# Topic V0E

Strings of Characters

Readings: (Section 2.9)

# Instructions to Manipulate Characters (Bytes)

RISC-V byte load/store

String processing is a common case

`lb rd, offset(rs1)`

Sign-extend to 32 bits in rd

`lbu rd, offset(rs1)`

Zero-extend to 32 bits in rd

`sb rs2, offset(rs1)`

Store just the rightmost byte

```
lb   t0, 0(s0)
lb   t1, 1(s0)
lbu  t2, 0(s0)
lbu  t3, 1(s0)
```

| Register | Value |
|---|---|
| s0 | 0x10001010 |
| t0 | 0x00000009 |
| t1 | 0xFFFFFF80 |
| t2 | 0x00000009 |
| t3 | 0x00000080 |

| Address | Value |
|---|---|
| 0x10001015 | 0x00 |
| 0x10001014 | 0x0B |
| 0x10001013 | 0x00 |
| 0x10001012 | 0x00 |
| 0x10001011 | **0x80** |
| 0x10001010 | **0x09** |

# Instructions to Manipulate Half words

In RISC-V a halfword is 16 bits

`lh rd, offset(rs1)`
Sign-extend to 32 bits in rd

`lhu rd, offset(rs1)`
Zero-extend to 32 bits in rd

`sh rs2, offset(rs1)`
Store just the rightmost halfword

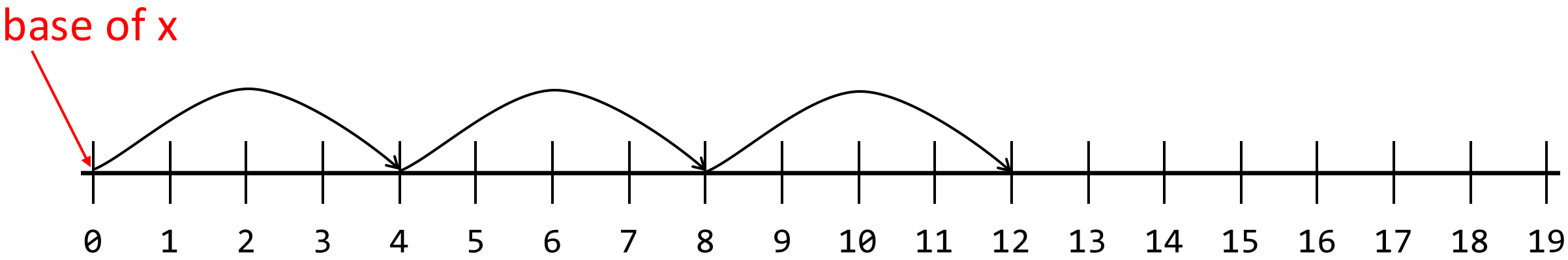
# How do I Know by How Much to Multiply the Index of an Array?

