

IV - Assembly - II

ICS 26Fall

Junqi Xu 2026.9.30

Recall

Concepts:

IR

PIC, ASLR, PC-Relative

GOT, PLT, Relocation

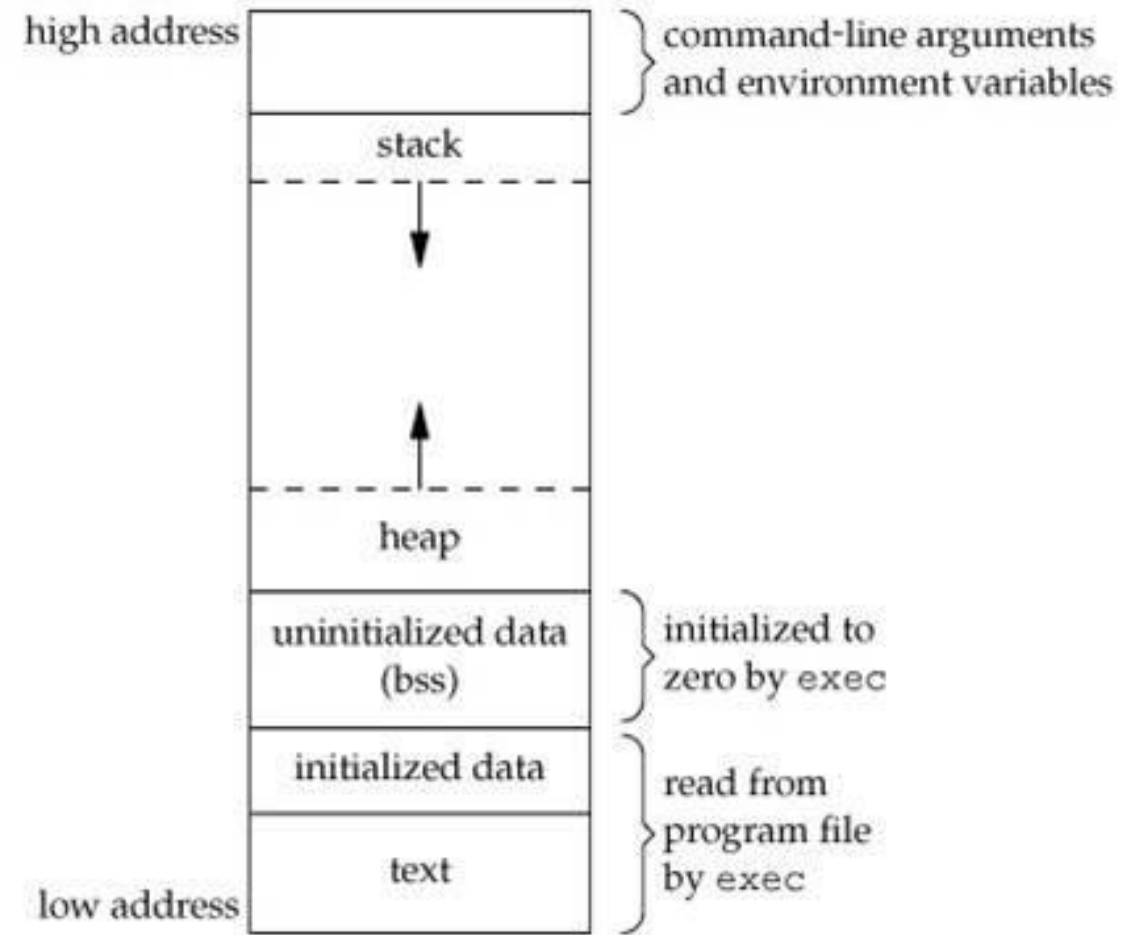


Fig.1^[1]

[1] Stevens, W. R., & Rago, S. A. (2013). Advanced Programming in the UNIX Environment (3rd ed.). Addison-Wesley.

Topics

- Registers
- Control Flow
- Data Flow

Registers - ABI: Caller-Saved vs. Callee-Saved

View: Current Function

Caller-Saved:

I own the value in the register.

If I want to save it, I have to do it by myself.

Callee-Saved:

The function who called me owns the value.

If I want to use the register, I have to save the value in it, and recover it before I pass the control back to the caller (that is, return).

Registers - ABI: Caller-Saved vs. Callee-Saved

Caller-Saved Registers (vs. Callee-Saved)

Pros:

Make it cheap to store temporary value.

Could be used to pass arguments.

Cons:

Value stored in it would be refreshed every time we make a call.

Discuss:

Could you give a case when callee-saved registers are more efficient?

Registers - ABI: Caller-Saved vs. Callee-Saved

One case when callee-saved register is more efficient:

```
for(int i = 0; i < size, ++i){  
    func(i)  
}
```

i and **size** would be stored in callee-saved registers.

Registers - ABI: Caller-Saved vs. Callee-Saved

All caller-saved registers could be used to pass arguments.

But, ABI says, **%r10** and **%r11** couldn't be used to do so...

Almost no functions need more than 6 arguments. (If you really need, consider to use a structure or class to pass context.)

So ABI uses them to accelerate other things...

Registers - ABI: %r10

%r10 is used to save **static link pointer**.

If you know Lisp well, it is exactly the pointer to the parent environment in the implementation of lambda expressions.

If you don't know Lisp, you should at least know Callable Class (or Closure Type, a.k.a. Lambda) in C++, or Closure in Python or Rust.

