4.3 Building a datapath

A reasonable way to start a datapath design is to examine the major components required to execute each class of LEGv8 instructions. Let's start at the top by looking at which *datapath elements* each instruction needs, and then work our way down through the levels of **abstraction**. When we show the datapath elements, we will also show their control signals. We use abstraction in this explanation, starting from the bottom up.

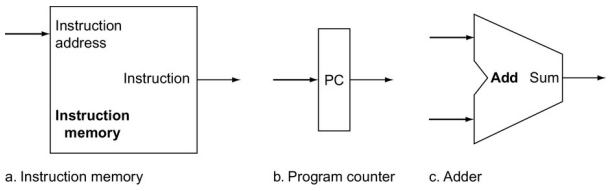
Datapath element: A unit used to operate on or hold data within a processor. In the LEGv8 implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

Item a in the figure below shows the first element we need: a memory unit to store the instructions of a program and supply instructions given an address. Item b in the figure below also shows the *program counter (PC)*, which as we saw in COD Chapter 2 (Instructions: Language of the Computer) is a register that holds the address of the current instruction. Lastly, we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU described in detail in COD Appendix A (The Basics of Logic Design) simply by wiring the control lines so that the control always specifies an add operation. We will draw such an ALU with the label *Add*, as in the figure below, to indicate that it has been permanently made an adder and cannot perform the other ALU functions.

Program counter (***PC***): The register containing the address of the instruction in the program being executed.

Figure 4.3.1: Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address (COD Figure 4.5).

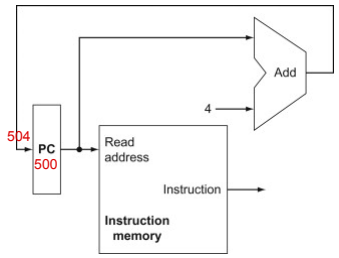
The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 64-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 64-bit inputs and place the sum on its output.



To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. The animation below shows how to combine the three elements from the above figure to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

PARTICIPATION ACTIVITY 4.3.1: A portion of the datapath used for fetching instructions and incrementing the program counter (COD Figure 4.6).

Start ☐ 2x speed



Now let's consider the R-format instructions (see COD Figure 2.21 (LEGv8 instruction formats)). They all read two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical operations). This instruction class includes `ADD`, `SUB`, `AND`, and `ORR`, which were introduced in COD Chapter 2 (Instructions: Language of the Computer). Recall that a typical instance of such an instruction is `ADD X1, X2, X3`, which reads `X2` and `X3` and writes the sum into `X1`.

The processor's 32 general-purpose registers are stored in a structure called a *register file*. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer. In addition, we will need an ALU to operate on the values read from the registers.

Register file: A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.

R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction. For each data word to be read from the registers, we need an input to the register file that specifies the *register number* to be read and an output from the register file that will carry the value that has been read from the registers. To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the *data* to be written into the register. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge. Item a in the figure below shows the result; we need a total of four inputs (three for register numbers and one for data) and two outputs (both for data). The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 64 bits wide.

As illustrated in the animation below, the register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in COD Section A.8 (Memory elements: Flip-flops, latches, and registers) of Appendix A (The Basics of Logic Design). The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 64 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in COD Appendix A (The Basics of Logic Design). We will use the Zero detection output of the ALU shortly to implement conditional branches.

PARTICIPATION ACTIVITY 4.3.2: Fetching instructions and incrementing the PC.

Consider a rising clock edge that causes 3000 to be written into the PC.

1) The 3000 waits at the instruction memory input for the next rising clock edge, at which time the instruction at address 3000 is read out.
   - ○ True
   - ○ False

2) After the address 3000 is read into the PC, the 3000 only propagates to the adder.
   - ○ True
   - ○ False

3) The 3000 waits at the adder input for the next rising clock edge.
   - ○ True
   - ○ False

4) 3001 will be waiting at the PC's input to be written on the next rising clock edge.
   - ○ True
   - ○ False

PARTICIPATION ACTIVITY 4.3.3: The two elements needed to implement R-format ALU operations are the register file and the ALU (COD Figure 4.7).

Start ☐ 2x speed

a. Registers b. ALU

Item b in the animation above shows the ALU, which takes two 64-bit inputs and produces a 64-bit result, as well as a 1-bit signal if the result is 0. The 4-bit control signal of the ALU is described in detail in COD Appendix A (The Basics of Logic Design); we will review the ALU control shortly when we need to know how to set it.

PARTICIPATION ACTIVITY 4.3.4: Register file and ALU.

1) The register file always outputs the two registers' values for the two input read addresses.
   - ○ True
   - ○ False

2)

The register file writes to one register on every rising clock edge.

- O True
- O False

3) The design can read from two registers and write to one register during the same clock cycle.

