

# Computer Organization and Networks

(INB.06000UF, INB.07001UF)

## Chapter 4: Processors

Winter 2023/2024



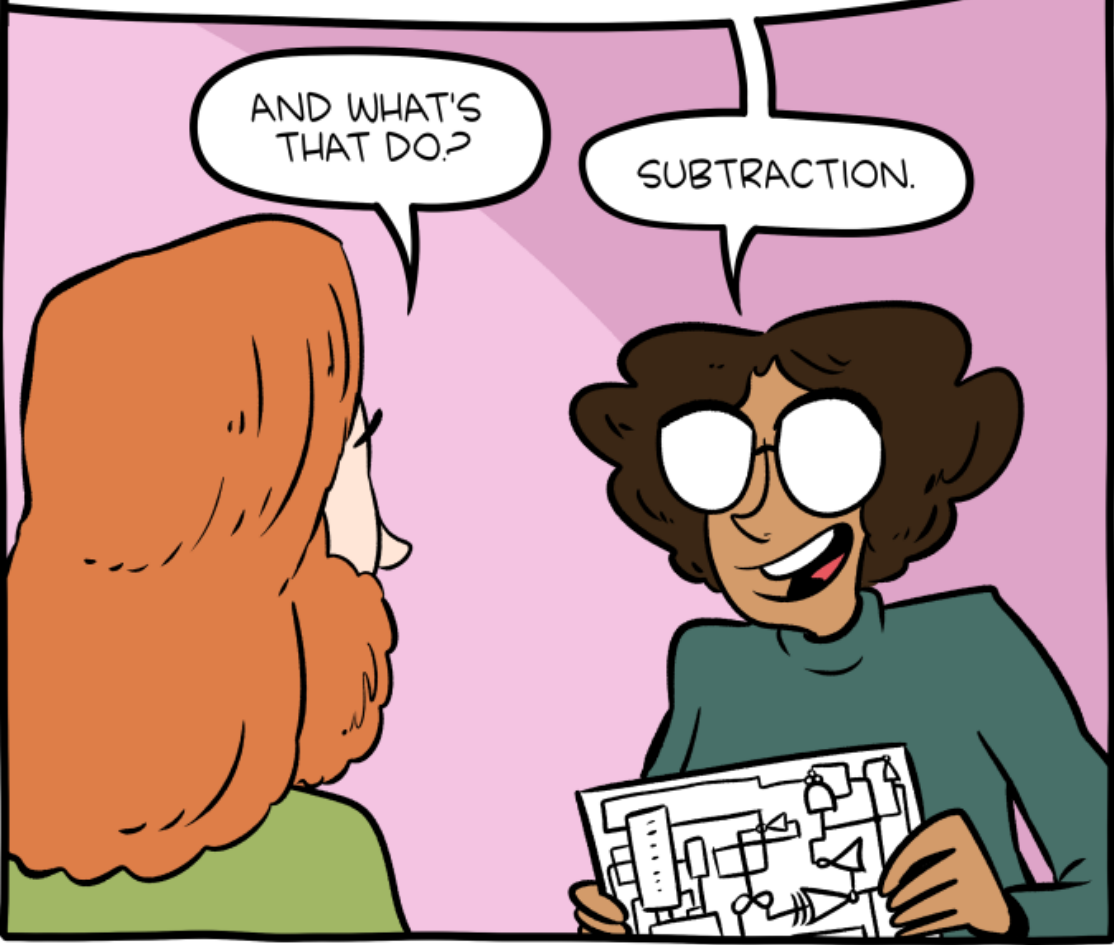
Stefan Mangard, [www.iaik.tugraz.at](http://www.iaik.tugraz.at)

THIS IS WHAT LEARNING LOGIC GATES FEELS LIKE

SEE, YOU JUST CONNECT THIS 12 INPUT REVERSE FLIP-FLOP TO THE CONTROLLED TWO-THIRDS ADDER, WHICH RESETS THE LATCHES IN THE NOT-NAND RELAY ARRAY, THEN LOOP BACK TO ODD-NUMBER INPUTS AND REVERSE ALL YOUR SWITCHES!

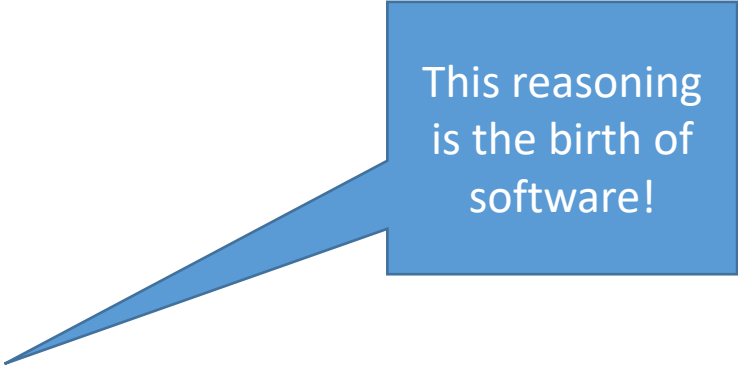
AND WHAT'S THAT DO.?

SUBTRACTION.



# Limitations of State Machines Discussed So Far

- The State machines that we have discussed so far have been designed for a specific application (e.g. controlling traffic lights)
  - Changing the application requires building a new state machine, new hardware, ...
  - We want to have a general-purpose machine that
    - Can be used for all kinds of different applications
    - Can be reconfigured quickly
- We want **general purpose hardware** that is “configured in memory” for a particular **application**



This reasoning  
is the birth of  
software!

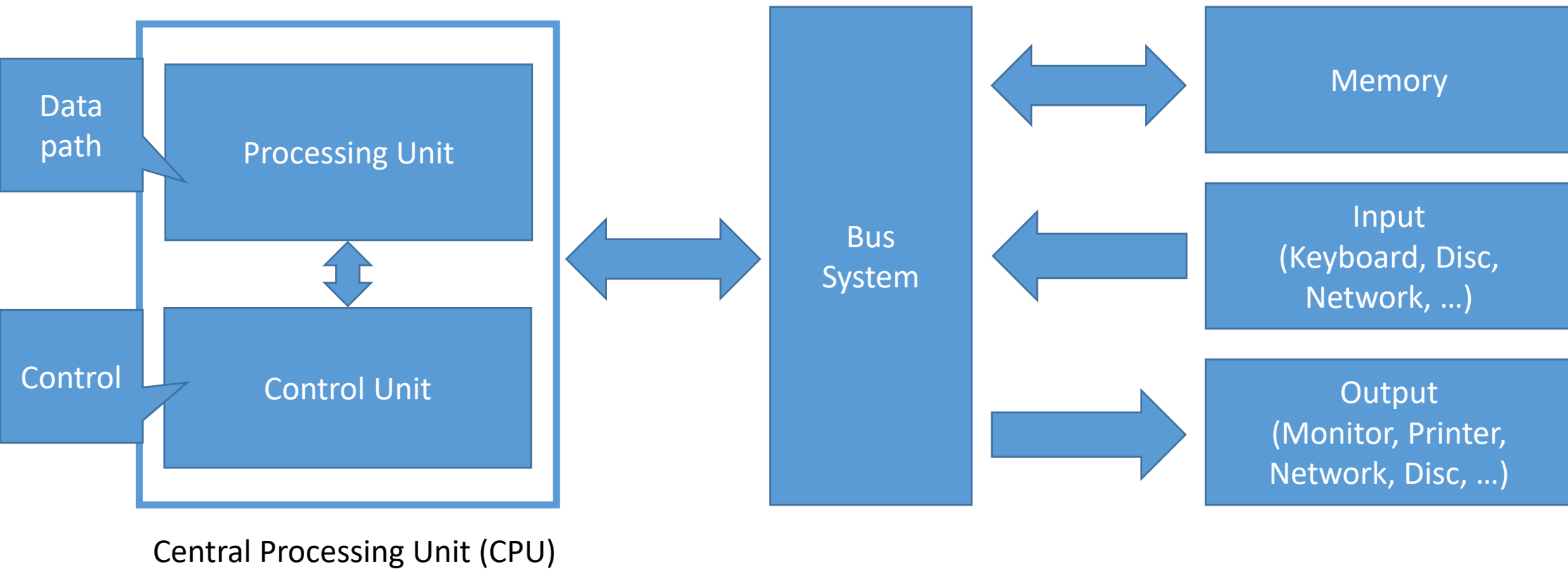
# Von Neumann Model

- Components of a computer built based on Von Neumann
  - Processing Unit
  - Control Unit
  - Memory
  - Input
  - Output
  - Buses



John Von Neumann  
(born 1945 in Budapest)

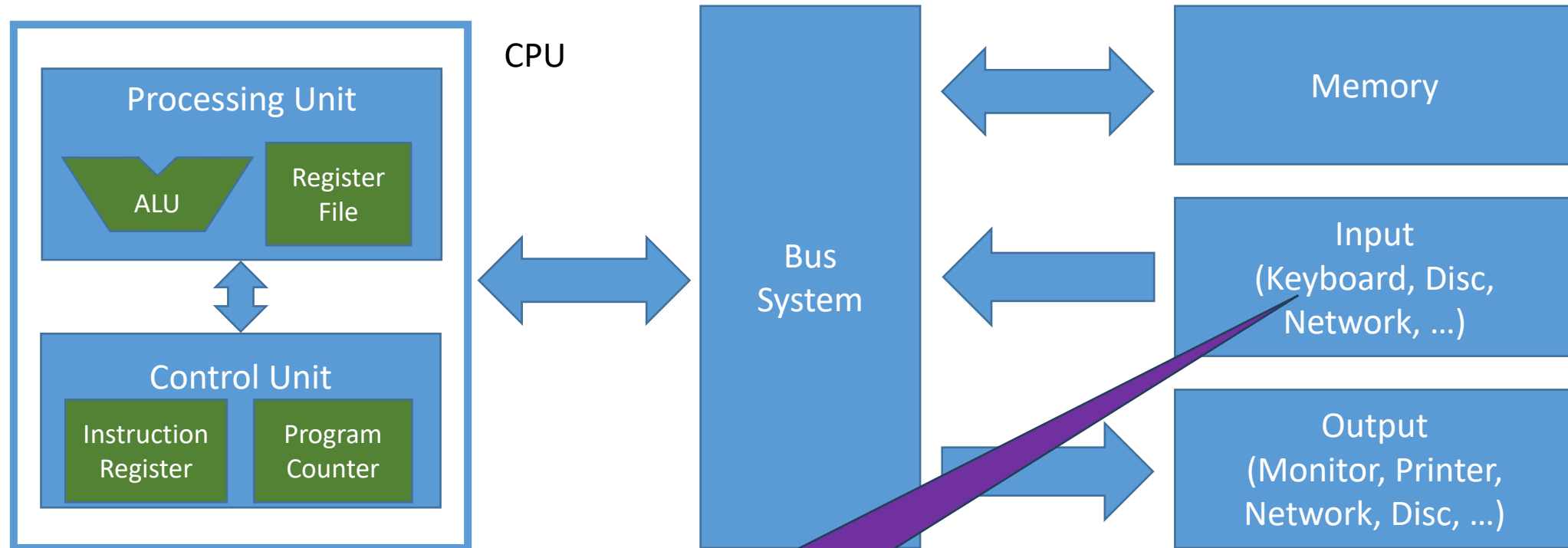
# Von Neumann Model



Central Processing Unit (CPU)

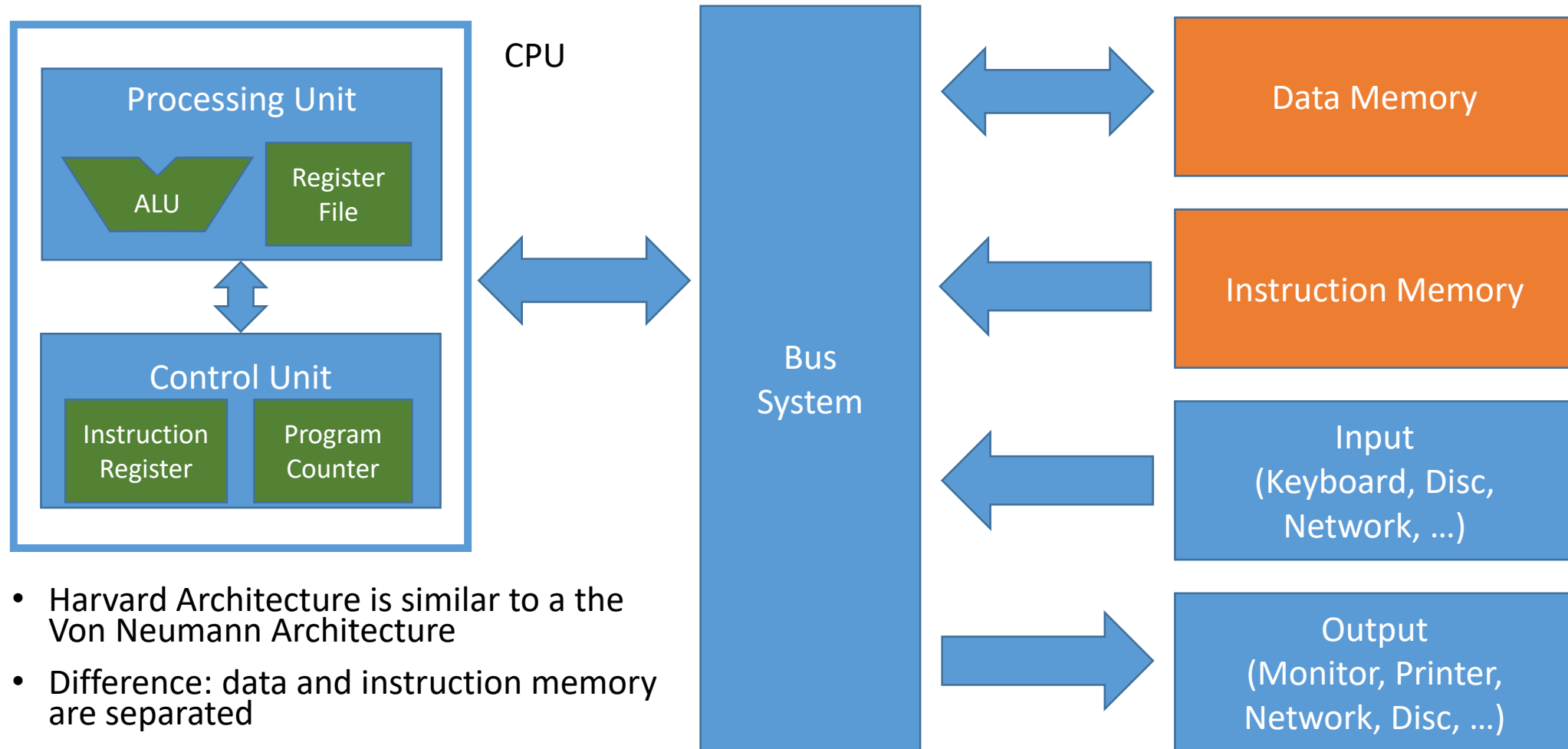
The Von Neumann Model is the classical computer model – it is the basis of most CPUs

# Von Neumann Model



**Note for Task 1 of the practical:**  
I/O Peripherals can be “memory mapped”  
(their registers are addressable just like memory locations)

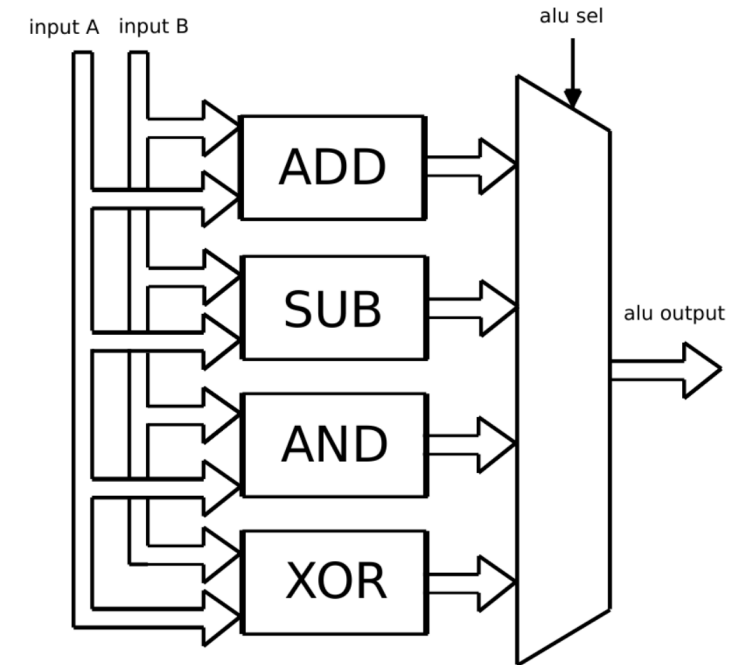
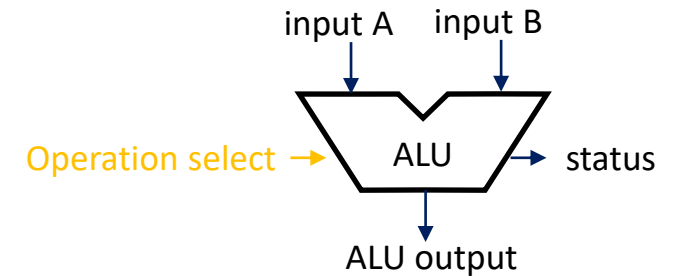
# Harvard Architecture



- Harvard Architecture is similar to a the Von Neumann Architecture
- Difference: data and instruction memory are separated
- We use a Harvard Architecture in the first lectures

# Arithmetic Logic Unit (ALU)

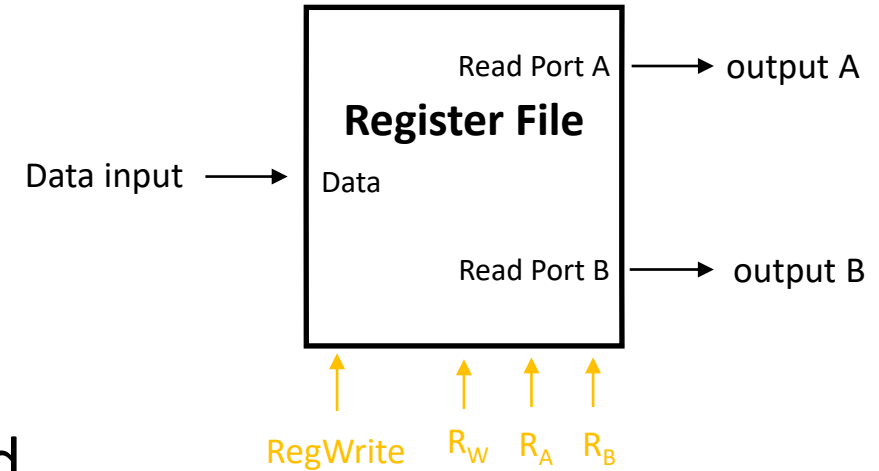
- The ALU is a combinational circuit performing calculation operations
- Basic Properties
  - Takes two n-bit inputs (A, B); today typically 32 bit or 64 bit
  - Performs an operation based on one or both inputs; the performed operation is selected by the control input `alu_sel`
  - Returns an n-bit output; It typically also provides a status output with flags to e.g. indicate overflows or relations of A and B, such as  $A==B$  or  $A<B$





# Register File

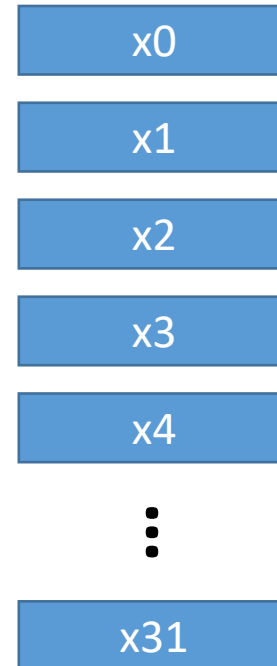
- The register file contains  $m$   $n$ -bit registers
- In a given clock cycle one  $n$ -bit value can be stored in the register selected via the signal  $R_W$ ; In case  $\text{RegWrite}$  is low, no register is written
- In each cycle two registers can be read and are provided at the outputs A and B. The registers to be read are selected via  $R_A$  and  $R_B$
- The register file is essentially a memory with **one write port** and **two read ports**



