Computer Architectures

Parameters Passing to Subroutines and Operating System Implemented Virtual Instructions (System Calls)

Pavel Píša

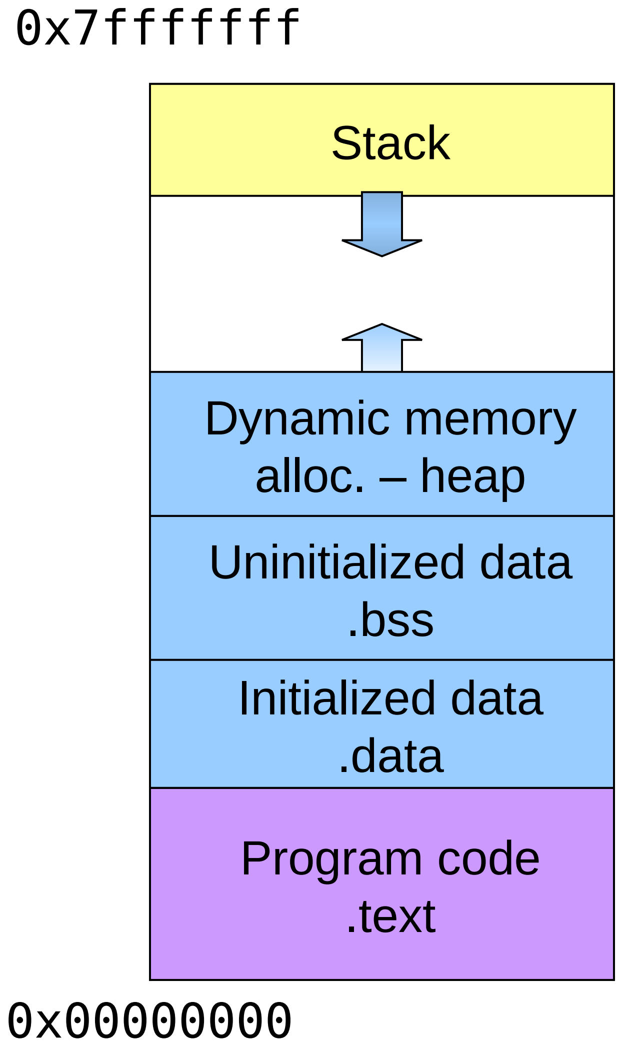
Czech Technical University in Prague, Faculty of Electrical Engineering

Kinds of Calling Conventions and Parameters Passing

- Regular (standard) functions (subroutines) calling
  - Parameters passing – in registers, on stack, through register windows
  - Calling convention (part of ABI – application binary interface) selected according to the CPU architecture supported options, agreement between compilers, system and additional libraries is required

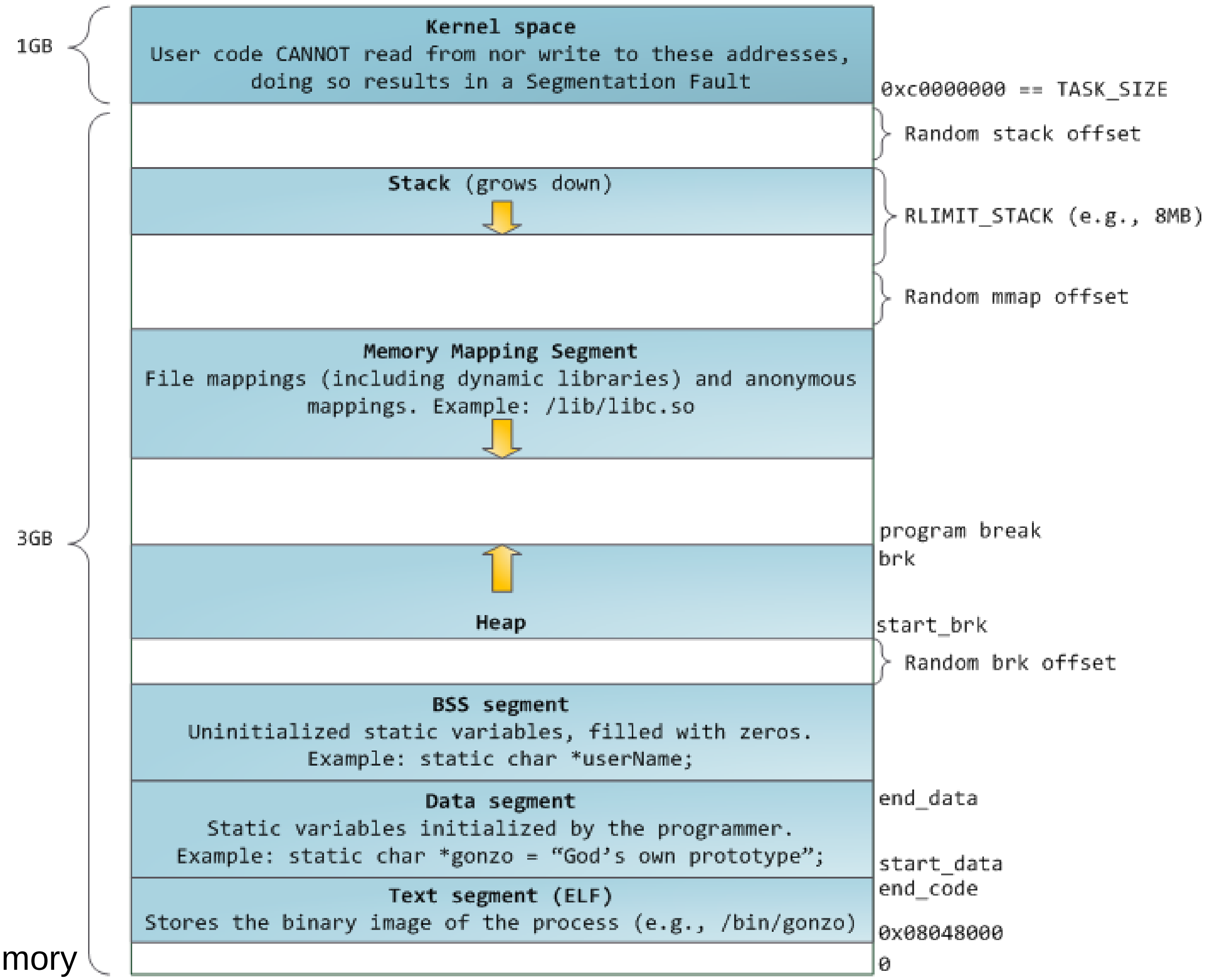
    (x86 Babylonian confusion of tongues - cdecl, syscall, optlink, pascal, register, stdcall, fastcall, safecall, thiscall MS/others)
  - Stack frames allocated space for local variables and C-library alloca()
- System calls (requests a service from OS kernel)
  - Control transfer (switching) from user to system (privileged) mode
- Remote functions calls and method invocations
  - Callee cannot read (easily) from memory space of caller process
  - Standards for network cases (RPC, CORBA, SOAP, XML-RPC)
  - Machine local same as networked + more: OLE, UNO, D-bus, etc.

Program Layout in Memory at Process Startup



- The executable file is mapped ("loaded") to process address space – sections **.data** and **.text** (note: LMA != VMA for some special cases)
- Uninitialized data area (**.bss – block** starting by symbol) is reserved and zeroed for C programs
- Stack pointer is set and control is passed to the function _**start**
- Dynamic memory is usually allocated above _**end** symbol pointing after **.bss**

Process Address Space (32-bit x86 Linux)



Based on:
Anatomy of a Program in Memory
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

Steps Processed During Subroutine Call

- Calling routine/program (caller) calculates the values of the parameters
- Caller prepares for overwrite of registers which ABI specifies as clobberable (can be modified by callee) storing values which are still neded
- Parameters values are placed in registers and or on stack according to used ABI calling convention
- Control is transferred to the first subroutine instruction while return address is saved on stack or stored in dedicated register
- Subroutine saves previous values of registers, which are to be used and are specified as callee saved/non-clobberable
- Space for local variables is allocated on stack
- The body of subroutine is executed
- The function result is placed to register(s) defined by calling convention
- Values of callee saved/non-clobberable registers are restored
- Return from subroutine instruction is executed, it can release stack space used for parameters but it is usually caller duty to adjust stack to free parameters

Example: RISC-V Calling Convention and Registers Usage

- **a0**, **a1**: arguments and function result value (registers x10, x11)
- **a2** – **a7**: arguments to call functions (x12 – x17)
- **t0** – **t6**: temporary/clobberable registers (x5 – x7, x28 – x31)
  - callee function can use them as it needs without saving
- **s0** – **s11**: saved (non-clobberable) registers (x8, x9, x18 – x21)
  - if used by callee, they has to be saved first and restored at return
- **gp**: global static data pointer (x3)
- **sp**: stack pointer (x2) – top of the stack, grows down to 0
- **fp/s0**: frame pointer (x8) – points to start of local variables on stack
- **ra**: return address register (x1) – implicitly written by **jal** instruction – jump and link – call subroutine
- **zero**: fixed zero and discard register (x0)

