Computer organization

Levels of abstraction

	Assembler Simulator	Applications
	C C++ Java	High-level language
		SOFTWARE
	add lw ori	Assembly language
Goal	0000 0001 0000 1001 0101	Machine instructions/Data
		HARDWARE
	CPU Memory I/O	Functional units
CMSC311	Multiplexor Adder Register	Components
	Combinational Sequential	Circuits
You are here	AND OR XOR	Gates
	Transistors Wires	Electronics
	Electrons	Atomic units

Datapath and control

- Objective
  - Implement hardware to execute simple instruction set
  - MIPS-lite
    - arithmetic/logical: `add, sub, and, or, slt`
    - memory access: `lw, sw`
    - branch/jump: `beq, j`
- Concepts
  - Stored program
    - memory stores both program instructions and data
    - simplifying assumption: data and instruction memory separate
  - Instruction set architecture (ISA)
    - load-store architecture
      - operations can be performed only on data in registers
- CPU
  - Datapath: performs operations on data (i.e., ALU)
  - Control: tells datapath, memory, etc. what to do

Instruction execution steps

1. IF (Instruction Fetch)
    - address of instruction to be executed is in PC (program counter, a hidden register)
    - instruction is copied from memory to IR (instruction register, another hidden register)
2. D (Decode the instruction, Fetch Operands)
    - determine what operation to perform (opcode, function)
    - get operand values
    - add
        - get 2 operands from registers
    - addi
        - get 1 operand from register, 1 from instruction itself (sign-extended immediate)
3. ALU (Perform the operation)
    - arithmetic, logical, etc.
    - action performed by ALU circuits
4. MEM (Memory access)
    - MIPS: only load or store instructions
5. WB (Write Back)
    - result of third step is written to the appropriate register
6. PC Update (Program Counter Update)
    - normally, PC <- PC + 4
    - branch or jump: some other address

Datapath

Datapath: flow of data during instruction execution

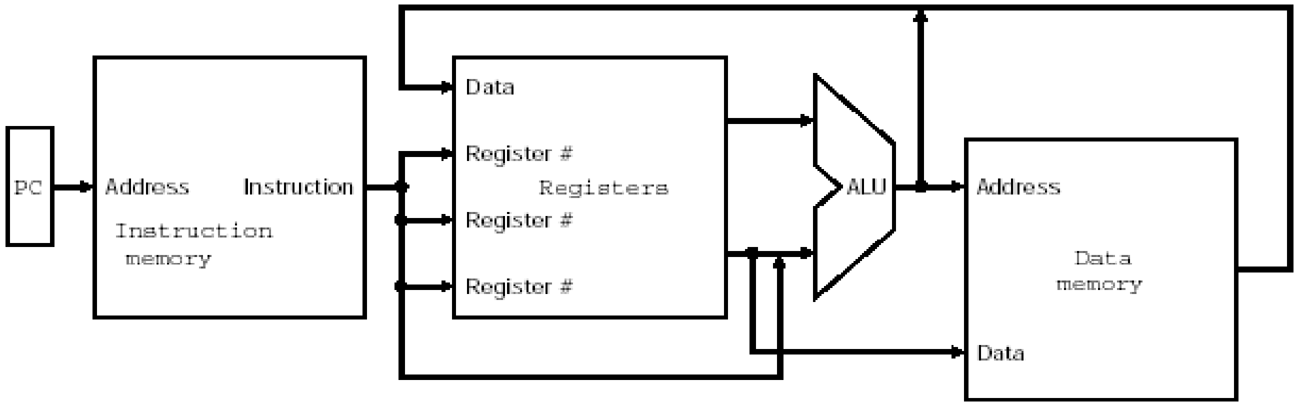


Fig. 5.1

Process

- **PC gives address of next instruction to (instruction) memory**
- **Memory gives instruction contents to IR (instruction register, not shown)**
- **IR gives**
  - **Register numbers (2 source, 1 destination) to registers**
  - **Immediate data to ALU**
- **Registers give data to**
  - **ALU**
  - **(Data) memory**
- **ALU processes operands and gives result to**

- Register (arithmetic/logical result)
- Memory (as data address for load or store)

Data memory gives data to registers (if load)

What's missing from this picture?

- How to update the PC?
  - Could have counter to increment, but what about branches and jumps?
  - Use ALU to compute branch address, but need more control
- Where do register addresses come from?
- How do we control register read/write?
- What about immediate operands?
- How to determine what operation for ALU to perform?

