# CS Basics
# 5) Stack, Syscalls, Branching

Fall Term 2023-24

E.Benoist, C.Fuhrer, Ch. Grothoff, L. Ith, P.Mainini | BFH-TI

# Stack, Syscalls, Branching

- **The Stack**
- **System Calls on x86-64**
- **Logical Operations**
- **Shift and Rotate Instructions**
- **Branching and Comparing**
- **Addressing Modes**
- **Example:** `hexdump1.asm`
- **Conclusion**

# The Stack

# The Stack

**Can be used to store information temporarily**

- Store the state of a program while another part of it is running
- Often used to save registers

**LIFO Data Structure**

- Last in, first out
- Examples: Dish dispenser at the cafeteria, “Pez (candy) dispenser”

**Semantics: Two primary ways to access a stack**

- *Push:* add an item on top of the stack
- *Pop:* remove an item from the top of the stack

# The Stack

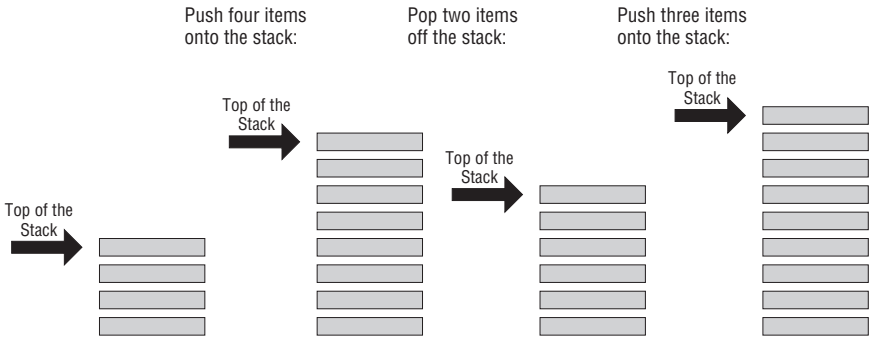


**Figure 8-1:** The stack

# The Stack in Program Memory

**The stack starts at a high address in program memory**

- It grows down in direction of free memory

**The other program parts are all stored at the “bottom” of a program’s memory**

- `.text` section
- `.data` section
- `.bss` section

**Heap: Programs can dynamically allocate memory for large structures**

- On the fly, while the program runs
- Allocation typically grows up towards the stack
- Hopefully, there is sufficient free memory between the two…

# The Stack in Program Memory

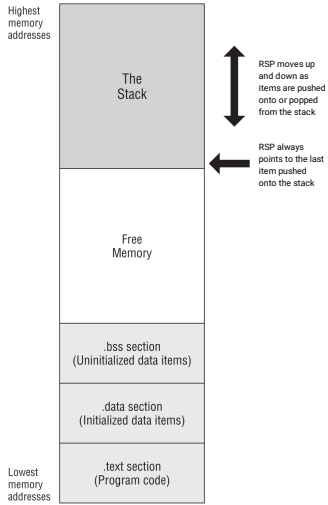


**Figure 8-2:** The stack in program memory

# `push` Instructions

## Add data on top of the stack

- `push` pushes a register or memory value
- `pushf` pushes the flags register

## examples

```
push 42             ; Push an immediate value
push ax             ; Push the ax register
; push eax          ; Not supported on x86-64!
push rax            ; Push the rax register
push word[Addr]     ; Push a word from mem at Addr
; push dword[Addr]  ; Not supported on x86-64!
push qword[Addr]    ; Push a quad word from mem at Addr
pushf               ; Push RFLAGS to stack
```

# `pop` Instructions

## Retrieve data from the top of the stack

- "Removes" the requested data from the stack and transfers it to given destination
- *The programmer needs to know type and order of data on the stack*

## Instructions

- `pop` retrieves a register or memory value
- `popf` retrieves the flags register

## Examples

```
popf                ; Pop RFLAGS from stack
pop qword[Addr]     ; Pop a quad word to mem at Addr
; pop dword[Addr]   ; Not supported on x86-64!
pop word[Addr]      ; Pop a word to mem at Addr
pop rbx             ; Pop to the rbx register
; pop ebx           ; Not supported on x86-64!
pop bx              ; Pop to the bx register
```

