E85

Digital Electronics & Computer Architecture

Lecture 17:

Function Calls

Function Calls

- **Caller:** calling function (in this case, `main`)
- **Callee:** called function (in this case, `sum`)

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

RISC-V Function Conventions

- **Call Function:** jump and link (`jal`)
- **Return** from function: jump register (`jr ra`)
- **Arguments**: `a0 – a7`
- **Return value**: `a0`

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

RISC-V assembly code

```
0x00000300 main:   jal  ra, simple  # call
0x00000304         add  s0, s1, s1
...                ...

0x0000051c simple: jr   ra         # return
```

`void` means that `simple` doesn't return a value

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

RISC-V assembly code

```
0x00000300 main:   jal  ra, simple  # call
0x00000304         add  s0, s1, s1
...                ...

0x0000051c simple: jr   ra          # return
```

`jal ra, simple:`

`ra` = PC + 4 (0x00000304)

jumps to simple label (PC = 0x0000051c)

`jr ra:`

PC = `ra` (0x00000304)

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

RISC-V assembly code

```
0x00000300 main:   jal  simple      # call
0x00000304         add  s0, s1, s1
...                ...

0x0000051c simple: jr   ra          # return
```

- Preferred instruction:

  `jal simple` - a pseudoinstruction for **`jal ra, simple`**

- **Pseudoinstructions** are not actual RISC-V instructions but they are often simpler for the programmer.
- They are converted to real RISC-V instructions by the assembler

Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

RISC-V Function Conventions

- **Call Function:** jump and link (`jal`)
- **Return** from function: jump and link register (`jr ra`)
- **Arguments**: `a0 – a7`
- **Return value**: `a0`

Input Arguments & Return Value

C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```

Input Arguments & Return Value

RISC-V assembly code

```
# s7 = y
main:
. . .
addi a0, zero, 2  # argument 0 = 2
addi a1, zero, 3  # argument 1 = 3
addi a2, zero, 4  # argument 2 = 4
addi a3, zero, 5  # argument 3 = 5
jal  diffofsums   # call function
add  s7, a0, zero # y = returned value
. . .
# s3 = result
diffofsums:
add  t0, a0, a1   # t0 = f + g
add  t1, a2, a3   # t1 = h + i
sub  s3, t0, t1   # result = (f + g) - (h + i)
add  a0, s3, zero # put return value in a0
jr   ra           # return to caller
```

`jal diffofsums`
is pseudocode for
`jal ra, diffofsums`

Input Arguments & Return Value

RISC-V assembly code

```
# s3 = result
diffofsums:
  add  t0, a0, a1    # t0 = f + g
  add  t1, a2, a3    # t1 = h + i
  sub  s3, t0, t1    # result = (f + g) - (h + i)
  add  a0, s3, zero  # put return value in a0
  jr   ra            # return to caller
```

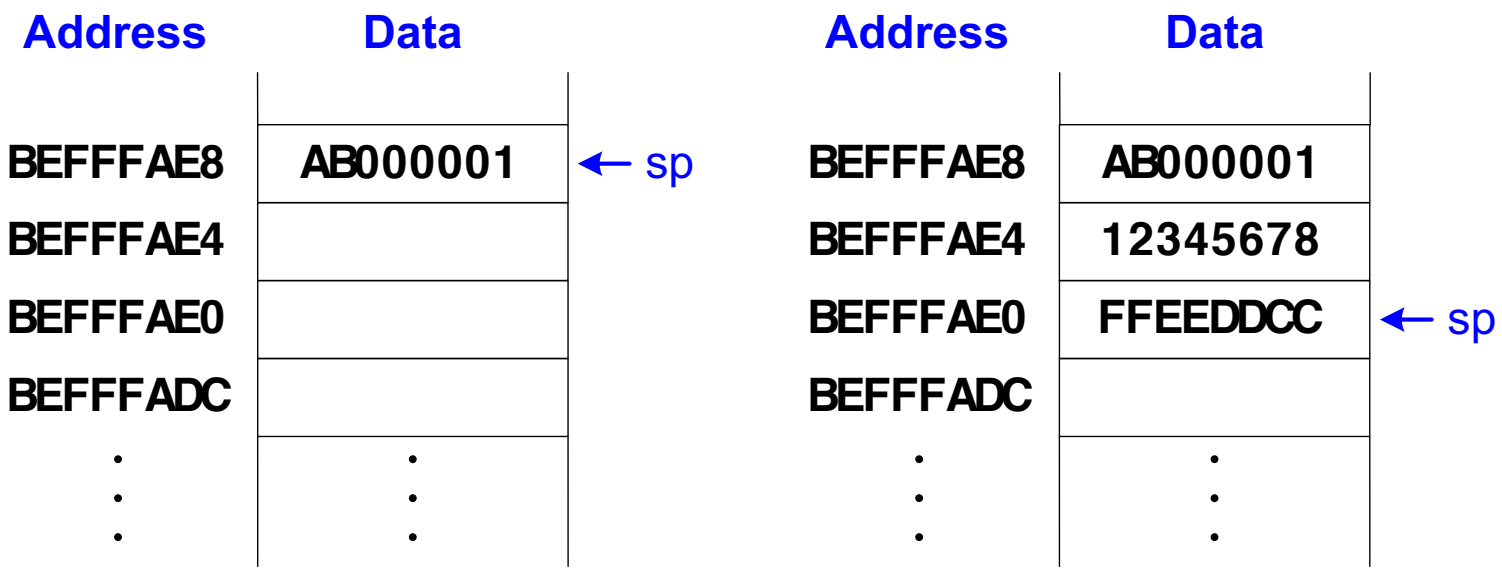
- `diffofsums` overwrote 3 registers: `t0`, `t1`, `s3`
- `diffofsums` can use the *stack* to temporarily store registers

