RISC-V Instruction Set

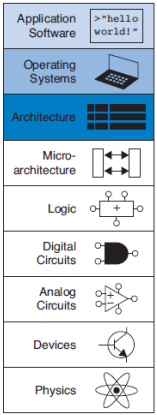
Computer Architecture

Dr. Mahmoudi

February 8, 2025

Outline

- Introduction
- From a High-Level Language to the Language of Hardware
- Arithmetic Operations
- Register/Memory Operands
- Data Transfer Operations
- Logical Operations
- Conditional Operations
- Supporting Procedures in Computer Hardware
- Representing Instructions in the Computer

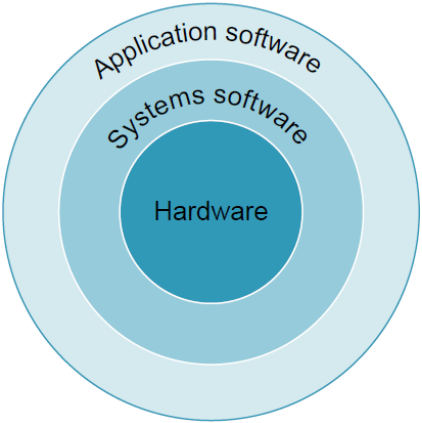


ensia

Introduction

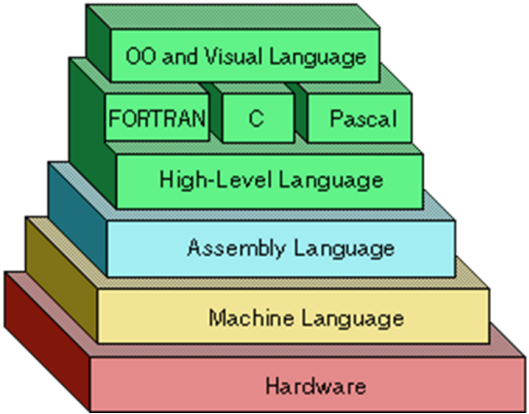
Introduction

- The hardware in a computer can only execute extremely simple low-level instructions.
- To go from a complex application to these primitive instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions **(GI: Abstraction)**



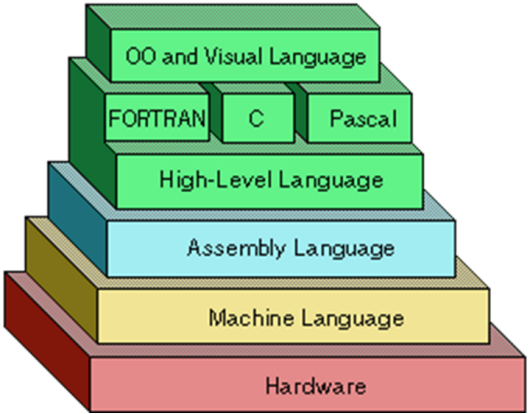
From a High-Level Language to the Language of Hardware

- The first programmers communicated to computers in binary numbers (**Machine language**).
- It is a low-level language understandable directly by the machine.



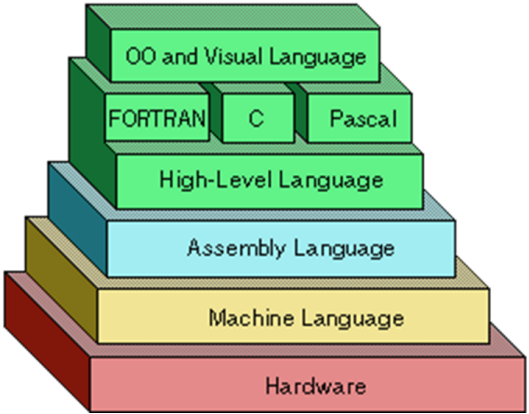
From a High-Level Language to the Language of Hardware

- In addition to knowledge of the machine architecture, the user must know:
  - All operation codes (in binary) of all instructions,
  - The format of all instructions,
  - All addressing modes wired into the machine.



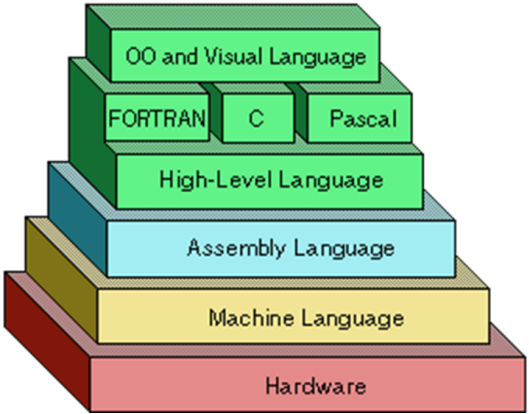
From a High-Level Language to the Language of Hardware

- They invented a new notation closer to the human language called the **assembly language** (*a symbolic representation of machine instructions*).



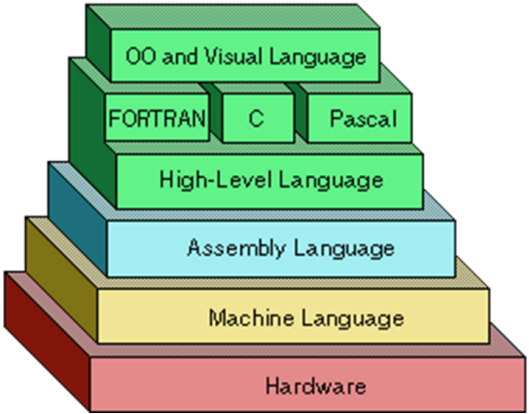
From a High-Level Language to the Language of Hardware

- **Assembly language** is a language closer to the machine where the user needs a minimum requirement on the architecture of the machine namely: The number of processor registers and their sizes, the size of the memory and the address of the area reserved for users etc.



From a High-Level Language to the Language of Hardware

- At first, these notations were translated to binary by hand.
- **Assembler:** a program that translates a symbolic version of instructions into the binary version called **machine language** (*a binary representation of machine instructions*).

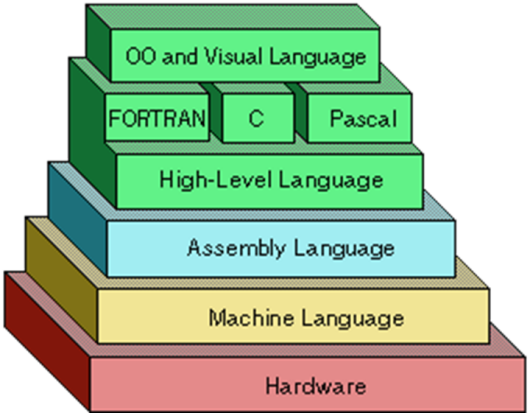


From a High-Level Language to the Language of Hardware

- Margaret Hamilton's Apollo code.

High-level language

- High-level language is a programming language that is closer to mathematical writing.
- The user of this language does not care much about the architecture of the Machine.
- There are several advanced languages including: COBOL, PL1, BASIC, FORTRAN4, FORTRAN90, PASCAL, C LANGUAGE, DELPHI, C++, MATLAB, JAVA, PYTHON, JAVASCRIPT.



From a High-Level Language to the Language of Hardware

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;
}
```

Compiler

Assembly language program (for RISC-V)

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      lw   x5, 0(x6)
      lw   x7, 4(x6)
      sw   x7, 0(x6)
      sw   x5, 4(x6)
      jalr x0, 0(x1)
```

Assembler

Binary machine language program (for RISC-V)

```
00000000001101011001100100010011
00000000011001010000001100110011
00000000000000011001100101000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000000100000000100111
```

RISC-V

- **Instruction set:** The vocabulary of commands understood by a given architecture.
- RISC-V was originally developed at UC Berkeley starting in 2010.
- RISC-V is an open architecture that is controlled by RISC-V International.
- It is not a proprietary architecture that is owned by a company like ARM, MIPS, or x86.
- In 2020, more than 200 companies are members of RISC-V International, and its popularity is growing rapidly.

