Topic #6

Processor Design

Major Goals

- To present the **single-cycle implementation** and to develop the student's understanding of combinational and clocked sequential circuits and the relationship between them.

- To present the **multiple-cycle implementation** and to further develop the student's understanding of combinational and clocked sequential circuits and the relationship between them.

- To introduce **microprogramming**.

How Are These Several Topics Related?

- The performance of a computer is determined by (**Topic #2**):
  - **Instruction count**
  - **Clock cycle time**
  - **Clock cycles per instruction** (**CPI**)

- The instruction count is determined by the compiler and the instruction set architecture (**Topics #3**, **#4** and **#5**).

- The clock cycle time and the CPI are determined by the implementation of the processor (**Topic #6** - this topic).

A MIPS Subset Implementation

- For illustration, we will study an implementation of a subset of the core MIPS instruction set:
    - **Memory-reference instructions**: **`lw`**, **`sw`**
    - **Arithmetic-logical instructions**: **`add`**, **`sub`**, **`and`**, **`or`**, **`slt`**
    - **Branch and jump instructions**: **`beq`**, **`j`**

- Instructions not included:
    - Integer instructions such as those for multiplication and division
    - Floating-point instructions

Steps to Execute MIPS Instructions

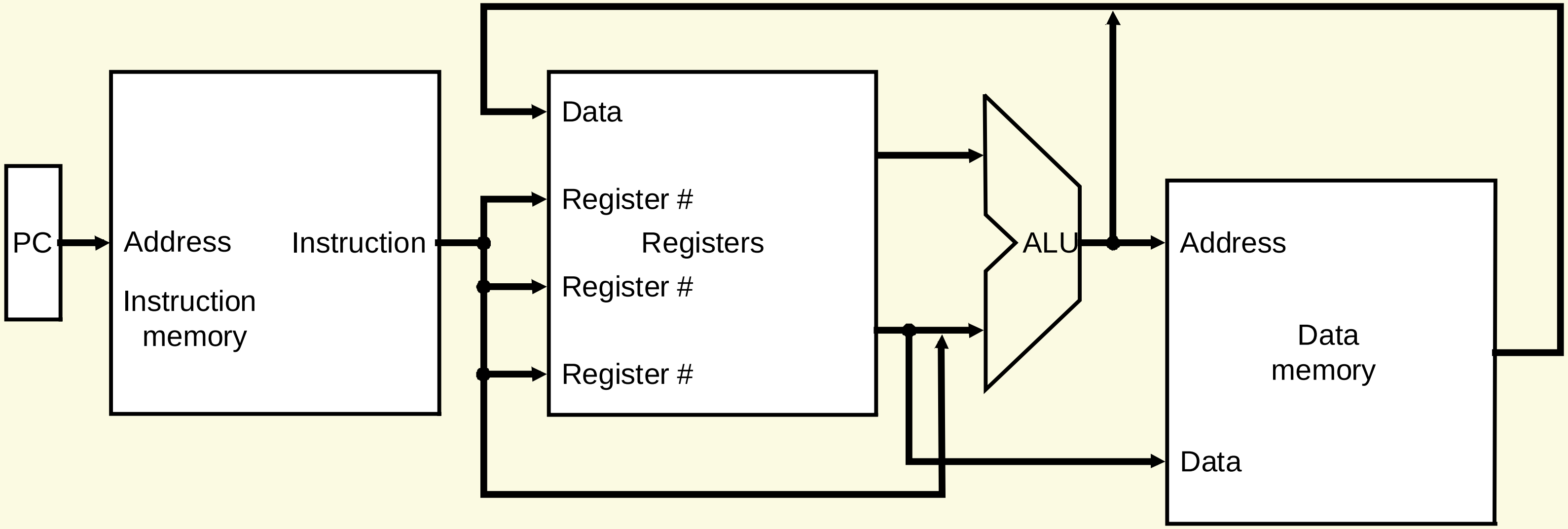
- Send the program counter (PC) to the memory location that contains the code and fetch the instruction from that memory location.

- Read one or two registers, using fields of the instruction to select the registers to read. For the load word and store word instructions we need to read only one register, but most other instructions require that we read two registers.

- Perform the operation required by the instruction using the ALU. Memory-reference instructions use the ALU for address calculation; arithmetic-logical instructions for operation execution; and branches for comparison.

- Store the result in registers or memory locations, and change the value of the program counter in case of a branch instruction.

High-Level View of the MIPS Subset Implementation



Sequential Logic Circuits

- MIPS computers are designed using both combinational and sequential logic circuits.

- **Sequential logic circuits** are circuits whose output depends on **both** the current input and the value stored in memory (called **state**).

- We will review: **clocks** and **memory elements**

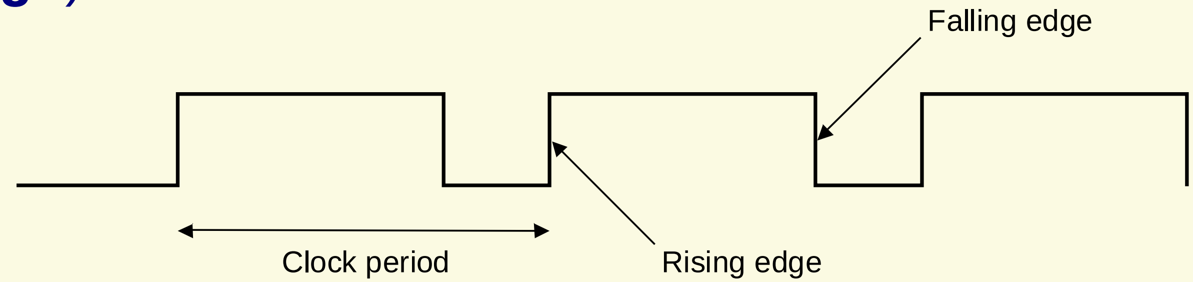
- Reference: Appendix B (B.4-B.5) of textbook

Clocks

- A **clock** is a free-running signal with a fixed **cycle time** (or called **clock period**) or, equivalently, a fixed **clock frequency** (i.e., inverse of the cycle time).

- Clocks are needed in sequential logic to decide when an element that contains state should be updated.

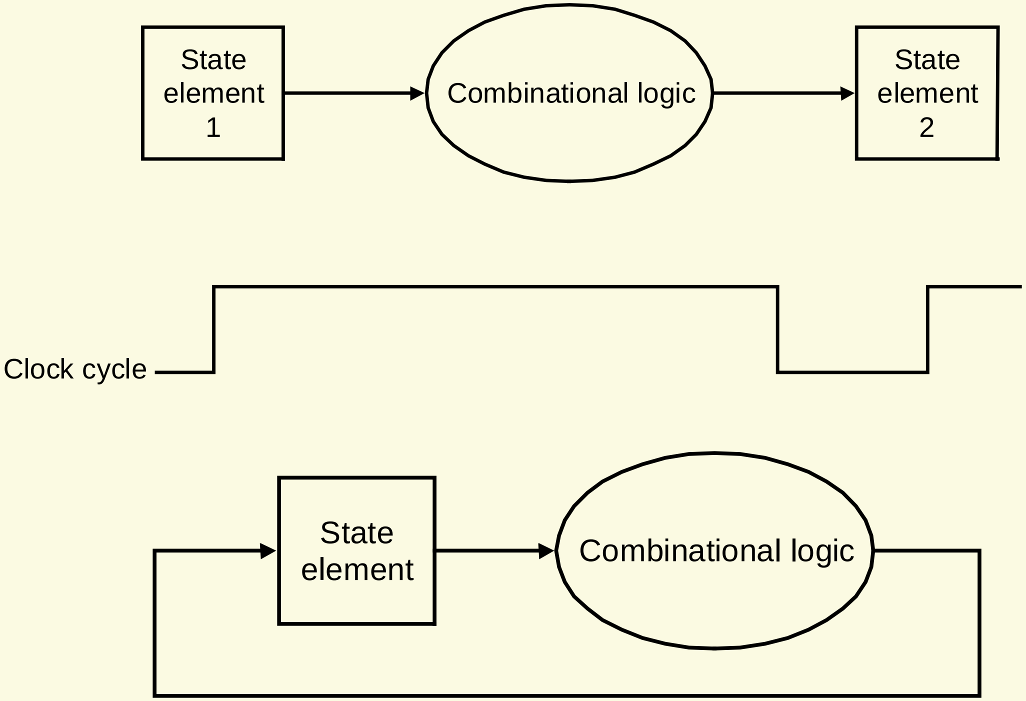
- **Edge-triggered clocking**:
  - Design methodology for sequential logic circuits in which all state changes occur on a clock edge (**rising edge** or **falling edge**).



Clocked Systems

- Clocked systems are also called **synchronous systems**.

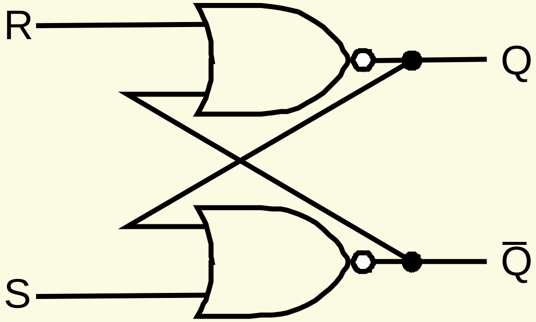
- Relationship among state elements and combinational logic blocks in a synchronous, sequential logic design:



Memory Elements

- **All memory elements store state**: the output from any memory element depends both on the current inputs and on the value that has been stored inside the memory element.

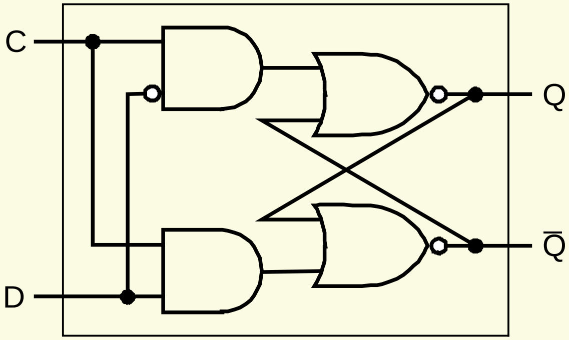
- **S-R latches** (set-reset latches):
  - **Unclocked** memory elements built from a pair of cross-coupled NOR gates (i.e., OR gates with inverted outputs).

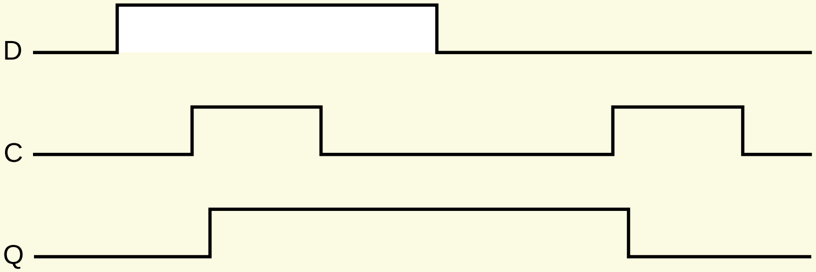


D Latches

- Unlike S-R latches which are unclocked, **D latches** are **clocked** (i.e., state changes are triggered by a clock).

- The output is equal to the value of the stored state inside it.



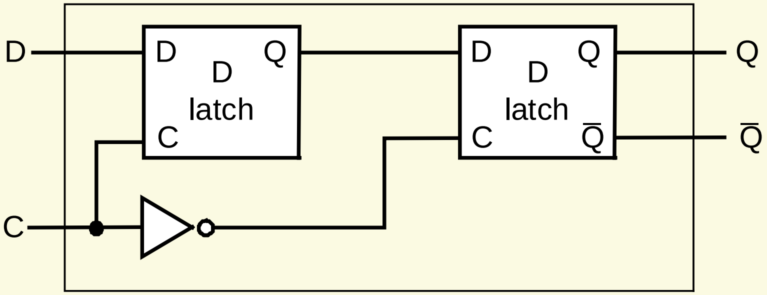


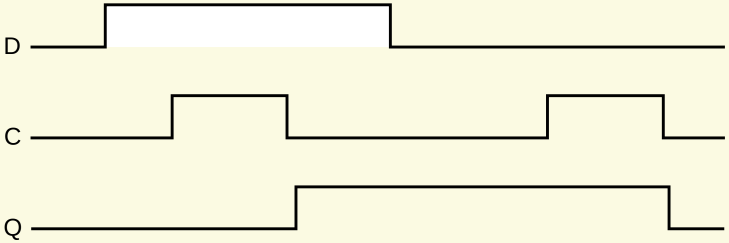
- **Operation**:
  - When the clock C is asserted, the latch is open and the Q output immediately assumes the value of the D input.

- Clocked latches are used to build **flip-flops**.

D Flip-Flops

- **D flip-flops**, like D latches, are **clocked**.
- The outputs change only on the (rising or falling) clock edge.
- **D flip-flop with a falling-edge trigger**:





- **Operation**:
  - When the clock input C changes from asserted to deasserted, the Q output stores the value of the D input.
- An array of D flip-flops can be used to build a **register** that can hold a multibit datum, such as a byte or word.

Register Files

- A **register file** is a structure in the datapath consisting of a set of registers that can be read and written by supplying a register number to be accessed.

- A register file can be implemented with a **decoder** for each read or write port and an array of **registers** built from D flip-flops.

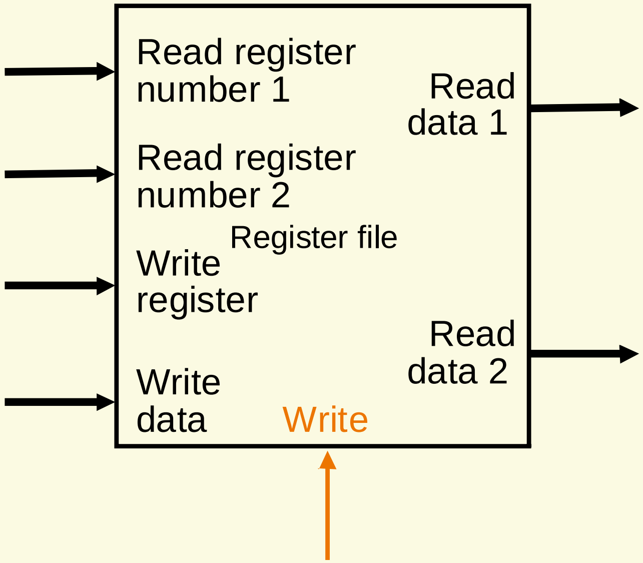
- **Reading a register**:
  - Input: a register number
  - Output: data contained in the specified register

- **Writing a register**:
  - Inputs: a register number, the data to write, and a clock that controls the writing into the register

Example: A Register File with Two Read Ports and One Write Port

- There are **five inputs** and **two outputs**.

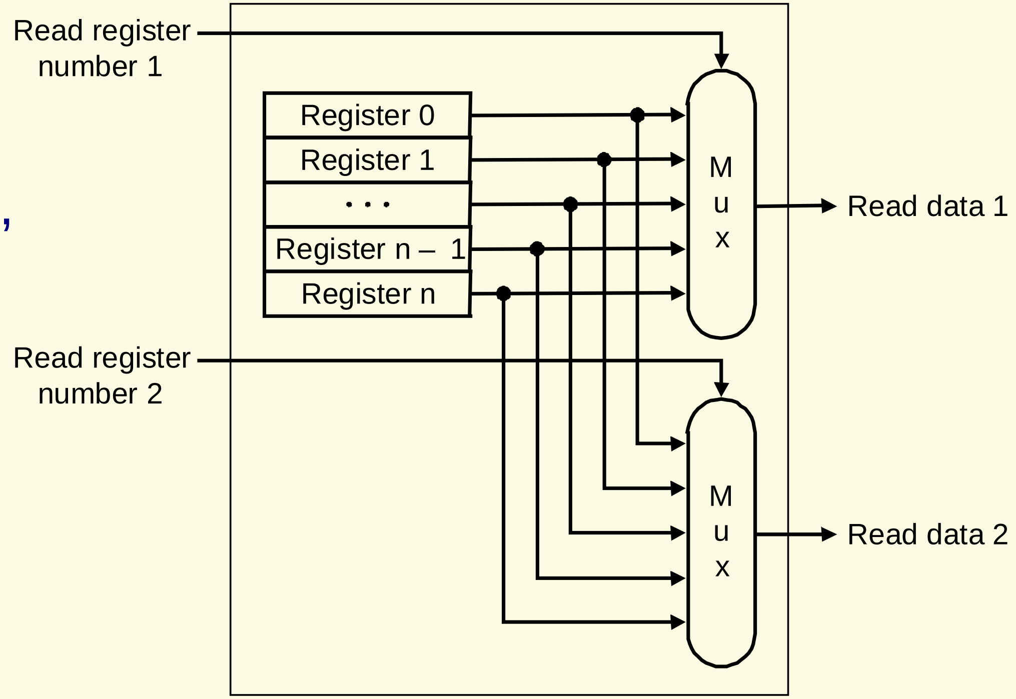
- The read ports can be implemented with a pair of **multiplexors**, each of which is as wide as the number of bits in the register file.



Implementation of Two Read Ports

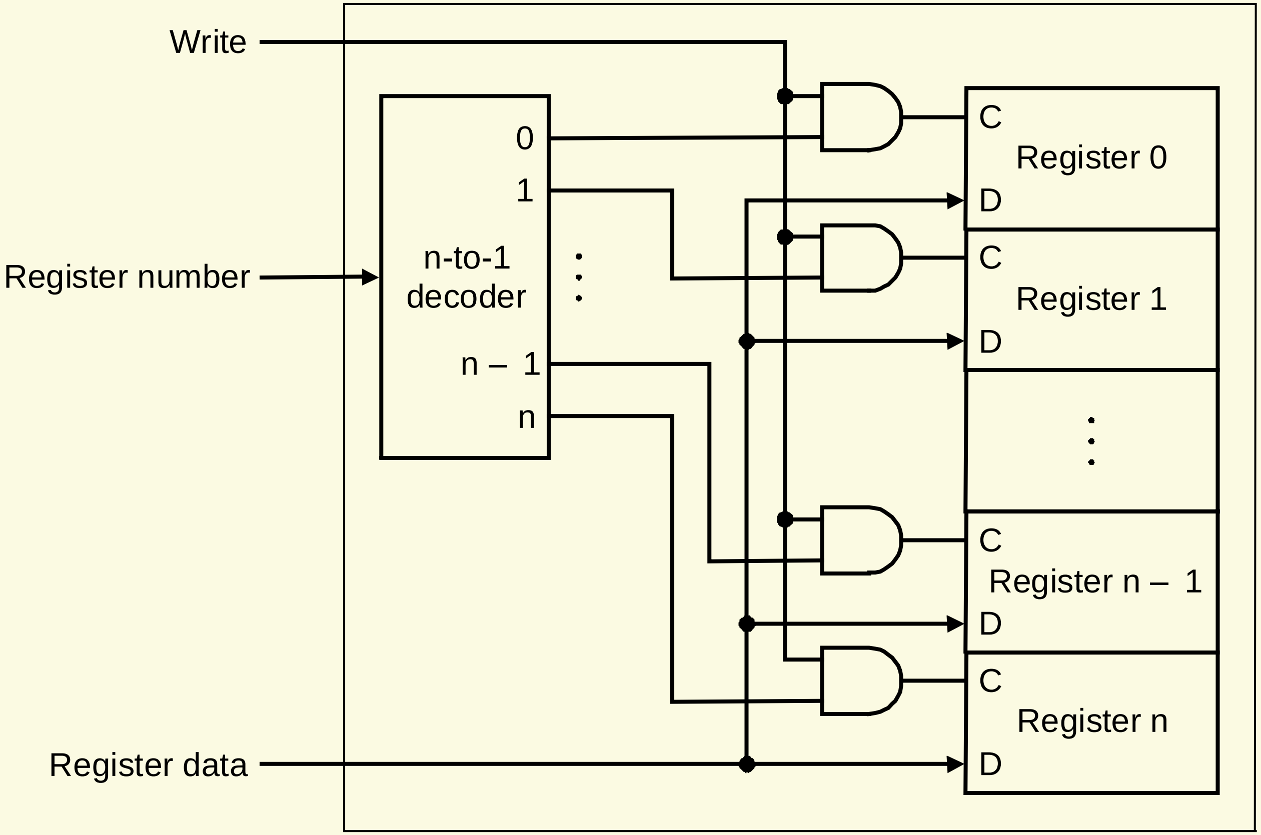
- To implement two read ports for a register file with n registers, we use two **n-to-1 multiplexors**, each of which is 32 bits wide.

- The read register number signal is used as the **multiplexor selector signal**.



Implementation of a Write Port

- To implement the write port for a register file with n registers, we use an **n-to-1 decoder** to generate a signal that can be used to determine which register to write.



SRAMs and DRAMs

- Unlike registers which are small, fast memories inside the processor, larger amounts of memory are in the form of:
  - **Static random access memories** (**SRAMs**)
  - **Dynamic random access memories** (**DRAMs**)

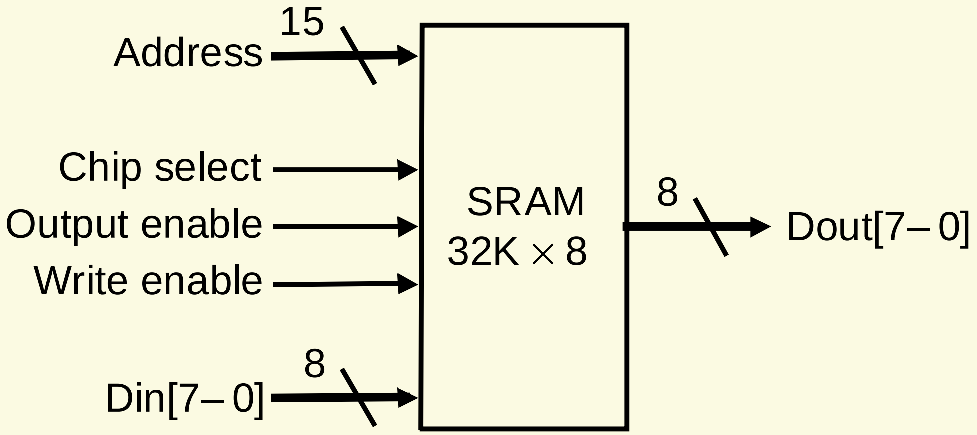
- SRAMs are usually used for relatively **small** memories (e.g., **cache**) while DRAMs are used for **larger** memories (e.g., main memory).

- SRAMs are typically built using **flip-flops** while DRAMs are built using **capacitors**.

- SRAMs are somewhat simpler in design than DRAMs, but they are also more expensive and less dense.

SRAMs

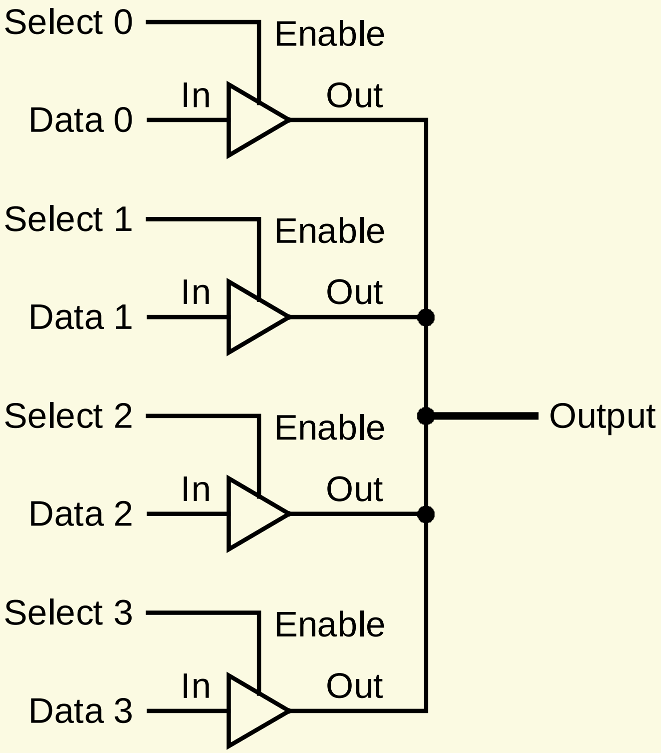
- **256K x 1 SRAM**:
  - Length (i.e., number of addressable locations) = 256K
  - Width (i.e., number of bits per entry) = 1
  - 18 address lines, 1 data input line, 1 data output line



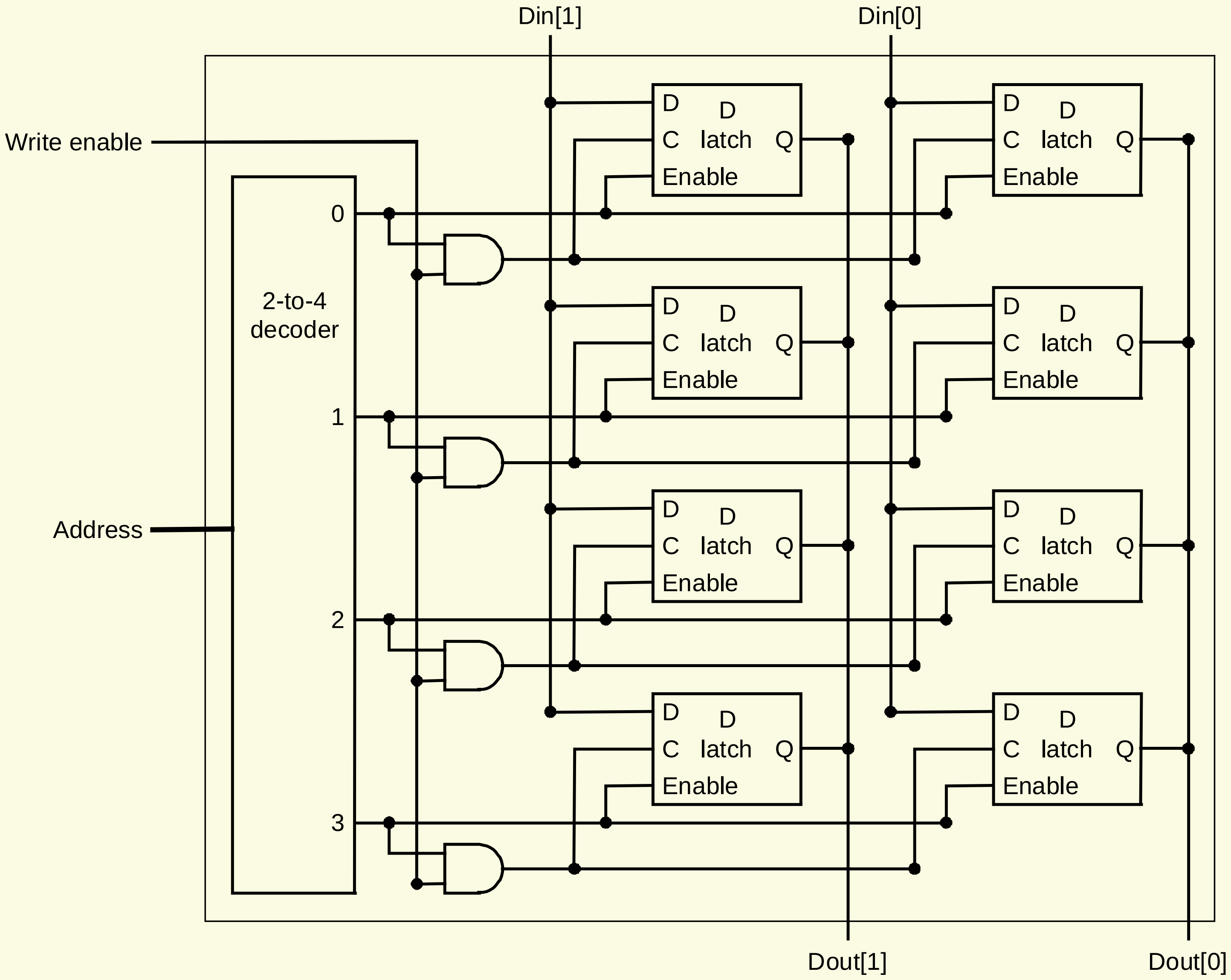
- **32K x 8 SRAM**:
  - 15 address lines, 8 data input lines, 8 data output lines

SRAMs

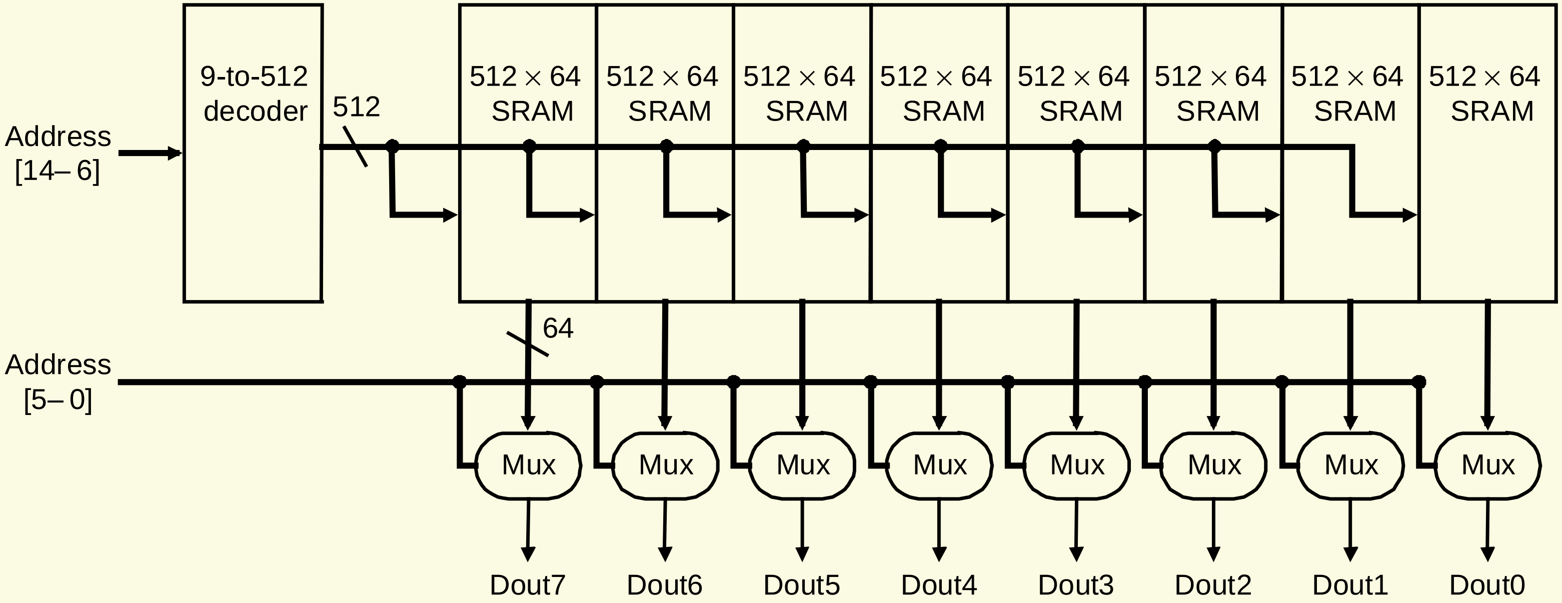
- ❑ Large SRAMs cannot be built like register files using multiplexors, as a 64K-to-1 multiplexor (too large!) would be needed for a 64K x 1 SRAM.
- ❑ Instead, a shared output line, called **bit line**, is used so that multiple memory cells in the memory array can assert.
- ❑ A **three-state buffer** (or **tri-state buffer**) is used to allow multiple sources to drive a single line.
- ❑ A **multiplexor** constructed from four three-state buffers:



A 4 x 2 SRAM


Din[1]
Din[0]
Write enable
Address
2-to-4
decoder
0
1
2
3
D
D
C
latch
Q
Enable
Dout[1]
Dout[0]

A 32K x 8 SRAM as an Array of 512 x 64 Arrays



DRAMs

- The value kept in a cell is stored as a **charge** in a **capacitor**.

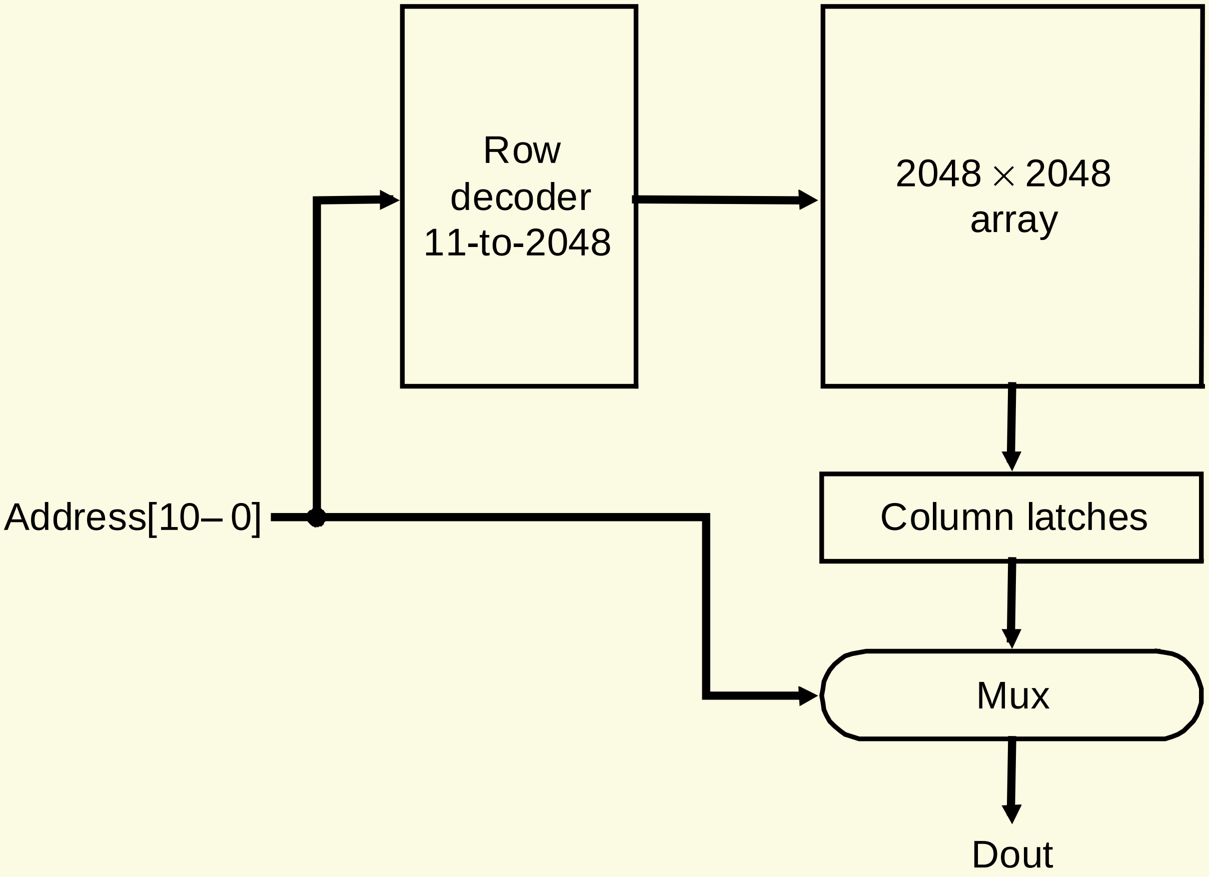
- Only a single transistor is needed to access the stored charge (either to read the value or to overwrite the charge stored there), and hence DRAMs are much denser and cheaper per bit.

- Since the charge is stored on a capacitor, it cannot be kept indefinitely but has to be **refreshed periodically**.

- To refresh a cell, we read its content and then write it back.

- Typically, refresh operations consume 1-2% of the active cycles, leaving 98-99% of the cycles available for reading and writing data.

A 4M x 1 DRAM Built with a 2048 x 2048 Array



Synchronous RAMs

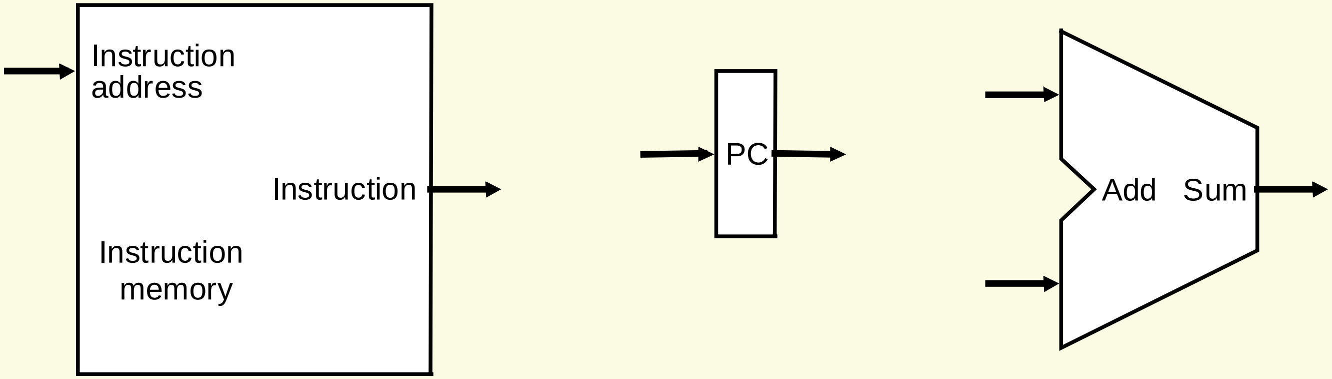
- **Synchronous SRAMs** (SSRAMs) and **synchronous DRAMs** (SDRAMs)

- The key capability provided by synchronous RAMs is the ability to transfer a burst of data from a series of sequential addresses within an array or row. The burst is defined by a **starting address**, supplied in the usual fashion, and a **burst length**.

- The speed advantage of synchronous RAMs comes from the ability to transfer the bits in the burst without having to specify additional address bits. Instead, a clock is used to transfer the successive bits in the burst.

Building a Datapath

- Some basic datapath elements:
  - **Instruction memory**: a memory unit that stores the instructions of a program and supplies an instruction given its address.
  - **Program counter**: a register that stores the address of the instruction being executed.
  - **Adder**: a unit that increments the program counter to the address of the next instruction.



a. Instruction memory

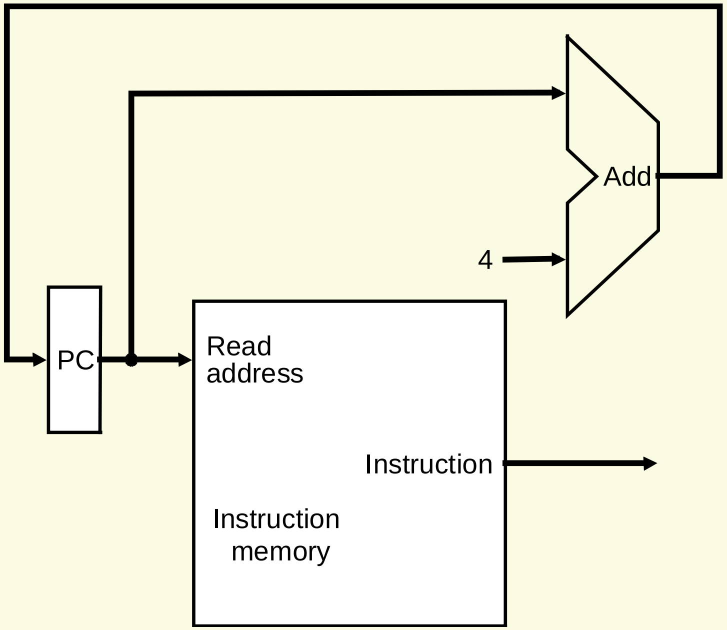
b. Program counter

c. Adder

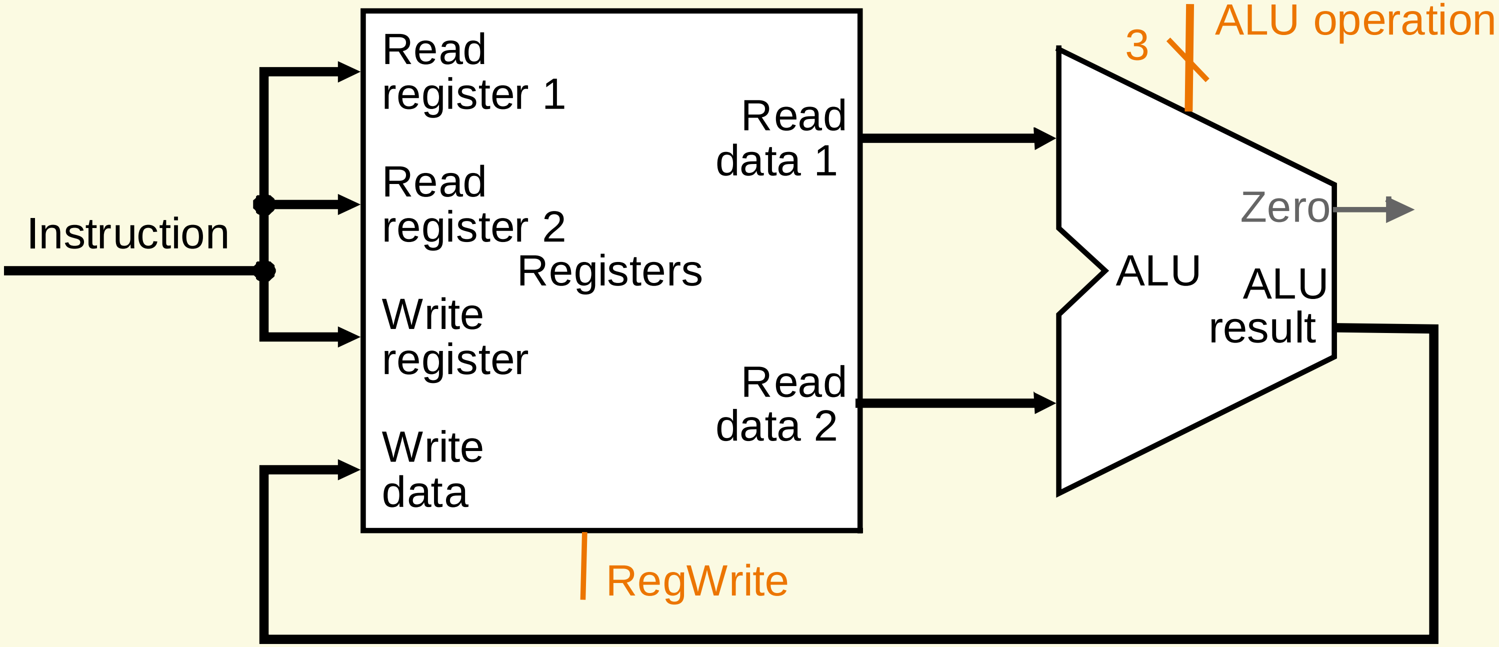
Building a Datapath

- ❑ To execute any instruction, we first fetch the instruction from memory.

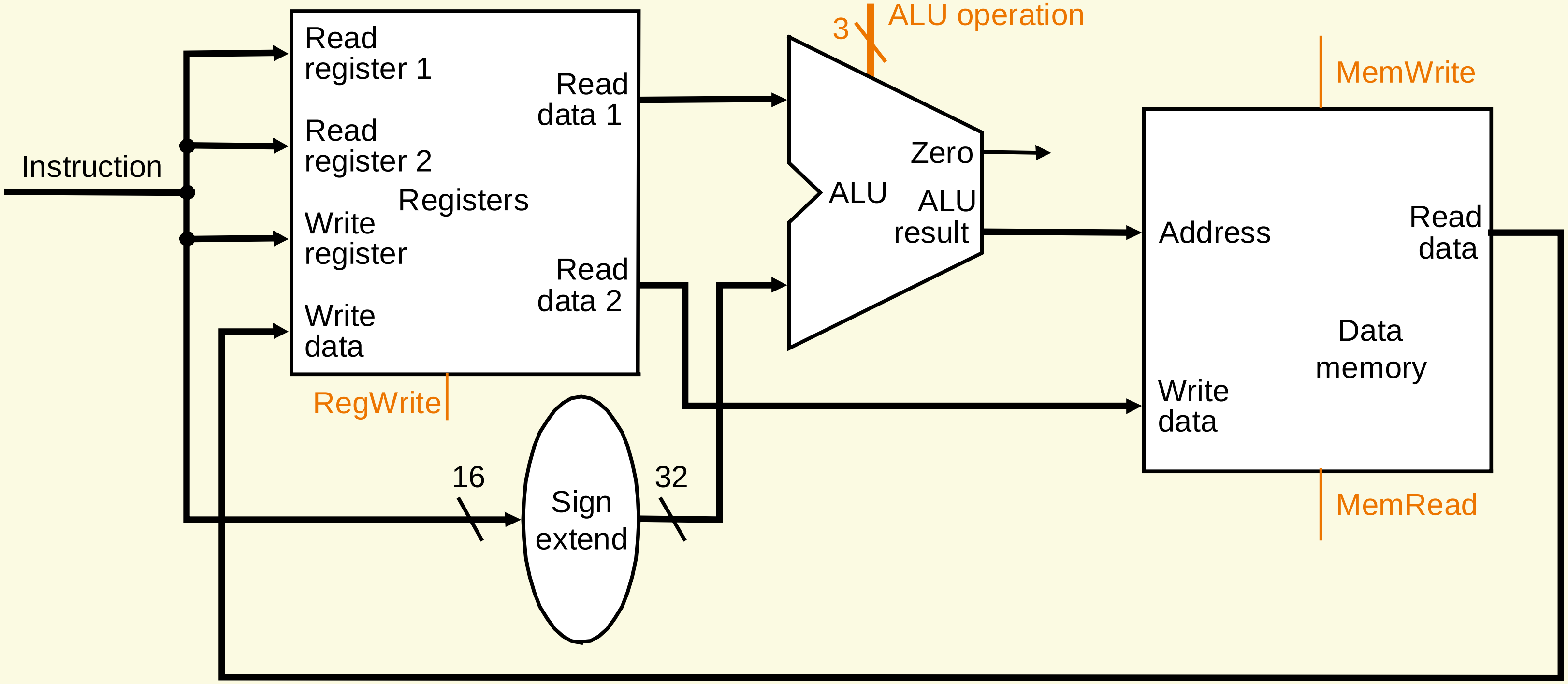
- ❑ To prepare for executing the next instruction, we increment the program counter so that it points at the next instruction (4 bytes later).



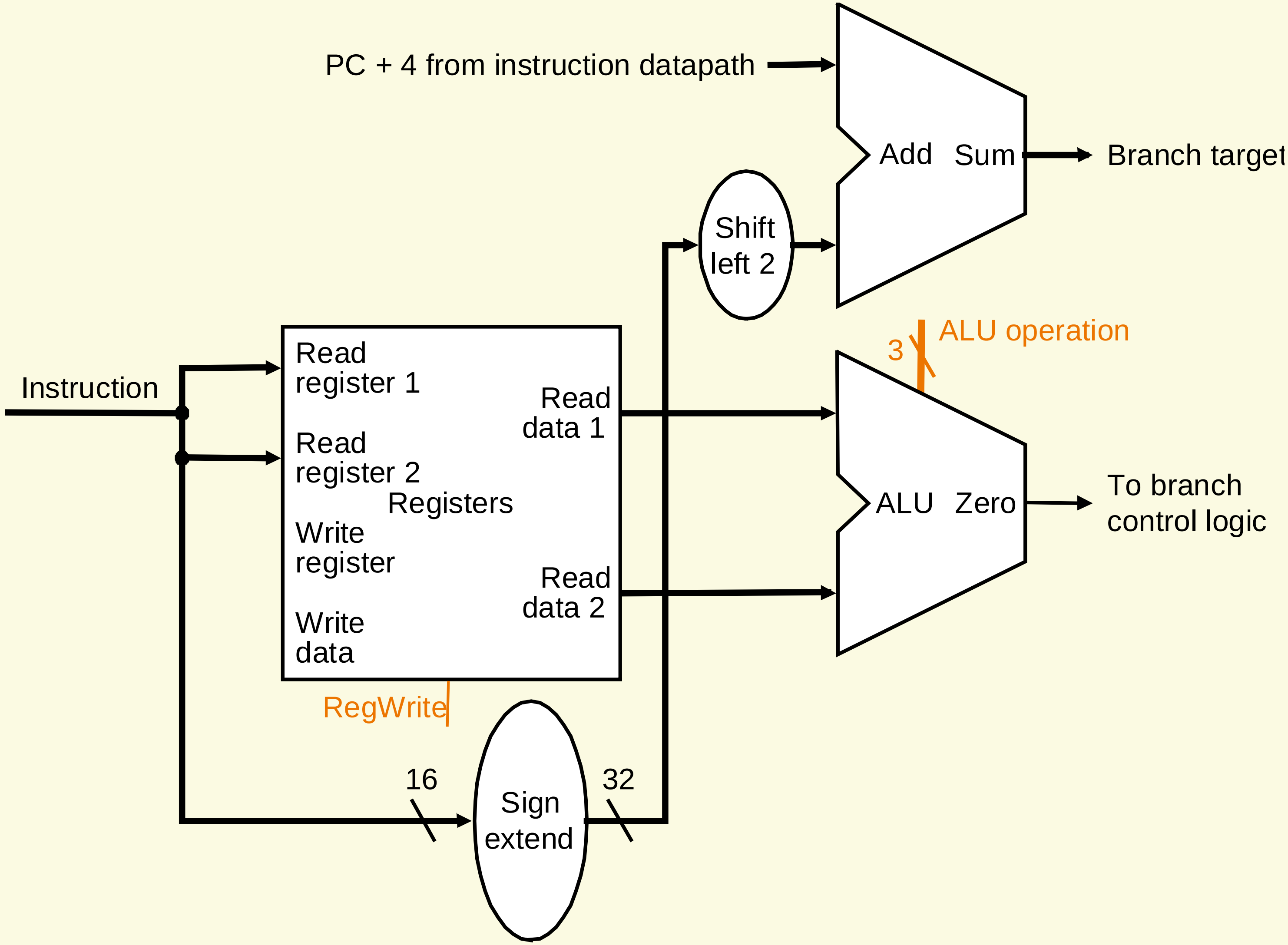
Datapath for Arithmetic-Logical (R-Format) Instructions



Datapath for Load and Store (I-Format) Instructions



Datapath for Branch (I-Format) Instructions

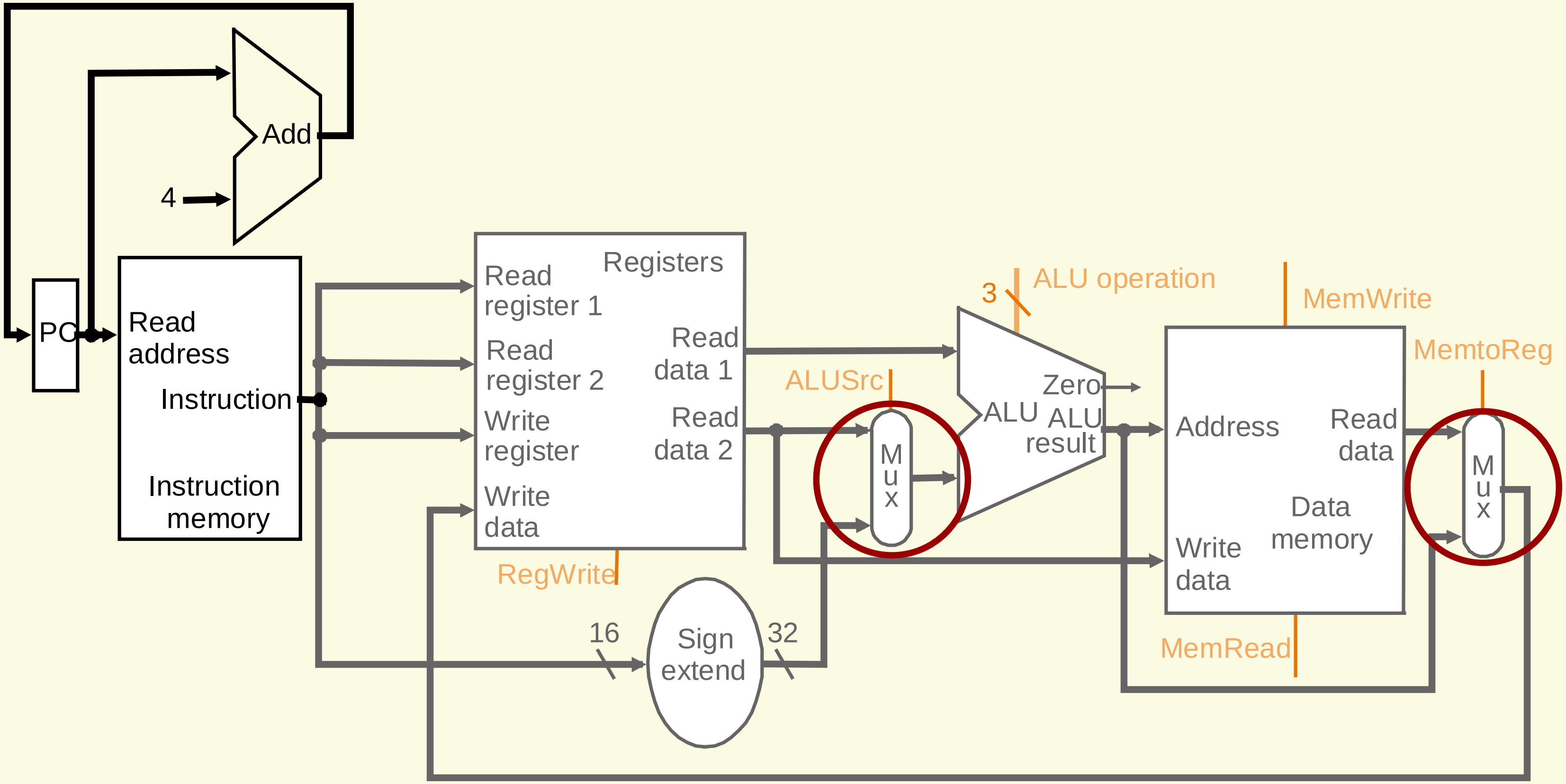


A Simple Single-Cycle Implementation

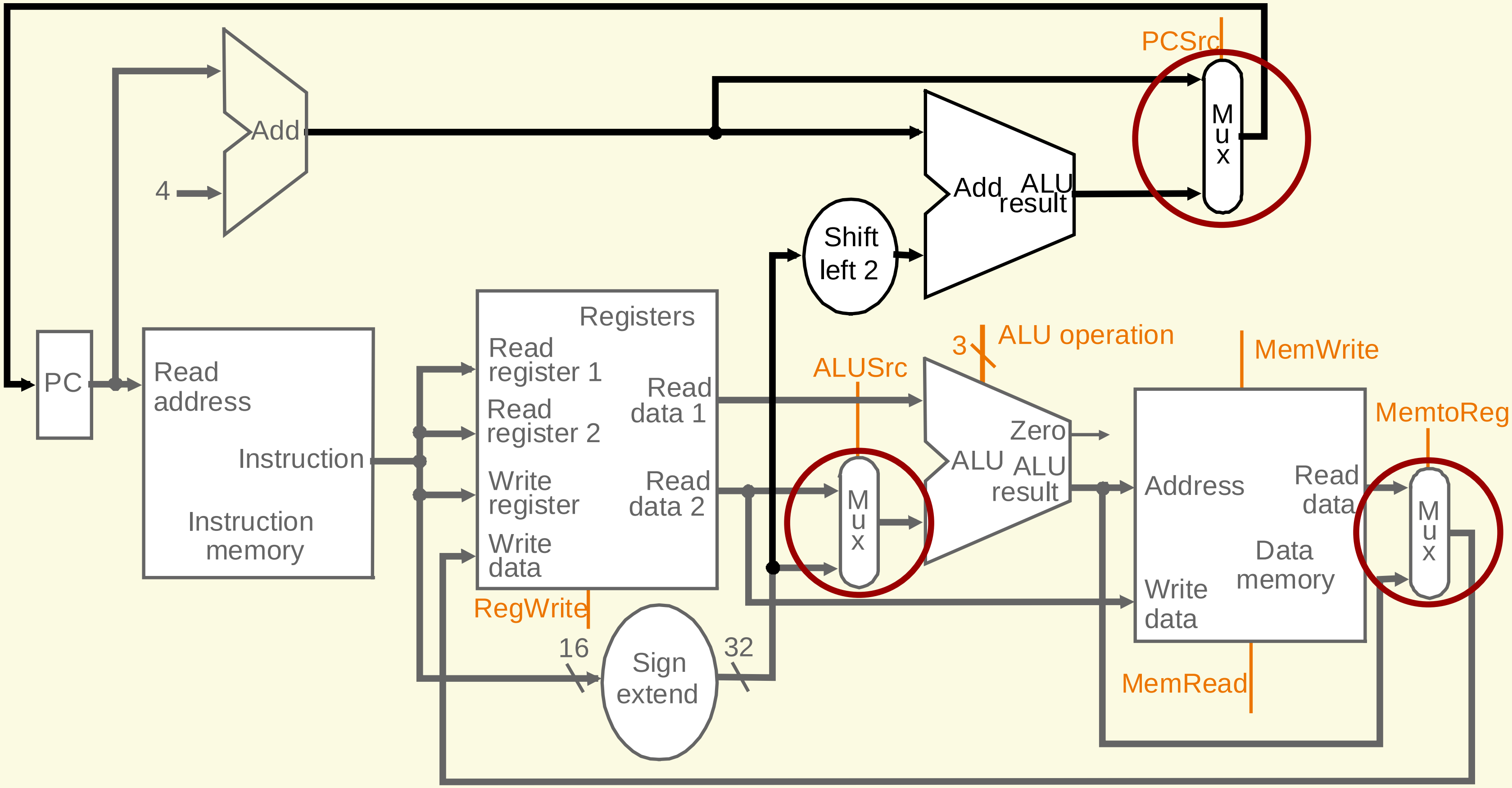
- We have already built a datapath for each instruction separately. Now, we need to combine them into a **single datapath**, by devising ways to share some of the resources (e.g., ALU) among the different instructions instead of duplicating them.

- This **simple implementation** is based on the (unrealistic) assumption that all instructions take just one clock cycle each to complete.
  - Implication: No datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. As a consequence, we need a memory for instructions separate from one for data.

Combined Datapath for R-Format Instructions and Memory Instructions



Combined Datapath for Different Instruction Classes



ALU Control

- The **control unit** controls the whole operation of the datapath by generating appropriate **control signals** (e.g., write signals for state elements, selector inputs for multiplexors, ALU control inputs) for the proper operation of the datapath.

- The **ALU control** is part of the **main control unit**.

- Control input bits for ALU:

ALU Control Input	Function
000	and
001	or
010	add
110	subtract
111	set on less than

ALU Control Input Bits

- Inputs used by control unit to generate ALU control input bits:
  - **ALUOp** (2 bits)
  - **Function code** of instruction (6 bits)

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
Load word	00	load word	XXXXXX	add	010
Store word	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	and	100100	and	000
R-type	10	or	100101	or	001
R-type	10	set on less than	101010	set on less than	111

Multiple Levels of Decoding

- **Level 1:** Generation of ALUOp bits by main control unit - to be discussed later

- **Level 2:** Generation of ALU control input bits from ALUOp bits and function code of instruction

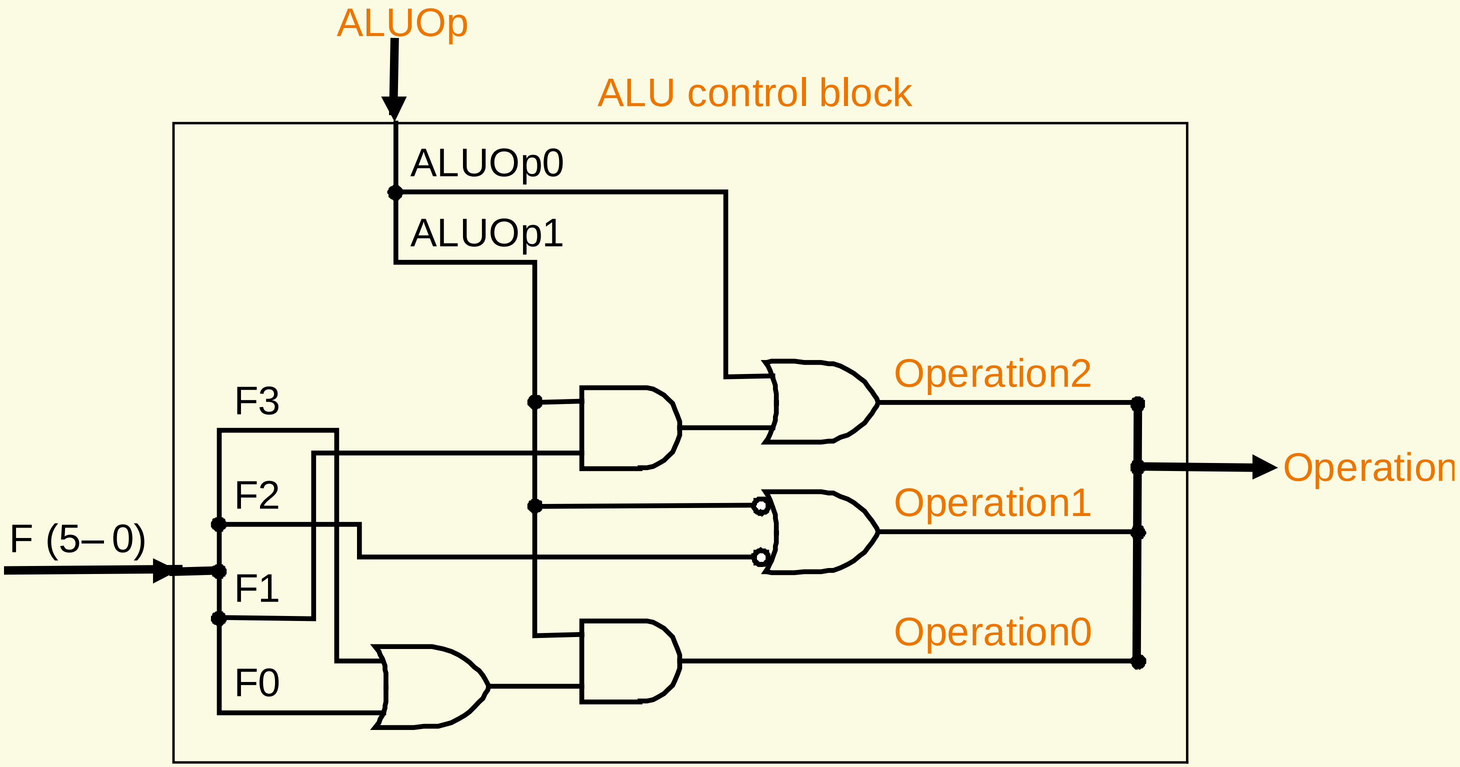
- **Why multiple levels**?
  - Using multiple levels of control can reduce the size of the main control unit, and may also potentially increase the speed of the control unit.
  - A common implementation technique.

Truth Table for ALU Control Block

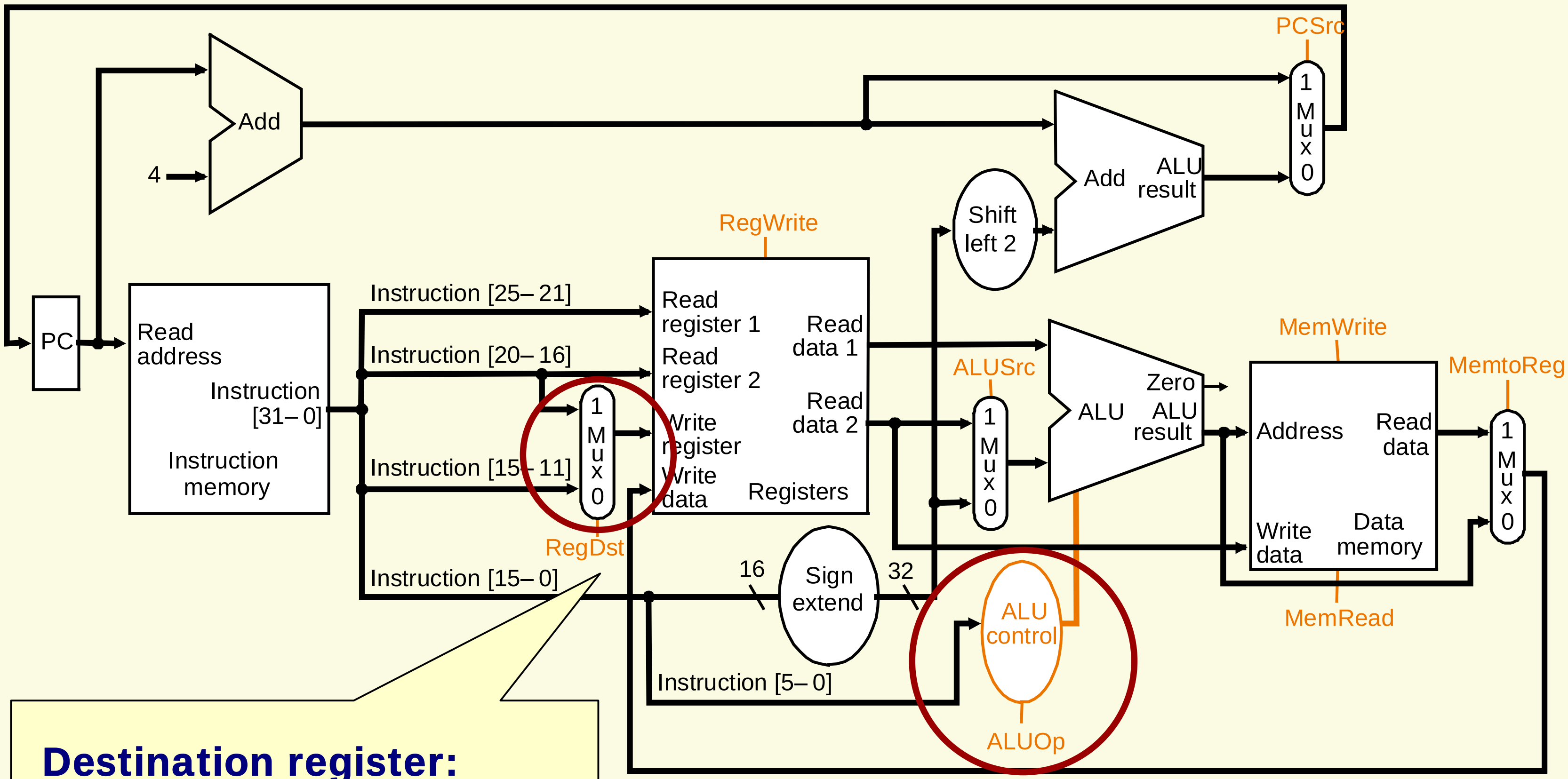
- The truth table contains many don't-care terms, which can lead to simplified hardware implementation.

ALUOp		Function code						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Hardware Implementation of ALU Control Block



Datapath with ALU Control



Destination register:

R-format: **rd** field (bits 15-11)

`lw`: **rt** field (bits 20-16)

Effects of Control Signals

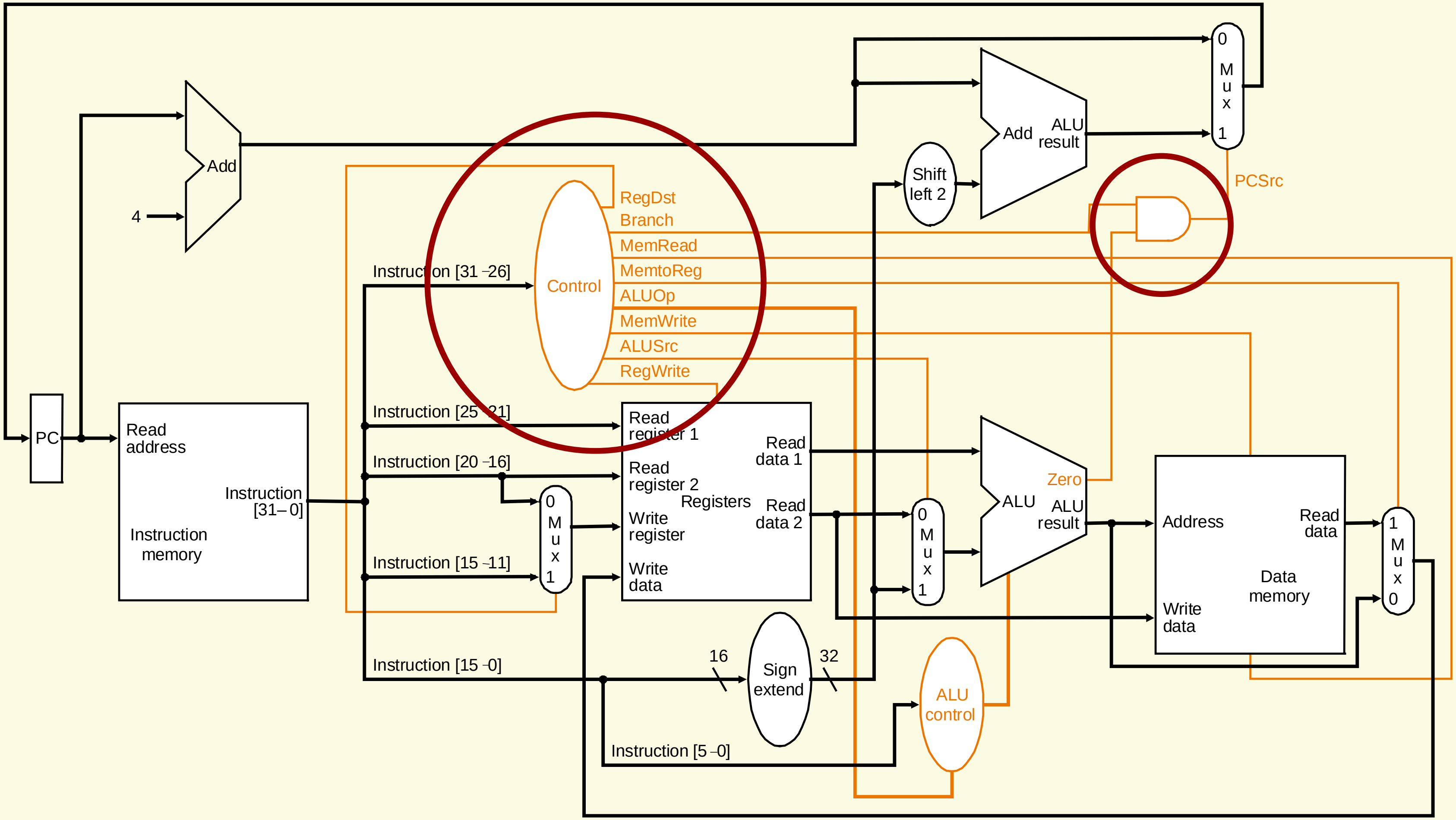
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the **rt** field (bits 20-16).	The register destination number for the Write register comes from the **rd** field (bits 15-11).
RegWrite	None	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Setting of Control Signals

- The 9 control signals (7 from the previous table + 2 from ALUOp) can be set based entirely on the 6-bit opcode, with the exception of PCSrc.

- PCSrc control line:
  - Set if both conditions hold simultaneously:
    - Instruction is **beq**.
    - Zero output of ALU is true (i.e., the two source operands are equal).

Datapath with Control Unit



Setting of Control Signals

- Setting of control lines is completely determined by opcode:

Instruction	Reg-Dst	ALU-Src	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

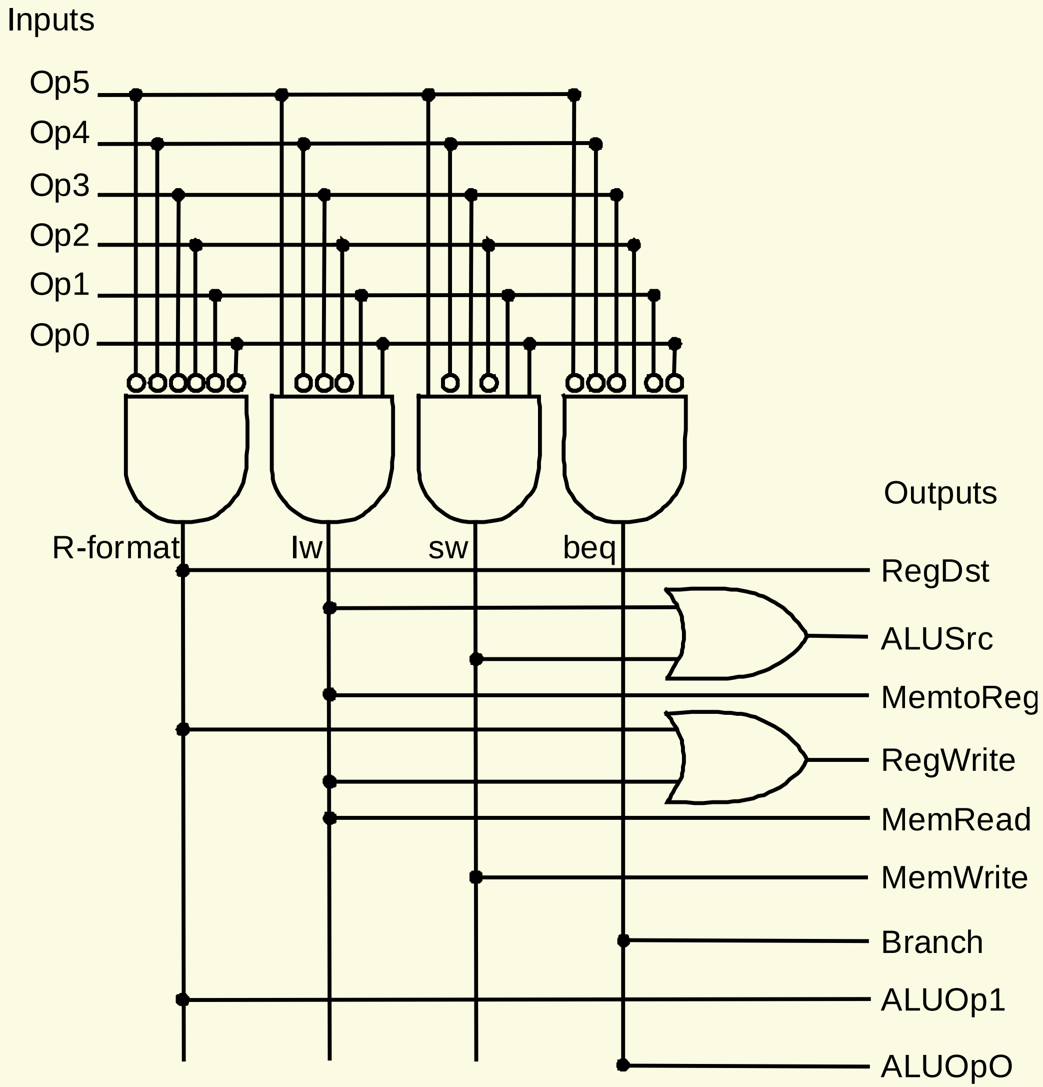
- Input to datapath control unit:

Instruction	Opcode in decimal	Opcode in binary					
		Op5	Op4	Op3	Op2	Op1	Op0
R-format	0	0	0	0	0	0	0
lw	35	1	0	0	0	1	1
sw	43	1	0	1	0	1	1
beq	4	0	0	0	1	0	0

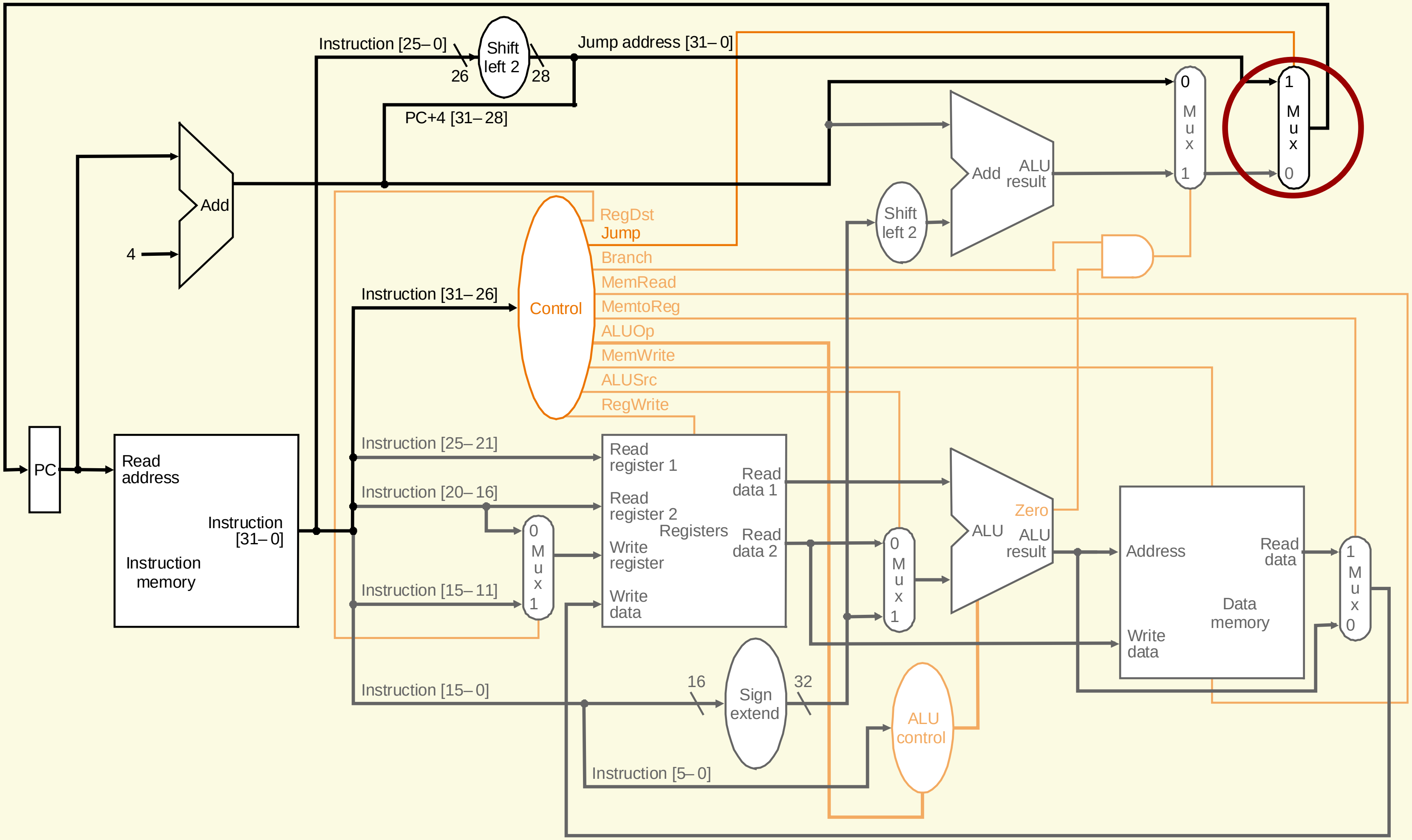
Truth Table for Datapath Control Unit

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Hardware Implementation of Datapath Control Unit



Simple Datapath and Control Extended to Handle the Jump Instruction



Problems with Single-Cycle Datapath Implementation

- Every instruction takes one clock cycle (CPI = 1). The clock cycle is determined by the longest possible path in the machine.

- The longest path is for a load instruction which involves five functional units in series: **instruction memory**, **register file**, **ALU**, **data memory**, **register file**.

- Even though each instruction takes just one clock cycle, the clock cycle is expected to be large and hence the overall performance is poor because many instructions cannot fully utilize the unnecessarily long clock cycle.

- No sharing of hardware functional units is possible.

Different Instruction Classes

Instruction class	Functional units used by the instruction class				
R-format	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

E.g., memory units: 2 ns   ALU: 2 ns   register file (read/write): 1 ns

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-format	2	1	2	0	1	6 ns
Load word	2	1	2	2	1	8 ns
Store word	2	1	2	2		7 ns
Branch	2	1	2			5 ns
Jump	2					2 ns

Problems with Single-Cycle Datapath Implementation

- Although a **variable clock implementation** would be faster, it is very difficult to implement a variable-speed clock.

- The penalty for using a single-length clock cycle becomes more severe if we also consider other computationally demanding instructions such as multiplication and floating-point operations.

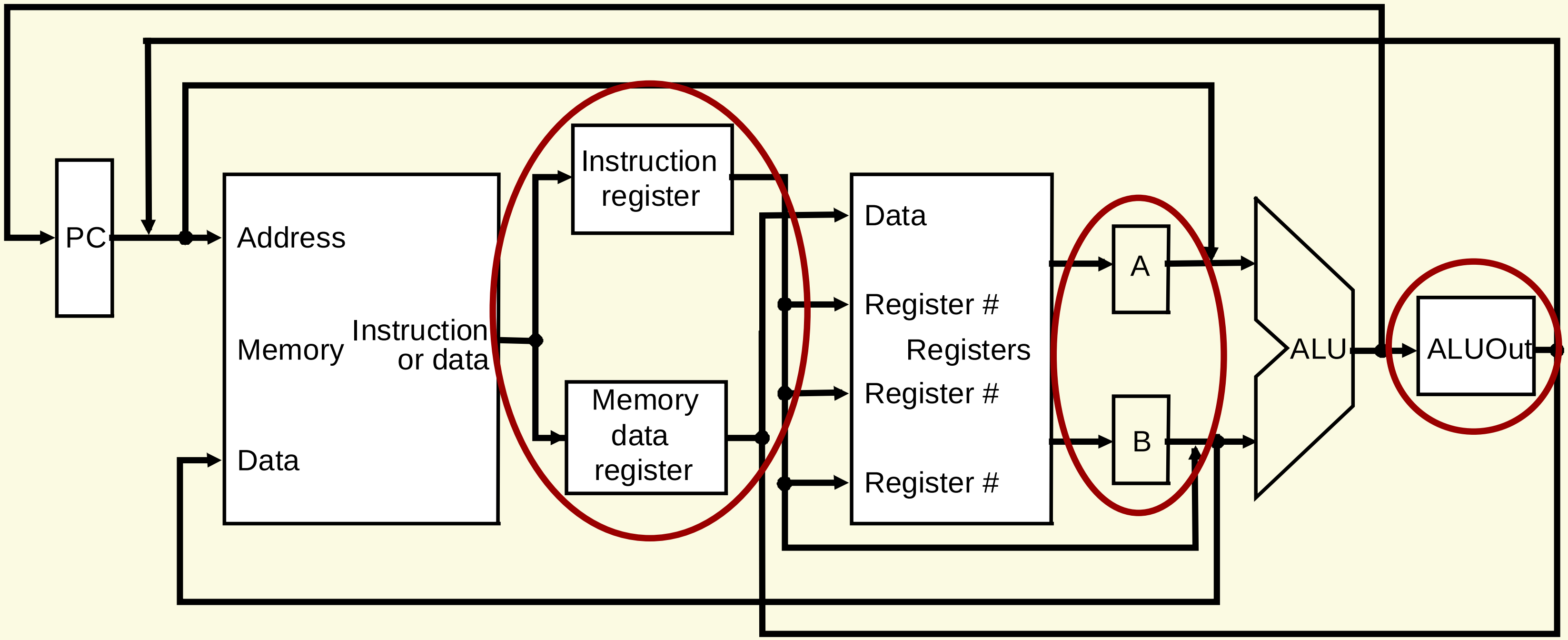
- As a result, the best solution is to consider a **shorter clock cycle** (derived from the basic functional unit delays) and allow different instructions to require **different numbers of clock cycles**.

A Multicycle Implementation

- The execution of each instruction is broken into a series of steps that correspond to the **functional unit operations**. Each step takes one clock cycle to complete.

- A single functional unit can be used more than once per instruction, as long as it is used on different clock cycles. This sharing can help to reduce the amount of hardware required. In particular,
  - A **single memory unit** is used for both instructions and data.
  - There is a **single ALU**, rather than an ALU and two adders.
  - **One or more registers are added** after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle.

High-Level View of the Multicycle Datapath



Need for Additional Registers

- **Instruction register** (IR) and **memory data register** (MDR):
  - Hold output of memory for an instruction read and a data read, respectively.
- **A** and **B registers**:
  - Hold register operand values from register file.
- **ALUOut register**:
  - Holds output of ALU.

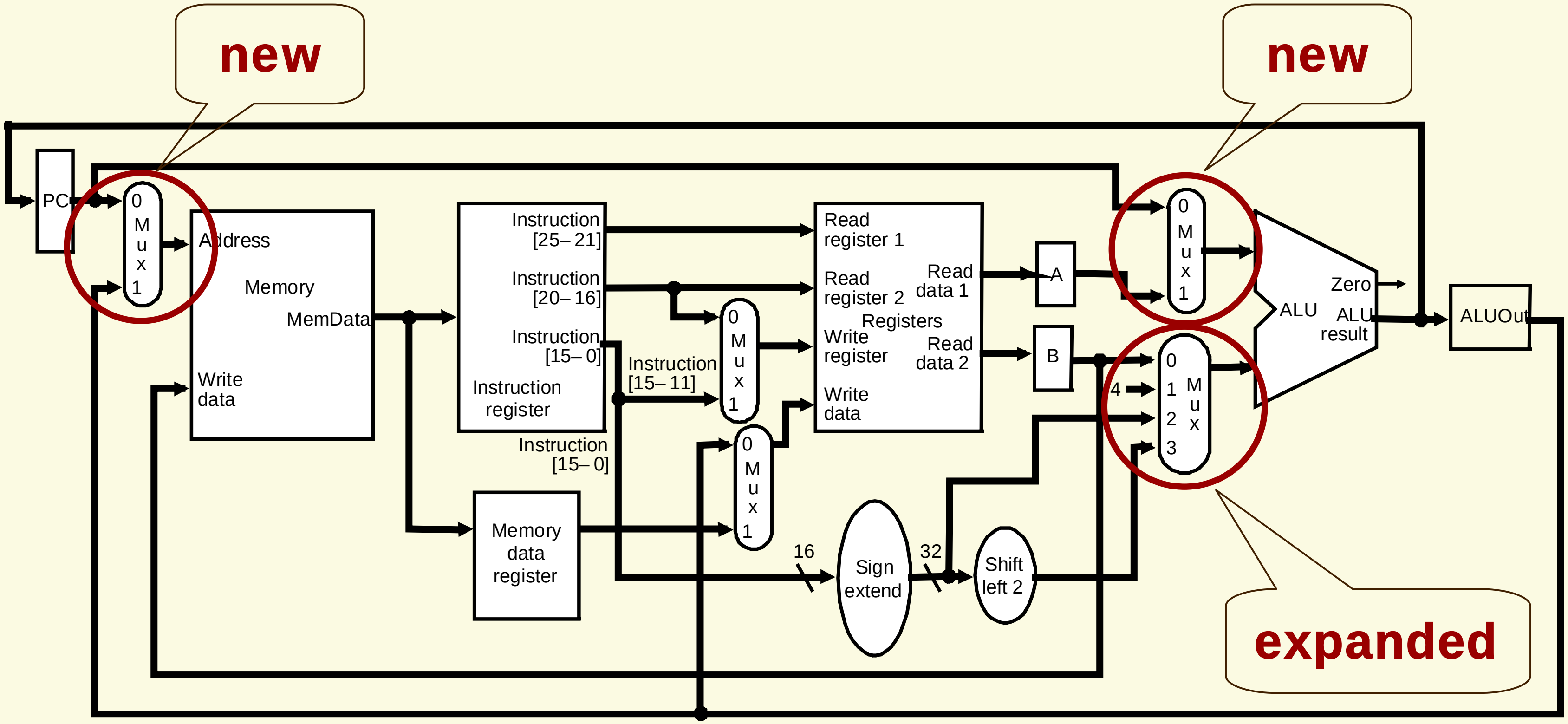
Need for Additional Control Signals

- All except the IR hold data only between a pair of adjacent clock cycles. Thus there is no need for a write control signal.

- The IR needs to hold the instruction until the end of execution of that instruction. Thus it requires a **write control signal**.

- Since several functional units are shared for different purposes, some **multiplexors** have to be added or expanded. Thus **additional control signals** are needed.

Simple Multicycle Datapath with Multiplexors Added

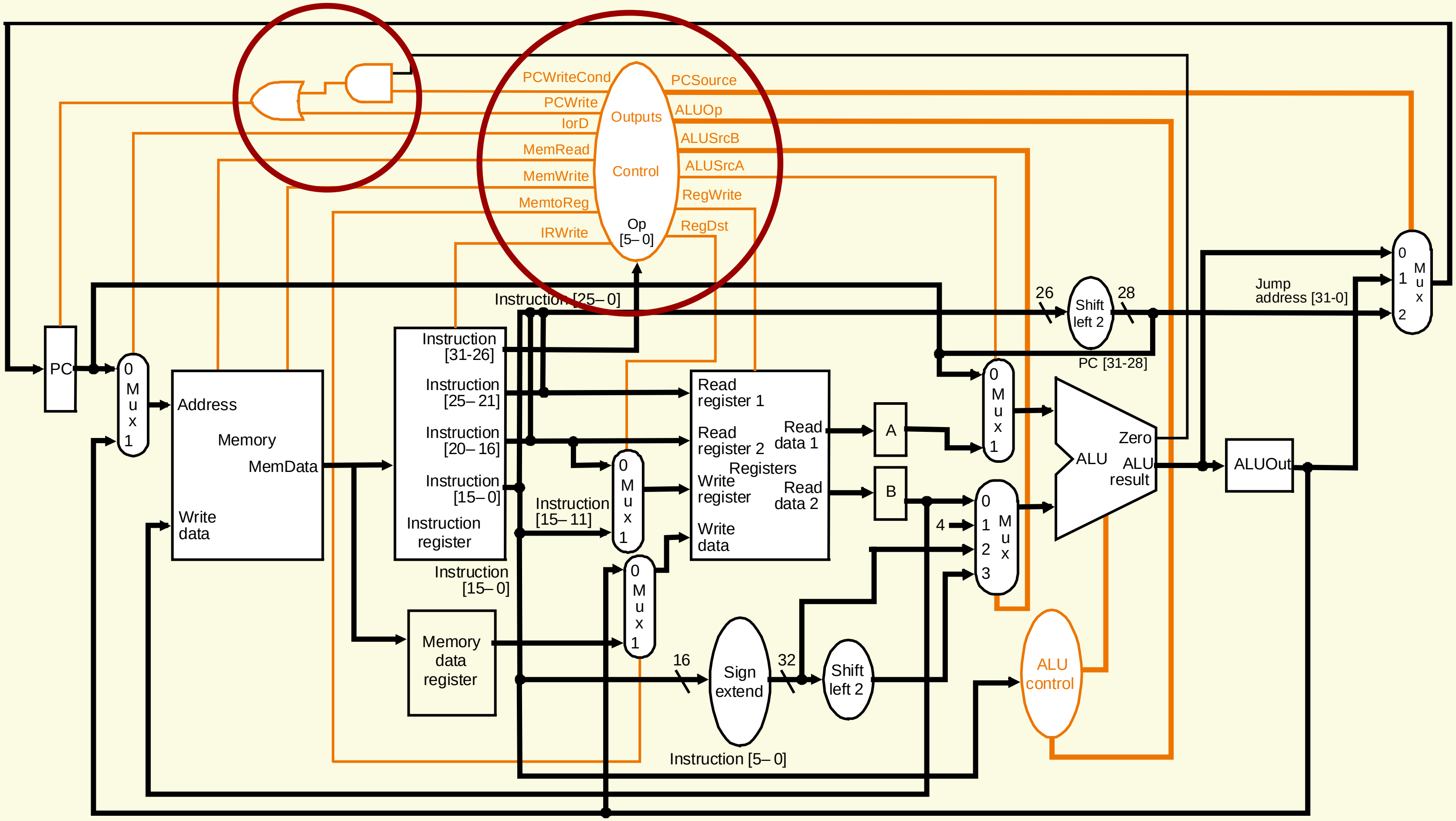


new

new

expanded

Complete Datapath and Control Unit for Multicycle Implementation



Multiple Execution Steps Per Instruction

- **Typical execution steps**:
  - **Instruction fetch**
  - **Instruction decode** and **register fetch**
  - **Execution**, **memory address computation**, or **branch completion**
  - **Memory access** or **R-type instruction completion**
  - **Memory read completion**

- Each instruction takes a few (3-5) steps.

Summary of Execution Steps

Step name	Action for R-type instructions	Action for memory reference instructions	Action for branches	Action for jumps

Instruction fetch	IR = Memory[PC]
	PC = PC + 4			

Instruction decode / register fetch	A = Reg[IR[25-21]]
	B = Reg[IR[20-16]]
	ALUOut = PC + (sign-extend(IR[15-0]) << 2)			

Execution, address computation, branch/jump completion	ALUOut = A op B	ALUOut = A + sign-extend(IR[15-0])	If (A == B) PC = ALUOut	PC = PC[31-28] || (IR[25-0] << 2)
Memory access or R-type completion	Reg[IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Instruction Fetch Step

- Fetch the instruction from memory and compute the address of the next sequential instruction:

```
IR = Memory[PC];
PC = PC + 4;
```

- **Operations**:
  - Send the PC to the memory as address.
  - Read an instruction from memory.
  - Write the instruction into the IR.
  - Increment the PC by 4.

Instruction Decode & Register Fetch Step

- Assuming the existence of two registers and an offset field (no harm to do the computation early even if they do not exist), fetch the two registers from the register file and compute the branch target address:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- **Operations**:
  - Access the register file to read rs and rt.
  - Store the results into registers A and B.
  - Compute the branch target address (sign extension and left shift).
  - Store the address in ALUOut.

Execution, Memory Address Computation, or Branch Completion Step

- In this step, the datapath operation is determined by the instruction class.

- **Memory reference instructions**:
  - Compute the memory address:

    ```
    ALUOut = A + sign-extend(IR[15-0]);
    ```

  - **Operations**:
    - Sign-extend the 16-bit offset to a 32-bit value.
    - Send both register A and the 32-bit offset to the ALU.
    - Add the two values.
    - Store the result in ALUOut.

Execution, Memory Address Computation, or Branch Completion Step

- **Arithmetic-logical (R-type) instructions**:
  - Perform the ALU operation specified by the function code:

    ```
    ALUOut = A op B;
    ```
  - **Operations**:
    - Send both registers A and B to the ALU.
    - Perform the specified operation on the two values.
    - Store the result in ALUOut.

Execution, Memory Address Computation, or Branch Completion Step

- **Branch instructions**:
  - Compare registers A and B and set the PC to the branch target address if A and B are equal:

    ```
    if (A == B)
      PC = ALUOut;
    ```

  - **Operations**:
    - Send both registers A and B to the ALU.
    - Compare A and B by performing subtraction in the ALU and set the Zero output signal to 1 if A and B are equal.
    - If Zero is equal to 1, then write ALUOut to the PC.

Execution, Memory Address Computation, or Branch Completion Step

- **Jump instructions**:
  - Compute the jump address and set the PC to this address:

    ```
    PC = PC[31-28] || (IR[25-0] << 2)
    ```

  - **Operations**:
    - Left-shift the 26-bit address field by 2 bits to give a 28-bit value.
    - Concatenate the four leftmost bits of the PC with the 28-bit value to form a 32-bit jump address.
    - Write the jump address to the PC.

Memory Access or R-Type Instruction Completion Step

- **Memory reference instructions**:
  - Read from or write to memory:

```
MDR = Memory[ALUOut];    // for load instruction
  or
Memory[ALUOut] = B;      // for store instruction
```

  - **Operations**:
    - Use the address computed during the previous step and stored in ALUOut.
    - For a load instruction, a data word is retrieved from memory with the specified address.
    - For a store instruction, a data word is written into memory with the specified address.

Memory Access or R-Type Instruction Completion Step

- **Arithmetic-logical (R-type) instructions**:
  - Write the result of the ALU operation into a destination register inside the register file:

    ```
    Reg[IR[15-11]] = ALUOut;
    ```
  - **Operations**:
    - Get from ALUOut the value which was the result of the ALU operation in the previous step.
    - Write the value into a register in the register file.

Memory Read Completion Step

- A load instruction completes by writing back the value from memory into a register in the register file:

  ```
  Reg[IR[20-16]] = MDR;
  ```

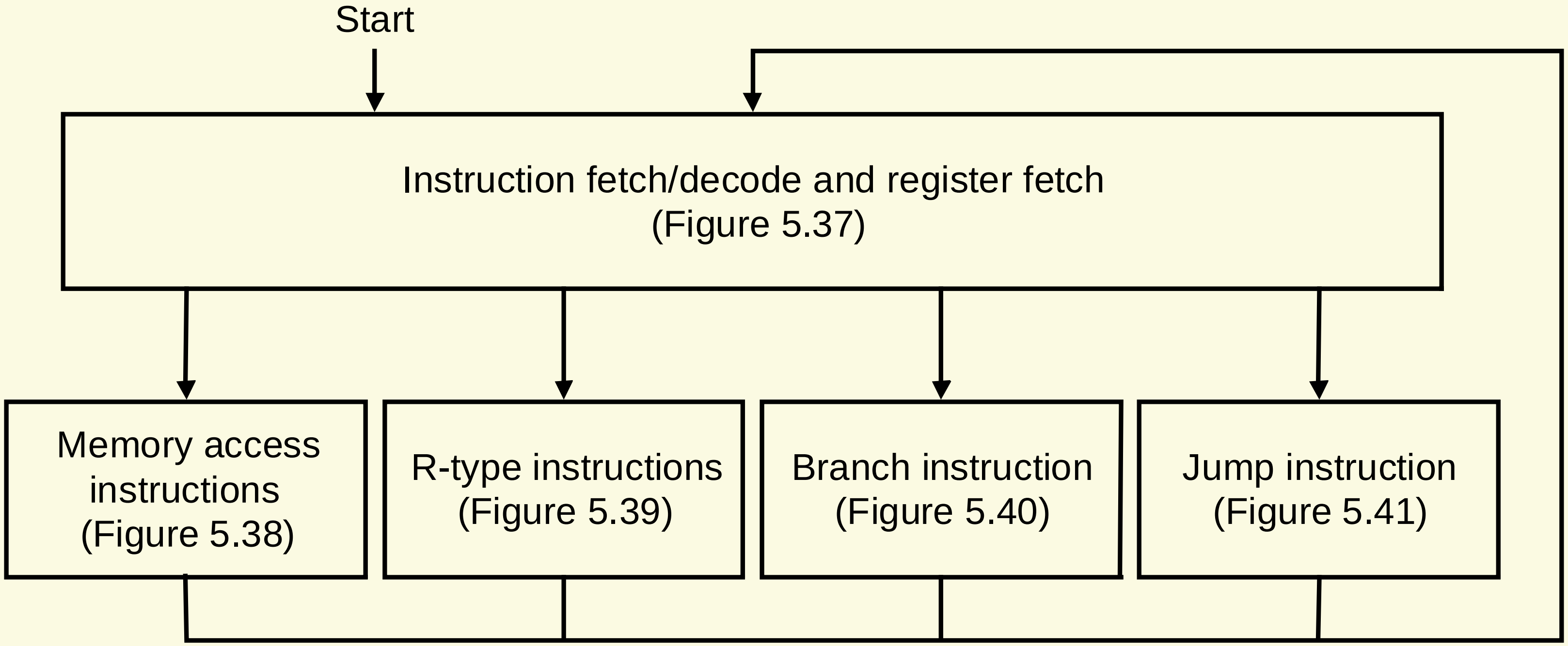
- **Operations**:
  - Get from MDR the value which was read from memory in the previous step.
  - Write the value into a register in the register file.

Control Unit Design

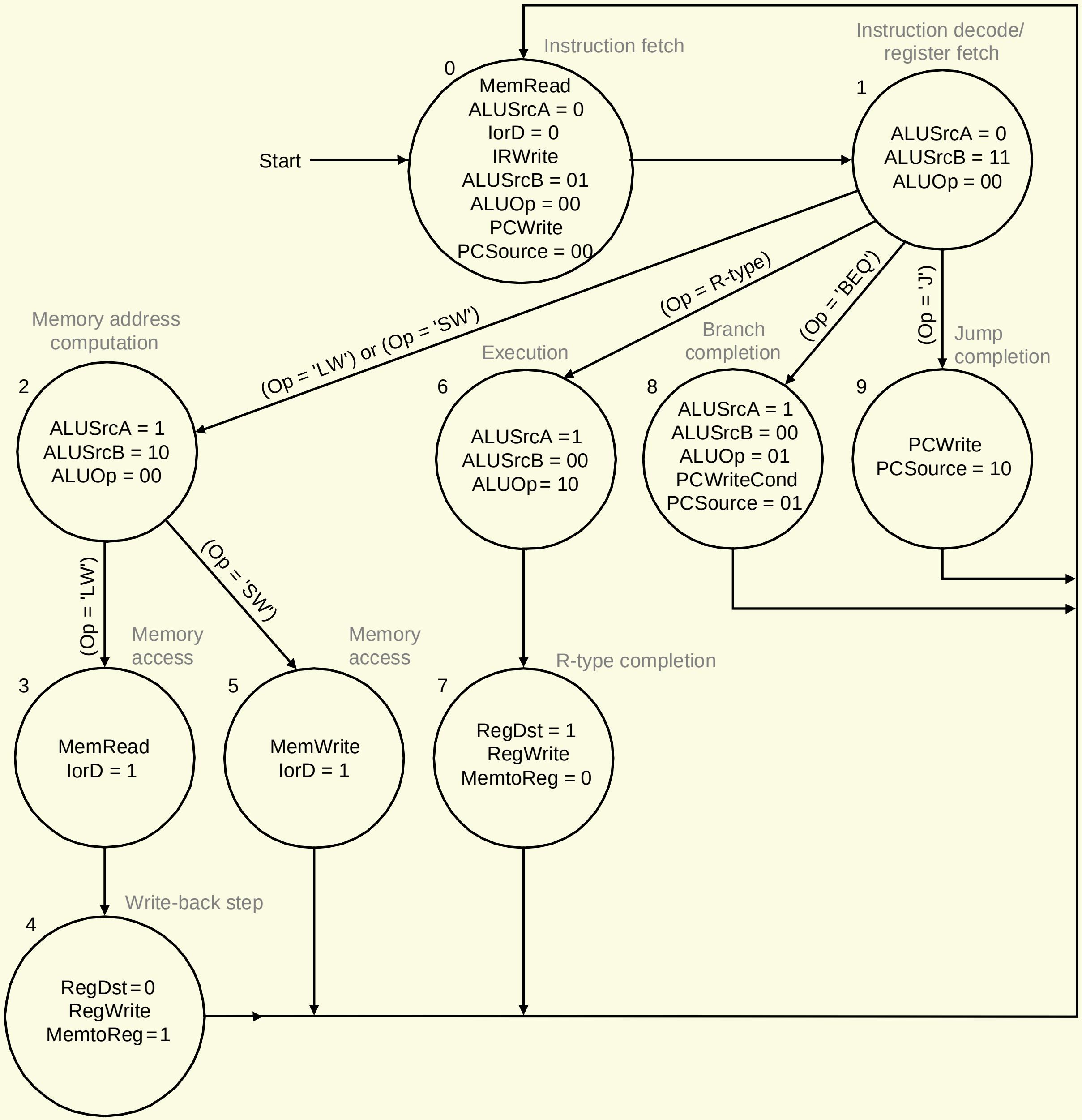
- Unlike the single-cycle implementation which only requires a set of truth tables to specify the setting of the control signals based on the instruction class, control for the multicycle implementation is more complex because an instruction is executed in a series of steps and hence the signals for both the current step and the next step have to be specified.

- **Two techniques for specifying the control**:
  - **Finite state machine representation**
  - **Microprogramming**

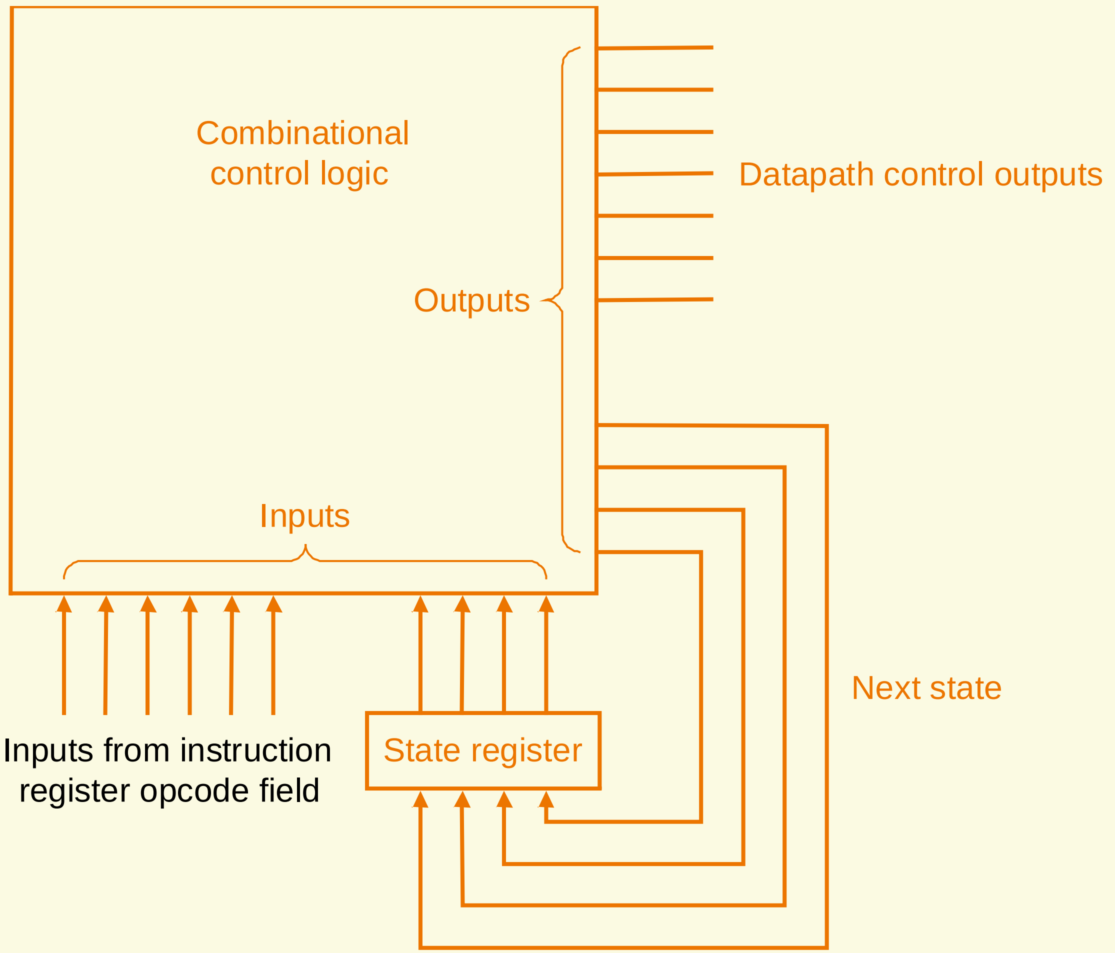
High-Level View of Finite State Machine Control for the Datapath



Complete Finite State Machine Control



Finite State Machine Controller



Simplifying Control Design by Microprogramming

- ❑ Representing the MIPS control using a finite state machine is relatively easy and feasible.

- ❑ However, in case we have a large instruction set and a large number of complex addressing modes (e.g., Intel x86/Pentium instruction set), the number of states in the finite state machine would go up to thousands, which becomes too large and cumbersome to handle.

- ❑ The solution to this problem comes from programming - **creating functions and procedures**.
  - ❍ As programs become large, additional structuring techniques are needed to keep the programs comprehensible.

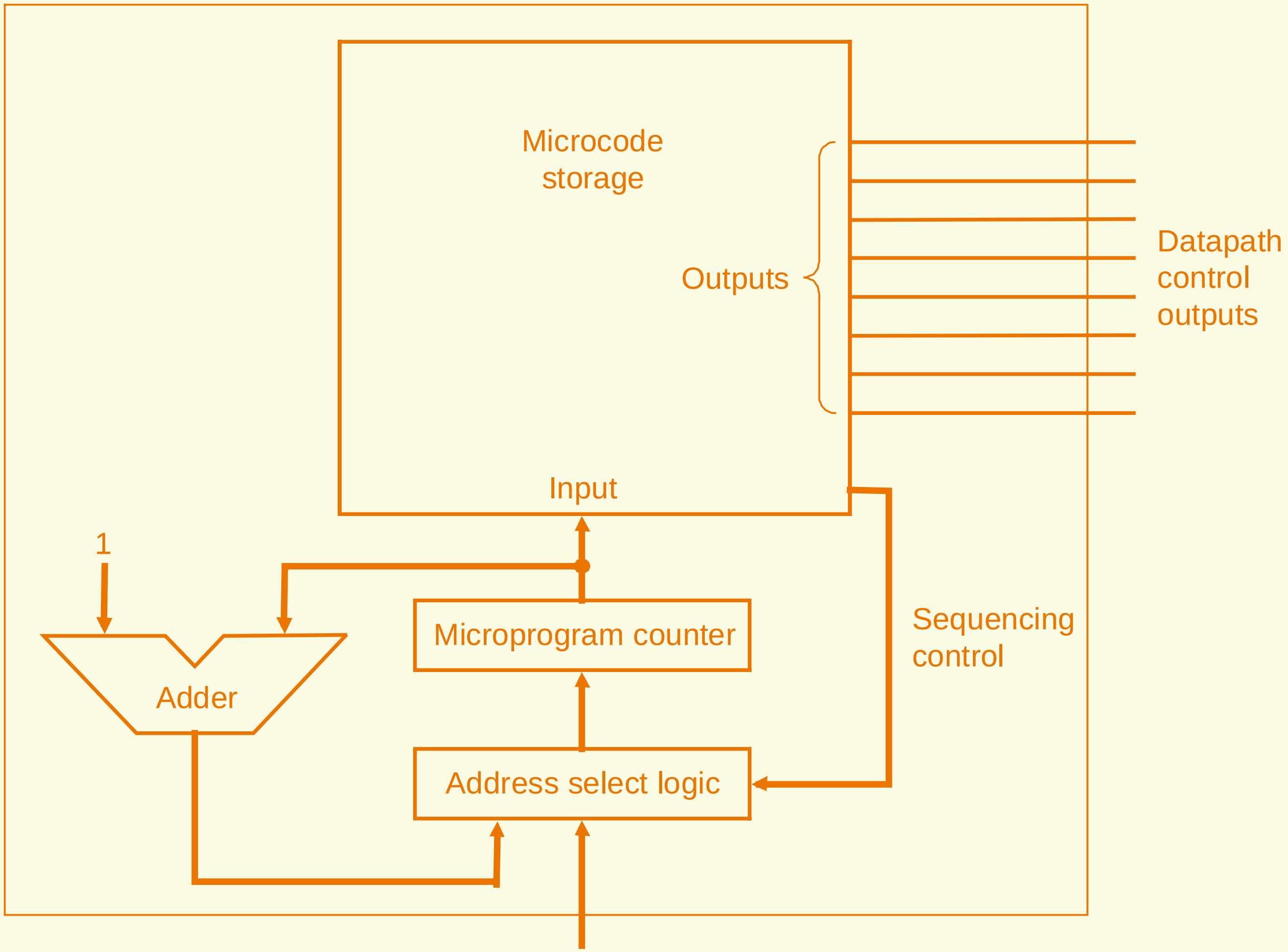
Microprogramming

- **Microinstructions**:
  - The set of control signals that must be asserted in a state can be considered as an instruction to be executed by the datapath.
  - To distinguish such low-level control instructions from instructions of the MIPS instruction set, they are called **microinstructions**.
  - Executing a microinstruction has the effect of asserting the control signals specified by the microinstruction.

- **Microprogramming**: designing the control unit as a program that implements machine instructions in terms of simpler microinstructions.

- The asserted values on the control lines are represented symbolically in a **microprogram**, which is a representation of the microinstructions.

Microcode Controller



Inputs from instruction register opcode field

Key Concepts to Remember

- **Sequential logic circuits** are needed for the implementation of **state elements** such as registers and memory units.

- **Latches** and **flip-flops** are building blocks for state elements such as registers and some types of RAM.

- A **register file** is a structure in the datapath consisting of a set of registers and some read and write ports.

- There are two types of RAM: **static RAM** and **dynamic RAM**.

- A **two-stage decoding process** is typically used in the addressing scheme of RAMs.

Key Concepts to Remember

- A **single-cycle implementation** of the dathpath and control assumes that each instruction takes only one clock cycle.

- **Disadvantages of single-cycle implementation**: long execution time and no sharing of hardware.

- A **multicycle implementation** of the datapath and control allows different instructions to take different numbers of clock cycles.

- Specification of the control for a multicycle implementation can be done using the **finite state machine** representation (for simple instruction sets) or through **microprogramming** (for complex instruction sets).