EE 361
16-Bit Single-Cycle LEGLite Project

Summary: You will implement, in verilog, a simpified version of the **single-cycle LEGv8**, which we will refer to as LEGLite.

LEGLite Description:

- Its data and address buses are 16-bits wide rather than 32-bits.
- All instructions and operands are 16-bits, i.e., "halfwords".
- Memory is byte addressable and is organized as Little Endian. Note that memory addresses of halfwords are divisible by two.

General Purpose Registers: X0, X1, ..., X7

Name	**Reg #**	**Usage**
X0-X6	0-6	General purpose registers
XZR	7	Zero register (always zero valued)

Instruction Formats

Name	Fields					Example
Format	4 bits	3 bits	3 bits	3 bits	3 bits	
R	op	Xm	NA	Xn	Xd	ADD Xd,Xn,Xm
I	op	constant		Xn	Xd	ADDI Xd,Xn,#constant
D	op	constant		Xn	Xd	STUR Xd,[Xn,#constant]
CB	op	constant		Xn	NA	CBZ Xn,#Label

Note: Instruction formats I, D, and CB are all the same, so we could just call them a single format. But we refer to them as different formats to be consistent with LEGv8.

Machine Instructions: In the table LD and ST are load and store, respectively, a 16-bit halfword from/to memory.

Name	**Format**	**Example**					**Comments**
		Bits 15-12	**Bits 11-9**	**Bits 8-6**	**Bits 5-3**	**Bits 2-0**	
ADD	**R**	**0**	**3**		**2**	**1**	**ADD X1,X2,X3**
SUB	**R**	**1**	**3**		**2**	**1**	**SUB X1,X2,X3**
B	**CB**	**4**	**(PC-relative offset)/2**				**B Label**
LD	**D**	**5**	**30**		**2**	**1**	**LD X1,[X2,#30]**
ST	**D**	**6**	**30**		**2**	**1**	**ST X1,[X2,#30]**
CBZ	**CB**	**7**	**(PC-relative offset)/2**		**2**		**CBZ X2,Label**
ADDI	**I**	**8**	**30**		**2**	**1**	**ADDI X1,X2,#30**
ANDI	**I**	**9**	**30**		**2**	**1**	**AND X1,X2,#30**
SUBI	**I**	**10**	**30**		**2**	**1**	**SUB X1,X2,#30**

Figure 1 has the circuit diagram for the single cycle LEGLite.   Not shown is the clock inputs and the reset input, where the reset input will clear the program counter (PC) to 0.