RISC-V

Krste Asanović started RISC-V as a summer project. He is a professor of computer science at the University of California, Berkeley and the Chairman of the Board for RISC-V International, formerly known as the RISC-V Foundation. He is also the cofounder of SiFive, a company that develops and commercializes RISC-V chips, boards, and supporting tools. (Photo printed with permission.)

Andrew Waterman designs microprocessors at SiFive, a company he cofounded with Krste Asanović in 2015 to provide low-cost RISC-V cores and custom chips. He earned his PhD in computer science from UC Berkeley in 2016, where, weary of the vagaries of existing instruction-set architectures, he co-designed the RISC-V ISA and the first RISC-V cores. (Photo printed with permission.)

David Patterson has been a professor of computer science at the University of California, Berkeley since 1976, and he coinvented *reduced instruction set computing* with John Hennessy in 1984. It was later commercialized as the SPARC architecture. He helped develop the RISC-V architecture and continues to play an integral role in its development. (Photo printed with permission.)

ensia

Arithmetic Operations

Arithmetic Operations

- Every computer must be able to perform arithmetic.
- All arithmetic operations have this form: two sources and one destination.
- This notation is rigid in that each RISC-V arithmetic instruction performs only one operation and must always have exactly three variables.

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands;
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands;
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants

Table: RISC-V Arithmetic Operations

Arithmetic Operations

- For example, suppose we want to place the sum of four variables b, c, d, and e into variable a.
- Each line of this language can contain at most one instruction.

```
add a, b, c        // The sum of b and c is placed in a
add a, a, d        // The sum of b, c, and d is now in a
add a, a, e        // The sum of b, c, d, and e is now in a
```

- The natural number of operands for an arithmetic operation is three.
- Requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple.
- hardware for a variable number of operands is more complicated than hardware for a fixed number.

Arithmetic Example

C/C++ code

```
f = (g + h) – (i + j);
```

RISC-V code

```
add t0, g, h // temporary variable t0 contains g + h
add t1, i, j // temporary variable t1 contains i + j
sub f,t0,t1 // f gets t0 – t1, which is (g + h) – (i + j)
```

ensia

Register/Memory Operands

Register Operands

- Unlike programs in high-level languages, the operands of RISC-V arithmetic instructions must be from a limited number of registers.
- RISC-V has 32 general purpose registers **x0** to **x31**.
- The size of a register in the RISC-V architecture is 32 bits **(called word)**.
- Another popular size is a group of 64 bits, called a **doubleword** in the RISC-V architecture.
- The reason for the limit of 32 registers is that very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther **(Design Principle 2: Smaller is faster)**.

RISC-V Registers

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

Register Operands Example

C/C++ code

```
f = (g + h) – (i + j);
```

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

RISC-V code

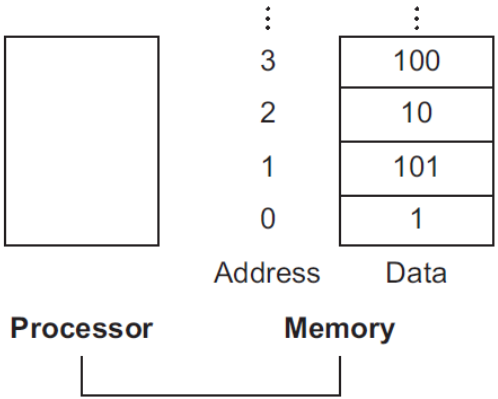
```
add x5, x20, x21  // register x5 contains g + h
add x6, x22, x23  // register x6 contains i + j
sub x19, x5, x6   // f gets x5 – x6, which is (g + h) – (i + j)
```

Memory Operands

- The processor can keep only a small amount of data in registers.
- Hence, data structures (arrays and structures) are kept in memory.
- Arithmetic operations occur only on registers in RISC-V instructions;
- thus, RISC-V must include instructions that transfer data between memory and registers.
- Such instructions are called **data transfer instructions**.
- **Data transfer instruction**: A command that moves data between memory and registers.

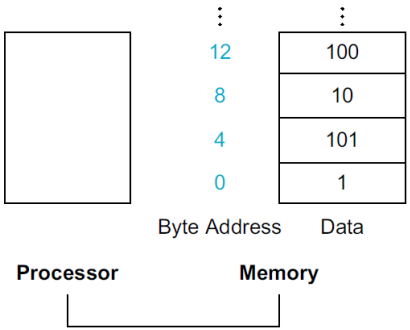
Memory Operands

- To access a word in memory, the instruction must supply the memory **address**.
- Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0.
- **Address** A value used to delineate (mark) the location of a specific data element within a memory array.



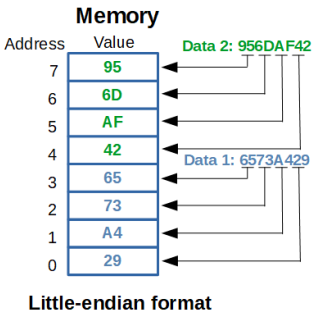
Memory Operands

- Memory is byte addressed: each address identifies an 8-bit byte
- Therefore, the address of a word matches the address of one of the 4 bytes within the word.
- the addresses of sequential words differ by 4.
- In many architectures, words must start at addresses that are multiples of 4. This requirement is called an **alignment restriction**. RISC-V and Intel x86 do not have alignment restrictions, but MIPS does.

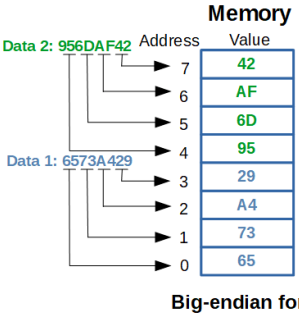


Memory Operands

- Computers either use the address of the leftmost or “big end” byte as the word address or use the rightmost or “little end” byte as the word address.
- RISC-V is Little Endian.



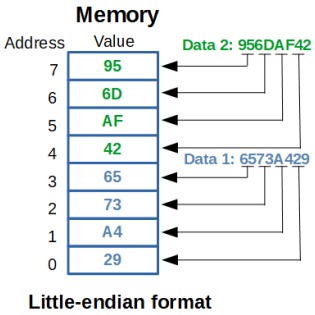
Little-endian format



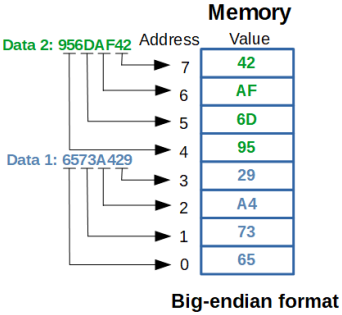
Big-endian format

Memory Operands

- **Little-endian**: The bytes are ordered with the least significant byte placed at the lowest address.
- **Big-endian**: The bytes are ordered with the most significant byte placed at the lowest address.



Little-endian format



Big-endian format

ensia

Data Transfer Operations

Data transfer: Load into registers

- The data transfer instruction that copies data from memory to a register is traditionally called **load**.
- The format of the load instruction is the name of the operation followed by the register to be loaded, then a register, and a constant used to access memory.
- The sum of the constant portion of the instruction and the contents of the second register forms the memory address.
- The real RISC-V name for this instruction is lw, standing for load word.

Data transfer: Load into registers Example

- A is an array and g and h are associated with the registers x20 and x21.
- The starting address, or base address, of the array is in x22.
- The register added to form the address (x22) is called the base register, and the constant in a data transfer instruction (8*4) is called the offset.

C/C++ code

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

RISC-V code

```
lw x9, 8*4 (x22) // Temporary reg x9 gets A[8]
add  x20, x21, x9 // g = h + A[8]
```

Data transfer: Store to memory

- The instruction complementary to load is traditionally called **store**.
- It copies data from a register to memory.
- The format of a store is similar to that of a load: the name of the operation, followed by the register to be stored, then the base register, and finally the offset to select the array element.
- The actual RISC-V name is sw, standing for store word.

