Basics

Crash Dump Analysis 2015/2016

CHARLES UNIVERSITY IN PRAGUE
faculty of mathematics and physics
Department of Distributed and Dependable Systems
D3S

ORACLE
SUSE
redhat

Processors

- **Basic functionality of processors**
  - Execute machine code
    - Sequence of instructions (simple operations)
      - Arithmetic, logic, conditional, jumps, branches, etc.
  - Access memory and peripherals
    - Using registers for internal data storage
    - Using memory management unit
      - Translating virtual addresses to physical addresses

Machine Code

```
fa
31 c0
8e d8
66 0f 01 16
29 82 0f 20 c0 66
83 c8 01
0f 22 c0
66 ea 1d 80 00 00
08 00
```

opcode

```
cli
xor   %eax, %eax
mov   %eax, %ds
lgdtw (%esi)
sub   %eax, 0x66c0200f(%edx)
or    $0x1, %eax
mov   %eax, %cr0
ljmpw $0x0, $0x801d
or    %al, (%eax)
```

instruction name

register name

dereference

displacement

constant

IA-32 machine code
Sequence of bytes stored in memory and processed by the processor

IA-32 instruction mnemonics (AT&T syntax)
Code written by hand or compiled by a high-level language compiler

Assembler compiles it to machine code

Machine Code (2)

```
86 03 a7 ff
89 57 c0 00
82 10 3f ff
83 28 70 0e
86 08 c0 01
c4 58 e0 18
81 c3 e0 08
c8 70 a4 78
```

(opcode)

```
add  %sp, 0x7ff, %g3
rdpr %ver, %g4
mov  -1, %g1
sllx %g1, 0xe, %g1
and  %g3, %g1, %g3
ldx  [ %g3 + 0x18 ], %g2
retl
stx  %g4, [ %g2 + 0x478 ]
```

(instruction name, register name, constant, displacement, dereference)

SPARC V9 machine code
All instructions are exactly 4 bytes long

SPARC V9 instruction mnemonics

Binary Interoperability

- **Many degrees of freedom for machine code**
  - How to call subroutines (functions)?
  - How to pass arguments to functions?
  - How to pass return values from functions?
  - Where to store local variables?
  - **A**pplication **B**inary **I**nterface
    - Set of conventions defining machine code usage
    - Interoperability between compilers, assemblers, linkers, libraries, user space and operating system, etc.

Registers

- **Small and fast static memory**
  - Integral part of most processors
  - Addressed usually by small index
    - In assembler mnemonics by a name (`eax`, `rflags`, `g7`, etc.)
  - Usually word-sized (8, 16, 32, 64, 128 bits)
  - Basic taxonomy
    - System Registers
    - Application Registers
      - General Purpose Registers (accumulator, base, source, destination, etc.)
      - Special Registers (flags, etc.)

General Purpose Registers

- **For general computation**
  - Values fetched from memory, results of calculations, addresses, etc.
  - Defined by the architecture of the processor
    - IA-32
      - `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`
    - AMD64
      - `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`, ..., `r15`
    - SPARC V9
      - `r0`, ..., `r31` (but it's more complicated)

Volatile GPRs

- **Scratch, Caller-Saved**
  - Defined by the ABI
  - When calling a function, these registers **can be** right away **used by the callee**
  - If the **caller** stores important values in these registers (and wants to preserve the values across function calls), it **must save** them before it calls the callee