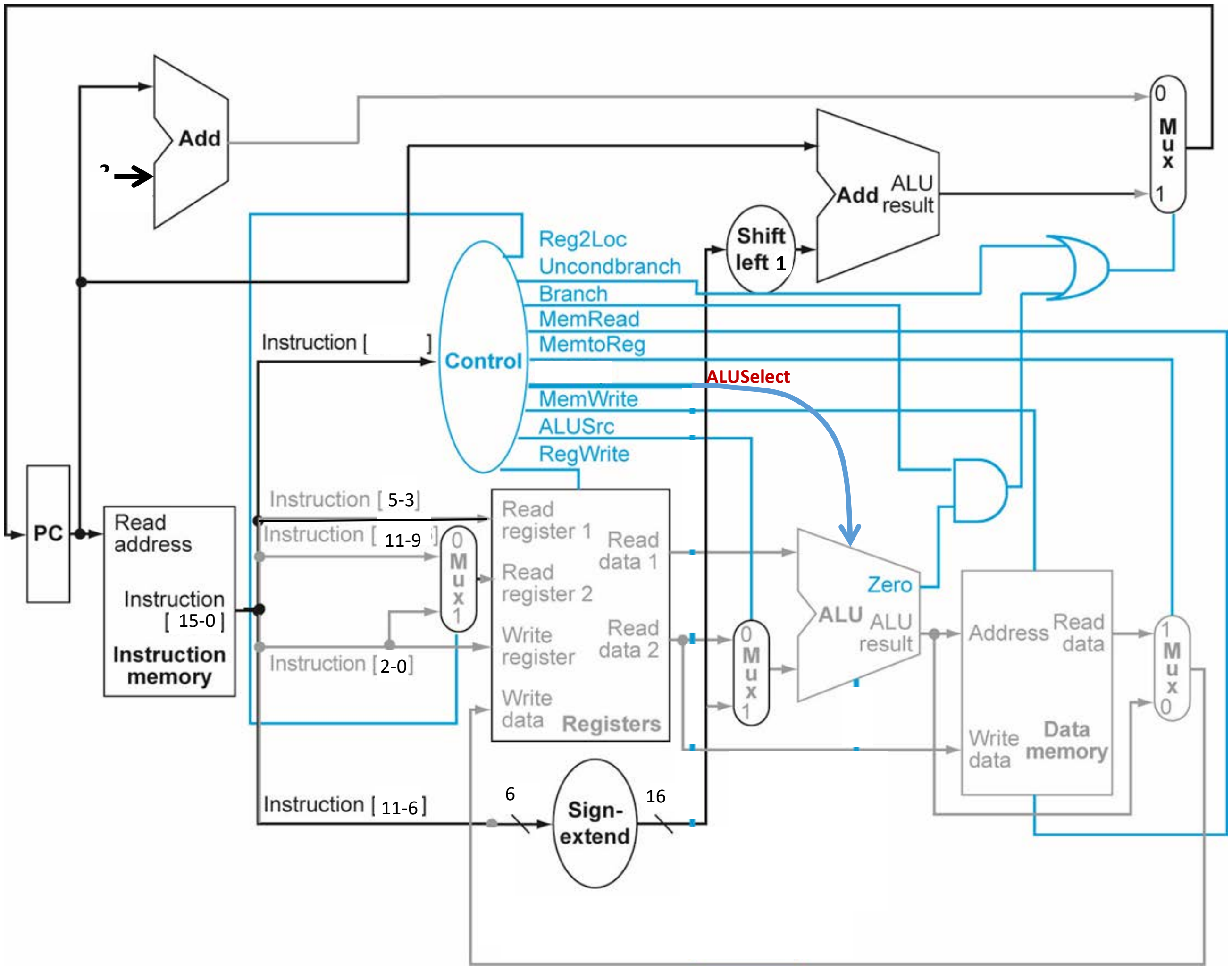


Figure 1. Single cycle LEGLite block diagram.

To simplify the block diagram in Figure 1, we can organize the program counter (PC) and all the circuitry that controls it into a single block, shown in Figure 2. We refer to this block as PC Logic. Note that the PC Logic is a synchronous sequential circuit. It has the output PC, which is the PC value. It has as inputs:

- clock: the clock input
- reset: asserting this will reset the PC in the next clock transition
- signext: this is the offset that is added to the PC to determine where to branch to, i.e., this can be used to compute the target branch address
- uncondbranch: asserting this will cause the computer to branch.
- branch: asserting this will cause the computer to branch if alu_zero equals 1. This implement CBZ.