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands*:** uses more memory when more space needed
- ***Contracts*:** uses less memory when the space is no longer needed

The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `sp` points to top of the stack



How Functions use the Stack

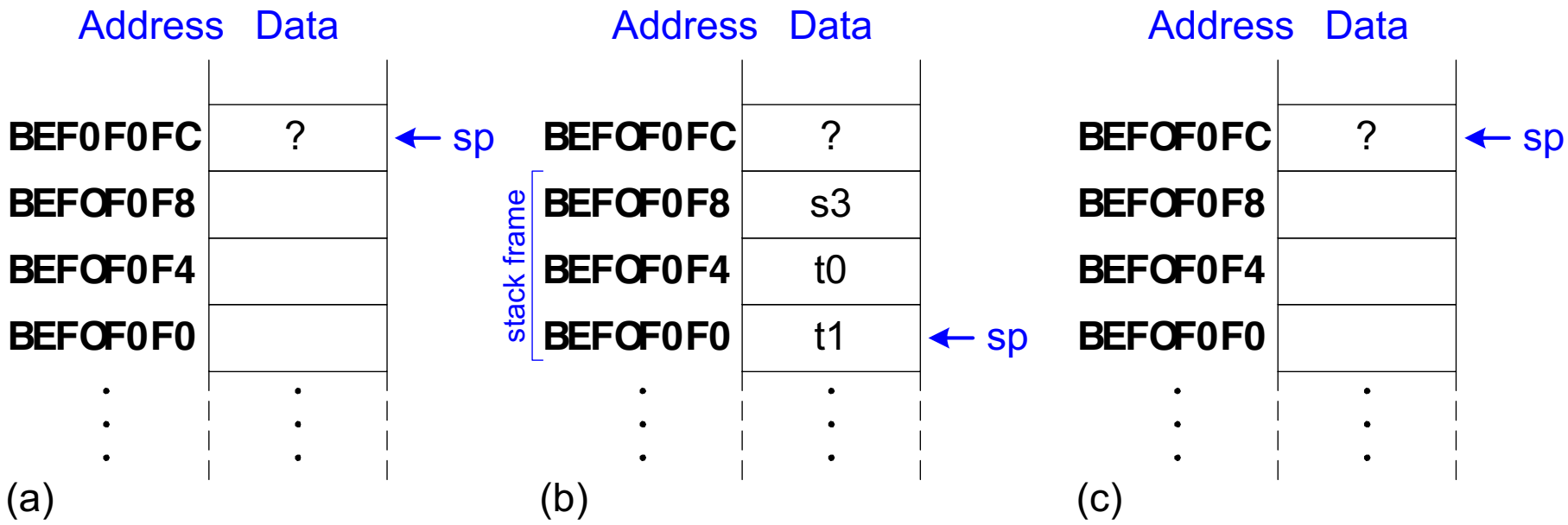
- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `t0`, `t1`, `s3`

```
# RISC-V assembly
# s3 = result
diffofsums:
  add  t0, a0, a1    # t0 = f + g
  add  t1, a2, a3    # t1 = h + i
  sub  s3, t0, t1    # result = (f + g) - (h + i)
  add  a0, s3, zero  # put return value in a0
  jr   ra            # return to caller
```

Storing Register Values on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -12       # make space on stack to
                         # store three registers
  sw   s3, 8(sp)         # save s3 on stack
  sw   t0, 4(sp)         # save t0 on stack
  sw   t1, 0(sp)         # save t1 on stack
  add  t0, a0, a1        # t0 = f + g
  add  t1, a2, a3        # t1 = h + i
  sub  s3, t0, t1        # result = (f + g) - (h + i)
  add  a0, s3, zero      # put return value in a0
  lw   t1, 0(sp)         # restore $t1 from stack
  lw   t0, 4(sp)         # restore $t0 from stack
  lw   s3, 8(sp)         # restore $s3 from stack
  addi sp, sp, 12        # deallocate stack space
  jr   ra                # return to caller
```

The Stack During `diffofsums` Call



Register Saving Conventions

Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
`s0-s11`	`t0-t6`
`sp`	`ra`
stack above `sp`	`a0-a7`
	stack below `sp`

Storing Saved Registers on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -4        # make space on stack to
                         # store one register
  sw   s3, 0(sp)         # save s3 on stack
  add  t0, a0, a1        # t0 = f + g
  add  t1, a2, a3        # t1 = h + i
  sub  s3, t0, t1        # result = (f + g) - (h + i)
  add  a0, s3, zero      # put return value in a0
  lw   s3, 0(sp)         # restore $s3 from stack
  addi sp, sp, 4         # deallocate stack space
  jr   ra                # return to caller
```

