Chapter 2: Instructions: Language of the Computer

2.1 – 2.3 Introduction, Operations and Operands, 2.4 – 2.5 Signed and Unsigned Numbers, Representing Instructions in the Computer

ITSC 3181, Introduction to Computer Architecture

https://passlab.github.io/ITSC3181/

Department of Computer Science

Yonghong Yan

yyan7@uncc.edu

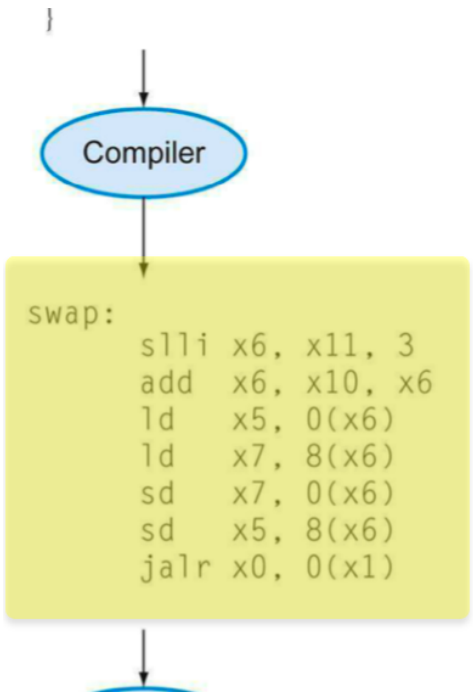
https://passlab.github.io/yanyh/

Chapter 2: Instructions: Language of the Computer

- ☛ Lecture
  - 2.1 Introduction
  - 2.2 Operations of the Computer Hardware
  - 2.3 Operands of the Computer Hardware
- Lecture
  - 2.4 Signed and Unsigned Numbers
  - 2.5 Representing Instructions in the Computer
- Lecture
  - 2.6 Logical Operations
  - 2.7 Instructions for Making Decisions
- Lecture
  - 2.8 Supporting Procedures in Computer Hardware
  - 2.9 Communicating with People
  - 2.10 RISC-V Addressing for Wide Immediate and Addresses
- Lecture
  - ~~2.11 Parallelism and Instructions: Synchronization~~
  - ~~2.12 Translating and Starting a Program~~
    - We covered before along with C Basics
  - 2.13 A C Sort Example to Put It All Together
  - 2.14 Arrays versus Pointers
    - We covered most before along with C Basics
  - ~~2.15 Advanced Material: Compiling C and Interpreting Java~~
  - 2.16 Real Stuff: MIPS Instructions
  - 2.17 Real Stuff: x86 Instructions
  - ~~2.18 Real Stuff: The rest of RISC-V~~
  - ~~2.19 Fallacies and Pitfalls~~
  - 2.20 Concluding Remarks
  - ~~2.21 Historical Perspective and Further Reading~~

Instruction Set

- The repertoire of instructions of a computer
- **Different computers have different instruction sets**
  - **But with many aspects in common**
- **Early computers had very simple instruction sets**
  - **Simplified implementation**
- Many modern computers also have simple instruction sets

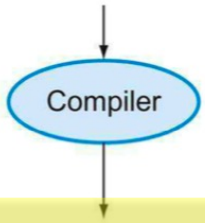



```
Disassembly of section __TEXT,__text:
_swap:
       0:        55         pushq   %rbp
       1:        48 89 e5           movq    %rsp, %rbp
       4:        48 89 7d f8        movq    %rdi, -8(%rbp)
       8:        89 75 f4           movl    %esi, -12(%rbp)
       b:        48 8b 7d f8        movq    -8(%rbp), %rdi
       f:        48 63 45 f4        movslq  -12(%rbp), %rax
      13:        8b 34 87           movl    (%rdi,%rax,4), %esi
      16:        89 75 f0           movl    %esi, -16(%rbp)
      19:        48 8b 45 f8        movq    -8(%rbp), %rax
      1d:        8b 75 f4           movl    -12(%rbp), %esi
      20:        83 c6 01           addl    $1, %esi
      23:        48 63 fe           movslq  %esi, %rdi
      26:        8b 34 b8           movl    (%rax,%rdi,4), %esi
      29:        48 8b 45 f8        movq    -8(%rbp), %rax
```

RISC-V and X86_64 Assembly Example

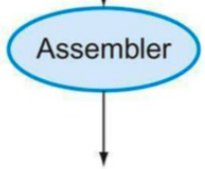
High-level language program (in C)

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



Assembly language program (for RISC-V)

```
swap:
        slli x6, x11, 3
        add  x6, x10, x6
        ld   x5, 0(x6)
        ld   x7, 8(x6)
        sd   x7, 0(x6)
        sd   x5, 8(x6)
        jalr x0, 0(x1)
```



Binary machine language program (for RISC-V)

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000011001001010000011
00000001000001100110011110000011
00000000111001100110000000100011
00000000101001100110100001000011
00000000000000001000000001100111
```

```
MacBook-Pro-8:exercises yanyh$ gcc -c swap.c
MacBook-Pro-8:exercises yanyh$ objdump -D swap.o