- O True
- O False

4) The programmer must take care not to create a program that writes to a register during the same cycle that the same register is read.

- O True
- O False

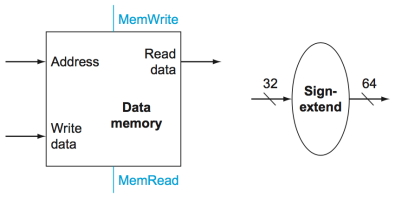
Next, consider the LEGv8 load register and store register instructions, which have the general form `LDUR X1, [X2, offset_value]` or `STUR X1, [X2, offset_value]`. These instructions compute a memory address by adding the base register, which is `X2`, to the 9-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in `X1`. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is `X1`. Thus, we will need both the register file and the ALU from the animation above.

In addition, we will need a unit to *sign-extend* the 9-bit offset field in the instruction to a 64-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory. The figure below shows these two elements.

Sign-extend: To increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.

Figure 4.3.2: The two units needed to implement loads and stores, in addition to the register file and ALU, are the data memory unit and the sign extension unit (COD Figure 4.8).

The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy). The sign extension unit has a 32-bit instruction as input that selects a 9-bit for load and store or a 19-bit field for compare and branch on zero that is sign extended into a 64-bit result appearing on the output (see COD Chapter 2 (Instructions: Language of the Computer)). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See COD Section A.8 (Memory elements: Flip-flops, latches, and registers) of Appendix A for further discussion of how real memory chips work.



a. Data memory unit b. Sign extension unit

The `CBZ` instruction has two operands, a register that is tested for zero, and a 19-bit offset used to compute the *branch target address* relative to the branch instruction address. Its form is `CBZ X1, offset`. To implement this instruction, we must compute the branch target address by adding the sign-extended offset field of the instruction to the PC. There are two details in the definition of branch instructions (see COD Chapter 2 (Instructions: Language of the Computer)) to which we must pay attention:

- The instruction set architecture specifies that the base for the branch address calculation is the address of the branch instruction.
- The architecture also states that the offset field is shifted left 2 bits so that it is a word offset; this shift increases the effective range of the offset field by a factor of 4.

To deal with the latter complication, we will need to shift the offset field by 2.

Branch target address: The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the LEGv8 architecture, the branch target is given by the sum of the offset field of the instruction and the address of the branch.

As well as computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address. When the condition is true (i.e., the operand is zero), the branch target address becomes the new PC, and we say that the *branch* is *taken*. If the operand is not zero, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the *branch* is *not taken*.

Branch taken: A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches.

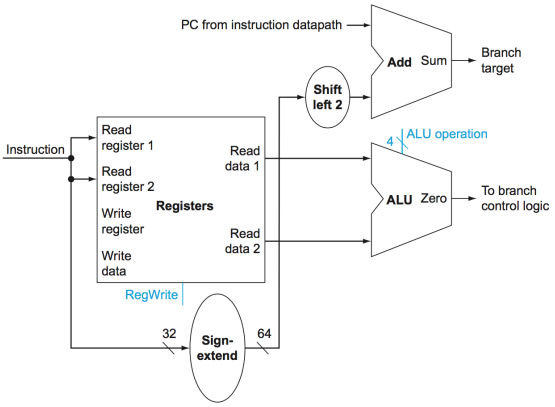
Branch not taken or (***untaken branch***): A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

Thus, the branch datapath must do two operations: compute the branch target address and test the register contents. (Branches also affect the instruction fetch portion of the datapath, as we will deal with shortly.) The figure below shows the structure of the datapath segment that handles branches. To compute the branch target address, the branch datapath includes a sign extension unit, from COD Figure 4.8 (The two units needed to implement loads and stores …) and an adder. To perform the compare, we need to use the register file shown in COD Figure 4.7a (The two elements needed to implement R-format ALU operations …) to supply the register operand (although we will not need to write into the register file). In addition, the comparison can be done using the ALU we designed in COD Appendix A (The Basics of Logic Design). Since that ALU provides an output signal that indicates whether the result was 0, we can send the register operand to the ALU with the control set to pass the register value. If the Zero signal out of the ALU unit is asserted, we know that the register value is zero. Although the Zero output always signals if the result is 0, we will be using it only to implement the compare test of conditional branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.

The branch instruction operates by adding the PC with the lower 26 bits of the instruction shifted left by 2 bits. Simply concatenating 00 to the branch offset accomplishes this shift, as described in COD Chapter 2 (Instructions: Language of the Computer).

Figure 4.3.3: 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 PC and the sign-extended 19 bits of the instruction (the branch displacement), shifted left 2 bits (COD Figure 4.9).

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 19 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.