Data transfer: Store to memory Example

- A is an array and h is associated with the register x21.
- The starting address, or base address, of the array is in x22.

C/C++ code

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

RISC-V code

```
lw    x9, 32(x22)    // Temporary reg x9 gets A[8]
add   x9, x21, x9    // Temporary reg x9 gets h + A[8]
sw    x9, 48(x22)    // Stores h + A[8] back into A[12]
```

Data transfer: Instructions

Category	Instruction	Example	Meaning	Comments
Data transfer	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits

Registers vs. Memory

- Programs have more variables than computers have registers.
- Consequently, the compiler tries to keep the most frequently used variables in registers and places the rest in memory, using loads and stores to move variables between registers and memory.
- The process of putting less frequently used variables (or those needed later) into memory is called **spilling registers**.

Registers vs. Memory

- Registers are faster to access than memory.
- Moreover, data are more useful when in a register.
- A RISC-V arithmetic instruction can read two registers, operate on them, and write the result.
- A RISC-V data transfer instruction only reads one operand or writes one operand, without operating on it.
- Thus, registers take less time to access and have higher throughput than memory,
- Accessing registers also uses much less energy than accessing memory.
- To achieve the highest performance and conserve energy, an instruction set architecture must have enough registers, and compilers must use registers efficiently.

Constant or Immediate Operands

- Many times a program will use a constant in an operation (i.e. incrementing an index of an array).
- More than half of the RISC-V arithmetic instructions have a constant as an operand (SPEC CPU2006 benchmarks).
- The most popular is called add immediate or addi:

```
addi      x22, x22, 4      // x22 = x22 + 4
```

ensia

Logical Operations

Logical Operations

- The first class of such operations is called shifts. They move all the bits in a word to the left or right, filling the emptied bits with 0s.
- 00001001 shift left by 4 10010000
- The dual of a shift left is a shift right.

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

Logical Operations

- The actual names of the two RISC-V shift instructions are shift left logical immediate (slli) and shift right logical immediate (srli).

```
slli x11, x19, 4 // reg x11 = reg x19 << 4 bits
```

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

Logical Operations

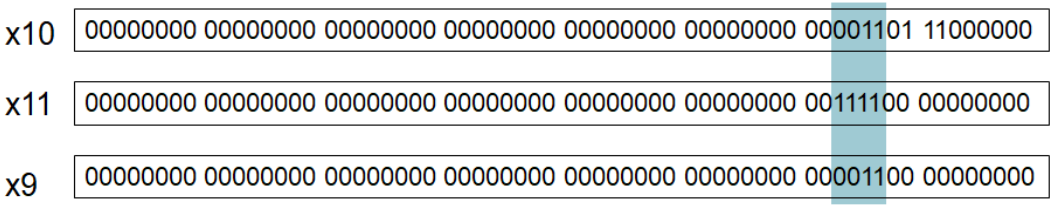
- RISC-V provides a third type of shift, shift right arithmetic (srai) which fills the vacant bits with the sign bit.
- It also provides variants of all three shifts that take the shift amount from a register, rather than from an immediate: sll, srl, and sra.

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`

Logical Operations: AND

- **Useful to mask bits in a word:** Select some bits, clear others to 0.
- The x11 bit pattern in conjunction with AND is traditionally called a **mask**, since the mask "conceals" some bits.

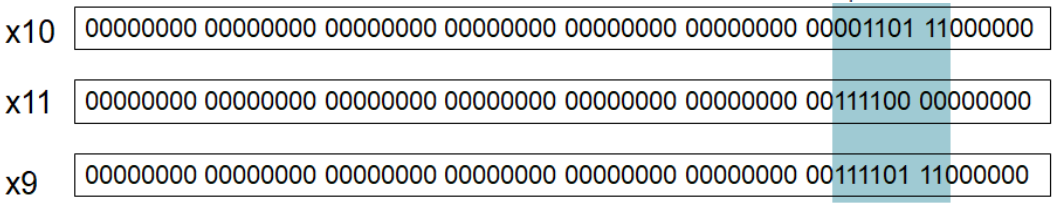
```
and x9, x10, x11      // reg x9 = reg x10 & reg x11
```



Logical Operations: OR

- **Useful to include bits in a word:** Set some bits to 1, leave others unchanged

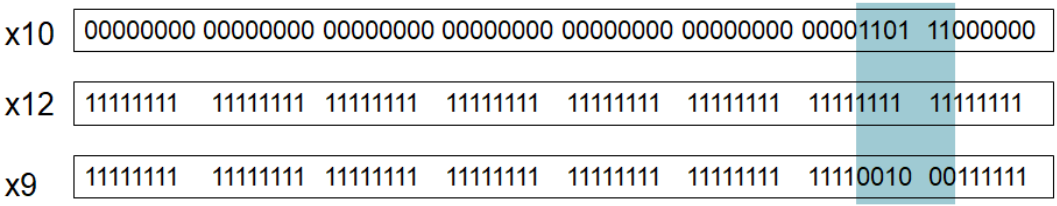
```
or x9, x10, x11 // reg x9 = reg x10 | reg x11
```



Logical Operations: XOR

- The designers of RISC-V decided to include the instruction XOR (exclusive OR) instead of NOT.

```
xor x9,x10,x12  // NOT operation
```



Logical Operations

Category	Instruction	Example	Meaning	Comments
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

ensia

Conditional Operations

Conditional Operations

- RISC-V assembly language includes two decision-making instructions, similar to an if statement with a go-to.
- The first instruction is:

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

- This instruction means go to the statement labeled L1 if the value in register rs1 equals the value in register rs2.
- Otherwise, continue sequentially.
- The mnemonic beq stands for “branch if equal”.

Conditional Operations

- The second instruction is:
- `bne rs1, rs2, L1`
- It means go to the statement labeled L1 if the value in register rs1 does not equal the value in register rs2.
- The mnemonic bne stands for “branch if not equal”.
- These two instructions are traditionally called **conditional branches.**

Conditional Operations: Example

- f …g in x19 …x20

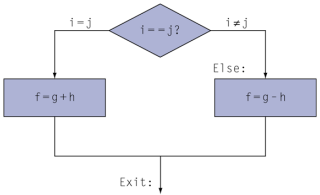
C/C++ Code

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

RISC-V Code

```
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:
```

Assembler calculates addresses



Conditional Operations

- The previous example introduces another kind of branch, often called an unconditional branch.
-
```
beq x0, x0, Exit        // if 0 == 0, go to Exit
```
- This instruction says that the processor always follows the branch.
- One way to express an unconditional branch in RISC-V is to use a conditional branch whose condition is always true.

Conditional Operations: Loops

- i in x22, k in x24, address of save in x25

C/C++ Code

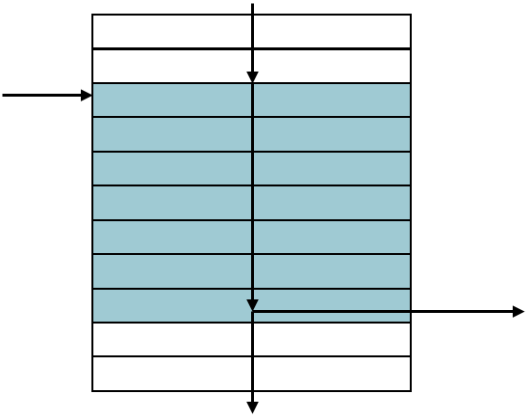
```
while  (save[i] == k)
       i += 1;
```

RISC-V Code

```
Loop: slli x10, x22, 2 // Temp reg x10 = i * 4
add x10, x10, x25      // x10 = address of save[i]
lw x9, 0(x10)          // Temp reg 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:
```

Basic Blocks

- A basic block is a sequence of instructions without branches, except possibly at the end,
- and without branch targets or branch labels, except possibly at the beginning.
- One of the first early phases of compilation is breaking the program into basic blocks.



