Computer Organization and Networks

(INB.06000UF, INB.07001UF)

Chapter 4: Basics on Processors

Winter 2020/2021

Stefan Mangard, www.iaik.tugraz.at

Limitations in 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" for a particular application by software

How to Build This?

- The most widely used approach is the Von Neumann Model – it is the basis of for example x86, ARM and RISC-V CPUs

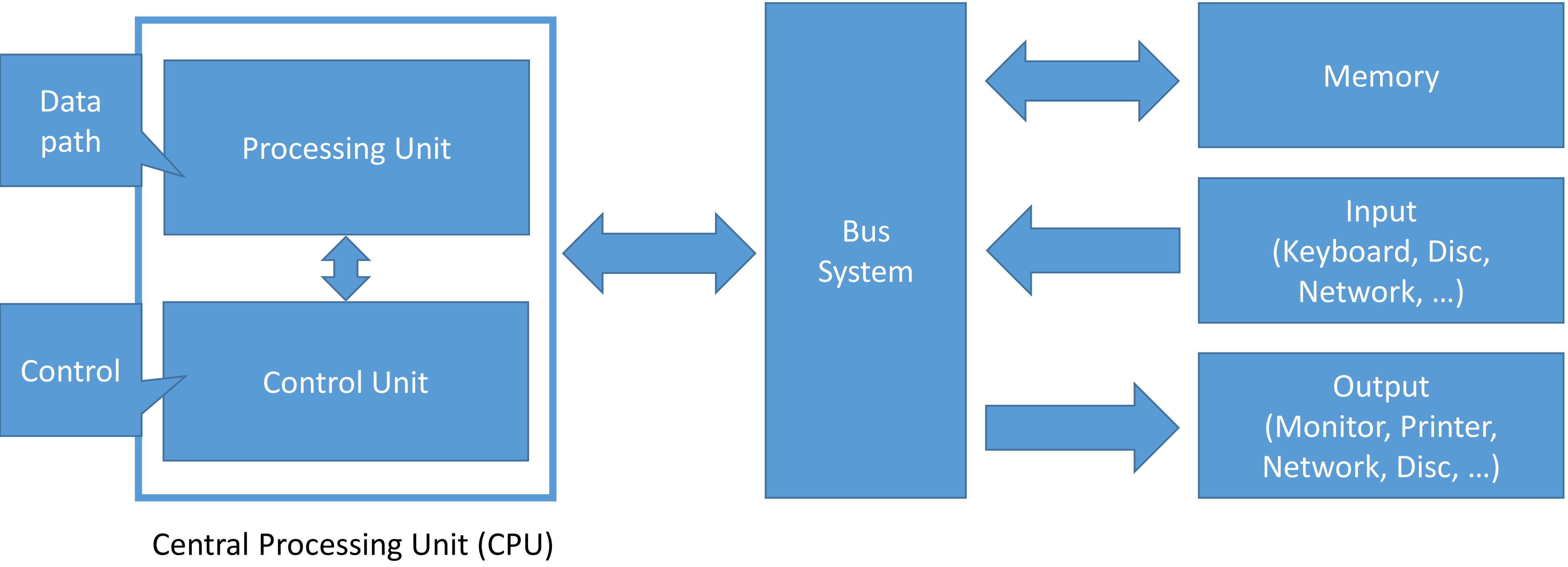
- It was proposed in 1945 by John Von Neumann (born in Budapest)

Von Neumann Model

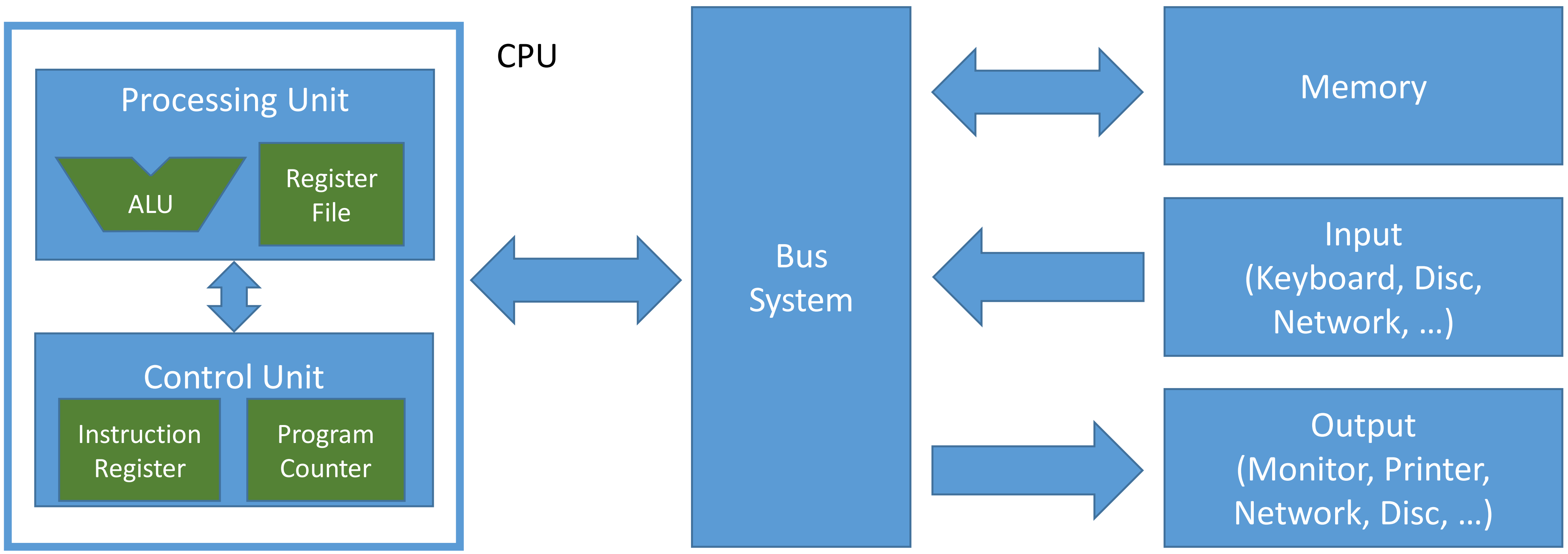
Von Neumann Model

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

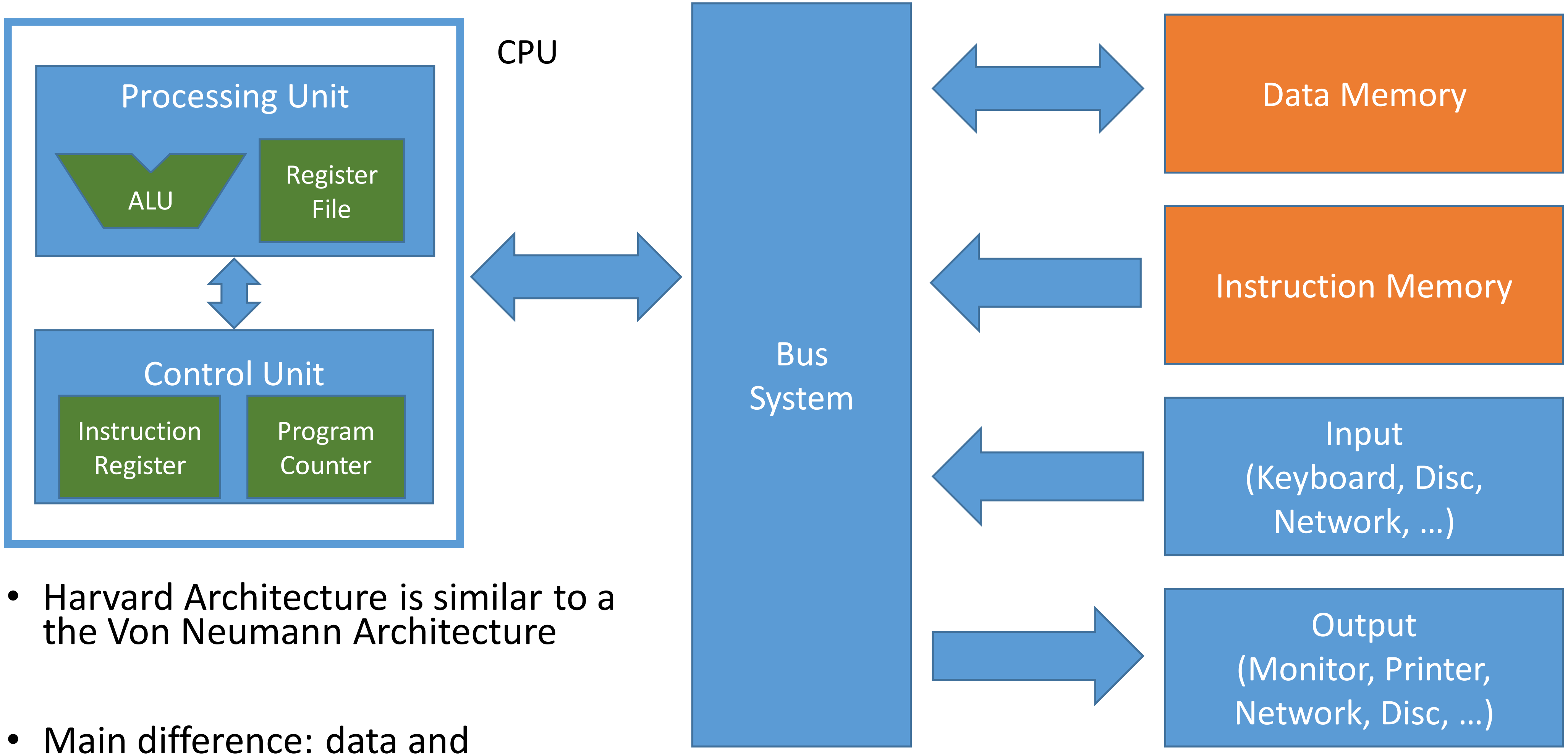
Von Neumann Model



Von Neumann Model

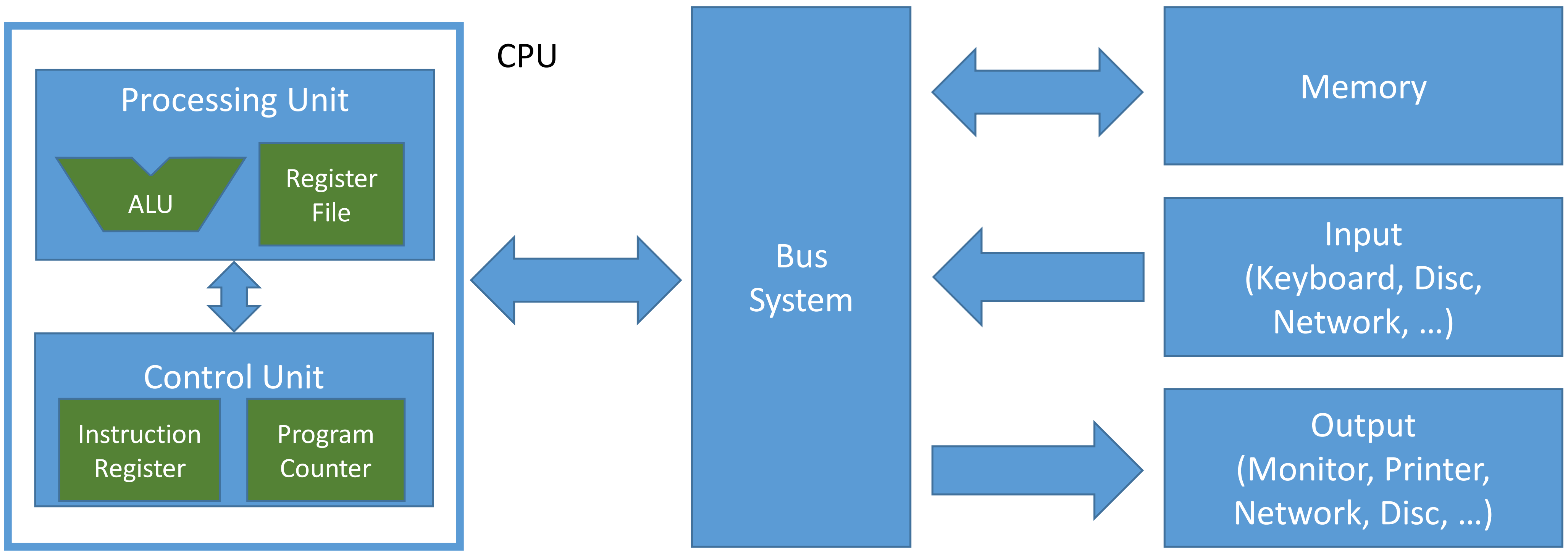


Harvard Architecture

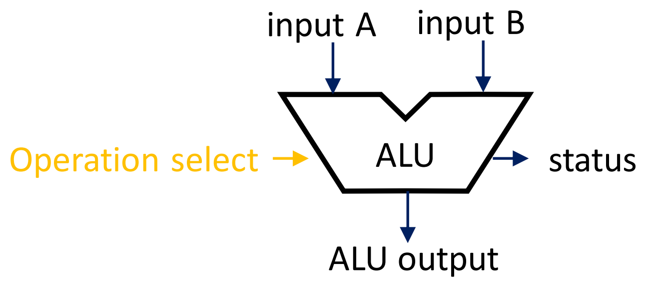


- Harvard Architecture is similar to a the Von Neumann Architecture
- Main difference: data and instruction memory are separated

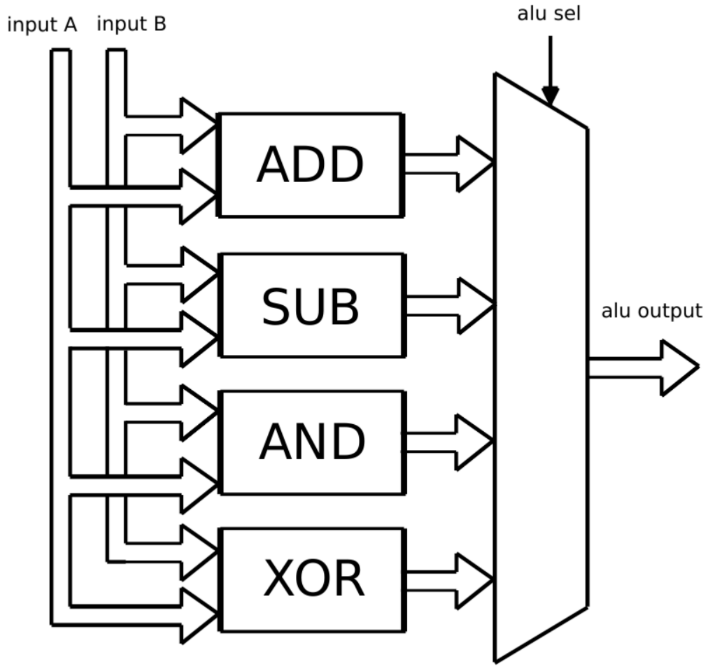
Von Neumann Model



Arithmetic Logic Unit (ALU)

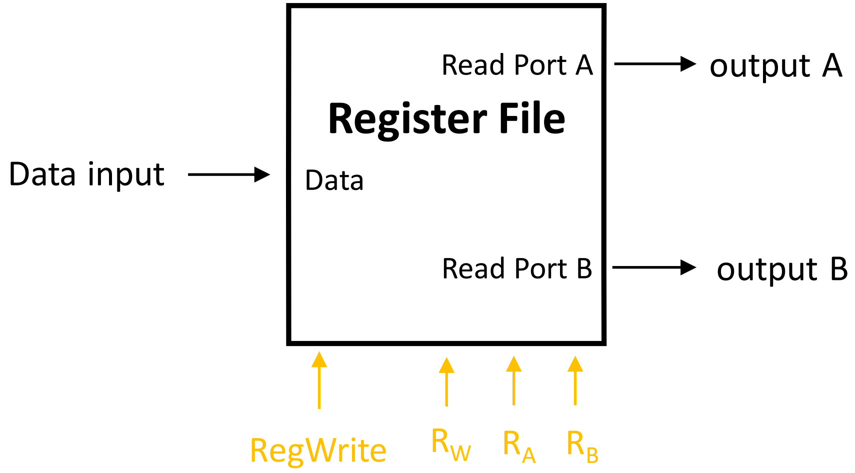


- The ALU is a combinational circuit performing calculation operations



- 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

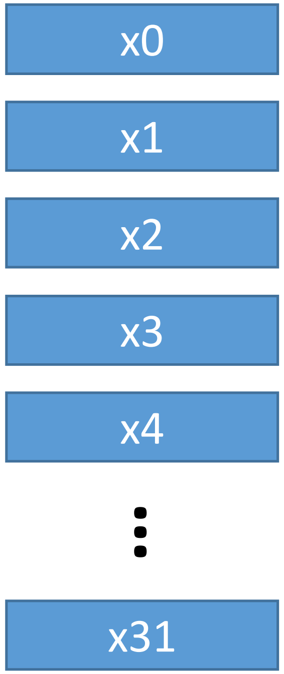
Register File



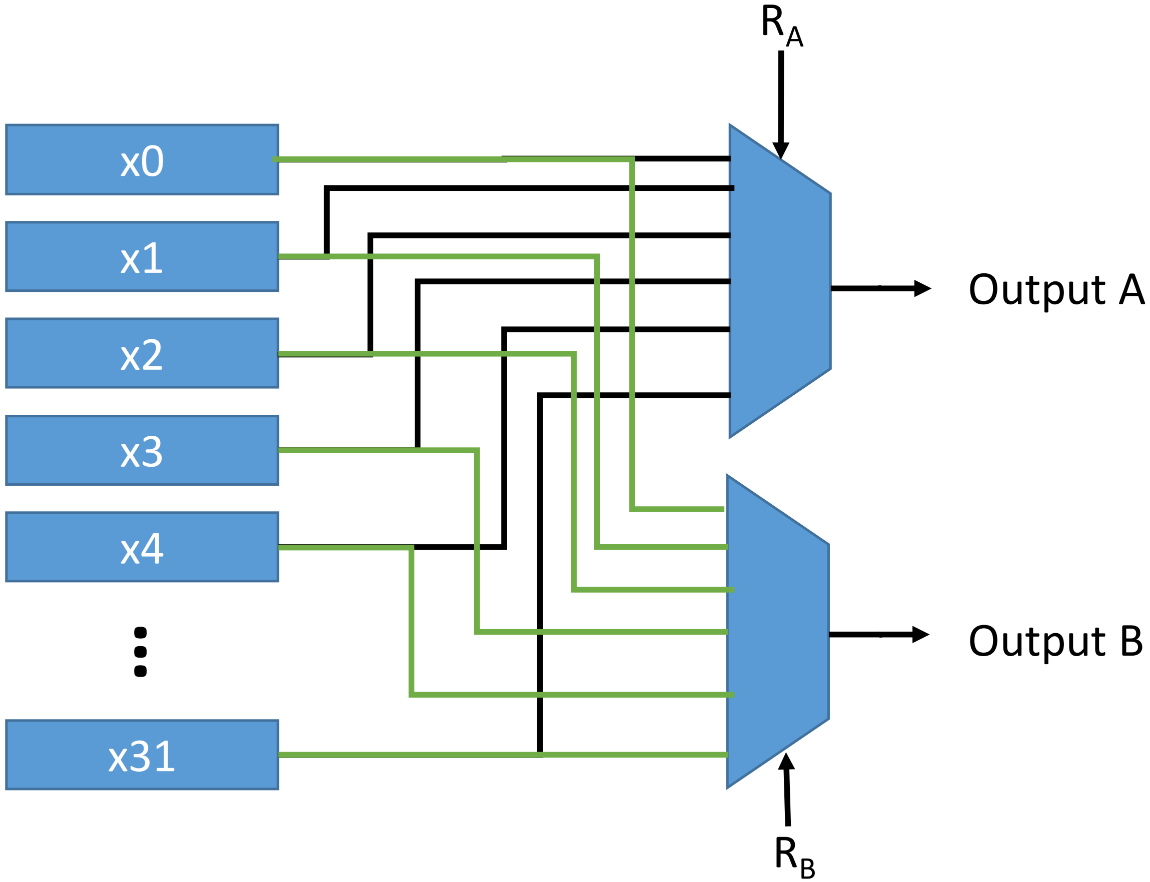
- The register file contains m n-bit registers
- In a given clock cycle one n-bit value can be stored in the register selected via the signal R_W; In case RegWrite is low, no register is written
- In each cycle two registers can be read and are provided at the outputs A and B. The registers to be read 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)

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



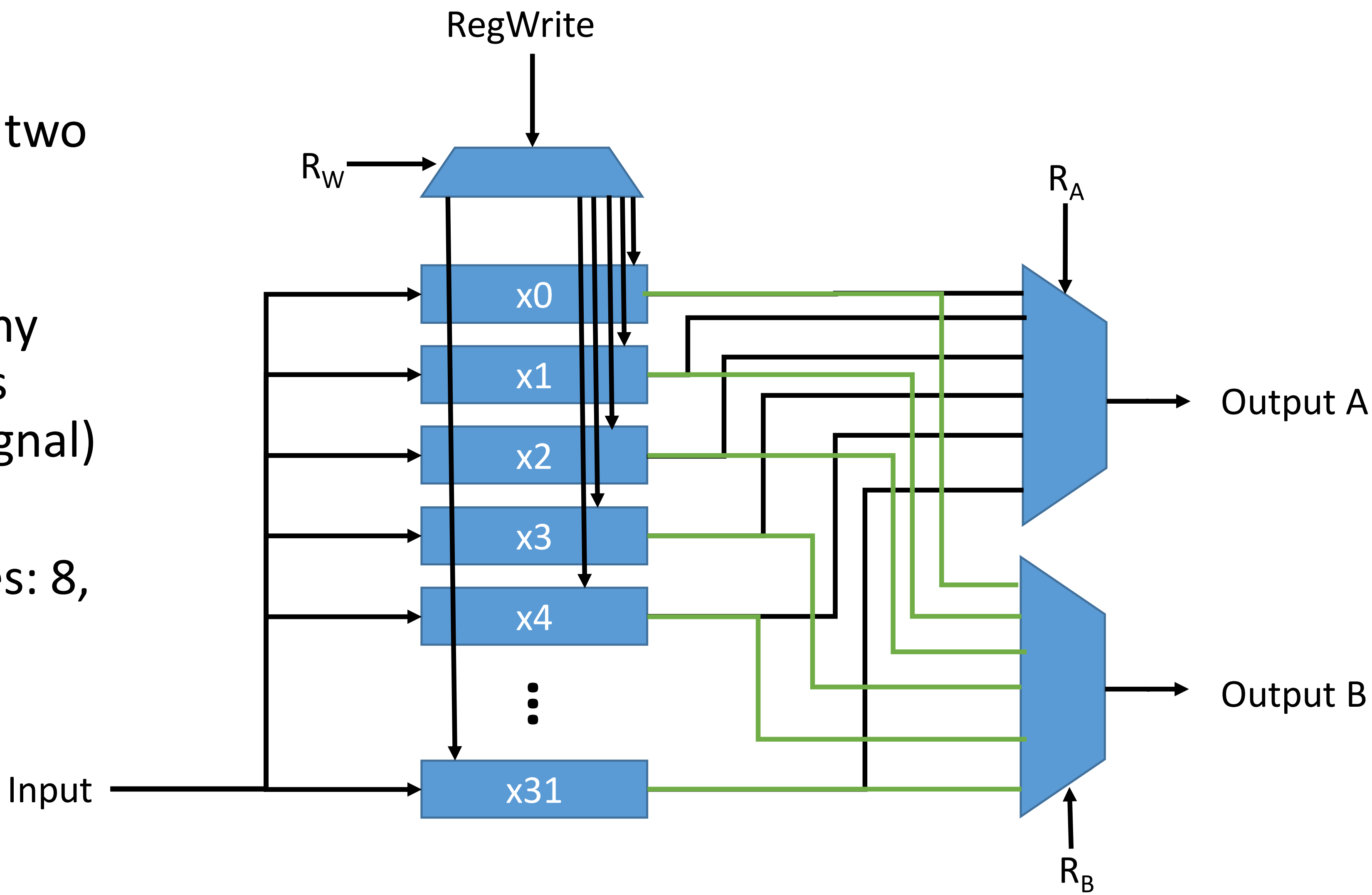
Data Registers (Register File)



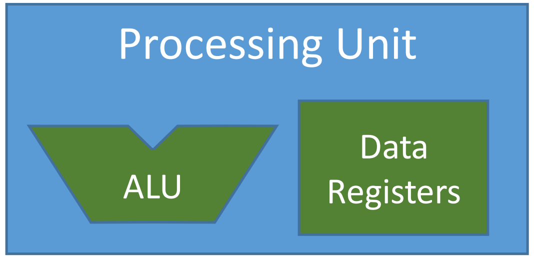
Data Registers (Register File)