Creating a single datapath

Now that we have examined the datapath components needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation. This simplest datapath will attempt to execute all instructions in one clock cycle. This design means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. We therefore need a memory for instructions separate from one for data. Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.

Example 4.3.1: Building a datapath.

The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are the following:

- The arithmetic-logical instructions use the ALU, with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign-extended 9-bit offset field from the instruction.
- The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

Show how to build a datapath for the operational portion of the memory-reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary multiplexors.

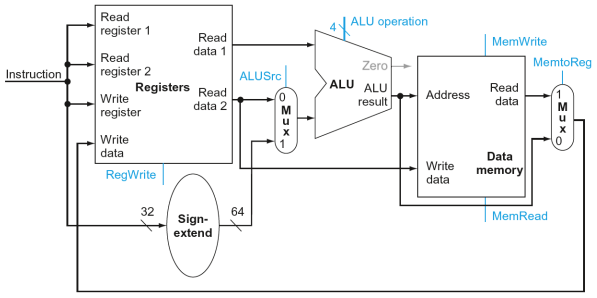
Answer

To create a datapath with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file. The figure below shows the operational portion of the combined datapath.

Figure 4.3.4: The datapath for the memory instructions and the R-type instructions (COD Figure 4.10).

This example shows how a single datapath can be assembled from the pieces in COD Figures 4.7

(The two elements needed to implement R-format ALU operations …) and 4.8 (The two units needed to implement loads and stores …) by adding multiplexors. Two multiplexors are needed, as described in the example.



Now we can combine all the pieces to make a simple datapath for the core LEGv8 architecture by adding the datapath for instruction fetch (COD Figures 4.6 (A portion of the datapath used for fetching instructions …)), the datapath from R-type and memory instructions (COD Figure 4.10 (The datapath for the memory instructions and the R-type instructions)), and the datapath for branches (COD Figure 4.9 (The datapath for a branch uses the ALU …)). The animation below shows the datapath we obtain by composing the separate pieces. The branch instruction uses the main ALU to test the register operand, so we must keep the adder from COD Figure 4.9 (The datapath for a branch uses the ALU …) for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC.

The components come from COD Figures 4.6 (A portion of the datapath used for fetching instructions and incrementing the program counter), 4.9 (The datapath for a branch uses the ALU …), and 4.10 (The datapath for the memory instructions and the R-type instructions). This datapath can execute the basic instructions (load-store register, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches. The support for unconditional branches will be added later.

PARTICIPATION ACTIVITY 4.3.5: The simple datapath for the core LEGv8 architecture combines the elements required by different instruction classes (COD Figure 4.11).

Start ☐ 2x speed

PARTICIPATION ACTIVITY 4.3.6: The LEGv8 datapath.

Consider the LEGv8 datapath above. Find the error in each of the following statements.

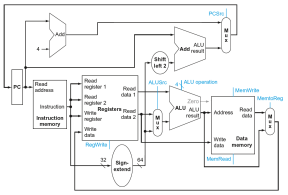
1) An R-type instruction like add uses three datapath units: the register file, the ALU, and the data memory.
2) In addition to the register file and ALU, a load or store instruction also involves the sign extension, data memory, and shift left units.
3) The branch datapath uses the sign extension, shift by 2, and data memory units.
4) During a load instruction, the mux to the data memory's right would pass the top / bottom input.
5) The mux at the upper right either passes PC + 1 (the normal case), or passes a target address from the instruction (for branches).

Now that we have completed this simple datapath, we can add the control unit. The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control. The ALU control is different in a number

of ways, and it will be useful to design it first before we design the rest of the control unit. The design of the control unit is discussed in another section (COD Section 4.4 (A Simple Implementation Scheme)).

PARTICIPATION ACTIVITY 4.3.7: Check yourself: Load instruction.

1) Which of the following is correct for a load instruction?



- O MemtoReg should be set to cause the data from memory to be sent to the register file.
- O MemtoReg should be set to cause the correct register destination to be sent to the register file.
- O We do not care about the setting of MemtoReg for loads.

2) The single-cycle datapath conceptually described in this section *must* have separate instruction and data memories, because _____.

- O the formats of data and instructions are different in LEGv8, and hence different memories are needed
- O having separate memories is less expensive
- O the processor operates in one cycle

Elaboration

The sign extension logic must choose between sign-extending a 9-bit field in instruction bits 20:12 for data transfer instructions or a 19-bit field (bits 23:5) for the conditional branch. Since the input is all 32 bits of the instruction, it can use the opcode bits of the instruction to select the proper field. LEGv8 opcode bit 26 happens to be 0 for data transfer instructions and 1 for conditional branch. Thus, bit 26 can control a 2:1 multiplexor inside the sign extension logic that selects the 9-bit field if it is 0 or the 19-bit field if it is 1.

Provide feedback on this section