SINGLE-CYCLE PROCESSOR DESIGN

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 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.

SUMMARY

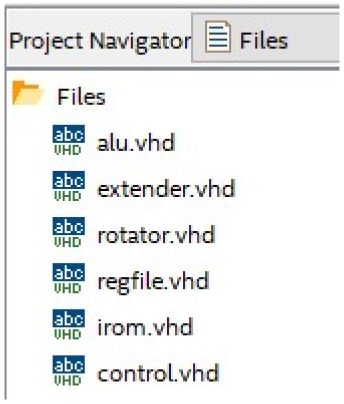
Instruction set architectures can be implemented in multiple ways. Engineers **organize** and **interconnect** components into a circuit capable of executing instructions. When micro-miniaturized and fabricated on silicon, these circuits become a **micro-architecture**. The **textbook organization** follows the ARMv4 instruction set architecture quite closely. ARMv4 reserves register R15 as the PC and any read from register R15 must return the current PC+8. In the textbook micro-architecture, the authors form PC+8 – the partial branch address – using an adder external to the register file and route this partial address through a register file input called R15 to register file output RD1 so that it can be used as the A input of the ALU. Branch instructions then use the ALU to finish forming the branch address as $(PC+8)_{ALU-A}$ + $(SExtImm24 << 2)_{ALU-B}$. The partial branch address PC+8 is never actually stored in R15 in the textbook circuit – it is simply passed through the register file as a bus between the R15 input and the RD1 output multiplexer. The textbook also presents a controller divided into three parts. The first part of the controller is a four-bit register storing the conditional code nibble CVNZ. The second part of the controller is a decoder that generates control signals after examining the opcode, I-bit, command field, and S-bit. Finally, the third part of the controller is a set of conditional execution logic that allows or prevents register file and memory updates during conditional execution.

An **alternate organization** is presented in this laboratory. The CE1921 ARMv4 organization simplifies the register file and the control logic for branching by using a circuit that calculates the full branch address without using the register file or ALU. Also, the instruction set implemented by the controller is kept to a foundational set of ARMv4 instructions including unconditional arithmetic and logic instructions, simply immediate-offset mode LDR and STR instructions, and only the basic branches B, BEQ, and BNE. This reduced instruction set results in a simpler controller that can be implemented quite cleanly using VHDL when-else statements for each control signal.

Always remember that engineering is both art and science. Through lecture, you have studied the ARMv4 instruction set architecture and basic assembly language programming. You have learned the computer science behind this numeric processor. You can see the art of engineering by studying both the textbook organization and this laboratory organization. Both organizations achieve the desired goal of allowing a small set of ARMv4 machine code instructions to execute. The choices made in this week's alternate laboratory organization prevent the processor from executing some ARM instructions that would be expected to work if presented as machine code instructions in a commercial processor. In particular, the LDR and STR instructions are greatly reduced in functionality.

PROJECT CREATION AND PREPARATION

- **Create** a new Quartus project for your single-cycle processor named **scp**.
- **Copy and paste** the VHDL of your completed ALU, extender, rotator, register file, instruction ROM, and controller into the new project. The simplest way is to use Quartus **File → Open → Browse** within the **scp** project to open the VHDL file from the older project and then **select-all → copy → close file**. Then use **new VHDL file** within the **scp** project and **paste** in the VHDL source code. Then **save the file** with the appropriate component name. Here is an example of the **scp project navigator** after adding these files to the project.



Project Navigator | Files

Files
- alu.vhd
- extender.vhd
- rotator.vhd
- regfile.vhd
- irom.vhd
- control.vhd

- **Set** Project Settings:
  - **Assignments→Settings…→Compiler Settings→Advanced Settings (Synthesis)→ Block Design Naming → Choose Quartus II**
    - This assignment allows signal busses in block diagram files to use the bracket [ ] syntax (example: D[13..0]). This option used to be the default. More recent versions of Quartus have changed to a different default option.
  - **Assignments →Settings…→Compiler Settings Sub-level VHDL Input → VHDL 2008**.
- **Right-click → Create Symbol Files** for every VHDL file in **Project Navigator**.
- **Complete** the exercises that follow.

IMPORTANT NOTE

USE ***START ANALYSIS AND SYNTHESIS*** AS THE PROCESSING OPTION THROUGHOUT YOUR WORK.

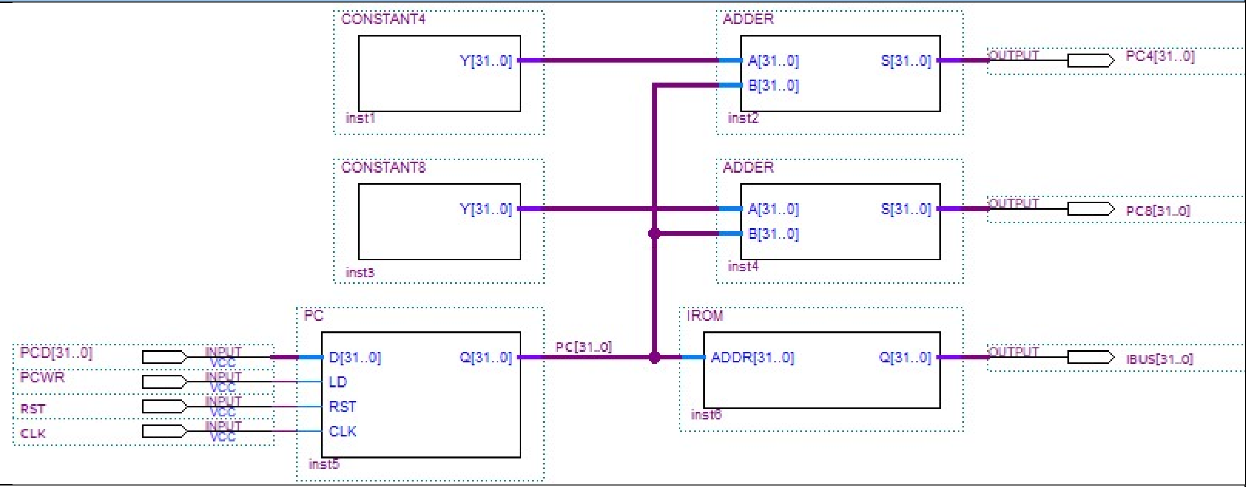
TOOLBAR ICON: KEYBOARD SHORTCUT: CTRL-K

DO NOT ATTEMPT A FULL COMPILATION TO FPGA.

