Digital Electronics & Computer Engineering (E85)

Harris Spring 2023

Final Exam

This is a closed-book take-home exam. Electronic devices including calculators are not allowed, except on the computer question on the last page. You are permitted two 8.5x11" sheets of paper with notes. Reference materials are provided for you on the first pages of the exam, including the RISC-V instruction set, single-cycle, multicycle, and pipelined processor diagrams (datapath, controller, FSM, truth tables), and Cyclone logic block diagram.

You are bound by the HMC Honor Code while taking this exam.

The first part of the exam is written, while the final page is done on the computer based on your E85 Lab 10. The entire Lab 10 that you use (including datapath and controller) must be your own work; it cannot, for example, include somebody else's controller. The exam is intended to be doable in 3 hours if you have prepared adequately. However, there will be no limit on the time you are allowed except that the written portion must be completed in one contiguous block of time and the computer part must be completed in another contiguous block of time. A contiguous block of time is a period of time working at a desk without breaking for meals, naps, socializing, etc. You cannot study for E85 or consult E85 resources between the two blocks of time. Please manage your time wisely and do not let the exam expand to take more time than is justified.

Return the exam under Prof. Harris' door no later than Friday 5/12 at noon (senior exams are due by May 5 at 5 pm).

Alongside each question, the number of points is written in brackets. All work and answers should be written directly on this examination booklet, except for printouts. Use the backs of pages if necessary. Write neatly; illegible answers will be marked wrong. Show your work for partial credit.

Name: _____

Do Not Write Below This Point

Page 6:	_____	/ 4
Page 7:	_____	/ 9
Page 8:	_____	/ 6
Page 9:	_____	/ 8
Page 10:	_____	/ 4
Page 11:	_____	/ 7
Page 9:	_____	/ 7
Total:	_____	/ 45

RISC-V Instruction Set Summary

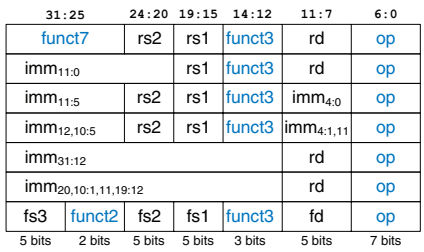


Figure B.1 RISC-V 32-bit instruction formats

- `imm`: signed immediate in $\mathbf{imm}_{11:0}$
- `uimm`: 5-bit unsigned immediate in $\mathbf{imm}_{4:0}$
- `upimm`: 20 upper bits of a 32-bit immediate, in $\mathbf{imm}_{31:12}$
- `Address`: memory address: **rs1** + SignExt($\mathbf{imm}_{11:0}$)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({$\mathbf{imm}_{12:1}$, 1'b0})
- `JTA`: jump target address: PC + SignExt({$\mathbf{imm}_{20:1}$, 1'b0})
- `label`: text indicating instruction address
- `SignExt`: value sign-extended to 32 bits
- `ZeroExt`: value zero-extended to 32 bits
- `csr`: control and status register

Table B.1 RV32I: RISC-V integer instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	`rd = [Address]31:0`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]15:0)`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	`[Address]7:0 = rs27:0`
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	`[Address]15:0 = rs215:0`
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	`[Address]31:0 = rs2`
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	`rd = rs1 – rs2`
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	`rd = rs1 << rs24:0`
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	`rd = (rs1 < rs2)`
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	`rd = (rs1 < rs2)`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	`rd = rs1 >> rs24:0`
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>> rs24:0`
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	`rd = rs1 & rs2`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