# How the Stack Works

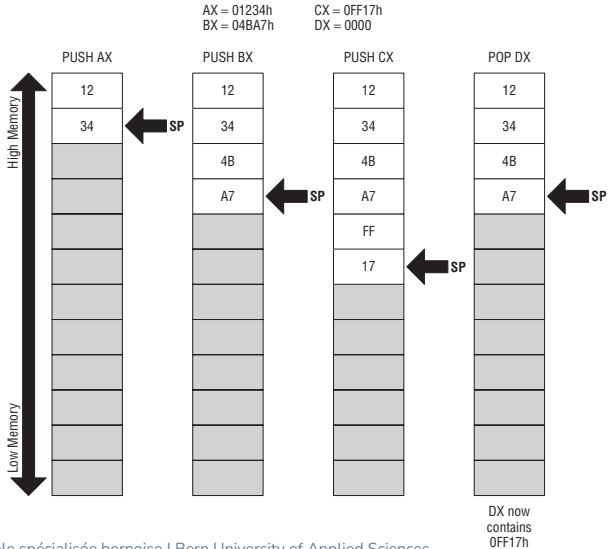


**Figure 8-3:** How the stack works

# Stack Operations with RSP

RSP can also be used directly for addressing elements on the stack.[1]
The following example demonstrates "pushing" two values to the stack using RSP:

```
; Note: mov does not support copying
; 64-bit immediates directly to memory
mov r8, 0x1234567890abcdef
sub rsp, 8
mov qword [rsp], r8

sub rsp, 2
mov word [rsp], 0x1234

pop bx
pop rax
```

[1] In fact, there are architectures without `push` and `pop` instructions!

# System Calls on x86-64

# `syscall` Instruction

- **System calls provide access to functions of the OS**
- **Making a System Call:** `syscall`
  - The instruction "`syscall`" is used for making a system call
  - On 32-bit x86, system calls where made using the "`int 80`" (interrupt) instruction
  - `syscall` is more efficient
  - A 32-bit compatibility layer ensures that software interrupts still work
  - Syscall-numbering is different between 32 and 64 bit![2]
- **Parameters**
  - `rax`: Specifies the syscall to be invoked
  - Additional parameters are passed in `rdi`, `rsi`, `rdx`, …
- **Return Values**
  - If a system call returns a value, it can be found in `rax`

[2] Full list (64-bit): `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/entry/syscalls/syscall_64.tbl`

# `syscall` Example I

## Display a string

- `rax` : `1` = sys_write
- `rdi` : `1` = Standard output
- `rsi` : Address of the string to be written
- `rdx` : Length of the string (number of bytes)

## Return value

- The number of bytes written is stored in `rax` (not necessarily the same as `rdx`!)

```
mov rax, 1      ; Code for sys_write
mov rdi, 1      ; File descriptor: 1 (Standard Output)
mov rsi, EatMsg ; Pass offset of the message
mov rdx, EatLen ; Pass the length of the message
syscall         ; Make kernel call
```

# File Descriptors

## Standard Input

- `stdin`
- File descriptor 0

## Standard Output

- `stdout`
- File descriptor `1`

## Standard Error

- `stderr`
- File descriptor `2`

## Other files

- A program can open more files (e.g. with the `sys_open` syscall)
- For each, a new file descriptor is allocated

# `syscall` Example II

**Exit program**

- `rax` : 60 = `sys_exit`
- `rdi` : Return code 0 (ok)

```
mov rax, 60 ; Code for sys_exit
mov rdi, 0  ; Return a code of zero
syscall     ; Make kernel call
```

# Example: `eatsyscall.asm`

```
;  Executable name : eatsyscall    ;  Version         : 1.0
;  Created date    : 08/30/2016    ;  Last update     : 08/30/2016
;  Author          : Emmanuel Benoist
;  Description     : A simple program in assembly for Linux, using NASM,
;    demonstrating the use of Linux 64-bit syscalls to display text.
;
;  Build using these commands:
;    nasm -f elf64 -g -F dwarf eatsyscall.asm
;    ld -o eatsyscall eatsyscall.o

SECTION .data                   ; Section containing initialized data
    EatMsg: db "Eat at Joe's!",10
    EatLen: equ $-EatMsg        ; Compute the length of the string

SECTION .bss                    ; Section containing uninitialized data

SECTION .text                   ; Section containing code

global _start                   ; Linker needs this to find the entry point!

_start:
    mov rax, 1                  ; Code for sys_write call
    mov rdi, 1                  ; Specify File Descriptor 1: Standard Output
    mov rsi, EatMsg             ; Pass offset of the message
    mov rdx, EatLen             ; Pass the length of the message
    syscall                     ; Make kernel call
    mov rax, 60                 ; Code for exit syscall
    mov rdi, 0                  ; Return a code of zero
    syscall                     ; Make kernel call
```

