Computer Organization and Networks

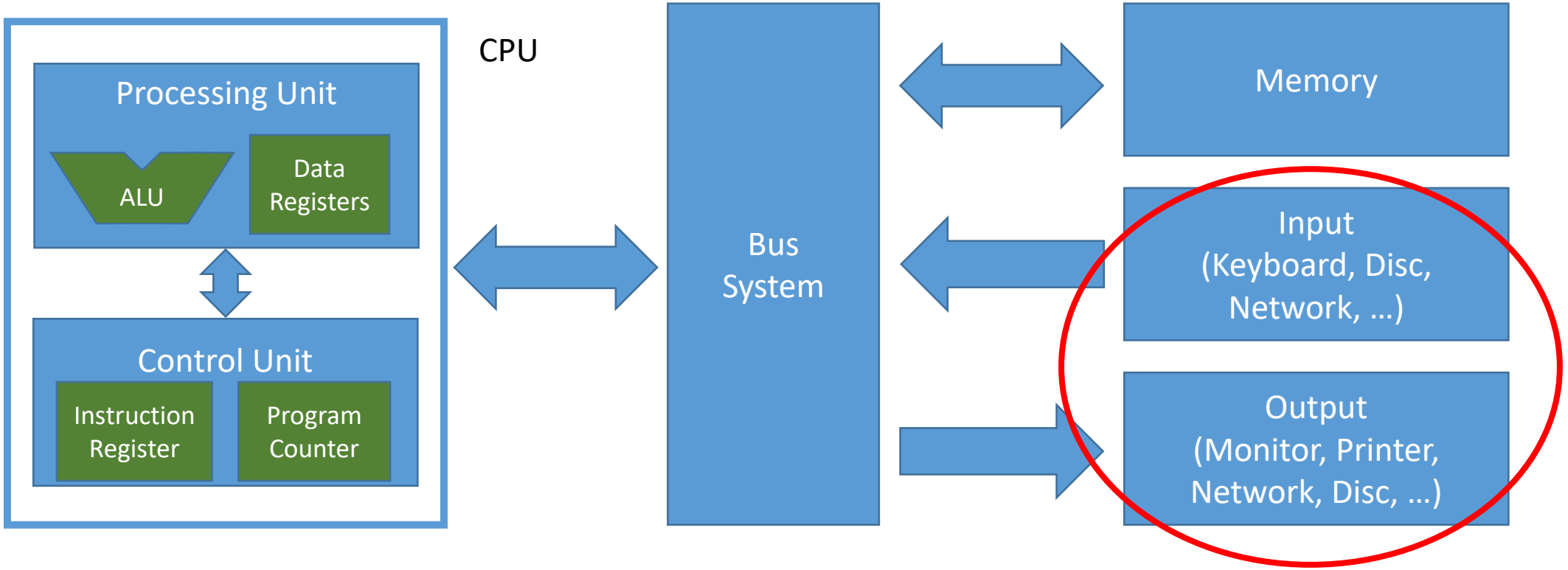
(INB.06000UF, INB.07001UF)

Chapter 6: Peripherals and Interrupts

Winter 2022/2023

Stefan Mangard, www.iaik.tugraz.at

How to Implement I/O?



RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

How to Implement I/O?

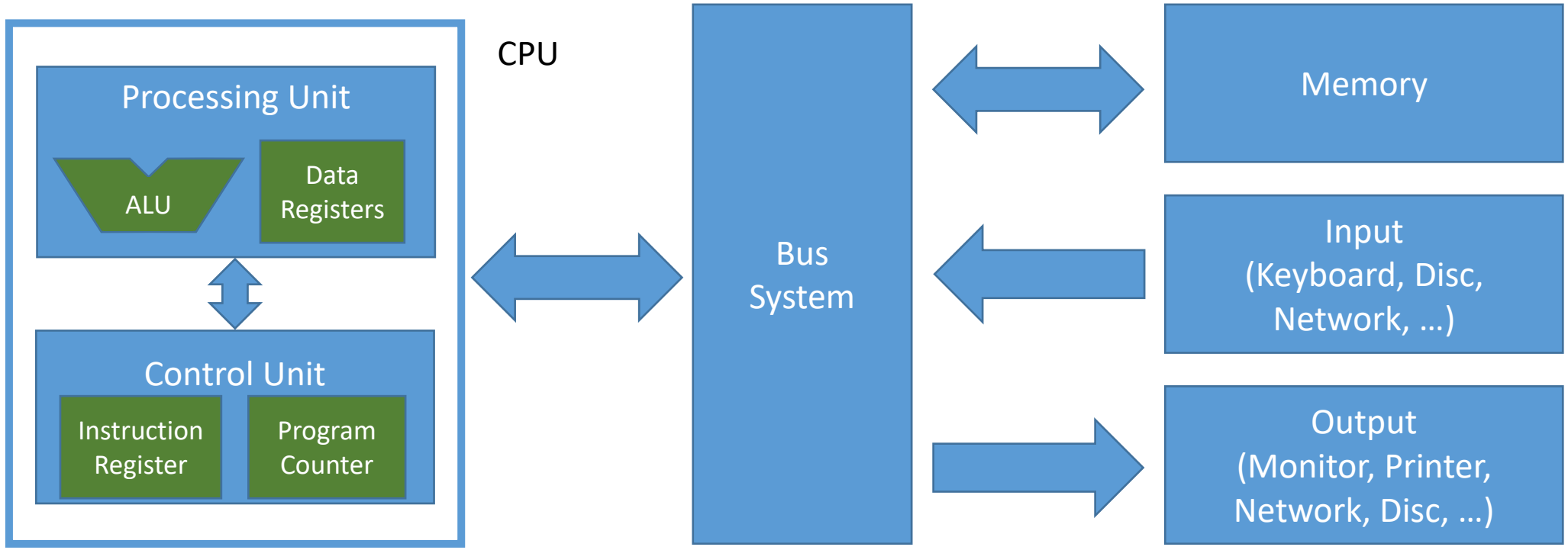
- We access I/O and other devices like memory

