4.4 A simple implementation scheme

In this section, we look at what might be thought of as a simple implementation of our LEGv8 subset. We build this simple implementation using the datapath of the last section and adding a simple control function. This simple implementation covers *load register* (`LDUR`), *store register* (`STUR`), *compare and branch zero* (`CBZ`), and the arithmetic-logical instructions `ADD`, `SUB`, `AND`, and `ORR`. We will later enhance the design to include an unconditional branch instruction (`B`).

The ALU control

The LEGv8 ALU in COD Appendix A (The basics of logic design) defines six following combinations of four control inputs, shown in the below animation's "ALU control lines / Function" table.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR

Depending on the instruction class, the ALU will need to perform one of these first five functions. (NOR can be used for other parts of the LEGv8 instruction set not found in the subset we are implementing.) For load register and store register instructions, we use the ALU to compute the memory address by addition. For the R-type instructions, the ALU needs to perform one of the four actions (AND, OR, subtract, or add), depending on the value of the 11-bit opcode field in the instruction (see COD Chapter 2 (Instructions: Language of the computer)). For compare and branch zero, the ALU just passes the register input value.

We can generate the 4-bit ALU control input using a small control unit that has as inputs the opcode field of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, pass input b (01) for `CBZ`, or be determined by the operation encoded in the opcode field (10). The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations shown previously.

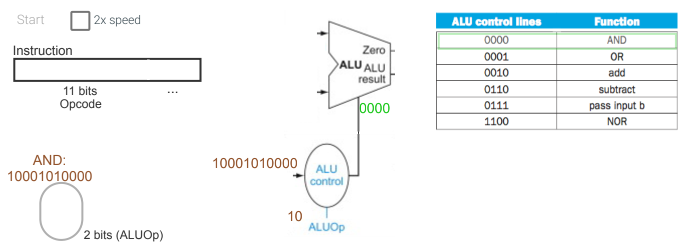
In the animation below, we show how to set the ALU control inputs based on the 2-bit ALUOp control and the 11-bit opcode. Later in this chapter, we will see how the ALUOp bits are generated from the main control unit.

The instruction, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the opcode field; in this case, we say that we "don't care" about the value of the opcode, and the bits are shown as Xs. When the ALUOp value is 10, then the opcode is used to set the ALU control input. See COD Appendix A (The basics of logic design).

PARTICIPATION ACTIVITY

4.4.1: How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction (COD Figure 4.12).

Start ☐ 2x speed



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR

Instruction	ALUOp	Instruction operation	Opcode field	Desired ALU action	ALU control input
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	ADD	10001011000	add	0010
R-type	10	SUB	11001011000	subtract	0110
R-type	10	AND	10001010000	AND	0000
R-type	10	ORR	10101010000	OR	0001

This style of using multiple levels of decoding—that is, the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit—is a common implementation technique. Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially reduce the latency of the control unit. Such optimizations are important, since the latency of the control unit is often a critical factor in determining the clock cycle time.

PARTICIPATION ACTIVITY

4.4.2: Consider the ALU control bits animation above.

1) If the instruction is STUR, then ALUOp should be _____.

- ○ 00
- ○ 01
- ○ 10
- ○ unknown

2) If the instruction is STUR, then the ALU's

four control inputs should be _____.

- 0000
- 0010
- 0110

3) For LDUR and STUR instructions, the ALU function _____.

- is the same
- differs

4) If the instruction is ORR, then ALUOp should be _____.

- 0001
- 10
- unknown

5) If the instruction is ORR, then as well as examining the ALUOp bits, the ALU control will also examine _____.

- Instruction[4:0] (the rightmost bits)
- Instruction[31:21] (the leftmost bits)

6) If the instruction is ORR, then the ALU control will (after examining the ALUOp and opcode bits) output _____.

- 10
- 0000
- 0001

There are several different ways to implement the mapping from the 2-bit ALUOp field and the 11-bit opcode to the four ALU operation control bits. Because only a small number of the 2048 possible values of the function field are of interest and the opcode field is used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and generates the appropriate ALU control signals.

As a step in designing this logic, it is useful to create a *truth table* for the interesting combinations of the opcode field and the ALUOp signals, as we've done in the figure below; this *truth table* shows how the 4-bit ALU control is set depending on these two input fields. Since the full truth table is very large (2^{13} = 8192 entries) and we don't care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value. Throughout this chapter, we will use this practice of showing only the truth table entries for outputs that must be asserted and not showing those that are all deasserted or don't care. (This practice has a disadvantage, which we discuss in COD Section C.2 (Implementing combinational control units) of Appendix C.)

