Computer Architecture and Operating Systems
Lecture 5: Assembly Programming – Branches and Memory

Andrei Tatarnikov

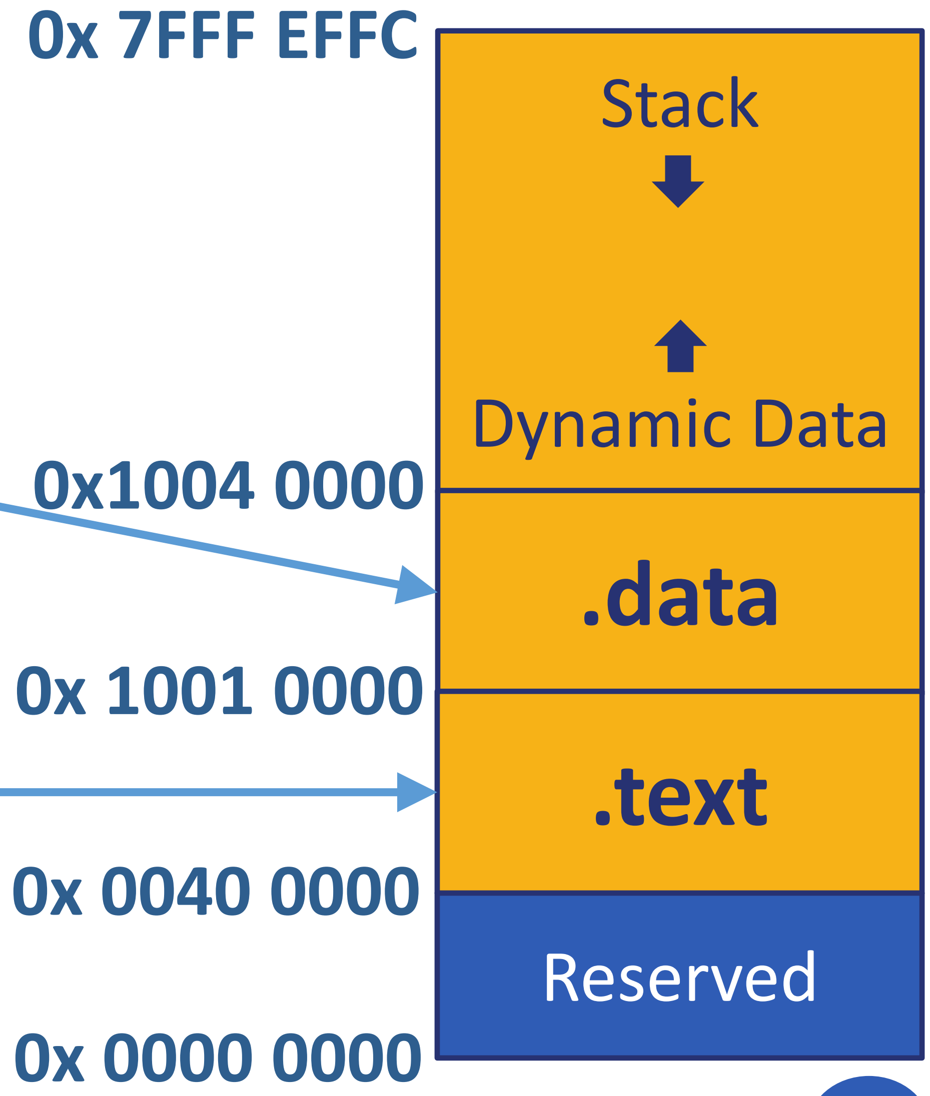
atatarnikov@hse.ru

@andrewt0301

Program Structure and Memory Layout

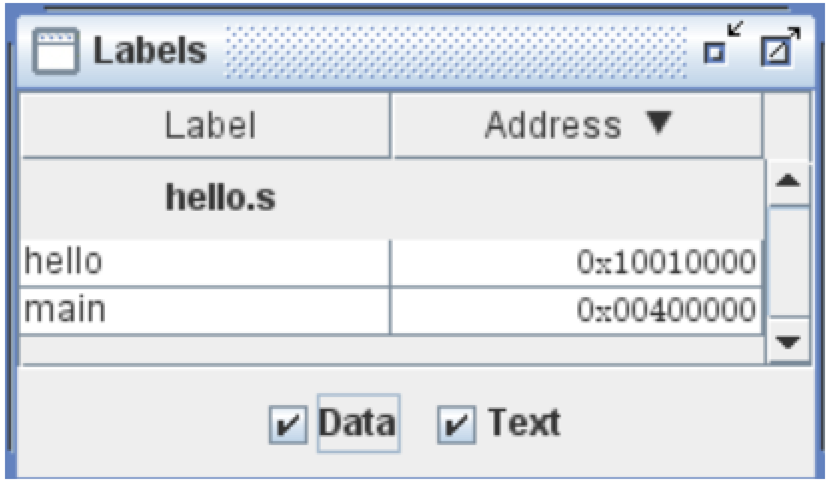
```
    .data
hello:
    .string "Hello, world!"

    .text
main:
    li a7, 4
    la a0, hello
    ecall
```



