EECS4030: Computer Architecture

Instructions: Language of the Computer (I)

Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University, Taiwan

(Adapted from textbook slides https://www.elsevier.com/books-and-journals/book-companion/9780128122754/lecture-slides)

Introduction

- To command a computer, you must speak its language
- Most computers understand high-level languages (HLL), e.g., C, C++, Java, with suitable compilers

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

Variables (memory locations storing data)

operands of operators

Operators

operations on operands

- How about language to a computer's hardware?
  - *Assembly language* ➔ *machine instructions*
  - Very close to control signals to control hardware operations

Outline

- Languages of computers (Sec. 2.1)
- Operations of computer hardware (Sec. 2.2)
- Operands of computer hardware (Sec. 2.3, 2.4)
  - Registers, memory, signed and unsigned numbers
- Representing instructions (Sec. 2.5)
- More operations: logic, decision making (Sec. 2.6, 2.7)
- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)

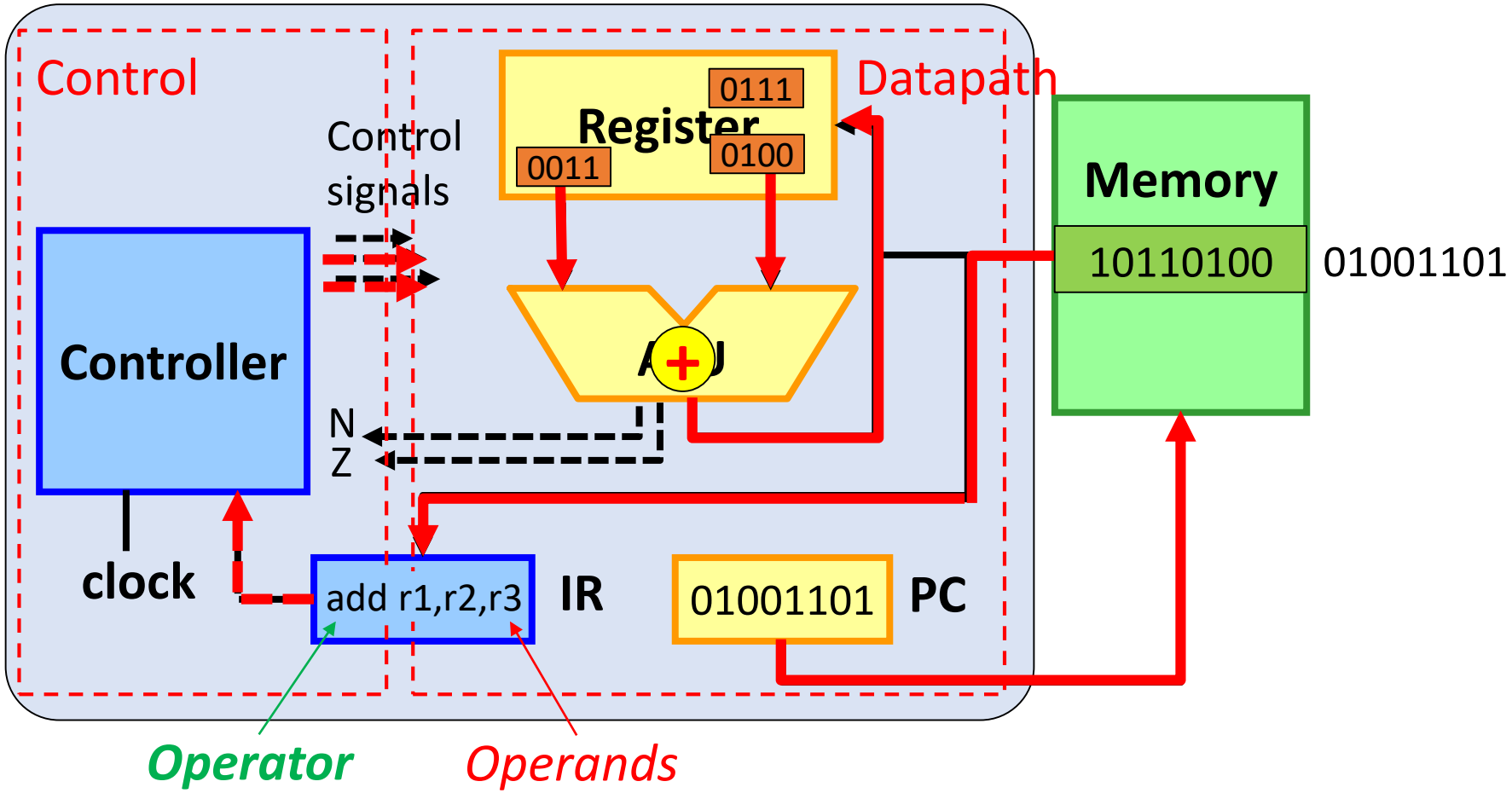
Outline

- A C sort example to put it all together (Sec. 2.13)
- Arrays versus pointers (Sec. 2.14)
- Compiling C and interpreting Java (Sec. 2.15)
- Other ISAs: MIPS, x86 (Sec. 2.16, 2.17)
- The rest of RISC-V (Sec. 2.18)
- Fallacies and pitfalls (Sec. 2.19)

Outline

- Languages of computers (Sec. 2.1)
- Operations of computer hardware (Sec. 2.2)
- Operands of computer hardware (Sec. 2.3, 2.4)
  - Registers, memory, signed and unsigned numbers
- Representing instructions (Sec. 2.5)
- More operations: logic, decision making (Sec. 2.6, 2.7)
- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)

(Command) Language to Hardware



Instruction: a sentence of the language, do something (on some data)

