Digital Electronics & Computer Engineering (E85)

Harris & Brake

Spring 2020

Final Exam

This is a closed-book take-home exam. Electronic devices including calculators are not allowed. You are permitted one side of two 8.5 × 11" sheets of paper with notes.

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

There is no time limit for this exam. You do not need to work on the exam in one contiguous block of time. However, you should record each of your start and stop times and the total time spent on the exam in the table below.

Return the exam as a single scanned or digitally filled PDF uploaded to Sakai no later than **12 pm on May 14th, 2020**. If you scan your exam, please ensure that your answers are legible. If possible, we encourage you to use a scanner or a smartphone scanning app with the flash enabled.

Alongside each question, the number of points is written in brackets. All work and answers should be written directly on this exam. Insert additional pages of work in your PDF submission as necessary. The work for each problem should appear with its respective problem, not at the end of your PDF. Show your work for partial credit.

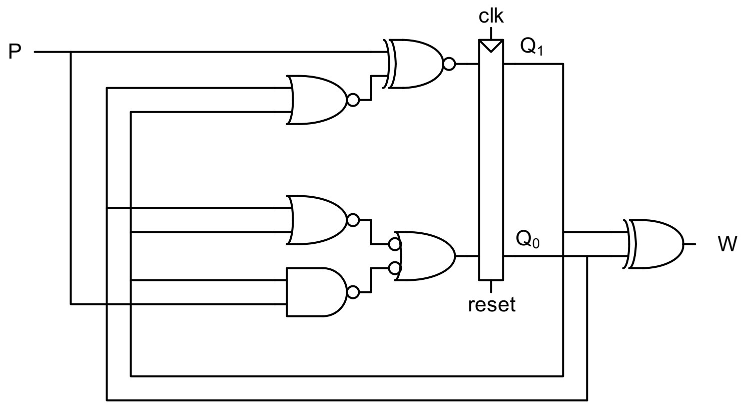
Name: ______________________________________

Page	Score	Total Available
2		**3**
3		**4**
4		**3**
5		**4**
6		**6**
7		**5**
8		**3**
9		**6**
10		**4**
11		**7**
Total		**45**

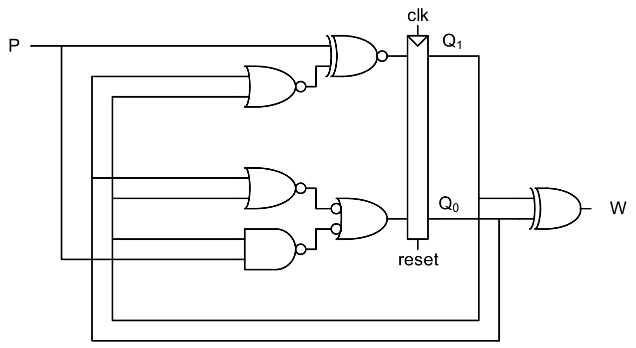
Date	Start Time	Stop Time

Total Time: ________________ hrs

Consider the following finite state machine.



[3] Sketch a state transition diagram for the FSM.

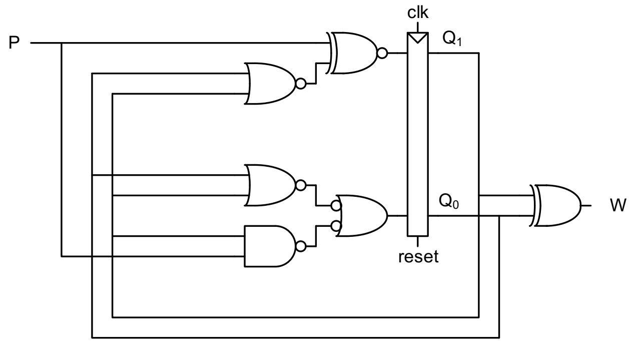


Assume the following component delays and ignore timing issues related to P and W to answer the following questions. The FSM schematic is repeated here for your convenience.

Cell	Propagation Delay (ps)	Contamination Delay (ps)	Setup Time (ps)	Hold Time (ps)
2-input gate	20	10		
Flip-flop	40	25	30	17

[2] If the clock skew is 15 ps, what is the minimum clock period for which the system will work reliably?

[2] How much clock skew may the system experience before it becomes unreliable at low frequencies?

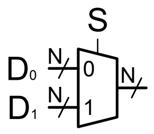


[3] Write a simplified behavioral Verilog description of the FSM (repeated here for your convenience).

