



NATIONAL RESEARCH
UNIVERSITY



Computer Architecture and Operating Systems

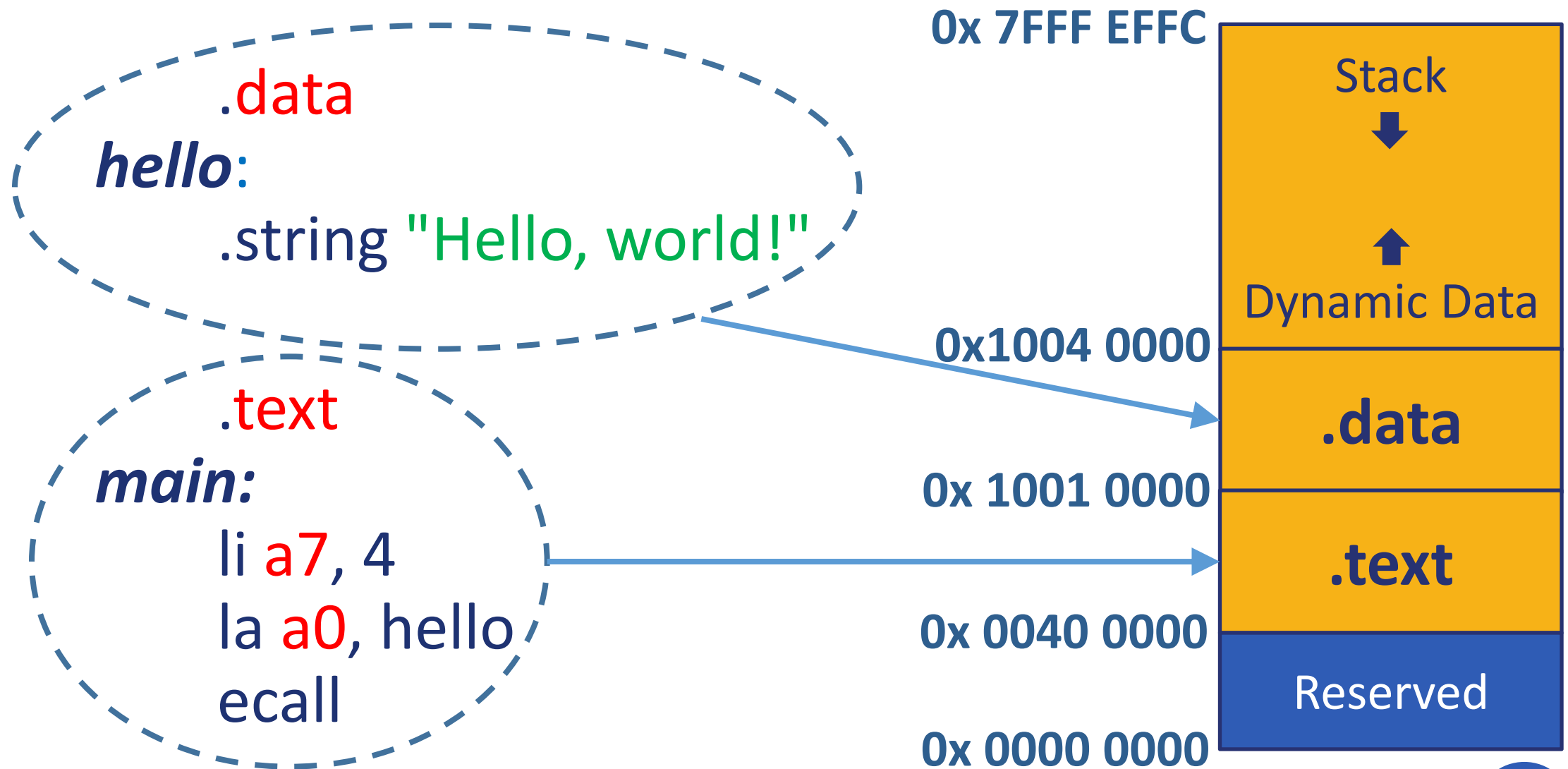
Lecture 5: Assembly Programming – Branches and Memory

