ROTATOR DESIGN AND SIMULATION

CLASS PROJECT SUMMARY

Instruction set architecture describes the programmer's view of the machine. This level of computer blueprinting allows design engineers to discover expected features of the circuit including:

- data processing instructions like ADD, SUB, AND, EOR, etc.,
- memory load-store instructions like LDR, STR, etc.,
- branch instructions with both conditional and unconditional flow control,
- the names and sizes of the registers provided for computational storage and flow control,
- the size of data memory provided for longer term storage during program execution, and
- the binary encodings for each instruction.

These features are then used by design engineers as they choose components to organize together into a working circuit. Multiple **micro-architectures** are possible for any given instruction set architecture because different design engineers can choose different components and different organizational strategies when implementing the features. In the end, however, any micro-architecture design must implement the features described by the instruction set architecture.

One organizational decision that leads to different micro-architectures is the number of clock periods used per instruction. The three common clock-period strategies are called ***single-cycle***, ***multi-cycle***, and ***pipelined***.

- Single-cycle processors use one clock-period per instruction and the clock-period is set by the total delay of the slowest instruction. This is a disadvantage as faster instructions cannot execute more quickly. The advantage, however, is straightforward control circuitry.
- Multi-cycle processors use multiple clock-periods per instruction and each instruction uses the minimum number of clock periods required for its execution. This allows faster instructions that do not access data memory, like ADD, to avoid the unnecessary delay of the data memory stage. Thus, the advantage is speed for faster instructions. The disadvantage is more complex control because a finite state machine controller must be built to coordinate control signals across multiple clock periods.
- Pipelined processors exploit instruction level parallelism to allow multiple instructions to be in execution at the same time. This is accomplished by adding state registers between the instruction fetch, instruction decode, execute, memory access, and write-back stages of the circuit.

The CE1921 laboratory is designed as a large multi-week project requiring students to design and simulate a ***single-cycle processor*** for a subset of the ARMv4 instructions. Students are required to:

- **Design** VHDL and schematic data path components including registers, a register file, instruction ROM, data memory, ALU, extenders, and controllers.
- **Organize** the components together into a top-level schematic that implements a single-cycle processor.
- **Simulate** the processor using a basic test program.

ROTATOR DESIGN AND SIMULATION

LABORATORY EXERCISE

Most instruction set architectures allow assembly language programmers to embed constants into instructions. For example, ARMV4 allows the assembly language programmer to write **ADD R0, R1, #5**. The constants embedded in instruction binary numbers are **immediately** available to the CPU circuitry once the instruction has been retrieved from instruction memory. This improves performance by eliminating an additional read from data memory.

Modern RISC instruction set architectures use fixed-width instruction binary numbers. ARMV4 encodes every instruction as a 32-bit binary number. Within these 32-bits, bit fields identify the instruction, the location of data, control information for the CPU circuit, and any constants used in the instruction. Thus, the size of the constant number bit field in the instruction binary number is fixed to something less than the 32-bit numbers expected by the ARMV4 ALU. Different ARMV4 instructions use different sized constants. Arithmetic and bitwise logic instructions use 8-bit constants while memory instructions use 12-bit constants and branch instructions use 24-bit constants.

The 8-bit wide **immediate** constant specified in an ARMV4 arithmetic or bitwise logic instruction appears to limit the instruction to a very small unsigned number line from 0 to 255. However, the use of a four-bit **rotation field** gives the instruction much more number line space to use. If a number doesn't fit on the eight-bit number line, then the assembler attempts to create it by rotating some other 8-bit constant right by two times the specified rotation amount. Not all numbers can be created using this rotation technique. If the assembler is not able to create a constant, it generates a program image that stores the constant in memory, and it inserts appropriate load-from-memory instructions to bring the value into a register. Let's look at some example ARM instructions.

Table 1: Example Immediate Mode ARM Data Processing Instructions

ARM INSTRUCTION	CONSTANT FITS ON 8-BIT NUMBER LINE?		8-BIT CONSTANT FIELD	4-BIT ROTATION FIELD
	YES	NO		
ADD R1, R2, #136	X		136	0
SUB R9, R11, #68	X		68	0
ADD R0, R11, #209	X		209	0
SUB R6, R1, #380		X	95	15
ADD R0, R1, #1536		X	6	12
ADD R1, R2, #457		X	convert to load from memory followed by ADD	

The first three instructions in Table 1 fit on the normal 8-bit number line. These numbers are encoded into the immediate field of the instruction binary number with the rotation field set to zero. The fourth instruction uses a constant that does not fit in 8-bits. Here is decimal number 380 as a 32-bit number with the lower 8-bits highlighted to show it exceeding the 8-bit number line.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	0	0

The rotation field is a 4-bit number encoding 0 through 15 two-bit rotations to the right. The assembler attempts to find an 8-bit number that can be rotated into the correct position. In this case, 8-bit number 95 can be rotated 30 positions right.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	0	0

ROTATOR DESIGN AND SIMULATION

The same process can be used to form constant 1536. This constant is encoding as an immediate field of six (6) rotated right by 12 two-bit rotations.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0

The constant 457 cannot be formed. This constant does not result in any 8-bit value that can be rotated.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	0	1

Figure 1 shows the rotate component inserted above the ALU in the CE1921 ARMV4 arithmetic circuit. The 8-bit immediate constant is first extended to 32-bits by the extender and then routed to the rotator by the multiplexer controlling the second ALU data operand. The rotator completes rotation to the right by a specified number of two-bit rotations.