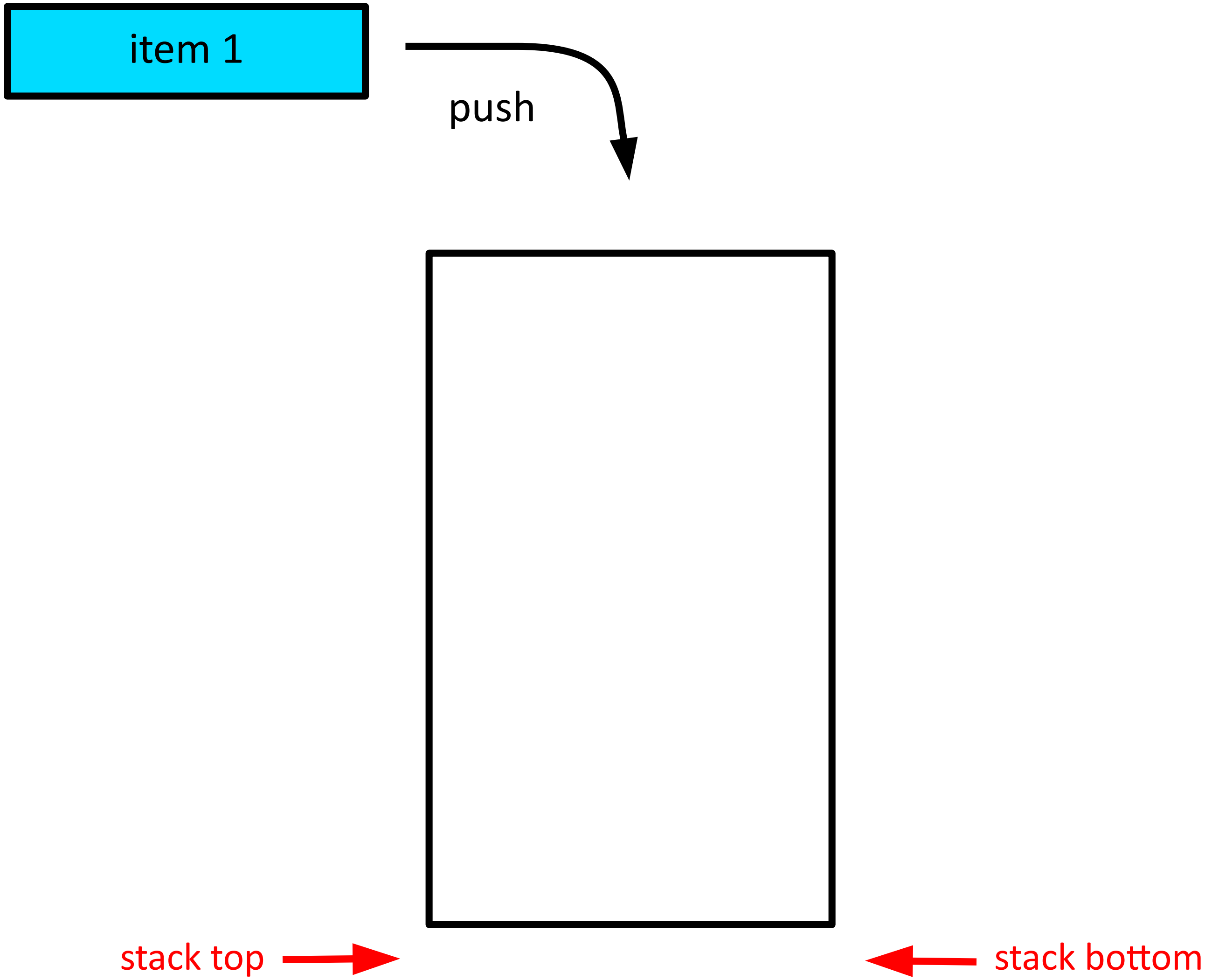
Non-Volatile GPRs

- **Preserved, Callee-Saved**
  - Defined by the ABI
  - When calling a function, the **callee must save the original values** before it can use these registers and **restore the original values** before returning to the caller
  - The caller does not need to care about these registers, the **values are preserved for the caller**

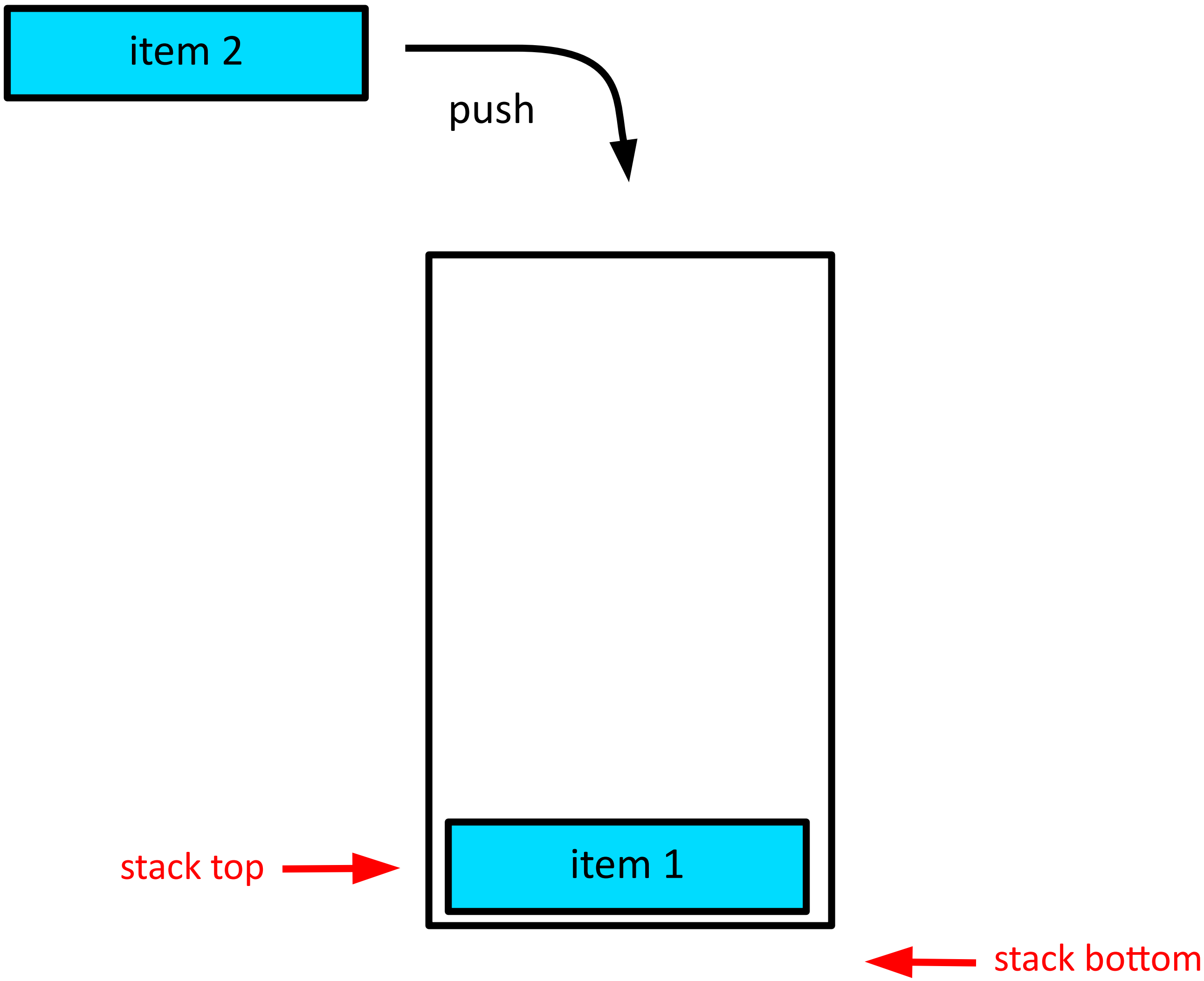
Memory Stack

- **Stack**
  - Generic data structure for storing items of the same kind
    - A queue with LIFO (last-in first-out) semantics
    - Two basic operations
      - Push (add an item to the stack)
      - Pop (remove the last inserted item from the stack)
- **Memory Stack**
  - A stack stored physically as an array of items in memory