Andrei Tatarnikov

atatarnikov@hse.ru

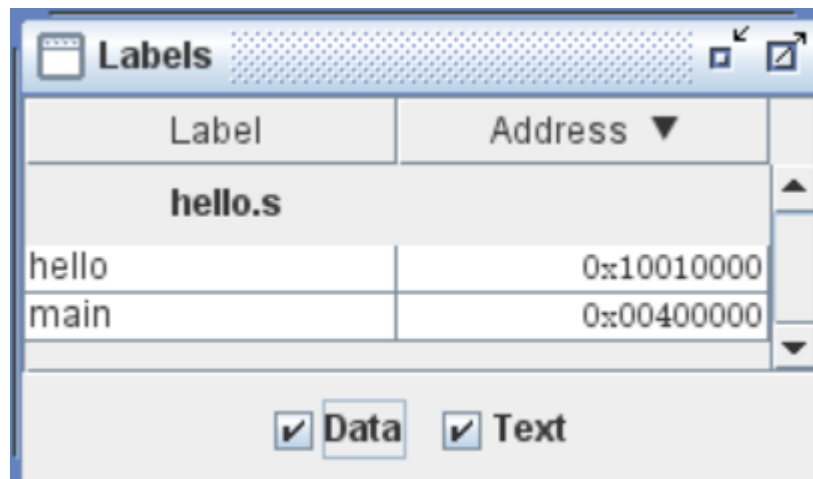
[@andrewt0301](#)

Program Structure and Memory Layout



Labels

- **Labels** are symbolic names for addresses (in the .data or .text segment).
- **Labels** are used by control-flow instructions (branches and jumps).
- **Labels** are used by load and store instructions.



The screenshot shows a window titled 'Labels' with a table of labels. The table has two columns: 'Label' and 'Address'. The file 'hello.s' is selected. The table lists two labels: 'hello' at address '0x10010000' and 'main' at address '0x00400000'. At the bottom, there are checkboxes for 'Data' and 'Text', both of which are checked.

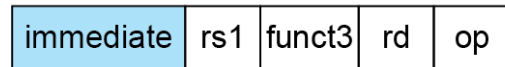
Label	Address ▼
hello.s	
hello	0x10010000
main	0x00400000

☒ Data ☒ Text

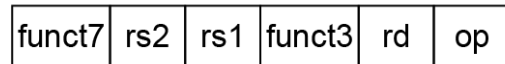
Addressing

Addresses can be represented in several ways

1. Immediate addressing



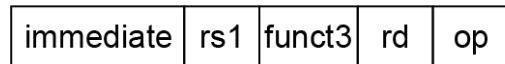
2. Register addressing



Registers

Register

3. Base addressing



Memory

Register

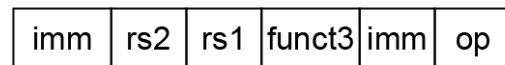
+

Byte Halfword

Word

Doubleword

4. PC-relative addressing



Memory

PC

+

Word

Program Counter

- **Program Counter (PC)** is a special register that stores the address of the currently executed instruction.
- When an instruction is executed, the **PC** is incremented by the size of the instruction (4 bytes) to point to the next instruction.
- Branch and jump instructions assign to the **PC** new addresses to change the control flow.
- Branch instructions use **PC**-relative addresses (increment or decrement current value by an offset).

Branch Instructions

Branch Instructions

- Branch = `beq rs1, rs2, label`
- Branch \neq `bne rs1, rs2, label`
- Branch $<$ `blt rs1, rs2, label`
- Branch \geq `bge rs1, rs2, label`
- Branch $<$ Unsigned `bltu rs1, rs2, label`
- Branch \geq Unsigned `bgeu rs1, rs2, label`

Branch Pseudo Instructions

Branch Pseudo Instructions

▪ Branch unconditionally	<code>b</code>	<code>label</code>
▪ Branch = 0	<code>beqz</code>	<code>rs1, label</code>
▪ Branch ≥ 0	<code>bgez</code>	<code>rs1, label</code>
▪ Branch >	<code>bgt</code>	<code>rs1, rs2, label</code>
▪ Branch > Unsigned	<code>bgtu</code>	<code>rs1, rs2, label</code>
▪ Branch > 0	<code>bgtz</code>	<code>rs1, label</code>
▪ Branch \leq	<code>ble</code>	<code>rs1, rs2, label</code>
▪ Branch \leq Unsigned	<code>bleu</code>	<code>rs1, rs2, label</code>
▪ Branch ≤ 0	<code>blez</code>	<code>rs1, label</code>
▪ Branch < 0	<code>bltz</code>	<code>rs1, label</code>
▪ Branch $\neq 0$	<code>bnez</code>	<code>rs1, label</code>