Truth table: From logic, a representation of a logical operation by listing all the values of the inputs and then in each case showing what the resulting outputs should be.

Figure 4.4.1: The truth table for the 4 ALU control bits (called Operation) (COD Figure 4.13).

The inputs are the ALUOp and opcode field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. While we show all 11 bits of the opcode, note that the only bits with different values for the four R-format instructions are bits 30, 29, and 24. Thus, we only need these three opcode bits as input for ALU control instead of all 11.

ALUOp		Opcode field											
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[24]	I[23]	I[22]	I[21]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	X	0111
1	X	1	0	0	0	1	0	1	1	0	0	0	0010
1	X	1	1	0	0	1	0	1	1	0	0	0	0110
1	X	1	0	0	0	1	0	1	0	0	0	0	0000
1	X	1	0	1	0	1	0	1	0	0	0	0	0001

Because in many instances we do not care about the values of some of the inputs, and because we wish to keep the tables compact, we also include *don't-care terms*. A don't-care term in this truth table (represented by an X in an input column) indicates that the output does not depend on the value of the input corresponding to that column. For example, when the ALUOp bits are 00, as in the first row of the figure above, we always set the ALU control to 0010, independent of the opcode. In this case, then, the opcode inputs will be don't cares in this line of the truth table. Later, we will see examples of another type of don't-care term. If you are unfamiliar with the concept of don't-care terms, see COD Appendix A (The basics of logic design) for more information.

Don't-care term: An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

PARTICIPATION ACTIVITY 4.4.3: Truth table for the four ALU control inputs.

Consider the truth table above, which has six rows (Row 1, Row 2, ..., Row 6). Indicate which instruction(s) correspond to the listed rows.

SUB CBZ ORR LDUR or STUR

Row 1

Row 2

Row 4

Row 6

Reset

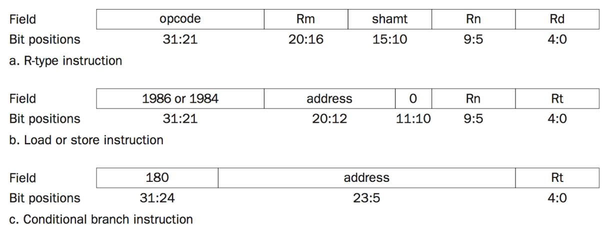
Once the truth table has been constructed, it can be optimized and then turned into gates. This process is completely mechanical. Thus, rather than show the final steps here, we describe the process and the result in COD Section C.2 (Implementing combinational control units) of Appendix C.

Designing the main control unit

Now that we have described how to design an ALU that uses the opcode and a 2-bit signal as its control inputs, we can return to looking at the rest of the control. To start this process, let's identify the fields of an instruction and the control lines that are needed for the datapath we constructed in COD Figure 4.11 (The simple datapath for the core LEGv8 architecture ...). To understand how to connect the fields of an instruction to the datapath, it is useful to review the formats of the three instruction classes: the R-type, branch, and load-store instructions. The figure below shows these formats.

Figure 4.4.2: The three instruction classes (R-type, load and store, and conditional branch) use three different instruction formats (COD Figure 4.14).

The unconditional branch instruction uses another format, which we will discuss shortly. (a) Instruction format for R-format instructions, have three register operands: Rn, Rm, and Rd. Fields Rn and Rm are sources, and Rd is the destination. The ALU function is in the opcode field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are `ADD`, `SUB`, `AND`, and `ORR`. The `shamt` field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 1986_{ten}) and store (opcode = 1984_{ten}) instructions. The register Rn is the base register that is added to the 9-bit address field to form the memory address. For loads, Rt is the destination register for the loaded value. For stores, Rt is the source register whose value should be stored into memory. (c) Instruction format for compare and branch on zero (opcode = 180). The register Rt is the source register that is tested for zero. The 19-bit address field is sign-extended, shifted, and added to the PC to compute the branch target address.