# Logical Operations

# Boolean Operators

Binary and unary Boolean operators are commonplace in Assembly programming.[3] They work on *individual* bits and assume that *1 is true* and *0 is false*.

- ***Binary* Boolean operators**
  - `AND` (conjunction)
    True if and only if *both* operands are true.
  - `OR` (disjunction)
    True if and only if *at least* one of the operands is true.
  - `XOR`
    True if and only if *exactly one* of the operands is true. Can be composed from `AND`, `OR` and `NOT`.
- ***Unary* Boolean operators**
  - `NOT` (negation)
    *Negates* the result ($1 \to 0, 0 \to 1$)

[3] Refer to module “discrete mathematics” for more details.

# Anatomy of an AND Instruction

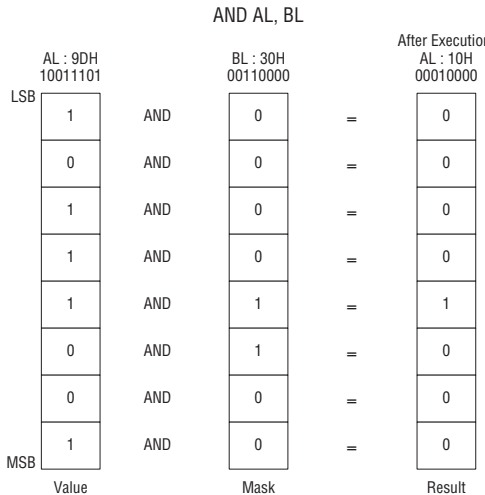


**Figure 9-2:** The anatomy of an AND instruction

# Masking Bits

## `AND` can be used to *mask* individual bits

- A *bitmask* is prepared, it contains a `1` for every bit we are interested in, the others are `0`
- The `AND` operation is applied to a given value and the bitmask
- The value of the desired bits can then be read from the result

## Example

```
; Read bits 0 and 3 of a byte in register AL
; The required mask is 0000 1001 (0x09)

mov al, 0xff
and al, 0x09    ; al is 0x09
mov al, 0xfe
and al, 0x09    ; al is 0x08
mov al, 0xf0
and al, 0x09    ; al is 0x00
```

# Setting Bits

## `OR` can be used to set individual bits to `1`

- A *bitmask* is prepared, it contains a `1` for every bit to set
- The `OR` operation is applied to a given value and the bitmask
- The desired bits in the result are `1`, independent of the input value

## Example

```
; Set bits 0 and 3 of the bytes in register AL
mov al, 0xf6
or al, 0x09     ; al is 0xff
```

# Shift and Rotate Instructions

# `shl` and `shr` Instructions

- **Syntax**

```
shl <register/memory>, count    ; Shift left
shr <register/memory>, count    ; Shift right
```

  - Shift all the bits in `register/memory` by `count`
  - `count` can be an immediate value or register CL
  - `shl`: bits pass through *carry flag (CF)*!
  - `sal` and `sar`: Arithmetic shifts (signed operation)[4]

- **Example**

```
mov al, 0x09    ; 0000 1001 = 9
shl al, 2       ; 0010 0100 = 36
shl al, 2       ; 1001 0000 = 144
shl al, 1       ; 0010 0000 = 32 + CF
shr al, 5       ; 0000 0001 = 1
```

[4] Actually, `shl` and `sal` are the same due to overflow…

# Rotate Instructions

**Syntax:** Same as `shl` and `shr`

**Rotate with/without carry**

- `rol` and `ror` rotate bits left or right respectively
- `rcl` and `rcr` do the same, but the bits pass through carry flag (CF) before re-entering

**`stc` and `clc`: Clear or set the carry flag (CF)**

**Example**

```
mov al, 0x90    ; 1001 0000
rol al, 4       ; 0000 1001 = 0x09

clc             ; Clear CF
mov al, 0x90    ; 1001 0000
rcl al, 4       ; 0000 0100 = 0x04 + CF
rcl al, 1       ; 0000 1001 = 0x09
```