*Encoded in $\text{instr}_{31:25}$, the upper seven bits of the immediate field

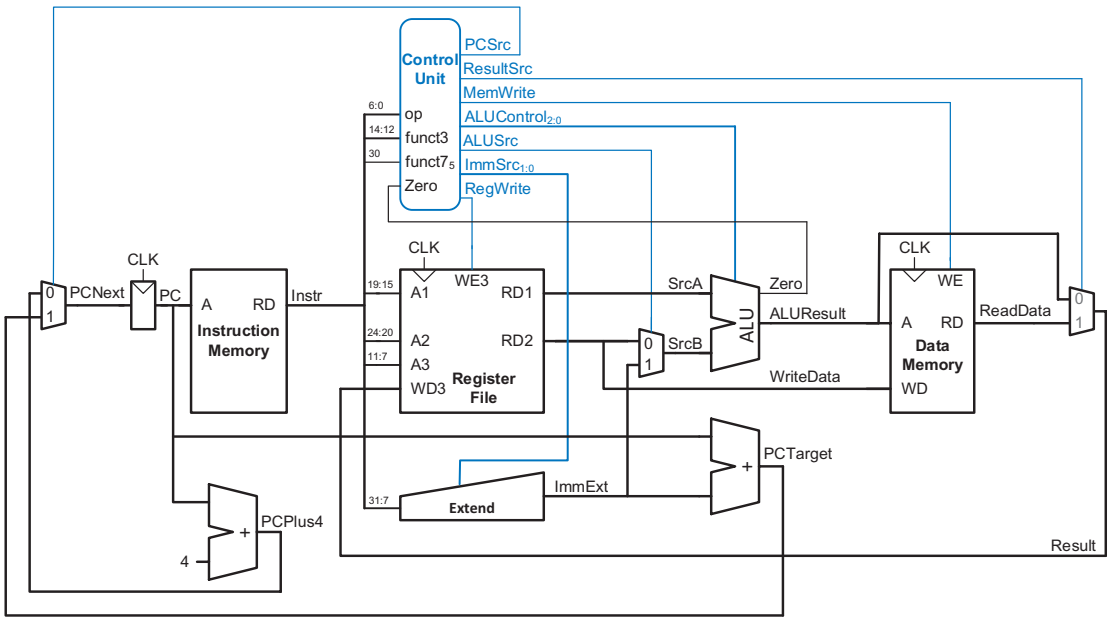
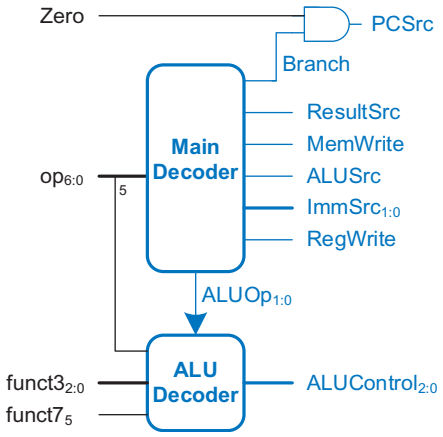


Figure 7.12 Complete single-cycle processor



ALUOp	funct3	{op₅, funct7₅}	ALUControl	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt`
	110	x	011 (or)	`or`
	111	x	010 (and)	`and`

Main Decoder

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
`lw`	0000011	1	00	1	0	1	0	00
`sw`	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
`beq`	1100011	0	10	0	0	x	1	01
`addi`	**0010011**	**1**	**00**	**1**	**0**	**0**	**0**	**10**