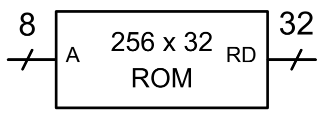
```
module fsm(input  logic clk, reset,
           input  logic P,
           output logic W);
```

```
endmodule
```

[4] Design a 2048 × 8 Read Only Memory (ROM) with input terminals A[10:0] and output terminals RD[7:0] using 256 × 32 ROMs (input terminals A[7:0] and output terminals RD[31:0]) and N-bit 2:1 multiplexers. Draw a schematic of your design. Use no more hardware than necessary. Blocks for your design are shown below.



(a) N-bit, 2:1 multiplexer



(b) 256 × 32 ROM

Block diagram elements

What is the minimum number of logic elements required to build each of the following circuits on the Cyclone IV FPGA that we used in lab and class? A block diagram of a single logic element is attached at the end of the exam for your reference.

```
module mux2(input  logic [31:0] d0, d1,       ELEMENTS: _______ [2]
            input  logic        s,
            output logic [31:0] y);

  assign y = s ? d0 : d1;
endmodule
```

```
module gates(input  logic a, b, c, d,         ELEMENTS: _______ [2]
             output logic y);

  assign y = ((a & b) ^ (c | d)) & ~b;
endmodule
```

```
module fsm(input  logic       clk, reset,     ELEMENTS: _______ [2]
           input  logic       a,
           output logic [1:0] q);

  logic [1:0] nextq;

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

  always_comb
    case(q)
      2'b00: if (a) nextq <= 2'b01;
             else   nextq <= 2'b00;
      2'b01: if (a) nextq <= 2'b10;
             else   nextq <= 2'b01;
      2'b10: if (a) nextq <= 2'b11;
             else   nextq <= 2'b10;
      2'b11: if (a) nextq <= 2'b11;
             else   nextq <= 2'b00;
   endcase
endmodule
```

[2] Express the IEEE 754 floating point number `C1BE0000` in base 10.

[3] Assume the array `c` is initialized at a base memory address of `0x2000_0000` and is running on a 32-bit architecture to answer the following questions.

(a) What is the value of carSize?

(b) What is the value of `y`?

```
typedef struct car {
   char make[56];
   char model[16];
   unsigned int year;
   unsigned long num_seats;
} car;

car myCar;
unsigned long carSize = sizeof(myCar);

car c[10];
unsigned long *y = &c[5].year;
```

[3] Translate the following RISC-V instruction from machine code to its corresponding assembly instruction. You may refer to registers by index rather than by name (e.g., `x9` instead of `s1`). Selected tables from Appendix B are attached to the end of this exam for your reference.

```
0x012E1A93
```

[6] Write RISC-V assembly code to implement the following function, which performs matrix-vector multiplication. The code snippet computes the product of a `3×3` matrix `A` and a `3×1` vector `x`. The result is stored in a `3×1` vector `y`.

Use `s0` for `i`, `s1` for `j`, and `s2` for `temp`. The base address of `A` is stored in `s3`, the base address of `x` is stored in `s4`, and the base address of `y` is stored in `s5`.

```
int A[3][3];

int x[3];
int y[3];
int i, j, temp;

for(i = 0; i < 3; i++)
{
    temp = 0;
    for(j = 0; j < 3; j++)
    {
        temp += A[i][j]*x[j];
    }
    y[i] = temp;
}
```

Consider the 5-stage pipelined RISC-V processor with hazard unit from Chapter 7 running the following program. The first `addi` instruction is issued on cycle 1.