# Rotate Instructions

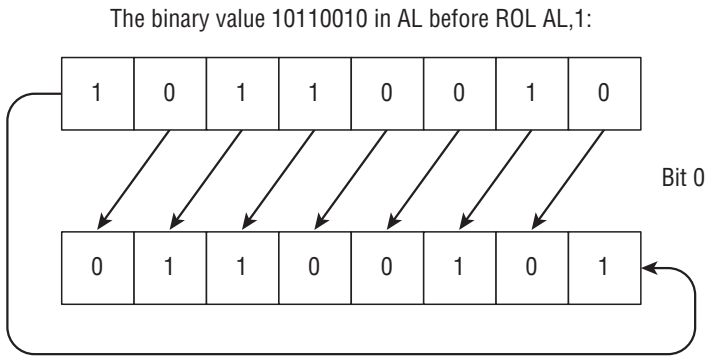


**Figure 9-4:** How the rotate instructions work

# Rotate Instructions with Carry

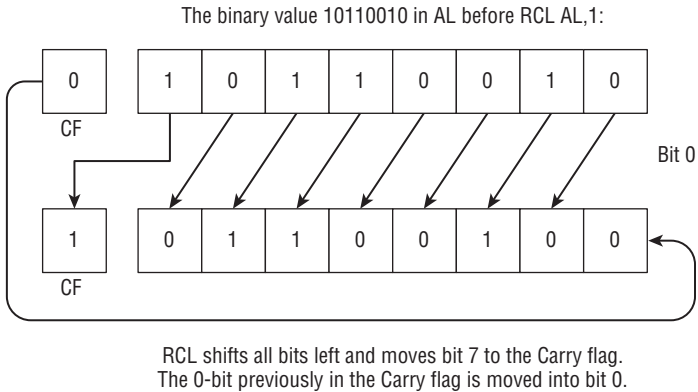


**Figure 9-5:** How the rotate through carry instructions work

# Divide and Multiply using Shifts

Shift instructions can be used to divide and multiply with *powers of two*. Combining shifts and `add`/`sub` may be more efficient than using `mul`. Examples (see also Slide 24):

- **Multiply by 9 by 3**

```
mov al, 0x09
shl al, 1       ; Multiply by 2
add al, 0x09    ; Add 9
```

- **Multiply 9 by 10**

```
mov al, 0x09
shl al, 3       ; Multiply by 8
add al, 0x09    ; Add 9
add al, 0x09    ; Add 9
```

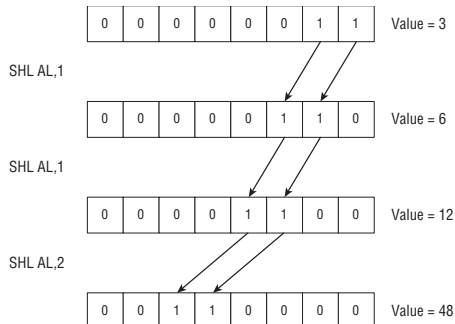


**Figure 9-8:** Multiplying by shifting

# Branching and Comparing

# Branch Instructions

## Unconditional branching

- Immediately continue processing at a different address
- Independent of any flags
- Jump instruction: `jmp <Address>`

## Conditional branching

- Test the value of one or more flags
- Jump only if flag(s) match expected value(s)
- Otherwise, program flow continues normally

## Example: Branching based on zero flag (ZF)

- Jump if zero (`jz`): Branch if ZF *is set*
- Jump if not zero (`jnz`): Jump if ZF *is not set*

# Example: `jz` and `jnz`

```
  mov word [RunningSum],0 ; Clear the running total
  mov rcx, 17             ; Counter for loop
WorkLookup:
  add word [RunningSum],3 ; Add 3 to the running total
  dec rcx                 ; Decrement counter
  jz SomewhereElse        ; Done if counter is zero!
  jmp WorkLookup          ; Loop otherwise
SomewhereElse:
  nop                     ; Rest of code
```