Language to Computer Hardware

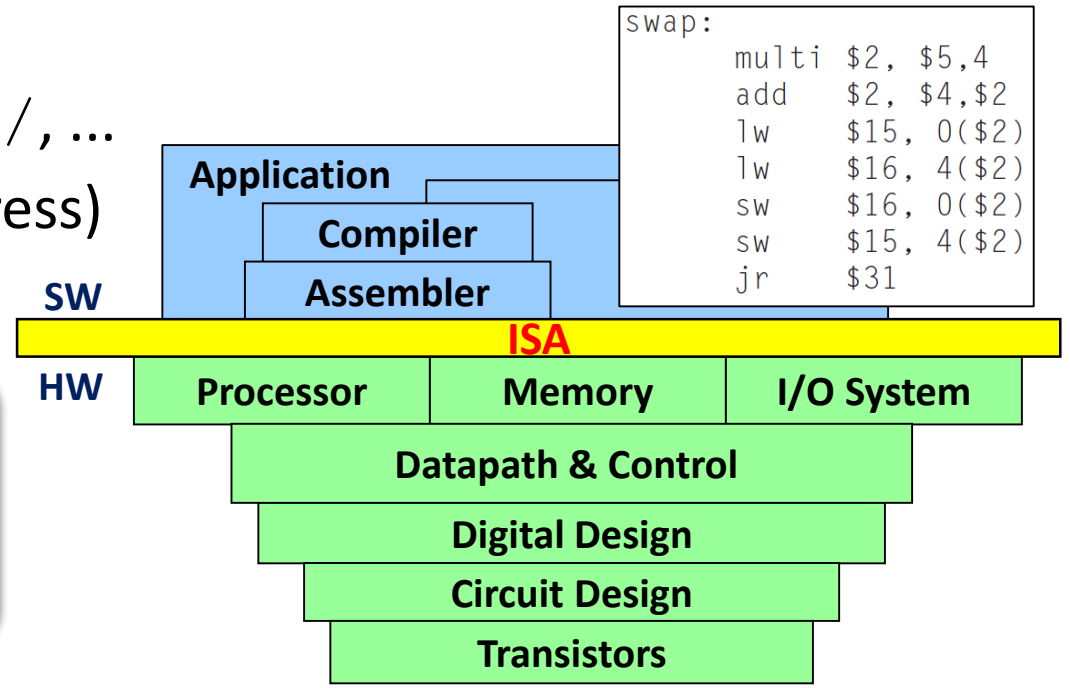
- Different computers have different languages (instruction sets)
  - But with many aspects in common
  - Because all computers are constructed from hardware technologies based on similar underlying principles
  - Because there are a few **basic** operations that all computers must provide
    - Makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and energy
    - Combinations of these basic operations allow a computer to run all sorts of programs written in various HLLs

What are the most basic operations and how to specify them?

Essentially, We Are Asking …

- What must be specified in an instruction set architecture (ISA)?
  - "ISA encompasses all the information necessary to write a machine language program that will run correctly"
  - Including:
    - operations: +, −, *, /, …
    - operands (data, address)
    - flow control

What are included in ISA and how are they specified have profound effects on computer design!



Components of an ISA

Operands: storage and data

- Organization of programmable storage:
  - Registers: # of bits, # of registers, their roles, …
  - Memory: **addressing**, access methods, …
- Data types and data structures
  - Integers, floating-point, text, signed number, …

Operations: on data and on execution flow

- Instruction set (or operation code)
  - ALU, control transfer, exceptional handling

Instruction specification:

- Instruction format and encoding

Use 64-bit RISC-V (RV64) as an Example

- Developed at UC Berkeley as open ISA
  - Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- We will not only describe what the RISC-V ISA is
- But also study the **principles** behind the design
- *And examine some different design **options** to appreciate why RISC-V chooses the specific option*

Outline

- Languages of computers (Sec. 2.1)
- Operations of computer hardware (Sec. 2.2)
- Operands of computer hardware (Sec. 2.3, 2.4)
  - Registers, memory, signed and unsigned numbers
- Representing instructions (Sec. 2.5)
- More operations: logic, decision making (Sec. 2.6, 2.7)
- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)

What Basic Operations for Hardware?

1. Operations to work on data
   - *Arithmetic* (+, -, *, /), *logic* (&, |, ~)
     → similar to a function, e.g. z=f(x,y), operating on values
   - 2-operand (a + b), 1-operand (++a)
2. Operations to move and copy data between storage locations
   - As storage, original location still retains the data, until explicitly overwritten
3. Operations to change execution flow
   - *Unconditional*: goto, jump to new location to execute
   - *Conditional*: if-then-else, jump if condition is true
   - *Procedural*: foo(), similar to goto but need to come back

Let's Start with Arithmetic Operations

- The most basic operation for any computer hardware is arithmetic operation on numbers, e.g.,

```
a = b + c
```

- Basic RISC-V arithmetic instructions are specified as:

```
add a,b,c          # a ← b + c
```

- All RISC-V arithmetic instructions have this form
  - 3-address: **a**, **b**, **c** are storage locations that hold data, but **c** may also be a value

Why storage concept indispensable?

Design Principle 1: *Simplicity favors regularity*

- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost

vs. Flexibility

Arithmetic Instruction Example

- C code:

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

- Compiled RISC-V (pseudo) code:

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

  - `t0` and `t1` are temporary variables, i.e., storage locations, normally created and allocated by the compiler

Why 3-Address Format: add a,b,c?

- Why not 2-address format?

  `add b,c` **`# b ← b + c`**

  – Still regular, and one address less to specify and encode!

  → Many 16-bit processors use this format

- How about 0-address format?

  `add`

  – Where to get the operands?

  ➔ must be specified implicitly so that the computer hardware knows where to get, e.g. in a stack

  `add #stack[top]←stack[top]+stack[next]`

Tradeoff between flexibility and complexity

Outline

- Languages of computers (Sec. 2.1)
- Operations of computer hardware (Sec. 2.2)
- Operands of computer hardware (Sec. 2.3, 2.4)
  - Registers, memory, signed and unsigned numbers
- Representing instructions (Sec. 2.5)
- More operations: logic, decision making (Sec. 2.6, 2.7)
- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)

Operands to an Operation

- Operands may be the value itself (*immdiates*) 3

  ```
  add a,b,3
  ```

  or the storage location holding the value

  ```
  add a,b,c
  ```

- The storages that can hold values may be in *memory* 2 or in *registers* 1
  - Registers are a small number of storage locations built directly into CPU hardware; memory refers to storage locations in the main memory ➔ need to specify addresses
  - How about disk? Why not disk locations as operands?
- RISC-V arithmetic instructions must use **register operands!**

Why Restricted to Register Operands?

- Why not allow memory operands, too?

```
add M[1000],M[2019],M[2020]
```

Design Principle 2: *Smaller is faster*
- Register vs. main memory (millions of locations)

  - Registers are inside CPU → same clock as CPU, connect directly to arithmetic units, short and fast signals
  - Registers are small → easy to activate to access data, short and fast signals
  - Registers are small → fewer number of bits to specify and encode ➔ smaller instructions
  - Suitable for holding frequently accessed data

How to Specify (1) Register Operands?

- Must know RISC-V register organization!
- RISC-V has a 32 × 64-bit *register file*
  - 32 registers specified as **x0** to **x31**, each with a 5-bit addr

doubleword

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

Why not 64, or 16?

Register Operand Example

- How to do the following C statement?

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

  - Assume that **f**, **g**, **h**, **i**, **j** are in **x19**, **x20**, **x21**, **x22**, **x23**

- Use intermediate temporary registers **x5**, **x6**

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

  - If data are in the memory, they must be loaded into the registers first before arithmetic operations can operate on them; after operations, they need to be stored back

IC↑
IPC↓

Use *data transfer instructions* (**ld**, **sd**, ...) discussed later

How to Specify (2) Memory Operands?