swap.o: file format Mach-O 64-bit x86-64

Disassembly of section __TEXT,__text:
_swap:
       0:       55      pushq   %rbp
       1:       48 89 e5        movq    %rsp, %rbp
       4:       48 89 7d f8     movq    %rdi, -8(%rbp)
       8:       89 75 f4        movl    %esi, -12(%rbp)
       b:       48 8b 7d f8     movq    -8(%rbp), %rdi
       f:       48 63 45 f4     movslq  -12(%rbp), %rax
      13:       8b 34 87        movl    (%rdi,%rax,4), %esi
      16:       89 75 f0        movl    %esi, -16(%rbp)
      19:       48 8b 45 f8     movq    -8(%rbp), %rax
      1d:       8b 75 f4        movl    -12(%rbp), %esi
      20:       83 c6 01        addl    $1, %esi
      23:       48 63 fe        movslq  %esi, %rdi
      26:       8b 34 b8        movl    (%rax,%rdi,4), %esi
      29:       48 8b 45 f8     movq    -8(%rbp), %rax
      2d:       48 63 7d f4     movslq  -12(%rbp), %rdi
      31:       89 34 b8        movl    %esi, (%rax,%rdi,4)
      34:       8b 75 f0        movl    -16(%rbp), %esi
      37:       48 8b 45 f8     movq    -8(%rbp), %rax
      3b:       8b 4d f4        movl    -12(%rbp), %ecx
      3e:       83 c1 01        addl    $1, %ecx
      41:       48 63 f9        movslq  %ecx, %rdi
      44:       89 34 b8        movl    %esi, (%rax,%rdi,4)
      47:       5d      popq    %rbp
      48:       c3      retq
```

The RISC-V Instruction Set

- Used as the example throughout the book
  - We will use and study only three classes of instructions for a handful of ins
  - Sufficient for most programs.
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage, cameras, printers, …

- Other Instruction Set Architectures:
  - X86 and X86_32: Intel and AMD, main-stream desktop/laptop/server
  - ARM: smart phone/pad
  - RISC-V: emerging and free ISA, closer to MIPS than other ISAs
    - The same textbook in RISC-V version
  - Others: Power, SPARC, etc

RISC vs. CISC

- Design "philosophies" for ISAs: RISC vs. CISC
  - CISC = Complex Instruction Set Computer
    - X86, X86_64 (Intel and AMD, main-stream desktop/laptop/server)
    - X86* internally are still RISC
  - RISC = Reduced Instruction Set Computer
    - ARM: smart phone/pad
    - RISC-V: free ISA, closer to MIPS than other ISAs, the same textbook in RISC-V version
    - Others: Power, SPARC, etc
- Tradeoff:

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- RISC:
  - Small instruction set
    - Easier for compilers
  - Limit each instruction to (at most):
    - three register accesses,
    - one memory access,
    - one ALU operation
    - => facilitates parallel instruction execution (ILP)
  - Load-store machine: minimize off-chip access

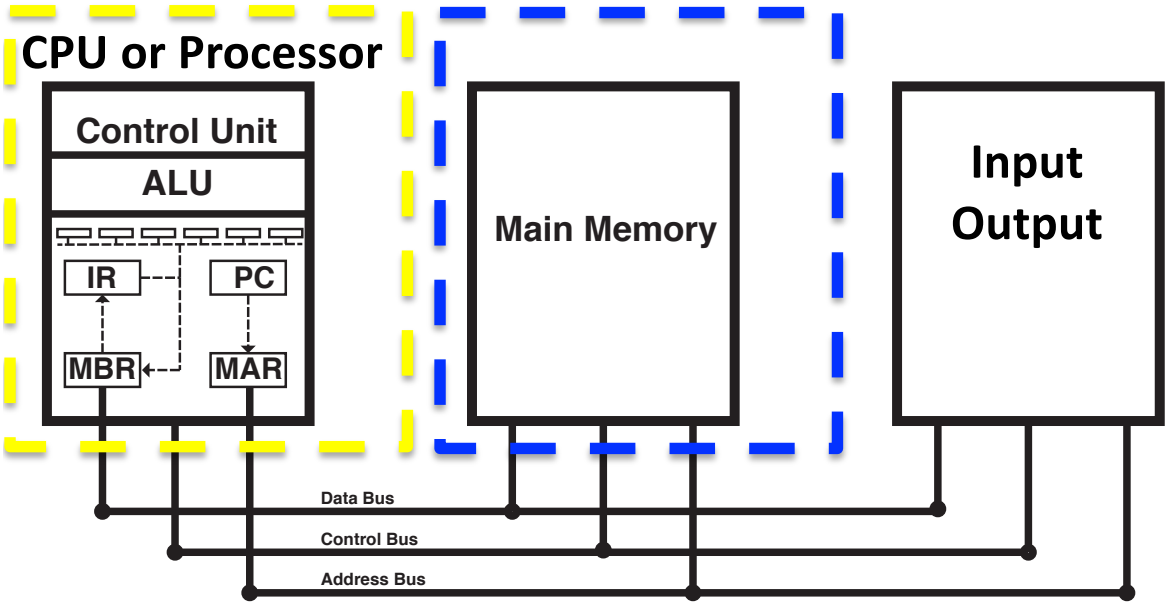
We Will Study Three Classes of Instructions

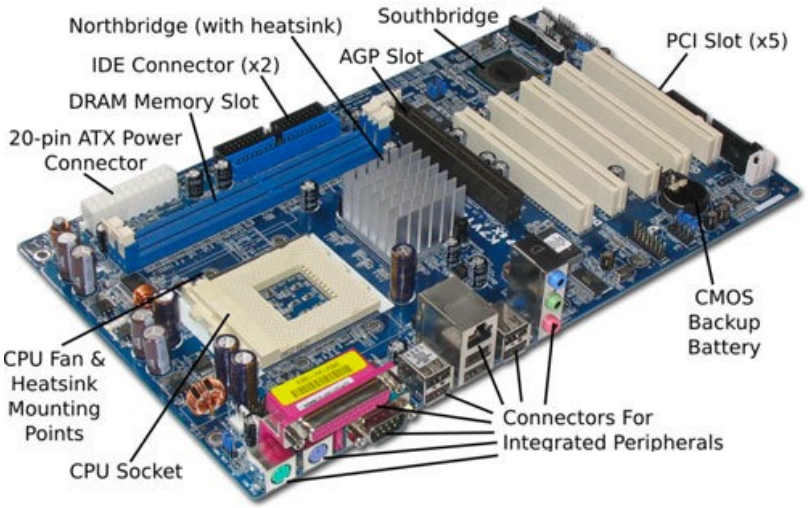
1. Arithmetic-logic instructions
   - add, sub, addi, and, or, shift left|right, etc

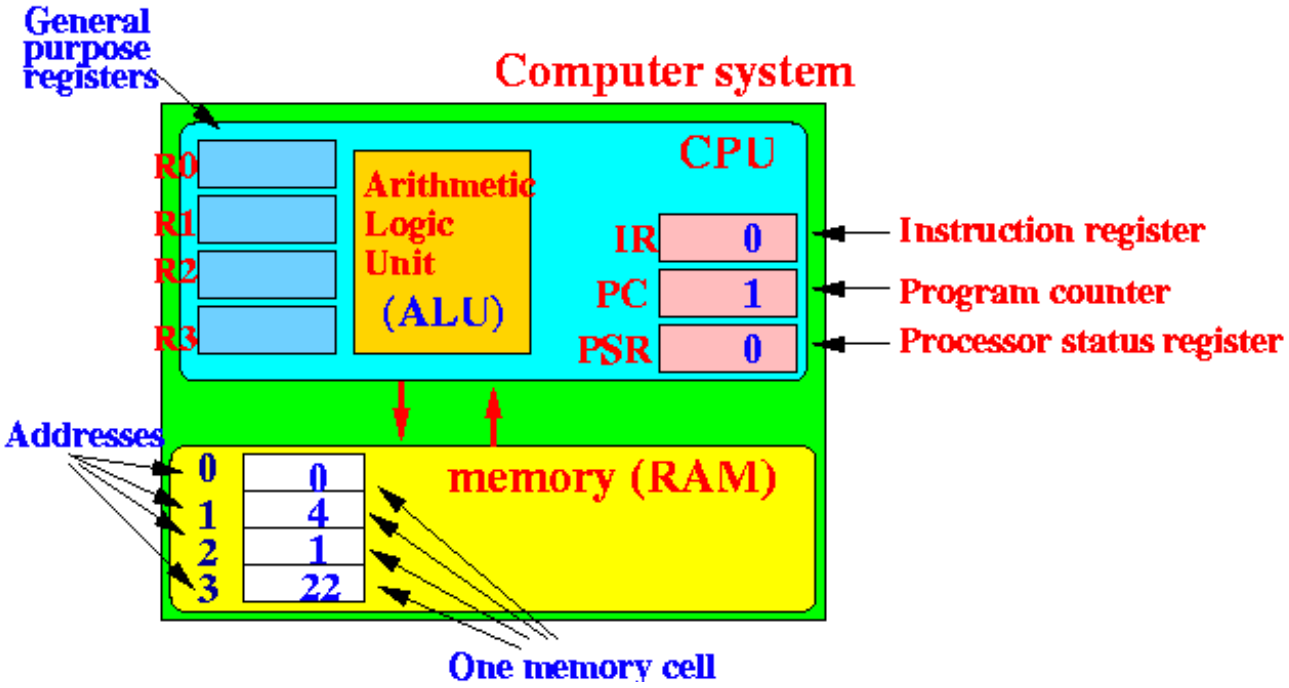
2. Memory load and store instructions
   - lw and sw: Load/store word
   - ld and sd: Load/store doubleword

3. Control transfer instructions (changing sequence of instruction execution)
   - Conditional branch: bne, beq
   - Unconditional jump: j (
   - Procedure call and return: jal and jr

Components of a Computer







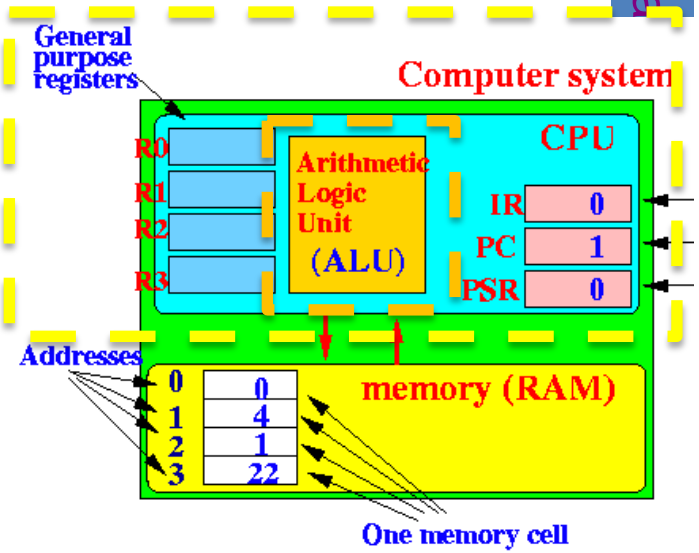
Arithmetic Operations (of the First Class Instrs)

- Add and subtract, three operands
  - Two sources operands: provide input or source data
  - One destination operand: where result goes to.

