

Computer Architecture

Processor implementation

<http://d3s.mff.cuni.cz/teaching/nswi143>



CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Lubomír Bulej

bulej@d3s.mff.cuni.cz

Implementing simplified RISC-V ISA

- **RISC-V: a family of related ISAs**

- Primary base integer variants: RV32I, RV64I
- RV32E (subset of RV32I), RV128I (future)

- **RV32I**

- 32 general-purpose 32-bit registers: $x0 - x31$
 - $x0$ /zero wired to zero
 - other uses defined by ABI
- 32-bit program counter register (PC)
- Control and status registers (CSRs)
 - Exception cause, exception instruction address, timer counter, etc.



Implementing simplified RISC-V ISA (2)

- **Memory**

- Access to 4-byte aligned addresses only
 - Corresponds to 32-bit word length of the processor
- Indirect addressing with immediate displacement
 - **Load:** $R2 := \text{mem}[R1 + \text{immediate}]$
 - **Store:** $\text{mem}[R1 + \text{immediate}] := R2$



Implementing simplified RISC-V ISA (3)

● Operations

■ Arithmetic and logic

- Fully orthogonal, three-operand instructions
- Source operands: register/register, register/immediate
- Target operand: register
- Includes data movement between registers

■ Load/store operations

- Move data between registers and memory (load/store architecture)

■ Conditional branch

- Tests equality/inequality of two registers

■ Unconditional jumps

- Including jumps to subroutine and indirect jumps (return from a subroutine)

■ Special instructions



Implementing simplified RISC-V ISA (4)

- **Single-cycle datapath**
 - Basic organization of data path elements
 - Combinational and sequential blocks
 - Operations executed in one long cycle
 - Suitable for operations of similar complexity
 - Writes to memory elements synchronized by clock
 - Clock signal is implicit, will not be shown
 - **Simplification:** separate instruction memory (Harvard architecture)



Implementing simplified RISC-V ISA (5)

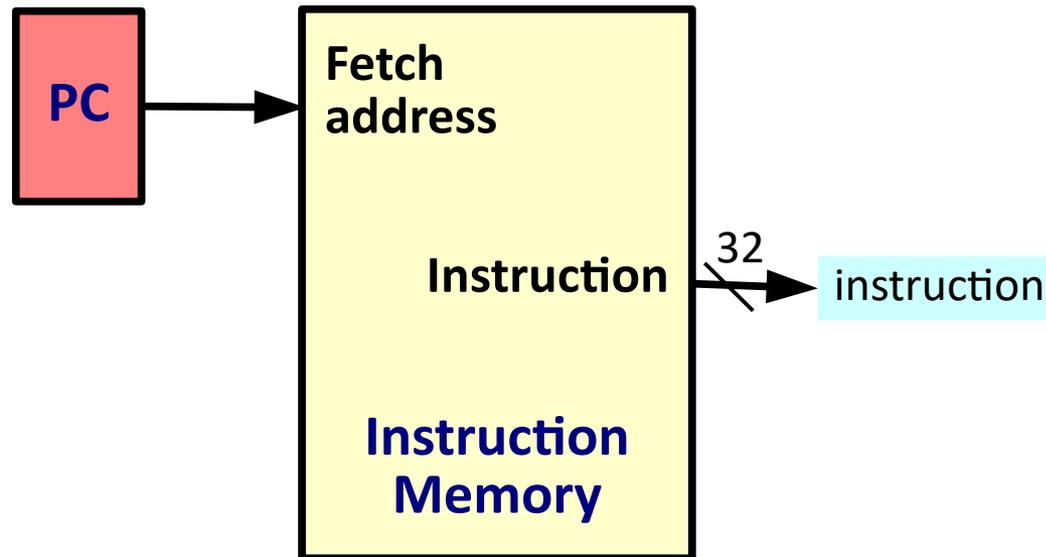
● Steps to execute an instruction

1. Fetch instruction from memory
 - Read from an address supplied by the PC register
2. Decode instruction and fetch instruction operands
3. Execute operation corresponding to the opcode
 - Register operations, computing address for accessing memory, comparing operands for conditional branch.
4. Store the result of the operation
 - Write data to register or memory
5. Adjust PC to point at next instruction
 - One that immediately follows the current
 - One that is a target of a jump or branch



Reading an instruction (fetch)

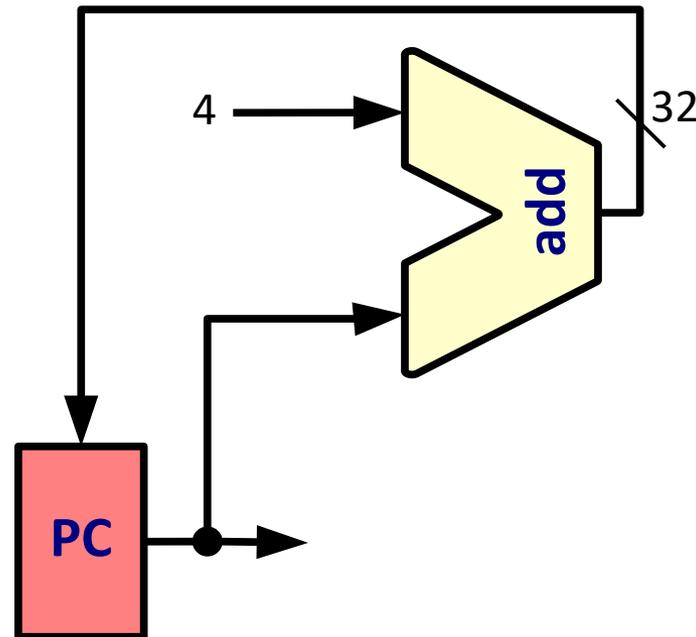
- **Address provided by PC register**
 - PC not directly accessible to a programmer



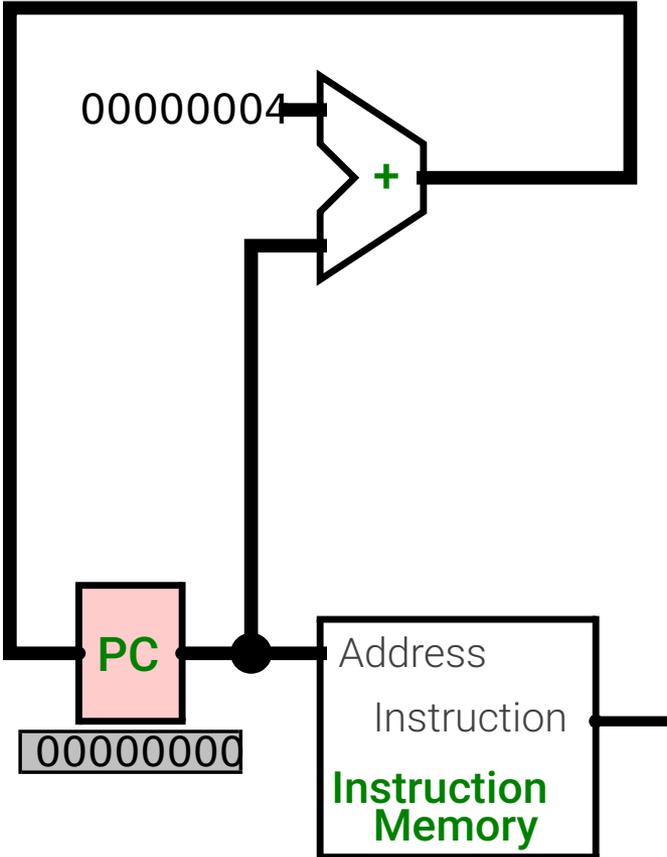
Advancing to next instruction

- **Increment PC by 4**

- Default unless branching or jumping
- Simple thanks to fixed instruction size

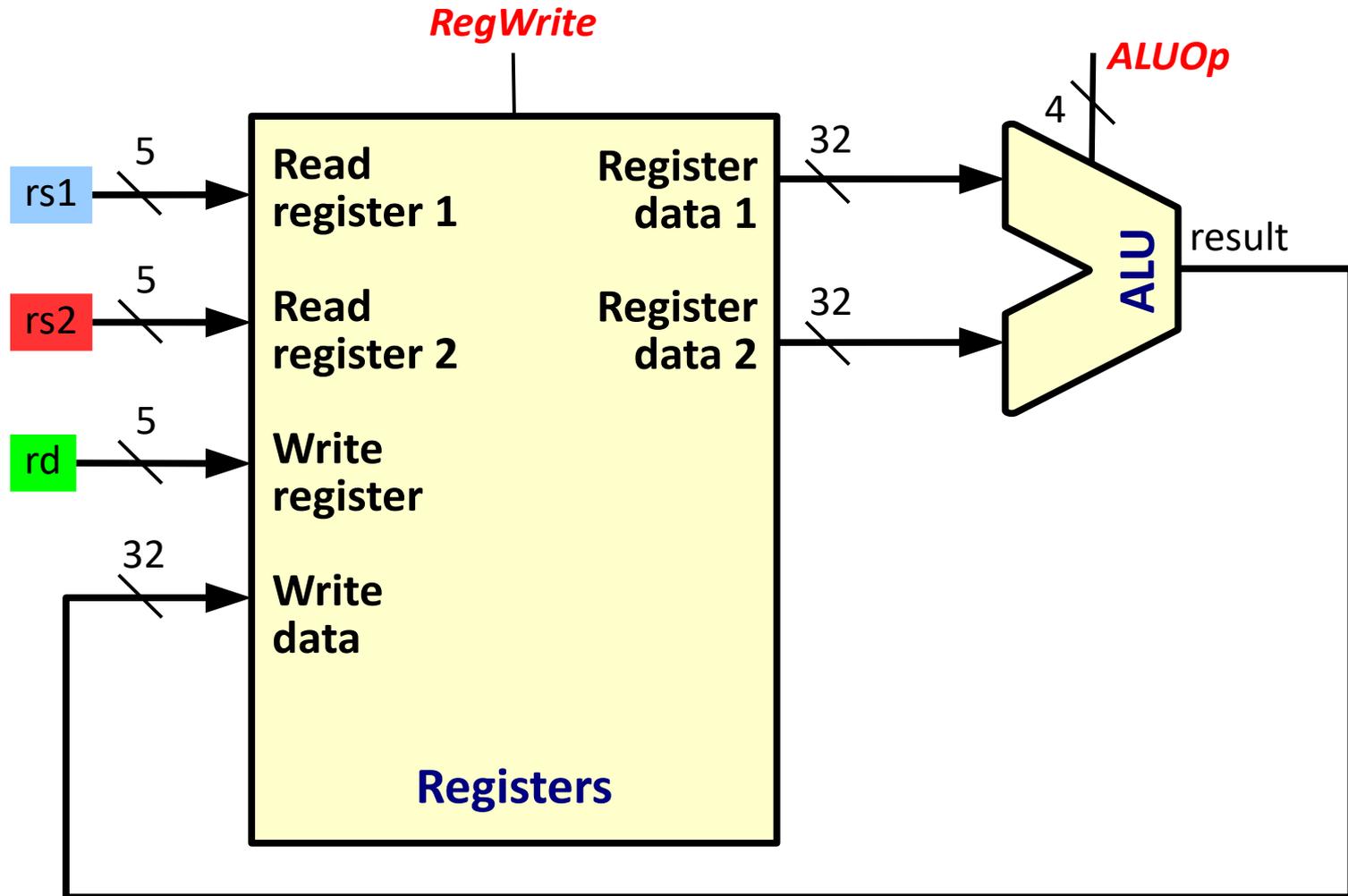


Support for reading instructions

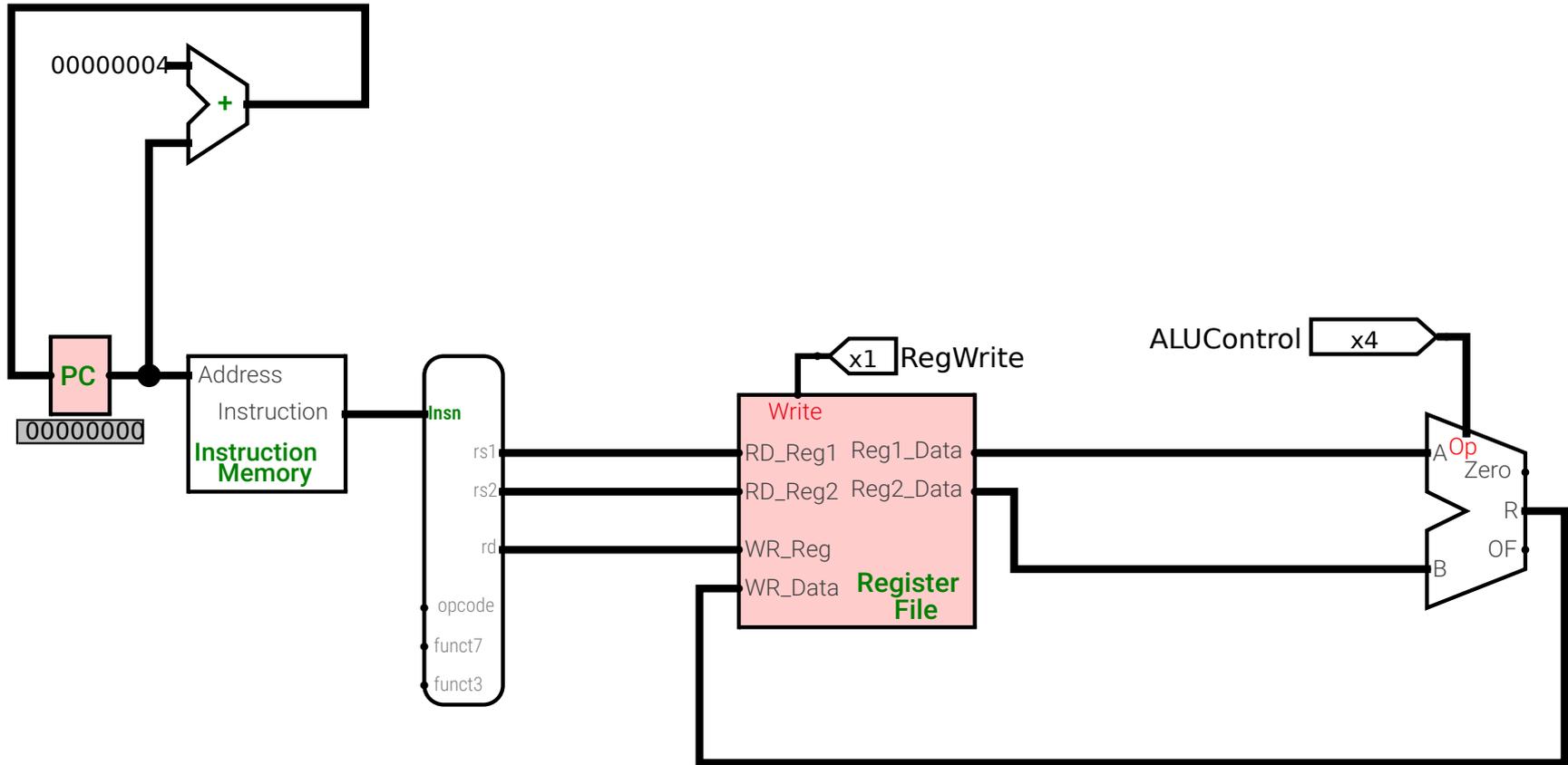


Register operations (add, sub, ...)

R-type format

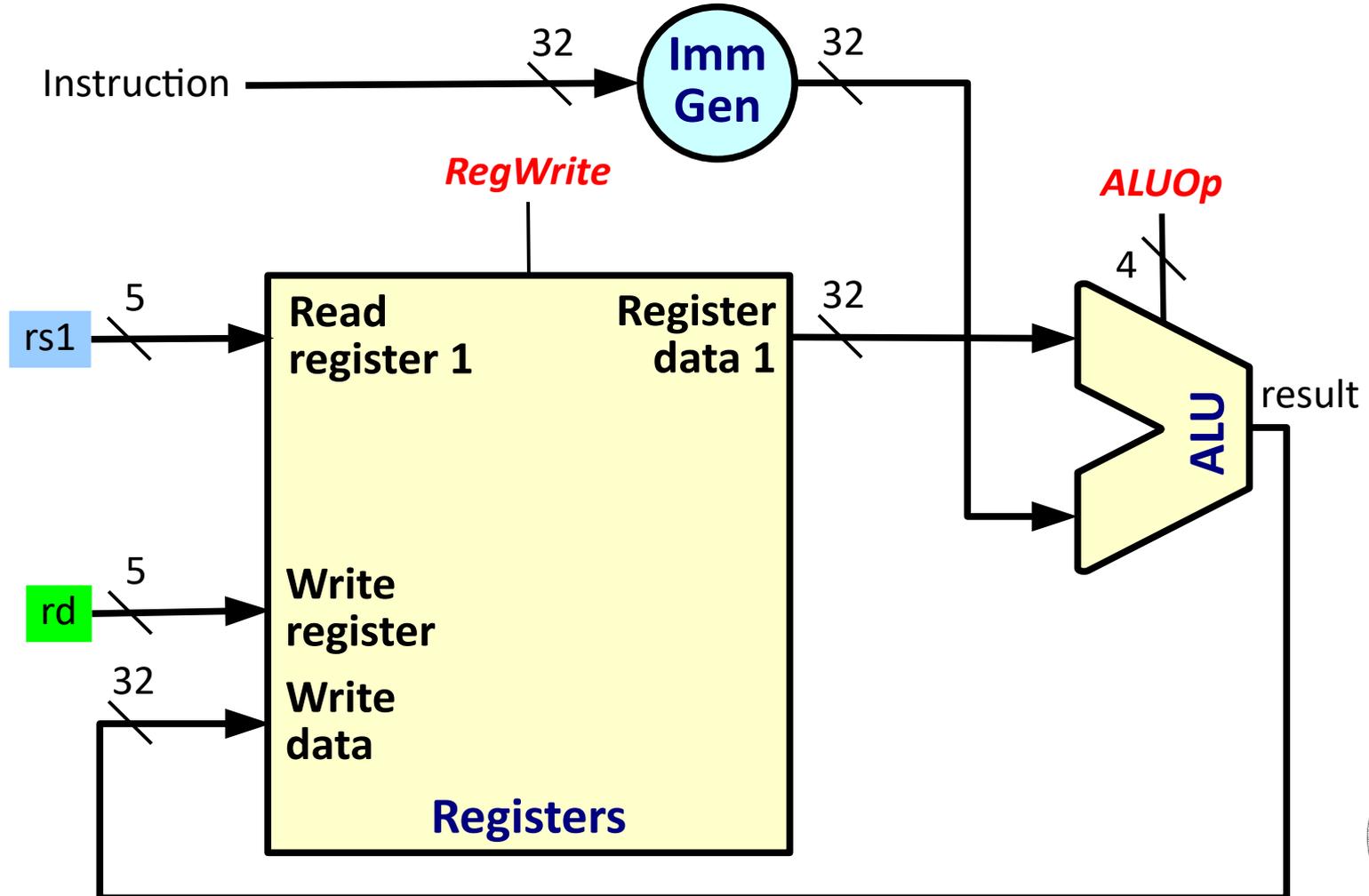


Support for register operations



Immediate operand operations (addi, ...)