Look at the size of each element of the array

# Array of Integers

```
int x[100];
```

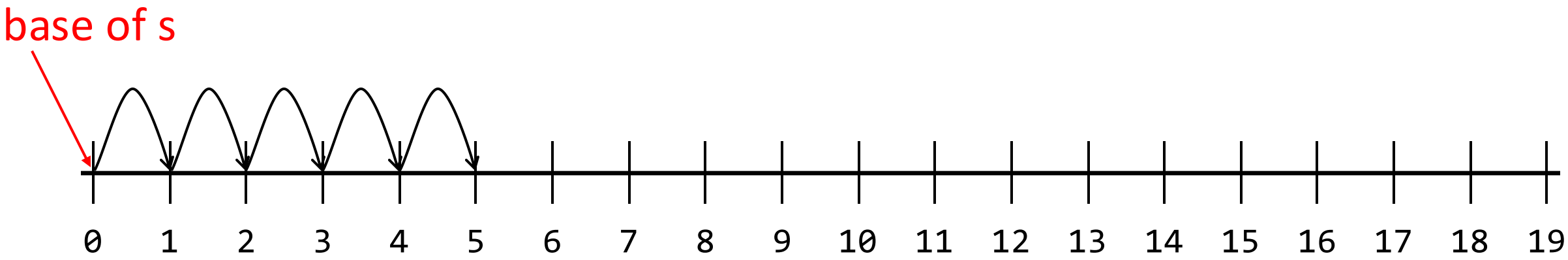
```
address(x[i]) = x + 4 * i;
```



# Array of characters

```
char s[100];
```

```
address(s[j]) = s + j;
```

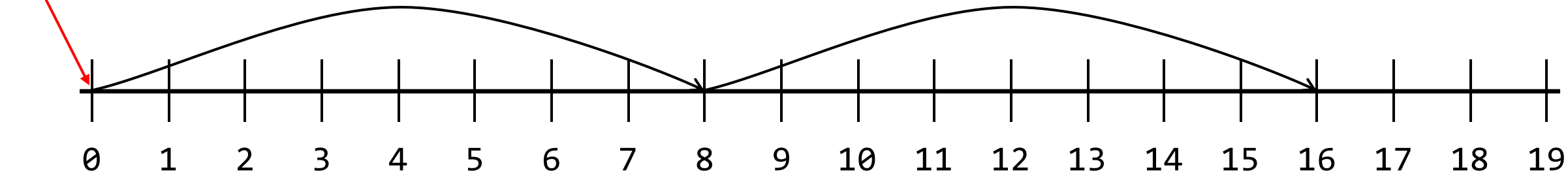


# Array of doubles

```
double f[100];
```

base of f

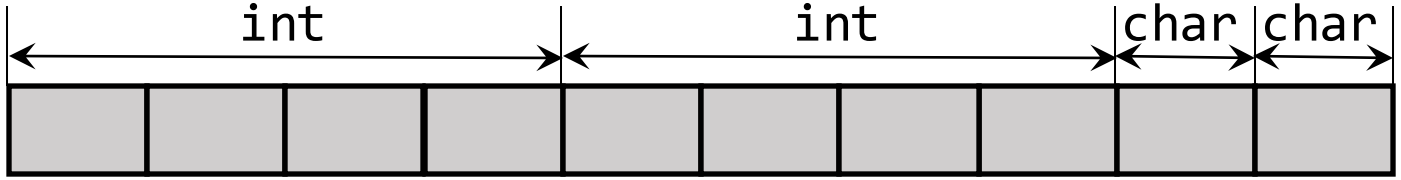
```
address(f[k]) = f + 8 * k;
```



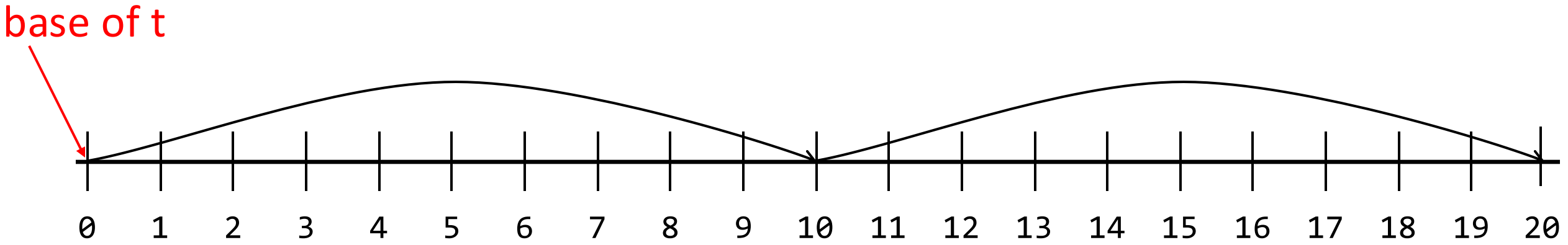
# Array of Structures

```
…
struct city{
      int DistanceFromCoast;
      int Population;
      char Subway;
      char RingRoad;
};
```

```
struct city t[150];
```




```
address(t[i]) = t + 10 * i;
```