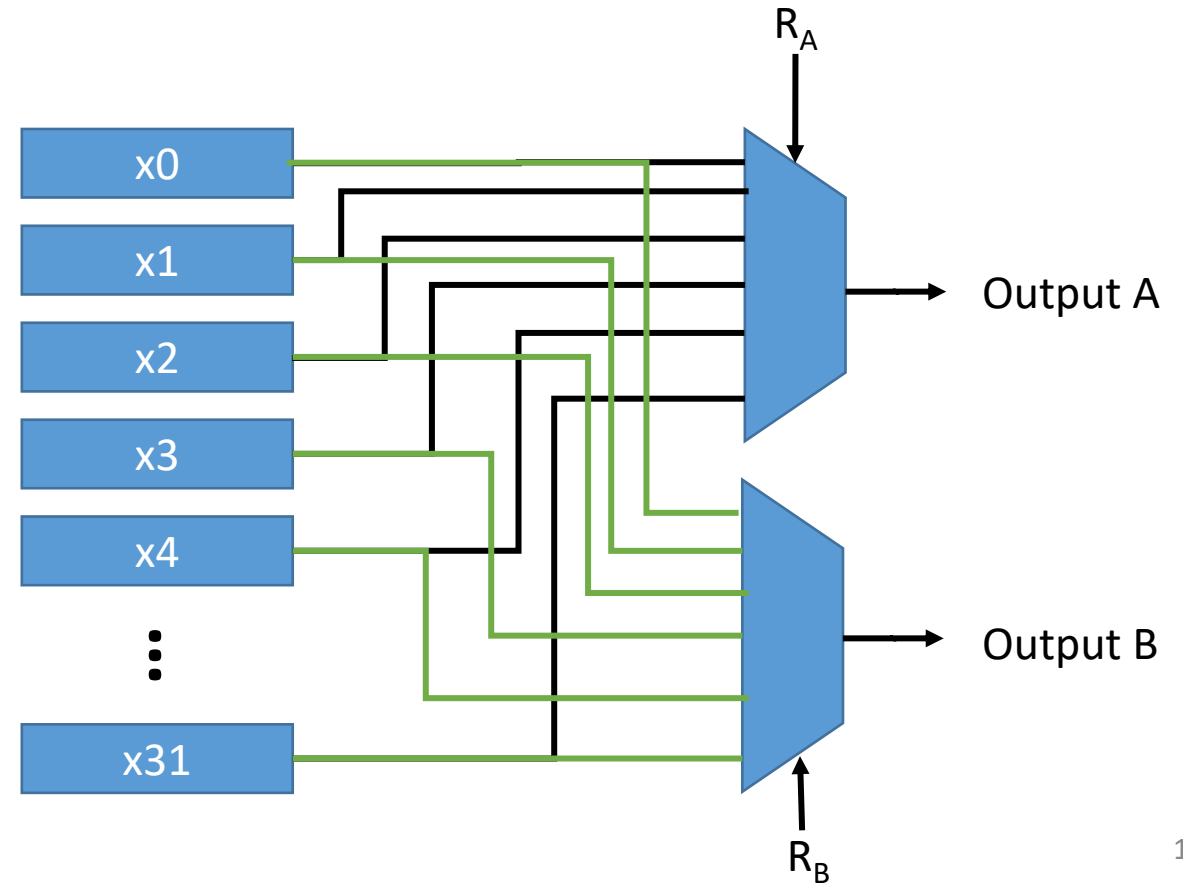
# Data Registers (Register File)

Register File

- In case of RISC-V, the register file consists of 32 registers
- 5 bit are needed for  $R_W$ ,  $R_A$ ,  $R_B$



# Data Registers (Register File)

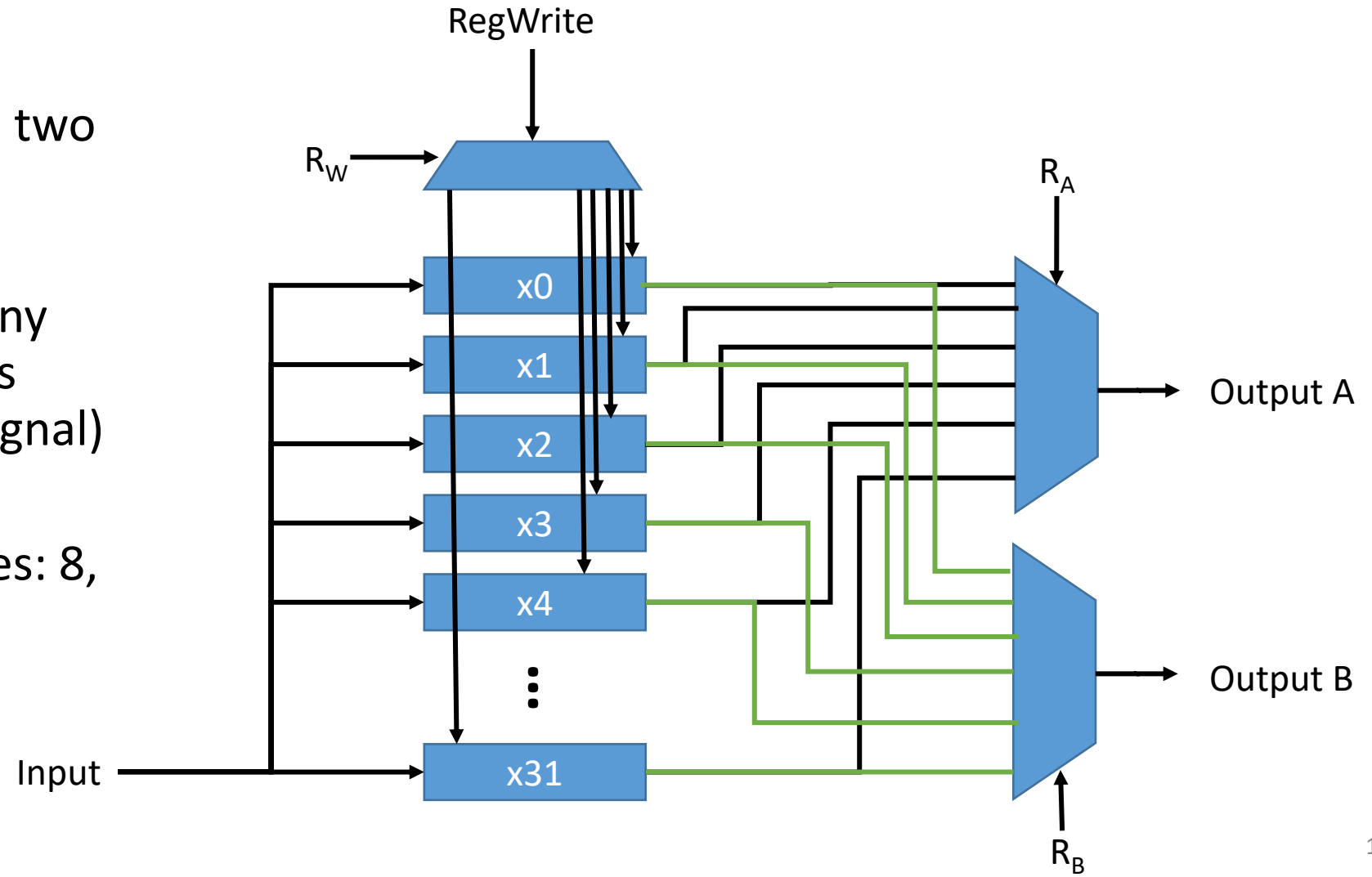


# Data Registers (Register File)

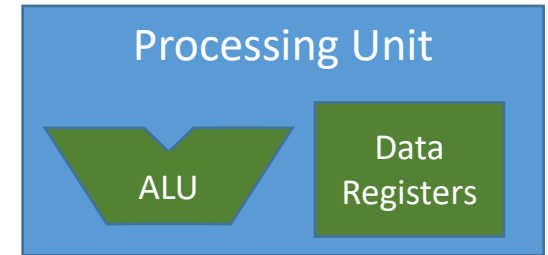
Register File

- Basic Properties

- Data registers with two output MUX
- Input is stored in any one of the registers (selection via  $R_W$  signal)
- Typical register sizes: 8, 16, 32, 64 bit

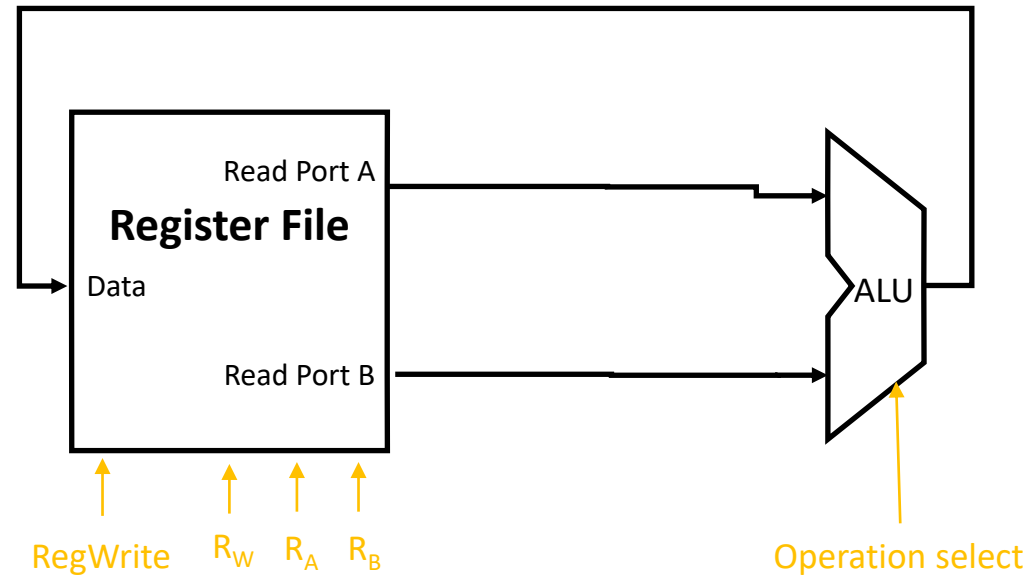


# Processing Unit

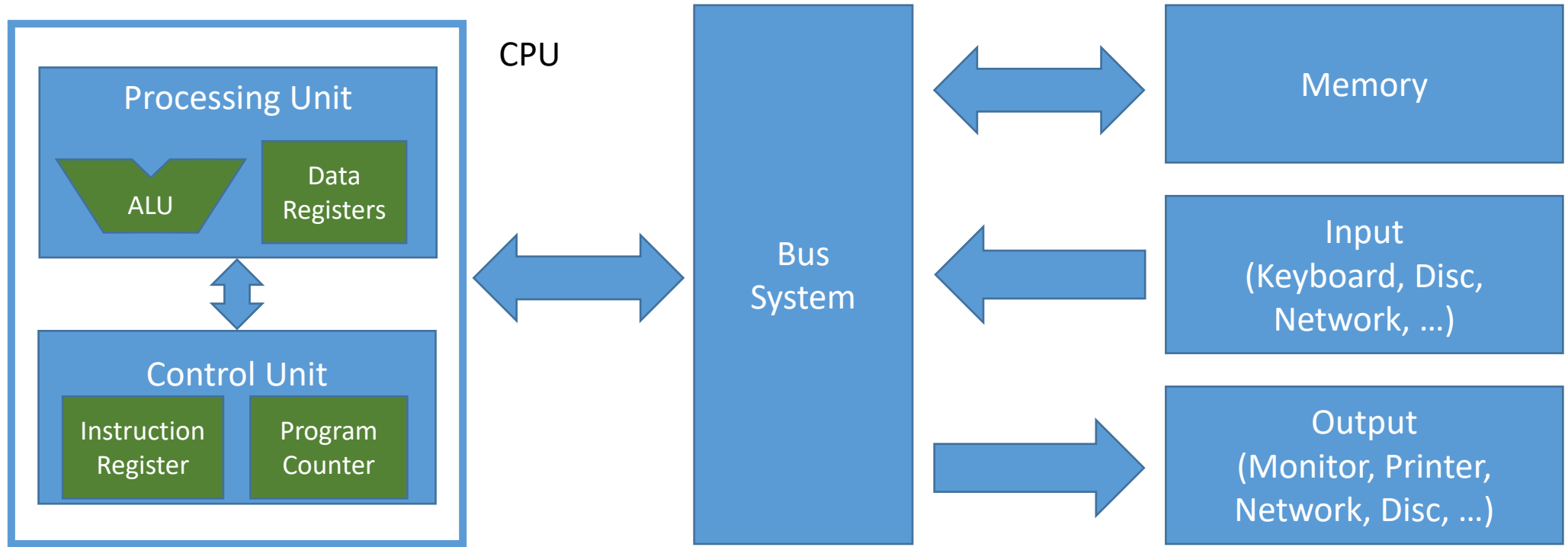


- The processing unit constitutes the data path of the CPU
- Based on control signals that are provided as inputs operations are performed in the ALU and data registers are updated

# A First Simple Datapath for Our CPU

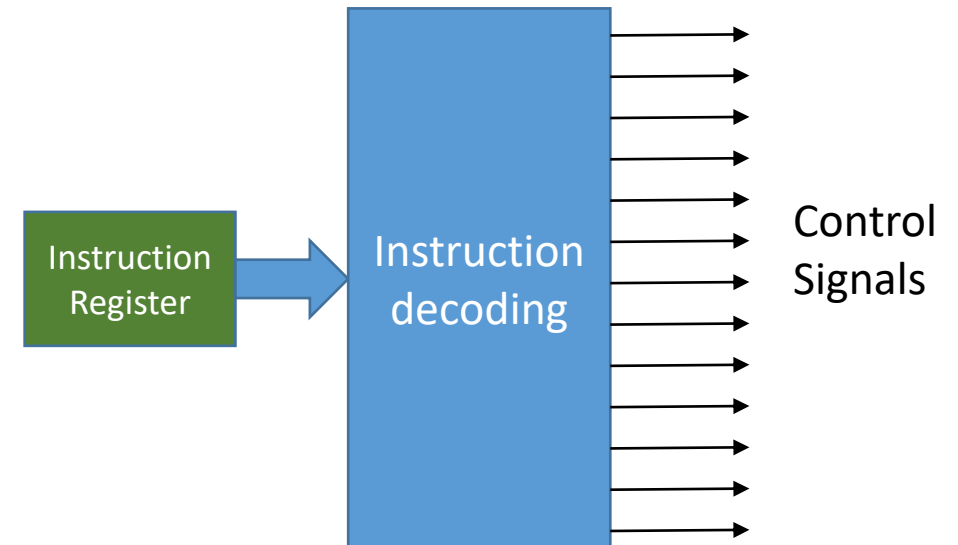
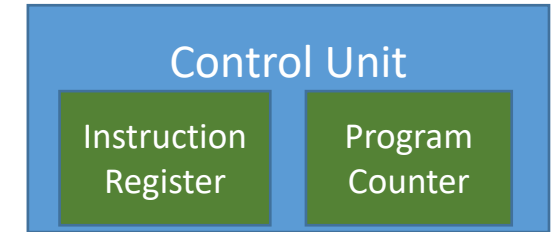


- How do we get data from “outside” into the register file?
- Where do we get the control signals from?



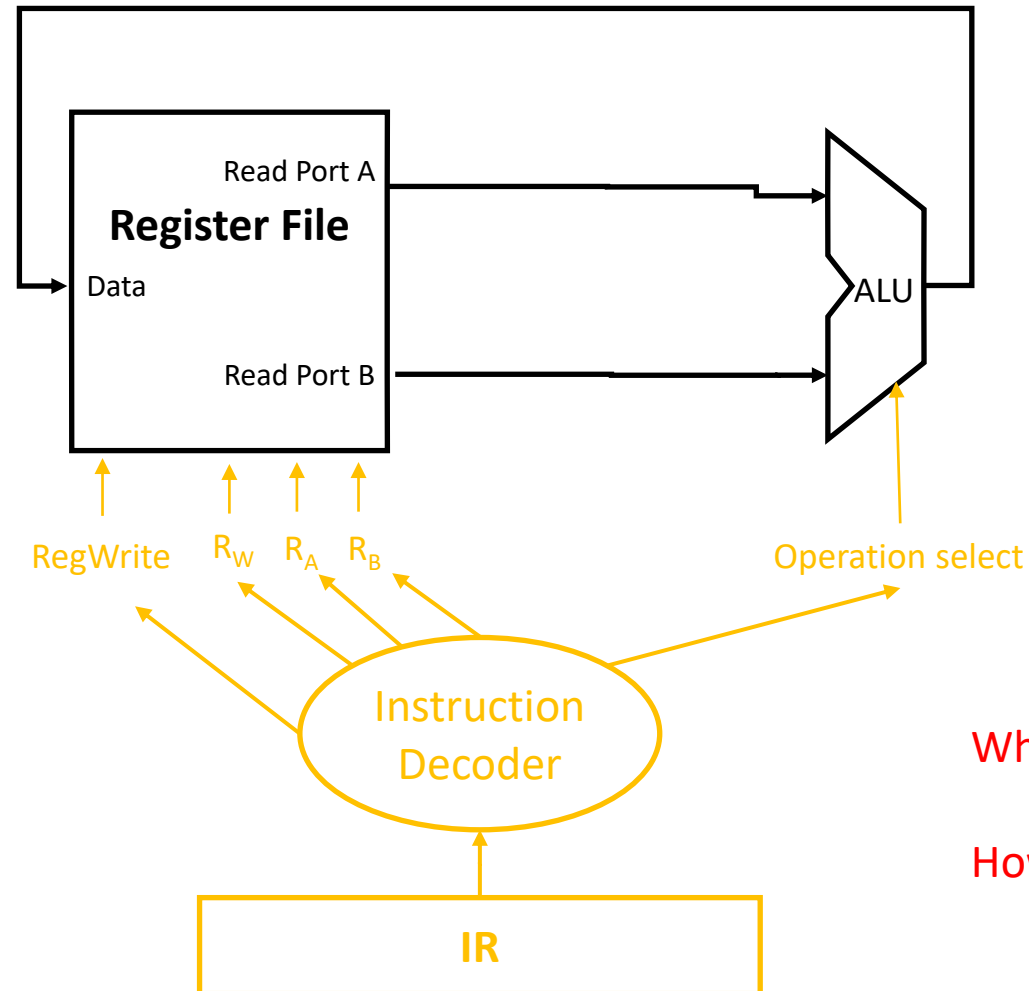
# Instruction Register

- The instruction register stores the instruction that shall be executed by the data path
- The instruction decoder maps the instruction register to control signals





# A First Simple Datapath with Control for Our CPU



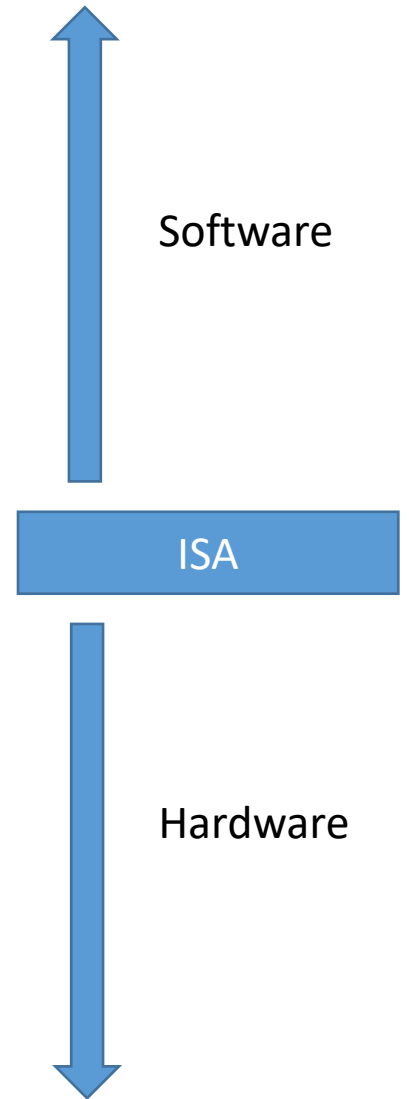
What is an instruction?

How do we encode an instruction?

# Instruction Set Architectures

# Instruction Set Architecture (ISA)

- An instruction is the basic unit of processing on a computer
- The instruction set is the set of all instructions on a given computer architecture
- The ISA is the interface between hardware and software
- Options to represent instructions
  - Machine language:
    - A sequence of zeros and ones, e.g. 0x83200002 → this is the sequence of zeros and ones the processor takes into its instruction register for decoding and execution
    - Length varies can be many bytes long (up to 15 bytes on x86 CPUs)
  - Assembly language:
    - This is a human readable representation of an instruction, e.g. ADD x3, x1, x2



# Instruction Set Architectures

- There are many instruction set architectures from different vendors
  - Examples: Intel x86, AMD64, ARM, MIPS, PowerPC, SPARC, AVR, RISC-V, ...
- Instruction sets vary significantly in terms of number of instructions
  - **Complex Instruction Set Computer (CISC)**
    - Not only load and store operations perform memory accesses, but also other instructions
    - Design philosophy: many instructions, few instructions also for complex operations
    - Hundreds of instructions that include instructions performing complex operations like entire encryptions
    - Examples: x86 and x64 families
  - **Reduced Instruction Set Computer (RISC)**
    - RISC architectures are **load/store architectures**: only dedicated load and store instructions read/write from/to memory
    - Design philosophy: fewer instructions, lower complexity, high execution speed.
    - Instruction set including just basic operations
    - Examples: ARM, RISC-V
  - **One Instruction Set Computer (OISC)**
    - Computers with a single instruction (academic), e.g. SUBLEQ  
see [https://en.wikipedia.org/wiki/One\\_instruction\\_set\\_computer](https://en.wikipedia.org/wiki/One_instruction_set_computer)

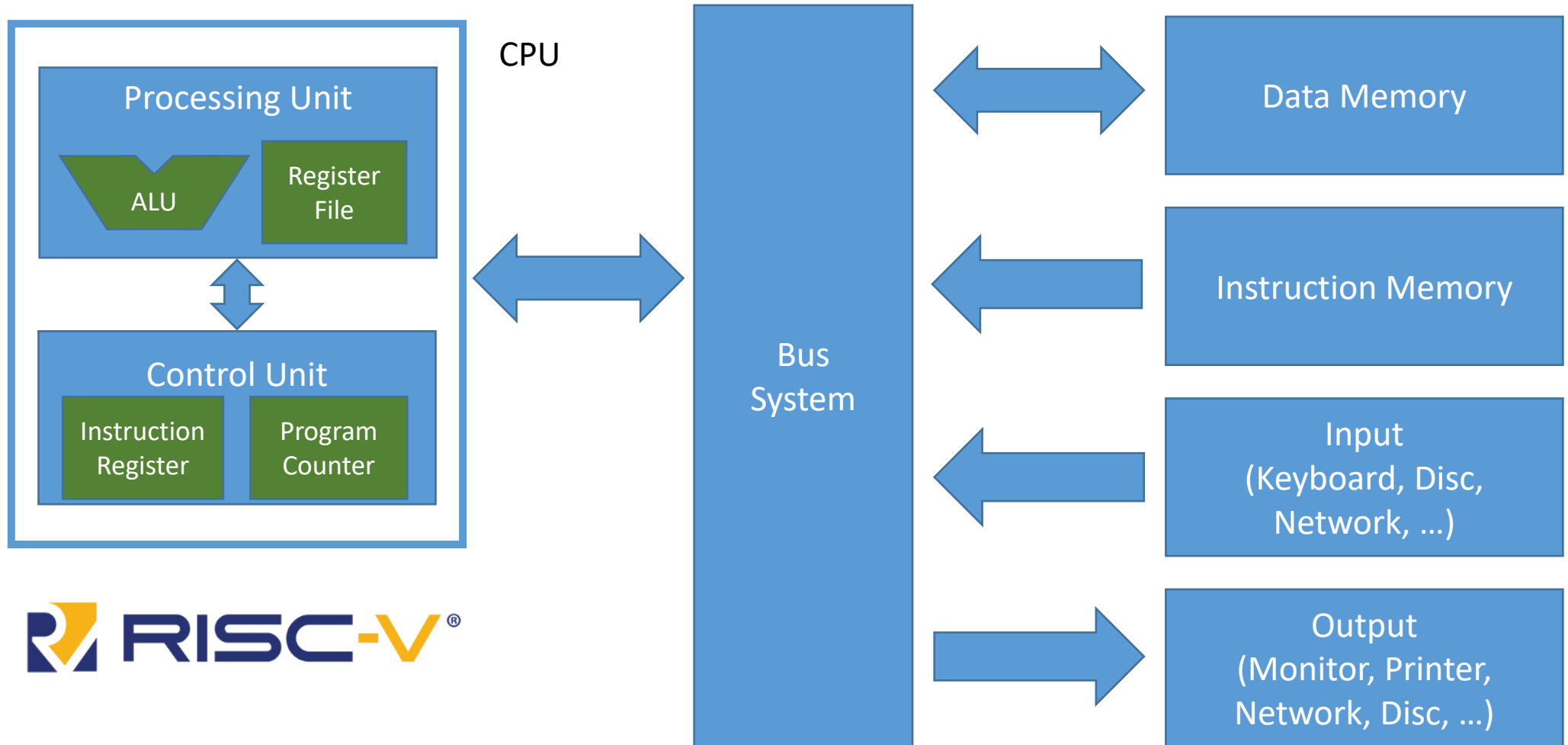
# Competition Between Instruction Sets

- Given a fixed program (e.g. written in C), which instruction set leads
  - to the smallest code size (the smallest number of instructions need to express the program)?
  - to best performance on a processor implementing the ISA?
  - lowest power consumption on a processor implementing the ISA?
  - ...

