CONTROLLER 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.

CONTROLLER DESIGN AND SIMULATION

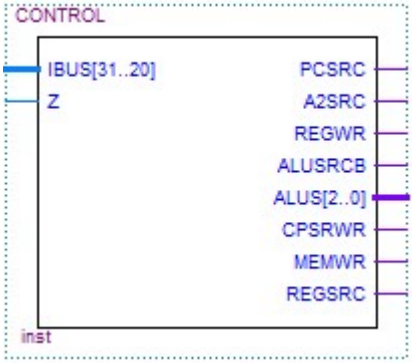
LABORATORY EXERCISE

The ARMv4 control circuit **decodes** bit fields of the machine code binary number and **generates control signals** to:

- route data through multiplexers,
- select the ALU operation, and
- control which registers or memory write data.

The three most-significant nibbles of the machine code binary number contain the fields that must be decoded by the controller. These nibbles become an input bus to the control circuit as shown in the block diagram symbol below. And, because the CE1921 single-cycle processor only supports the BNE and BEQ conditional branches, only the Z-bit from the condition code nibble is required as a circuit input. Unlike the textbook circuit, the CE1921 processor does not include the condition code register as part of the controller. Rather, it is a separate register that sits decide the ALU. This separate register uses an active-low write control signal, called CPSRWR, that becomes active when the CMP instruction is executing.

Use the information, design tables, and the skeleton code provided on the next pages to guide your work as you **complete** the VHDL description for the CONTROL component.

COMPONENT	BEHAVIOR
	• PCSRC routes PC+4 or the Branch Address to the PC input . • A2SRC routes Rm or Rd bit field to the A2 address input of the REGFILE. • REGWR is the active-low REGFILE write control signal. • ALUSRCB routes REGFILE RD2 or the IMM32 to ALU input B. • ALUS selects the ALU operation. • CPSRWR is the active-low write control signal active used to store the CVNZ bits in the CPSR register that sits next to the ALU. It is active only for CMP instructions. • MEMWR is the active-low DATA MEMORY write control signal. • REGSRC routes the data processing ALU result or the LDR DATA MEMORY value to the REGFILE for storage. For those reading the textbook, this signal is called MemtoReg in the textbook design.

SIMULATION REQUIREMENTS

- Overwrite arbitrary value E3A onto the IBUS input.
- Overwrite Z=0 on the Z input.
- Run the simulation.
- Explain your simulation results. How do you know they are correct?
- Repeat for arbitrary value 0A0.
- Repeat for arbitrary value E59.

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.

"I know that this simulation is correct because E3A is a _____ instruction. Checking against my instruction table, all the outputs match."

CONTROLLER DESIGN AND SIMULATION

REFERENCE TABLES FOR CE1921 SINGLE-CYCLE PROCESSOR CIRCUIT COMPONENT CONTROL SIGNALS

PC DATA MUX	
PCSRC	**Y**
0	BRADDR
1	PC+4

REGFILE A2 MUX	
A2SRC	**Y**
0	RM BITFIELD
1	RD BITFIELD

ALU SELECT	
ALUS	**F**
0	ADD
1	SUB
2	AND
3	OR
4	XOR
5	A
6	B
7	constant 1

ALUSRCB MUX	
ALUSRCB	**Y**
0	IMM32
1	REGFILE RD2

REGFILE DATA SRC MUX	
REGSRC	**Y**
0	MEMORY
1	ALU

The CE1921 laboratory processor does not implement the ARMv4 ISA using the textbook circuit. A different circuit is used that simplifies the register file and the calculation of branch addresses. And, the CE1921 laboratory uses VHDL to implement the controller – eliminating the need to break the controller up into smaller pieces that enable paper design. For this reason, some components of Figure 7.13 in the textbook do not appear in the CE1921 circuit while other components do. Here is a mapping between the textbook control signals and the control signals used in laboratory.

TEXTBOOK CONTROL SIGNAL IN FIGURE 7.13	CE1921 LABORATORY CIRCUIT CONTROL SIGNAL
PCSRC	PCSRC
MemtoReg	REGSRC
MemWrite	MEMWR
AluControl	ALUS
ALUSrc	ALUSRCB
ImmSrc	Not used. The opcode bitfield is directly wired to the extender in the lab circuit as the extender selection choices correspond directly with opcode.
RegWrite	REGWR
RegSrc	A2SRC
Not used.	CPSRWR (active-low controls signal active for only CMP instructions)

CONTROLLER DESIGN AND SIMULATION

EXAMPLE TRUTH TABLE UNDER CONSTRUCTION – TRUTH TABLES ARE PROVIDED TO STUDENTS IN EXCEL FORMAT FOR COMPLETION

- **Don't cares are noted with theta (θ) symbols because Dr. Meier generally zeros don't care values when writing equations in VHDL. Use X if you prefer that symbol for don't care values.**