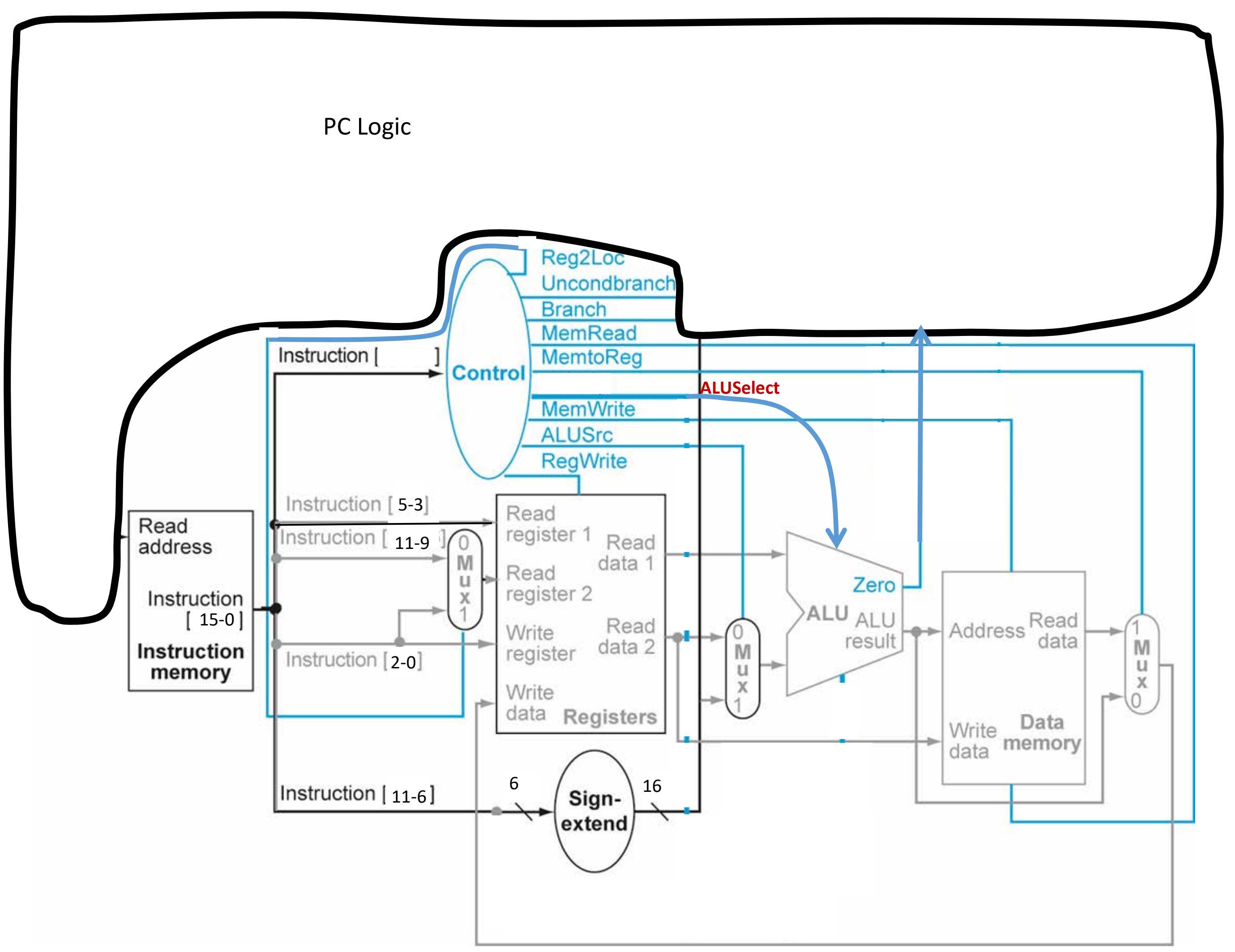


Figure 3 more clearly shows the inputs and output of the PC Logic block (not shown is the clock input).

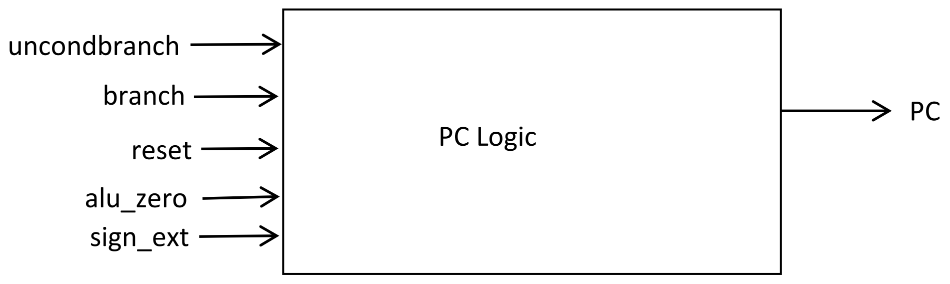


Figure 4 shows a block diagram of the computer. There is the CPU, instruction memory, and data memory, which includes I/O. Note that the CPU is basically all the components in Figure 1 except the instruction and data memories. The following is a description of the instruction and data memories.