# Open vs. Closed Instruction Sets

- Most instruction sets are covered by patents
  - Building a computer that is compatible with that instruction set requires patent licensing
- RISC-V (the instruction set of this course)
  - is open
  - developed at UC Berkeley
  - An instruction family from low-end 32bit devices to large 64bit CPUs
  - Significant momentum in industry and academia
  - More information and full specs available at <https://riscv.org/>





# First RISC-V Basics



# RISC-V Instruction Sets

- **Base instruction sets**

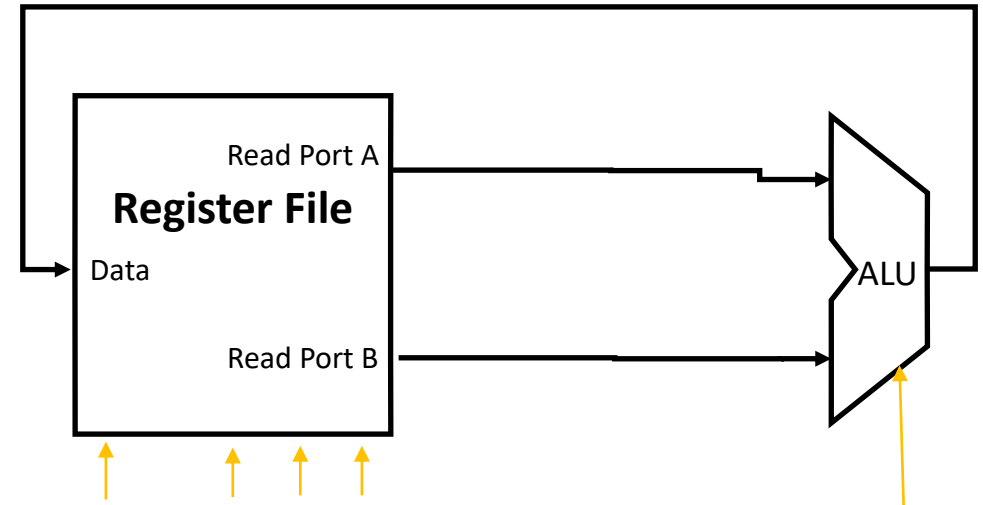
- **RV32I** (RV32E is the same as RV32I, except the fact that it only allows 16 registers)
- RV64I
- RV128I

- **Extensions**

- “M” Standard Extension for Integer Multiplication and Division
- “A” Standard Extension for Atomic Instructions
- “Zicsr”, Control and Status Register (CSR) Instructions
- “F” Standard Extension for Single-Precision Floating-Point
- ....

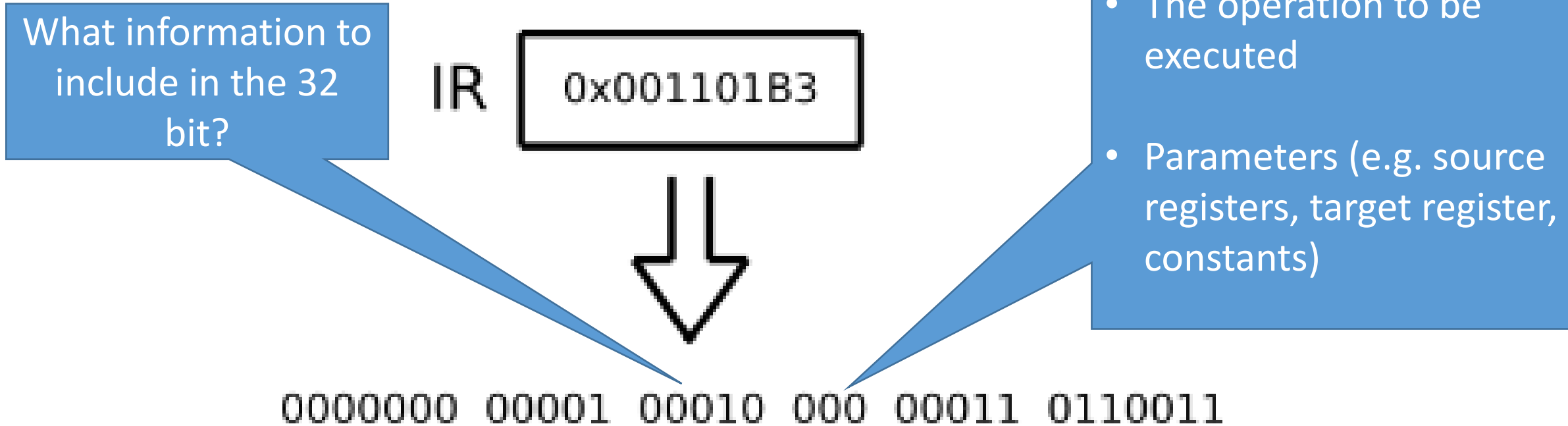
# Register File and ALU

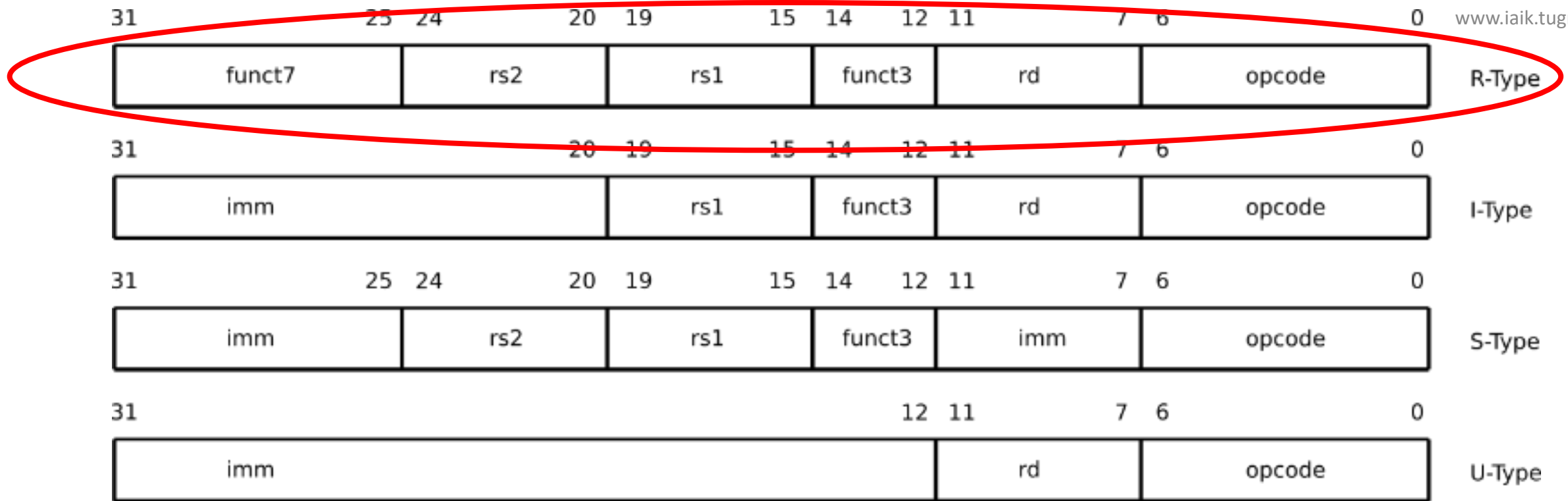
- We focus on RV32I
- The ALU and the register file are all 32 bit
- Our register file consists of 32 registers (**Note**: register x0 always reads zero; writing to x0 does not lead to storing a value)



# Basics

The base instruction set has fixed-length 32-bit instructions

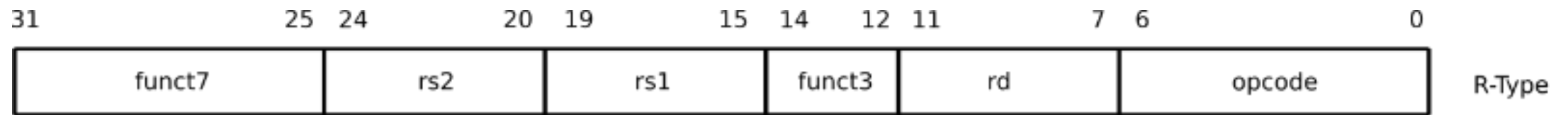




- **Opcode, funct3, funct7:** definition of the functionality
- **Imm:** immediate values (constants)
- **rs1, rs2:** source registers
- **rd:** destination register

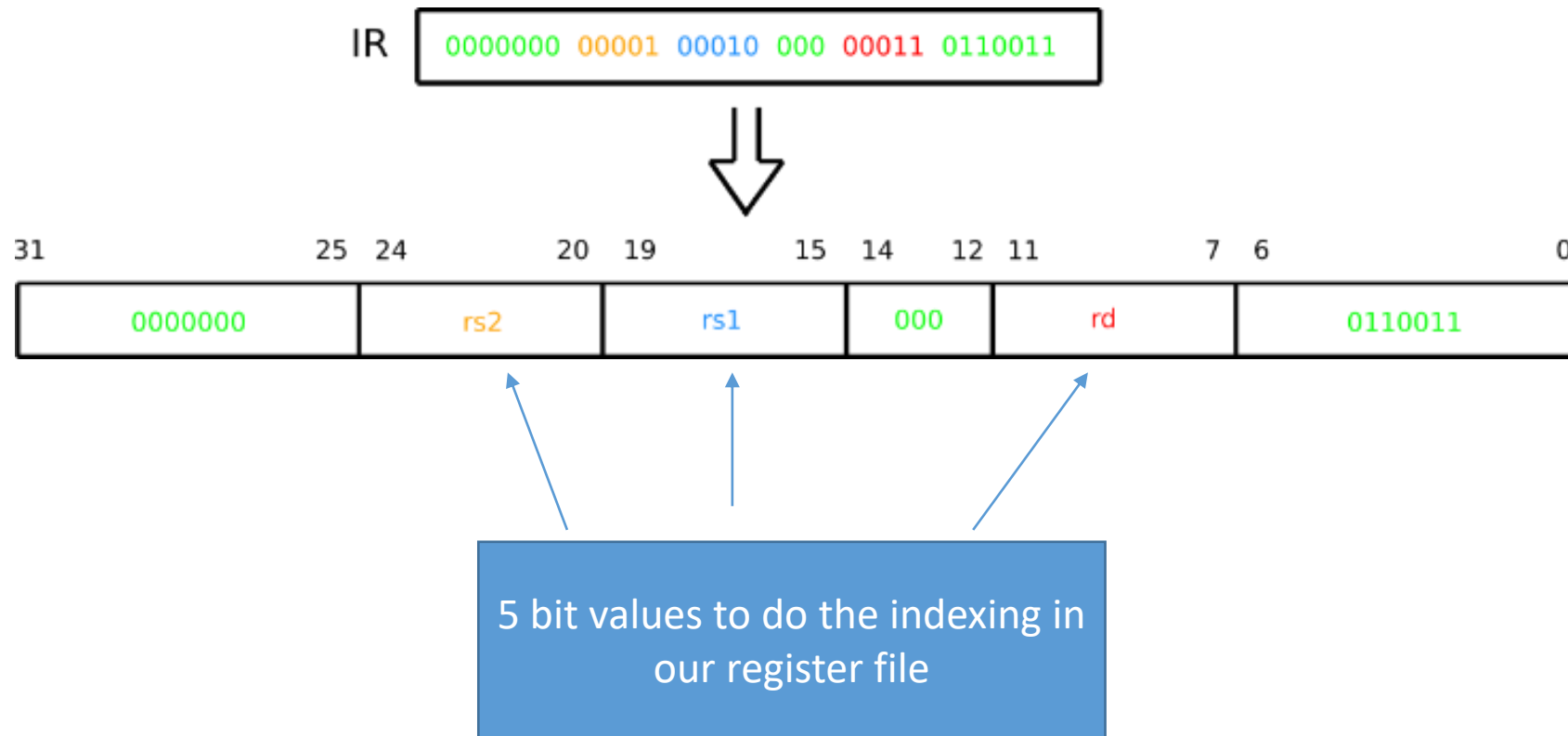
# R-Type Instructions

- These are instructions that perform arithmetic and logic operations based on two input registers



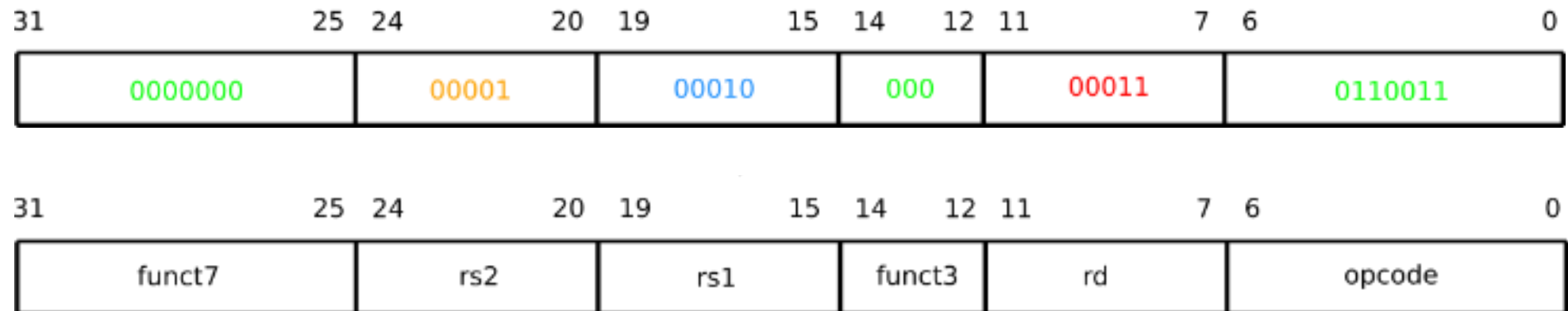
- funct7, funct4 and opcode define the operation to be performed
- rs1 defines source register 1
- rs2 defines source register 2
- rd defines the destination register

# Example



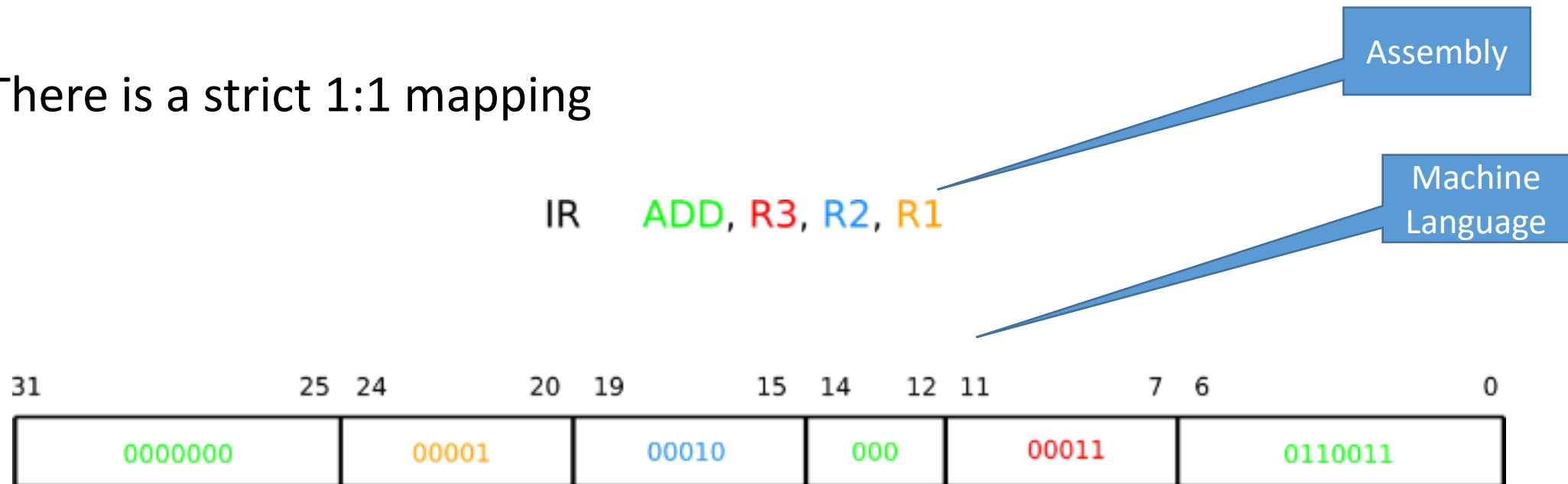
# Example

IR    **ADD**, **R3**, **R2**, **R1**



# Machine Language and Assembly

- Every instruction can be represented in human readable form → **assembly**
- Every instruction can be represented in machine readable form → **machine language**
- There is a strict 1:1 mapping





# The RV32I Instruction Set

- 40 instructions
- Categories:
  - Integer Computational Instructions
  - Load and Store Instructions
  - Control Transfer Instructions
  - Memory Ordering Instructions
  - Environment Call and Breakpoints

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

# Integer Computational Instructions

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

- All instructions take two input registers (**rs1** and **rs2**) and compute the result in **rd**
- Example: `sub r3, r1, r2` computes  $r3 = r1 - r2$

# Integer Computational Instructions

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

## • Logic Functions

- AND
- OR
- XOR

## • Arithmetic

- ADD (Addition)
- SUB (Subtraction)

