RISC-V Architecture I

’22H2

송 인 식

Outline

- RISC-V Overview
- Arithmetic / Logical Operations
- Data Transfer Operations

Architecture

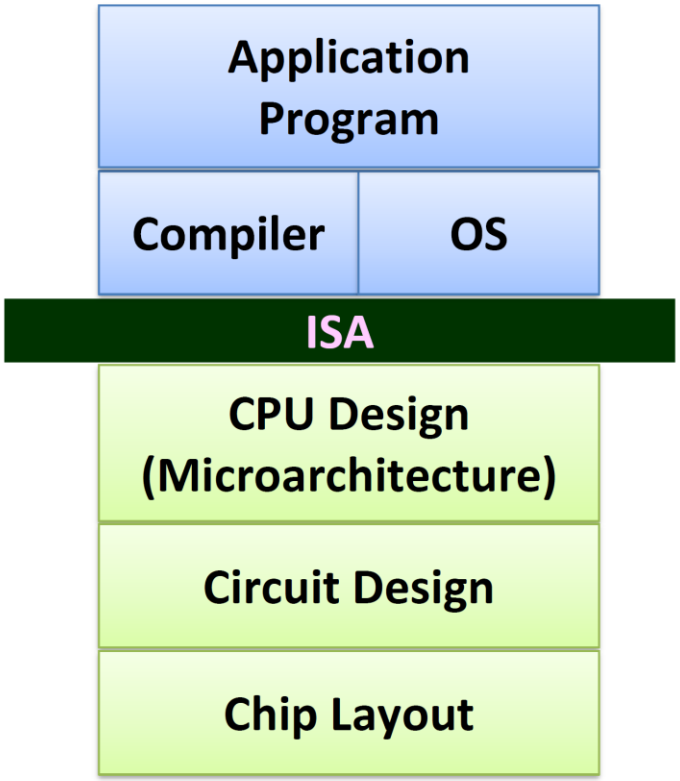
"the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation"

-- Amdahl, Blaauw, and Brooks, Architecture of the IBM System/360, IBM Journal of Research and Development, April 1964.

- The visible interface between software and hardware
- What the user (OS, compiler, …) needs to know to reason about how the machine behaves
- Abstracted from the details of how it may accomplish its task

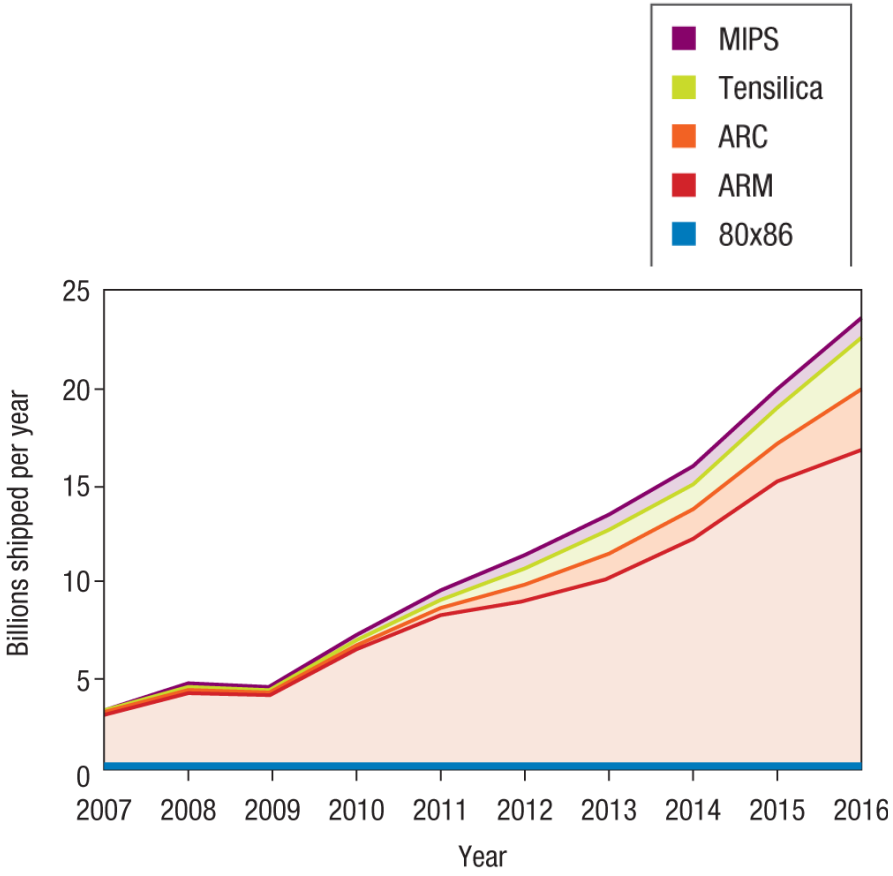
Instruction Set Architecture

- Above: how to program machine
  - Processors execute instructions in sequence
- Below: what needs to be built
  - Use variety of tricks to make it run fast
- Instruction set
- Processor registers
- Memory addressing modes
- Data types and representations
- Byte ordering, …



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
  - RISC (Reduced Instruction Set Computer)



The RISC-V Instruction Set

- A completely open ISA that is freely available to academia and industry
- Fifth RISC ISA design developed at UC Berkeley
  - RISC-I (1981), RISC-II (1983), SOAR (1984), SPUR (1989), and RISC-V (2010)
- Now managed by the RISC-V Foundation (http://riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Card (or Green Card)
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Why Freely Open ISA?