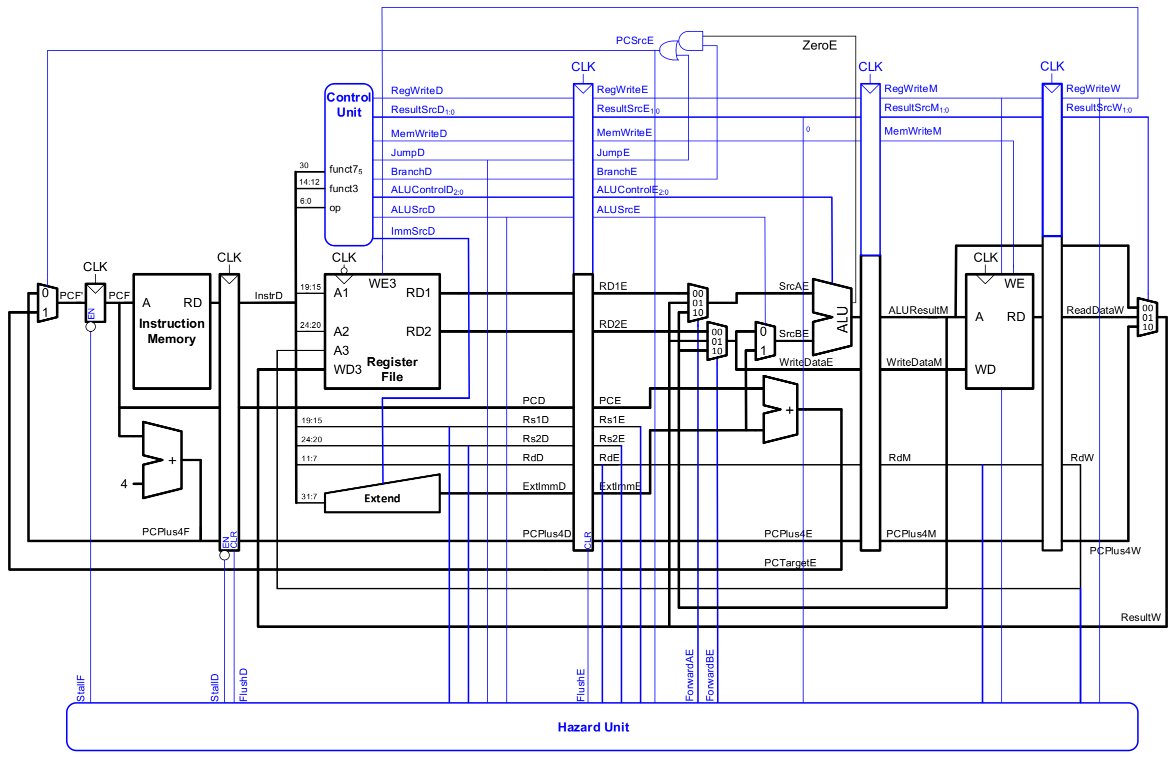
```
       addi t0, zero, 42
       addi t1, zero, 12
       and  t2, t0, t1
       lw   t4, 0(t1)
       add  t5, t4, t4
       beq  ZERO, ZERO, else
if:    or   t3, t0, t1
       sub  t3, t3, t0
       beq  zero, zero, done
else:  sw   t2, 0(t1)

done:
```

[1] On which cycle is `t2` written?

[1] What value is written to `t2`?

[1] On which cycle is `t5` written?

[1] On which cycle is `MemWrite` asserted?



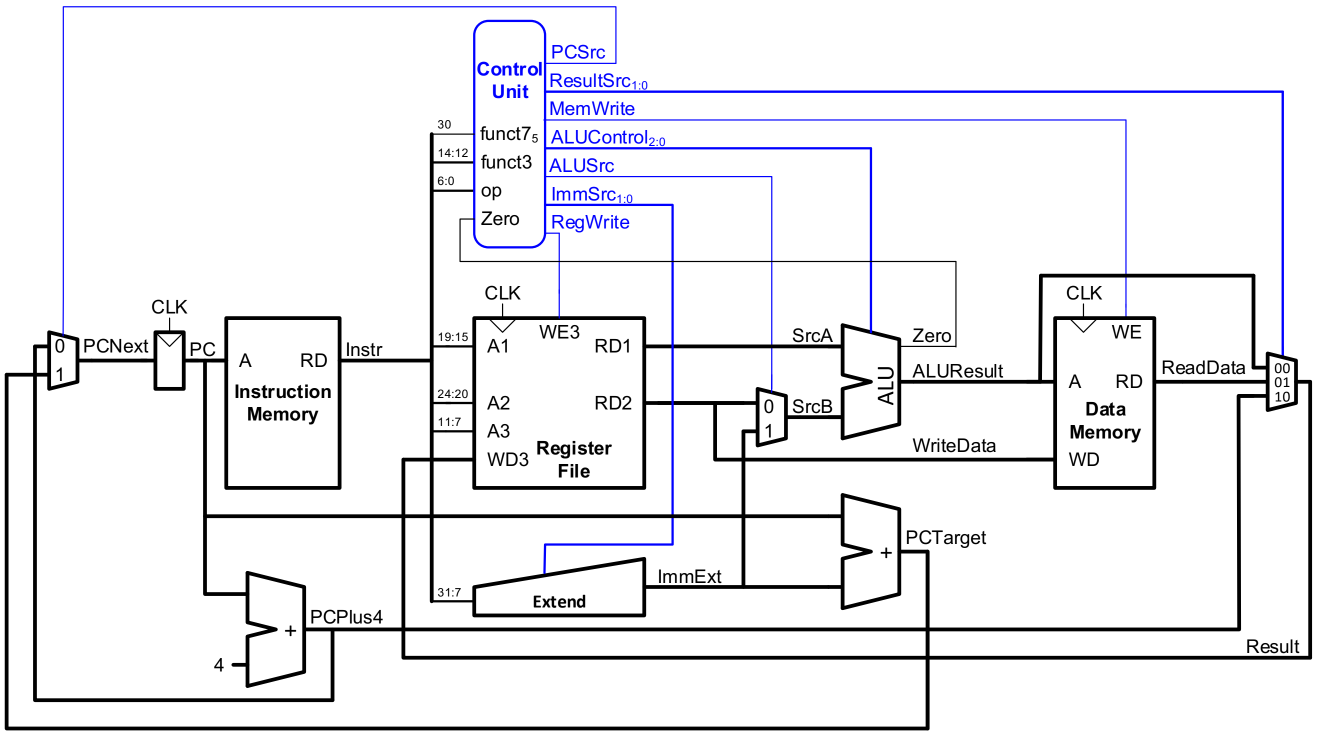
Consider modifying the single-cycle RISC-V processor to support the `lhu` (load halfword unsigned) instruction. `lhu rd, imm(rs1)` reads a 16-bit half-word from memory address (`rs1 + imm`) and places it in the bottom 16 bits of register `rd`. The upper 16 bits of register `rd` are filled with zeros. The address must be even. `lhu` has op = 3 and funct3 = 5. In comparison, `lw` also has op = 3 but funct3 = 2. A table of the instruction formats is included in the material attached to the end 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].

