VietNam National University
University of Engineering and Technology

THIẾT KẾ MẠCH TÍCH HỢP SỐ
(DIGITAL IC DESIGN)

TS. Nguyễn Kiêm Hùng
Email: kiemhung@vnu.edu.vn

Lecture 3.2: CPU Architecture and Design

Objectives

In this lecture you will be introduced to:

- **General-purpose Processor Architecture**
- **Techniques for designing General-purpose processor at RTL**

Outline

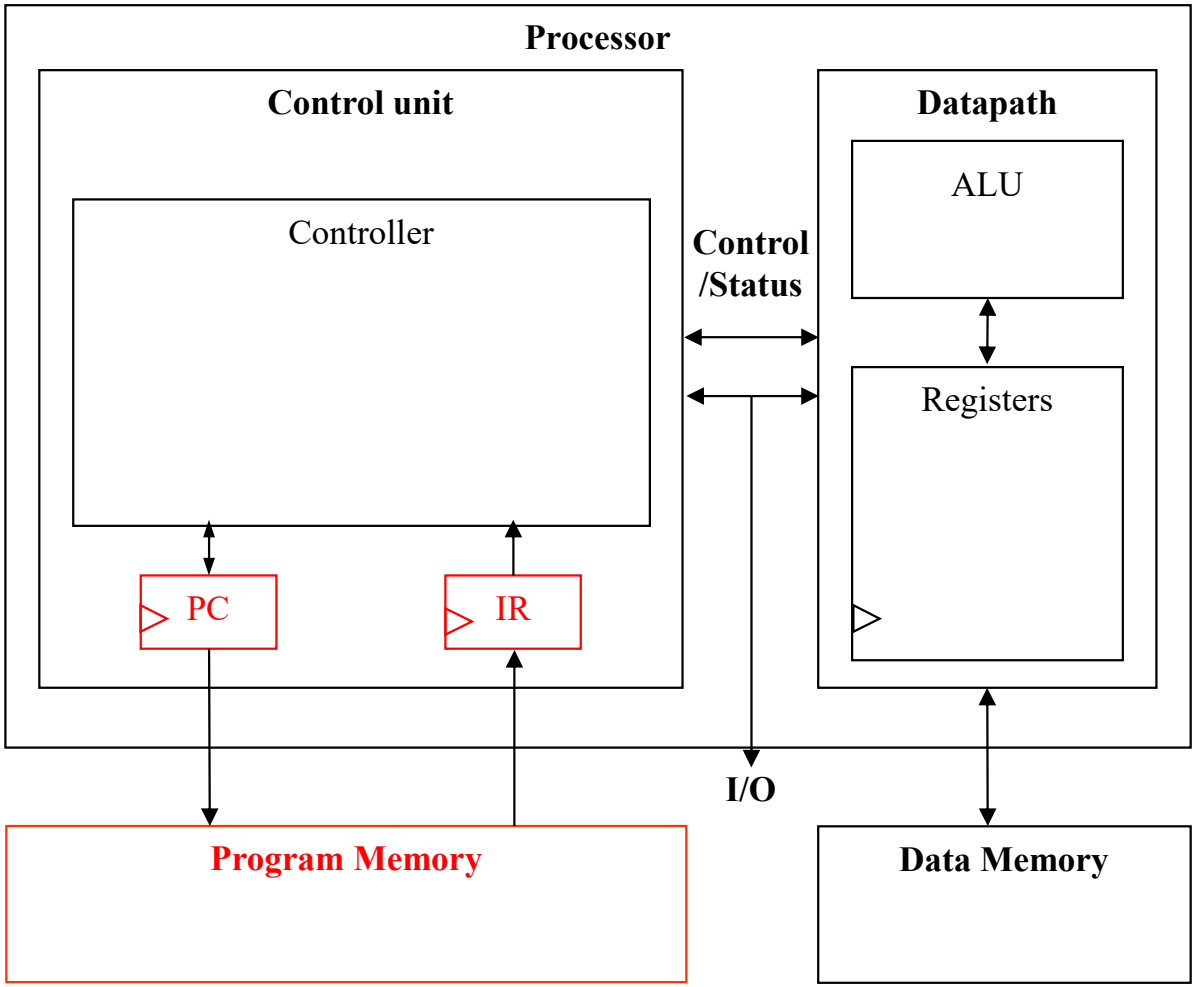
- **Processor Architecture**
- Programmer's Model
- RT-level custom general-purpose processor design

Introduction

- General-Purpose Processor
  - Processor designed for a variety of computation tasks
  - Low unit cost, because manufacturer spreads NRE (Non-recurring engineering) over large numbers of units
    - Two billions ARM-based microcontrollers was sold *in 2008*
  - Carefully designed since higher NRE is acceptable
    - Can yield good performance, size and power
  - Low NRE cost, short time-to-market/prototype, high flexibility
    - User just writes software; no processor design
  - a.k.a. “microprocessor” – “micro” used when they were implemented on one or a few chips rather than entire rooms

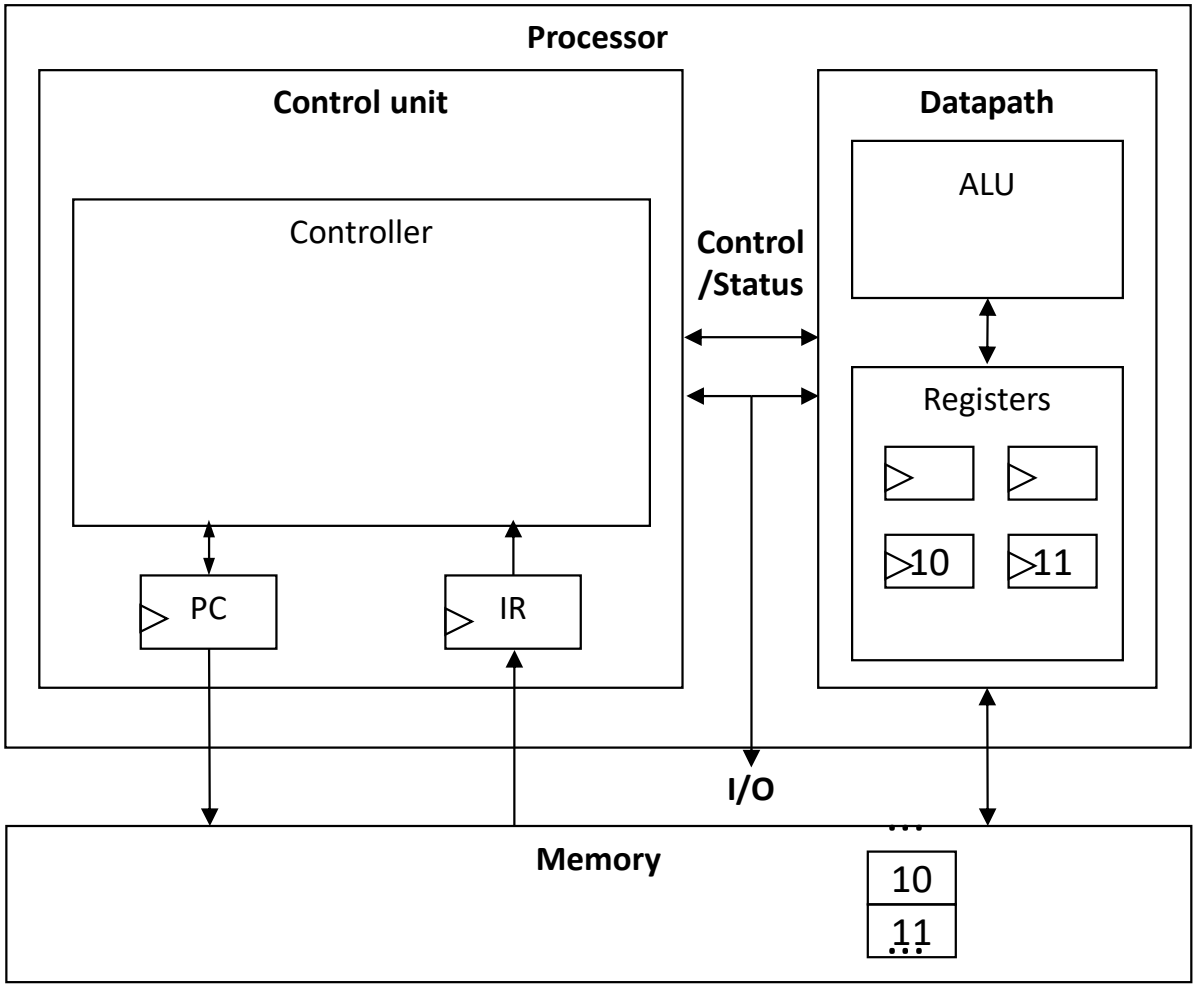
Basic Architecture

- Control unit and datapath
  - Note similarity to single-purpose processor
- Key differences
  - Datapath is general
  - Control unit doesn't store the algorithm – the algorithm is "programmed" into the memory
  - CPU registers: program counter (PC), instruction register (IR), general-purpose registers, etc.



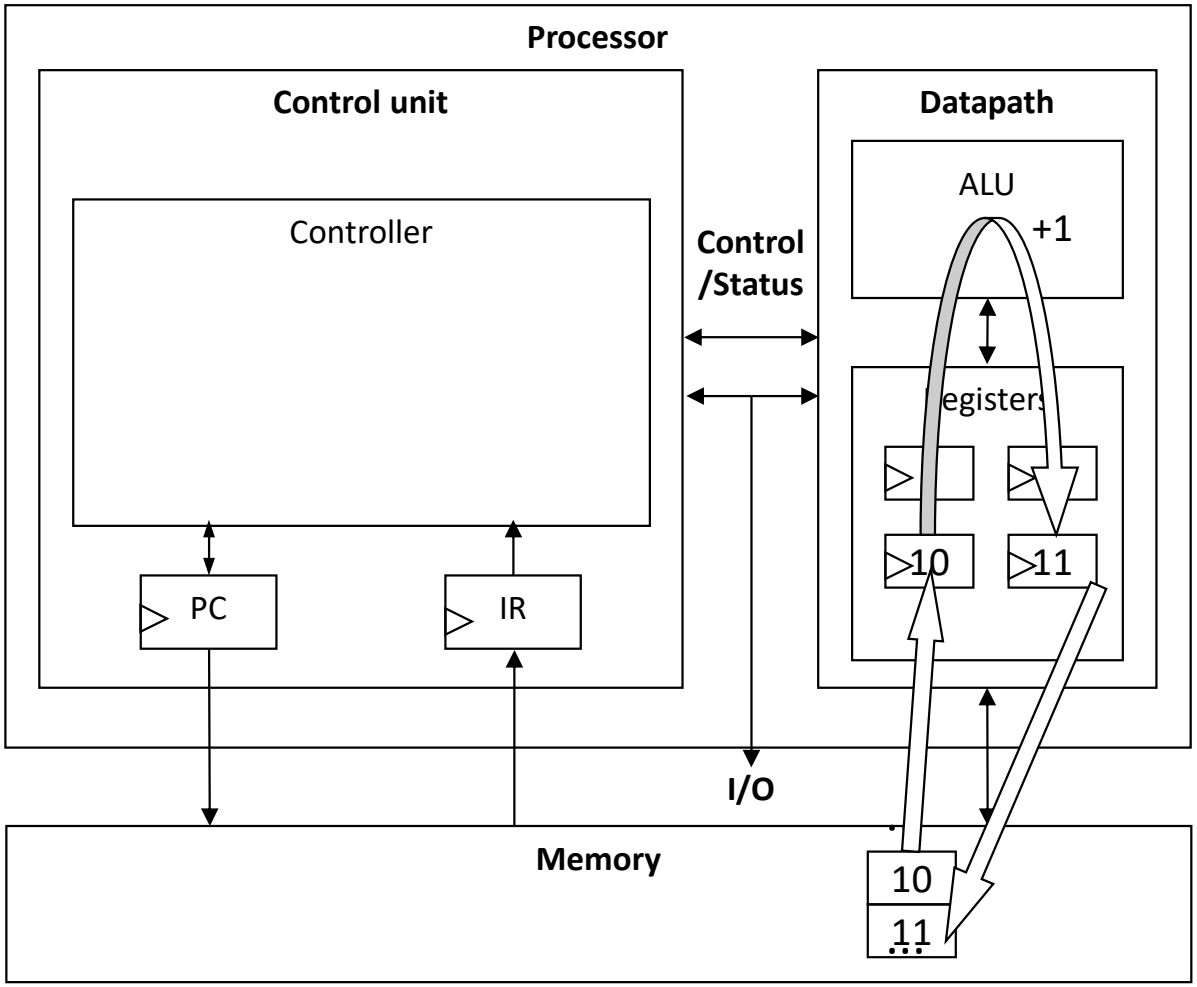
Datapath

- **Function**
  - transforming data and storing temporary data
- **Structure:**
  - ALU (arithmetic-logic unit): transforming data through operations
  - Registers: storing temporary data
  - internal data bus



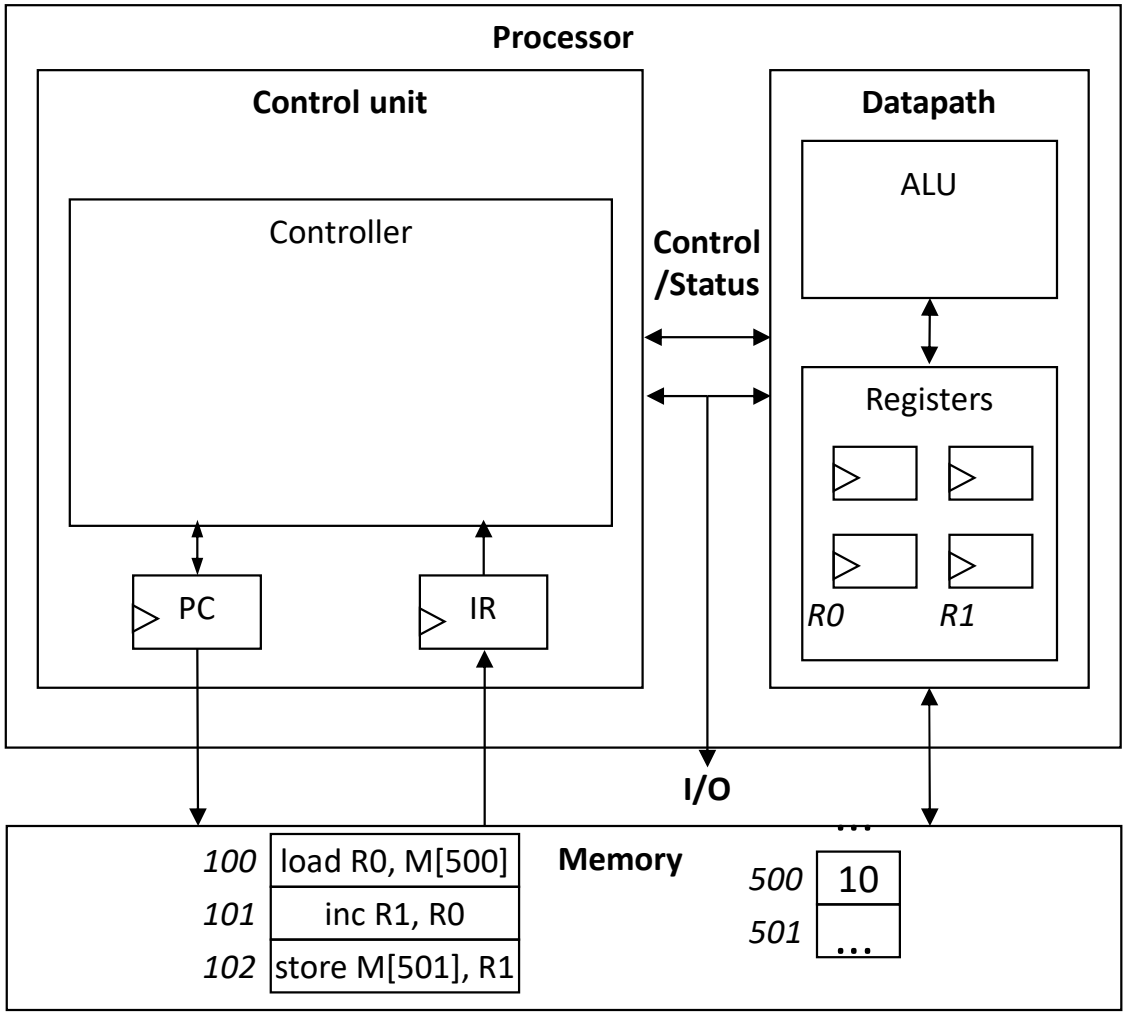
Datapath Operations

- Load
  - Read memory location into register
- ALU operation
  - Input certain registers through ALU, store back in register
- Store
  - Write register to memory location