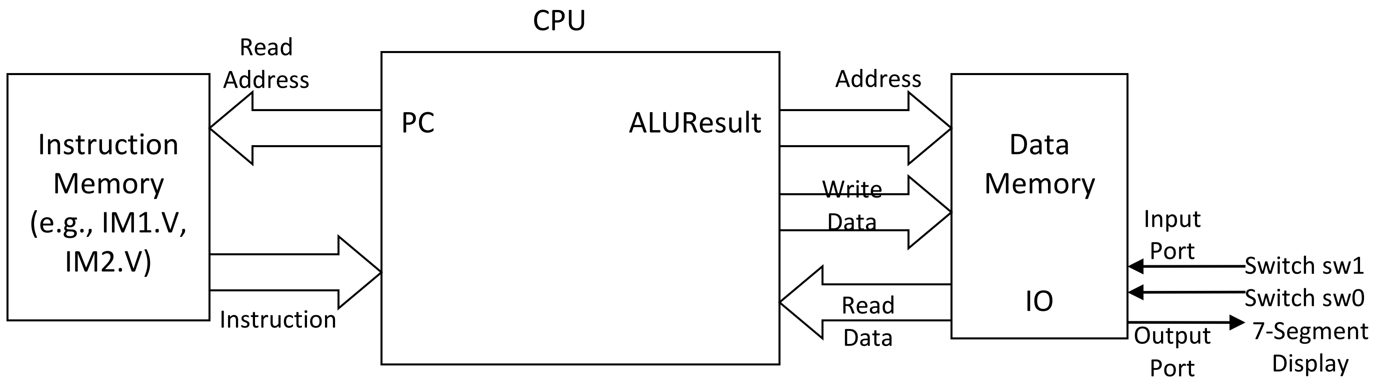


Figure 4. Block diagram of the computer, clock not shown.

Instruction Memory (IM1.V and IM2.V):
The instruction memory will be given to you, and they are verilog modules IM1.v or IM2.v. Each of these modules will have a program that starts at address 0.

Data Memory:
Data memory has 128 memory cells, each with 16-bits. The addresses of the memory cells are 0, 2, 4, ..., 254.

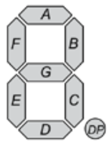
The data memory also has an I/O subcomponent which is connected to two switches and a 7-segment display. The two switches are labeled in Figure 4 as sw0 and sw1. These are inputs to the computer. The 7-segment display is connected to the output of the computer.

The address of the switches is `0xfff0`. The switches are connected to the least significant bits at the address. In particular, sw0 is connected to bit 0, and sw1 is connected to bit 1. The following is an example loop that will keep looping until switch sw1 equals 1.

```
Loop:   LD      X0, [ XZR, #-16]     // Note that  0xfff0  = -16 in twos complement
        ANDI   X0, X0, #2
        CBZ     X0, Loop
```

Note that to input the values of these switches, we load the value in memory at address `0xfff0.` In the loop-example above, the value is loaded into register X0. Then we can mask bits to determine whether a switch has value 0 or 1.

The output to the 7-segment display is at address `0xfffa` (which is twos complement for -6):



There is a 7-bit register at this address. Let (b7, b6, b5, b4, b3, b2, b1, b0) be the bits of this register. Then

b7 is connected to A,
b6 is connected to B,
...,
b0 is connected to G.

Thus, if the bits are 0110000 then the display will be "1", if the bits are 1111110 then the display will be "0", and if the bits are 1011011 then the display will be "5".

As an example, the following will display "0" on the 7-segment display:

```
LD      X0, [XZR, -6]    // X0 points to the output to the 7-segment display
ADDI  X1, XZR, #-2     //  -2 in twos complement is 111...1110
ST      X1, [X0, #0]
```

The input at address `0xfff0` is known as an *input port*, while the output at address `0xfffa` is known as an *output port*.

The following is the portion of the module that controls reading data from the unit.

```
always_comb
     begin
     if (read == 0) rdata = 0;
     else // read = 1
          begin
          if (addr >= 0 && addr < 256)     rdata = memcell[addr[7:1]];
          else if (addr == 16'hfff0)       rdata = {14'd0,io_sw1,io_sw0};
          else rdata = 0; // default
          end
     end
```

The following is the portion of the module that controls writing data to the unit.

```
always_ff @(posedge clock)
     if (write == 1 && addr == 16'hfffa)
        io_display <= wdata[6:0];

always_ff @(posedge clock)
     if (write == 1 && addr >= 0 && addr < 256) memcell[addr[7:1]] <= wdata;
```

Assignment:
There are three stages to this homework. Each stage takes about a week. DON'T FALL BEHIND.