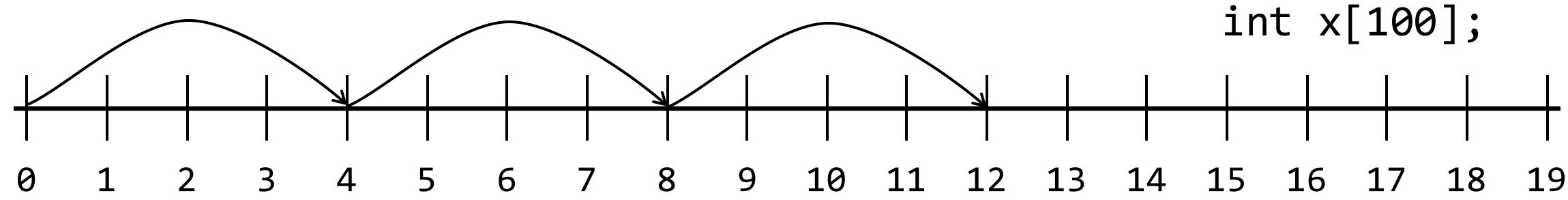


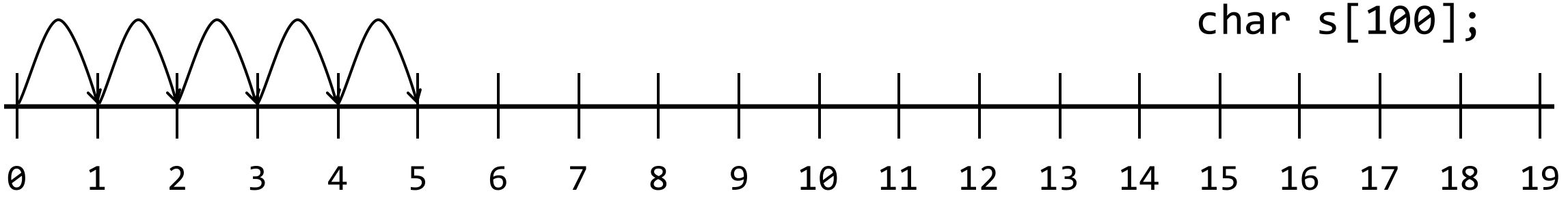
```
lb   t0, 0(s0)
lb   t1, 1(s0)
lbu  t2, 0(s0)
lbu  t3, 1(s0)
```

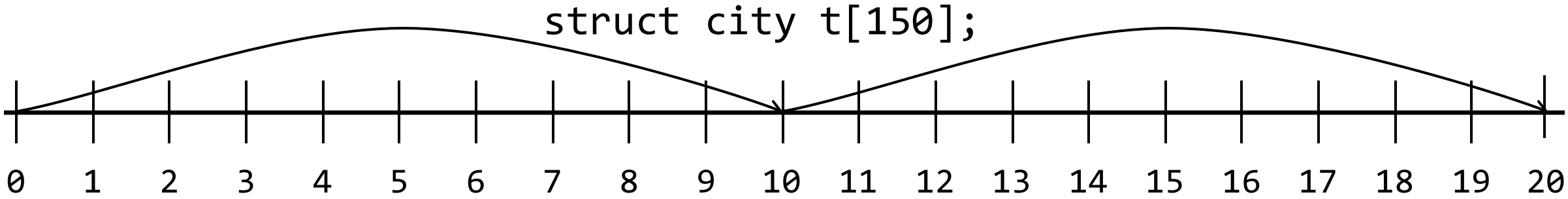
| Register | Value |
|---|---|
| s0 | 0x10001010 |
| t0 | 0x00000009 |
| t1 | 0xFFFFFF80 |
| t2 | 0x00000009 |
| t3 | 0x00000080 |

| Address | Value |
|---|---|
| 0x10001015 | 0x00 |
| 0x10001014 | 0x0B |
| 0x10001013 | 0x00 |
| 0x10001012 | 0x00 |
| 0x10001011 | **0x80** |
| 0x10001010 | **0x09** |