I-type format



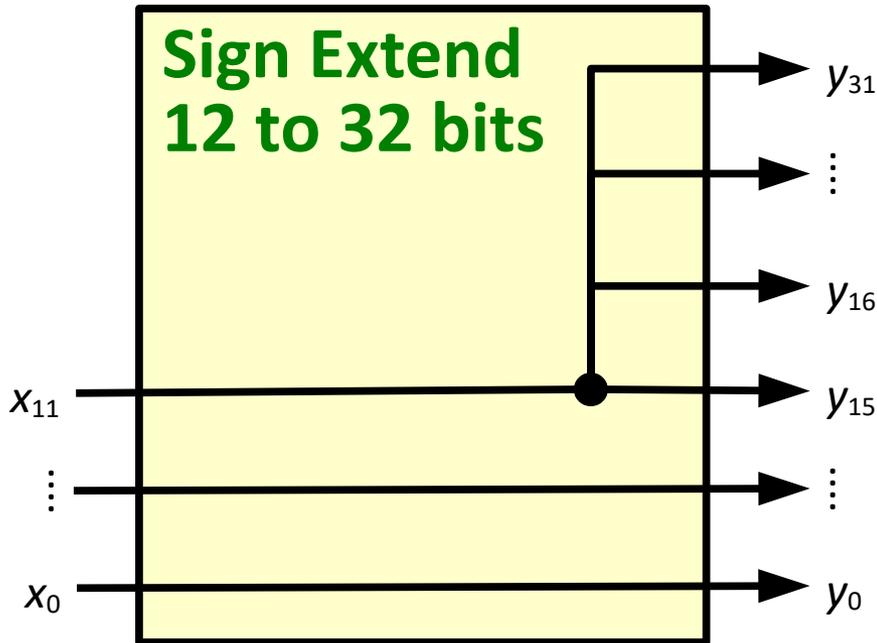
Generating immediate values

- **Values from instruction word**
 - Different bit lengths (12 or 20 bits)
 - Some values shifted left by one bit
 - Bit positions depend on format
 - All values sign-extended to 32-bits

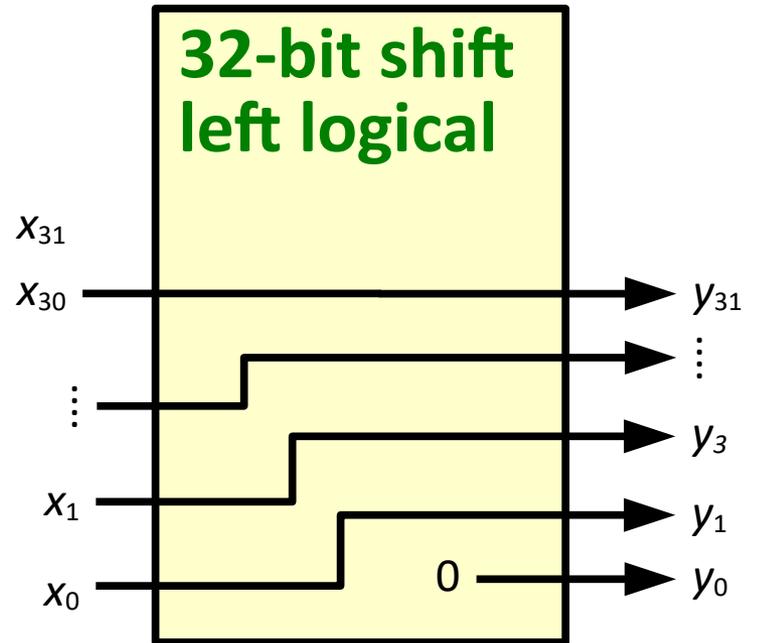


Generating immediate values (2)

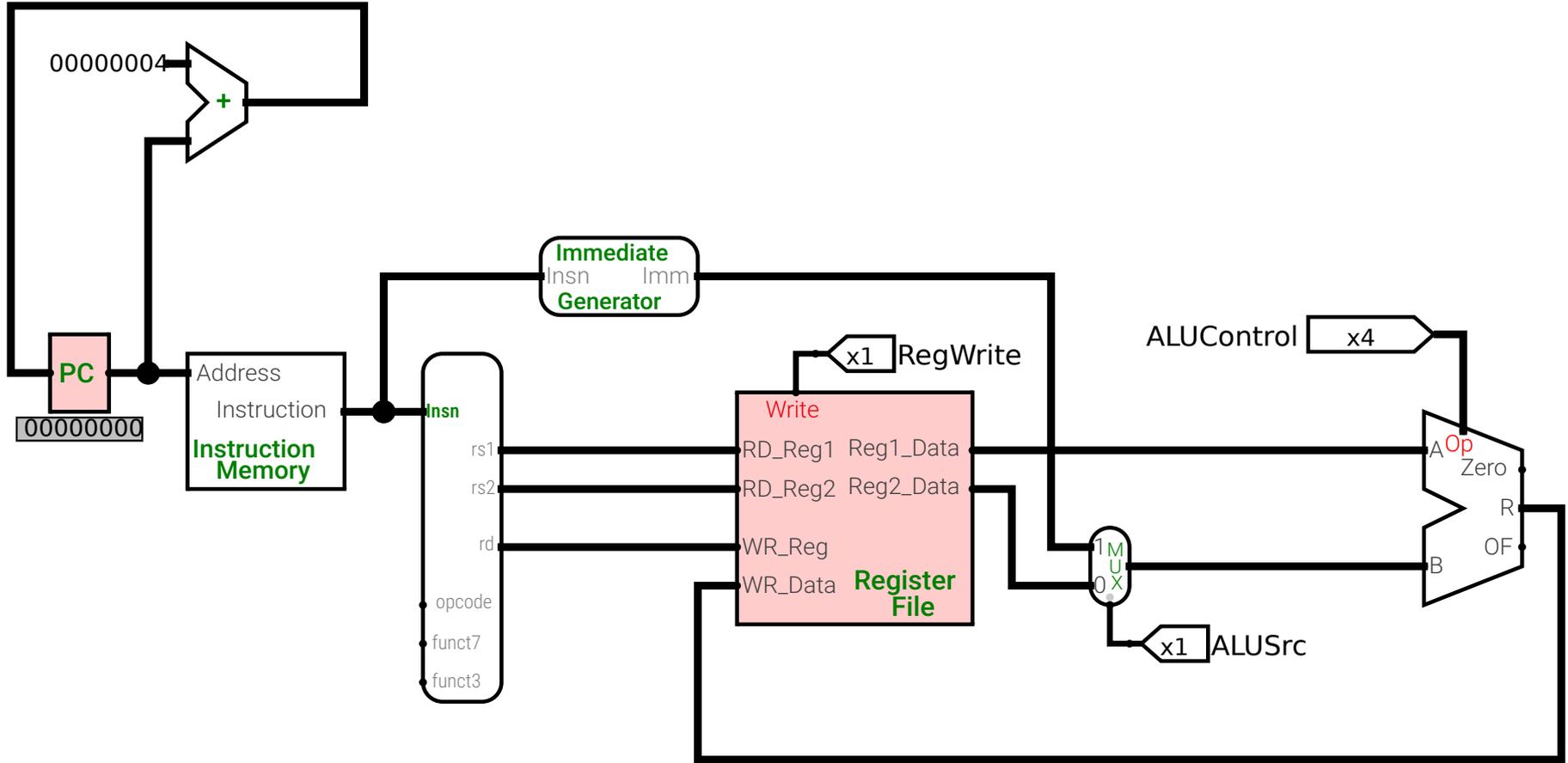
- **Sign extension**



- **Logical shift**

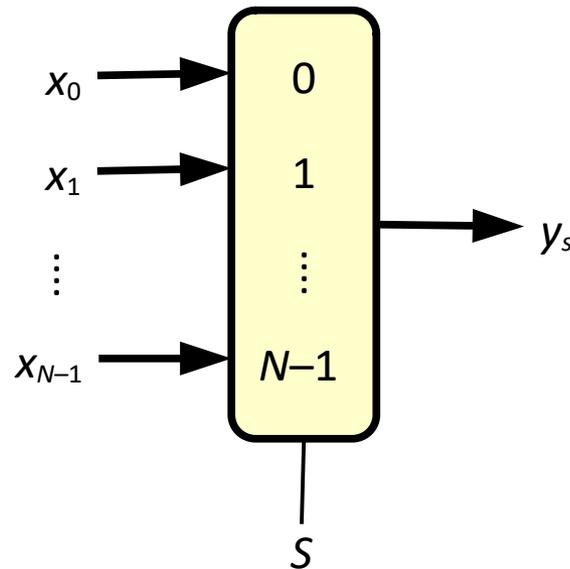


Support for immediate operands



Multiplexer (mux)

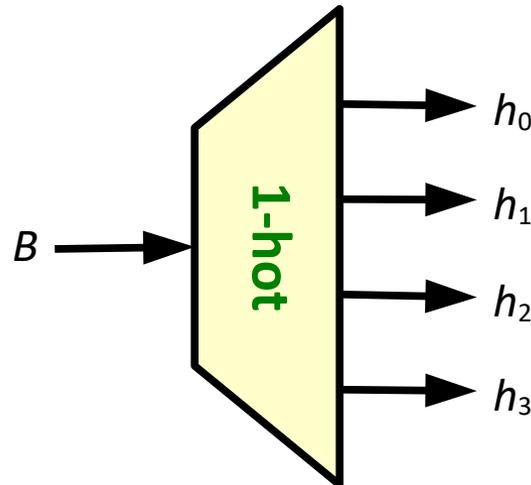
- **Selects one of several inputs**
 - **Selector:** n -bit number $S \in \{0, \dots, 2^n - 1\}$
 - **Data input:** $N = 2^n$ m -bit values x_0, x_1, \dots, x_{N-1}
 - **Data output:** m -bit value $y = x_S$



Implementing a multiplexer

- **Binary to “1-hot” decoder**

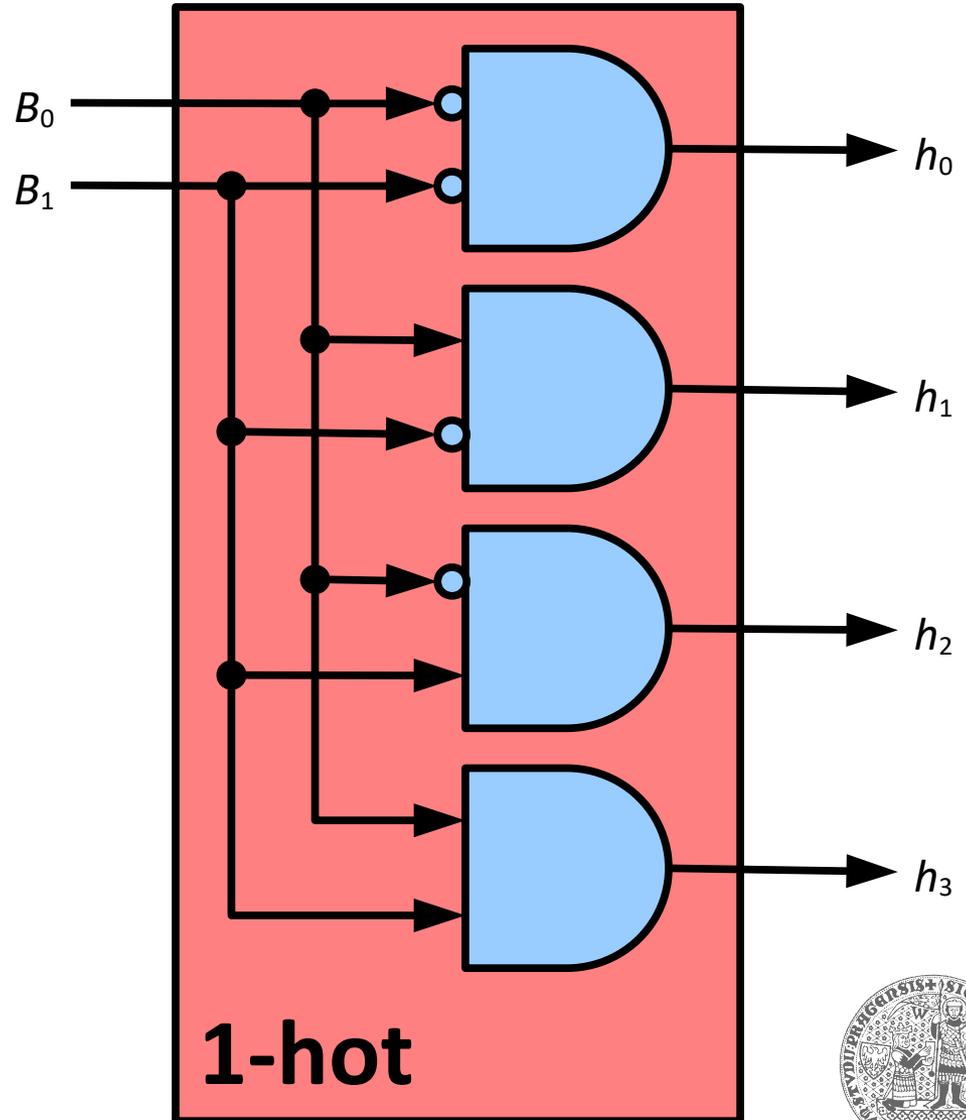
- Activates 1 (selected output) of N outputs
- **Input:** n -bit number $B \in \{0, \dots, 2^n - 1\}$
- **$N=2^n$ outputs:** B -th output logical 1 (hot), other outputs logical 0



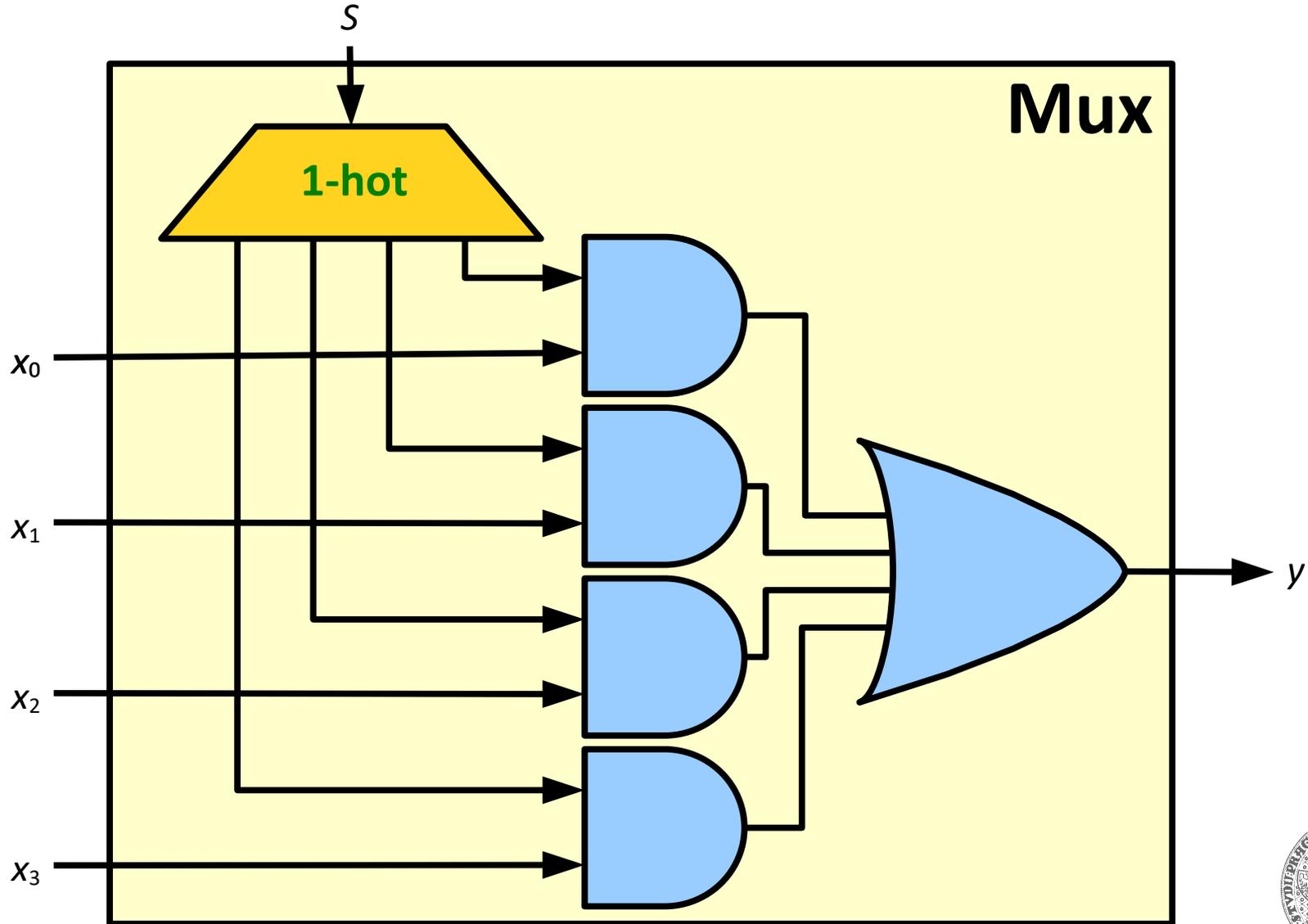
Binary to 1-hot for $N=4$ outputs



Inputs		Outputs			
B_1	B_0	h_3	h_2	h_1	h_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

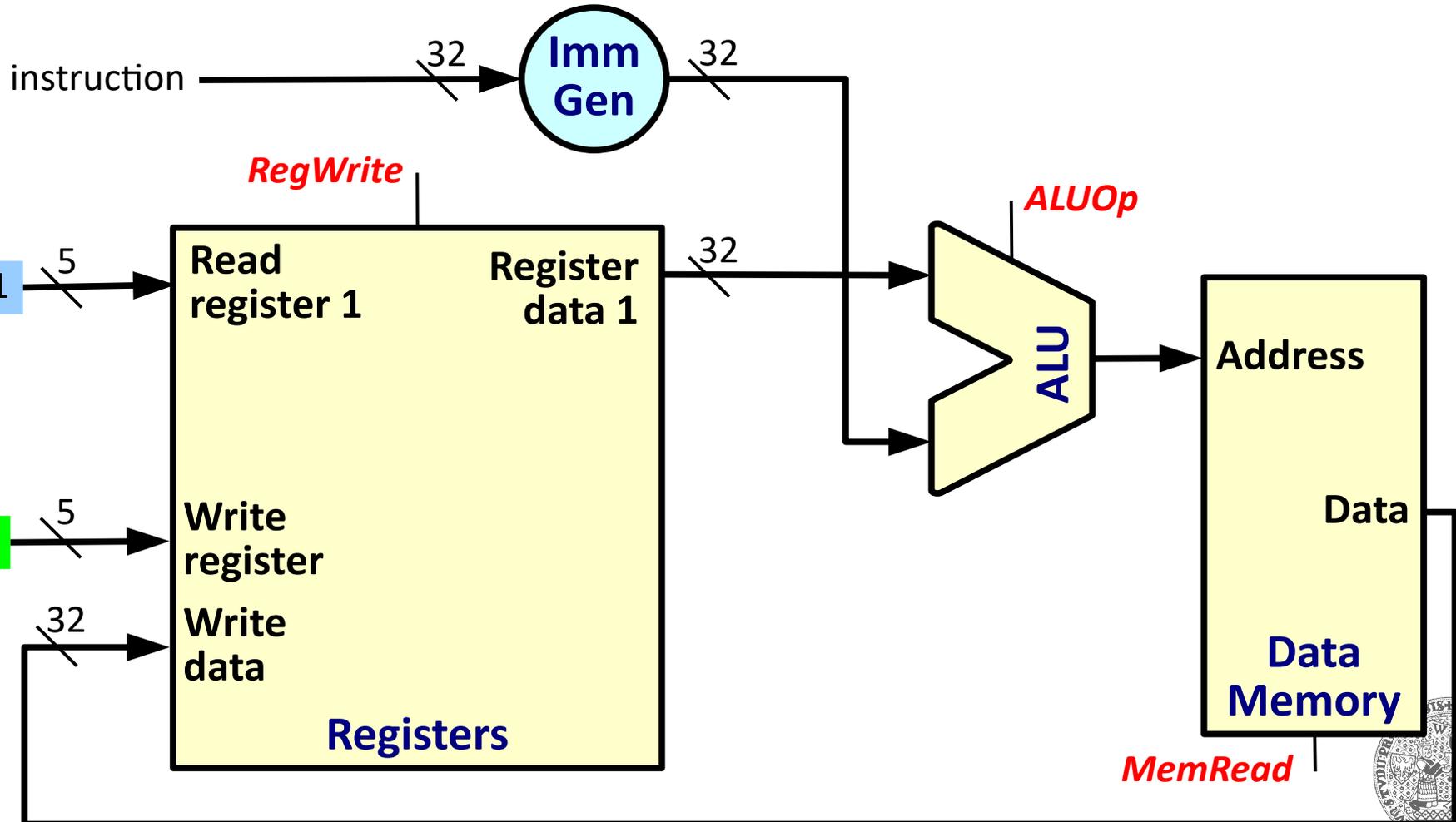


Implementing a multiplexer (4x 1-bit)