Example: MIPS Calling Convention and Registers Usage

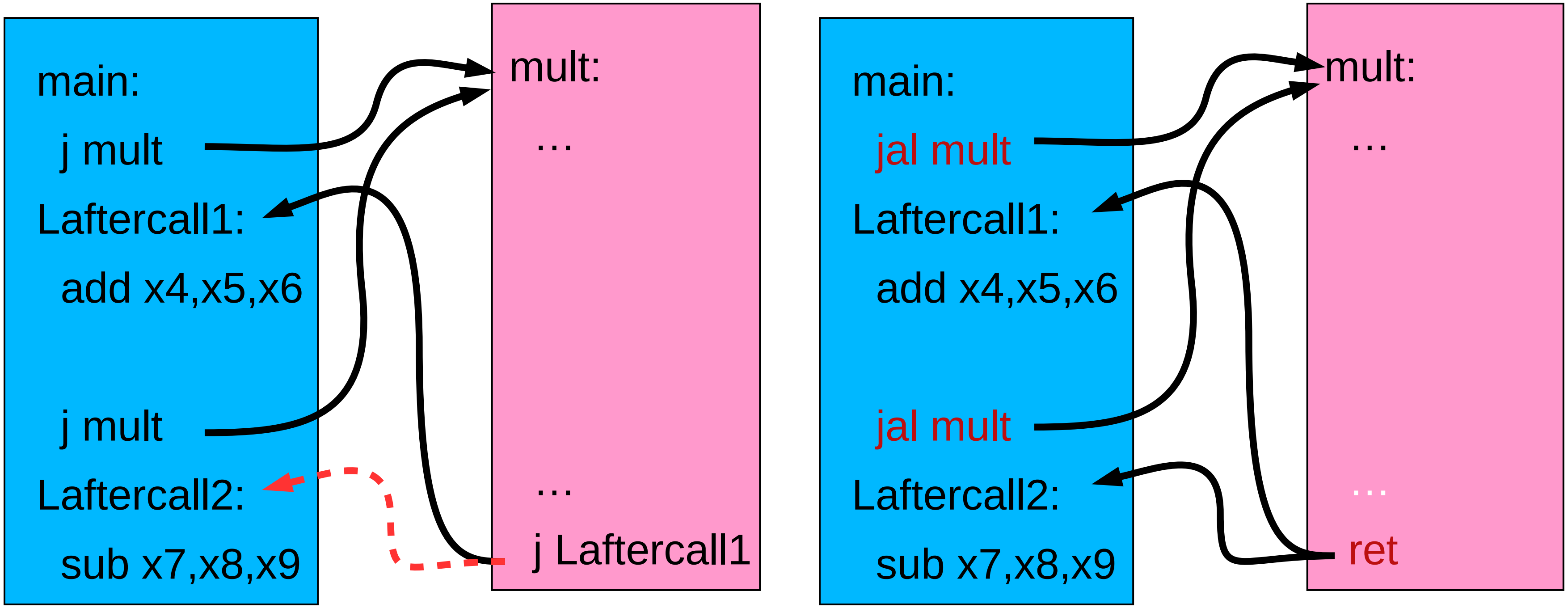
- a0 – a3: arguments (registers $4 – $7)
- v0, v1: registers to hold function result value ($2 and $3)
- t0 – t9: temporary/clobberable registers ($8-$15,$24,$25)
  - callee function can use them as it needs without saving
- at: temporary register for complex assembly constructs ($1)
- k0, k1: reserved for operating system kernel ($26, $27)
- s0 – s7: saved (non-clobberable) registers ($16-$23)
  - if used by callee, they has to be saved first and restored at return
- gp: global static data pointer ($28)
- sp: stack pointer ($29) – top of the stack, grows down to 0
- fp/s8: frame pointer ($30) – points to start of local variables on stack
- ra: return address register ($31) – implicitly written by **jal** instruction – jump and link – call subroutine

RISC-V: Call Instruction and Return

- Subroutine invocation (calling): jump and link
  - **jal** ProcedureLabel
  - On RISC-V implemented as **jal** ra, ProcedureLabel
  - Address of the instruction following **jal** is stored to **ra** (x1) register
  - Target (subroutine start) address is filled to **pc**
- Return from register: jump register
  - **ret** on RISC-V implemented as **jalr** x0, 0(x1) same as **jr ra**
  - Loads **ra** register content to **pc**
  - The same instruction is used when jump target address is computed or chosen from table – i.e. case/switch statements in C source code

Jump and Subroutine Call Differences

Jump does not save return address
⇒ cannot be used to call subroutine from more locations



on the contrary, call using register **ra** allows you to call a subroutine from as many locations as needed

Jump and Link Instruction in Detail

C/C++

```
int main() {
  simple();
  …
}

void simple(){
  return;
}
```

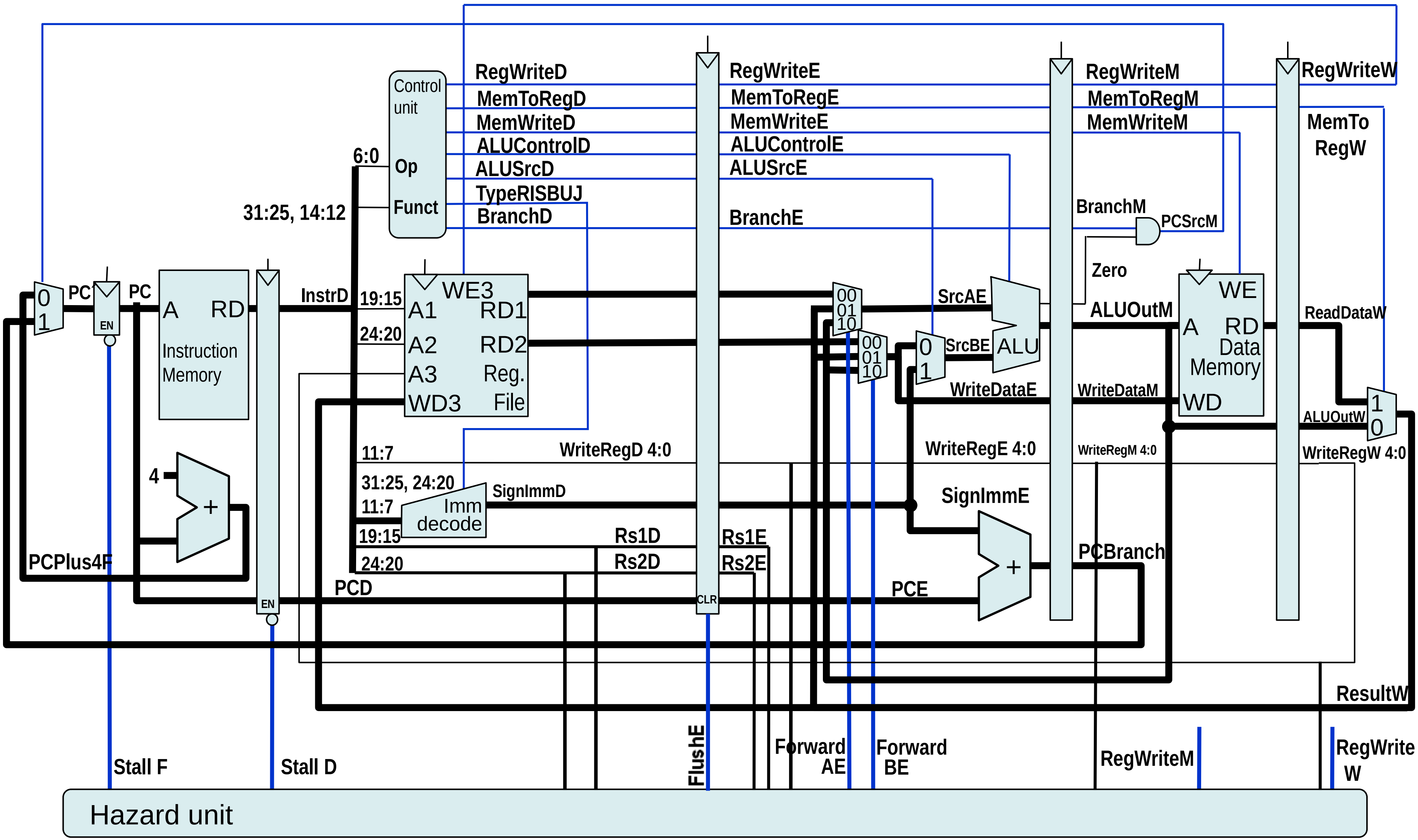
RISC-V

```
0x00400200 main: jal simple
0x00400204 …

0x00401020 simple: ret
```

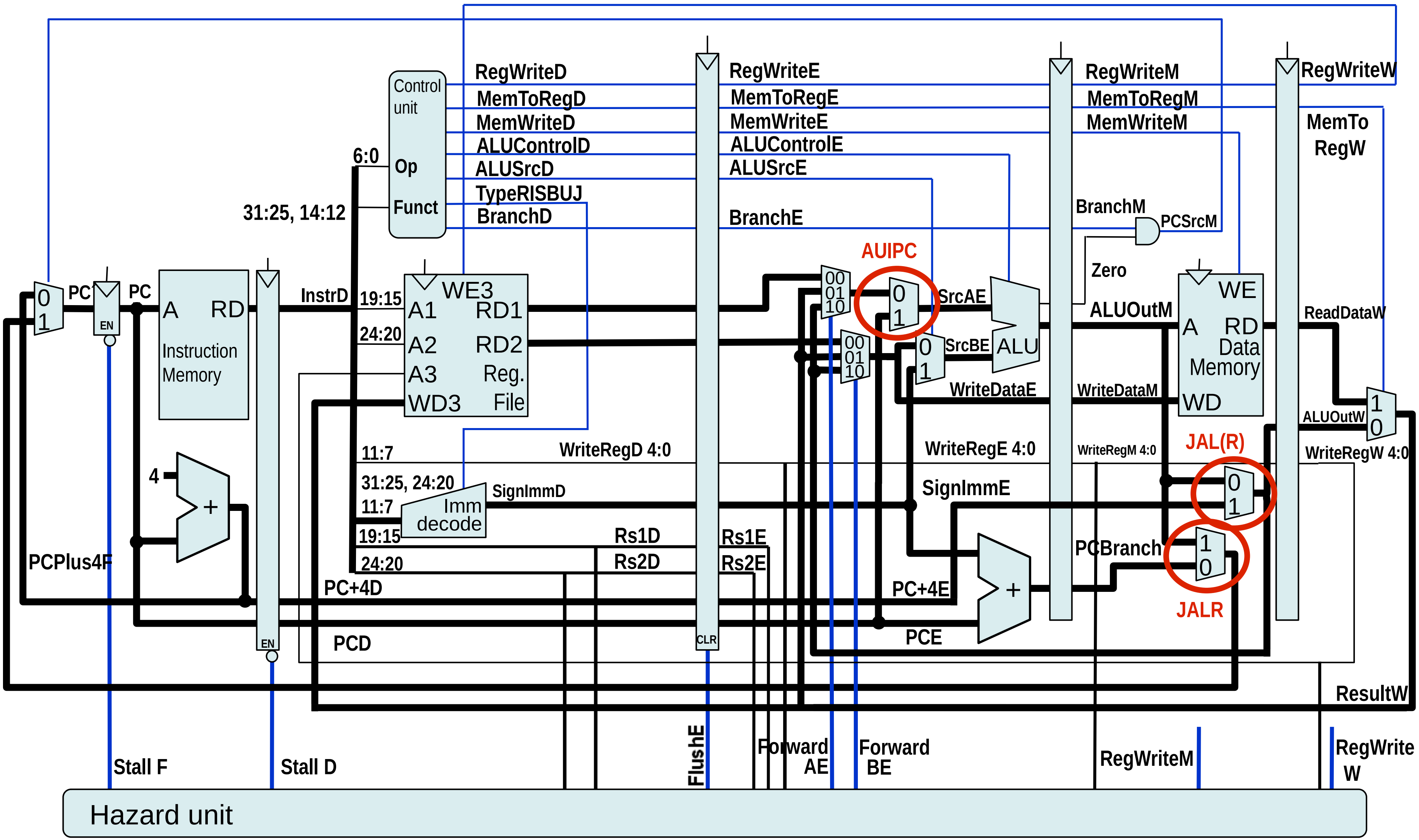
Description:	For procedure call.
Operation:	rd = PC + 4; PC = (PC+ target)
Syntax:	jal target
Encoding:	`iiii iiii iiii iiii iiii dddd d110 1111`

Caller/**`jal`** stores return address into rd → **`ra`** (reg x1) and transfers control to the callee/subroutine first instruction. Call returns by jump to return **ret** address (**`jr ra`**).