# Recap


int x[100];
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19


char s[100];
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19


struct city t[150];
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

# Recap

# String Copy

# String Copy (example)

C code (naïve):

Null-terminated string

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

Assumption

$i \leftrightarrow s0$

$x \leftrightarrow a0$

$y \leftrightarrow a1$

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

Assumption

i ↔ s0

x ↔ a0

y ↔ a1

```
strcpy:
       i ← 0;
L1:    t2 ← y[i];
       x[i] ← t2;
       if (y[i] == 0) goto L2;
       i ← i + 1;
       goto L1;
L2:    return;
```

```
strcpy:
       s0 ← 0;
L1:    t2 ← y[s0];
       x[s0] ← t2;
       if (t2 == 0) goto L2;
       s0 ← s0 + 1;
       goto L1;
L2:    return;
```

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

Assumption

$i \leftrightarrow s0$

$x \leftrightarrow a0$

$y \leftrightarrow a1$

```
strcpy:
       s0 ← 0;
L1:    t2 ← y[s0];
       x[s0] ← t2;
       if (t2 == 0) goto L2;
       s0 ← s0 + 1;
       goto L1;
L2:    return;
```

```
strcpy:
       s0 ← 0;
L1:    t1 ← a1 + s0;
       t2 ← M[t1];
       x[s0] ← t2;
       if (t2 == 0) goto L2;
       s0 ← s0 + 1;
       goto L1;
L2:    return;
```

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

Assumption

$i \leftrightarrow s0$

$x \leftrightarrow a0$

$y \leftrightarrow a1$

```
strcpy:
        s0 ← 0;
L1:     t1 ← a1 + s0;
        t2 ← M[t1];
        x[s0] ← t2;
        if (t2 == 0) goto L2;
        s0 ← s0 + 1;
        goto L1;
L2:     return;
```

```
strcpy:
        s0 ← 0;
L1:     t1 ← a1 + s0;
        t2 ← M[t1];
        t3 ← a0 + s0;
        M[t3] ← t2;
        if (t2 == 0) goto L2;
        s0 ← s0 + 1;
        goto L1;
L2:     return;
```

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

Assumption

i ↔ s0

x ↔ a0

y ↔ a1

```
strcpy:
      <save registers>
      s0 ← 0;
L1:   t1 ← a1 + s0;
      t2 ← M[t1];
      t3 ← a0 + s0;
      M[t3] ← t2;
      if (t2 == 0) goto L2;
      s0 ← s0 + 1;
      goto L1;
L2:   <restore registers>
      return;
```

```
strcpy:
      sp ← sp – 4;
      M[sp] ← s0;
      s0 ← 0;
L1:   t1 ← a1 + s0;
      t2 ← M[t1];
      t3 ← a0 + s0;
      M[t3] ← t2;
      if (t2 == 0) goto L2;
      s0 ← s0 + 1;
      goto L1;
L2:   s0 ← M[sp];
      sp ← sp + 4;
      return;
```

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

## Assumption

i ↔ s0

x ↔ a0

y ↔ a1

```
strcpy:
        sp ← sp - 4;
        M[sp] ← s0;
        s0 ← 0;
L1:     t1 ← a1 + s0;
        t2 ← M[t1];
        t3 ← a0 + s0;
        M[t3] ← t2;
        if (t2 == 0) goto L2;
        s0 ← s0 + 1;
        goto L1;
L2:     s0 ← M[sp];
        sp ← sp + 4;
        return;
```

```
strcpy:
        addi   sp, sp, -4    # adjust stack for 1 item
        sw     s0, 0(sp)     # save s0
        add    s0, zero, zero      # i ← 0
L1:     add    t1, a1, s0    # addr of y[i] in t1
        lbu    t2, 0(t1)     # t2 ← y[i]
        add    t3, a0, s0    # addr of x[i] in t3
        sb     t2, 0(t3)     # x[i] ← y[i]
        beq    t2, zero, L2 # exit loop if y[i] == '\0'
        addi   s0, s0, 1     # i ← i + 1
        jal    zero, L1      # next iteration of loop
L2:     lw     s0, 0(sp)     # restore saved s0
        addi   sp, sp, 4     # pop 1 item from stack
        jalr   zero, ra, 0   # and return
```