```
add a, b, c //sum of b and c is placed in a
```

- All arithmetic operations have this form
  - Three operands, two sources and one destination
  - 3-operands instructions

- *Design Principle 1:* Simplicity favors regular
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lov



Arithmetic Example

- C code:

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

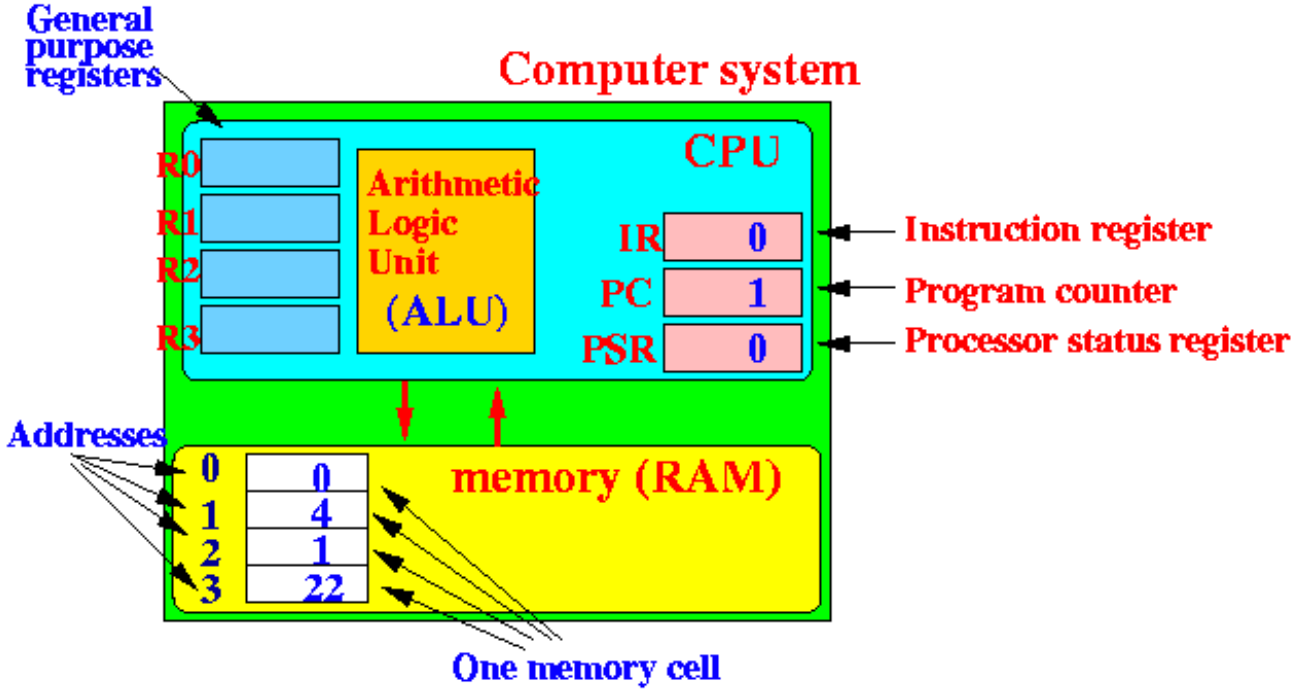
- Compiled RISC-V code:

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

- **What are those symbols (t0, g, h, …) and where are their values are stored?**
  - **Recall variables refers to *memory* locations**
  - **Registers:** super-fast small memory/storage used inside a CPU chip

Registers in CPU

- Registers are super-fast small memory/storage used in CPU.
  - General-purpose registers, program counter, instruction register, status register, floating-point register, etc
  - **32 GP Registers** in RISC-V CPU, 32-bit or 64-bit size for each
- **Data and instructions need to be loaded to memory *and then register* in order to be processed.**



Register Operands

- **Arithmetic instructions use register operands**
  - **add <dest>, <src1>, <src2>**
- **64-bit RISC-V has 32 64-bit general purpose *registers***
  - The storage for all GP registers is called a register file
    - It is storage, i.e. to store data
  - Use for frequently accessed data
  - **Numbered x0 to x31**
    - **the "memory address" for register**
  - 64-bit data is called a "doubleword"
  - 32-bit data called a "word"

x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

- ***Design Principle 2:* Smaller is faster**
  - **c.f. main memory: millions of locations**

RISC-V 32 64-Bit Registers, x0 to x31

- Usage **convention** for most programs:
  - **x0: the constant value 0**
  - x1: return address of a function
  - x2: stack pointer of a functon
  - 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

x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

Register Operand Example

- C code:

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

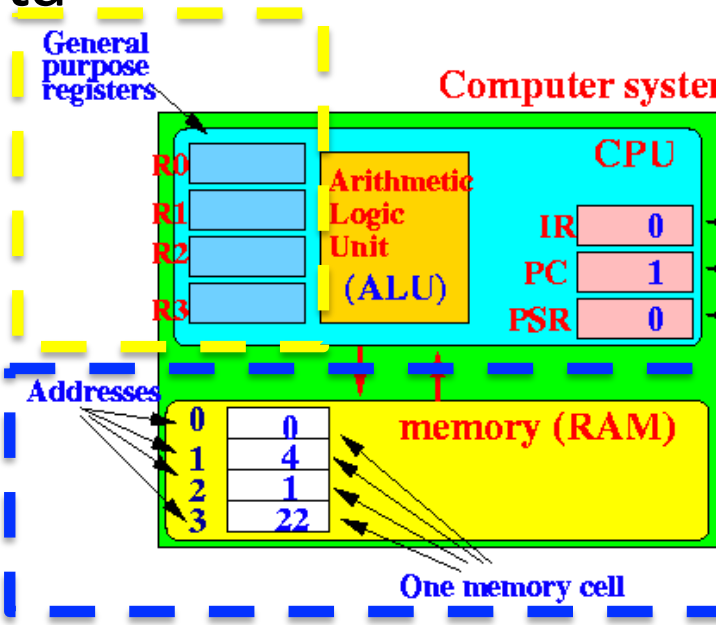
  - f, …, j values are already loaded in x19, x20, …, x23

- Compiled RISC-V code, all are register operands
  - **Three operands: the first operand is destination, last two are source operands**

