
CSE 220 Computer Organization

Lecture 19

Hussein Badr

Computer Science, Stony Brook University

<http://www.cs.sunysb.edu/~cse220>

Design & Control of Datapath (1/3)

- How do we connect the various building blocks that we have seen, such as the register file and the ALU?
- Goal is to execute instructions efficiently.
- Once various blocks are connected, how do we control the movement of data?
- We need to generate various control signals to accomplish this.
- The first step is to group machine instructions that are similar.
- Implement 'datapath' connections for each group.
- Then combine the results.

Design & Control of Datapath (2/3)

- For example, all instructions that have the R format and refer to ALU operations will be executed in a similar fashion.
- `add`, `sub`, `and`, `or`, `slt`, (ALU type instructions).
- 3 registers are specified.
- Control selects different operation of ALU.
- Instruction fetch, read two registers, send values to ALU for required operation, write result back to the destination register.

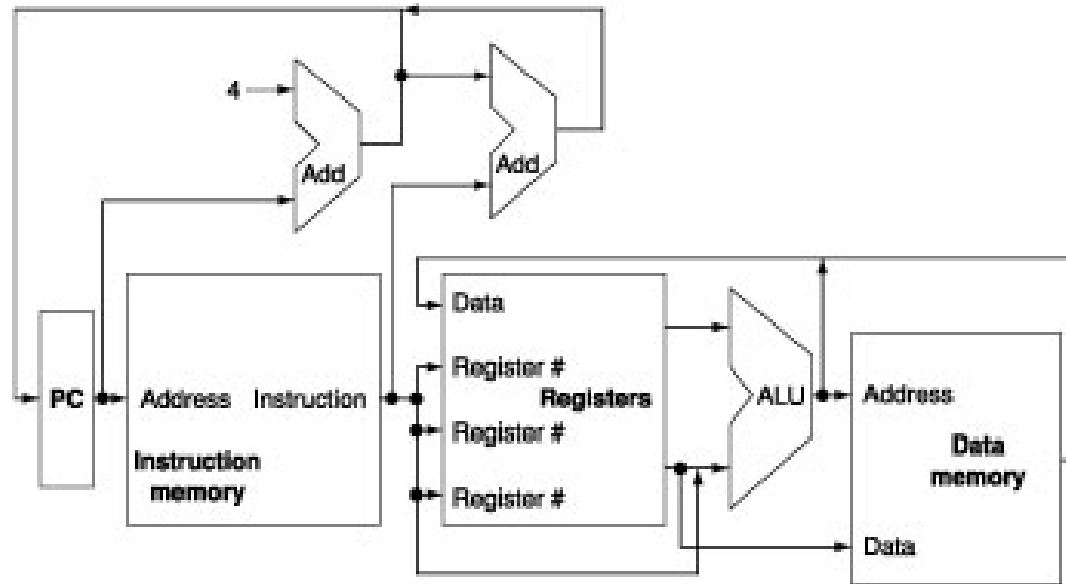
Design & Control of Datapath (3/3)

- Load and store instructions may be another group.
- These use base addressing + offset.
- They add an offset to the contents of a base register to calculate the correct memory address.
- For a load, we read from memory. For a store, we write to memory.

Load	Store
Instruction Fetch	Instruction Fetch
Register Read	Register Read
Calculate Address using ALU	Calculate Address using ALU
Read from memory	Write to memory
Write to a register	

Note that load takes more time than store

Simplified Datapath (1/2)



- **Abstract view of the datapath.**
- **Instruction memory (I-Cache).**
- **Instruction register.**
- **Data memory (D-Cache).**
- **ALU is used for address calculation and operand arithmetic.**

Simplified Datapath (2/2)

(repeat of figure on preceding slide)

