Fakultät für
Informatik und
Mathematik

HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN
MÜNCHEN

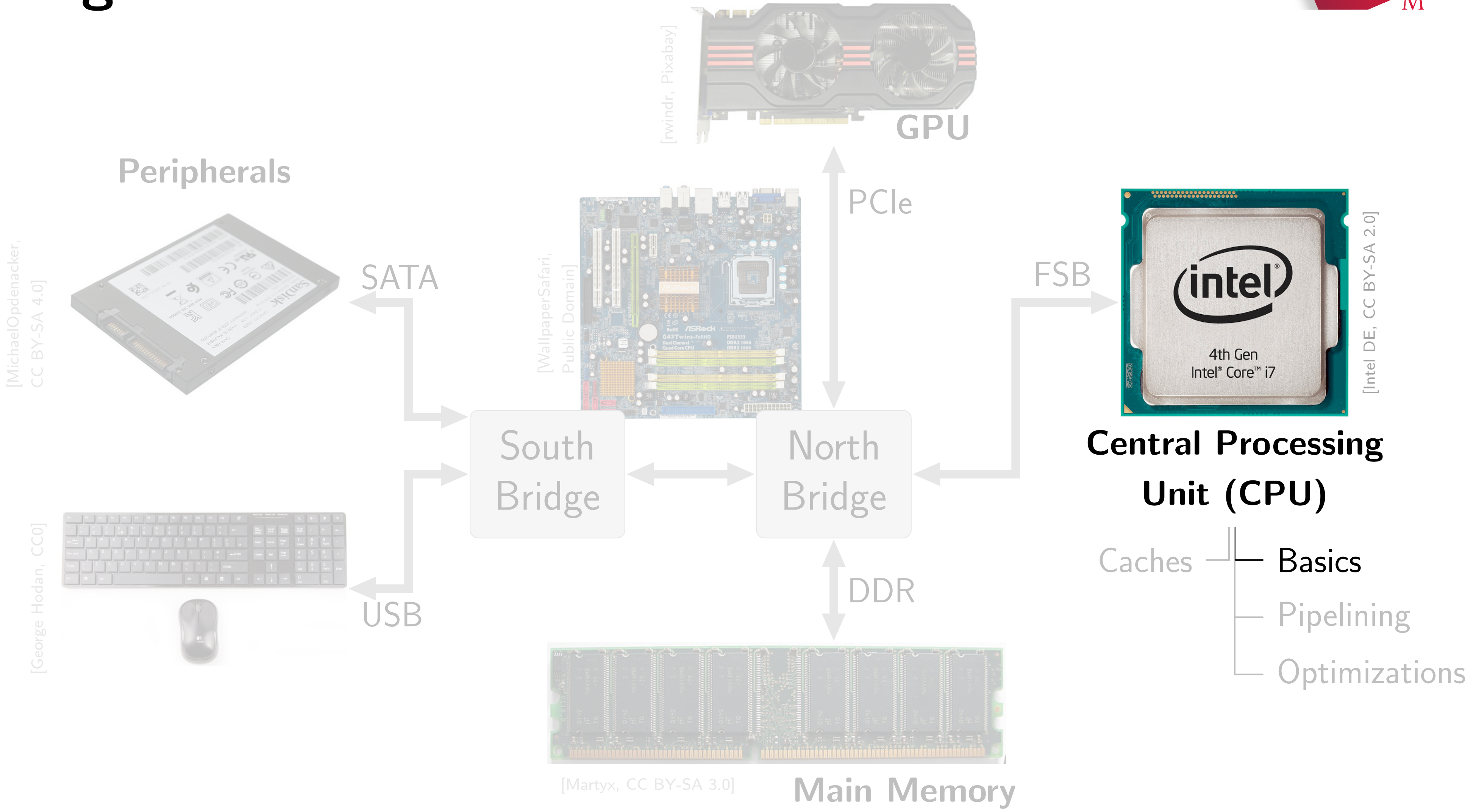
COMPUTER ARCHITECTURE

Chapter 2 – CPU Basics

Prof. Dr.-Ing. Stefan Wallentowitz

Department 07 – Munich University of Applied Sciences

Course Organization

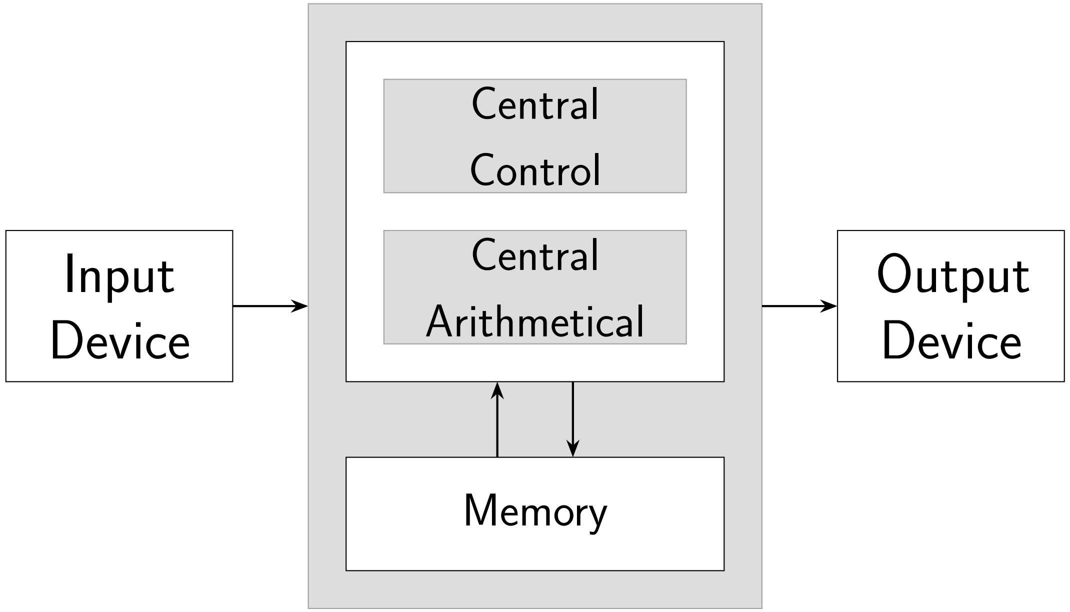


Learning Objectives

- Understand the role of the instruction set architecture
- Design programs in assembly/machine code from an instruction set architecture
- Apply rules and definition from an instruction set architecture in software development
- Understand the general principle and draw conclusions from it for practical tasks

Von Neumann Architecture

- Description of generic computer
- Computer organization independent of problem
- Memory
  - Intermediate results
  - Stored programs
  - Organized in homogeneous cells
  - Linearly addressed (data and program)
- Program counter in "central control" points to next instruction