Table 7.1 ***ImmSrc*** **encoding**

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate

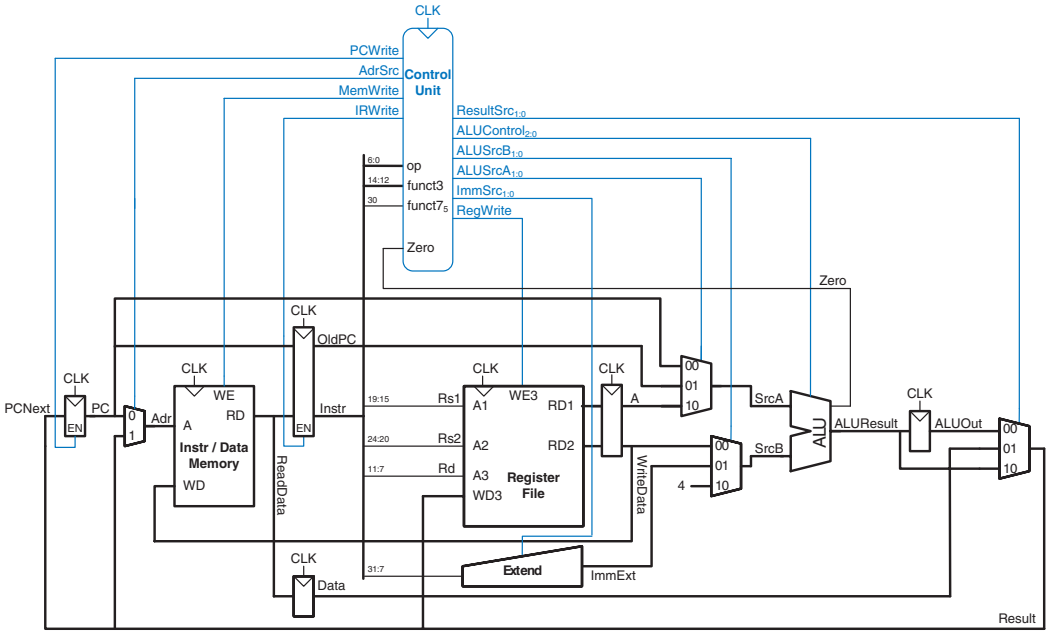
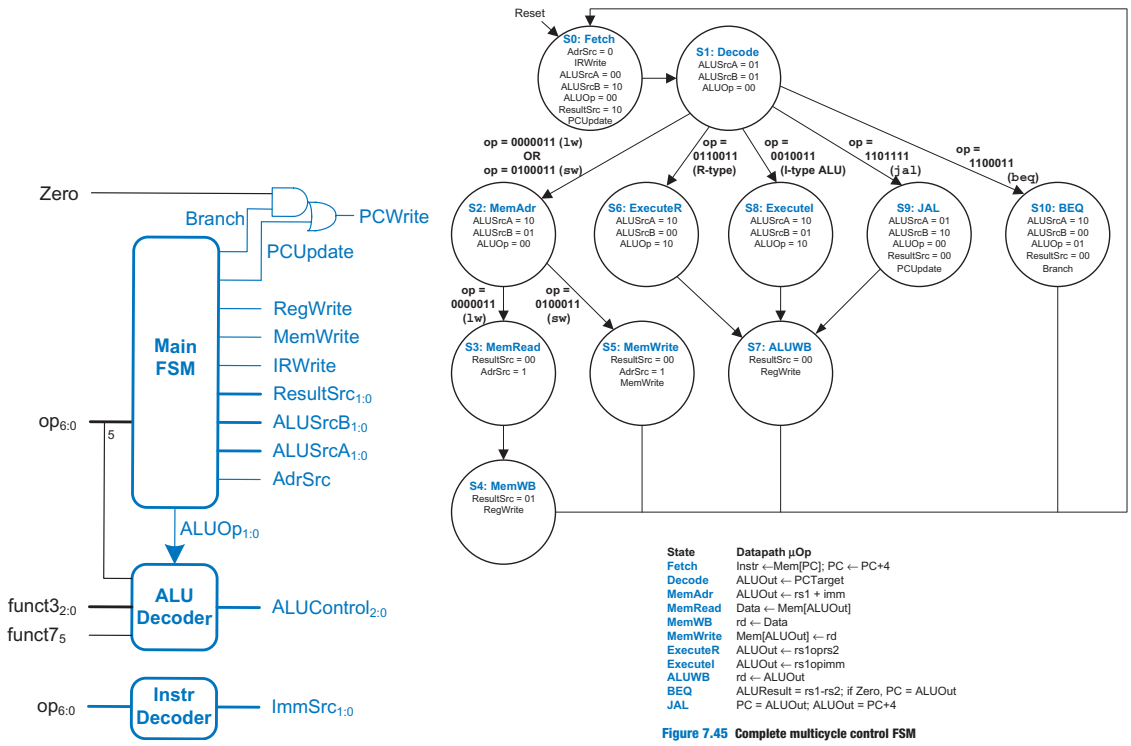
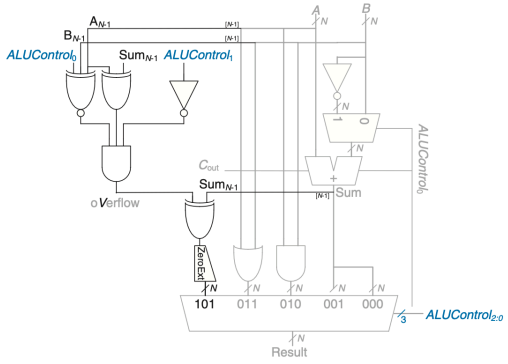