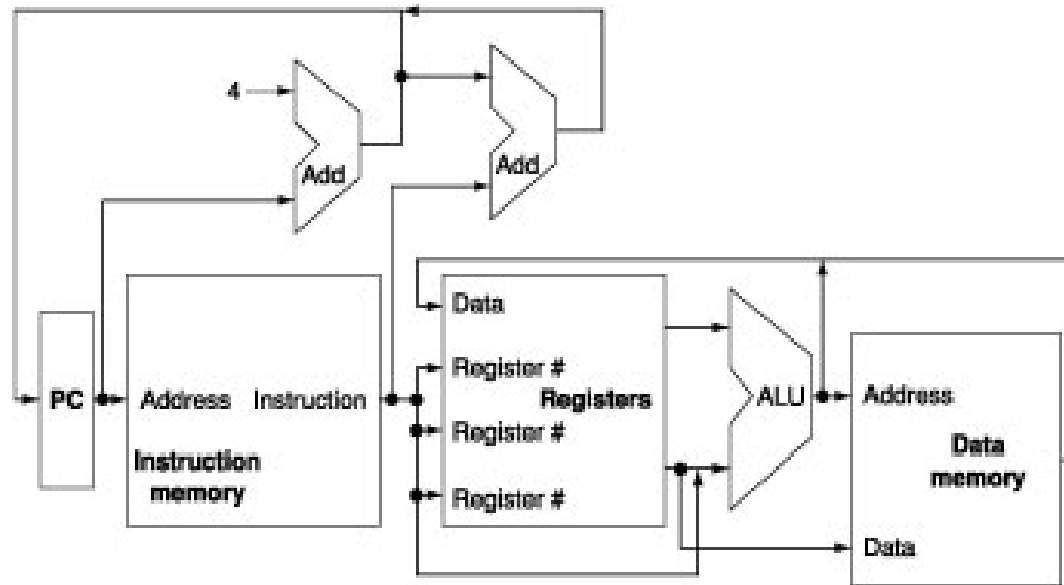
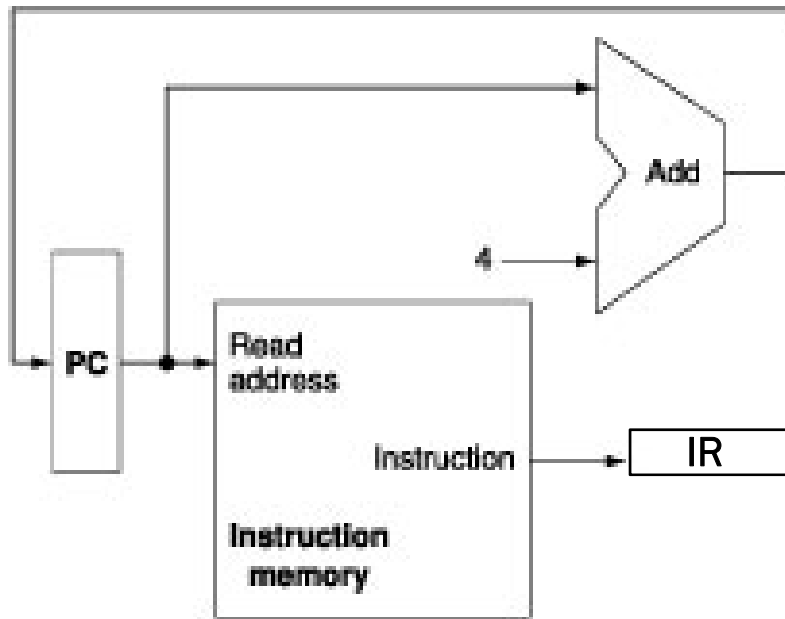


FIGURE 4.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

Datapath for Fetch



- **Instruction fetch**
- **After fetch, $PC \leftarrow PC+4$**
- **Instruction \rightarrow IR (Instruction Register)**
- **IR has various fields**
- **opcode, register numbers, 16 bit offset, *etc.***

FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

Datapath for Load / Store

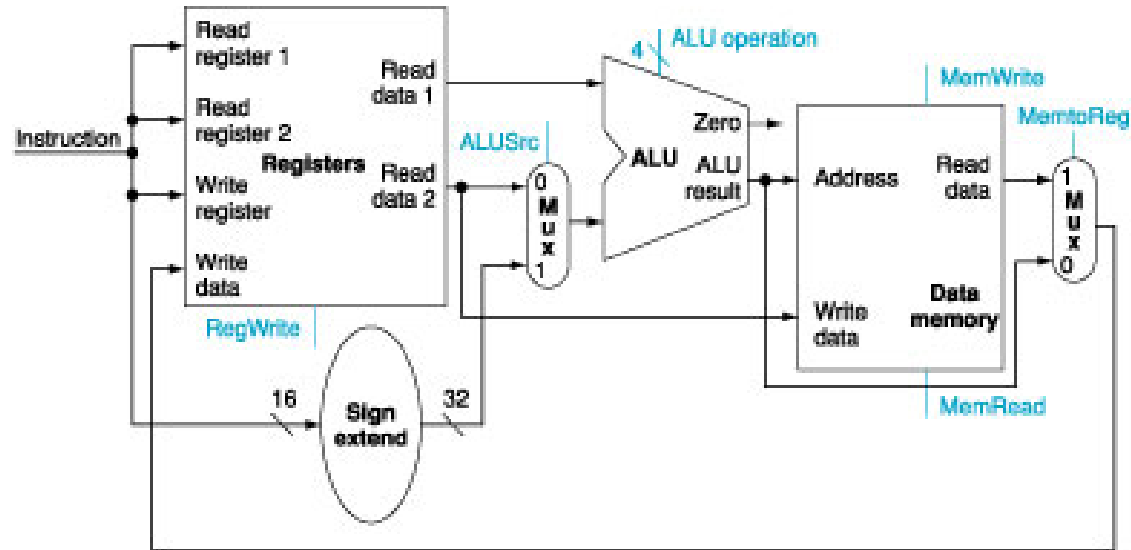


FIGURE 4.10 The datapath for the memory instructions and the R-type instructions.

- **Base addressing with 16-bit offset.**
- **Sign-extend the offset (16 → 32 bits): the ALU takes 32-bit operands.**
- **Contents of base register are read.**
- **ALU calculates final address = base address + offset .**
- **Load: memory read & register write.**
- **Store: memory write only.**

Datapath for R-format ALU-type Instruction

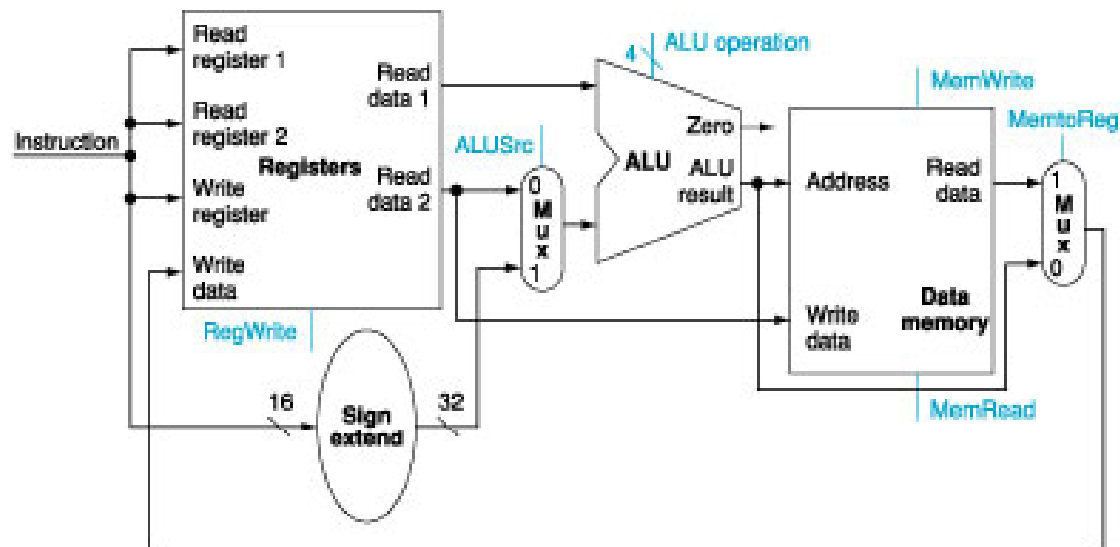


FIGURE 4.10 The datapath for the memory instructions and the R-type instructions.