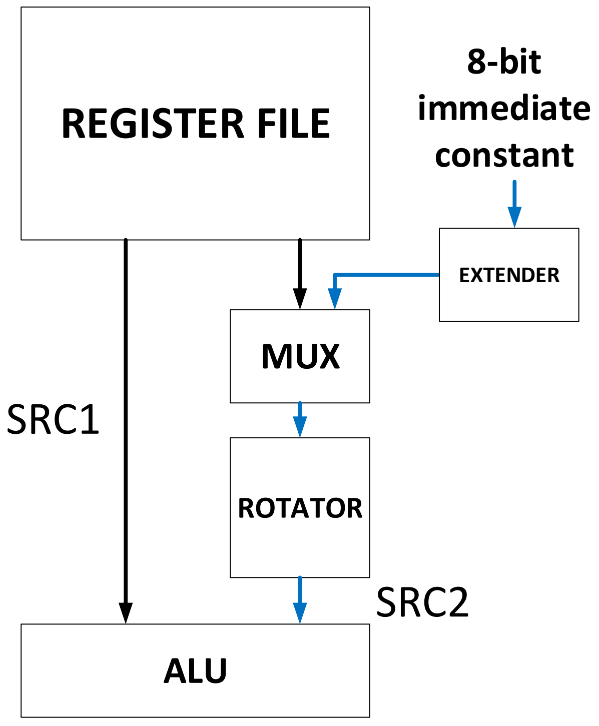
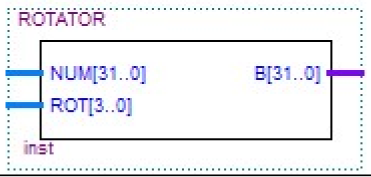


Figure 1: The CE1921 ARMV4 ARITHMETIC CIRCUIT

ROTATOR DESIGN AND SIMULATION

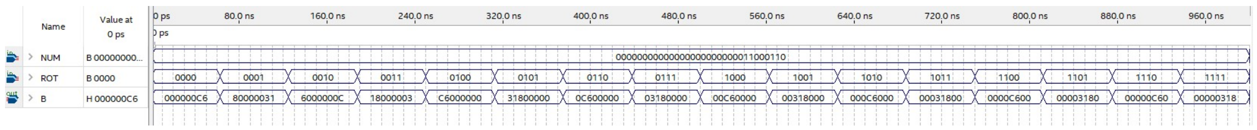
Use the skeleton code provided on the next page to guide your work as you **complete** the VHDL description for the CE1921 extender component.

COMPONENT	BEHAVIOR
	B = NUM rotated right by ROT two-bit rotations

SIMULATION REQUIREMENTS
• Write one arbitrary binary value onto NUM. • Write a count sequence on ROT through simulation time. • Simulation verifies that NUM has been correctly rotated.

SUBMISSION

- You must submit well-commented VHDL code and simulation waveform diagrams using your instructor's preferred submission method. Simulation can be completed using a Quartus university waveform file or a Quartus VHDL testbench with Modelsim-Altera waveform results.
- You must comment on how you know the simulation is correct.
- You are not allowed to use the same arbitrary numbers as this example.
- You are not allowed to set the input number to 0.



```
Name   Value at 0 ps   0 ps  80.0 ns  160.0 ns  240.0 ns  320.0 ns  400.0 ns  480.0 ns  560.0 ns  640.0 ns  720.0 ns  800.0 ns  880.0 ns  960.0 ns
NUM    B 00000000...   00000000000000000000000011000110
ROT    B 0000          0000      0001      0010      0011      0100      0101      0110      0111      1000      1001      1010      1011      1100      1101      1110      1111
B      H 000000C6      000000C6  80000031  6000000C  18000003  C6000000  31800000  0C600000  03180000  00C60000  00318000  000C6000  00031800  0000C600  00003180  00000C60  00000318
```

"I know that this simulation is correct because all values reflect the expected hexadecimal numbers when rotating the original binary number (0x000000C6) right by the selected amount. I made a table that shows the rotation values in binary and in hexadecimal. The table follows. As you can see…"

ROTATOR DESIGN AND SIMULATION

```
-- ********************************************************
-- * project:     rotator
-- * filename:    rotator.vhd
-- * author:      << insert your name here >>
-- * date:        MSOE Spring Quarter 2020
-- * provides:    a component to right rotate a 32-bit
-- *              number a specified number of 2-bit
-- *              rotations
-- * approach:    use a multiplexer to route bits
-- ********************************************************

-- use library packages
-- std_logic_1164: 9-valued logic signal voltages
library ieee;
use ieee.std_logic_1164.all;

-- function block symbol
-- NUM is the input 32-bit number
-- ROT4 is a 4-bit number encoding the number of 2-bit right rotates
-- B is the rotated output for use by the ALU
entity ROTATOR is
port(NUM:  in std_logic_vector(31 downto 0);
     ROT:  in std_logic_vector(3 downto 0);
     B: out std_logic_vector(31 downto 0));
end entity ROTATOR;

-- circuit description
architecture MULTIPLEXER of ROTATOR is
begin

   -- USE CONCATENATION & TO FORM ROTATED OUTPUT
   -- HINT: make a paper table of bits labled 31 downto 0
   --       then start rotating right by two-bit rotations
   --       remember that rotation wraps bits around to the other end
   with ROT select

        -- rotate right 15 two-bit rotations
   B <= NUM(29 downto 0)&B"00" when B"1111",

        -- rotate right 14 two-bit rotations
        NUM(27 downto 0)&B"0000" when B"1110",

        -- << complete other rotations >>

        -- rotate right 3 two-bit rotations
        NUM(5 downto 0)&B"000000000000000000000000"&NUM(7 downto 6) when B"0011",

        -- << complete other rotations >>

        -- rotate right zero two-bit rotations
        NUM when others;

 end architecture MULTIPLEXER;
```