LABORATORY EXERCISES

1. **Complete** the schematic block diagram of the **fetch** circuit stage (**fetch.bdf**).

FETCH CIRCUITRY



DESIGN DISCUSSION

The fetch circuit is responsible for reading the current instruction from the instruction memory and for calculating the control flow numbers needed to advance the program counter to the next instruction.

- Instruction memories are usually flash-ROMs in the microcontrollers used in embedded systems. ROM memories produce a stored value based on an address input but do not write new data in user mode. Thus, this component has no load control signal or sample clock. In addition, large memories generally do not include reset signals because it would take too much time to reset millions or billions of storage locations.
- The ARMv4 control flow equations are **PC 🡐 PC+4** and **PC 🡐 (PC+8)+(SExtImm << 2)**. The fetch circuit uses constant4 and constant8 drivers and adders to calculate PC+4 and PC+8. These calculations occur at the same time because of two separate adders. Architects say that these calculations occur **in parallel**.

NEW VHDL COMPONENTS

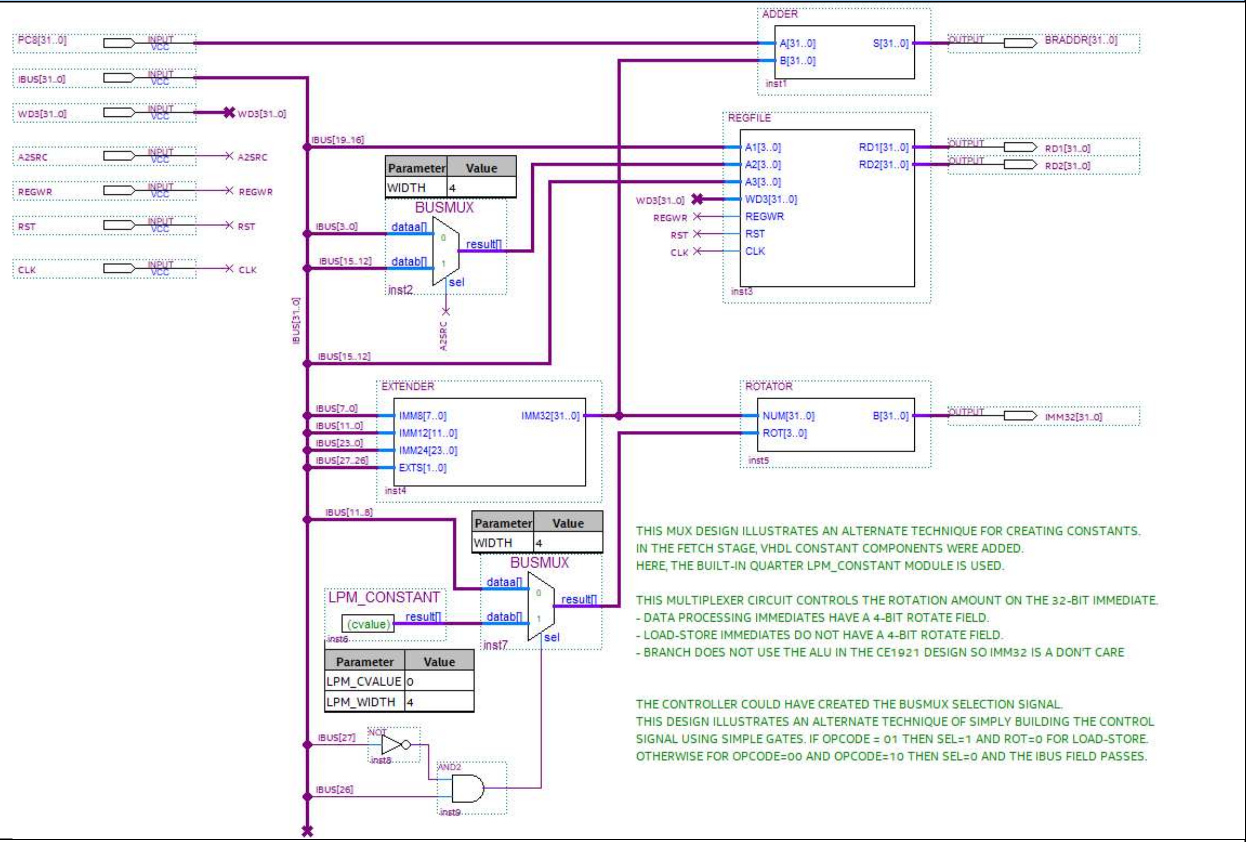
- A rising-edge triggered program counter with active-low reset and active-low load (**pc.vhd**)
- A constant-4 component with an output always equal to 0x00000004 (**constant4.vhd**)
- A constant-8 component with an output always equal to 0x00000008 (**constant8.vhd**)
- An adder using the + operator from package **ieee.std_logic_unsigned** (**adder.vhd**)

NOTES

- The schematic shows all components renamed with incrementing **inst** names.

2. **Complete** the schematic block diagram of the **decode** circuit stage (**decode.bdf**).