Datapath

Datapath: flow of data during instruction execution

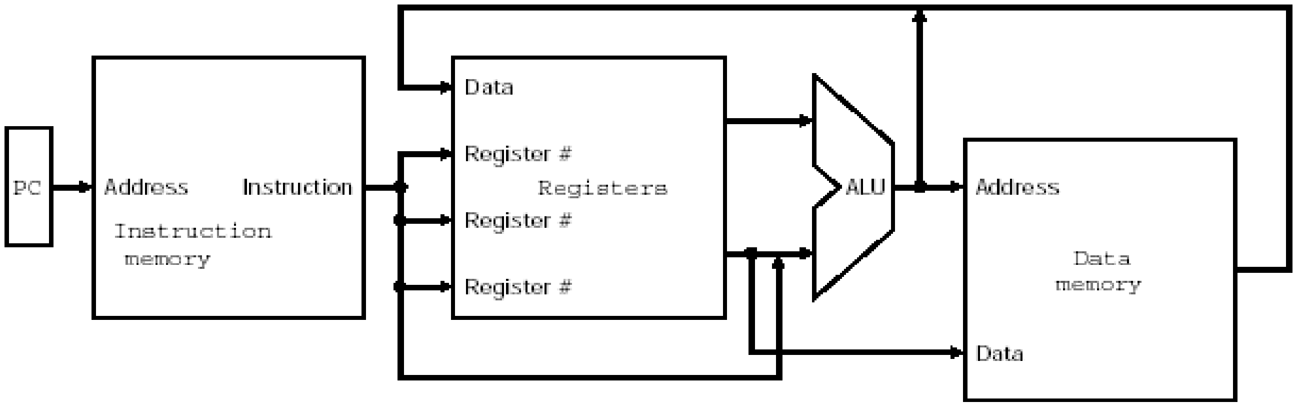


Fig. 5.1

Components

- **Registers: sequential circuits**
- **Register file: group of registers (32 in MIPS)**
- **ALU: combinational circuit**
- **Memory: to be defined later**
  - **Instructions**
  - **Data**
- **Control (not shown): selects operands and tells what to do with them**

Register file

Need a group of registers to store operands (32 in MIPS)

- Registers are very fast memory, part of the CPU
- Why not use only registers for all data?
- Costs more to make registers than RAM
- More registers mean more circuits to control them, therefore slower

Consider 32 integer registers inside a black box called a register file

- Set of registers with combinational logic to select

Instruction to run:

```
    add $1, $2, $3  # R[1] <- R[2] + R[3]
```

Fetch operands:

- Need values in registers 2 and 3

To tell the register file which registers to use:

- Need a value from 0 to 31

How many bits? ceil(lg(32)) = 5 bits

- Need to specify 5 bits for each register

Where to get the 10 bits for register numbers?

- From the instruction itself
- IR (instruction register)
  - Hidden register containing the instruction currently being executed

Register file: black box

Encoding for the instruction

000000	00010	00011	00001	00000	100000
b_{31-26}	b_{25-21}	b_{20-16}	b_{15-11}	b_{10-6}	b_{5-0}
opcode	$rs	$rt	$rd	shamt	function

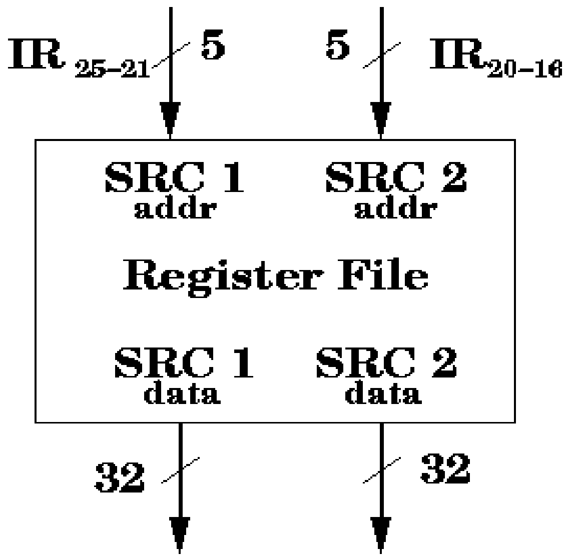
Good ISA design helps:

Can get the desired bits for the source registers without doing much decoding.