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

## Assumption

i ↔ s0

x ↔ a0

y ↔ a1

```
strcpy:
        sp ← sp – 4;
        M[sp] ← s0;
        s0 ← 0;
L1:     t1 ← a1 + s0;
        t2 ← M[t1];
        t3 ← s0 + a0;
        M[t3] ← t2;
        if (t2 == 0) goto L2;
        s0 ← s0 + 1;
        goto L1;
L2:     s0 ← M[sp];
        sp ← sp + 4;
        return;
```

```
strcpy:
        addi   sp, sp, -4    # adjust stack for 1 item
        sw     s0, 0(sp)     # save s0
        add    s0, zero, zero      # i ← 0
L1:     add    t1, a1, s0    # addr of y[i] in t1
        lbu    t2, 0(t1)     # t2 ← y[i]
        add    t3, a0, s0    # addr of x[i] in t3
        sb     t2, 0(t3)     # x[i] ← y[i]
        beq    t2, zero, L2  # exit loop if y[i] == '\0'
        addi   s0, s0, 1     # i ← i + 1
        jal    zero, L1      # next iteration of loop
L2:     lw     s0, 0(sp)     # restore saved s0
        addi   sp, sp, 4     # pop 1 item from stack
        jalr   zero, ra, 0   # and return
```

# Better versions of the same string copy

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

An assignment inside a conditional expression may be missed when reading the code

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

```
void strcpy(char x[], char y[]){
        int i;
        x[0] = y[0];
        for(i=0; y[i-1] != 0 ; i++){
     x[i] = y[i];
    }
}
```

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

```
strcpy:
      sp ← sp – 4;
      M[sp] ← s0;
      s0 ← 0;
L1:   t1 ← s0 + a1;
      t2 ← M[t1];
      t3 ← s0 + a0;
      M[t3] ← t2;
      if (t2 == 0) goto L2;
      s0 ← s0 + 1;
      goto L1;
L2:   s0 ← M[sp];
      sp ← sp + 4;
      return;
```

```
strcpy:
      addi   sp, sp, -4   # adjust stack for 1 item
      sw     s0, 0(sp)    # save s0
      add    s0, zero, zero     # i ← 0
L1:   add    t1, s0, a1   # addr of y[i] in t1
      lbu    t2, 0(t1)    # t2 ← y[i]
      add    t3, s0, a0   # addr of x[i] in t3
      sb     t2, 0(t3)    # x[i] ← y[i]
      beq    t2, zero, L2 # exit loop if y[i] == '\0'
      addi   s0, s0, 1    # i ← i + 1
      jal    zero, L1     # next iteration of loop
L2:   lw     s0, 0(sp)    # restore saved s0
      addi   sp, sp, 4    # pop 1 item from stack
      jalr   zero, ra, 0  # and return
```