→ we build memory-mapped peripherals

Memory-Mapped Peripherals

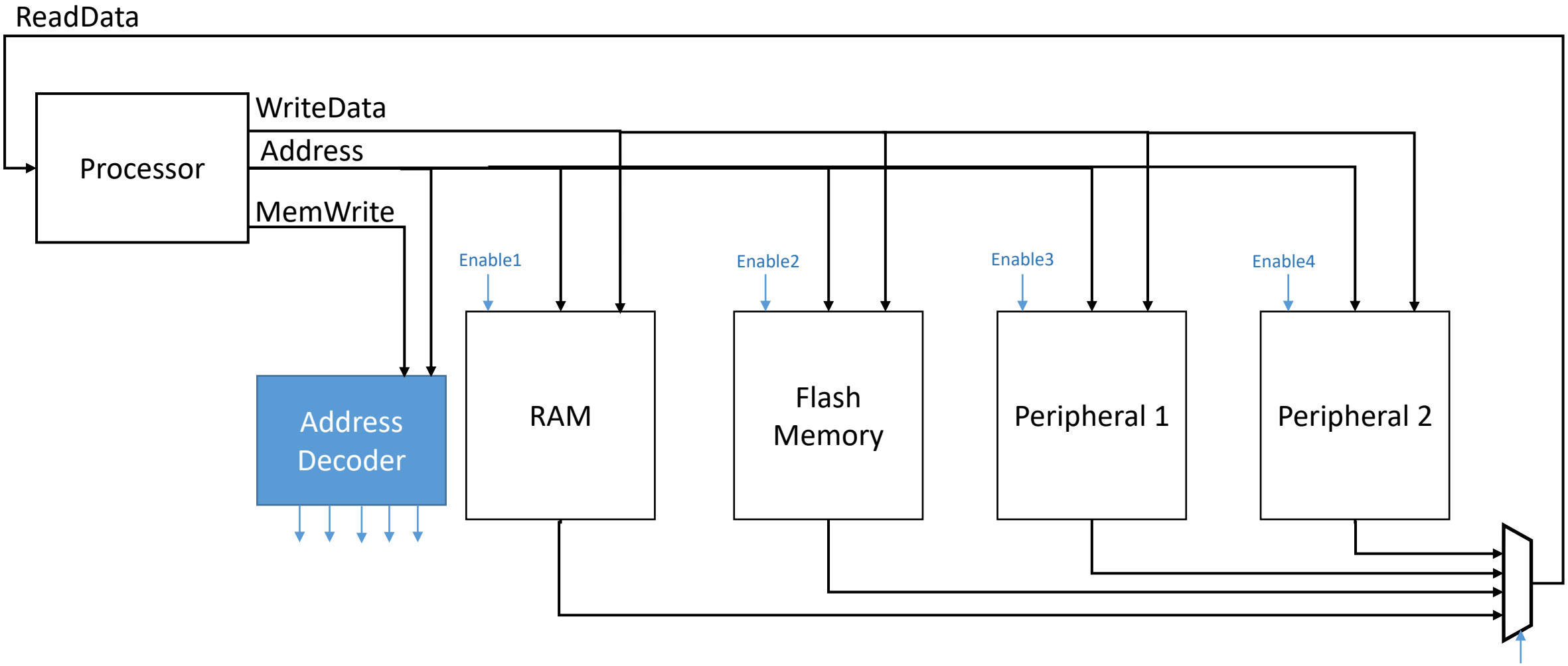
- Store and load instructions allow addressing 32-bit of memory space
- Not all the memory space that is addressable is used for actual memory
- We can split the memory space in pieces and assign a certain range to actual memory and other ranges to peripherals:

  → load/store operations write to registers of state machines with additional functionality (I/O, Co-processors, sound, graphics, …)



The bus system takes care of routing the load/store operations to the correct physical device as defined by the memory ranges

The Hardware View



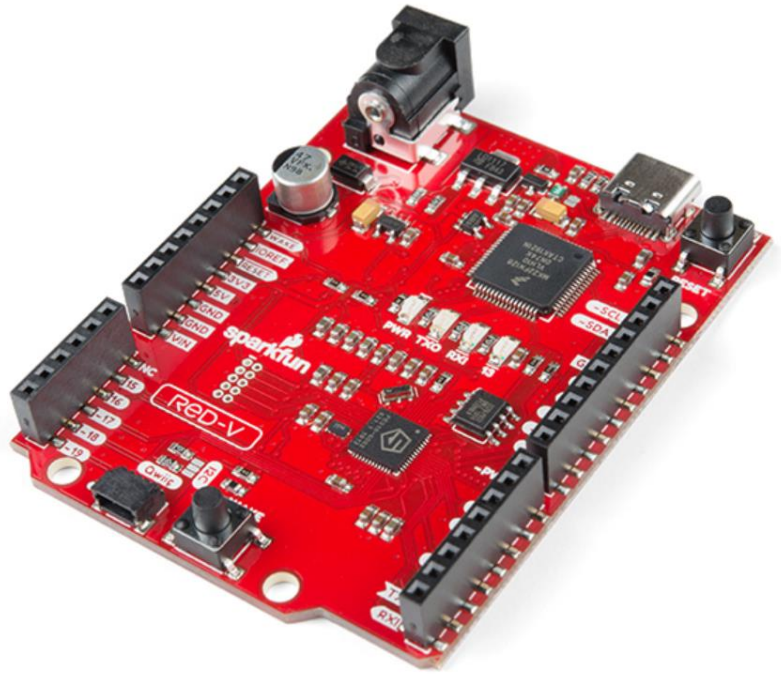
Memory Mapping – Different for different Systems

- The memory map is not part of the instruction set architecture and it is also not defined by RISC V
- There are commonalities, but in the end the memory map is individual for every device

Example Memory Map

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	Debug Address Space
0x0000_1000	0x0000_1FFF	R XC	Mode Select	On-Chip Non Volatile Memory
0x0000_2000	0x0000_2FFF		Reserved	
0x0000_3000	0x0000_3FFF	RWX A	Error Device	
0x0000_4000	0x0000_FFFF		Reserved	
0x0001_0000	0x0001_1FFF	R XC	Mask ROM (8 KiB)	
0x0001_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region	
0x0002_2000	0x001F_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	On-Chip Peripherals
0x0201_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x0800_1FFF	RWX A	E31 ITIM (8 KiB)	
0x0800_2000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	AON	
0x1000_1000	0x1000_7FFF		Reserved	
0x1000_8000	0x1000_8FFF	RW A	PRCI	
0x1000_9000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	
0x1001_3000	0x1001_3FFF	RW A	UART 0	
0x1001_4000	0x1001_4FFF	RW A	QSPI 0	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0x1001_7000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	

Table 4: FE310-G002 Memory Map. Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable, **A** - Atomics



The FE310-G002 supports booting from several sources, which are controlled using the Mode Select (`MSEL[1:0]`) pins on the chip. All possible values are enumerated in Table 5.

