CS152 Computer Architecture and Engineering

Bus-Based MIPS Implementation

Last Updated:
1/26/2010 12:59 AM

http://inst.eecs.berkeley.edu/~cs152/sp10

General Overview

Figure H1-A shows a diagram of a bus-based implementation of the MIPS architecture. In this architecture, the different components of the machine share a common 32-bit bus through which they communicate. Control signals determine how each of these components is used and which components get to use the bus during a particular clock cycle. These components and their corresponding control signals are described below.

For this handout, we shall use a positive logic convention. Thus, when we say signal X is "asserted", we mean that signal X is a logical 1, and that the wire carrying signal X is raised to the "HIGH" voltage level.

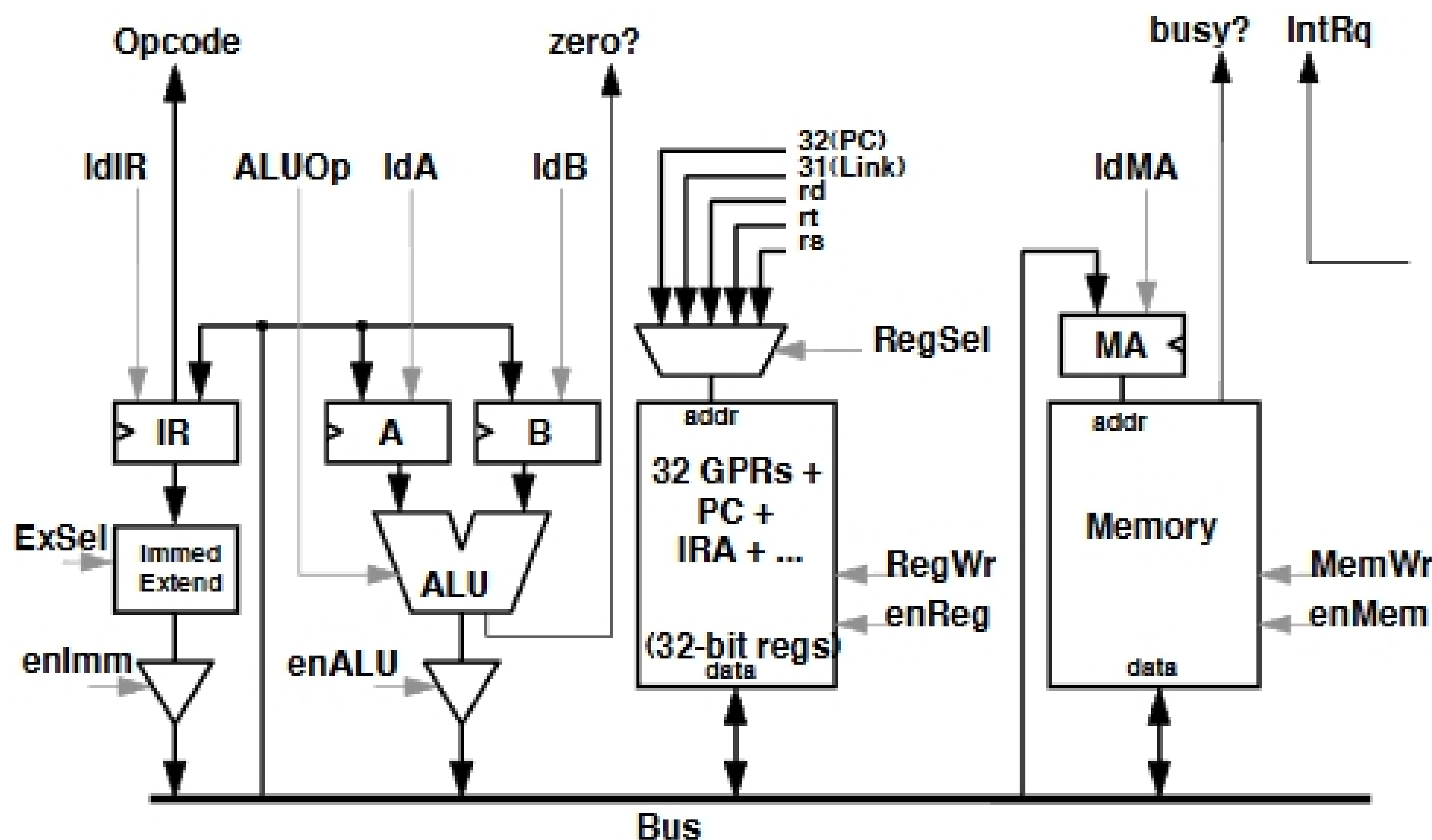

Figure H1-A: A bus-based datapath for MIPS.