```
add x5, x20, x21 // x5 = x20 + x21
add x6, x22, x23 // x6 = x22 + x23
sub x19, x5, x6  // x19 = x5 – x6
```

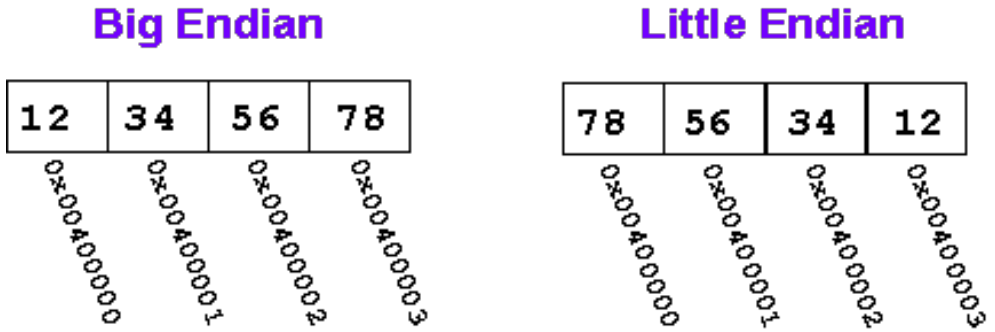
Second Class Instr: Memory Access

- Main memory used for composite data
  - Arrays, structures, dynamic data
- **To apply arithmetic operations**
  - **Load values from memory into registers**
  - **Store result from register to memory**



- Memory is byte addressed
  - Each address identifies an 8-bit byte
- RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: Most-significant byte at least address

To store number 12345678



Memory Access Example

- C code:

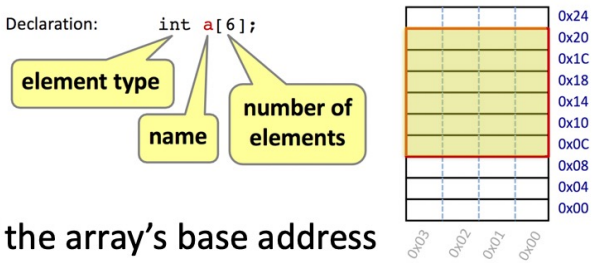
```
double A[N]; //double size is 8 bytes
A[12] = h + A[8];
```

  - h in x21, base address of A in x22

- Compiled RISC-V code:
  - Index 8 requires offset of 64
  - A[8] right-val, A[12]: left-val

```
ld  x9, 64(x22)      // load doubleword
add x9, x21, x9
sd  x9, 96(x22)      // store doubleword
```

- `int a[6];`



- a is the name of the array's base address
  - 0x0C

&a[i]: (char*)a + i * sizeof(int)

64(x22) and 96(x22) are **memory operands**, in contrast to register operands (x9)

Load and Store Instructions

```
Format:  ld  rd, offset(rs1)
Example: ld  x9, 64(x22)  // load doubleword to x9
```

- ld: load a doubleword from a memory location whose address is specified as rs1+offset (base+offset, x22+64) into register rd (x9)
  - Base should be stored in an register, **offset MUST be a constant number**
  - Address is specified similar to array element, e.g. A[8], for ld, the address is offset(base), e.g. 64(x22)

```
Format:  sd  rs2, offset(rs1)
Example: sd  x9, 96(x22)    // store a doubleword
```

- sd: store a doubleword from register rs2 (x9 in the example) to a memory location whose address is specified as rs1+offset(base+offset, x22+96). **Offset MUST be a constant number.**

- **Load and store are the ONLY two instructions that access memory**
- lw: load a word from memory location to a register
- sw: store a word from a register to a memory location

More Load/Store Examples: Addressing Memory

- int A[100]; base address (A, or &A[0]) is in x23, int is 4 bytes

```
Format:  lw rd, offset(rs1)
Example: lw x6, 16(x23) // load word from A[4] to x6
Format:  sw rs2, offset(rs1)
Example: sw x7, 32(x23) // store a word from x7 to A[8]
```

- L/S A[0]: address can be specified as 0(x23).

- A scalar variable (e.g. int f;) can be considered as one-element array (e.g. int f[1]) for load/store its value between mem and reg
  - L/S a variable's (e.g. int f) 32-bit value stored in a specific memory address which is stored in register x6 to register x8
    - lw x8, 0(x6) //offset is 0
    - sw x8, 0(x6)

A[8] = A[10], base is in x23, each element 4 bytes

- Lw x6, 40(x23)
- Sw x6, 32(x23)

- The context of the terms we use: **base and offset**
  - For array/variable: **base**: &A[0], **offset**: bytes between A[0] and A[i];
  - For LW/SW: **base**: base register, **offset**: the constant in the instr

  - If you have address of A[4] in x9
    - LW x5, 0(x9): load A[4]
    - SW x5, 8(x9): store to A[6]
    - SW x5, -8(x9): store to A[2]

- Lab 02 helps you step-by-step for address

More Load/Store Examples: Addressing Memory B[i], i is NOT constant

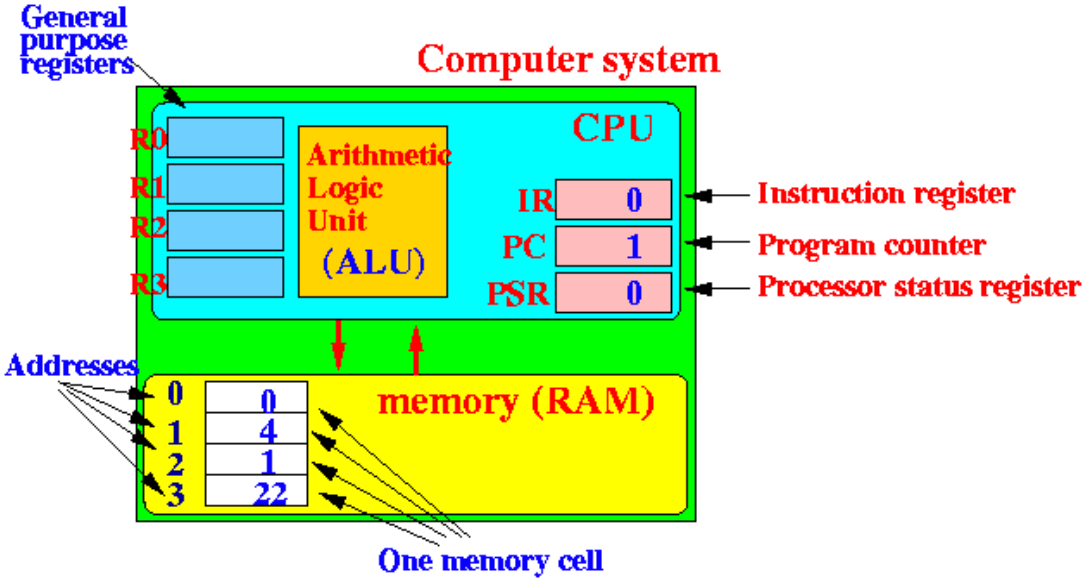
int B[N], B2[N]; // int type, 4 byptes
B2[i] = B[i];

- Base address for B and B2 are in register x22 and x23. i is stored in register x5
  - We need load B[i] to a register, e.g. x9, and then store x9 to B[2]
  - Need to use the address for B[i] and B2[i] in load and store
    - base+offset: B+i*4, and B2+i*4
  - But i*4 is not constant, cannot be the offset of load and store
  - **Solution: Calculate the address of B[i] and B2[i] and store in registers as base for LW/SW, and then use 0 as offset in L/S**