# Recap

jalr x0, ra, 0

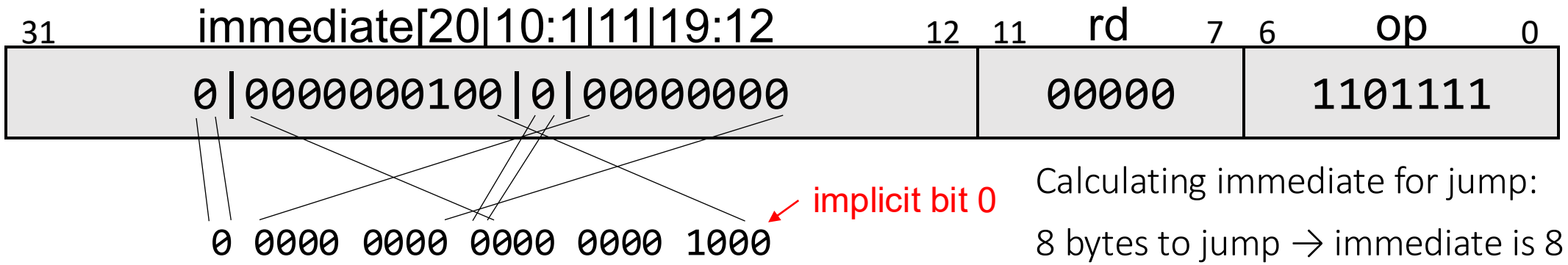
Let us recap how we use jal for an unconditional jump.

# Jump Instructions: UJ-Type

Memory Address

```
0x1000 0000          bne    s3, s4, Subtr      # if i ≠ j goto Subtr
0x1000 0004          add    s0, s1, s2         # f ← g + h
0x1000 0008          jal    zero, Exit         # goto Exit
0x1000 000C Subtr:   sub    s0, s1, s2         # f ← g – h
0x1000 0010 Exit:           …
```

jal zero, Exit ↔ R[rd] = PC + 4 ; PC = PC + {imm, 1b'0}




Calculating immediate for jump:

8 bytes to jump → immediate is 8

What is 8 expressed in 21 bit signed binary?

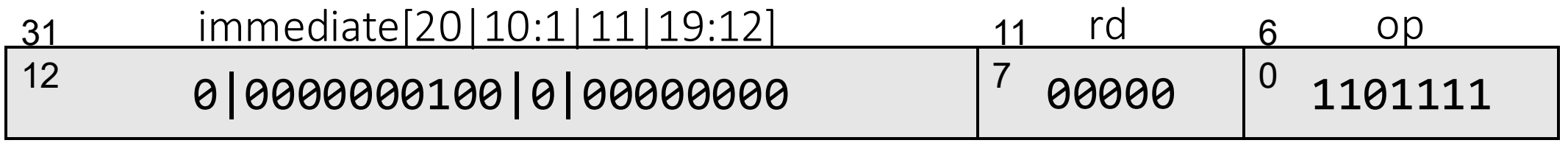
```
0000 0000 0000 0000 0000 0000 0000 1000   ← sign-extended (written as hex)
+                     0x1000 0008         ← PC
-------------------------------------
                      0x1000 0010         ← New PC
```

# Jump Instructions: UJ-Type

Memory Address

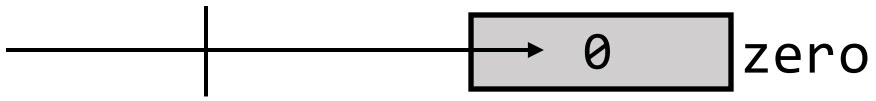
```
0x1000 0000          bne    s3, s4, Subtr      # if i ≠ j goto Subtr
0x1000 0004          add    s0, s1, s2         # f ← g + h
0x1000 0008          jal    zero, Exit         # goto Exit
0x1000 000C Subtr:   sub    s0, s1, s2         # f ← g – h
0x1000 0010 Exit:           …
```

jal zero, Exit ↔ R[rd] = PC + 4 ; PC = PC + {imm, 1b'0}




Saving return address:

```
PC      →    0x1000 0008
4       →  + 0x0000 0004
PC + 4  ←    0x1000 000C
```