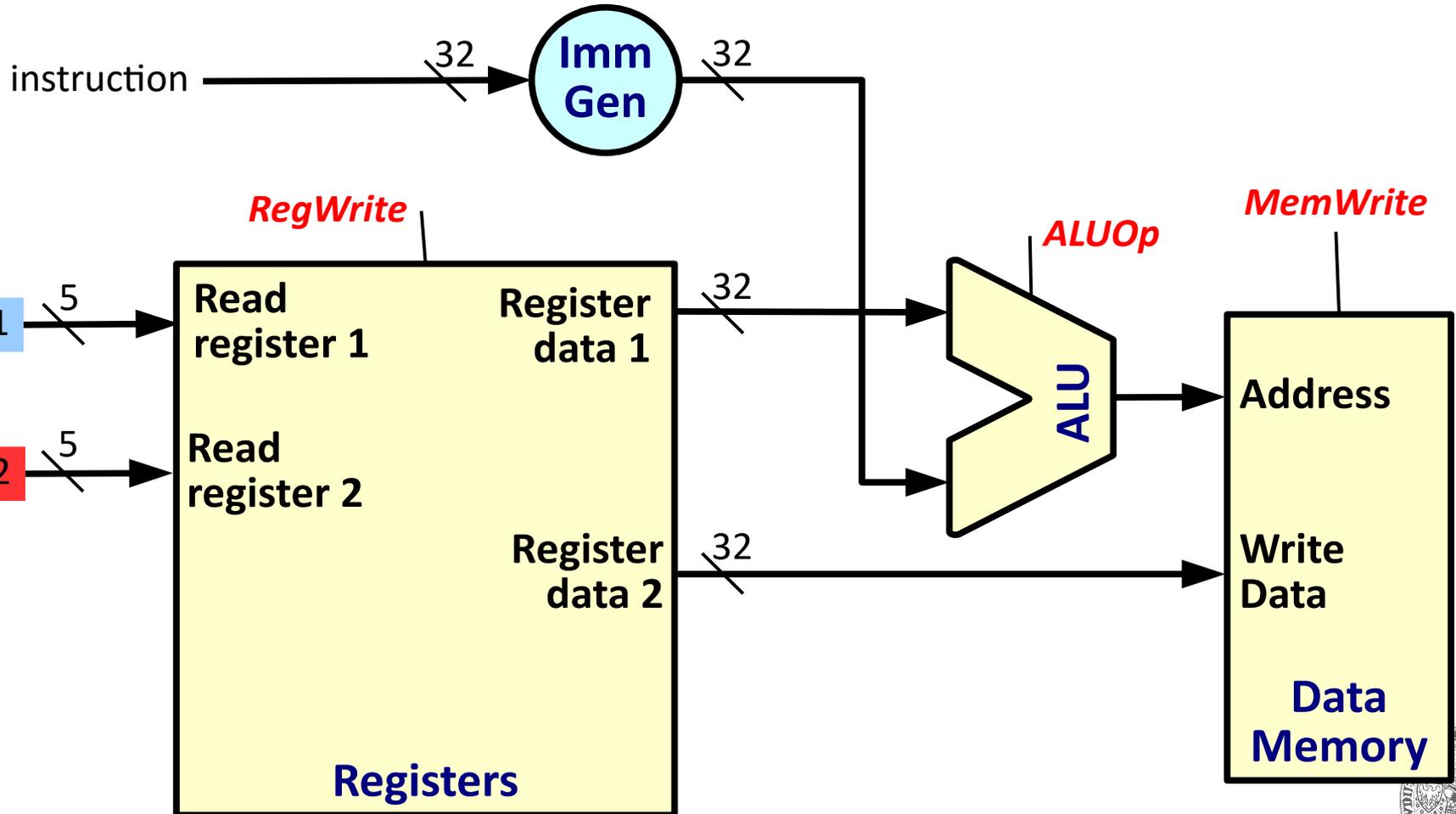
Loading words from memory (lw)

I-type
format

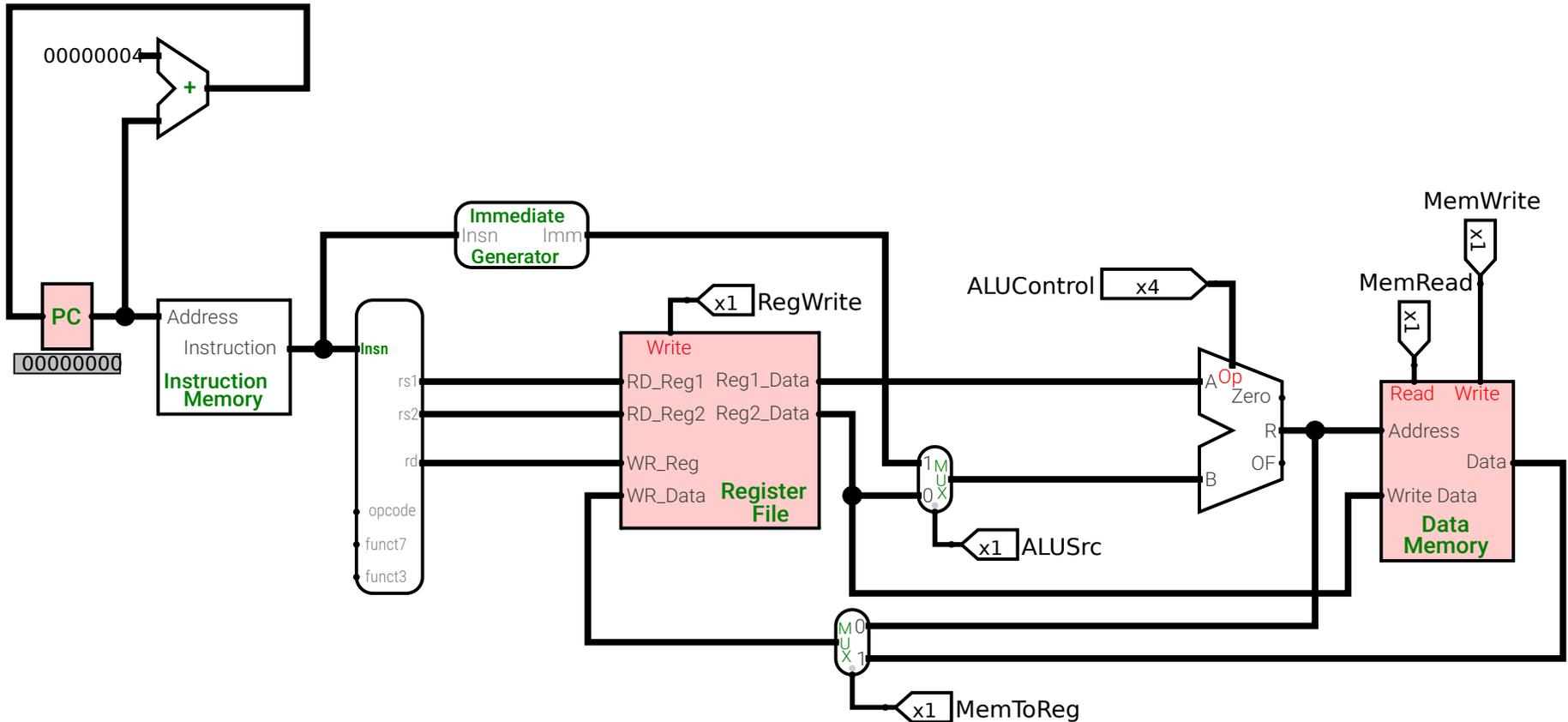


Storing words to memory (sw)

S-type
format

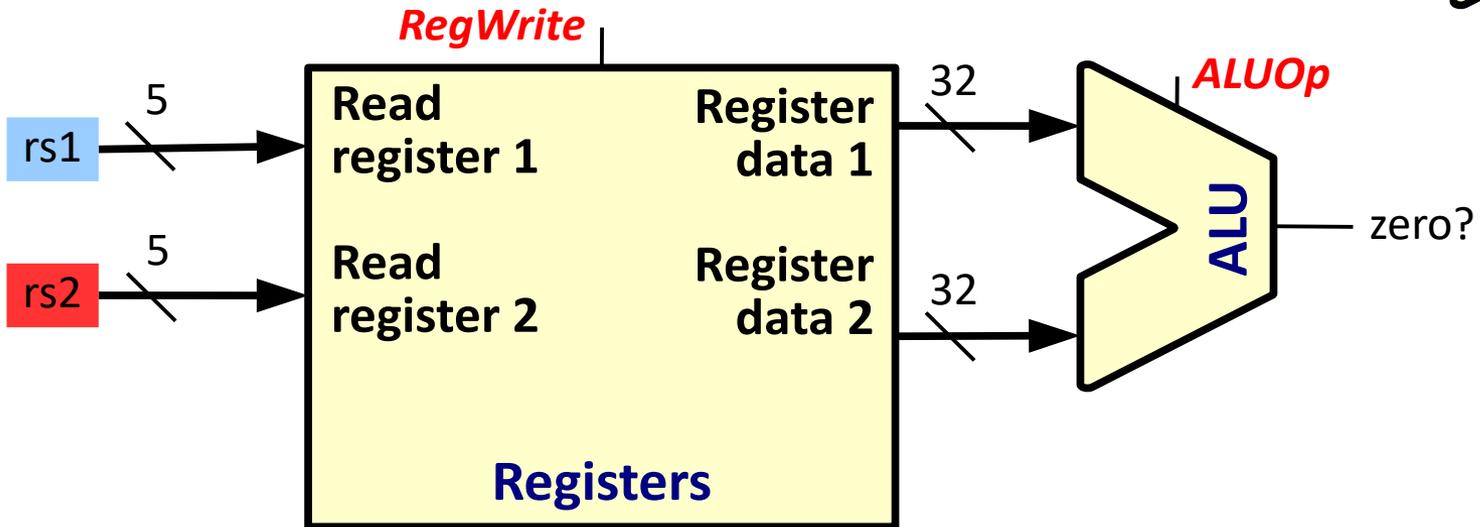
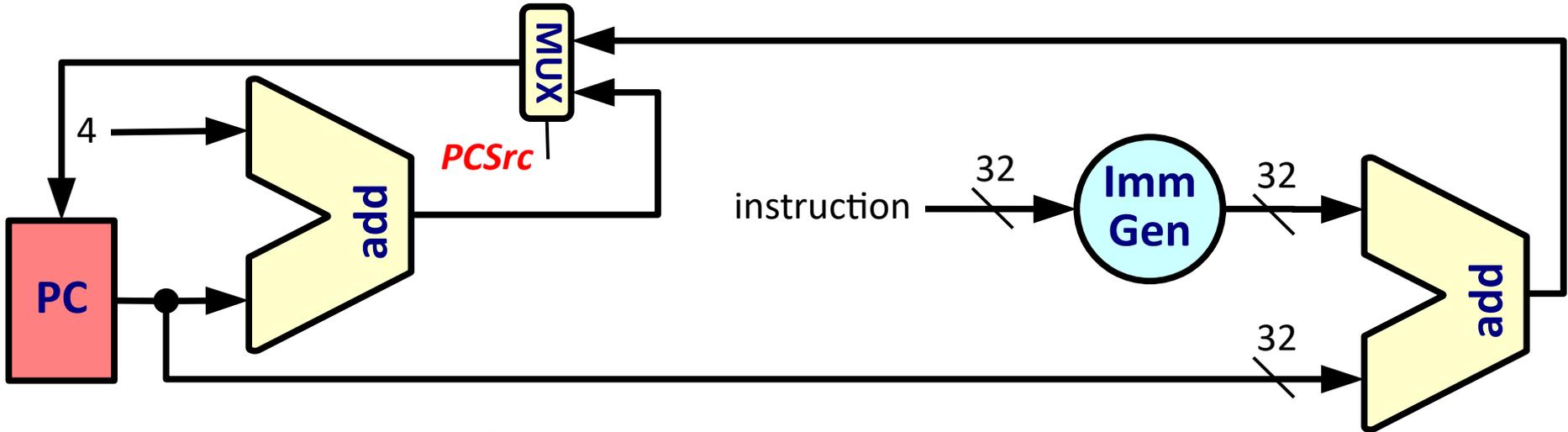


Support for memory access (load/store)

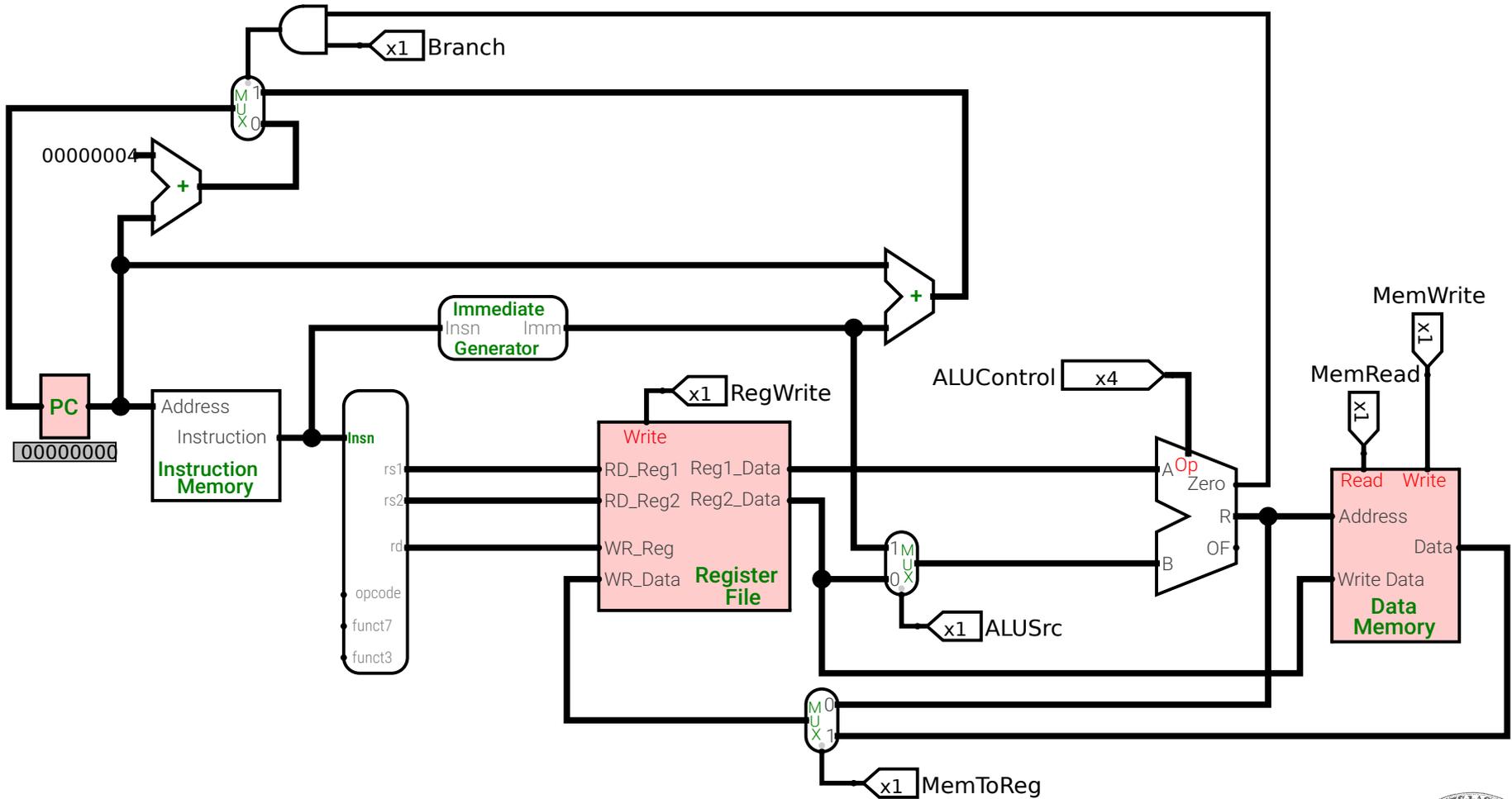


Conditional branch, PC-relative (beq)