**Is there a better solution?**

# The `cmp` Instruction

**Syntax:** `cmp <op1>, <op2>`

- Instruction `cmp` compares two operands
- Think of a subtraction: `op1 - op2`
- Depending on the result, the same flags are set as for `sub`: CF, OF and SF (as well as ZF and PF)
- *The result is not stored, only the flags are modified*

**Using conditional branching instructions (Slide 33), branching can be performed depending on the outcome**

- E.g. Equality (ZF=1), inequality (ZF=0), less than (unsigned: CF=1, signed: SF $\neq$ OF), greater than or equal (unsigned: CF=0, signed: SF=OF)

# Conditional Branch Instructions

| Condition | Unsigned | Signed |
|---|---|---|
| Equal ($=$) | `je` | `je` |
| Not Equal ($\neq$) | `jne` | `jne` |
| Greater than ($>$) | `ja` | `jg` |
| Not less than or equal ($\not\leq$) | `jnbe` | `jnle` |
| Less than ($<$) | `jb` | `jl` |
| Not greater than or equal ($\not\geq$) | `jnae` | `jnge` |
| Greater than or equal ($\geq$) | `jae` | `jge` |
| Not less than ($\not<$) | `jnb` | `jnl` |
| Less than or equal ($\leq$) | `jbe` | `jle` |
| Not greater than ($\not>$) | `jna` | `jng` |

Note: “greater than” and “less than” are used for *signed values*; “above” and “below” for *unsigned* values.

# Addressing Modes

# Protected Mode Memory Addressing

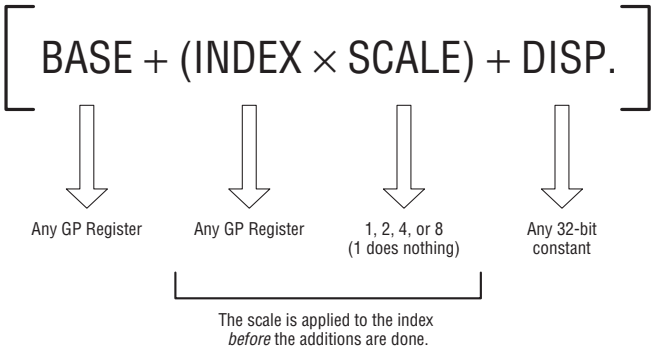


**Figure 9-9:** Protected mode memory addressing

# Addressing Modes I

***Effective address***

- The “real” address used to read or write memory

**Base addressing**

- A register contains the address

```
mov byte [rax], 42 ; Copy 42 to memory at address
                   ; given in rax
```

**Displacement addressing**

- Also called absolute- or static addressing
- Symbolic address (i.e. a label)

```
mov [Label], rax   ; Copy value from rax to memory
                   ; at address "Label"

mov [Label+4], rax ; Computed at assembly time!
```

# Addressing Modes II

## Base + Displacement addressing

- Constant start address plus varying index
- Example usage: array indexing, function arguments

```
mov [Label+rcx], rax
mov [Label+4+rcx], rax
```

## Base + Index and Base + Index + Displacement addressing

- Address is computed using two registers
- Optional displacement
- Example usage: 2-dimensional arrays

```
mov [rcx+rdx], rax
mov [Label+rcx+rdx], rax
```

# Addressing Modes III

**(Index · Scale) + Displacement addressing**

- Typical mode for addressing arrays if element size matches
- *Note: Scale may only be 1, 2, 4 or 8!*

```
mov [Label+rcx*4], rax
```

**Base + (Index · Scale) + Displacement addressing**

- Example usage: 2-dimensional arrays if element size matches

```
mov [Label+rcx*4+rdx], rax
```

# Array Addressing I

## Using Base + Displacement

- Can be used for addressing array elements of any size
- Base register must be adjusted by element size

```
  mov rcx, 0
loop:
  mov rax, [Label+rcx]
  add rcx, 6 ; Element size
```

## (Index · Scale) + Displacement

- Can be used for addressing array elements of size 1, 2, 4, 8
- Base register is adjusted by 1

```
  mov rcx, 0
loop:
  mov rax, [Label+rcx*4]
  inc rcx
```

# Array Addressing II

Example: For a label `DDTable` pointing to an array of double words:

[DDTable + RCX * 4]