Branches and Program Counter

- Branch instructions are **PC**-relative
- They add a **12-bit** signed immediate to **PC**
- The immediate is an offset from **PC** to the target label
- The branch address range is $\pm 2^{12}$ (4096 B = 4 KB)
- **PC** can be read with the **auipc** instruction

main:

```
auipc a0, 0    # a0 = PC + 0  
li     a7, 34  # Print as hex  
ecall                # Print a0
```


Assembly Code for “If-Then-Else”

```
if (t0 == 0) {  
    t1 = 1;  
} else if (t0 < 0) {  
    t1 = 2;  
} else if (t0 >= 10) {  
    t1 = 3;  
} else {  
    t1 = 4;  
}
```

if_0:
bnez t0, if_less_0
li t1, 1
b end_if

if_less_0:
bgez t0, if_greater_10
li t1, 2
b end_if

if_greater_10:
li t3, 10
blt t0, t3, else
li t1, 3
b end_if

else:
li t1, 4
end_if:

Assembly Code for “While”

```
while((t0 = read_int()) != 0) {  
    print_int(t0)  
    print_char('\n')  
}
```



```
while:  
    li    a7, 5  
    ecall  
    mv    t0, a0  
    beqz  a0, end_while  
    li    a7, 1  
    ecall  
    li    a7, 11  
    li    a0, '\n'  
    ecall  
    b     while  
end_while:
```

Assembly Code for “For”

```
for (t0 = 0; t0 < t1; ++t0) {  
    print_int(t0)  
    print_char('\n')  
}
```



```
for:  
    li    a7, 5  
    ecall  
    mv    t1, a0  
    mv    t0, zero  
next:  
    beq    t0, t1, end_for  
    mv    a0, t0  
    li    a7, 1  
    ecall  
    li    a7, 11  
    li    a0, '\n'  
    ecall  
    addi   t0, t0, 1  
    b      next  
end_for:
```

Assembly Code for Nested “For”

```
for (t0 = 0; t0 < s0; ++t0) {  
    for (t1 = 0; t0 < s1; ++t1) {  
        print_int(t0)  
        print_char(':', '  
        print_int(t1)  
        print_int(' '  
    }  
    print_char('\n')  
}  
}
```

Assembly code for the nested for loop:

```
mv t0, zero  
next_t0:  
beq t0, s0, end_for_t0  
mv t1, zero  
next_t1:  
beq t1, s1, end_for_t1  
print_int(t0)  
print_char(':', '  
print_int(t1)  
print_char(' '  
addi t1, t1, 1  
b next_t1  
end_for_t1:  
print_char('\n')  
addi t0, t0, 1  
b next_t0  
end_for_t0:
```

Diagram illustrating the mapping between the C code and the assembly code:

- The outer for loop header `for (t0 = 0; t0 < s0; ++t0) {` maps to the assembly code `mv t0, zero` and `next_t0:`.
- The inner for loop header `for (t1 = 0; t0 < s1; ++t1) {` maps to the assembly code `beq t0, s0, end_for_t0` and `mv t1, zero`.
- The inner loop body `print_int(t0)`, `print_char(':', 'print_int(t1), and print_int(' ' maps to the assembly code print_int(t0), print_char(':', 'print_int(t1), and print_char(' '.`
- The inner loop increment `++t1` maps to the assembly code `addi t1, t1, 1`.
- The inner loop branch `}` maps to the assembly code `b next_t1`.
- The outer loop body `print_char('\n')` maps to the assembly code `print_char('\n')`.
- The outer loop increment `++t0` maps to the assembly code `addi t0, t0, 1`.
- The outer loop branch `}` maps to the assembly code `b next_t0`.
- The end of the outer loop `}` maps to the assembly code `end_for_t0:`.