Figure 7.27 Complete multicycle processor



State	Datapath µOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1opsr2
ExecuteI	ALUOut ← rs1opimm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4

Figure 7.45 Complete multicycle control FSM



ALU

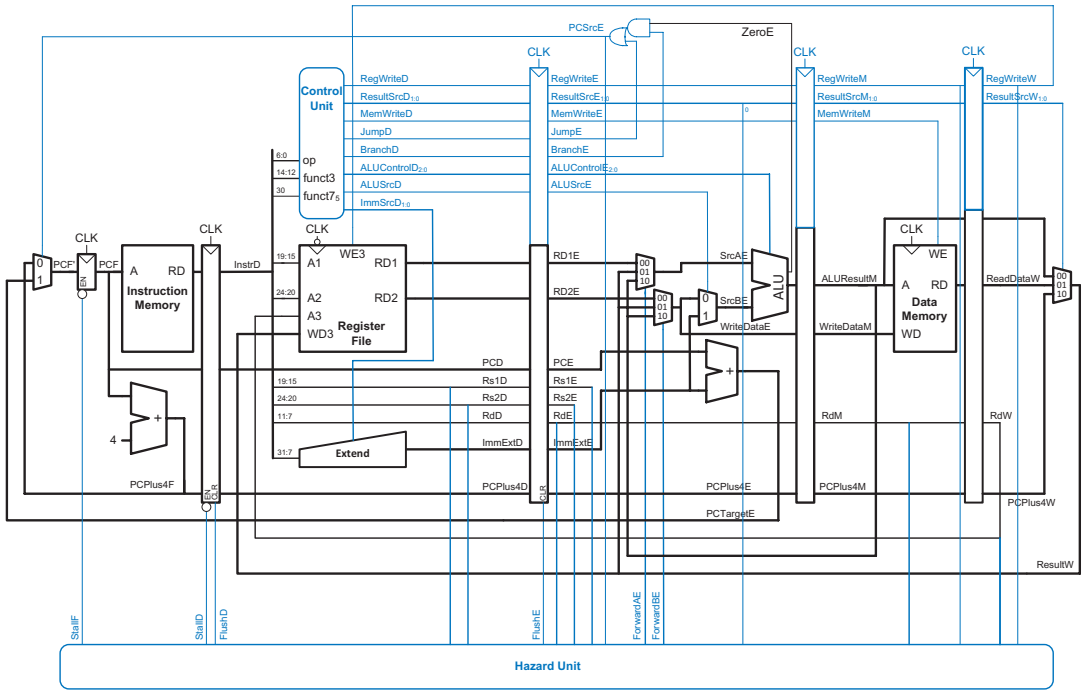


Figure 7.61 Pipelined processor with full hazard handling

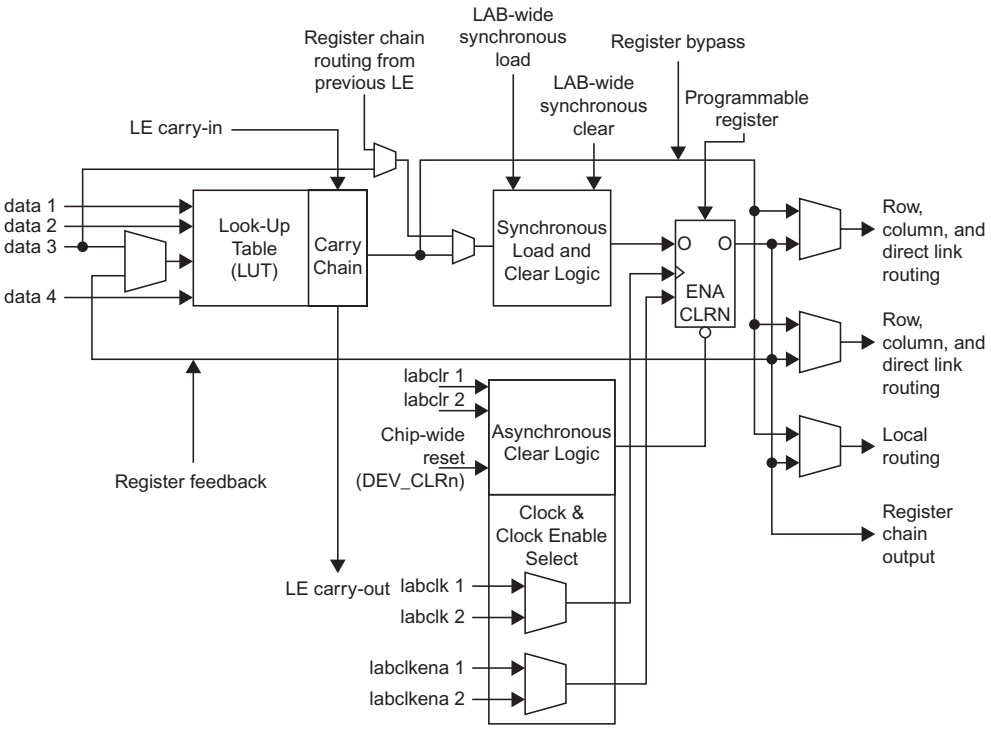


Figure 5.60 Cyclone IV Logic Element (LE)

(Reproduced with permission from the Altera Cyclone™ IV Handbook © 2010 Altera Corporation)

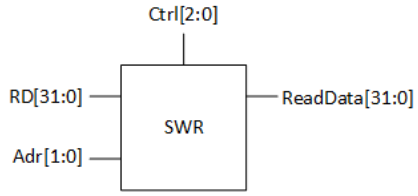
[4] Compute 0x3FA00000 * 0xC0600000, treating both numbers as IEEE single-precision floating point. Express your result in hexadecimal.

Hexadecimal Result: _____

1.25 x -3.5 = -35/8 = -4.375 = C08C0000

The critical path some implementaions of the pipelined processor is through the branch logic. Your boss asks you to speed up the branch logic by designing a dedicated branch comparator for bltu (branch less than unsigned) comparaisons. The comparator takes two 32-bit inputs SrcA and SrcB. It produces an output LTU if SrcA < SrcB when interpreted as unsigned inputs.

A Sub-Word Read (SWR) block extracts a byte or half-word from a word and possibly sign-extends it to word length. The block interface is shown in the diagram below and the function is described by the table. Byte indicates one of the four bytes within the word, specified by Adr[1:0] with byte 00 being Word[7:0] and byte 11 being Word[31:24]. Similarly, Halfword is one of the two 16-bit halfwords within the word, specified by Adr[1].