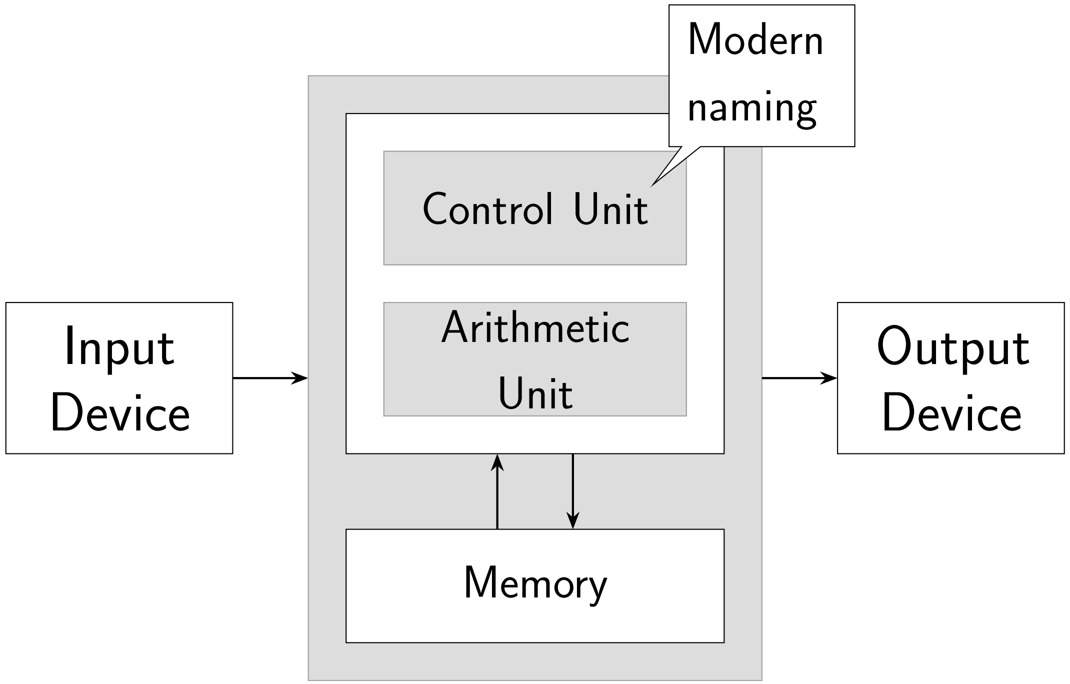
Von Neumann Architecture

Control Flow

- Load current instruction from memory
- Store instruction in control register
- Decode instruction
- Execute instruction based on operation

Types of operations

- Arithmetic and logical: Data manipulation
- Transport: Transfer data between elements
- Control flow: Change instruction stream
- Input/Output: Communication

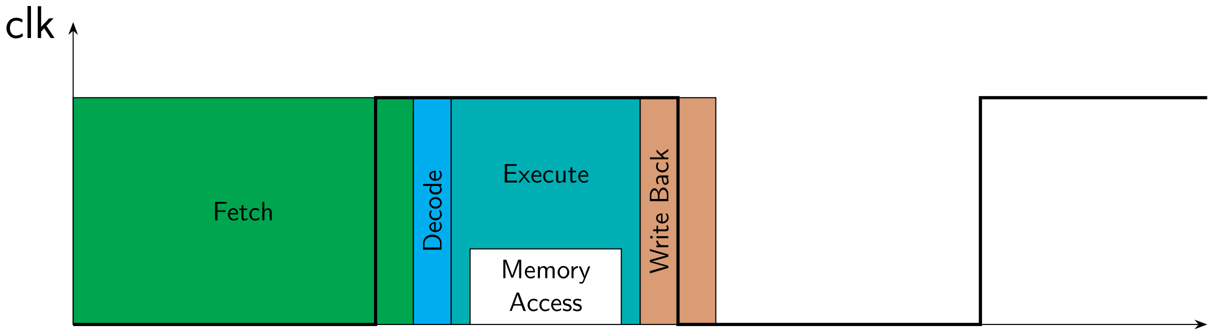


Instruction Processing

Remember: Microprocessor in Computer Engineering (Technische Informatik)