Pipelined Processor Designed in the Lecture 3 and 5

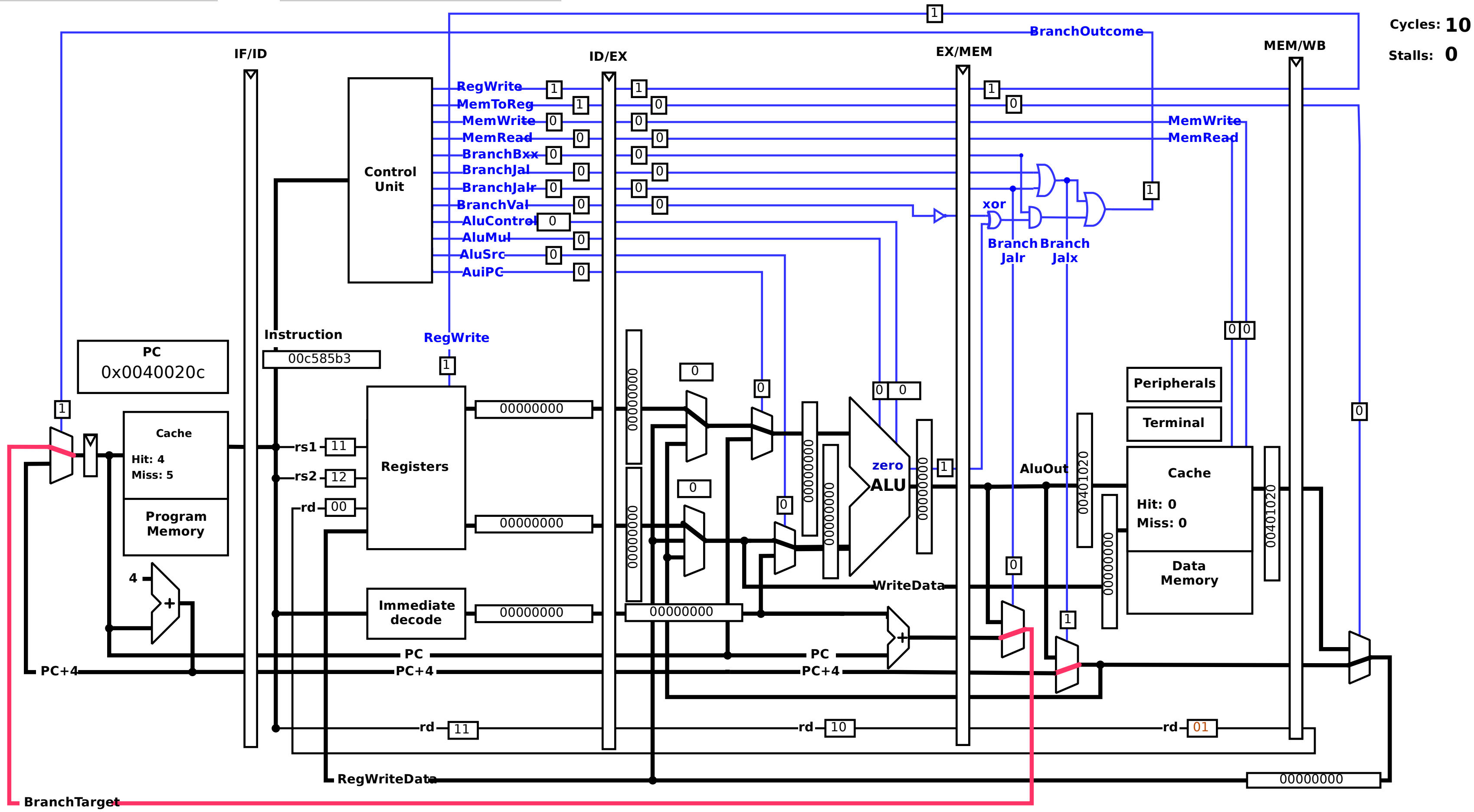


Pipelined Processor with JAL, JALR (RET, J, JR) Support

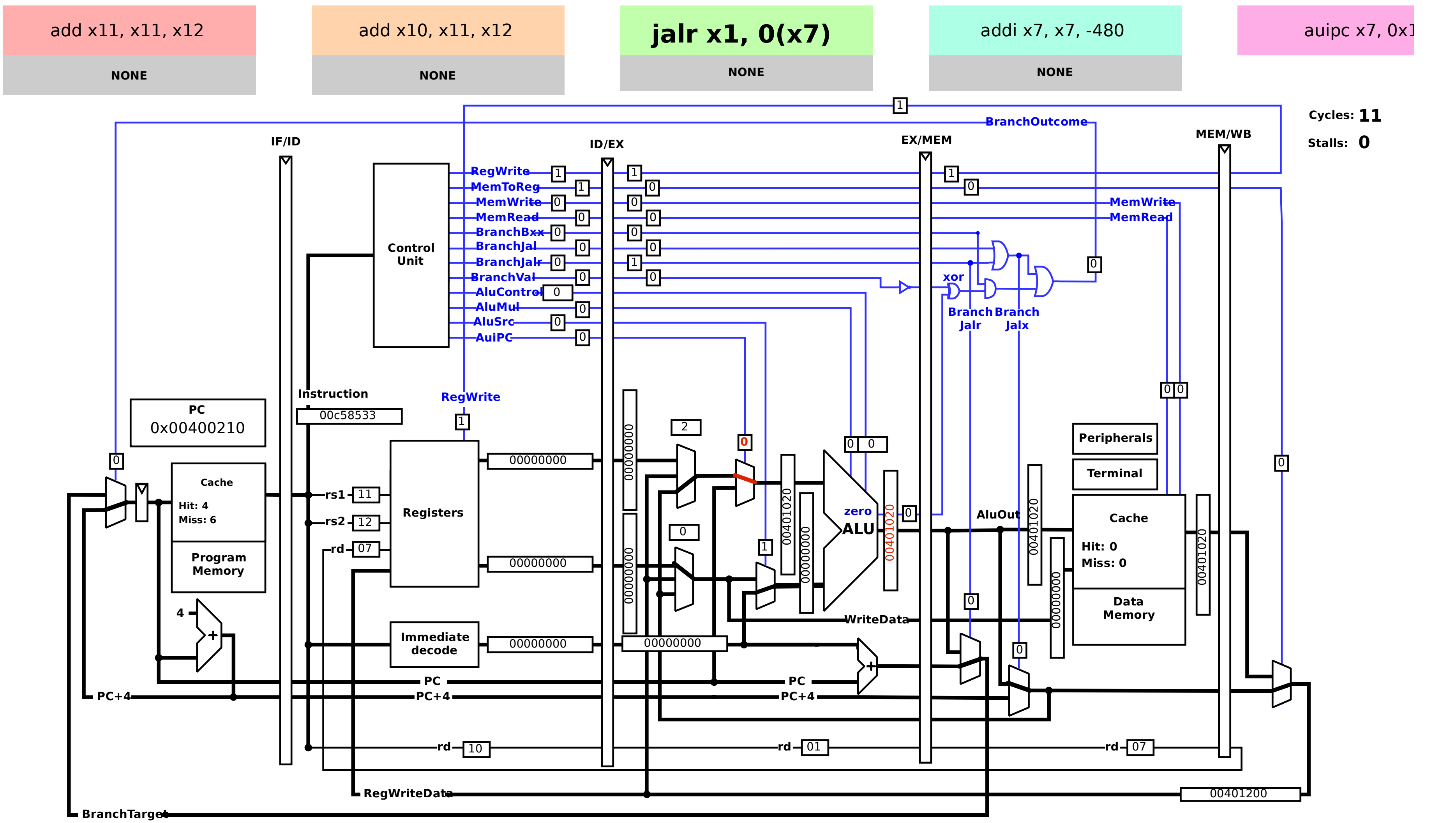


RISC-V JAL Instruction Execution

add x12, x11, x12	add x11, x11, x12	add x10, x11, x12	**jal x1, 0x401020**	nop
NONE	NONE	NONE	NONE	