- Stage 1: This stage is to test components of the computer. There's nothing to submit. The files for this stage are in the folder HwLEGLite-Stage1. This stage is straight forward but it will take time. Do this right now.
- Stage 2: You will implement a computer so that it runs a simple program, which multiplies two integers by using a loop. The computer should be able to run the instructions: ADD, ADDI, and CBZ. The files for this stage are in the folder HwLEGLite-Stage2

- Stage 3: This stage improves the computer from stage 2. In particular, it can also run LD, ST, and ANDI. The program that it will run will access the IO ports. The files for this stage are in the folder HwLEGLite-Stage3

For each of these stages are verilog files for component and testbenches. Read the testbenches to understand them. They have comments to guide you.

Grading: The Homework is worth 30 points according to the following

Complete	**Points**
Partially working Stage 2	15
Completely working Stage 2	24
Completely working Stage 3	30

Submission Instructions:

Submit a single Windows folder (zipped), which should have

1. All your verilog files including the testbench
2. Also include IM2.V if you have Stage 3 completely working, or IM1.V if you don't have Stage 3 working but you have Stage 2 partially or completely working.
3. Project file that will run the testbench
4. README file which includes
   a. Your name
   b. Description of what you have completed (and not completed). For example
      i. "I have completed Stage 3. I have included IM2.V"
      ii. "I have completed Stage 2, I didn't get Stage 3 to work completely. I have included IM1.V"
      iii. "I didn't get Stage 2 to work completely but I have it partially working. I have included IM1.V"
      iv. "I didn't get anything to work in Stage 2"

      If you don't have this description, you won't get a grade.
   c. Description of what you have in this folder including a list of all the files including all verilog and project files.

Zip the folder and submit it through laulima as an attachment on the due date indicated in the assignment.

Description of the Stages:

Stage 1: Here are the files in HwLEGLite-Stage1 folder (note some files may need some editing)

- Parts.V: This has
  - Register file
  - ALU
  - 16-bit 2:1 multiplexer
  - 16-bit 4:1 multiplexer
  - Data Memory / IO Component
    - The data memory has 128 memory cells that are 16-bits
    - Memory starts from address 0.
    - Memory is byte addressable, each 16-bit word has an address that is divisible by 2
    - The word addresses are 0, 2, 4, ...
    - There are two IO ports
      - Output port to a 7-segment display at address 0xfffa.

        - Input port from two switches at address 0xfff0.
    - There are three testbenches for this file
        - testbench-Parts-CombCirc.V: Tests the combinational circuits.
        - testbench-Parts-Rfile.V: Tests the register file
        - testbench-Parts-Dmem.V: Tests the data memory and I/O

Read all the code and test all the parts. The code may be buggy. You can make any corrections. Read the code and comments, so you understand them completely.

There is nothing to turn in for this stage.

The next two stages are to build a working processor that can run some of the instructions.

Stage 2: In the HwLEGLite-Stage 2 folder there are the following files (some files may need editing, i.e., buggy)

- LEGLiteSingle.V: This has the LEGLite processer module. It's incomplete (actually it's basically empty) and you must complete it.
    - It must implement the ADD, ADDI, and CBZ instructions. You can ignore the other instructions for now.
- testbench-LEGLiteSingle-Stage2.V: This is the test bench for LEGLiteSingle.V. It has instantiations of the
    - LEGLiteSingle module
    - Data memory (with IO)
    - Instruction memory IM1.V. This is a program that multiplies 3 by 5 using a loop. It uses the ADD, ADDI, and CBZ, instructions. The following is the program

```
L0:     ADDI  X2,XZR,#3         # X2 = 3, X2 is used as a counter
        ADD   X4,XZR,XZR        # Clear X4, which will contain the final product
L1:     CBZ   X2,L0       # If counter = 0 then we've completed the multiply and we can start all over
        ADDI  X4,X4,#5  # Increment X4 by 5
        ADDI  X2,X2,#-1 # Decrement counter
        CBZ   XZR,L1      # Continue looping
```

    - Note: We can check if the LEGLite is working correctly by viewing the ALU-result output. For example, if the instruction is ADDI X4,XZR,#3 then the ALU output = 3. Then if the next instruction is ADDI X4,X4,#3, the ALU output is 6.