DECODE CIRCUIT



THIS MUX DESIGN ILLUSTRATES AN ALTERNATE TECHNIQUE FOR CREATING CONSTANTS.
IN THE FETCH STAGE, VHDL CONSTANT COMPONENTS WERE ADDED.
HERE, THE BUILT-IN QUARTER LPM_CONSTANT MODULE IS USED.

THIS MULTIPLEXER CIRCUIT CONTROLS THE ROTATION AMOUNT ON THE 32-BIT IMMEDIATE.
- DATA PROCESSING IMMEDIATES HAVE A 4-BIT ROTATE FIELD.
- LOAD-STORE IMMEDIATES DO NOT HAVE A 4-BIT ROTATE FIELD.
- BRANCH DOES NOT USE THE ALU IN THE CE1921 DESIGN SO IMM32 IS A DON'T CARE

THE CONTROLLER COULD HAVE CREATED THE BUSMUX SELECTION SIGNAL.
THIS DESIGN ILLUSTRATES AN ALTERNATE TECHNIQUE OF SIMPLY BUILDING THE CONTROL SIGNAL USING SIMPLE GATES. IF OPCODE = 01 THEN SEL=1 AND ROT=0 FOR LOAD-STORE.
OTHERWISE FOR OPCODE=00 AND OPCODE=10 THEN SEL=0 AND THE IBUS FIELD PASSES.

DESIGN DISCUSSION

The decode circuit is responsible for preparing the data for calculation.

- Register-mode data processing instructions require R[Rn] and R[Rm].
- Immediate-mode data processing instructions require R[Rn] and an extended immediate.
- Memory processing instructions require R[Rn] and an extended immediate.
- The decode circuit calculates the branch address (PC+8) + (SExtImm32 << 2) for branch instructions because it is considered a “piece of decoded data” by architects.

NEW VHDL COMPONENTS

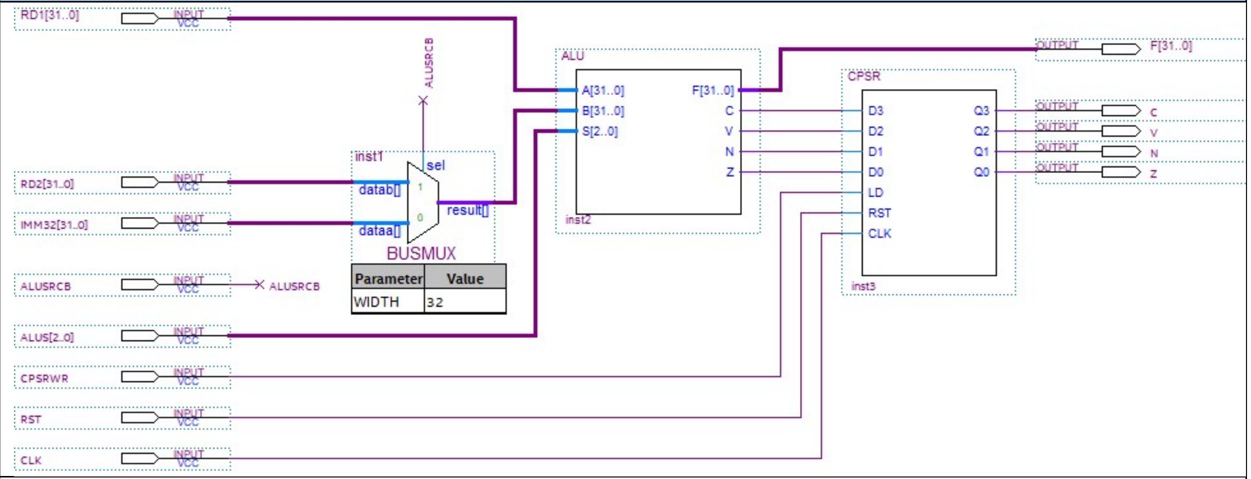
- None

NOTES

- Quartus includes a 2:1 bus multiplexer. Type **busmux** when searching for the symbol.
- Set the **width** of each busmux to 4-bits because they choses between 4-bit wide bitfields.
- The schematic shows all components renamed with incrementing **inst** names.
- The schematic shows the vertical IBUS labeled with the bus name **IBUS[31..0]**.

3. **Complete** the schematic design of the **execute** circuit stage (**execute.bdf**).

EXECUTE CIRCUIT



DISCUSSION

The execute circuit completes the calculation.

- Control signal ALUSRCB routes the correct SRC2 value through the ALUSRCB multiplexer.
- Control signal ALUS selects the ALU function.
- Control signal CPSRWR asserts only when executing the CMP instruction.