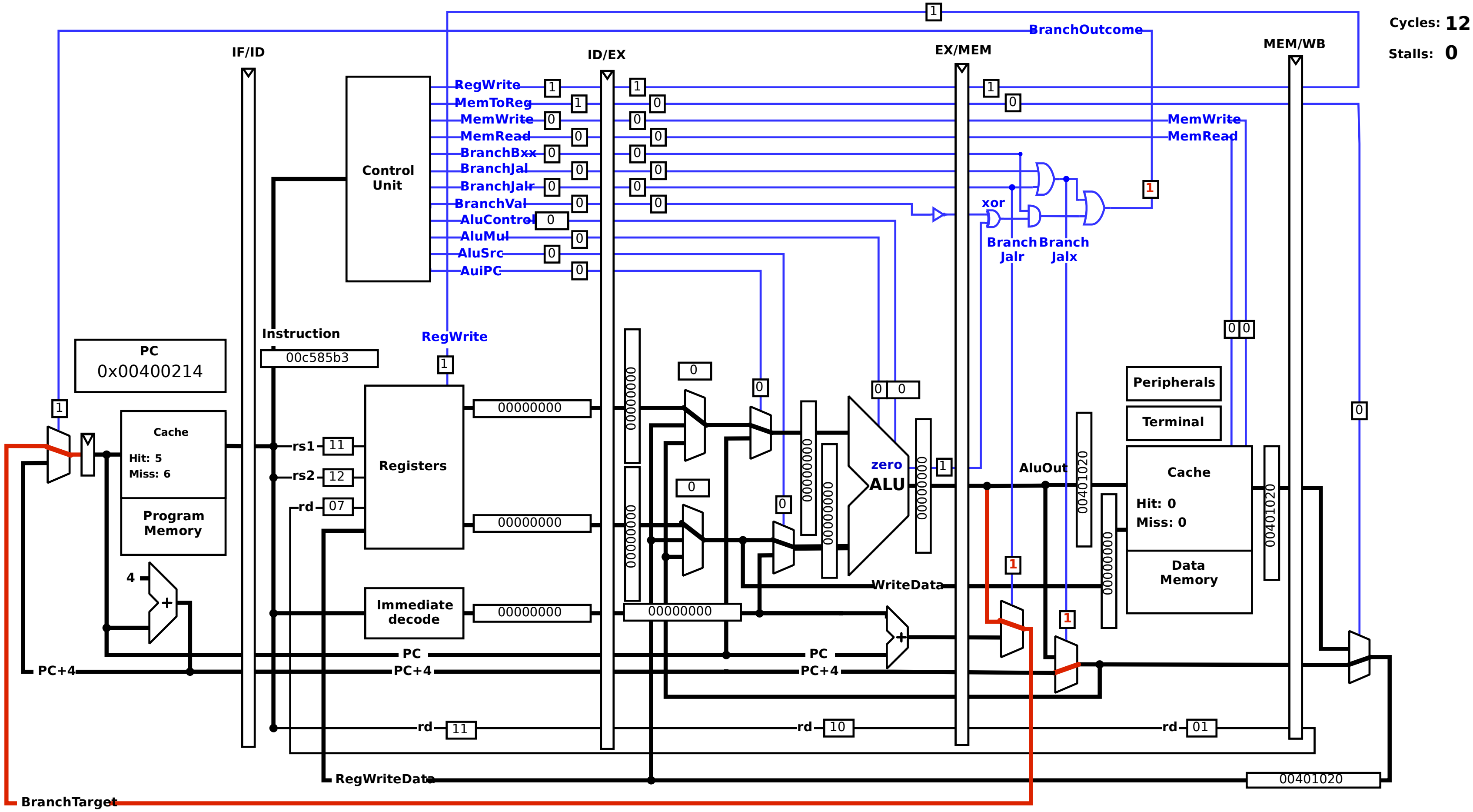


RISC-V JALR Target Address Computation in ALU



RISC-V JALR Instruction Execution, MEM Stage – Set PC

add x12, x11, x12	add x11, x11, x12	add x10, x11, x12	**Jalr x1, 0(x7)**	Addi x7, x7,
NONE	NONE	NONE	NONE	



Leaf Node Function Code Example

C language source code:

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

Parameters **g**, …, **j** are placed in registers **a0**, …, **a3**

f function result is computed in **s0** (**s0** non-clobberable and previous value has to be saved on stack)

Function return value is expected in **a0** register by caller

RISC-V: Caller and Callee with Parameters Passing

- a0-a7 available for arguments values, a0-a3 used
- a0-a1 function return value(s)

C/C++

```
int main() {
  int y;
  y=fun(2,3,4,5)
  …
}

int leaf_fun(int g,
   int h, int i, int j)
{
  int res;
  res = g + h – (i + j);
  return res;
}
```

RISC-V

```
main:
 addi a0, zero, 2
 addi a1, zero, 3
 addi a2, zero, 4
 addi a3, zero, 5
 jal fun
 add s0, a0, zero

fun:
 add t0, a0, a1
 add t1, a2, a3
 sub s0, t0, t1
 add a0, s0, zero
 ret   // jr ra
```

Clobbers caller guaranteed to be saved register s0

source: stud-support: seminaries/qtrvsim/call-syscall/lec10-01-call4-clobber.S

Example Code Compiled for RISC-V Architecture

int leaf_fun (int g, h, i, j)

g→a0, h→a1, i→a2, j→a3, a0 – ret. val, sX – save, tX – temp, ra – ret. addr

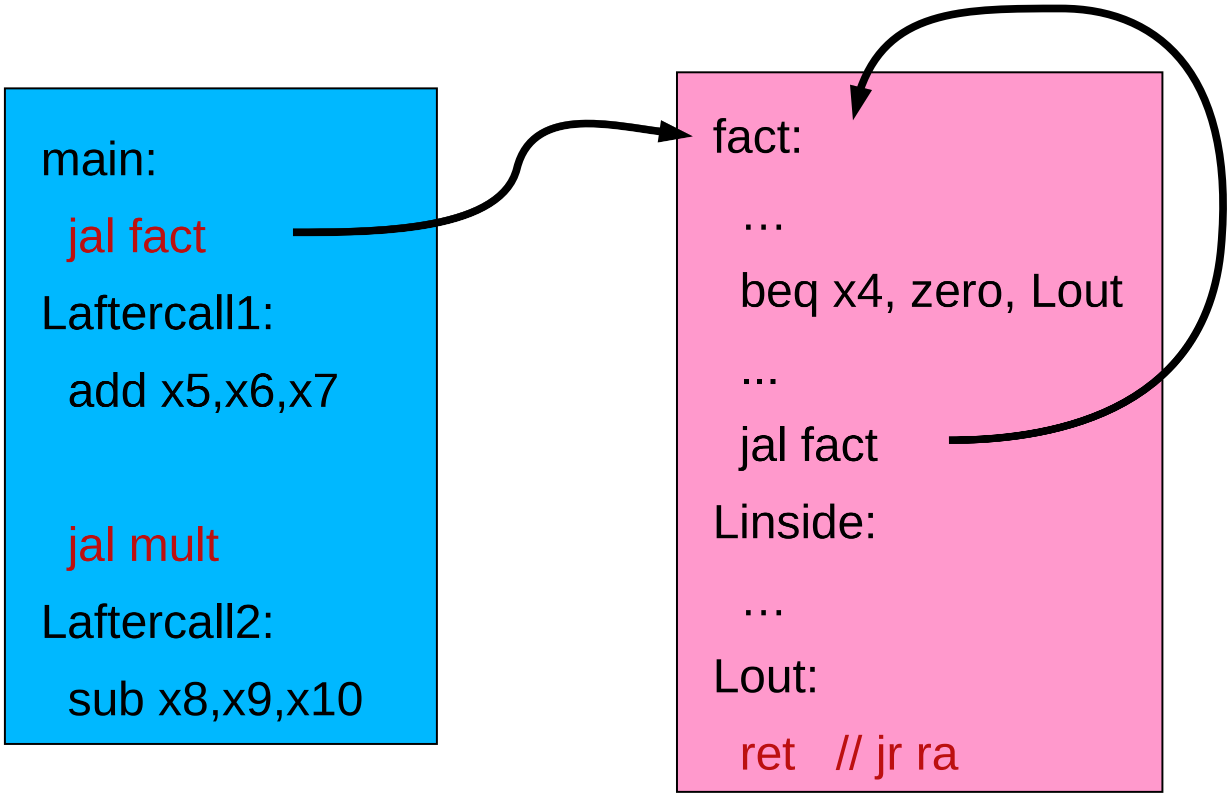
leaf_fun:	
addi sp, sp, -4 sw s0, 0(sp)	Save s0 on stack

add t0, a0, a1
add t1, a2, a3
sub s0, t0, t1
	Procedure body

add a0, s0, zero	Result
lw s0, 0(sp) addi sp, sp, 4	Restore s0
ret // jr ra	Return

source: stud-support: seminaries/qtrvsim/call-syscall/lec10-02-call4-save.S

Link-register and Nested or Recursive Calling Problem



The value of **ra** register has to be saved before another (nested or recursive) call same as for non-clobberable (saved) registers **s***X*

source: stud-support: seminaries/qtrvsim/call-syscall/lec10-03-fact-buggy.S

Recursive Function or Function Calling Another One

C language source code:

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

Function parameter **n** is located and passed in **a0** register

Function return value in **a0** register

RISC-V: Recursive Function Example

```
fact:
    addi   sp, sp, -8          // adjust stack for 2 items
    sw     ra, 4(sp)           // save return address
    sw     a0, 0(sp)           // save argument
    slti   t0, a0, 1           // test for n < 1
    beq    t0, zero, L1
    addi   a0, zero, 1         // if so, result is 1
    addi   sp, sp, 8           //   pop 2 items from stack
    ret                        //   and return (jr ra)
L1: addi   a0, a0, -1          // else decrement n
    jal    fact                // recursive call
    lw     t0, 0(sp)           // restore original n
    lw     ra, 4(sp)           //   and return address
    addi   sp, sp, 8           // pop 2 items from stack
    mul    a0, a0, t0          // multiply to get result
    ret                        // and return (jr ra)
```

Listing before filling/moving instructions into delay-slots

source: stud-support: seminaries/qtrvsim/call-syscall/lec10-04-fact-ok.S

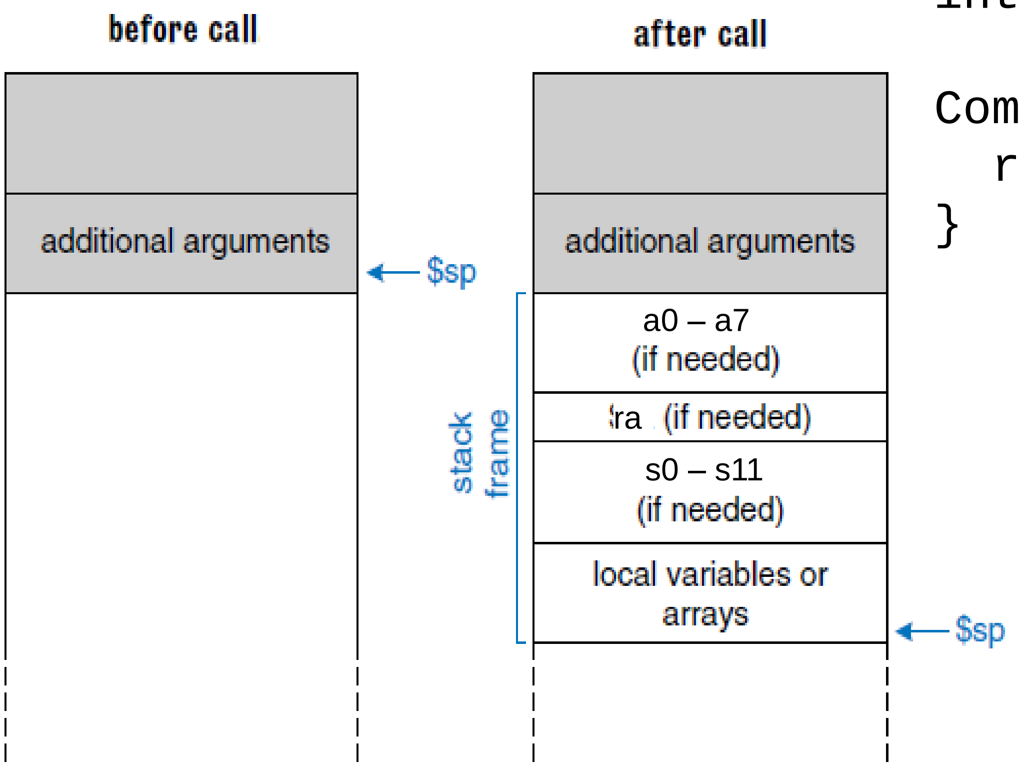
RISC-V Calling Convention and ABI

- Application binary interface (ABI) for given architecture
- Defines which registers needs to be saved by callee
- Specifies how arguments are passed etc.

