# CS152 Computer Architecture and Engineering

## Bus-Based RISC-V Implementation

---

**http://inst.eecs.berkeley.edu/~cs152/sp18**

---

## General Overview

Figure H1-A shows a diagram of a bus-based implementation of the RISC-V architecture.[1] 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.
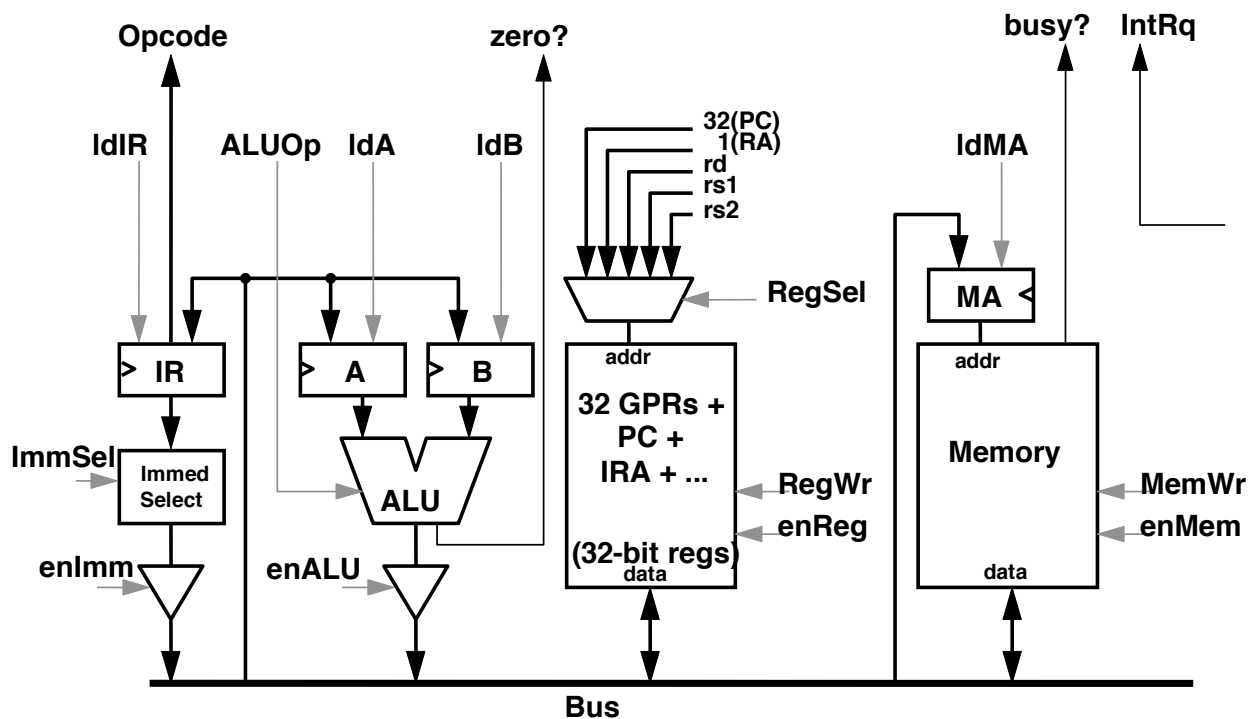


Figure H1-A: A bus-based datapath for RISC-V.

---

[1] This document discusses an implementation of RV32, the 32-bit address space variant of RISC-V. RV32 uses 32-bit wide general purpose registers and a 32-bit wide PC register. A RV64 micro-coded processor would appear identical to the one presented in this document, with the exception of 64-bit registers, a 64-bit bus, and some additional ALU operations to support the new instructions added to RV64.

**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.[2]

In the bus-based RISC-V, 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 RISC-V 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 "RISC-V Instruction Set Manual", the opcode and function fields (see the RISC-V 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 an

---

[2] 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.

immediate selector and 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 Immediate Selector & Sign Extender

The box named "Immed Select" in the diagram is an immediate select and sign extender module that selects the immediate from the instruction register and sign extends the immediate to a 32-bit number. The sign extender can have one of five possible values: IType, LType, JType, BsType, and BrType. IType selects the immediate for I-Type instructions; LType selects for L-type instructions, JType selects for J-Type instructions, BsType selects for B-Type instructions used by store instructions; and BrType selects for B-Type instructions used by branch instructions. All immediates are signed-extended.