SB-type
format

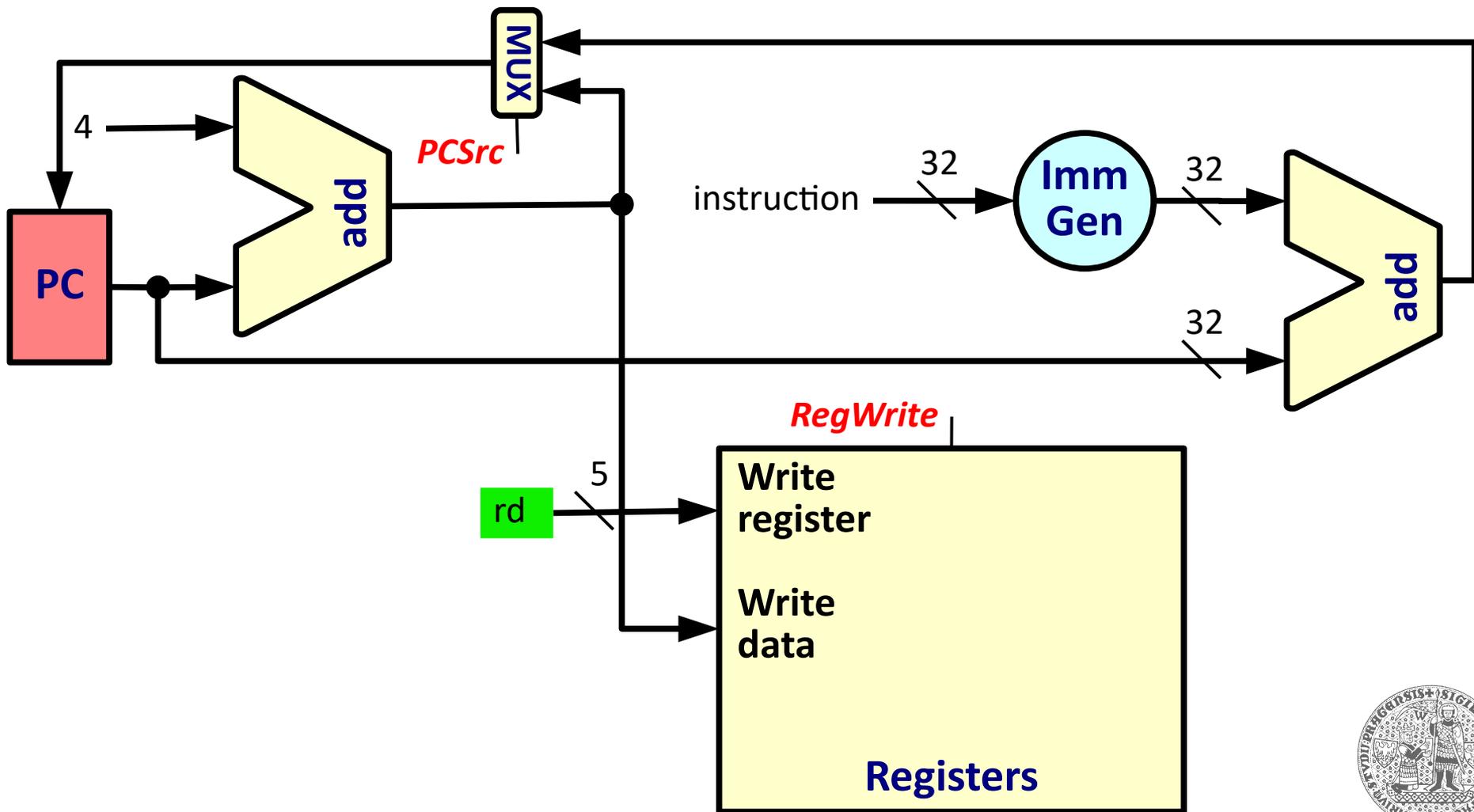
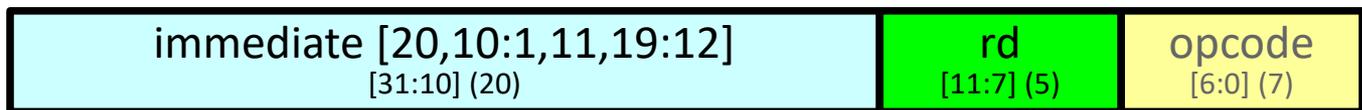


Support for conditional branch



Uncond. jump and link, PC-relative (jal)

UJ-type format

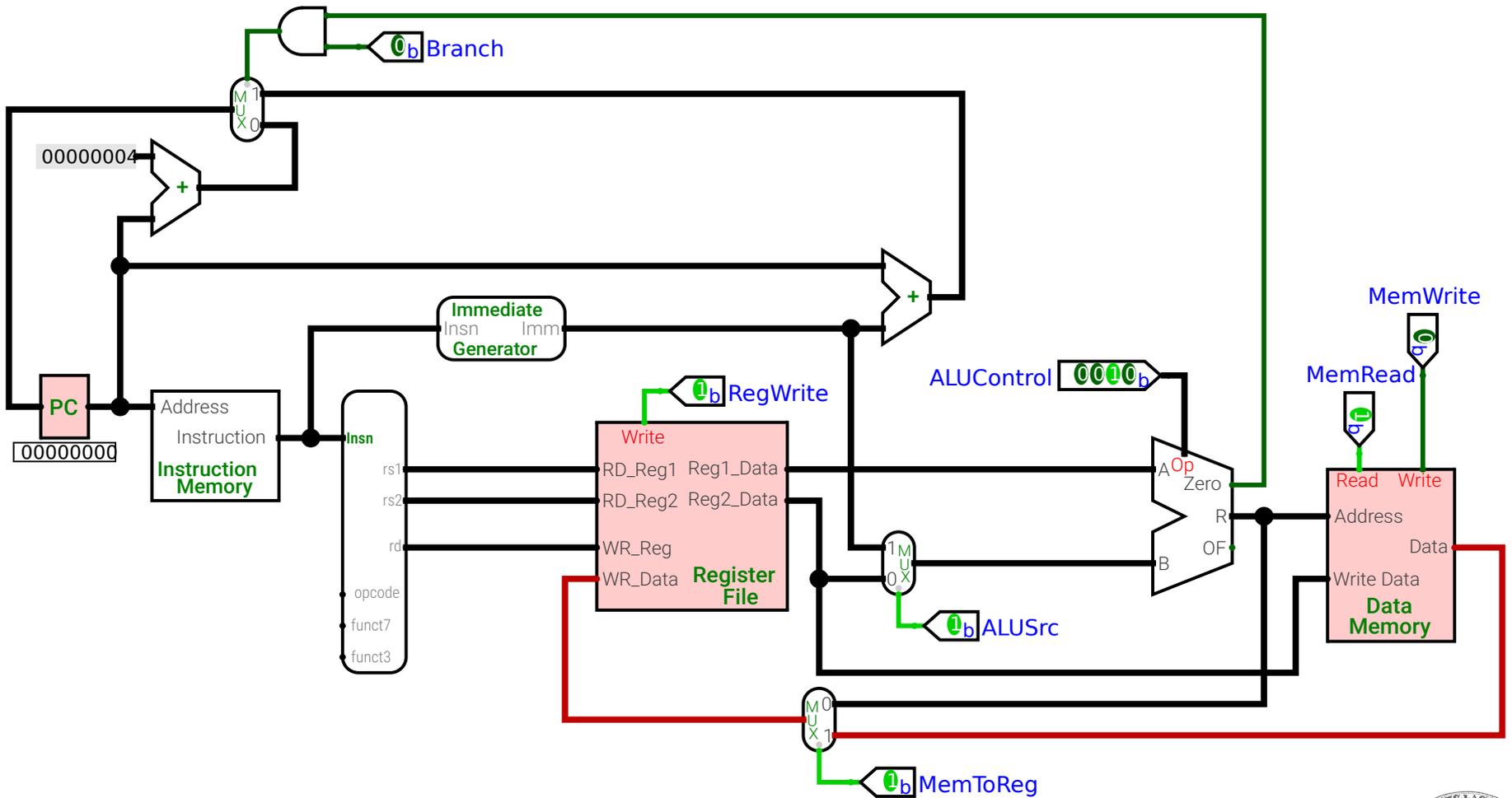


Single-cycle datapath control

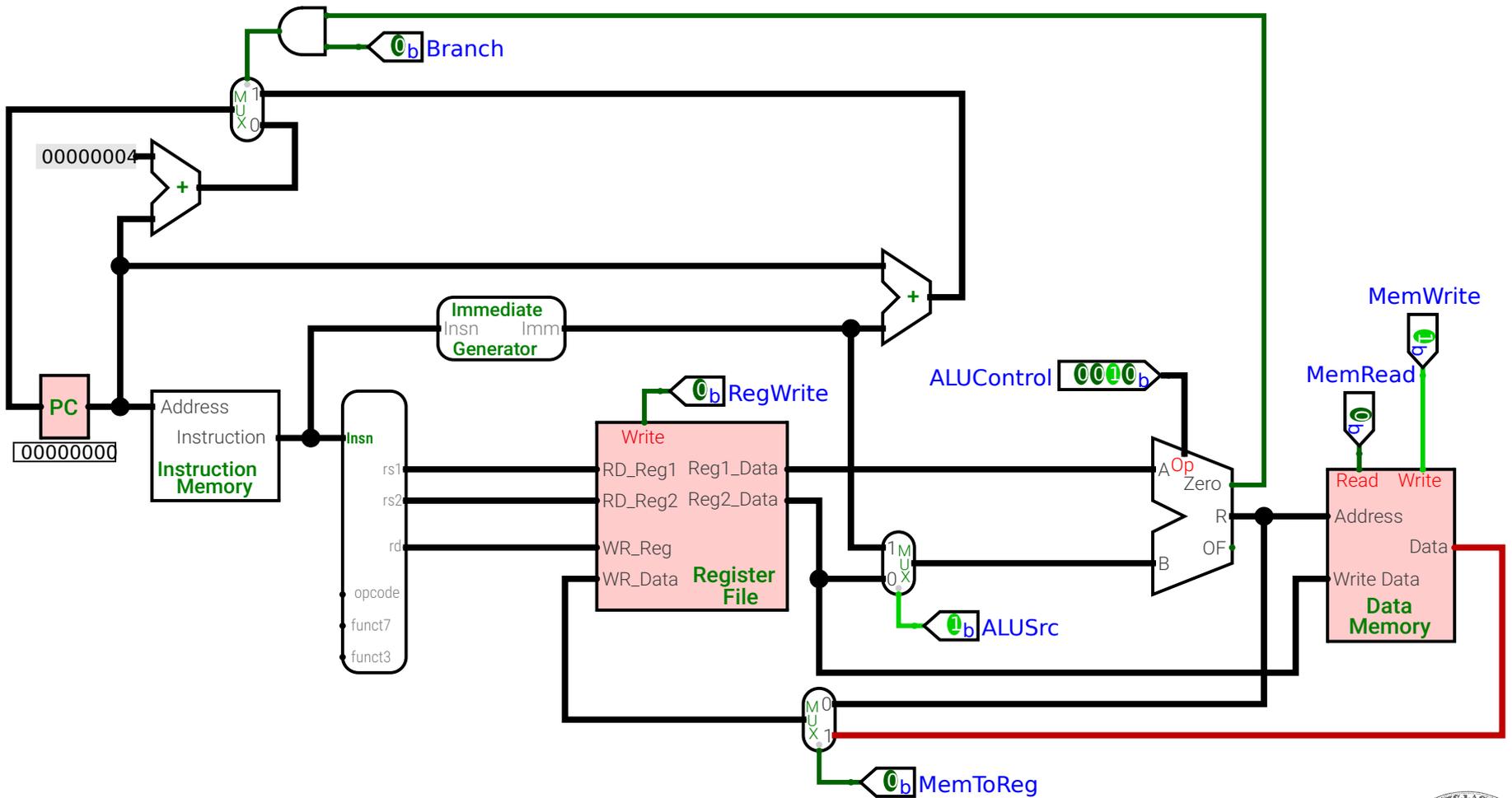
- **Controls the flow of data**
 - Depending on the type of operation
 - Responsible for control signals
 - Source of the next value of PC
 - Write to registers
 - Write to memory
 - ALU operations
 - Mux configuration



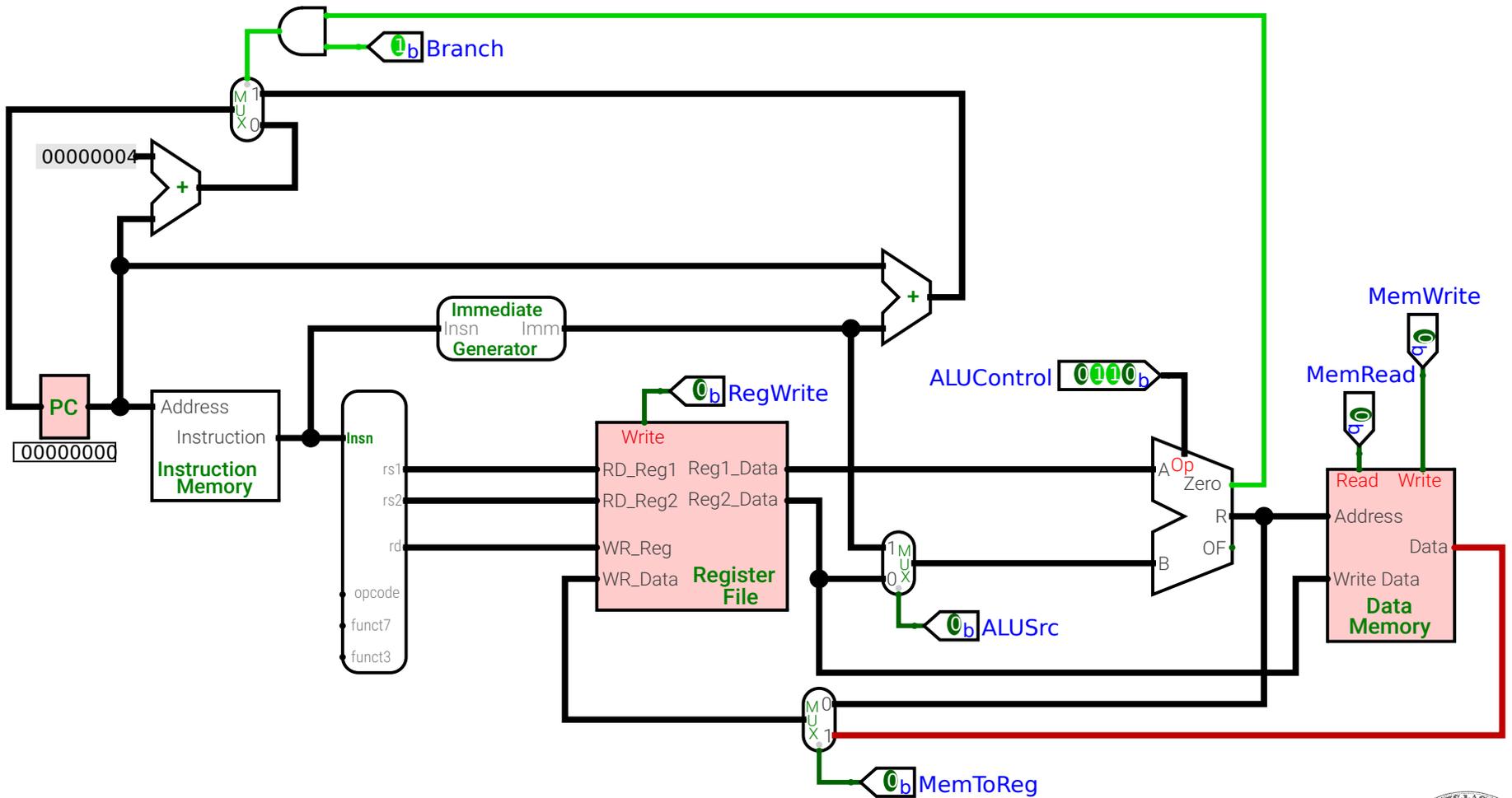
Example: datapath control for *lw*



Example: datapath control for sw



Example: datapath control for *beq*



Datapath controller

- **Responsible for generating control signals**
 - Signal values determined by instruction opcode
 - Some control signals can be directly embedded in the instruction word

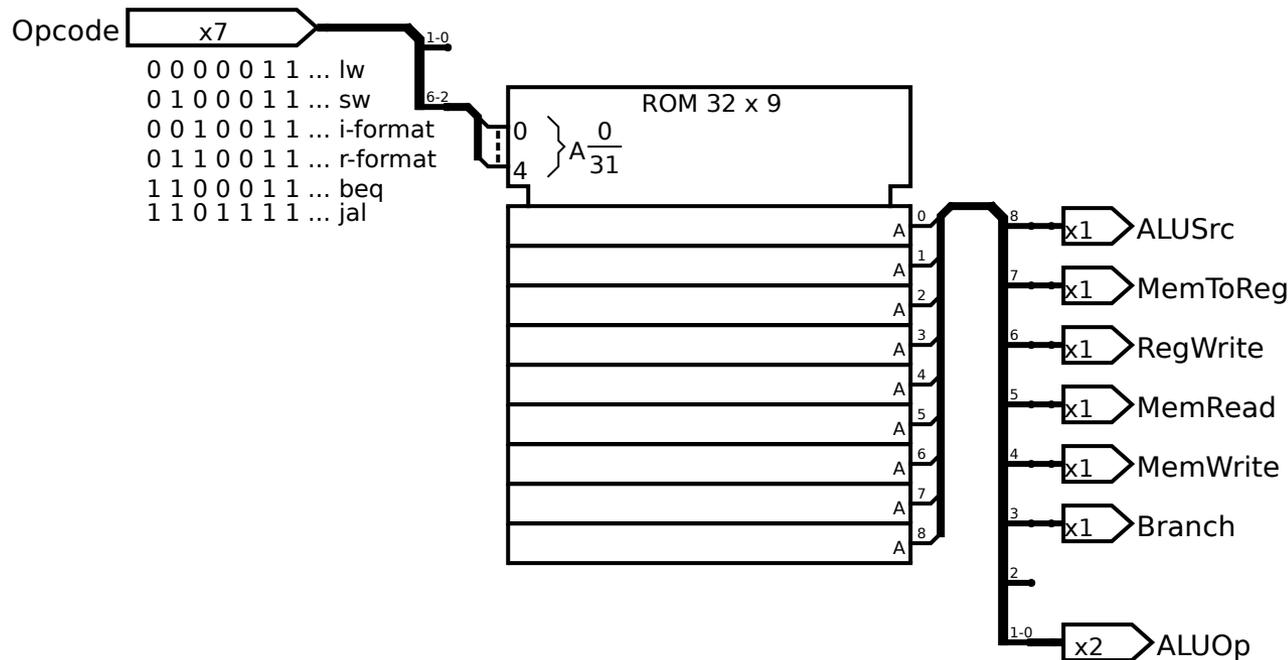
opcode	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp
R-format	0	0	1	0	0	0	10 (func)
I-format	1	0	1	0	0	0	10 (func)
lw	1	1	1	1	0	0	00 (add)
sw	1	?	0	0	1	0	00 (add)
beq	0	?	0	0	0	1	01 (sub)



ROM-based controller

- **Signal values stored in read-only memory**

- Each word contains the values of all control signals
- Words addressed by the opcode



ROM-based controller (2)

- **Implementation issues**

- Making ROM faster than the datapath

- **Real MIPS**

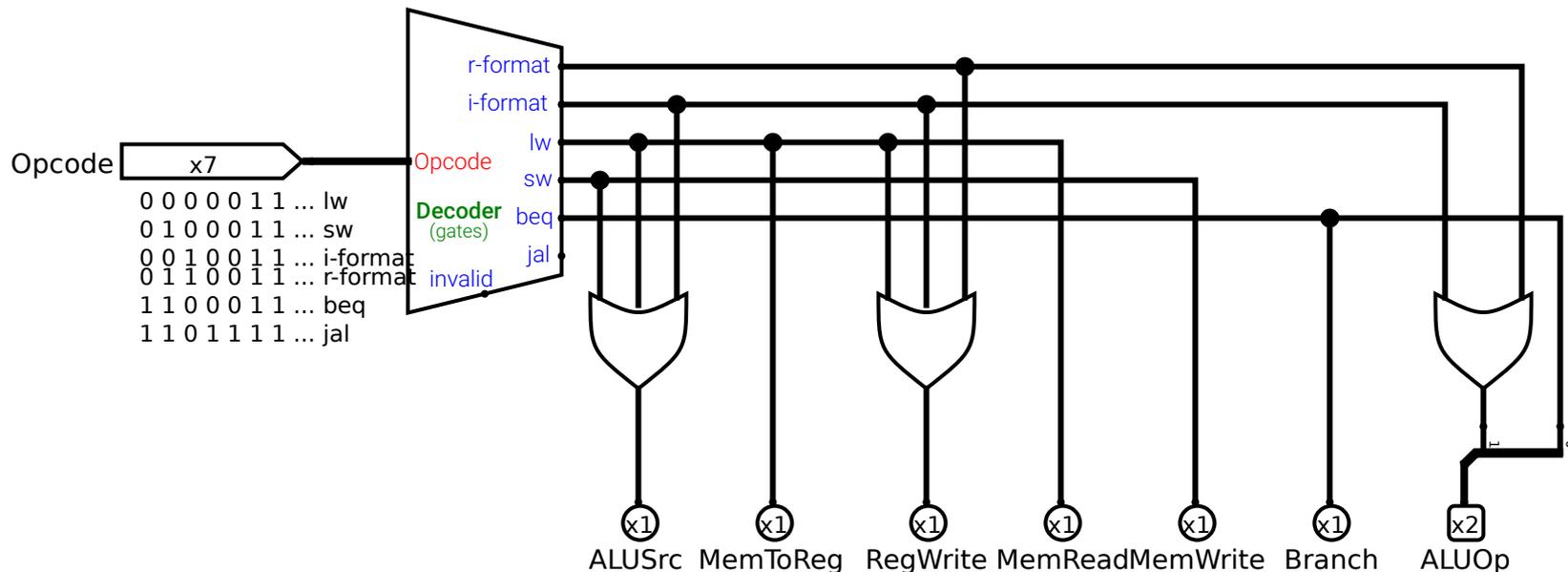
- Approx. 100 instructions and 300 control signals
 - Control ROM capacity needed: 30000 bits (~ 4 KB)
- Real RISC-V at least as complex
 - Optional extensions can add significant complexity