The Enable Signals

Since the bus is shared by different components, there is a need to make sure that only one component is driving ("writing") the bus at any point in time. We do this by using tri-state buffers at the output of each component that can write to the bus. A tri-state buffer is a simple combinational logic buffer with enable control. When enable is 1, then the output of the buffer simply follows its input. When enable is 0, then the output of the buffer "floats" -- i.e., regardless of its input, the tri-state buffer will not try to drive any voltage on the bus. Floating the buffer's output allows some other component to drive the bus.[1]

In the bus-based MIPS, we have four enable signals: enImm, enALU, enReg, and enMem. As shown, enImm and enALU are connected directly to the enable of a tri-state buffer. On the other hand, enReg and enMem are used in more complex circuitry to be explained later in the section on the register file. When setting these signals, it is important to make sure that at any one time, *at most one* device should be driving the bus. It is possible not to assert any of the four signals. In this case, the bus will float and will have an undefined value.

Special Registers and Load Signals

In addition to the registers in the register file, the bus-based MIPS has four other special internal registers: IR, A, B, and MA. These registers are 32-bit *edge triggered* registers with load enable control. As shown, these registers take their data inputs from the bus. If a register's enable is asserted during a particular cycle, then the value on the bus during that cycle will be copied into the register at the *next* clock edge. If the enable control is 0, then register's value is not changed.

We call these register enable signals *load* signals, and give them names of the form "ldXX" (i.e., ldIR, ldA, ldB, and ldMA). In addition, the RegWr and MemWr signals are also load signals, but their exact functionality will be discussed later. It is possible to assert more than one load signal at a time. In this case, the value on the bus will be loaded to all registers whose load signals are asserted.

The Instruction Register

The instruction register is used to hold the current 32-bit instruction word. As explained in the Handout #4 (RISC ISA- MIPS64), the opcode and function fields (see the MIPS64 instruction format) are used by the microcode control hardware to identify the instruction and run the appropriate microcode. As shown in Figure H1-A, the immediate field is connected to a sign

[1] You can also think of a tri-state buffer as an electronically controlled switch. If enable is 1, then the switch connects the input and the output as if they were connected by a wire. If enable is 0, then the input is electrically disconnected from the output. Note that the tri-state buffer is a memoryless device, and is *not* the same as a latch or a flip-flop.

extender and then to the bus. Finally, as described below, the register specifier fields go to a multiplexer connected to the register file address input.

The Sign Extender

The box named "Immed Extend" in the diagram is a sign extender module that extends a number (16 bits or 26 bits) to a 32-bit number. The sign extender can have one of four possible values. If ExSel is uExt16, then no sign extension is performed, and the most significant bits are just padded with 0's. In this case, the input to the sign extender must be a 16-bit number. If ExSel is uExt26, then no sign extension is performed either, but the input to the sign extender must be a 26-bit number. If ExSel is sExt16, then the number is sign extended by taking its MSB and using it to pad the most significant bits of the 32-bit value. (Note: we use 2's complement to represent negative numbers.) In this case, the input to the sign extender must be a 16-bit number. If ExSel is sExt26, then the input must be a 26-bit number, and it is sign extended by taking its MSB and using it to pat the most significant bits of the 32-bit value.

The ALU

The ALU takes 3 inputs: two 32-bit operand inputs, connected to the A and B registers, and an ALUOp input. ALUOp selects the operation to be performed on the operands. Assume that the ALU can perform the following operations by default, if not explicitly stated otherwise:

ALUOp	ALU Result Output
COPY_A	A
COPY_B	B
INC_A_1	A+1
DEC_A_1	A-1
INC_A_4	A+4
DEC_A_4	A-4
ADD	A+B
SUB	A-B

Table H1-1: ALU Operations for Handout #5.

In order to implement the entire MIPS ISA, we will need a few more ALU operations.

The ALU is purely combinational logic. It has two outputs, a 32-bit main result output, and 1-bit zero flag output, *zero*. The result output is computed as in Table H1-1. The zero flag simply indicates if the ALU result output equals to zero. If the result is 0 then *zero* is 1, otherwise *zero* is 0. For example, if A=2, B=2, and ALUOp=SUB, then the ALU result will be 0, and *zero* will be 1. This flag is used to do conditional branches in microcode.