Registers - ABI: %r10

There's no concepts like closure in C standards!

But **gcc** does support it (while **clang** doesn't).

```
int func_p(int arg1, int arg2) {  
    int a_p = arg1 + arg2;  
    int func_c(int arg1, int arg2) {  
        int a_c = arg1 - arg2;  
        return a_p + a_c;  
    }  
    return func_c(arg1, arg2);  
}
```

Registers - ABI: %r10

```
int func_p(int arg1, int arg2) {  
    int a_p = arg1 + arg2;  
    int func_c(int arg1, int arg2) {  
        int a_c = arg1 - arg2;  
        return a_p + a_c;  
    }  
    return func_c(arg1, arg2);  
}
```

Stack Frame:func_p

arg1

arg2

a_p

func_c

Stack Frame:func_c

static chain pointer

arg1

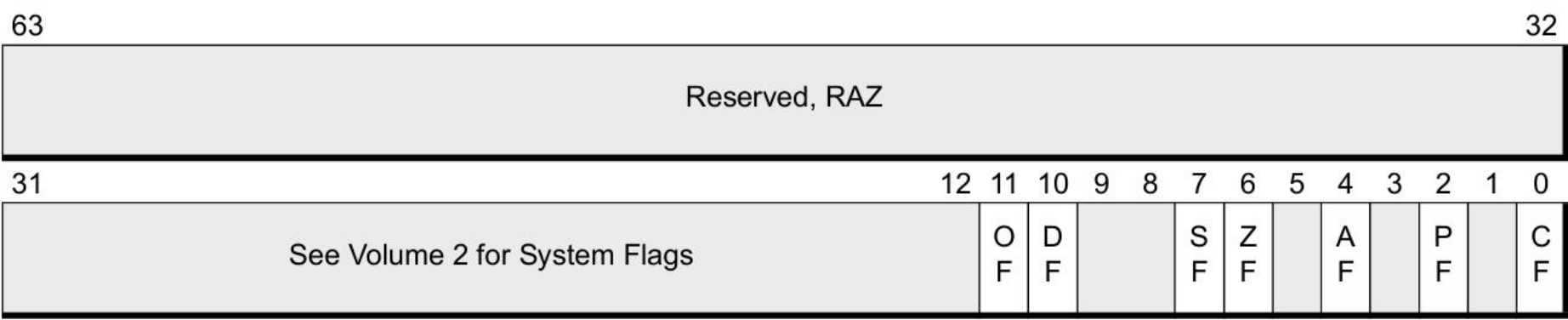
arg2

a_c

Registers - ABI: RFLAGS

RFLAGS is also a register (like `%rip`).

It stores runtime states of a process.



Bits	Mnemonic	Description	R/W
11	OF	Overflow Flag	R/W
10	DF	Direction Flag	R/W
7	SF	Sign Flag	R/W
6	ZF	Zero Flag	R/W
4	AF	Auxiliary Carry Flag	R/W
2	PF	Parity Flag	R/W
0	CF	Carry Flag	R/W

Fig.2^[2]

[2] Advanced Micro Devices, Inc. (2024). AMD64 Architecture Programmer's Manual Volume 1: Application Programming (Publication No. 24592, Rev. 3.42). Advanced Micro Devices, Inc.

Registers - ABI: **syscall**

A common process only owns the memory allocated by the kernel.

What if it needs to access other memory?

e.g. Read & Write a file; Execute another program; Need more memory...

ABI provides lots of **syscalls** to make it possible for process to request the kernel to help them do these things!

Registers - ABI: syscall

Different from common call, syscall needs the current process hand over control to the kernel, which means RFLAGS would be changed.

In particular, syscall should save the value of RFLAGS, and sysret should recover it.

Registers - ABI: %r11

%r11 is used to pass the value of RFLAGS to the kernel when syscall happens.

Then the kernel would save the value of **%r11** into somewhere safe and use the value to recover RFLAGS when sysret happens.

Registers - Physical Registers

Physical Registers:

- Real registers

- Lots of constraints (ISA & ABI)

- Hard to analyze.

Registers - Virtual Registers

Virtual Registers:

Registers only used to store values.

No constraints, unlimited number.

Named as %1, %2, ..., %920, %921, ...

Easy (not that easy) to analyze!

Control Flow

In what order do the instructions be executed?

SEQ: Execute instructions one by one.