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

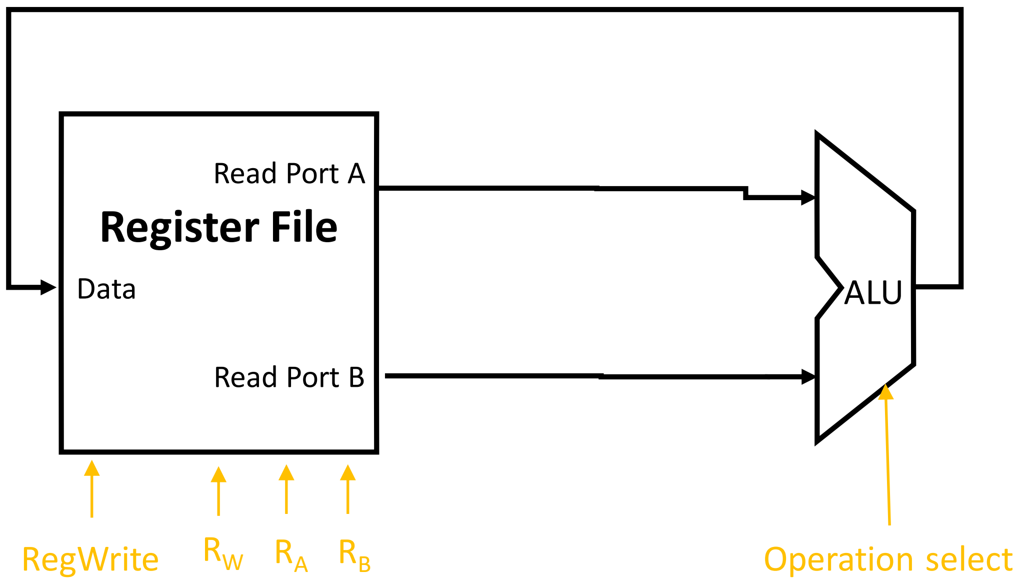


Processing Unit

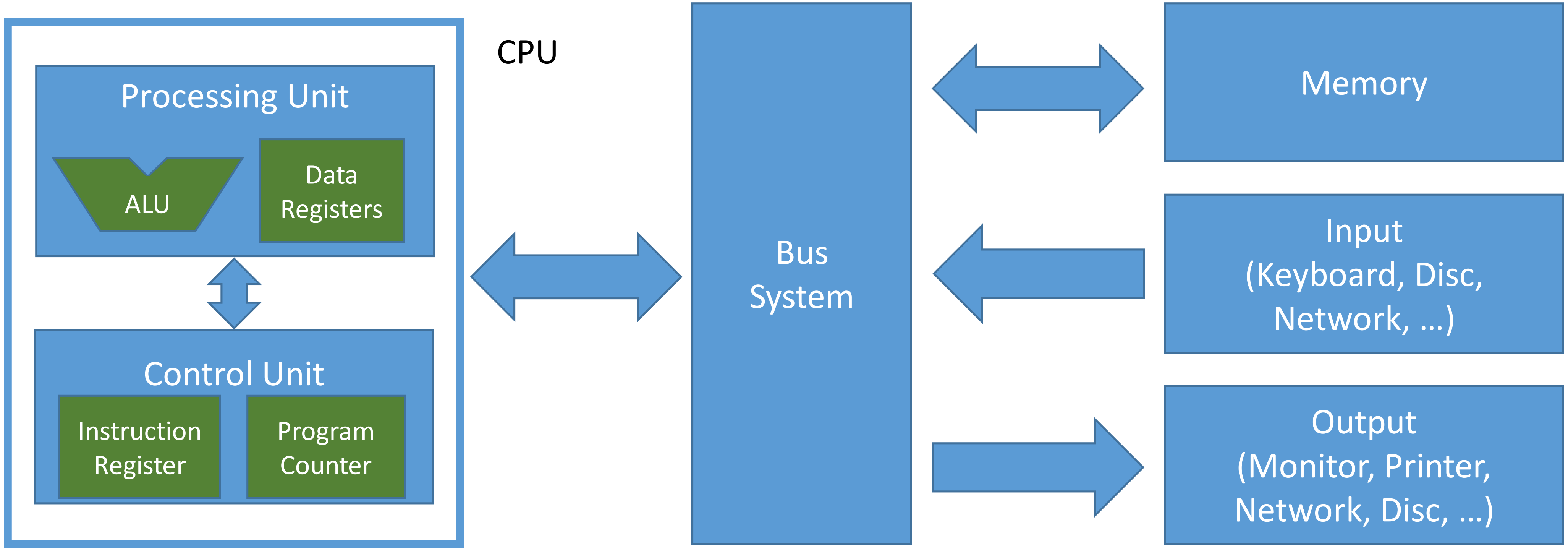


- The processing unit constitutes the data path of the CPU
- Based on control signals that are provided as inputs operations are performed in the ALU and data registers are updated

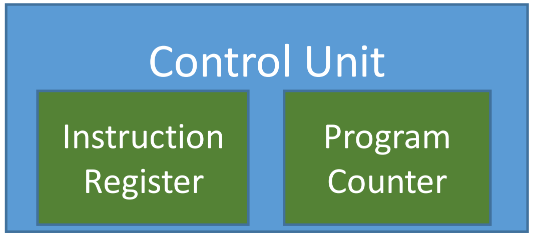
A First Simple Datapath for Our CPU



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

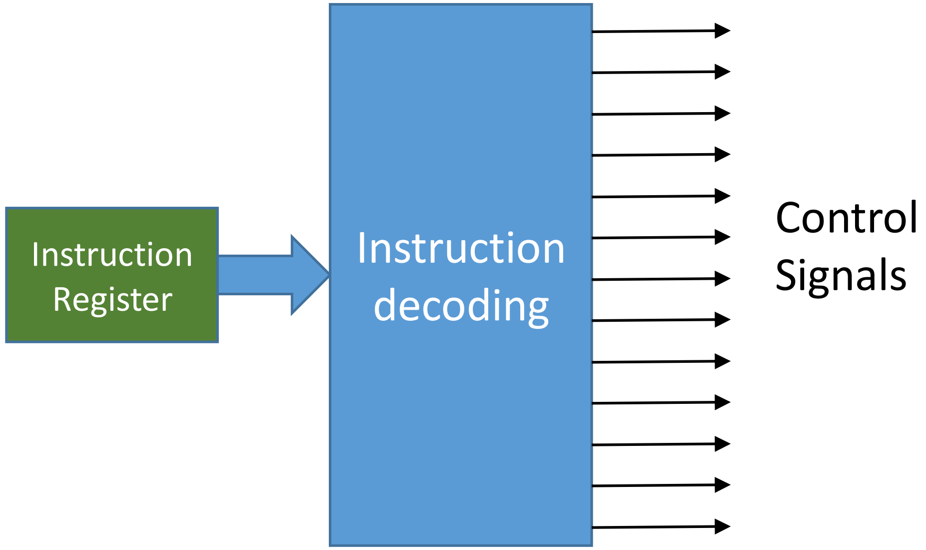

CPU
Processing Unit
ALU
Data Registers
Control Unit
Instruction Register
Program Counter
Bus System
Memory
Input (Keyboard, Disc, Network, …)
Output (Monitor, Printer, Network, Disc, …)

Instruction Register

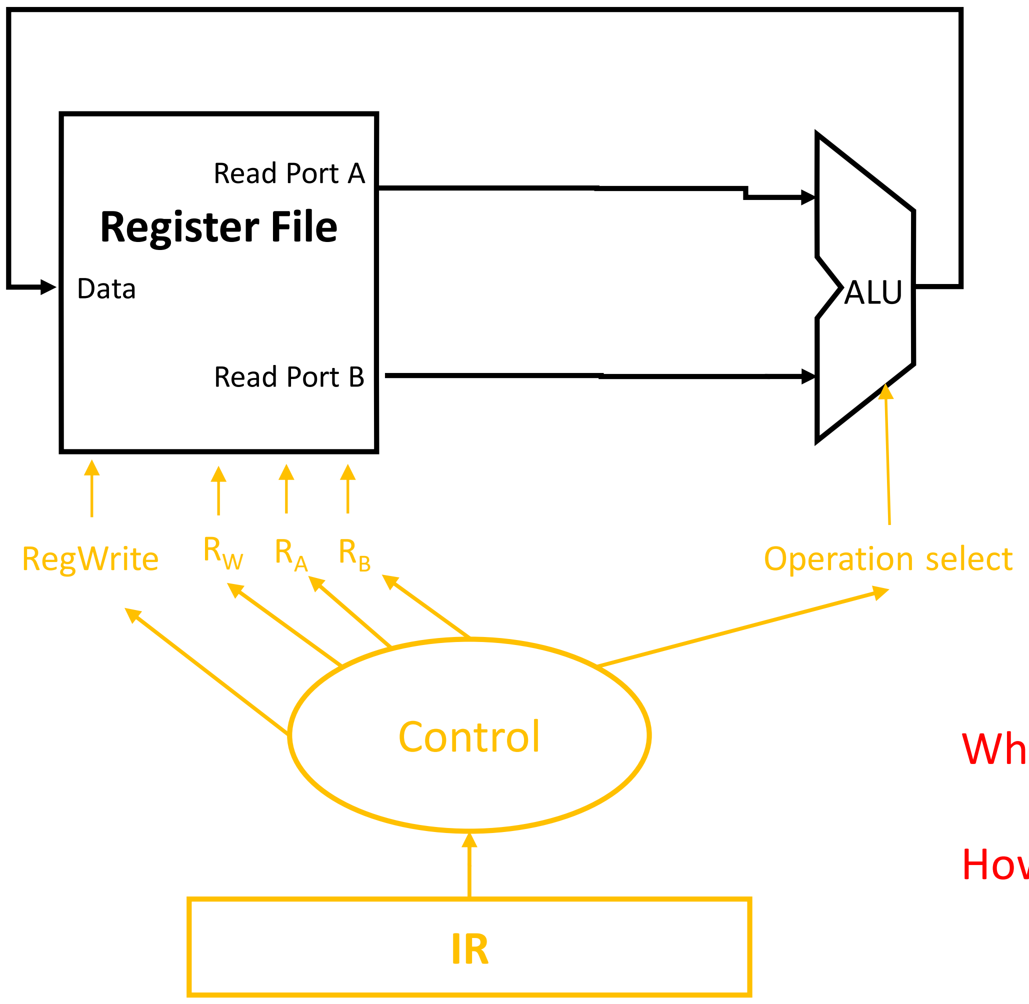


- The instruction register stores the instruction that shall be executed by the data path

- The instruction decoder maps the instruction register to control signals



A First Simple Datapath with Control for Our CPU



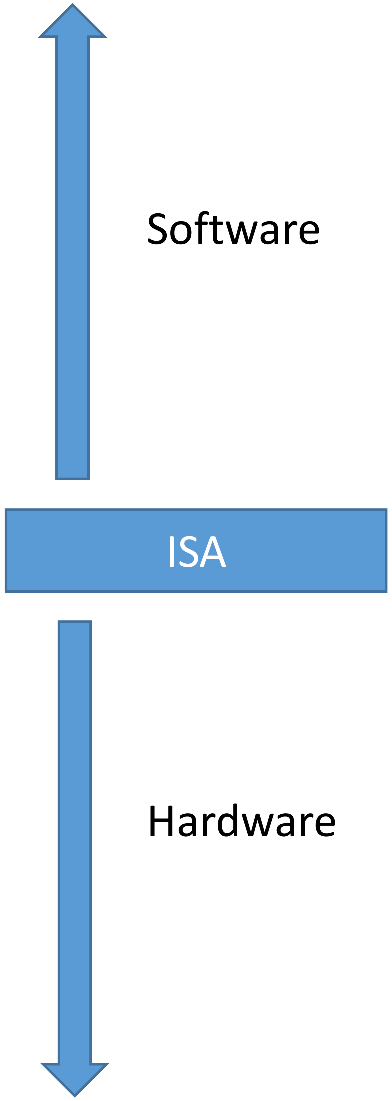
What is an instruction?

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
- 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
- The ISA is the interface between hardware and software



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

Competition Between Instruction Sets

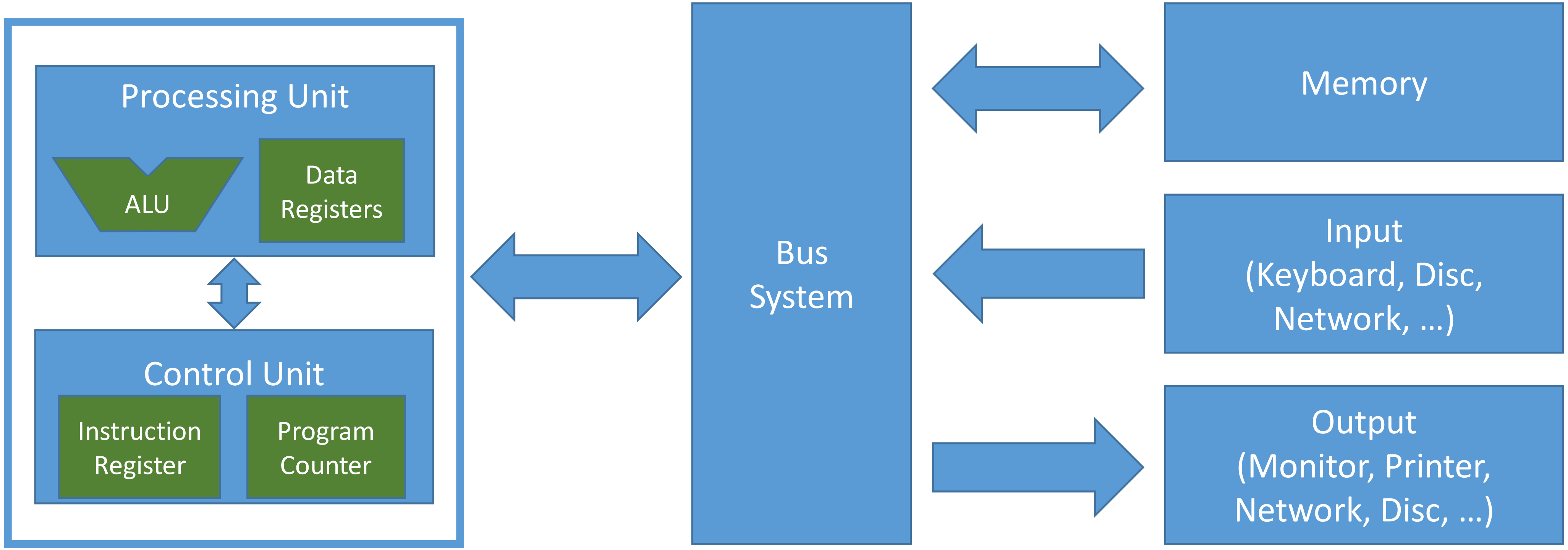
- Given a fixed program (e.g. written in C), which instruction set leads
  - to the smallest code size (the smallest number of instructions need to express the program)?
  - to best performance on a processor implementing the ISA?
  - lowest power consumption on a processor implementing the ISA?
  -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

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/


Processing Unit
ALU
Data Registers
Control Unit
Instruction Register
Program Counter
Bus System
Memory
Input (Keyboard, Disc, Network, …)
Output (Monitor, Printer, Network, Disc, …)

First RISC-V Basics

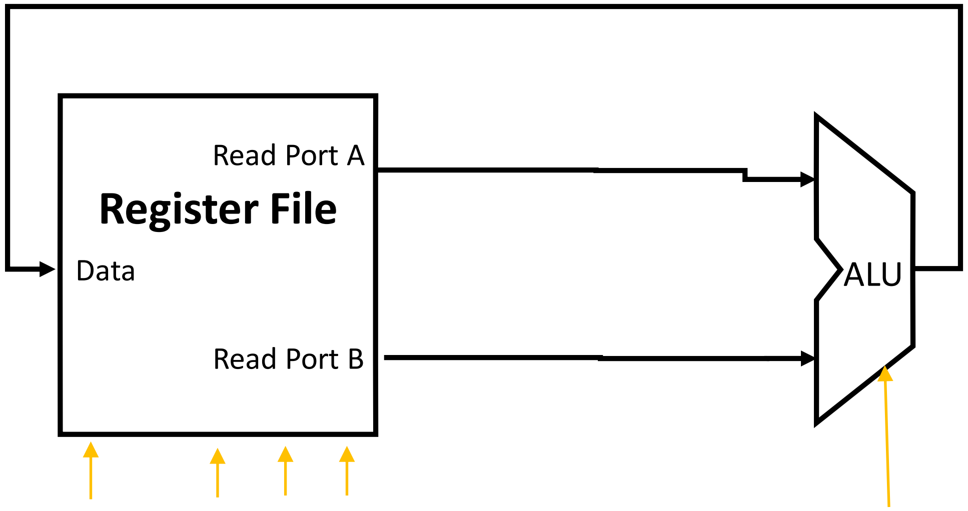
RISC-V Instruction Sets

- **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

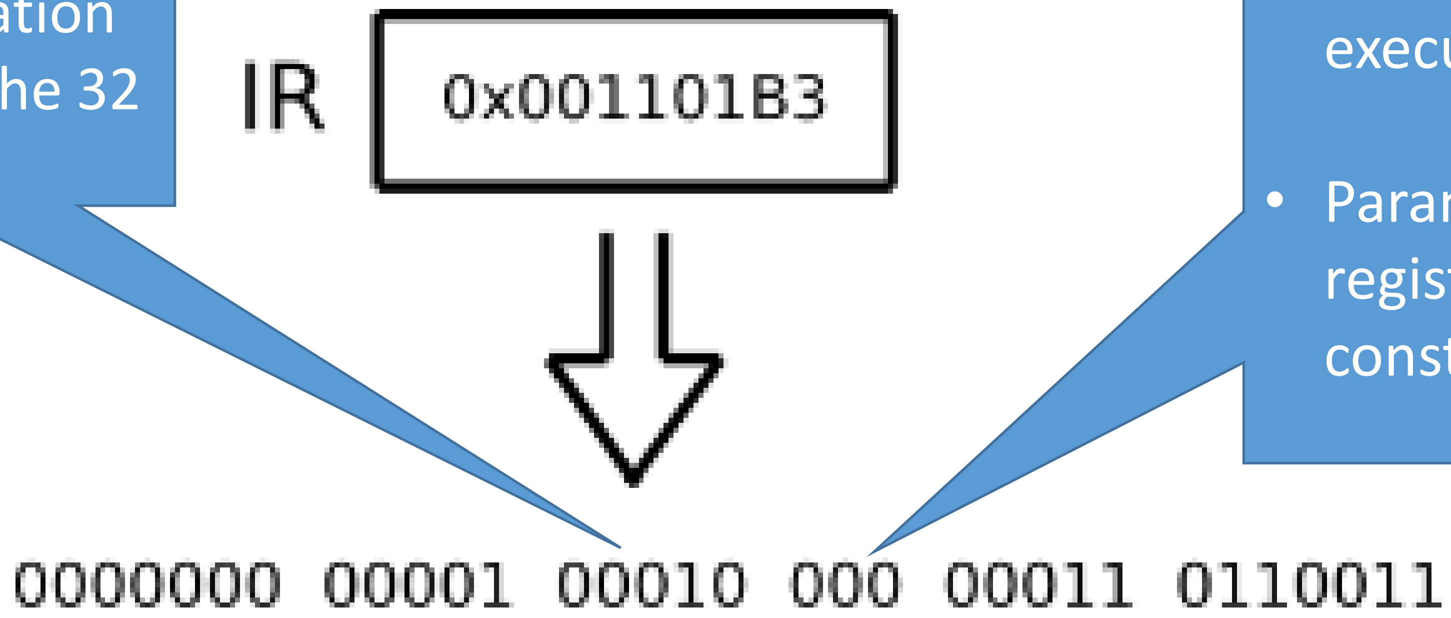
- We focus on RV32I
- The ALU and the register file are all 32 bit
- Our register file consists of 32 registers (Note: register x0 always reads zero; writing to x0 does not lead to storing a value)



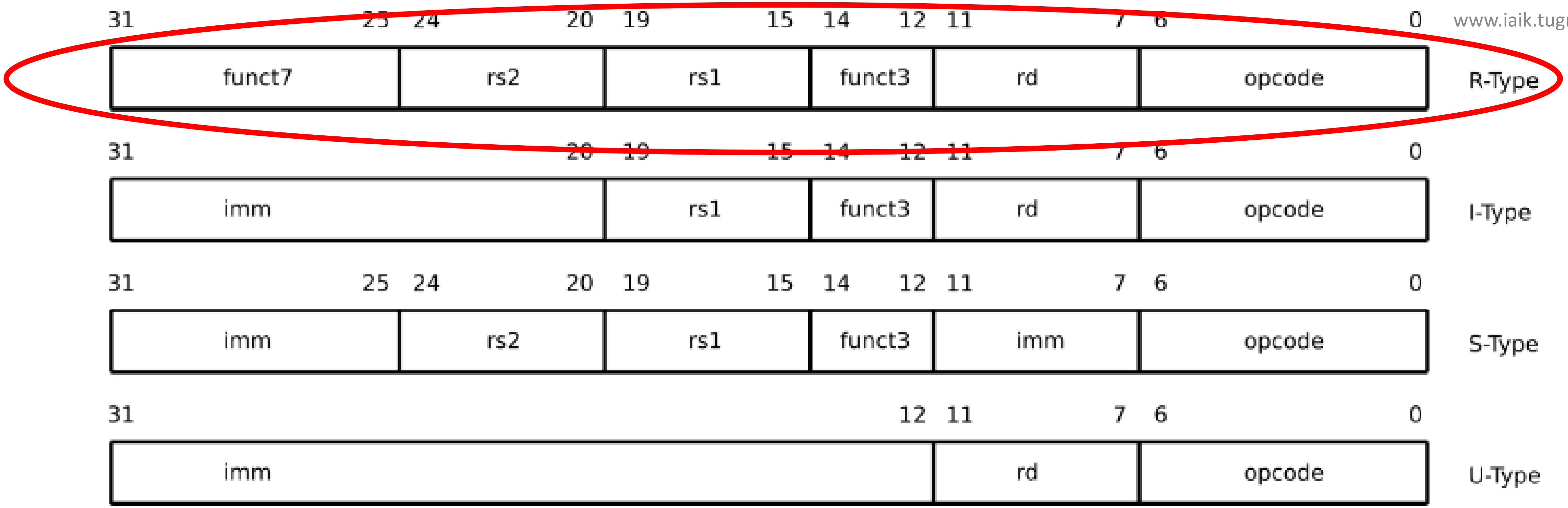
Basics

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

What information do include in the 32 bit?



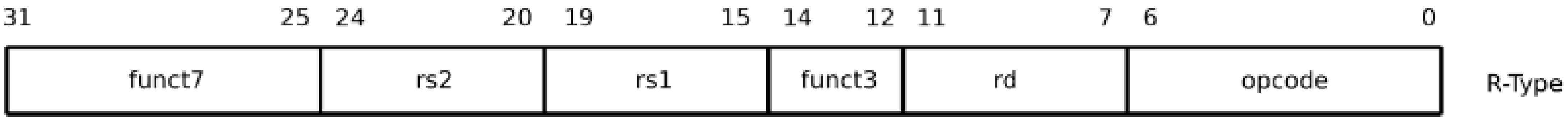
- The operation to be executed
- Parameters (e.g. source registers, target register, constants)



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

R-Type Instructions

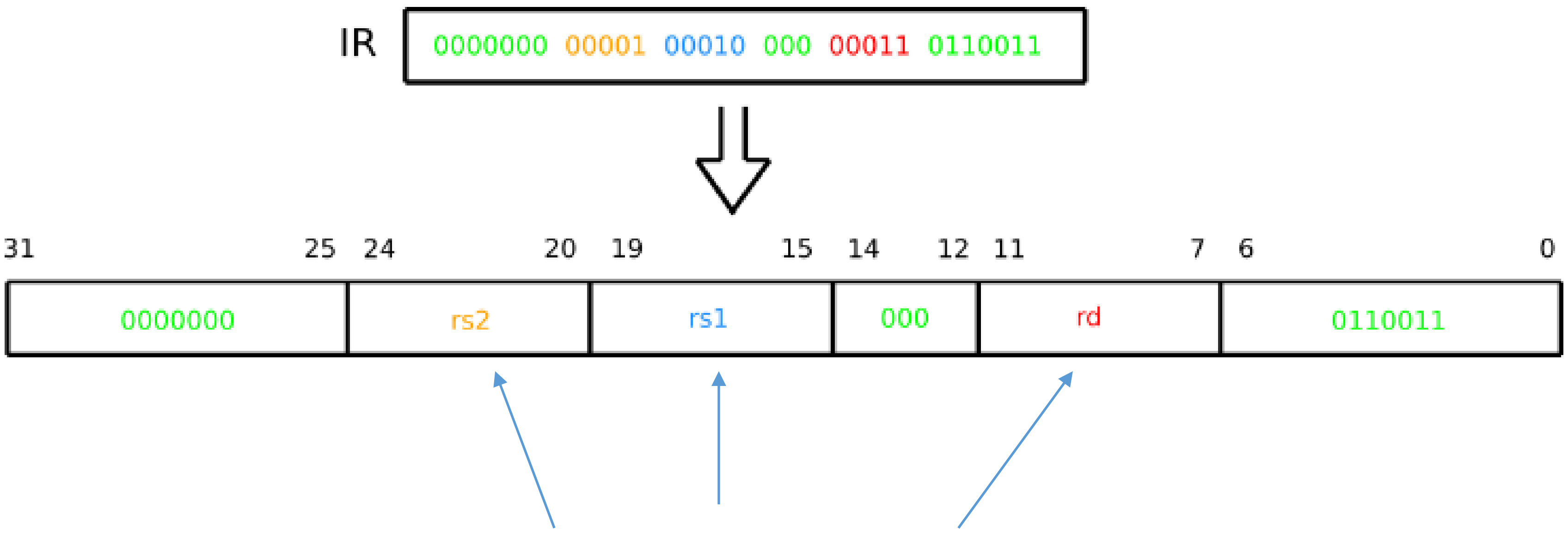
- These are instructions that perform arithmetic and logic operations based on two input registers



- 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



5 bit values to do the indexing in our register file

Example

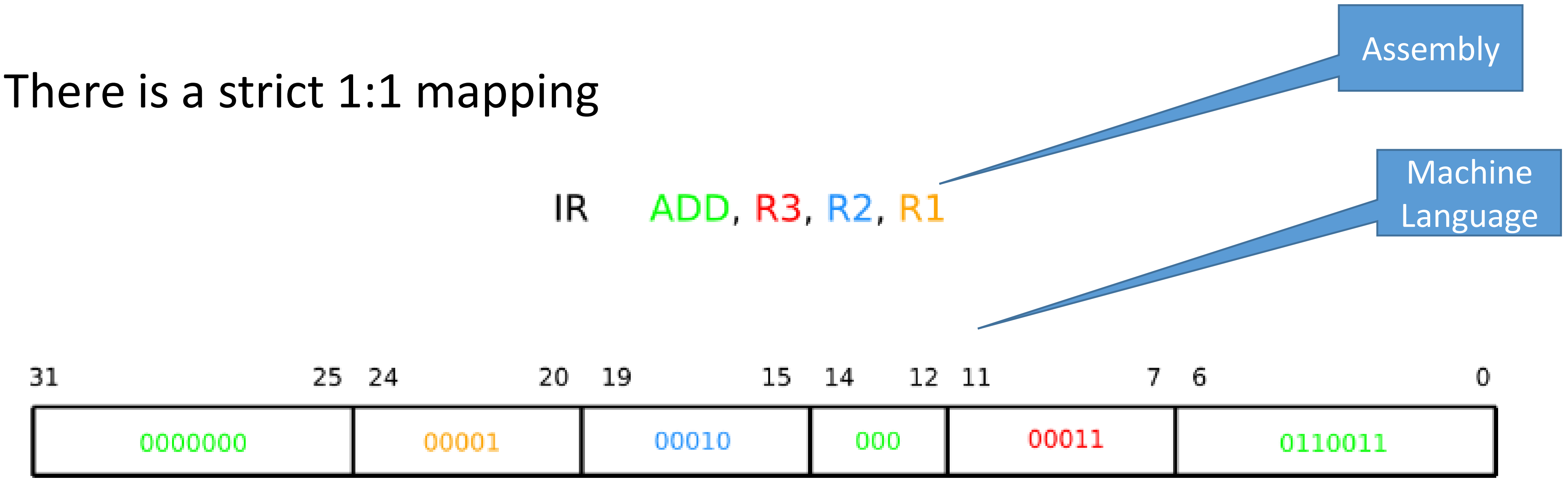
IR ADD, R3, R2, R1

31 25	24 20	19 15	14 12	11 7	6 0
0000000	00001	00010	000	00011	0110011

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode

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



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

The RV32I Instruction Set

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

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

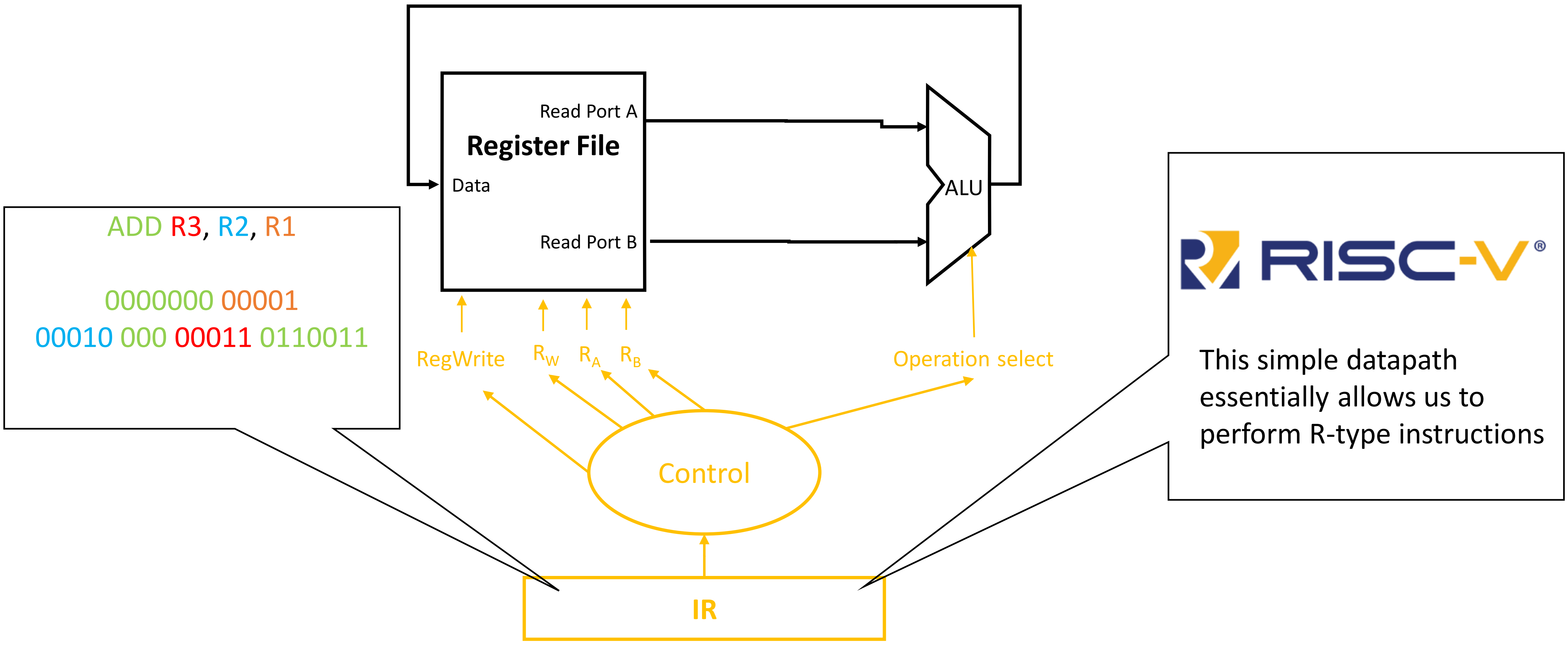
- **Logic Functions**
  - AND
  - OR
  - XOR