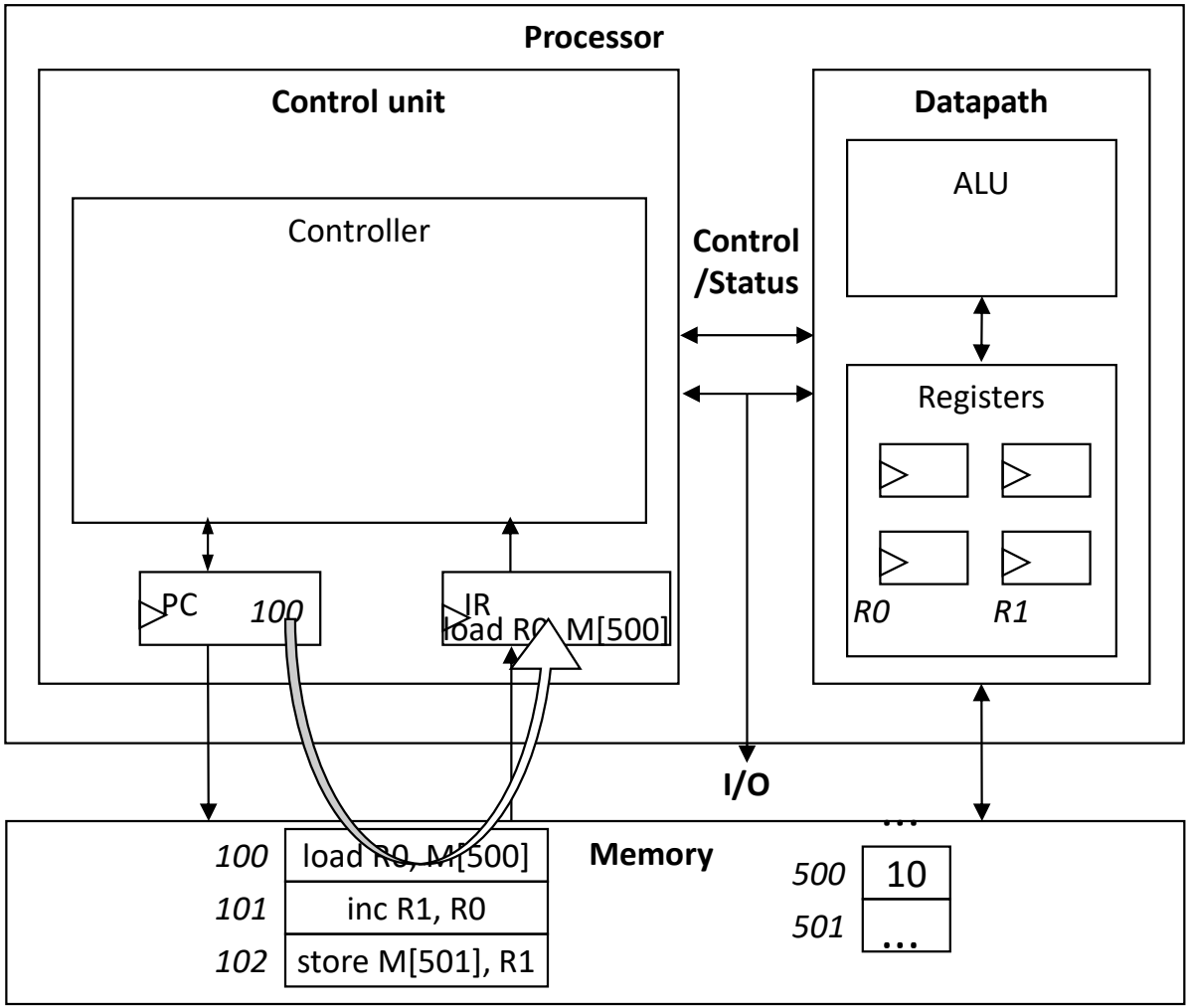
Control Unit

- **Control unit**: configures the datapath operations
  - Sequence of desired operations ("instructions") stored in memory – "Program"
- **Instruction cycle** – broken into several sub-operations, each one clock cycle, e.g.:
  - *Fetch*: Get next instruction into IR
  - *Decode*: Determine what the instruction means
  - *Fetch operands*: Move data from memory to datapath register
  - *Execute:* Move data through the ALU
  - *Store results*: Write data from register to memory



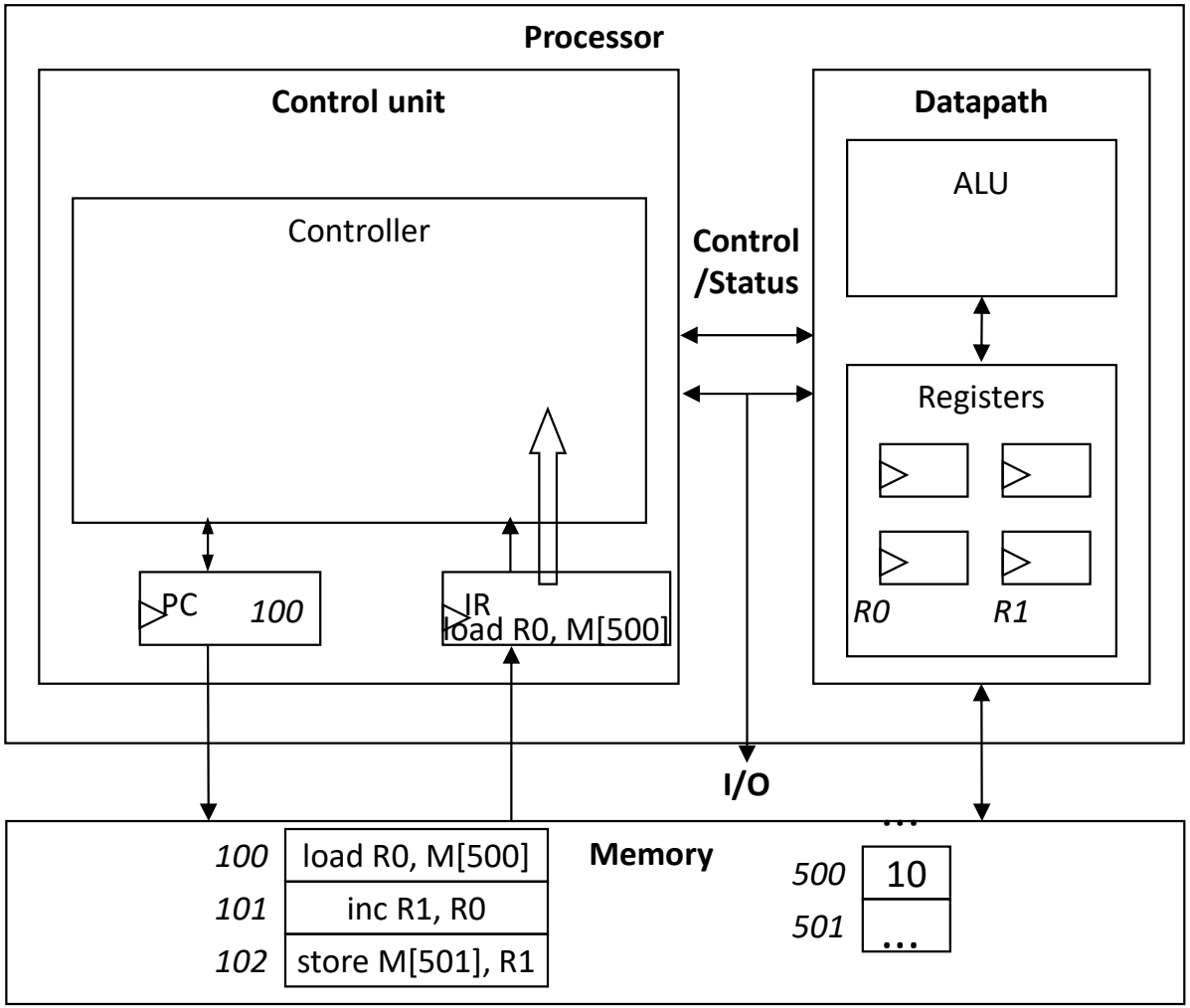
Control Unit Sub-Operations

- **Fetch**
  - Get next instruction into IR
  - PC: program counter, always points to next instruction
  - IR: Instructions Register, holds the fetched instruction



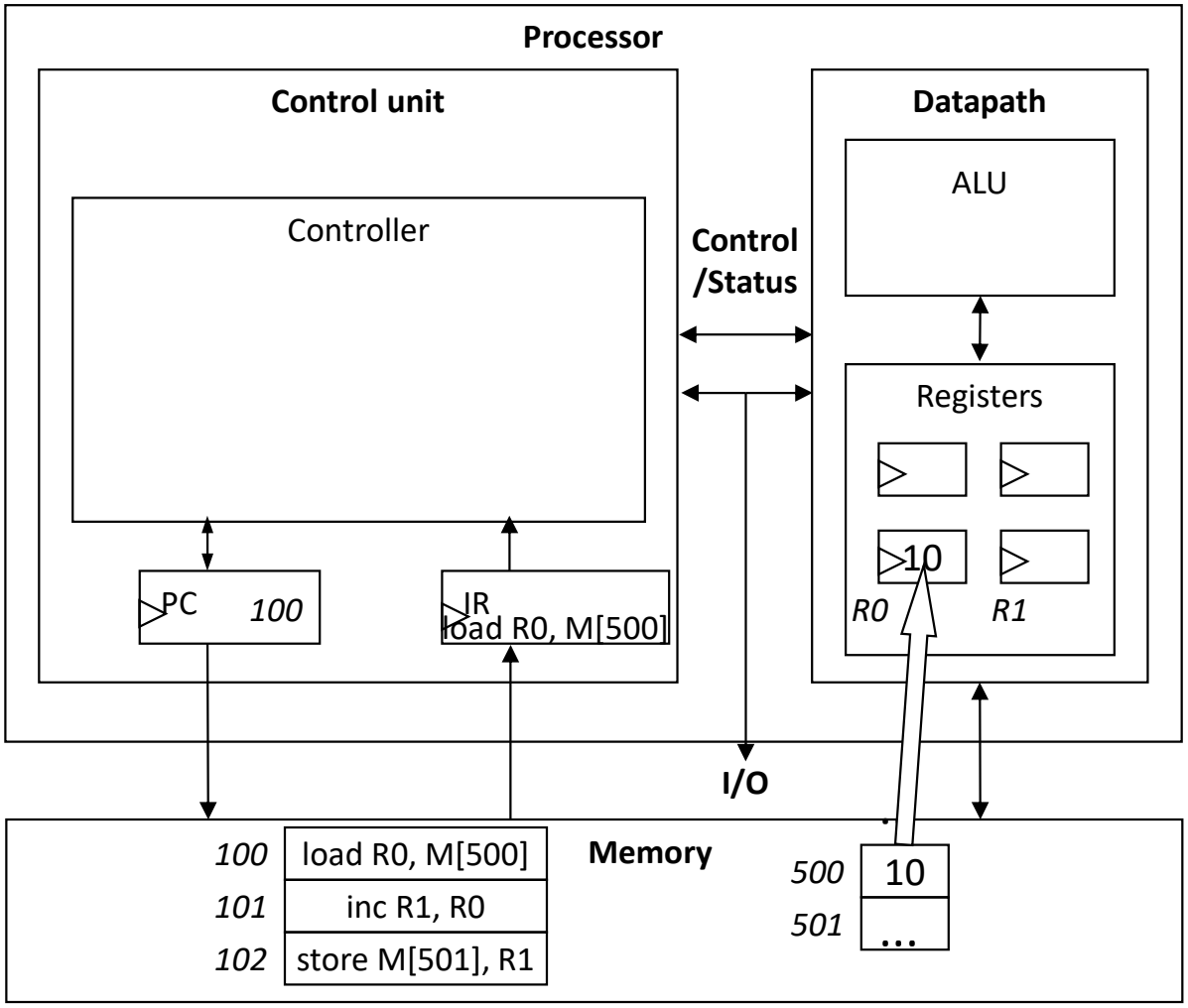
Control Unit Sub-Operations

- **Decode**
  - Determine what the instruction means



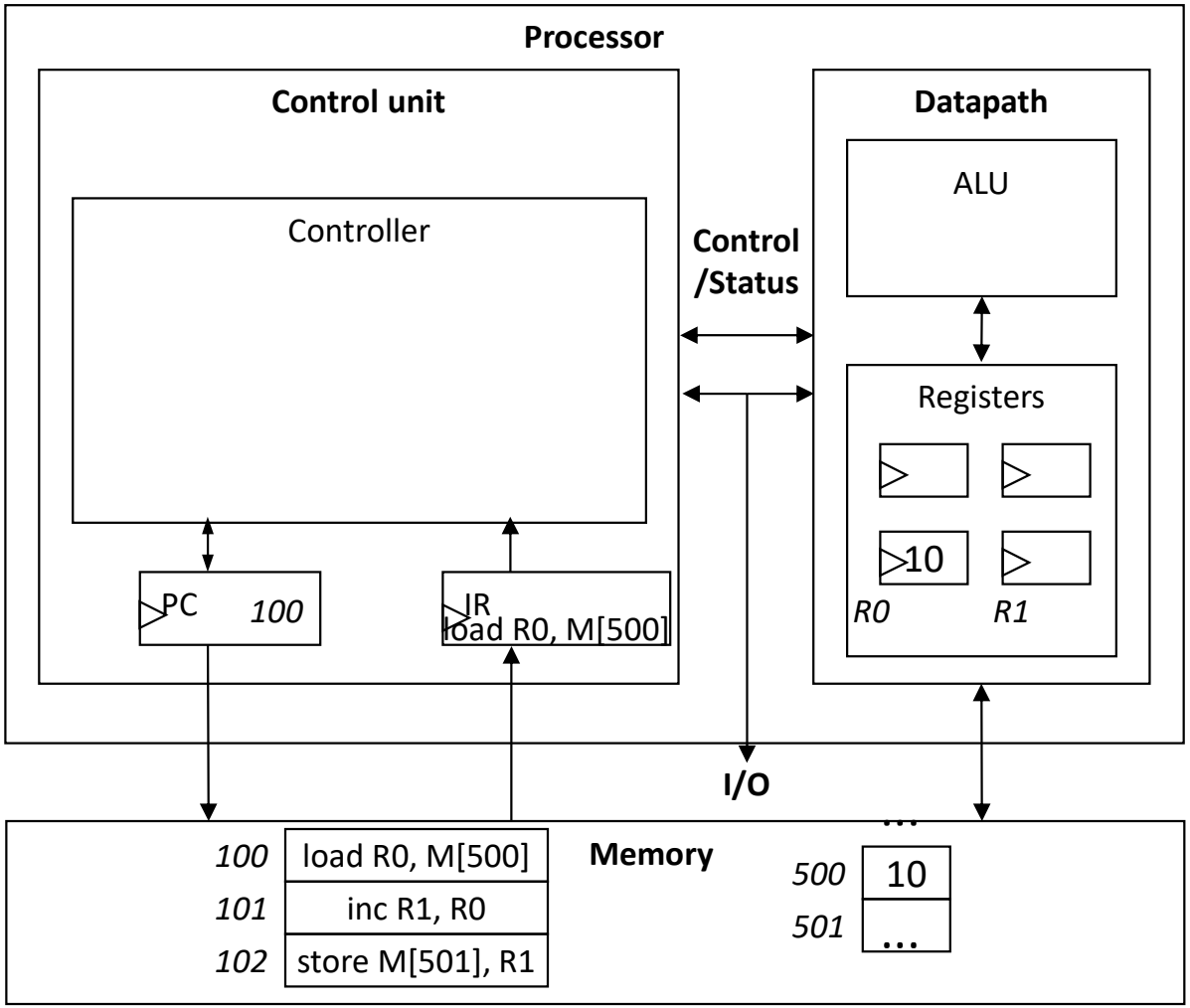
Control Unit Sub-Operations

- **Fetch operands**
  - Move data from memory to datapath register



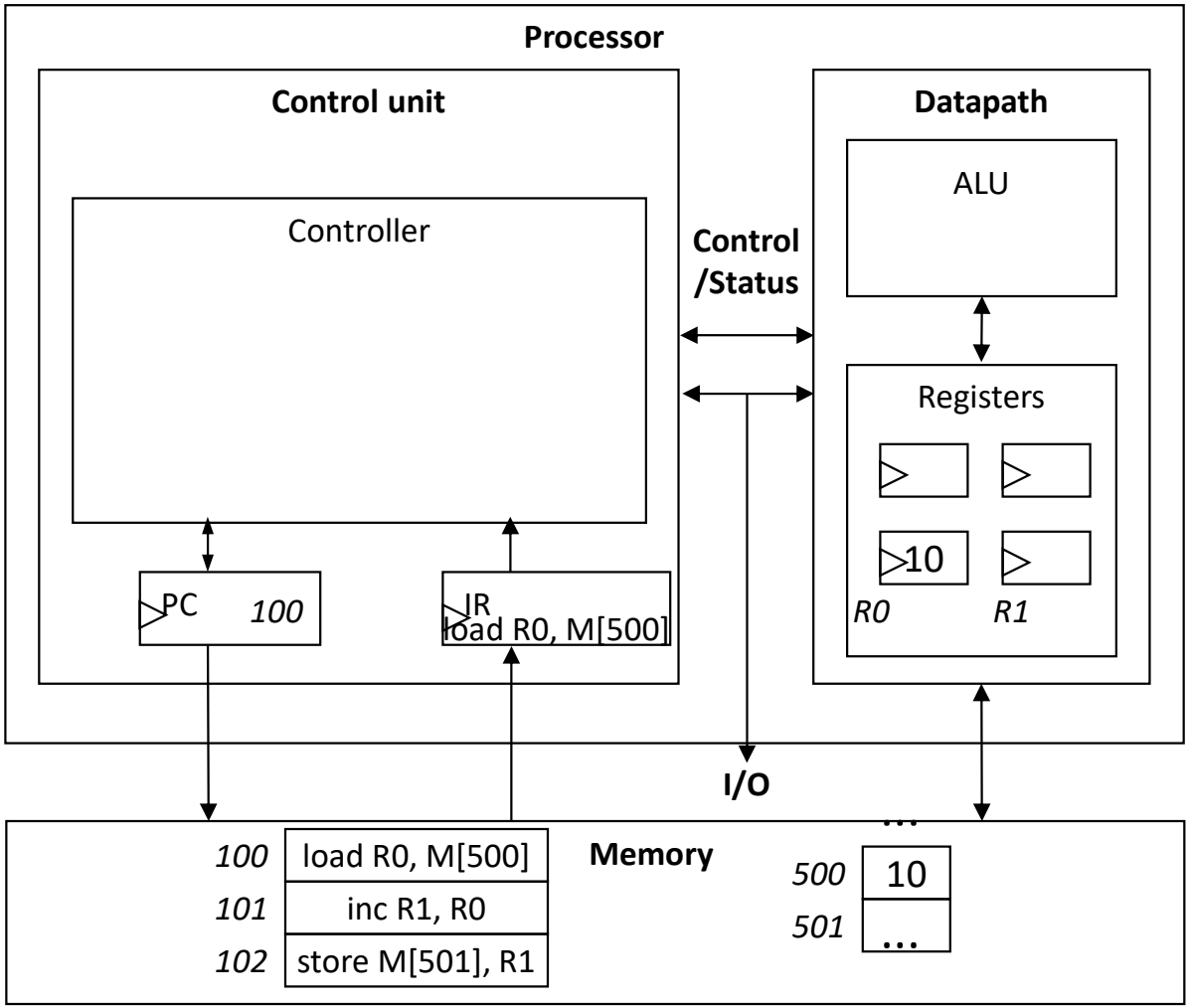
Control Unit Sub-Operations

- **Execute**
  - Move data through the ALU
  - This particular instruction does nothing during this sub-operation



Control Unit Sub-Operations

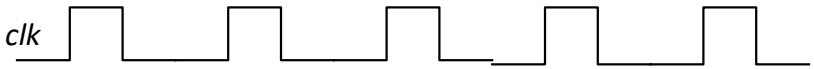
- **Store results**
  - Write data from register to memory
  - This particular instruction does nothing during this sub-operation

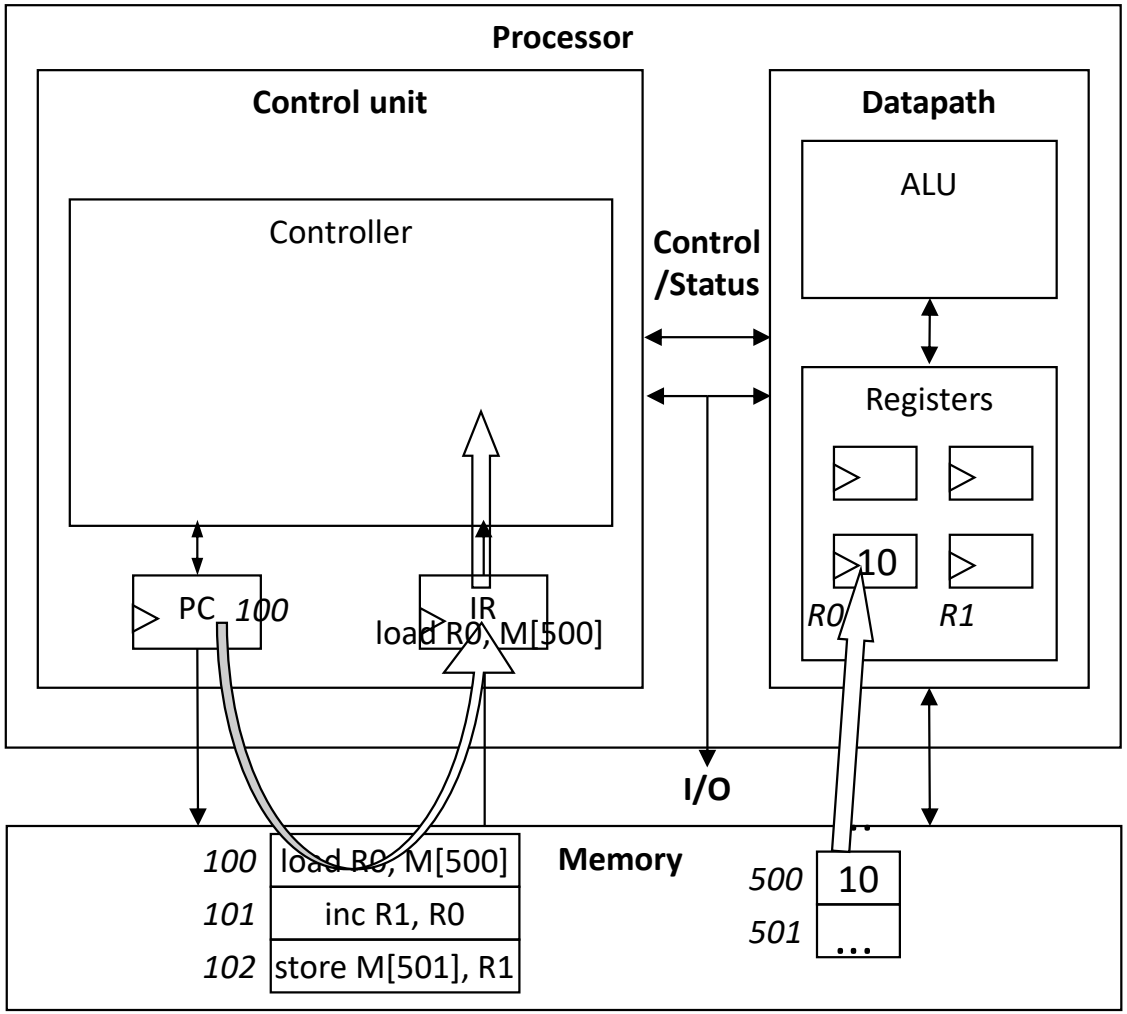


Instruction Cycles

PC=100

Fetch Decode Fetch ops Exec. Store results





Instruction Cycles

PC=100

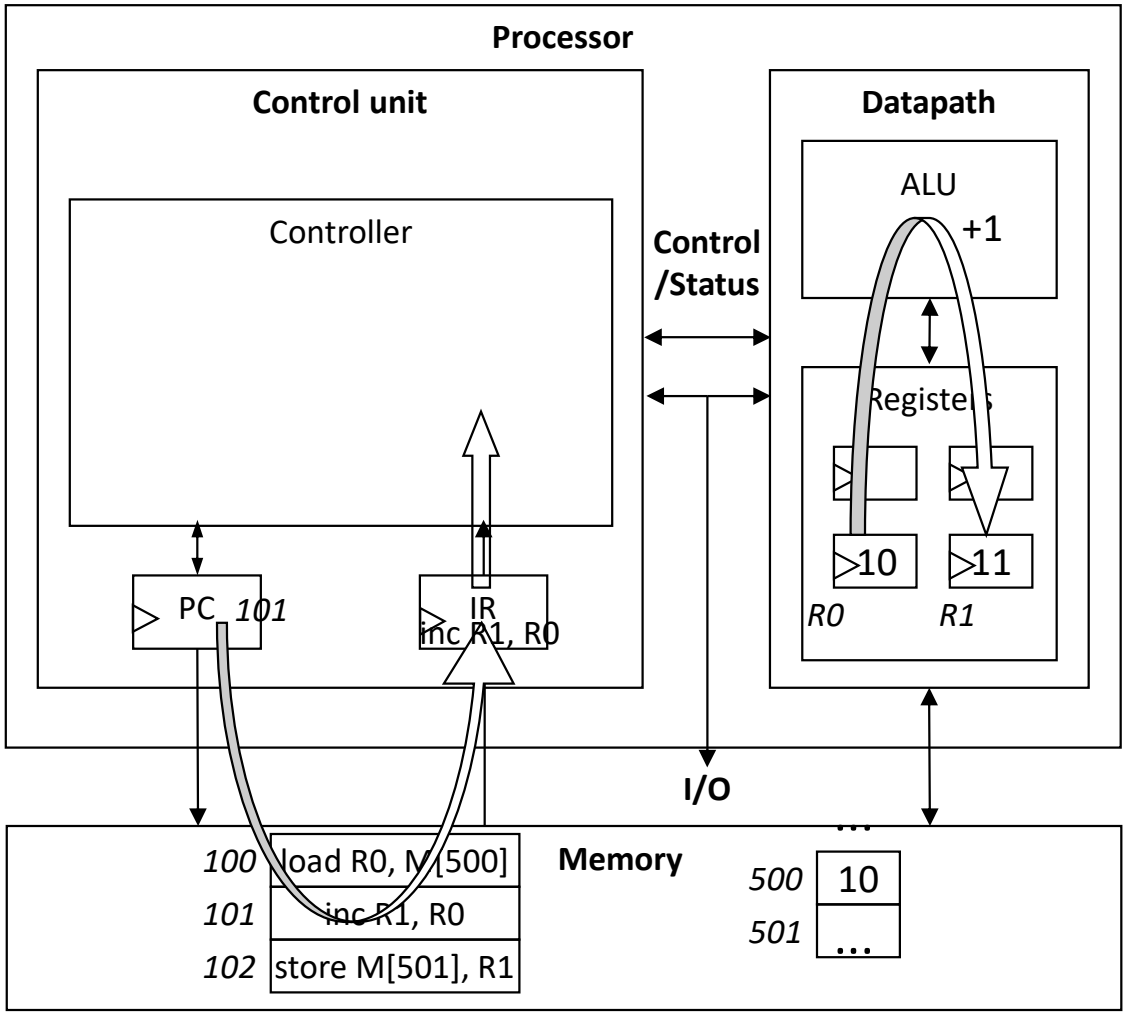
Fetch | Decode | Fetch ops | Exec. | Store results

clk

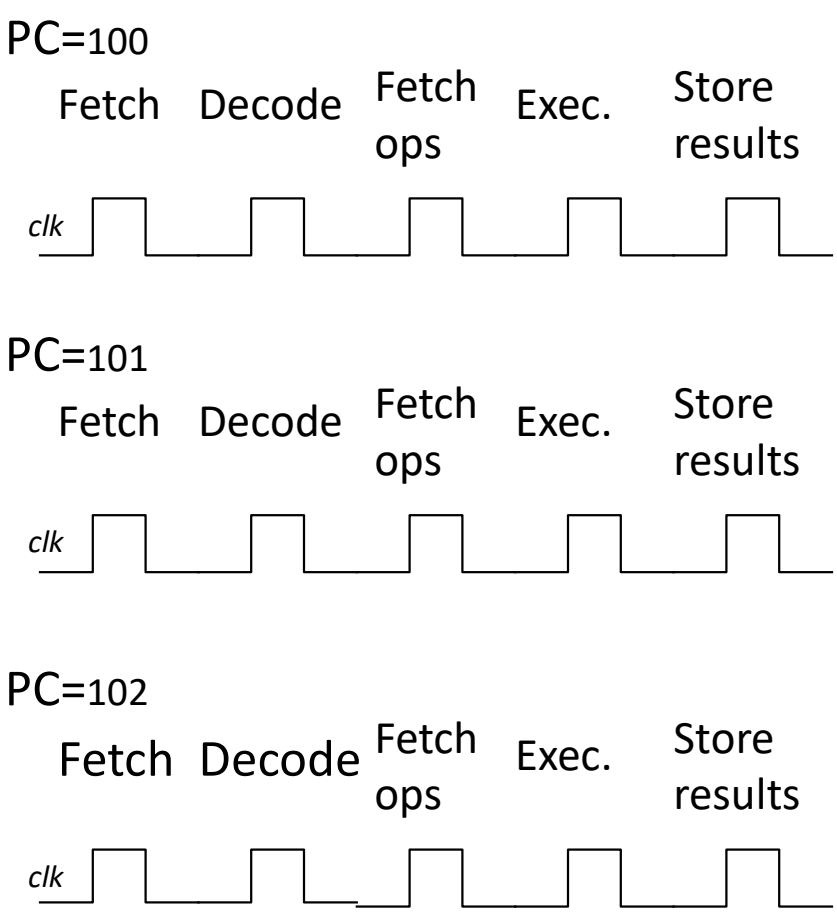
PC=101

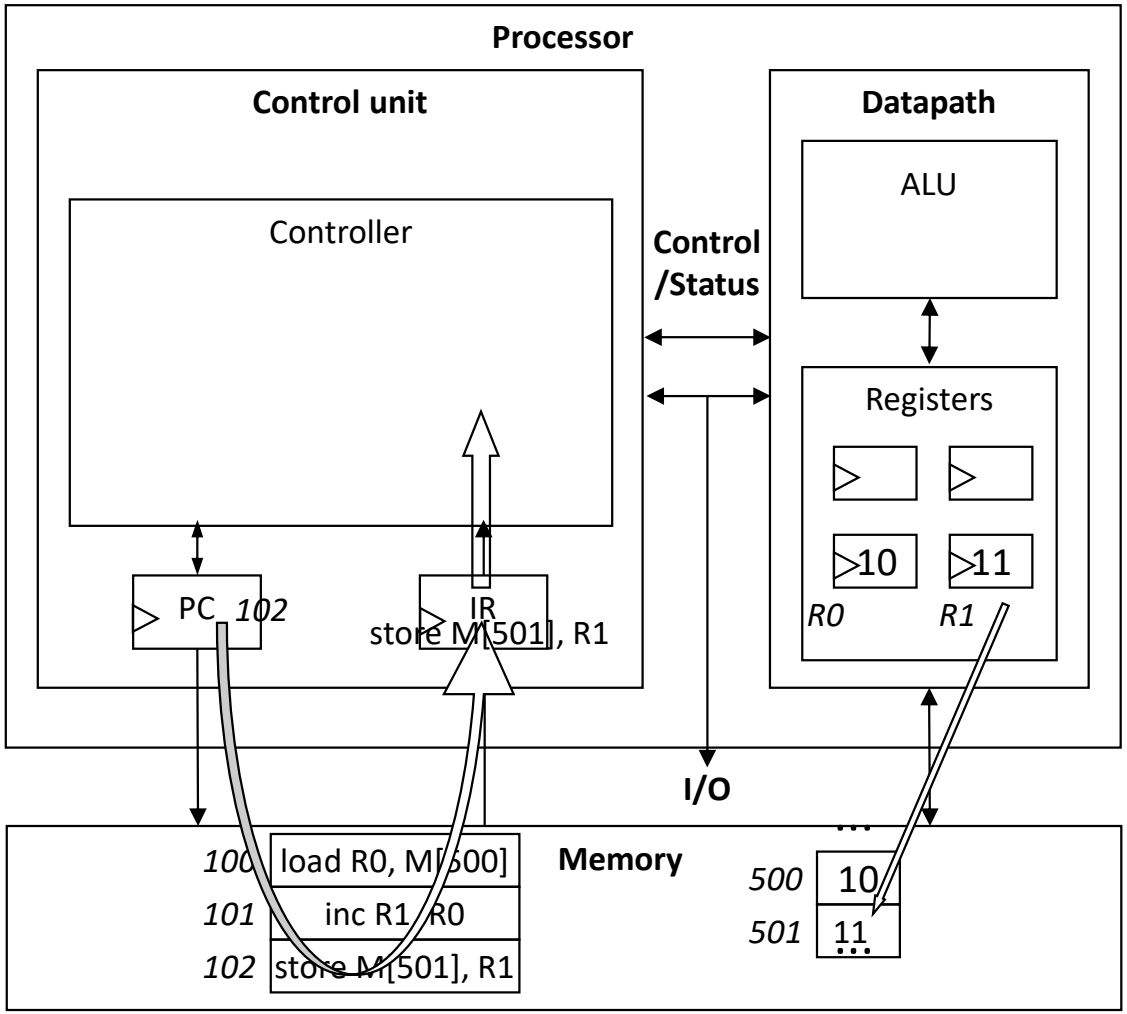
Fetch | Decode | Fetch ops | Exec. | Store results

clk



Instruction Cycles





Outline

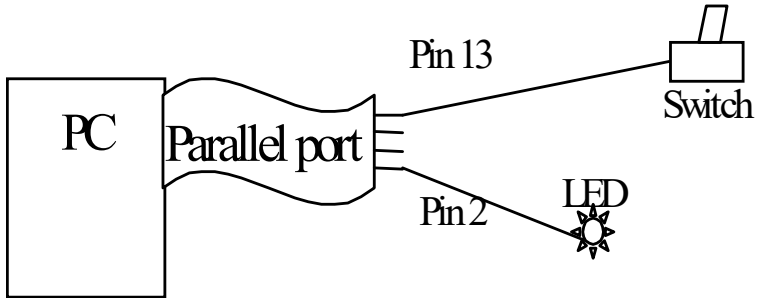
- Processor Architecture
- **Programmer's View**
- RT-level custom general-purpose processor design

Programmer's View (1)

- Programmer doesn't need detailed understanding of processor's architecture
  - Instead, needs to know what instructions can be executed
- Two levels of instructions:
  - Assembly level
  - Structured languages (C, C++, Java, etc.)
- Most development today done using structured languages
  - But, some assembly level programming may still be necessary
  - Drivers: portion of program that communicates with, and/or drives, peripheral devices
    - Often have detailed timing considerations, extensive bit manipulation
    - Assembly level may be best for these

Example: parallel port driver

LPT Connection Pin	I/O Direction	Register Address
1	Output	0^{th} bit of register #2
2-9	Output	0^{th} ~7^{th} bit of register #0
10,11,12,13,15	Input	6,7,5,4,3^{th} bit of register #1
14,16,17	Output	1,2,3^{th} bit of register #2

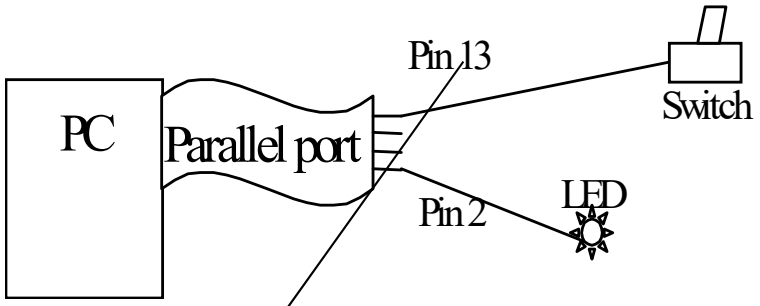


- Using assembly language programming we can configure a PC parallel port to perform digital I/O
  - write and read to three special registers to accomplish this table provides list of parallel port connector pins and corresponding register location
  - Example : parallel port monitors the input switch and turns the LED on/off accordingly

Parallel Port Example

```
; This program consists of a sub-routine that reads
; the state of the input pin, determining the on/off state
; of our switch and asserts the output pin, turning the LED
; on/off accordingly
        .386

CheckPort     proc
        push    ax              ; save the content
        push    dx              ; save the content
        mov     dx, 3BCh + 1    ; base + 1 for register #1
        in      al, dx          ; read register #1
        and     al, 10h         ; mask out all but bit # 4
        cmp     al, 0           ; is it 0?
        jne     SwitchOn        ; if not, we need to turn the LED on

SwitchOff:
        mov     dx, 3BCh + 0    ; base + 0 for register #0
        in      al, dx          ; read the current state of the port
        and     al, fEh         ; clear first bit (masking)
        out     dx, al          ; write it out to the port
        jmp     Done            ; we are done

SwitchOn:
        mov     dx, 3BCh + 0    ; base + 0 for register #0
        in      al, dx          ; read the current state of the port
        or      al, 01h         ; set first bit (masking)
        out     dx, al          ; write it out to the port

Done:   pop     dx              ; restore the content
        pop     ax              ; restore the content
CheckPort     endp
```

```
extern "C" CheckPort(void);        // defined in
                                   // assembly
void main(void) {
        while( 1 ) {
                CheckPort();
        }
}
```



LPT Connection Pin	I/O Direction	Register Address
1	Output	0th bit of register #2
2-9	Output	0th ~7th bit of register #0
10,11,12,13,15	Input	6,7,5,4,3th bit of register #1
14,16,17	Output	1,2,3th bit of register #2

Assembly language

- Textual description of instructions, instead of their binary representation.
- Basic features:
  - One instruction per line.
  - Labels provide names for addresses (usually in first column).
  - Instructions often start in later columns.
  - Columns run to end of line.

ARM assembly language example

```
label1  ADR r4,c
        LDR r0,[r4] ; a comment
        ADR r4,d
        LDR r1,[r4]
        SUB r0,r0,r1 ; comment
```

Instruction Set

- **The Instruction Set defines:**
  - the interface between software modules and the underlying hardware;
  - what the hardware will do under certain circumstances.
- **Characteristics:**
  - fixed versus variable length;
  - addressing modes;
  - numbers of operands;
  - types of operations supported

Instruction Set

- Why do we need to know the instruction set when designing embedded systems?
  - Most software design effort for many systems is possible to be implemented in high-level languages and knowledge of the instruction set is not required
- But…..
  - Embedded systems require initialisation code and interrupt routines
  - the instruction set is the key to analyzing the performance of programs; Performance gains can be made by writing assembly routines
  - All systems require debugging – possibly at the instruction level
  - Some features of the processor architecture are not available with compilers
  - Etc.

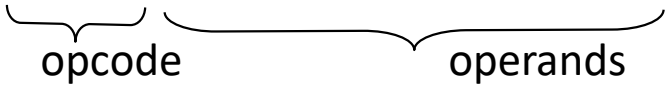
Instruction Set

Instruction 1	opcode	operand1	operand2
Instruction 2	opcode	operand1	operand2
Instruction 3	opcode	operand1	operand2
Instruction 4	opcode	operand1	operand2
	…		

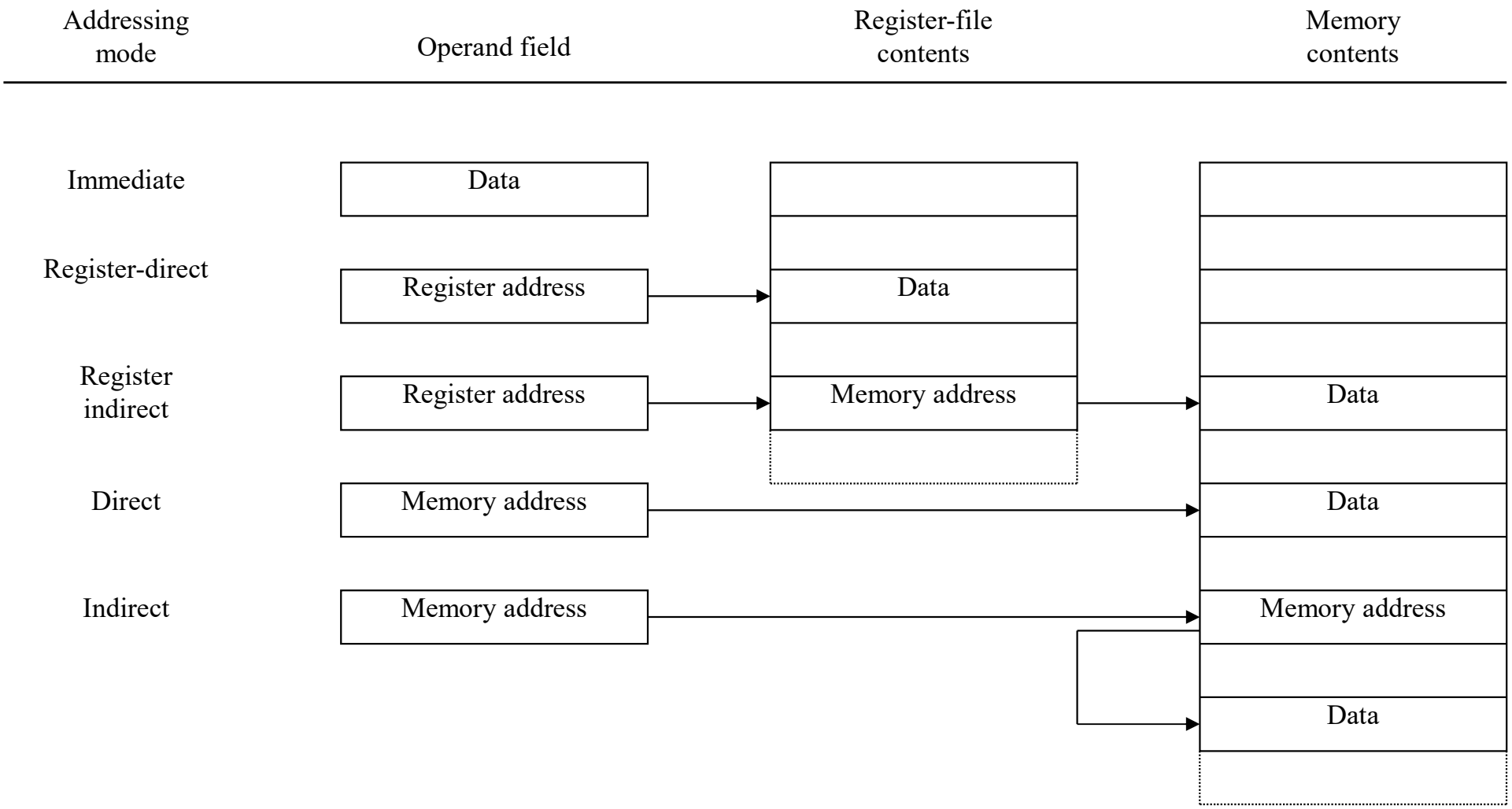
- Instruction Set
  - Defines the legal format of instructions for that processor
  - *An opcode* specifies the operation to take place during the instruction:
    - **Data transfer:** memory/register, register/register, I/O, etc.
    - **Arithmetic/logical:** move register through ALU and back
    - **Branches:** determine next PC value when not just PC+1
  - An *operand* field specifies the location of the actual data that takes part in an operation.

A Simple (Trivial) Instruction Set

Assembly instruct.	First byte	Second byte	Operation
MOV Rn, direct	0000 | Rn	direct	Rn = M(direct)
MOV direct, Rn	0001 | Rn	direct	M(direct) = Rn
MOV @Rn, Rm	0010 | Rn	Rm	M(Rn) = Rm
MOV Rn, @Rm	0111 | Rn	Rm	Rn = M(Rm)
MOV Rn, #immed.	0011 | Rn	immediate	Rn = immediate
ADD Rn, Rm	0100 | Rn	Rm	Rn = Rn + Rm
SUB Rn, Rm	0101 | Rn	Rm	Rn = Rn - Rm
JZ Rn, Address	0110 | Rn	Address	PC = Address (only if Rn = 0)
Jmp Address	1010 | Rn	Address	PC = Address (always)



Addressing Modes



Sample Programs

C program

```
int total = 0;
for (int i=10; i!=0; i--)
  total += i;
// next instructions...
```

Equivalent assembly program

```
      0   MOV R0, #0;      // total = 0
      1   MOV R1, #10;     // i = 10
      2   MOV R2, #1;      // constant 1
      3
Loop: JZ R1, Next;         // Done if i=0
      4   ADD R0, R1;      // total += i
      5   SUB R1, R2;      // i--
      6   JMP Loop;        // Jump always
Next: // next instructions...
```

- Exercise: Try some others
  - Handshake: Wait until the value of M[254] is 0, set M[255] to 1, wait until M[254] is 1, set M[255] to 0 (assume those locations are ports: M[254] -> Switch; M[255] -> LED).
  - (Harder) Count the occurrences of zero in an array stored in memory locations 100 through 199.

Programmer's Model (2)

- Program and data memory space
  - Embedded processors often have very limited memory for program and for data
    - e.g., 64 Kbytes program, 256 bytes of RAM (expandable)
- Registers: How many are there?
  - Which registers is visible to the programmer.
- I/O
  - How communicate with external signals?
- Interrupts allow a device to change the flow of control in the CPU and jump to *Interrupt Service Routine* (ISR)
  - Which is the types of interrupts supported by the processor

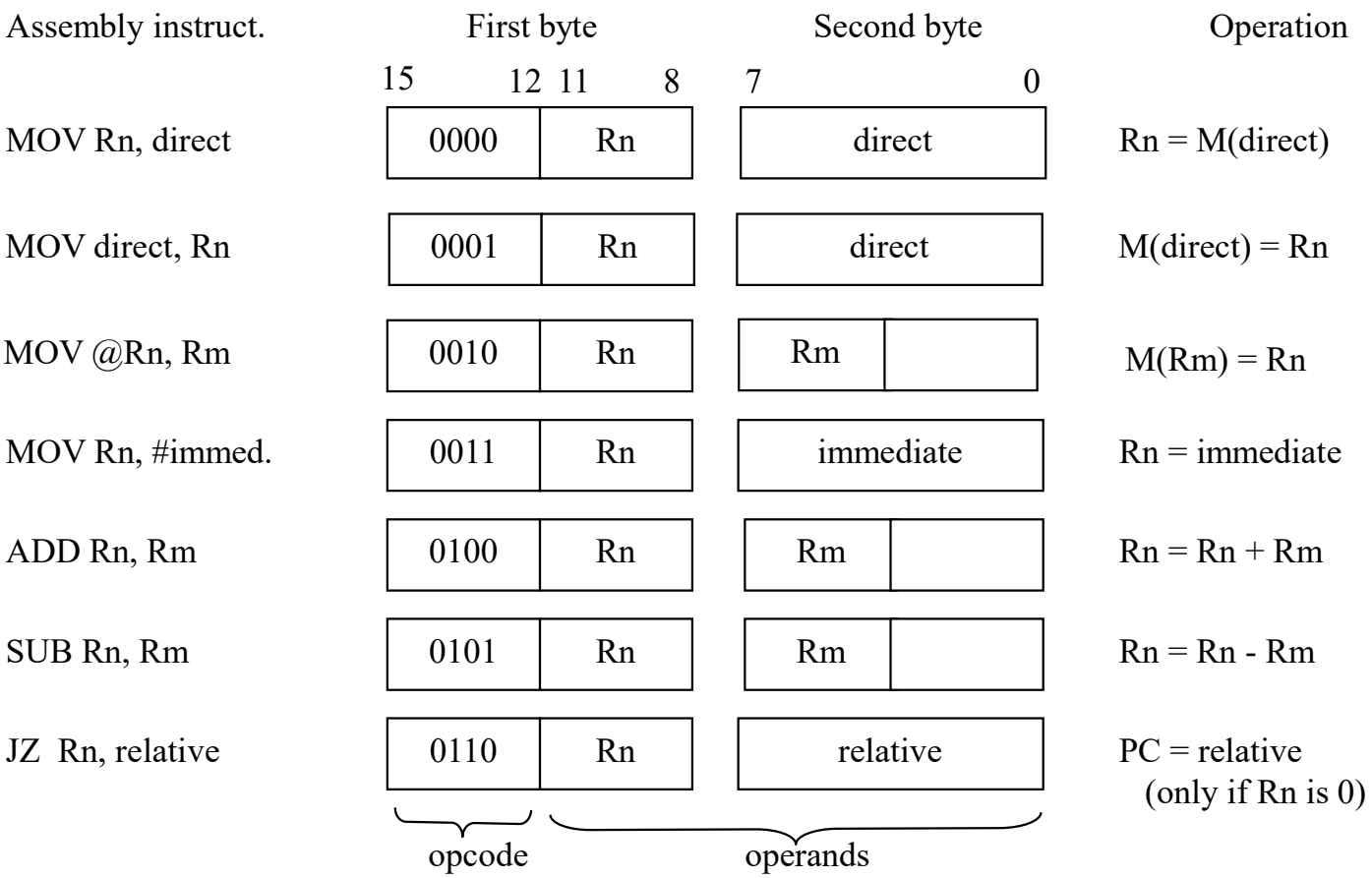
Outline

- Processor Architecture
- Programmer's Model
- **RT-level custom microarchitecture design of general-purpose processor**

RTL Design Method

- First create instruction set
- Convert instruction set to “complex” state machine
  - Known as FSMD: finite-state machine with datapath (or CDFG)
- Partition FSMD into a datapath part and controller part
  - The datapath contains a netlist of functional units like multiplexors, registers, subtractors and a comparator, …
    - This design is structural
  - The controller is an FSM which issues control signals to the datapath based on the current state and the external inputs.
    - This can be a behavioral description.

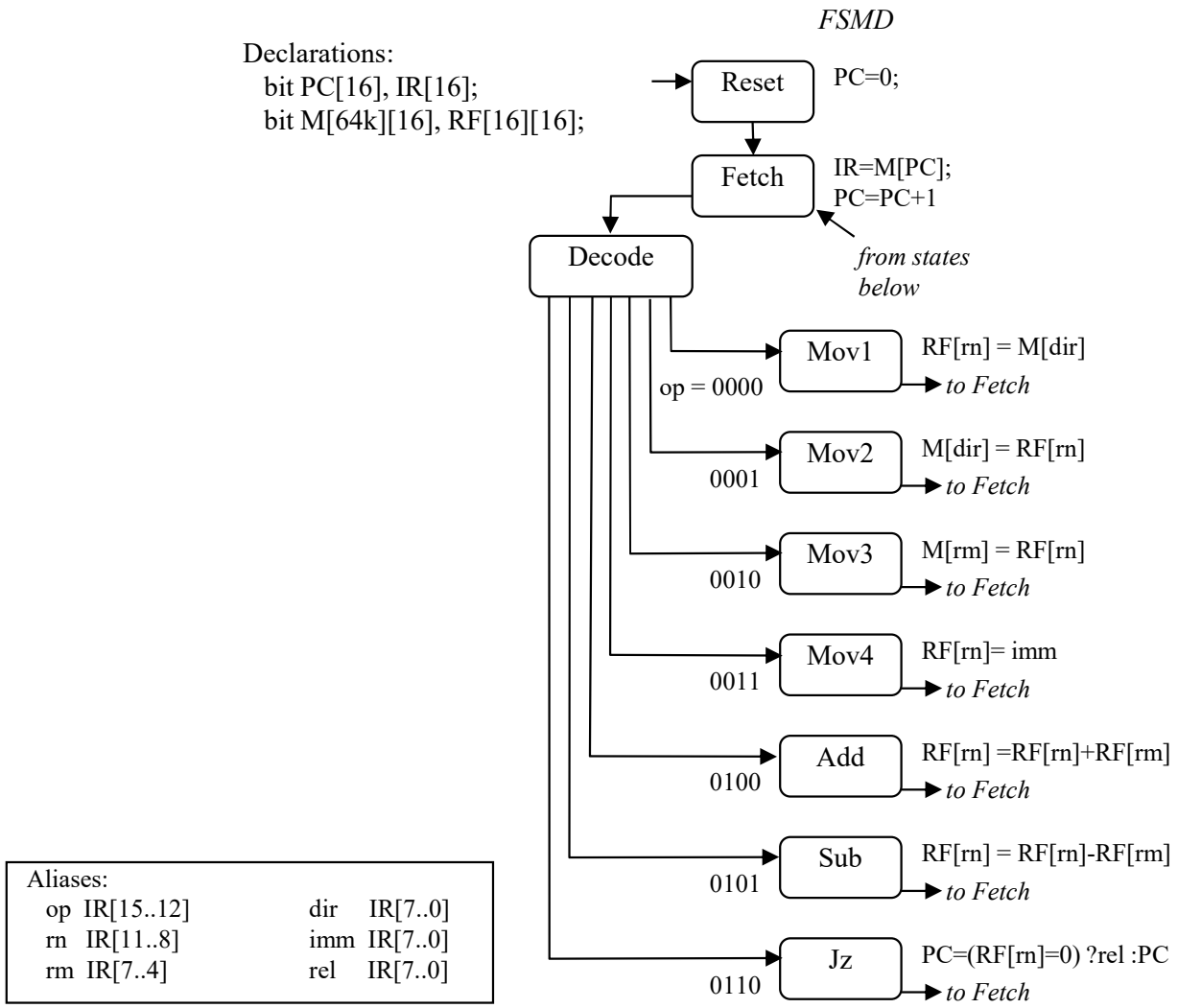
Designing a General Purpose Processor



A Simple Instruction Set

Designing a General Purpose Processor

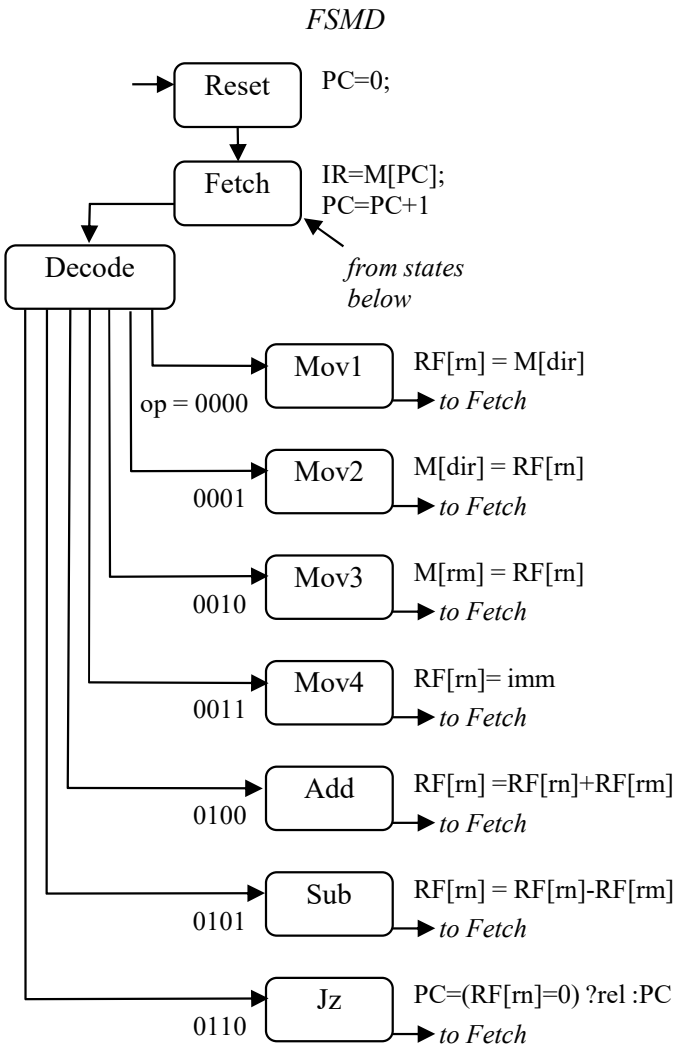
- **Creating the FSMD**



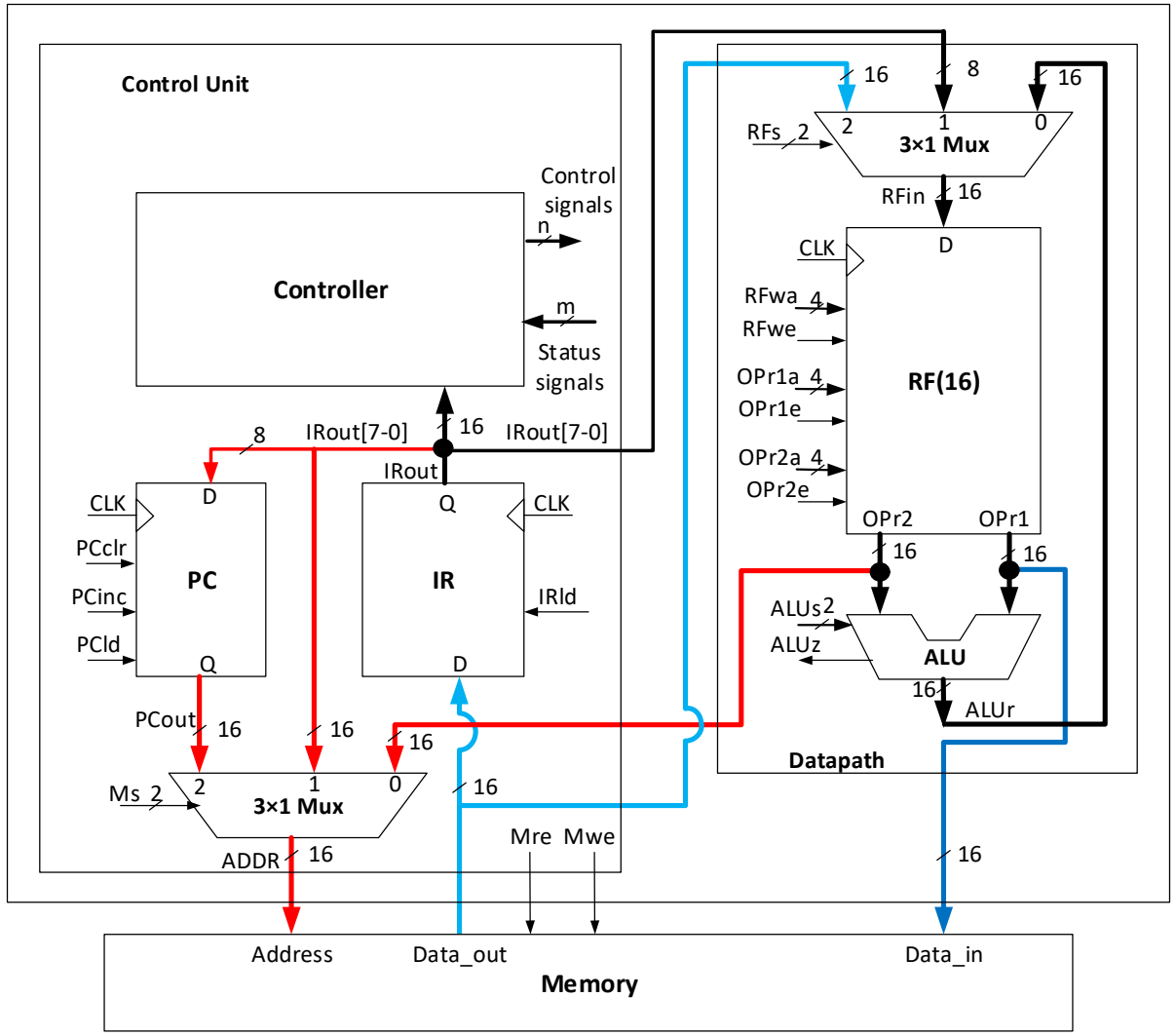
Architecture of a Simple Microprocessor

Creating the Datapath

(c) state diagram FSMD



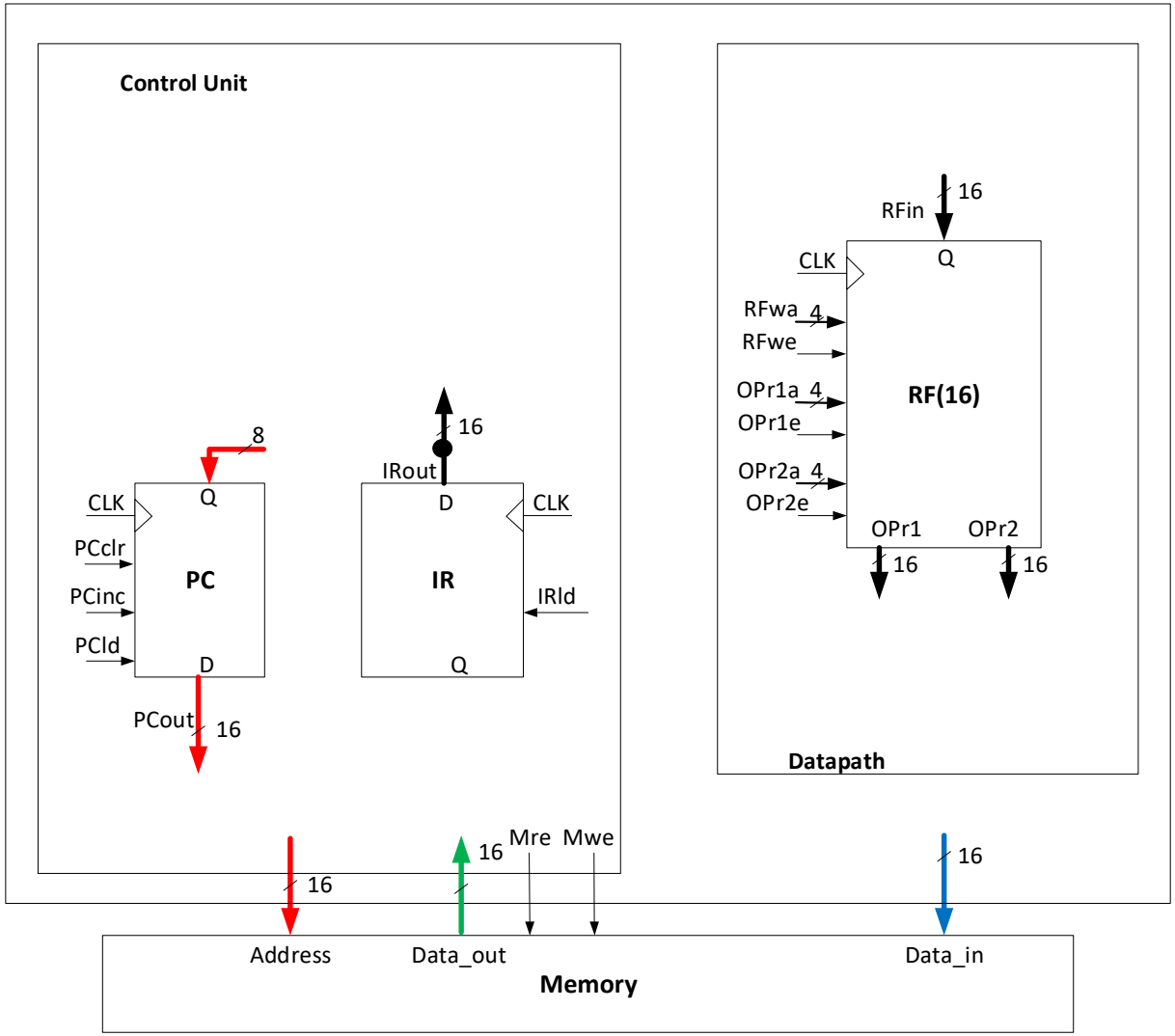
(d) Datapath



Architecture of a Simple Microprocessor

Creating the Datapath

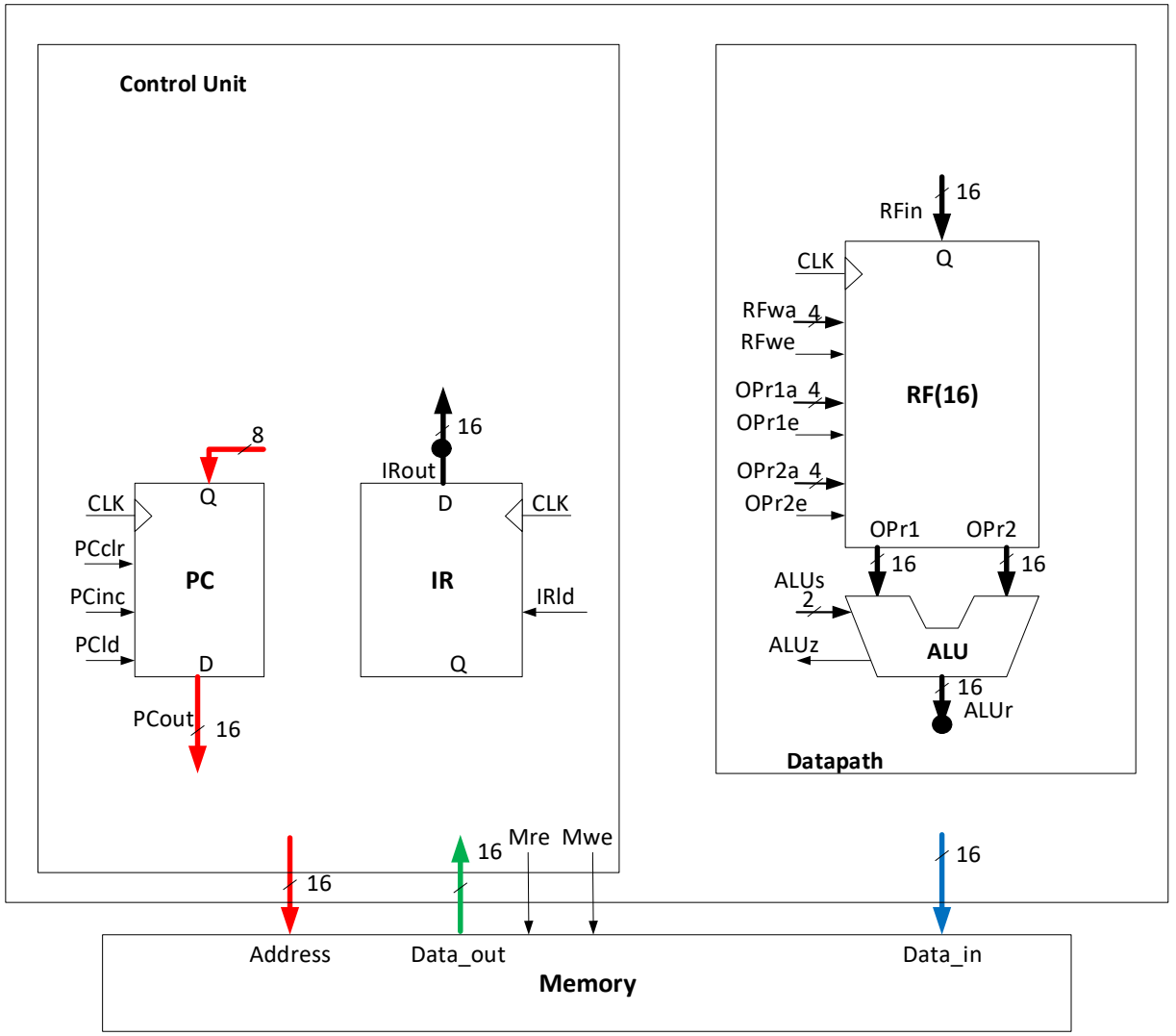
- Storage devices for each declared variable
  - register file holds each of the variables



Architecture of a Simple Microprocessor

Creating the Datapath

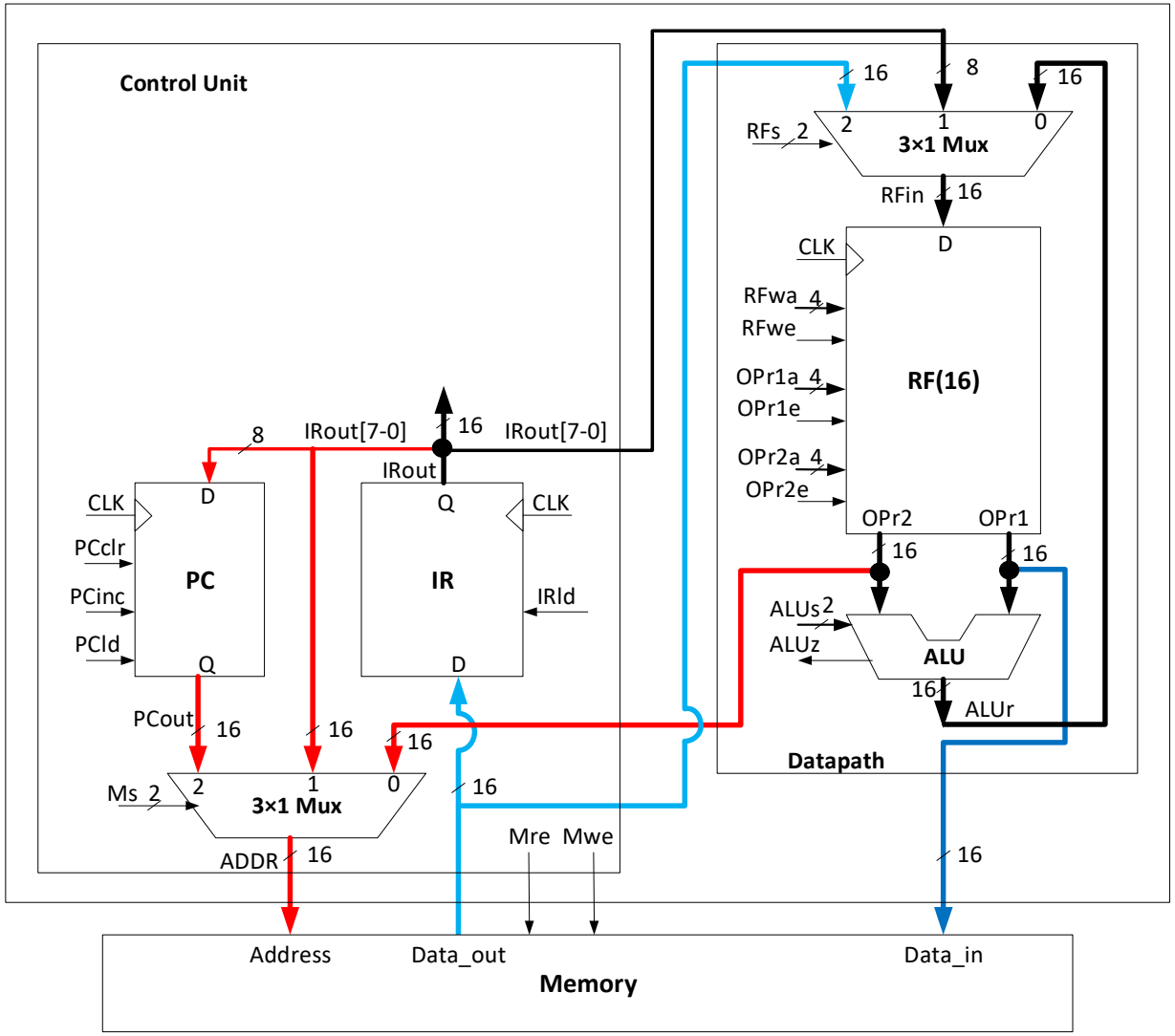
- Storage devices for each declared variable
  - PC, IR, register file holds each of the variables
- Functional units to carry out the FSMD operations
  - One ALU carries out every required operations



Architecture of a Simple Microprocessor

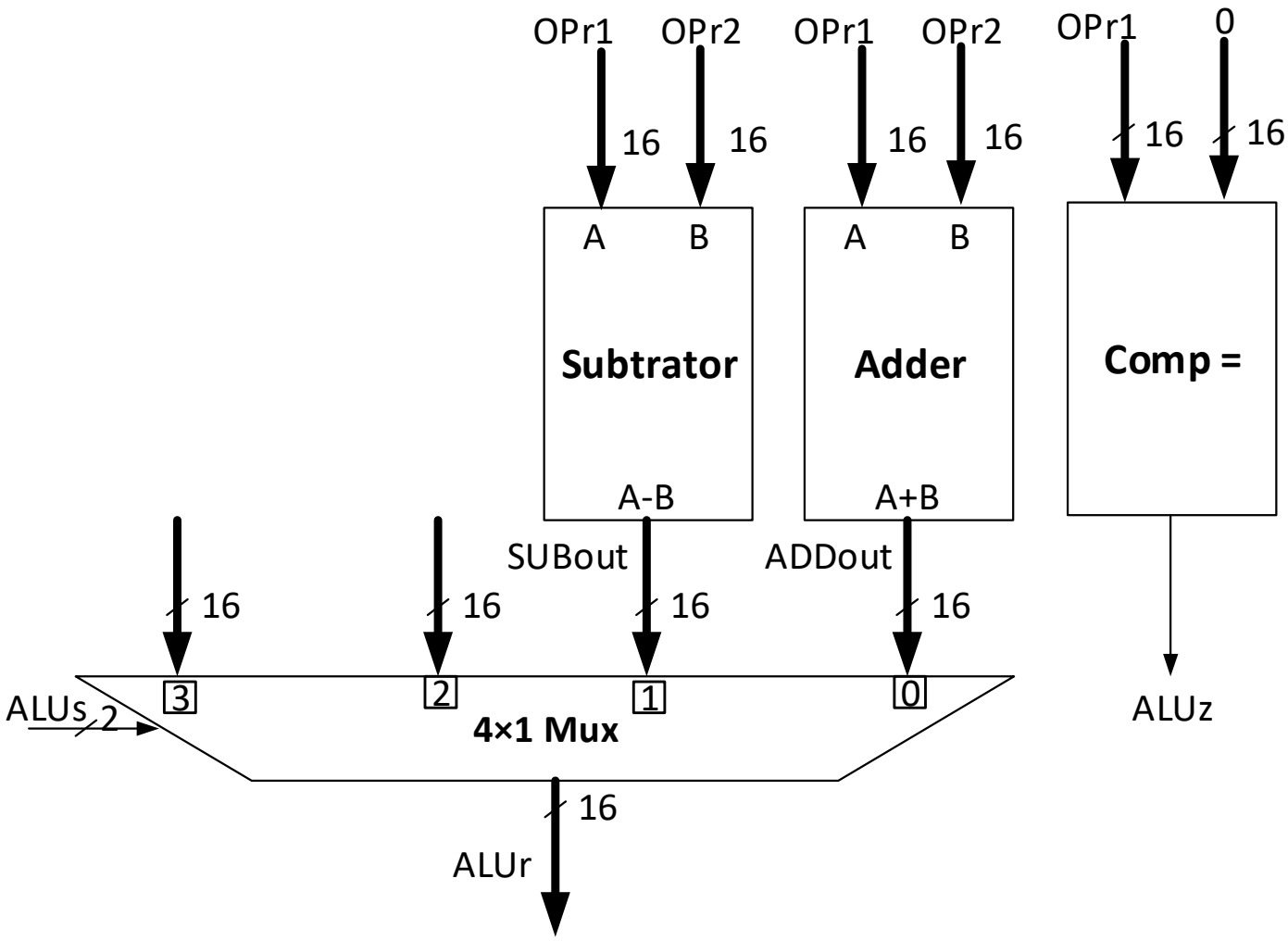
Creating the Datapath

- Storage devices for each declared variable
  - register file holds each of the variables
- Functional units to carry out the FSMD operations
  - One ALU carries out every required operations
- Connections added among the components' ports corresponding to the operations required by the FSM

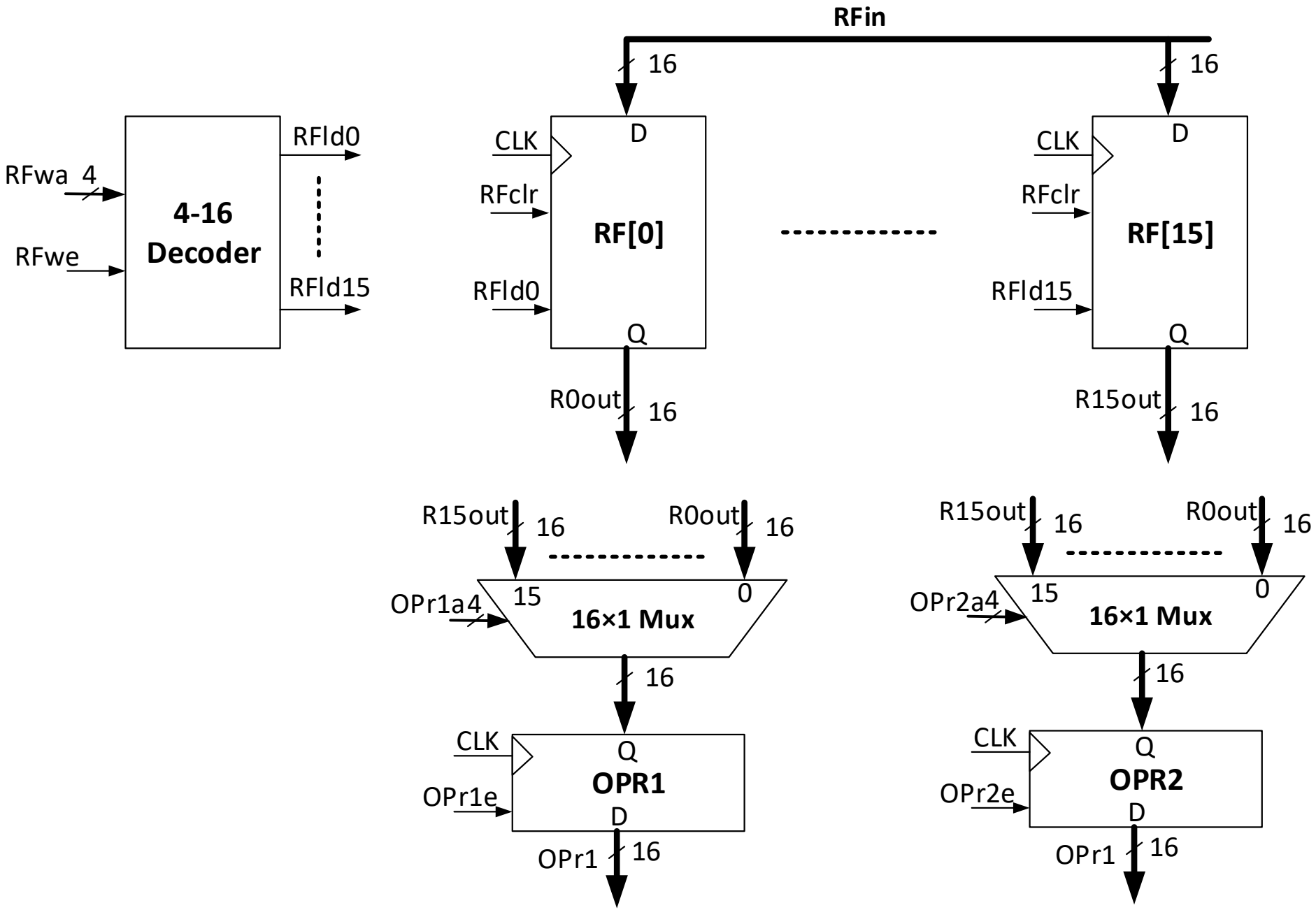


Architecture of a Simple Microprocessor

ALUs	ALUr
00	OPr1 + OPr2
01	OPr1 - OPr2
10	OPr1 **OR** OPr2
10	OPr1 **AND** OPr2

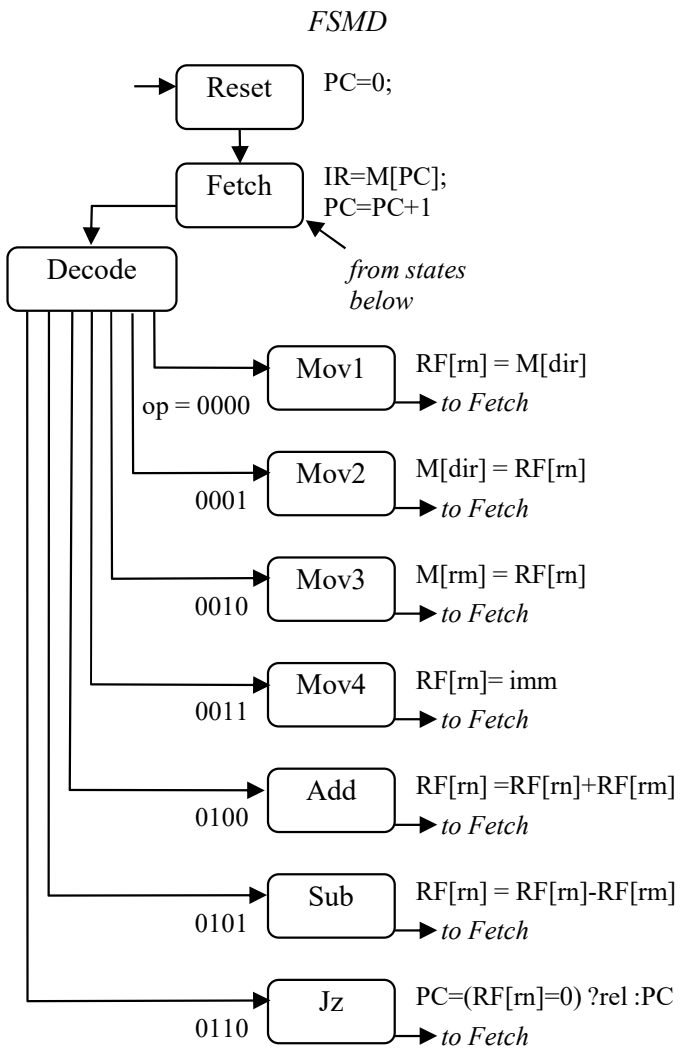


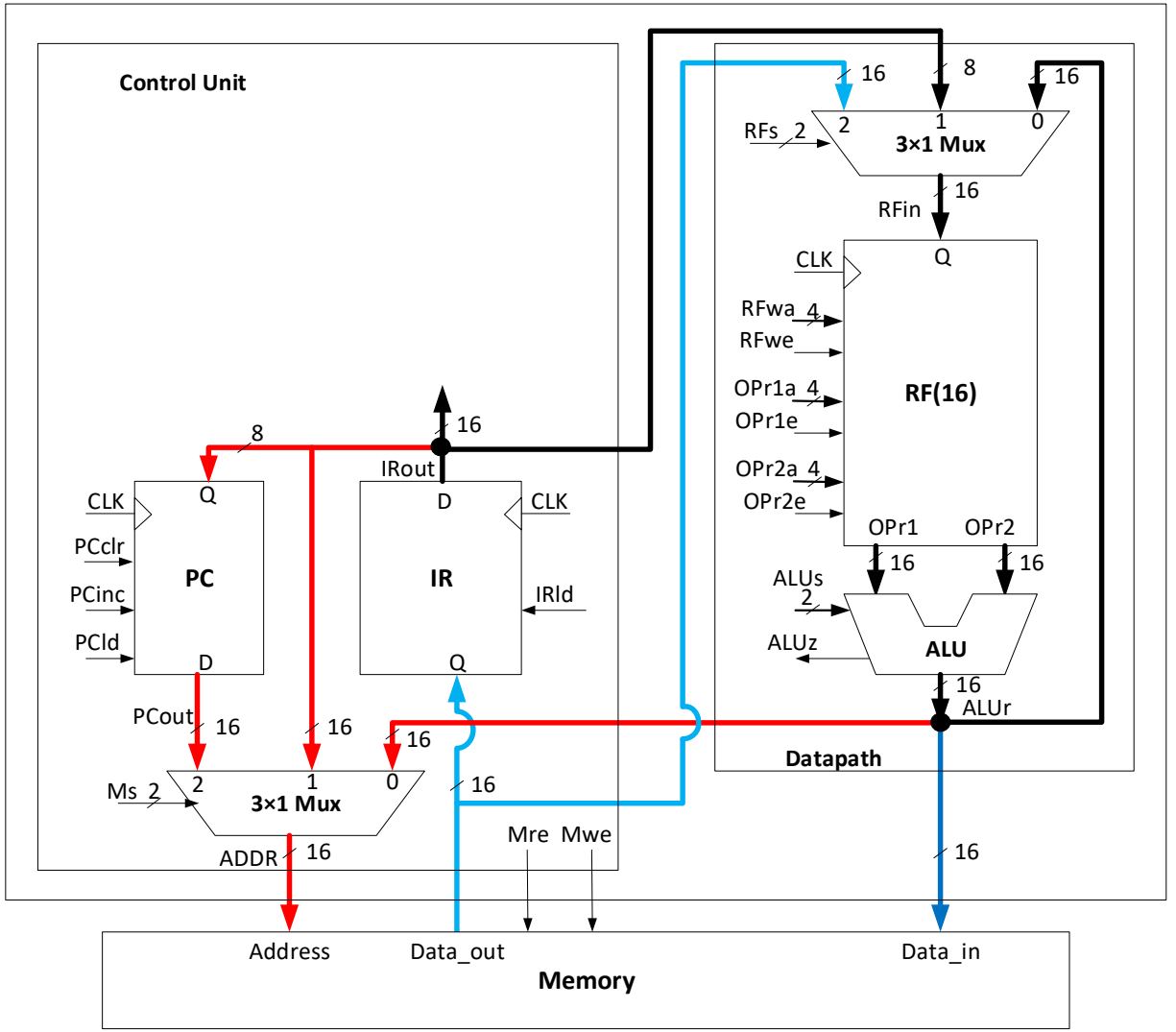
Architecture of a Simple Microprocessor



Architecture of a Simple Microprocessor

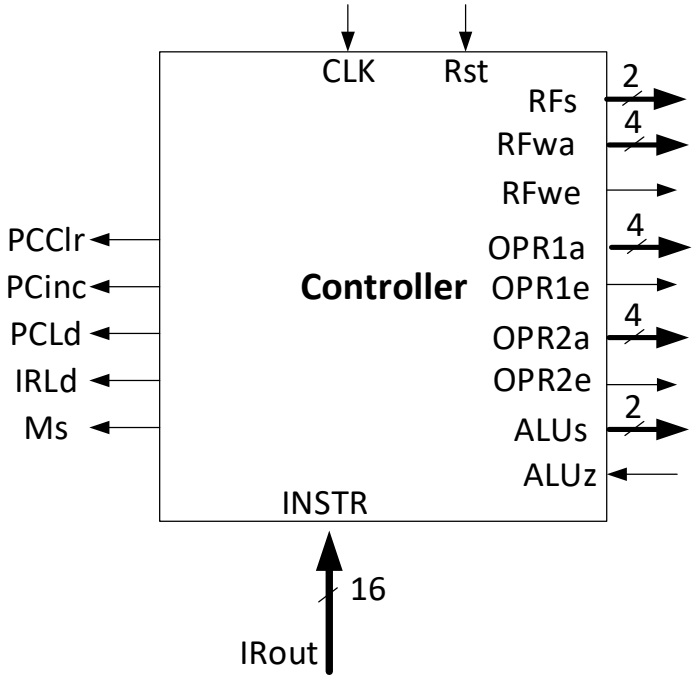
Creating the Controller



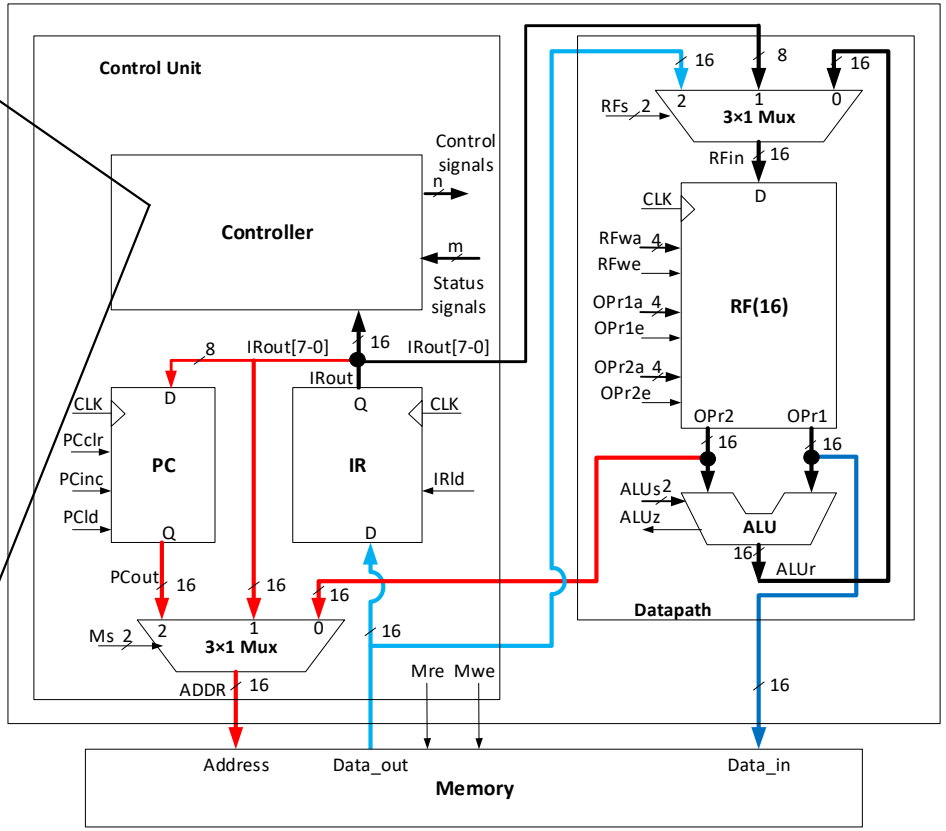


A Simple Microprocessor (here)

(e) Designing Controller

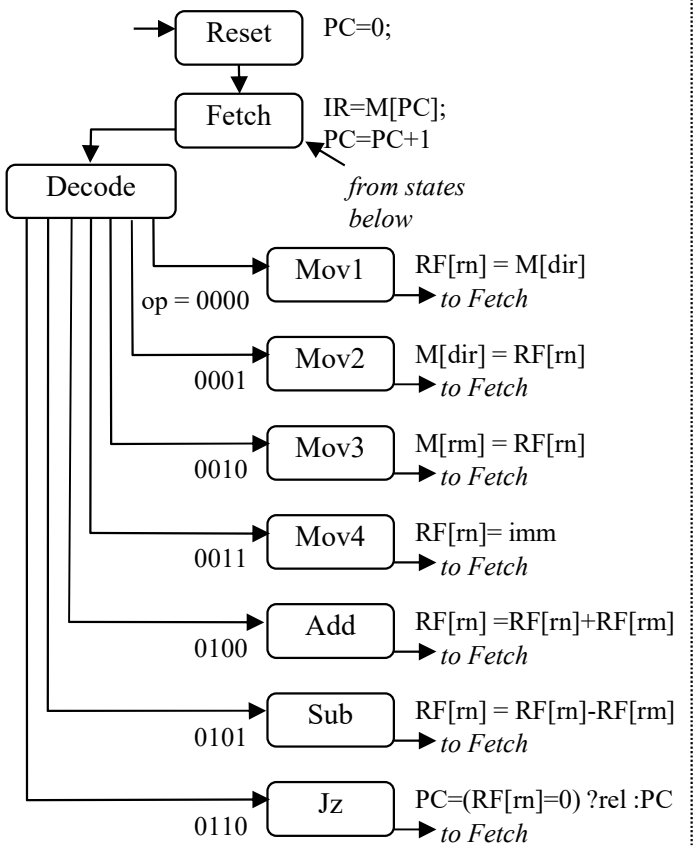


(d) Datapath



A Simple Microprocessor

(c) state diagram FSMD → **(f) state diagram FSM** ← **(d) Datapath**



FSMD

PCclr=‘1’;

MS=“10”;
Mre=‘1’;
Irld=‘1’;
PCinc=‘1’;

Ms=“01”; Mre=‘1’;
RFwa=rn; RFwe=‘1’; RFs=“10”;

OPr1a=rn; OPr1e=‘1’;
Ms=“01”; Mwe=‘1’;

OPr1a=rn; OPr1e=‘1’;
OPr2a=rm; OPr2e=‘1’;
Ms=“00”; Mwe=‘1’;

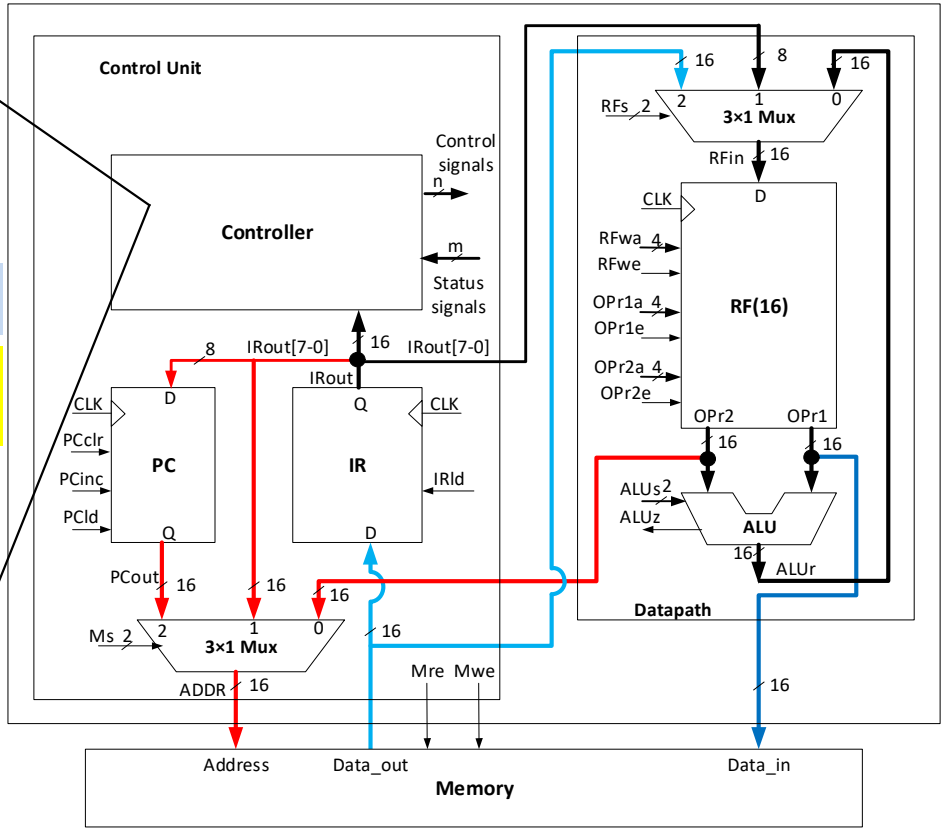
RFwa=rn; RFwe=‘1’; RFs=“01”;

OPr1a=rn; OPr1e=‘1’;
OPr2a=rm; OPr2e=1; ALUs=“00”;
RFwa=rn; RFwe=‘1’; RFs=“00";

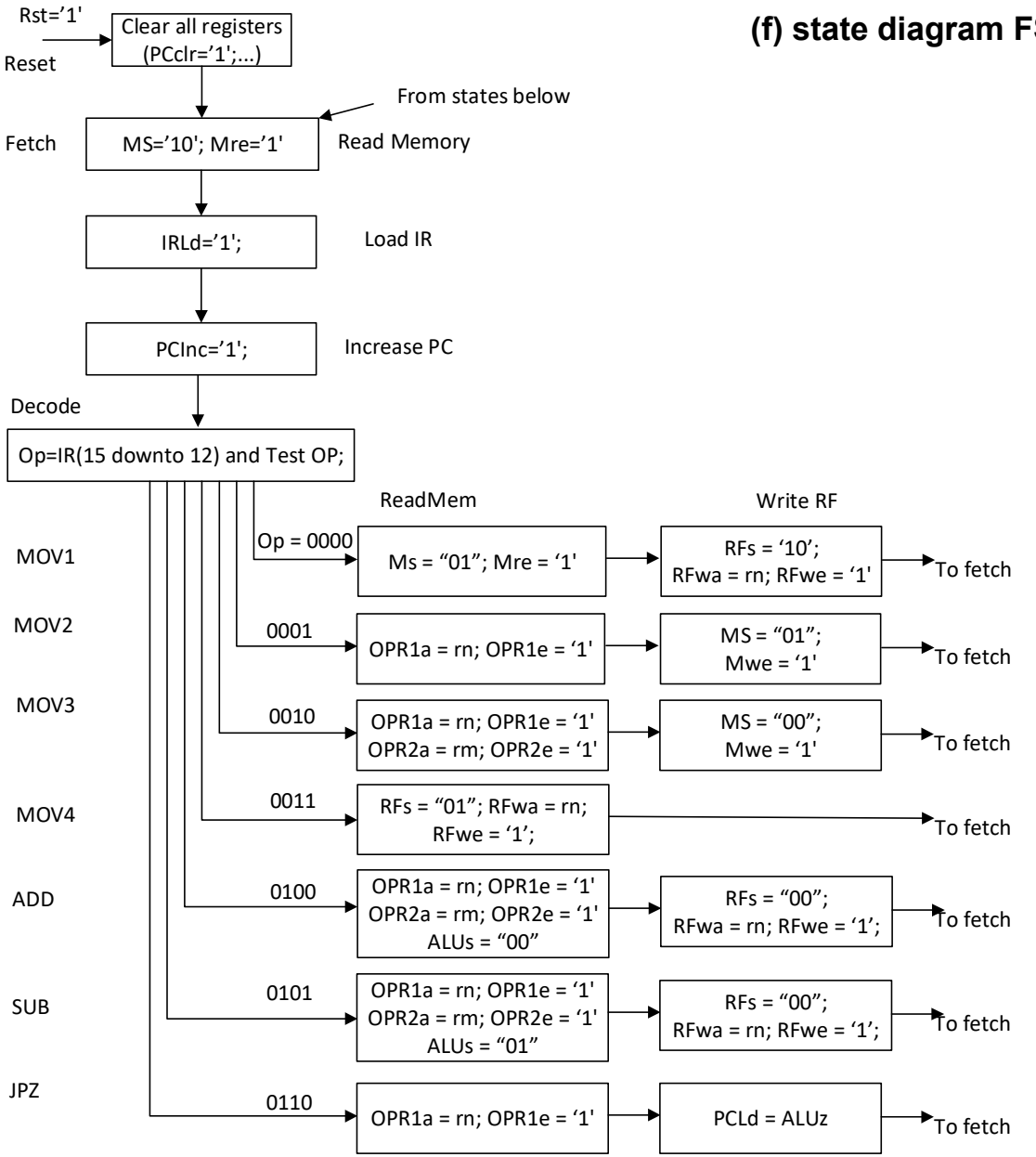
OPr1a=rn; OPr1e=‘1’;
OPr2a=rm; OPr2e=1; ALUs=“01”;
RFwa=rn; RFwe=‘1’; RFs=“00”;

OPrla=rn;
OPrle=1;
PCld= ALUz;

FSM operations that replace the FSMD operations after a datapath is created

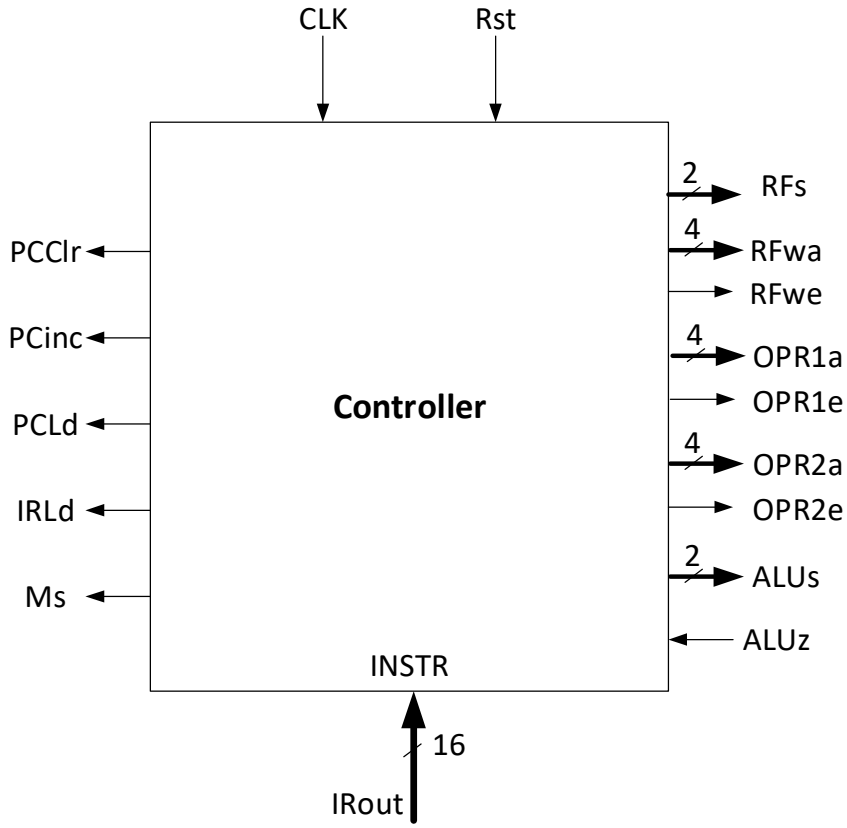


A Simple Microprocessor

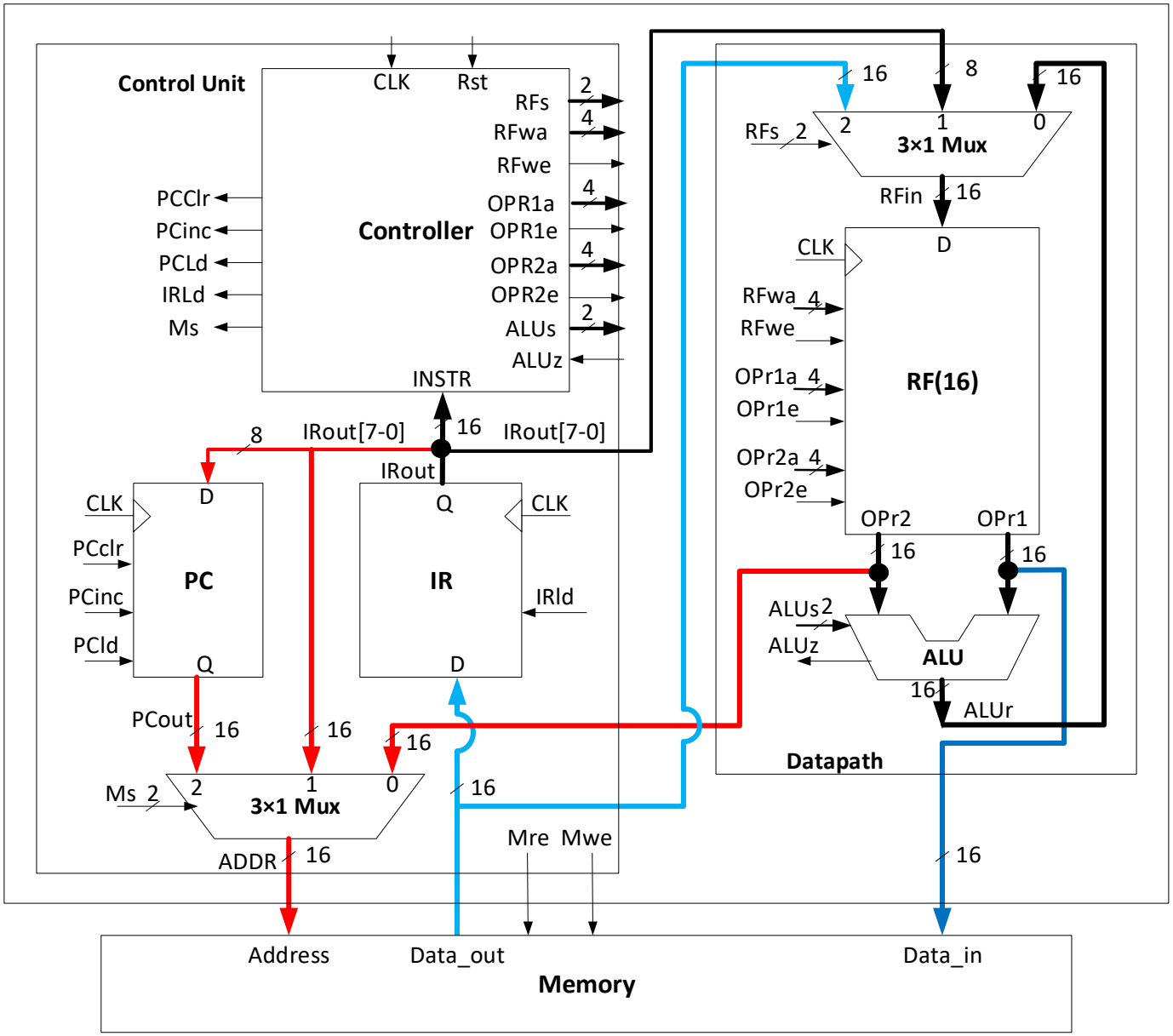


(f) state diagram FSM

(e) I/O port of Controller



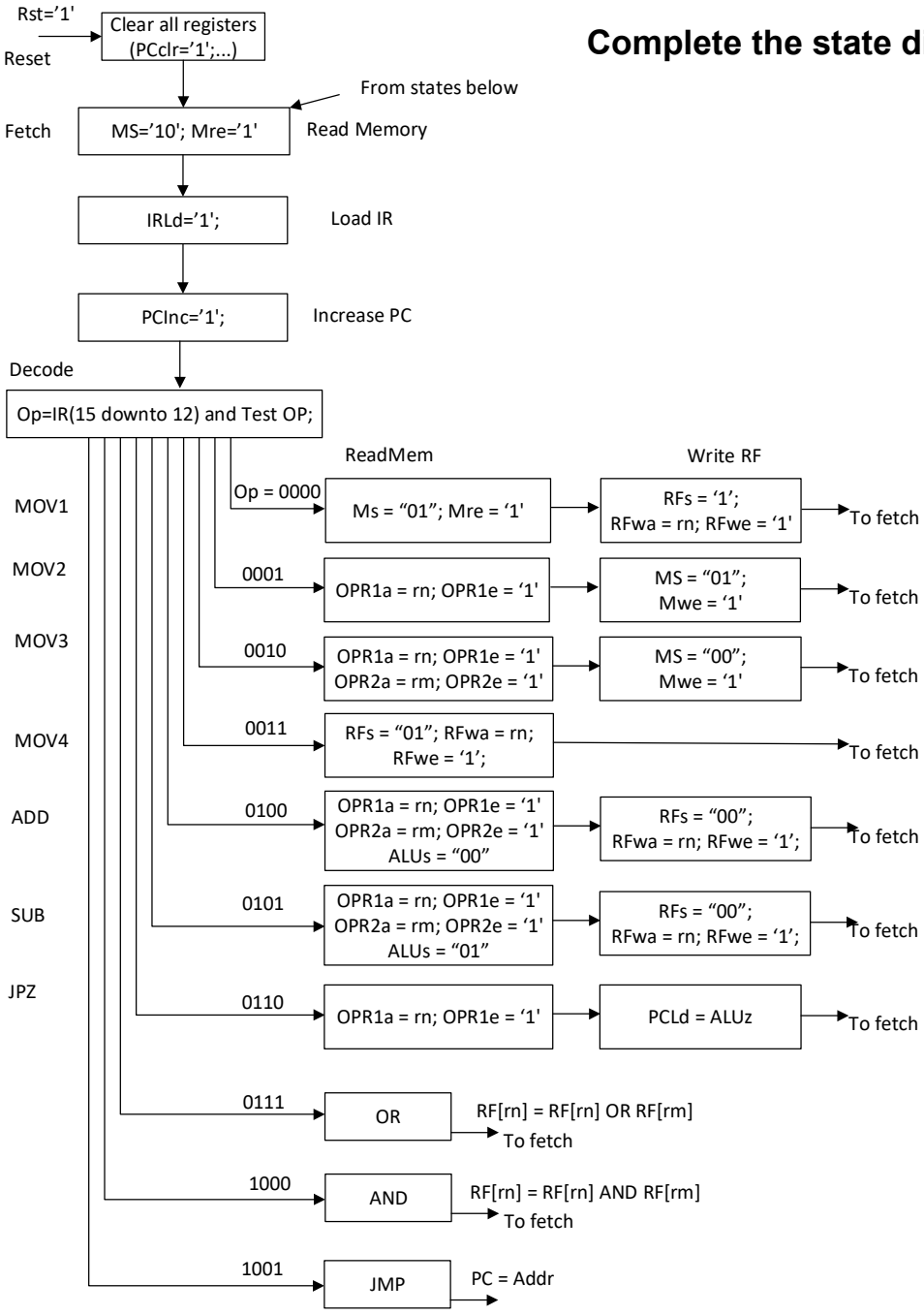
A Simple Microprocessor



You just built a simple microprocessor!

Quiz

Complete the state diagram FSM for instructions: OR, AND, JMP



RTL Modeling of Processor design in VHDL

- **Controller**: is behavior description of the FSM
- **Datapath:** structural description, including:
  - **Mux3to1:** choose one of three 16-bit inputs to connect to output port under the control of one select line.
  - **Register File:** 16 Registers, each takes a 16-bit input, a load signal, reset, and a clock signal.
  - **ALU:** adds or subtracts two 16-bit numbers,.
  - **PC Register:** 16 bits, holds the address of the next instruction.
  - **IR Register: 16 bits,** holds the next instruction
  - **Decoder4to16:** take a 4-bit input and generate a 16-bit output signal

Testbench for testing the Processor

- **Instruction/Data Memory Model**: behavior description
- **Data and Program for test**: stored in .mif file
- **Reset/Clock circuits:**

Summary

- General-purpose processors
  - Good performance, low NRE, flexible
- Structure includes Controller, datapath, and memory
- Many tools required for developing applications
  - Including compiler and debuger, instruction-set simulators, and in-circuit emulators
- Programmer's Model
  - Assembly Instructions, Registers, Addressing Models, …
- Techniques for designing a general-purpose processor