Once the ten bits are sent from the IR as ADDRESS inputs to the register file,

the register file produces 64 bits of DATA output

Implementation as a black box:



Notice that we haven't said anything yet about
HOW the address inputs
are turned into data outputs

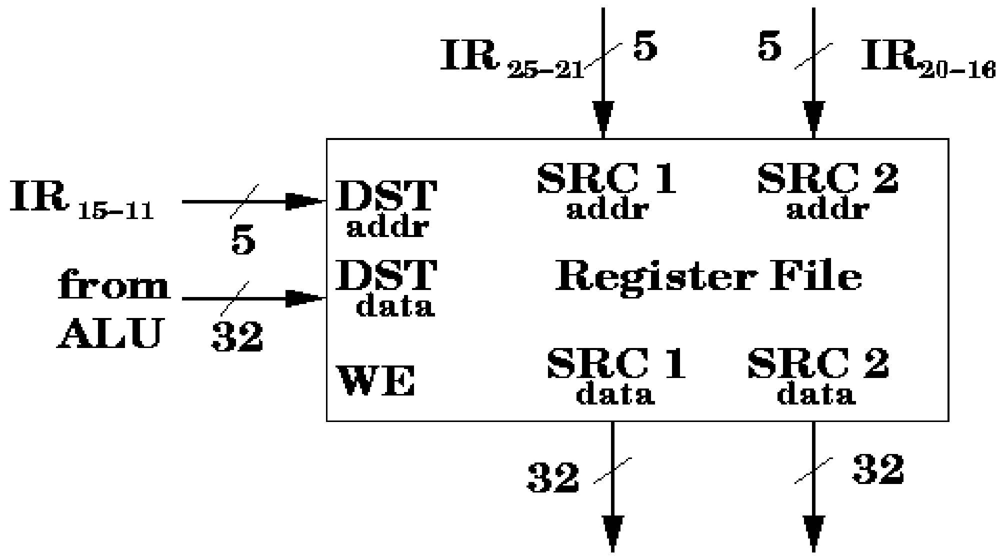
Register file: writing

Reading from a register file is only part of the story

Writing to a register file

- Control input called write enable (WE)
- When WE = 1, then we want to write to the register file
- Once we add R[2] to R[3], we need to save the result to destination register R[1]
- Require 5 bits to indicate the destination register (DST addr)
    - Can come from IR: bits $B_{15\text{-}11}$ are used for destination register (R-type)
- Need 32 bits of data to write to the register (DST data)
    - Data comes from the output of the ALU

How the register file looks now:



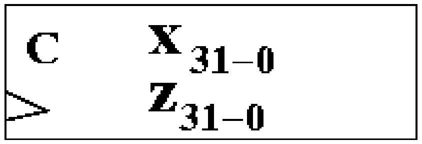
Summary:

- **Inputs**
  - **SRC 1 Addr: index of the first source register (5 bits)**
  - **SRC 2 Addr: index of the second source register (5 bits)**
  - **DST Addr: index of the destination register (5 bits)**
  - **DST Data: data to be written to the destination register (32 bits)**
  - **WE: DST Data is written to the register at index DST Addr only when WE = 1.**
- **Outputs**
  - **SRC 1 Data: contents of the first source register, as specified by SRC 1 Addr.**
  - **SRC 2 Data: contents of the second source register, as specified by SRC 2 Addr.**

Register file: implementing

Example: register file with 4 registers

Each parallel-load register



Input

- 32-bit data, x_{31-0}
- 1 control bit, c
- clock

Output

- 32-bit data, z_{31-0}

When c = 1, the register parallel loads, i.e., $z_{31-0} = x_{31-0}$.

When c = 0, it holds the value.

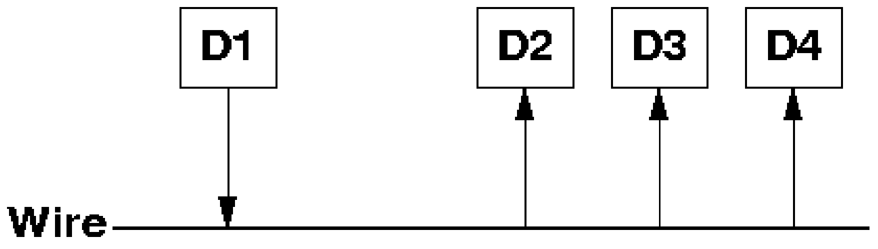
In order to organize the registers into a register file, we need the concept of a bus:

- composed of multiple wires

Wire

Wire: transmits a one-bit signal

- Connected device can write 0 or 1 to the wire
- At most 1 device can write to a wire at any given time
    - Result is undefined if:
        - More than 1 device attempts to write to the wire
        - No device is writing to the wire
- More than 1 device can read from the wire
- Wire has no memory: signal must be continuously asserted to be valid



- Example: 4 devices
    - D1 is writing to the wire
    - D2, D3, D4 are reading from the wire
- Assume each device has 2 connections to the wire (1 to read, 1 to write)
- What are the devices?
    - ALU, registers, gates, flip-flops, etc.