- **After fetch, two registers are read; values pass through ALU.**
- **The result is stored in destination register.**
- **Used for all basic ALU operations instructions.**
- **Opcode will generate proper control signals.**

Datapath for Branch

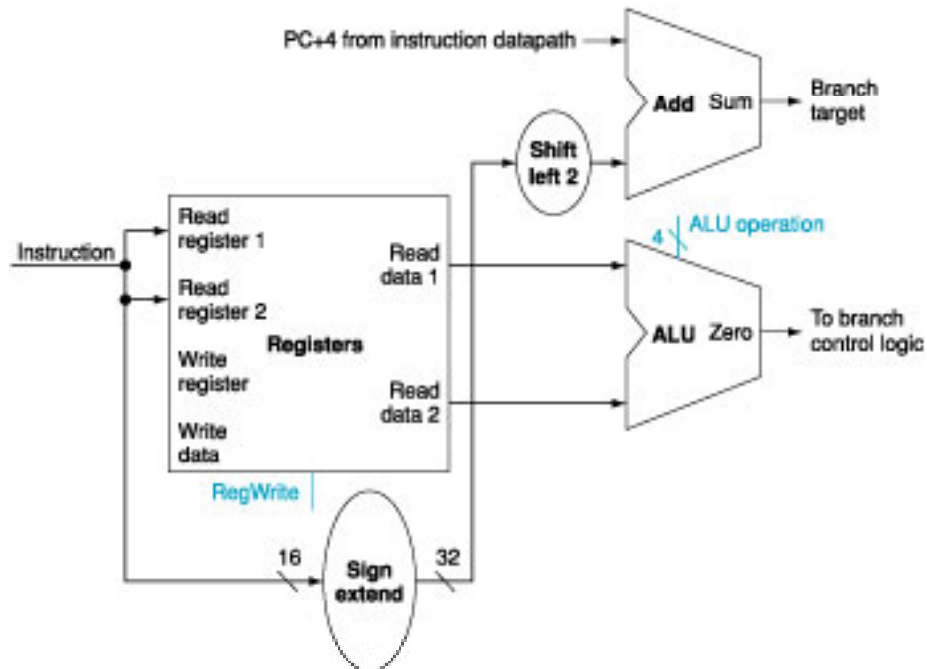
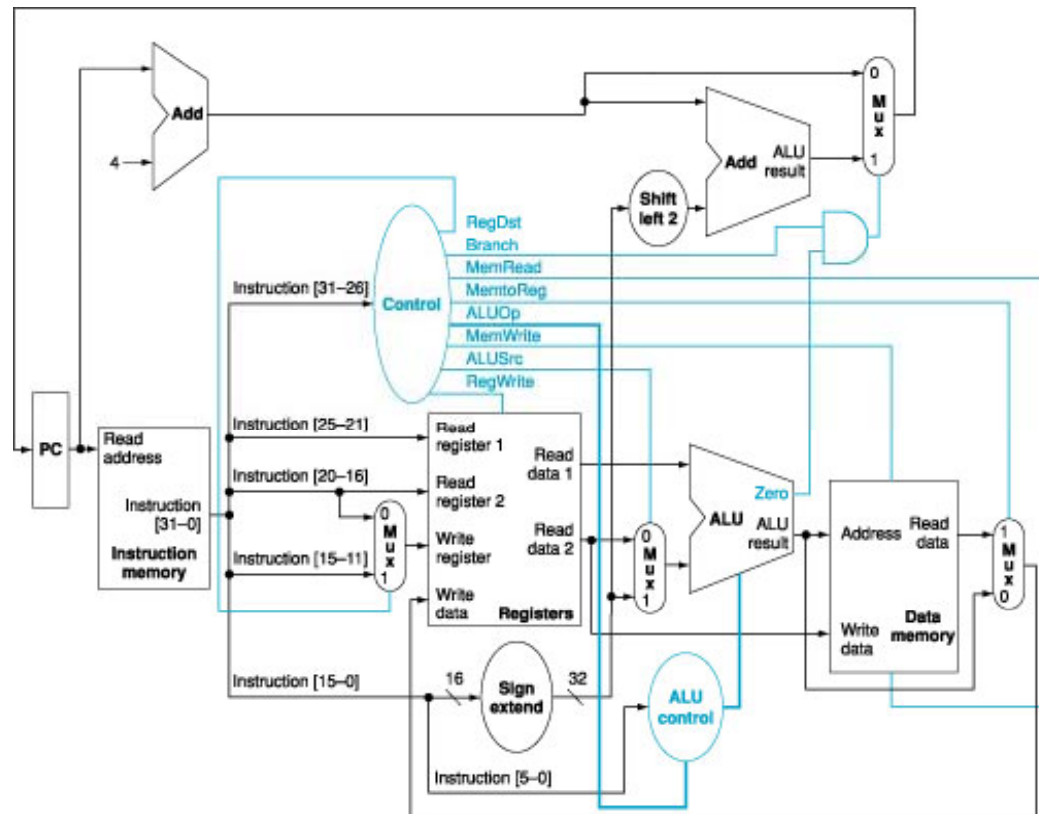