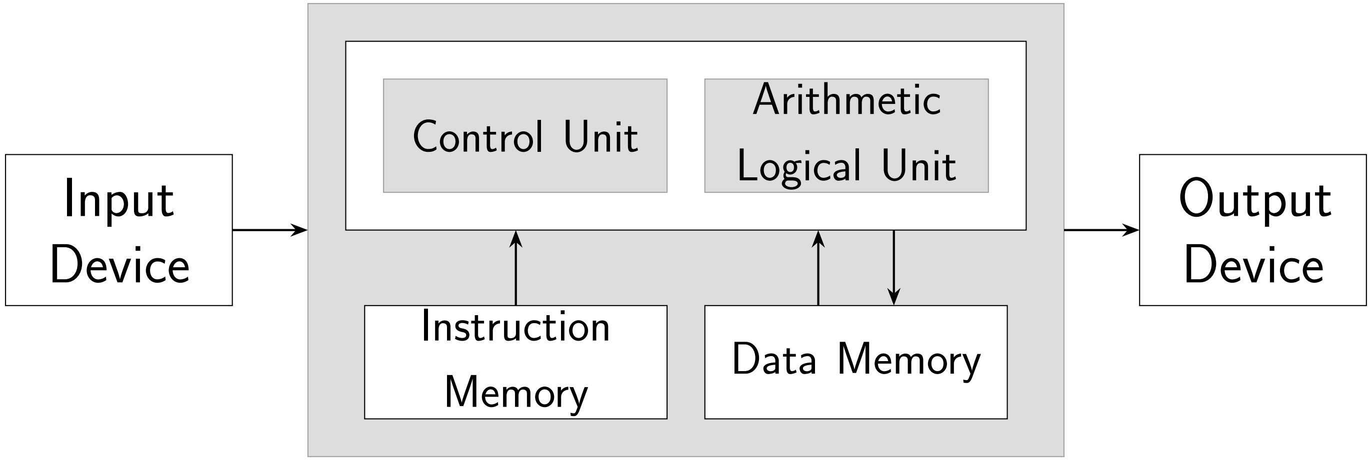
5 phases of execution

- Fetch instruction
- Decode instruction
- Execute
- Memory Access
- Write Back



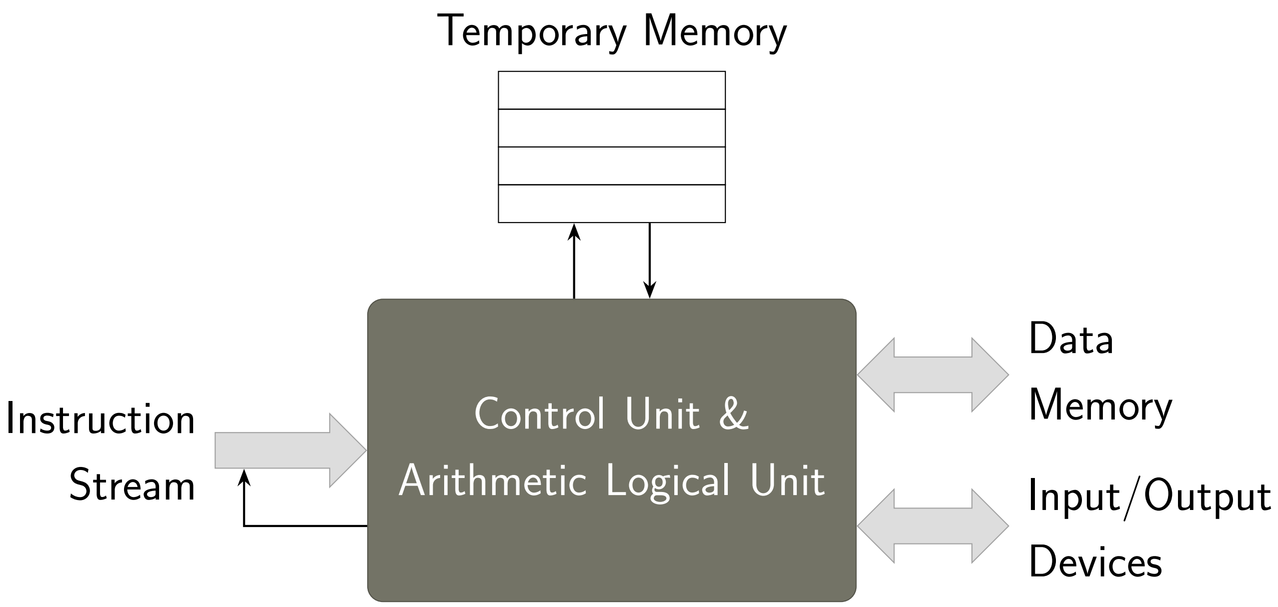
Harvard Architecture

- Separated data and instruction memory
- Today mostly "modified harvard architecture": Separated level 1 caches (see later)



Generic CPU Architecture

- Control unit and ALU as brain and heart of CPU
- Fetch instruction stream
- Diversion of (sequential) instruction stream
- Access to data memory
- Temporary memory much faster
- Often I/O interface



Instruction Set Architecture vs. Microarchitecture

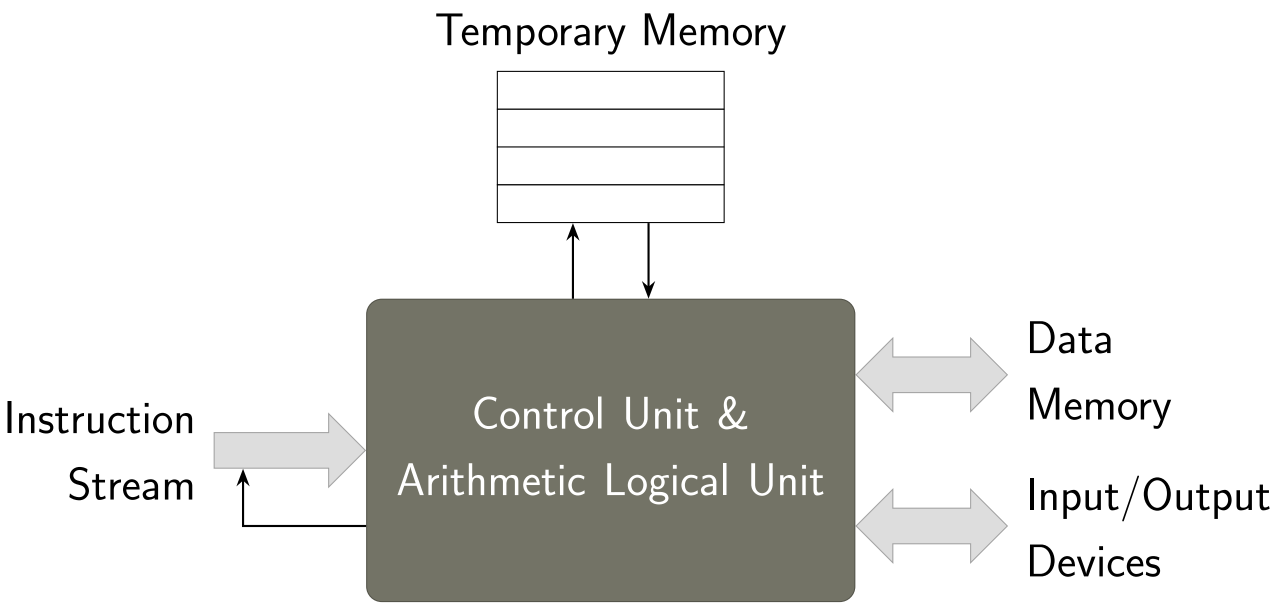
Instruction Set Architecture (ISA)

- General operation of a CPU
- "Contract" with programmer/compiler
- Defines instructions, states, memory access and interface to outside world
- Often many (optional) extensions

Microarchitecture

- Actual implementation of the ISA
- Many design alternatives and optimizations
- Must obey rules set out by ISA

Instruction Set Architecture vs. Microarchitecture



Examples: ISA and Microarchitecture 1/3

ISA	Manufacturer	Microarchitecture/Product
x86	Intel	8086, 80186, 80286, 80386, 80486, P5 (Pentium), P6 (Pentium II/III)
	AMD	8086, Am386, K5, K6, Athlon
	VIA/Cyrix	C3, C7
ia64	Intel/HP	Itanium, Itanium 2, Itanium 9300
x86-64	AMD	Opteron, Athlon 64, Turion, ..., Ryzen/Epyc
	Intel	Pentium 4, Xeon, Atom, Core 2, ..., Cannon Lake (e.g. Core i3 8121U)

Examples: ISA and Microarchitecture 2/3

ISA	Manufacturer	Microarchitecture/Product
ARMv2, .., ARMv6	ARM	ARM1, ARM2, ARM6, ARM7, .. ARM11
	DEC	StrongARM
	Intel	Xscale
ARMv7-A	ARM	Cortex-A5, Cortex-A7, ..., Cortex-A17
	Qualcomm	Krait, Scorpion
	Apple	A6
ARMv8-A	ARM	Cortex-A35, Cortex-A53, ..., Cortex-A76
	Qualcomm	Kryo
	Apple	A7, A8, A9, A10, A11, A12
	Samsung	M1, M2

Examples: ISA and Microarchitecture 3/3

ISA	Manufacturer	Microarchitecture/Product
MIPS64	MIPS	5K, 20K, Warrior-P, Warrior-M, ...
	Broadcom	BCM1125H, BCM1255
	Cavium	Octeon (CN30xx, CN31xx, ...), ...
m68k	Motorola	680x0
SPARC V7, V8, V9	Sun	SPARC, UltraSPARC, UltraSPARC II, ...
	Fujitsu	SPARClite, microSPARC II, ...
VAX, Alpha, PA-RISC, AVR, ...		