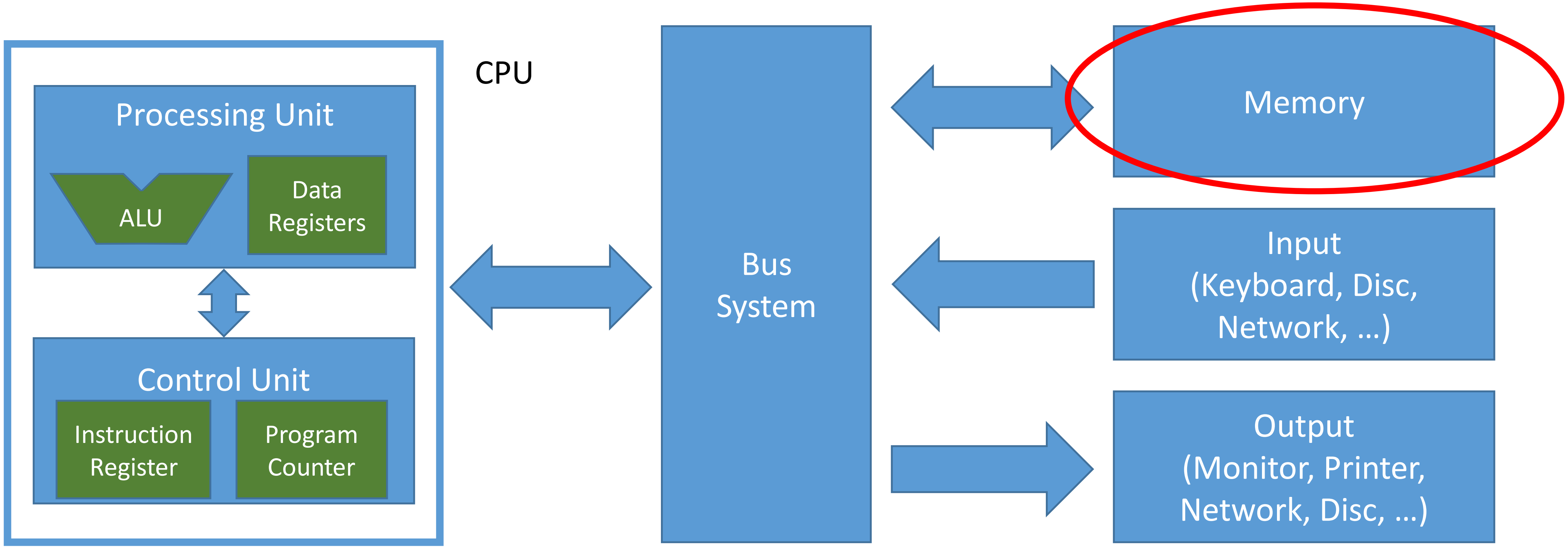
- **Arithmetic**
  - ADD (Addition)
  - SUB (Subtraction)

- **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

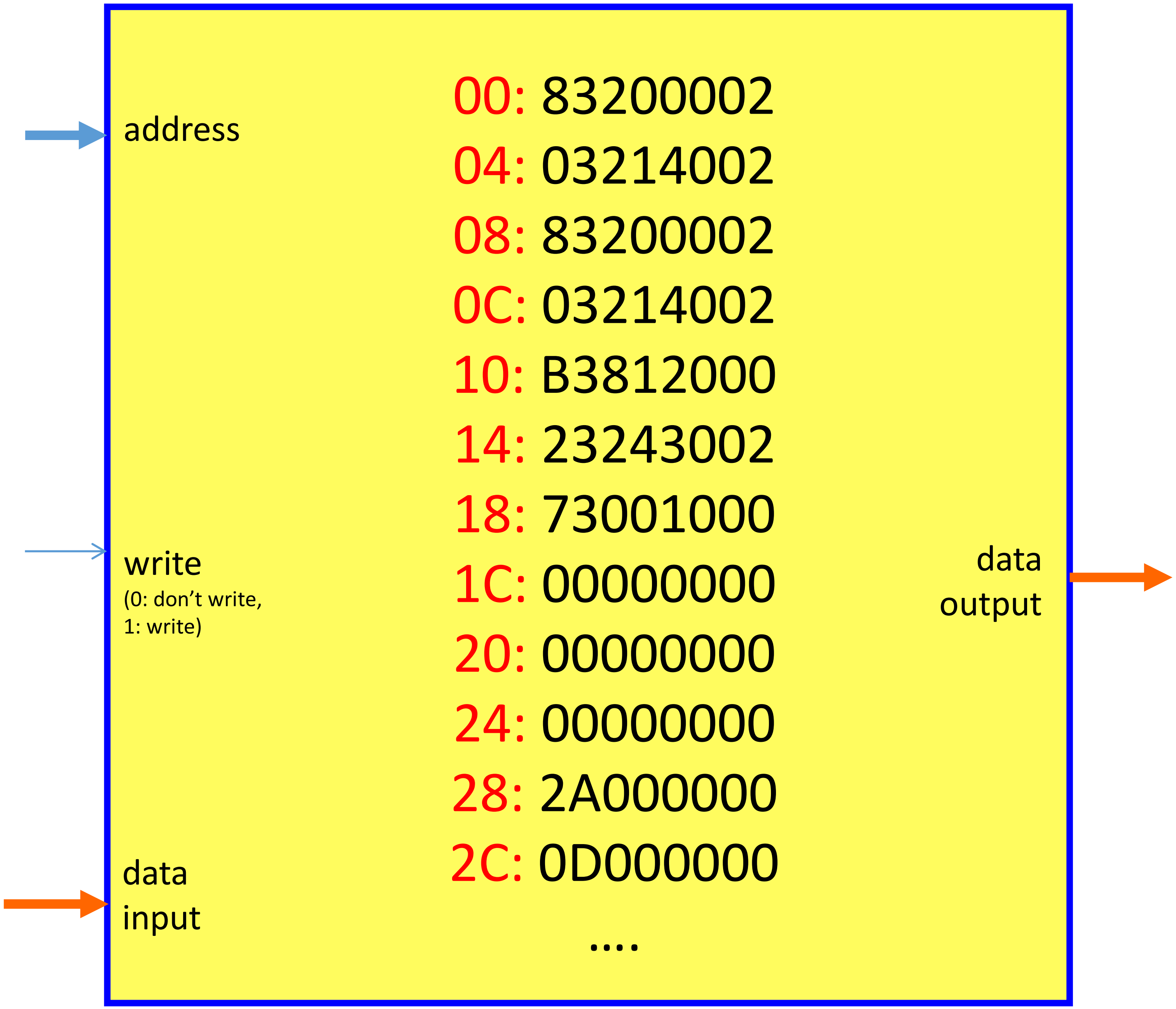




Let’s learn about memories!

Memory

Memory

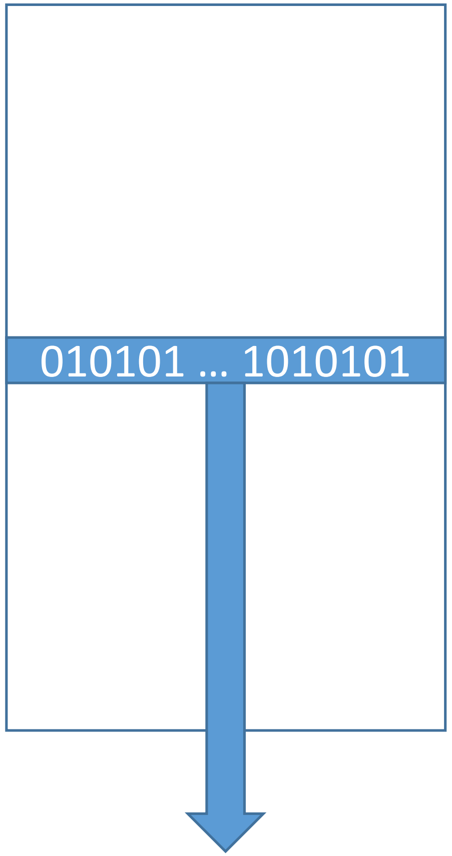

address
write
(0: don't write,
1: write)
data
input
00: 83200002
04: 03214002
08: 83200002
0C: 03214002
10: B3812000
14: 23243002
18: 73001000
1C: 00000000
20: 00000000
24: 00000000
28: 2A000000
2C: 0D000000
....
data
output

Main memory is a “RAM”

Random **A**ccess **M**emory

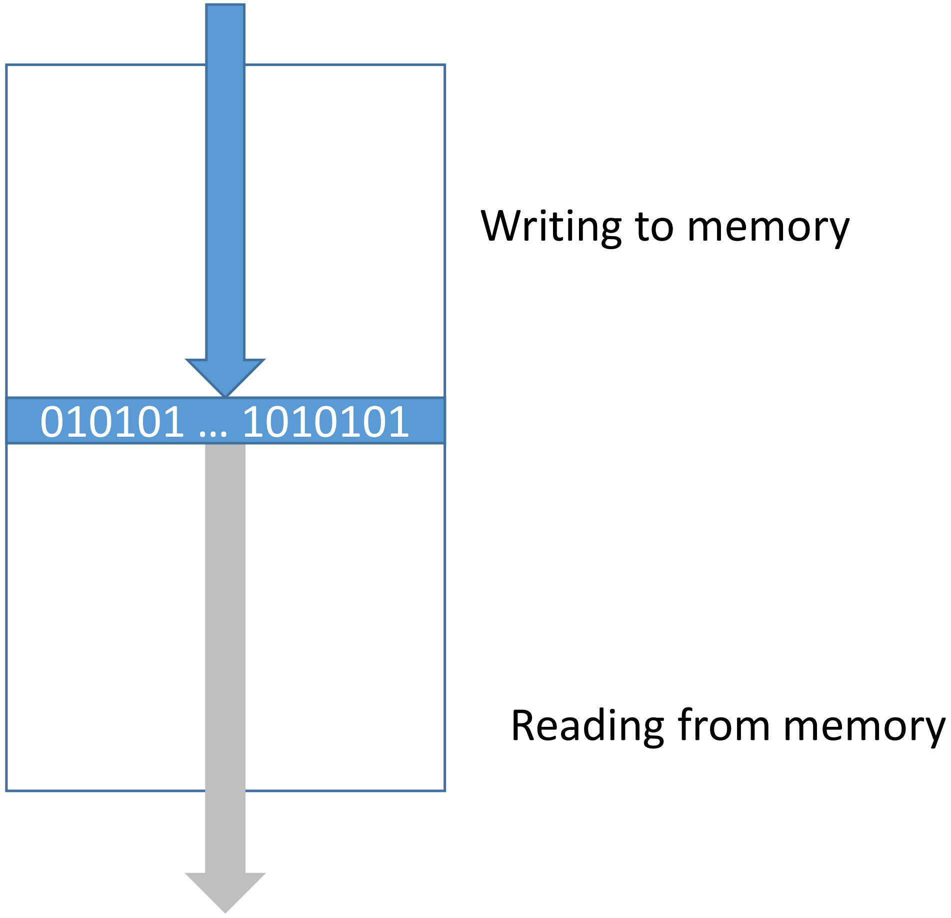
(“Memory where arbitrary read and write accesses can be performed”)

Reading from memory

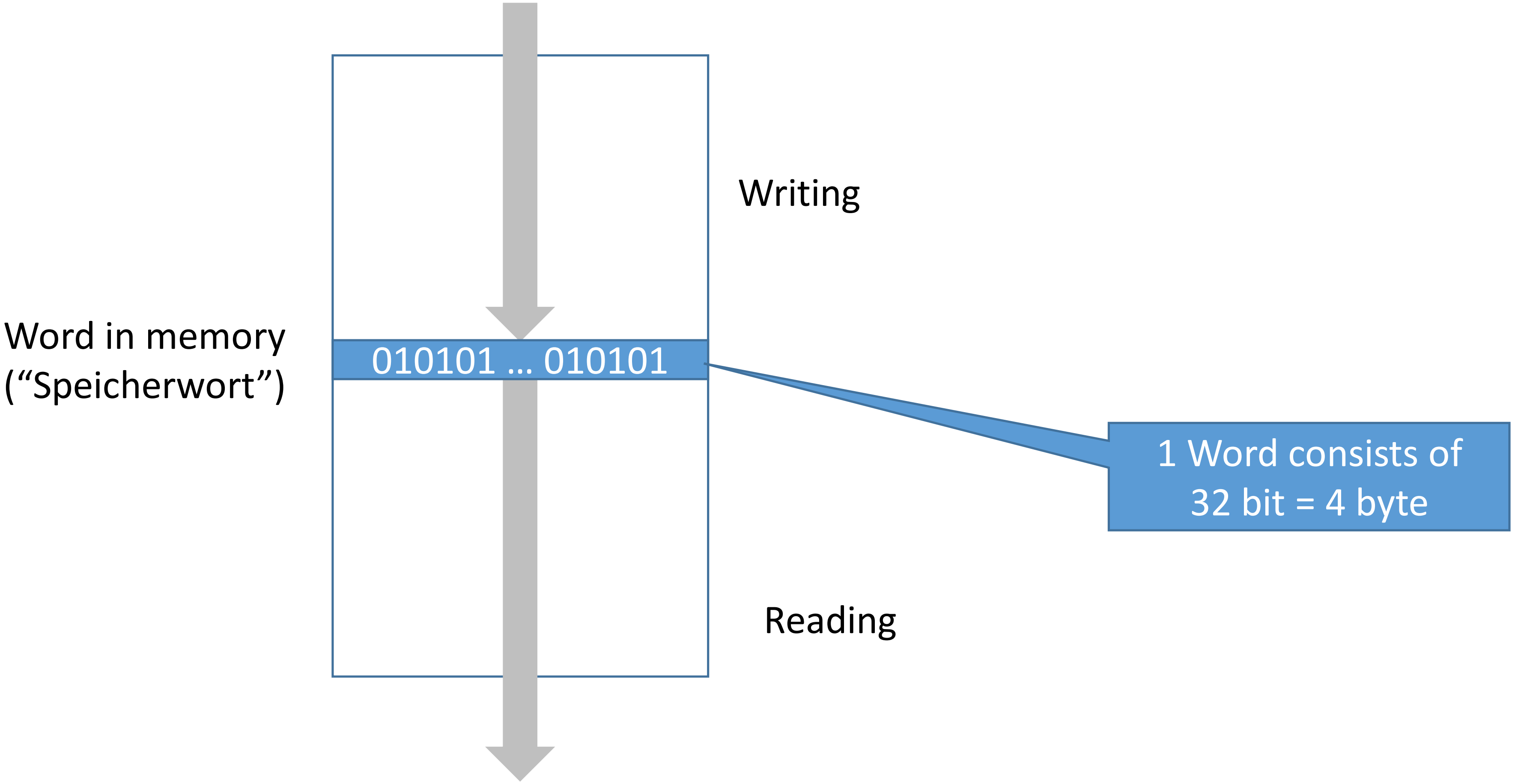


Reading from memory

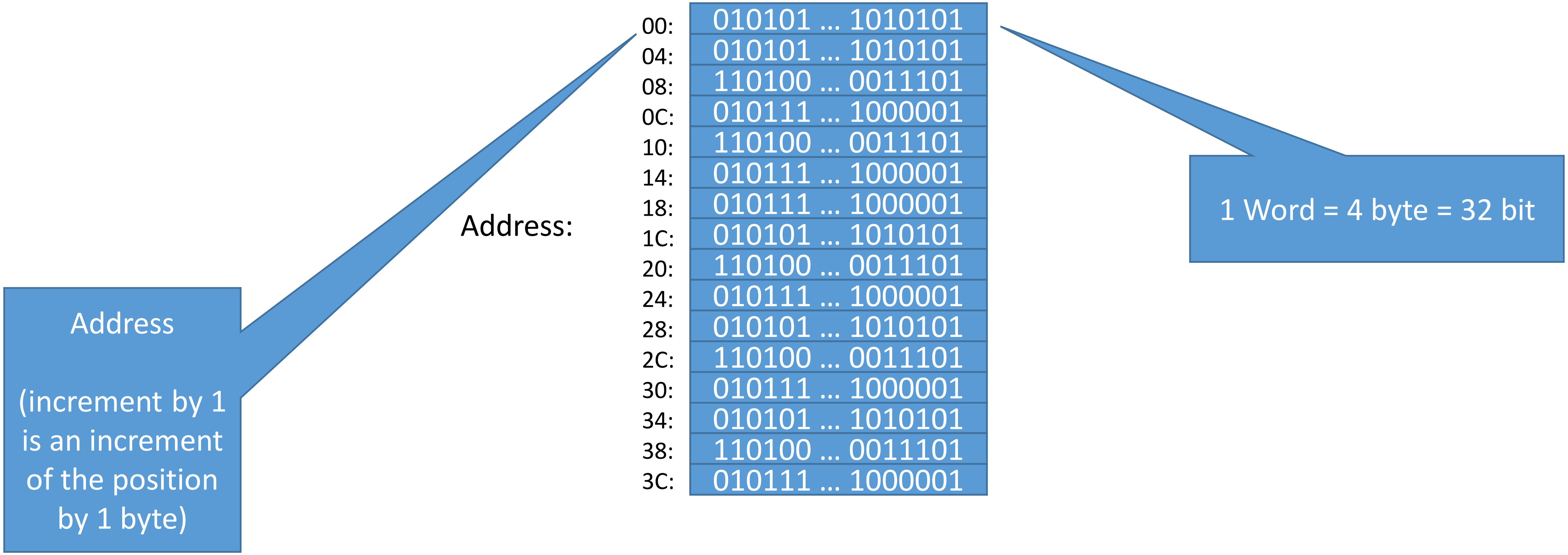
Writing to Memory



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



Each Byte in Memory Has an Address



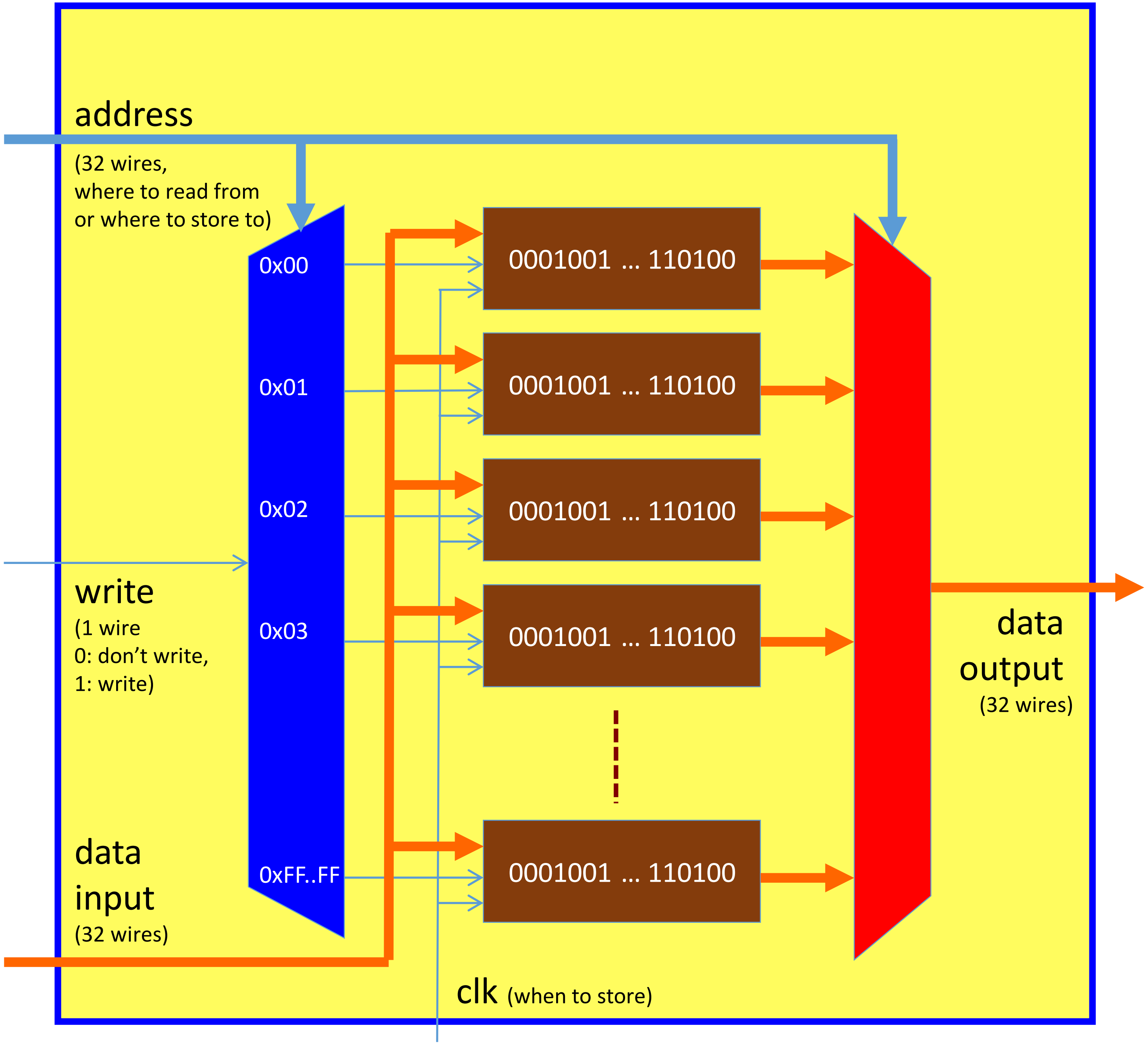
The Indices of the Bits Within a Word in Memory

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


address
(32 wires,
where to read from
or where to store to)
0x00
0x01
0x02
0x03
0xFF..FF
0001001 … 110100
0001001 … 110100
0001001 … 110100
0001001 … 110100
0001001 … 110100
write
(1 wire
0: don't write,
1: write)
data
input
(32 wires)
data
output
(32 wires)
clk (when to store)

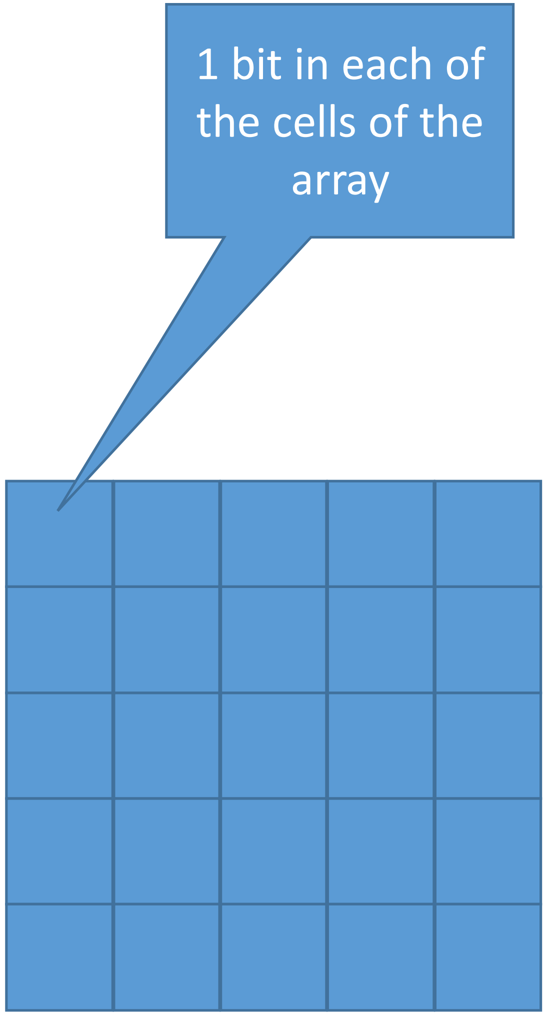
Building Memories in Practice

- 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

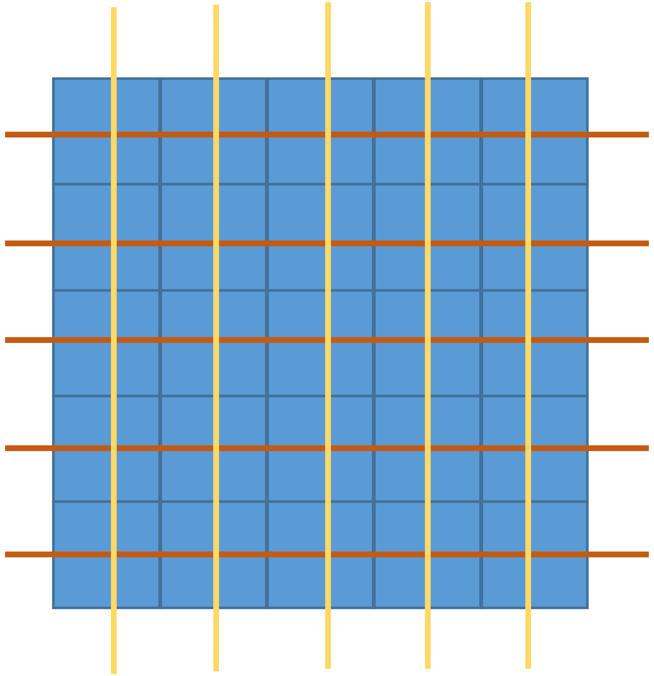
Basic Idea of Memory Design

- 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.