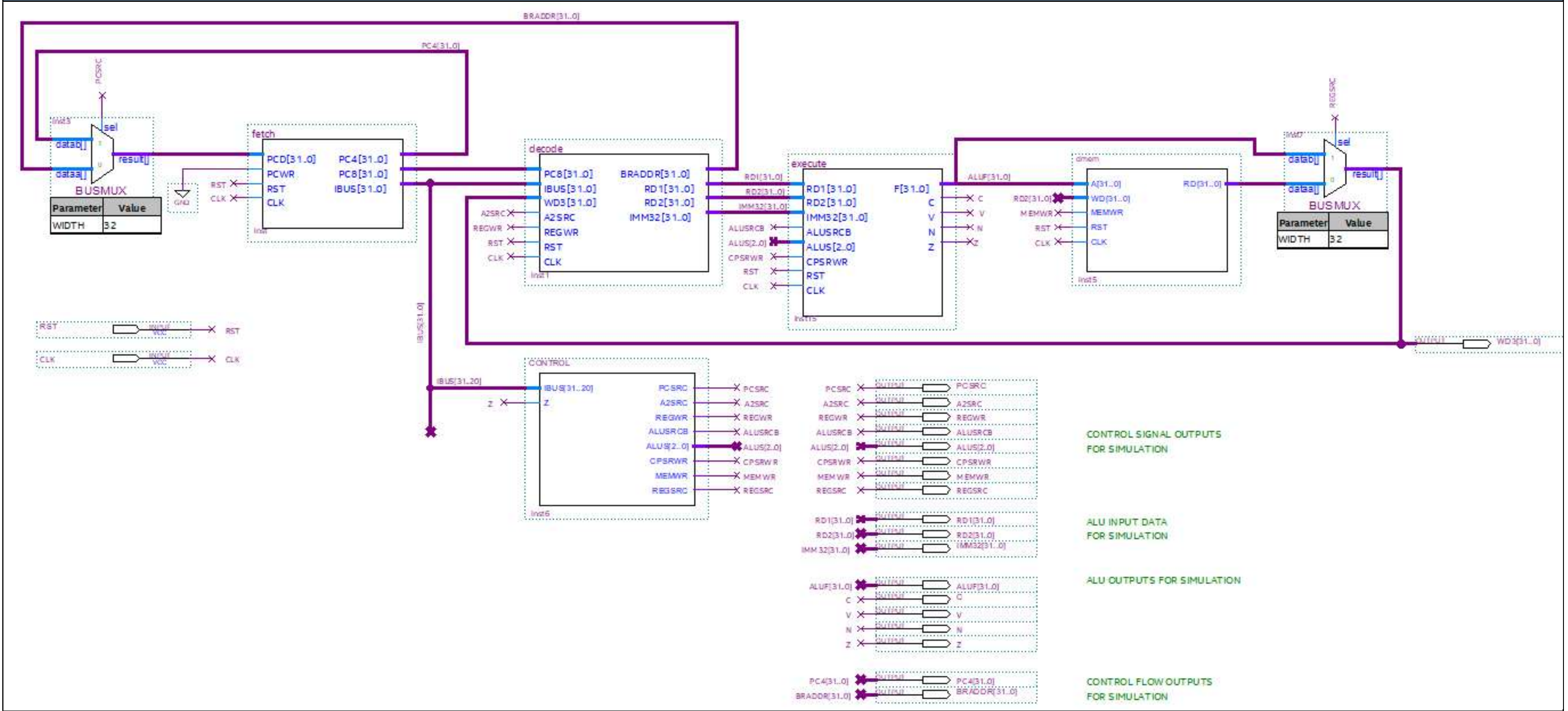
NEW VHDL COMPONENTS

- A current program status register with active-low reset and active-low load (**cpsr.vhd**)

NOTES

- The schematic shows all components renamed with incrementing **inst** names.

TOP LEVEL DESIGN



DISCUSSION

- The processor begins executing the program stored in the instruction ROM after reset. One instruction executes per clock period.
- Data flow returns to the register file on write back signal WD3. Control flow returns to the program counter on PC4 and BRADDR.
- **Basic simulation** of the processor must include RST, CLK, and WD3 because they are sufficient to check data flow.
- **Good simulation** needs additional output pins so that all the control signals can be visualized, the ALU input data can be visualized, the ALU output data can be visualized, and the control flow equations PC4 and BranchAddress can be visualized.

LABORATORY SIMULATION

Simulating a large circuit is an art that develops as engineers-in-training mature this skill through practice over multiple years. When thinking about simulating any circuit, begin by identifying the expected behavior. In the case of the single-cycle processor, it is running a program stored in instruction ROM that calculates the sum of the first 10 numbers using a do-while loop. The sum of the first ten numbers is decimal 55. If the sum is greater than 32, the program writes the value 1 = "yes" into memory location 4 using STR. If the sum is not greater than 32, the program writes the value 0 = "no" into memory location 4. It then enters an infinite loop of two instructions: the first reads memory location 4 using LDR and the second branches back to the LDR. Thus, simulation to show operation should clearly show:

- A do-while loop and the associated PCSRC changes that proof BRADDR is written to the PC.
- A correct sum calculation.
- An infinite loop of LDR, B producing 1, branch, 1, branch, 1 branch, etc. because 55 is greater than 32.