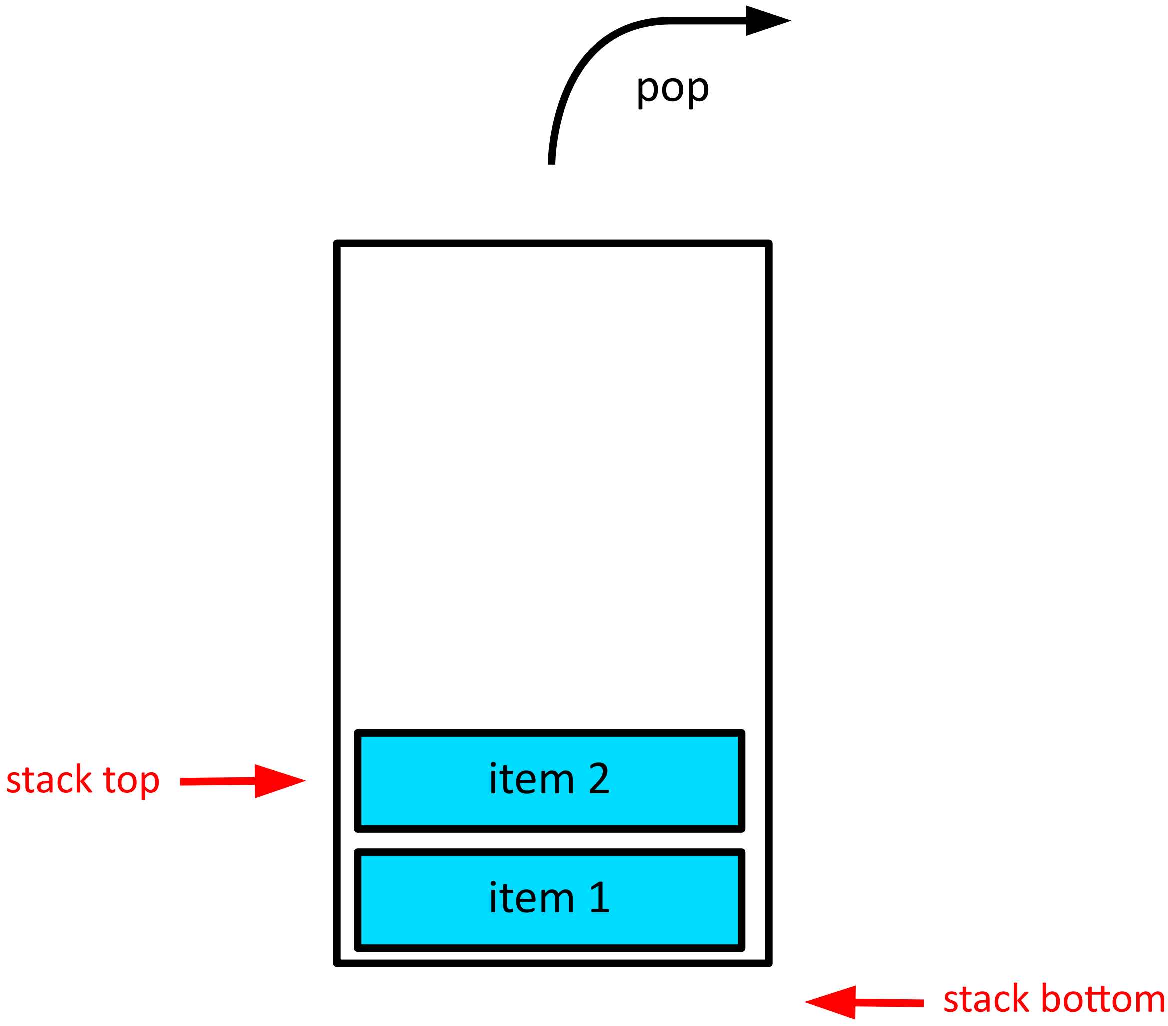
Memory Stack (2)



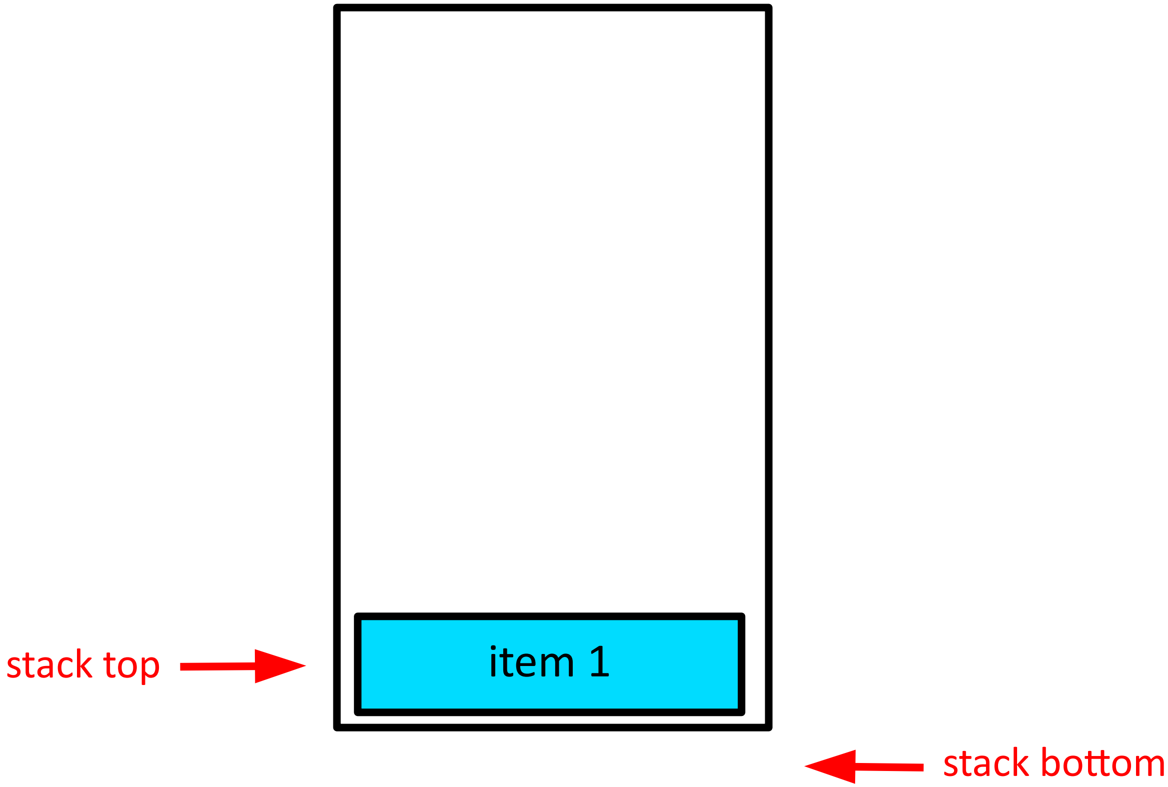
Memory Stack (2)