- Greater innovation via free-market competition
  - From many core designers, closed-source and open-source
- Shared open core designs
  - Shorter time to market, lower cost from reuse, fewer errors given more eyeballs, transparency makes it difficult for government agencies to add secret trap doors
- Processors becoming affordable for more devices
  - Help expand the Internet of Things (IoTs), which could cost as little as $1
- Software stack survive for long time
- Make architectural research and education more real
  - Fully open hardware and software stacks

RISC-V ISAs

- Three base integer ISAs, one per address width
  - RV32I, RV64I, RV128I
  - RV32I: Only 40 instructions defined
  - RV32E: Reduced version of RV32I with 16 registers for embedded systems
- Standard extensions
  - Standard RISC encoding in a fixed 32-bit instruction format
  - C extension offers shorter 16-bit versions of common 32-bit RISC-V instructions (can be intermixed with 32-bit instructions)

Name	Extension
M	Integer Multiply/Divide
A	Atomic Instructions
F	Single-precision FP
D	Double-precision FP
G	General-purpose (= IMAFD)
Q	Quad-precision FP
C	Compressed Instructions

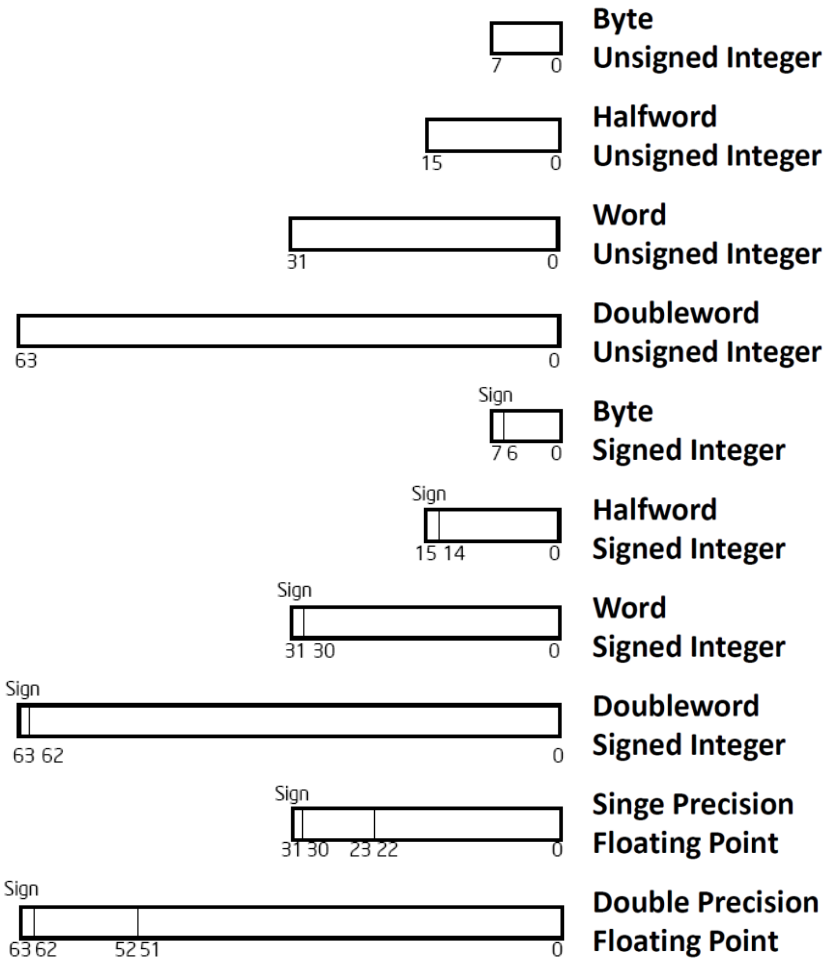
RISC-V Registers

Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

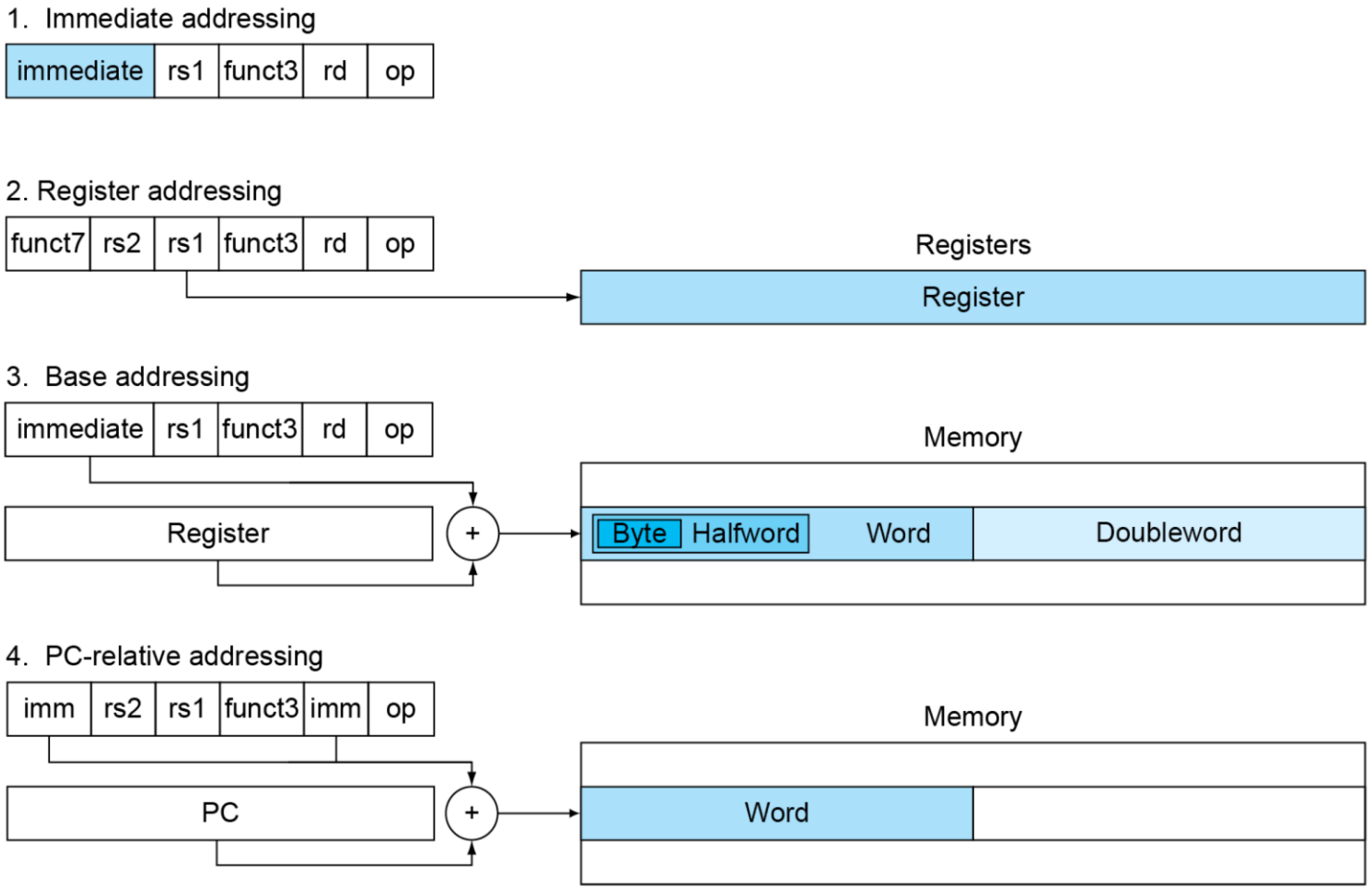
pc Program counter

Data Types

- Integer data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4 or 8 bytes (with F or D extension)
- No aggregated types such as arrays or structures
  - Just contiguously allocated bytes in memory



RISC-V Addressing



Operations

- Perform an arithmetic or logical function on register data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jump
  - Conditional branch
  - Procedure call and return

RISC-V Tutorial @ HotChips ‘19

- https://youtu.be/nPXdbm9lc3A

Outline

- RISC-V Overview
- Arithmetic / Logical Operations
- Data Transfer Operations

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  // a ← b + c
sub a, b, c  // a ← b - c
```

- All arithmetic operations have this form
- Design Principle 1: Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

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
```

Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32 × 64-bit register file: x0 ~ x31
  - Use for frequently accessed data
  - 64-bit data is called a “doubleword”
  - 32-bit data is called a “word”
- Design Principle 2: Smaller is faster
  - c.f. main memory: millions of locations

Register Operand Example

- C code:

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

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

- Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Registers vs. Memory

- Registers are faster to access than memory
- In RISC-V, data in memory cannot be directly address ed by ALU instructions
- 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!