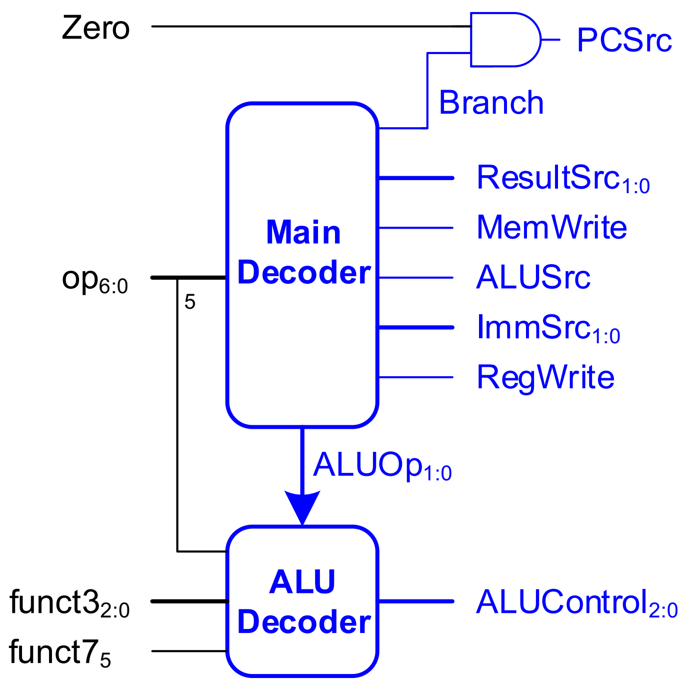
[2] Write `lhu x5, 28(x3)` in machine language. Express your answer in hexadecimal.

[5] Modify the attached single-cycle processor datapath and controller to support `lhu`. Feel free to add new hardware blocks, but keep your design as simple as possible.

Single Cycle Processor



Single Cycle Controller



Main Decoder truth table

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	00	1	0	01	0	00	0
`sw`	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
`beq`	1100011	0	10	0	0	xx	1	01	0
`addi`	0010011	1	00	0	0	00	0	10	0
`jal`	1101111	1	11	x	0	10	0	xx	1

ALU Decoder truth table

ALUOp	funct3	op_5, $funct7_5$	ALUControl	Instruction
00	xxx	xx	010 (add)	`lw, sw`
01	xxx	xx	110 (subtract)	`beq`
10	000	00, 01, 10	010 (add)	`add`
10	000	11	110 (subtract)	`sub`
10	010	xx	111 (set less than)	`slt`
10	110	xx	001 (or)	`or`
10	111	xx	000 (and)	`and`

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
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate

Immediate encodings

Type	Bits Encoded	Immediate	Field Width
I	$imm_{11:0}$	{20{imm_{11}}, $imm_{11:0}$}	12 bits
S	$imm_{11:0}$	{20{imm_{11}}, $imm_{11:0}$}	12 bits
B	$imm_{12:1}$	{19{imm_{12}}, $imm_{12:0}$, 1'b0}	12 bits
J	$imm_{20:1}$	{11{imm_{20}}, $imm_{20:1}$, 1'b0}	20 bits
U	$imm_{31:12}$	{$imm_{31:12}$, 12'b0}	20 bits
R	$rs2_{4:0}$	$shamt_{4:0}$	5 bits

RISC-V instruction formats

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$				rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$				rd	op	**J-Type**
20 bits				5 bits	7 bits	

RV32I instructions – sorted by opcode, then funct3

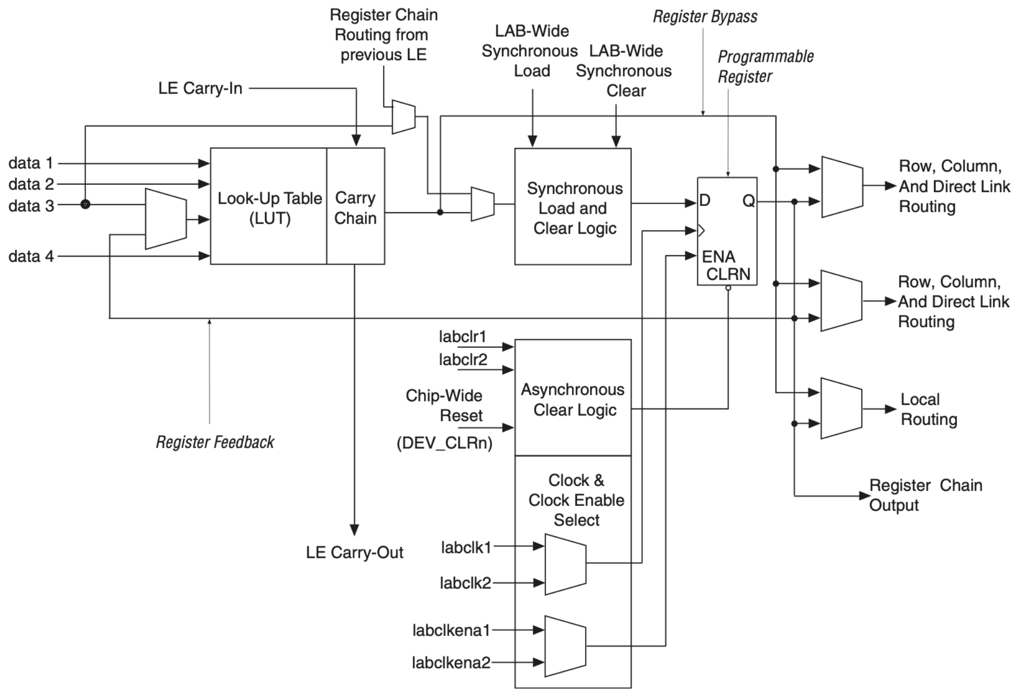
op	f3	f7	Type	Instruction	Description	Operation
3	0	-	I	`lb rd, imm(rs1)`	load byte	rd = SignExt([Address]$_{7:0}$)
3	1	-	I	`lh rd, imm(rs1)`	load half	rd = SignExt([Address]$_{15:0}$)
3	2	-	I	`lw rd, imm(rs1)`	load word	rd = [Address]
3	4	-	I	`lbu rd, imm(rs1)`	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
3	5	-	I	`lhu rd, imm(rs1)`	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
19	0	-	I	`addi rd, rs1, imm`	add immediate	rd = rs1 + SignExt(imm)
19	1	0	R	`slli rd, rs1, shamt`	shift left logical immediate	rd = rs1 << shamt
19	2	-	I	`slti rd, rs1, imm`	set less than immediate	rd = (rs1 < SignExt(imm))
19	3	-	I	`sltiu rd, rs1, imm`	set less than immediate unsigned	rd = (rs1 < SignExt(imm))
19	4	-	I	`xori rd, rs1, imm`	xor immediate	rd = rs1 ^ SignExt(imm)
19	5	0	R	`srli rd, rs1, shamt`	shift right logical immediate	rd = rs1 >> shamt
19	5	32	R	`srai rd, rs1, shamt`	shift right arithmetic immediate	rd = rs1 >>> shamt
19	6	-	I	`ori rd, rs1, imm`	or immediate	rd = rs1 | SignExt(imm)
19	7	-	I	`andi rd, rs1, imm`	and immediate	rd = rs1 & SignExt(imm)
23	-	-	U	`auipc rd, imm`	add upper immediate to PC	rd = {imm$_{31:12}$, 12'b0} + PC
35	0	-	S	`sb rs2, imm(rs1)`	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
35	1	-	S	`sh rs2, imm(rs1)`	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
35	2	-	S	`sw rs2, imm(rs1)`	store word	[Address] = rs2
51	0	0	R	`add rd, rs1, rs2`	add	rd = rs1 + rs2
51	0	32	R	`sub rd, rs1, rs2`	sub	rd = rs1 - rs2
51	1	0	R	`sll rd, rs1, rs2`	shift left logical	rd = rs1 << rs2$_{4:0}$
51	2	0	R	`slt rd, rs1, rs2`	set less than	rd = (rs1 < rs2)
51	3	0	R	`sltu rd, rs1, rs2`	set less than unsigned	rd = (rs1 < rs2)
51	4	0	R	`xor rd, rs1, rs2`	xor	rd = rs1 ^ rs2
51	5	0	R	`srl rd, rs1, rs2`	shift right logical	rd = rs1 >> rs2
51	5	32	R	`sra rd, rs1, rs2`	shift right arithmetic	rd = rs1 >>> rs2
51	6	0	R	`or rd, rs1, rs2`	or	rd = rs1 | rs2
51	7	0	R	`and rd, rs1, rs2`	and	rd = rs1 & rs2
55	-	-	U	`lui rd, imm`	load upper immediate	rd = {imm$_{31:12}$, 12'b0}
99	0	-	B	`beq rs1, rs2, label`	branch if =	if (rs1==rs2) PC = BTA
99	1	-	B	`bne rs1, rs2, label`	branch if !=	if (rs1!=rs2) PC = BTA
99	4	-	B	`blt rs1, rs2, label`	branch if <	if (rs1< rs2) PC = BTA
99	5	-	B	`bge rs1, rs2, label`	branch if ≥	if (rs1>=rs2) PC = BTA
99	6	-	B	`bltu rs1, rs2, label`	branch if < unsigned	if (rs1< rs2) PC = BTA
99	7	-	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	if (rs1>=rs2) PC = BTA
103	0	-	I	`jalr rd, rs1, imm`	jump and link register	rd = PC + 4, PC = rs1 + SignExt(imm)
111	-	-	J	`jal rd, label`	jump and link	rd = PC + 4, PC = JTA