Memory Stack (2)

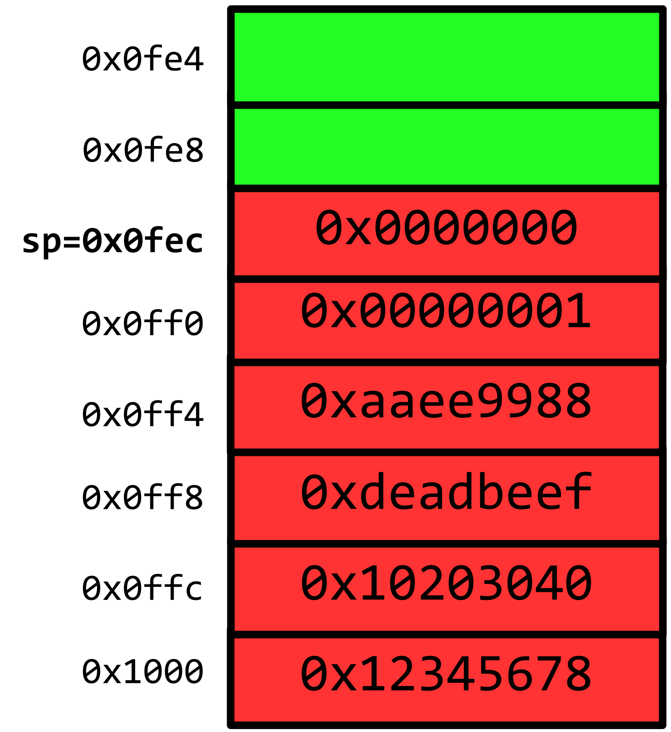


Memory Stack (2)



Memory Stack (3)

- **Storing register values**
  - Return addresses, local variables, etc.
  - Stack top is addressed by stack pointer (register)
    - Aligned, usually grows towards lower addresses
      - Push decreases the pointer
      - Pop increases the pointer
  - All items are also accessible directly
    - Stack pointer as a base address, plus displacement



Memory Stack (4)

- **Implicit stack support**
  - The processor has inherent support for the memory stack
    - Automatically pushes the return address onto the memory stack when calling a subroutine (function)
    - Pops the return address from the memory stack when returning from the function
    - Usually provides dedicated instructions (PUSH, POP)
    - Hard-wired stack pointer register
    - IA-32, AMD64

Memory Stack (5)

- **Explicit stack**
  - The processor does not provide any dedicated primitives to work with the memory stack
    - The code operates with the memory stack as with any other memory area (by hand)
      - PUSH: Decrease the stack pointer and store the value
      - POP: Fetch the value and increase the stack pointer
      - Stack pointer register is defined by ABI (not hard-wired)
    - Return address is stored in a dedicated register
      - Optionally pushed to the stack by hand
    - MIPS, SPARC V9 (in flat mode)

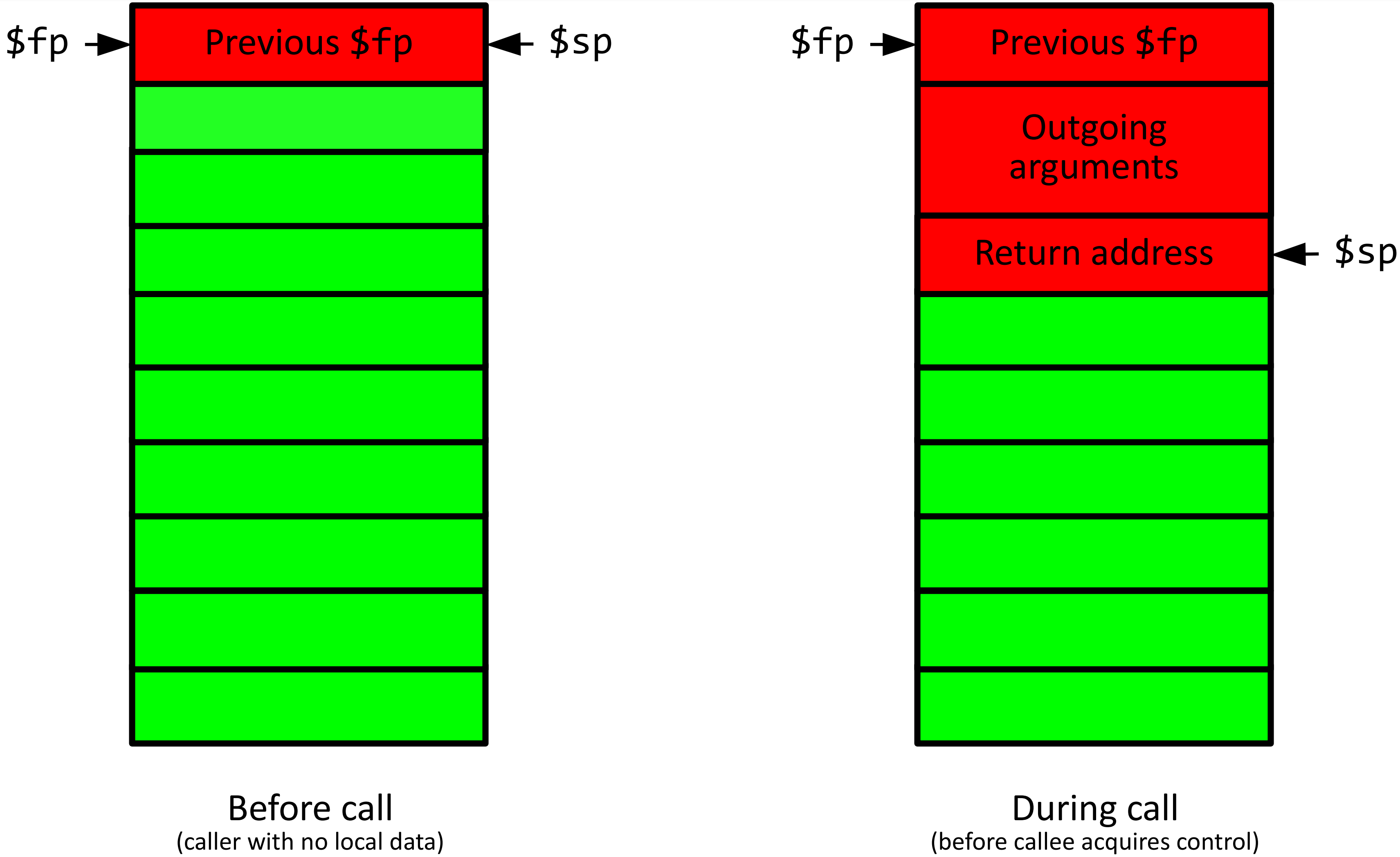
Memory Stack (6)