RISC-V Instruction Set Architecture

- Started as academic project at UC Berkeley (Asanovic/Patterson)
- Open instruction set architecture
- Widely adopted in industry
- Clean and clear ISA
- Open and proprietary implementations

➔ Used in course labs

INSTRUCTION SET ARCHITECTURE

Example: RISC-V®

Register and Register Files

Registers are the fastest memory elements of a CPU (much faster than memory access)

Differentiation in ISA

General Purpose Register (GPR): Intermediate results of program execution

Special Purpose Register (SPR)/Control and Status Register (CSR)

- Instruction pointer/Program counter
- Stack pointer, frame pointer
- Status registers (flags)
- Link register
- Index register (for address calculations)

Register and Register Files RISC-V®

32 General purpose registers

- Register 0 always tied to 0 (not writeable)
- Standard naming: `x0` .. `x31`
- Semantic use of registers (link register, stack pointer, ..), see later

Size of registers depends on ISA variant

- RV32, RV64, RV128: 32-bit, 64-bit or 128-bit registers
- Generalized naming of register size: `XLEN`

Control and Status Registers (CSR)

- Up to 4,096 registers, organized in groups
- Different access rights depending on processor mode

Instructions

Instructions are fetched by processor

Instructions are essentially data in memory:
Interpretation of instruction coding defined by ISA

Sequential execution

- Instructions are stored sequentially in memory
- Program counter points to current instructions
- CPU decodes an instruction and executes it

Variation in instruction stream

- Control unit can change the program counter non-sequentially
- Program flow: Loops, branches, function calls
- Exception handling of synchronous and asynchronous exceptions

Instruction Types

Four basic types of instructions

1. Integer Computational Instructions
2. Memory access (load/store)
3. Control flow
4. Input/Output

Operands

- Maximum number of operands per instructions
  - Important for arithmetic and logical operations
  - Influences the length of instructions
- Maximum number of memory addresses of those operands (typical: 1)

Integer Computational Instructions RISC-V

Basic Addition and Subtraction

- `add rd, rs1, rs2` (rd = rs1 + rs2)
- `sub rd, rs1, rs2` (rd = rs1 - rs2)

Basic Logical Operations

Example: `a = b + c - d`

Instruction Coding RISC-V®

Instructions are encoded in a unified format

- Standardized fields at same positions reduce the hardware overhead
- Assembler: Generate instructions from *mnemonics*

Example: `add x9, x20, x21`

31 … 0

funct7	rs2	rs1	funct3	rd	opcode
0	21	20	0	9	51
7 Bit	5 Bit	5 Bit	3 Bit	5 Bit	7 Bit
0000000	10101	10100	000	01001	0110011

0x015A04B3 ← 0000000 10101 10100 000 01001 0110011

`sub x9, x20, x21`

0100000	10101	10100	000	01001	0110011

0x415A04B3 ← 0100000 10101 10100 000 01001 0110011

"R" Format Instruction Coding

Constants/Immediates RISC-V®

Constants are often needed Examples: Offset in data structure, loop increment, etc.

How to add a constant?

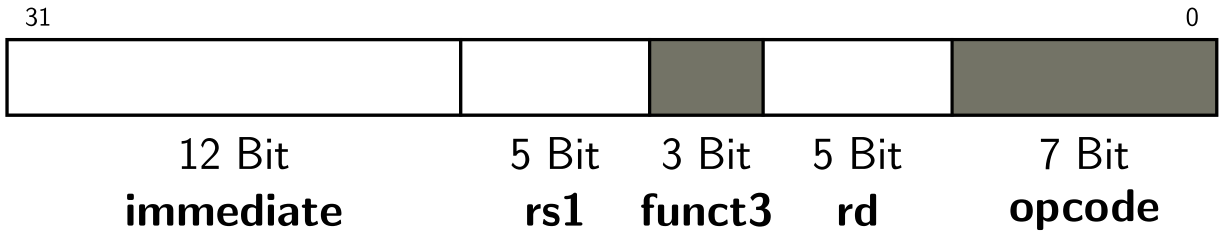
Problem with known instruction: Need constant in register. From memory?

Better solution: Encode into instructions with immediate

- `addi rd, rs1, imm` (rd = rs1 + imm)
- `andi rd, rs1, imm` (rd = rs1 & imm)
- `ori rd, rs1, imm` (rd = rs1 | imm)
- `xori rd, rs1, imm` (rd = rs1 ^ imm)

Immediate Instruction Coding RISC-V®

- Opcode and 3-bit function field at same position
- Source register and destination address at same position
- So called "I"-Format
- Immediate in 12 bit: Two's complement (-2^{11} to $2^{11} - 1$)



Shift Operations RISC-V®

Logical shifts:

- `sll rd, rs1, rs2` (rd = rs1 << rs2)
- `slli rd, rs1, shamt` (rd = rs1 << shamt)
- `srl rd, rs1, rs2` (rd = rs1 >> rs2)
- `srli rd, rs1, shamt` (rd = rs1 >> shamt)

Arithmetic shifts (preserves sign):

- `sra rd, rs1, rs2` (rd = rs1 >> rs2)
- `srai rd, rs1, shamt` (rd = rs1 >> shamt)

31					0
7 Bit	5 Bit	5 Bit	3 Bit	5 Bit	7 Bit
type	**shamt**	**rs1**	**funct3**	**rd**	**opcode**

Instruction Length & Code Size

Code size is a function of

- Number of instructions in program
- Length of instructions (shorter ⇒ smaller code size)
  (RISC-V basic instruction set: 32-bit, RISC-V "compact" extension: 16 bit)
- Density of instruction set ("more dense" ⇒ less instructions)

Instruction length is a function of

- Number of operands
- Number of instructions in ISA (increases opcode length)
- Special operands, especially constants

Often: Variable instruction length

- Common instructions in short form with less bits
- RISC-V: Compact (C) ISA extension, ARM: Thumb ISA extension

Instruction Set Complexity

Conflicting goals

- More instructions in ISA $\Rightarrow$ less instructions needed to complete certain task
- More instructions in ISA $\Rightarrow$ increased length of instructions

Instruction Set Complexity: CISC

- *Complex* Instruction Set Computer
- Instructions cover multiple operations
- Example:
  1. Load data from memory
  2. Arithmetic operation with two registers and this data
  3. Write other register value to memory address computed by this operation
  4. Increment value in register
- Problems: Hardware complexity
- Examples: x86, most computers until 1990s

Instruction Set Complexity: RISC

- *Reduced* Instruction Set Computer
- Small number of instructions, low instruction complexity
- Limited number of operands: max. 3 (destination + 2 sources)
- Most instructions are register-register operations
- *Load-Store architectures*: only a few, simple memory-register instructions

- Introduced as Berkeley RISC and Stanford MIPS in the 1980s
- Examples that follow the RISC paradigm: ARM, SPARC, MIPS, PowerPC, RISC-V
- Under the hood modern CISC processors are actually RISC processors: newline Translation of CISC commands into RISC microcode

Instruction Set Complexity: Video

Krste Asanovic - RISC-V: Instruction Sets Want To Be Free, MeetBSD 2016

`https://youtu.be/QTYiH1Y5UV0?t=371`

(6:11 to 9:16 are of interest in this context, but the entire video is a great watch!)