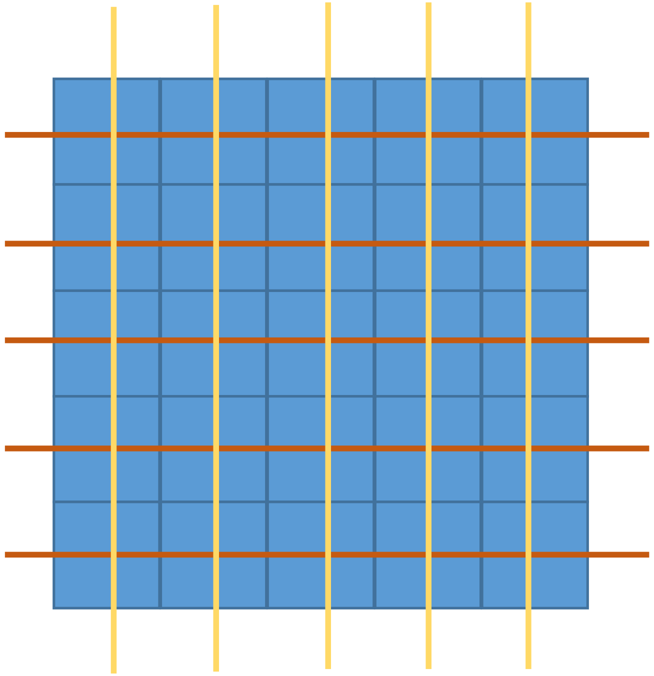


Basic Idea of Memory Design

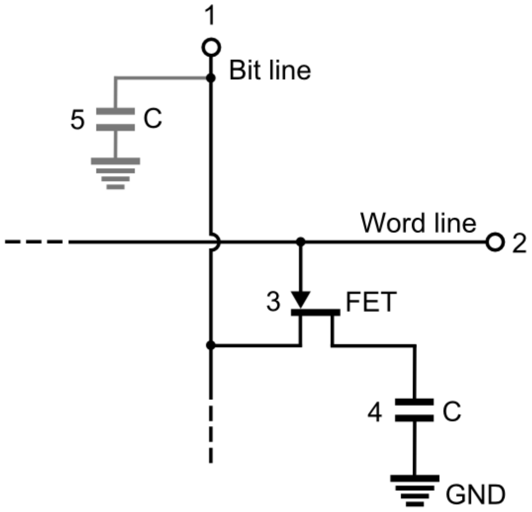
- A bitline connects all memory cells vertically (yellow)
- A wordline connects all memory cells horizontally
- This basic structure is used for all kinds of memories:
  - Non-volatile memory (NVM)
  - Static memory (SRAM)
  - Dynamic memory (DRAM)
  - DDR memory
- Each memory type is for different trade-offs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



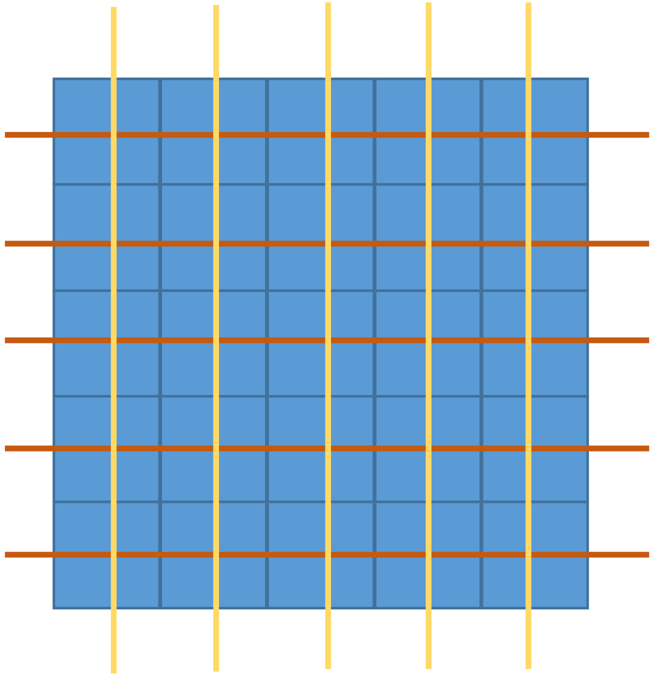
Basic Idea of Read/Write for DRAM



- 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

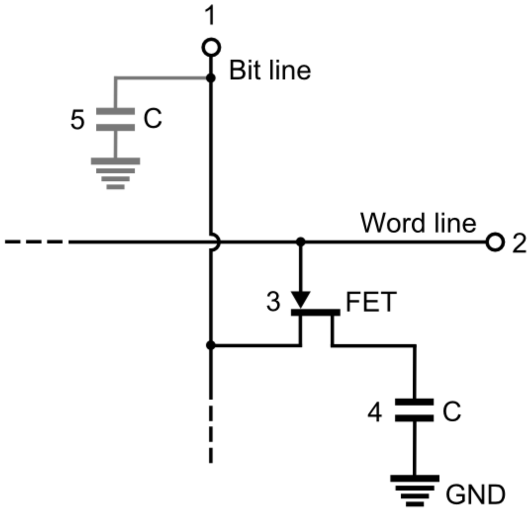


- Writing a cell:
  - Set corresponding bitline to the desired storage value
  - Set corresponding wordline to high

  →This charges the capacitance of the desired cell to the desired storage value

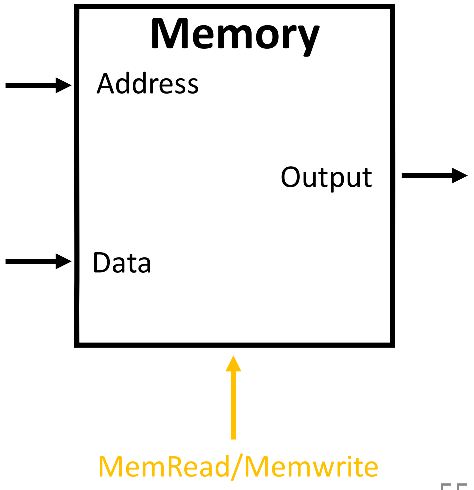
- Reading a cell:
  - Pre-charge the corresponding bitline to the desired voltage value
  - Disconnect the bitline
  - Set the corresponding wordline to high

  → 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 one of the most highly optimized components of a computer system
- 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



Sign Extension

- Memory operations in RISC–V require to combine signed values of different bit sizes when computing addresses
- This requires to perform “sign extension”
- Sign extension means that the MSB of the shorter value is replicated until the bit size of the larger value is reached. This ensures correct arithmetic handling.

Sign Extension – Example

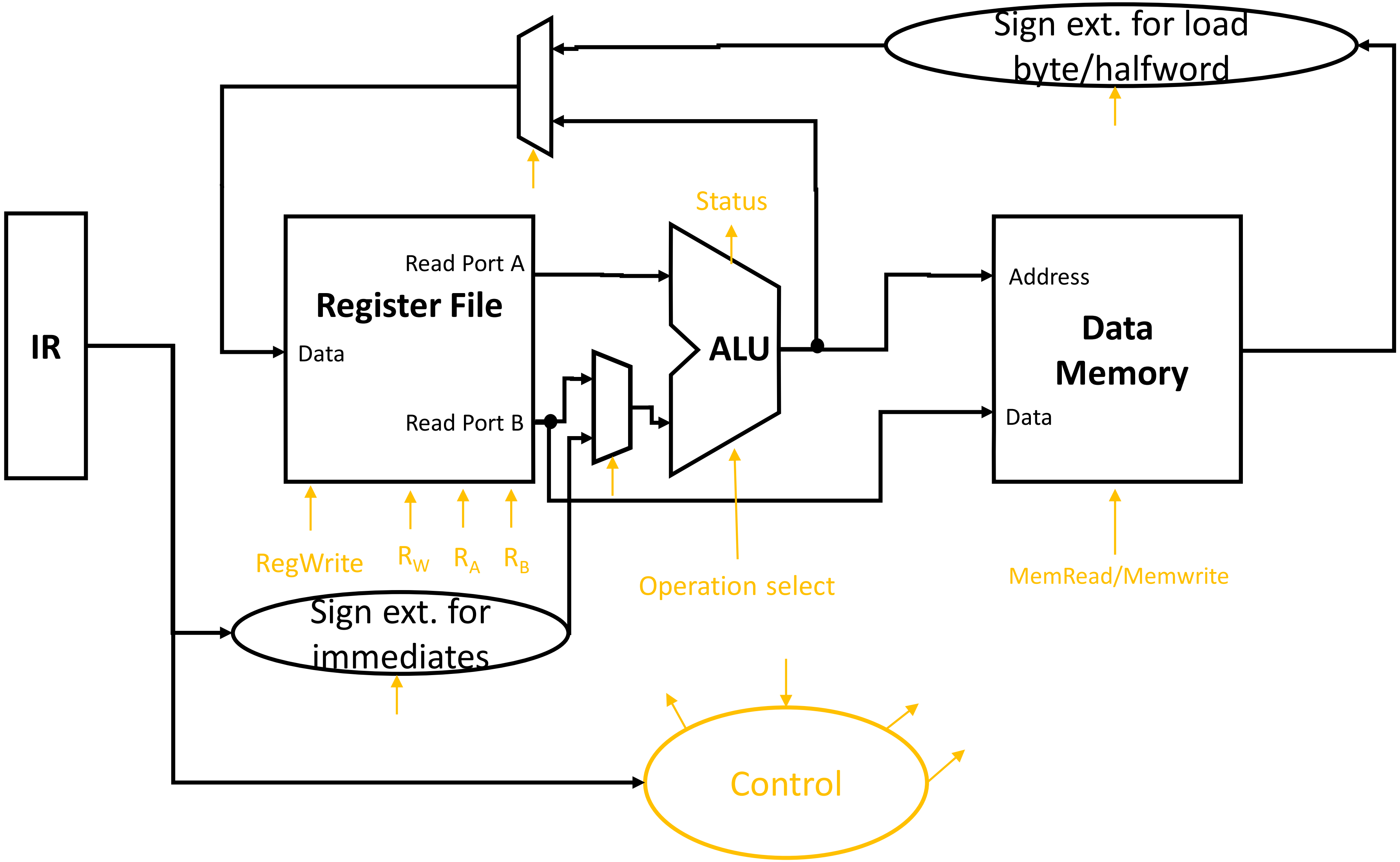
- Example: compute A + B
  - Value A (16 bit): 7 (Binary: 00000000 00000111)
  - Value B (8 bit): -1 (Binary: 11111111)

  - If we would simply add the values without sign extension, this would lead to an incorrect result: 262 decimal (Binary: 00000001 00000110)

- Correct computation with sign extension:
  - Value A (16 bit): 7 (Binary: 00000000 00000111)
  - Value B after sign extension (16 bit): -1 (Binary: 11111111 11111111)

  - Result A + B : 6 (Binary: 00000000 00000110)

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

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

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

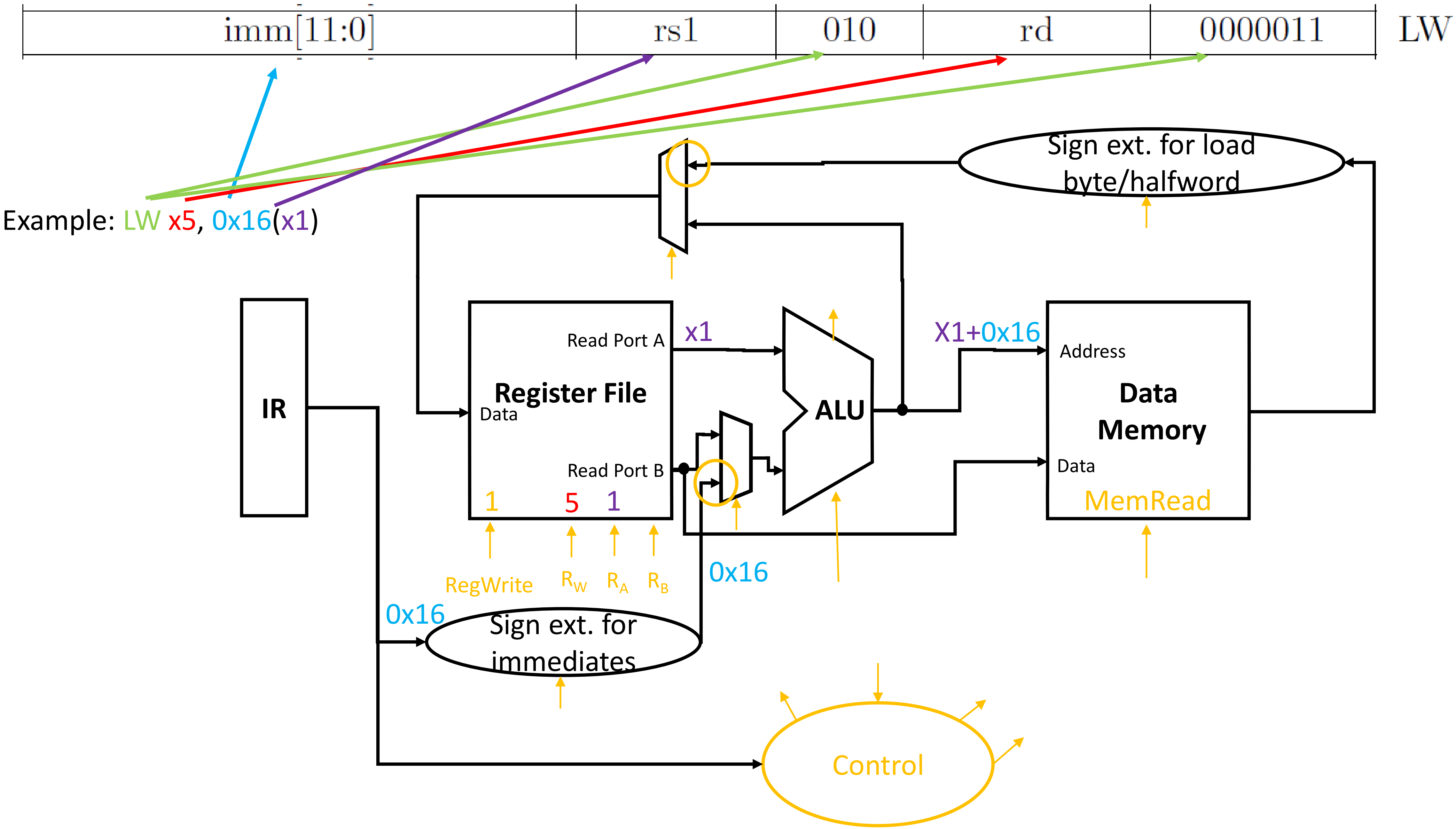
- Assembly:
  - LW rd, offset(rs1)

- Machine language

imm[11:0]	rs1	010	rd	0000011	LW

  - 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 fixed address by setting rs1 to x0 (LW rd, addr(x0))
    - load data from a pointer providing a relative offset (LW rd, offset(rs1))


imm[11:0]
rs1
010
rd
0000011
LW
Example: LW x5, 0x16(x1)
Sign ext. for load byte/halfword
IR
Read Port A
Register File
Data
Read Port B
1
5
1
x1
ALU
X1+0x16
Address
Data Memory
Data
MemRead
RegWrite
RW
RA
RB
0x16
0x16
Sign ext. for immediates
Control

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

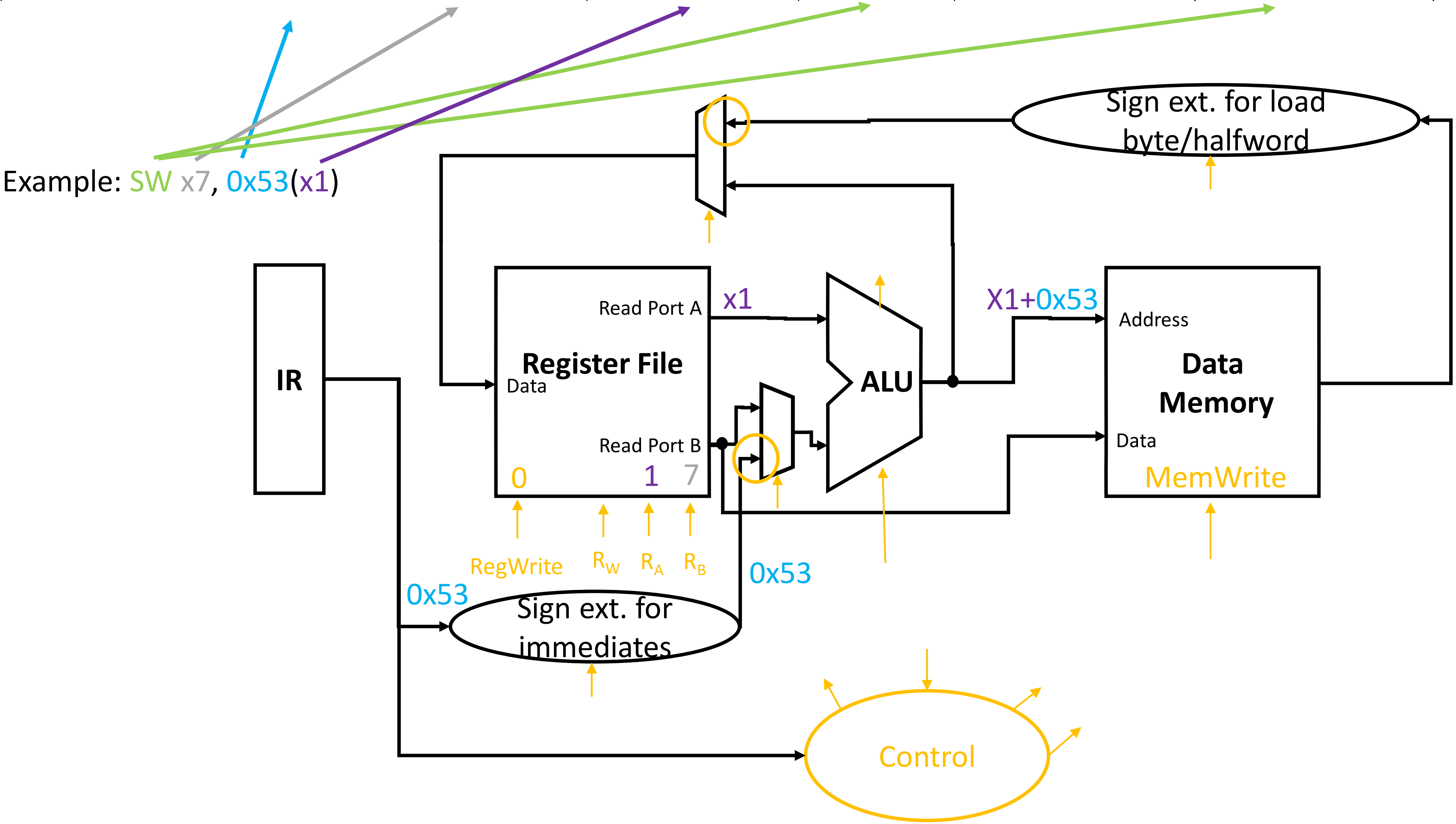
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

  - 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 an absolute address (SW rs2, addr(x0))
    - store data to pointer + offset (SW rs2, offset(rs1))

imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

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.

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

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

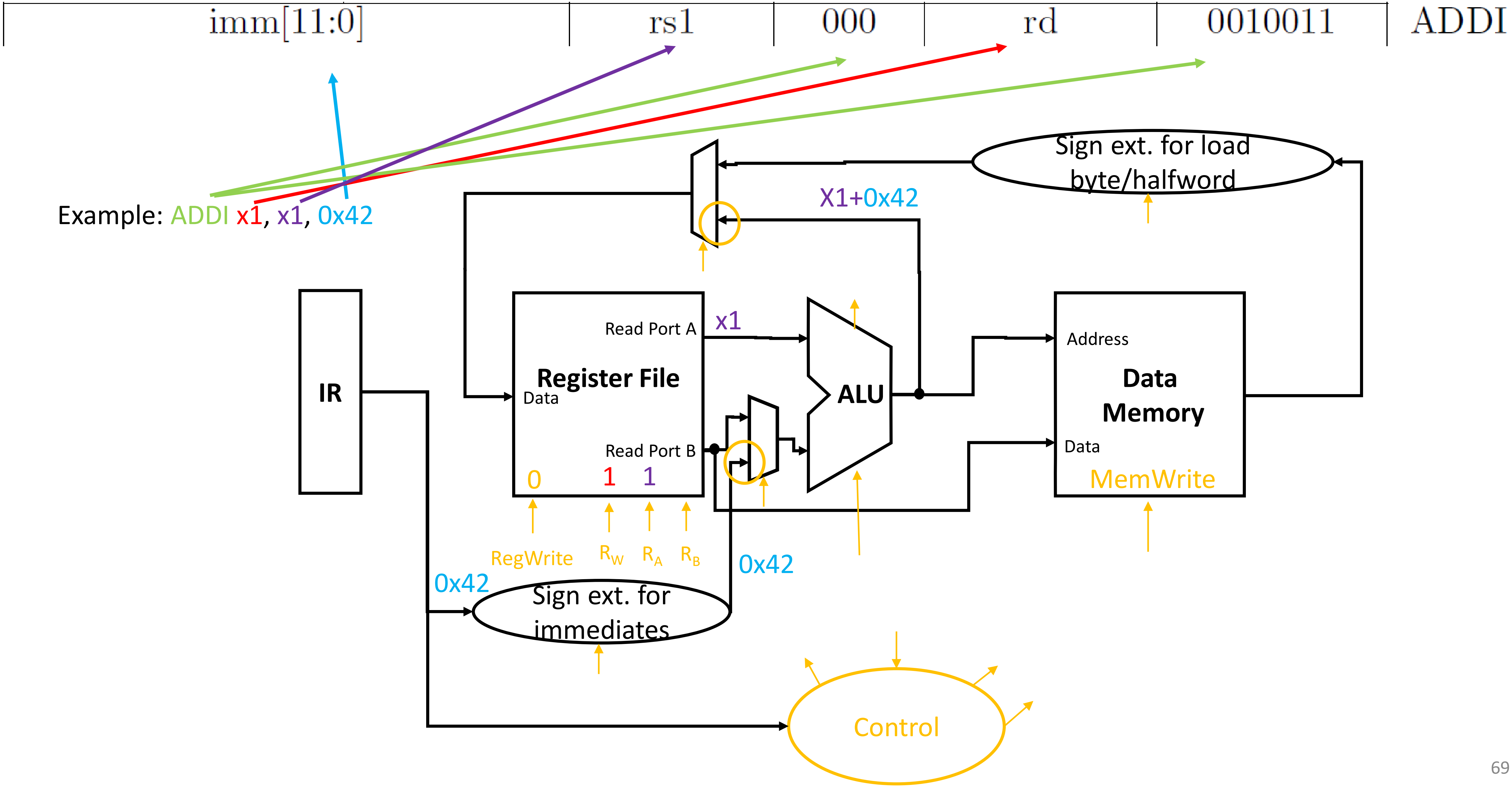
- Assembly:
  - ADDI rd, rs1, immediate

- Machine language

imm[11:0]	rs1	000	rd	0010011	ADDI

  - Computes rd = rs1 + imm

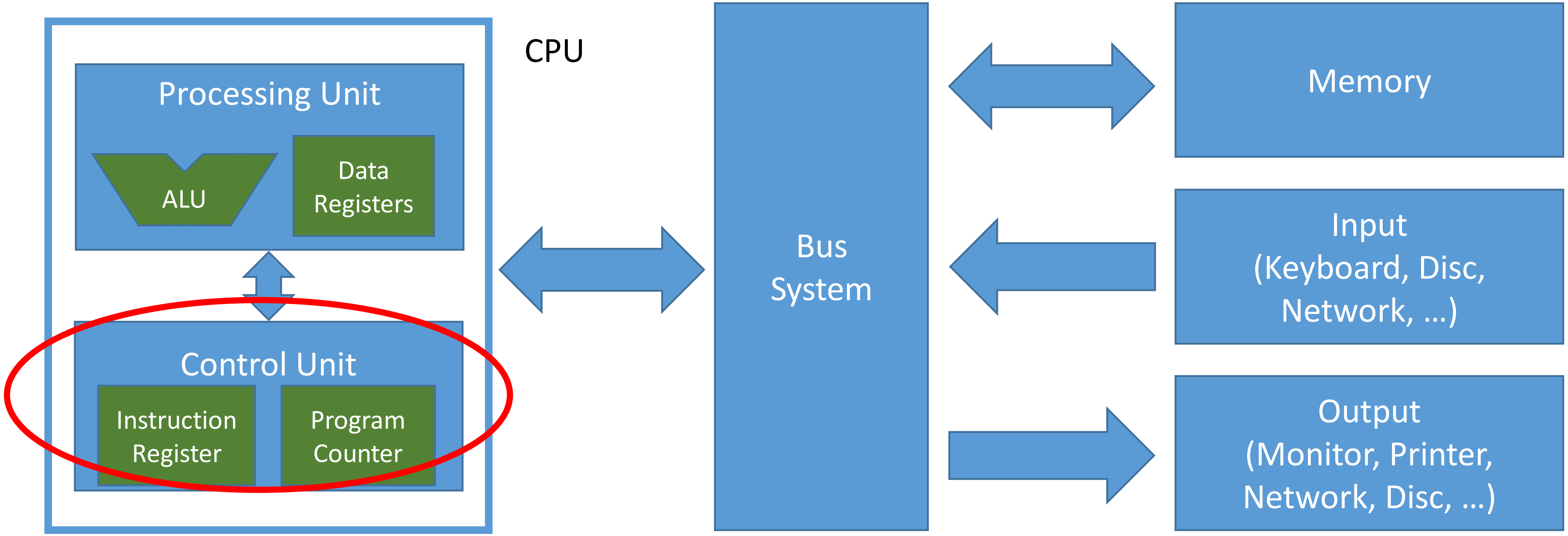
- Functionality:
  - Computes rd = rs1 + imm
  - Example applications
    - Move content of one register to another register by setting immediate to 0 (ADDI rd,rs1,0)
    - Set a register to a constant value by using x0 as source: (ADDI rd, x0, immediate)
    - Increment/decrement a register by setting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


imm[11:0]
rs1
000
rd
0010011
ADDI
Example: ADDI x1, x1, 0x42
Sign ext. for load byte/halfword
X1+0x42
IR
Read Port A
x1
Register File
Data
Read Port B
0
1
1
RegWrite
RW
RA
RB
ALU
Address
Data Memory
Data
MemWrite
0x42
0x42
Sign ext. for immediates
Control