FIGURE 4.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the “shift” is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only “sign bits.” Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

- **Conditional branch datapath.**
- **ALU evaluates the condition.**
- **16-bit offset is sign extended to 32 bits.**
- **‘Left shift’ by 2 converts words to bytes (multiply by 4).**
- **Add this to (PC+4) using another adder.**

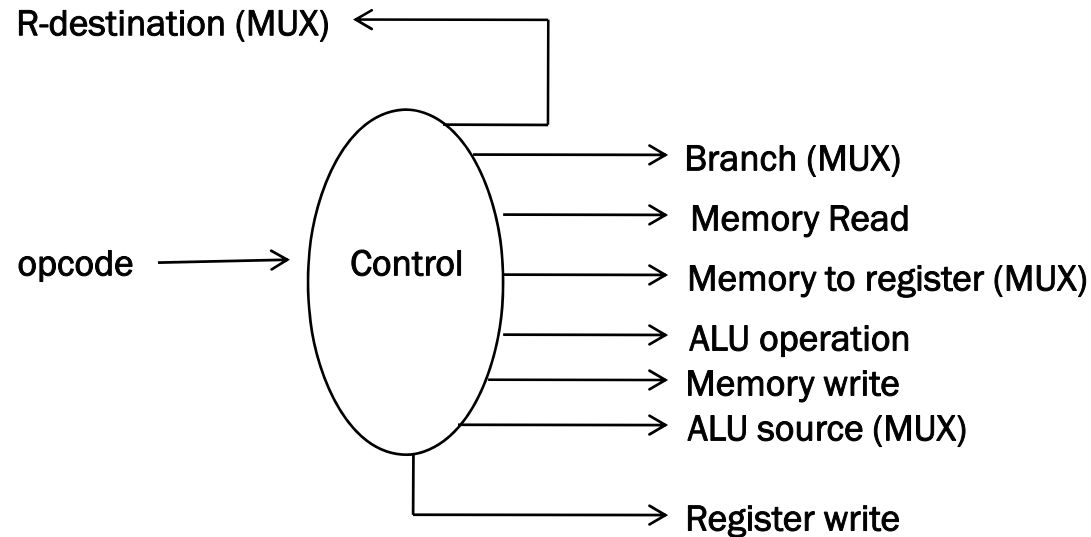
Combining the Datapaths

(figure is repeated on a larger scale in the next slide)



- All 3 data paths combined.
- Control signals operate ALU & MUXes.
- MUXes are required, as each instruction is controlled differently.

Opcode Control Signals



- **Each distinct opcode will have its specific control signals generated by the control unit.**
- **These signals control every sub-step of the instruction execution.**
- **The figure above gives a very simplified view.**
- **In reality, things are much more complex.**