Preserved by callee (Non-clobberable)	Nonpreserved (clobberable)
Callee-saved	Maintained/saved by caller
Saved registers: s0/fp, s1 … s11	Temporary registers: t0 … t6
Return address: ra	Argument registers: a0 … a7
Stack pointer: sp	Return value registers: a0 … a1
Stack above the stack pointer	Stack bellow the stack pointer

RISC-V: Subroutine with More than 8 Arguments

- RISC-V calling convention: The eight four in **a0** … **a7**, other placed on stack such way that fifth argument is found directly at **sp** and next at **sp**+4. Space is allocated and deallocated and released by caller.




```
int main(){

Complex(1,2,3,4,5,6,7,8,9,10);
  return 0;
}
```

```
addiu  sp,sp,-32
addi   t0,0,10    // 0x10
sw     t0,20(sp)
addi   t0,0,9     // 0x9
Sw     t0,16(sp)
addi   a0,zero,1 // 0x1
addi   a1,zero,2 // 0x2
...
addi   a7,zero,8 // 0x8
jal     complex
addiu  sp,sp,32
```


source: stud-support: seminaries/qtrvsim/call-syscall/lec10-05-call-10args.S

C Calling Convention for x86 – 32-bit Mode Registers

General-Purpose Registers

	31 … 16	15 … 8	7 … 0	16-bit	32-bit
Accumulator		AH	AL	AX	EAX
Base		BH	BL	BX	EBX
Count		CH	CL	CX	ECX
Data		DH	DL	DX	EDX
Source Index		SI			ESI
Destination Index		DI			EDI
Base Pointer		BP			EBP
Stack Pointer		SP			ESP

31 … 0	
Program Status and Control Register	EFLAGS
Instruction Pointer	EIP

Segment Registers

15 … 0	
Code Segment	CS
Data Segment	DS
Stack Segment	SS
Extra Segment	ES
	FS
	GS

x86 32-bit Mode Intel and GNU Assembler Syntax

GAS AT&T syntax	INTEL/MASM
movb $8, %al	mov al, 8
addw $0x1234, %bx	add bx, 1234h
subl (%ecx), %edx	sub edx, [ecx]
memory operands	memory operands
100	[100]
%es:100	[es:100]
(%eax)	[eax]
(%eax,%ebx)	[eax+ebx]
(%ecx,%ebx,2)	[ecx+ebx*2]
(,%ebx,2)	[ebx*2]
-10(%eax)	[eax-10]
%ds:-10(%ebp)	[ds:ebp-10]

x86 32-bit Mode Intel and GNU Assembler Syntax

GAS AT&T

```
# Full example: load *(ebp + (edx * 4) - 8) into eax
movl    -8(%ebp, %edx, 4), %eax

 # Typical example: load a stack variable into eax
movl    -4(%ebp), %eax

# No index: copy the target of a pointer into a register
movl    (%ecx), %edx

# Arithmetic: multiply eax by 4 and add 8
leal    8(,%eax,4), %eax

# Arithmetic: multiply eax by 2 and add edx
leal    (%edx,%eax,2), %eax
```

Standard C Calling Convention for x86 in 32-bit Mode

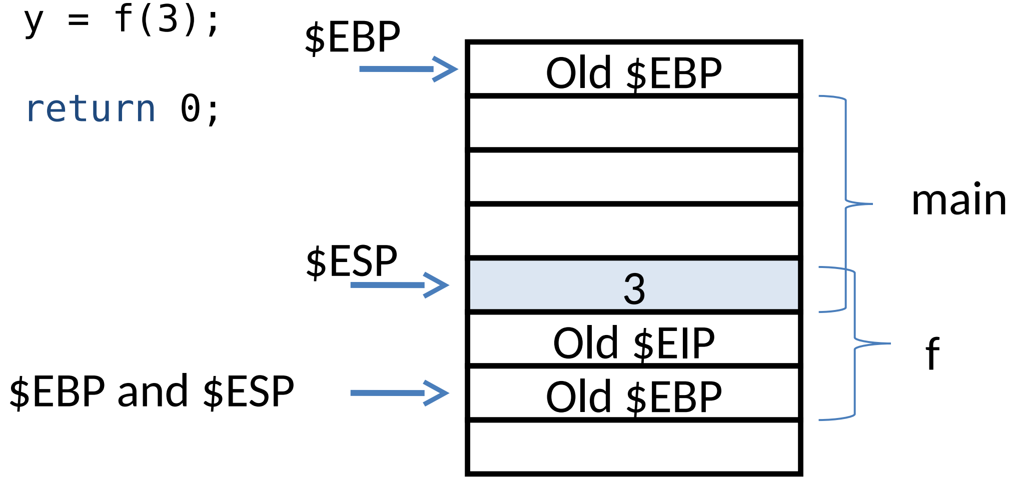
- Registers $EAX, $ECX and $EDX are clobberable/can be modified by subroutine without saving
- Registers $ESI, $EDI, $EBX values has to be preserved
- Registers $EBP, $ESP have predefined use, sometimes even $EBX use can be special (dedicated for GOT access)
- Three registers are usually not sufficient even for local variables of leaf-node functions
- Function result is expected in $EAX register; for 64-bit return values $EDX holds MSB part of the result
- Everything other – all parameters, local variables atc. have to be placed on stack

SYSTEM V APPLICATION BINARY INTERFACE
Intel386™ ArchitectureProcessor Supplement

x86: Calling Function with one Parameter

```c
#include <stdio.h>

int f(int x)
{
        return x;
}

int main()
{
        int y;

        y = f(3);

        return 0;
}
```

```
f:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        leave   ; same as
                ; mov %ebp, %esp
                ; pop %ebp
        ret

main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $3, (%esp)
        call    f
        movl    %eax, -4(%ebp)
        movl    $0, %eax
        leave
        ret
```



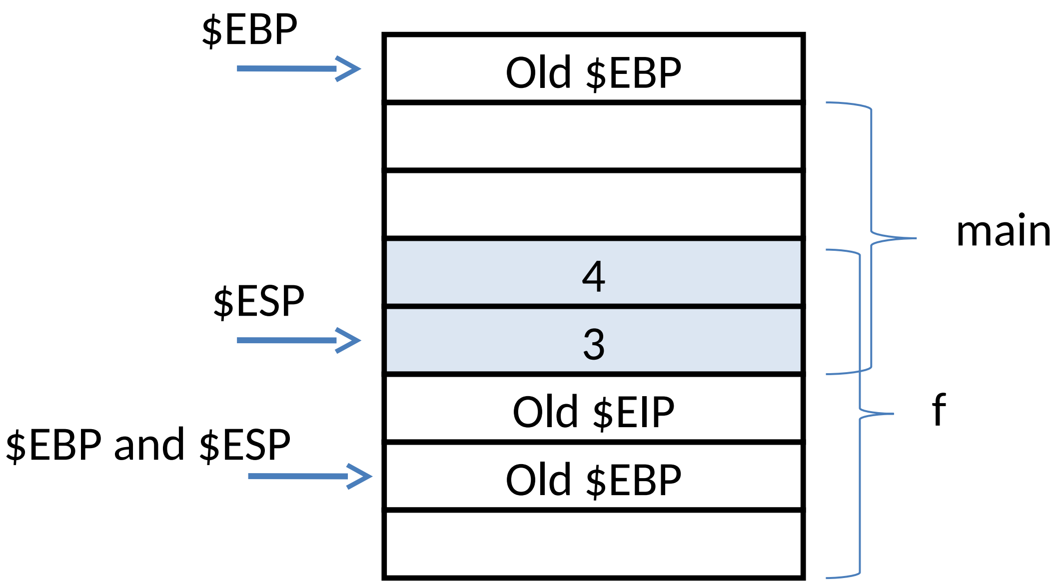
Examples found in the work of David Kyle a Jonathan Misurda

x86: Calling Function with two Parameters

```c
#include <stdio.h>

int f(int x, int y)
{
  return x+y;
}
```

```c
int main()
{
  int y;
  y = f(3, 4);
  return 0;
}
```

```asm
f:      pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        leave
        ret
main:   pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $4, 4(%esp)
        movl    $3, (%esp)
        call    f
        movl    %eax, 4(%esp)
        movl    $0, %eax
        leave
        ret
```

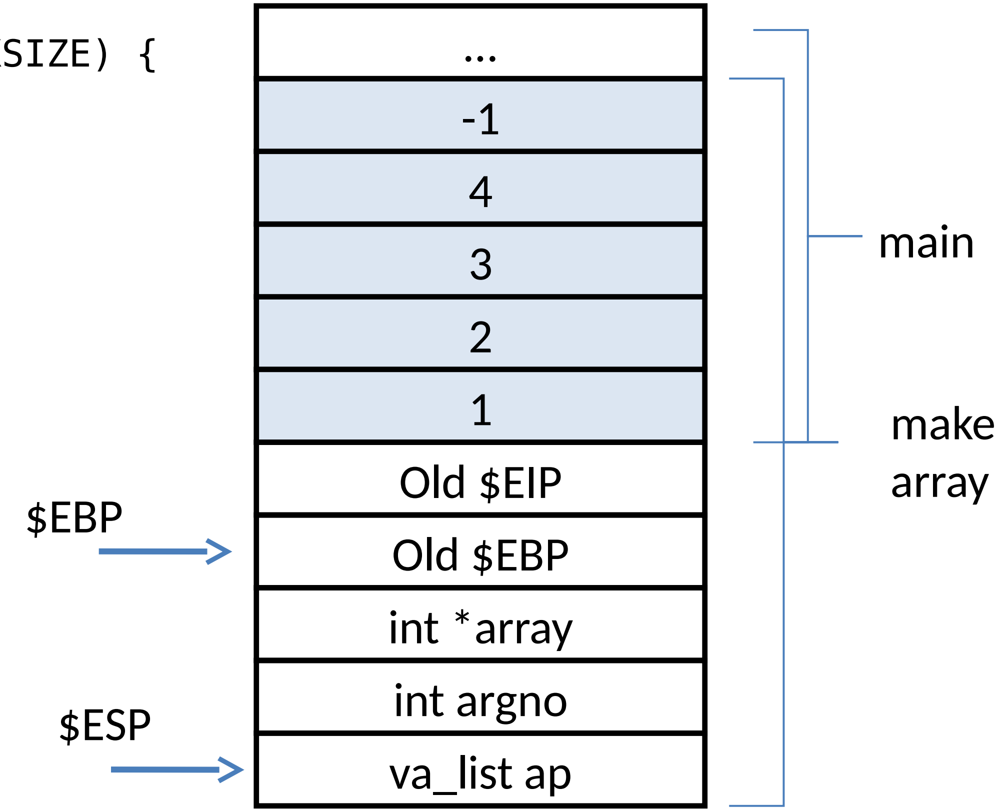


Variable Parameters Count - stdarg.h

```
int *makearray(int a, ...) {
   va_list ap;
   int *array = (int *)malloc(MAXSIZE * sizeof(int));
   int argno = 0;
   va_start(ap, a);
   while (a > 0 && argno < MAXSIZE) {
        array[argno++] = a;
        a = va_arg(ap, int);
   }
   array[argno] = -1;
   va_end(ap);
   return array;
}
```

va_list
va_start, va_arg,
va_end, va_copy

```
Call the variadic function
  int *p;
  int i;
  p = makearray(1,2,3,4,-1);
  for(i=0;i<5;i++)
    printf("%d\n", p[i]);
```