More Conditional Operations

- The **branch if less than (blt)** instruction compares the values in registers rs1 and rs2 and takes the branch if the value in rs1 is smaller, when they are treated as two's complement numbers.
- **Branch if greater than or equal (bge)** takes the branch in the opposite case, that is, if the value in rs1 is at least the value in rs2.
- **Branch if less than, unsigned (bltu)** takes the branch if the value in rs1 is smaller than the value in rs2 when the values are treated as unsigned numbers.
- **Branch if greater than or equal, unsigned (bgeu)** takes the branch in the opposite case.

Conditional Operations: Example

- **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*/
blt x22, x23, Exit     // branch if a >= b
addi x22, x22, 1
Exit:
```

Conditional Operations: Bounds Check Shortcut Example

- Treating signed numbers as if they were unsigned gives us a low-cost way of checking if $0 \leq x < y$, which matches the index out-of-bounds check for arrays.
- The key is that negative integers in two's complement notation look like large numbers in unsigned notation;
- Thus, an unsigned comparison of $x < y$ checks if x is negative as well as if x is less than y.

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

ensia

Supporting Procedures in Computer Hardware

Procedure Calling

- In the execution of a procedure, the program must follow these six steps:
  1. Put parameters in a place where the procedure can access them (registers x10 to x17).
  2. Transfer control to the procedure.
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin (address in x1), since a procedure can be called from several points in a program.

Procedure Call Instructions

- RISC-V assembly language includes an instruction just for the procedures.
- It branches to an address and simultaneously saves the address of the following instruction to the destination register **rd (x1)**.
- The **jump-and-link instruction (jal)** is written:

```
jal x1, ProcedureAddress
// jump to ProcedureAddress and write return address to x1
```

- The link portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address.
- This “link,” stored in register x1, is called the **return address**.

Procedure Return Instructions

- To support the return from a procedure, computers like RISC-V use an indirect jump.
- The **jump-and-link register** instruction branches to the address stored in register x1:
- ```
  jalr x0,  0(x1)
  ```
- Like jal, but jumps to 0 + address in x1.
- Use x0 as rd (x0 cannot be changed).
  ```

Procedure Calling: Caller/Callee

- The calling program, or **caller**:
  1. puts the parameter values in **x10–x17**;
  2. Uses **jal x1, P1** to branch to procedure **P1**.
- **P1** is sometimes named the **callee**;
  1. The **callee** then performs the calculations;
  2. Places the results in the same parameter registers;
  3. Returns control to the **caller** using **jalr x0, 0(x1)**.

- **caller:***The program that instigates a procedure and provides the necessary parameter values*
- **callee:***A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller*.

More on jump-and-link instruction

- The **jal** instruction saves **PC + 4** in its designation register **(x1)** to link to the byte address of the following instruction to set up the procedure return.
- The **jal** instruction can also be used to perform an unconditional branch within a procedure by using **x0** as the destination register.
- Since **x0** is hard-wired to zero, the effect is to discard the return address:

```
jal x0, Label  // unconditionally branch to Label
```

Using More Registers

- Suppose a compiler needs more registers for a procedure than the eight argument registers.
- All registers needed by the caller must be restored to the values that they contained before the procedure was invoked.
- We need to spill registers to memory.
- The ideal data structure for spilling registers is a **stack**—a last-in-first-out queue.

Using More Registers

- A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where old register values are found.
- In RISC-V, the stack pointer is the register **x2**, also known by the name **sp**.
- The stack pointer is adjusted by one word for each register that is saved or restored.
- By historical precedent, stacks **"grow"** from higher addresses to lower addresses.
- This convention means that you push values onto the stack by subtracting from the stack pointer.
- Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Leaf Procedure Example

C/C++ Code

```
int leaf_example (int g, int h, int i, int j)
{
        int f;

        f = (g + h) – (i + j);
        return f;
}
```

- **Caller** put arguments g,..., j in x10, ..., x13 **(function arguments)**.
- **We** will put the result f in x20 **(saved registers)**.
- **We** will need two **temporaries x5, x6** to calculate the result.
- Need to **save** x5, x6, x20 on the **stack.**

Leaf Procedure Example

- The compiled program starts with the label of the procedure.
- The next step is to save the registers used by the procedure.
- We **“push”** the old values onto the stack by creating space for **three words (12 bytes)** on the stack and then store them.

```
leaf_example:
addi sp, sp, -12          // adjust stack to make room for 3 items
sw   x5, 8(sp)      // save register x5 for use afterwards
sw   x6, 4(sp)            // save register x6 for use afterwards
sw   x20, 0(sp)           // save register x20 for use afterwards
```

Leaf Procedure Example

- The next three statements correspond to the body of the procedure.
- To return the value of f, we copy it into a parameter register **x10**.

```
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)
```

Leaf Procedure Example

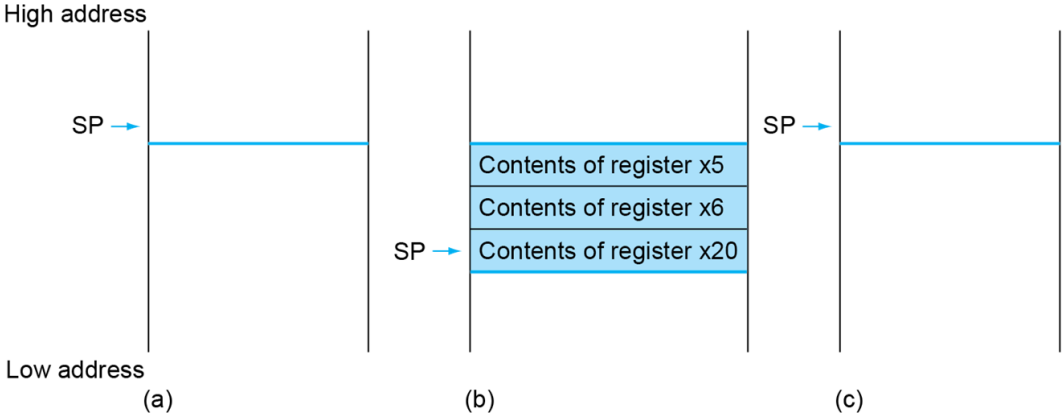
- Before returning, we restore the three old values of the registers we saved by **“popping”** them from the stack.
- The procedure ends with a branch register using the return address.

```
lw x20, 0(sp)    // restore register x20 for caller
lw x6, 4(sp)     // restore register x6 for caller
lw x5, 8(sp)     // restore register x5 for caller
addi sp, sp, 12  // adjust stack to delete 3 items
jalr x0, 0(x1)       // branch back to calling routine
```

Leaf Procedure Example

```
leaf_example:
addi sp, sp, -12          // adjust stack to make room for 3 items
sw   x5, 8(sp)      // save register x5 for use afterwards
sw   x6, 4(sp)            // save register x6 for use afterwards
sw   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)
lw x20, 0(sp)      // restore register x20 for caller
lw x6, 4(sp)       // restore register x6 for caller
lw x5, 8(sp)       // restore register x5 for caller
addi sp, sp, 12  // adjust stack to delete 3 items
jalr x0, 0(x1)       // branch back to calling routine
```

Leaf Procedure Example



Leaf Procedure

- In the previous example, we used temporary registers and assumed their old values must be saved and restored.
- To avoid this RISC-V software separates **19** of the registers into two groups:
  1. **x5−x7 and x28−x31**: temporary registers that are not preserved by the callee (called procedure) on a procedure call
  2. **x8−x9 and x18−x27**: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)
- This simple convention reduces register spilling.

Nested Procedures

- Procedures that do not call others are called **leaf procedures**.
- Procedures may invoke other procedures **non-leaf procedures**.
- Recursive procedures even invoke “clones” of themselves.
- **We need to be careful when using registers when invoking non-leaf procedures**.