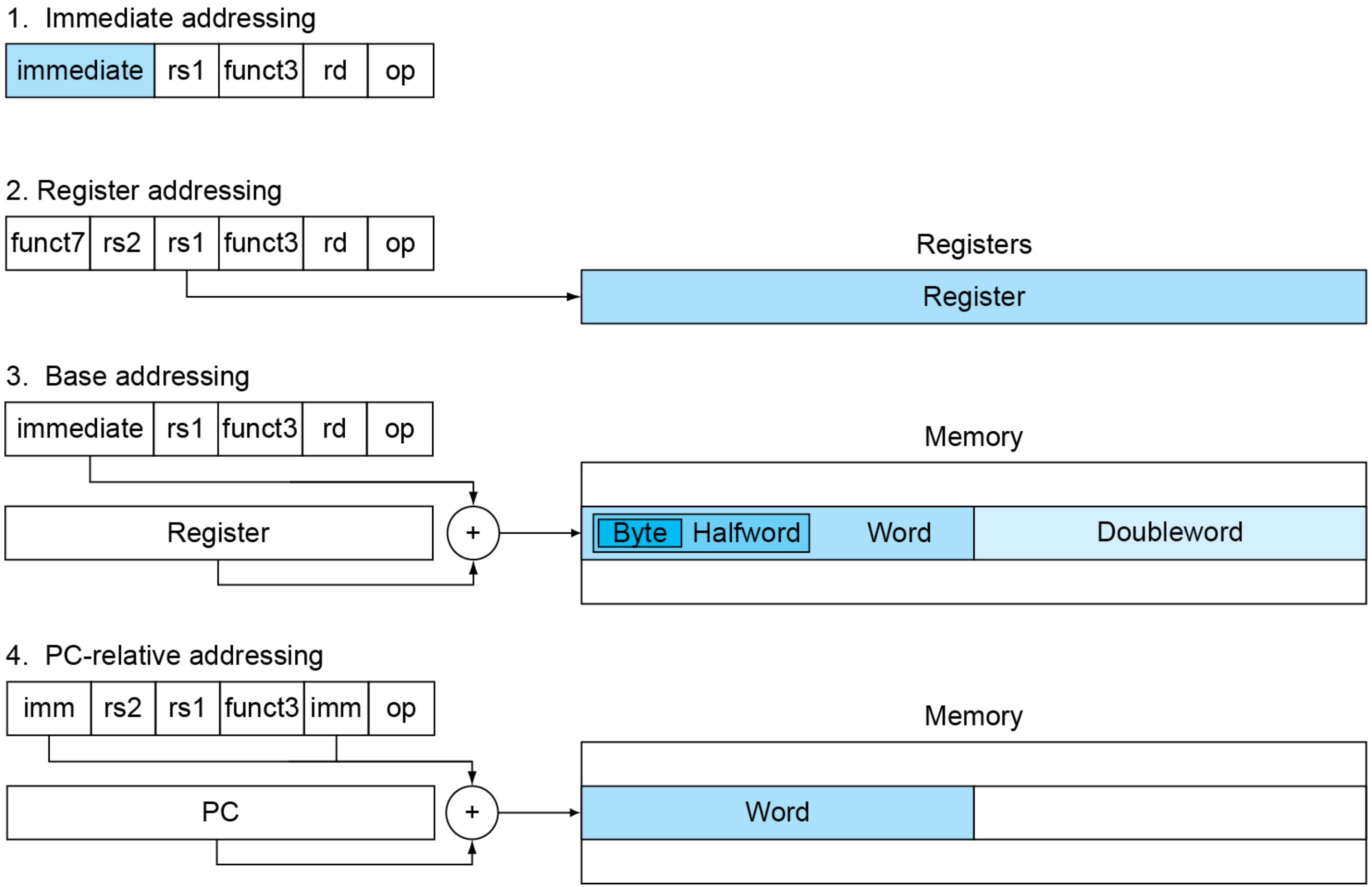
Labels

- **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.