Variable Parameters Count, Multiple Iterations

```
int *makearray(int a, ...) {
	va_list ap, ap1;
	int tmp;
	size_t count = 1;
	int argno = 0;
	va_start(ap, a);
	va_copy(ap1, ap);
	tmp = a;
	while (tmp > 0) {
		tmp = va_arg(ap1, int);
		count++;
	}
	int *array = (int *)malloc(count * sizeof(int));
	while (argno < count) {
		array[argno++] = a;
		a = va_arg(ap, int);
	}
	array[argno] = -1;
	va_end(ap);
	return array;
}
```

```
#include <stdlib.h>
#include <stdarg.h>
```

va_list
va_start, va_arg,
va_end, va_copy

Functions printf and vprintf

```
#include <stdio.h>
#include <stdarg.h>

int vprintf ( const char * format, va_list arg ) {
    for arguments in format switch
      case 'd':  int val_i = va_arg(arg, int); break;
      case 'f':  float val_f = va_arg(arg, float); break;
}

void printf( const char * format, ... ) {
   va_list args;
   va_start (args, format);
   vprintf (format, args);
   va_end (args);
}

int main () {
   printf ("Call with %d variable argument.\n",1);
   return 0;
}
```

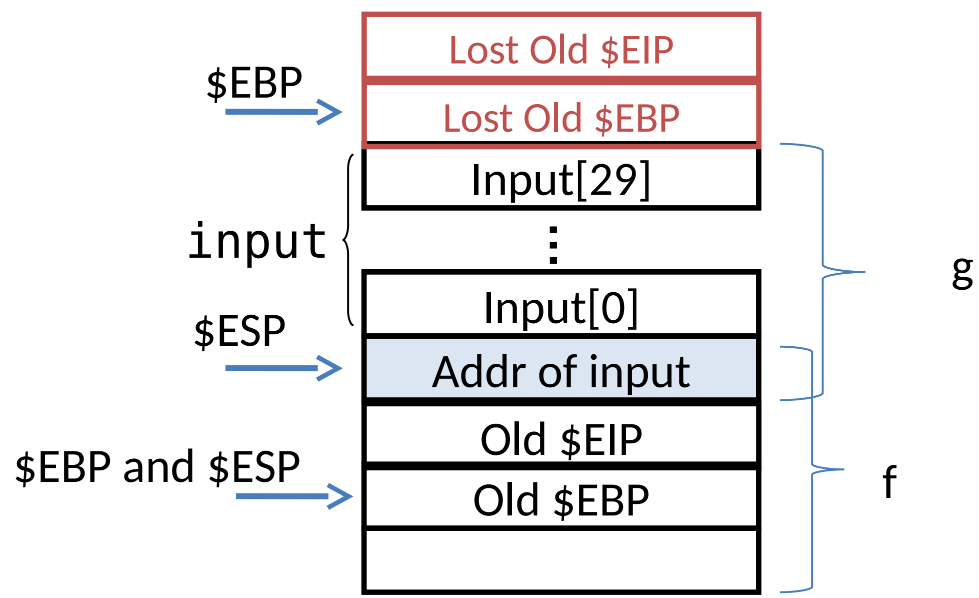
```
#include <stdlib.h>
#include <stdarg.h>

va_list
va_start, va_arg,
va_end, va_copy
```

Stack/buffer Overflow Error Example

```
void f(char *s)
{
  gets(s);
}

int g()
{
  char input[30];
  f(input);
}
```

```
g:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $40, %esp
    andl    $-16, %esp
    subl    $16, %esp
    leal    -40(%ebp), %eax
    movl    %eax, (%esp)
    call    f
    leave
    ret
```

Real one CVE-2022-26809
Critical Remote Code Execution Vulnerabilities in Windows RPC Runtime
Ben Barnea and Ophir Harpaz
April 13, 2022 URL
700,000 machines on public Internet with port open



x86: Example with More Arguments

```
int simple(int a, int b, int c, int d, int e, int f){
  return a-f;
}
int main(){
  int x;
  x=simple(1, 2, 3, 4, 5, 6);
  return 0;
}
```

```
_simple:
pushl %ebp
movl  %esp, %ebp
movl  28(%ebp),%eax
movl  8(%ebp),%edx
movl  %edx, %ecx
subl  %eax, %ecx
movl  %ecx, %eax
popl  %ebp
ret
```

```
_main:
pushl %ebp            ebp saved on stack, push modifies esp
movl  %esp, %ebp      esp to ebp
andl  $-16, %esp      allign stack to 16-bytes
subl  $48, %esp       esp = esp – 48 (allocate space)
call  ___main
movl  $6, 20(%esp)    the last argument
movl  $5, 16(%esp)    the fifth argument
movl  $4, 12(%esp)    ...
movl  $3, 8(%esp)     ...
movl  $2, 4(%esp)     ...
movl  $1, 0(%esp)     the first argument
call  _simple         call the function
movl  %eax, 44(%esp)  store result to global x = simple(…);
movl  $0, %eax        return 0;
leave
ret
```

x86 Calling Convention for 64-bit Mode – Registers

Not modified for 8-bit operands (bits 63–8)

Not modified for 16-bit operands (bits 63–16)

Zero-extended for 32-bit operands (bits 63–32)

Register encoding	63 … 32	31 … 16	15 … 8	Low 8-bit (7 … 0)	16-bit	32-bit	64-bit
0			AH†	AL	AX	EAX	RAX
3			BH†	BL	BX	EBX	RBX
1			CH†	CL	CX	ECX	RCX
2			DH†	DL	DX	EDX	RDX
6				SIL‡	SI	ESI	RSI
7				DIL‡	DI	EDI	RDI
5				BPL‡	BP	EBP	RBP
4				SPL‡	SP	ESP	RSP
8				R8B	R8W	R8D	R8
9				R9B	R9W	R9D	R9
10				R10B	R10W	R10D	R10
11				R11B	R11W	R11D	R11
12				R12B	R12W	R12D	R12
13				R13B	R13W	R13D	R13
14				R14B	R14W	R14D	R14
15				R15B	R15W	R15D	R15

† Not legal with REX prefix ‡ Requires REX prefix

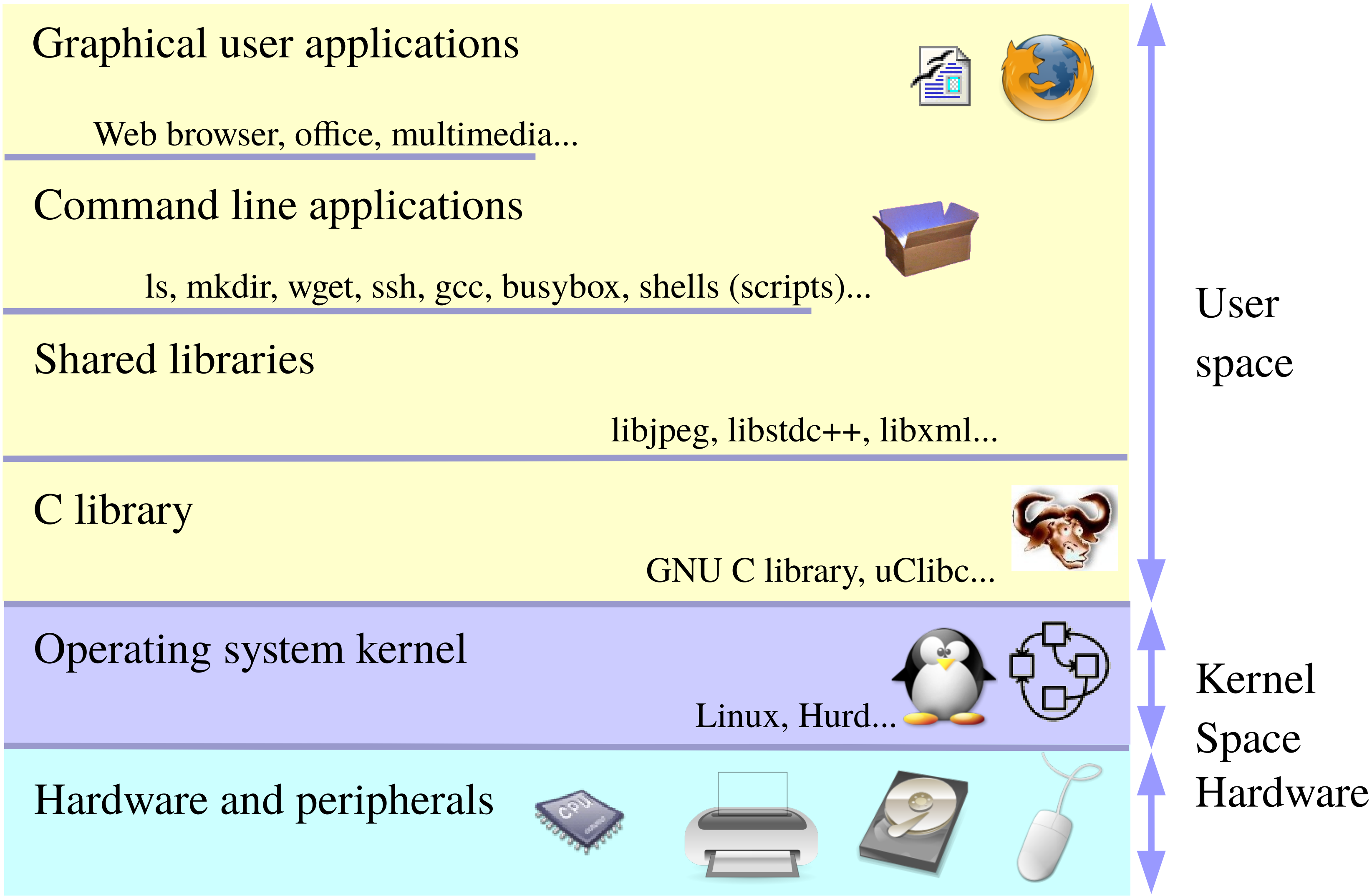
RIP – program counter

Source: Yasm manual https://www.tortall.net/projects/yasm/manual/manual.pdf

x86 Calling Convention for 64-bit Mode (for Linux)