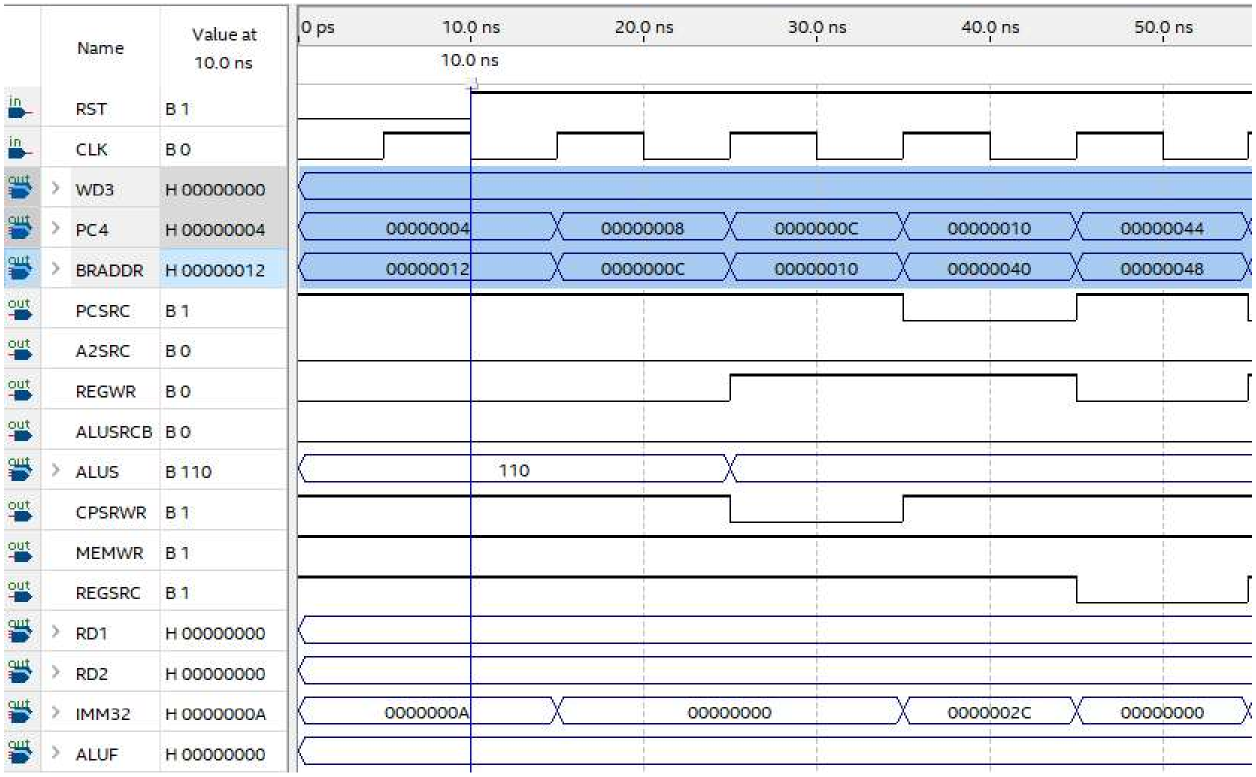
The first step is preparation of the simulation by adding appropriate input, output, and internal test signals. One organization of the waveforms on the Quartus simulation might be:

- RST and CLK clear state registers and advance the program counter. Place at the top of the simulation.
- WD3, PC4, and BRADDR are the data flow and control flow outputs. Place them next in the simulation.
- Control signals are critical to operation. They should be visualized to help identify any design errors. Place them next in the simulation in the order given in the controller design truth table.
- ALU inputs are the data that drive calculation. Place RD1, RD2, and IMM32 next in the simulation.
- ALU outputs are useful when debugging errors in WD3. Place ALUF, C, V, N, and Z next in the simulation.

The second step is thinking about how input signals to the circuit must be controlled by the simulator.

- There are only two input signals: reset (RST) and clock (CLK).
- Overwrite a clock waveform to simulate the system clock. For this exercise, the frequency and period are not important so the default 10ns period is acceptable.
- In motherboard designs, a component called an econo-reset is often used to hold the system in reset while the power supply stabilizes after power-on or return to power after power-failure. This component can also be integrated with a reset switch (https://datasheets.maximintegrated.com/en/ds/DS1233.pdf).
- Simulate the econo-reset by writing logic-1 on the RST signal through simulation time. Then write logic-0 on the RST signal from the start of simulation time to the first falling edge of the clock.

The images on the next pages show steps to verify operation of the single-cycle processor during the design-build-test of this laboratory. Read the comments for each simulation.



Name	Value at 10.0 ns
RST	B 1
CLK	B 0
WD3	H 00000000
PC4	H 00000004
BRADDR	H 00000012
PCSRC	B 1
A2SRC	B 0
REGWR	B 0
ALUSRCB	B 0
ALUS	B 110
CPSRWR	B 1
MEMWR	B 1
REGSRC	B 1
RD1	H 00000000
RD2	H 00000000
IMM32	H 0000000A
ALUF	H 00000000

0 ps
10.0 ns
20.0 ns
30.0 ns
40.0 ns
50.0 ns
10.0 ns
PC4: 00000004, 00000008, 0000000C, 00000010, 00000044
BRADDR: 00000012, 0000000C, 00000010, 00000040, 00000048
ALUS: 110
IMM32: 0000000A, 00000000, 0000002C, 00000000

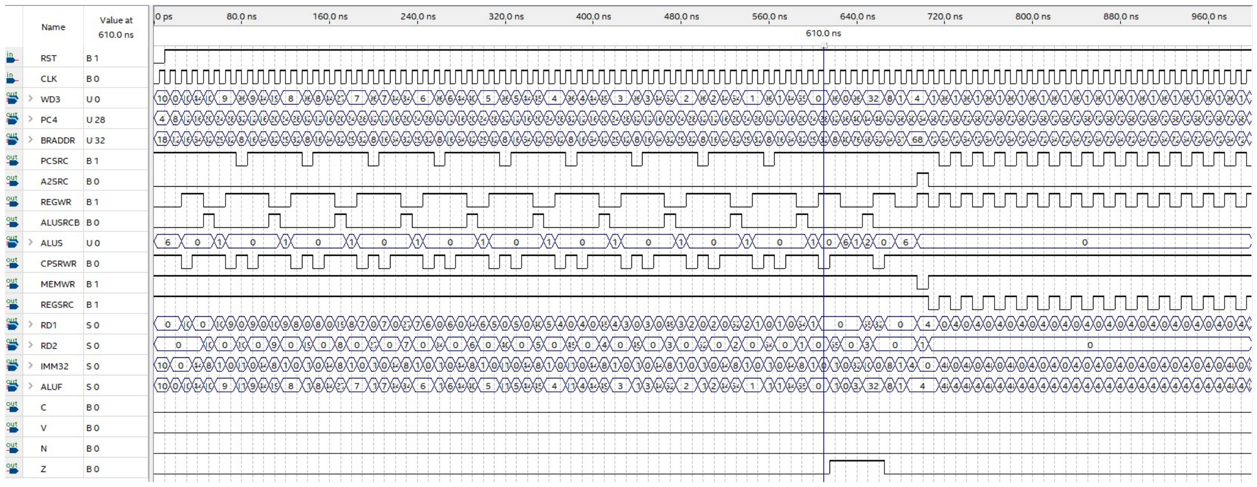
The initial simulation step should examine the first executing instruction to see if the control signals are routing data correctly and calculating the correct result.