Addressing

Addresses can be represented in several ways



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 ≠	bne	rs1, rs2, *label*
▪ Branch <	blt	rs1, rs2, *label*
▪ Branch ≥	bge	rs1, rs2, *label*
▪ Branch < Unsigned	bltu	rs1, rs2, *label*
▪ Branch ≥ Unsigned	bgeu	rs1, rs2, *label*

Branch Pseudo Instructions

Branch Pseudo Instructions

▪ Branch unconditionally	j	*label*
▪ Branch = 0	beqz	rs1, *label*
▪ Branch ≥ 0	bgez	rs1, *label*
▪ Branch >	bgt	rs1, rs2, *label*
▪ Branch > Unsigned	bgtu	rs1, rs2, *label*
▪ Branch > 0	bgtz	rs1, *label*
▪ Branch ≤	ble	rs1, rs2, *label*
▪ Branch ≤ Unsigned	bleu	rs1, rs2, *label*
▪ Branch ≤ 0	blez	rs1, *label*
▪ Branch < 0	bltz	rs1, *label*
▪ Branch ≠ 0	bnez	rs1, *label*

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
    j     end_if
if_less_0:
    bgez  t0, if_greater_10
    li    t1, 2
    j     end_if
if_greater_10:
    li    t3, 10
    blt   t0, t3, else
    li    t1, 3
    j     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
    j       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
    j       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')
}
```

```
    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
    j       next_t1
end_for_t1:
    print_char('\n')
    addi  t0, t0, 1
    j       next_t0
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.

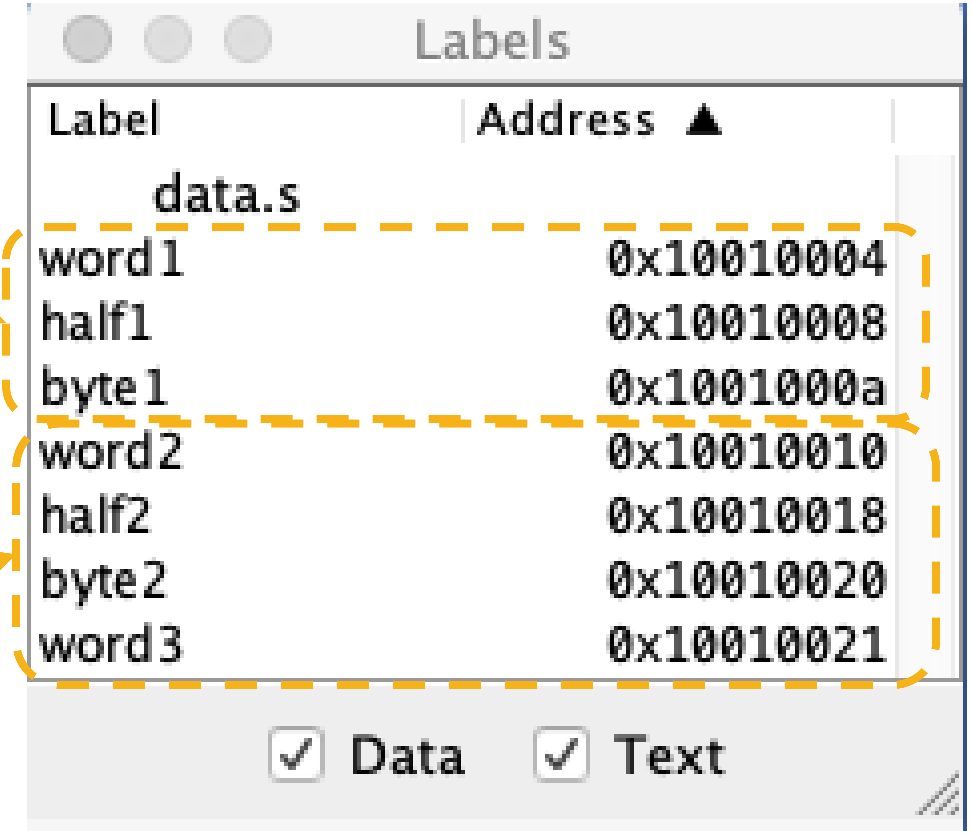
- .align 0 # 1 byte
- .align 1 # 2 bytes
- .align 2 # 4 bytes
- .align 3 # 8 bytes
- etc.

Data Alignment Example

```
        .data
        .space  3
word1:
        .word   0x12345678
half1:
        .half   0x1234
byte1:
        .byte   0x12
        .align  4
word2:
        .word   0x12345678
        .align  3
half2:
        .half   0x1234
        .align  3
byte2:
        .byte   0x12
        .align  0
word3:
        .word   0x12345678
```

Default Alignment

Custom Alignment



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, y, z
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

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

Store Pseudo Instructions

```
sw t1,(t2)         # Store t1 to address t2
sw t1, imm         # Store t1 to address in imm
sw t1, imm, t2     # Store t1 in to address in imm using t2 as temp
sw t1, label, t2   # Store t1 to label's address using t2 as temp
```

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
```