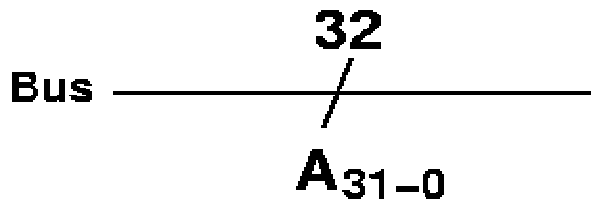
Bus

Bus consists of 1 or more wires

Real computer has multiple buses, for example: data, address, control

Size of bus: number of wires

- B_{31-0} can be used to indicate 32 wires in the bus
- B_{20-16} can be used to indicate a subset of wires



- Slash indicates multiple wires and the number indicates how many

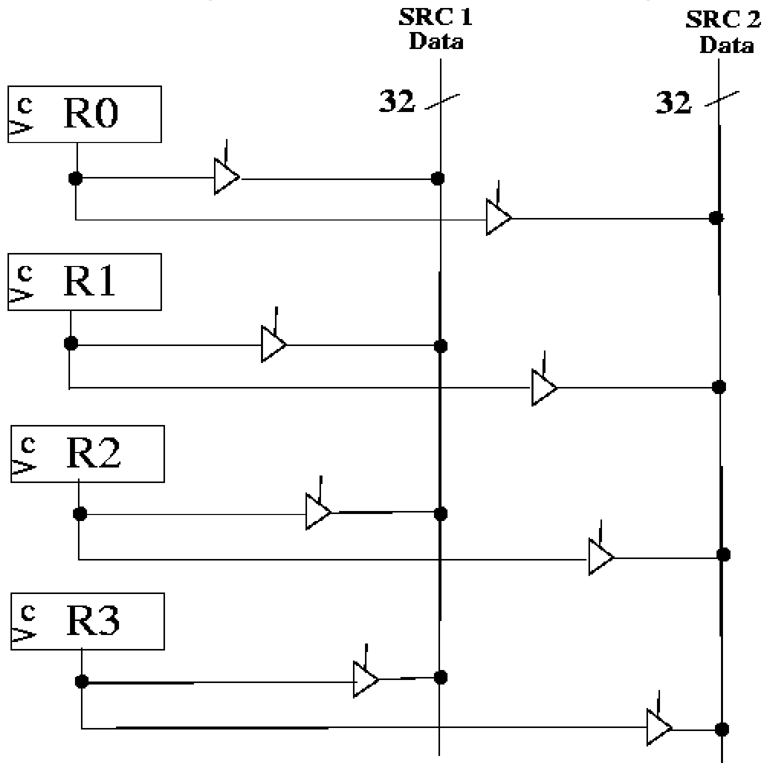
Bus allows any number of devices to connect, but only 1 may write at any given time

Why use bus?

- Alternative: connect all pairs of devices
  - Requires order of N^2 connections

Register file: implementing

Register file uses combinational logic with a set of registers
We attach the outputs of each register to two buses, using tri-state buffers for each.



How to select the right registers?