Macros

Macro is a pattern-matching and replacement facility that provides a simple mechanism to name a frequently used sequence of instructions.

```
.macro print_int (%x)
li    a7, 1
mv    a0, %x
ecall
.end_macro
```

```
.macro read_int (%x)
li    a7, 5
ecall
mv    %x, a0
.end_macro
```

Use Macros to
Simplify Your Code



main:

```
read_int(t0)
print_int(t0)
```

Including Macro Libraries

It is possible to place macros in a **library** file and **include** it in other assembly programs.

```
.include "macrolib.s"
```

```
main:
```

```
read_int(t0)
```

```
print_int(t0)
```

The *read_int* and *print_int* macros are defined in the *macrolib.s* file.

The file must be in the same directory as the program.

Macro Constants and Single-Line Macros

The **.eqv** directive can be used to define macro constants and single-line macros.

```
.eqv VAL 0x123
```

```
.eqv X t0
```

```
.eqv Y t1
```

```
.eqv SUM addi Y, X, VAL
```

```
main:
```

```
li    X, 0x111
```

```
SUM
```

Data Segment

Segment **.data** stores static data (global variables and constants), which are described with the following directives:

.data

.word 0xDEADBEEF # 32-bit value

.half 0x1234, 0x4567 # 16-bit values

.byte 0x98, 0x76, 0x65, 0x43 # 8-bit values

.space 8 # 8 bytes of empty space

.ascii "Hello " # String

.asciz "World! " # Zero-terminated string

Data Alignment

Data items are aligned in memory by their size for convenience of access. This means ***address is multiple of size***. Default alignment is as follows:

- **.byte** # 1 byte
- **.half** # 2 bytes
- **.word** # 4 bytes

It is possible to specify a ***custom alignment by 2^n bytes*** for a next data item with the .align directive.

- **.align 0** # 1 byte
- **.align 1** # 2 bytes
- **.align 2** # 4 bytes
- **.align 3** # 8 bytes
- etc.

Load and Store Instructions

Load Instructions

lb **t1**, offset(**t2**) # t1 <- sign-extended 8-bit value from address t2 + offset
lbu **t1**, offset(**t2**) # t1 <- zero-extended 8-bit value from address t2 + offset
lh **t1**, offset(**t2**) # t1 <- sign-extended 16-bit value from address t2 + offset
lhu **t1**, offset(**t2**) # t1 <- zero-extended 16-bit value from address t2 + offset
lw **t1**, offset(**t2**) # t1 <- contents of address t2 + offset

Store Instructions

sb **t1**, offset(**t2**) # Store low-order 8 bits (byte) of t1 to address t2 + offset
sh **t1**, offset(**t2**) # Store low-order 16 bits (half) of t1 to address t2 + offset
sw **t1**, offset(**t2**) # Store contents of t1 to address t2 + offset

Load Address Pseudo Instruction

la **t2**, label # t1 <- address of label

Load and Store Example

x, y, and z are static variables

int x, y, z;

x = read_int();

y = read_int();

z = x + y;

.data
x:
 .word 0
y:
 .word 0
z:
 .word 0
 .text
main:
 read_int(t0)
 la t2, x
 sw t0, 0(t2)

 read_int(t0)
 la t2, y
 sw t0, 0(t2)

 la t2, x
 lw t0, 0(t2)
 la t2, y
 lw t1, 0(t2)
 add t3, t0, t1
 la t2, z
 sw t3, 0(t2)

Load and Store With Offset Example

data[3] is a static array that stores three integer variables

int data[3];

x = read_int();

y = read_int();

z = x + y;

.data
data:
 .word 0, 0, 0
 .text
main:
 la t2, data

 read_int(t0)
 sw t0, 0(t2)

 read_int(t0)
 sw t0, 4(t2)

 lw t0, 0(t2)
 lw t1, 4(t2)
 add t3, t0, t1
 sw t3, 8(t2)

Load and Store Pseudoinstruction Example

x, y, and z are static variables

int x, y, z;

x = read_int();