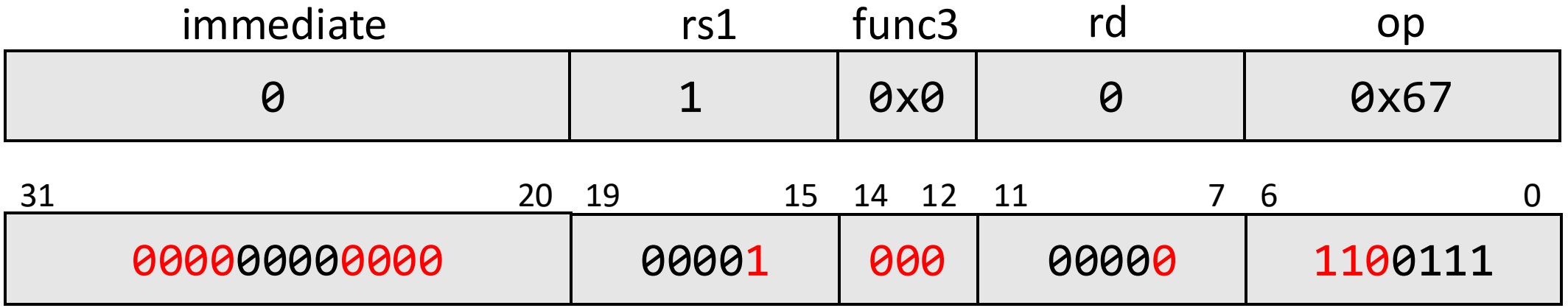



Register zero cannot be overwritten so PC + 4 is discarded

# Jump And Link Register

x1

```
jalr x0, ra, 0   # return
```

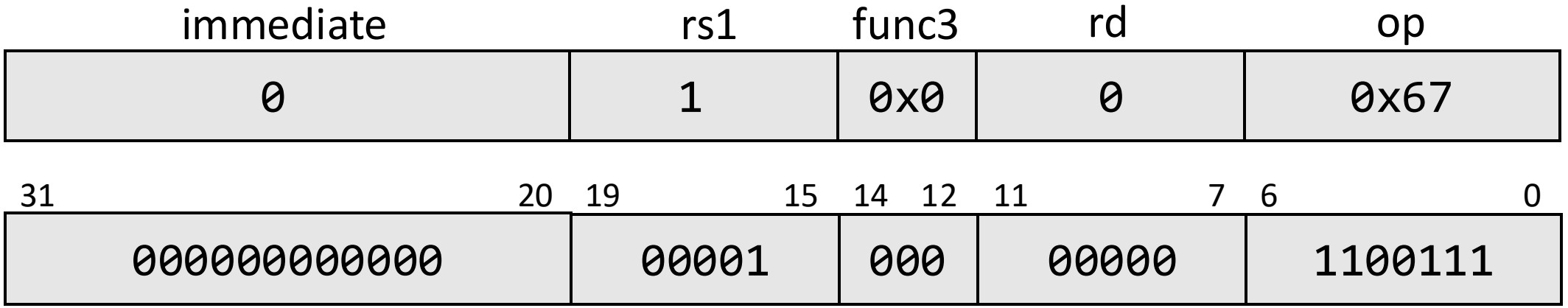



Hexadecimal Representation: 0x00008067

# Jump And Link Register

x1

```
jalr x0, ra, 0   # return
```




x0 ← PC+4

PC ← ra + Immediate # bit zero of immediate is zeroed

```
jalr x0, ra, 1  # PC ← ra
…
jalr x0, ra, 2   # alignment error
…
jalr x0, ra, 5   # PC ← ra+4
```

```
foo:   add   t0, x0, x0     # t0 <- 0
       auipc t1, 0          # t1 <- PC this inst.
       bne   t0, x0, after  # has been here?
       addi  t0, x0, 1      # t0 <- 1
       jalr  x0, t1, 1      # jump to auipc inst.
after: jalr  t2, t1, 21     # jump to LA instr.
LA:    jalr  zero, t1, 2    # Exc: Alignment error
       jalr  zero, ra, 0
```

rd ← PC+4

PC ← rs + immed

Bit zero of immediate is zeroed

jalr rd, rs, immed

General unconditional jump relative to the address in register **rs**.

Used for dynamic dispatching: a function call via a function pointer.

Used for indirect jump: p. e. code for a switch-case statement.

Recap

A return statement:

jalr x0, ra, 0