INSTRUCTION	INPUTS						OUTPUTS							
	COND	OP	I	CMD	S		PCSRC	A2SRC	REGWR	ALUSRCB	ALUS	CPSRWR	MEMWR	REGSRC
ADD Rd, Rn, Rm	E	0	0	4	0		1	0	0	1	0	1	1	1
ADD Rd, Rn, imm32	E	0	1	4	0		1	θ	0	0	0	1	1	1
AND Rd, Rn, Rm														
AND Rd, Rn, imm32														
CMP Rn, Rm														
CMP Rn, imm32	E	0	1	A	1		1	θ	1	0	1	0	1	θ
EOR Rd, Rn, Rm														
EOR Rd, Rn, imm32														
MOV Rd, Rm	E	0	0	D	0		1	0	0	1	6	1	1	1
MOV Rd, imm32														
ORR Rd, Rn, Rm														
ORR Rd, Rn, imm32														
SUB Rd, Rn, Rm														
SUB Rd, Rn, imm32														

ADD Rd, Rn, Rm	Rd ← Rn + Rm	PC = PC + 4	PCSRC=PC+4, A2SRC=RM, REGWR=active-low, ALUSRCB=RM=REGFILE RD2, ALUS=ADD=0, CPSRWR=not-active, MEMWR=not-active, REGSRC=ALU
CMP Rn, imm32	Rn - Imm CSPR ←CVNZ	PC = PC + 4	PCSRC=PC+4, A2SRC=RM, REGWR=not-active, ALUSRCB=IMM32, ALUS=SUB=1, CPSRWR=active-low, MEMWR=not-active, REGSRC=don't care (θ) because REGFILE is not writing

```
-- ******************************************************
-- * project:     control
-- * filename:    control.vhd
-- * author:      << insert your name here >>
-- * date:        MSOE Spring Quarter 2020
-- * provides:    a control circuit for the ARMv4 ISA
-- *              instructions implemented in the CE1921
-- *              single-cycle processor
-- * approach:    use when-else statements
-- ******************************************************

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

-- functional block symbol
-- inputs
--    IBUS is the upper 12-bits of the 32-bit machine code
--    Z is the zero condition code flag from the CPSR
-- outputs
--    PCSRC:   0 = BranchAddress    1 = PC+4
--    A2SRC:   0 = Rm               1 = Rd
--    REGWR:   0 = Regfile Write    1 = Regfile does not write
--    ALUSRCB: 0 = imm32            1 = RD2 from Regfile
--    ALUS:    0 = ADD              1 = SUb
--             2 = AND              3 = OR
--             4 = XOR              5 = A
--             6 = B                7 = 0X00000001
--    CSPRWR:  0 = CPSR Write       1 = CPSR does not write
--    MEMWR:   0 = Data Mem Write   1 = Data Mem does not write
--    REGSRC:  0 = Data Mem Value   1 = ALU Value

entity CONTROL is
port(IBUS:     in  std_logic_vector(31 downto 20);
     Z:        in  std_logic;
     PCSRC:    out std_logic;
     A2SRC:    out std_logic;
     REGWR:    out std_logic;
     ALUSRCB:  out std_logic;
     ALUS:     out std_logic_vector(2 downto 0);
     CPSRWR:   out std_logic;
     MEMWR:    out std_logic;
     REGSRC:   out std_logic);
end entity CONTROL;

-- circuit description
architecture DATAFLOW of CONTROL is
   -- declare signals for the IBUS bit fields
   -- data processing
   signal COND : std_logic_vector(3 downto 0);
   signal OPCODE: std_logic_vector(1 downto 0);
   signal I: std_logic;
   signal CMD: std_logic_vector(3 downto 0);
   signal S: std_logic;
   -- load-store
   signal IBAR: std_logic;
   signal PUBW: std_logic_vector(3 downto 0);
   signal L: std_logic;
   -- branch
   signal BRL: std_logic; -- the branch L bit is a different bit than Load/Store L

begin
```

```
   -- assign IBUS bits to internal signals
   COND <= IBUS(31 downto 28);
   OPCODE <= IBUS(27 downto 26);
   I <= IBUS(25);
   -- << continue writing these internal signal assignments >>




   -- write output equations using when-else syntax
   -- include rows from data processing, load-store, and branch truth tables
   PCSRC <= '0' when COND=X"0" and OPCODE=B"10" and BRL='0' and Z='1' else -- beq taking branch
            -- << complete other equations taking branch >>
            else
           '1'; -- PC+4 for all other instructions

   A2SRC <= '0' when COND=X"E" and OPCODE=B"00" and I='0' and CMD=X"4" and S='0' else -- add reg
            '0' when COND=X"E" and OPCODE=B"00" and I='1' and CMD=X"4" and S='0' else -- add imm
            -- << complete>>

   -- complete all equations

end architecture DATAFLOW;
```