MSEL	Purpose
00	loops forever waiting for debugger
01	jump directly to 0x2000_0000 (memory-mapped QSPI0)
10	jump directly to 0x0002_0000 (OTP)
11	jump directly to 0x0001_0000 (Mask ROM: Default Boot Mode)

Table 5: Boot media based on `MSEL` pins

Micro RISC-V

- Micro RISC-V is a very simple CPU that we use for our introductory programming examples
- Micro RISC-V implements a subset of R32I
- Tools and code for micro RISC-V
  - Code for Micro RISC-V and examples are available in the examples-2021 repo
  - Assembler: riscvasm.py
  - Simulator: riscvsim.py

Micro RISC-V Overview

Registers:

- Zero Register: `x0`
- General Purpose Registers: `x1` - `x31`

Memory:

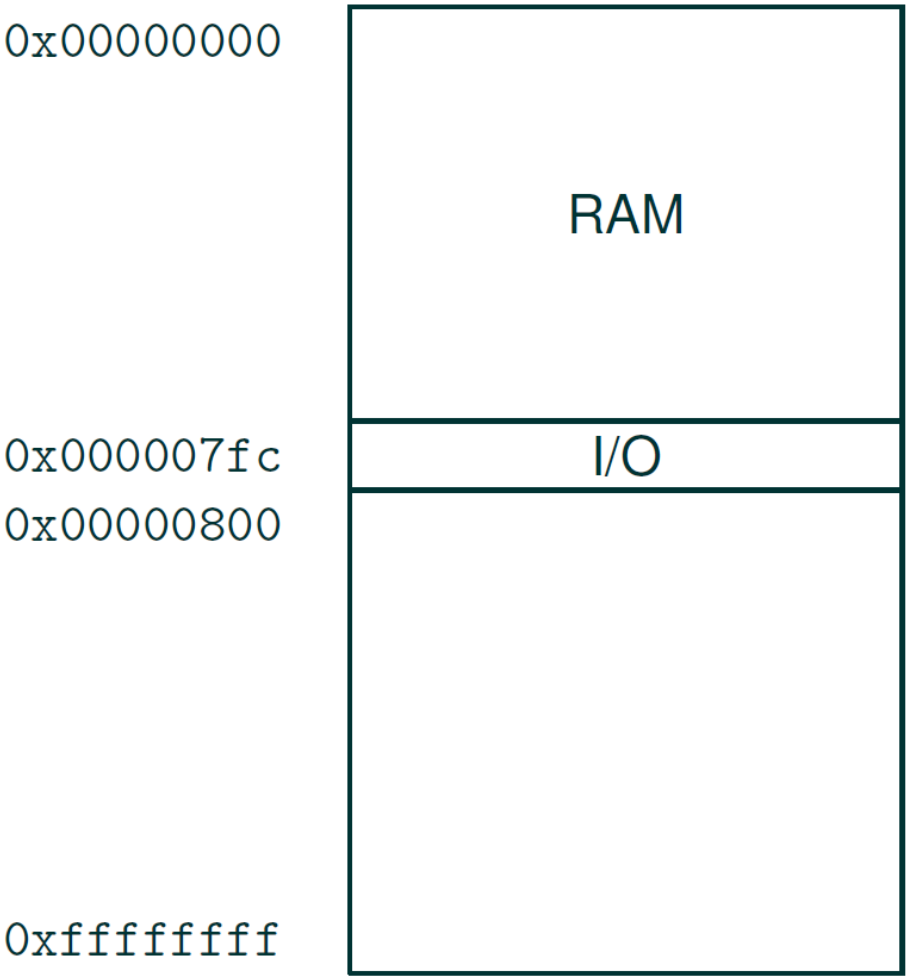
- almost 2 KiB of Memory (`0x000` - `0x7fc`)
- memory-mapped I/O at address `0x7fc`

Instructions:

- ALU: `OP rd, rs1, rs2`
  - ADD, SUB, AND, OR, XOR, SLL, SRL, SRA
- Add immediate: `ADDI rd, rs1, value`
- Load upper immediate: `LUI rd, value`
- Branch: `OP rs1, rs2, offset`
  - BEQ, BNE, BLT, BGE
- Jump / Call: `JAL rd, offset`
- Jump / Call indirect: `JALR rd, offset(rs1)`
- Load: `LW rd, offset(rs1)`
- Store: `SW rs2, offset(rs1)`
- Halt: `EBREAK`

Memory Map in Micro RISC-V

- In Micro RISC-V, the physical memory map is as follows:
  - RAM is located from 0x00000000 to 0x000007FB
  - I/O is located at address 0x000007FC
  - The remaining memory range is not connected (write has no effect; read returns 0)
- The physical memory map is defined for each device depending on size of memory, peripherals, etc.



Programming in Assembly

Note on ASM Examples

- Run “make” to generate .hex files
- Run “make run” to assemble and run the .asm file in the current working directory with the RTL simulator (micro-RISCV)
- Run “make sim” to simulate the .asm file in the current working directory with the python asmlib RISC-V simulator

If there are more than one asm files in the current working directory, you need to specify the target explicitly using “make run=the_asm_file_without_file_extension_suffix” (and accordingly for “make sim”).