### 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 |
| SLT | Signed(A) < Signed(B) |
| SLTU | A < B |

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

In order to implement the entire RISC-V 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.

**The Register File**

The register file contains the 32 general-purpose registers (GPRs), the PC, and a few other special-purpose registers. The register file itself has a 6-bit address input (addr) and a 32-bit data port. The address input determines which register is to be read or written to.

Two control signals determine how the register file is used during a certain cycle: RegWr and enReg. RegWr determines whether the operation to be performed, if any, is a read or a write. If RegWr is 1, then it is a write, otherwise it's a read. enReg is a general enable control for the register file. If enReg is 1, then the register reads or writes depending on RegWr. If enReg is 0, then nothing is done, regardless of the value of RegWr. Figure H1-B shows exactly how enReg and RegWr are wired to the register controls.

The address input (addr) determines which register in the file is to be used. Read operations are assumed to be combinational. That is, if you change the value at the addr input, then the value at the data output will change appropriately (after some delay), *even if* no clock edge occurs. Write operations, on the other hand, are edge triggered, such that data from the register's data input is only stored in the selected address at the next clock edge.
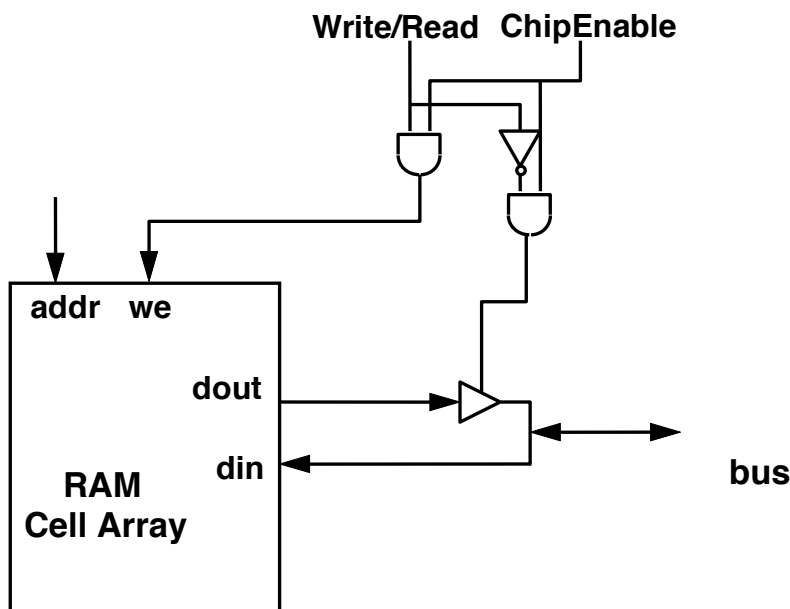


Figure H1-B: Control signals and bus connections for the register file and the memory.

While the addr input selects which register is used, the RegSel control signal selects what value is used as the addr input. As shown, RegSel controls a multiplexer that chooses between (at least) 5 possible values. The first two are the hardwired values 32 and 1, corresponding to the PC and the return address (RA) register respectively. The next three, rs1, rs2, and rd, are taken from the register specifier fields of the IR. The following figure shows which fields rs1, rs2, and rd correspond to:

| rd | rs1 | rs2 | … | opcode |
|---|---|---|---|---|
| 5 bits | 5 bits | 5 bits | 10 bits | 7 bits |

The exact meaning of these fields depends on the instruction and instruction-type. Please refer to the RISC-V ISA manual and the lecture notes. Note that these fields are 5 bits wide, while addr is 6 bits wide. This is because our register file here contains more than just the 32 GPR's. Just assume that the 5-bit fields are converted to a 6-bit address by padding the MSB with a 0. When specifying the value of RegSel, you should just use the symbols PC, RA, rs1, rs2, and rd.

**The Memory**

The memory module is very similar to the register file except that the address input is taken from an edge triggered register, MA. Thus, it takes two steps to access a particular memory location. First, you load the MA with the desired memory address. Then, you perform the desired memory operation. The MemWr and enMem controls work just like RegWr and enReg. Any operations to be performed on memory are performed on the location specified by the current value of MA. As in the register file, we assume that reads are combinational, while writes are edge triggered.

The main difference between memory module and the register file is the busy signal. Unlike the register file, the memory may take more than one cycle to complete a read or write (e.g., if you get a cache miss). The busy signal indicates that the memory isn't done reading or writing yet. The microcode can then respond to the busy signal appropriately.