Ctrl	Operation	ReadData
000	lb	SignExt(Byte)
001	lh	SignExt(Halfword)
010	lw	RD[31:0]
100	lbu	ZeroExt(Byte)
101	lhu	ZeroExt(Halfword)

[4] Complete the following table to show the ReadData output in hexadecimal, assuming RD[31:0] = 0xAB0042CD. The first row is done for you.

Ctrl	Adr[1:0]	ReadData
000	11	FFFFFFAB
001	00	
010	00	
100	00	
101	10	

[1] Is SWR combinational or sequential? Combinational Sequential

[4] Sketch a circuit implementing the SWR function. You may use standard building blocks including multiplexers, sign extenders, and zero extenders. Label each bus with the bit width.

What is the smallest number of Cyclone IV logic elements that each of the following modules require?

```
module adventuregame(input  logic clk, reset,
                     input  logic up, down, fly,
                     output logic win);

  logic [1:0] level, nextlevel;

  always_ff @(posedge clk, posedge reset)
     if (reset) level <= 2'b00;
     else       level <= nextlevel;

  always_comb
    case (level)
      2'b00: if      (up)   level = 2'b01;
             else           level = 2'b00;
      2'b01: if      (up)   level = 2'b10;
             else if (down) level = 2'b00;
             else           level = 2'b01;
      2'b10: if      (up)   level = 2'b11;
             else if (down) level = 2'b01;
             else           level = 2'b10;
      2'b11: if      (down) level = 2'b10;
             else           level = 2'b11;
    endcase

  assign win = (level == 2'b11) & fly;

endmodule
```

[2] LEs: _____

```
module lt2(input  logic [1:0] a, b,
           output logic       lt);

  assign lt = a < b;
endmodule
```

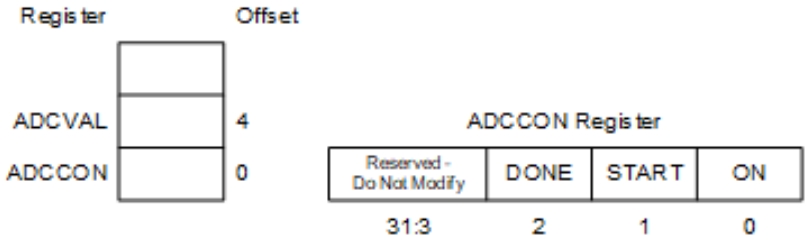
[2] LEs: _____

```
module alu(input  logic [31:0] a, b,
           input  logic [1:0]  op,
           output logic [31:0] y);

  always_comb
    case (op)
      2'b00: y = a + b;
      2'b01: y = a & b;
      2'b10: y = a | b;
      2'b11: y = a ^ b;
    endcase
endmodule
```

[2] LEs: _____

Consider adding an analog-to-digital converter to the FE310 chip on the Red-V board so that you can record audio. The register map is given below, with a base address of 0x10019000. The bitfields within the ADCCON register are also shown. Before using the converter, initialize it by writing a 1 to the ON bit of ADCCON. To take a reading, write a 1 to the START bit. Wait for the DONE bit to become 1, then read the result from ADCVAL.



[2] Write C code declaring pointers ADCCON and ADCVAL to access the peripheral.

[2] Write a C function to initialize the ADC.

```
void ADCInit(void) {




}
```

[4] Write a C function to read the ADC.

```
uint32_t ADCRead(void) {




}
```

Suppose the 5-stage pipelined RISC-V processor (see page 5) runs the following program, fetching the `ori` instruction on cycle 1. Suppose the initial memory contents are shown below.

```
          ori  t0, zero, 4
          lw   s0, 8(t0)
          beq  s0, s0, else
          lw   s0, 12(t0)
else:     and  s1, s0, t0
```

Memory Address	Initial Contents
0	42
4	69
8	47
12	28
16	32

[1] What value is written to s1? _____

[1] On which cycle is s1 written? _____

[1] Is the branch taken? _____

[1] What is ALUResultM on cycle 5? _____

Consider modifying the single-cycle RISC-V processor to support the `lbu` (load byte unsigned) instruction. `lbu rd, imm(rs1)` reads an 8-bit byte from memory address (`rs1 + imm`) and places it in the bottom 8 bits of register `rd`. The upper 24 bits of register `rd` are filled with zeros. The address does not need to be a multiple of 4. `lbu` has op = 3 and funct3 = 4. In comparison, `lw` also has op = 3 but funct3 = 2. A table of the instruction formats is included in the material attached to the front of the exam for your reference.