Read/Write from Memory vs. Read Write from I/O

adding-two-constants

```
.org 0x00
  # read from memory
  LW x1, 0x20(x0)
  LW x2, 0x24(x0)
  ADD x3, x1, x2
  # write to memory
  SW x3, 0x24(x0)
  EBREAK
```

adding-stdin-numbers

```
.org 0x00
  # read from I/O
  LW x1, 0x7fc(x0)
  LW x2, 0x7fc(x0)
  ADD x3, x1, x2
  # write to I/O
  SW x3, 0x7fc(x0)
  EBREAK
```

Summing Up 10 Input Values

Sum up 10 numbers and print the result:

```
.org 0x00
  ADD x1, x0, x0 # clear x1

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2   # x1 += input
```

Let's improve this code using control flow instructions to build a loop

```
LW x2, 0x7fc(x0) # load input
ADD x1, x1, x2   # x1 += input

LW x2, 0x7fc(x0) # load input
ADD x1, x1, x2   # x1 += input

LW x2, 0x7fc(x0) # load input
ADD x1, x1, x2   # x1 += input

LW x2, 0x7fc(x0) # load input
ADD x1, x1, x2   # x1 += input

SW x1, 0x7fc(x0) # output sum
EBREAK
```

Loops

- start with the code for one iteration

```
.org 0x00
  ADD x1, x0, x0    # clear x1

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input

  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Loops

- start with the code for one iteration
- add loop variables

```
.org 0x00
  ADD x1, x0, x0    # clear x1
  ADD x3, x0, x0    # clear counter
  ADDI x4, x0, 10   # iteration count

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input




  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Loops

- start with the code for one iteration
- add loop variables
- increment the counter

```
.org 0x00
  ADD x1, x0, x0    # clear x1
  ADD x3, x0, x0    # clear counter
  ADDI x4, x0, 10   # iteration count

  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input

  ADDI x3, x3, 1    # counter++


  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Loops

- start with the code for one iteration
- add loop variables
- increment the counter
- branch to the start of the loop

```
.org 0x00
  ADD x1, x0, x0    # clear x1
  ADD x3, x0, x0    # clear counter
  ADDI x4, x0, 10   # iteration count

→ LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input

  ADDI x3, x3, 1    # counter++
  BLT x3, x4, ???   # if (counter < 10) loop

  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Loops

Counting offsets is not a nice job for a programmer

→ Let the compiler do it

- start with the code for one iteration
- add loop variables
- increment the counter
- branch to the start of the loop

```
.org 0x00
  ADD x1, x0, x0    # clear x1
  ADD x3, x0, x0    # clear counter
  ADDI x4, x0, 10   # iteration count

  L  x2, 0x7fc(x0) # load input
  ADD  , x1, x2    # x1 += input

  ADDI x3, x3, 1    # counter++
  BLT x3, x4, -12   # if (counter < 10) loop

  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Symbols

- Basic idea:
  - We label memory addresses
  - Each address we label is assigned a symbol (“a name”)

- When programming, we can replace memory addresses by symbols to simplify the complexity of programming

Loop Using a Label

```
.org 0x00
  ADD x1, x0, x0    # clear x1
  ADD x3, x0, x0    # clear counter
  ADDI x4, x0, 10   # iteration count
loop:
  LW x2, 0x7fc(x0) # load input
  ADD x1, x1, x2    # x1 += input

  ADDI x3, x3, 1    # counter++
  BLT x3, x4, loop # if (counter < 10) loop

  SW x1, 0x7fc(x0) # output sum
  EBREAK
```

Variables, Having Fun With the Memory Layout