**The Clock Period and Timing Issues**

We will assume that the clock period is long enough to guarantee that the results of all *combinational* logic paths are valid and stable before the setup time of any edge triggered components attempting to latch these results.

Remember that these combinational paths include not only computational elements like the sign extender, and the ALU, but also the register file and the memory during *read* operations. Specifically, the path from addr to the data output in both the register file and the memory is purely combinational. As mentioned above, if you change the value at the addr input, then the value at the data output will change appropriately (after some delay), *even if* no clock edge occurs.

Also remember that the path from *data input* to data output is *not* combinational. This applies not only to the register file and memory, but also to other edge triggered registers (i.e., IR, A, B, and MA). When performing a write to any of these components, the value at the data input is not stored until the *next* clock edge, and can thus only be read during the next clock cycle.

Finally, we will assume that there are no hold time violation problems.

## Microprogramming on the Bus-Based RISC-V Implementation

In the past, simple CISC machines employed a bus-based architecture wherein the different components of the machine (ALU, memory, etc.) communicated through a common bus. This type of architecture is easy and inexpensive to implement, but has the disadvantage of requiring all data movement between components to use the bus. Because of this bottleneck, an instruction on such an architecture typically took several cycles to execute.

Microprogramming makes it easy to generate the control signals for such multi-cycle instructions. A microcode program is basically a finite state machine description. Each line of microcode corresponds to some state of the machine, and contains a "microinstruction" which specifies the outputs for that state, and the next state. The outputs are used to control the different components in the datapath. The next state depends on the current state and possibly on certain other signals (e.g., condition flags). As we shall see, this simple FSM model proves to be very powerful, allowing complex operations like conditional branches and loops to be performed.

Table H1-3 in Appendix B shows a microcode table for the bus-based RISC-V implementation. Some lines have been filled in as examples. The first column in the microcode table contains the state label of each microinstruction. For brevity, we assume that the states are listed in increasing numerical order, and we only label important states. For example, we label state FETCH0, and assume that the unlabeled line immediately following it corresponds to state (FETCH1). The second column contains a pseudo-code explanation of the microinstruction in RTL-like notation. The rest of the columns, except the last two, represent control signals that are asserted during the current cycle. 'Don't care' entries are marked with a '*'.

The last two columns specify the next state. The μBr (*micro*branch) column represents a 3-bit field with six possible values: N, J, EZ, NZ, D and S. If μBr is N (next), then the next state is simply (*current state* + 1). If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column (i.e., it's an unconditional microbranch). If it is EZ (branch-if-equal-zero), then the next state depends on the value of the ALU's *zero* output signal (i.e., it's a conditional microbranch). If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1). NZ (branch-if-not-zero) behaves exactly like EZ, except NZ branches to the next state is *zero* is not asserted (!= 1). If μBr is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state. In this handout, we assume that the dispatch goes to the state labeled (RISC-V-instruction-name + "0"). For example, if the instruction in the IR is SW, then the dispatch will go to state SW0. If μBr is S, then the next state depends on the value of the *busy?* signal. If *busy?* is asserted, then the next state is the same as the *current state* (the μPC "spins" on the same microaddress), otherwise, the next state is (*current state* + 1).

The first three lines in the table (starting at FETCH0) form the *instruction fetch* stage. These lines are responsible for fetching the next instruction opcode, incrementing the PC, and then going to the appropriate microcode line based on the particular opcode fetched. The instruction fetch stage is performed for every instruction, and every instruction's microcode should always end by executing an instruction fetch for the next instruction. The easiest way to do this is by having the last microinstruction of an instruction's microcode do a microbranch to FETCH0. The microcode for NOP provides a simple example. For the rest of the instructions, their microcodes were not given intentionally; we expect students to fill out the table themselves.

## Appendix A. A Cheat Sheet for the Bus-based RISC-V Implementation

For your reference, we've also included the actual bus-based datapath as well as rehash of some important information about microprogramming in the bus-based architecture.