- Must know RISC-V memory organization!
  - Main memory is organized as a 1D array of locations
  - Each location can hold one byte of data and is identified by an *address* → memory is *byte addressed*, i.e. each address identifies an 8-bit byte of data, ranging from 0 to 2^{61}

- A nature way of specifying memory operands in RISC-V is like addressing an array, e.g., **`A[100]`**
  - **`A`** is the **base address** and 100 is the **offset** and memory address of **`A[100]`** is (**`A`** + 100)
  - In RISC-V, memory operand is specified as: **`100(x22)`**, where **`x22`** contains base addr. and **`100`** is offset (in bytes)
  - A memory operand normally gets a doubleword (64 bits), and thus word addresses differ by 8

Memory Operand Example

- C code:

```
A[9] = h + A[8];
```

  - **h** in **x21**, *base address* of **A** in **x22**
  - **A[8]**: the 8th doubleword in array **A[]**, which is at the 64th byte from start of **A[]**

- Compiled RISC-V code:

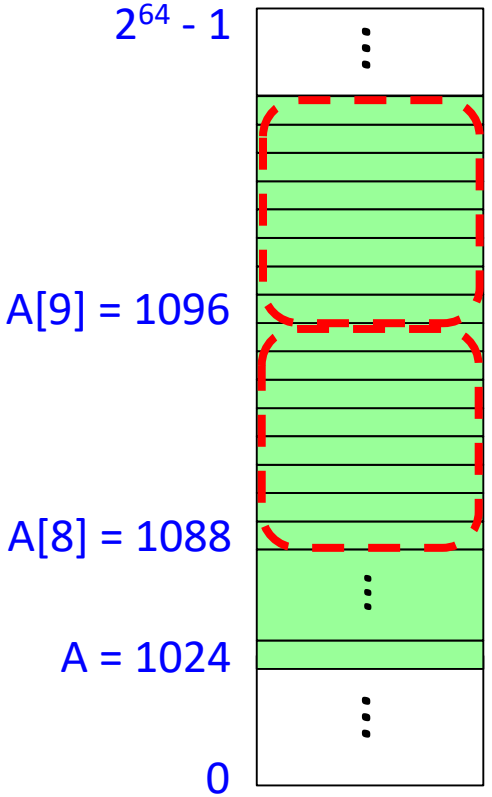
```
ld  x9, 64(x22)
add x9, x21, x9
sd  x9, 72(x22)
```