RISC-V multiply and divide instructions (RVM extension)

op	f3	f7	Type	Instruction	Description	Operation
51	0	1	R	`mul rd, rs1, rs2`	multiply	`rd = {rs1 x rs2}`$_{31:0}$
51	1	1	R	`mulh rd, rs1, rs2`	multiply high (signed signed)	`rd = {rs1 x rs2}`$_{63:32}$
51	2	1	R	`mulhsu rd, rs1, rs2`	multiply high signed unsigned	`rd = {rs1 x rs2}`$_{63:32}$
51	3	1	R	`mulhu rd, rs1, rs2`	multiply high unsigned	`rd = {rs1 x rs2}`$_{63:32}$
51	4	1	R	`div rd, rs1, rs2`	divide (signed)	`rd = rs1 / rs2`
51	5	1	R	`divu rd, rs1, rs2`	divide unsigned	`rd = rs1 / rs2`
51	6	1	R	`rem rd, rs1, rs2`	remainder (signed)	`rd = rs1 % rs2`
51	7	1	R	`remu rd, rs1, rs2`	remainder unsigned	`rd = rs1 % rs2`

Diagram of a single Logic Element (LE) in a Cyclone IV FPGA

Figure 2–1. Cyclone IV Device LEs



Register Chain Routing from previous LE
LAB-Wide Synchronous Load
LAB-Wide Synchronous Clear
Register Bypass
Programmable Register
LE Carry-In
data 1
data 2
data 3
data 4
Look-Up Table (LUT)
Carry Chain
Synchronous Load and Clear Logic
D
Q
ENA
CLRN
Row, Column, And Direct Link Routing
Row, Column, And Direct Link Routing
labclr1
labclr2
Chip-Wide Reset (DEV_CLRn)
Asynchronous Clear Logic
Local Routing
Register Feedback
Clock & Clock Enable Select
Register Chain Output
LE Carry-Out
labclk1
labclk2
labclkena1
labclkena2