Remember that you can use the following ALU operations:

| 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 |
| SLT | Signed(A) < Signed(B) |
| SLTU | A < B |

Table H1-2: Available ALU operations

Also remember that μBr (*micro*branch) column in Table H1-3 represents a 3-bit field with six possible values: N, J, EZ, NZ, D, and S. If μBr is N (next), then the next state is simply (*current state* + 1). If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column (i.e., it's an unconditional microbranch). If it is EZ (branch-if-equal-zero), then the next state depends on the value of the ALU's *zero* output signal (i.e., it's a conditional microbranch). If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1). NZ (branch-if-not-zero) behaves exactly like EZ, but instead performs a microbranch if *zero* is not asserted (!= 1). If μBr is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state. If S, the μPC spins if *busy?* is asserted, otherwise goes to (*current state* +1).
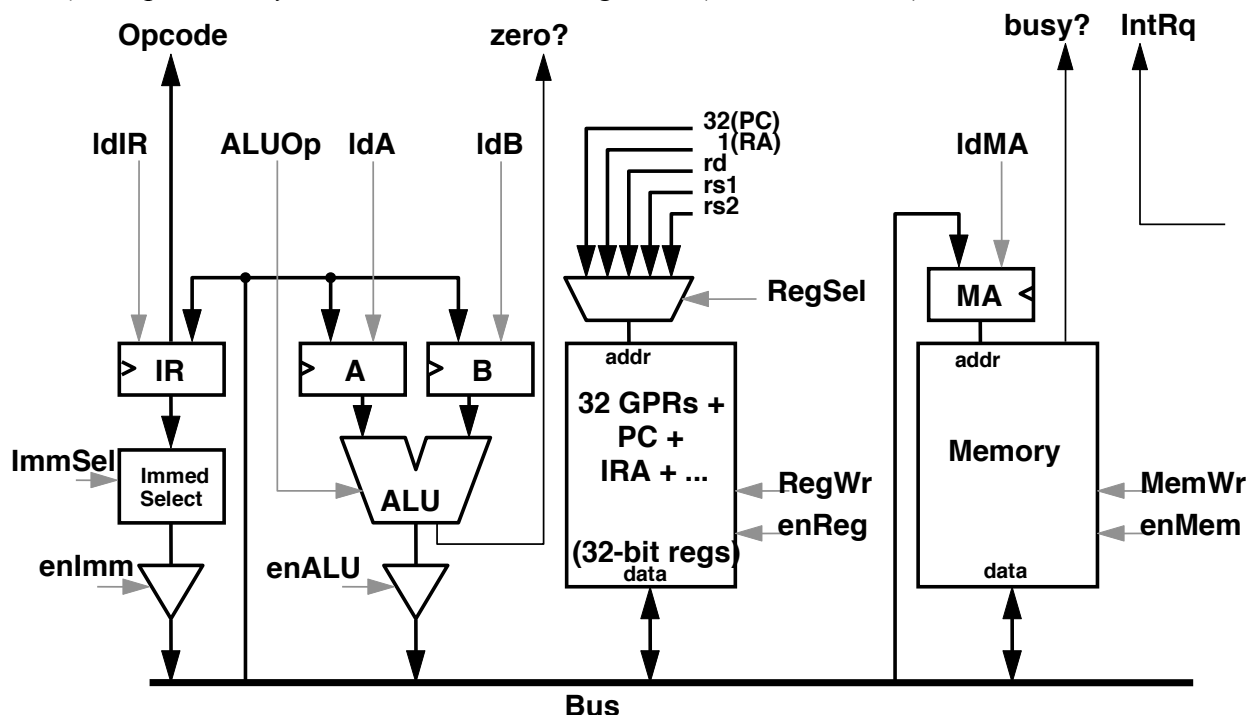
## Appendix B. Microcode Table for the Bus-based RISC-V Implementation

| State | PseudoCode | ldIR | Reg Sel | Reg Wr | en Reg | ldA | ldB | ALUOp | en ALU | ld MA | Mem Wr | en Mem | Imm Sel | en Imm | μBr | Next State |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FETCH0: | MA <- PC; A <- PC | * | PC | 0 | 1 | 1 | * | * | 0 | 1 | * | 0 | * | 0 | N | * |
|  | IR <- Mem | 1 | * | * | 0 | 0 | * | * | 0 | 0 | 0 | 1 | * | 0 | S | * |
|  | PC <- A+4 | 0 | PC | 1 | 1 | 0 | * | INC_A_4 | 1 | * | * | 0 | * | 0 | D | * |
| . . . |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| NOP0: | microbranch back to FETCH0 | * | * | * | 0 | * | * | * | 0 | * | * | 0 | * | 0 | J | FETCH0 |
| ADDM0: |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Table H1-3 (Worksheet 1)