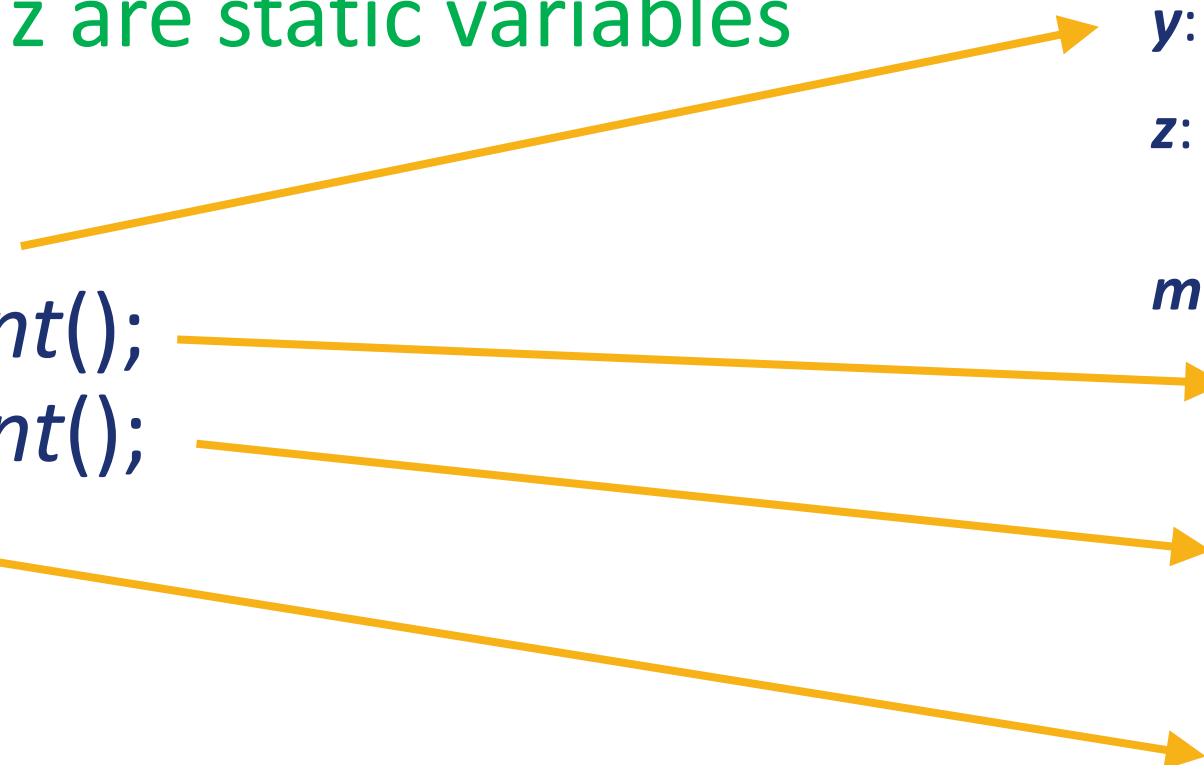
y = read_int();

z = x + y;

```
.data
x:
.word 0
y:
.word 0
z:
.word 0
.text
main:
read_int(t0)
sw t0, x, t2

read_int(t0)
sw t0, y, t2

lw t0, x
lw t1, y
add t3, t0, t1
sw t3, z, t2
```



Load and Store Pseudo Instructions

Load Pseudo Instructions

<code>lw t1, (t2)</code>	<code># t1 <- contents of memory at address t2</code>
<code>lw t1, imm</code>	<code># t1 <- contents of memory address in imm</code>
<code>lw t1, label</code>	<code># t1 <- contents of memory at label's address</code>

Store Pseudo Instructions

<code>sw t1, (t2)</code>	<code># Store t1 to address t2</code>
<code>sw t1, imm</code>	<code># Store t1 to address in imm</code>
<code>sw t1, imm, t2</code>	<code># Store t1 in to address in imm using t2 as temp</code>
<code>sw t1, label, t2</code>	<code># Store t1 to label's address using t2 as temp</code>

For instructions `lb`, `lbu`, `lh`, `lhu`, `sb`, and `sh` similar pseudo instructions are provided.

Any Questions?

```
                .text
__start:      addi t1, zero, 0x18
                addi t2, zero, 0x21
cycle:        beq t1, t2, done
                slt t0, t1, t2
                bne t0, zero, if_less
                nop
                sub t1, t1, t2
                j cycle
                nop
if_less:      sub t2, t2, t1
                j cycle
done:         add t3, t1, zero
```