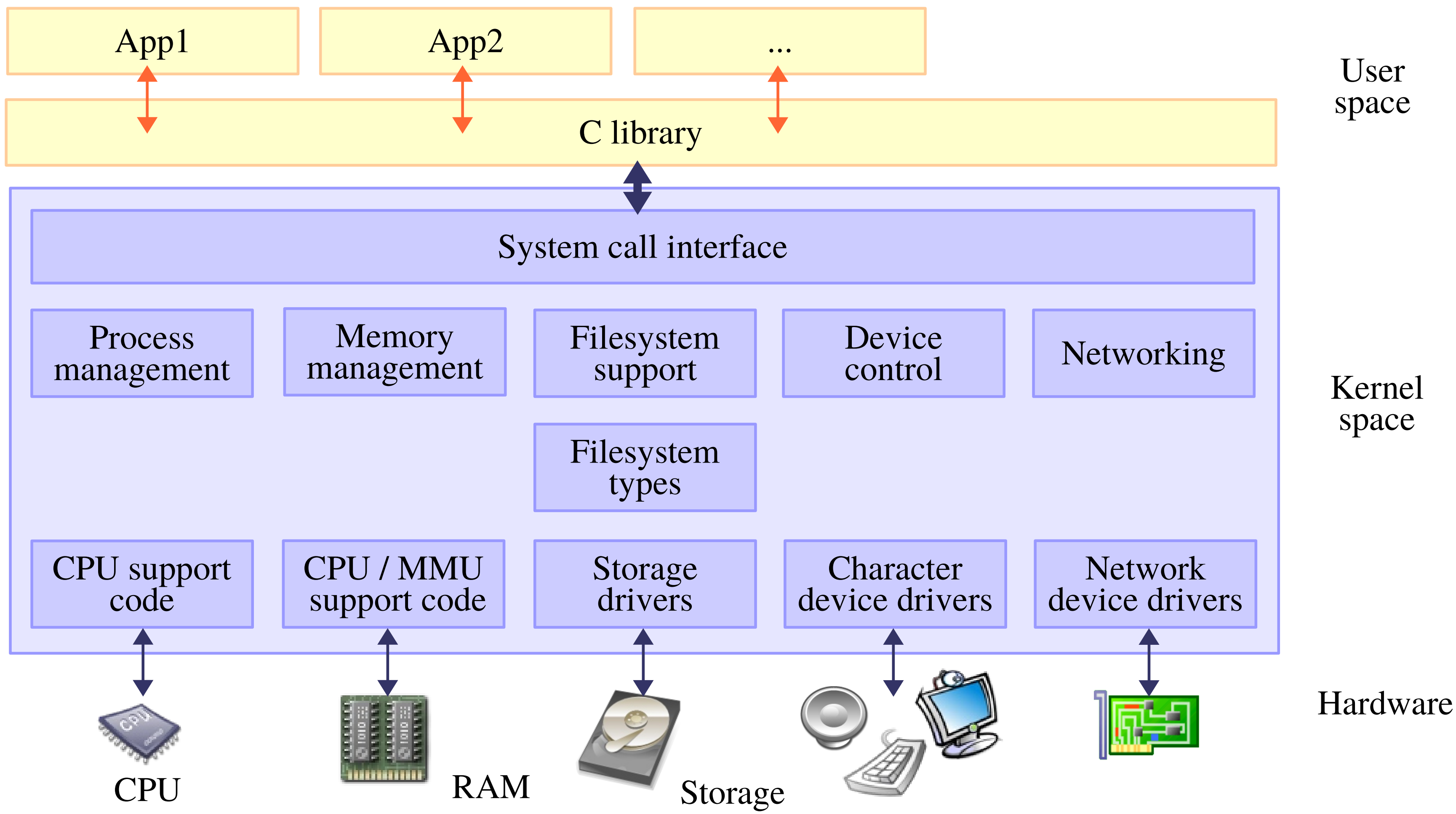
- Original 32-bit calling convention is expensive, too many main memory (stack) accesses
- 64-bit registers $RAX, $RBX, $RCX, $RDX, $RSI, $RDI, $RBP, $RSP, $R8 … R15 and many multimedia registers
- AMD64 ABI/calling convention places up to the first 6 integral data type parameters in $RDI, $RSI, $RDX, $RCX, $R8 and $R9 registers
- The first 8 double and float data type parameters in XMM0-7
- Return/function result value in $RAX
- Stack is always 16-byte aligned when a call instruction is executed
- If calling function without prototype or function with variable arguments count (va_arg/...) then $RAX has to be set to number of parameters passed in the vector registers (never use va_arg twice without va_copy)

Operating System – Layers



Source: https://bootlin.com/ (Michael Opdenacker)

Operating System Structure



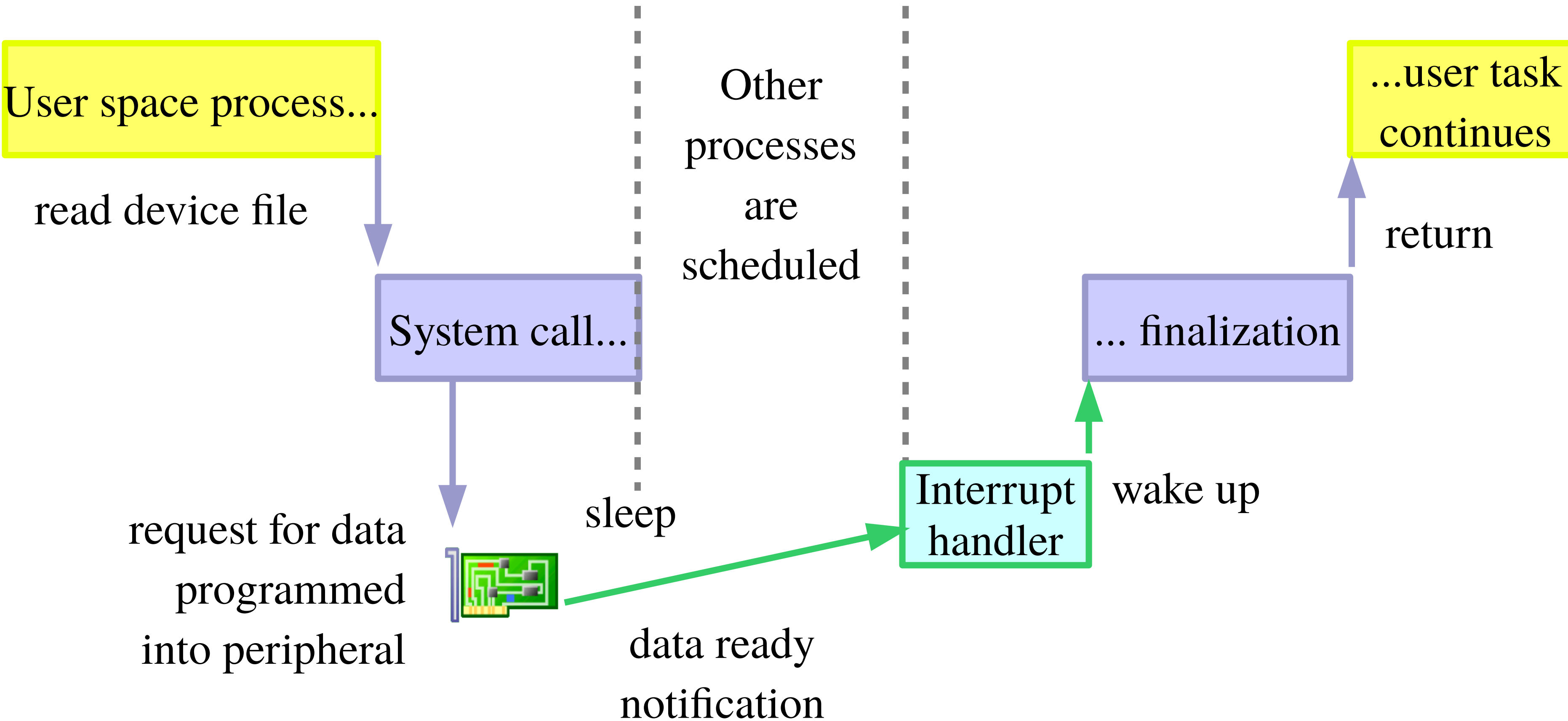
Source: https://bootlin.com/ (Michael Opdenacker)

System Calls

- The main interface between the operating system kernel and user space is the set of system calls
- On Linux, about 400 system calls that provide the main kernel services
- File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- This interface is stable over time: only new system calls can be added by the kernel developers
- This system call interface is wrapped by the C library, and user space applications usually never make a system call directly but rather use the corresponding C library function

System Call and Device with Interrupt handler

When peripheral transfers data, task is suspended/waiting (and other work could be done by CPU). Data arrival results in IRQ processing, CPU finalizes transfer and original task continues