Immediate Operands

- Constant data specified in an instruction

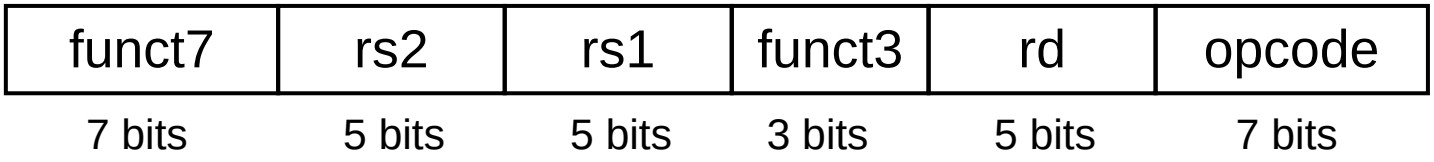
```
addi x22, x22, 4
```

- Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Representing Instructions

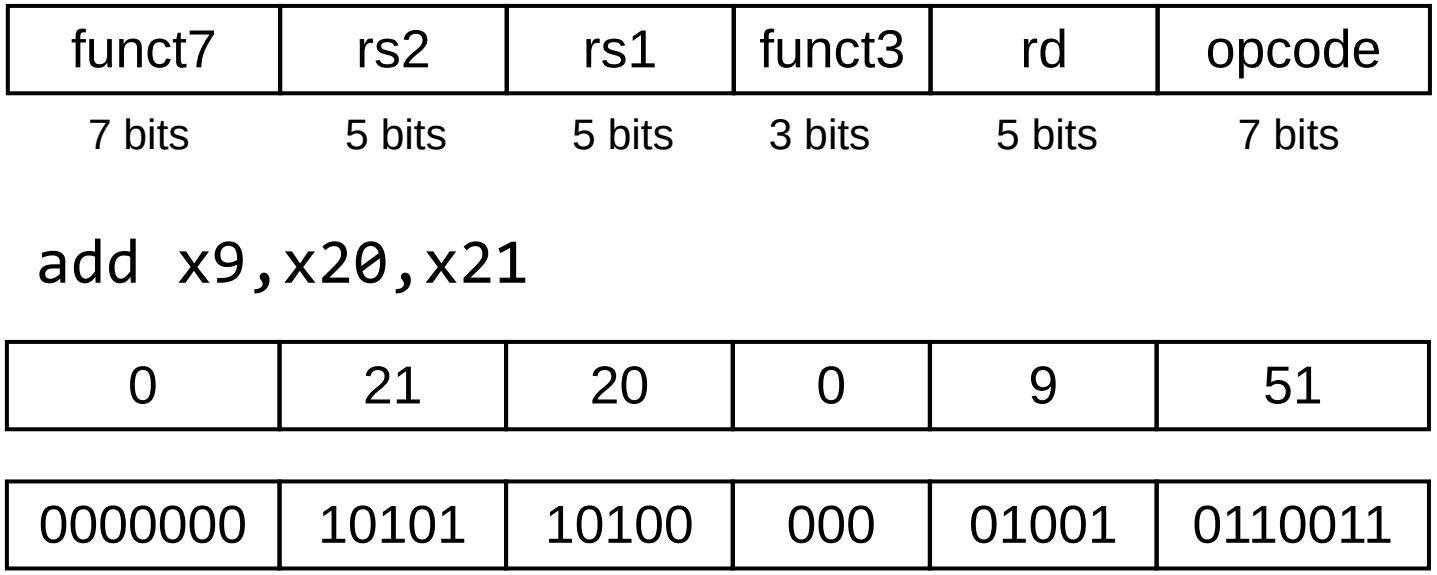
- Instructions are encoded in binary
  - Called machine code
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

RISC-V R-format Instructions



- 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 Example

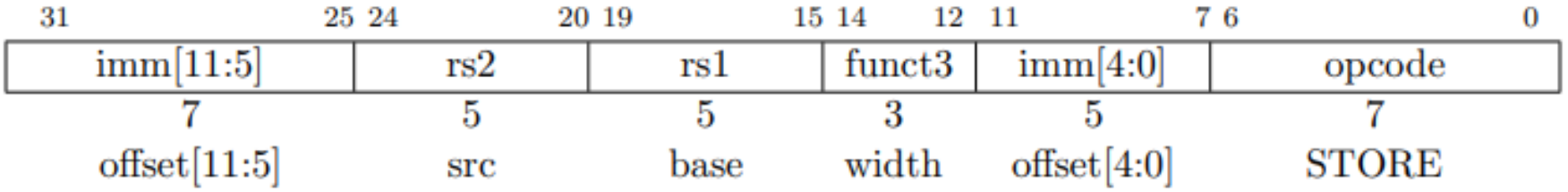


$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

RISC-V I-format Instructions

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
- *Design Principle 3*: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