More Operations with Immediates

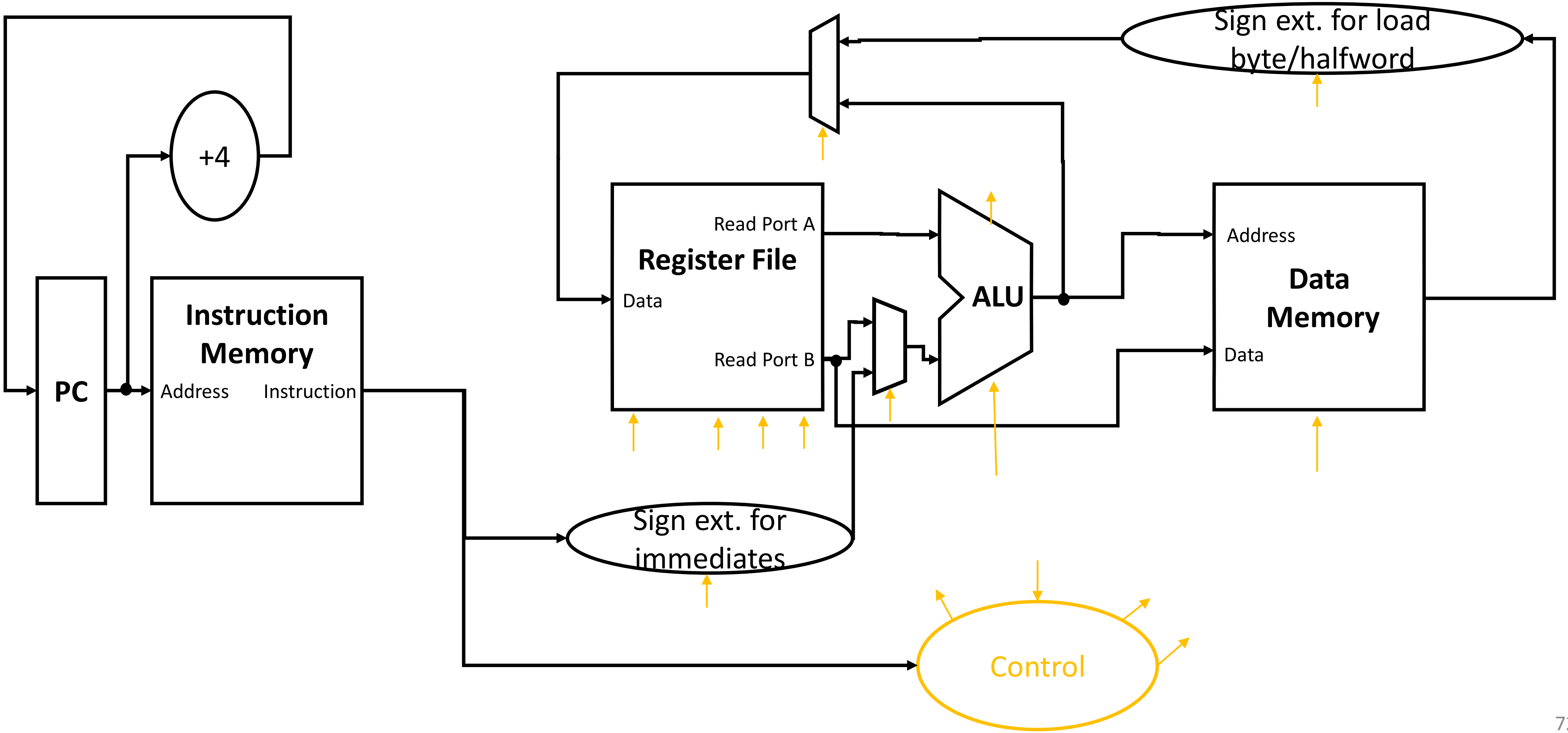
imm[31:12]				rd	0110111	LUI
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	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 load a full 32 bit 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



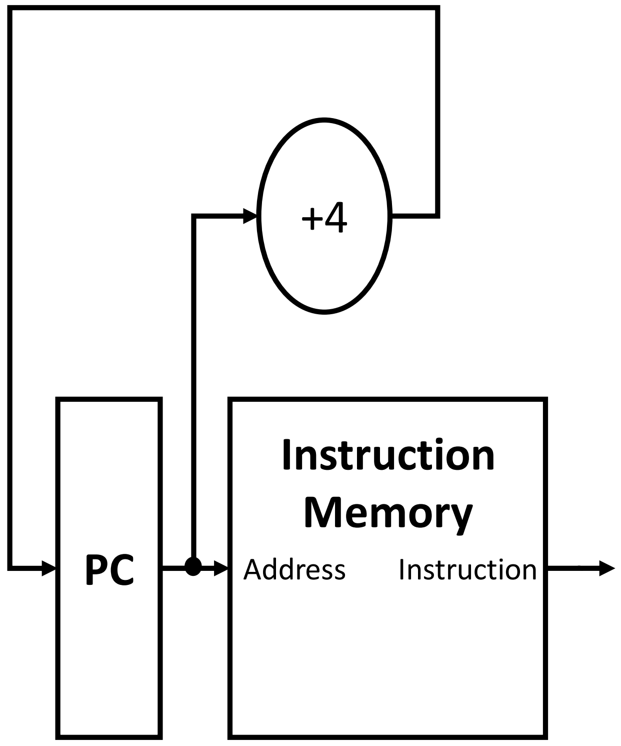
Let’s learn about control!

Adding Instruction Memory

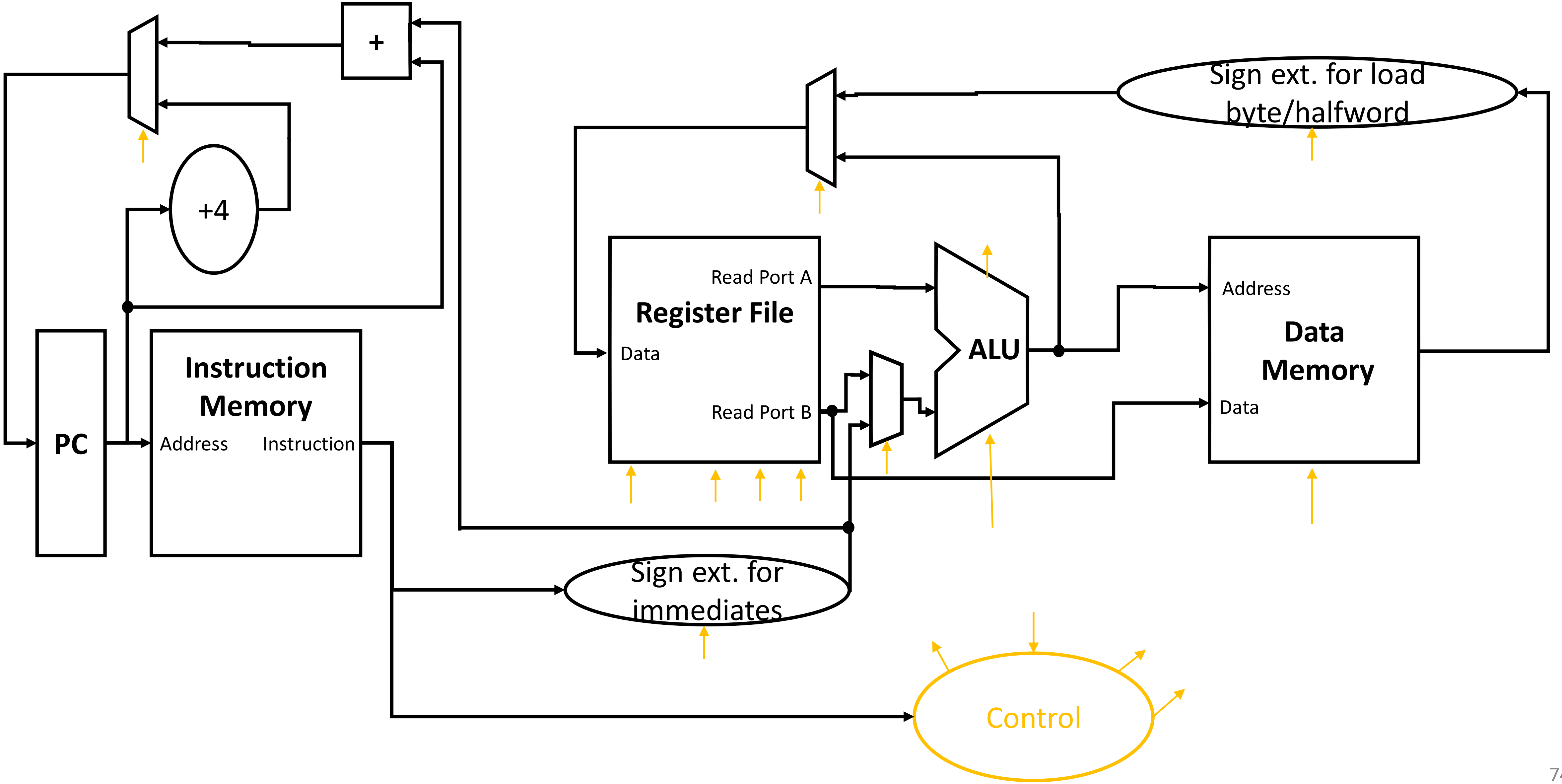


Instruction Memory

- The instruction memory stores a sequence of instruction
- 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



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

Additional operations that we can perform with our updated datapath:

Conditional Branch Operations

Example: BEQ

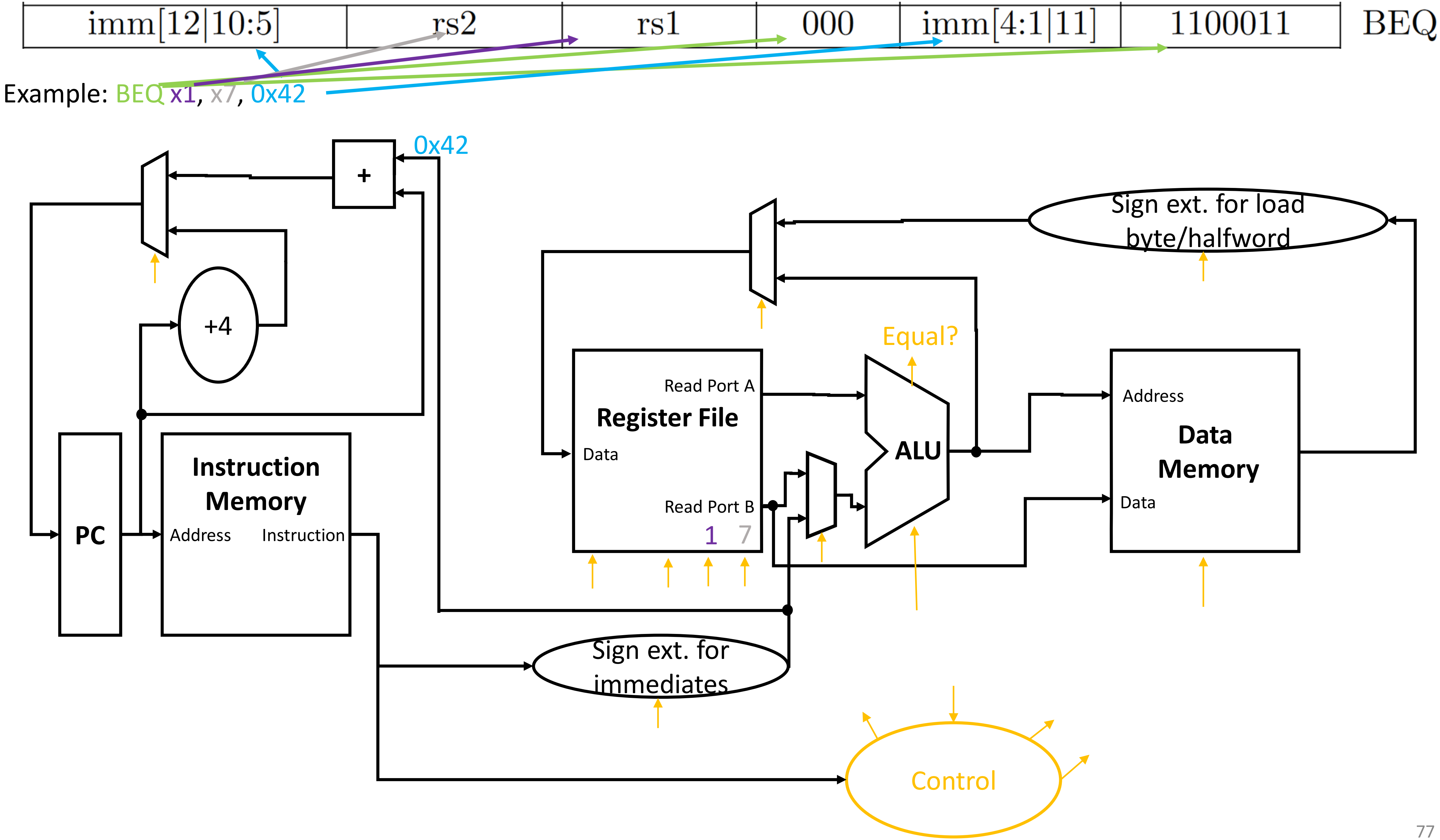
- Assembly:
  - BEQ rs1, rs2, offset

- 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



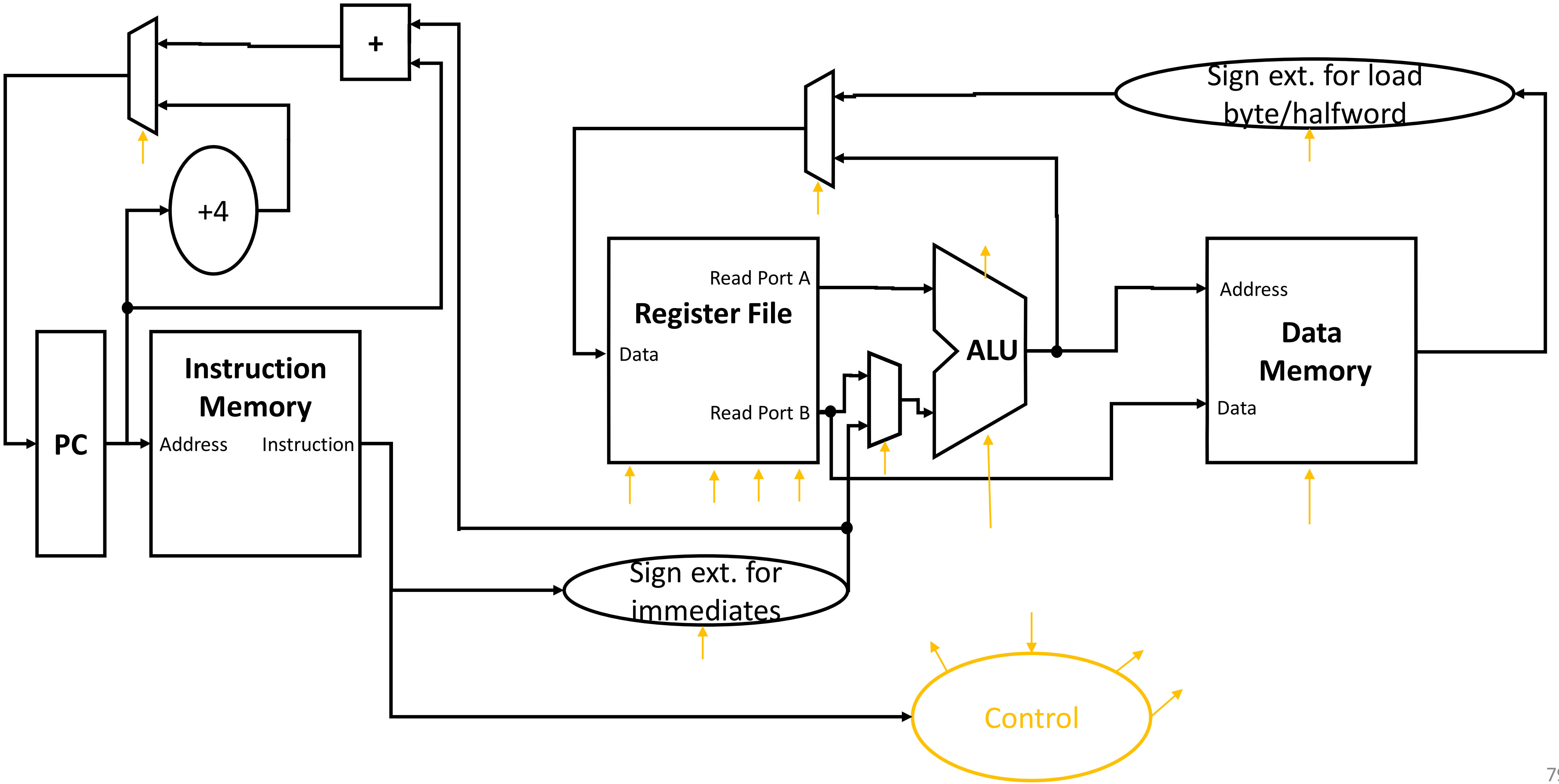
imm[12|10:5]
rs2
rs1
000
imm[4:1|11]
1100011
BEQ
Example: BEQ x1, x7, 0x42
0x42
+
+4
Sign ext. for load byte/halfword
Equal?
Read Port A
Register File
Data
Read Port B
1
7
ALU
Address
Data Memory
Data
PC
Instruction Memory
Address
Instruction
Sign ext. for immediates
Control

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)



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

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)

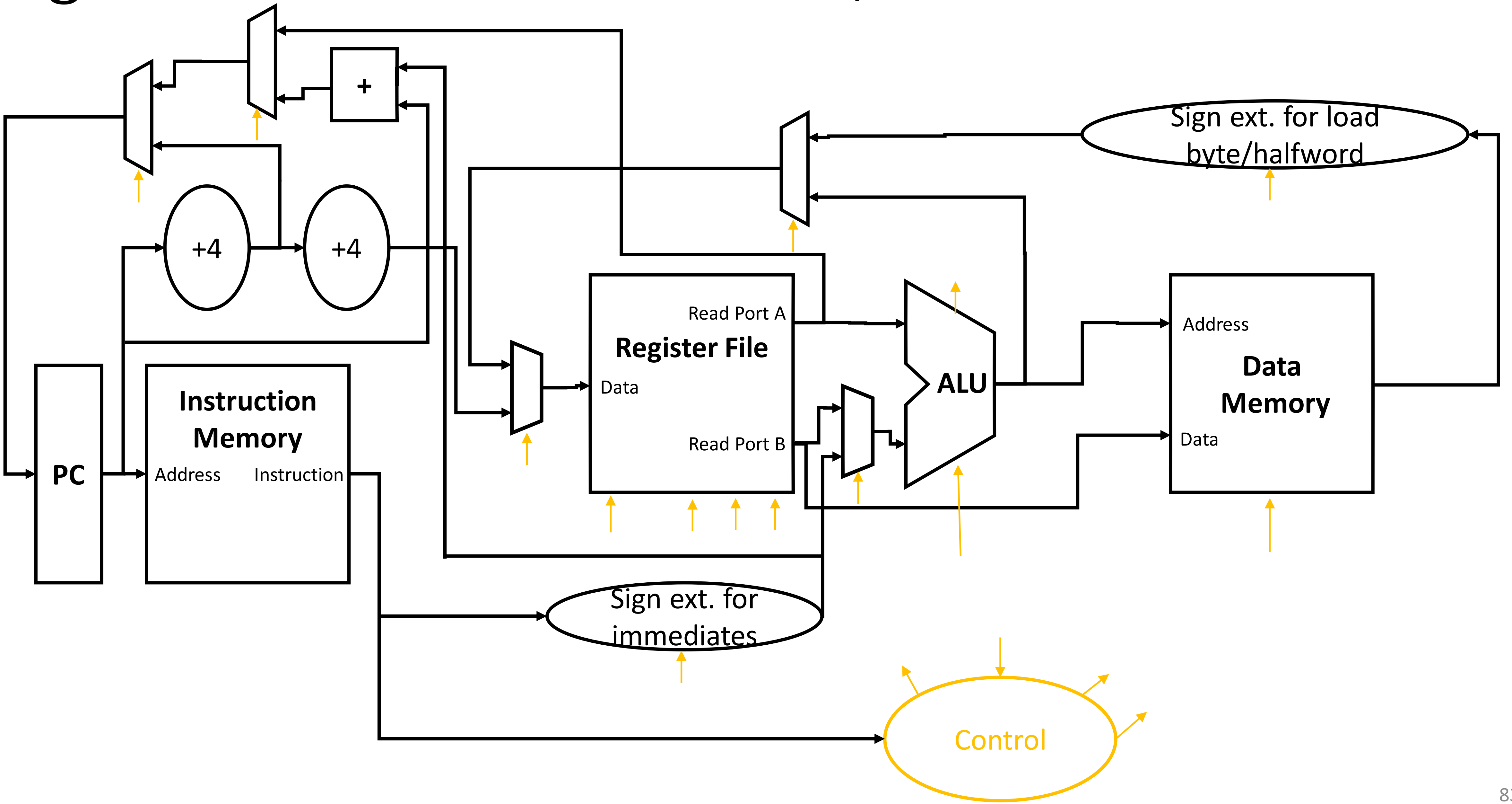
JAL/JALR

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 (will be discussed later)

High-Level Overview incl. JAL/JALR



Performance

- The goal of processor design is maximize the executed number of instructions per time

- This is determined by two factors
  - The needed clock cycles per instruction (CPI)
  - The clock frequency, which determines the number of cycles per second

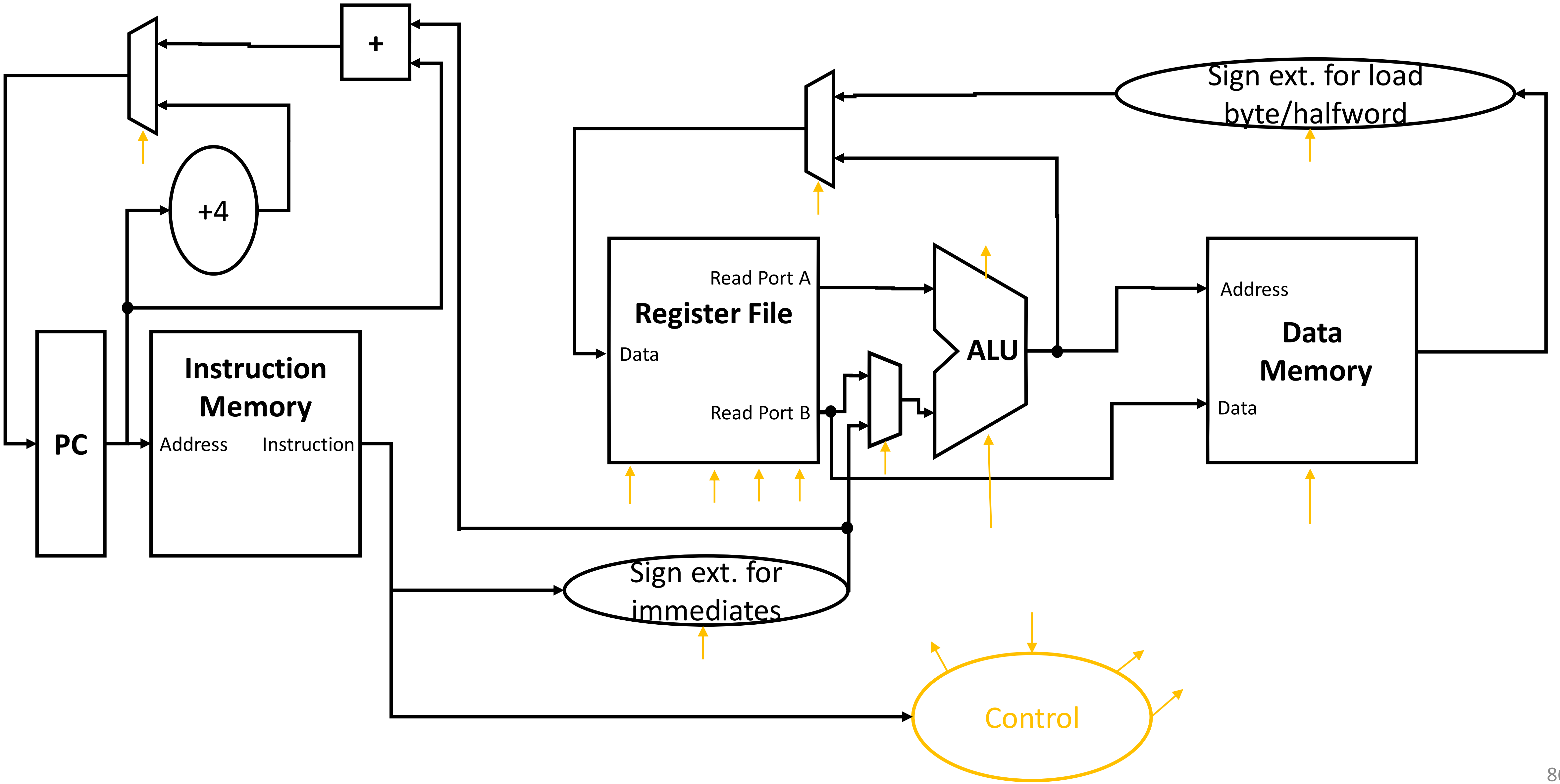
- The execution time for a program with N instructions is N * CPI * (1/f)
  - f is the clock frequency (1/f is the clock period)
  - CPI is the average number of cycles per instruction

Performance of the Single-Cycle Design

- Each instruction takes exactly one cycle to execute

- The maximum clock frequency is defined by the slowest instruction of the design
    - Remember: the critical path is the longest combinational path in the design.
    - The critical path of the slowest instruction therefore defines the clock frequency of our processor

High-Level Overview (Single Cycle Datapath)



Single Cycle Machine in Practice?