Memory Access

Transport data between memory and registers

Main memory is required as temporary storage, registers are limited

Difference main memory and long time storage (disk) later in course

Properties of memory access (endianess, alignment) and operation (adressing modes, instructions)

Memory Access: Endianess

Order of data in memory

Big Endian

- Byte with most significant bit at lowest memory address
- Can be found in: AVR32 (Arduino), network protocols

Little Endian

- Byte with most significant bit at highest memory address
- Can be found in: x86, x86-64, ARM, RISC-V

Memory Access: Granularity and Alignment

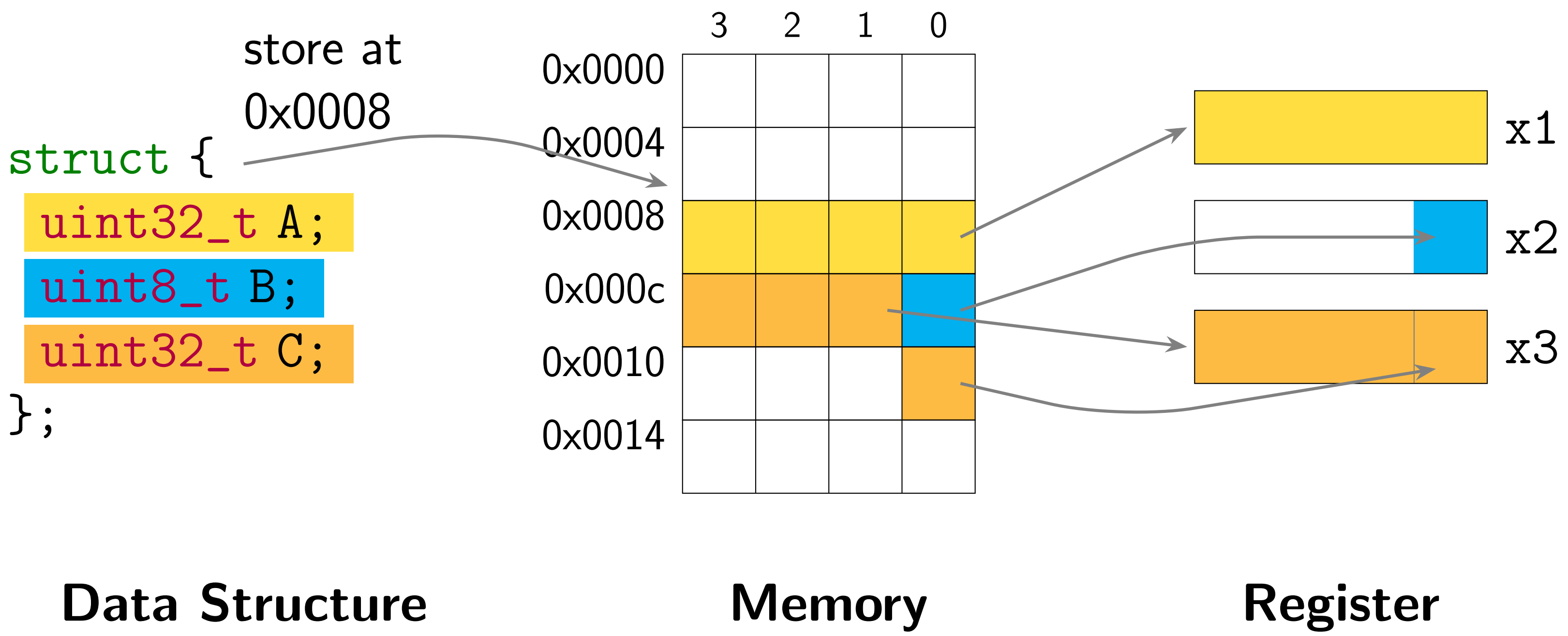
Granularity of memory and memory access

- Memory organized as blocks: 1 Byte, 2 Byte, 4 Byte, 8 Byte
- Transport between register and these blocks in memory
- Often: Memory block size == `XLEN`

Alignment: How data is stored in memory

- Data structures can be arbitrarily stored in memory
- Alignment: Base address and size of data item
- A data item is *misaligned* when it spans multiple memory blocks

Alignment: Example



Memory Address: Addressing Modes

Generally three options where *operands are loaded from* and *results are stored*

1. From the **instruction word**:
   Immediate in arithmetic/logical operation, offsets
2. From a **register**
3. From **memory**:
   Memory address that actual access is to: *effective address*

Example Motorola 68000 "full-relative mode":

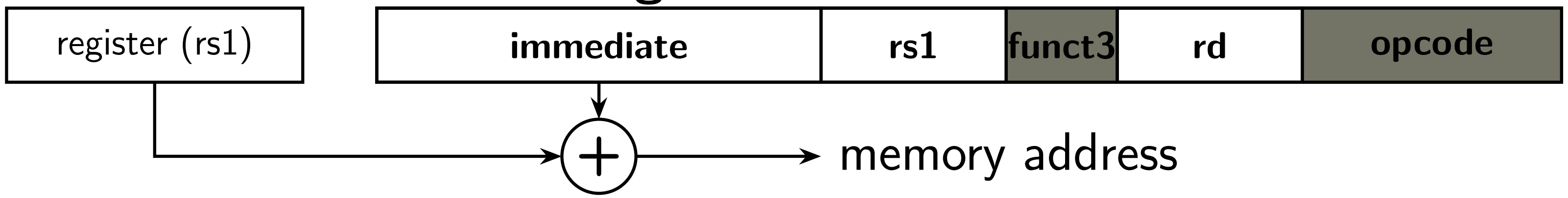
```
SUB 5(A3,D0), (A1) ➔ Mem[A1] = Mem[A1] - Mem[A3+D0+5]
```

RISC concept limits memory operands to transport operations (load-store architecture)

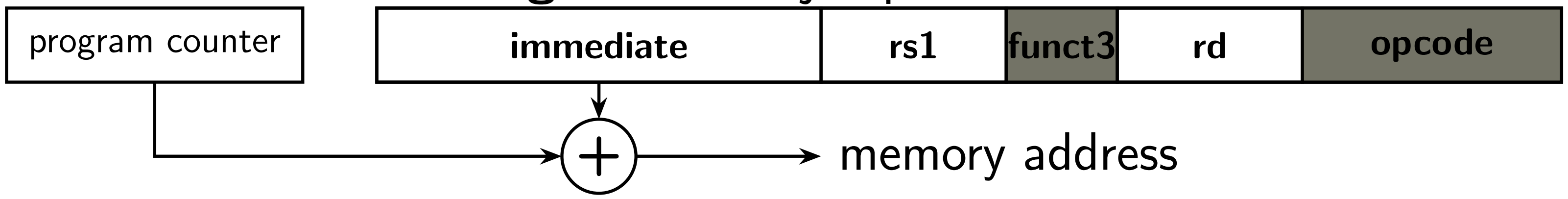
Memory Access in RISC-V®

Two adressing modes

- **Base-and-offset addressing mode** for data accesses



- **PC-relative addressing mode** for jumps



No alignment required, either hardware supports misaligned or software emulation

Memory Access: Load Instructions RISC-V®

Different granularities (byte, half-word, word, double-word) and signedness

- `lb rd, imm(rs1)` (rd = {sign,Mem[rs1+imm](7:0)})
- `lbu rd, imm(rs1)` (rd = {0,Mem[rs1+imm](7:0)})
- `lh rd, imm(rs1)` (rd = {sign,Mem[rs1+imm](15:0)})
- `lhu rd, imm(rs1)` (rd = {0,Mem[rs1+imm](15:0)})
- `lw rd, imm(rs1)` (rd = {sign,Mem[rs1+imm](31:0)})
- `lwu rd, imm(rs1)` (rd = {0,Mem[rs1+imm](31:0)})
- `ld rd, imm(rs1)` (rd = {sign,Mem[rs1+imm](7:0)})