Field	opcode	Rm	shamt	Rn	Rd
Bit positions	31:21	20:16	15:10	9:5	4:0

a. R-type instruction

Field	1986 or 1984	address	0	Rn	Rt
Bit positions	31:21	20:12	11:10	9:5	4:0

b. Load or store instruction

Field	180	address	Rt
Bit positions	31:24	23:5	4:0

c. Conditional branch instruction

There are several major observations about this instruction format that we will rely on:

- The *opcode* field, which as we saw in COD Chapter 2 (Instructions: Language of the computer), is between 6 and 11 bits wide and found in bits 31:26 to 31:21.
- The first register operand is always in bit positions 9:5 (Rn) for both R-type instructions and for the base register for load and store instructions.
- The other register operand is in one of two places. It is in bit positions 20:16 (Rm) for R-type instructions and it is in bit positions 4:0 (Rt) for the register to be written by load. That is also the field that specifies the register to be tested for zero for compare and branch on zero. Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be read.
- Another operand can also be a 19-bit offset for compare and branch on zero or a 9-bit offset for load and store.
- The destination register for R-type instructions (Rd) and for loads (Rt) is in bit positions 4:0.

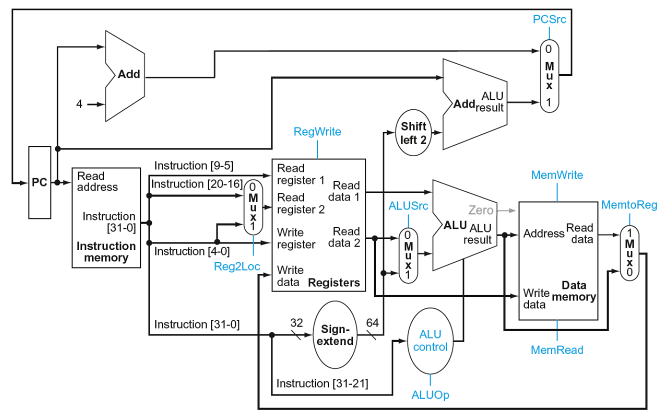
Opcode: The field that denotes the operation and format of an instruction.

The first design principle from COD Chapter 2 (Instructions: Language of the computer)—*simplicity favors regularity*—pays off here in specifying control.

Using this information, we can add the instruction labels and extra multiplexor (for the Read register 2 number input of the register file) to the simple datapath. The figure below shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a single control line.

Figure 4.4.3: The datapath of COD Figure 4.11 (The simple datapath for the core LEGv8 architecture ...) with all necessary multiplexors and all control lines identified (COD Figure 4.15).

The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.



The figure above shows seven single-bit control lines plus the 2-bit ALUOp control signal. We have already defined how the ALUOp control signal works, and it is useful to define what the seven other control signals do informally before we determine how to set these control signals during instruction execution. The figure below describes the function of these seven control lines.

Figure 4.4.4: The effect of each of the seven control signals (COD Figure 4.16).

When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See COD Appendix A (The basics of logic design) for further discussion of this problem.)

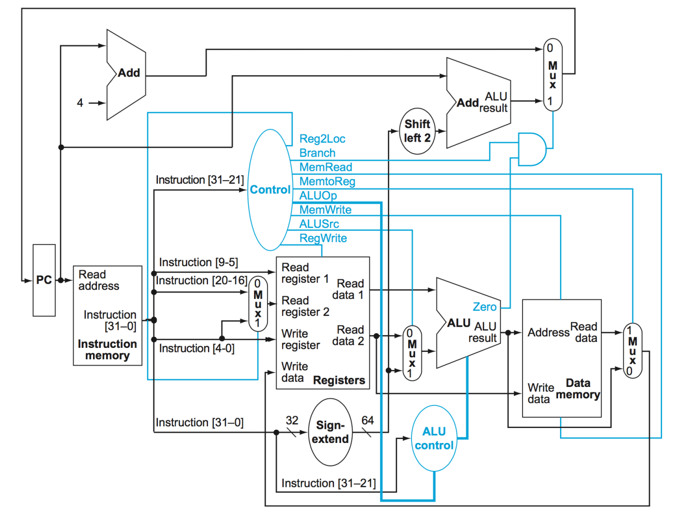
Signal name	Effect when deasserted	Effect when asserted
Reg2Loc	The register number for Read register 2 comes from the Rm field (bits 20:16).	The register number for Read register 2 comes from the Rt field (bits 4:0).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 32 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Now that we have looked at the function of each of the control signals, we can look at how to set them. The control unit can set all but one of the control signals based solely on the opcode field of the instruction. The PCSrc control line is the exception. That control line should be asserted if the instruction is compare and branch on zero (a decision that the control unit can make) *and* the Zero output of the ALU, which is used for the zero test, is asserted. To generate the PCSrc signal, we will need to AND together a signal from the control unit, which we call *Branch*, with the Zero signal out of the ALU.