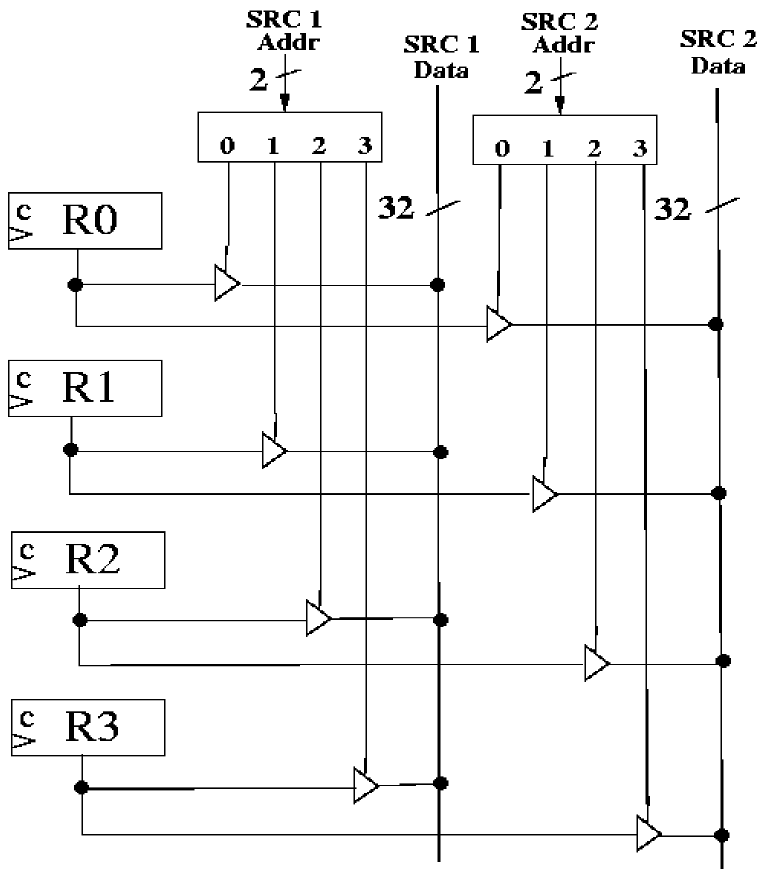
SRC 1 addr and SRC 2 addr

How many bits for 4 registers? ceil(lg(4)) = 2 bits

Register file: implementing

What device to use?

- 2 bits of input (register number)
- 4 bits of output, of which exactly one should have the value 1
  - (tri-state buffer control)
- 2-4 decoder
  - 2 bits of input (UB)
  - 1 bit of output

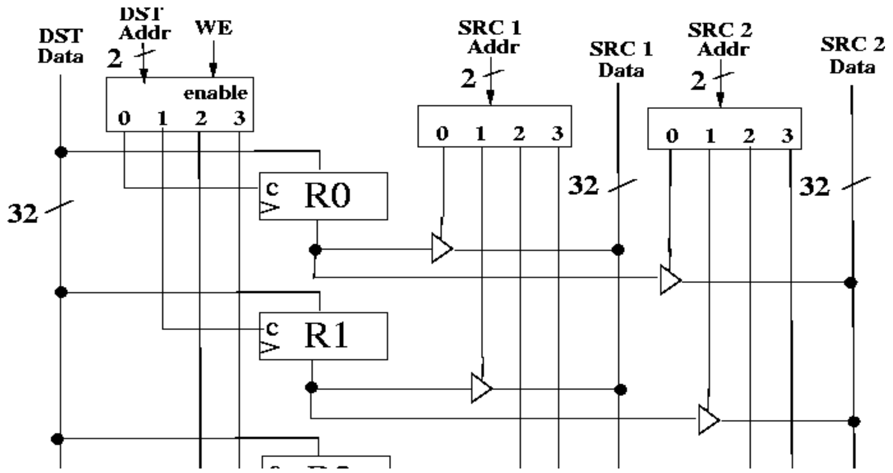


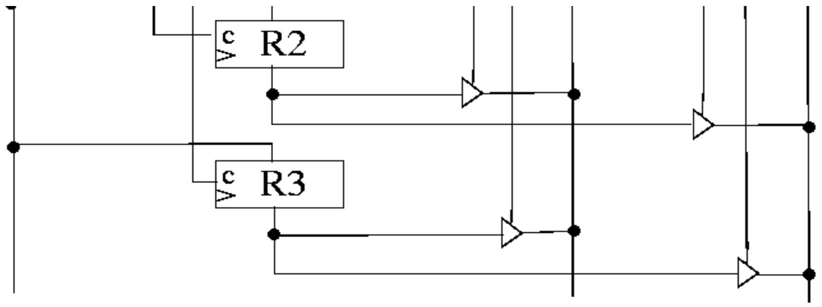
What about destination register?

Register file: implementing

Destination register

- **DST Addr: destination register address**
- **DST Data: data to store in destination register**
- **Write Enable (WE) indicates when to write the destination register**
  - **WE = 1: write to the register specified by DST Addr**
  - **WE = 0: don't do anything**
  - **2 bits of DST Addr tell which register to write**
  - **Want control input c of destination register to be 1 when WE = 1,**
    - **all other registers must have c = 0**
  - **What device to use?**
    - **2-4 decoder with enable or 1-4 DeMUX**
    - **choose decoder**





parts list:

- four parallel load registers
- two 2-4 decoders
- one 2-4 decoder with enable
- eight tri-state buffers
- three buses

anything else?