- In practice, we typically do not build single-cycle machines with separated instruction and data memory
  - Main drawbacks:
    - Low performance (the clock rate is defined by the slowest instructions)
    - We need separate instruction and data memory

  → we need to have a more fine granular view of the operations that are performed for each instructions

- First improvement: we split the actions of the CPU in three basic steps:
  - Fetch: Read an instruction from memory
  - Decode: Prepare the necessary control signals for the instruction
  - Execute: Execute the actual instruction

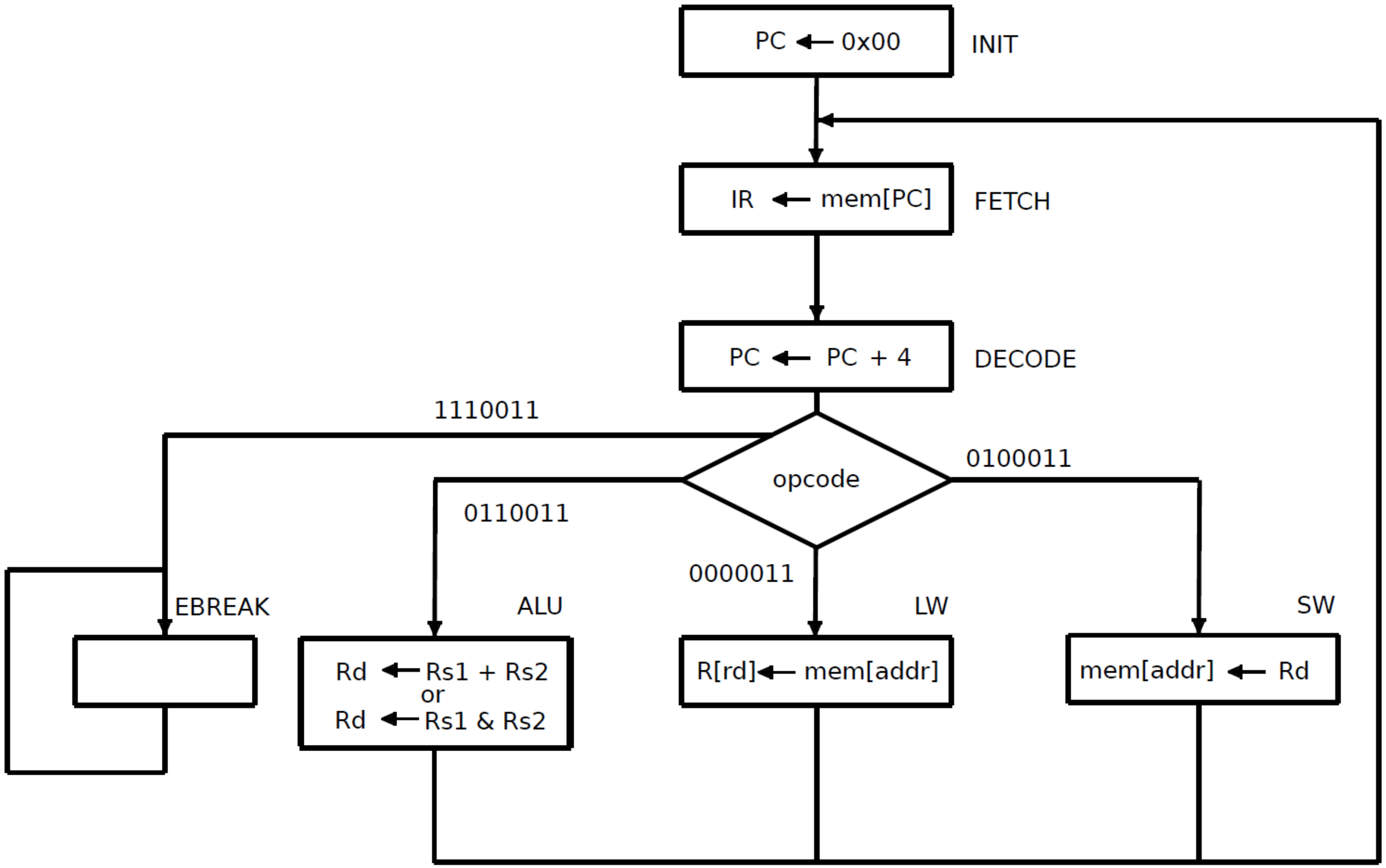
- This splitting does not immediately lead to higher performance, but it allows to have a single memory for instructions and data

Modelling with an ASM Graph

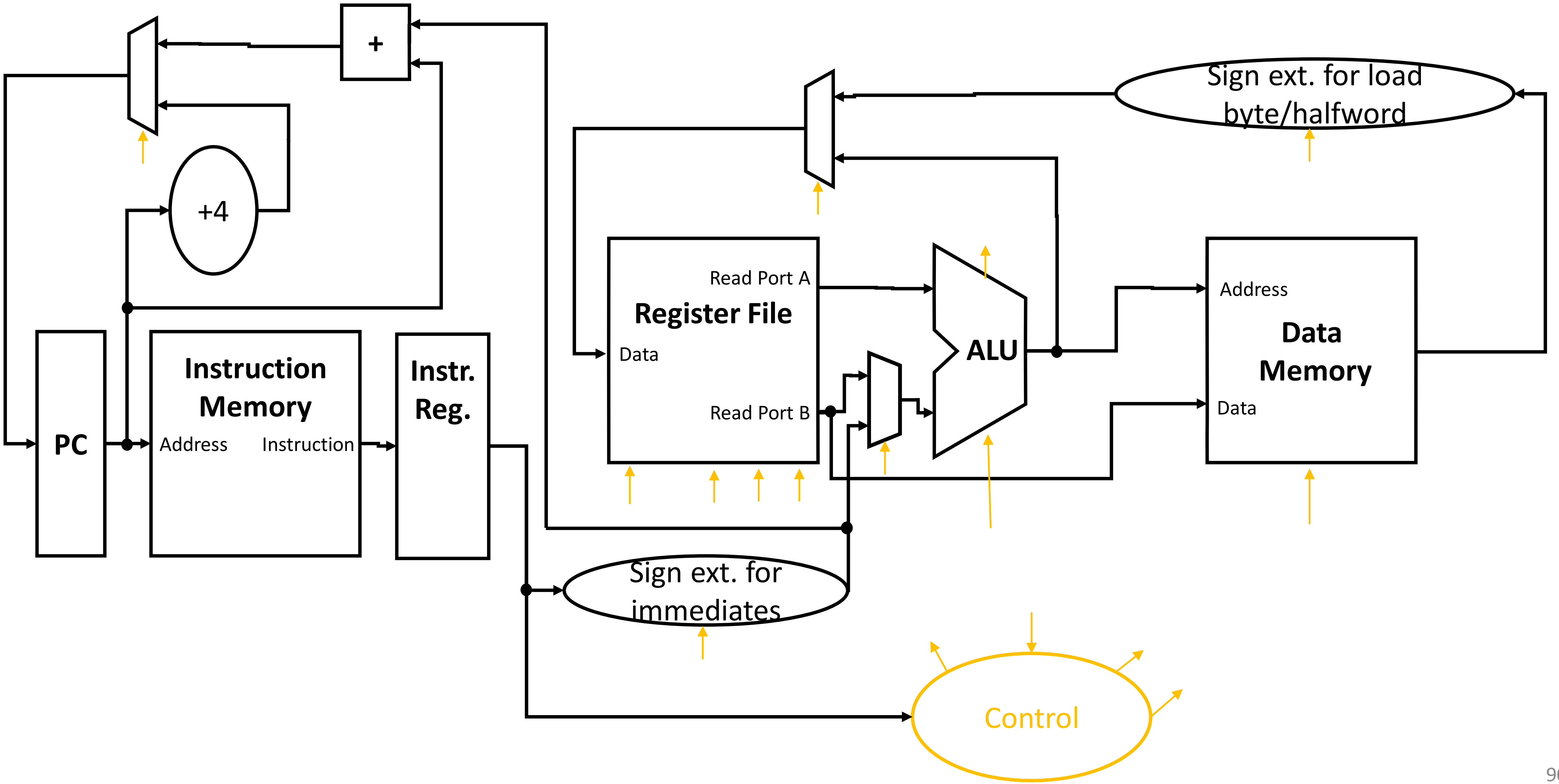
- Given that there are multiple cycles per instructions, it makes sense to draw an ASM graph for the CPU

- Example:
  - Simple CPU implementing LW, SW, ADD, AND and EBREAK (EBREAK is used in our simulation environment to halt the CPU)

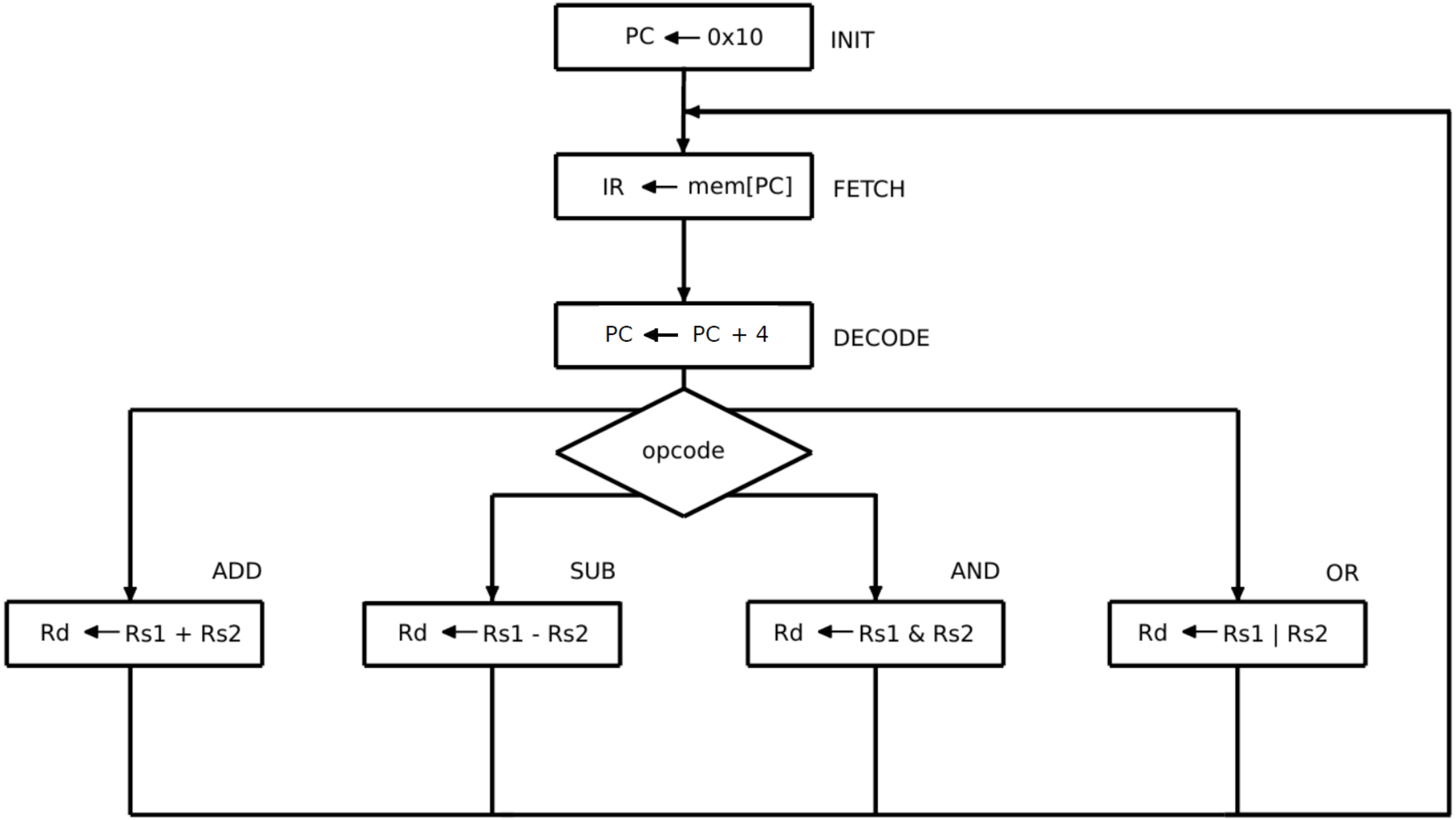
Simple Fetch/Decode/Execute ASM



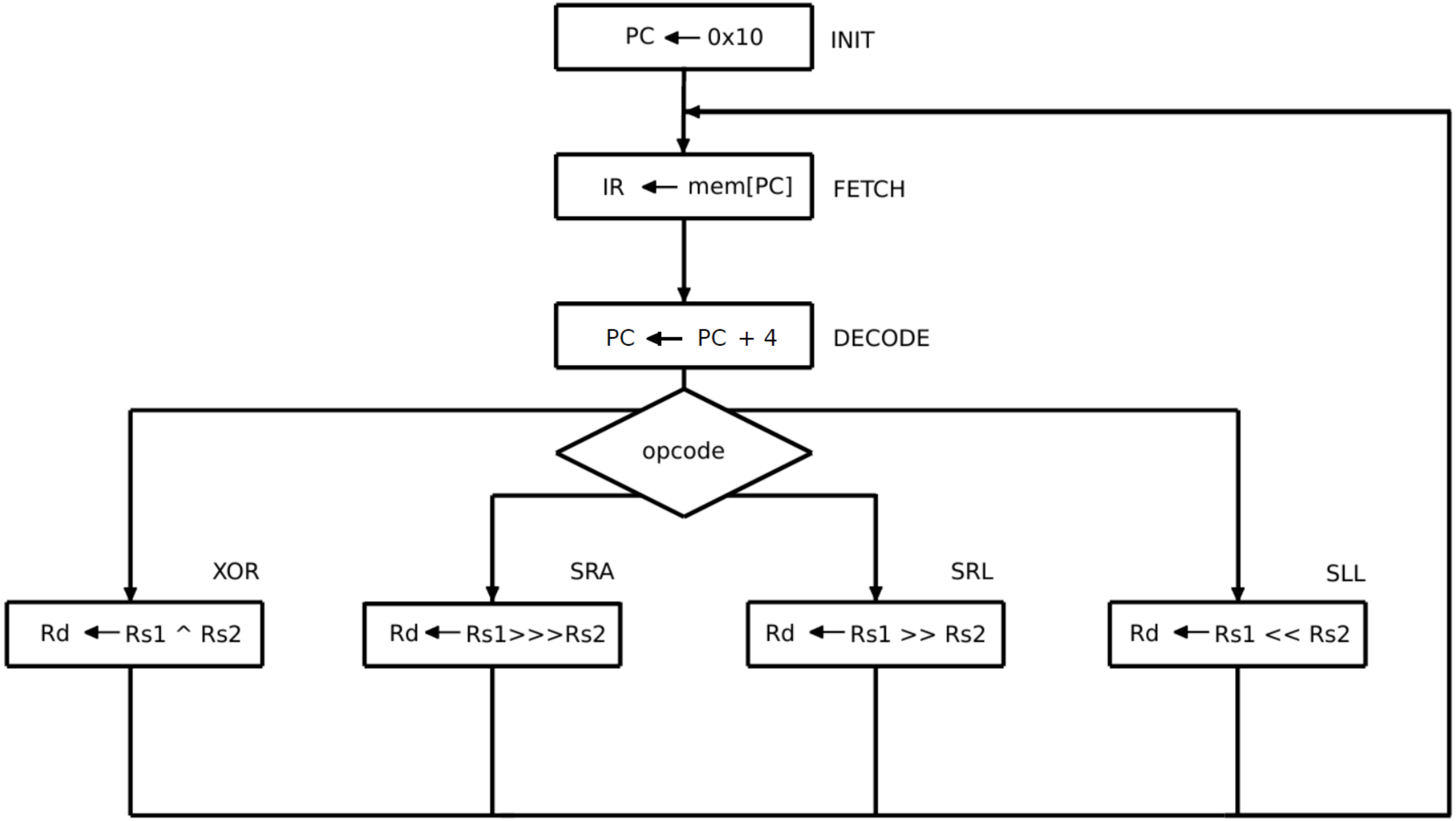
High-Level Overview (Single Cycle Datapath)



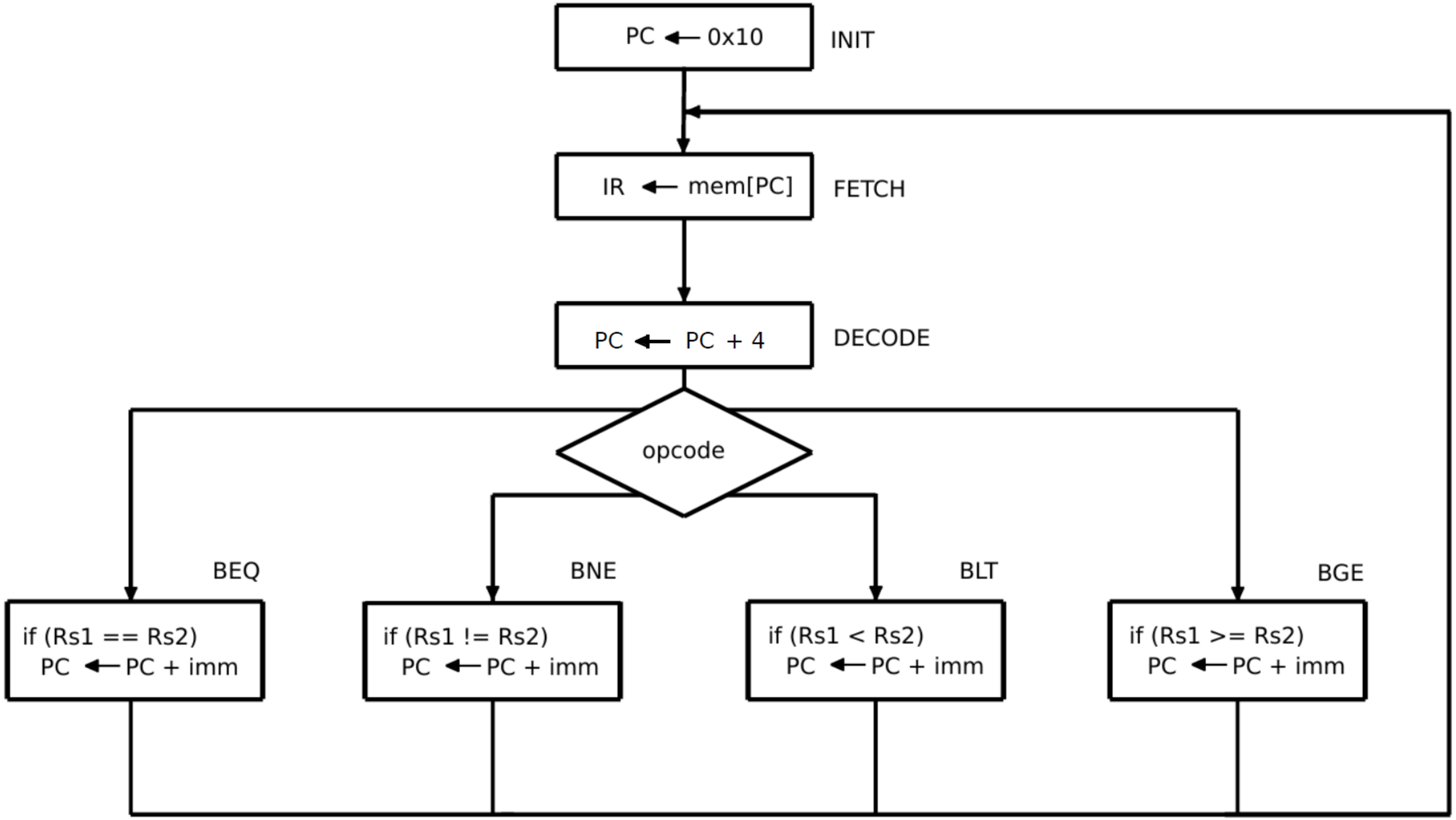
Extending the ASM Graph (Arithmetic/Logic Operations)



Extending the ASM Graph (Arithmetic/Logic Operations)



Extending the ASM Graph (Conditional Branches)



The Programmer’s View

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	$\mathtt{rd} = \mathtt{x1}$	$\mathtt{rs1} = \mathtt{x0}$	$\mathit{offset} = 0x20$
LW	$\mathtt{rd} = \mathtt{x2}$	$\mathtt{rs1} = \mathtt{x0}$	$\mathit{offset} = 0x24$
ADD	$\mathtt{rd} = \mathtt{x3}$	$\mathtt{rs1} = \mathtt{x1}$	$\mathtt{rs2} = \mathtt{x2}$
SW	$\mathtt{rs2} = \mathtt{x3}$	$\mathtt{rs1} = \mathtt{x0}$	$\mathit{offset} = 0x28$
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

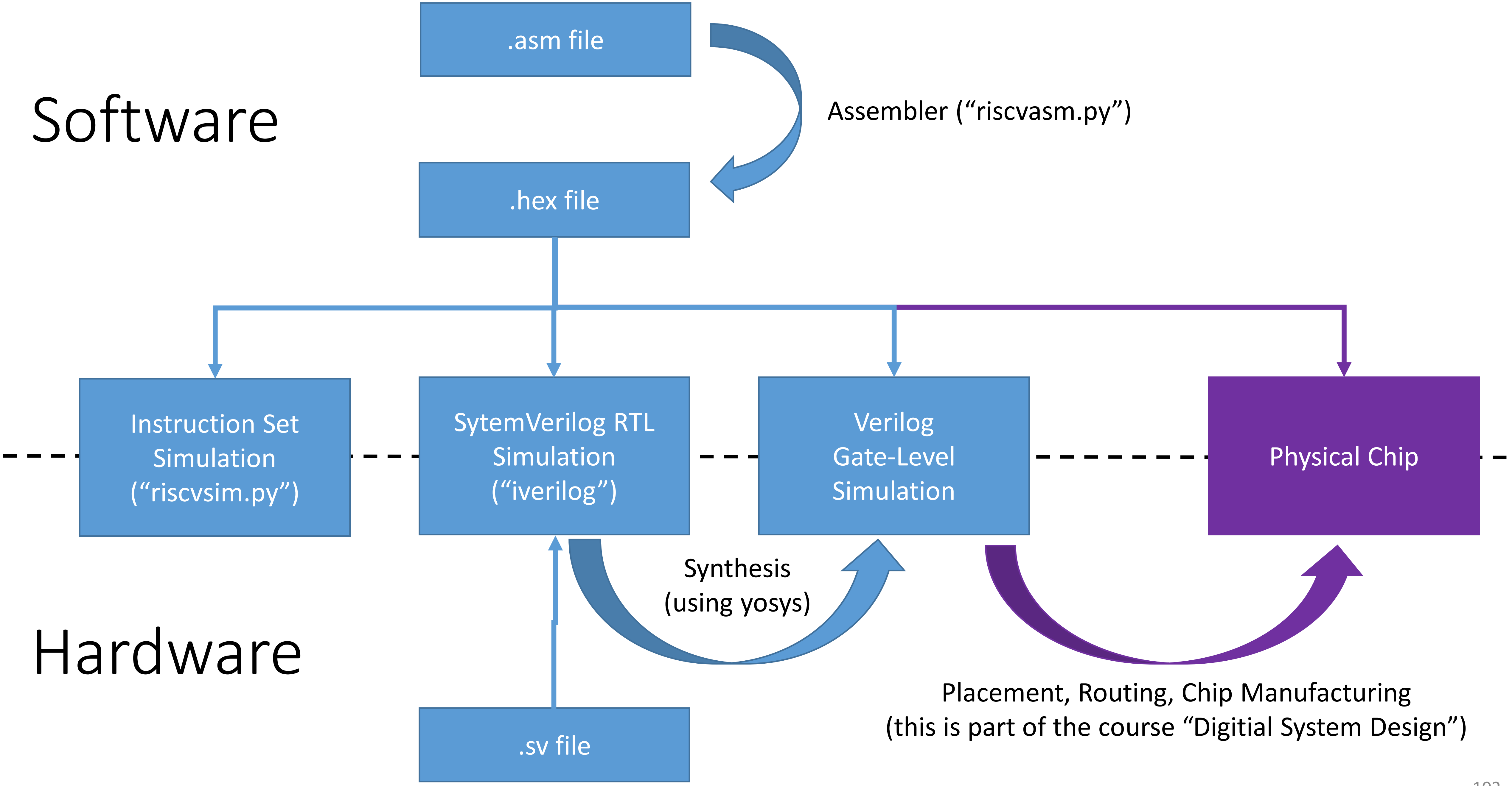
Instruction	Binary	Hexadecimal	Bytes
LW	00000010000000000010000010000011	0x02002083	83 20 00 02
LW	00000010010000000010000100000011	0x02402103	03 21 40 02
ADD	00000000001000001000000110110011	0x002081b3	b3 81 20 00
SW	00000010001100000010010000100011	0x02302423	23 24 30 02
EBREAK	00000000000100000000000001110011	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


.asm file
Assembler ("riscvasm.py")
Software
.hex file
Instruction Set Simulation ("riscvsim.py")
SytemVerilog RTL Simulation ("iverilog")
Verilog Gate-Level Simulation
Physical Chip
Synthesis (using yosys)
Hardware
.sv file
Placement, Routing, Chip Manufacturing
(this is part of the course "Digitial System Design")