These nine control signals (seven from the figure above and two for ALUOp) can now be set based on the input signals to the control unit, which are the opcode bits 31 to 21. The figure below shows the datapath with the control unit and the control signals.

Figure 4.4.5: The simple datapath with the control unit (COD Figure 4.17).

The input to the control unit is the 11-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (Reg2Loc, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.



PARTICIPATION ACTIVITY

4.4.4: The control unit's signals.

Consider the figure above showing the datapath and control unit (labeled as "Control"), and table further above listing control unit output signals.

1) The control unit sends _____ bits to the ALU control.

- 0
- 1
- 2

2) The control unit enables a write to the register file using the _____ signal.

- Reg2Loc
- MemWrite
- RegWrite

3) When MemToReg is 0, the data appearing at the register file's data input comes from the _____.

- ALU's output
- data memory's output
- register file's output

4) The ALU's top input always comes from the Read data 1 output of the register file. The ALU's bottom input can come from two possible places: The Read data 2 output of the register file, or the instruction's lower 32 bits, sign extended to 64 bits. Which control unit output select among those two places?

- ALUOp
- ALUSrc
- Zero

5) The control unit's Branch output will be 1 for a compare and branch on zero instruction. However, the branch's target address is only loaded into the PC if the ALU's Zero output is _____. Otherwise, PC is loaded with PC + 4.

- 0
- 1
- (actually, Zero is not involved)

Before we try to write a set of equations or a truth table for the control unit, it will be useful to try to define the control function informally. Because the setting of the control lines depends only on the opcode, we define whether each control signal should be 0, 1, or don't care (X) for each of the opcode values. The figure below defines how the control signals should be set for each opcode; this information follows directly from COD Figures 4.12 (How the ALU control bits are set ...), 4.16 (The effect of each of the seven control signals), and 4.17 (The simple datapath with the control unit).

Figure 4.4.6: The setting of the control lines is completely determined by the opcode fields of the instruction (COD Figure 4.18).

The first row of the table corresponds to the R-format instructions (`ADD`, `SUB`, `AND`, and `ORR`). For all these instructions, the source register fields are Rn and Rm, and the destination register field is Rd; this defines how the signals ALUSrc and Reg2Loc are set. Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the opcode field. The second and third rows of this table give the control signal settings for `LDUR` and `STUR`. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, Reg2Loc and RegWrite are set for a load to cause the result to be stored into the Rt register. The ALUOp field for branch is set for a pass input b (ALU control = 01), which is used to test for zero. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. A don't care can also be added to Reg2Loc for `LDUR`, which doesn't use a second register. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

Instruction	Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1

PARTICIPATION ACTIVITY 4.4.5: Determining control unit outputs from the opcode fields.

Consider the above figure showing control unit outputs for four kinds of instructions, using four rows (Rows 1, 2, 3, and 4).

1) In Row 1, RegWrite is 1, meaning the register is always written for an R-type instruction.

○ True

○ False

2) In Row 1, the last two bits, ALUOp, are 10, meaning the ALU will perform an add function.

○ True

○ False

3) MemWrite is 1 for Row 3 (STUR), but is 0 for Row 2. The reason is because while a store register instruction writes to the data memory, a _____ instruction does not.

○ R-type

○ load register

4) In CBZ's Row 4, MemToReg is X because the value appearing at the register file's Write data input is irrelevant.

○ True

○ False

Operation of the datapath

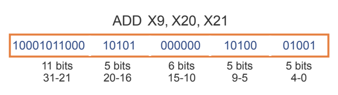
With the information contained in COD Figures 4.16 (The effect of each of the seven control signals) and 4.18 (The setting of the control lines ...), we can design the control unit logic, but before we do that, let's look at how each instruction uses the datapath. In the next few figures, we show the flow of three different instruction classes through the datapath. The asserted control signals and active datapath elements are highlighted in each of these. Note that a multiplexor whose control is 0 has a definite action, even if its control line is not highlighted. Multiple-bit control signals are highlighted if any constituent signal is asserted.