load doubleword

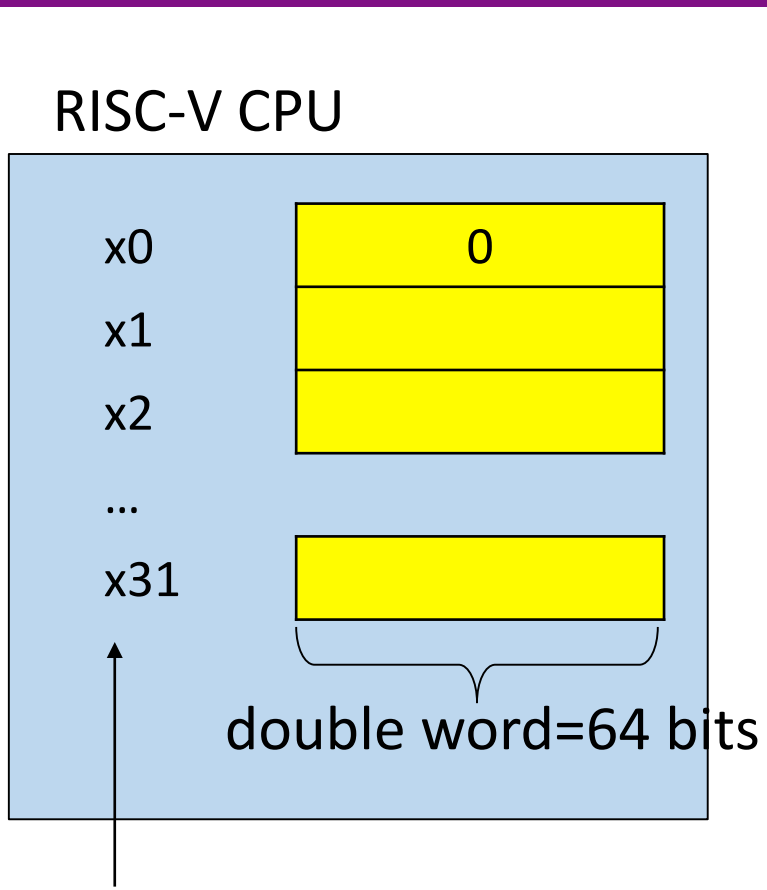
store doubleword

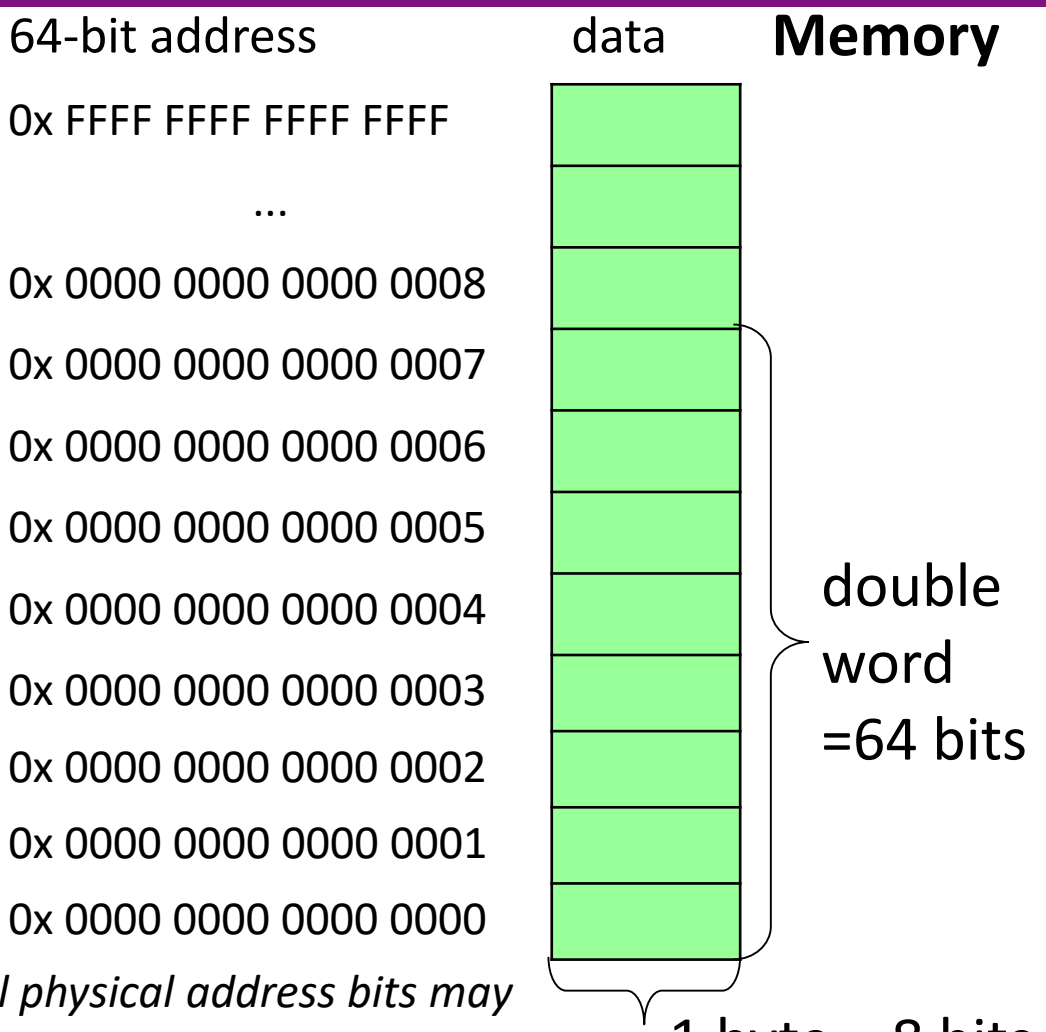
offset

base register



Registers and Memory Size

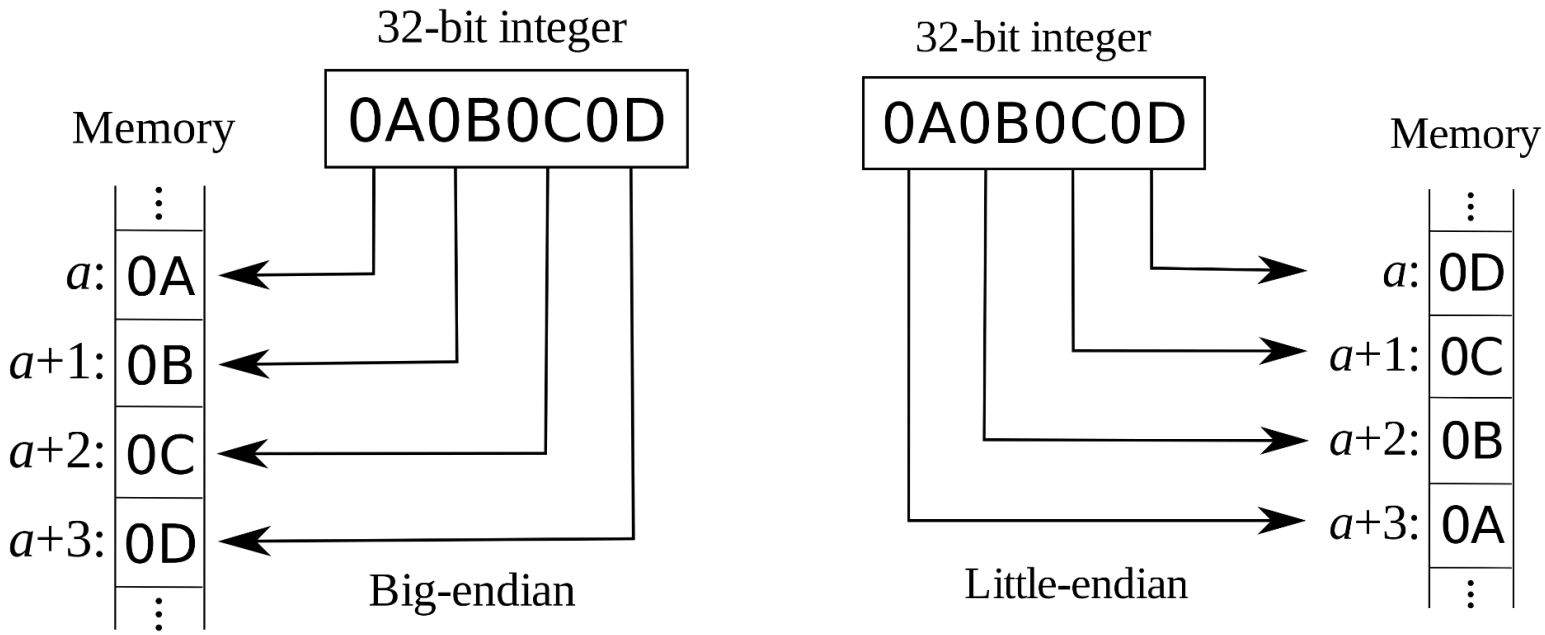




Note: *actual physical address bits may be fewer than 64, e.g., 42 for Intel x86_64*

More about RISC-V Memory Organization

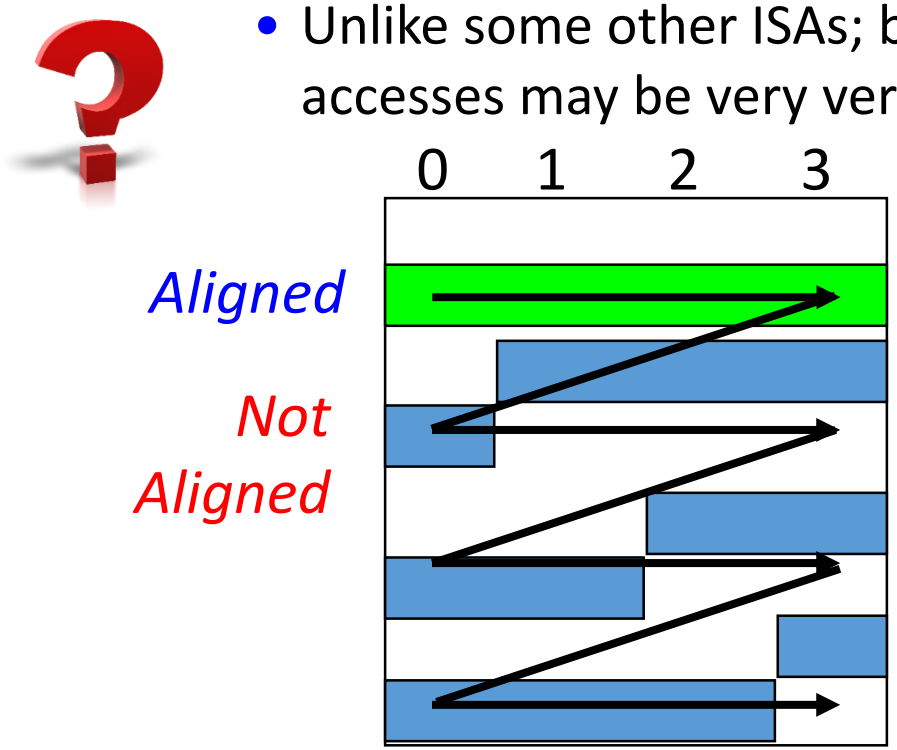
- RISC-V is *Little Endian*
  - Least-significant byte at least address of a word
  - c.f. Big Endian: most-significant byte at least address