The Demo Program Written in Assembly

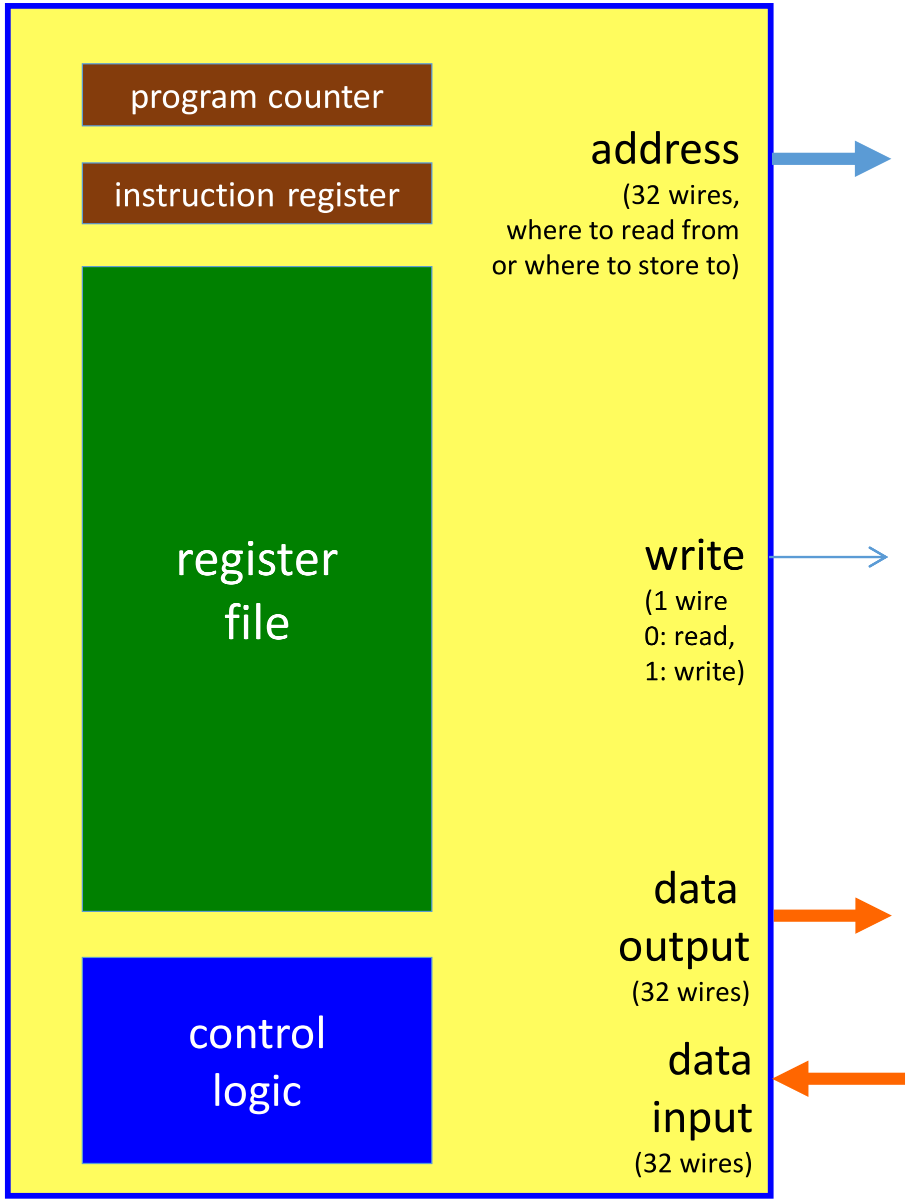
```
.org 0x00 # start program at address 0x00
  LW x1, 0x20(x0)
  LW x2, 0x24(x0)
  ADD x3, x1, x2
  SW x3, 0x28(x0)
  EBREAK

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

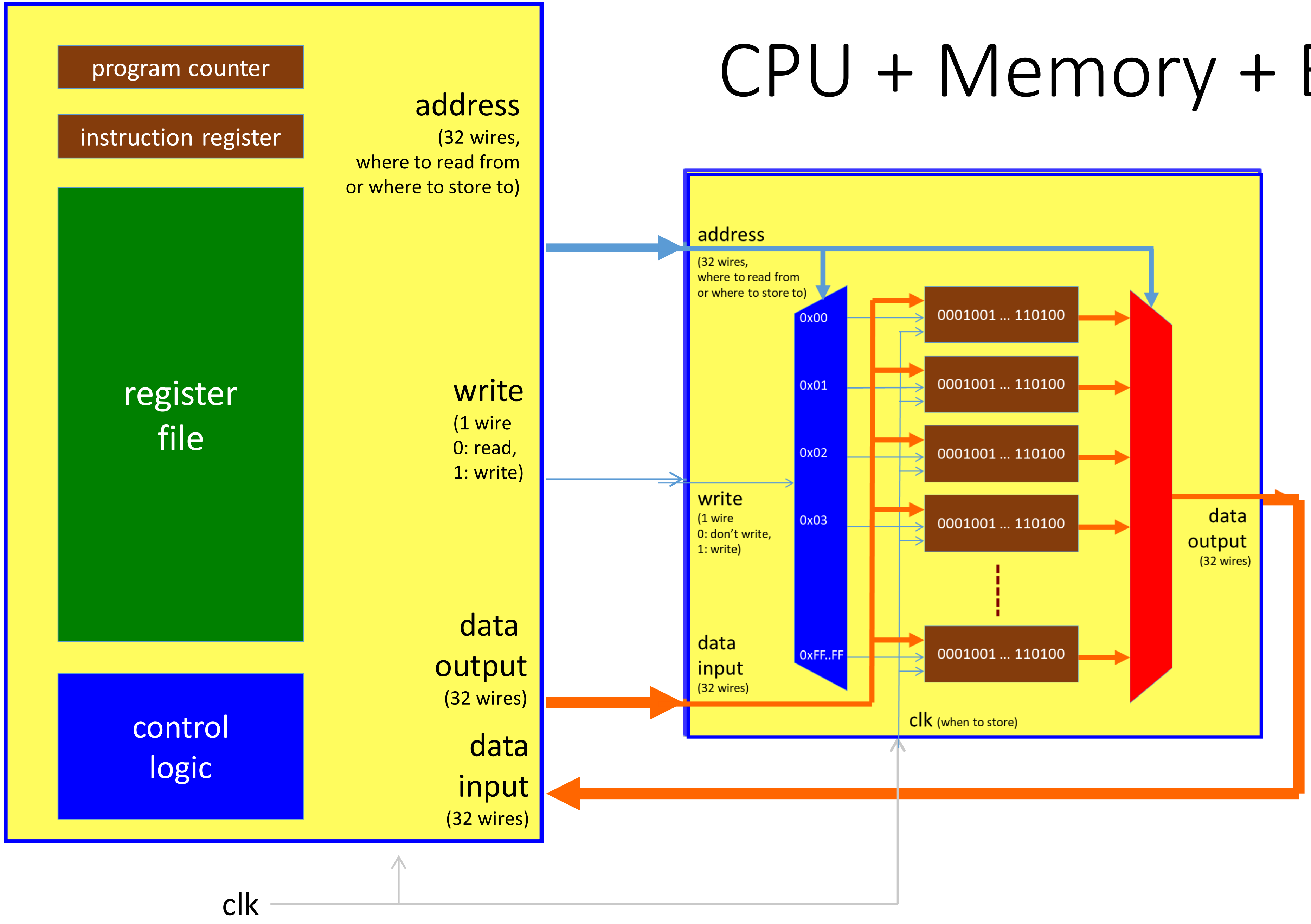
Try out to assemble and simulate your own code based on

con04_adding-two-constants

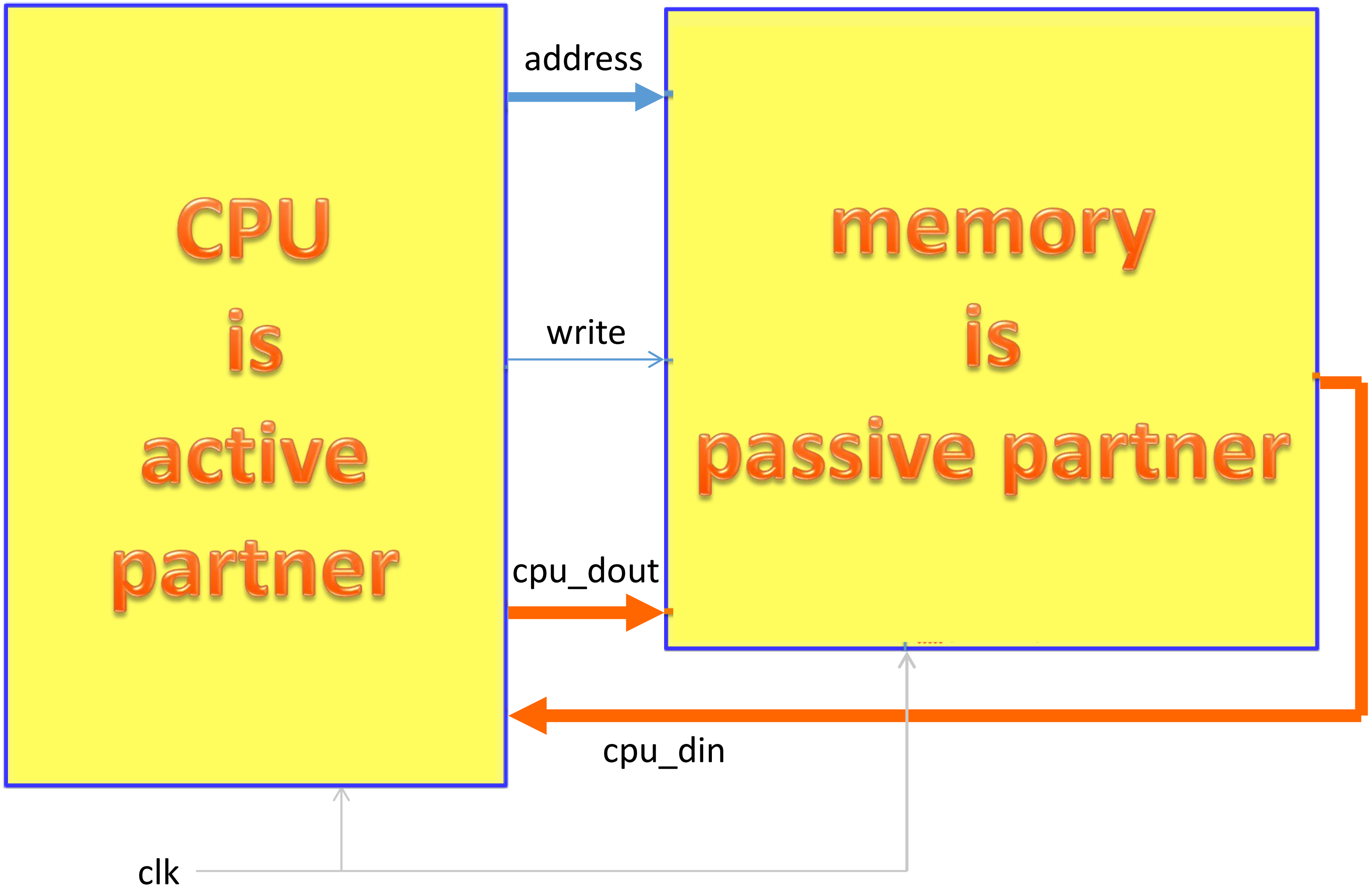
Programmer's View on the CPU



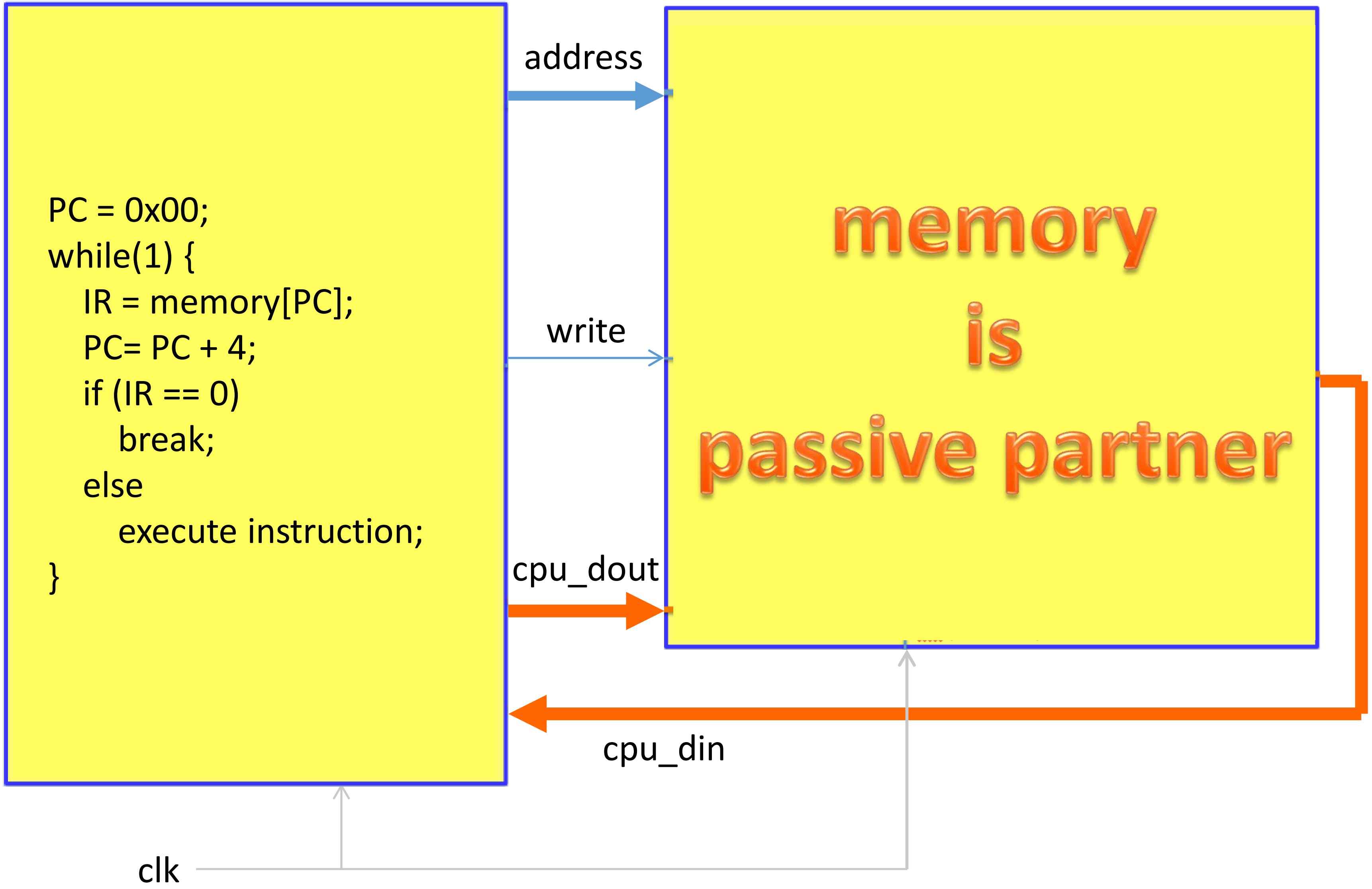
CPU + Memory + Bus



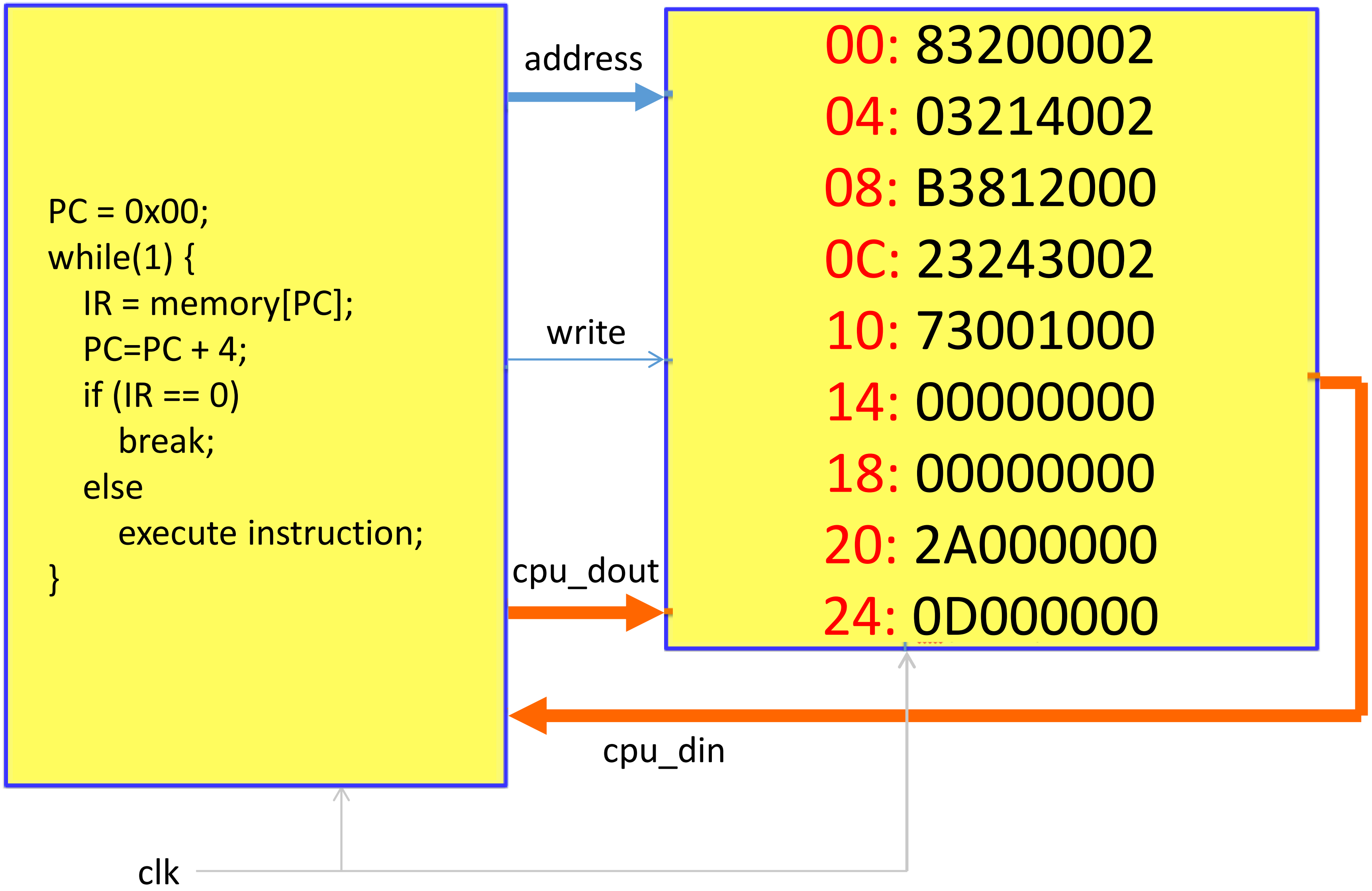
CPU is Active, Memory is Passive



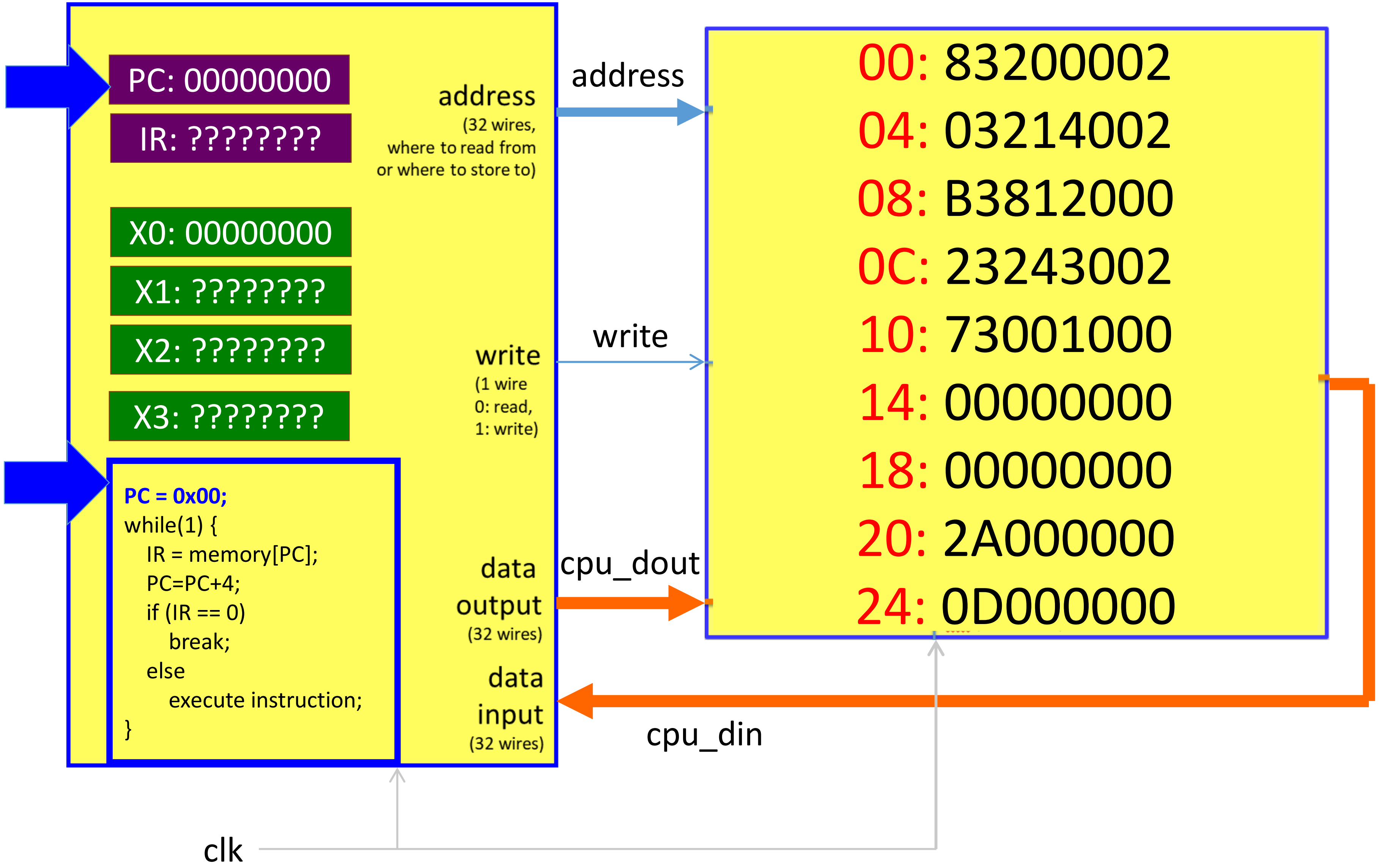
CPU’s Job: Fetch, Decode, and Execute



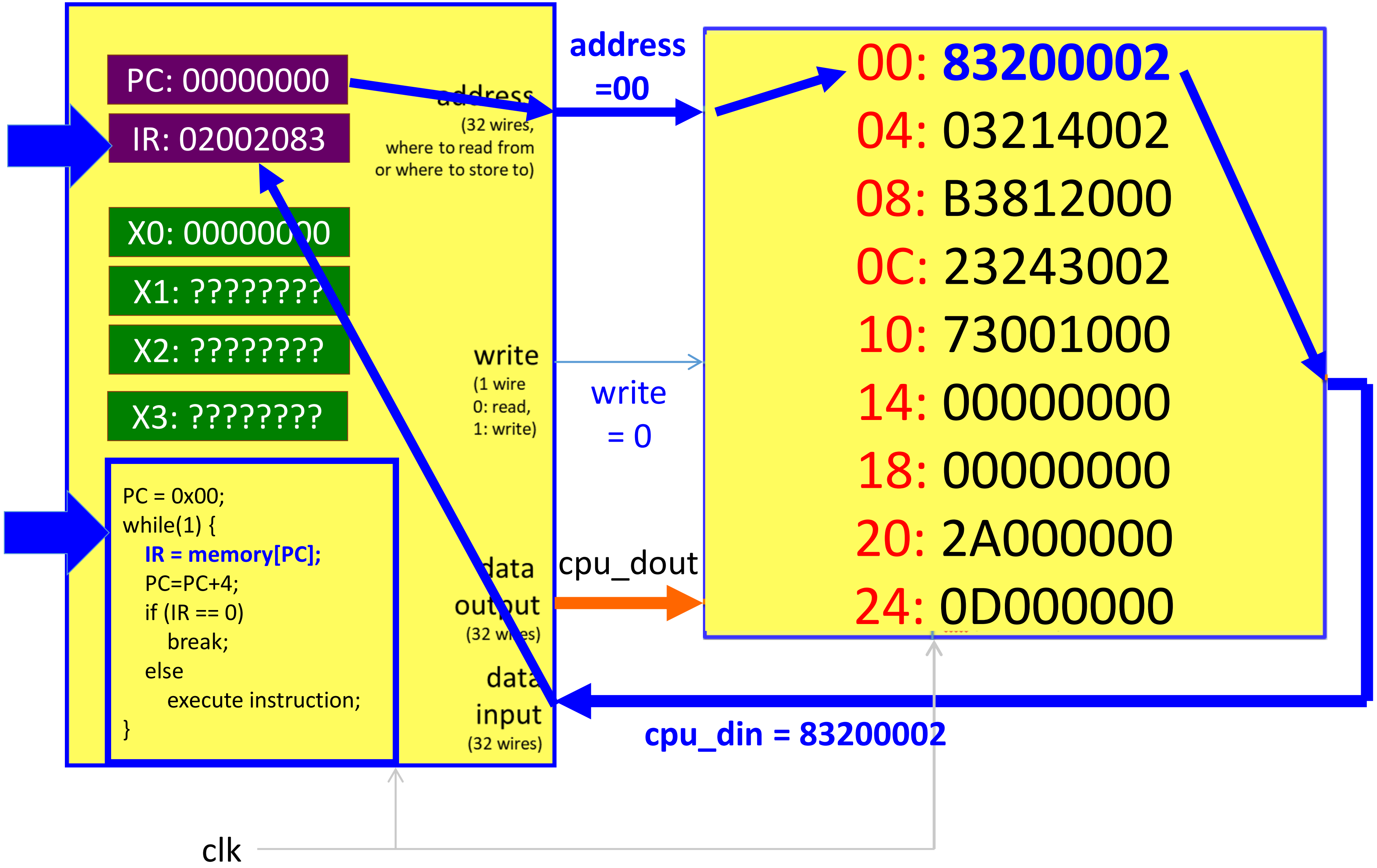
Example: Content in Main Memory



(1) Initialize PC with 0x00000000

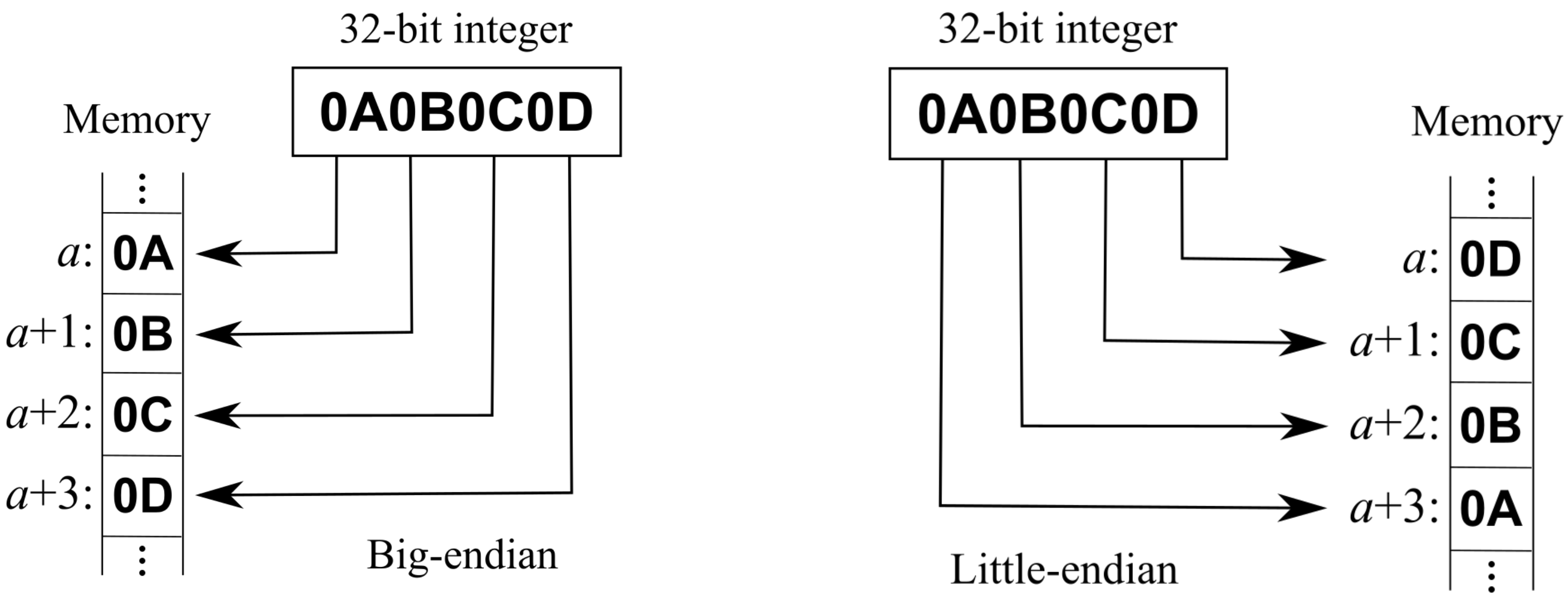


(2) Fetch First Instruction