Recall that our memory reads out a 32-bit word on RD from the address specified by A[31:2].

[4] Mark up the multicycle processor diagrams on page 4 of this exam packet to implement `lbu` with as little new hardware as feasible. Using the SWR block from page 7 of this exam is acceptable even though it supports more than just `lbu`. Explain your changes.

[1] The riscvtest.s test code (next page) has highlighted modifications to test the new instruction. As compared to the riscvtest.s from Lab 10, it reverses the inputs to the `sub` at address 30 and replaces a `lw` with `lbu` at 38. Convert `lbu` to machine language.

```
lbu  x2, 96(x0)_____________________________
```

[2] Predict what value should be written to memory address 100 at instruction address 50.

Predicted Value: _____

```
# riscvtest.s
# If successful, it should write the value ??? to address 100

#       RISC-V Assembly         Description             Address   Machine Code
main:   addi x2, x0, 5          # initialize x2 = 5     0         00500113
        addi x3, x0, 12         # initialize x3 = 12    4         00C00193
        addi x7, x3, -9         # initialize x7 = 3     8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7     C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4   10        0041F2B3
        add  x5, x5, x4         # x5 = 4 + 7 = 11       14        004282B3
        beq  x5, x7, end        # shouldn't be taken    18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0     1C        0041A233
        beq  x4, x0, around     # should be taken       20        00020463
        addi x5, x0, 0          # shouldn't happen      24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5) = 1      28        0023A233
        add  x7, x4, x5         # x7 = 1 + 11 = 12      2C        005203B3
        sub  x7, x7, x2         # x7 = 12 - 5 = 7       30        402383B3
        sub  x7, x2, x7         # x7 = 5 - 12 = -7      30        407103B3
        sw   x7, 84(x3)         # [96] = FFFFFFF9 = -7  34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 7         38        06002103
        lbu  x2, 96(x0)         # x2 = [96] = ???       38        ???
        add  x9, x2, x5         # x9 = ??? + 11 = ???   3C        005104B3
        jal  x3, end            # jump to end, x3=0x44  40        008001EF
        addi x2, x0, 1          # shouldn't happen      44        00100113
end:    add  x2, x2, x9         # x2 = ??? + 18 = ???   48        00910133
        sw   x2, 0x20(x3)       # write mem[100] = ???  50        0221A023
done:   beq  x2, x2, done       # infinite loop         54        00210063
```

END OF WRITTEN PORTION OF EXAM

DO NOT PROCEED PAST THIS POINT UNTIL YOU ARE PREPARED TO CEASE ALL WORK ON THE WRITTEN PORTION AND MOVE ON TO THE COMPUTER PORTION.

COMPUTER PORTION OF EXAM

Once you start this question, you may refer to the written portion of the exam, but may not spend any more time on the written portion or change any of your answers on that portion.

Modify your multicycle processor from Lab 10 to support the `lbu` instruction as simply as possible. Your changes should be in accordance with your plan from the previous question, but you can make corrections if necessary. Modify your riscvtest.txt to replace the existing `lw` instructions with the new `lbu` instructions.

[1] Print out the Verilog modules you modified, highlight or circle the changes, and attach them to your exam. Make sure your name is on the attached papers.

[4] Print out a simulation waveform showing at least the value being written to memory location at the end of the program. Circle this value in the waveform, making sure it is legible.

[2] What is the hash printed at the end of your simulation?