Nested Procedures: Problem

- Main program calls procedure A with an argument of 3, by placing it into register x10.
- Procedure A calls procedure B with an argument of 7, also placed in x10.
- **What could be the problem here?**

```
addi x10, x0, 3
jal x1, A
addi x10, x0, 7
jal x1, B
```

Nested Procedures: Problem

- Since A hasn't finished its task yet, there is a conflict over the use of register x10.
- Similarly, there is a conflict over the return address in register x1, since it now has the return address for B.
- **This conflict will eliminate procedure A's ability to return to its caller**.

```
addi x10, x0, 3
jal x1, A
addi x10, x0, 7
jal x1, B
```

Nested Procedures: Solution

- **Push all registers that must be preserved on the stack**.
- The **caller** pushes any argument registers **(x10–x17)** or temporary registers **(x5-x7 and x28-x31)** that are needed after the call.
- The **callee** pushes the return address register **x1** and any saved registers **(x8-x9 and x18-x27)** used by the **callee**.
- The stack pointer sp is adjusted to account for the number of registers placed on the stack.
- Upon the return, the registers are restored from memory, and the stack pointer is readjusted.

Non-Leaf Procedure Example

- The parameter variable n corresponds to the argument register x10.

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

Non-Leaf Procedure Example

- The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and x10.

```
fact:
addi sp, sp, -8    // adjust stack for 2 items
  sw x1, 4(sp)     // save the return address
  sw x10, 0(sp)    // save the argument n
```

Non-Leaf Procedure Example

- The next two instructions test whether n is less than 1, going to L1 if n $\geq$ 1.
- If n is less than 1, fact returns 1 by putting 1 into x10.
- It then pops the two saved values off the stack and branches to the return address:

```
addi    x5, x10, -1        // x5 = n - 1
bge     x5, x0, L1         // if (n - 1) >= 0, go to L1
addi    x10, x0, 1         // return 1
addi    sp, sp, 8          // pop 2 items off stack
jalr    x0, 0(x1)          // return to caller
```

Non-Leaf Procedure Example

- If n is not less than 1, the argument n is decremented and then fact is called again with the decremented value.
- The next instruction is where fact returns; its result is in x10.
- Now the old return address and old argument are restored, along with the stack pointer.

```
L1: addi x10, x10, -1  // n >= 1: argument gets (n – 1)
    jal  x1, fact      // call fact with (n – 1)
addi  x6, x10, 0    // return from jal: move result of fact
                       (n - 1) to x6:
lw    x10, 0(sp)    // restore argument n
lw    x1, 4(sp)     // restore the return address
addi  sp, sp, 8     // adjust stack pointer to pop 2 items
```

Non-Leaf Procedure Example

- Next, argument register x10 gets the product of the old argument and the result of fact(n - 1), now in x6.
- The multiply instruction will be covered later.
- Finally, fact branches again to the return address.

```
mul     x10, x10, x6     // return n * fact (n – 1)
jalr    x0, 0(x1)        // return to the caller
```

Non-Leaf Procedure Example

```
addi x10, x0, 5
fact:
addi sp, sp, -8 # adjust stack for 2 items
sw x1, 4(sp) # save the return address
sw x10, 0(sp) # save the argument n
addi x5, x10, -1 # x5 = n - 1
bge x5, x0, L1 # if (n - 1) >= 0, go to L1
addi x10, x0, 1 # return 1
addi sp, sp, 8 # pop 2 items off stack
jalr x0, 0(x1) # return to caller
L1: addi x10, x10, -1 # n >= 1: argument gets (n - 1)
jal x1, fact # call fact with (n - 1)
addi x6, x10, 0 # return from jal: move result of fact (n - 1) to x6:
lw x10, 0(sp) # restore argument n
lw x1, 4(sp) # restore the return address
addi sp, sp, 8 # adjust stack pointer to pop 2 items
mul x10, x10, x6 # return n * fact (n - 1)
jalr x0, 0(x1) # return to the caller
```

Nested Procedures

- The stack above **sp** is preserved simply by making sure the callee does not write above **sp**;
- **sp** is itself preserved by the callee adding exactly the same amount that was subtracted from it;
- The other registers are preserved by saving them on the stack (if they are used) and restoring them from there.

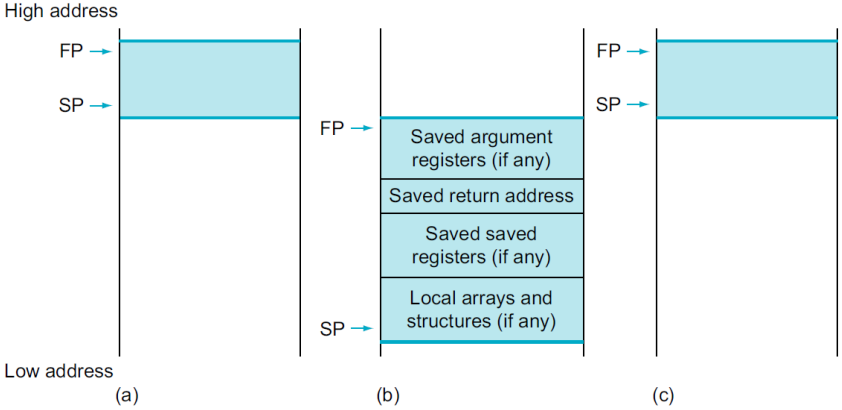
Preserved	Not preserved
Saved registers: `x8-x9, x18-x27`	Temporary registers: `x5-x7, x28-x31`
Stack pointer register: `x2(sp)`	Argument/result registers: `x10-x17`
Frame pointer: `x8(fp)`	
Return address: `x1(ra)`	
Stack above the stack pointer	Stack below the stack pointer

Allocating Space for New Data on the Stack

- The final complexity is that the stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures.
- The segment of the stack containing a procedure's saved registers and local variables is called a **procedure frame** or **activation record**.
- Some RISC-V compilers use a **frame pointer fp**, or register **x8** to point to the first word of the frame of a procedure.

Allocating Space for New Data on the Stack

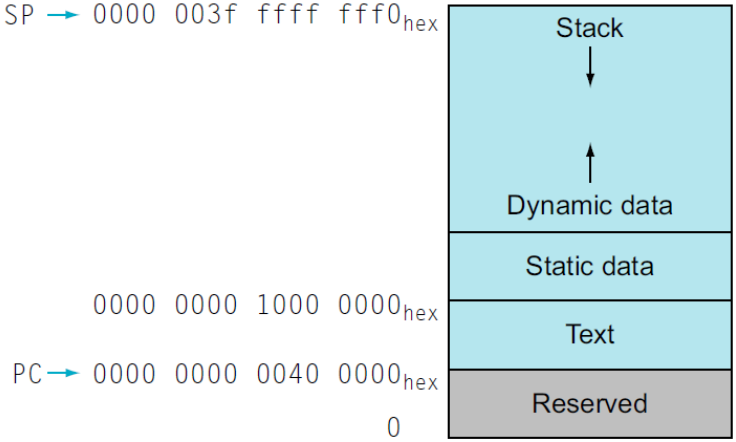
- The state of the stack before, during, and after the procedure call.



Allocating Space for New Data on the Stack

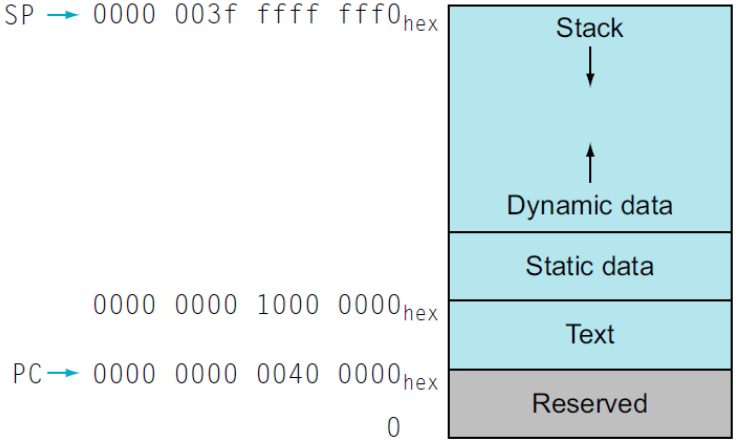
- A stack pointer might change during the procedure, so references to a local variable in memory might have different offsets.
- Alternatively, a frame pointer offers a stable base register within a procedure for local memory references.
- Note that an activation record. appears on the stack whether or not an explicit frame pointer is used.