Logic-based controller (combinational)

- **Faster alternative to ROM**

- Observation: only a few control signals need to be set to one (zero) at the same time
- Contents of ROM can be efficiently expressed using logic functions

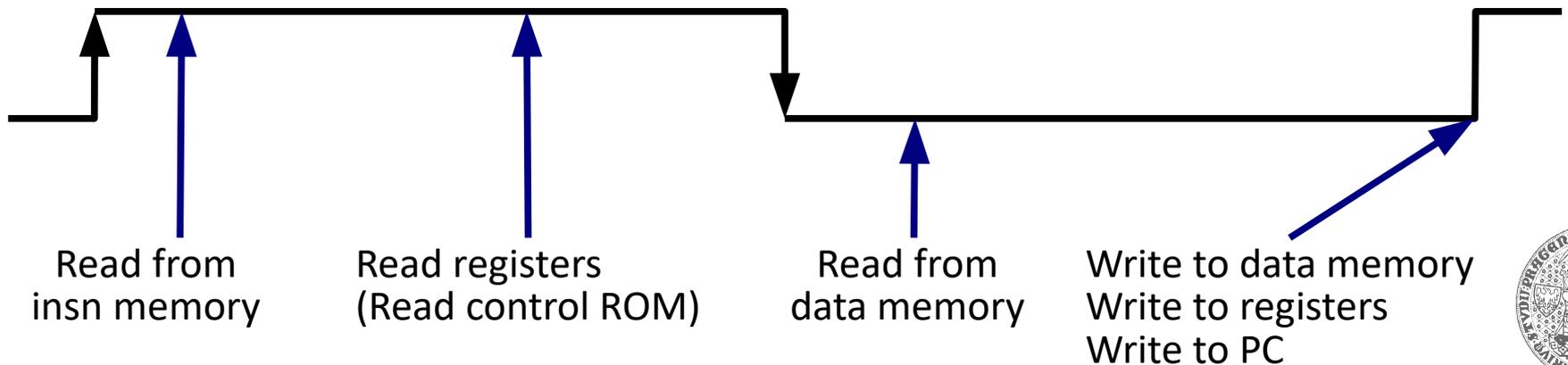


Instruction cycle

- **Datapath with continuous read**

- No problem in our design

- Writes (PC, RF, DM) are independent
- No read follows write in the instruction cycle
- Instruction fetch does not need control
 - After instruction is read, the controller decodes instruction opcode into control signals for the rest of the datapath
 - When PC changes, datapath starts processing another instruction



Single-cycle processor performance

- **Each instruction executed in 1 cycle (CPI=1)**
 - Single-cycle controller (control ROM or a combinational logic block)
 - Generally lower clock frequency
 - Clock period respects the “longest” instruction
 - Load Word (lw) in our case
 - Usually multiplication, division, or floating point ops
 - Datapath contains duplicate elements
 - Instruction and data memory, two extra adders



Multi-cycle datapath

- **Basic idea**

- Simple instructions should not take as much time to execute as the complex ones

- **Variable instruction execution time**

- Clock period is constant (cannot be changed dynamically), we need a „digital“ solution
- We can make clock faster (shorter period) and split instruction execution into multiple stages
 - Clock period corresponds to **one execution stage**
 - Fixed **machine cycle** (clock period)
 - Variable **instruction cycle**



Example: multi-cycle CPU performance

- **Rough estimate, assuming the following**
 - Simple instructions take 10 ns to execute
 - Multiplication takes 40 ns
 - Instruction mix with 10% of multiplications
- **Single-cycle datapath**
 - Clock period 40 ns, CPI=1 → **25 MIPS**
- **Multi-cycle datapath**
 - Clock period 10 ns, 13 ns per instruction (average)
 - CPI=1.3 → **77 MIPS** (3x improvement)



Multi-cycle datapath (2)

- **Instruction cycle**

1. Read instruction from memory
2. Decode instruction, read registers, compute branch target address
3. Execute register operation / compute address for memory access / finish branch or jump
4. Write register operation results / access memory
5. Finish load from memory



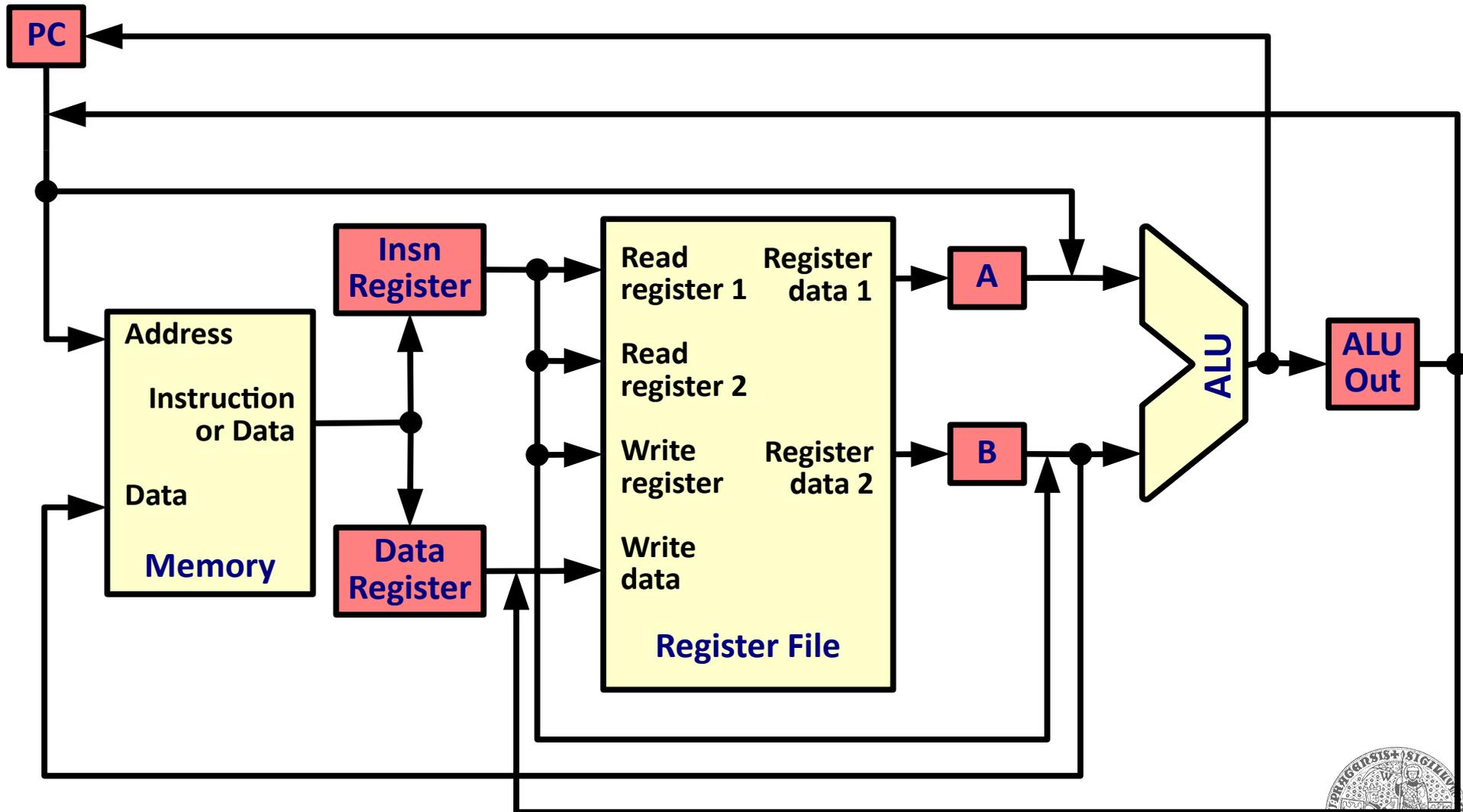
Multi-cycle datapath (3)

- **Implementation issues**

- Instruction execution split to stages
 - Need to isolate stages using latch registers to “remember” results from previous stage
- Need to keep track of stages
 - Different sequences for different instruction types
 - Some instructions may skip stages and finish early
 - Controller needs to remember state → sequential logic



Multi-cycle datapath (4)



Stage 1: Instruction Read

- **Common for all instructions**

- $IR \leftarrow \text{Memory}[PC]$

- Read instruction into Instruction Register
- Memory is used for both instruction and data access
- Need to “remember” the instruction being executed

- $ALUOut \leftarrow PC + 4$

- Compute the address of next instruction
- Do not update PC before computing branch target



Stage 2: Instruction Decode, Read Regs.

- **Common for all instructions**

- $A \leftarrow \text{Reg}[\text{IR.rs1}]$

- Read contents of source register 1
- Store value into latch A for next stage

- $B \leftarrow \text{Reg}[\text{IR.rs2}]$

- Read contents of source register 2
- Store value into latch B for next stage

- $\text{ALUOut} \leftarrow \text{PC} + \text{Immediate}$

- Compute branch/jump target (type-B/type-J)
- Relative to current instruction (before updating PC)
- Remains unused if not a branch/jump

- $\text{PC} \leftarrow \text{ALUOut} (\text{PC} + 4)$

- Advance PC to next instruction. This will not change the instruction being executed: it was stored in the Instruction Register



Stage 3: Execute / address calc.

● Branch instruction (*finish*)

- $(A == B) \Rightarrow PC \leftarrow \text{ALUOut}$ (branch target)
 - Branch target in ALUOut from previous stage

● Jal instruction (*finish*)

- $\text{Reg}[\text{IR.rd}] \leftarrow PC + 4$ (next instruction)
- $PC \leftarrow \text{ALUOut}$ (jump target)
 - Jump target in ALUOut from previous stage

● Register operation

- $\text{ALUOut} \leftarrow A \text{ *funct* } B$, or alternatively
- $\text{ALUOut} \leftarrow A \text{ *funct* } \text{Immediate}$ (type-I, type-U)

● Memory access

- $\text{ALUOut} \leftarrow A + \text{Immediate}$ (type-I, type-S)
 - Compute address for memory access



Stage 4: Write Results / memory access

- **Register operation (*finish*)**
 - $\text{Reg}[\text{IR.rd}] \leftarrow \text{ALUOut}$
 - Result in ALUOut (from previous stage)
- **Write to memory (*finish*)**
 - $\text{Memory}[\text{ALUOut}] \leftarrow B$
 - Address in ALUOut (from previous stage)
- **Read from memory**
 - $\text{DR} \leftarrow \text{Memory}[\text{ALUOut}]$
 - Address in ALUOut (from previous stage)
 - Store data into latch DR for next stage

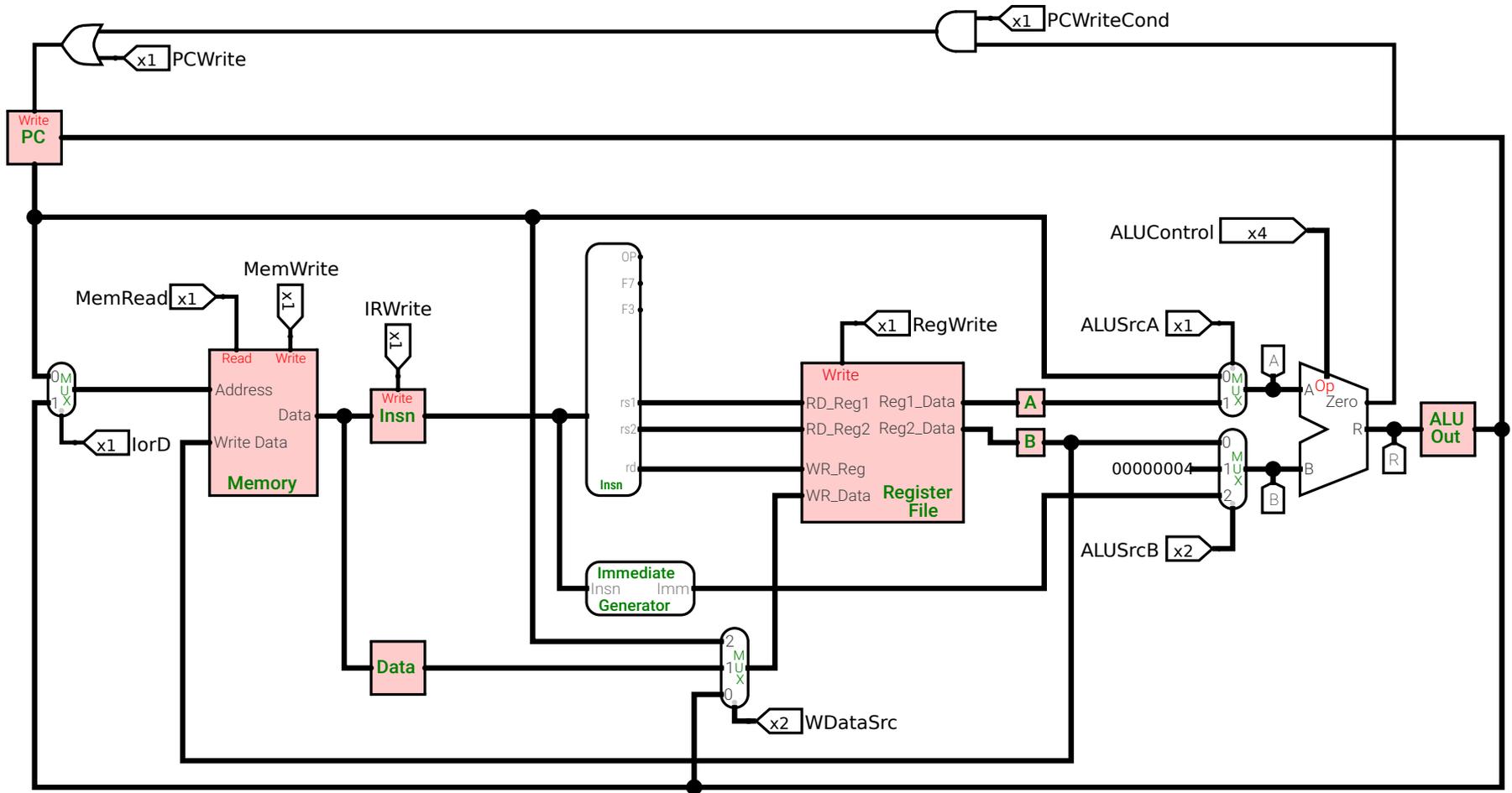


Stage 5: Finish reading from memory

- **Read from memory (*finish*)**
 - $\text{Reg}[\text{IR.rd}] \leftarrow \text{DR}$
 - Value stored in DR (from previous stage)



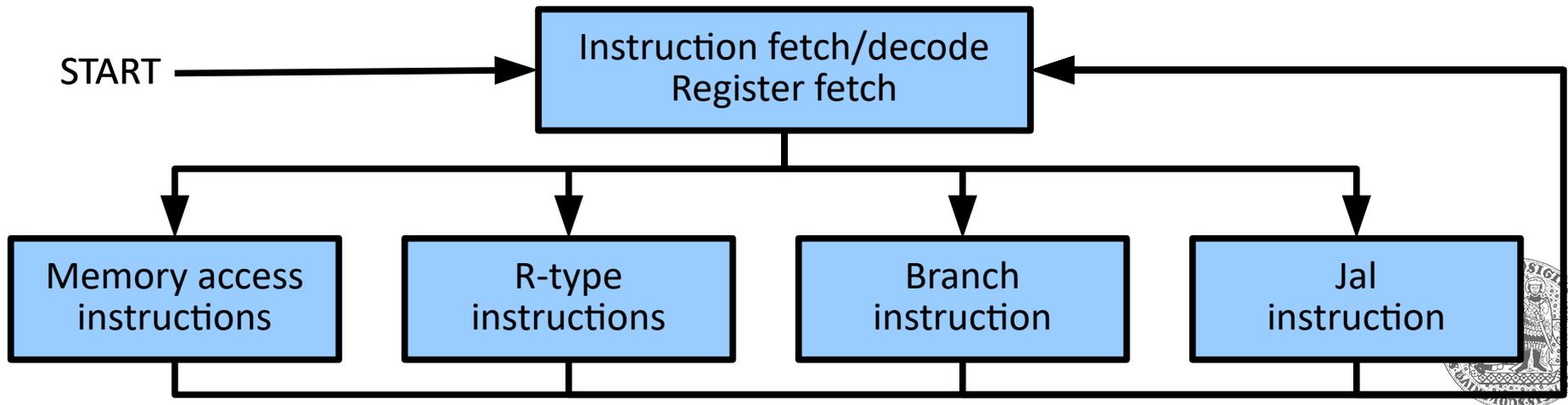
Multi-cycle datapath implementation



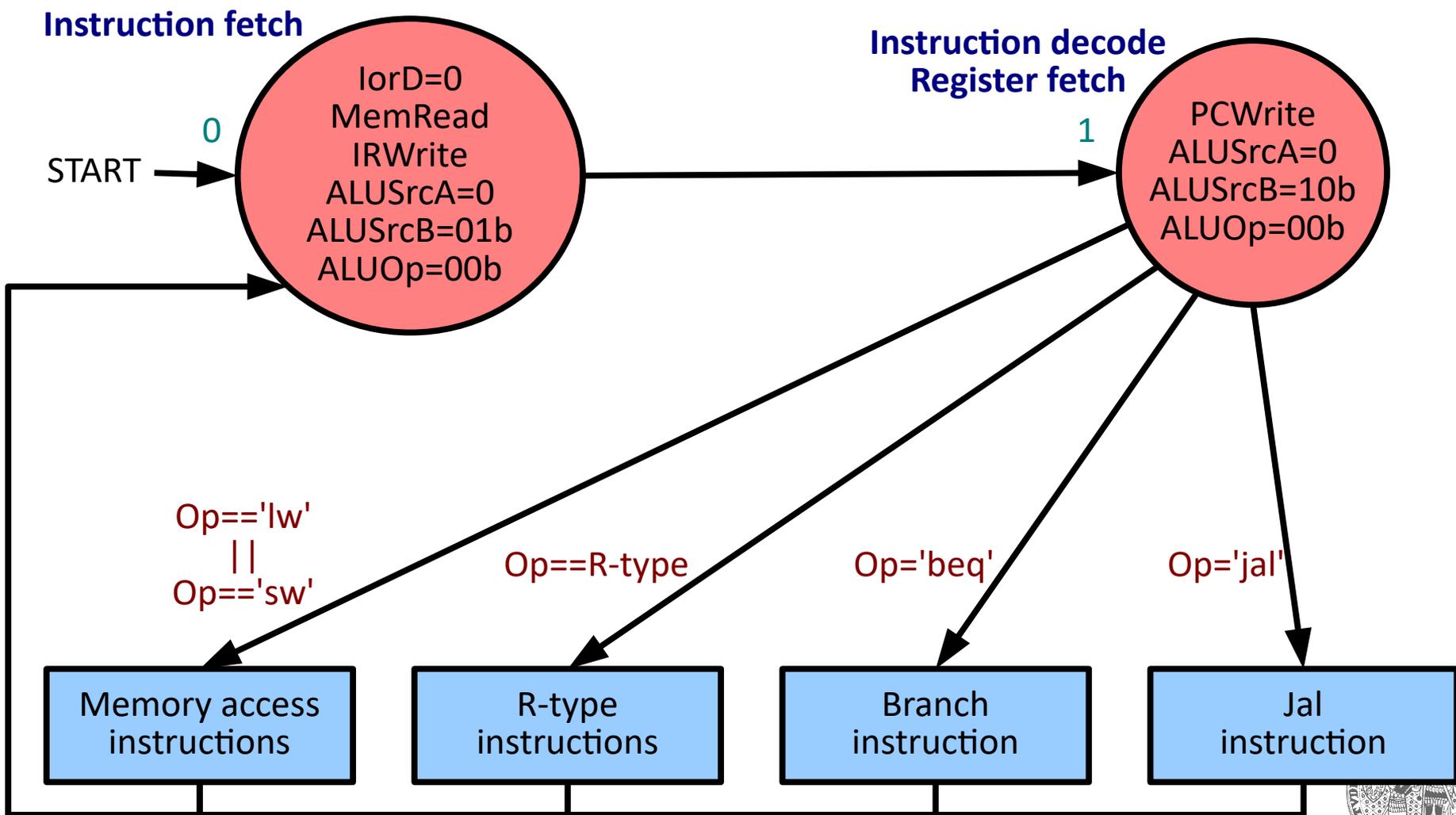
Multi-cycle datapath control

- **Sequential process**

- Instructions executed in multiple cycles
- Controller is a sequential circuit (automaton)
 - Current state stored in a state register
 - Combinational block determines next state
 - Depends on current state and instruction being executed
 - Updated on rising edge of the clock signal

