Computer Architecture and Operating Systems
Lecture 6: Assembly Programming – Stack

Andrei Tatarnikov

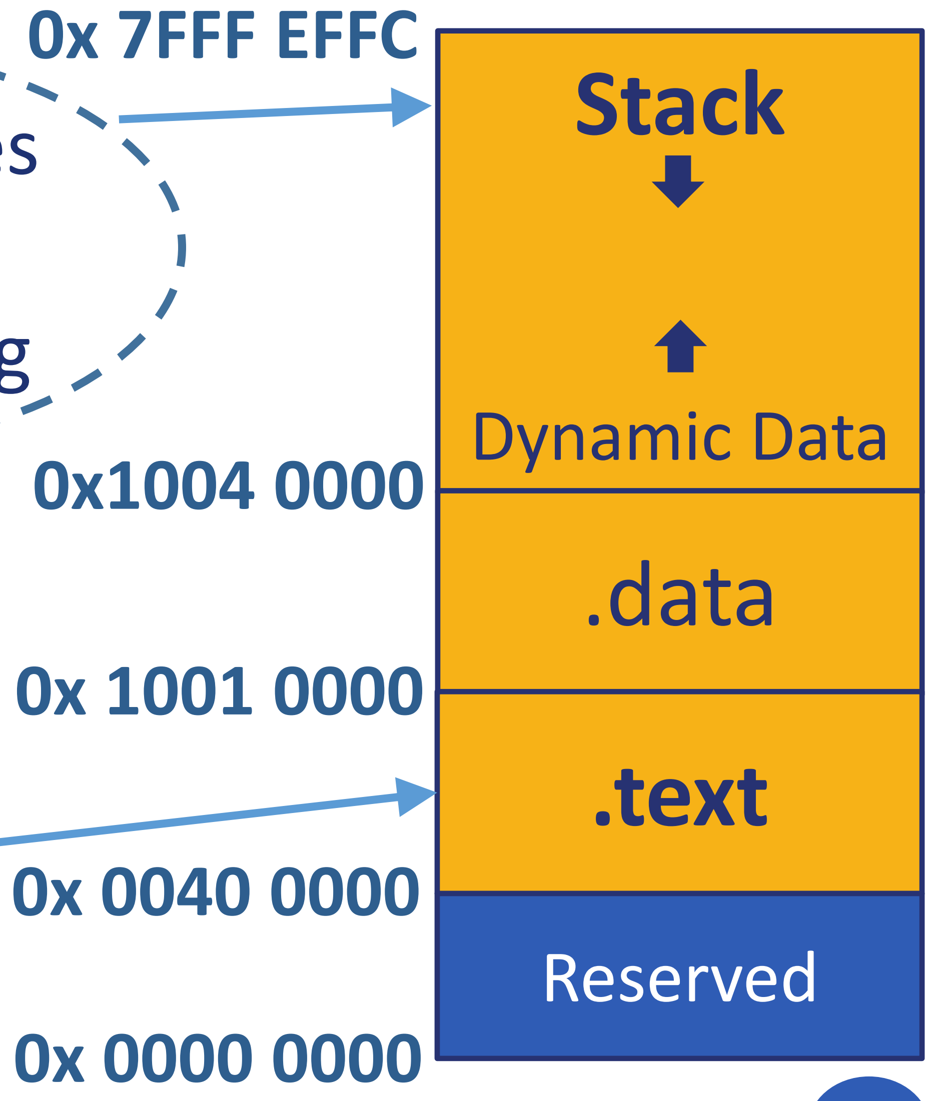
atatarnikov@hse.ru

@andrewt0301

Program Structure and Memory Layout

```
.text
main:
    li a7, 5
    ecall
    # function call
    func(a0)
    li a7, 10
    ecall
    # function
func:
    # do something
    return a0
```

Local variables
Temp values
Housekeeping



Notion of Function

- **Function** (procedure) is a code that performs some task based on the arguments with which it is provided
- **Caller** is a code that calls a function and provides it with the necessary arguments
- **Callee** is a function that executes instructions based on arguments provided by the caller and then returns control to the caller
- **Return address** is a link that allows the callee to return control to the caller
- **Jump-and-link instruction** is an instruction that branches to an address and simultaneously saves the address of the next instruction in to a register

Function Call Steps

- Place arguments in registers a0 (x10) to a7 (x17)
- Save return address in ra (x1) and jump to function
- Allocate stack memory for the function
- Perform function's operations
- Free stack memory allocated for the function
- Place result in register a0 for caller
- Return to place of call (address in ra)