Allocating Space for New Data on the Heap



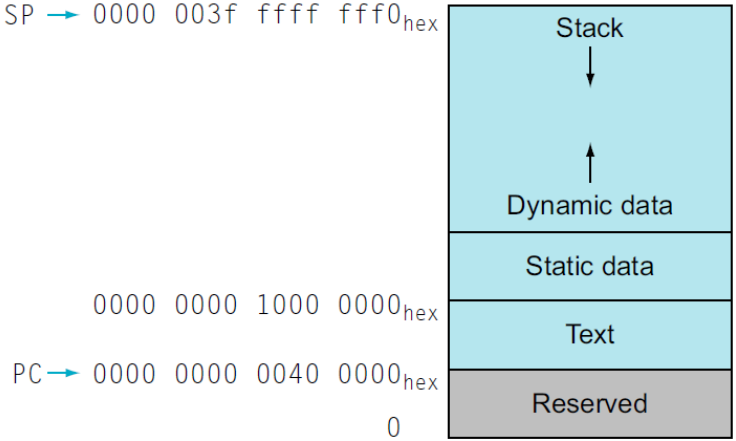
- Programmers also need space in memory for static variables and dynamic data structures.
- The stack starts in the high end of the user addresses space and grows down.

Allocating Space for New Data on the Heap



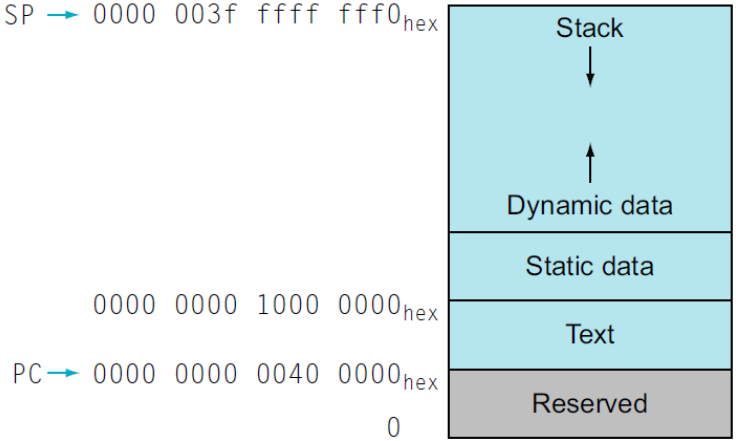
- The first part of the low end of memory is reserved.
- It is followed by the home of the RISC-V machine code, traditionally called the **text segment**.

Allocating Space for New Data on the Heap



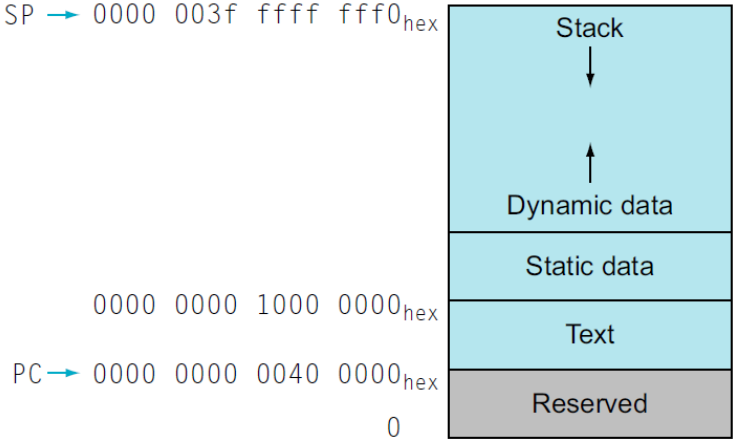
- Above the code is the **static data** segment, which is the place for constants and other static variables.

Allocating Space for New Data on the Heap



- Data structures like linked lists tend to grow and shrink during their lifetimes. The segment for such data structures is traditionally called the **heap**, and it is placed next in memory.

Allocating Space for New Data on the Heap



- Note that this allocation allows the **stack** and **heap** to **grow toward each other**, thereby allowing the efficient use of memory as the two segments wax and wane.

Allocating Space: Example

- A C variable is generally a location in storage, and its interpretation depends both on its **type** (int, char..., etc.) and **storage class**.
- C has two storage classes: **automatic** and **static**.
- **Automatic** variables are local to a procedure and are discarded when the procedure exits.
- **Static** variables exist across exits from and entries to procedures.
- C variables declared outside all procedures are considered static, as are any variables declared using the keyword static.
- The rest are automatic.
- To simplify access to static data, some RISC-V compilers reserve a register **x3** for use as the **global pointer**, or **gp**.

Allocating Space: Example

- C allocates and frees space on the heap with explicit functions.
- **malloc()** allocates space on the **heap** and returns a pointer to it, and **free()** releases space on the heap to which the pointer points.
- Forgetting to free space leads to a **“memory leak”**.
- Freeing space too early leads to **“dangling pointers”**.
- Java uses automatic memory allocation and garbage collection primarily to avoid such bugs.

RISC-V register conventions

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

RISC-V register conventions

- **What if there are more than eight parameters?**
- The RISC-V convention is to place the extra parameters on the stack **just above the frame pointer**.
- The procedure then expects the first eight parameters to be in registers x10 through x17 and the rest in memory, addressable via the frame pointer.

ensia

Representing Instructions in the Computer

Representing Instructions in the Computer

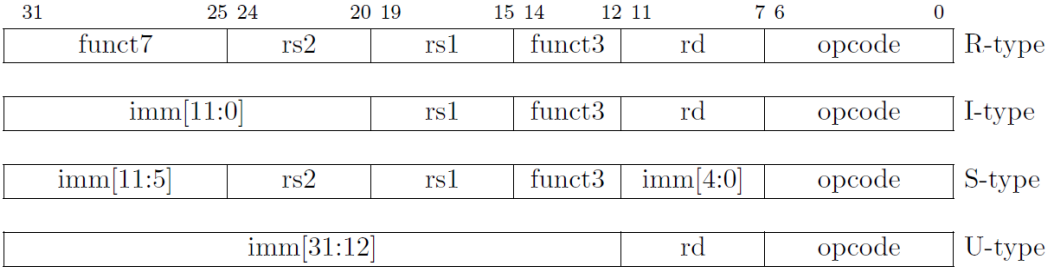
- Each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction.
- Each of these segments of the instruction is called a **field**.
- This layout of the instruction is called the **instruction format**.
- RISC-V instructions are all 32 bits long.
- ***Instruction format:*** *A form of representation of an instruction composed of fields of binary numbers.*

Representing Instructions in the Computer

- To distinguish it from assembly language, we call the numeric version of instructions **machine language** and a sequence of such instructions **machine code**.
- ***Machine language:*** *Binary representation used for communication within a computer system.*
- The 32 registers of RISC-V are just referred to by their number, from 0 to 31 (instead of x0 to x31).

Base Instruction Formats

- In the base ISA, there are four core instruction formats **(R/I/S/U)**.



RISC-V Fields: R-format (for register)

- **opcode**: Basic operation of the instruction, and this abbreviation is its traditional name. It denotes also the format of an instruction.
- **rd**: The register destination operand. It gets the result of the operation.
- **funct3**: An additional opcode field.
- **rs1**: The first register source operand.
- **rs2**: The second register source operand.
- **funct7**: An additional opcode field.