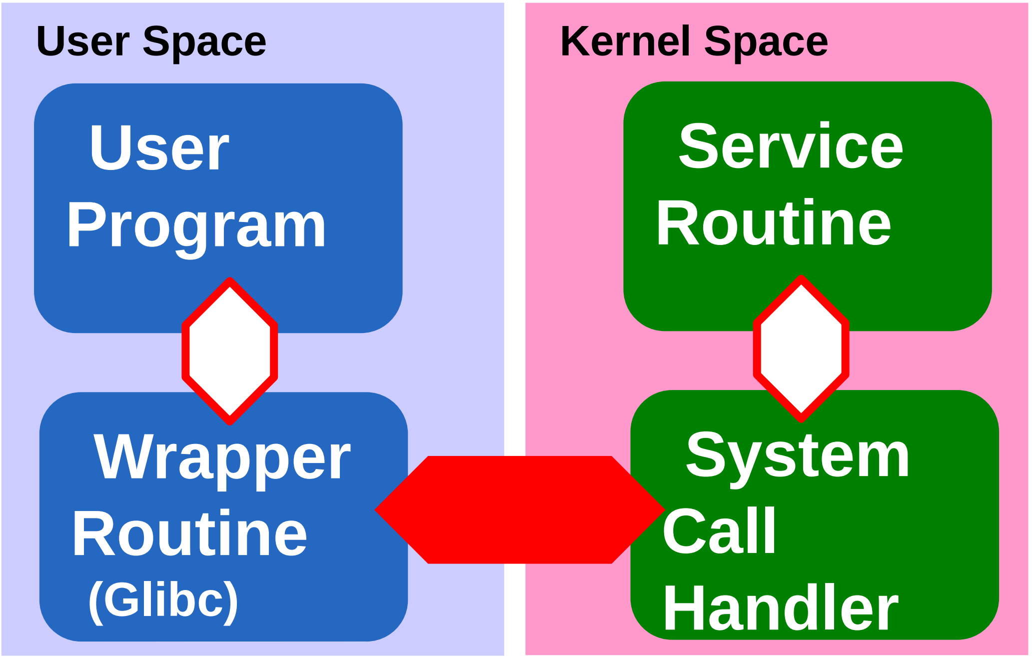


Source: https://bootlin.com/ (Michael Opdenacker)

System Call Processing Steps

- Operating system services (i.e. open, close, read, write, ioctl, mmap) are wrapped by common runtime libraries (GLIBC, CRT atd.) and parameters are passed to wrappers same way as to the usual functions/subroutines
- Library in the most cases moves parameters values into registers specified by given system call ABI (differs from function calls ABI)
- Service identification / syscall number is placed in defined register (EAX for x86 architecture)
- Syscall exception/interrupt instruction is executed (int 0x80 or sysenter for x86 Linux)
- Syscall entry handler decodes parameters according to syscall number and calls system service function usual function ABI way
- Kernel return code is placed to one of the registers and return from exception/switch to user mode follow
- Library wrapper does error processing (sets errno) and regular function return passes execution back to calling program

System Calls – Wrapper and System Service Routine



Parameters Placement for Selected Syscalls (Linux i386)

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int				
3	sys_read	fs/read_write.c	unsigned int	char *	size_t		
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t		
5	sys_open	fs/open.c	const char *	int	mode_t		
6	sys_close	fs/open.c	int				
15	sys_chmod	fs/open.c	const char *	mode_t			
20	sys_getpid	kernel/timer.c	void				
21	sys_mount	fs/namespace.c	char *	char *	char *	unsigned long	void *
88	sys_reboot	kernel/sys.c	int	int	unsigned int	void *	

System Call Name

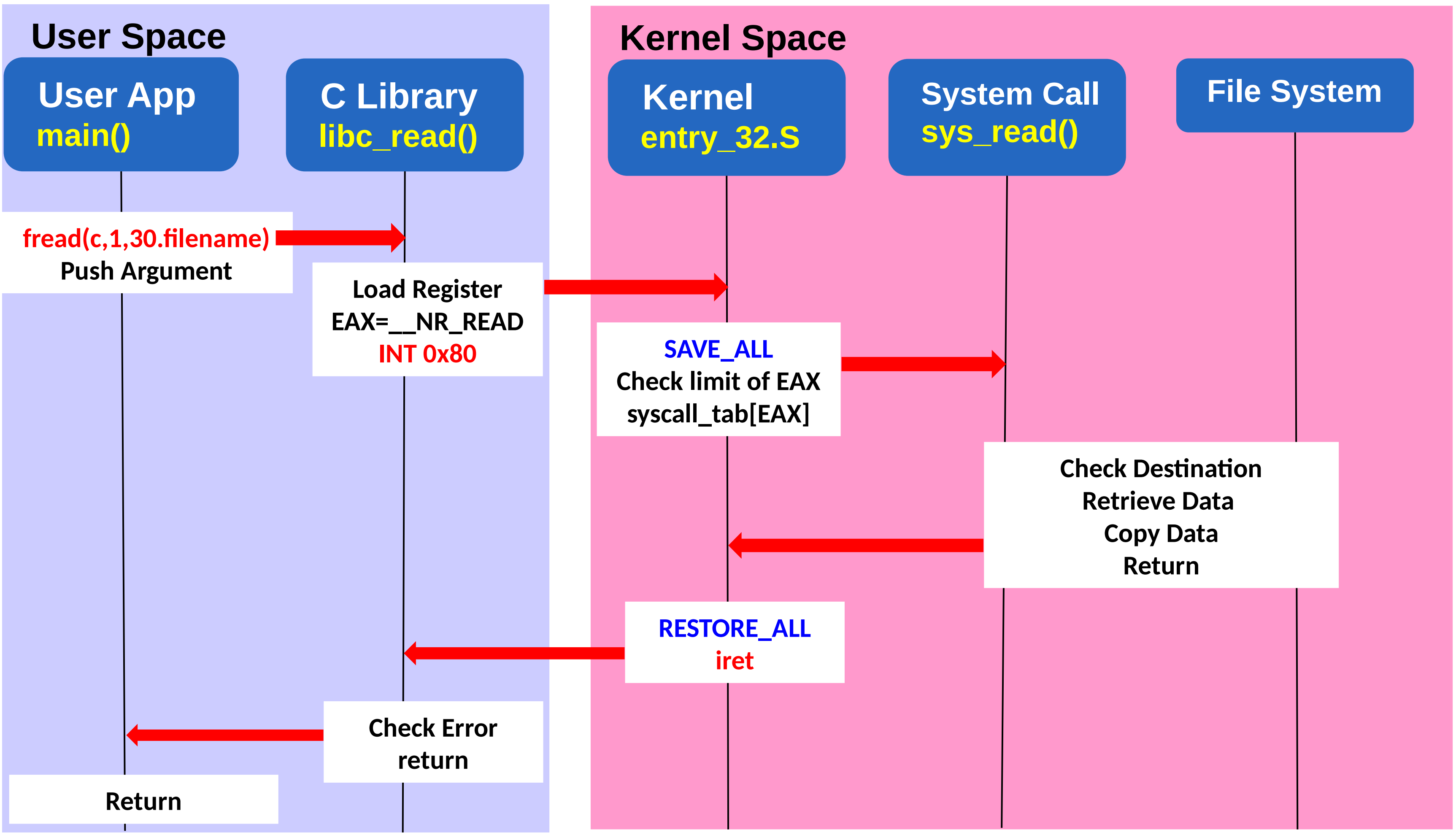
First parameter

Second parameter

Third parameter

System Call Number

System Call Processing Steps



System Call and Parameters Placement for RISC-V

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5	t0	Temporary/alternate link register	Caller
x6–7	t1– 2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
pc	pc	Program Counter	
f0–31		Floating point	
		Machine control and status	

Linux Hello World application on RISC-V

```
.globl _start
.globl __start
.option norelax
.text

_start:
        addi  s0, zero, 1

open:                   // optional
//      fd = open("/dev/tty1", O_RDWR, 0);
        addi  a7, zero, __NR_open
        addi  a0, zero, file_name
        addi  a1, zero, O_RDWR
        ecall
        add   s0, a0, zero

#       write(fd, "hello, world.\n", 14);
        addi  a7, zero, __NR_write
        addi  a0, s0, zero
        addi  a1, zero, text_1
        addi  a2, zero, text_1_e - text_1
        ecall
```

```
close:                  // optional
//      close(fd);
        addi  a7, zero, __NR_close
        addi  a0, s0, zero
        ecall
final:
//      exit(0);
        addi  a7, zero, __NR_exit
        addi  a0, zero, 0
        ecall

        ebreak
        jal   zero, final

.data

// store ASCII text, no termination
text_1:     .ascii "Hello world.\n"
text_1_e:

file_name: .asciz “/dev/tty0”
```


source: stud-support: seminaries/qtrvsim/call-syscall/lec10-06-linus-hello.S

System Call and Parameters Placement for MIPS Architecture

Register	use on input	use on output	Note
$at	—	(caller saved)	
$v0	syscall number	return value	
$v1	—	2nd fd only for pipe(2)	
$a0 ... $a2	syscall arguments	returned unmodified	O32
$a0 ... $a2, $a4 ... $a7	syscall arguments	returned unmodified	N32 and 64
$a3	4th syscall argument	$a3 set to 0/1 for success/error	
$t0 ... $t9	—	(caller saved)	
$s0 ... $s8	—	(callee saved)	
$hi, $lo	—	(caller saved)	

Actual system call invocation is realized by **SYSCALL** instruction, the numbers are defined in **http://lxr.linux.no/#linux+v3.8.8/arch/mips/include/uapi/asm/unistd.h**

Source: http://www.linux-mips.org/wiki/Syscall

Hello World – the First Linux Program Ever Run on MIPS

```
#include <asm/unistd.h>
#include <asm/asm.h>
#include <sys/syscall.h>

#define O_RDWR              02
        .set    noreorder
        LEAF(main)
#        fd = open("/dev/tty1", O_RDWR, 0);
        la      a0,tty
        li      a1,O_RDWR
        li      a2,0
        li      v0,SYS_open
        syscall
        bnez    a3,quit
        move    s0,v0                   # delay slot
#        write(fd, "hello, world.\n", 14);
        move    a0,s0
        la      a1,hello
        li      a2,14
        li      v0,SYS_write
        syscall
```

```
#        close(fd);
        move    a0,s0
        li      v0,SYS_close
        syscall

quit:
        li      a0,0
        li      v0,SYS_exit
        syscall

        j       quit
        nop

        END(main)

        .data
tty:    .asciz  "/dev/tty1"
hello:  .ascii  "Hello, world.\n"
```


source: stud-support: seminaries/qtmips/call-syscall/lec10-06-linus-hello.S

Materials and References

- B35APO – Computer Architectures
  https://cw.fel.cvut.cz/wiki/courses/b35apo
- B35APO – The Fourth Homework - Code Analysis
  https://cw.fel.cvut.cz/wiki/courses/b35apo/en/homeworks/04/start
- B35APO – students support repository
  https://gitlab.fel.cvut.cz/b35apo/stud-support
- System V ABI @ OSDev Wiki
  https://wiki.osdev.org/System_V_ABI
- Linux system calls table for several architectures
  https://fedora.juszkiewicz.com.pl/syscalls.html
- Linux: arch/mips/kernel/syscalls/syscall_o32.tb
- MIPSpro N32 ABI Handbook
- Xinuos – OpenServer 10 based on FreeBSD
- https://en.wikipedia.org/wiki/Xinuos
- A4M35OSP – Open Source Programming
  https://support.dce.felk.cvut.cz/osp/