RISC-V Register Conventions

Register	Name	Use	Saver
x0	zero	constant 0	n/a
x1	ra	return address	caller
x2	sp	stack pointer	callee
x3	gp	global pointer	
x4	tp	thread pointer	
x5-x7	t0-t2	temporaries	caller
x8	s0/fp	saved/ frame pointer	callee
x9	s1	saved	callee
x10-x17	a0-a7	arguments	caller
x18-x27	s2-s11	saved	callee
x28-x31	t3-t6	temporaries	caller

Jump-and-Link Instructions

- Function call: jump and link

  ```
  jal ra, FunctionLabel (UJ-type)
  ```

  - Address of the next instruction is put in **ra** (x1)
  - Jumps to target address
- Function return: jump and link register

  ```
  jalr zero, 0(ra)   (I-type)
  ```

  - Like **jal**, but jumps to 0 + address in **ra** (x1)
  - Use **zero** (x0) as rd (**zero** cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Jump-and-Link Pseudo Instructions

j *label* # Jump to label and do not save return address

jal *label* # Jump to label and set return address to ra

jalr t0 # Jump to address in t0 and set return address to ra

jalr t0, -100 # Jump to address t0-100 and set return address to ra

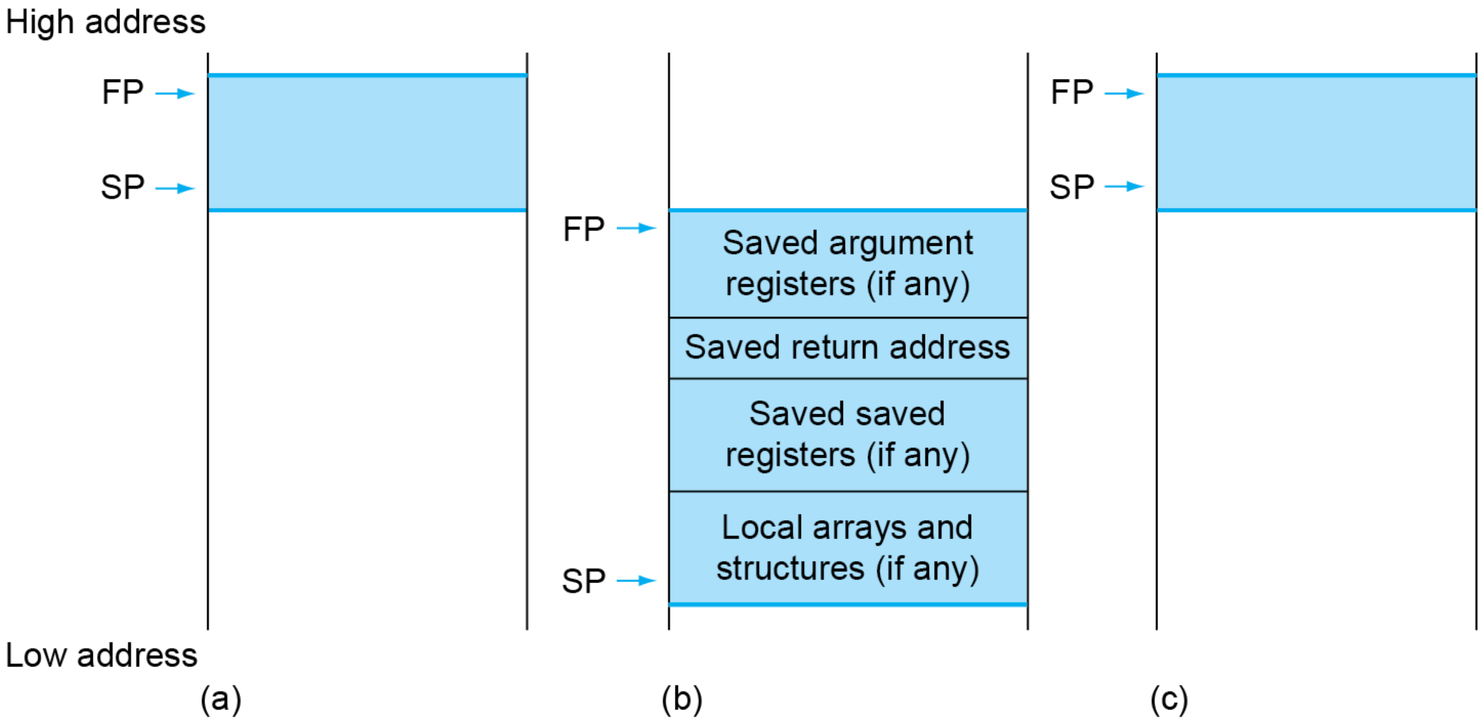
jr t0 # Jump Register: Jump to address in t0

jr t0, -100 # Jump Register: Jump to address t0-100

Stack

- **Stack** is a data structure for spilling registers organized as a last-in-first-out queue
- Dynamic memory for storing data (such as **local variables**) for function calls is organized as a task
- **Stack pointer** is a value denoting the most recently allocated address on the stack
- **Push** means to add element to stack
- **Pop** means to remove element from stack