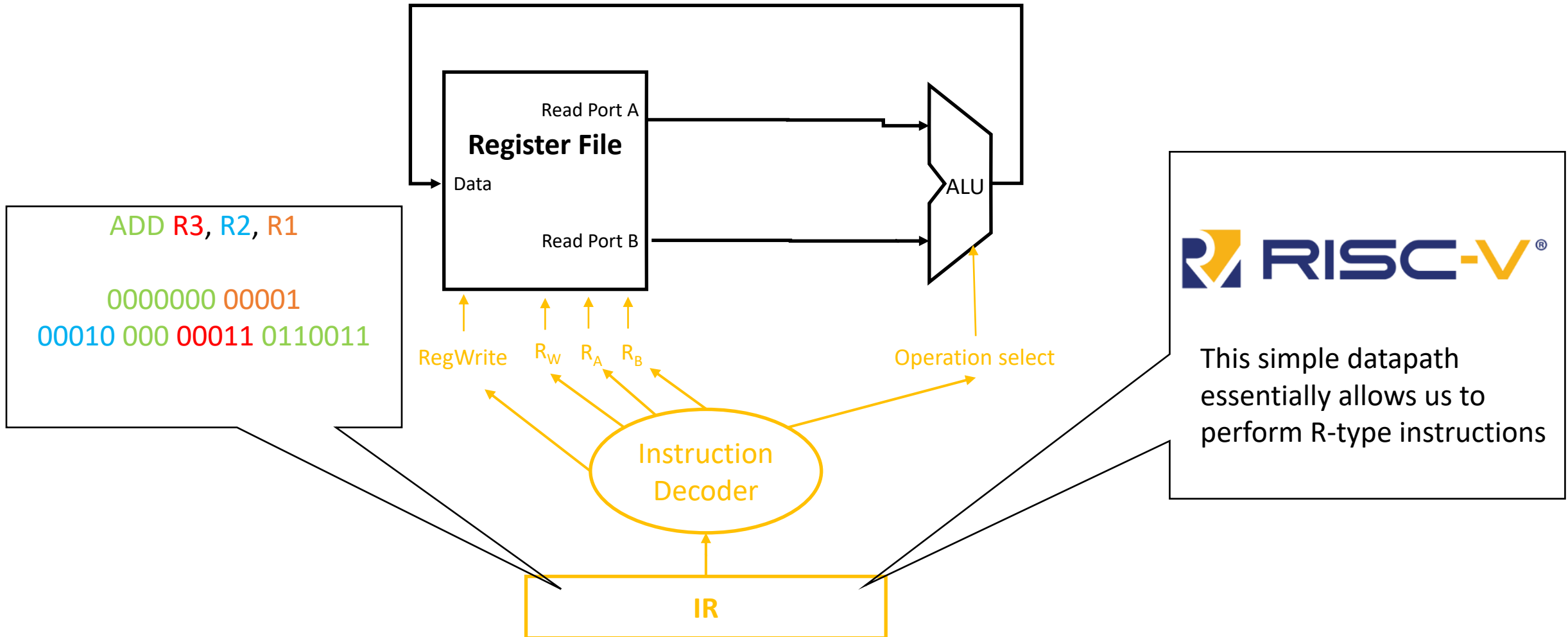
## • Shifts

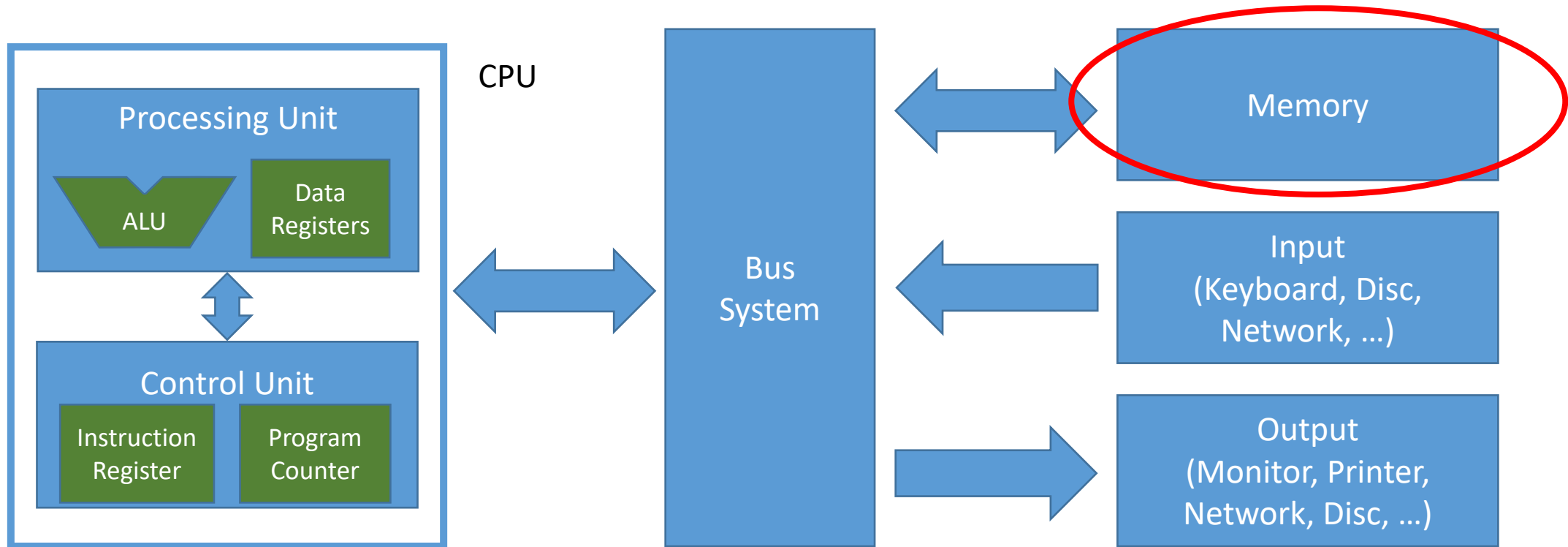
- SLL (Logical Shift Left)
- SRL (Logical Shift Right)
- SRA (Arithmetic Shift Right)

## • Compares

- SLT (Set on Less Than)
- SLTU (Set on Less Than – unsigned)

# A First Simple Datapath with Control for Our CPU

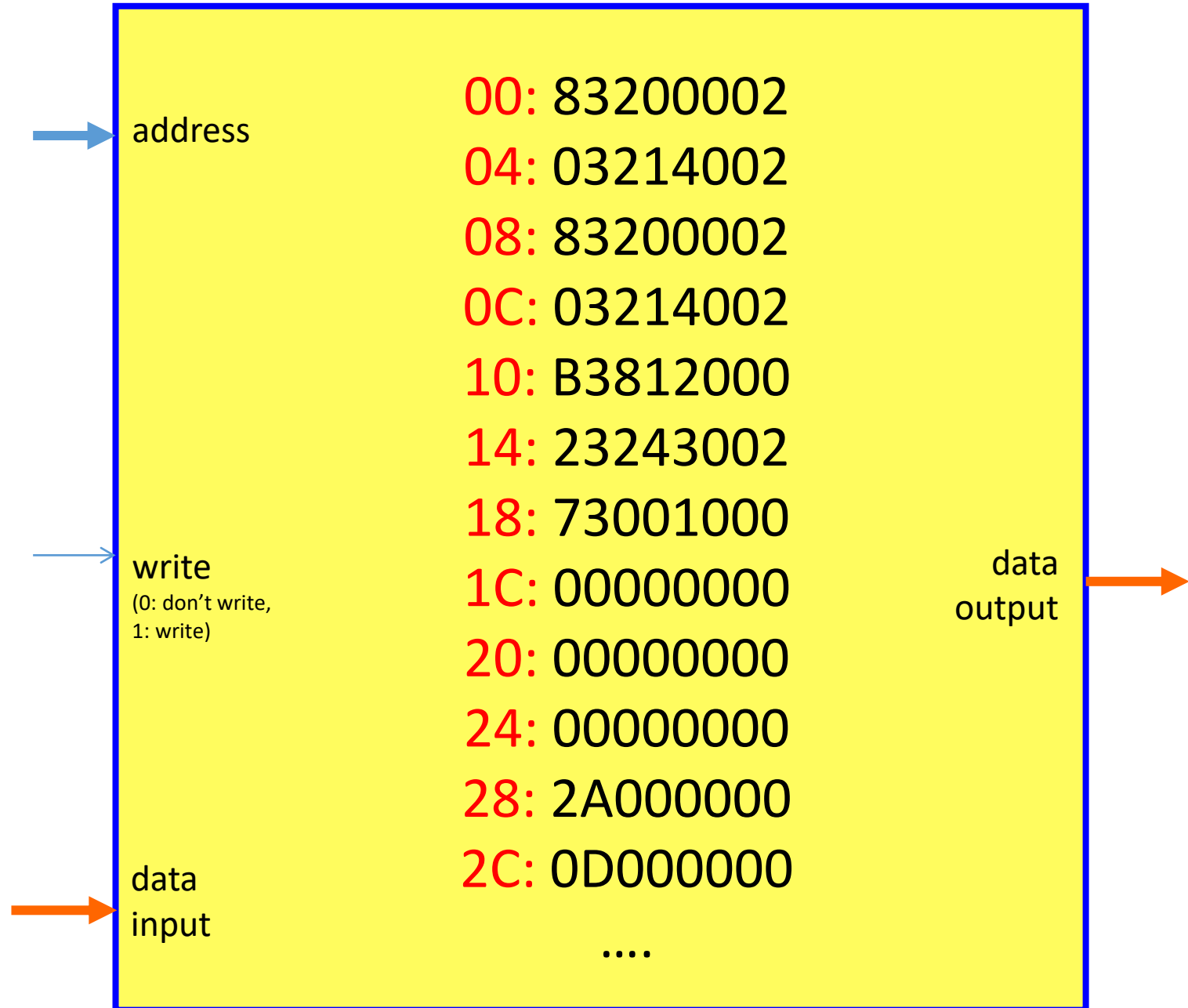




Let's learn about memories!

# Memory

# Memory



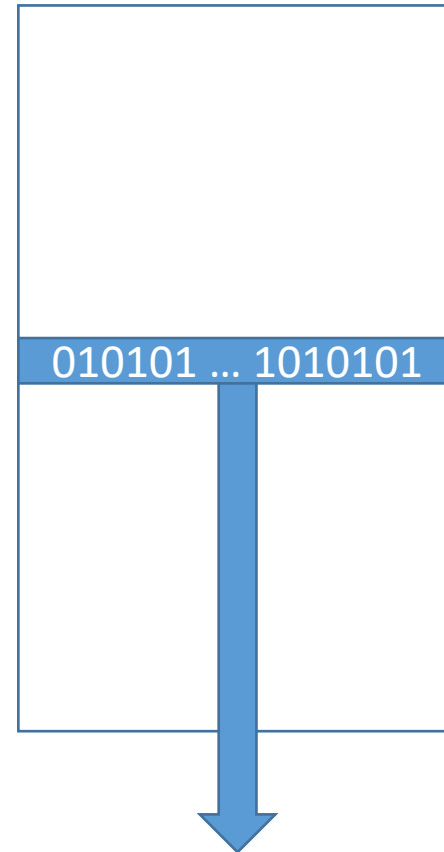
# Our Memories are RAMs

**R**andom **A**ccess **M**emory

(“Memory where arbitrary read and write accesses can be performed”)

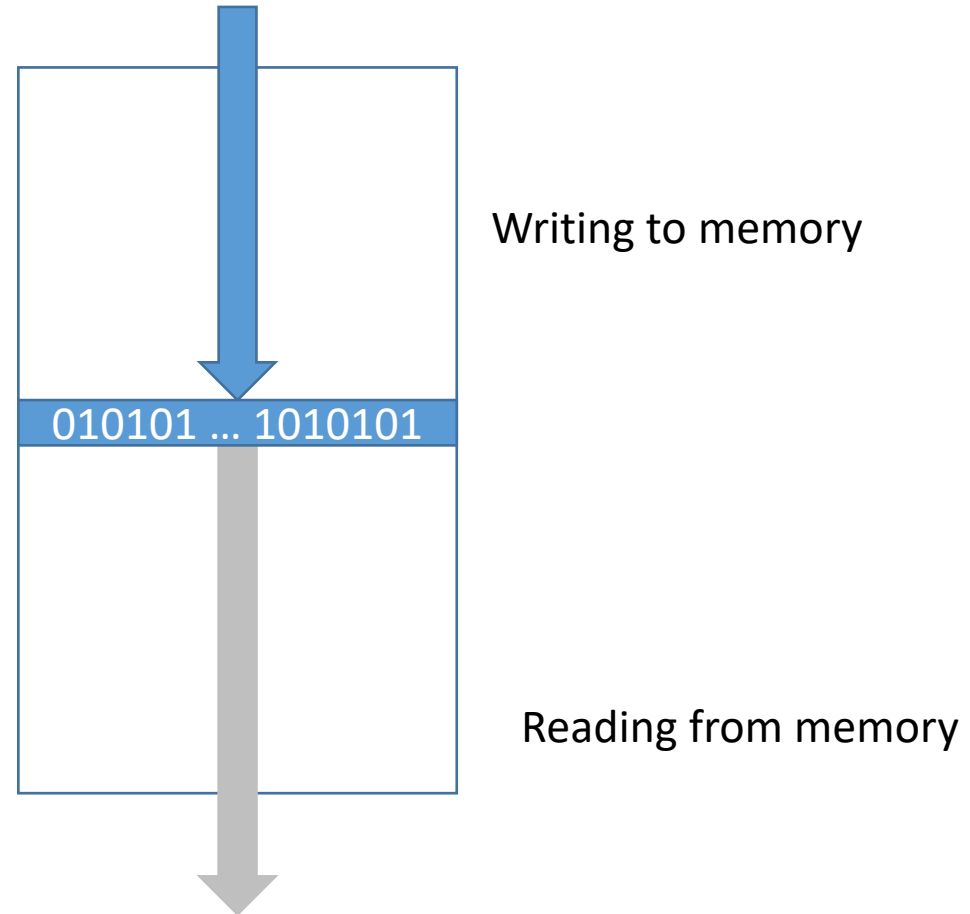


# Reading from memory

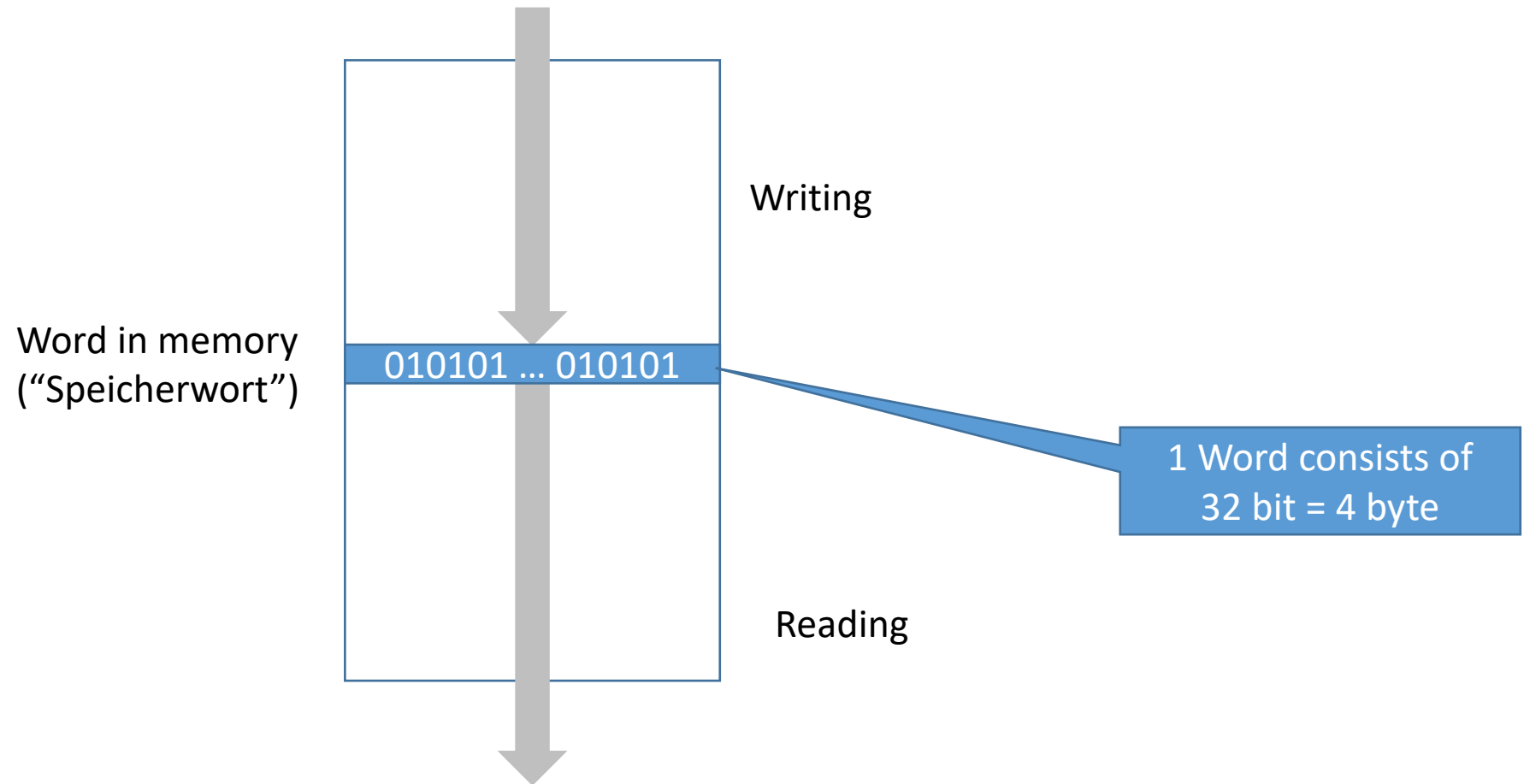


Reading from memory

# Writing to Memory



# A Word in Memory in Case of a 32-bit System



# Each Byte in Memory Has an Address

Address  
(increment by 1  
is an increment  
of the position  
by 1 byte)

Address:

00:	010101 ... 1010101
04:	010101 ... 1010101
08:	110100 ... 0011101
0C:	010111 ... 1000001
10:	110100 ... 0011101
14:	010111 ... 1000001
18:	010111 ... 1000001
1C:	010101 ... 1010101
20:	110100 ... 0011101
24:	010111 ... 1000001
28:	010101 ... 1010101
2C:	110100 ... 0011101
30:	010111 ... 1000001
34:	010101 ... 1010101
38:	110100 ... 0011101
3C:	010111 ... 1000001

1 Word = 4 byte = 32 bit

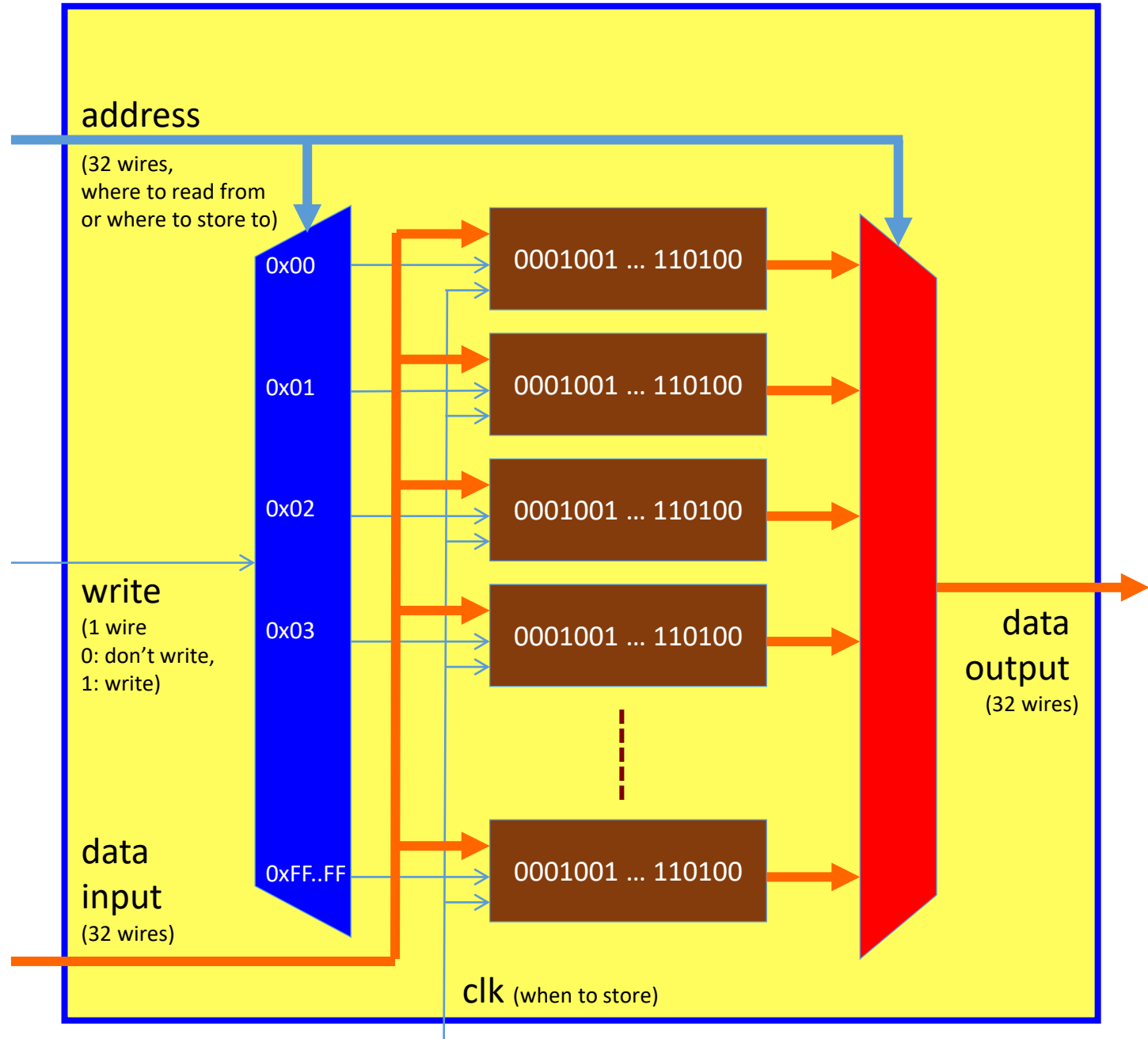
# The Indices of the Bits Within a Word in Memory

Address:

00:	010101 ... 1010101
04:	010101 ... 1010101
08:	110100 ... 0011101
0C:	010111 ... 1000001
10:	110100 ... 0011101
14:	010111 ... 1000001
18:	010111 ... 1000001
1C:	010101 ... 1010101
20:	110100 ... 0011101
24:	010111 ... 1000001
28:	010101 ... 1010101
2C:	110100 ... 0011101
30:	010111 ... 1000001
34:	010101 ... 1010101
38:	110100 ... 0011101
3C:	010111 ... 1000001