```
1   # micro riscv loop demo with symbols
2       .org 0x00
3       JAL x0, main
4
5       .org 0x120
6   counter: .word 0
7   sum:     .word 0
8
9       .org 0x3a0
10  main:
11      ADDI x4, x0, 10         # iteration count
12                              #    +-----------------------+
13  loop_start:                 #    v                       |
14      LW x1, sum              # load sum                     |
15      LW x2, 0x7fc(x0)        # load input                   |
16      ADD x1, x1, x2          # sum += input                 |
17      SW x1, sum              # store sum                    |
18                              #                              |
19      LW x3, counter          # load counter                 |
20      ADDI x3, x3, 1          # counter++                    |
21      SW x3, counter          # store counter                |
22      BLT x3, x4, loop_start # if (counter < 10) goto loop_start
23
24      SW x1, 0x7fc(x0)        # output sum
25      EBREAK
```

- We can choose the memory layout as we like
- We can mix data and code
- Try it out with your own code

Pseudo-Instructions

To ease programming, there are pseudo-instructions for

- common instruction sequences and
- instructions that can be derived from another instruction

	Examples	
nop	addi x0, x0, 0	No operation
li rd, immediate	lui rd, imm[31:12] addi rd, rd, imm[11:0]	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
bgez rs, offset	bge rs, x0, offset	Branch if ≥ zero
bltz rs, offset	blt rs, x0, offset	Branch if < zero
bgtz rs, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if ≤
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if ≤, unsigned
j offset	jal x0, offset	Jump

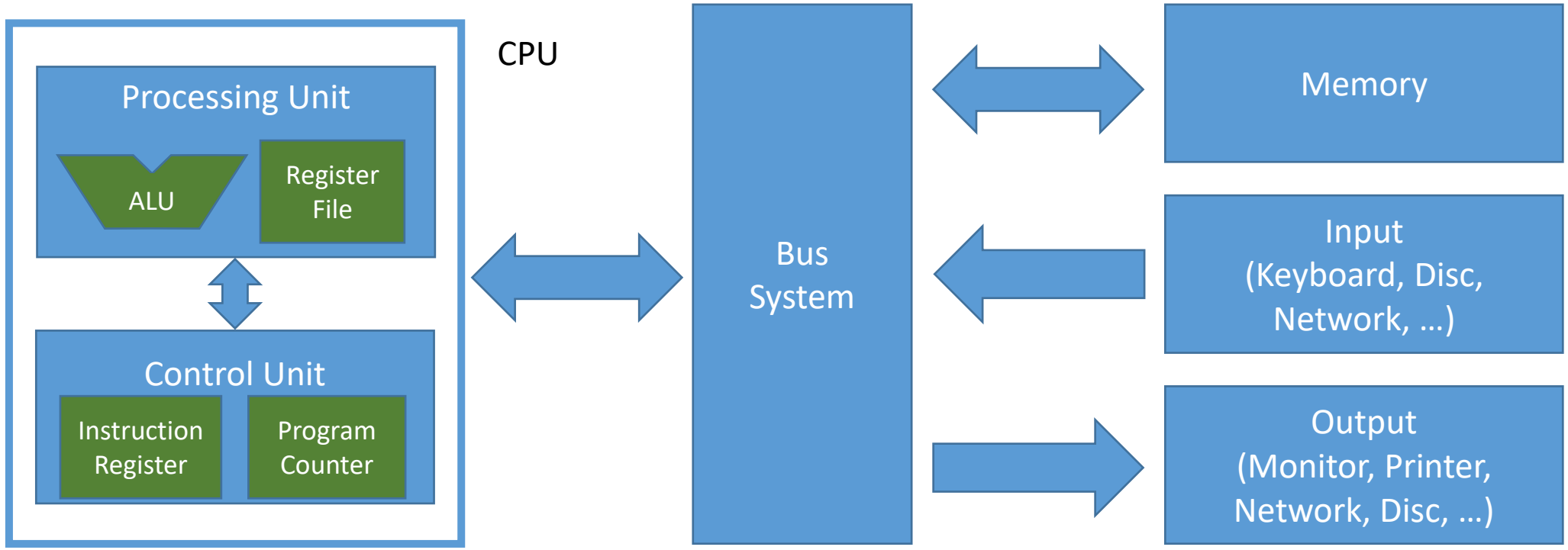
Communication Interfaces

Examples in QtRVSim

- https://comparch.edu.cvut.cz/qtrvsim/app/

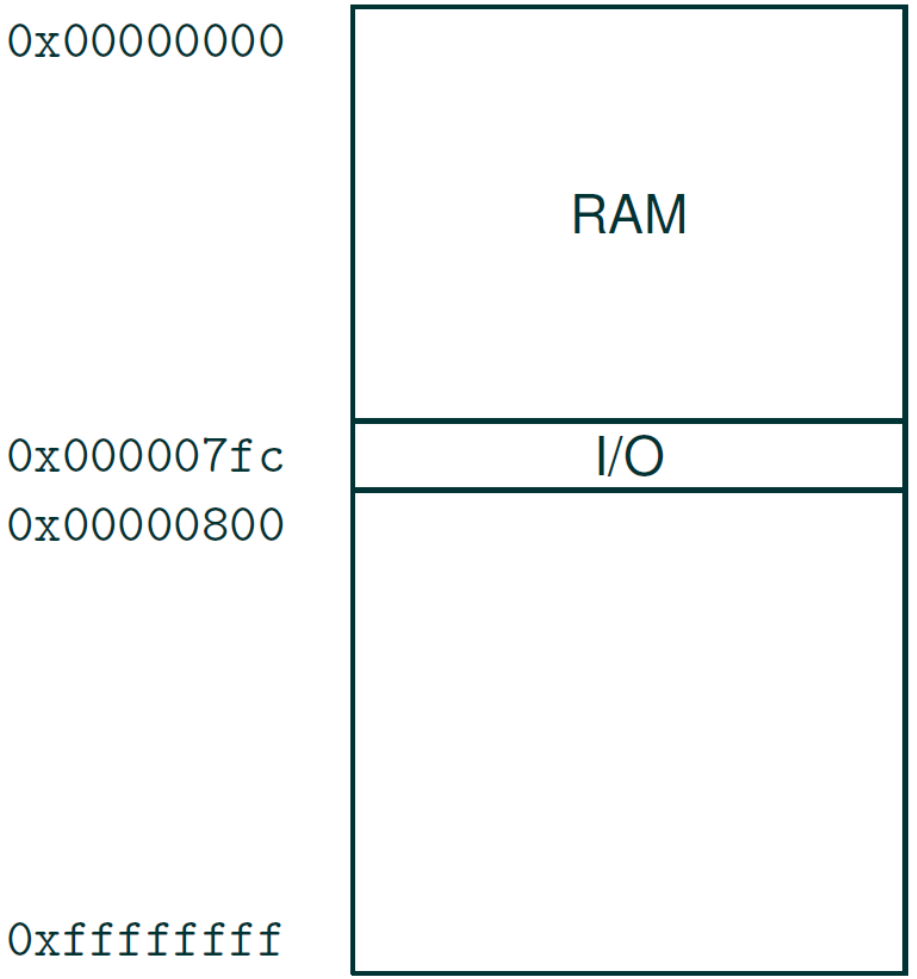