- The first instruction in the program is MOV R8, #10 at memory location 0x0000_0000.
- Placing the time bar at the falling edge of the first clock period allows verification of all control signals, data flow, and control flow for the instruction.
- The current PC is at 0x0000_0000 because PC4 reports 0x0000_0004.
- The BRADDR is not relevant for a MOV instruction and can be ignored.
- The control signals correctly route PC+4, Rm, the imm32, and the ALU result through multiplexers.
- The control signals correctly command the ALU to pass B for MOV.
- The control signals correctly change only the register file state using REGWR = ‘0’. All other state registers are not writing data.
- The IMM32 has been properly extended to 32-bits.
- The ALUF is not the correct value.
- **THIS IDENTIFIES AN ERROR. ALUF SHOULD BE 0x0000_000A IF THE ALU IS CORRECTLY PASSING B.**

The circuit and its associated VHDL components must be scanned for design mistakes.

- In this case, the error was discovered in the data memory VHDL. The active edge of the MEMWR control signal was wrong.

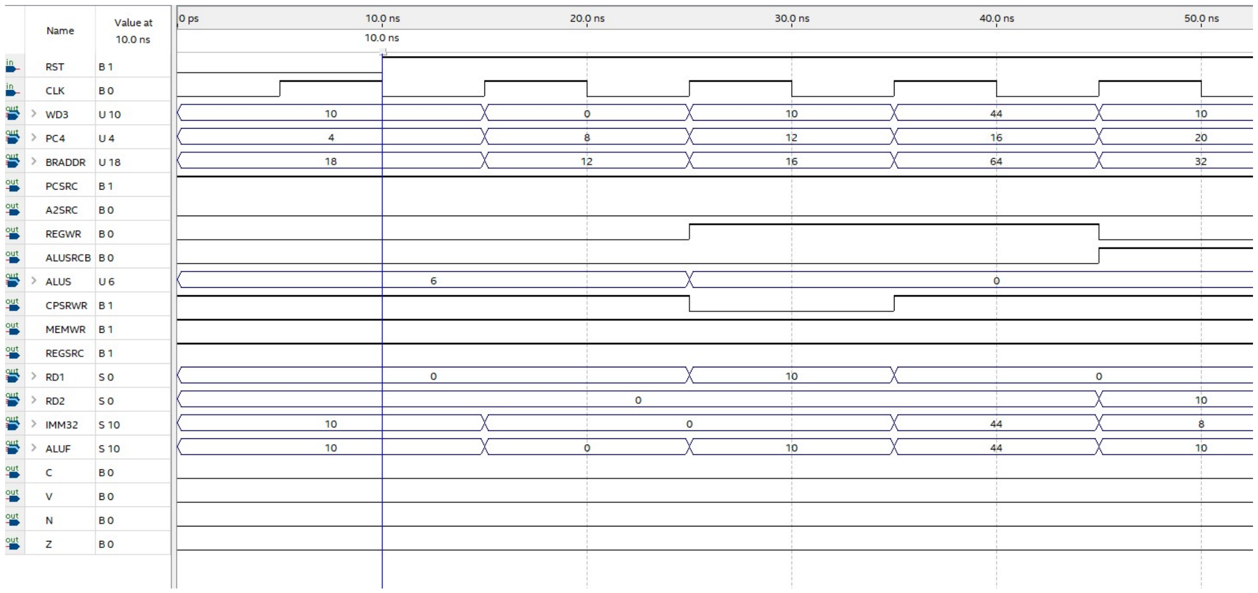
After correction, the circuit must be simulated again.



This time, the simulation shows significant energy transfer on the data flow and program flow signals.

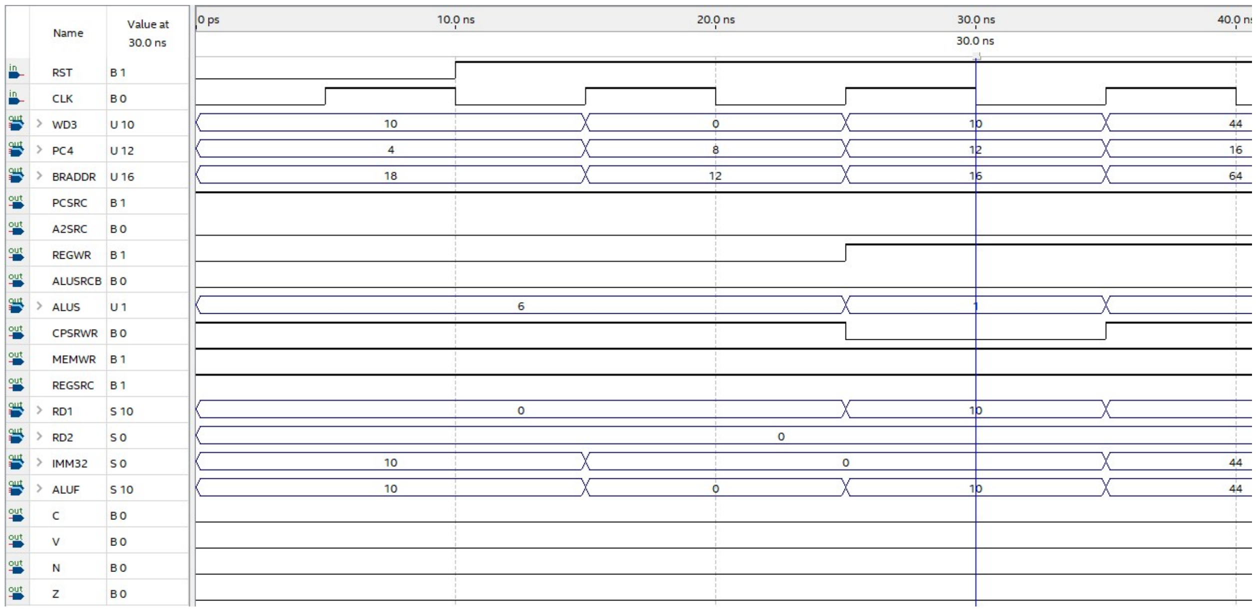
- A loop is clearly visible on the WD3 output. The loop is a down-loop counting from 10 to 0. The time bar is placed just to the right of the time where WD3 becomes 0.
- The loop branching is clearly visible on the PCSRC signal. Nine PCSRC=0 points are noted.
- Compare instructions are clearly visible on the active-low CPSRWR control signal. When compare is executing, CPSRWR=0 and REGWR=1.
- The end of the loop is also clearly visible when compare discovers that the register is equal to zero. At this point the Z bit in the condition code register becomes true.
- There is only one STR instruction in the program and thus MEMWR=0 for only one time period. This is clearly visible to the right of the time bar.
- The end of the simulate shows a two-instruction infinite loop as a 50% duty cycle square wave on PCSRC. This is LDR, B, LDR, B, LDR, B occurring to infinity.
- While difficult to see, the LDR WD3 value is 1 – as expected based the program should have stored a 1 into memory location 4 when it determined that 55 was greater than 32.