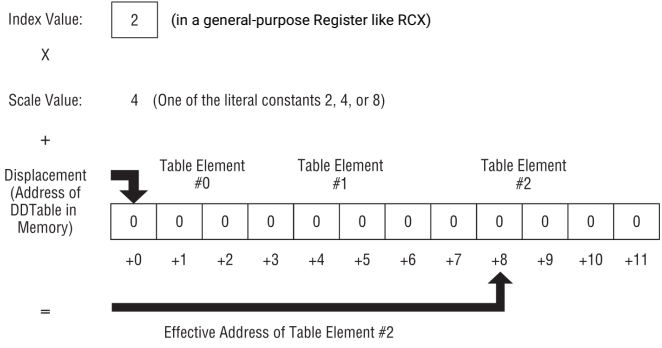


**Figure 9-10:** How address scaling works

# Load Effective Address: `lea`

**`lea` can be used to store an effective address in a register**

```
lea rax, [Label]
lea rax, [Label+rcx*4]
```

**Only the address is stored, not the value**

- Can be (ab)used to perform certain computations
- Values are not required to be (valid) addresses

**Example**

```
; Multiply rax by 3
mov rbx, rax
shl rax, 1
add rax, rbx
; The same, using lea
lea rax, [rax*2+rax]
```

# Example: Looping Through Memory

```
    xor rax, rax        ; Register to work with data
    xor rbx, rbx        ; Loop counter
loop:
    mov al, [Data+rbx]  ; Copy current byte from mem
    ; ...
    inc rbx             ; Increment counter
    cmp rbx, DataLen    ; Compare to len of Data
    jb loop             ; Loop if below (<)
    ; ...
```

# Example: `hexdump1.asm`

# Overview

## Displays binary data

- Individual bytes are printed as hexadecimal digits
- Similar to `hexdump` and `xxd` (GNU/Linux)

## Input and output

- Input file or data is given on *standard input* (`stdin`)
- May be typed in directly (press `Ctrl-d` to end input)
- Alternative: redirect input from a file: `./hexdump1 < file`
- Output is printed to *standard output* (`stdout`)

## How it works

- Reads `16` bytes at a time
- Transforms each byte into its hexadecimal representation
- Prints out `16` hexadecimal values separated by spaces, terminated by a line feed

# hexdump1.asm

See `examples/` directory.

# Lookup Table

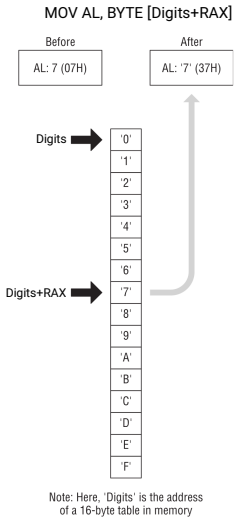


**Figure 9-6:** Using a lookup table

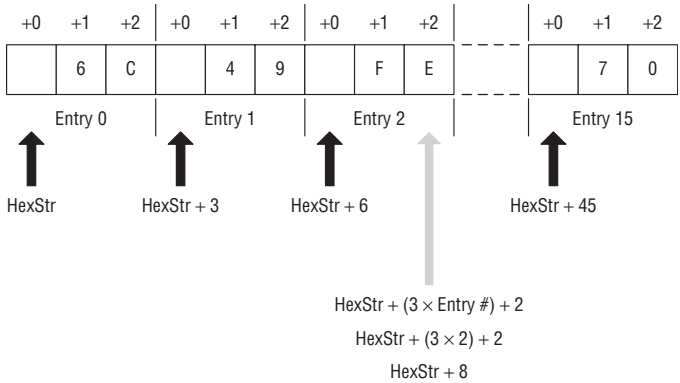


**Figure 9-7:** A table of 16 three-byte entries

# Conclusion

# Conclusion

- **Stack**
  - Used to store temporary information
  - Useful for restoring environment when calling an interrupt
  - Will be used to call functions
- **Syscall**
  - Possibility to access kernel functionality
- **Manipulating bits and bytes**
  - Using Boolean functions (`AND`, `OR`, `XOR` or `NOT`)
  - Using shift and rotate
  - Manipulate data as arrays of bytes
- **Conditional branching**
  - Works using `cmp` and flags
  - Many different branching instruction (jumps)
- **Addressing modes**
  - Different modes depending on usage
  - Planning memory layout is crucial!
- **Example: `hexdump1`**
  - Transforms (binary) input to hexadecimal output