# EXTENDER 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.

# EXTENDER DESIGN AND SIMULATION

## LABORATORY EXERCISE

Instruction set architectures allow assembly language programmers to embed constants into instructions. An example ARMV4 instruction is **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 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 instructions use 8-bit constants while memory instructions use 12-bit constants and branch instructions use 24-bit constants.

ARMV4 calls the constant bit field the **immediate field**. This immediate field must be **extended** by the arithmetic circuitry to become a 32-bit number for use by the ALU. Figure 1 diagrams the CE1921 ARMV4 arithmetic circuit. The **extender** component converts – or extends – the immediate field to a 32-bit number that is routed on to the ALU. In Figure 1, blue arrows show how the extended constant moves toward the ALU.

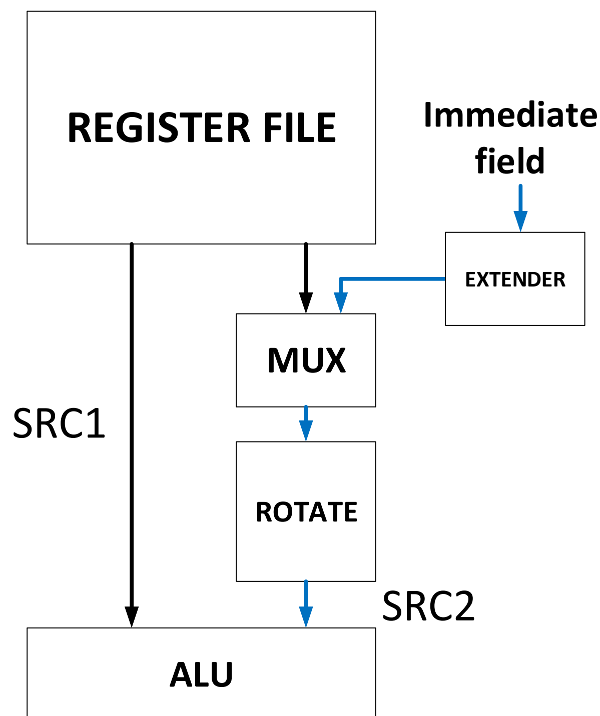



Figure 1: The CE1921 ARMV4 ARITHMETIC CIRCUIT

CC BY NC

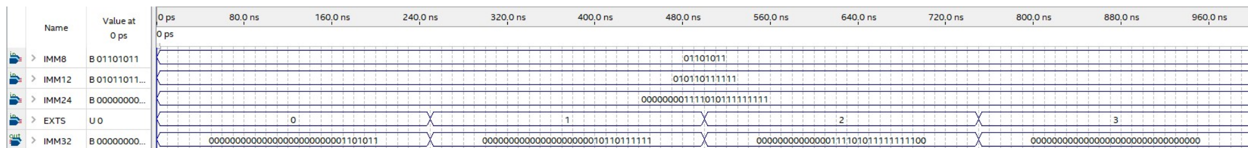
# EXTENDER DESIGN AND SIMULATION

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

<table>
<tr><th>COMPONENT</th><th>BEHAVIOR</th></tr>
<tr><td>EXTENDER<br>IMM8[7..0]<br>IMM12[11..0]<br>IMM24[23..0]<br>EXTS[1..0]<br>IMM32[31..0]<br>inst</td><td>EXTS IMM32<br>------------------------<br>0 000000000000000000000000 & imm8(7 downto 0)<br>1 00000000000000000000 & imm12(11 downto 0)<br>2 { 6 copies of imm24(23) } & imm24(23 downto 0) & 00<br>3 0<br><br>These extension values will be explained in class discussions.</td></tr>
<tr><th colspan="2">SIMULATION REQUIREMENTS</th></tr>
<tr><td colspan="2">• Write one arbitrary binary value onto imm8.<br>• Write one arbitrary binary value onto imm12.<br>• Write one arbitrary binary value onto imm24.<br>• Write a count sequence on EXTS through simulation time.<br>• Simulation verifies that IMM32 has been correctly calculated.<br>• Good simulation practice suggests that both positive IMM24 and negative IMM24 binary values be simulated because option two makes six copies of the sign bit.</td></tr>
</table>

## 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 any of the input immediate values to 0.




"I know that this simulation is correct because all values reflect the expected extensions selected using EXTS. For example, ...."

# EXTENDER DESIGN AND SIMULATION

## SKELETON CODE

```
-- ********************************************************
-- * project:     extender
-- * filename:    extender.vhd
-- * author:      << insert your name here >>
-- * date:        MSOE Spring Quarter 2020
-- * provides:    a component to convert 8, 12, or 24-bit
-- *              ARMV4 constants to 32-bit ALU values
-- * approach:    use a multiplexer to choose 32-bit values
-- *              created from smaller values
-- ********************************************************

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

-- function block symbol
-- IMM8 is the 8-bit arithmetic immediate field input
-- IMM12 is the 12-bit load-store immediate field input
-- IMM24 is the 24-bit branch immediate field input
-- EXTS selects the size of extension based on instruction
-- IMM32 is the extended constant output for use by the ALU
entity EXTENDER is
port(IMM8:  in std_logic_vector(7 downto 0);
     IMM12: in std_logic_vector(11 downto 0);
     IMM24: in std_logic_vector(23 downto 0);
     EXTS:  in std_logic_vector(1 downto 0);
     IMM32: out std_logic_vector(31 downto 0));
end entity EXTENDER;

-- circuit description
architecture MULTIPLEXER of EXTENDER is
begin

   -- truth table
   -- START WITH IMM32 OUTPUT AS ALL ZEROS
   -- USE CONCATENATION & TO FORM OUTPUT
   -- EXTS  SELECTS THIS BEHAVIOR
   -- 0     put IMM8 into the lower 8-bits of IMM32
   -- 1     put IMM12 into the lower 12 bits of IMM32
   -- 2     put IMM24 into IMM32(25 downto 2)
   --       and set IMM32(31 downto 26) = IMM24(23)
   -- 3     unused: output IMM32 = 0
   with EXTS select
   IMM32 <= B"000000000000000000000000"&IMM8(7 downto 0) when B"00", -- ALU
            B"                        "                   when B"01", -- complete

            IMM24(23)&IMM24(23)                           when B"10", -- complete
            B"                         "                  when others; -- complete

 end architecture MULTIPLEXER;
```