Optimized `diffofsums`

```
# a0 = result
diffofsums:
  add  t0, a0, a1    # t0 = f + g
  add  t1, a2, a3    # t1 = h + i
  sub  a0, t0, t1    # result = (f + g) - (h + i)
  jr   ra            # return to caller
```

Non-Leaf Function Calls

Non-leaf function:

a function that calls another function

```
func1:
  addi sp, sp, -4     # make space on stack
  sw   ra, 0(sp)      # save ra on stack
  jal  func2
  ...
  lw   ra, 0(sp)      # restore ra from stack
  addi sp, sp, 4      # deallocate stack space
  jr   ra             # return to caller
```

Function Call Summary

- **Caller**
  - Put arguments in `a0-a7`
  - Save any needed registers (`ra`, maybe `t0-t6`)
  - Call function: `jal callee`
  - (Possibly restore registers)
  - Look for result in `a0`
- **Callee**
  - Save registers that might be disturbed `(s0-s11)`
  - Perform function
  - Put result in `a0`
  - Restore registers
  - Return: `jr ra`

Recursive Functions

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n - 1));
}
```

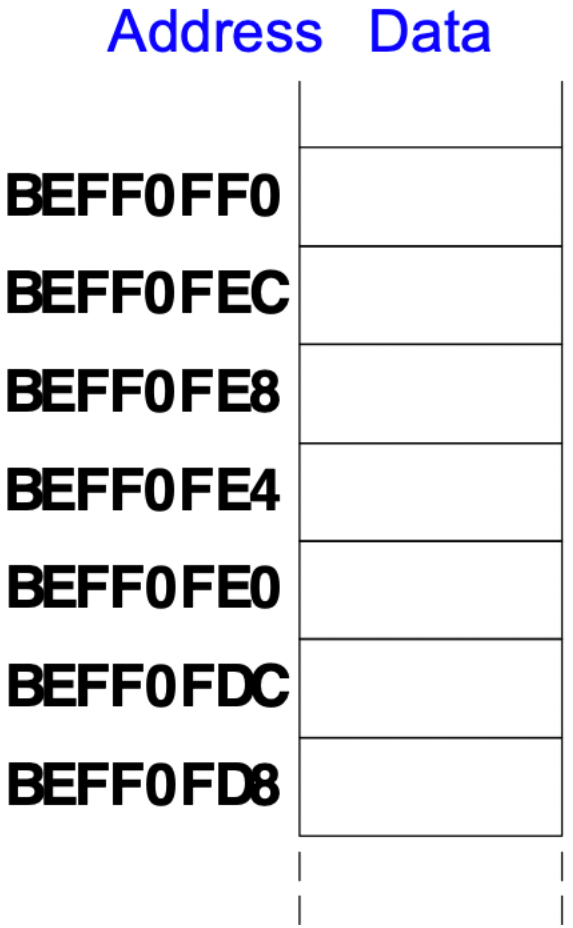
Recursive Functions

```
0x8500 factorial: addi sp, sp, -8     # make room for a0, ra
0x8504            sw   a0, 4(sp)
0x8508            sw   ra, 0(sp)
0x850C            addi t0, zero, 1    # temporary = 1
0x8510            bgt  a0, t0, else   # if n>1, go to else
0x8514            addi a0, zero, 1    # otherwise, return 1
0x8518            addi sp, sp, 8      # restore sp
0x851C            jr   ra             # return
0x8520 else:      addi a0, a0, -1     # n = n - 1
0x8524            jal  factorial      # recursive call
0x8528            lw   ra, 0(sp)      # restore ra
0x852C            lw   t1, 4(sp)      # restore n into t1
0x8530            addi sp, sp, 8      # restore sp
0x8534            mul  a0, t1, a0     # a0 = n*factorial(n-1)
0x8538            jr   ra             # return
```

Recursive Functions

What does the stack look like when executing `factorial(3)`?

```
0x8500 factorial: addi sp, sp, -8
0x8504            sw   a0, 4(sp)
0x8508            sw   ra, 0(sp)
0x850C            addi t0, zero, 1
0x8510            bgt  a0, t0, else
0x8514            addi a0, zero, 1
0x8518            addi sp, sp, 8
0x851C            jr   ra
0x8520 else:      addi a0, a0, -1
0x8524            jal  factorial
0x8528            lw   ra, 0(sp)
0x852C            lw   t1, 4(sp)
0x8530            addi sp, sp, 8
0x8534            mul  a0, t1, a0
0x8538            jr   ra
```



Recursive Functions

Stack (a) before, (b) during, and (c) after recursive call.