- Examples
  - 03_simple_write.S
  - 04_playing_with_knobs.S

Von Neumann Model



Our Example I/O

- The I/O interface that we discussed so far is idealized debug interface (data is always valid)
- In practice there is the following challenge:
  - The CPU executes one instruction after the other.
  - How should it know when the input is valid? Is it valid always (in every clock cycle)?



Example

- Assume an input port of a computer is set to a value 1 in one clock cycle
- It is still 1 in the next clock cycle
- Does this mean this is the “same” 1 or does this mean that there is a “second” 1?
- How should the computer know?

We Need to Add a Flag

No Mail for You

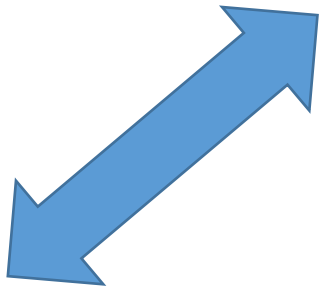
You've Got Mail

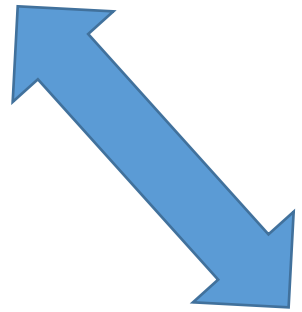
Synchronization with Control Signals

- On real communication channels, data is not always ready
- We need synchronization with control signals
- There exist different protocols and standards.
  - Serial protocols: RS232, SPI, USB, SATA, . . .
  - Parallel protocols: PATA/IDE, IEEE 1284 (Printer), . . .

- We use a simple interface with few control signals to illustrate this
  - 8-bit data port
  - Simple valid/ready flow-control
  - Registers (memory mapped)
    - 0x7D0 (control register)
    - 0x7D4 (data register)

Implementing an Interface With a Control Register



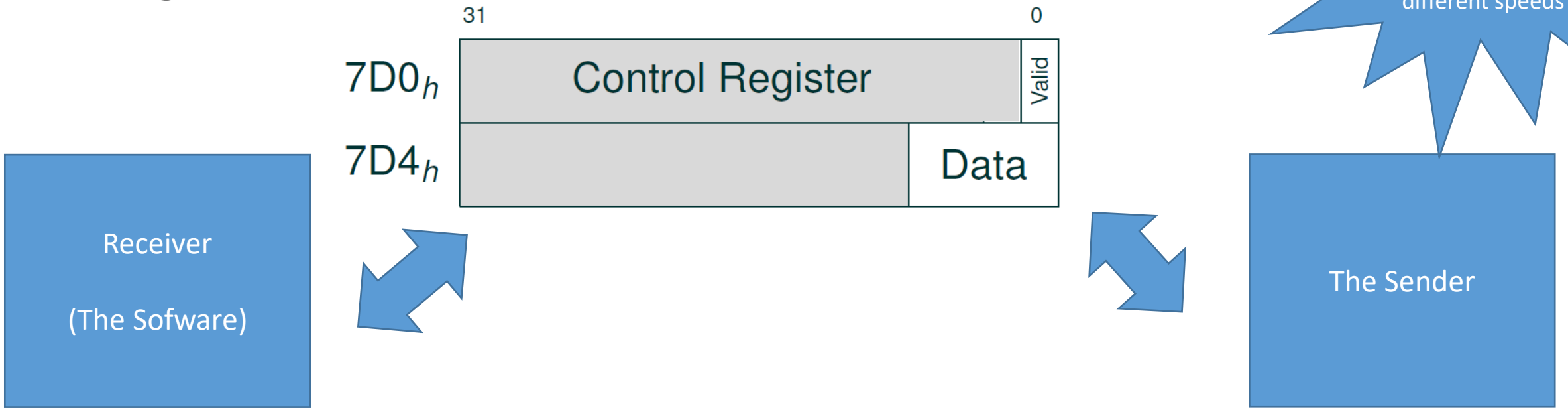


Receiver

(The Sofware)

The Sender

Implementing an Interface With a Control Register



Example of a basic protocol:

(4) Receiver (the software) waits until valid bit is set

(5) Receiver reads the data

(6) Receiver clears the valid bit

(1) Sender waits until valid bit is cleared (set to 0)

(2) Sender sets the data value

(3) Sender sets the valid bit

Polling Using a Control Register by the sender

Pseudoinstruction for

beq t1,zero, POLL_PARIN