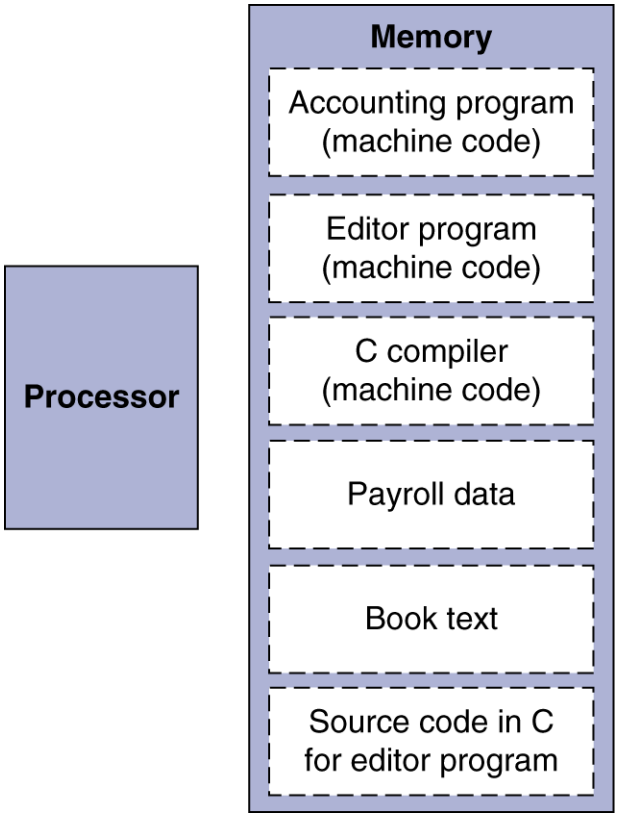
RISC-V S-format Instructions



- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

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

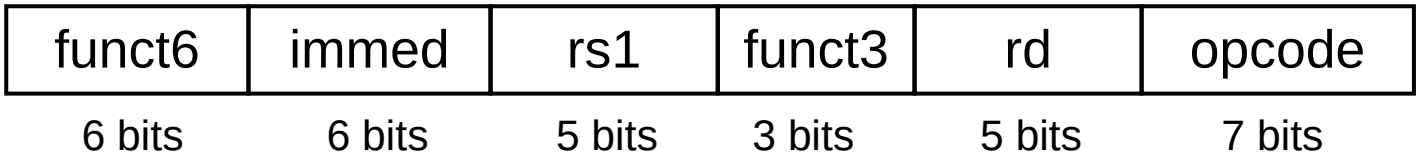
Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	slli
Shift right	>>	>>>	srli
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	

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

Shift Operations

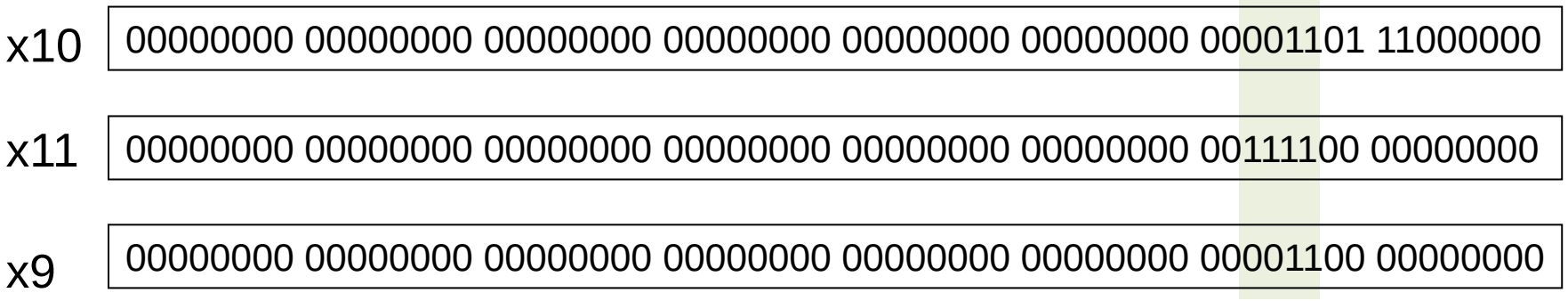


- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `slli` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srli` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