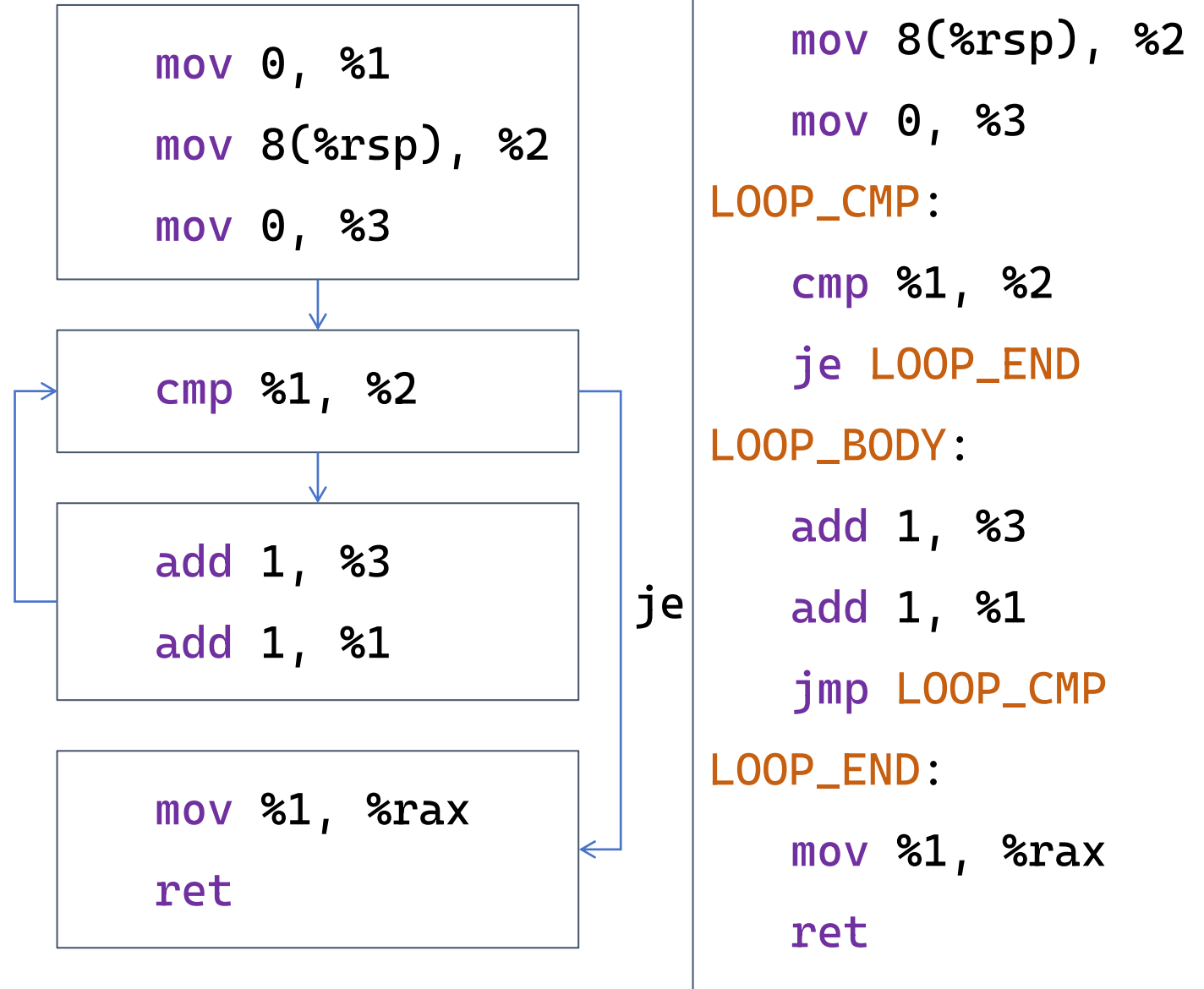
JMP: Execute an instruction in another place of the function.

CALL: Execute another function.

Control Flow - Basic Blocks & CFG

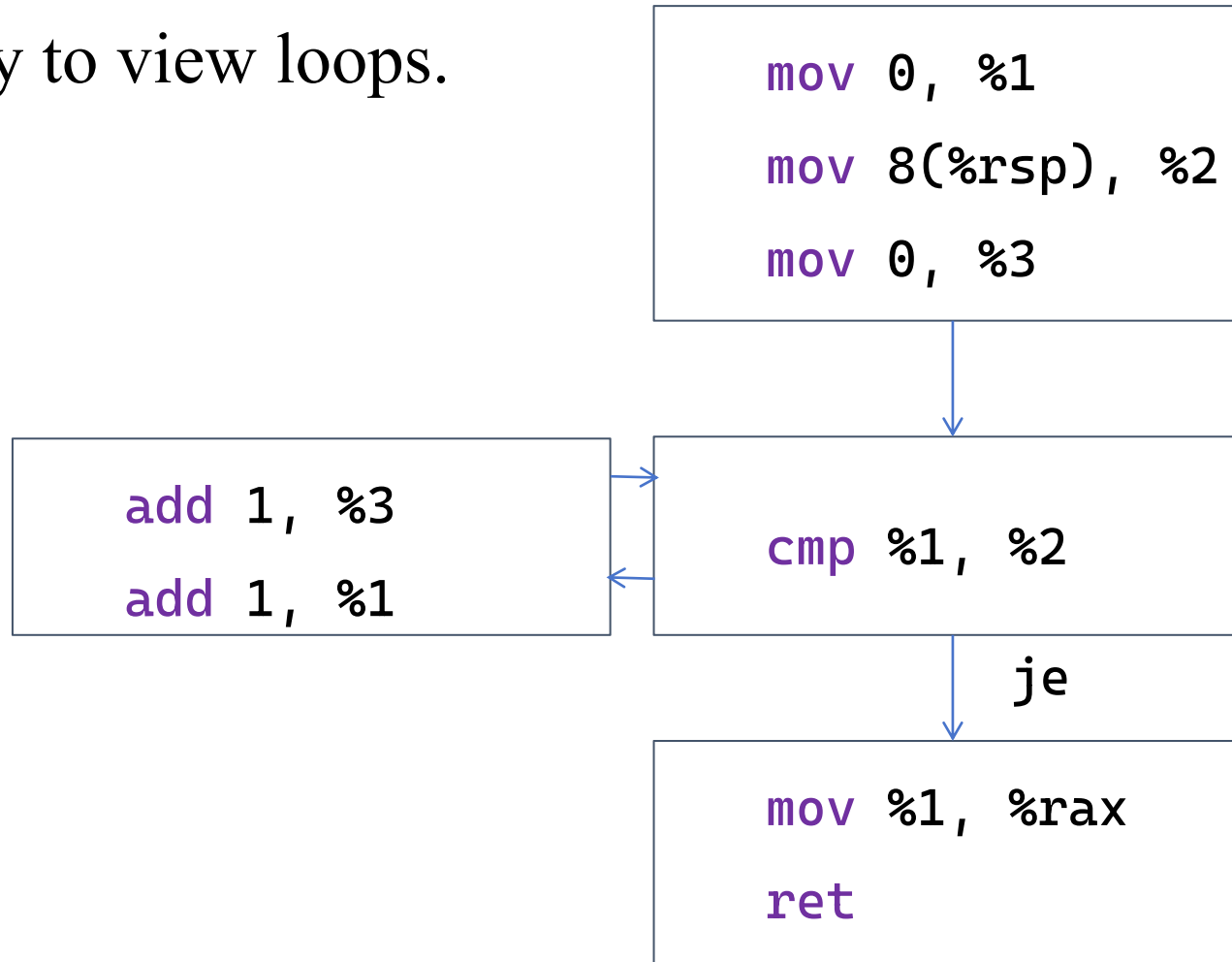
It is JMP or CALL that really matters.