```
POLL_PARIN:
  LW t1, 0x7D0(x0)     # Read PARIN CTRL REG
  ANDI t1, t1, 1
  BEQZ t1, POLL_PARIN

  LW t2,   0x7D4(x0)   # Read available data
  SW zero, 0x7D0(x0)   # Acknowledge Data
```

Control Signals

- There is a wide range of options for implementing communication between entities (FSMs, software, humans, …) of with different speeds

- However, in all cases, there needs to be signals to ensure that
  - The sender knows that the resource (bus, register, …) is available
  - The receiver knows that there is valid input
  - The sender knows that the receiver has received the signal (acknowledge)

Polling Example in QtRVSim

- https://comparch.edu.cvut.cz/qtrvsim/app/

- Example
  - 05_polling.S

Communication via a Slow Communication Interface

- Polling is highly inefficient: the CPU is stuck in a loop until e.g.
  - an I/O peripheral sets a ready signal
  - a timer has reached a certain value
  - the user has pressed a key
  - ….
- Alternative
  - CPU keeps executing some useful code in the first place
  - We use concept of interrupts to react to “unexpected” events
  - Basic idea: Instead of waiting for an event, we execute useful code and then let an event trigger a redirection of the instruction stream

How to handle unexpected external events?

- We add an input signal to the CPU called “interrupt”.
- An external source can activate this input signal “interrupt”.
- After executing an instruction, the CPU checks for the value of this input signal “interrupt” before it fetches the next instruction.
- If the signal “interrupt” is active, the next instruction to be executed is the first instruction of the “interrupt-service routine”.
- After “handling” the interrupt by executing the interrupt-service routine, the CPU returns to the interrupted program.

Interrupts in RISC-V

- **Hardware Aspects**
  - External interrupt is an input signal to the processor core
  - Control & Status registers (CSRs) for interrupt configuration (e.g. mie, mtvec, mip, …)
  - Additional instructions for interrupt handling (mret)
  - Dedicated interrupt controllers on bigger processors

- **Software Aspects**
  - When an interrupt occurs, the program execution is interrupted
  - Functions have to be provided to handle interrupts → Interrupt Service Routines (ISR)
  - Software needs to configure and enable interrupts
  - Software has to preserve the interrupted context
    → Interrupt entry points are typically written in assembly

Control & Status Registers (CSRs) in RISC-V

- We so far only considered memory-mapped peripherals whose registers can be accessed via standard load and store instructions

- RISC-V also features dedicated so called “Control & Status Registers”
  - The ISA allows addressing 4096 registers (32 bit each)