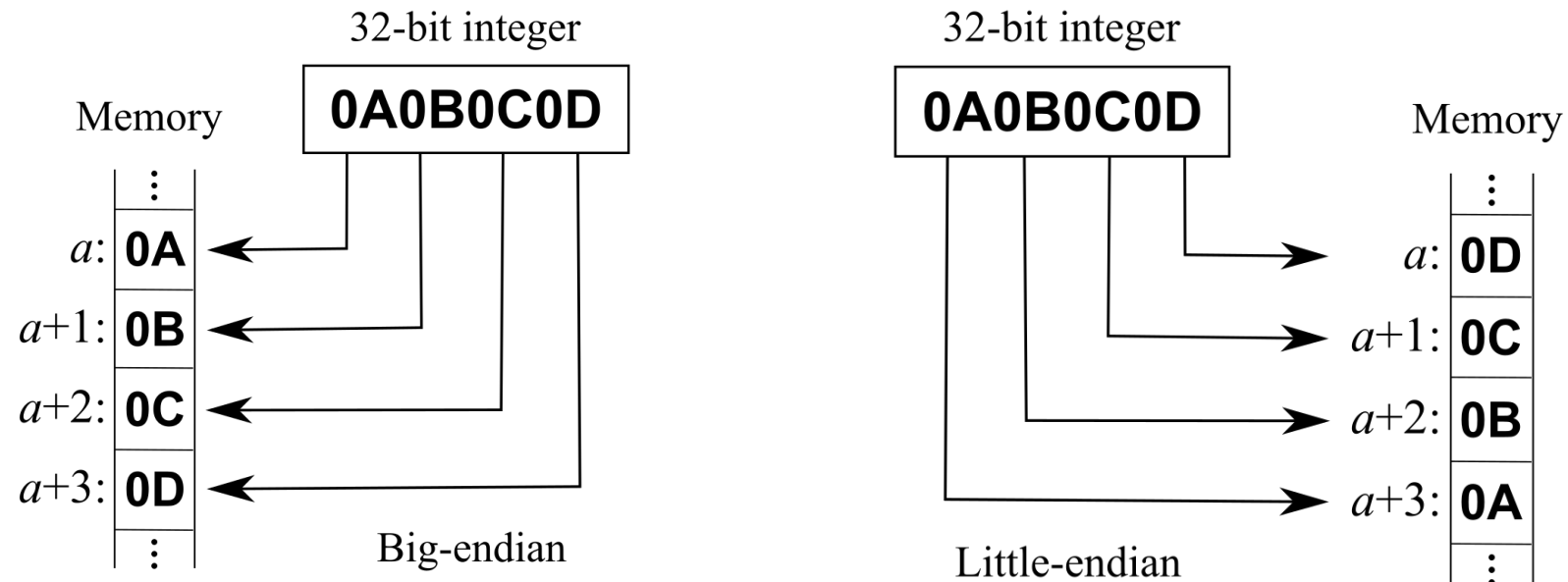
↑ Bit 31                      ↑ Bit 0

# Memory



# Endianess

- There are two options for the sequence of storing the bytes of a word in memory:
  - Little endian: least significant byte is at the lowest address
  - Big endian: most significant byte is at lowest address



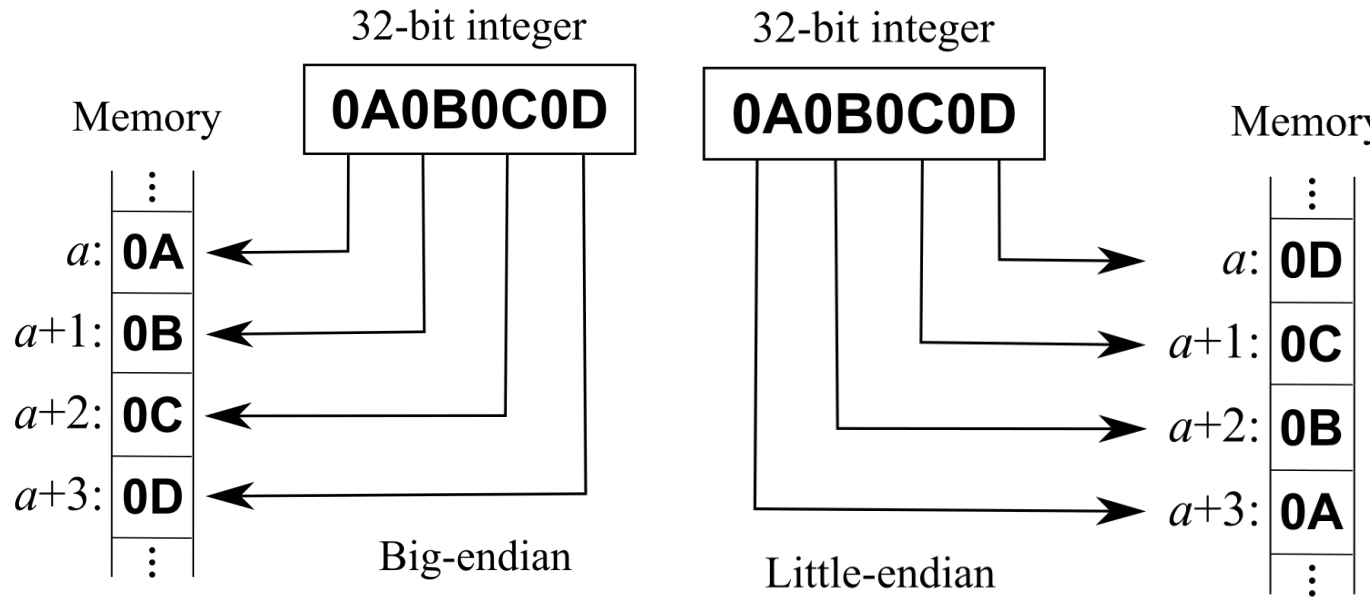
# Endianess - Example

CPU Register Content  
(little endian)

02002083

Representation  
in Memory

00:	83200002
04:	03214002
08:	B3812000
0C:	23243002
10:	73001000
14:	00000000
18:	00000000
20:	2A000000
24:	0D000000



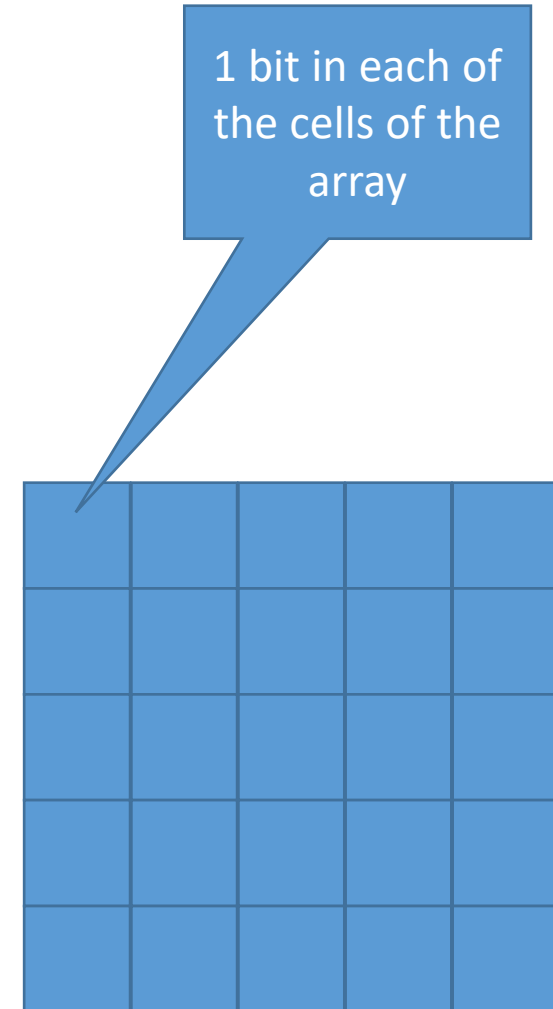


# Building Memories in Practice

- Building Memories based on standard flip flops (FFs), decoders and multiplexers would be extremely expensive!
  - Note: The functionality of a memory is less than what is available in a set of FFs:
    - A set of FFs allows that in each cycle a different value is written to each FF
    - A set of FFs allows that in each cycle the content of each FF is read
- A single port read/write memory requires only that it is possible to read/write one memory cell at a time

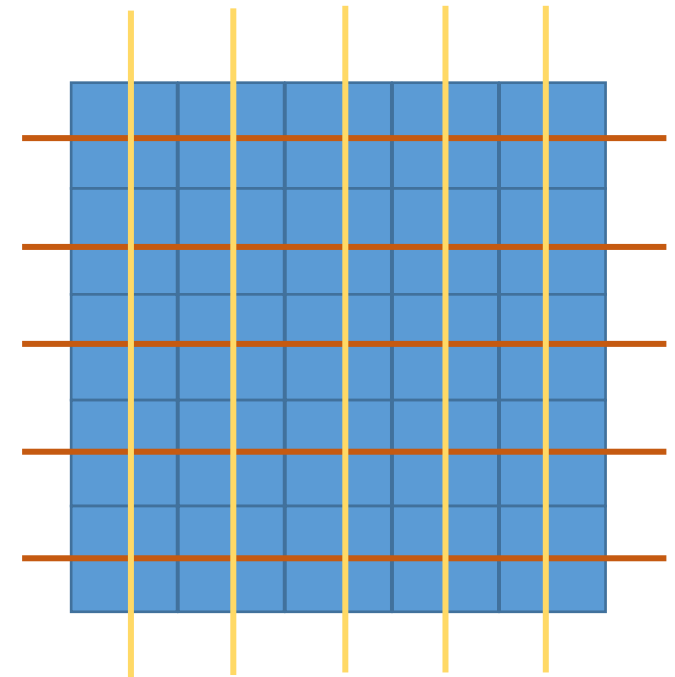
# Basic Idea of Memory Design

- Example: A RAM with a one bit read/write port
- Memories are built using so-called memory cells. Each cell can store one bit
- The memory cells are placed on a chip next to each other and form a rectangular structure: the so-called cell array.



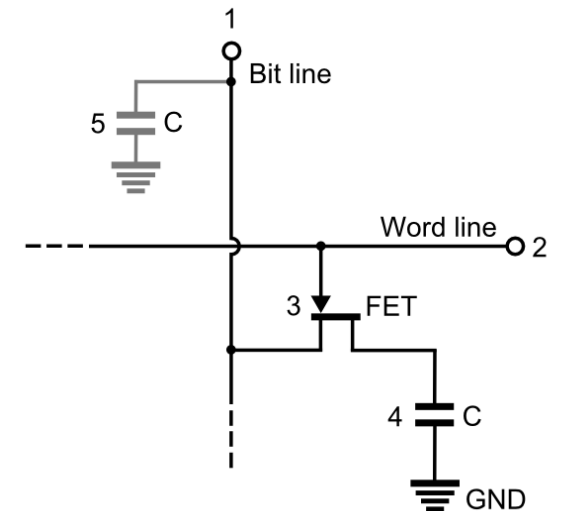
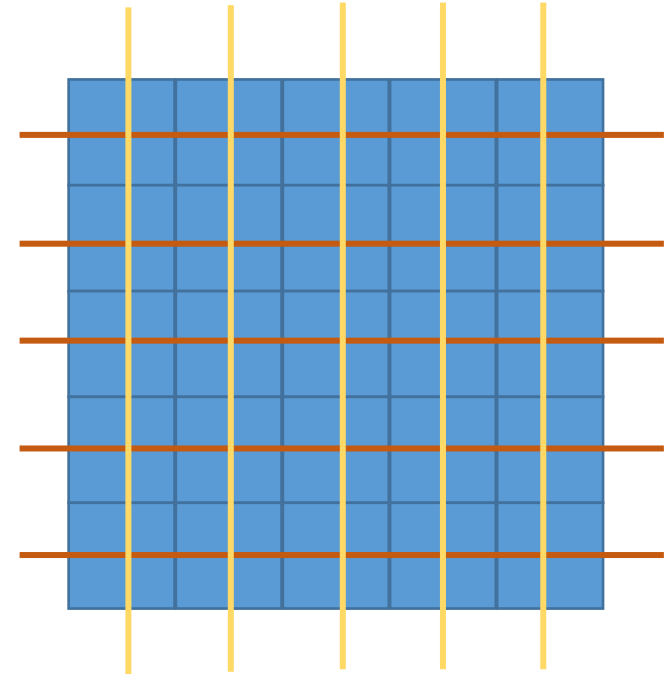
# Basic Idea of Memory Design

- A bitline connects all memory cells of a column vertically (yellow)
- A wordline connects all memory cells of a row horizontally
- This basic structure is used for all kinds of memories:
  - Non-volatile memory (Flash memory)
  - Static memory (SRAM)
  - Dynamic memory (DRAM)
  - DDR memory
- Each memory type is for different trade-offs with respect to size, speed, ...



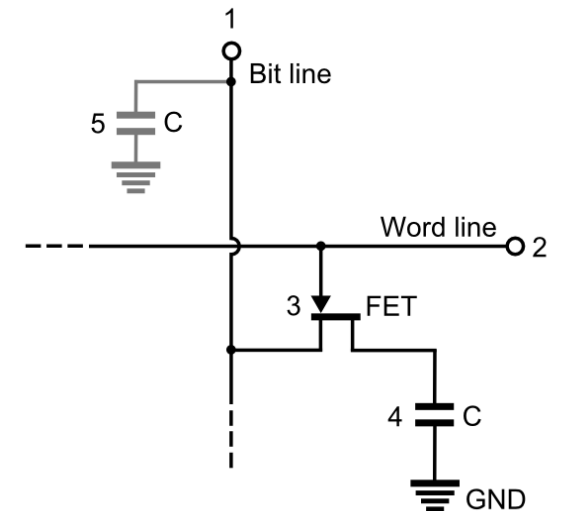
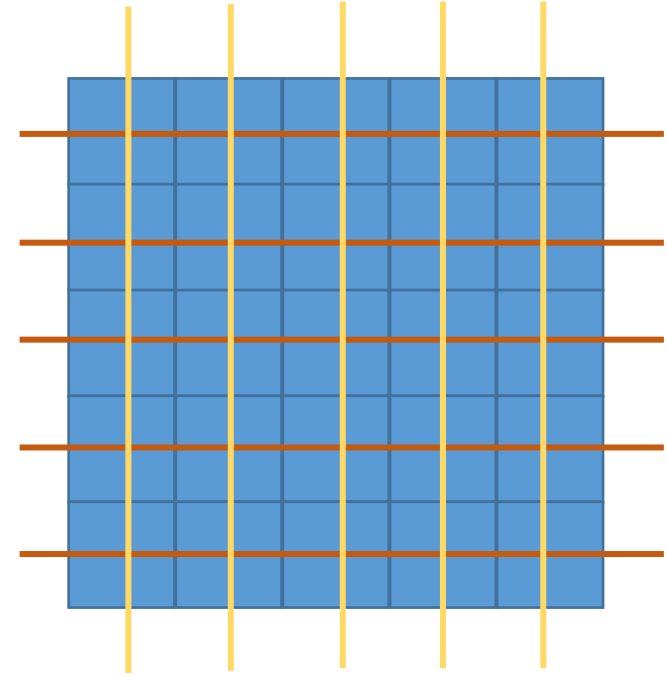
# Basic Idea of Read/Write for DRAM

- A DRAM cell just consists of a single transistor and a capacitance that stores the data value
- In steady state (no access) all bitlines and wordlines are disconnected from the power supply (i.e. they are floating)



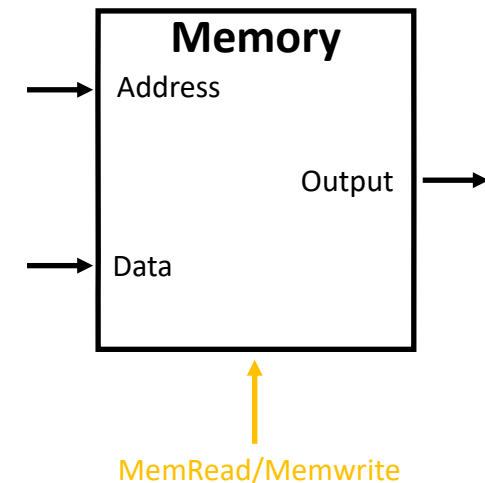
# Basic Idea of Read/Write for DRAM

- Writing a cell:
  - Set corresponding bitline to the desired storage value
  - Set corresponding wordline to high
  - This charges the capacitance of the desired cell to the desired storage value
  
- Reading a cell:
  - Pre-charge the corresponding bitline to the desired voltage value
  - Disconnect the bitline
  - Set the corresponding wordline to high
  - The bitline keeps its value, if the stored value is high or is pulled to low, if the stored value is zero

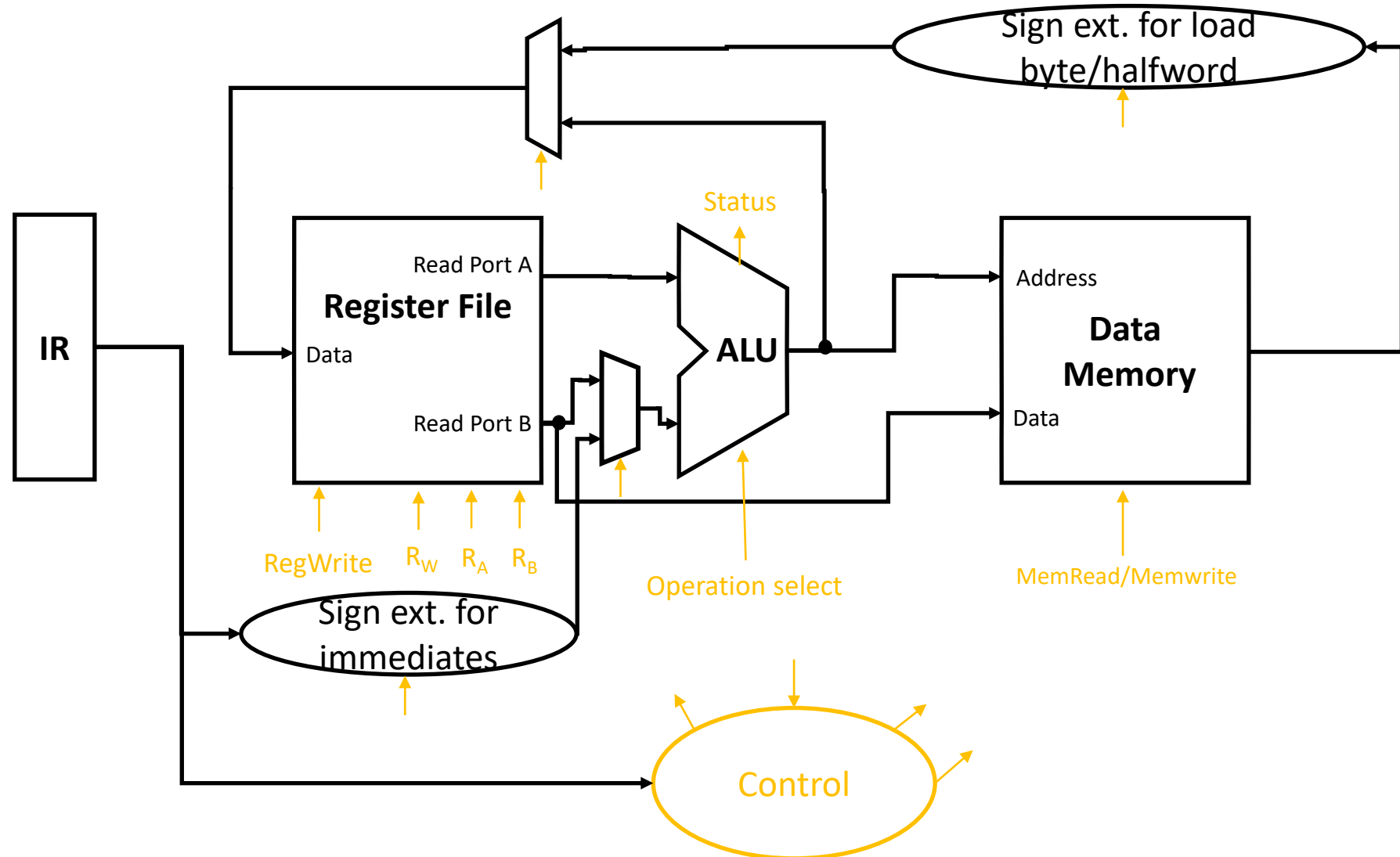


# Memories

- There are many details to know and learn about memories → memories are highly optimized components of a computer system (The size of a memory cell has a huge multiplication effect)
- In this lecture, we focus on the top-level view
- With “memory” we mean a single-port read and single-port write memory for 32-bit values



# Datapath Including Data Memory and Sign Extension



## 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

Arithmetic/Logic operations  
were already possible with  
our first version of the ALU



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

Additional operations that we can perform with our updated datapath:

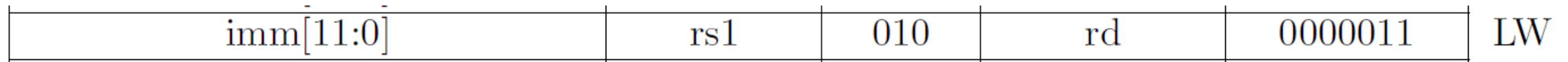
Load/Store Operations

Additional operations that we can perform with our updated datapath:

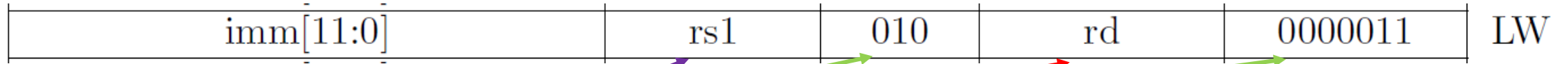
Operations using immediate values

# Example: Load Word

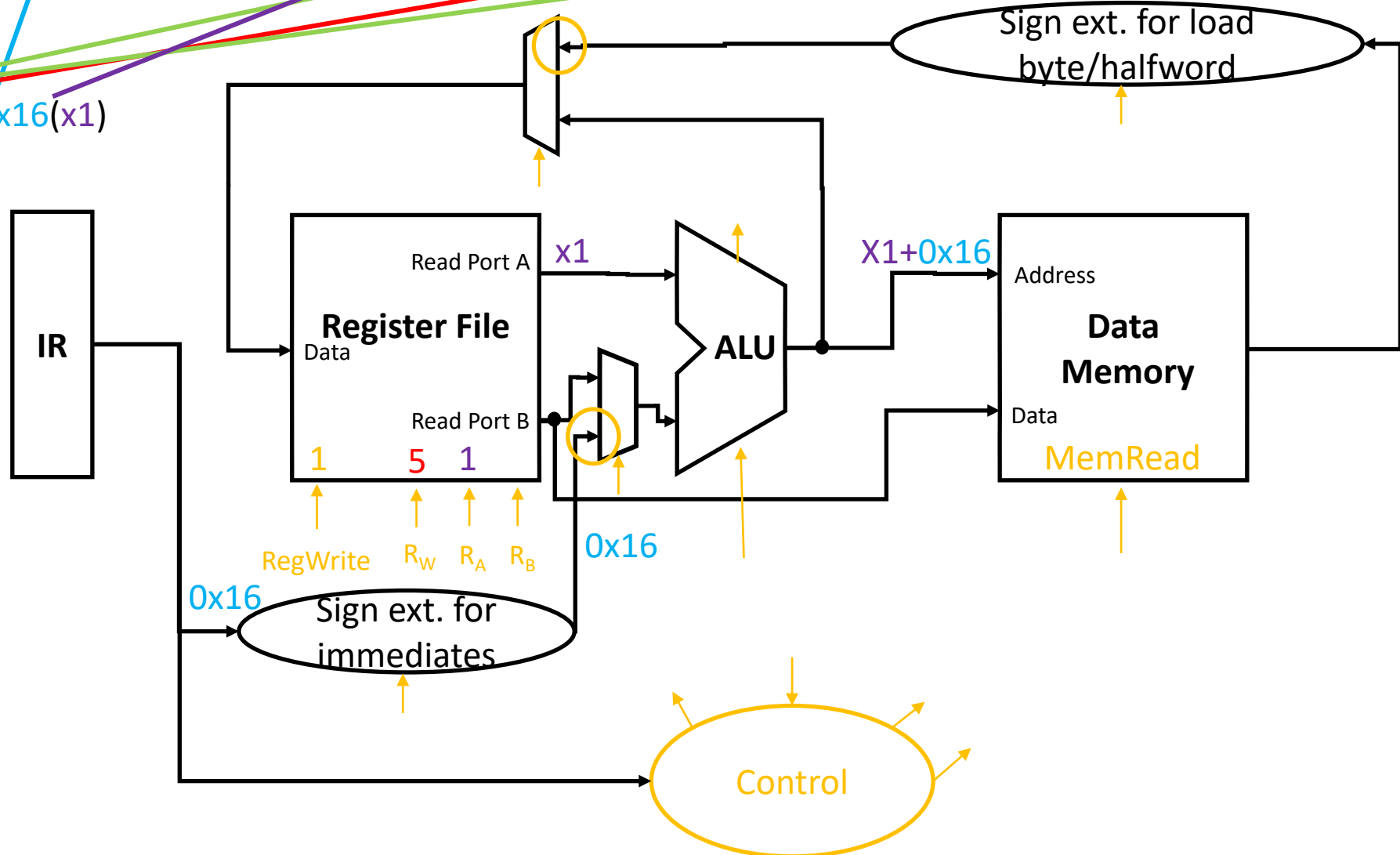
- Assembly:
  - LW rd, offset(rs1)
- Machine language



- Load from data from memory at address (rs1+imm) and store in rd
- Functionality:
  - Loads a word (32 bits / 4 bytes) from memory into a register
  - Example applications
    - load data from a pointer by setting offset to zero (LW rd, 0x0(rs1))
    - load data from a pointer providing a relative offset (LW rd, offset(rs1))
    - load data from a fixed address by setting rs1 to x0 (LW rd, addr(x0))



Example: LW x5, 0x16(x1)



# More Load Instructions

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

- LBU (Load Byte Unsigned) and LHU (Load Halfword Unsigned) work exactly the same way as LW (Load Word) except for the fact that they only load 8 bit /16 bit instead of 32 bit. The unused bits are zero
- LB and LH work like LBU und LHU, but perform sign extension for the upper bits

# Example: Store Word

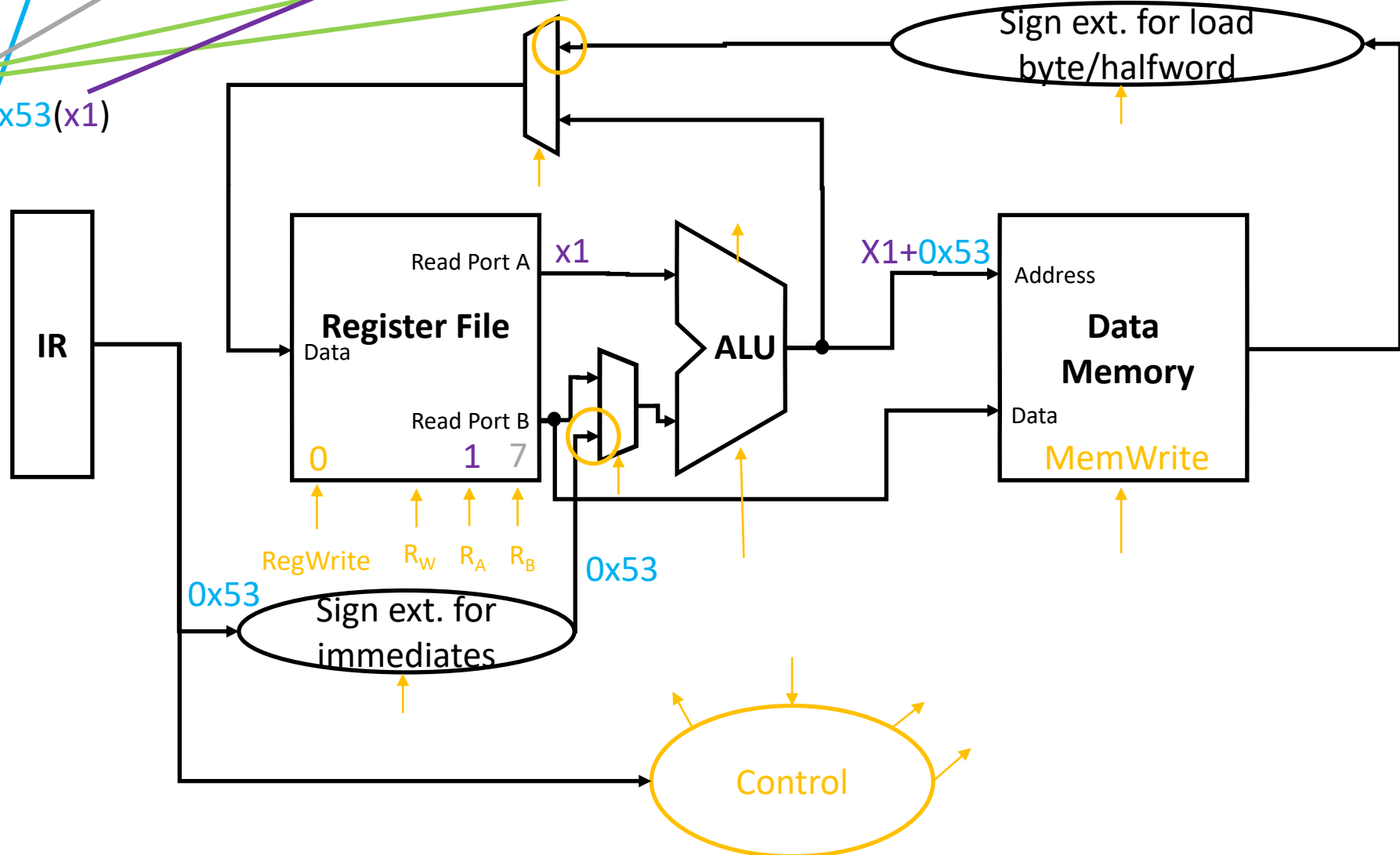
- Assembly:
  - SW rs2, offset(rs1)
- Machine language



- Store the value in rs2 to memory address (rs1+imm)
- Functionality:
  - Store a word (32 bits / 4 bytes) to memory
  - Example applications
    - store data to a pointer stored in a register by setting offset to 0 (SW rs2, 0x0(rs1))
    - store data to pointer + offset (SW rs2, offset(rs1))
    - store data to an absolute address (SW rs2, addr(x0))



Example: SW x7, 0x53(x1)



# More Store Instructions

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

- SB (Store Byte) and SH (Store Halfword) work exactly the same way as SW (Store Word) except for the fact that they only store the lowest 8 bit /16 bit of the rs2 register instead of the full 32 bit.
- Note that sign extension is not necessary for storing. To illustrate this consider the representation of -1 as 32 bit value and as 8 bit value.

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

Additional operations that we can perform with our updated datapath:

Load/Store Operations

Additional operations that we can perform with our updated datapath:

Operations using immediate values



# Example: ADDI

- Assembly:
  - ADDI rd, rs1, immediate
- Machine language

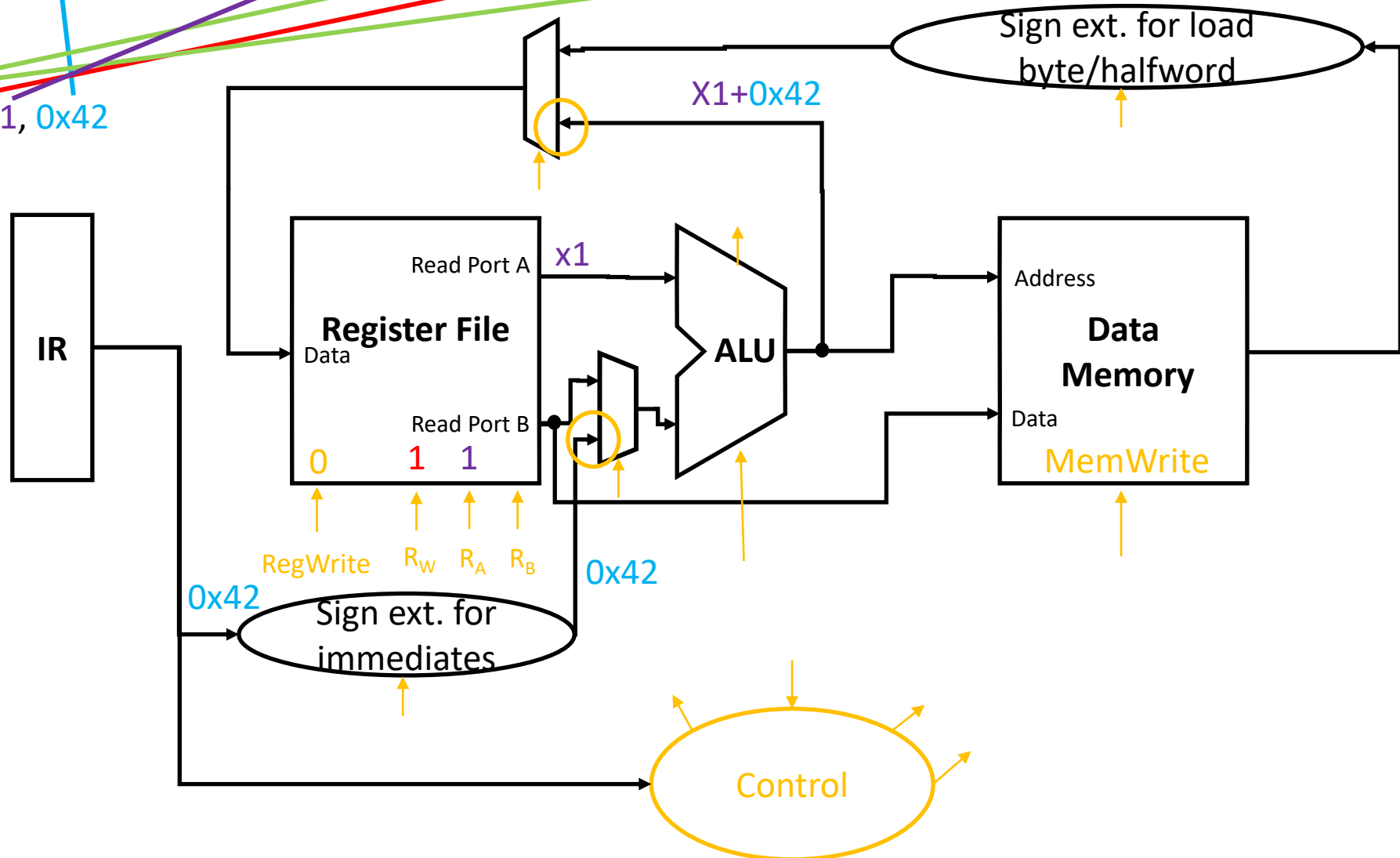
imm[11:0]	rs1	000	rd	0010011	ADDI
-----------	-----	-----	----	---------	------

- Computes  $rd = rs1 + imm$

- Functionality:
  - Computes  $rd = rs1 + imm$
  - Example applications
    - Move content of one register to another register by setting immediate to 0 (ADDI rd,rs1,0)
    - Set a register to a constant value by using x0 as source: (ADDI rd, x0, immediate)
    - Increment/decrement a register by setting rd=rs (e.g. ADDI x1, x1, 1)

imm[11:0] | rs1 | 000 | rd | 0010011 | ADDI

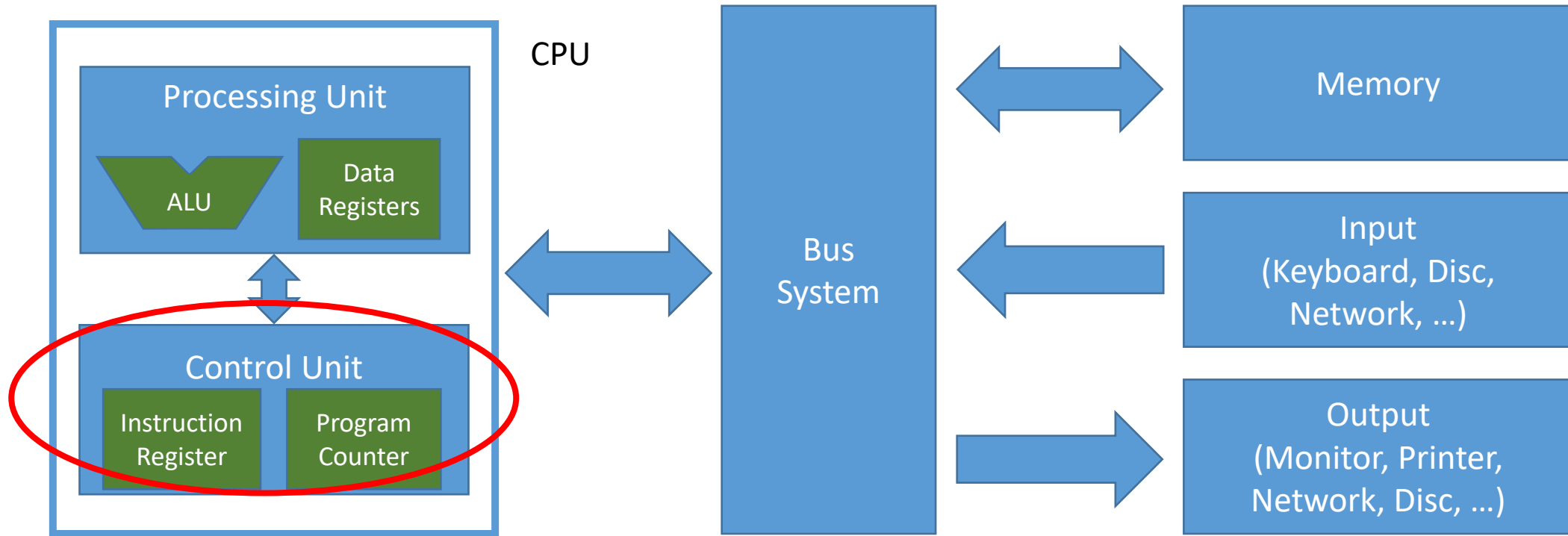
Example: **ADDI** x1, x1, 0x42



# More Operations with Immediates

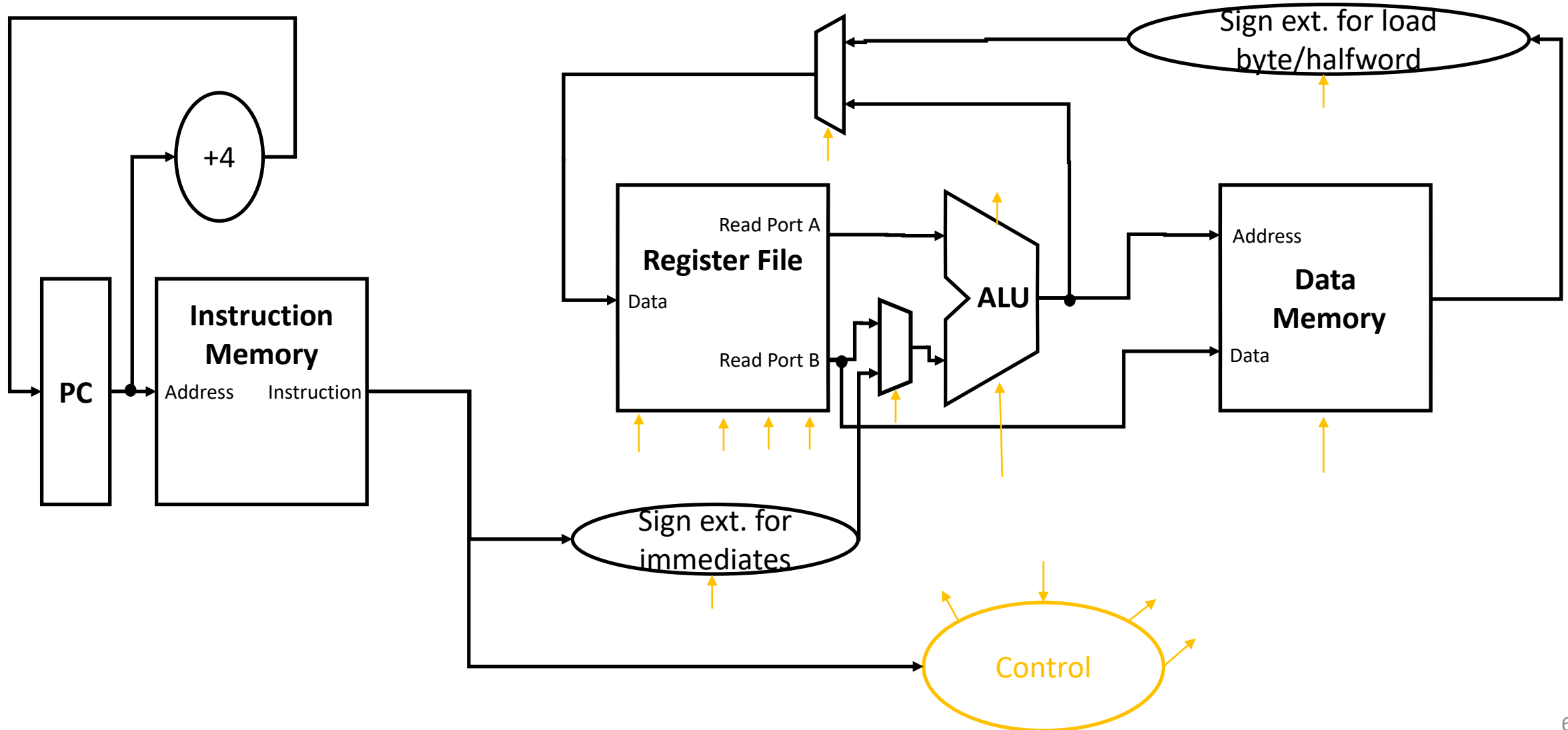
imm[31:12]				rd	0110111	LUI
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

- LUI allows to load 20 bits into the upper bits of a register; together with ADDI this allows to set a register to a 32 bit constant value
- SLTI sets the register rd to 1, if rs1 is less than the sign-extended immediate; SLTIU is the unsigned version
- XORI, ORI, ANDI are logic operations with immediates
- SLLI, SRLI, SRAI are shift operations, where the 5 bit immediate “shamt” defines the shift amount



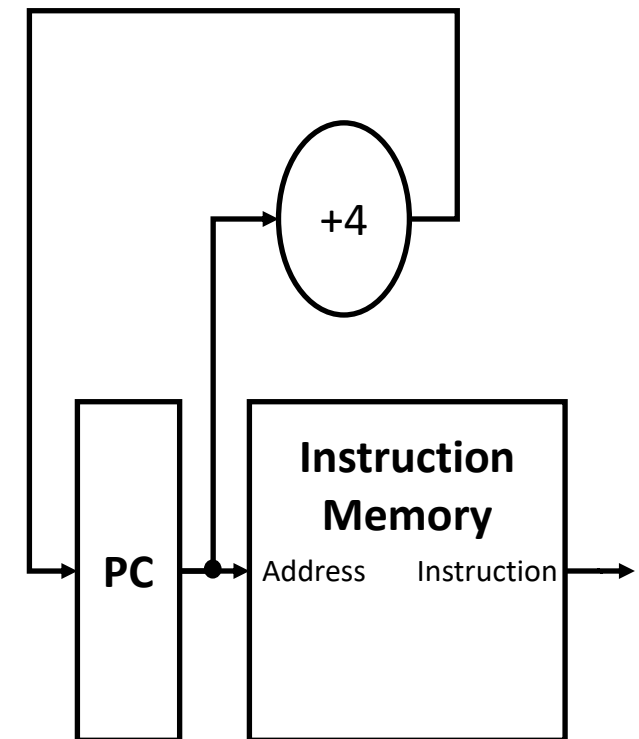
Let's learn about control!