Instructions that are executed in SEQ could be abstracted into a **block**.



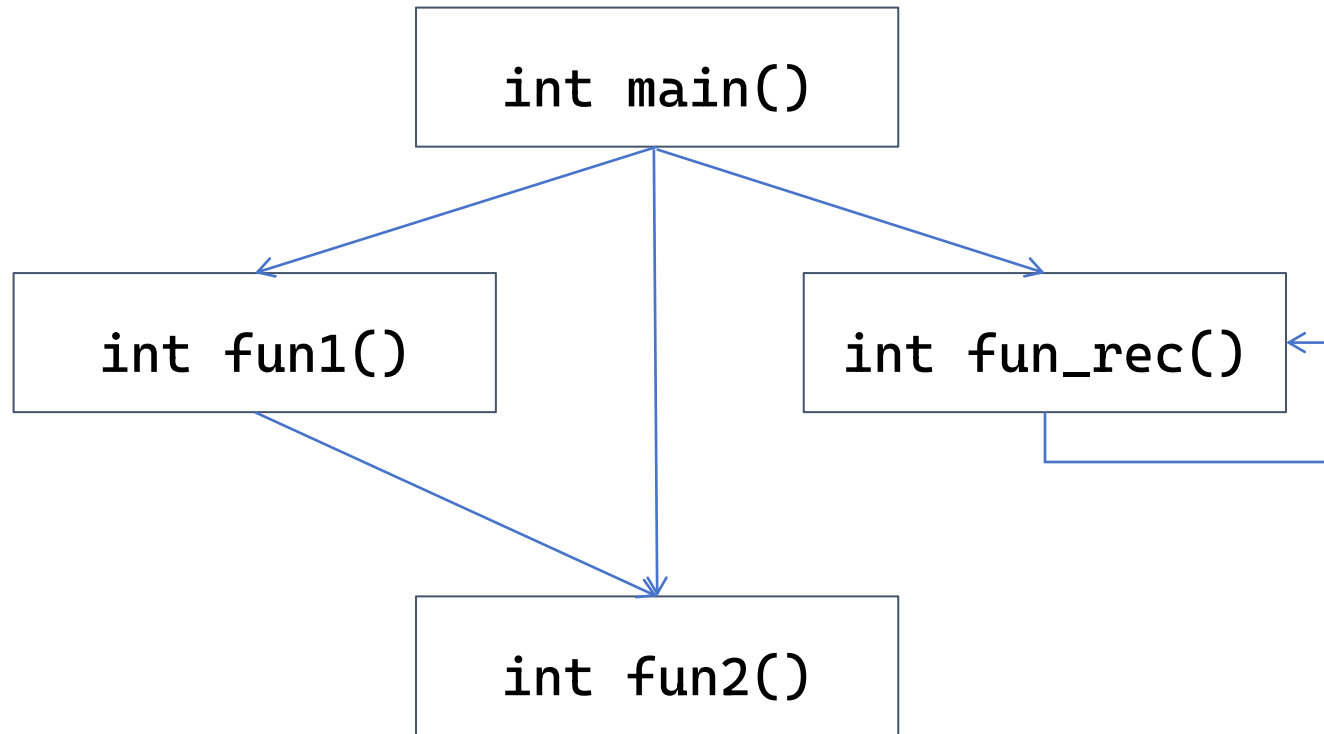
Control Flow - Basic Blocks & CFG

Another way to view loops.