```
slliw x6, x5, 2      // x6 now store i*4, slliw is i<<2 (shift left logic)
add x7, x22, x6   // x7 now stores address of B[i].
lw x9, 0(x7)       // load a word from memory location (x7+0), which is B[i], into
                   // reg x9
add x8, x23, x6  // x8 now stores the address of B2[i]
sw x9, 0(x8)       // store a word from register x9 to memory location (x8+0)
                   // which is B2[i]
```

Registers vs. Memory

- Registers are faster to access than memory
  - ~100x faster, ~10 more expensive, and takes more space
- **Operating on memory data requires loads and stores**
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only **spill** to memory for less frequently used variables
  - Register optimization is important!



Constant or Immediate Operands

- **Constant** data specified in an instruction

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

- No subtract immediate instruction
  - Just use a negative constant

```
addi x2, x1, -1
```

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

The Constant Zero

- RISC-V register x0 is the constant 0 always
  - Cannot be overwritten

- Useful for common operations
  - E.g., move between registers

```
add x9, x5, x0
addi x9, x5, 0
```

Two Classes of Instructions so Far

- Arithmetic instructions
  - Three operands, could be either register or immediate (for source operands only)
    - add x10, x5, x6; sub x5, x4, x7
    - addi x10, x5, 10;
- Load and store (L/S) instructions: Load data from memory to register and store data from register to memory
  - Remember the way of specifying memory address (base+offset)
  - ```
    ld  x9, 64(x22)     // load doubleword
    sd  x9, 96(x22)     // store doubleword
    ```
- With these two classes instructions, you can implement the following high-level code, and different ways of combining them
  - `f = (g + h) - (i + j);`
  - `A[12] = h + A[8];`
  - For L/S: **Left-value (of =) to Store**, **Right-value of (=) to Load**
    ```

Psuedo-instructions Used in RARS

- Are NOT machine instructions
- Are assembly instructions that help programmers
  - Translated to machine instructions by assembler
- For example
  - mv x6, x7 //move/copy value from x7 to x6
    - Machine instruction: add x6, x7, x0 //since x0 is always 0
    - Machine instruction: addi x6, x7, 0
  - li x8, 100 //set the value of a register to be an immediate (load immediate)
    - Machine instruction: addi x8, x0, 100
  - la x10, label //load address of label to register
    - Need two machine instructions
      - auipc x8, xxx
      - addi x0, x0, xxx

Clarifying the Terms

- For ALU to access register
  - Fetch and set
- For move data between mem and register
  - Load and store
- For move data between storage and mem
  - Read and write

Chapter 2: Instructions: Language of the Computer

- Lecture
  - 2.1 Introduction
  - 2.2 Operations of the Computer Hardware
  - 2.3 Operands of the Computer Hardware
- ☛ Lecture
  - 2.4 Signed and Unsigned Numbers
  - 2.5 Representing Instructions in the Computer
- Lecture
  - 2.6 Logical Operations
  - 2.7 Instructions for Making Decisions
- Lecture
  - 2.8 Supporting Procedures in Computer Hardware
  - 2.9 Communicating with People
  - 2.10 RISC-V Addressing for Wide Immediate and Addresses
- Lecture
  - ~~2.11 Parallelism and Instructions: Synchronization~~
  - ~~2.12 Translating and Starting a Program~~
    - We covered before along with C Basics
  - 2.13 A C Sort Example to Put It All Together
  - 2.14 Arrays versus Pointers
    - We covered most before along with C Basics
  - ~~2.15 Advanced Material: Compiling C and Interpreting Java~~
  - 2.16 Real Stuff: MIPS Instructions
  - 2.17 Real Stuff: x86 Instructions
  - ~~2.18 Real Stuff: The rest of RISC-V~~
  - ~~2.19 Fallacies and Pitfalls~~
  - 2.20 Concluding Remarks
  - ~~2.21 Historical Perspective and Further Reading~~

Unsigned (Positive) Binary Integers

- Given an n-bit number

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

- Range: 0 to $+2^n - 1$
  - 3 digits for 000 to 111 (0 to 2^3-1)
  - 0000 0000 0000 0000 0000 0000 0000 $\mathbf{1011}_2$
    = 0 + … + $\mathbf{1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0}$
    = 0 + … + **8 + 0 + 2 + 1** = 11_{10}
- Using 32 bits
  - 0 to +4,294,967,295

Representing Signed (Positive and Negative) Numbers: Two's Complement

- The most significant bit indicates the sign (1 = negative, 0 = positive)
  - **0**111: $+7_{10}$
  - **1**010: -6_{10}
- Positive numbers: **Sign***Magnitude*
  - **0**111: $+7_{10}$
- Negative numbers: **Special, NOT Sign***Magnitude* format
  - **1010 is** -6_{10}, why it is "1 010"
  - **1***110* is NOT -6_{10} (110 is binary 6)

- Given a 2's-complement binary number, what is its decimal value:
  - Look at the most significant bit for the sign of the number:
    - 1 = negative, 0 = positive
  - Positive: most significant bit is 0, the absolute value is the value of the binary number
    - 0111 = $+7_{10}$, 0 is sign (+), 111 is 7, so it is +7
  - Negative: most significant bit is 1, **the absolute value is the inverted bits of the number + 1. inverting/flipping bit: 0 → 1, 1 → 0**
    - 1010 = (-1) (**0101**+1) = (-1)(0110) = -6_{10} (inverting 1010 yields 0101)
    - 1110 = (-1) (0001+1) = (-1) (0010) = -2_{10}

What is the decimal value of the two's complement number 1001_2?