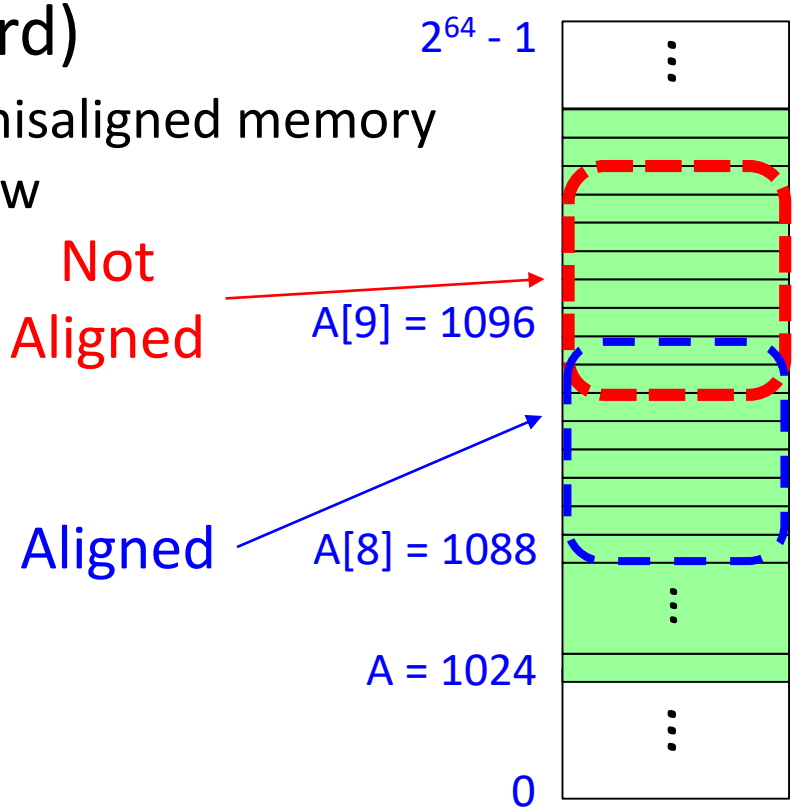


(By R. S. Shaw - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2974661)

More about RISC-V Memory Organization

- RISC-V does not require words to be aligned in memory, i.e., at an address that is a multiple of 4 (for a word) or 8 (for a doubleword)
  - Unlike some other ISAs; but misaligned memory accesses may be very very slow





Registers vs. Memory

- Registers are faster to access than memory
  - Also, operating on memory data requires loads and stores
    - More instructions to be executed, larger code size → IC↑
- Compilers should use registers for variables as much as possible → IPC↓
  - Only *spill* to memory for less frequently used variables, if more variables than available registers are used
  - Register optimization is important!
- Why not keep all variables in memory?
  - Smaller is faster: registers are faster than memory
  - Registers are more versatile:
    - Register file can read 2 and write 1 registers per instruction
    - Memory read/write 1 operand per instruction, no operation

How about (3) Immediate Operands?

- Small constants used frequently (50% of operands)

  e.g., A = B + 4; if (C > 1) … ;

- Put 'typical constants' in memory and load them

```
ld   x9, AddrConstant4(x3)
add  x22, x22, x9
```

Too slow

- Alternative: constant data specified in the instruction

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

Immediate operand

Design Principle 3: *Make the common case fast*

- Small constants are common
- Immediate operand avoids a load instruction

Why define a new instruction, **addi**? Why not reuse **add**?

Handling Constant Zero and Negatives

- The number zero (0) is so common in code that RISC-V hardwired register 0 (**`x0`**) to 0
  - Cannot be overwritten, e.g., **`addi x0,x0,5`** (no effect)
  - Useful for other common operations, too, e.g., move between registers: **`add x22,x20,x0`**
    - Thus, there is no need to implement a move instruction in hardware → simpler CPU design and implementation

- No need for subtract immediate instruction either
  - Just use a negative constant: **`addi x22, x22, -8`**

Time to review signed and unsigned numbers

Represent Unsigned Binary Integers

- Given an *n*-bit number

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

- Use all the bits to represent **values**
- Example: (32 bits)

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$

  $= 0\times2^{31} + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$

  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- Range: 0 to $+2^n - 1$
  - Using 32 bits: 0 to +4,294,967,295
  - Using 64 bits: 0 to +18,446,774,073,709,551,615

Can get the max value

Signed Binary Integers (2's-Complement)

- Given an *n*-bit number

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

- Use bit 31/63 to represent sign (*sign bit*)
  - 1 for negative and 0 for non-negative numbers
- Signed negation: complement (1 → 0, 0 → 1) & add 1
  - Non-negative numbers have the same representation
  - $-(-2^{n-1})$ can't be represented
- Example: (32 bits)

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

  $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$

  $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

2's-Complement Signed Integers

- Range: -2^{n-1} to $+2^{n-1} - 1$
  - Using 32 bits: –2,147,483,648 to +2,147,483,647
  - Using 64 bits: –9,223,372,036,854,775,808 (1000 … 0000_2) to 9,223,372,036,854,775,807 (0111 1111 … 1111_2)
  - 0: 0000 0000 … 0000_2
  - –1: 1111 1111 … 1111_2 (least negative number)
  - Most-negative number (1000 … 0000_2) does not have a corresponding positive number

One adder, no distinction

SW interprets meaning

Signed/unsigned +/- treated the same!

Sign Extension

- Representing a number using more bits while preserving the numeric value
- How to do? replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 ➔ 0000 0000 0000 0010
  - –2: 1111 1110 ➔ 1111 1111 1111 1110
- In RISC-V instruction set
  - **lb**: load byte (sign-extend loaded byte)
  - **lbu**: load byte unsigned (zero-extend loaded byte)

Sign?

```
lb  x22,0(x21)
```

FFFFFF F7

```
lbu x22,0(x21)
```

000000 F7

Outline

- Languages of computers (Sec. 2.1)
- Operations of computer hardware (Sec. 2.2)
- Operands of computer hardware (Sec. 2.3, 2.4)
  - Registers, memory, signed and unsigned numbers
- Representing instructions (Sec. 2.5)
- More operations: logic, decision making (Sec. 2.6, 2.7)
- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)

How to Specify RISC-V Instructions?

- Now that we have learned *operations* and *operands* of computer hardware, let us study how to put them together to form computer instructions
  - Remember: Computer only understands 1s and 0s, so "`add x21,x22,x23`" is meaningless to hardware
  - Instructions to instruct computer hardware how to operate should also be encoded in binary, called *machine code*
- RISC-V's strategies for instruction encoding
  - Encode every instruction in 32-bit words, divided into *fields*
  - Small number of formats encoding *operation code* (*opcode*), register numbers, …
- Let's look at three of them first

Regularity!

Why not just 1 format?

RISC-V R-format Instructions

- For register to register operations
- 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)

Why not combine op and funct?