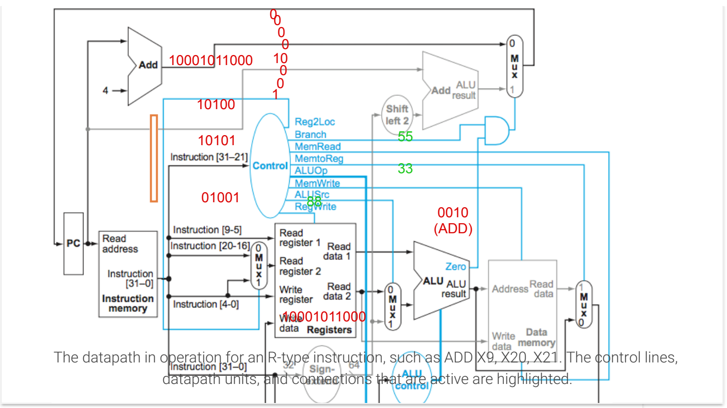
The animation below shows the operation of the datapath for an R-type instruction, such as `ADD X1, X2, X3`. Although everything occurs in one clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:

1. The instruction is fetched, and the PC is incremented.
2. Two registers, `X2` and `X3`, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using portions of the opcode to generate the ALU function.
4. The result from the ALU is written into the destination register (`X1`) in the register file.

PARTICIPATION ACTIVITY 4.4.6: The datapath in operation for an R-type instruction (COD Figure 4.19).

Start ☐ 2x speed





The datapath in operation for an R-type instruction, such as ADD X9, X20, X21. The control lines, datapath units, and connections that are active are highlighted.

Similarly, we can illustrate the execution of a load register, such as

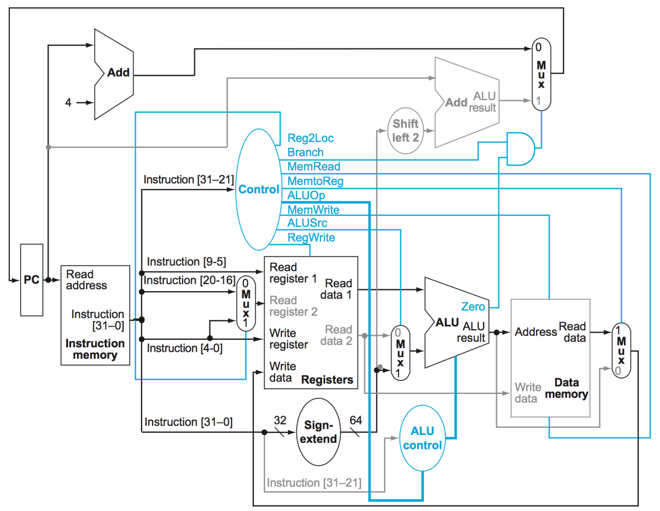
```
LDUR X1, [X2,offset]
```

in a style similar to the animation above. The figure below shows the active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps (similar to how the R-type executed in four):

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register (`X2`) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended 9 bits of the instruction (`offset`).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file (`X1`).

Figure 4.4.7: The datapath in operation for a load instruction (COD Figure 4.20).

The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.



PARTICIPATION ACTIVITY 4.4.7: The datapath for a load instruction.

Consider the above figure illustrating the datapath in action for a load instruction. Indicate the values for the listed control signals.

1) Branch
- ○ 0
- ○ 1

2) MemRead
- ○ 0
- ○ 1

3) MemToReg
- ○ 0
- ○ 1

4) ALUSrc
- ○ 0
- ○ 1

5)

RegWrite

○ 0

○ 1

6) For a store register (STUR) instruction, MemRead would be ____.

○ 0

○ 1

7) For a store register (STUR) instruction, RegWrite would be ____.