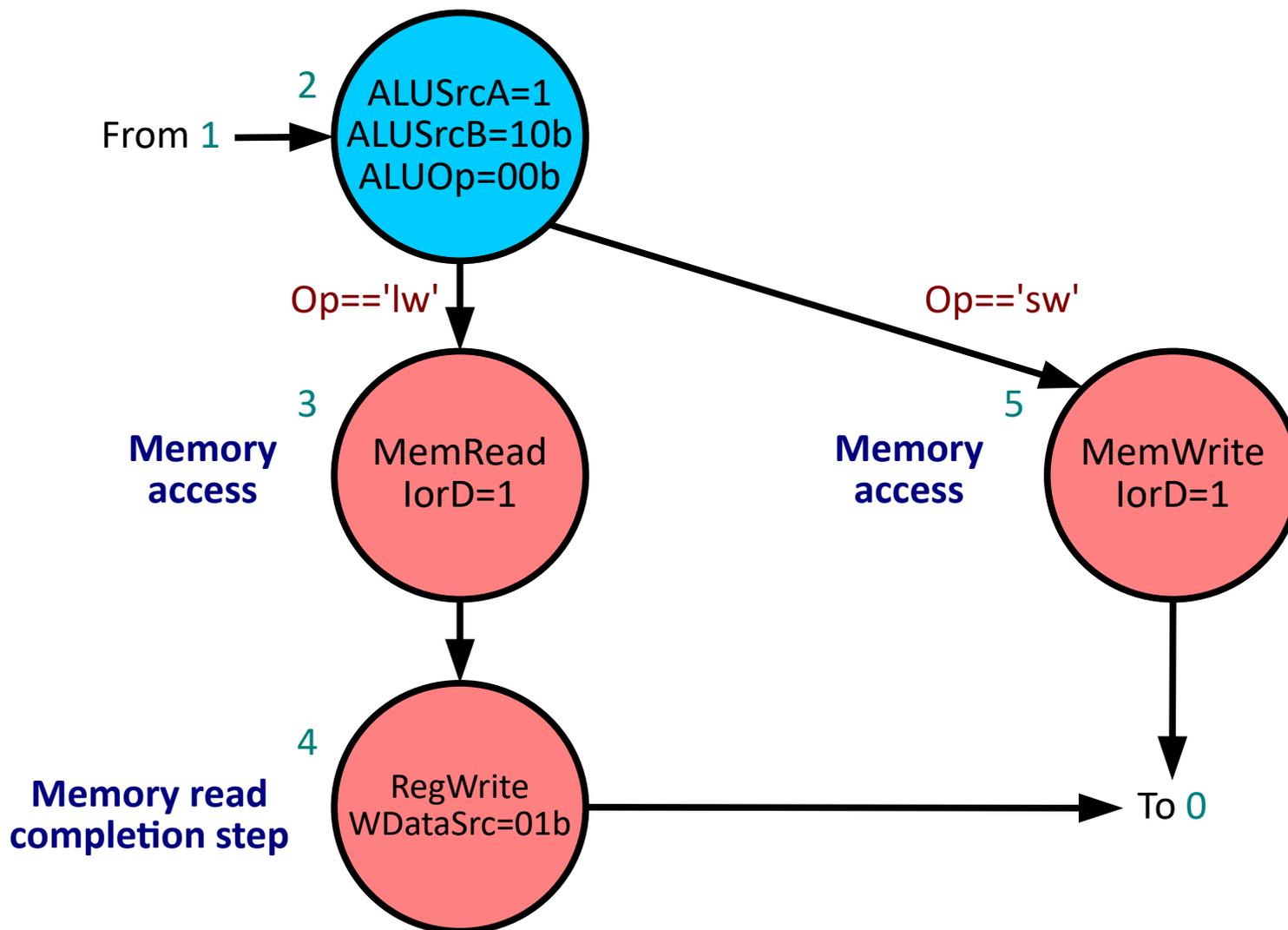


Instruction fetch/decode, Register fetch

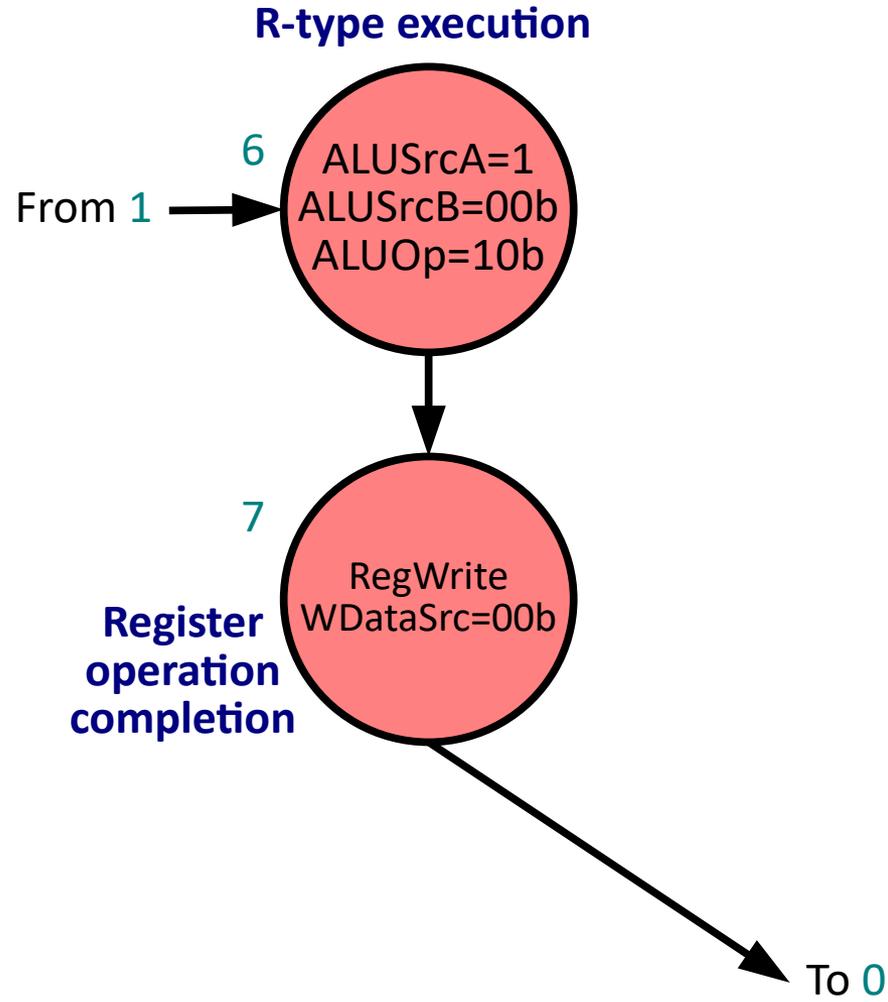


Memory access instructions

Memory address computation



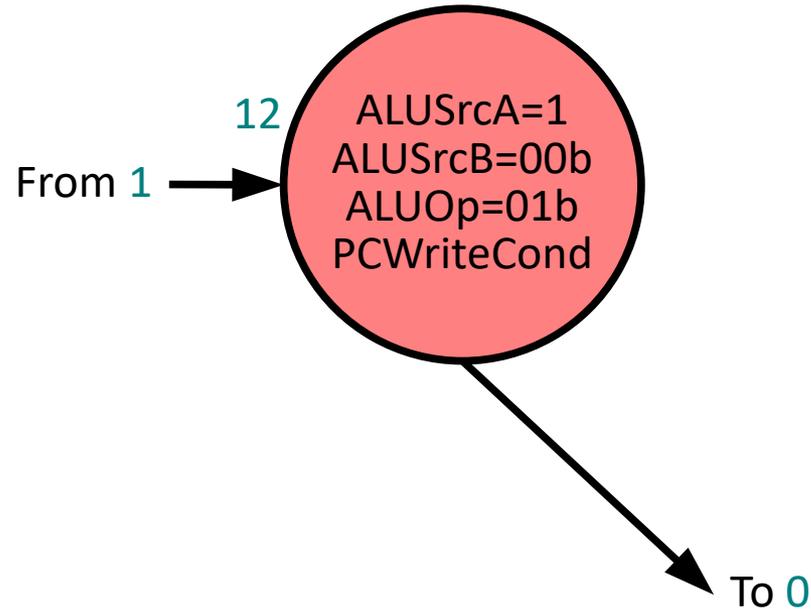
R-type instructions



Branch instruction



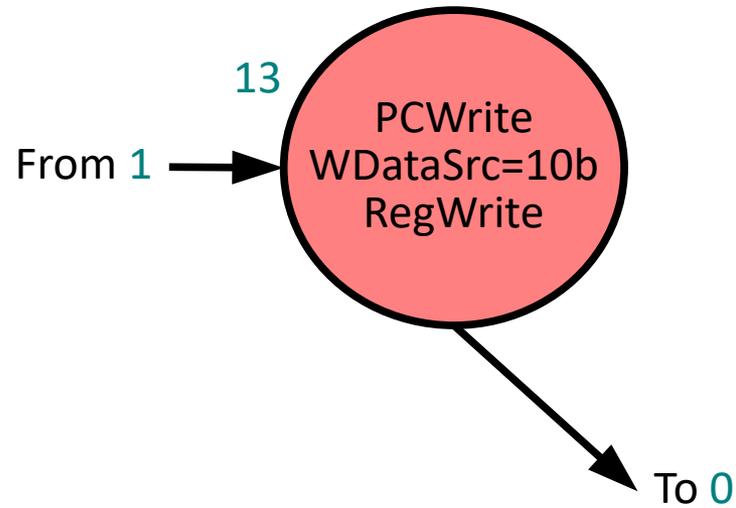
Branch completion



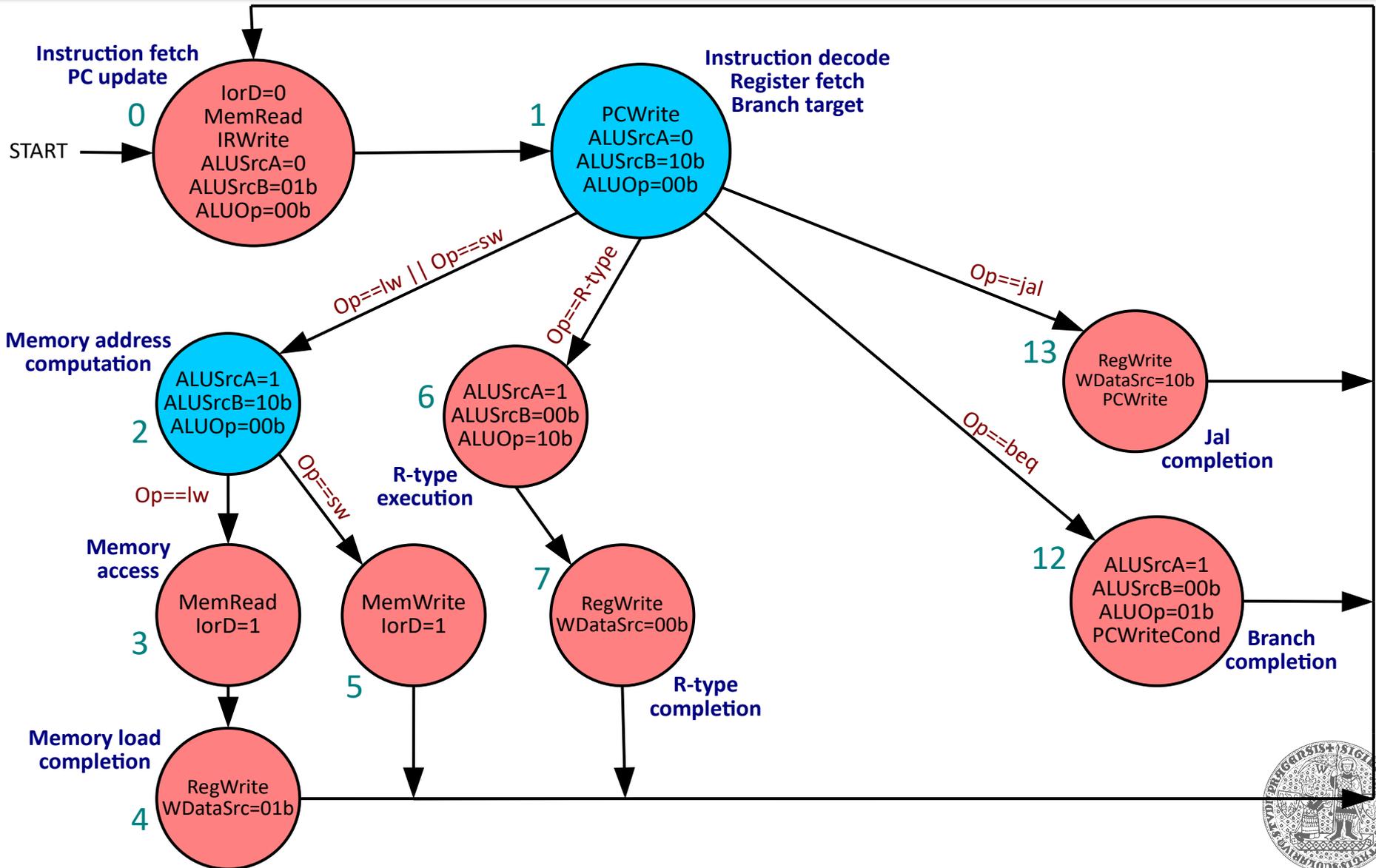
Jal instruction



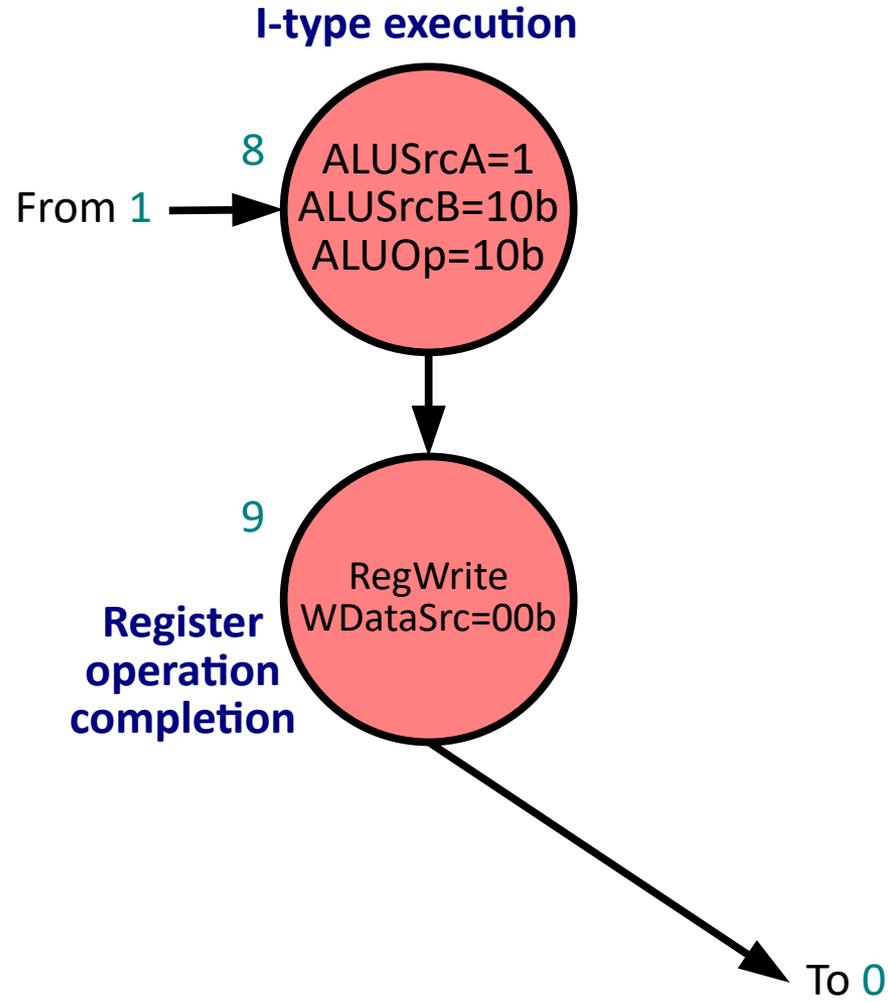
Jal completion



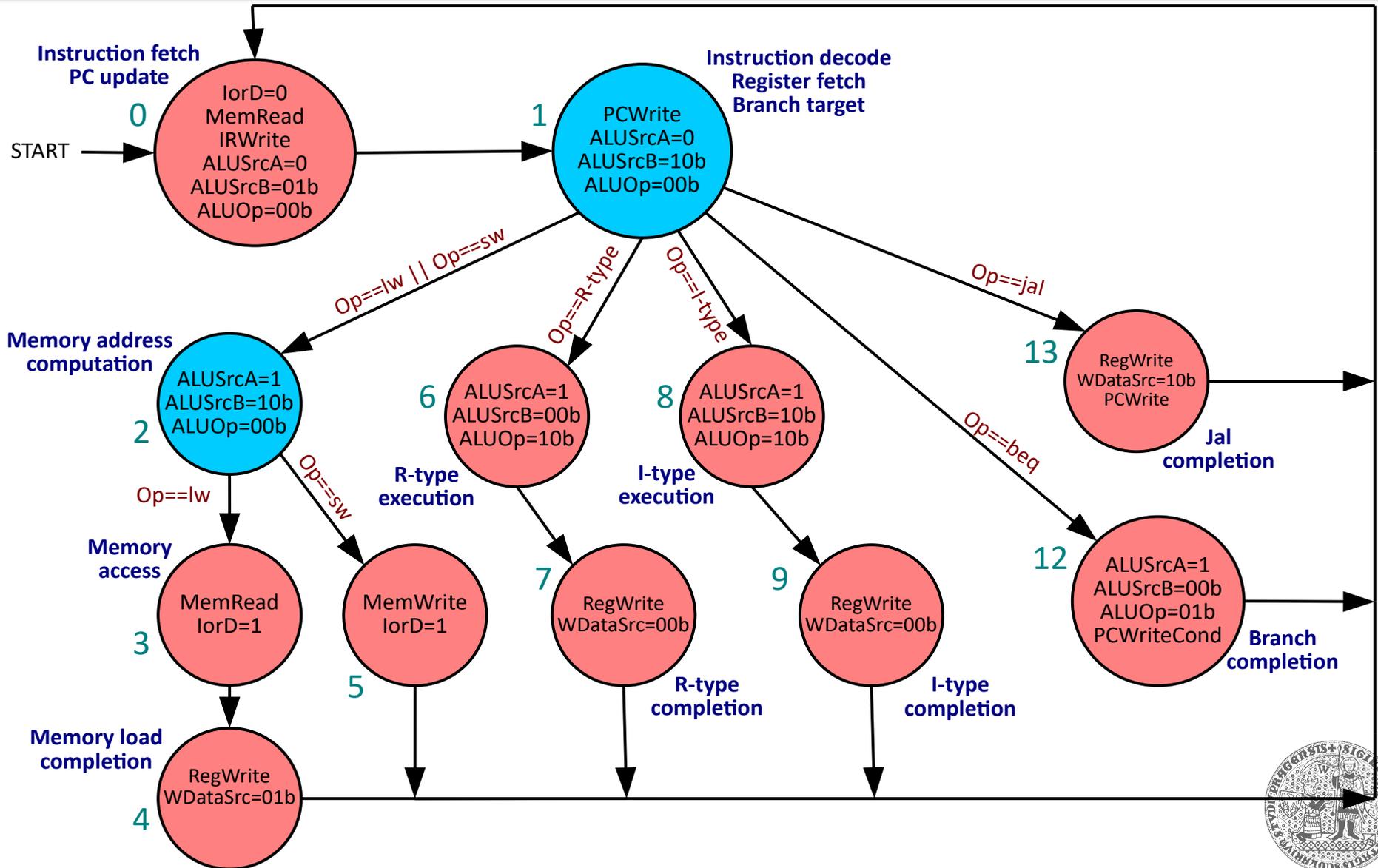
Multi-cycle datapath control (2)



I-type instructions



Multi-cycle datapath control (3)



Flow of instructions

- **Normal/expected flow**

- Sequential: common code operating on data
- Non-sequential: branches and jumps

- **Unexpected flow**

- Internal (*Exception/Trap*)
 - **Invalid instruction**
 - Arithmetic overflow, division by zero (not on RISC-V)
 - Unauthorized access to memory
 - Requesting service from operating system (system call)
 - Hardware failure
- External (*Interrupt*)
 - Request for “attention” from an I/O device
 - Hardware failure

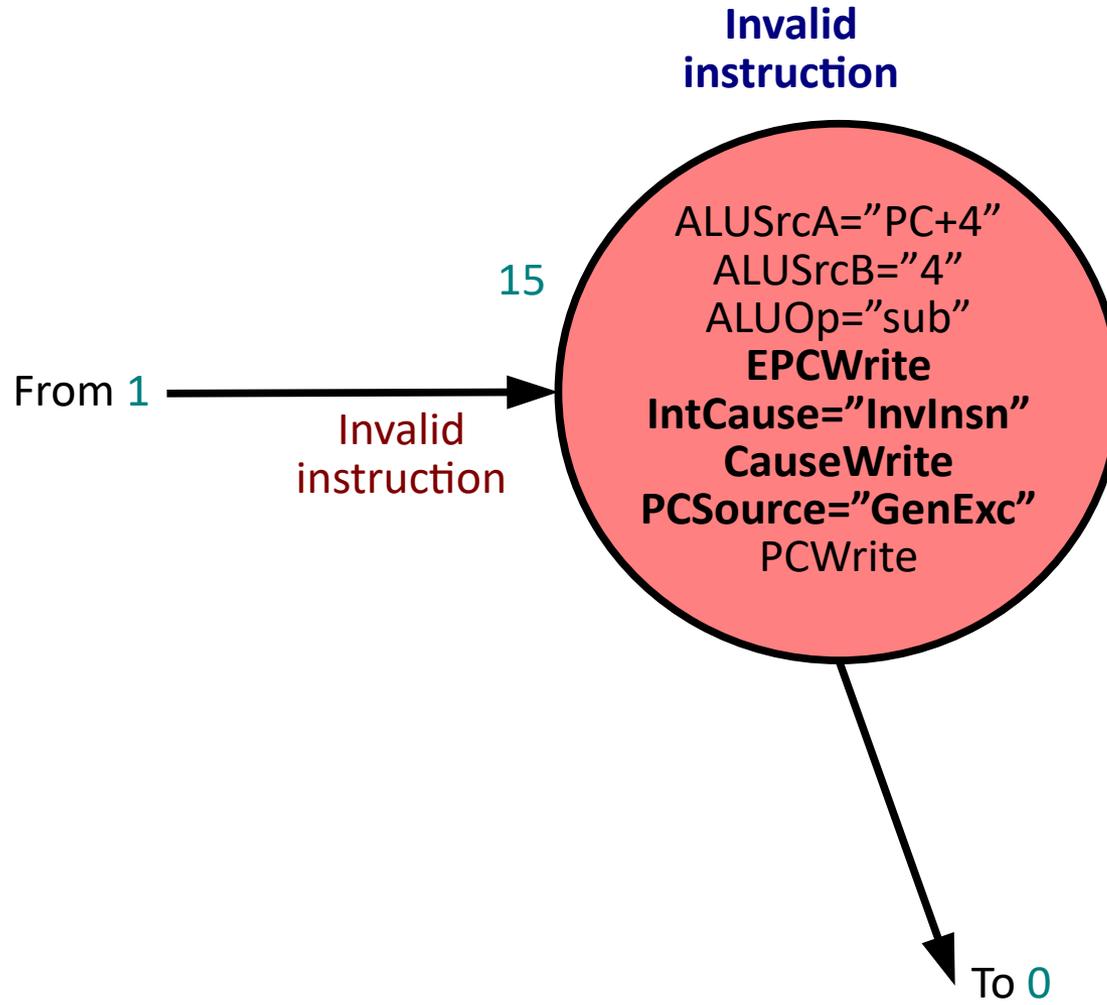


Supporting exceptions and interrupts

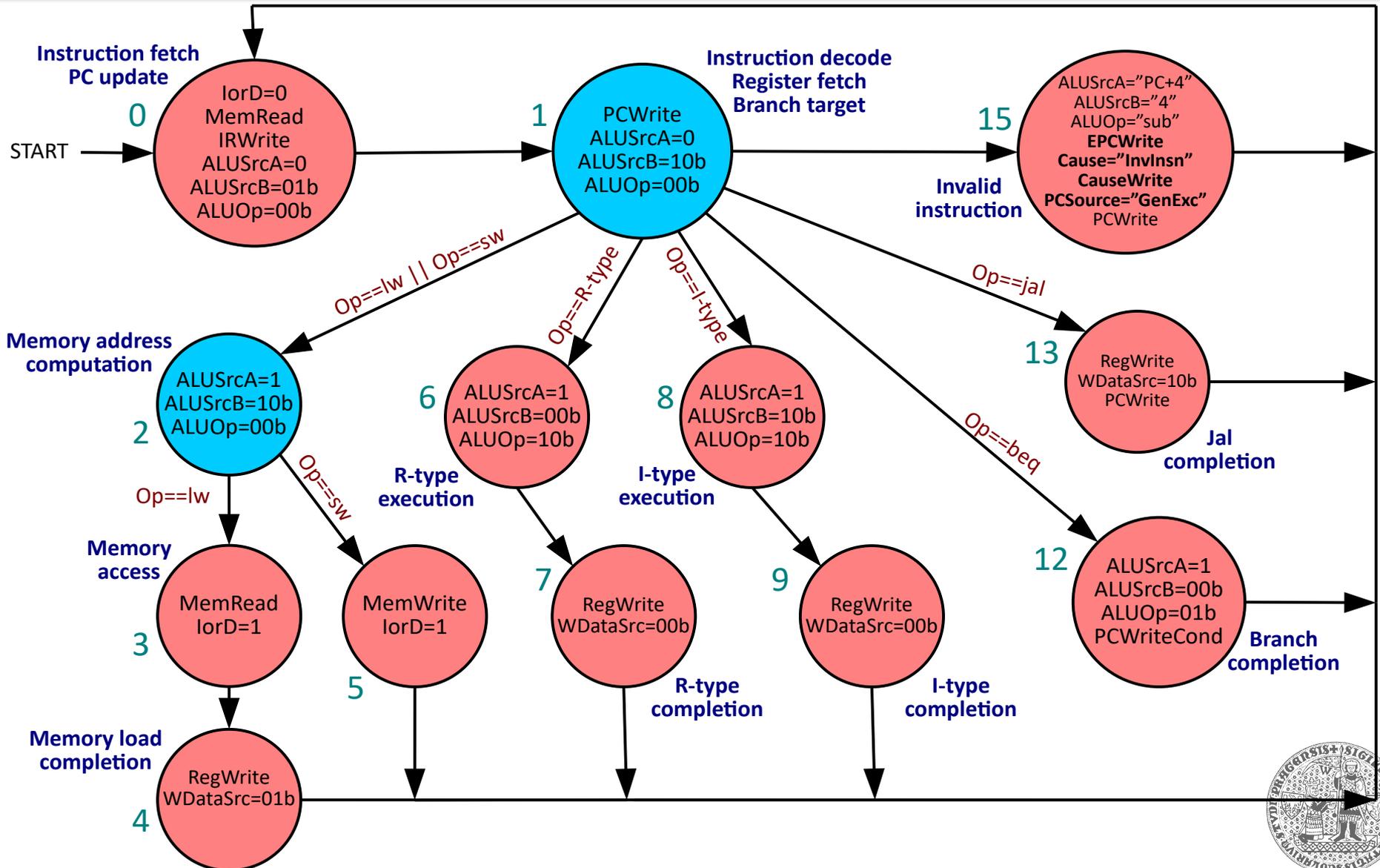
- **Hardware support (minimum necessary)**
 - Stop executing an instruction
 - Maintain valid processor and computation state
 - Allow to identify cause
 - Flag bits in a special register
 - Identifier of exception type
 - Store address of instruction that caused exception
 - Allows re-executing or skipping an instruction on resume
 - Jump to exception/interrupt handler
 - Single address for all exceptions/interrupts
 - Multiple addresses corresponding to exception type



Invalid instruction exception



Multi-cycle datapath control (4)