R-format Example

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

`add x9,x20,x21`

0	x21	x20	0	x9	add
0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

RISC-V I-format Instructions

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or byte offset to base addr
    - 2's-complement, sign extended, -2^{11} to $2^{11}-1$

Design Principle 4: *Good design demands good compromises*

- Different formats complicate decoding, but allow 32-bit instructions uniformly
- Keep formats as similar as possible (only 1 field differs)

I-Format Example 1

- RISC-V instruction:

```
addi    x9,x22,-51
```

  - opcode = 19 (look up in RISC-V Manual)
  - rs1 = 22 (source operand register number)
  - rd = 9 (destination register number)
  - immediate = -51 (by default, this is decimal)
- Decimal representation:

-51	22	0	9	19

- Binary representation:

1111 1100 1101	10110	000	01001	0010011

I-Format Example 2

- RISC-V instruction:

```
ld    x9,120(x22)
```

  - opcode = 3 (look up in RISC-V Manual)
  - rs1 = 22 (base address register number)
  - rd = 9 (destination register number)
  - immediate = 120 (offset)
- Decimal representation:

120	22	3	9	3

- Binary representation:

0000 0111 1000	10110	011	01001	0000011

RISC-V S-format Instructions

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
- RISC-V instruction:

```
sd    x9,120(x22)
```

120(1)	9	22	3	120(2)	35
0000011	01001	10110	011	11000	0100011

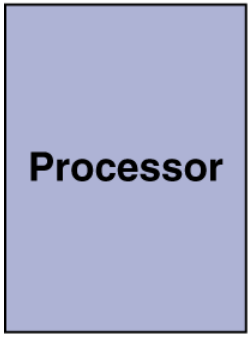
Key Computer Design Principles

- Computer instructions represented as numbers
- Program instructions can be stored at the same place as data (in memory) and be manipulated in the same way as data ➔ **stored-program computer**
  - Instructions and data look the same, just 0s and 1s
  - Differ only in how you interpret the bits
- One consequence: everything addressed
  - Everything has a memory address: instructions, data
    - Both branches and jumps use these
  - One register keeps address of the instruction being executed: "Program Counter" (PC)
  - Register can hold any 64-bit value, (signed) int, address, etc.

Key to programmable general-purpose

Stored-Program Computers

- Consequence #2:
  programs can operate on programs
  - e.g., compilers, linkers, …
  - The latter is data to the former
- Consequence #3:
  binary compatibility
  - Programs can be shipped as files of binary numbers and executed on any computers with a compatible ISA
- Consequence #4:
  fast context switches of processes



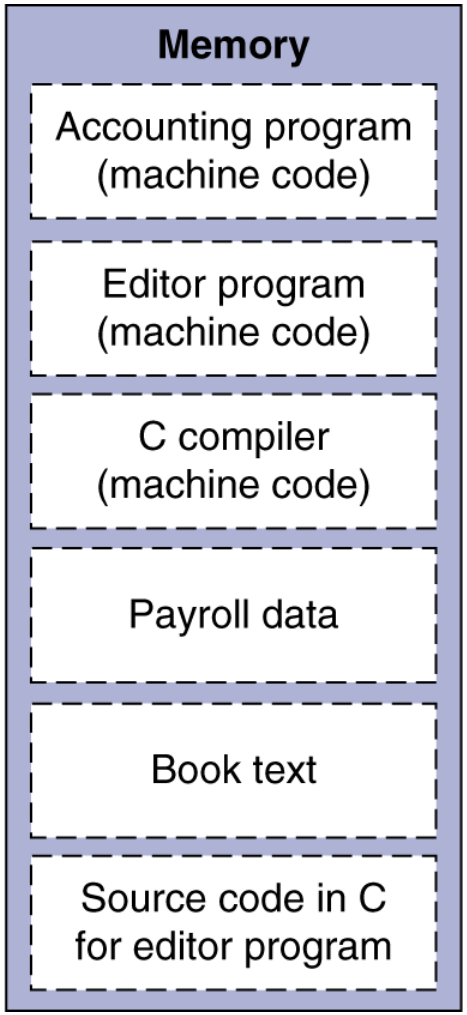
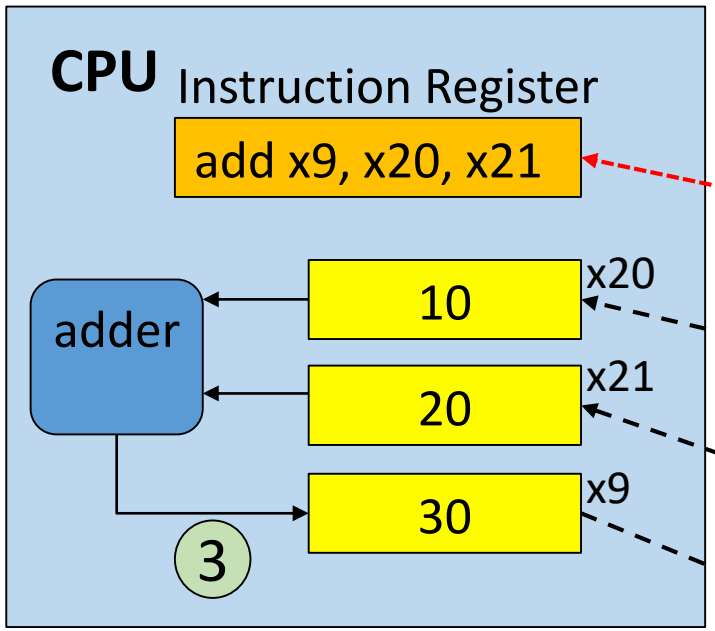
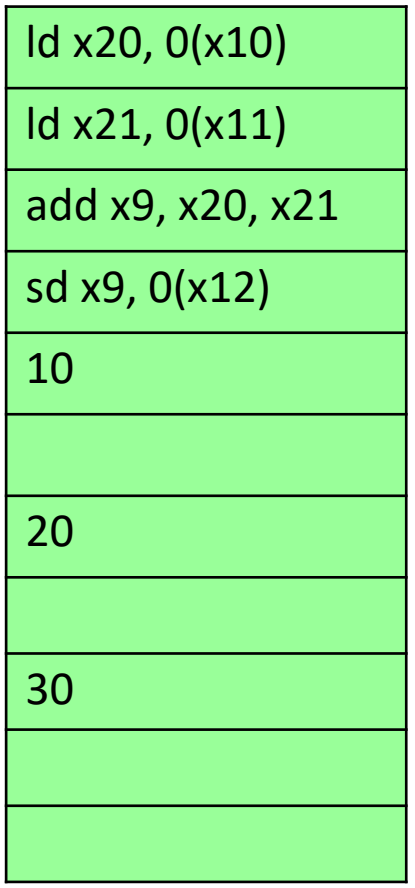