- 1001_2 is negative,
- For the absolute value: invert it, which is 0110, and then plus 1:

0110

\+ 1

$0111_2 = 7_{10,}$ so $1001_2 = -7_{10}$

- For computer, the value of a 2's complement number can be calculated using regular position binary form recognizing the sign bit
  - **$1001 = (-1) * 2^3 + 0 + 0 + 1*2^0 = -7$**

Representing using 2's-Complement Binary

- Given a decimal number, what is its 2's-complement binary representation?
- Positive: its binary representation
  - $7_{10} = 0111_2$, most significant big MUST be 0 to indicate it is positive

- Negative: **invert bits of the binary representation of the absolute value of the number and add up 1**
  - $-3 = xxxx_2$
  - Flip bits of 3_{10} (0011_2)
    = 1100
    + 1
    $1101_2 = -3_{10}$

2's-Complement Binary Representation of -6_{10}

- $6_{10} = 0110_2$, then invert +1

1001

+ 1

$1010_2 = -6_{10}$

Rang of Two's Complement Numbers

- 4-digit 2's-complment numbers:
  - Most positive 4-bit number: **0111, 7**
  - Most negative 4-bit number: **1000, -8**
    - **1111 is -1**

- Range of an *N*-bit two's comp number:

$$[-(2^{N-1}), 2^{N-1}-1]$$

2's-Complement Signed Integers (32 bits)

- Given an n-bit number

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

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
  - **Or do invert+1: -(0011+1) = -(0100) = -4_{10}**
- For 32 bits
  - **–2,147,483,648 to + 2,147,483,647**

2s-Complement Signed Integers of 32 Bits

- **Bit 31 is sign bit**
  - 1 for negative numbers
  - 0 for non-negative numbers

- **Range: [-(2^{32}-1), 2^{32-1}-1]**

- 2^{n-1} can't be represented
  - 1000… is negative now

- Non-negative numbers have the same unsigned and 2s-complement representation

- Some specific numbers
  - **0: 0000 0000 … 0000**
  - **–1: 1111 1111 … 1111**
  - Most-negative: 1000 0000 … 0000, which is **–2,147,483,648**
  - Most-positive: 0111 1111 … 1111, which is **2,147,483,647**

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$$
$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$$
$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$
$$\ldots \qquad \ldots$$
$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$$
$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$$
$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$$
$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$$
$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$$
$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$$
$$\ldots \qquad \ldots$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$$

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

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
0000 0000 0000 0000 0000 0000 0000 0010two = 2ten
...                                          ...
0111 1111 1111 1111 1111 1111 1111 1101two = 2,147,483,645ten
0111 1111 1111 1111 1111 1111 1111 1110two = 2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two = 2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0000two = -2,147,483,648ten
1000 0000 0000 0000 0000 0000 0000 0001two = -2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0010two = -2,147,483,646ten
...                                          ...
1111 1111 1111 1111 1111 1111 1111 1101two = -3ten
1111 1111 1111 1111 1111 1111 1111 1110two = -2ten
1111 1111 1111 1111 1111 1111 1111 1111two = -1ten
```

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = $\overline{+2}$ +1 = $\overline{0000\ 0000\ \ldots\ 0010_2}$ + 1
    = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

Add Two 2's Complement Numbers: Just Bit-wise Addition

- Add 6 + (-6) using two's complement numbers

$$\begin{array}{r} 0110 \\ +\ 1010 \\ \hline \end{array}$$

- Add -2 + 3 using two's complement numbers

$$\begin{array}{r} 1110 \\ +\ 0011 \\ \hline \end{array}$$

Add Two 2's Complement Numbers:

Just Bit-wise Addition, no need to recognize positive/negative number, easier to implement in hardware

- Add 6 + (-6) using two's complement numbers

```
  111
   0110
+  1010
------
 10000
```

- Add -2 + 3 using two's complement numbers