Immediate ("I") instruction coding

Memory Access: Store Instructions RISC-V

Notation similar to load instructions

- `sb rs1, imm(rs2)` (Mem[rs1+imm](7:0) = rs2(7:0))
- `sh rs1, imm(rs2)` (Mem[rs1+imm](15:0) = rs2(15:0))
- `sw rs1, imm(rs2)` (Mem[rs1+imm](31:0) = rs2(31:0))
- `sd rs1, imm(rs2)` (Mem[rs1+imm](63:0) = rs2(63:0))

Special instruction format

31					0
7 Bit	5 Bit	5 Bit	3 Bit	5 Bit	7 Bit
immediate	**rs2**	**rs1**	**funct3**	**imm.**	**opcode**

Comparisons

Comparisons are needed in programming (depend execution on data)

Condition Codes

- Flags that are implicitly set by arithmetic or logical operations
- Examples: **Z**ero flag, **C**arry flag, **N**egative flag, **O**verflow flag
- Those flags are architecture state (part of state registers)
- Commonly used for control flow instructions (see later)

Predications

- Execute operation only if flag is set
- Alternative to control flow instruction
- x86: `CMOV` instruction (conditional move)
- ARM: Most instructions have `cond` field in machine code (mnemonic: suffix)

```
cmp   r1, r2
subgt r1, r1, r2
```

ARM Predication

Comparison in RISC-V

RISC-V does not have condition codes or predication

Instructions to set `rd` to 1 iff condition is true, else 0

- `slt rd, rs1, rs2` (`rd = (rs1<rs2) ? 1 : 0`)
- `sltu rd, rs1, rs2` (`rd = (rs1<rs2) ? 1 : 0`, unsigned)
- `slti rd, rs1, imm` (`rd = (rs1<imm) ? 1 : 0`)
- `sltiu rd, rs1, imm` (`rd = (rs1<imm) ? 1 : 0`, unsigned)

Reasoning why only "less than"

- Remember: Limited coding space
- Set less than considered most useful
  - Greater than and comparisons to zero are easy
  - Typical boundary checks
- Observation: Other comparisons (==, <=, >=) commonly used with branches

Control Transfer Instructions

Change of instruction stream

Need to change the program counter

Unconditional control transfer instructions (jumps), example: function calls

Conditional control transfer instructions (branches), example: loops, if-then-else

Branches in RISC-V®

Comparison of two registers and change program counter iff condition is met

- `beq rs1, rs2, offset` (`if rs1==rs2 then pc+=offset`)
- `bne rs1, rs2, offset` (`if rs1!=rs2 then pc+=offset`)
- `blt rs1, rs2, offset` (`if rs1<rs2 then pc+=offset`)
- `bge rs1, rs2, offset` (`if rs1>=rs2 then pc+=offset`)
- `bltu` and `bgeu` accordingly

"B" Instruction format

31 … 0

7 Bit	5 Bit	5 Bit	3 Bit	5 Bit	7 Bit
imm	**rs2**	**rs1**	**funct3**	**imm**	**opcode**

Jumps in RISC-V

Branches are generally limited in distance (±4kB), larger jumps as unconditional control transfer instructions (±1MB)

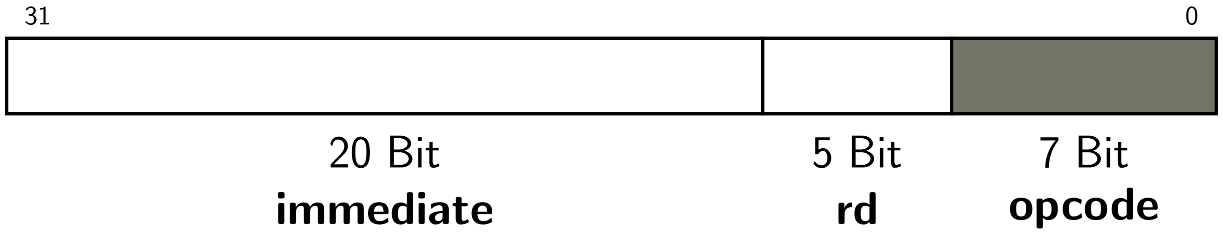
Jump and Link

- `jal rd, offset` (rd=pc+4, pc=pc+offset)
- `jalr rd, offset(rs1)` (rd=pc+4, pc=rs1+offset)

Link: Store address of next instruction

- Makes information available at destination where jump came from
- Main use in function calls (return address), `x0` as link register: simple "jump"

"J" format for `jal`, (known) "I" format for `jalr`



Large Constants and Addresses

Limitations of immediate operations and branches/jumps

- Only 12 bit immediate, how to set 32 bit?
- Only 20 bit jump offset, how to jump in large programs?

Immediate and offset sizes are *limited* by instruction size
usually: `XLEN` $\geq$ instruction size

ISAs provide operations for **constant/address formation** to solve the issue

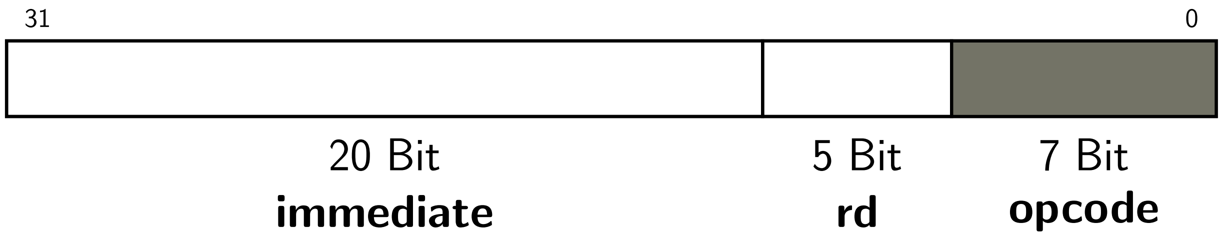
Large Constants in RISC-V®

Special "upper" instructions: Load upper part of register

- `lui rd, imm` (load upper immediate, `rd=imm,0`)
- `auipc rd, offset` (add upper immediate to pc, `pc=pc+offset,0`)