      Note that the ALU output is the write data output of the LEGLite which goes to data memory. So by viewing the write data output from the LEGLite we will view the ALU output.

- To create a project you need the following files:
    - IM1.V
    - LEGLiteSingle.V
    - LEGLite-Control.V (described below)
    - LEGLite-PC.V (described below)
    - Parts.V
    - testbench-LEGLiteSingle-Stage2.V.

- LEGLite-Control.V: This has the Controller module of LEGLite. This will be instantiated in the LEGLite module.
    - LEGLite-Control.V: This has an incomplete Controller module. Shown below is the function tables for the ALU for your reference. This is straight forward if you understand how the LEGLite datapath works.

- There's a sample testbench testbenchControl (it may be buggy). You can make your own testbench too.

ALU function table

alu select	Operation
0	add
1	subtract
2	pass input 1 to output ALUresult
3	or
4	and

- LEGLite-PC.V: This has the PC control logic described earlier (see Figures 3 and 4). This is a component of MIPS-L and will be instantiated in the LEGLite module.
    - LEGLite-PC.V: has an incomplete PCLogic module. Currently, it only increments PC by 2 and resets to 0. You should also have it operate for conditional branch CBZ. This is straight forward (really).
    - testbench-PC.V: this is a test bench for PCLogic module, which is the module that does PC Control.

Suggestion 1: LEGLite may be difficult to debug, especially since the circuit is not trivial and you are still learning about verilog and the Xilinx Webpack tools. In addition, the LEGLite is a circuit that forms a "loop" of circuitry. Then a bug can circulate throughout the circuit, making it difficult to locate its origins.

It's helpful to build the LEGLiteSingle in stages. For example, you can start with testing PC Logic. Next, you can add the the Instruction memory. Then set "default" inputs to specific values (e.g., zero), and subsequently run this incomplete system to check if it works. Then you can add other components such as the sign extender, register file, and ALU.

Stage 3: Implement LEGLiteSingle so that it can run LD and SW instructions as well as the instructions of Stage 2. It should run Instruction Memory IM2.V. The files are in the folder LEGLite-Stage3. It also includes the testbench.

The IM2.V program interacts with the I/O ports:
- Input switch 0 at address `0xfff0`
- Output port connected to the seven segment display at address 0xfffa

It will continually check switch 0. If the switch = 0 then it outputs "0" to the seven segment display. If the switch = 1 then it outputs "2" to the seven segment display.

Description of IM2.V

The program is an infinite loop. It first does the following

- Set X3 to 0xfff0. This is a reference point to the I/O ports. For example, note that
    - LD X4,[X3,#0] will load a value from the input switches into X4
    - ST X4,[X3,#0] will store the value in X4 into the 7 segment display

Next it checks the input switch sw0. If it's 0 then the computer will display "0" on the 7 segment display; otherwise, it displays "2".

```
#  Initialization Phase
L0:     ADDI    X3,XZR,#0xfff0          # X3=0xfff0, the addr to sw0/display
        LD      X5,[X3,#0]              # X5 = input switch value
        ANDI    X5,X5,#1                # Mask all bits except bit 0 (sw0)
        CBZ     X5,Disp0                # if bit = 0 then X4 = pattern “0” else X4 = pattern “1”
        ADDI    X4,XZR,1101101          # X4 = bit pattern “2”
        CBZ     XZR,Skip
Disp0:  ADDI    X4,XZR,1111110          # X4 = bit pattern “0”
Skip:
        ST      X4,[X3,#10]             # 7-segment display = bit pattern
        CBZ     XZR,L0                  # Repeat loop
```