```
  111
   1110
+  0011
------
 10001
```

Sign Extension

- Representing a number using more bits
  - **E.g. char a = -5; int b = a;**
    - **A char variable takes 1 byte of memory, and an int variable takes 4 bytes.**
    - **How to fill in the bits of the 4 bytes of memory for an int variable with a 8-bit number?**
  - **Preserve the numeric value**
- **Replicate the sign bit to the left**
  - **c.f. unsigned values: extend with 0s**
  - **1011 (-5 with 4 bits) ➔ 1111 1011 (-5 in 8 bits char)**
  - **11111011 (-5 in char) ➔ 11111111 11111111 11111111 11111011 (-5 in 32-bits int)**
- More examples: 8-bit to 16-bit
  - +2: **0**000 0010 => **0000 0000 0**000 0010
  - –2: **1**111 1110 => **1111 1111 1**111 1110

- In RISC-V instruction set
  - `addi`: extend immediate value
  - **lb,lh,lw,ld**: extend loaded byte/halfword/word/doubleword
  - **lbu, lhu, lwu, ldu**: zero extend loaded byte/halfword/word/doubleword
  - `beq, bne`: extend the displacement

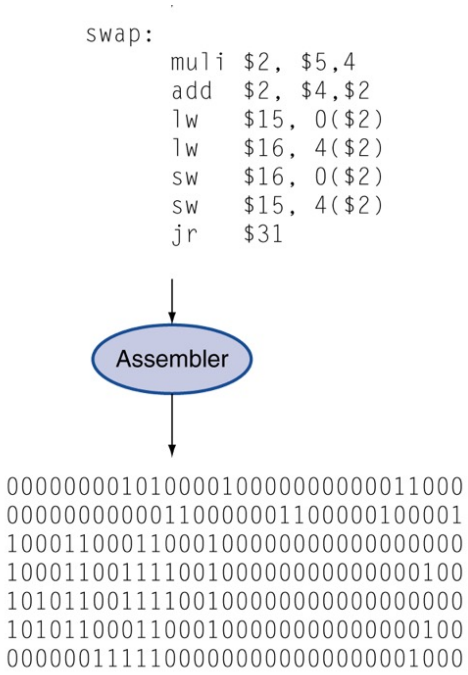
Three Classes of Instructions

1. **Arithmetic-logic instructions**
   - **add, sub, addi, and, or, shift left|right, etc**

2. **Memory load and store instructions**
   - **lw and sw: Load/store word**
   - **ld and sd: Load/store doubleword**

- **Control transfer instructions (changing sequence of instruction execution)**
  - **Conditional branch: bne, beq**
  - **Unconditional jump: j (**
  - **Procedure call and return: jal and jr**

Representing Instructions



- Instructions are encoded in binary
  - Using binary numbers to represent operations, operands, and immediate
  - Called machine code
- RISC-V instructions
  - **Each instruction is encoded as a 32-bit instruction word**
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Instructions use 32 registers:
  - We need 5 bit to identify 32 registers (0 to 31)
  - 00000 to 11111

Hexadecimal

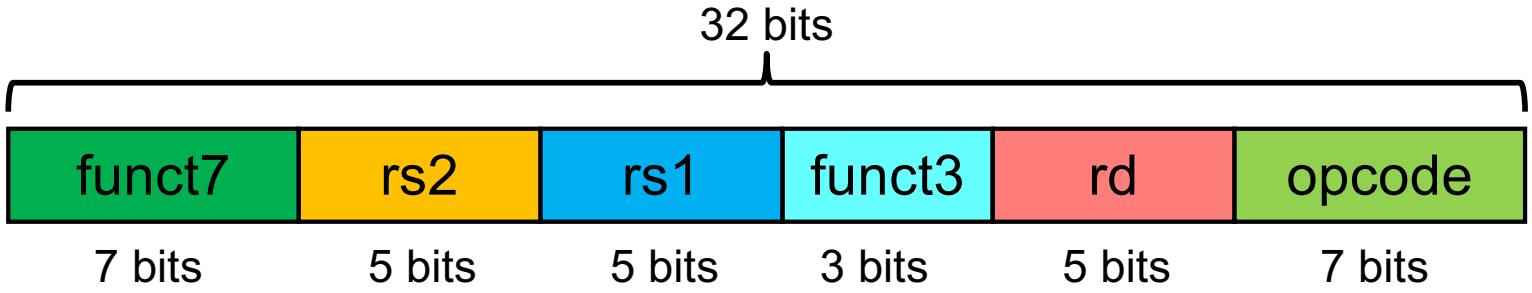
- Base 16 format to easily show binary number
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V R-Format Instructions

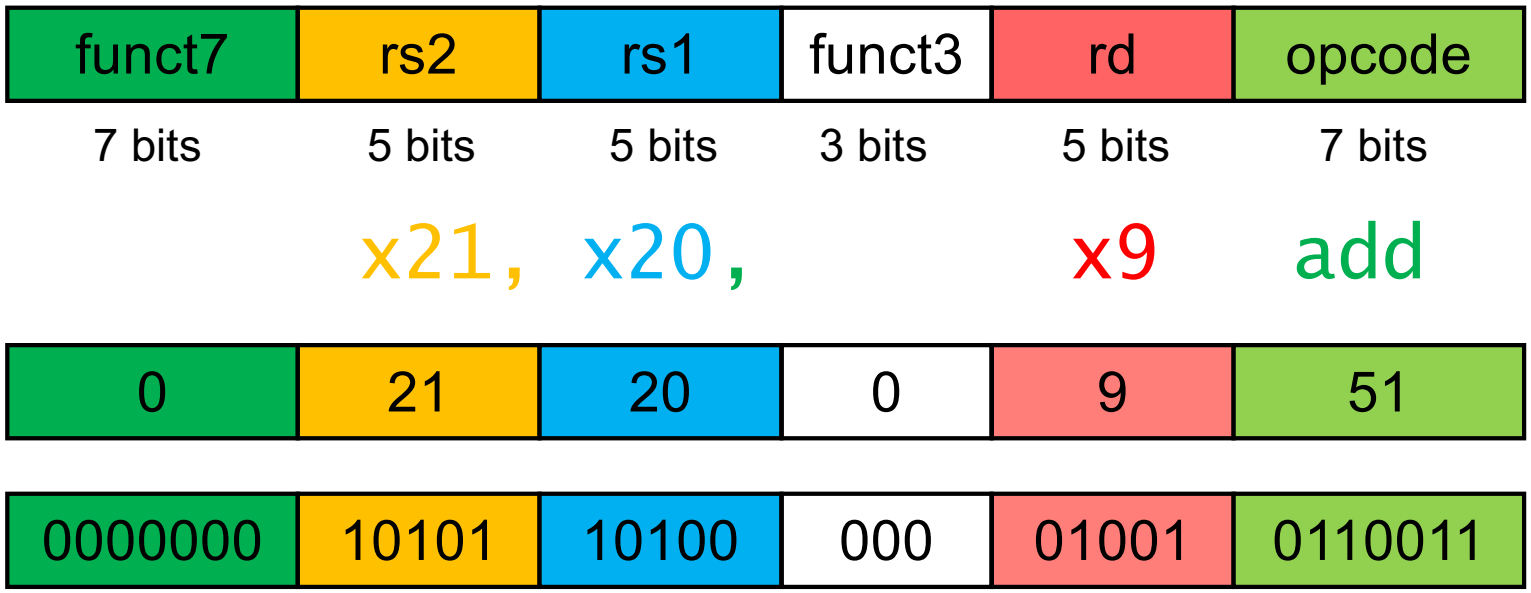
- R-Format: Operands are all from **Registers**
- **Arithmetic and logic instructions that use registers for ALL operands. Format: op rd, rs1, rs2.**



- 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 Encoding Example 1

`add x9,x20,x21` `(add rd, rs1, rs2)`



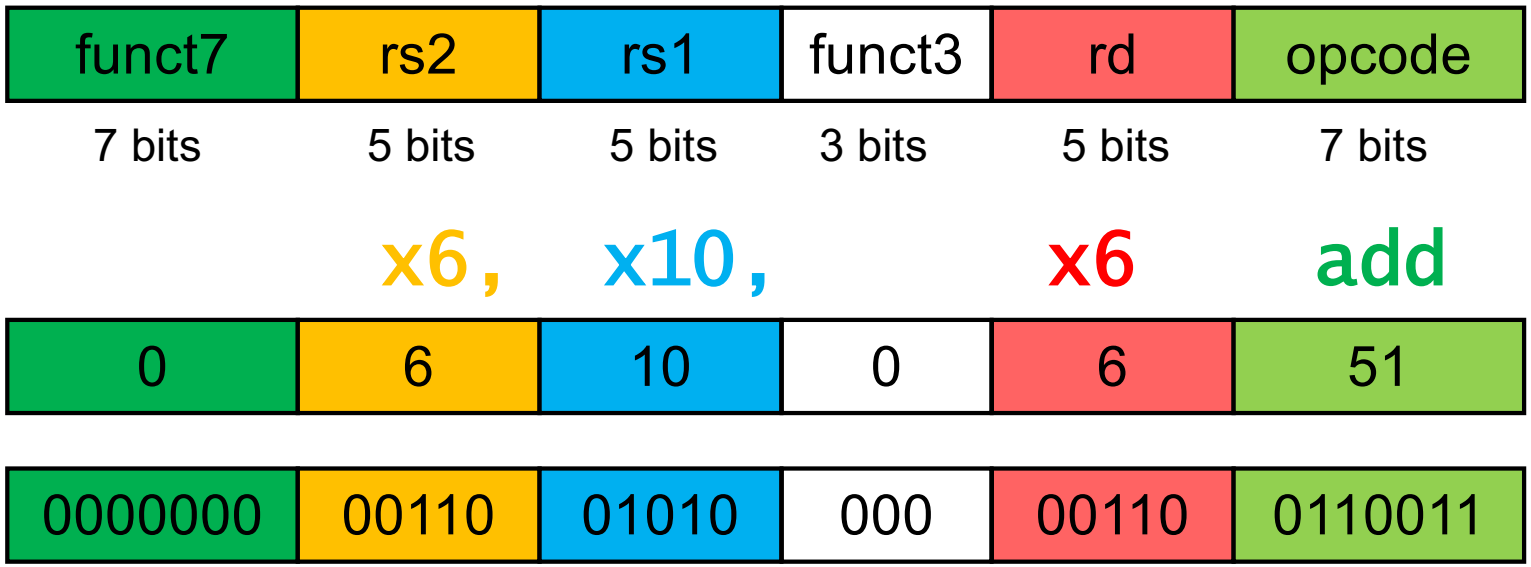
$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two}$ =
$015A04B3_{16}$

5 bits for rd, rs1 and rs2 because we have 32 registers,

thus only needs 5 bit to address a register

R-Format Encoding Example 2

`add x6, x10, x6` `(add rd, rs1, rs2)`



$0000\ 0000\ 0110\ 0101\ 0000\ 0011\ 0011\ 0011_{two}$ =

00650333_{16}

Opcode (51), funct3 (0) and funct7(0) for each instruction are defined by the ISA standard.

R-Format Instruction Encoding

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116

Arithmetic instructions

Logic instructions

RV32I Base Instruction Set

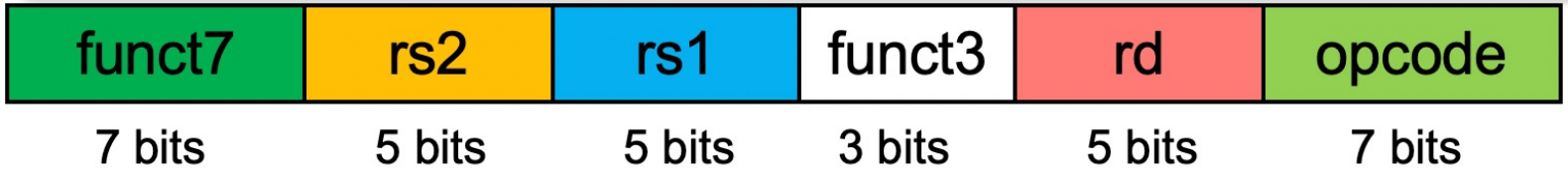
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

RV64I Base Instruction Set (in addition to RV32I)

0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

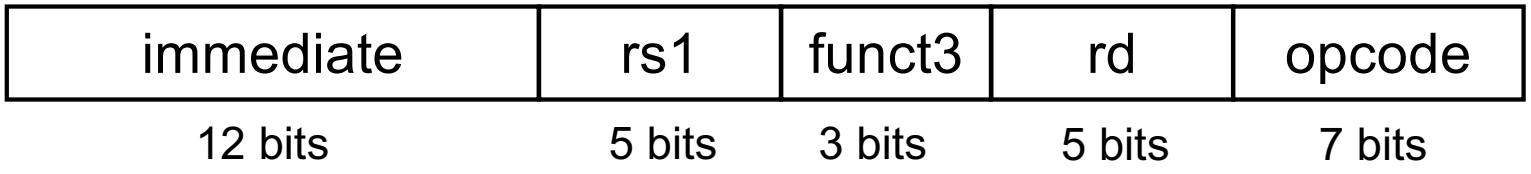
RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU



RISC-V I-Format Instructions