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

R-format instructions

funct7	rs2	rs1	funct3	rd	opcode	
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 Fields: I-format

- The 12-bit immediate is interpreted as a two's complement value, so it can represent integers from -2^{11} to $2^{11}-1$.
- When the I-type format is used for load instructions, the immediate represents a byte offset, so the load word instruction can refer to any word within a region of $\pm 2^{11}$ or 2048 bytes (± 28 or 512 words) of the base address in the base register rd.

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

I-format example: lw

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

```
. lw x9, 32(x22) // Temporary reg x9 gets A[8]
```

- Here, 22 (for x22) is placed in the rs1 field, 32 is placed in the immediate field, and 9 (for x9) is placed in the rd field.

I-format instructions

immediate		rs1	funct3	rd	opcode	
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

One of the higher-order immediate bits is used to distinguish “shift right logical” (SRLI) from “shift right arithmetic” (SRAI)

“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

I-format instructions

immediate	rs1	funct3	rd	opcode	
imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	010	rd	0000011	lh
imm[11:0]	rs1	011	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	110	rd	0000011	lhu

funct3 field encodes size and 'signedness' of load data

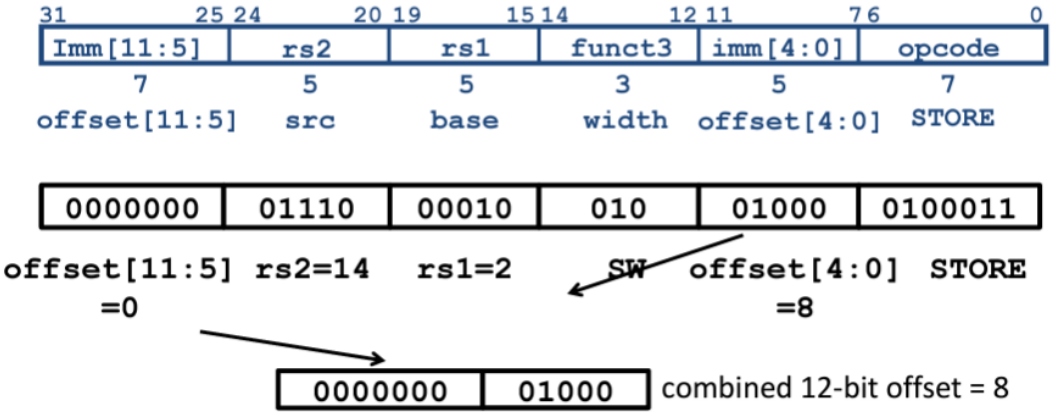
RISC-V Fields: S-format

- The 12-bit immediate in the S-type format is split into two fields, which supply the lower 5 bits and upper 7 bits.
- The RISC-V architects chose this design because it keeps the rs1 and rs2 fields in the same place in all instruction formats.

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

RISC-V Fields: S-format

- **sw x14, 8(x2)**

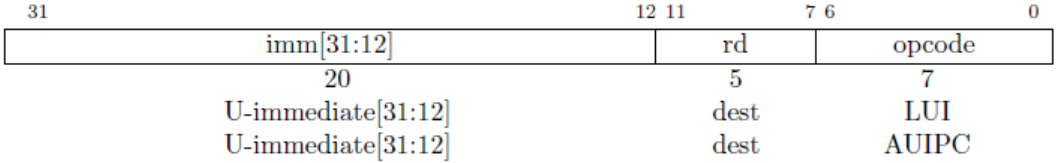


S-format instructions

immediate	rs2	rs1	funct3	immediate	opcode	
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

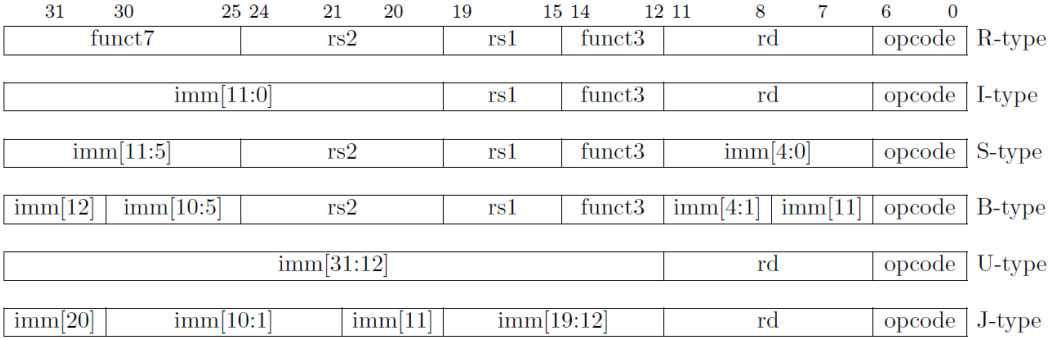
RISC-V Fields: U-format

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word.
- One destination register, rd.
- Used for two instructions
  1. **LUI**: Load Upper Immediate.
  2. **AUIPC**: Add Upper Immediate to PC.



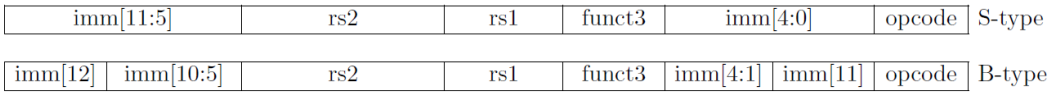
Immediate Encoding Variants

- There are two variants of the instruction formats **(B/J)** based on the handling of immediates.



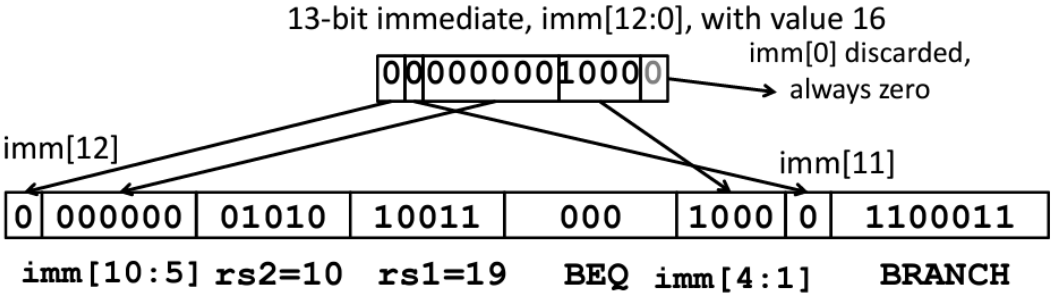
RISC-V Fields: B-format

- The only difference between the **S** and **B** formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format.
- Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done.
- The middle bits (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format **imm[11]**.



B-format: Example

```
beq   x19,x10, offset = 16 bytes
```

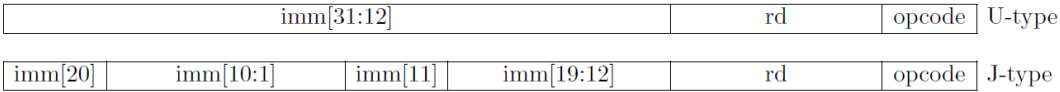


B-format instructions

immediate	rs2	rs1	funct3	immediate	opcode	
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

RISC-V Fields: J-format

- Similarly, the only difference between the **U** and **J** formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediate.
- The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.



ensia

RISC-V Addressing for Wide Immediates and Addresses

Wide Immediate Operands

- Although constants are frequently short and fit into the 12-bit fields, sometimes they are bigger.
- The RISC-V instruction set includes the instruction **Load upper immediate (lui)** to load a 20-bit constant into bits 12 through 31 of a register.
- The rightmost 12 bits are filled with zeros.
- This instruction allows, for example, a 32-bit constant to be created with two instructions.
- **lui** uses U-type instruction format, as the other formats cannot accommodate such a large constant.