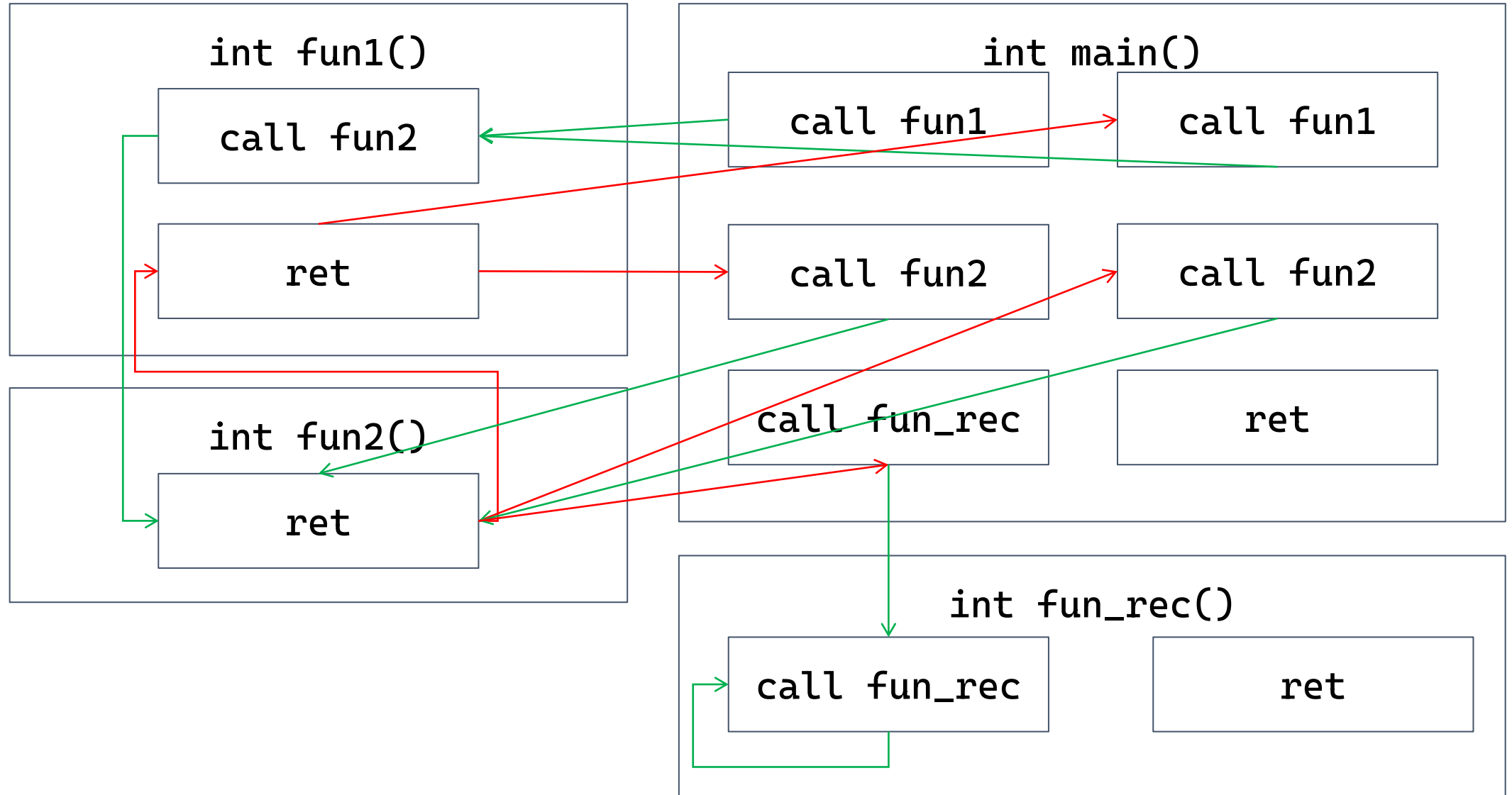
```
mov 0, %1
mov 8(%rsp), %2
mov 0, %3
LOOP_CMP:
    cmp %1, %2
    je LOOP_END
LOOP_BODY:
    add 1, %3
    add 1, %1
    jmp LOOP_CMP
LOOP_END:
    mov %1, %rax
    ret
```

Control Flow - Call Graph



```
int fun1() { return fun2(); }  
int fun2() { return 42; }  
int fun_rec() { return fun_rec(); }  
  
int main() {  
    int x = fun1() * fun1();  
    int y = fun2() + fun2();  
    fun_rec();  
}
```

Control Flow - ICFG



Data Flow - Lifetime

How the data change through the execution.

```
mov 8(%rsp), %1
mov 16(%rsp), %2
mov 24(%rsp), %3
add %1, %2
mov 32(%rsp), %4
mov 40(%rsp), %5
add %2, %3
mul %4, %5
call fun
sub %3, %5
mov %5, %rax
ret
```

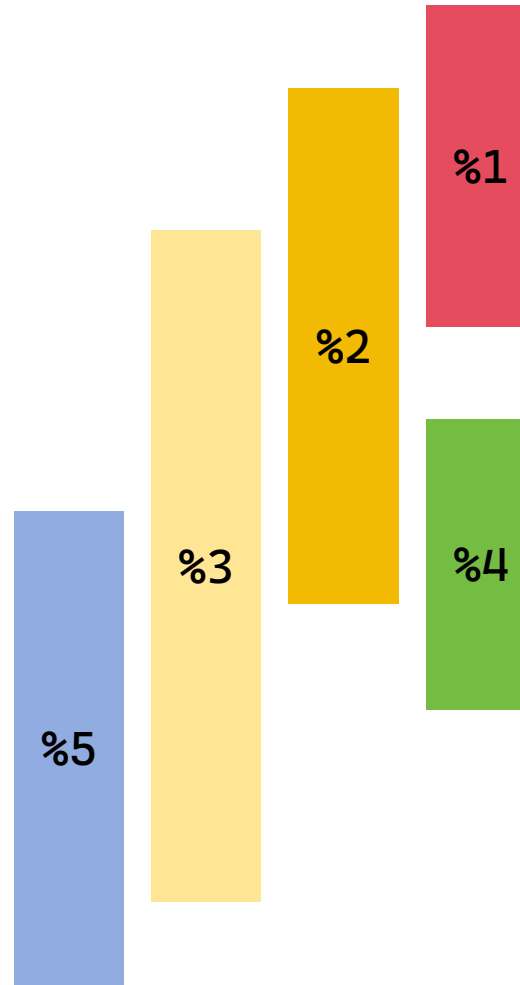
The diagram illustrates the data flow and lifetime of registers %1 through %5 across the assembly code. Arrows indicate the flow of data between instructions:

- Red arrow:** From `mov 8(%rsp), %1` to `add %1, %2`.
- Yellow arrow:** From `mov 16(%rsp), %2` to `add %2, %3`.
- Yellow arrow:** From `mov 24(%rsp), %3` to `add %2, %3`.
- Green arrow:** From `mov 32(%rsp), %4` to `mul %4, %5`.
- Blue arrow:** From `mov 40(%rsp), %5` to `mul %4, %5`.
- Yellow arrow:** From `add %2, %3` to `sub %3, %5`.
- Blue arrow:** From `mul %4, %5` to `sub %3, %5`.
- Blue arrow:** From `sub %3, %5` to `mov %5, %rax`.

Data Flow - Lifetime

%1 and **%4** would never live at the same time, so they can be assigned the same physical register.

%3 and **%5** live through a **call** instruction, so they better be assigned callee-saved registers.

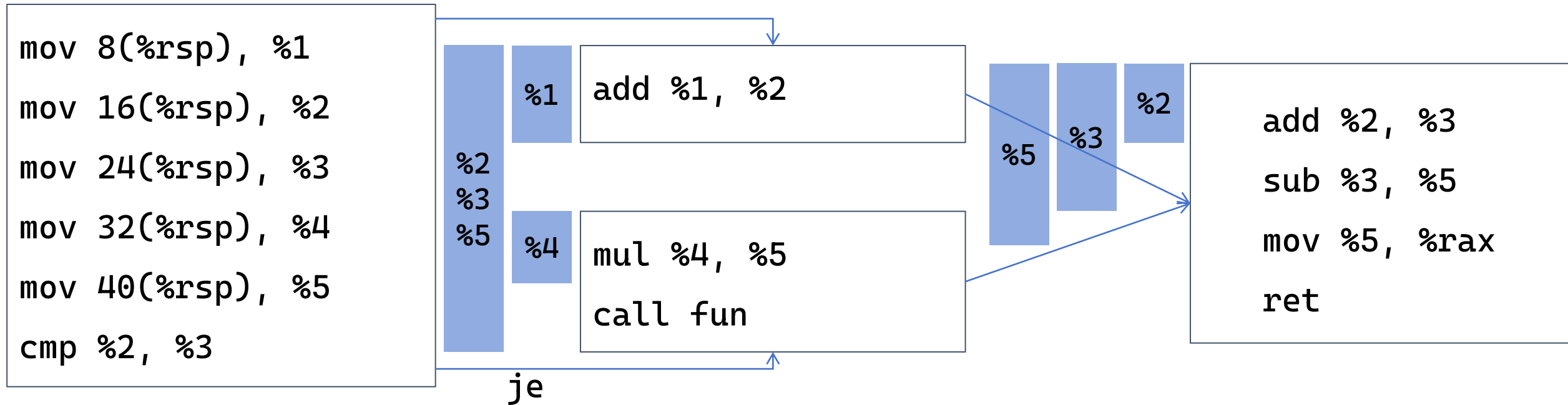


```
mov 8(%rsp), %1
mov 16(%rsp), %2
mov 24(%rsp), %3
add %1, %2
mov 32(%rsp), %4
mov 40(%rsp), %5
add %2, %3
mul %4, %5
call fun
sub %3, %5
mov %5, %rax
ret
```