- **Stack pointer**
  - Points to the last pushed item
    - Except for possible stack pointer bias
  - Stored in a GPR
    - Either defined by the architecture or by the ABI
- **Stack frame**
  - Items on the stack that belong to a single function
- **Frame pointer**
  - Optional pointer pointing to the beginning of the current stack frame
    - Can be stored in a GPR (defined by the ABI)

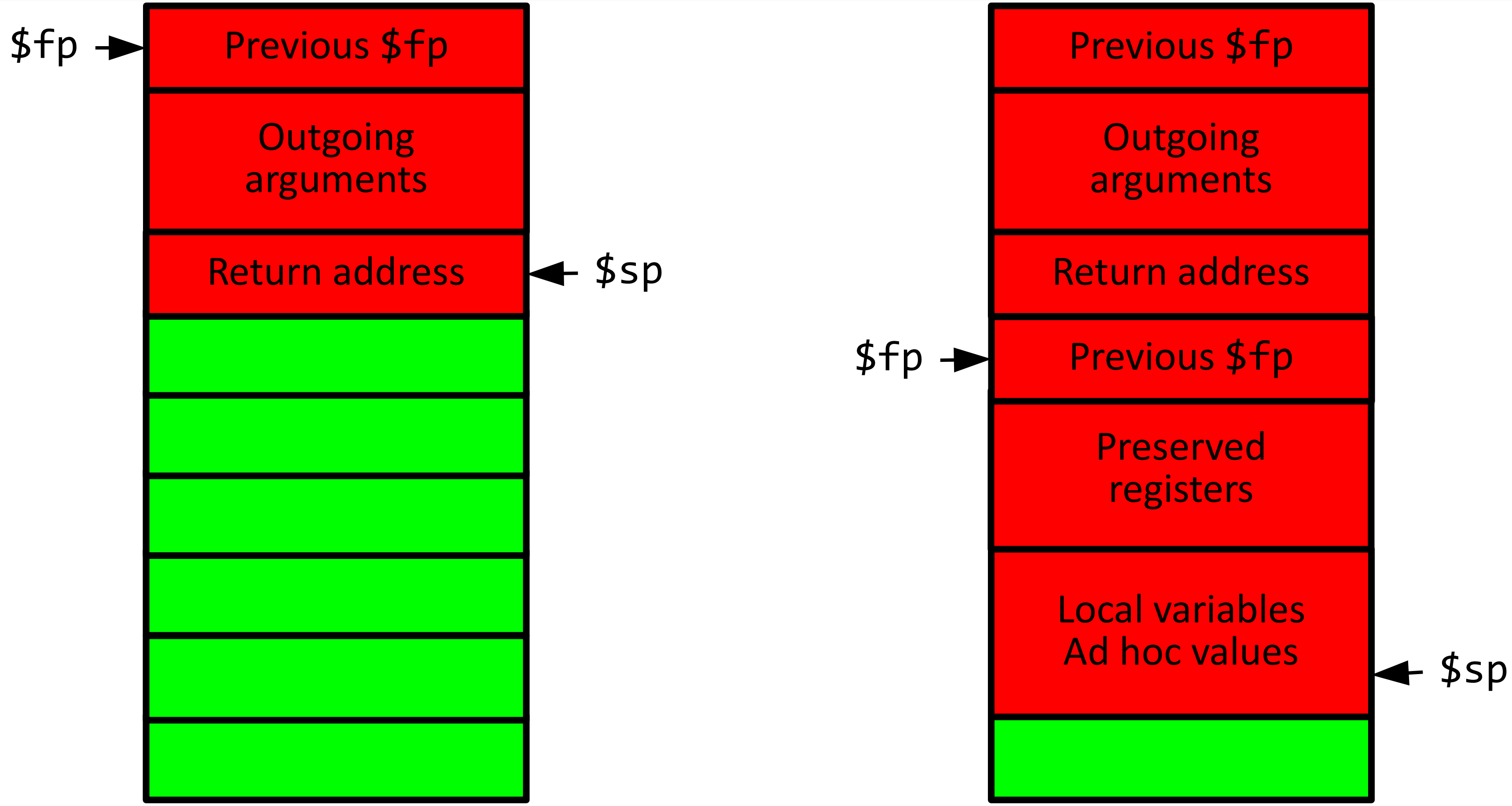
Stack Frame

- **Inserted on the stack when a subroutine is called**
  - Contents
    - Pointer to the previous stack frame (of the caller)
    - Values of preserved registers (if needed)
    - Local variables
    - Ad hoc values (if any)
      - Other preserved registers, stack allocations, etc.
    - When calling other subroutine
      - Outgoing arguments and return address (if any)
  - Defined by the ABI
    - Who creates the stack frame (on subroutine entry)
    - Who removes the stack frame (on subroutine exit)