  - Dedicated instructions allow to read and write these registers: CSRRW, CSRRS, CSRRC, CSRRWI, CSRRSI, CSRRCI

The Interrupt Service Routine (ISR)

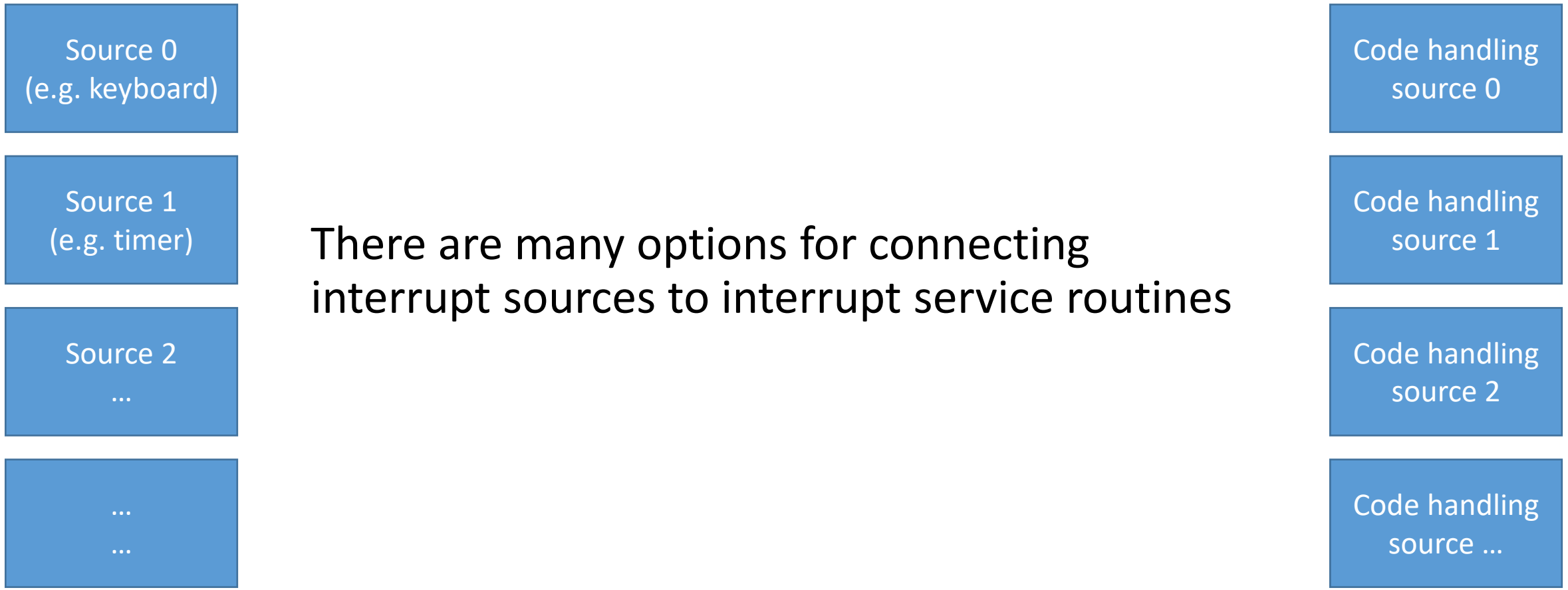
- Entering the ISR
  - Upon an interrupt, the processor
    - jumps to a location in memory specified by the **mtvec** CSR.
    - automatically stores the previous location into **mepc** CSR.
- Executing the ISR
  - The ISR can execute arbitrary code; However, the processor context (program counter, register) needs to have exactly the same values when returning to the interrupted code → “From the view of the interrupted program, the execution after the interrupt continues as if nothing had happened”
- Leaving the ISR
  - Upon the execution of the **mret** instruction, the processor
    - returns to the original location stored in the mepc CSR

Finding the Interrupt Service Routine

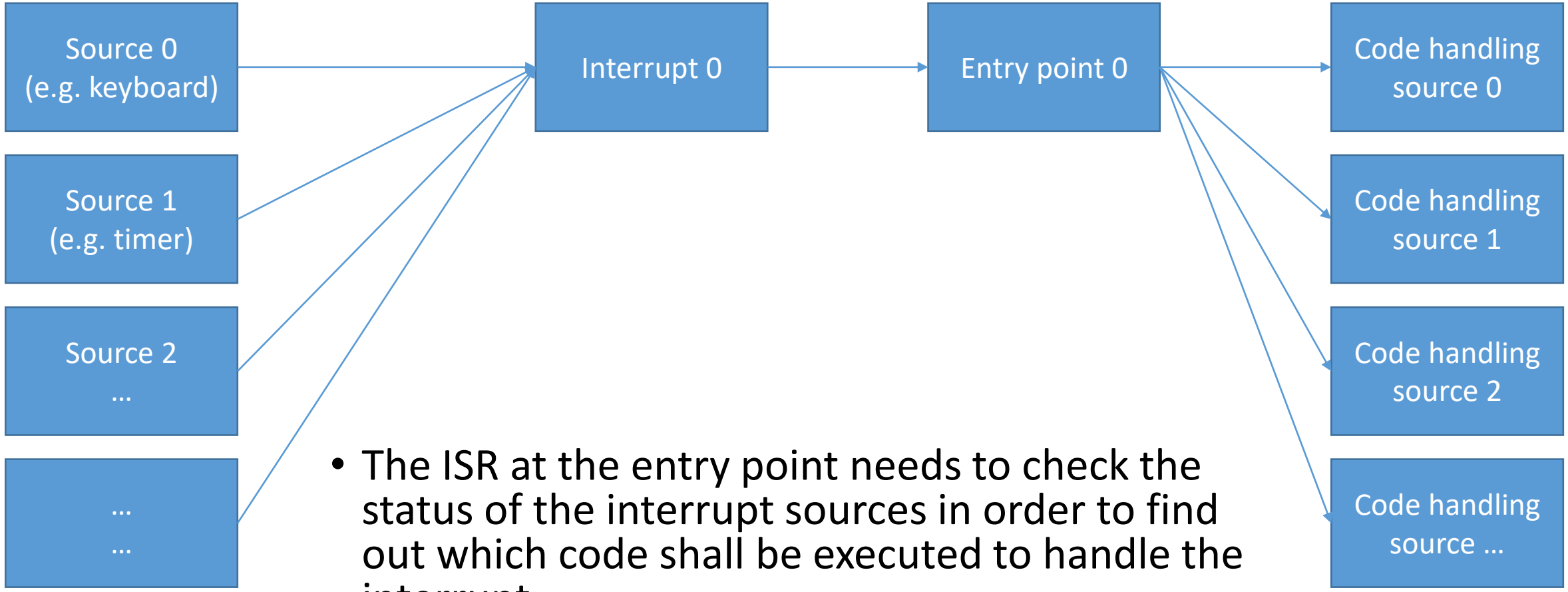
- Two approaches are common:
  - Single entrypoint for all interrupts.
    - the ISR has to determine what caused the interrupt and then handles the corresponding interrupt
  - Multiple entrypoints for different interrupts organized in a table (**vectored interrupts**)
    - A table defines the entry point for different causes of interrupts
    - E.g. each interrupt vector table entry has 4 bytes
      - Interrupt cause 0 leads to a jump to mtvec
      - Interrupt cause 1 leads to a jump to mtvec+4
      - Interrupt cause 2 leads to a jump to mtvec+8
      - …

      →just enough space to place a single **jal** instruction to the actual ISR handler code at each entry location
- RISC-V permits both approaches

Connecting Interrupt Sources to Interrupt Service Routines

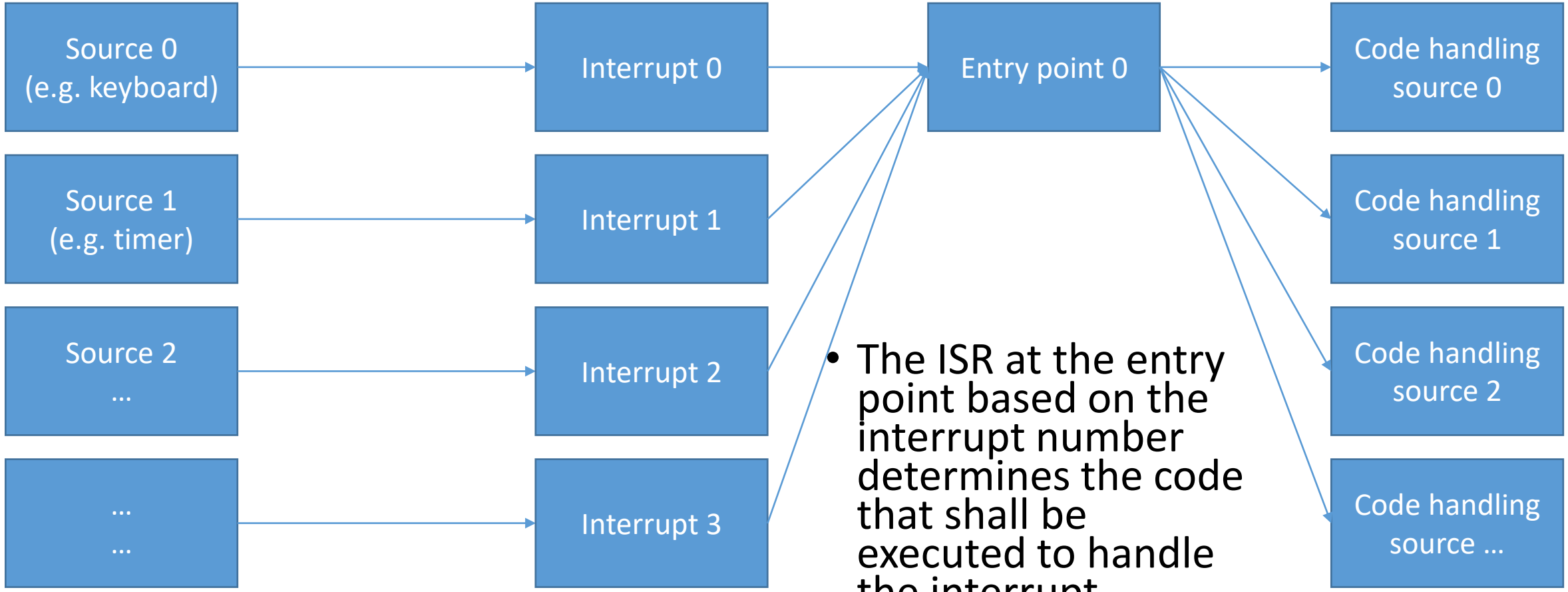


Connecting Interrupt Sources to Interrupt Service Routines (one Interrupt)



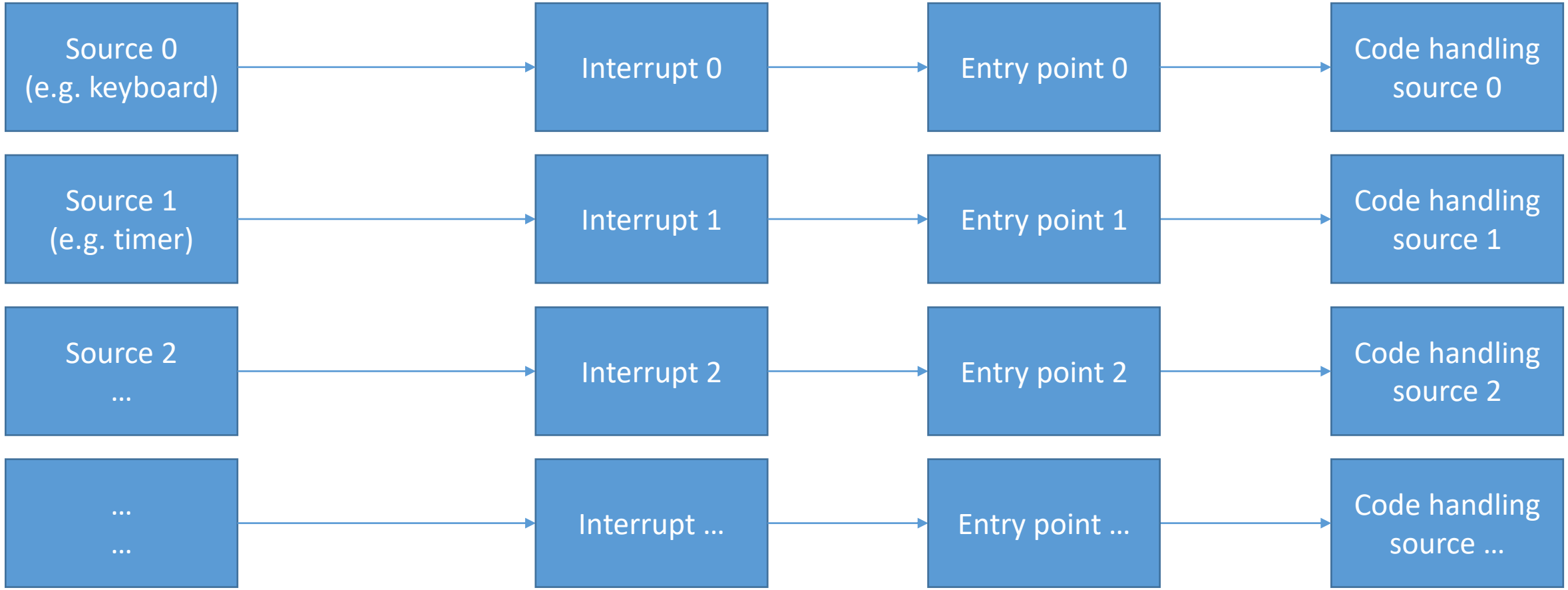
- The ISR at the entry point needs to check the status of the interrupt sources in order to find out which code shall be executed to handle the interrupt

Connecting Interrupt Sources to Interrupt Service Routines (one entry point)



- The ISR at the entry point based on the interrupt number determines the code that shall be executed to handle the interrupt

Connecting Interrupt Sources to Interrupt Service Routines (vectored approach)



- Vectored handling with different entry points for different interrupts

Connecting Interrupt Sources to Interrupt Service Routines

- In practice all kinds of combinations are possible for interrupt handling
- There is also the option for having interrupts with different priorities
- Dedicated interrupt controllers are available on larger systems to handle priorities, entry points, nested interrupts, …