○ 0

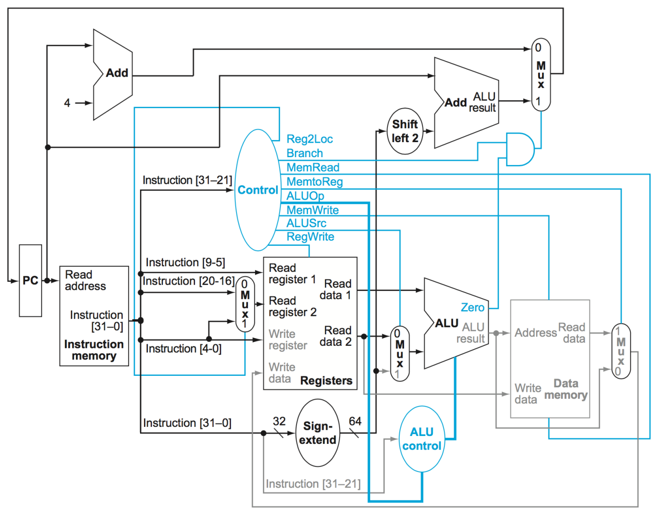
○ 1

Finally, we can show the operation of the compare-and-branch-on-zero instruction, such as `CBZ X1, offset`, in the same fashion. It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address. The figure below shows the four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. The register, `X1` is read from the register file using bits 4:0 of the instruction (Rt).
3. The ALU passes the data value read from the register file. The value of PC is added to the sign-extended, 19 bits of the instruction (`offset`) are shifted left by two; the result is the branch target address.
4. The Zero status information from the ALU is used to decide which adder result to store in the PC.

Figure 4.4.8: The datapath in operation for a compare-and-branch-on-zero instruction (COD Figure 4.21).

The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.



PARTICIPATION ACTIVITY 4.4.8: The datapath for a compare-and-branch-on-zero instruction.

Consider the above figure illustrating the datapath in action for a compare-and-branch-on-zero instruction.

1) The value for the ALUSrc control signal is ____.

○ 0

○ 1

2) The value for the ALUOp control signal is ____.

○ 00

○ 01

3) Assume the current instruction is `CBZ X1, offset`, and X1 is 85. The value for the Branch control signal is ____.

○ 0

○ 1

4) Assume the current instruction is `CBZ X1, offset`, and X1 is 85. The value for the Zero control signal is ____.

○ 0

○ 1

5) Assume the current instruction is `CBZ`

`X1, offset`, and X1 is 85. How will the PC be updated next?

- ○ PC + 4
- ○ Target address

6) The datapath shown requires four clock cycles to execute a branch instruction.

- ○ True
- ○ False

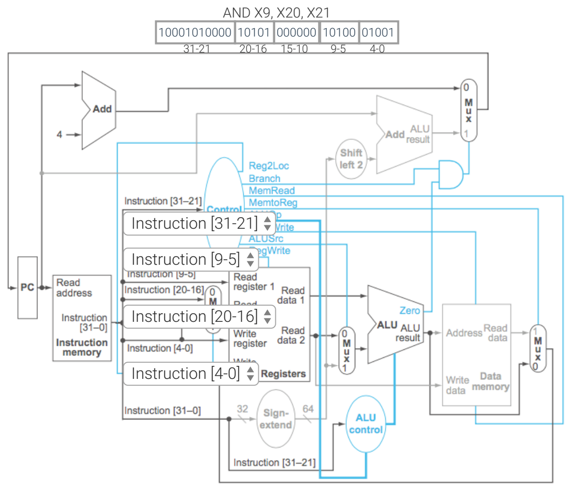
7) Consider a compare and branch on zero instruction. If the register is equal to 0, Zero becomes 1, causing a new target address to pass through the mux on the upper right and be waiting to enter the PC on the next rising clock edge.

- ○ True
- ○ False

CHALLENGE ACTIVITY 4.4.1: Datapath operation: Select bits on each wire.

Start

1
2
3
4
5



Instruction [31-21]
Instruction [9-5]
Instruction [20-16]
Instruction [4-0]

1 | 2 | 3 | 4 | 5

Check Next

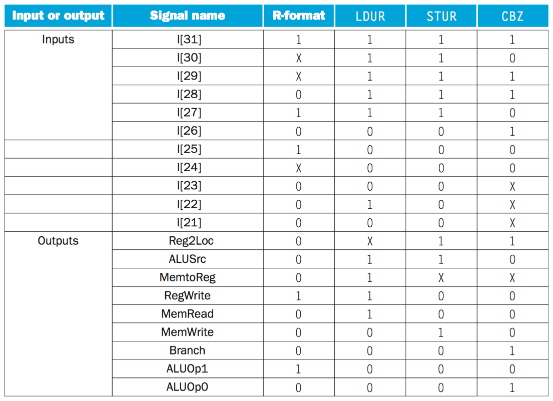
Finalizing control

Now that we have seen how the instructions operate in steps, let's continue with the control implementation. The control function can be precisely defined using the contents of COD Figure 4.18 (The setting of the control lines is completely determined by the opcode fields of the instruction). The outputs are the control lines, and the input is the opcode field. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.