Reduces the number of instructions needed to form constants/addresses

"U" format



Summary of instruction formats

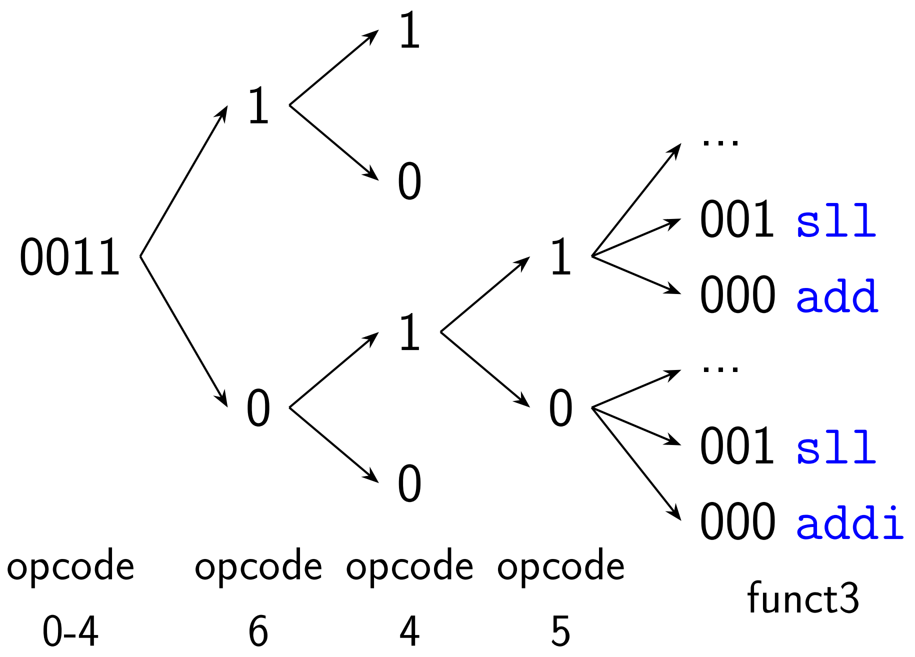
funct7	rs2	rs1	funct3	rd	opcode	**R** format
immediate		rs1	funct3	rd	opcode	**I** format
immediate		rs1	funct3	imm.	opcode	**S** format
immediate		rs1	funct3	imm.	opcode	**B** format
immediate				rd	opcode	**U** format
immediate				rd	opcode	**J** format

Differences between I/S/B and U/J in arrangement of bits throughout immediates: optimized to support hardware implementation

Instruction Anatomy in RISC-V

Formats/instruction coding are optimized for hardware design

Example of a decoding



The opcode and function bits can be used to directly drive control lines (multiplexers, etc.)

APPLICATION BINARY INTERFACE

Example: RISC-V®

Application Binary Interface

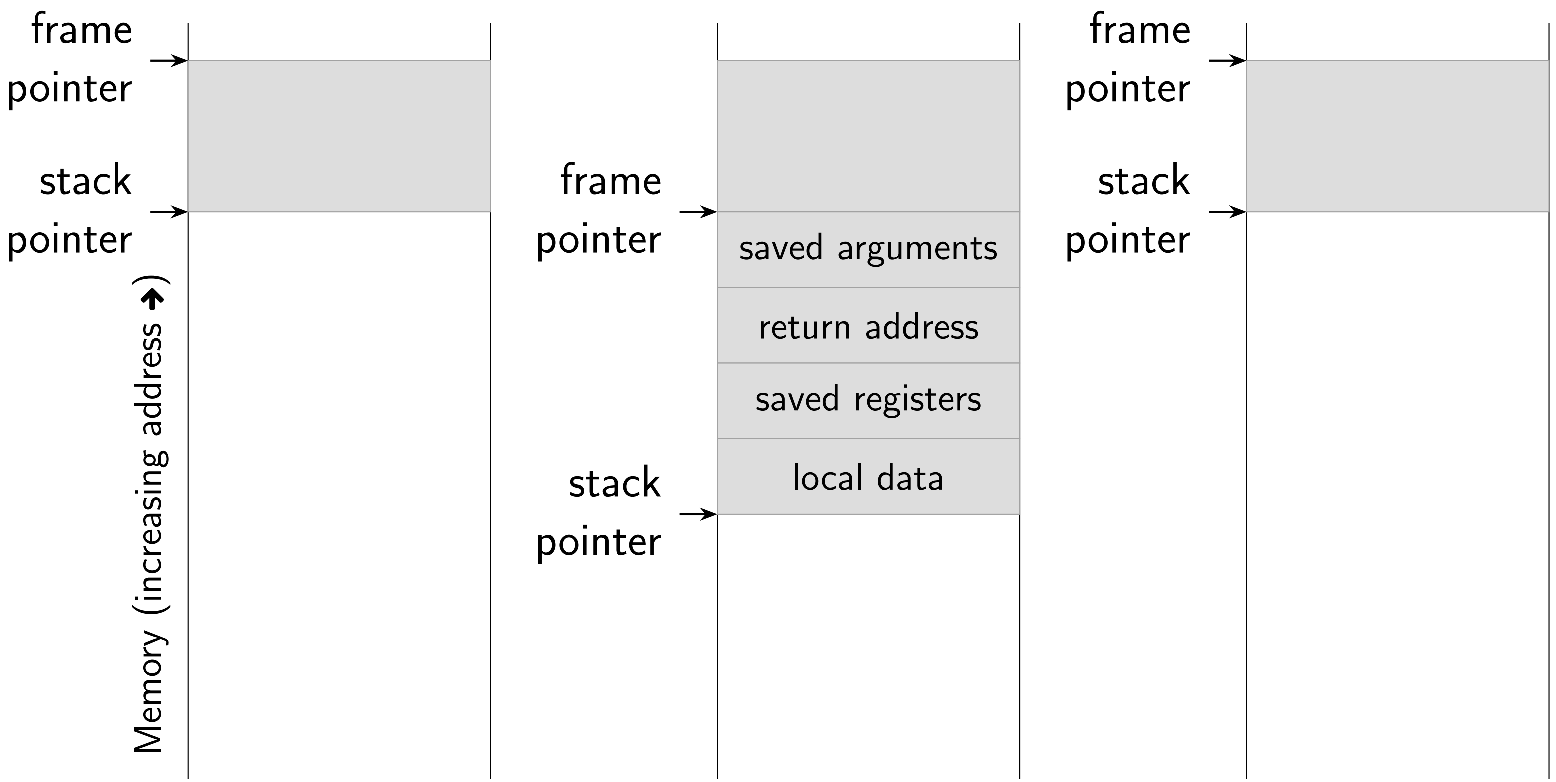
ABI defines **interoperability** of binary programs: operating system, library, etc.