Supporting exceptions and interrupts (2)

- **Software handler**

- Store the current state of computation
 - Save contents of CPU registers to memory
- Determine the cause of exception/interrupt and execute the corresponding handler routine
 - Deal with I/O device
 - Deal with memory management
 - Continue/terminate current process
 - Switch to another process
- Restore state of current (next) process
- Resume execution (jump into) of current (next) process
 - Restart instruction that caused an exception
 - Continue from next instruction



Multi-cycle datapath performance

- **Instruction mix**

- 30% load (5ns), 10% store (5ns)
- 50% add (4ns), 10% mul (20ns)

- **Single-cycle datapath (clock period 20ns, CPI = 1)**

- **20ns** per instruction → **50 MIPS**

- **Coarse-grained multi-cycle datapath (clock period 5ns)**

- **$CPI \approx (90\% \times 1) + (10\% \times 4) = 1.3$**
- **6.5ns** per instruction → **153 MIPS**

- **Fine-grained multi-cycle datapath (clock period 1ns)**

- **$CPI \approx (30\% \times 5) + (10\% \times 5) + (50\% \times 4) + (10\% \times 20) = 6$**
- **6ns** per instruction → **166 MIPS**



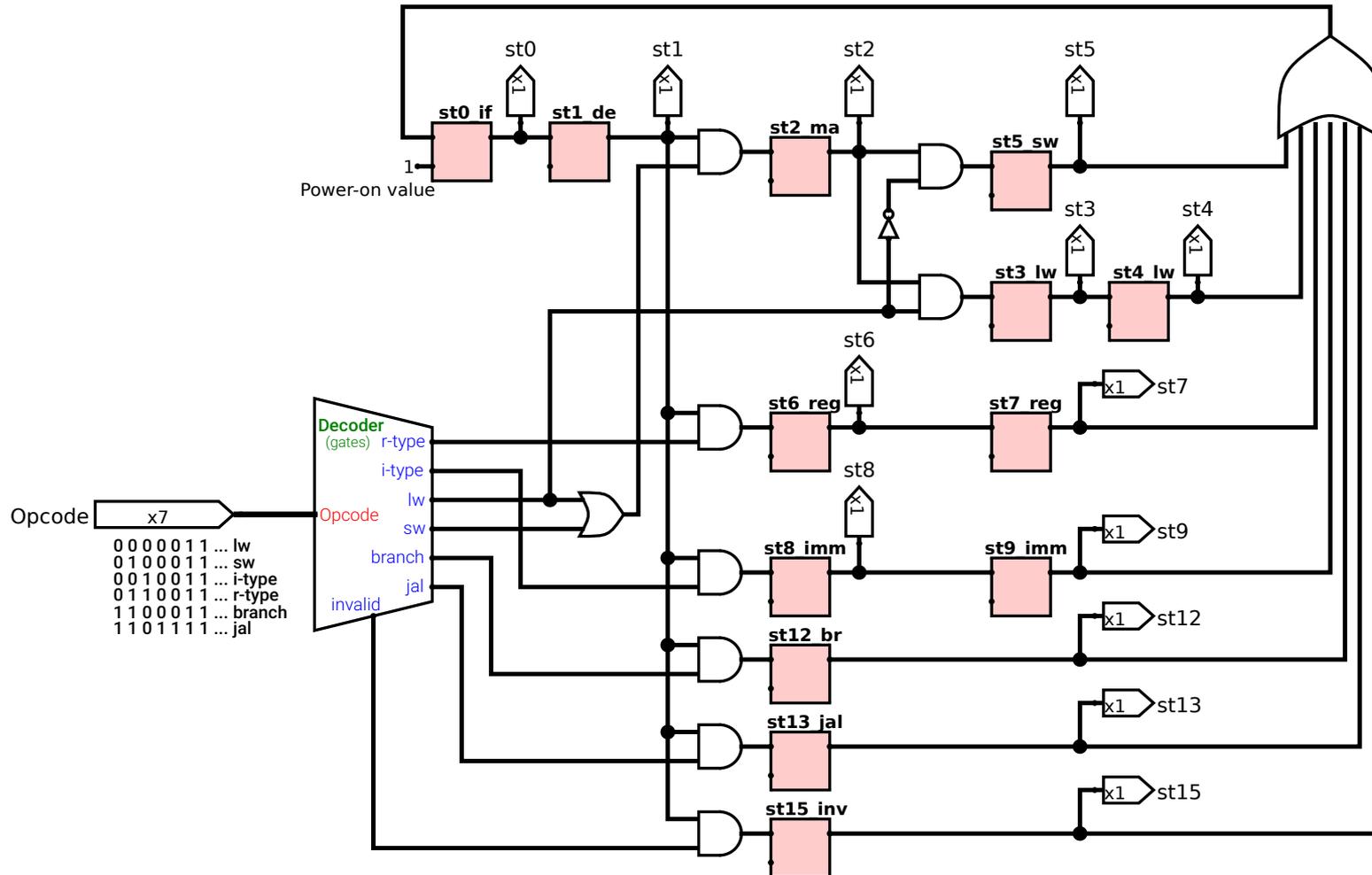
Implementing a sequential controller (1)

- **Implementing a finite-state automaton**
 - State + transition conditions = memory + combinational logic → sequential logic
 - Implementation depends on internal state representation
 - Sequential circuitry
 - 1 flip-flop per state (only one active at a time), active state shifted through enabling gates between flip-flops
 - State register + combinational logic
 - Simple sequencer + control memory
 - Micro- and nano- programming



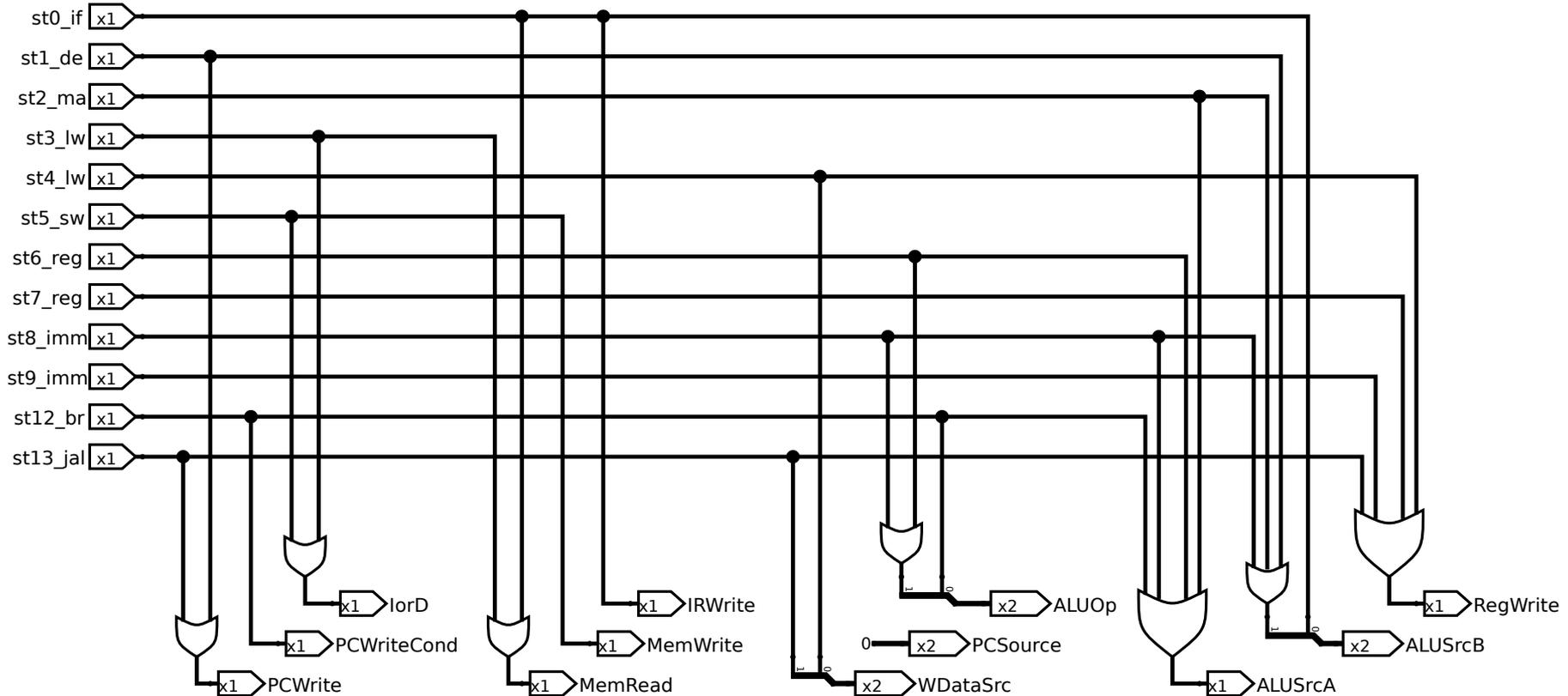
Implementing a sequential controller (2)

- State and transitions as flip-flop chains



Implementing a sequential controller (3)

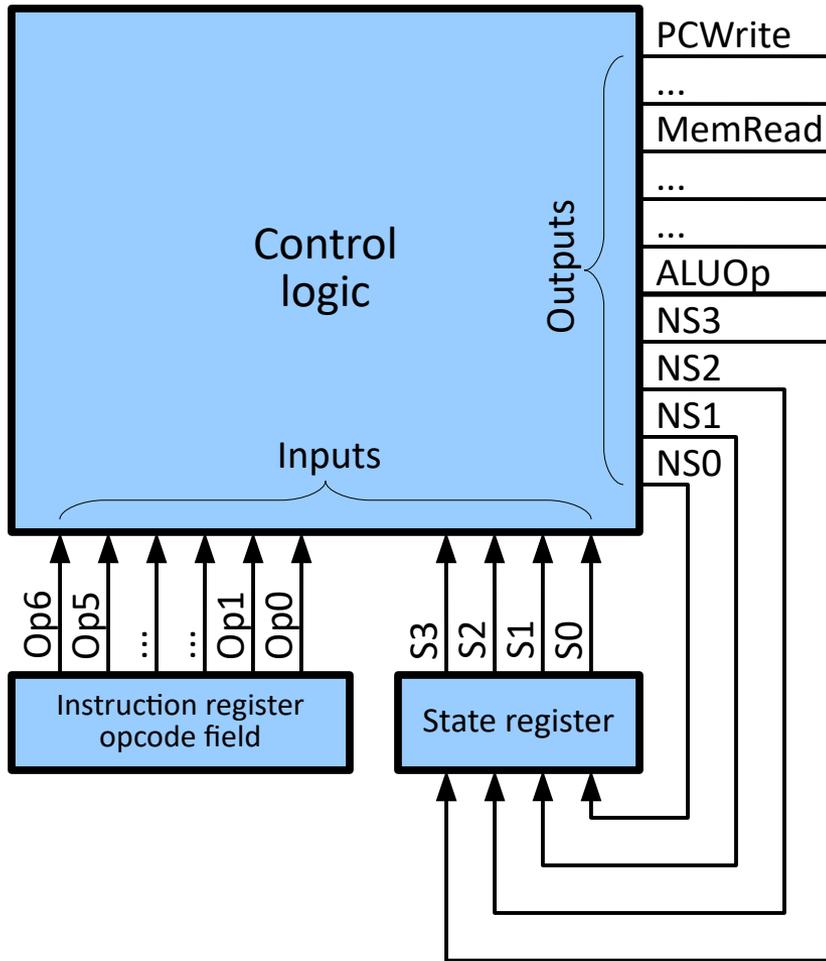
● Outputs as a functions of state



Derive control signals from controller state
Each signal collects states in which it is active



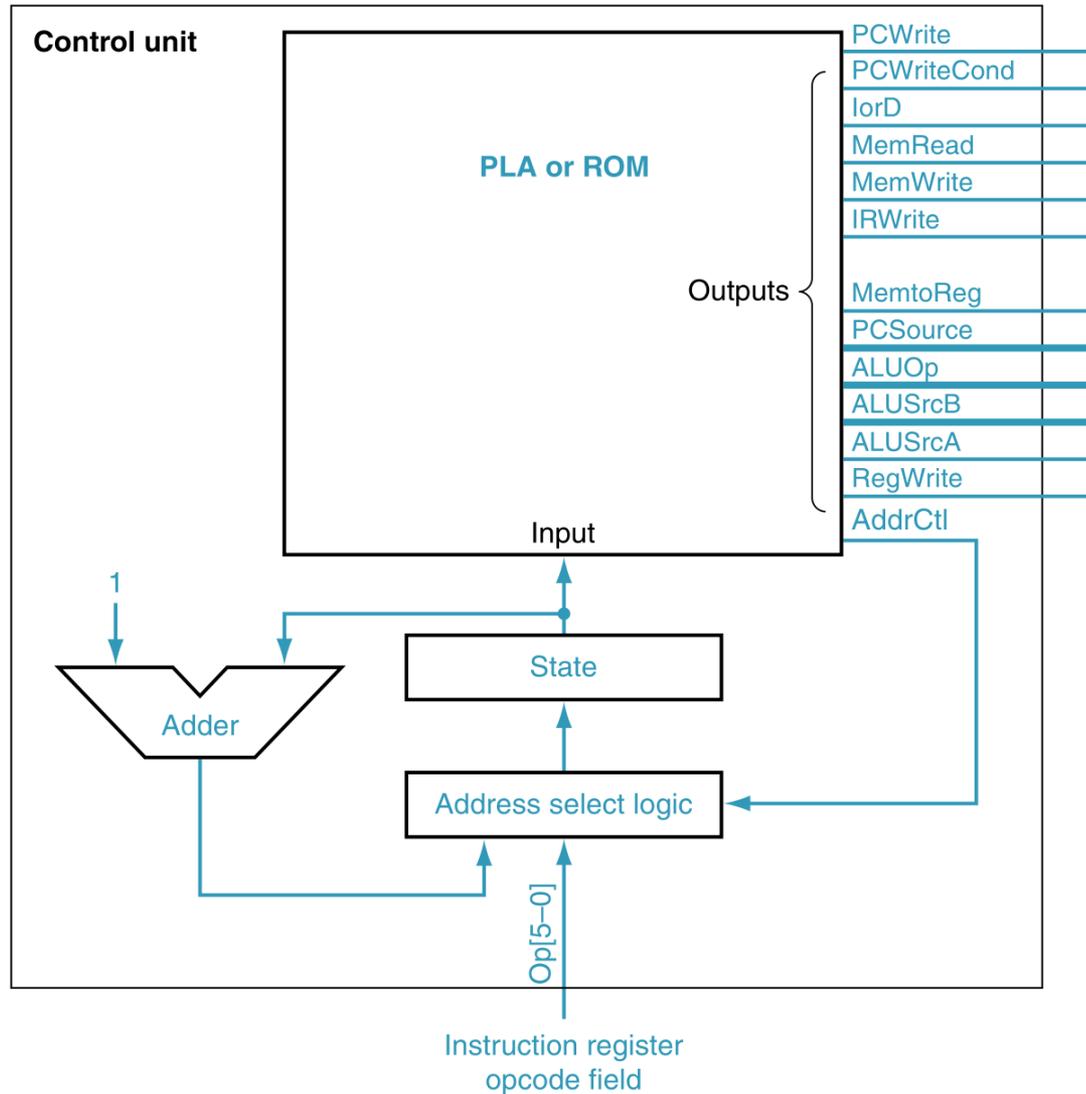
Implementing a sequential controller (4)



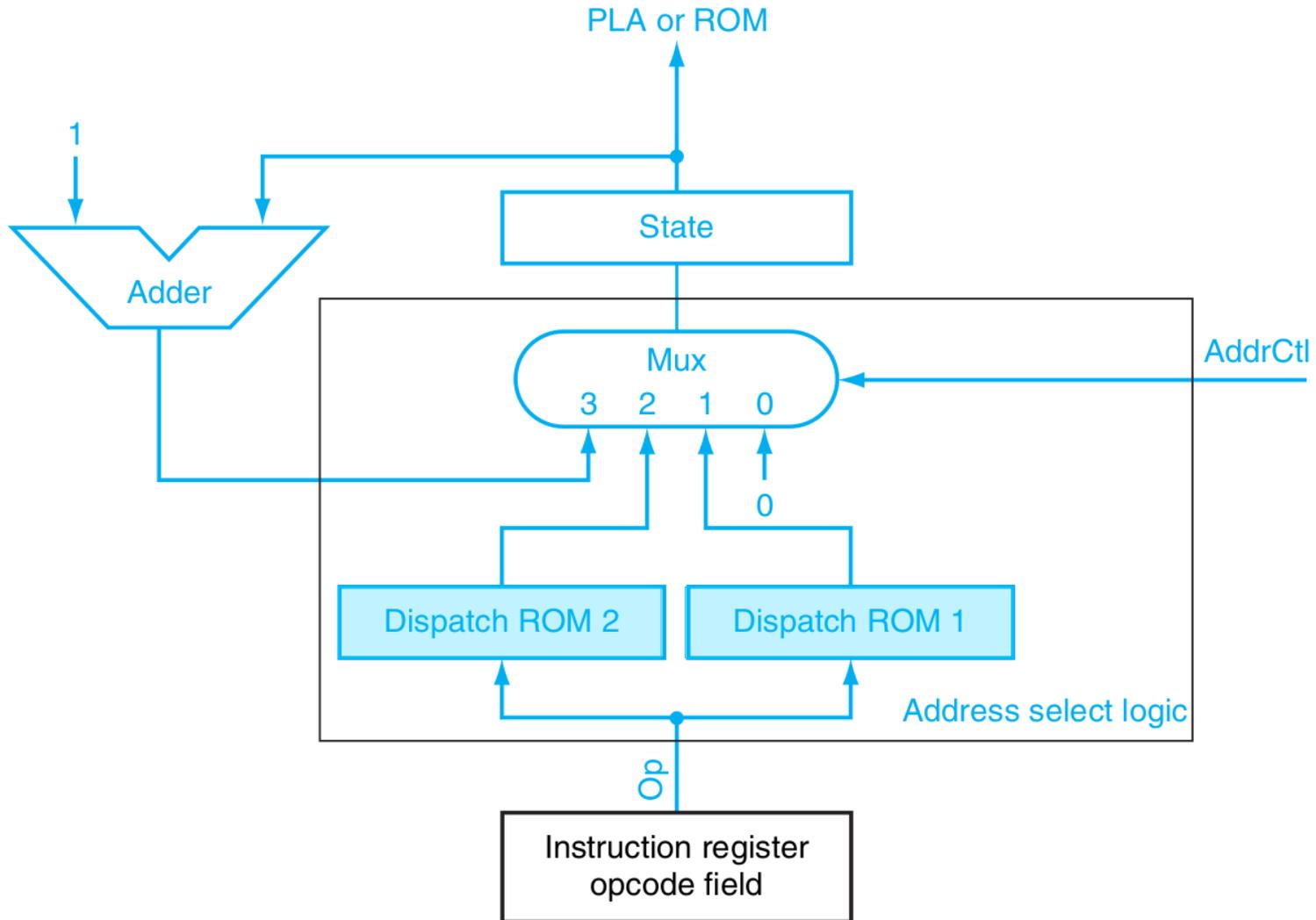
- **State register**
- **Control logic**
 - Combinational logic
 - ROM, FPGA



Next state is next control ROM address

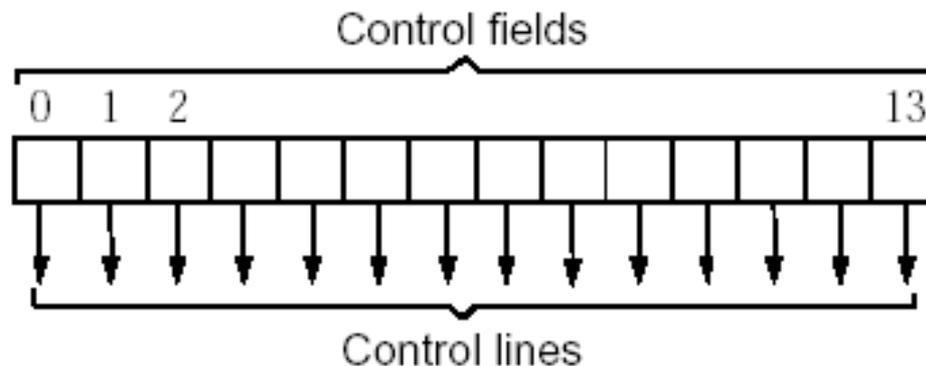


Control memory address select logic



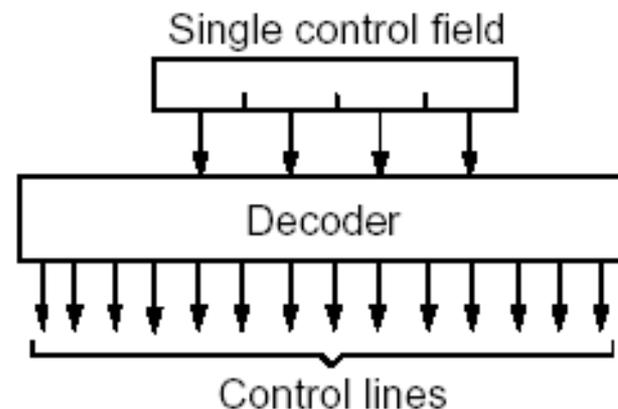
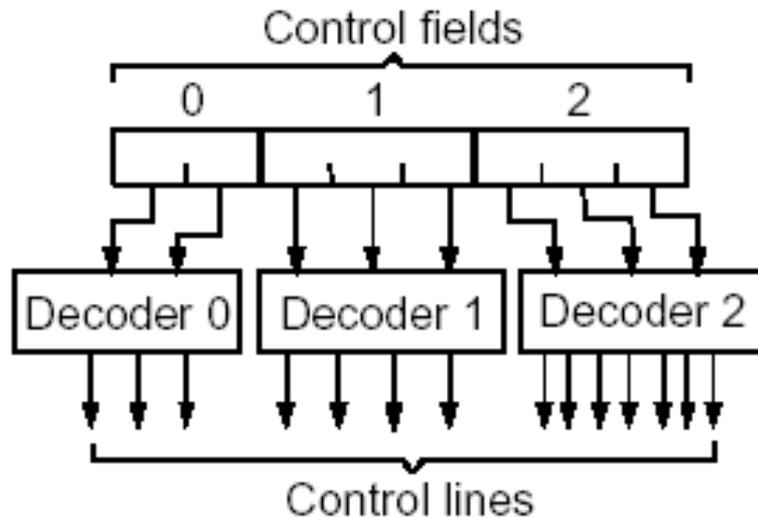
Horizontal micro-instructions

- **Direct representation of control signals**
 - Control memory contains raw control signals
 - Micro-instruction = set of control signal values
 - No need to decode (fast)
 - Any combination is possible (flexible)
 - Requires a lot of space



Vertical micro-instructions

- **Encoded representation of control signals**
 - Microinstructions identify valid combinations of control signals
 - Decoded into actual control signals using a decoder
 - Reduces space at the cost of flexibility and latency

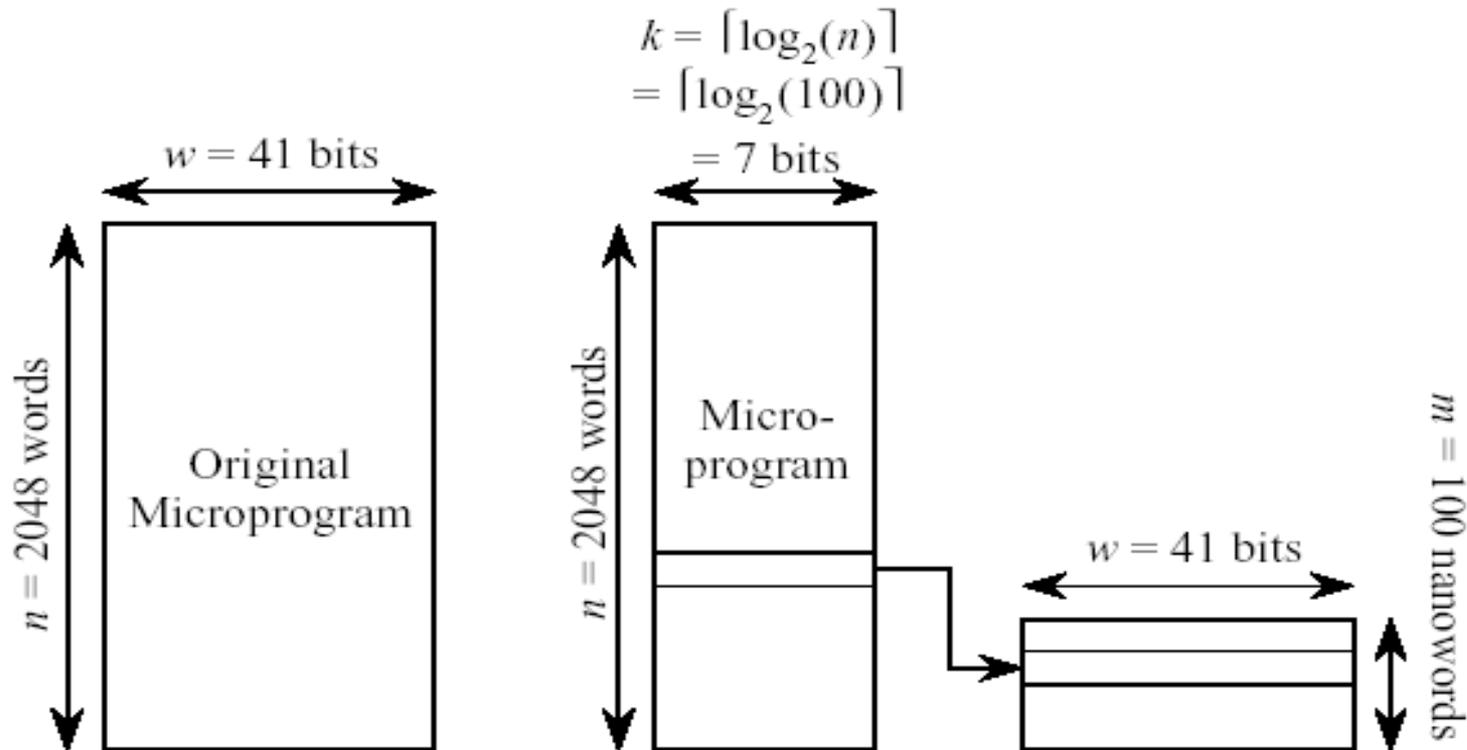


Nano-programming

- **Combines horizontal & vertical encoding**
 - Microprogram memory only contains numbers representing valid combinations of control signals (vertical format)
 - Decoding to horizontal format is realized using another memory (instead of a combinational circuit) which contains the control signal combination corresponding to microprogram code
 - Significantly reduces the amount of space required to store the microprogram, but increases decoding latency



Micro- vs nano-programming



$$\text{Total Area} = n \times w = 2048 \times 41 = 83,968 \text{ bits}$$

$$\text{Microprogram Area} = n \times k = 2048 \times 7 = 14,336 \text{ bits}$$

$$\text{Nanoprogram Area} = m \times w = 100 \times 41 = 4100 \text{ bits}$$

$$\text{Total Area} = 14,336 + 4100 = 18,436 \text{ bits}$$