Data Flow - Branch

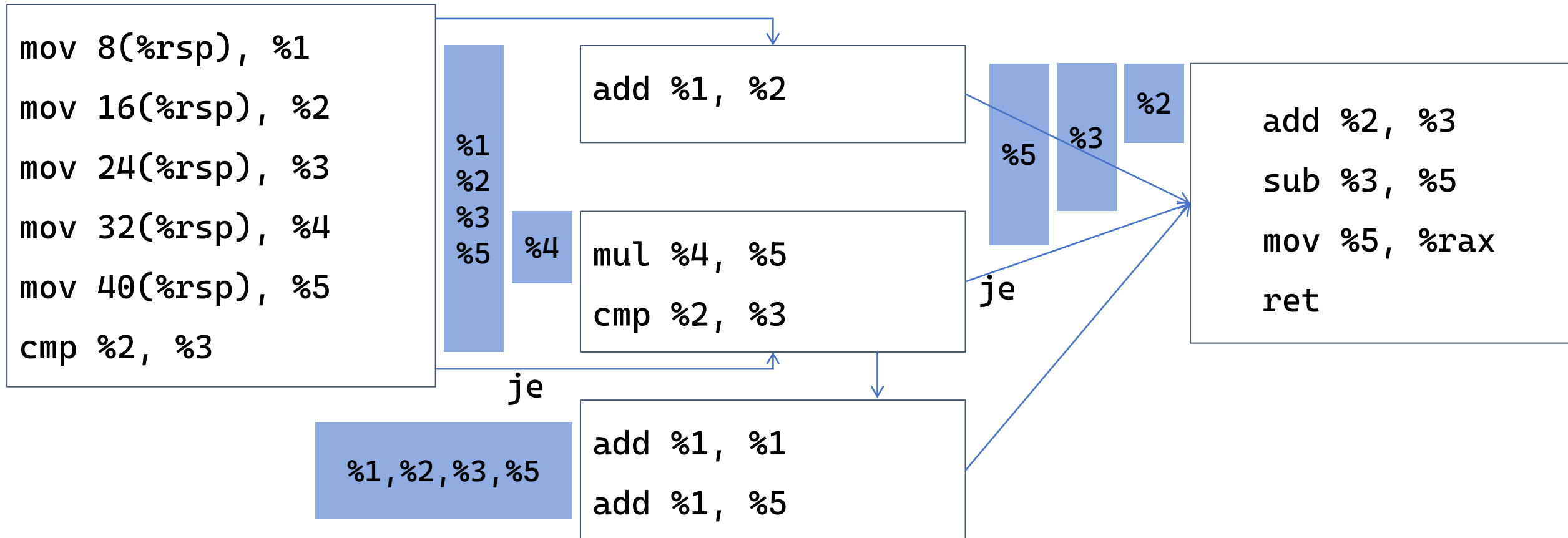
%1 and **%4** won't live at the same time.

%2, **%3** and **%5** all have to be callee-saved.



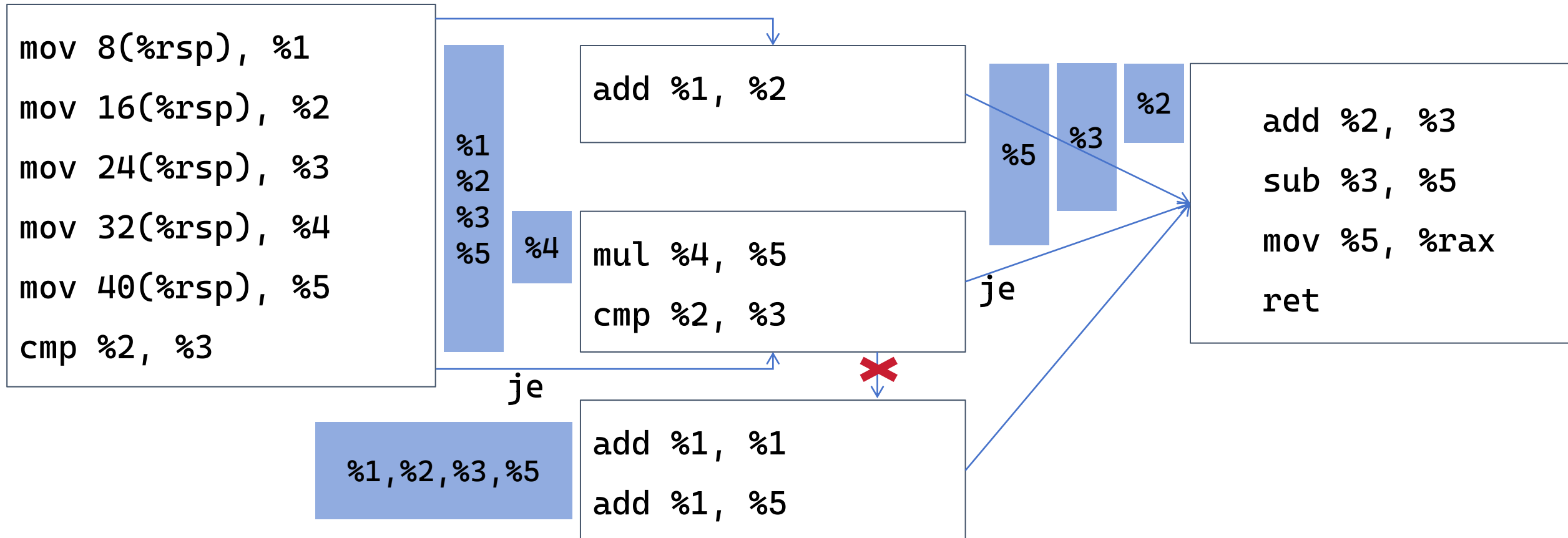
Data Flow - Dead Code Elimination

Branch gives us additional message.