Wide Immediate Operands: Example

- load this 32-bit constant into register x19:
  `00000000 00111101 00000101 00000000`
- First, we would load bits 12 through 31 with that bit pattern, which is 976 in decimal, using **lui**:
  `lui x19, 976 // 976`$_{\text{decimal}}$ `= 0000 0000 0011 1101 0000`
- The value of register x19 afterward is:
  `00000000 00111101 00000000 00000000`
- The next step is to add in the lowest 12 bits, whose decimal value is 1280:
  `addi x19, x19, 1280 // 1280`$_{\text{decimal}}$ `= 00000101 00000000`

Addressing in Branches

- The RISC-V branch instructions use a RISC-V instruction format with a 12-bit immediate.
- This format can represent branch addresses from -4096 to 4094 ($\pm 2^{10}$ words), in multiples of 2.
- The unconditional jump-and-link instruction (jal) uses a 20-bit address immediate, thus $\pm 2^{18}$ words.
- The address uses an unusual encoding, which simplifies datapath design but complicates assembly.

Addressing in Branches

- Conditional branches are found in loops and in if statements, so they tend to branch to a nearby instruction.
- About half of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away.
- On the other hand, procedure calls may require jumping more than 2^{18} words away, since there is no guarantee that the callee is close to the caller.
- Hence, RISC-V allows very long jumps to any 32-bit address with a two-instruction sequence: **lui** writes bits 12 through 31 of the address to a temporary register, and **jalr** adds the lower 12 bits of the address to the temporary register and jumps to the sum.

Addressing in Branches: Example

```
Loop:slli x10, x22, 2     // Temp reg x10 = i * 4
     add  x10, x10, x25   // x10 = address of save[i]
     lw   x9, 0(x10)      // Temp reg 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:
```

Addressing in Branches: Example

- The **bne** instruction on the fourth line adds 3 words or 12 bytes to the address of the instruction, specifying the branch destination relative to the branch instruction (12 + 80012) and not using the full destination address (80024).

Address	Instruction					
80000	0000000	00010	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Addressing in Branches: Example

- The branch instruction on the last line does a similar calculation for a backwards branch (−20 + 80020), corresponding to the label Loop.

Address	Instruction					
80000	0000000	00010	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Branching Far Away: Example

- Given a branch on register x10 being equal to zero,

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

- These instructions replace the short-address conditional branch:

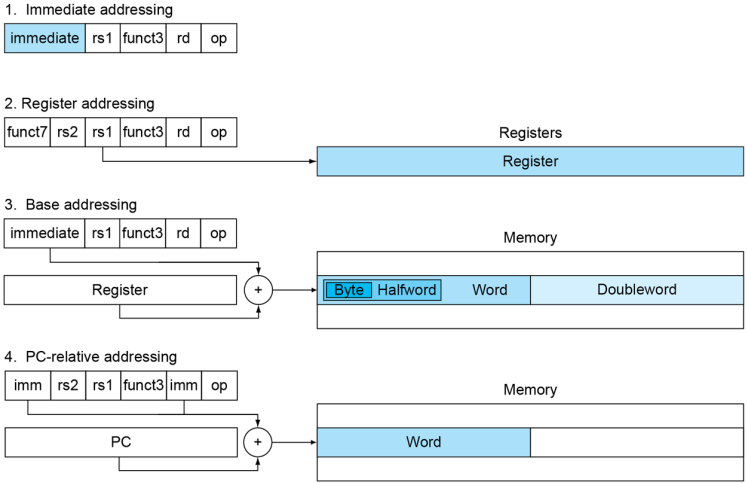
```
        bne     x10, x0, L2
        jal     x0, L1
L2:
```

RISC-V Addressing Summary

Multiple forms of addressing are generically called **addressing modes**.

- **Immediate addressing**, where the operand is a constant within the instruction itself.
- **Register addressing**, where the operand is a register.
- **Base or displacement addressing**, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction.
- **PC-relative addressing**, where the branch address is the sum of the PC and a constant in the instruction.

RISC-V Addressing Summary



ensia

Translating RISC-V Assembly Language into Machine Language

Translating RISC-V Assembly Language into Machine Language

- If x10 has the base of the array A and x21 corresponds to h, the assignment statement:

```
A[30] = h + A[30] + 1;
```

- is compiled into:

```
lw   x9, 120(x10)   // Temporary reg x9 gets A[30]
add  x9, x21, x9    // Temporary reg x9 gets h+A[30]
addi x9, x9, 1      // Temporary reg x9 gets h+A[30]+1
sw   x9, 120(x10)   // Stores h+A[30]+1 back into A[30]
```

Translating RISC-V Assembly Language into Machine Language

```
lw   x9, 120(x10)   // Temporary reg x9 gets A[30]
```

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
lw (load word)	I	address	reg	010	reg	0000011

- The lw instruction is identified by 3 in the opcode field and 2 in the funct3 field. The base register 10 is specified in the rs1 field, and the destination register 9 is specified in the rd field. The offset to select A[30] (120 = 30 × 4) is found in the immediate field.

immediate	rs1	funct3	rd	opcode
120	10	2	9	3

Translating RISC-V Assembly Language into Machine Language

```
add  x9, x21, x9     // Temporary reg x9 gets h+A[30]
```

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

- The add instruction is specified with 51 in the opcode field, 0 in the funct3 field, and 0 in the funct7 field. The three register operands (9, 21, and 9) are found in the rd, rs1, and rs2 fields.

funct7	rs2	rs1	funct3	rd	opcode
0	9	21	0	9	51

Translating RISC-V Assembly Language into Machine Language

```
addi x9, x9, 1        // Temporary reg x9 gets h+A[30]+1
```

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
lw (load word)	I	address	reg	010	reg	0000011

- The subsequent addi instruction is specified with 19 in the opcode field and 0 in the funct3 field. The register operands (9 and 9) are found in the rd and rs1 fields, and the constant addend 1 is found in the immediate field.

immediate	rs1	funct3	rd	opcode
1	9	0	9	19

Translating RISC-V Assembly Language into Machine Language

```
sw    x9, 120(x10)    // Stores h+A[30]+1 back into A[30]
```

Instruction	Format	immed -iate	rs2	rs1	funct3	immed -iate	opcode
sw (store word)	S	address	reg	reg	010	address	0100011

- The sw instruction is identified with 35 in the opcode field and 2 in the funct3 field. The register operands (9 and 10) are found in the rs2 and rs1 fields, respectively. The address offset 120 is split across the two immediate fields. The upper part of the immediate holds the quotient, 3, and the lower part holds the remainder, 24, since 120 = 0000011 11000

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
3	9	10	2	24	35

Translating RISC-V Assembly Language into Machine Language: Example 2

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	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
lw (load word)	001111101000		00010	010	00001	0000011	lw x1, 1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sw (store word)	0011111	00001	00010	010	01000	0100011	sw x1, 1000(x2)

Decoding Machine Language

- Sometimes you are forced to reverse-engineer machine language to create the original assembly language.
- One example is when looking at “core dump.”

Decoding Machine Language

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100

Decoding Machine Language

Format	Instruction	Opcode	Funct3	Funct6/7
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.

Decoding Machine Language

Format	Instruction	Opcode	Funct3	Funct6/7
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

Decoding Machine Language: Example

- 00578833_{hex}
- The first step is converting hexadecimal to binary:

  0000 0000 0101 0111 1000 1000 0011 0011
- Then, we need to determine the opcode (the rightmost 7 bits) **0110011**=**R-type.**

funct7	rs2	rs1	funct3	rd	opcode
0000000	00101	01111	000	10000	0110011

- The funct7 and funct3 fields are both zero, indicating the instruction is **add**.

  `add x16, x15, x5`

The Rest of the RISC-V Instruction Set

- RISC-V architects partitioned the instruction set into a base architecture and several extensions.
- Each is named with a letter of the alphabet, and the base architecture is named I for integer.

Mnemonic	Description	Insn. Count
I	Base architecture	51
M	Integer multiply/divide	13
A	Atomic operations	22
F	Single-precision floating point	30
D	Double-precision floating point	32
C	Compressed instructions	36

ensia

Thank you!