Stack Frame (2)



Stack Frame (3)



During call
(before callee acquires control)

After call
(callee has control)

Stack Frame (4)

- **Consecutive stack frames**
  - Organized in a linked list
  - Linked using the the pointer to the previous stack frame
    - Embedded in any given stack frame
  - The printout of this linked list is a stack trace

Stack Trace

```
int c(int i) { return i; }

int b(int i) { return c(i); }

int a(int i) { return b(i); }

int main(int argc, char *argv[]) {
    return a(argc);
}
```

stack frame address

```
08046b78:  c+3(1, 0, 0, fefa3000)
08046b98:  b+0x11(1, 80509e4, feffb350, 8)
08046bb8:  a+0x11(1, fee9431c, 29, fefab85c)
08046bec:  main+0x27(1, 8046c10, 8046c18)
08046c04:  _start+0x80(1, 8046d60, 0, 8046d68, 8046d8a, 8046da1)
```

return address

function arguments
(not reliable, must confront with reality)

Stack Frames

```
0x8046b78:  0x8046b98
0x8046b7c:  b+0x11
0x8046b80:  1
0x8046b84:  0
0x8046b88:  0
0x8046b8c:  0xfefa3000
0x8046b90:  0x8047ff3
0x8046b94:  libc.so.1`_fpstart
0x8046b98:  0x8046bb8
0x8046b9c:  a+0x11
0x8046ba0:  1
0x8046ba4:  0x80509e4
0x8046ba8:  ld.so.1`dbg_desc
0x8046bac:  8
0x8046bb0:  1
0x8046bb4:  0
0x8046bb8:  0x8046bec
0x8046bbc:  main+0x27
0x8046bc0:  1
0x8046bc4:  libc.so.1`_fpstart+0x2c
0x8046bc8:  0x29
0x8046bcc:  libc.so.1`_fp_hw
0x8046bd0:  0x133f
0x8046bd4:  0x8050cf6
0x8046bd8:  0x8060d5c
0x8046bdc:  0x8046bcc
0x8046be0:  0x8046bec
0x8046be4:  _init+0x1a
0x8046be8:  0xfeffb7dc
0x8046bec:  0x8046c04
```

Stack Frames

c()'s frame

```
0x8046b78:   0x8046b98
```

b()'s frame

```
0x8046b7c:   b+0x11
0x8046b80:   1
0x8046b84:   0
0x8046b88:   0
0x8046b8c:   0xfefa3000
0x8046b90:   0x8047ff3
0x8046b94:   libc.so.1`_fpstart
0x8046b98:   0x8046bb8
```

a()'s frame

```
0x8046b9c:   a+0x11
0x8046ba0:   1
0x8046ba4:   0x80509e4
0x8046ba8:   ld.so.1`dbg_desc
0x8046bac:   8
0x8046bb0:   1
0x8046bb4:   0
0x8046bb8:   0x8046bec
```

main()'s frame

```
0x8046bbc:   main+0x27
0x8046bc0:   1
0x8046bc4:   libc.so.1`_fpstart+0x2c
0x8046bc8:   0x29
0x8046bcc:   libc.so.1`_fp_hw
0x8046bd0:   0x133f
0x8046bd4:   0x8050cf6
0x8046bd8:   0x8060d5c
0x8046bdc:   0x8046bcc
0x8046be0:   0x8046bec
0x8046be4:   _init+0x1a
0x8046be8:   0xfeffb7dc
0x8046bec:   0x8046c04
```

Stack Frames

c()'s frame

```
0x8046b78:   0x8046b98
```

b()'s frame

```
0x8046b7c:   b+0x11
0x8046b80:   1
0x8046b84:   0
0x8046b88:   0
0x8046b8c:   0xfefa3000
0x8046b90:   0x8047ff3
0x8046b94:   libc.so.1`_fpstart
0x8046b98:   0x8046bb8
```

a()'s frame

```
0x8046b9c:   a+0x11
0x8046ba0:   1
0x8046ba4:   0x80509e4
0x8046ba8:   ld.so.1`dbg_desc
0x8046bac:   8
0x8046bb0:   1
0x8046bb4:   0
0x8046bb8:   0x8046bec
```

main()'s frame

```
0x8046bbc:   main+0x27
0x8046bc0:   1
0x8046bc4:   libc.so.1`_fpstart+0x2c
0x8046bc8:   0x29
0x8046bcc:   libc.so.1`_fp_hw
0x8046bd0:   0x133f
0x8046bd4:   0x8050cf6
0x8046bd8:   0x8060d5c
0x8046bdc:   0x8046bcc
0x8046be0:   0x8046bec
0x8046be4:   _init+0x1a
0x8046be8:   0xfeffb7dc
0x8046bec:   0x8046c04
```

Prologue and Epilogue

- **Stack frame bookkeeping code**
  - Function code generated by the compiler
  - Prologue
    - Sequence of instructions at the beginning of a function
    - Creates the stack frame and saves the used preserved registers
  - Epilogue
    - Sequence of instructions before the function returns
    - Restores the saved preserved registers and destroys the stack frame

Stack Optimizations

- **Unusual stack frame constriction**
  - For speed optimization
  - Obfuscation of the stack trace
  - Leaf optimization
    - Leaf functions don't create their own stack frame and reuse the caller's stack frame
  - Tail call optimization
    - If a function is called from the tail of the current function, the caller's stack frame is recycled
  - Function inlining
    - The function call is replaced by direct inclusion of the code of the callee