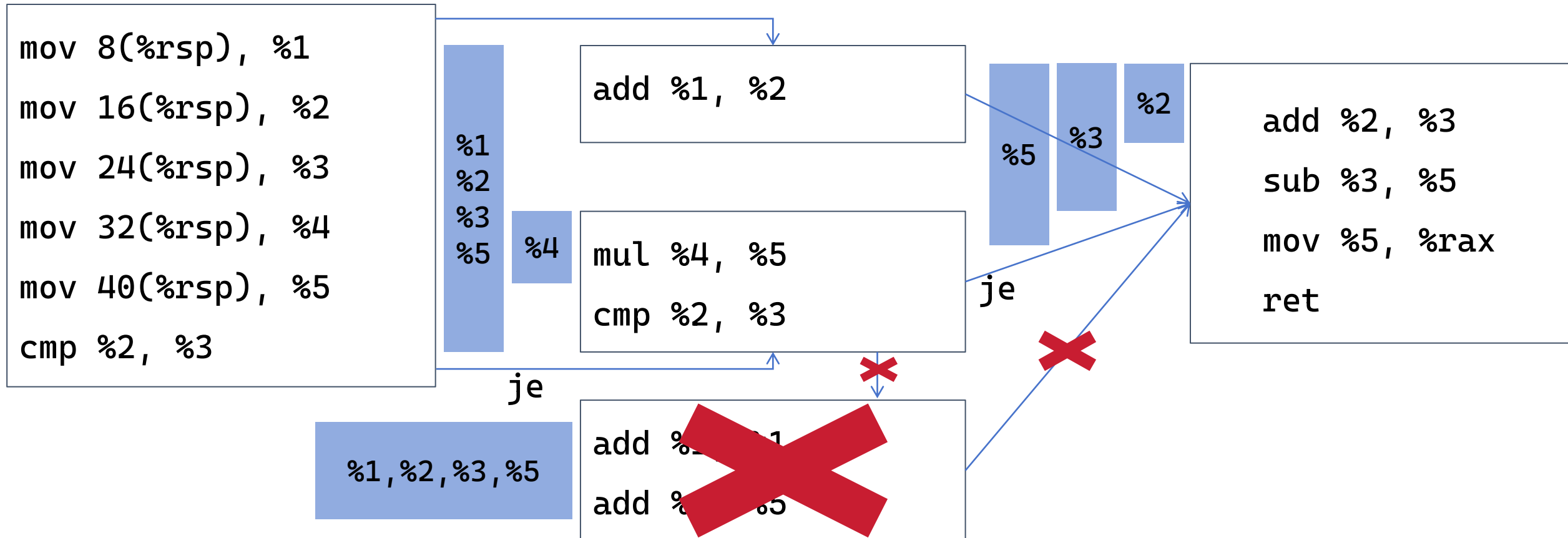
Data Flow - Dead Code Elimination

Branch gives us additional message.



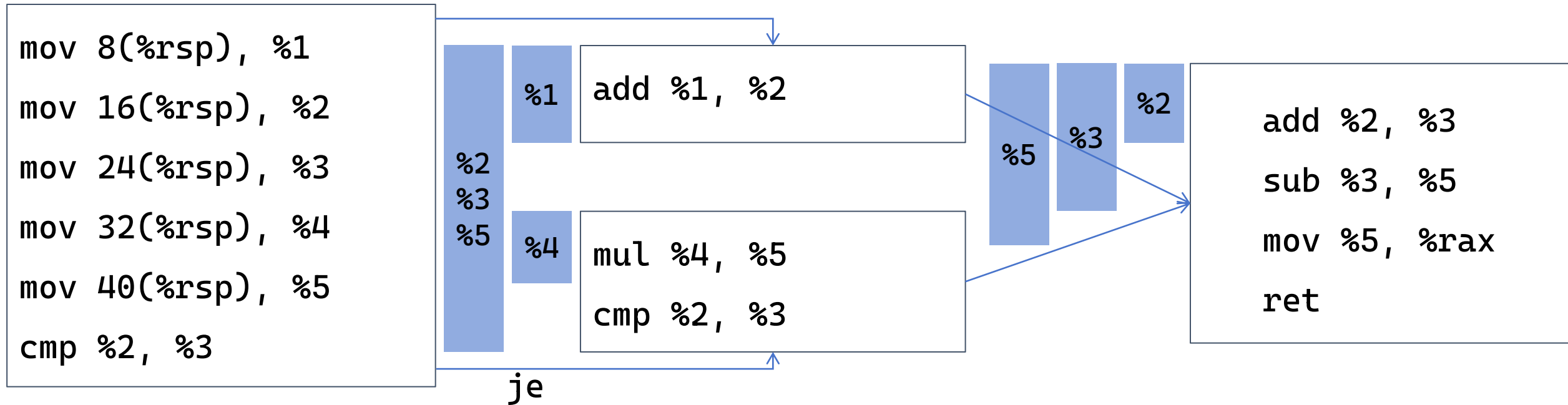
Data Flow - Dead Code Elimination

Branch gives us additional message.



Data Flow - Dead Code Elimination

Branch gives us additional message.



Takeaway

Concepts:

Caller-Saved Registers, Callee-Saved Registers

Static Link Pointer (%r10)*

RFLAGS, Syscall, Sysret, %r11

Virtual Registers

Basic Blocks, CFG, CG (Call graph), ICFG*

Data Flow Analysis*

Extra

VOTE: What's for next week?

Context: Pipeline is complex, but we only have one week to talk about it.

Solution: Subsequent 3 weeks for Chapter 4. (2 weeks for pipeline)

If you don't want it, I would show you some compilation optimization techniques (Chapter 6) next week.

Reference

- [1] Stevens, W. R., & Rago, S. A. (2013). **Advanced Programming in the UNIX Environment (3rd ed.)**. Addison-Wesley.
- [2] Advanced Micro Devices, Inc. (2024). **AMD64 Architecture Programmer's Manual Volume 1: Application Programming (Publication No. 24592, Rev. 3.42)**. Advanced Micro Devices, Inc.
- [3] Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2023). **Operating Systems: Three Easy Pieces (Version 1.10)**. Arpaci-Dusseau Books, LLC.
- [4] Yingfei, X. (2025). **软件分析技术 [Course website]**. Peking University.