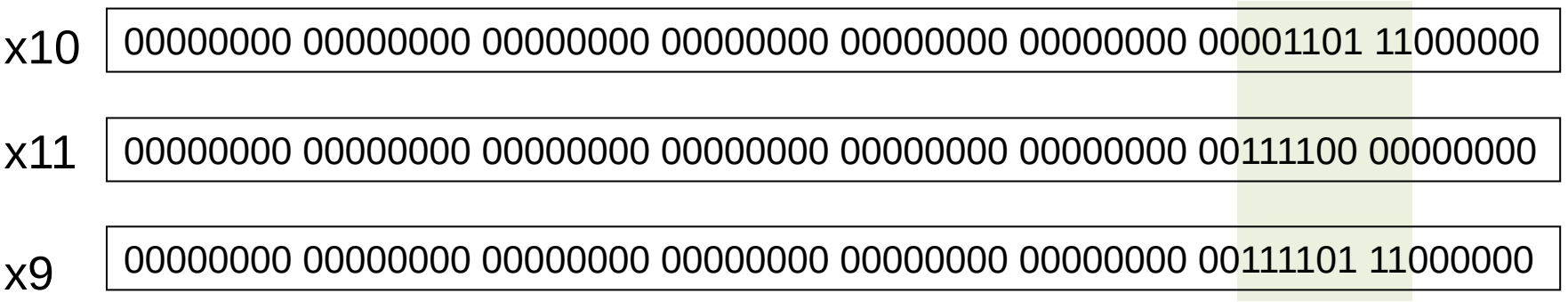
```
and x9,x10,x11
```



OR Operations

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

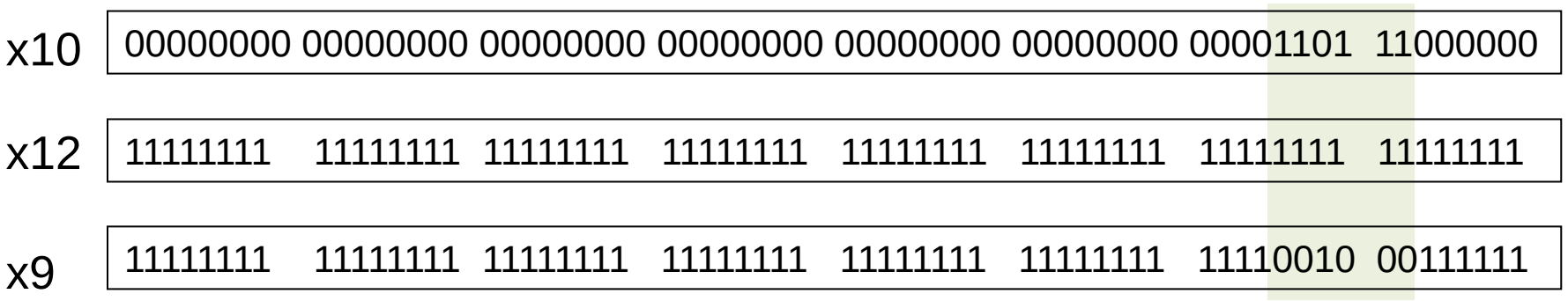
```
or x9,x10,x11
```



XOR Operations

- Differencing operation
  - Set some bits to 1, leave others unchanged

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



32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant:
  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0
- Example: x19 <- 0x003D0500

```
lui rd, constant
```

```
lui  x19, 0x003D0
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

```
addi x19,x19,0x500
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

Arithmetic Operations

Instruction	Type	Example	Meaning
Add	**R**	`add rd, rs1, rs2`	`R[rd] = R[rs1] + R[rs2]`
Subtract	**R**	`sub rd, rs1, rs2`	`R[rd] = R[rs1] – R[rs2]`
Add immediate	**I**	`addi rd, rs1, imm12`	`R[rd] = R[rs1] + SignExt(imm12)`
Set less than	**R**	`slt rd, rs1, rs2`	`R[rd] = (R[rs1] < R[rs2])? 1 : 0`
Set less than immediate	**I**	`slti rd, rs1, imm12`	`R[rd] = (R[rs1] < SignExt(imm12))? 1 : 0`
Set less than unsigned	**R**	`sltu rd, rs1, rs2`	`R[rd] = (R[rs1]` $<_u$ `R[rs2])? 1 : 0`
Set less than immediate unsigned	**I**	`sltiu rd, rs1, imm12`	`R[rd] = (R[rs1]` $<_u$ `SignExt(imm12))? 1 : 0`
Load upper immediate	**U**	`lui rd, imm20`	`R[rd] = SignExt(imm20 << 12)`
Add upper immediate to PC	**U**	`auipc rd, imm20`	`R[rd] = PC + SignExt(imm20 << 12)`

Logical Operations

Instruction	Type	Example	Meaning
AND	**R**	`and rd, rs1, rs2`	`R[rd] = R[rs1] & R[rs2]`
OR	**R**	`or rd, rs1, rs2`	`R[rd] = R[rs1] | R[rs2]`
XOR	**R**	`xor rd, rs1, rs2`	`R[rd] = R[rs1] ^ R[rs2]`
AND immediate	**I**	`andi rd, rs1, imm12`	`R[rd] = R[rs1] & SignExt(imm12)`
OR immediate	**I**	`ori rd, rs1, imm12`	`R[rd] = R[rs1] | SignExt(imm12)`
XOR immediate	**I**	`xori rd, rs1, imm12`	`R[rd] = R[rs1] ^ SignExt(imm12)`
Shift left logical	**R**	`sll rd, rs1, rs2`	`R[rd] = R[rs1] << R[rs2]`
Shift right logical	**R**	`srl rd, rs1, rs2`	`R[rd] = R[rs1] >> R[rs2]` *(logical)*
Shift right arithmetic	**R**	`sra rd, rs1, rs2`	`R[rd] = R[rs1] >> R[rs2]` *(arithmetic)*
Shift left logical immediate	**I**	`slli rd, rs1, shamt`	`R[rd] = R[rs1] << shamt`
Shift right logical imm.	**I**	`srli rd, rs1, shamt`	`R[rd] = R[rs1] >> shamt` *(logical)*
Shift right arithmetic immediate	**I**	`srai rd, rs1, shamt`	`R[rd] = R[rs1] >> shamt` *(arithmetic)*

Example: arith

```
long arith (long x,
            long y,
            long z) {
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 - t5;
    return rval;
}
```

```
arith:
   add   a5, a0, a1        # a5 = x + y (t1)
   add   a2, a5, a2        # a2 = t1 + z (t2)
   addi  a0, a0, 4         # a0 = x + 4 (t3)
   slli  a5, a1, 1         # a5 = y * 2
   add   a1, a5, a1        # a1 = a5 + y
   slli  a5, a1, 4         # a5 = a1 * 16 (t4)
   add   a0, a0, a5        # a0 = t3 + t4 (t5)
   sub   a0, a2, a0        # a0 = t2 - t5 (rval)
   ret
```

```
x in a0
y in a1
z in a2
```