This simulation looks very good. It appears that the processor is working. Additional verification requires zooming in to look at key values and ensure that an error isn't overlooked.

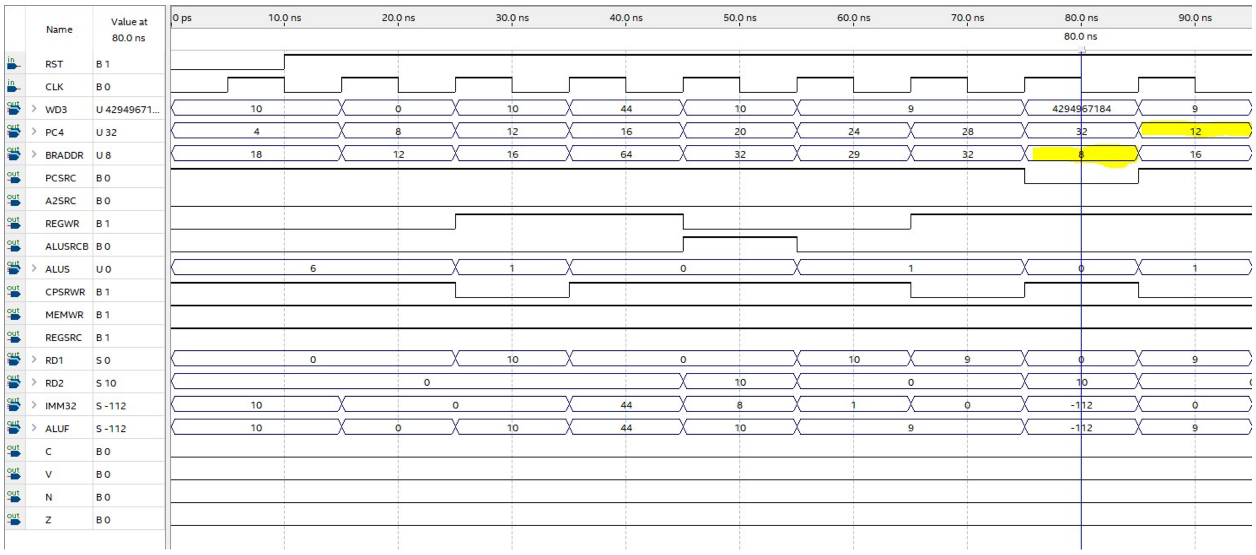


- Returning to the first instruction shows MOV operating correctly: WD3=10 for MOV R0,#10.
- The second instruction is also operating correctly: WD3=0 for MOV R9,#0.

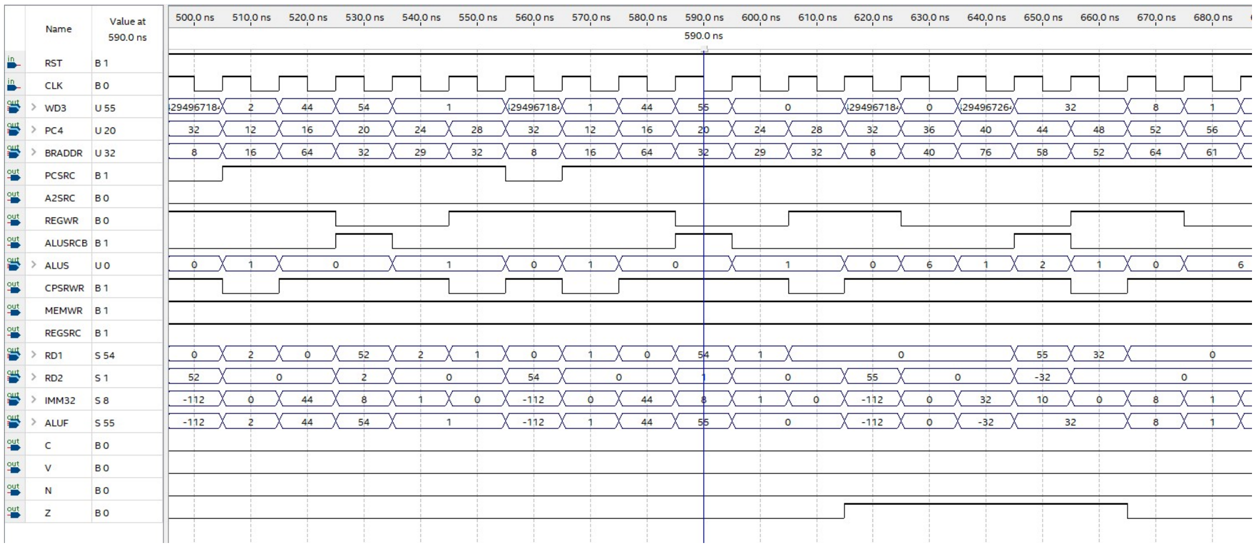
Every instruction in the program can be verified using this zoom-in technique to compare the instruction to its control signals, data flow, data result, and control flow values.