The figure below defines the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion. We show this final step in COD Section C.2 (Implementing combinational control units) in Appendix C.

Figure 4.4.9: The control function for the simple single-cycle implementation is completely specified by this truth table (COD Figure 4.22).

The top half of the table gives the combinations of input signals that correspond to the four instruction classes, one per column, that determine the control output settings. The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. We simplified the truth table by using don't cares in the input portion to combine the four R-format instructions together in one column; we could have instead replaced that single column with four columns for the instructions `ADD`, `SUB`, `AND`, and `ORR`. The outputs would have been the same for all four of these R-format instructions.



Input or output	Signal name	R-format	LDUR	STUR	CBZ
Inputs	I[31]	1	1	1	1
	I[30]	X	1	1	0
	I[29]	X	1	1	1
	I[28]	0	1	1	1
	I[27]	1	1	1	0
	I[26]	0	0	0	1
	I[25]	1	0	0	0
	I[24]	X	0	0	0
	I[23]	0	0	0	X
	I[22]	0	1	0	X
	I[21]	0	0	0	X
Outputs	Reg2Loc	0	X	1	1
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Now that we have a *single-cycle implementation* of most of the LEGv8 core instruction set, let's add the unconditional branch instruction to show how the basic datapath and control can be extended to handle other instructions in the instruction set.

Single-cycle implementation: Also called ***single clock cycle implementation***. An implementation in which an instruction is executed in one clock cycle. While easy to understand, it is too slow to be practical.

Example 4.4.1: Implementing Unconditional Branches.

COD Figure 4.17 (The simple datapath with the control unit) shows the implementation of many of the instructions we looked at in COD Chapter 2 (Instructions: Language of the computer). One instruction that is missing is the unconditional branch instruction. Extend the datapath and control of COD Figure 4.17 (The simple datapath with the control unit) to include the unconditional branch instruction. Describe how to set any new control lines.

Answer

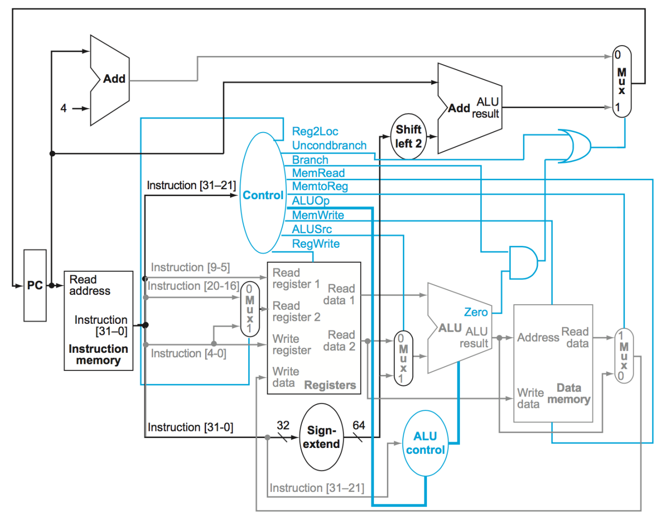
The unconditional branch instruction looks like a branch instruction with a longer offset but is not conditional. Like a branch, the lower-order 2 bits of a branch address are always 00_{two}. The next 26 bits of this 64-bit address come from the 26-bit immediate field in the instruction and then are sign extended. Thus, we can implement a branch by storing into the PC sum of the PC and the sign extended and shifted 26-bit offset. COD Figure 4.23 (The simple control and datapath are extended ...) shows the addition of the control for branch added to COD Figure 4.17 (The simple datapath with the control unit). An additional OR-gate is used with a control signal to select the branch target PC always. This control signal, called *UncondBranch*, is asserted only when the instruction is an unconditional branch.

Elaboration

We also need to modify the sign-extended unit to include the 26-bit address of this instruction. We solve the problem by expanding the 2:1 multiplexor that was controlled by opcode bit 26 mentioned in an earlier elaboration to include this address and then control it with the two opcode bits 31 and 26. The value 01 means select the address for B, 10 means address for ***`LDUR`*** *or* ***`STUR`****, and 11* ***`CBZ`****.*

Figure 4.4.10: The simple control and datapath are extended to handle the unconditional branch instruction (COD Figure 4.23).