Example: logical

```
long logical (long x,
              long y) {
    long t1 = x ^ y;
    long t2 = t1 >> 17;
    long mask = (1 << 8) - 7;
    long rval = t2 & mask;
    return rval;
}
```

```
logical:
   xor   a0, a0, a1      # a0 = x ^ y (t1)
   srai  a0, a0, 17      # a0 = t1 >> 17 (t2)
   andi  a0, a0, 249     # a0 = t2 & ((1 << 8) - 7)
   ret
```

```
x in a0
y in a1
```

Outline

- RISC-V Overview
- Arithmetic / Logical Operations
- Data Transfer Operations

Memory Operands

- 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
- RISC-V does not require words to be aligned in memory
  - Unlike some other ISAs

Memory Operand Example

- C code:

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

  - h in x21, base address of A in x22
- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword
    - &A[8] = A + 64

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

Byte/Halfword/Word Operations

- Load byte/halfword/word:
  Sign extend to 64 bits in rd

```
lb    rd, offset(rs1)
lh    rd, offset(rs1)
lw    rd, offset(rs1)
```

- Load byte/halfword/word:
  Zero extend to 64 bits in rd

```
lbu   rd, offset(rs1)
lhu   rd, offset(rs1)
lwu   rd, offset(rs1)
```

- Store byte/halfword/word:
  Store rightmost 8/16/32 bits

```
sb    rs2, offset(rs1)
sh    rs2, offset(rs1)
sw    rs2, offset(rs1)
```

Data Transfer Operations

Instruction	Type	Example	Meaning
Load doubleword	**I**	`ld rd, imm12(rs1)`	R[rd] = Mem_8[R[rs1] + SignExt(imm12)]
Load word	**I**	`lw rd, imm12(rs1)`	R[rd] = SignExt(Mem_4[R[rs1] + SignExt(imm12)])
Load halfword	**I**	`lh rd, imm12(rs1)`	R[rd] = SignExt(Mem_2[R[rs1] + SignExt(imm12)])
Load byte	**I**	`lb rd, imm12(rs1)`	R[rd] = SignExt(Mem_1[R[rs1] + SignExt(imm12)])
Load word unsigned	**I**	`lwu rd, imm12(rs1)`	R[rd] = ZeroExt(Mem_4[R[rs1] + SignExt(imm12)])
Load halfword unsigned	**I**	`lhu rd, imm12(rs1)`	R[rd] = ZeroExt(Mem_2[R[rs1] + SignExt(imm12)])
Load byte unsigned	**I**	`lbu rd, imm12(rs1)`	R[rd] = ZeroExt(Mem_1[R[rs1] + SignExt(imm12)])
Store doubleword	**S**	`sd rs2, imm12(rs1)`	Mem_8[R[rs1] + SignExt(imm12)] = R[rs2]
Store word	**S**	`sw rs2, imm12(rs1)`	Mem_4[R[rs1] + SignExt(imm12)] = R[rs2](31:0)
Store halfword	**S**	`sh rs2, imm12(rs1)`	Mem_2[R[rs1] + SignExt(imm12)] = R[rs2](15:0)
Store byte	**S**	`sb rs2, imm12(rs1)`	Mem_1[R[rs1] + SignExt(imm12)] = R[rs2](7:0)

Swap Example

- Source code in C:

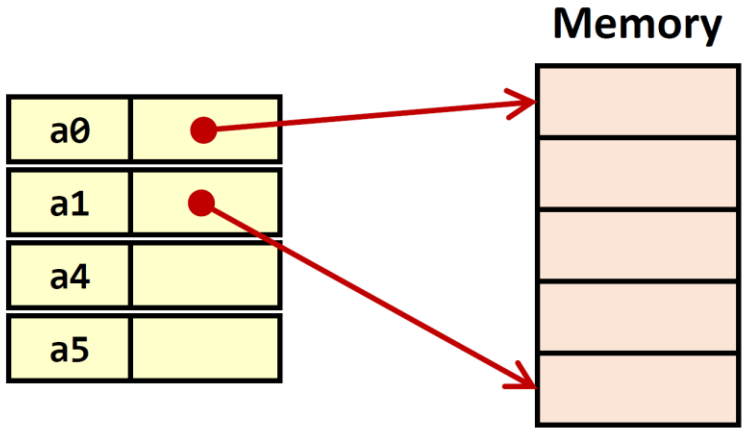
```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- Corresponding assembly code:

```
swap:
    ld    a4, 0(a0)
    ld    a5, 0(a1)
    sd    a5, 0(a0)
    sd    a4, 0(a1)
    ret
```

Understanding Swap - 1

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

Understanding Swap - 2

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

a0	0x120
a1	0x100
a4	
a5	

Memory

0x120	123
0x118	
0x110	
0x108	
0x100	456

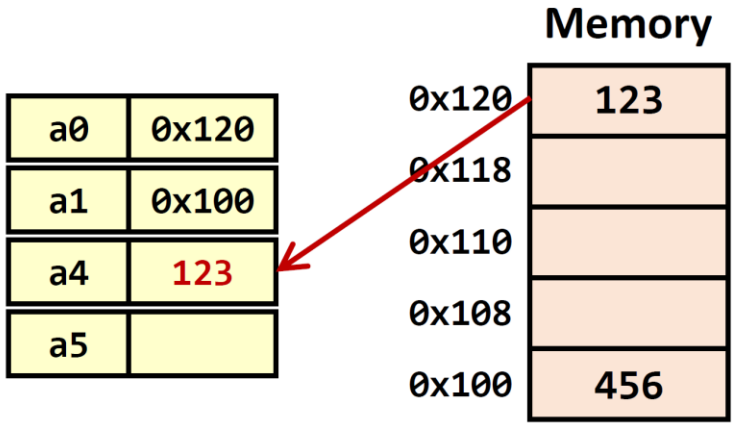
Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

Understanding Swap - 3

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



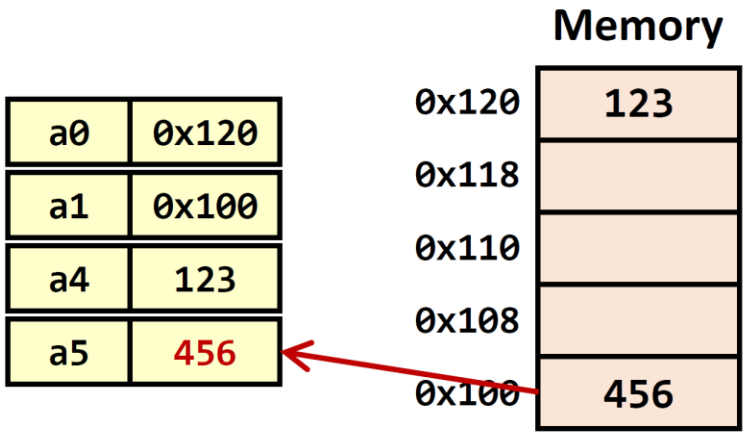
Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

Understanding Swap - 4

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



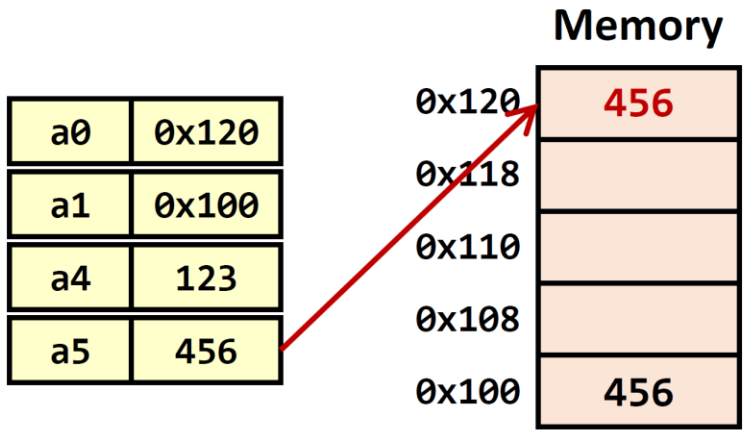
Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

Understanding Swap - 5

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



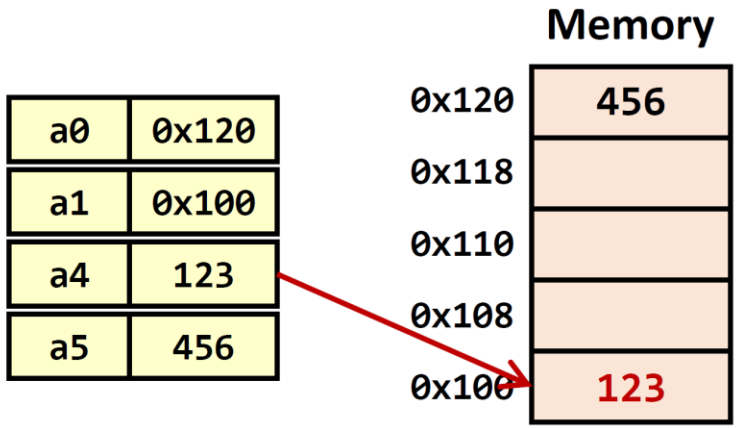
Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

Understanding Swap - 6

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation (By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ size_t i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

String Copy Example

- RISC-V code:

```
strcpy:
        addi sp,sp,-8          // adjust stack for 1 doubleword
        sd   x19,0(sp)         // push x19
        add  x19,x0,x0         // i=0
L1:     add  x5,x19,x11        // x5 = addr of y[i]
        lbu  x6,0(x5)          // x6 = y[i]
        add  x7,x19,x10        // x7 = addr of x[i]
        sb   x6,0(x7)          // x[i] = y[i]
        beq  x6,x0,L2          // if y[i] == 0 then exit
        addi x19,x19,  1       // i = i + 1
        jal  x0,L1             // next iteration of loop
L2:     ld   x19,0(sp)         // restore saved x19
        addi sp,sp,8           // pop 1 doubleword from stack
        jalr x0,0(x1)          // and return
```

Questions?