- Size and alignment of data types
- Calling conventions define how programs call functions in other binary programs
- System calls to the operating system

Calling conventions and the **stack** are generally defined for software
Required so that compiler can generate programs, **standardized** for interoperability

ABI adds **semantics** to instructions that is reflected in register "ABI names"
Examples in RISC-V: `a0` for `x10` as argument register, `t0` for `x5` as temporary, `zero` for `x0`, see chapter 25 in ISA spec

Stack Frame



Before function call | **During function call** | **After function call**

Calling Conventions

Specified per ISA

Defines the flow of function calls

- In which registers arguments are stored (RISC-V: `a0-a7`)
- How extra arguments are given (RISC-V: stack pointer points to next argument)
- In which registers results are returned (RISC-V: `a0-a1`)
- Which register contains the return address (RISC-V: `ra`)
- Which registers are saved by the caller or the callee (RISC-V: see next)

Calling Convention in RISC-V

Register values must be preserved during function calls

Define which saved by calling function (caller) and called function (callee)

- **Callee-saved**
  Stack pointer (sp/x2), "saved registers" (s0-s11/x8,x9,x18-x27)
- **Caller-saved**
  Return address (ra/x1), arguments (a0-a7/x10-x17), "temporary registers" (t0-t6/x5-x7,x28-x31)

Function call (generic, differs for leaf functions and can be optimized)

- Caller saves caller saved registers on the stack
- Caller calls function with `jal(r)`
- Callee saves stack pointer on stack reserves space on stack, saves callee saved if needed
- Callee function..
- Callee restores return address and executes `jalr` with it as target

Other Things

Assembler **mnemonic pseudoinstructions** (aliases)

- Defined by the instruction manual (Chapter 25)
- Expand to other assembler instruction or sequence of instructions
- Examples: no operation `nop`, load immediate `li rd, imm`, move `mv rd, rs`

Some assembler programs (e.g., GNU AS in our lab) provide convenient use
For example: generic `add x2, x1, -2` as alias for `addi`

PRIVILEGE LEVELS AND EXCEPTIONS

Example: RISC-V®

Prof. Dr.-Ing. Stefan Wallentowitz – Department 07 – Munich University of Applied Sciences

Interaction with environment

Programs commonly run in an environment, for example:

- **Baremetal environment**: Direct access to hardware
- **Operating system environment**: Access abstracted and multiplexed by operating system or runtime system
- **Virtualization environment**: Computer shared by multiple operating systems

Basic abstraction principle:

- Execution environments abstract from underlying hardware
- Potentially protects from malicious code controlling the system with **privileges**

Application Execution Environment RISC-V®

Applications usually have an underlying execution environment

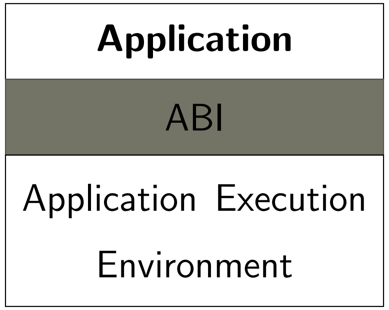
Most fundamental is application execution environment

- Provides system functions (such as I/O)
- No actual operating system, but basic abstraction

Recap: ABI

- Interoperability between software pieces of application
- Access to system functions via **system calls**

Very common AEE (also in our lab): simulators



Supervisor Privileges

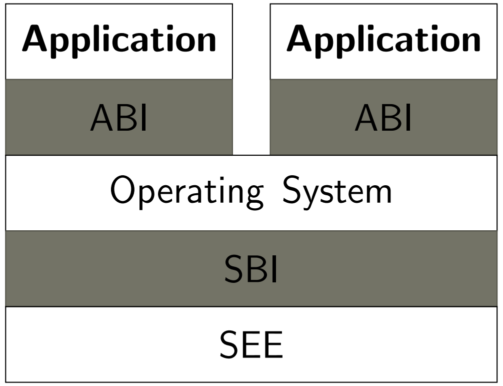
Extend AEE with multitasking

- Provide each application the impression it is running alone
- Strong separation properties
- Fundamental functionality of an **operating system**

Differentiation between application and **supervisor** privileges

RISC-V: Supervisor execution environment (SEE)

- Provides abstraction from hardware platform (portability) via *Supervisor Binary Interface*
- Basic SEEs: BIOS-style IO system, boot loader



Hypervisor Privileges

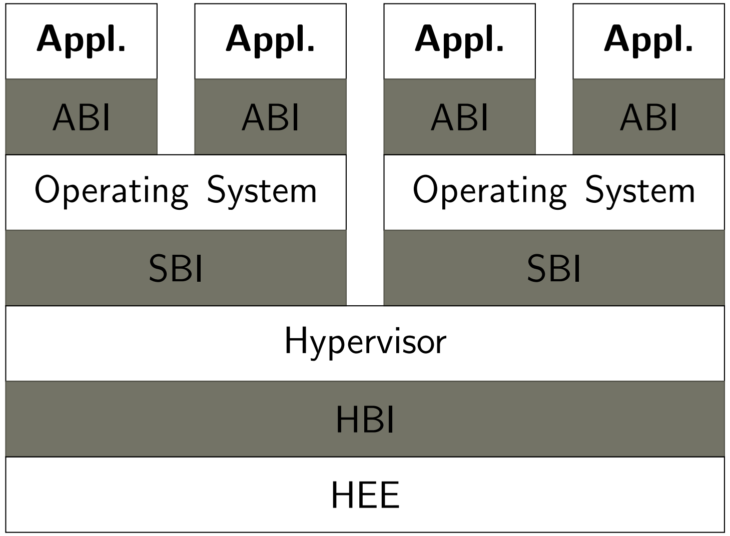
Hypervisor: SEE multiplexes between multiple operating systems

- Relevant system functions accessed via SBI
- Full system virtualization

Differentiation between application, supervisor and **hypervisor** privileges

RISC-V: Hypervisor execution environment (HEE)

- Portability by *Hypervisor Binary Interface* (HBI)



Privilege Levels

Privileges of application and different execution environment managed by **privilege levels*

Differences:

- Control and Status Registers (CSRs) depend on privilege level
- Access to hardware resources managed by privilege levels

Privilege levels are encoded in the *CPU mode* (RISC-V: U-mode, S-mode, M-mode)

Switch between privilege levels have to be explicit, example RISC-V:

- `ecall` leaves current mode and traps to next lower mode
- return from trap in each mode with `mret`, `sret`, `uret`

Exceptions

Exceptions: "Disturbance" in instruction stream by an event

- *Synchronous* exceptions: exceptions that relate to an instruction (divide by zero, page fault, etc.)
- *Asynchronous* exceptions/*interrupts*: external events (such as I/O)

Performance improvement: delegation

- Let "higher" modes handle exceptions
- Delegation reduces overhead of switching, typical example: Let guest OS in virtualization directly handle page fault and not fault to machine mode

Summary

Key takeaways

- Generic model of a processor
- Difference between ISA and Microarchitecture
- RISC vs. CISC
- Basic instruction set architecture, example RISC-V