- **I-Format: The second source operand is an Immediate**, the first source operand is register, destination operand is register.
- **Immediate arithmetic/logic, and load instructions (NOT store instruction)**
  - `addi x22, x22, 4; Format: addi rd, rs1, #immediate`
  - `ld  x9, 64(x22); Format: ld|lw, rd, #immediate(rs1)`
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended



  - **NOT for store: because destination for store is the memory location (not a register), thus no rd for store.**
- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

I-Format Instruction Encoding

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116

Immediate arithmetic/logic **load instructions**

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

imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

imm[11:0]	rs1	000	rd	0010011	ADDI
imm[11:0]	rs1	010	rd	0010011	SLTI
imm[11:0]	rs1	011	rd	0010011	SLTIU
imm[11:0]	rs1	100	rd	0010011	XORI
imm[11:0]	rs1	110	rd	0010011	ORI
imm[11:0]	rs1	111	rd	0010011	ANDI
0000000 shamt	rs1	001	rd	0010011	SLLI
0000000 shamt	rs1	101	rd	0010011	SRLI
0100000 shamt	rs1	101	rd	0010011	SRAI

imm[11:0]	rs1	110	rd	0000011	LWU
imm[11:0]	rs1	011	rd	0000011	LD

imm[11:0]	rs1	000	rd	0011011	ADDIW

Shift Operation Encoding

- Use immediate operands, I-Format
  - Immediate: slli, sri, srai, etc

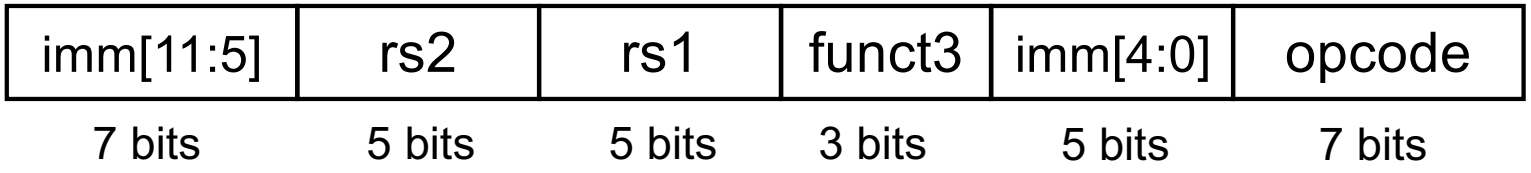
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits	
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

- If use registers for all operands, R-Format
  - Sll, sri, sra

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

RISC-V S-Format Instructions

- S-Format: instructions that use two source register operands and NO destination operand register (rd), **only store instruction**
- **`Format: sd|sw, rs2, #immediate(rs1)`**



- Different immediate format for store instructions
  - `sd  x9, 96(x22);`
  - rs1: base address register number (x22)
  - rs2: source operand register number (x9), which provide the value to be stored to memory
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place as for R- or I-Format

S-Format Instruction Encoding

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD

Store instructions

More Examples from Textbook 2.5

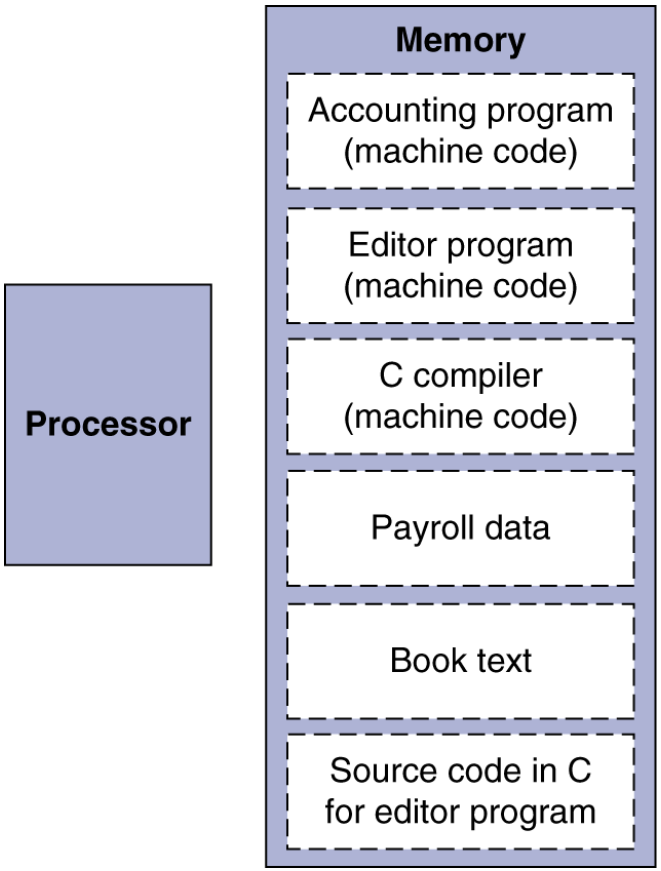
Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (Add)	R	0000000	reg	reg	000	reg	0110011
sub (Sub)	R	0100000	reg	reg		reg	0110011
Instruction	**Format**	**immediate**		**rs1**	**funct3**	**rd**	**opcode**
addi (Add Immediate)	I	constant		reg	000	reg	0010011
ld (Load doubleword)	I	address		reg	011	reg	0000011
Instruction	**Format**	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**
sd (Store doubleword)	S	address	reg	reg	011	address	0100011

More Examples from Textbook 2.5

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (Add)	0000000	00011	00010	000	00001	0110011	add x1,x2,x3
sub (Sub)	010000	00011	00010	000	00001	0110011	sub x1,x2,x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (Add Immediate)	001111101000		00010	000	00001	0010011	addi x1,x2, 1000
ld (Load doubleword)	001111101000		00010	011	00001	0000011	ld x1,1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sd (Store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1,1000(x2)

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs