CSC 3210 Computer organization and programming

Chapter 3 & Chapter 4

Chunlan Gao

Georgia State University

Attendance Number

- The attendance number you should calculate it …….

Last class

- Floating-Point Addition, FP Adder Hardware, Floating-Point Multiplication, temperature conversion example

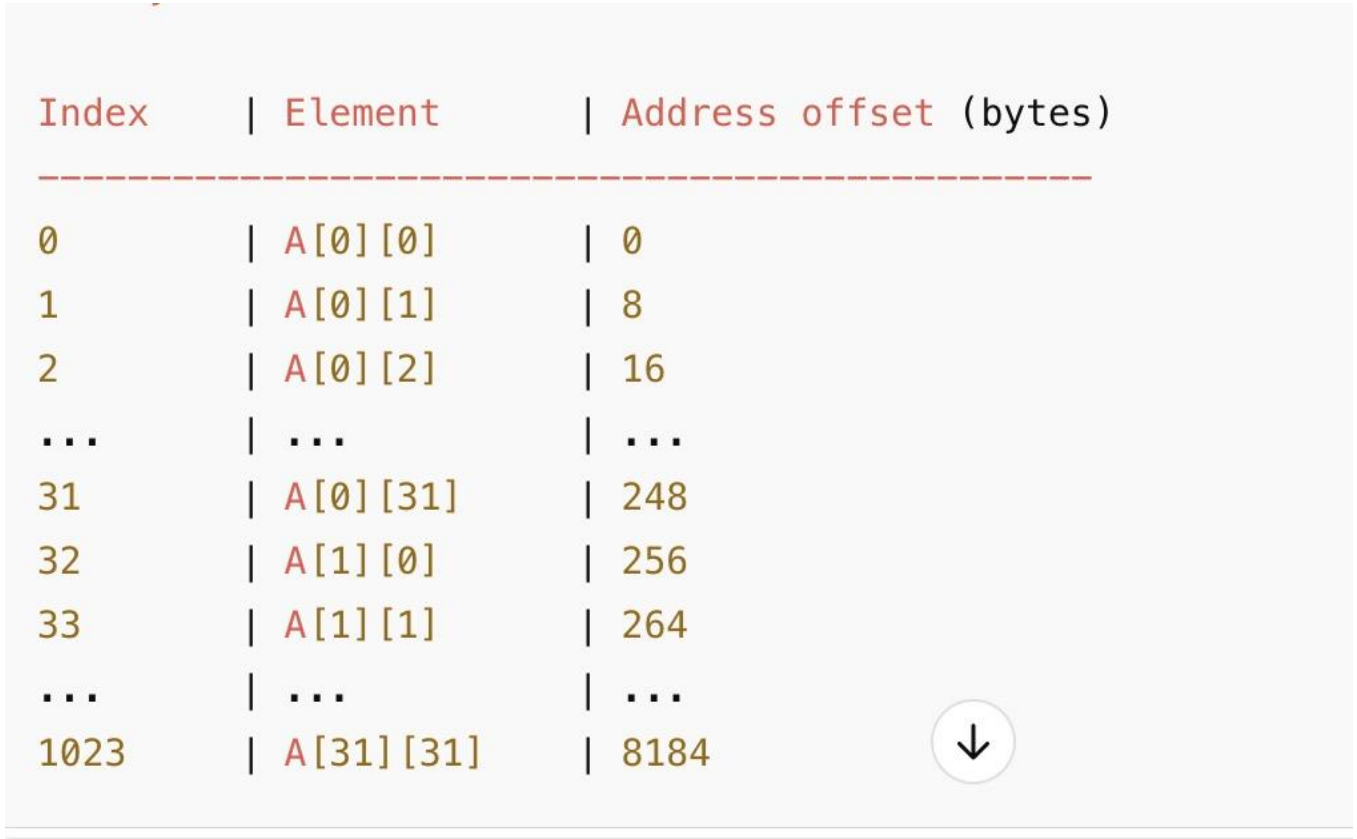
FP Example: Array Multiplication

- C = C + A × B
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double c[][],
         double a[][], double b[][]) {
  size_t i, j, k;
  for (i = 0; i < 32; i = i + 1)
    for (j = 0; j < 32; j = j + 1)
      for (k = 0; k < 32; k = k + 1)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

  - Addresses of `c`, `a`, `b` in x10, x11, x12, and `i`, `j`, `k` in x5, x6, x7
  - Assume that the integer variables are in x5, x6, and x7, respectively

Array in the memory



Index	Element	Address offset (bytes)
0	A[0][0]	0
1	A[0][1]	8
2	A[0][2]	16
...	...	...
31	A[0][31]	248
32	A[1][0]	256
33	A[1][1]	264
...	...	...
1023	A[31][31]	8184

FP Example: Array Multiplication

- RISC-V code:

```
    mm:...
          li    x28,32        // x28 = 32 (row size/loop end)
          li    x5,0          // i = 0; initialize 1st for loop

   L1:    li    x6,0          // j = 0; initialize 2nd for loop
   L2:    li    x7,0          // k = 0; initialize 3rd for loop
# f0 = c[i][j]
             slli  x30,x5,5     // x30 = i * 2**5 (size of row of c)
             add   x30,x30,x6   // x30 = i * size(row) + j
             slli  x30,x30,3    // x30 = byte offset of [i][j]
             add   x30,x10,x30  // x30 = byte address of c[i][j]
             fld   f0,0(x30)    // f0 = c[i][j]
# f1 = b[k][j] and f2= a[k][j]
   L3:    slli  x29,x7,5     // x29 = k * 2**5 (size of row of b)
          add   x29,x29,x6   // x29 = k * size(row) + j
          slli  x29,x29,3    // x29 = byte offset of [k][j]
          add   x29,x12,x29  // x29 = byte address of b[k][j]
          fld   f1,0(x29)    // f1 = b[k][j]
```

Index	Element	Address offset (bytes)
0	A[0][0]	0
1	A[0][1]	8
2	A[0][2]	16
...	...	...
31	A[0][31]	248
32	A[1][0]	256
33	A[1][1]	264
...	...	...
1023	A[31][31]	8184

FP Example: Array Multiplication

```
        slli   x29,x5,3     // x29 = i * 2**5 (size of row of a)
        add    x29,x29,x7   // x29 = i * size(row) + k
        slli   x29,x29,3    // x29 = byte offset of b[i][k]
        add    x29,x11,x29  // x29 = byte address of a[i][k]
        fld    f2,0(x29)    // f2 = a[i][k]
#f[0]= c[i][j] + a[i][k] * b[k][j]
        fmul.d f1, f2, f1   // f1 = a[i][k] * b[k][j]
        fadd.d f0, f0, f1   // f0 = c[i][j] + a[i][k] * b[k][j]
      # k++, check if k<32
        addi   x7,x7,1      // k = k + 1
        bltu   x7,x28,L3    // if (k < 32) go to L3
 # save f0 to c[i][j]
         fsd    f0,0(x30)    // c[i][j] = f0
        addi   x6,x6,1      // j = j + 1
        bltu   x6,x28,L2    // if (j < 32) go to L2
        addi   x5,x5,1      // i = i + 1
        bltu   x5,x28,L1    // if (i < 32) go to L1
```

Accurate Arithmetic

IEEE 754 always keeps two extra bits

on the right during intervening additions, called guard and round

Add $2.56 * 10^0$ to $2.34 * 10^2$, assuming we have three significant decimal digit, (the guard digit hold5 and round digit hold 6)

$$\begin{array}{r} 2.3400_{ten} \\ +0.0256_{ten} \\ \hline 2.3656_{ten} \end{array}$$

Round to 3 digit 2.37

Without guard and round

the result is 2.36

$$\begin{array}{r} 2.34_{ten} \\ +0.02_{ten} \\ \hline 2.36_{ten} \end{array}$$

Chapter 4

- CPU performance factors:
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- CPU time = Instruction Count * CPI* Clock Cycle Time
  = Instruction Count * CPI / Clock Rate
- We will examine two RISC-V implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `ld`, `sd` …
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `xor` ….
  - Control transfer: `beq` …

Chapter 4

Logic Design Basic review

- Information encoded in binary

--low voltage =0, high voltage =1

--one bit per wire

--multi-bit data encoded on multi-wire buses

- Combinational element (and, alu)

-operate one data

- output just depend on input (a function of input)

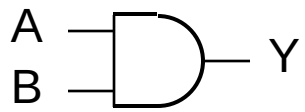
- State (sequential) elements (register, memory)

-store information

-operate on at least two inputs - input data, clock

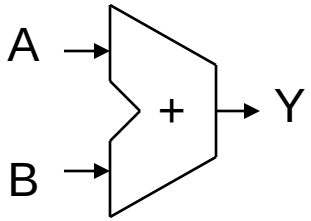
Chapter 4 :Combinational Elements

- AND-gate
  - Y = A & B



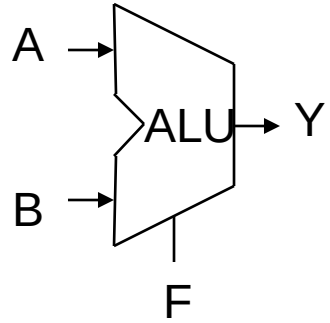
- Adder

  Y = A + B



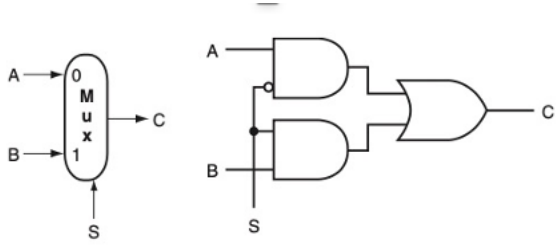
Arithmetic/Logic Unit

Y = F(A, B)

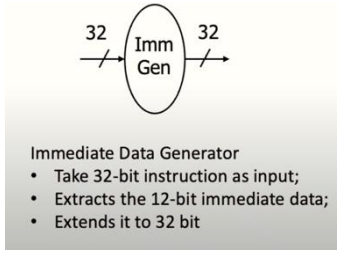


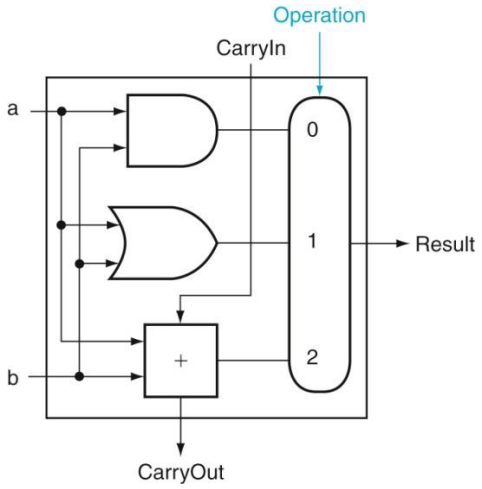
- Multiplexer

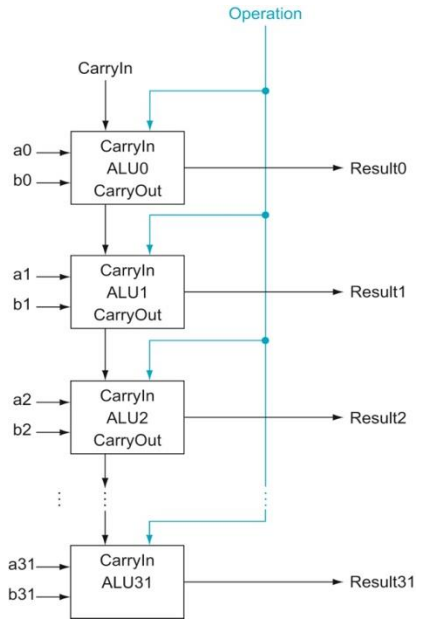
  Y = S ? A : B



Immediate Generator

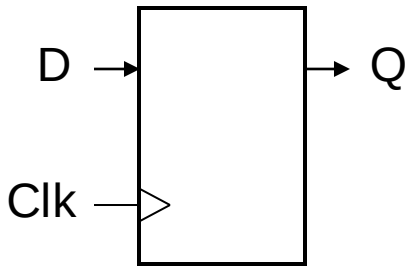




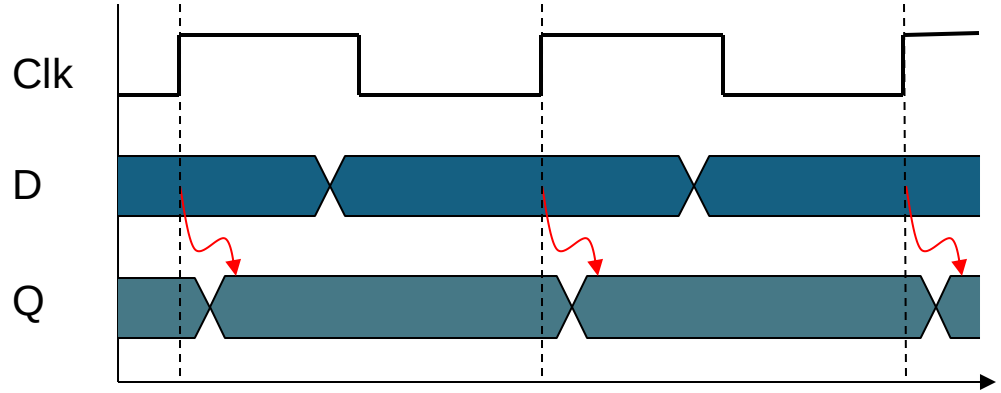


Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

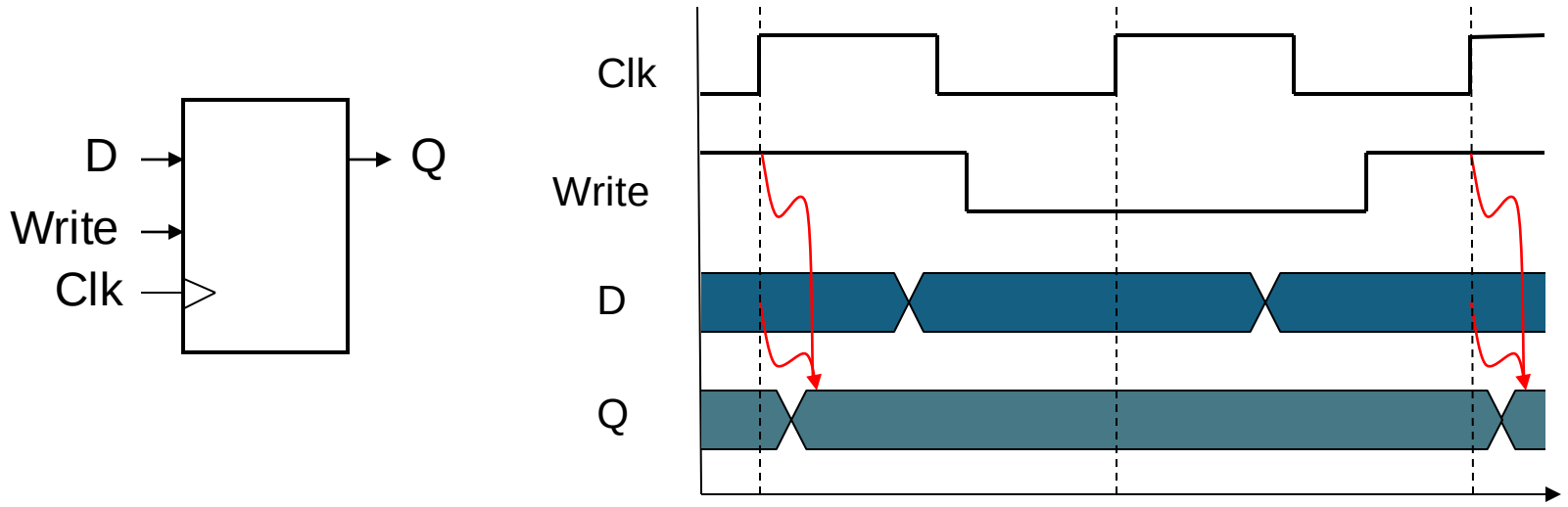


Delay flip-flop



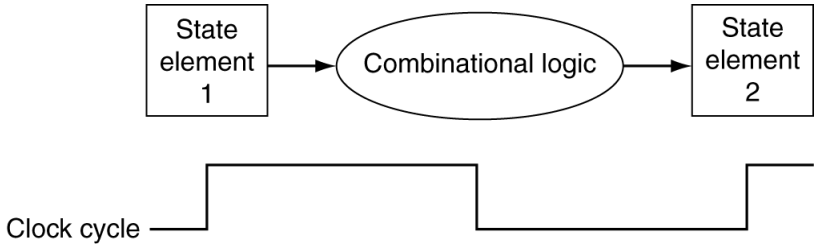
Sequential Elements

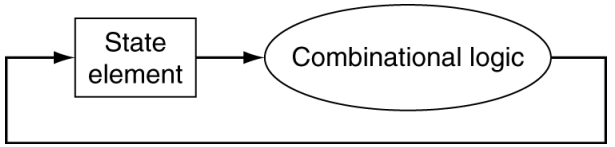
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



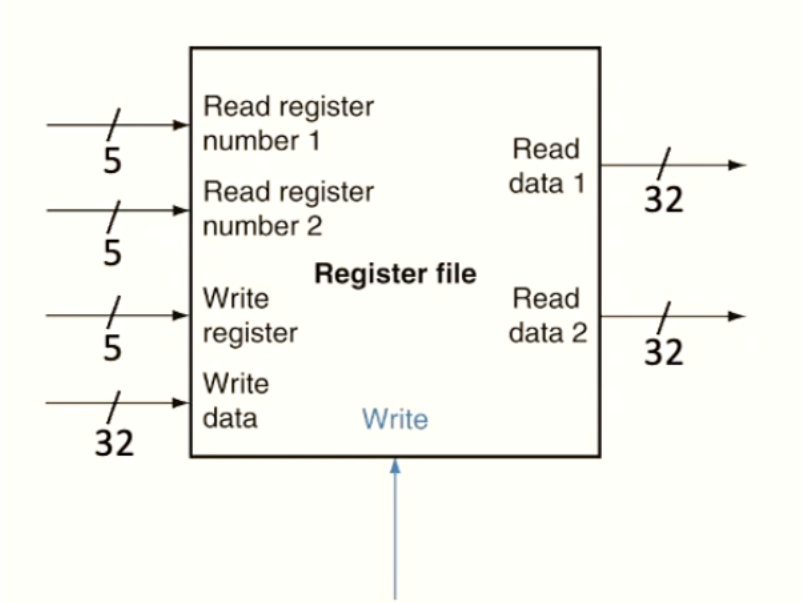
Clocking control

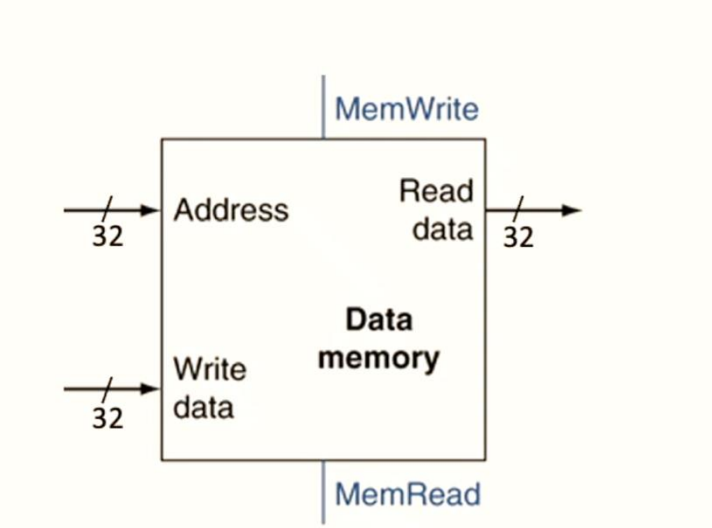
- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period





Sequential Elements





Instruction Execution (5 steps)

In a typical RISC-V 5-stage pipeline:

1. **IF (Instruction Fetch)** – IM (Instruction Memory)
2. **ID (Instruction Decode/Register Fetch)** – Reg
3. **EX (Execute)** – ALU
4. **MEM (Memory Access)** – DM (Data Memory)
5. **WB (Write Back)** – Reg

Instruction Execution(5 steps)

IF – Instruction Fetch

- Components involved:
  - Instruction memory
  - PC (Program Counter)
- Purpose: Fetch the instruction from memory using the PC.

ID – Instruction Decode / Register Fetch

- Components involved:
  - Register File (for reading source registers)
  - Immediate Generator (if needed)
  - Control Logic
- Purpose: Decode the instruction, read registers, and generate control signals.

EX – Execute / ALU

- Components involved:
  - ALU (performs operations)
  - Multiplexers (for choosing ALU inputs)
  - Branch Target Adder (if needed)
- Purpose: Perform ALU operation, compute addresses, evaluate branches.

MEM – Memory Access

- Components involved:
  - Data Memory
- Purpose:
  - For lw: Load data from memory
  - For sw: Store data to memory

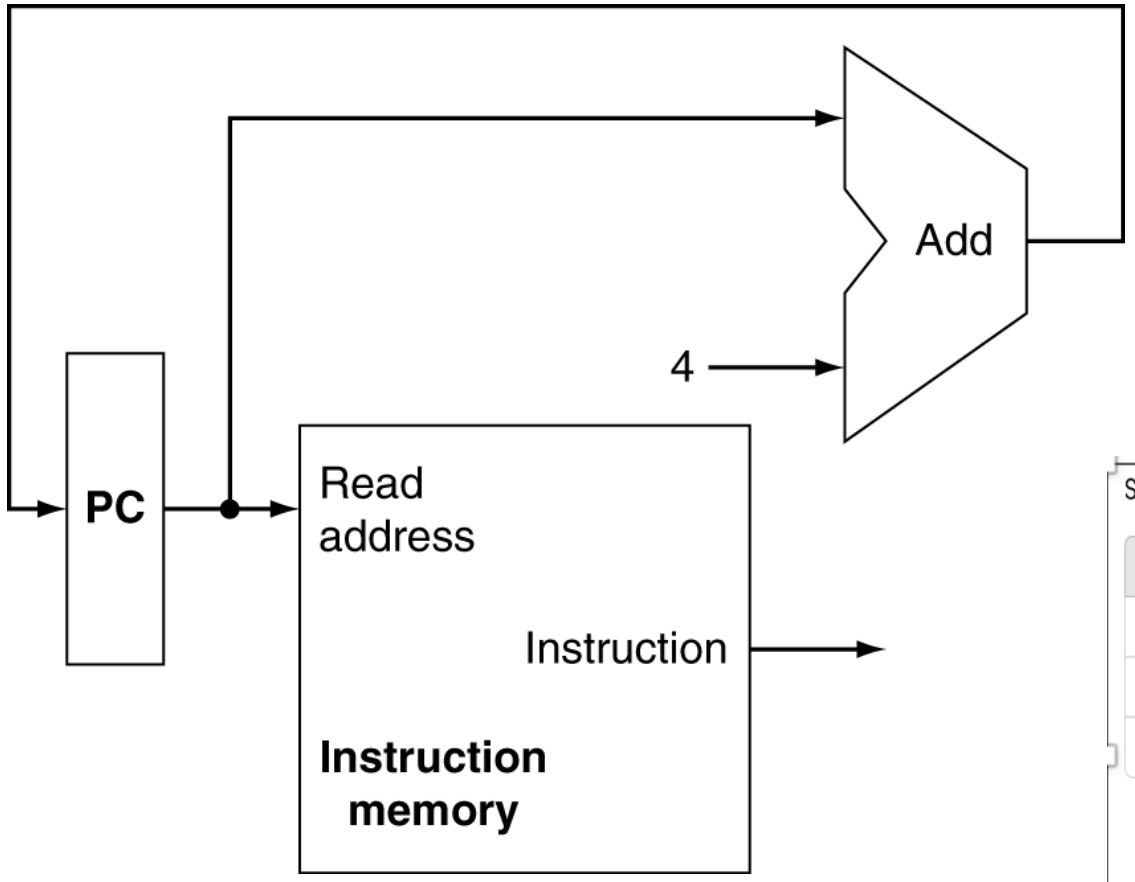
WB – Write Back

- Components involved:
  - MUX for selecting final result
  - Register File (write port)
- Purpose: Write result back to destination register (rd)

Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a RISC-V data path incrementally

Step 1 **Instruction Fetch**



Suppose we have these instructions in memory:

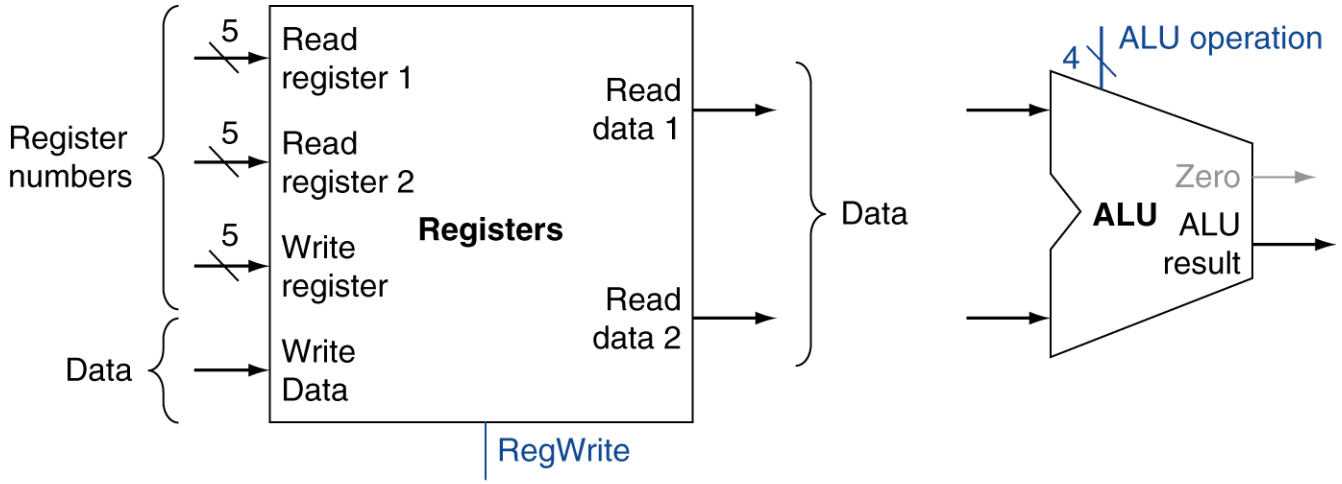
Address	Instruction
`0x1000`	`LOAD R1, 0x50`
`0x1004`	`ADD R1, R2`
`0x1008`	`JMP 0x2000`

- Initially, `PC = 0x1000` → fetches the first instruction.
- After execution, `PC` becomes `0x1004`.

ID (Instruction Decode/Register Fetch)&ALU

R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



a. Registers

b. ALU

Combination

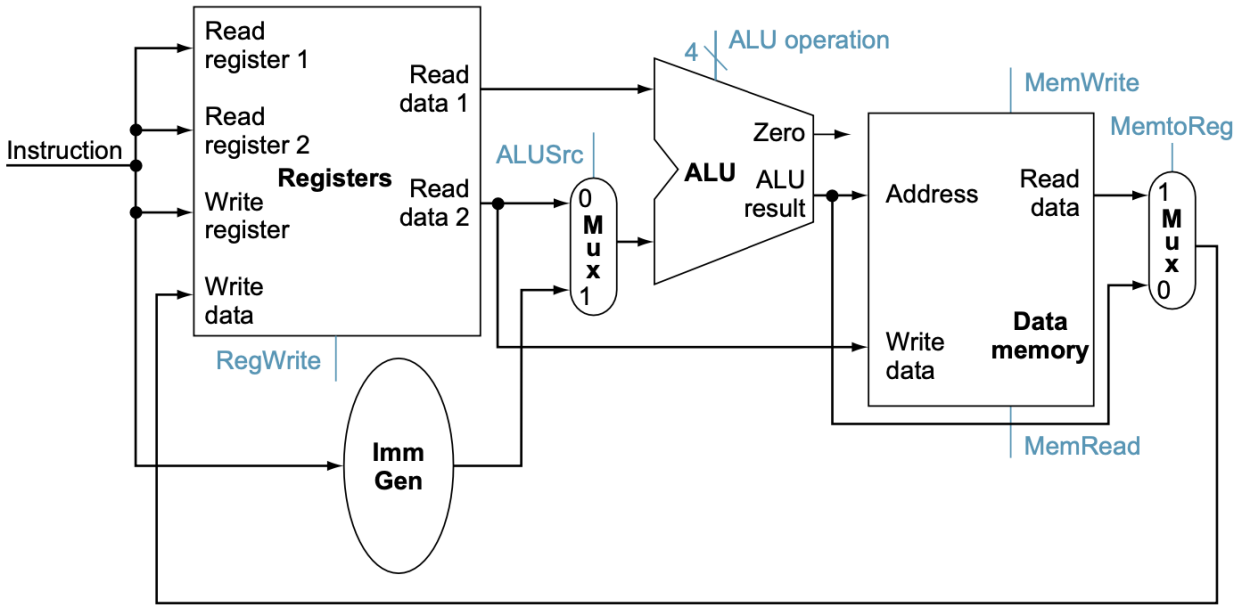


FIGURE 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example.