An additional OR-gate (at the upper right) is used to control the multiplexer that chooses between the branch target and the sequential instruction following this one. One input to the OR-gate is the Uncondbranch control signal. Although not shown, the Sign-extend logic would recognize the unconditional branch opcode and sign-extend the lower 26 bits of the branch instruction to form a 64-bit address to be added to the PC.



PARTICIPATION ACTIVITY 4.4.9: The datapath for an unconditional branch instruction.

Consider the above figure illustrating the datapath extended for an unconditional branch instruction.

1) The ALU is used for both compare and branch on zero and unconditional branch instructions.
   - True
   - False

2) For an unconditional branch instruction, the datapath forms the destination address by appending 00 to Instruction[25-0], and also _____ the sum of the PC and the sign extended and shifted 26-bit offset.
   - prepending
   - appending

3) For an unconditional branch instruction, the control unit sets UncondBranch to 1. The control unit must also set Branch to 0.
   - True
   - False

Why a single-cycle implementation is not used today

Although the single-cycle design will work correctly, it is too inefficient to be used in modern designs. To see why this is so, notice that the clock cycle must have the same length for every instruction in this single-cycle design. Of course, the longest possible path in the processor determines the clock cycle. This path is most likely a load instruction, which uses five functional units in series: the instruction memory, the register file, the ALU, the data memory, and the register file. Although the CPI is 1 (see COD Chapter 1 (Computer abstractions and technology)), the overall performance of a single-cycle implementation is likely to be poor, since the clock cycle is too long.

The penalty for using the single-cycle design with a fixed clock cycle is significant, but might be considered acceptable for this small instruction set. Historically, early computers with very simple instruction sets did use this implementation technique. However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design wouldn't work well at all.

Because we must assume that the clock cycle is equal to the worst-case delay for all instructions, it's useless to try implementation techniques that reduce the delay of the common case but do not improve the worst-case cycle time. A single-cycle implementation thus violates the great idea from COD Chapter 1 (Computer abstractions and technology) of making the **common case fast**.

In the next section, we'll look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath but is much more efficient by having a much higher throughput. Pipelining improves efficiency by executing multiple instructions simultaneously.

PARTICIPATION ACTIVITY 4.4.10: Single cycle datapath.

1) A single-cycle implementation is uncommon today because a single-cycle implementation _____.
   - often yields incorrect values
   - is harder to design than a multiple-cycle implementation
   - is slower than a multi-cycle

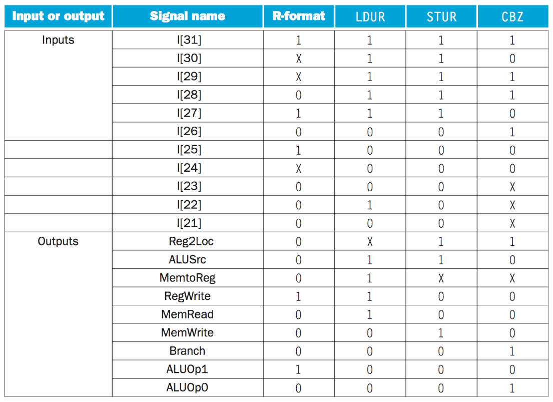
design

2) Suppose all instructions could potentially execute with a 1 ns clock cycle, except a load instruction requiring 2 ns. Assuming each instruction runs one at a time, how long would 1 load instruction plus 39 other instructions take to execute in a single-cycle implementation using a 2 ns clock cycle?

- ○ 40 ns
- ○ 41 ns
- ○ 80 ns

PARTICIPATION ACTIVITY 4.4.11: Check yourself: Control functions.

Refer to the control function truth table below.



Input or output	Signal name	R-format	LDUR	STUR	CBZ
Inputs	I[31]	1	1	1	1
	I[30]	X	1	1	0
	I[29]	X	1	1	1
	I[28]	0	1	1	1
	I[27]	1	1	1	0
	I[26]	0	0	0	1
	I[25]	1	0	0	0
	I[24]	X	0	0	0
	I[23]	0	0	0	X
	I[22]	0	1	0	X
	I[21]	0	0	0	X
Outputs	Reg2Loc	0	X	1	1
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

1) Can any control signal be combined together?

- ○ Yes
- ○ No

2) Can any control signal output be replaced by the inverse of another?

- ○ Yes
- ○ No

Provide feedback on this section