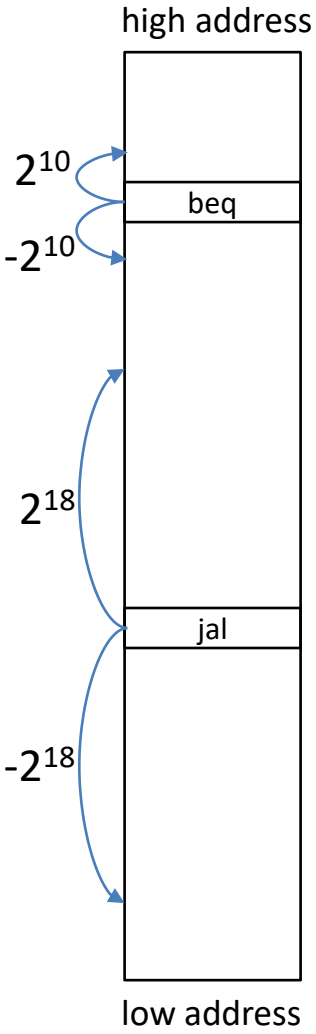
  →Program size = $2^{immediate\_field\_size}$
- new PC ← register + branch offset

  PC
  - Allow program to be 2^{64} and use branches
- PC-relative addressing enables:
  - Branch within $\pm 2^{10}$ instructions
  - Jumps within $\pm 2^{18}$ instructions
- Branching far away

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

→

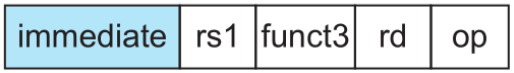
```
    bne x10, x0, L2
    jal x0, L1
L2:
```



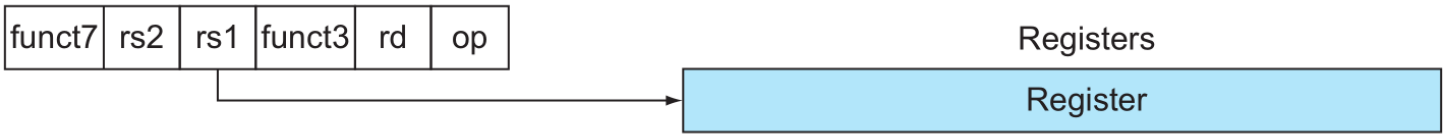
RISC-V Addressing Modes

- Addressing mode: How to locate data needed as operands

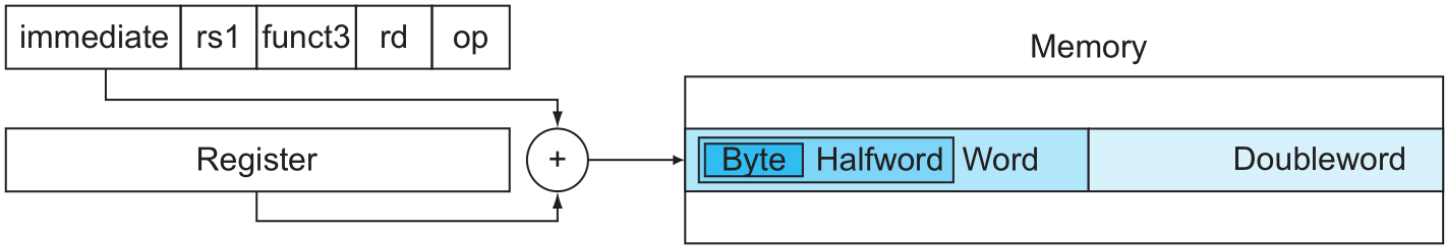
1. Immediate addressing



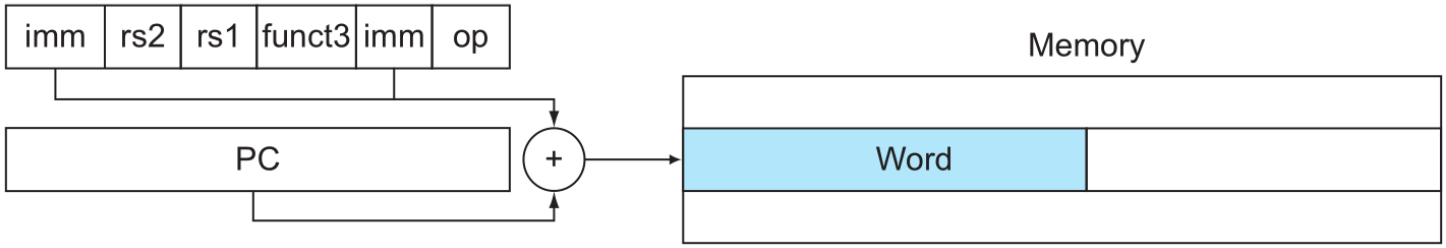
2. Register addressing



3. Base addressing



4. PC-relative addressing



Jumping Very Far Away

- RISC-V allows very long jumps to any 32-bit address
  - lui write bits 12~31 of the address to a temporary register
  - jalr adds the lower 12 bits of the address to the register and jumps to the sum

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

```
      lui x9, L1_U20
      addi x9,x9, L1_L12
      bne x10,x0, L2
      jalr x0, 0(x9)
L2:
```