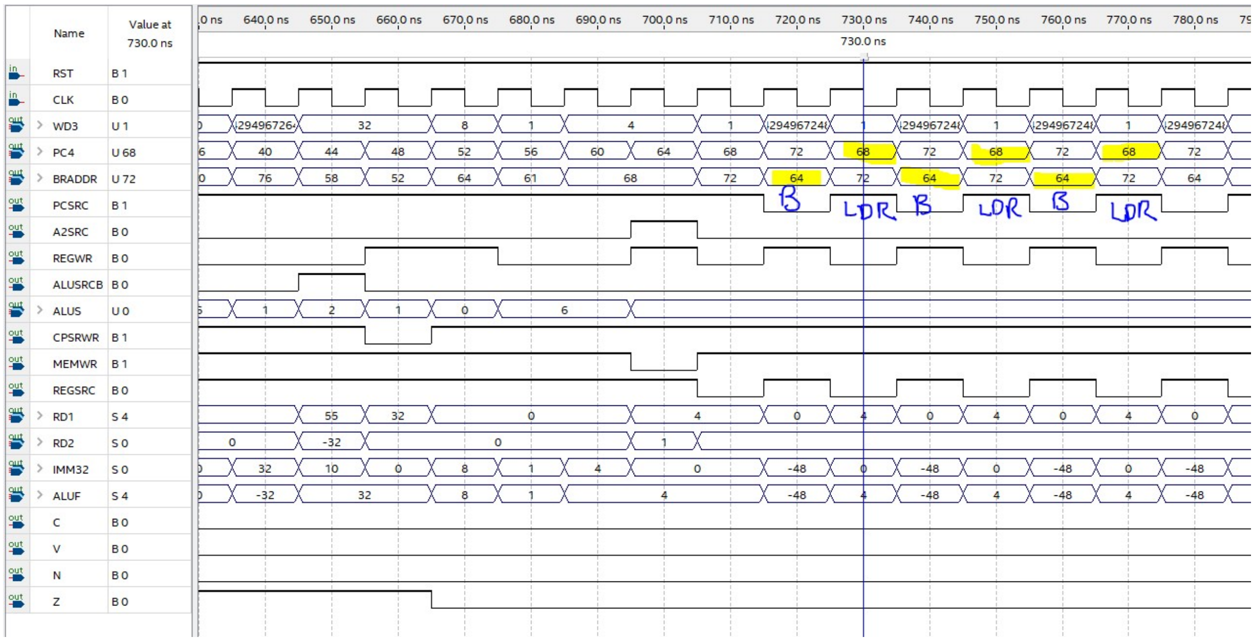
- The third instruction is the previous figures was not operating correctly. CMP should have ALUS=1 but the previous figures show ALUS=0 for the third instruction. The error was fixed, the design rebuilt and re-simulated. This figure now shows the third instruction operating correctly.
- WD3=10 for CMP R8,#0.
- CPSRW=0, REGWR=1, MEMWR=1 shows only the CPSR changing state value for CMP
- WD3 shows 10 because the REGSRC don't care for CMP has defaulted to 1 and is passing the ALUF value.
- The ALU is calculating 10 – 0 = 10. ALUS=1 for subtraction. The result of 10 is seen on WD3 even though it is ignored because REGWR=1 because REGSRC is routing ALUF as WD3.



- The eighth instruction at memory address 0x0000_001C (decimal 28) is operating correctly. This is a BNE instruction at the bottom of the do-while loop.
- PCSRC=0 because BNE **takes the branch** if the previous compare did not find equal numbers (Z=0).
- The WD3 data is not relevant for branch and can be ignored. It is not relevant because the BRADDR is calculated by a separate circuit and does not use the ALU. WD3 shows as a "garbage" value because the ALU **is calculating something** that just doesn't get used in the instruction. It is a random number based on the don't care values for the ALU circuit control signals.
- The highlight PC4 value shows the branch occurring. The BRADDR of 8 is written into the PC on the next rising edge of the clock – causing PC4 to show 12.



- The end of the loop should show the correct answer calculated for the sum of the first 10 numbers. The time bar verifies SUM=55 on WD3.
- The compare that occurs two instructions later (CPSRWR=0) causes the update of the CVNZ bits and after this final time through the loop, the counter has reached 0 and Z becomes 1.



- One final check of the infinite loop verifies that the logic-1 **has been stored in memory** and now is being retrieved successfully from memory using LDR.
- The infinite loop is clearly visible when PCSRC becomes active-low.
- The infinite loop is also clearly visible on the PC4 and BRADDR signal pattern.

SUBMISSION

- You must submit a Quartus archive of your working single-cycle processor to the instructor using the preferred submission method. You can form an archive using **Project → Archive Project**…
- The project archive will be in your project folder with an extension of **.QAR**.
- You must also submit well-commented simulation waveform diagrams that show how you verified the processor worked. **Do not copy the text from the explanations in the simulation section above**. You are responsible for writing your own text.