Fig. 2.7

Stored Program with Memory Access





a ← b + c

(1) load b to x20=10

(2) load c to x21=20

(3) add x9=x20+x21

(4) store x9 to a

Assume

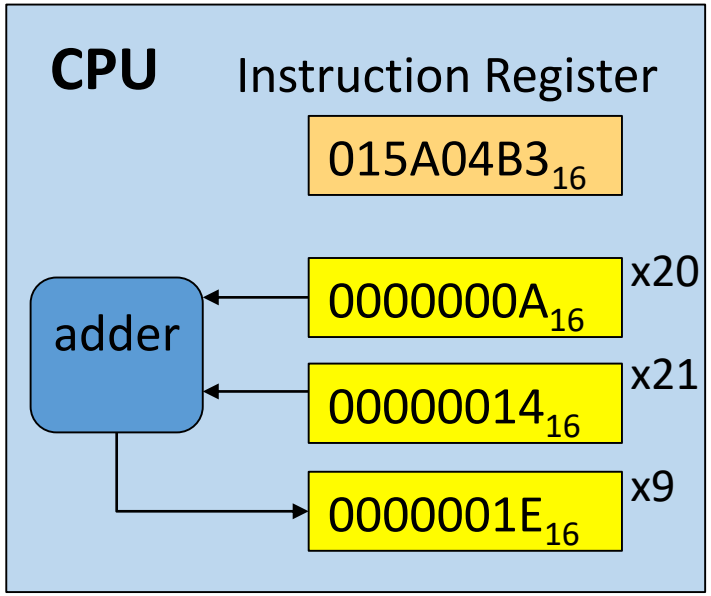
x10=b's address

x11=c's address

x12=a's address

(Prof. Jing-Jia Liou)

Instruction and Data Encoding (in Hex)



Memory

address	instruction/data
	$00053A03_{16}$
	$0005BA83_{16}$
	$015A04B3_{16}$
	00963023_{16}
b	$0000000A_{16}$
c	00000014_{16}
a	$0000001E_{16}$
...	

a ← b + c

(1) load b to x20=10
(2) load c to x21=20
(3) add x9=x20+x21
(4) store x9 to a

Assume
x10=b's address
x11=c's address
x12=a's address

(Prof. Jing-Jia Liou)

Outline

- Languages of computers (Sec. 2.1)
- Operations of computer hardware (Sec. 2.2)
- Operands of computer hardware (Sec. 2.3, 2.4)
  - Registers, memory, signed and unsigned numbers
- Representing instructions (Sec. 2.5)
- More operations: logic, decision making (Sec. 2.6, 2.7)
- Supporting procedures in hardware (Sec. 2.8)
- More operand representations (Sec. 2.9, 2.10)
  - Characters, wide immediates and addresses
- Parallelism and instructions (Sec. 2.11)
- Translating and starting a program (Sec. 2.12)

Bitwise Operations

- Up until now, we have seen arithmetic (**`add`**, **`addi`**) and memory access (**`ld`**, **`sd`**)
  - All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)
- New perspective: View contents of register as 64 individual bits rather than as a single 64-bit number
  - Since registers are composed of 64 bits, we may want to access individual bits rather than the whole
- Introduce two new classes of instructions:
  - Logical operators
  - Shift instructions

Logical and Shift Operations

- Instructions for bitwise manipulation
  - Useful for extracting and inserting groups of bits in a word

Operation	C	Java	RISC-V
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arith.	>>	>>	sra, srai
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise XOR	^	^	xor, xori
Bitwise NOT	~	~	

Fig. 2.8

Shift Operations

- sll: shift left logical (R-type)
  - Shift rs1 left for rs2 bits and fill rs1 with 0 bits
  - Shift left by *i* bits = multiplies by 2^i (faster than multiply)

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

- Example: **`sll x9,x20,x21`**

0	x21	x20	1	x9	sll
0000000	10101	10100	001	01001	0110011

  - **`x20`** = 0000 … 0001; **`x21`** = 0000 … 0011 = 3
  - After instruction executed, **`x9`** will contain 0000 … 1000

Shift Operations

- srl: shift right logical (R-type)
  - Shift rs1 right for rs2 bits and fill rs1 with 0 bits
  - Shift right by *i* bits = divides by 2^i (unsigned only)
    (much faster than division, used in compiler optimization)
  - Example: **`srl x9,x20,x21`**

0000000	10101	10100	101	01001	0110011

- sra: shift right arithmetic (R-type)
  - Shift rs1 right and fill with sign bits (no **`sla`**!)
  - Shift right by *i* bits = divides by 2^i (signed)
  - Example: **`sra x9,x20,x21`**

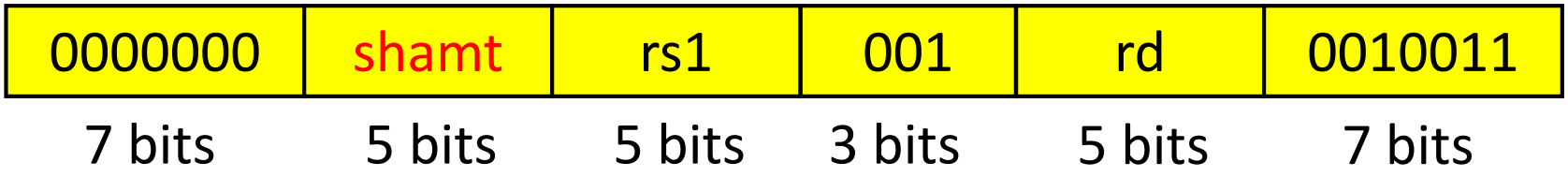
0100000	10101	10100	101	01001	0110011

Note the encoding

  - **`x20`** = 1111 … 1000; **`x21`** = 0000 … 0011 = 3
  - After instruction executed, **`x9`** will contain 1111 … 1111

Shift Immediate Operations

- slli: shift left logical immediate (32-bit RISC-V: RV32)



  - *shamt*: how many bits to shift
    - 32-bit data can only be shifted 32 bit positions → 5-bit shamt
    - Same format as R-type, except rs2 becomes shamt
- slli: how about 64-bit RISC-V (RV64)?
  - We can shift 64 bit positions and need 6 bits for shamt
  - How?

Logic Operations

- **`and`**: useful for setting certain portions of a word to 0s, while leaving the rest alone → *masking*
  - Example: **`and x9,x10,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`**: useful for forcing certain bits of a word to 1s, while leaving others unchanged
  - Example: **`or x9,x10,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