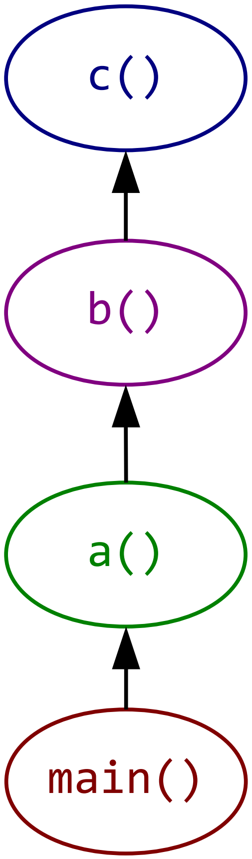
Stack Optimizations (2)

- **In the following example ...**
  - Functions `main()`, `a()`, `b()`, `c()` compiled with various compiler optimization levels
  - Breakpoint in the business code of `c()`
    - Avoiding the prologue and epilogue of `c()`
  - Displaying the stack trace when the breakpoint is hit

No Optimization

- **gcc -O0**
  - All stack frames are present

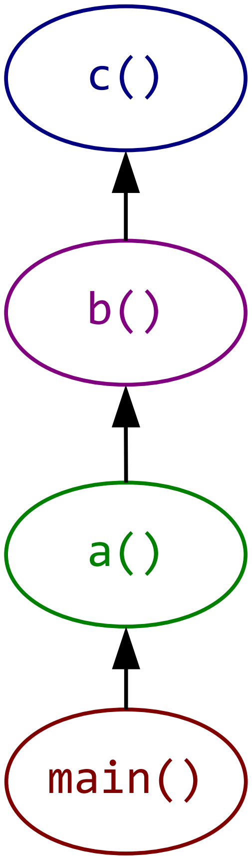
```
08046b78:   c+3(1, 0, 0, fefa3000)
08046b98:   b+0x11(1, 80509e4, feffb350, 8)
08046bb8:   a+0x11(1, fee9431c, 29, fefab85c)
08046bec:   main+0x27(1, 8046c10, 8046c18)
08046c04:   _start+0x80(1, 8046d60, 0, ...)
```



Better Optimization

- **gcc -O1**
  - c() is leaf-optimized
    - c() is not creating its own stack frame
    - c() is using b()'s stack frame
      - Thus b() is not shown in the stack trace

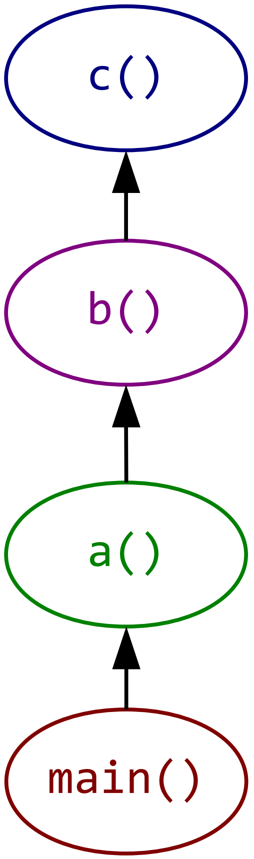
```
08046b78:   c(1, 80509e8, feffb350, 8)
08046bb8:   a+0xe(1, fee9431c, 29, fefab85c)
08046bec:   main+0x14(1, 8046c10, 8046c18)
08046c04:   _start+0x80(1, 8046d60, 0, ...)
```



Even Better Optimization

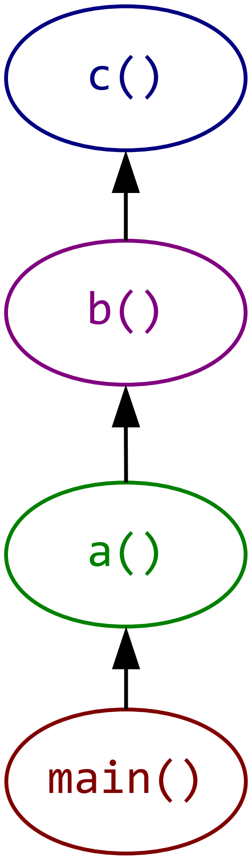
- **gcc -O2**
  - a() and b() is tail-call-optimized
    - a() and b() recycle the caller's stack frame
      - Thus a(), b() and main() are not shown in the trace
  - c() is leaf-optimized (as previously)

```
08046b78:   c(1, 8046c10, 8046c18)
08046c04:   _start+0x80(1, 8046d60, 0, ...)
```



Best Optimization

- `gcc -O3`
  - Function inlining on `a()`, `b()` and `c()`
    - Thus `a()`, `b()`, `c()` not even called by `main()`
    - Our breakpoint is not even hit
  - `a()` and `b()` is tail-call-optimized (as previously)
  - `c()` is leaf-optimized (as previously)



Morale of the Example

- **Compiler optimizations produce better (faster) code**
- **Optimized code is harder to debug**
  - Inlined functions
  - Missing stack frames
  - Untraceable argument values
- **Optimized code is harder to understand**
  - Relation between source code and machine code is not straightforward

Special Stack Frames

- **Trap frames**
  - Created by interrupt or exception (trap) handlers
    - Contain values of all registers in the time the trap occurred
    - Also saved on the stack
  - Usually not visible in a core dump (user space)
  - Can be seen in a crash dump (when code is in kernel)
  - If time permits, more details later in the course