Local Data on Stack



- Local data allocated by callee
  - Local variables, arrays, etc.
- Function frame (activation record)
  - Segment of stack containing function's saved registers and local variables

Saving Registers

A function can overwrite values of registers. Sometimes is undesirable. There are special rules to handle this issues. They specify who is responsible for saving the registers.

- **Callee-saved register** is a register saved by the routine being called
- **Caller-saved register** is register saved by the routine making a function call

Function Example

```
int leaf_example (int g, int h, int i, int j) {
  int f = (g + h) - (i + j);
  return f;
}
```

Requirements:

- arguments g, ..., j in a0 (x10)...a3 (x13)
- f in s4 (x20)
- temporaries t0 (x5), t1 (x6)
- need to save t0, t1, s4 on stack

Function Assembly Code

```
main:
    read_int(t0) # read g
    read_int(t1) # read h
    read_int(t2) # read i
    read_int(t3) # read j
    mv a0, t0
    mv a1, t1
    mv a2, t2
    mv a3, t3
    jal ra, leaf_example
    mv  t4, a0
    print_int(t0, t1, t2, t3, t4)
    li a7, 10
    ecall
```

```
leaf_example:
    addi sp, sp, -12
    sw   t0, 8(sp)
    sw   t1, 4(sp)
    sw   s4, 0(sp)
    add  t0, a0, a1
    add  t1, a2, a3
    sub  s4, t0, t1
    mv   a0, s4
    lw   s4, 0(sp)
    lw   t1, 4(sp)
    lw   t0, 8(sp)
    addi sp, sp, 12
    jalr x0, 0(ra)
```

Preserving Callee-Saved Registers

- **Preserve registers:**

```
addi sp,  sp, -20  # make room on stack for 5 registers
sw   ra,   16(sp)  # save ra (x1) on stack
sw   s1,   12(sp)  # save s1 (x9) on stack
sw   s2,    8(sp)  # save s2 (x18) on stack
sw   s3,    4(sp)  # save s3 (x19) on stack
sw   s4,    0(sp)  # save s4 (x20) on stack
```

- **Restore registers:**

```
lw   s4,   0(sp)   # restore s4 (x20) from stack
lw   s3,   4(sp)   # restore s3 (x19) from stack
lw   s2,   8(sp)   # restore s2 (x18) from stack
lw   s1,  12(sp)   # restore s1 (x9) from stack
lw   ra,  16(sp)   # restore ra (x1) from stack
addi sp, sp, 20    # restore stack pointer
jalr zero, 0(ra)   # return to caller
```

Preserving Caller-Saved Registers

- **Preserve registers:**

```
addi sp, sp, -16   # make room on stack for 4 registers
sw   t0, 12(sp)    # save t0 (x5) on stack
sw   t1,  8(sp)    # save t1 (x6) on stack
sw   t2,  4(sp)    # save t2 (x7) on stack
sw   t3,  0(sp)    # save t3 (x28) on stack
jal  ra, callee    # jump to callee
```

- **Restore registers:**

```
lw   t3,  0(sp)    # restore t3 (x28) from stack
lw   t2,  4(sp)    # restore t2 (x7) from stack
lw   t1,  8(sp)    # restore t1 (x6) from stack
lw   t0, 12(sp)    # restore t0 (x5) from stack
addi sp, sp, 16    # restore stack pointer
```