# Adding Instruction Memory

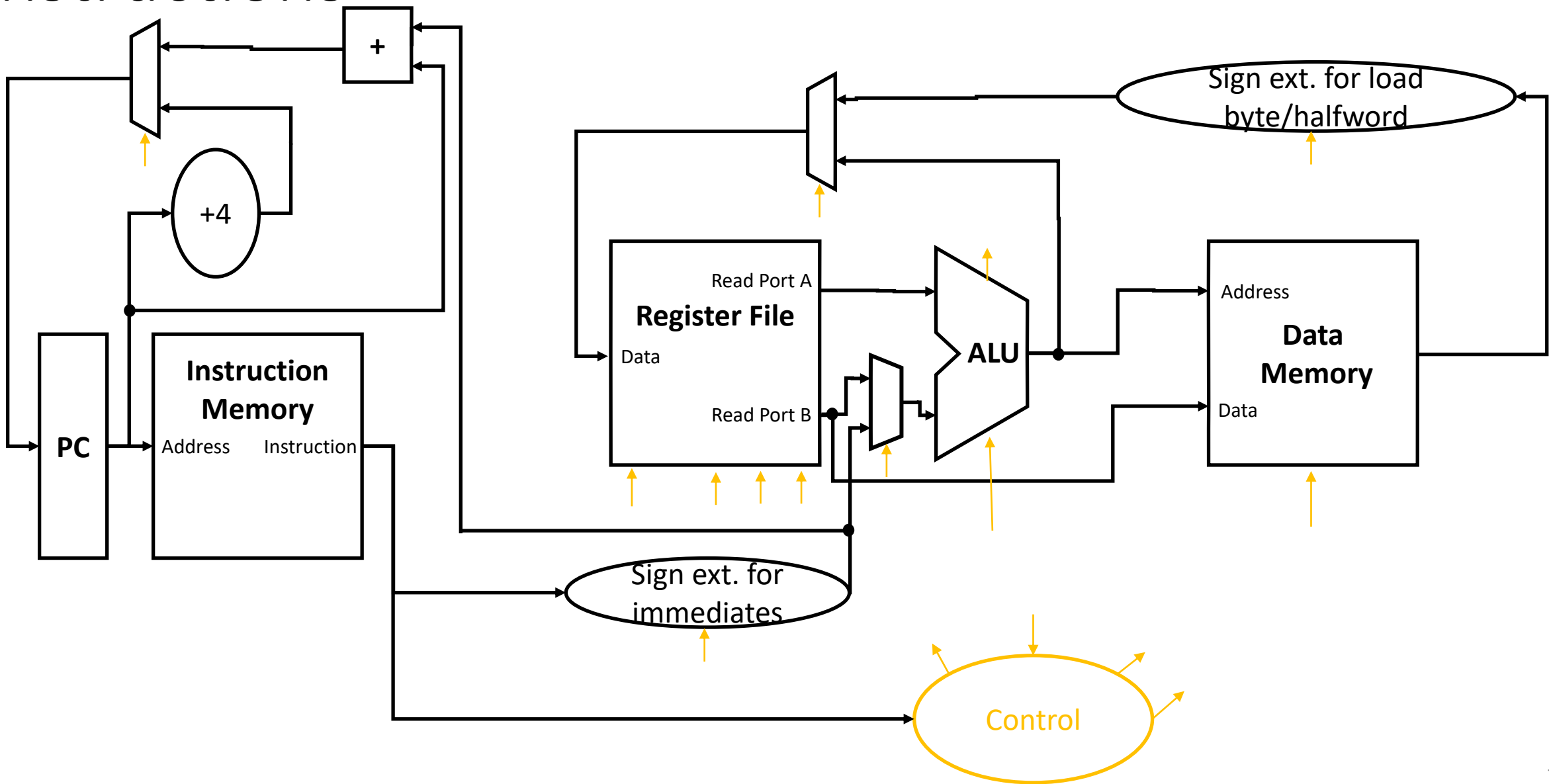


# Instruction Memory

- The instruction memory stores a sequence of instructions
- The program counter (PC) is incremented by 4 in each cycle and reads one instruction after the other
- This allows executing a static batch of instructions



# Extending the datapath for conditional branch instructions



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

Additional operations that we can perform with our updated datapath:

Conditional Branch Operations

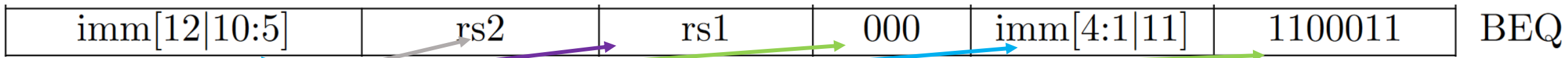


# Example: BEQ

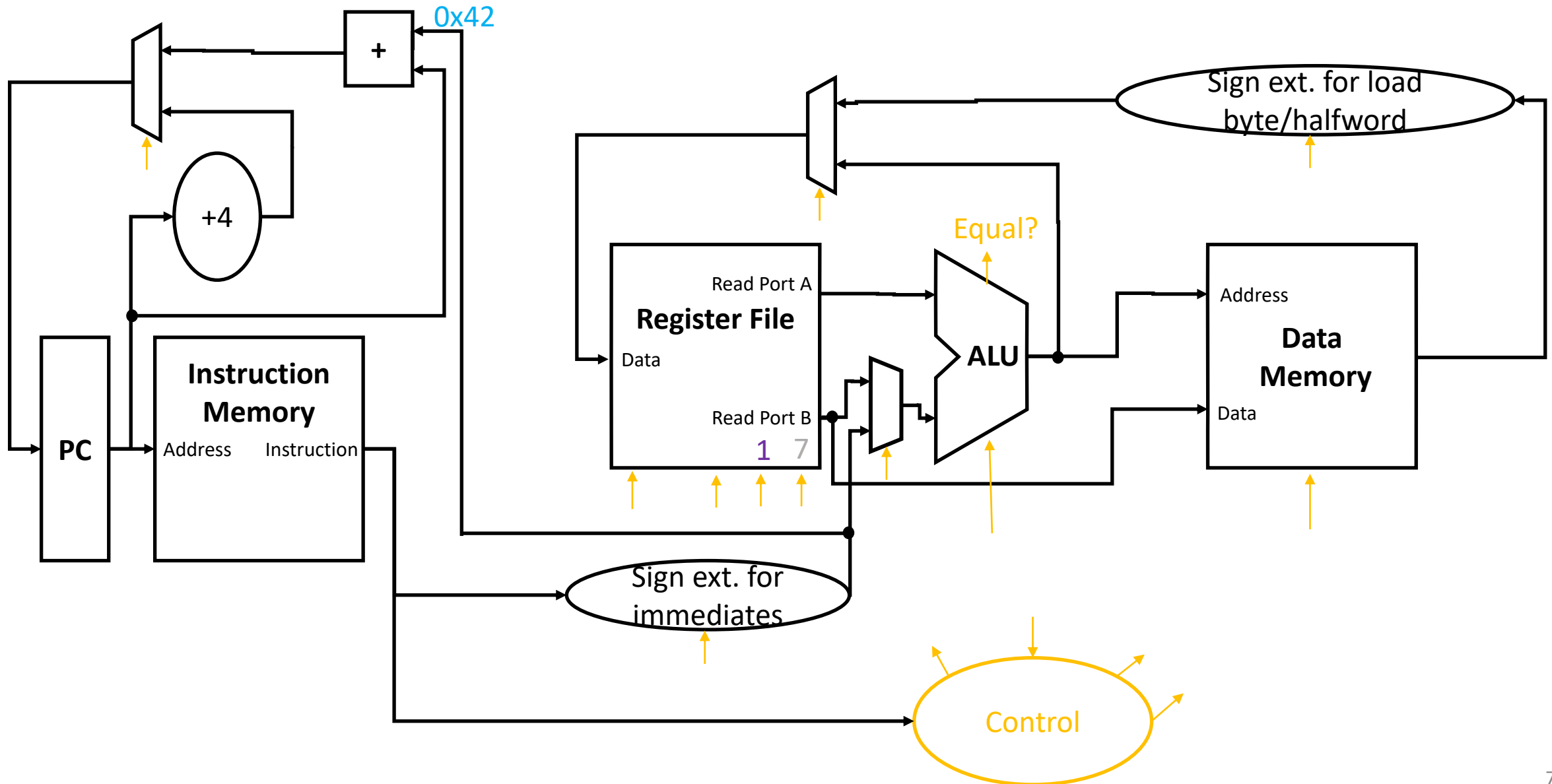
- Assembly:
  - BEQ rs1, rs2, offset
- Machine language

imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ
--------------	-----	-----	-----	-------------	---------	-----

- Branch to location PC + offset, if rs2 == rs1
- Functionality:
  - Branch if equal by to address PC + imm\*2
  - Example applications
    - Implement a branch to secure code, if password was entered correctly



Example: BEQ x1, x7, 0x42

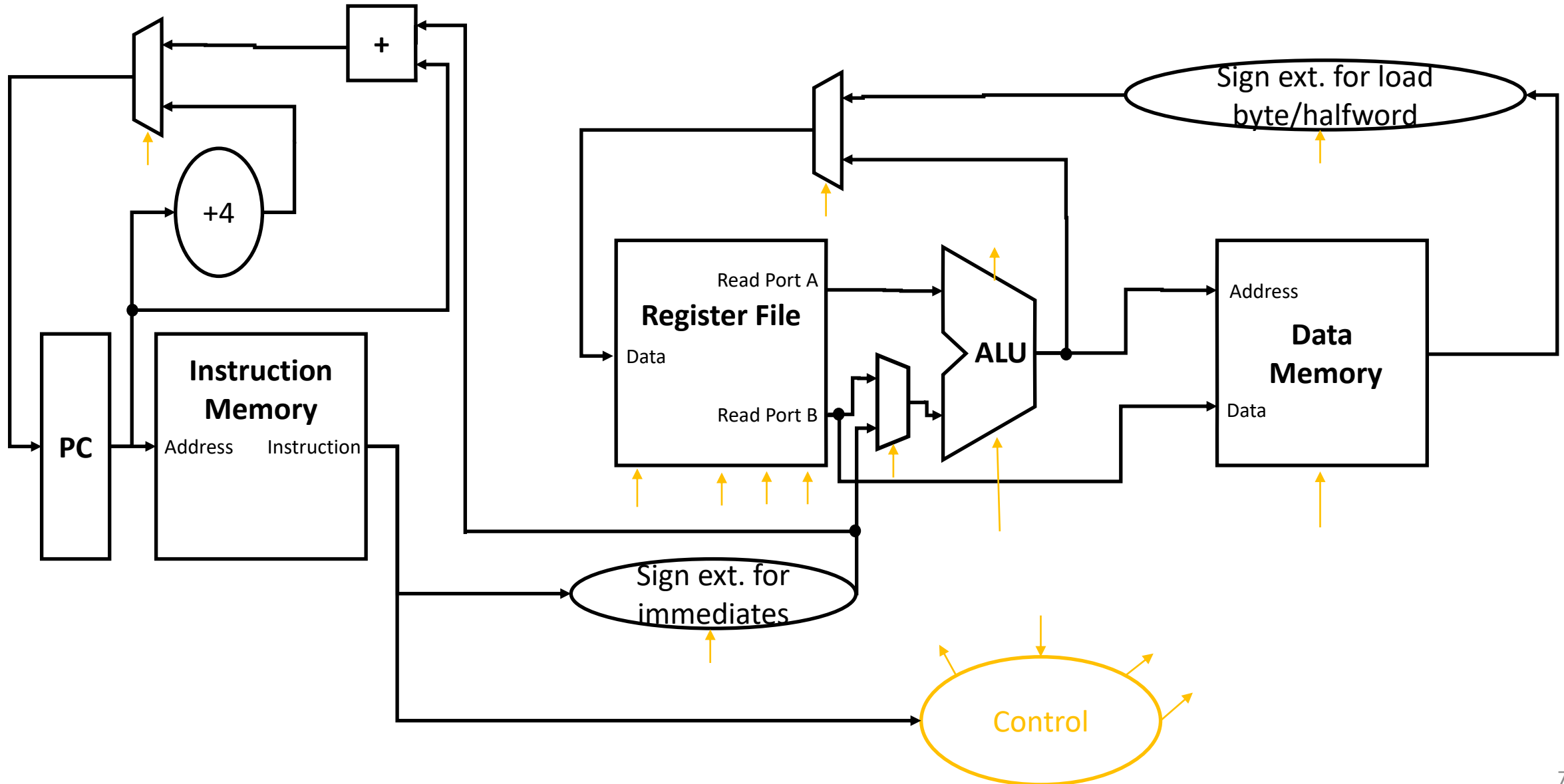


# More Conditional Branches

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

- BNE (Branch if not equal)
- BLT (Branch if less than)
- BGE (Branch if greater of equal)
- BLTU (Branch if less than unsigned)
- BGEU (Branch if greater of equal unsigned)

# High-Level Overview (Single Cycle Datapath)



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	

Conditional Branch Operations

Load/Store Operations

Operations using immediate values

Arithmetic/Logic operations

# JAL/JALR

imm[20 10:1 11 19:12]			rd	1101111	JAL
imm[11:0]	rs1	000	rd	1100111	JALR

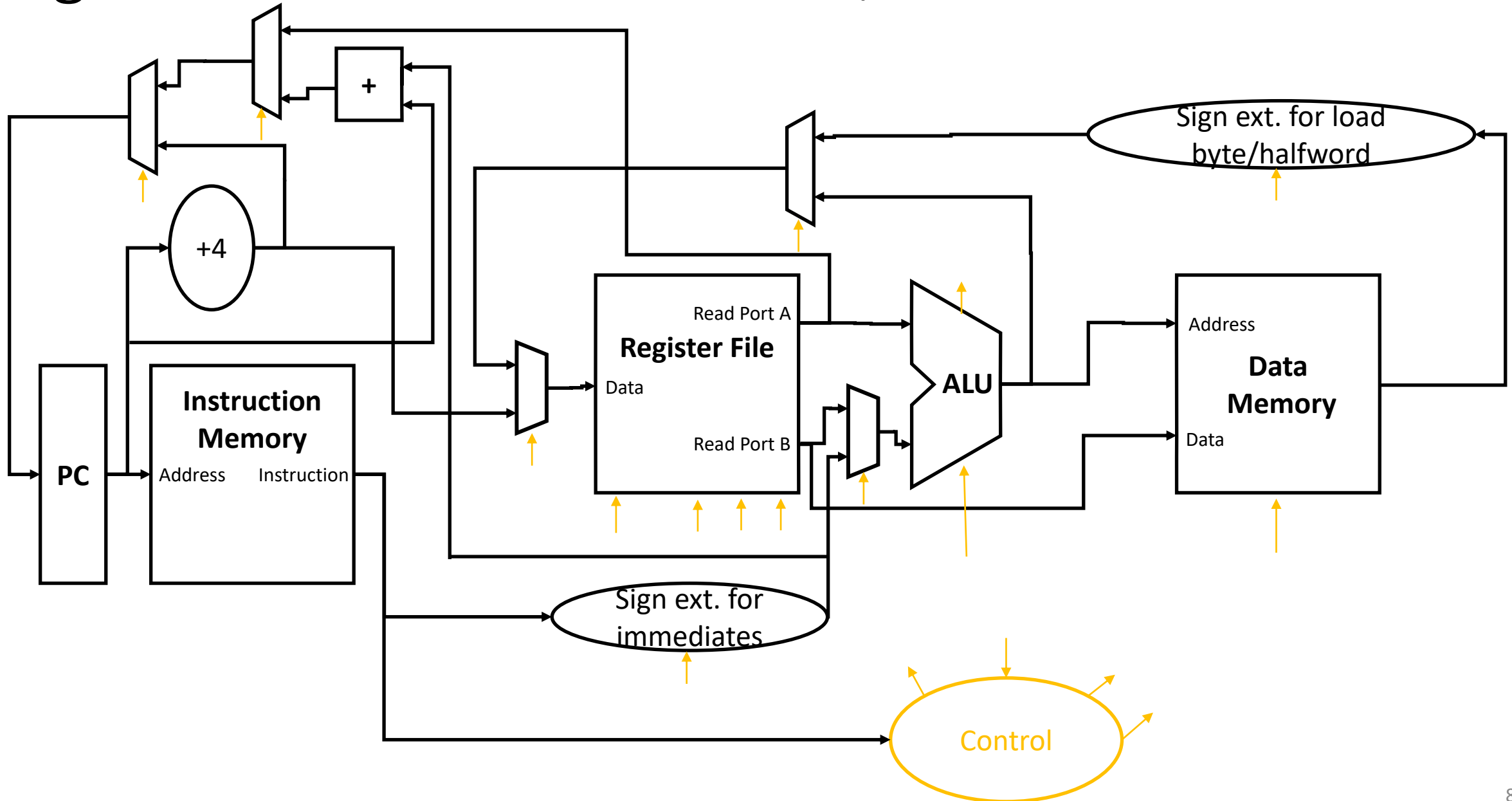
- Jump and Link (JAL):
  - Performs an unconditional jump to  $PC + imm * 2$
  - Stores the PC of the next instruction in rd
- Example applications
  - Unconditional jump (rd is set to x0 in this case)
  - Subroutine call (will be discussed later)

# JAL/JALR

imm[20 10:1 11 19:12]			rd	1101111	JAL
imm[11:0]	rs1	000	rd	1100111	JALR

- Jump and Link Register (JALR):
  - Performs an unconditional jump to  $rs1 + imm$
  - Stores the PC of the next instruction in rd
- Example applications
  - Subroutine call/return (will be discussed later)

# High-Level Overview incl. JAL/JALR





# Writing a First Program

# Simple Demo Program

- Load values from memory address 0x20, 0x24 into registers
- Add the registers together
- Store the result back to memory at 0x28
- Halt the CPU

# A First Mapping to Instructions

LW	rd = x1	rs1 = x0	<i>offset</i> = 0x20
LW	rd = x2	rs1 = x0	<i>offset</i> = 0x24
ADD	rd = x3	rs1 = x1	rs2 = x2
SW	rs2 = x3	rs1 = x0	<i>offset</i> = 0x28
EBREAK			

# Mapping to Encoding

Type	funct7	rs2	rs1	funct3	rd	opcode
I-Type	0x20		0	LW	1	LOAD
I-Type	0x24		0	LW	2	LOAD
R-Type	DEFAULT	2	1	ADD	3	ALU
S-Type	hi(0x28)	3	0	SW	lo(0x28)	STORE
I-Type	EBREAK		0	PRIV	0	SYSTEM

# Mapping to Binary

Type	funct7	rs2	rs1	funct3	rd	opcode
I-Type	0000001	00000	00000	010	00001	0000011
I-Type	0000001	00100	00000	010	00010	0000011
R-Type	0000000	00010	00001	000	00011	0110011
S-Type	0000001	00011	00000	010	01000	0100011
I-Type	0000000	00001	00000	000	00000	1110011

Instruction	Binary	Hexadecimal	Bytes
LW	000000100000000000010000010000011	0x02002083	83 20 00 02
LW	000000100100000000010000100000011	0x02402103	03 21 40 02
ADD	00000000001000001000000110110011	0x002081b3	b3 81 20 00
SW	00000010001100000010010000100011	0x02302423	23 24 30 02
EBREAK	00000000000100000000000001110011	0x00100073	73 00 10 00

# Putting the Program (Code and Data) into a single Memory

Instruction	Address	Value	Bytes
LW	0x00	0x02002083	83 20 00 02
LW	0x04	0x02402103	03 21 40 02
ADD	0x08	0x002082b3	b3 81 20 00
SW	0x0c	0x02302423	23 24 30 02
EBREAK	0x10	0x00100073	73 00 10 00
	0x14	0	00 00 00 00
	0x18	0	00 00 00 00
	0x1c	0	00 00 00 00
	0x20	42	2a 00 00 00
	0x24	13	0d 00 00 00
	0x28	0	00 00 00 00

# Tools to Write Assembler Code

- Writing instruction opcodes by hand is tedious
- An assembler is a tools to assemble machine code for us
- For this lecture we use `riscvasm.py`
- usage: `riscvasm.py program.asm -o program.hex`



# The Demo Program Written in Assembly

```
.org 0x00 # start program at address 0x00
LW x1, 0x20(x0)
LW x2, 0x24(x0)
ADD x3, x1, x2
SW x3, 0x28(x0)
EBREAK

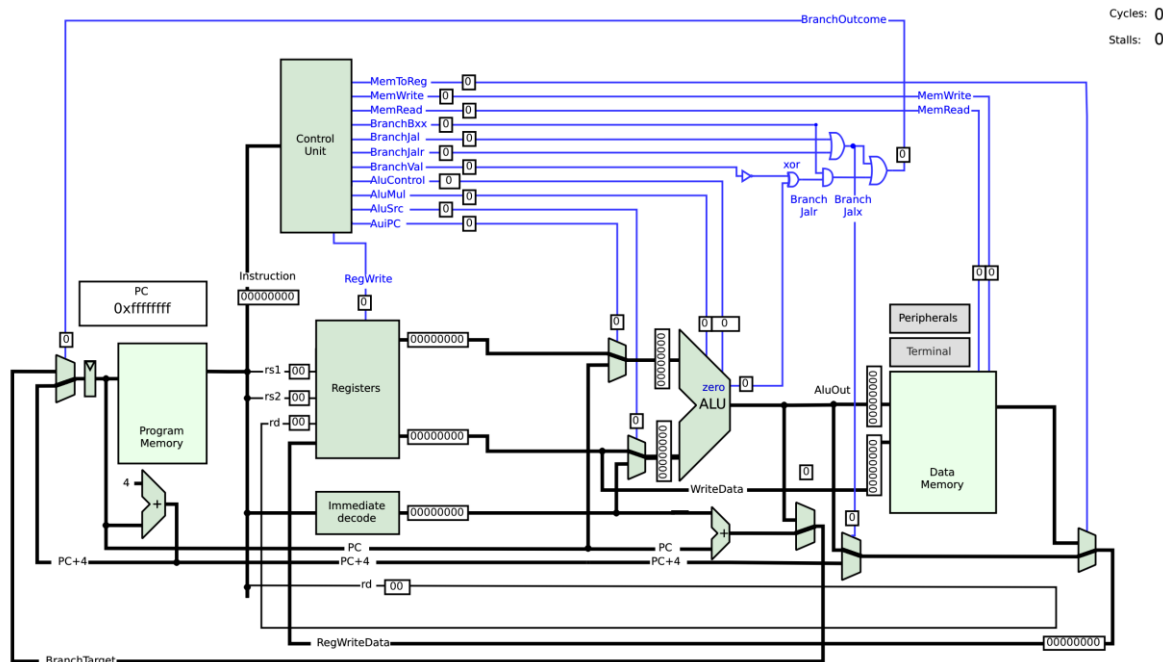
.org 0x20 # place data at address 0x20
# insert raw data instead of instructions
.word 42
.word 13
```