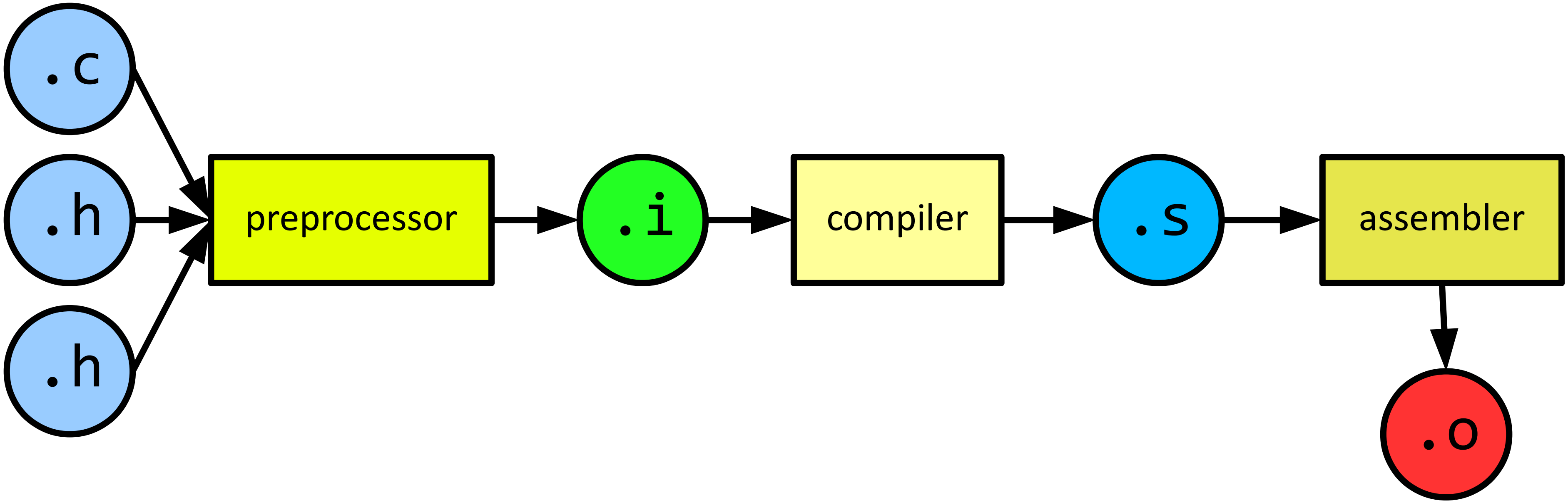
ABI Revisited

- **Two parts of every ABI**
  - Defined by architecture/platform (hard-wired)
  - Defined by the run-time environment
- **What is specified (among others)**
  - Set of volatile and non-volatile GPRs
  - Calling convention
    - Who creates and destroys stack frames
    - The way function arguments and return values are passed
  - Stack layout

ABI Revisited (2)

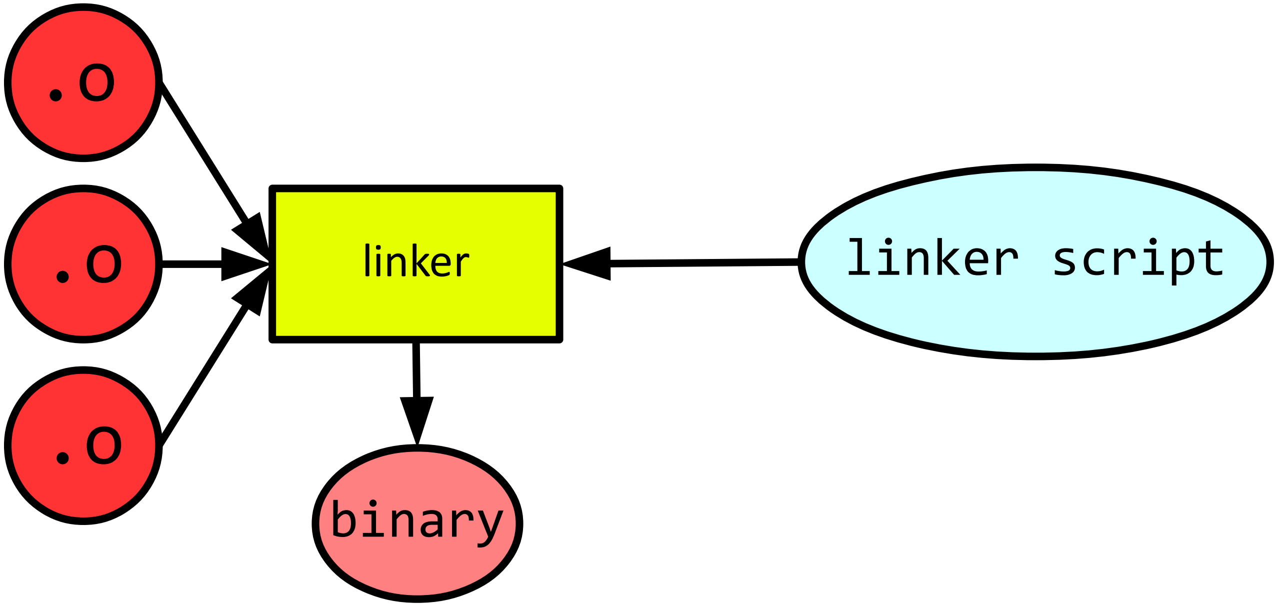
- **UNIX System V Application Binary Interface**
  - Intel386 Architecture Processor Supplement
  - AMD64 Architecture Processor Supplement
  - SPARC Compliance Definition
- **Windows ABI**
  - stdcall, fastcall, etc.
  - x64 Software Conventions

Compiler Toolchain




```
cc -c -o output.o input.c
as -o output.o input.s
```

Linking




```
ld -T link.ld -o output.bin input0.o input1.o
```

Linking (2)

input0.c

```
void global_fnc01() { }
void global_fnc02() { }

int global_int;
void *global_ptr;

int another_symbol __attribute__((section(“another_section”)));
```

input0.o

```
.text
    global_fnc01
    global_fnc02
.bss
    global_int
    global_ptr
another_section
    another_symbol
```

Linking (3)

link.ld

```
SECTIONS {
    .output 0x80000000 : {
        *(.text)
        *(.bss)
        *(another_section)
        _output_end = .;
    }
}
```

output.bin

```
.output (displacement: 0x80000000)
    global_fnc01
    global_fnc02
    global_int
    global_ptr
    another_symbol
    _output_end
```

Useful Tools

- `objdump`
  - Display various information about binary files

    `-x`
    - Display all headers

    `-d`
    - Disassemble executable sections

    `-D`
    - Disassemble all sections

    `-S`
    - Disassemble with source file intermixed
    - Debugging information is needed (`gcc -g`)