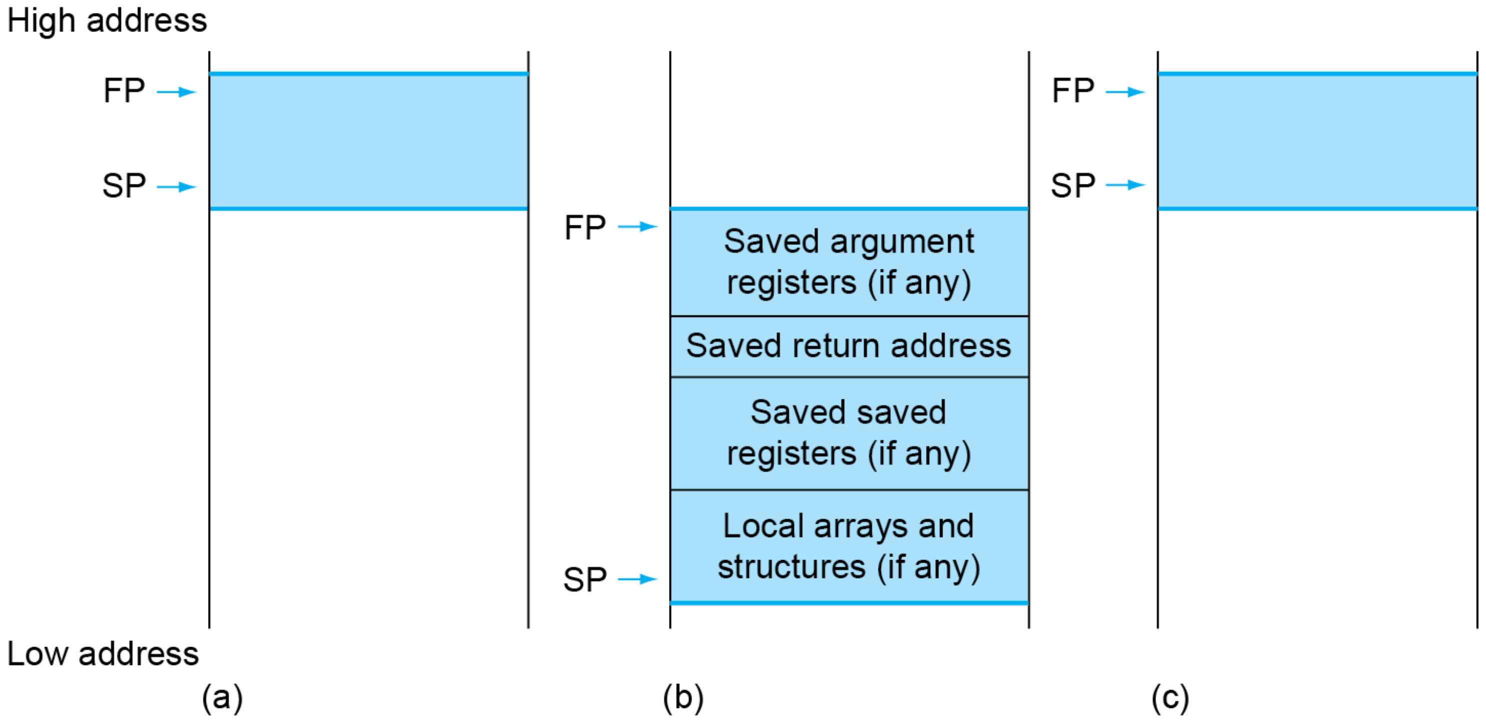
Recursive Function Example

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

```
fact:
    addi  t0, a0, -1
    bgez  t0, fact_else
    li    a0, 1
    jalr  zero, 0(ra)
fact_else:
    addi  sp, sp, -8
    sw    ra, 4(sp)
    sw    a0, 0(sp)
    addi  a0, a0, -1
    jal   ra, fact
    mv    t1, a0
    lw    a0, 0(sp)
    lw    ra, 4(sp)
    addi  sp, sp, 8
    mul   a0, a0, t1
    jalr  zero, 0(ra)
```

Frame Pointer

Frame pointer is a value denoting the location of the saved registers and local variables for a given procedure. Simplifies programming because when stack-pointer changes programmers have to use different offsets to access the same values.



Using Frame Pointer

```
main:
    mv   fp, sp          → frame pointer initialization
    jal  ra, func        → function call
    li   a7, 10
    ecall                → system call "exit"
func:
    addi sp, sp, -4
    li   t0, 1
    sw   t0, 0(sp)
    addi sp, sp, -4
    li   t0, 2
    sw   t0, 0(sp)       → sp-relative stores
    addi sp, sp, -4
    li   t0, 3
    sw   t0, 0(sp)
    lw   t0, 0(fp)
    print_int(t0)
    lw   t0, -4(fp)
    print_int(t0)        → fp-relative loads
    lw   t0, -8(fp)
    print_int(t0)
    addi sp, sp, 12
    jalr zero, 0(ra)     → return
```

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
```