Try out to assemble and simulate your own code

**con04\_adding-two-constants**

# Watch the Hardware in Action with QtRVSim

Visit <https://comparch.edu.cvut.cz/qtrvsim/app/> or use qtrvsim in your virtual machine to visualize how a sequence of instructions becomes executed on the single-cycle datapath that we have built

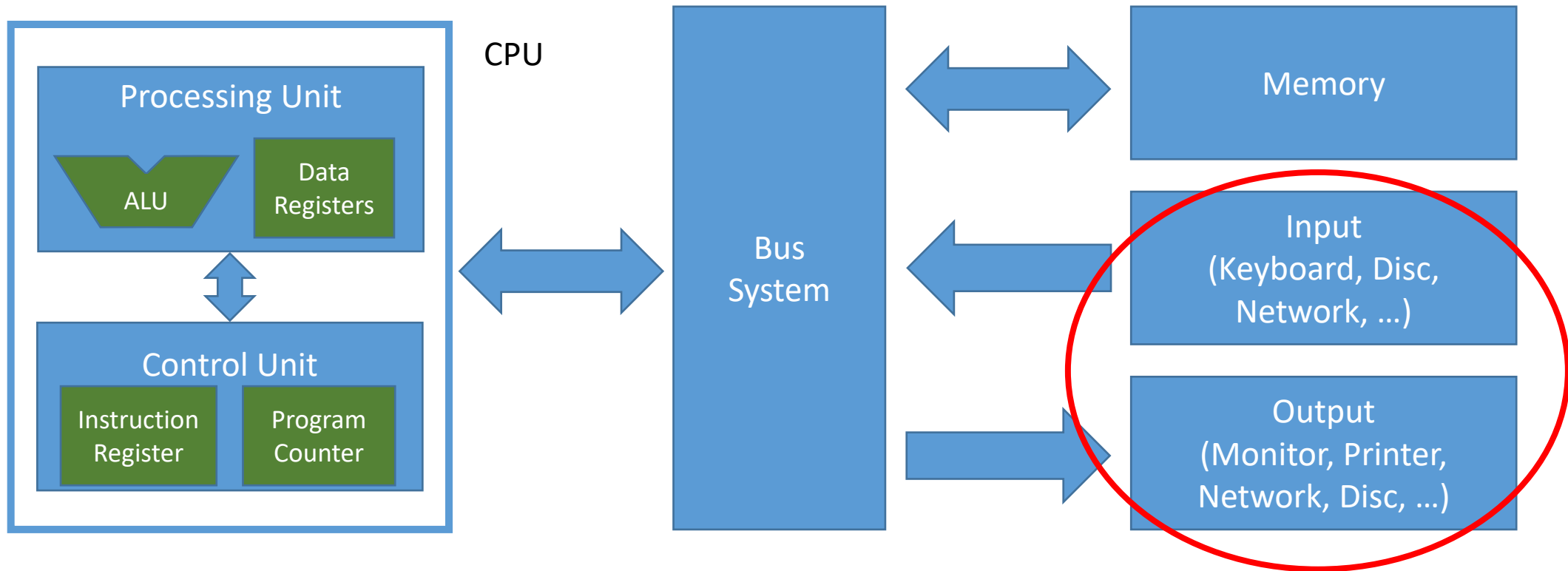


## Notes:

- riscviasm.py assumes that the instruction pointer starts at 0x0000, while QtRVSim starts at 0x0200
- You can upload and build assembly Code on the website; suitable source files are available in the QtRVSim directories of the examples repository

# Interfacing with I/O Devices

# 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	LLI	
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

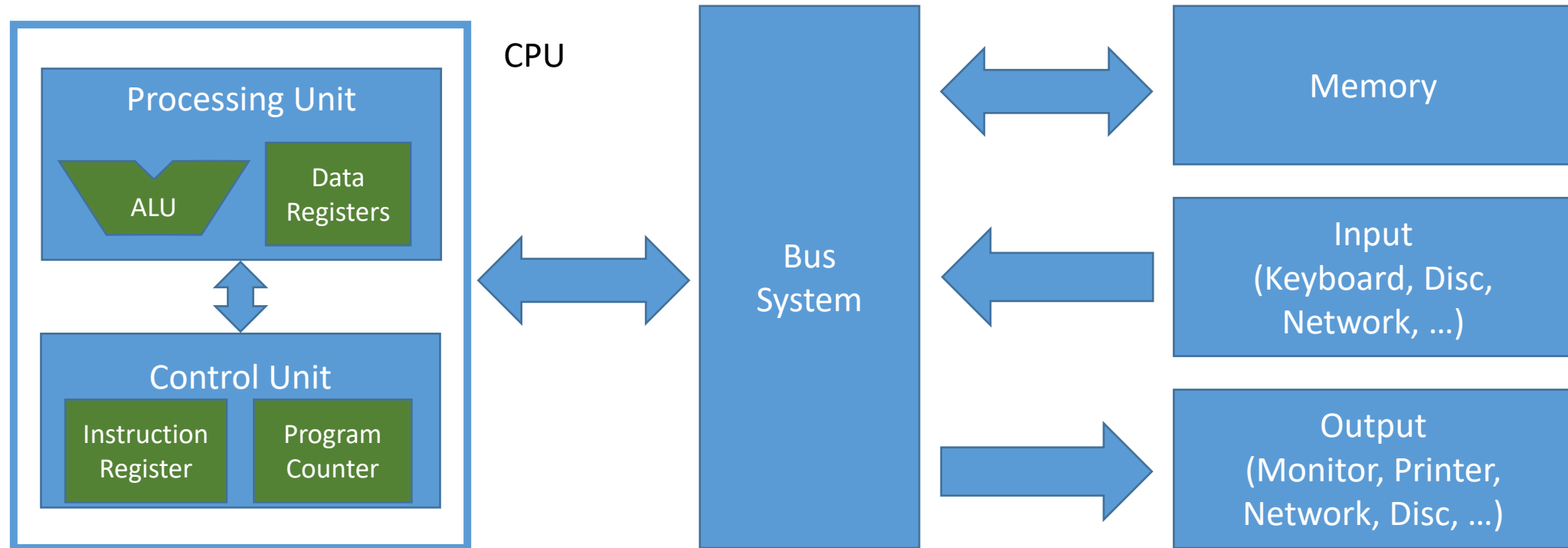
# How to Implement I/O?

- We access I/O and other devices like memory

→ we build memory-mapped peripherals

# Memory-Mapped Peripherals

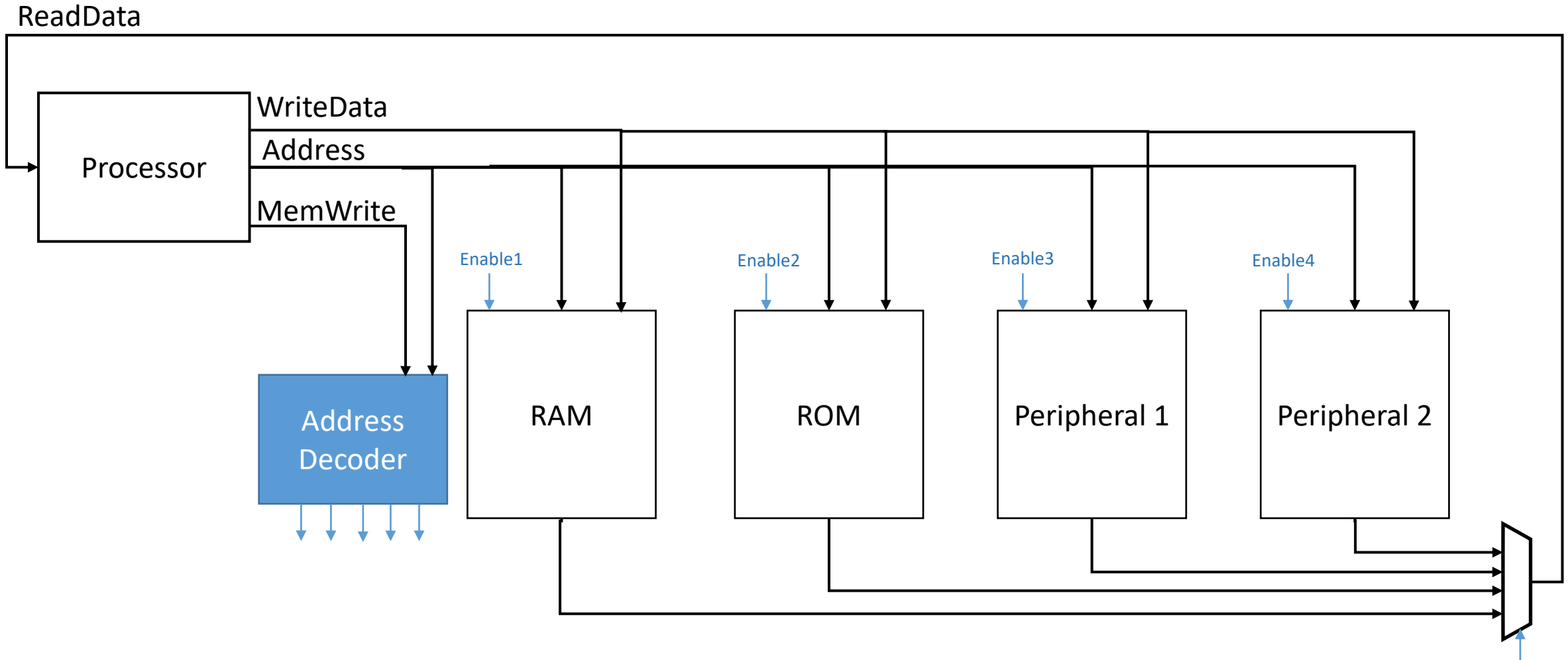
- Store and load instructions allow addressing 32-bit of memory space
- Not all the memory space that is addressable is used for actual memory
- We can split the memory space in pieces and assign a certain range to actual memory and other ranges to peripherals:
  - load/store operations write to registers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

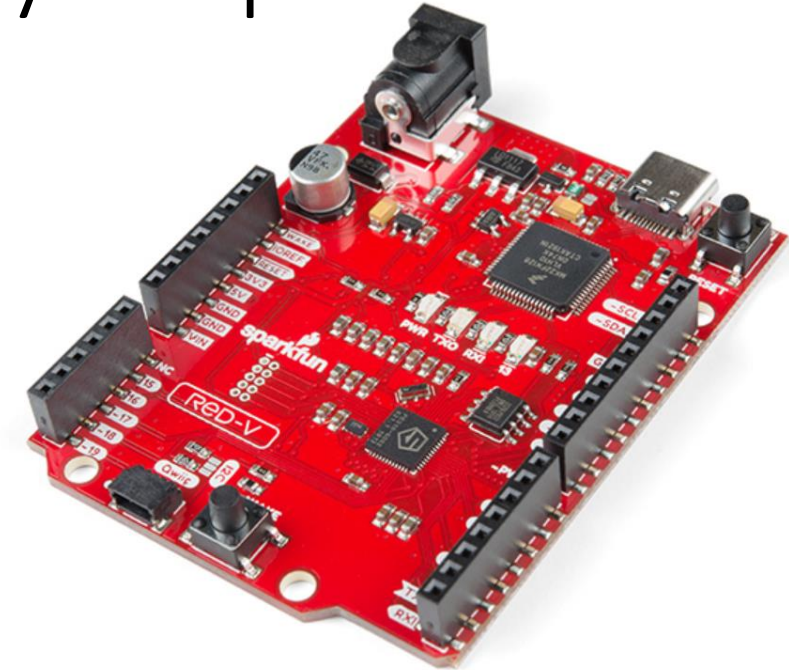
# Real-World Example of a Memory Map

Copyright © 2019, SiFive Inc. All rights reserved.

22

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	On-Chip Volatile Memory
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	
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

# The Core Used in Our Course: Micro RISC-V

- Micro RISC-V is a very simple CPU that we use for our introductory programming examples and the practical
- Micro RISC-V implements a large subset of R32I
- Tools and code for micro RISC-V
  - Code for Micro RISC-V and examples are available in the examples 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)

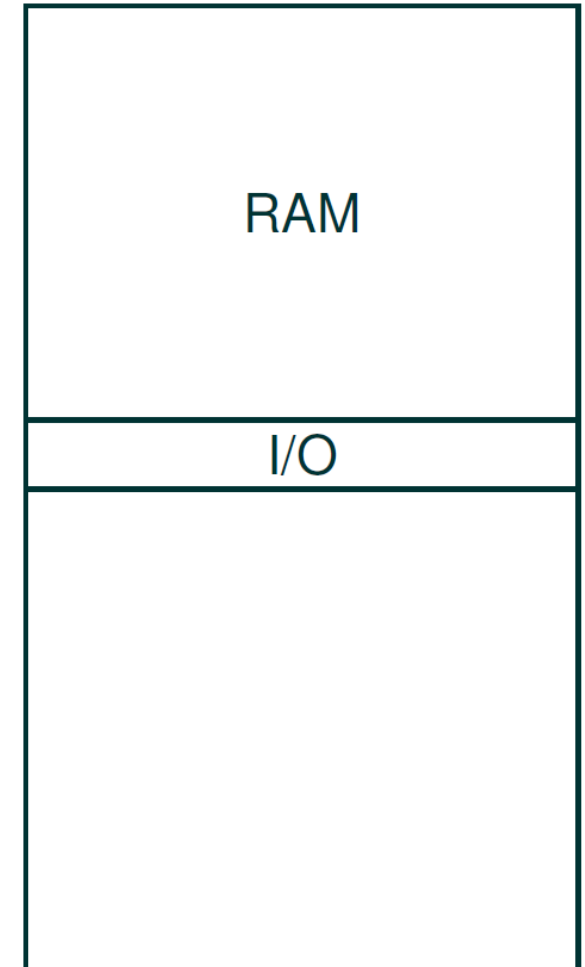
- The physical memory map is defined for each device depending on size of memory, peripherals, etc.

0x00000000

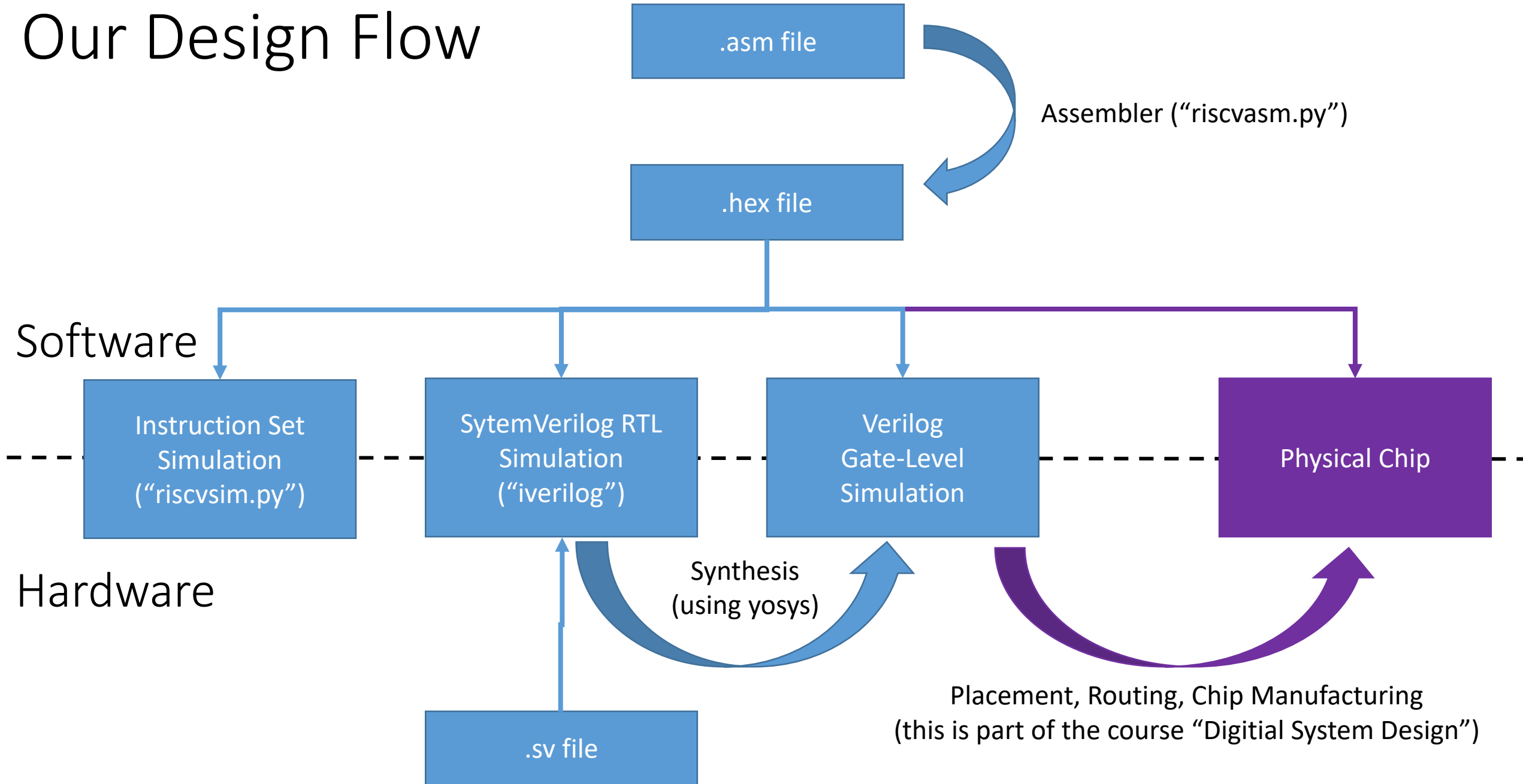
0x000007fc

0x00000800

0xffffffff



# Our Design Flow



# Notes 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 on Our Micro RISC-V CPU

## 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
```

# Peripherals of QtRVSim

- Also QtRVSim has peripherals.
- Let's set the color of LED RGB 1:

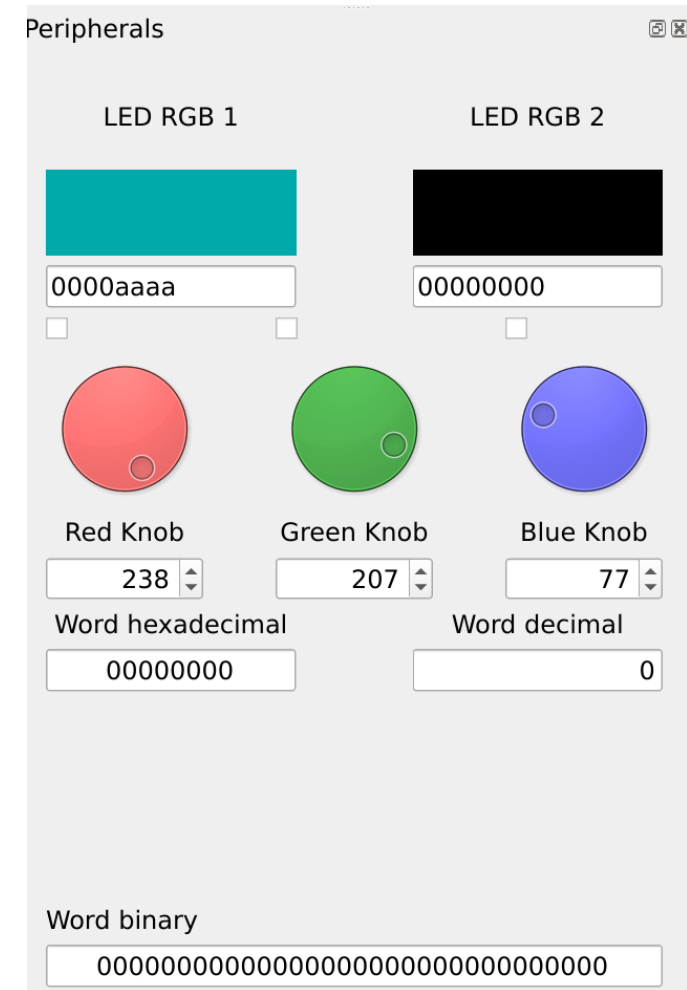
**see con04.02\_QtRVSim\_simple\_examples**

```

_start:
  li x1, SPILED_REG_BASE           // load base address of LED port
  li x2, 0xaaaa                   // Load color
  sw x2, SPILED_REG_LED_RGB1_o(x1) // Write color to memory mapped IO

  ebreak                          // Stop the program

```



# Common Pseudo-Instructions

These examples nicely show the value of having an x0 register that is always 0

To ease programming, there are pseudo-instructions for

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

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 $\geq$ 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 $\leq$
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if $>$ , unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if $\leq$ , unsigned
j offset	jal x0, offset	Jump