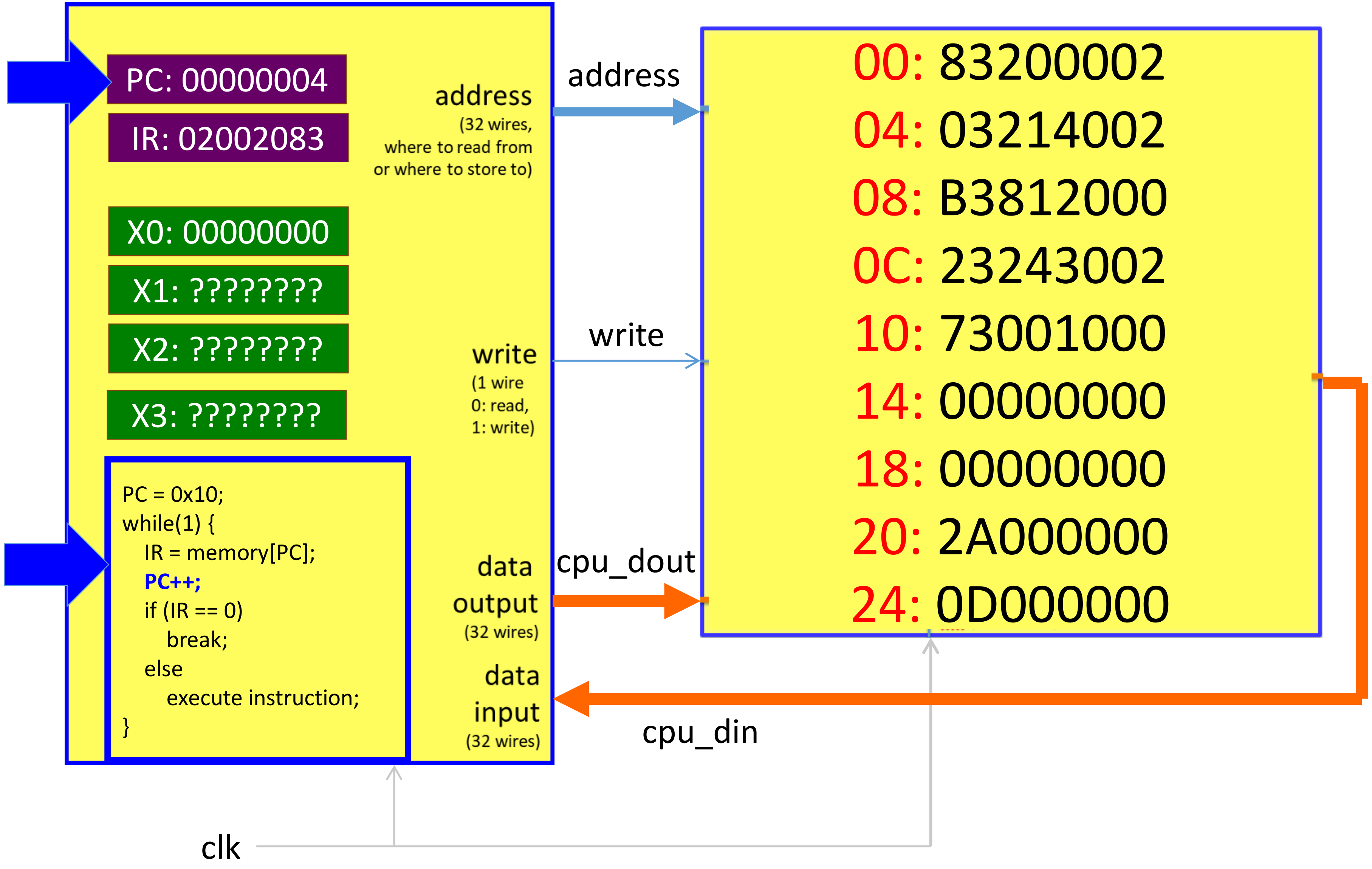


Note on Endianess

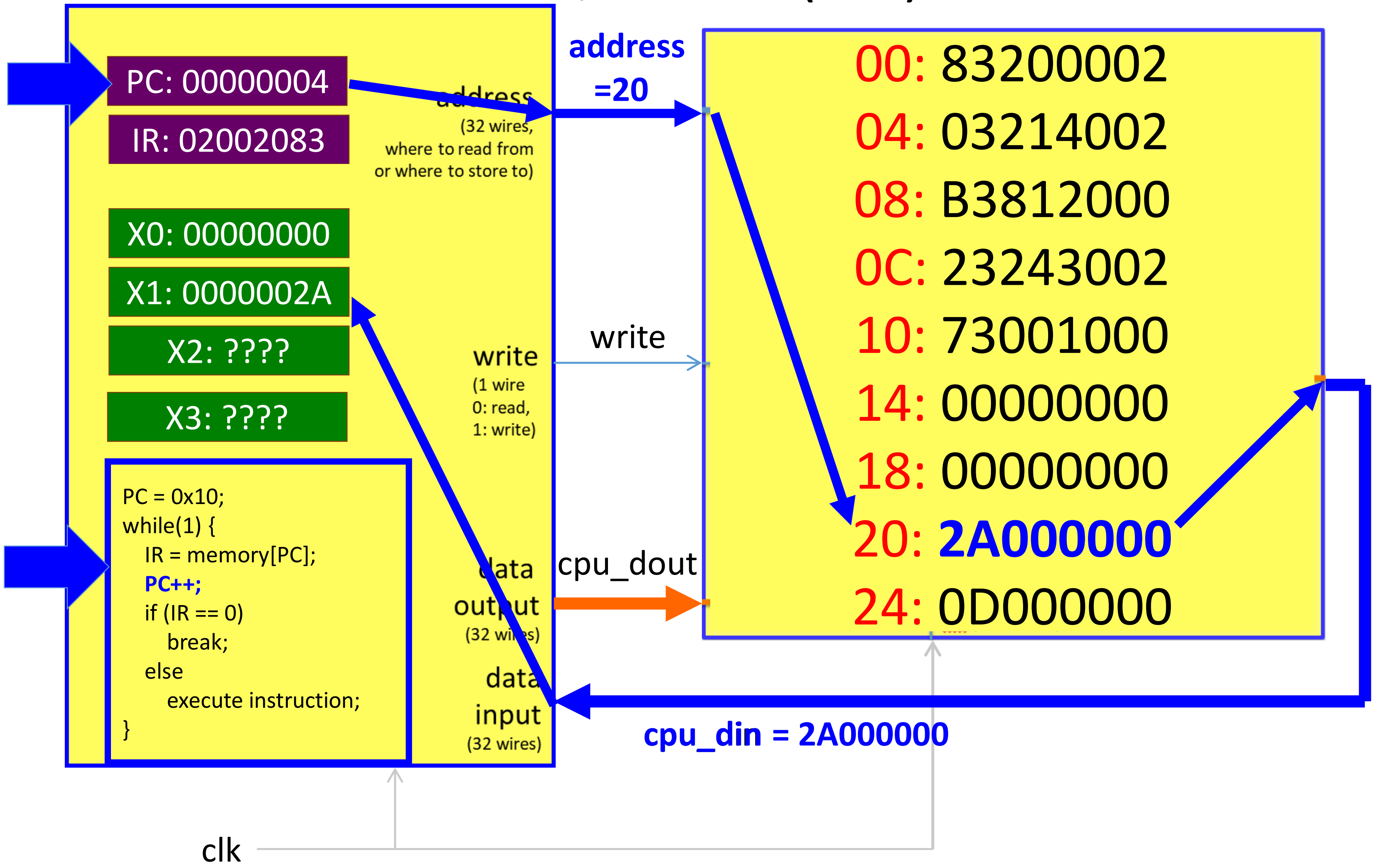
- There are two options for the sequence of storing the bytes of a word in memory:
  - Little endian: least significant byte is at the lowest address
  - Big endian: most significant byte is at lowest address



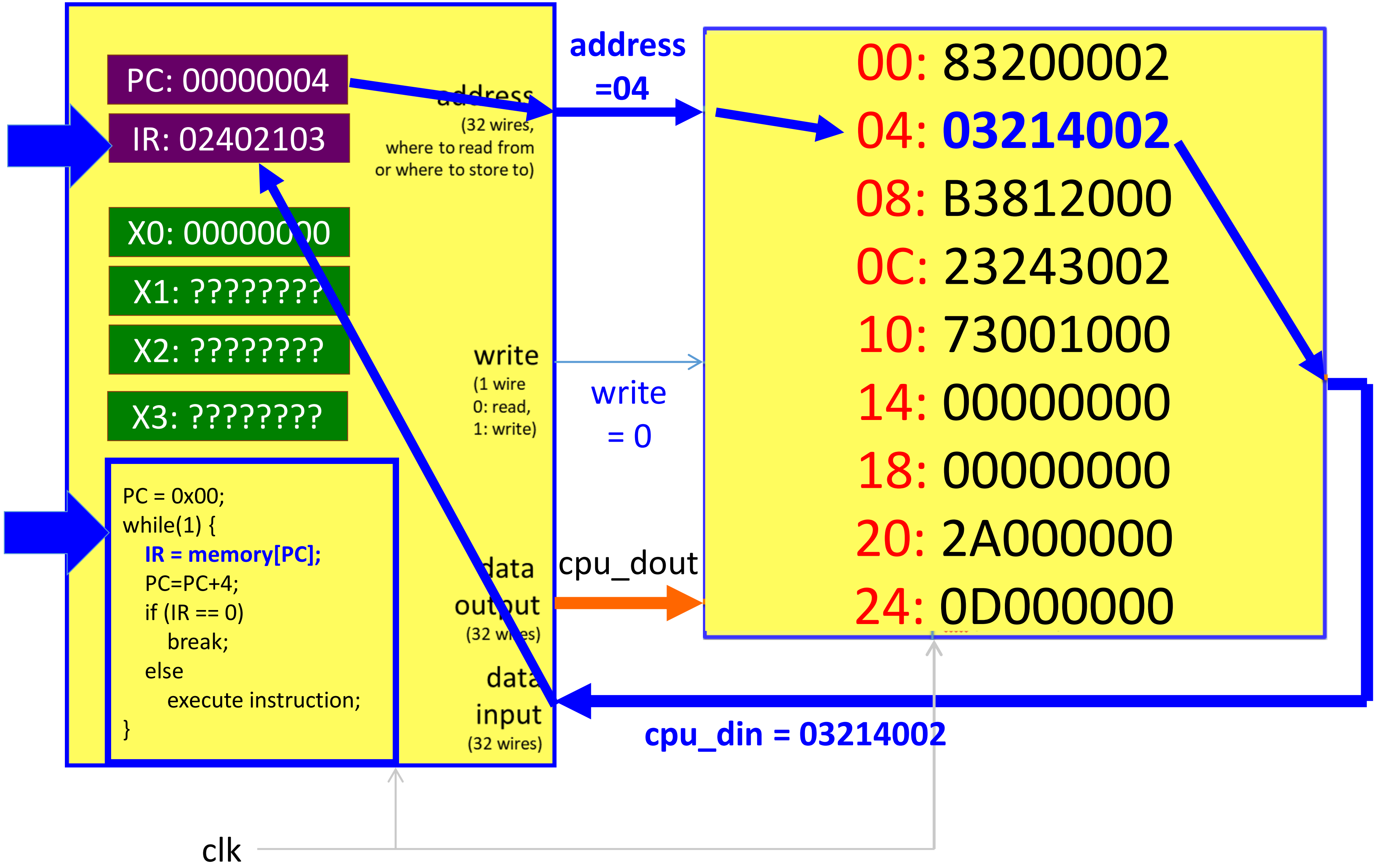
(3) Increment Value in PC



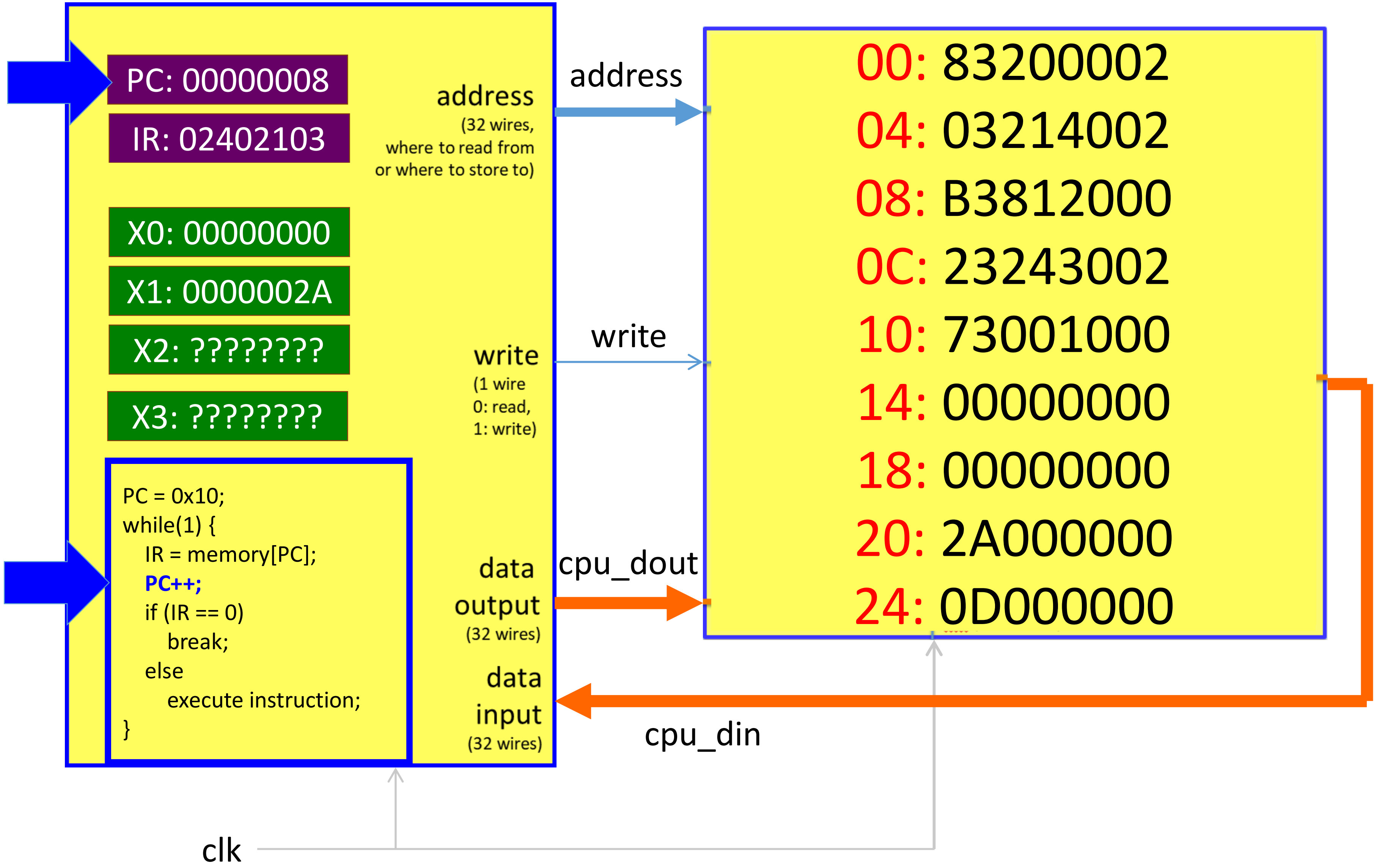
(4) Decode and Execute Machine Instruction 0x83200002: LW x1, 0x20(x0)



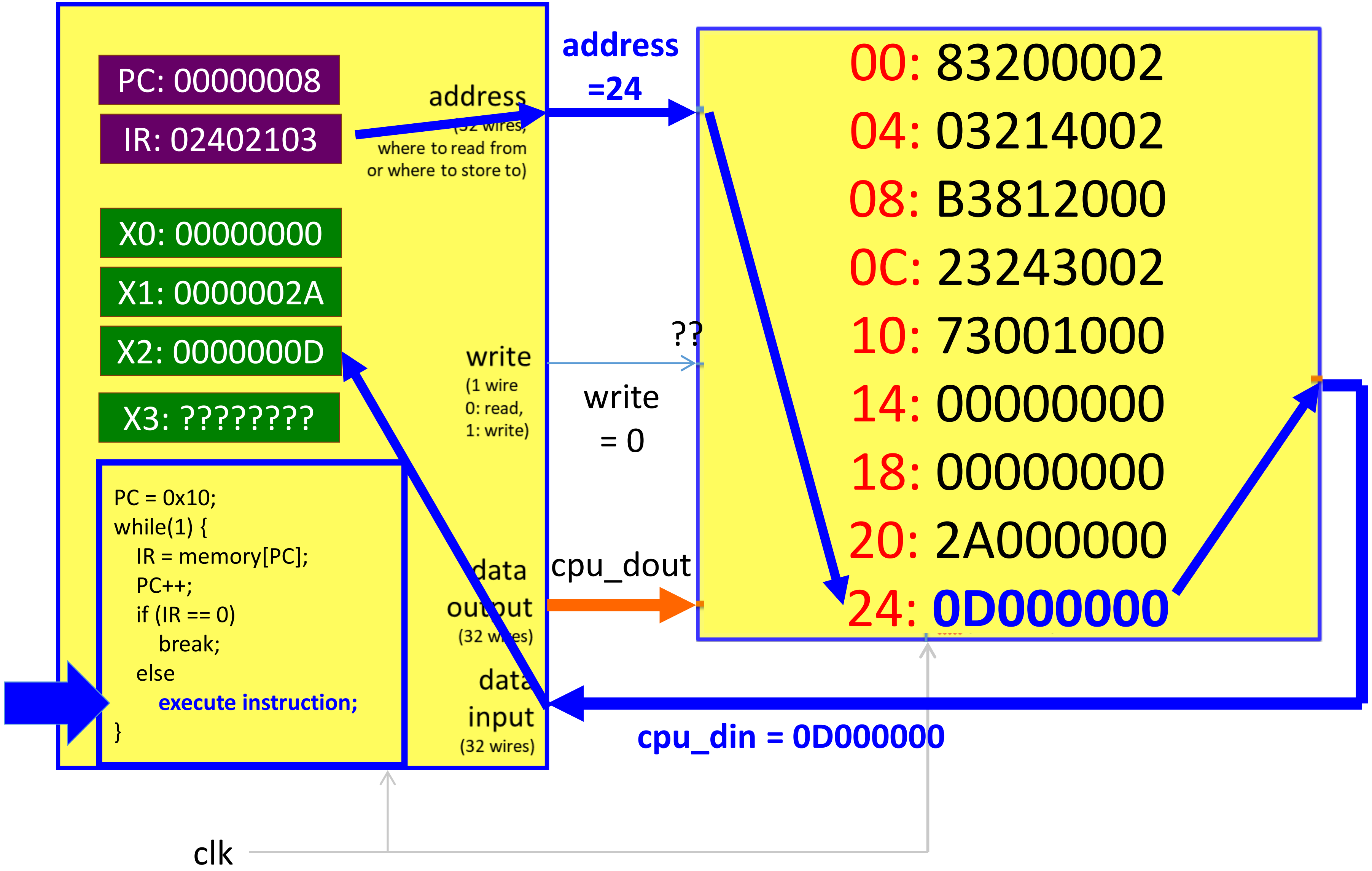
(5) Fetch Second Instruction



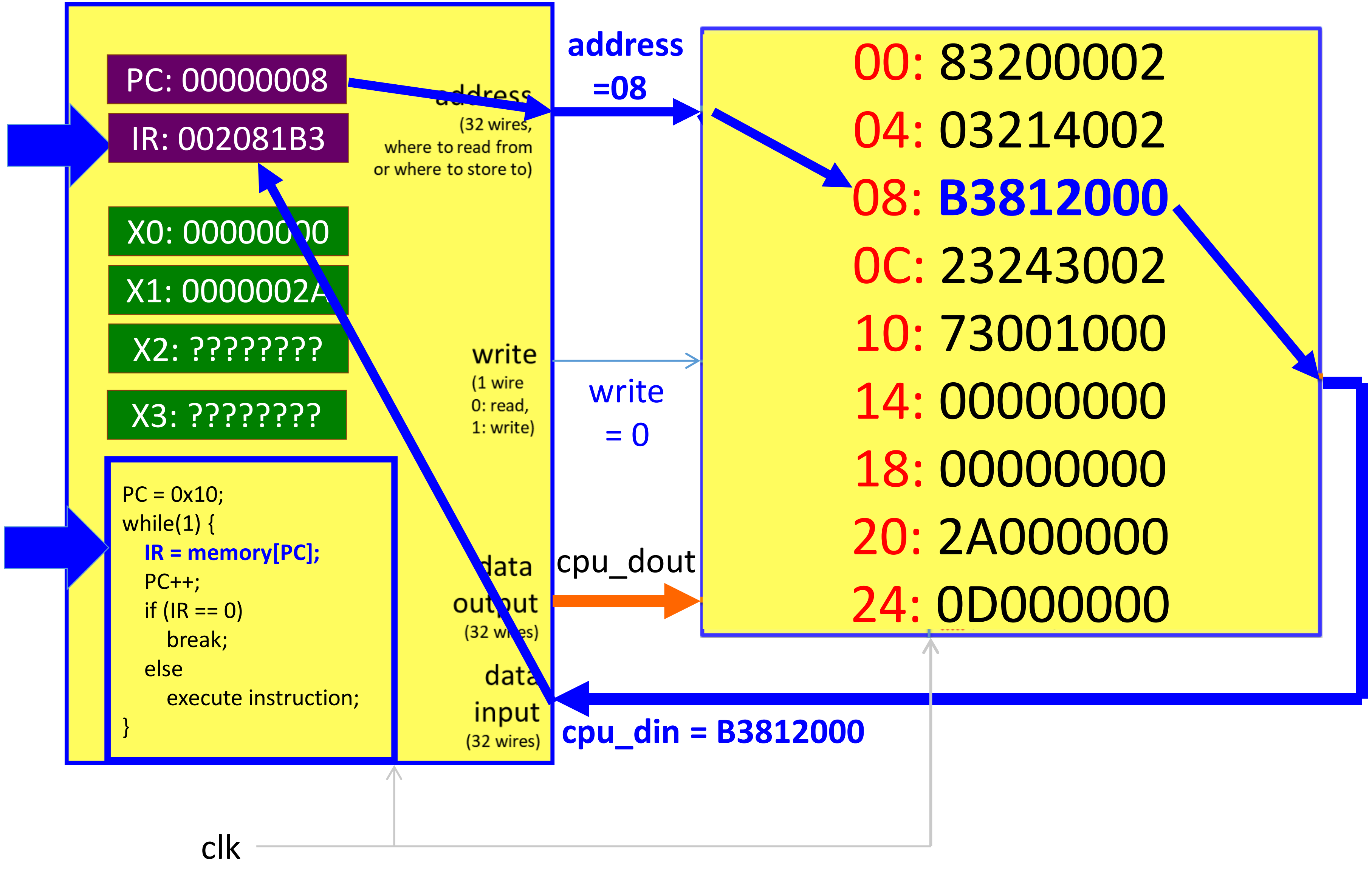
(6) Increment value in PC



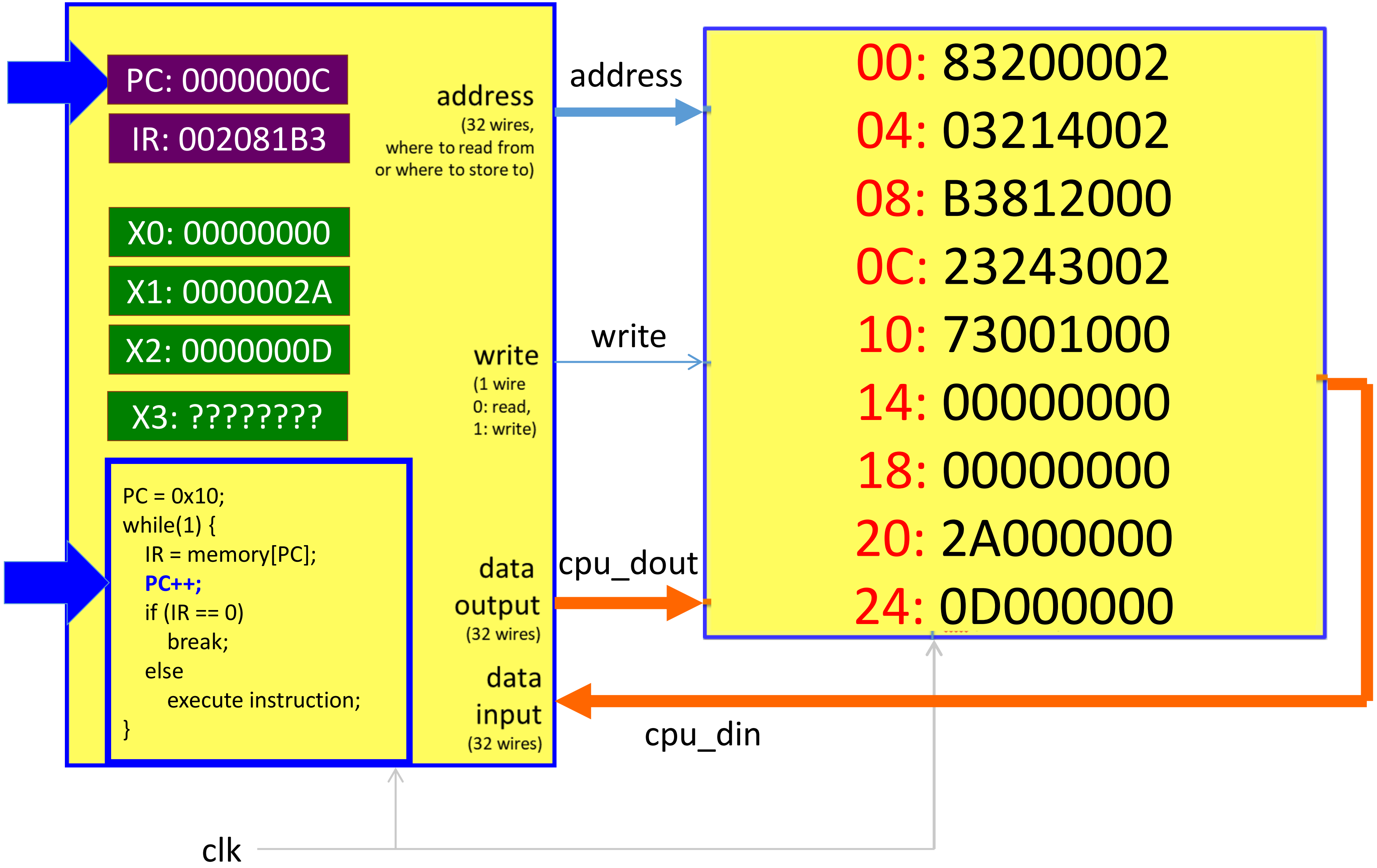
(7) Decode and Execute Machine Instruction 0x03214002 : LW x2, 0x24(x0)



(8) Fetch third instruction



(9) Increment value in PC



(10) Decode and Execute Machine Instruction 0x03214002 : ADD x3, x1, x2