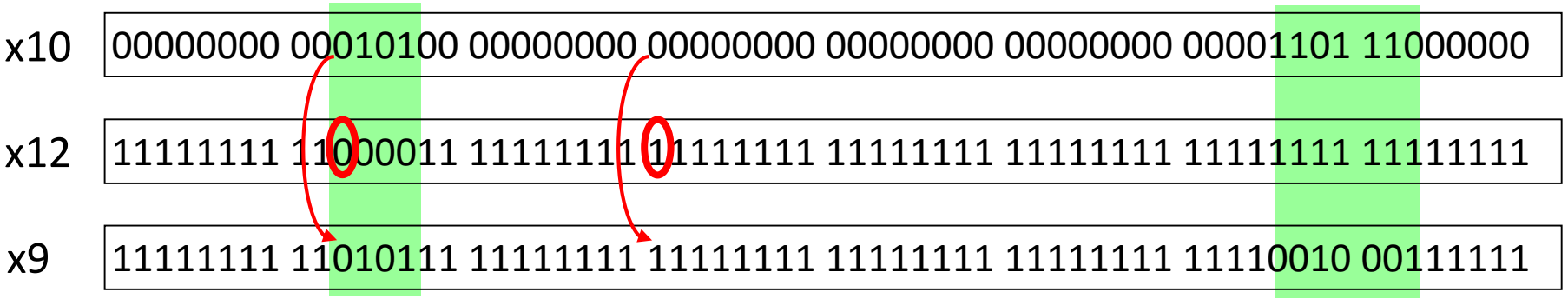
Logic Operations

XOR

0	0	0
0	1	1
1	0	1
1	1	0

- **`xor`**: differencing operation
  - Set some bits to 1, leave others unchanged

```
xor x9,x10,x12
```



  - If mask (x12) = 1, 0 → 1 and 1 → 0
  - If mask (x12) = 0, 0 → 0 and 1 → 1
  - So, if mask is all 1s, XOR becomes a NOT operation
    → no need for a **`not`** instruction (2-address) in RISC-V

So Far...

- All instructions studied so far have allowed us to manipulate data
- So we have built a calculator
- However, in order to build a computer, we need the ability to make decisions and change the execution flow

Conditional Branch Operations

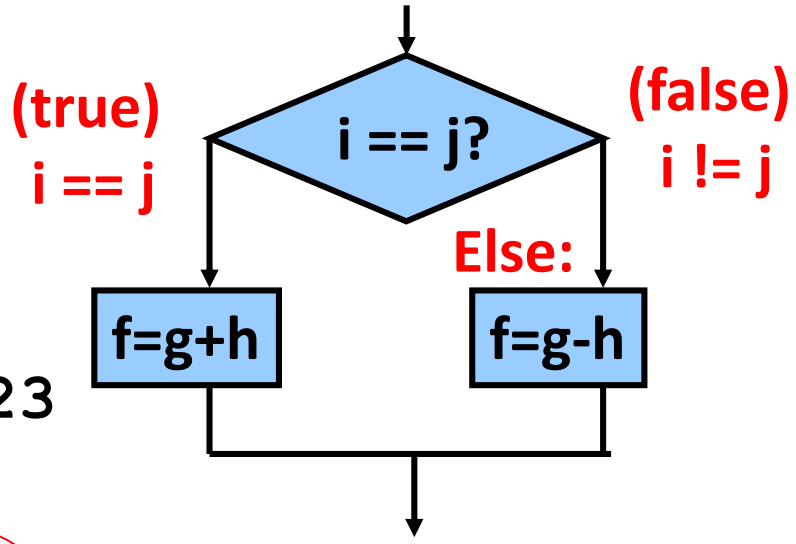
- Branch to a *labeled instruction* if a condition is true
  - Otherwise, continue sequentially
- **`beq rs1,rs2,L1`** ← Memory address
  - if (rs1 == rs2) branch to instruction labeled L1, i.e., load memory address of that instruction to PC (Program Counter), so that the CPU will next fetch and execute the branch-to instruction
- **`bne rs1,rs2,L1`**
  - If (rs1 != rs2) branch to instruction labeled L1 by storing L1 (an address) into PC

Compiling If Statements

- C code:

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

  - f, g, h, i, j in x19, x20, x21, x22, x23



- Compiled RISC-V code:

```
        bne x22,x23,Else
        add x19,x20,x21
        beq x0,x0,Exit  // unconditional
Else:   sub x19,x20,x21
Exit:   …
```

Why not unconditional jump?

Assembler calculates address (discussed later)

Compiling Loop Statements

- C code:

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

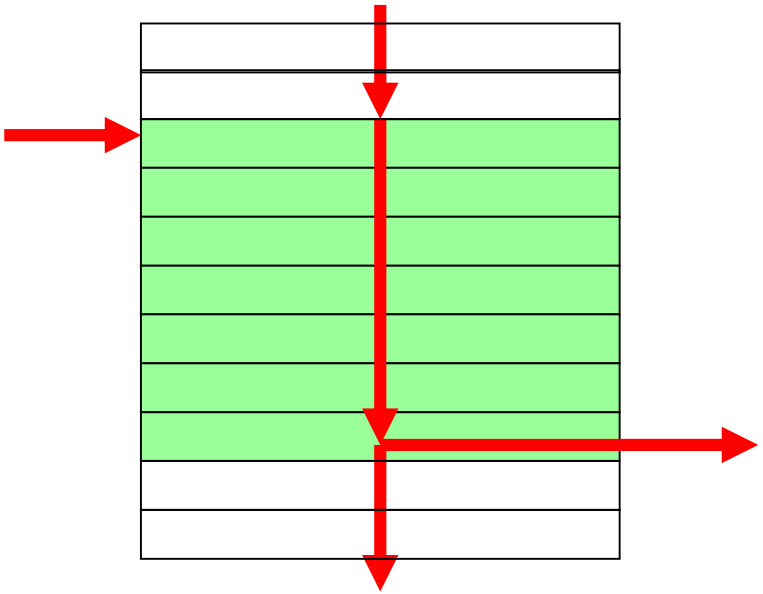
  - `i` in `x22`, `k` in `x24`, base address of `save[]` in `x25`

- Compiled RISC-V code:

```
Loop: slli x10,x22,3    // x10←i*8
      add  x10,x10,x25  // x10←save[i]
      ld   x9,0(x10)
      bne  x9,x24,Exit
      addi x22,x22,1
      beq  x0,x0,Loop
Exit: …
```

Concept of Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)
- A compiler identifies basic blocks for optimization
  - All instructions in a basic block will surely be executed once the basic block is entered
    → can rearrange execution seq.
- An advanced processor can accelerate execution of basic blocks with hardware scheduling



More Conditional Branch 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:
```

Signed vs. Unsigned Comparison

- Signed comparison: **`blt, bge`**
- Unsigned comparison: **`bltu, bgeu`**
- Example:
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - **`blt  x22,x23,L1  # signed`**

    –1 < +1 ⇒ branch to L1
  - **`bltu x22,x23,L1  # unsigned`**

    +4,294,967,295 > +1 ⇒ do not branch

Branch Instruction Design

- Alternative 1: (MIPS approach)
  - Set a register based on comparison result (e.g., **`slt`**), then branch on the value in that register with **`beq`** or **`bne`**
  - Pro: hardware for <, ≥, … slower than =, ≠, because combining with branch involves more work per instruction, requiring a slower clock → all instructions penalized!
  - Con: need two instructions instead of one
- Alternative 2: (ARM approach)
  - Keep extra bits (*condition codes* or *flags*) that record what occurred during an instruction, e.g., overflow, negative, …
  - Conditional branches then test on condition codes
  - Con: will create dependencies on condition codes