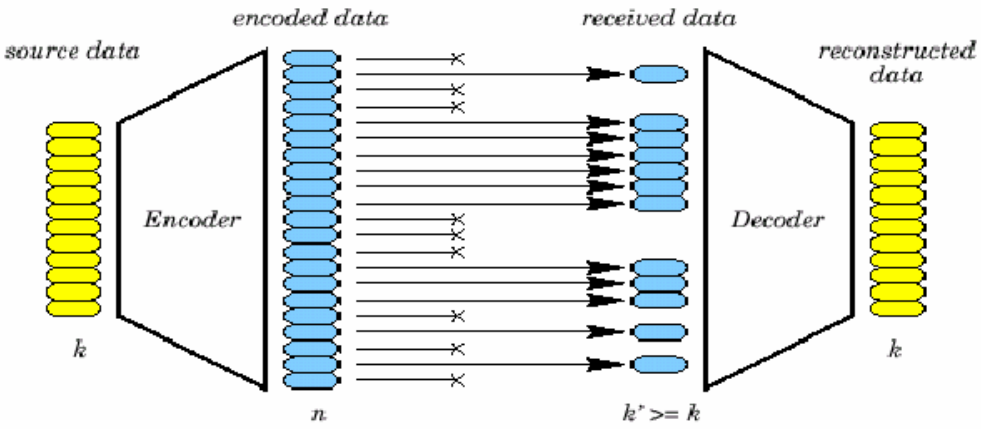
CSE 141L –Winter 2023

This document describes the entire course project for CSE 141L.

This is a very large document. Please do not be too intimidated, it is a full quarter's worth of work! Normally, courses dole our assignments in smaller pieces. One of the goals of 141L, however, is to help you develop skills to manage a large-scale, longer term project.
Skim it once, digest it, and then take a moment. Then go back and re-read smaller pieces in more detail. This document is broken down into the major milestones, so focus on each in turn.

Introduction

Your task this quarter is to design a custom processor that supports specific Forward Error Correction (FEC) tasks. FEC is commonly used in radio communications with lossy links where retransmission is expensive, challenging, or impractical (as well as other domains). As an interesting and motivating example, consider one of the longer-lived computational systems humanity has ever built, the Voyager Probe, which passed out of the heliosphere last year but is *still* sending us data! As an example closer to home, satellite radio streams allow for 4-5s of signal loss (e.g. driving under a highway underpass) without any interruption in the audio stream. Sirius receivers use a custom chip, historically the STA210/240, which is in many ways simply an advanced form of what you are designing and implementing in this course.[1]



Conceptual model of what a generic error correction scheme does

[1]To be clear, you do not need to know anything about FEC design for this course, Voyager's implementation, or Sirius's implementation. Your processor just needs to run the provided programs correctly. However, for those who are interested in learning more on their own for fun, NASA has an excellent technical document on the Voyager communication system: voyager.gsfc.nasa.gov/Library/DeepCommo_Chapter3--141029.pdf. Similarly, here are some details on how early Sirius worked (a lot is still proprietary, unfortunately): spectrum.ieee.org/amp/the-consumer-electronics-hall-of-fame-siriusxm-satellite-radio-system-2650279127.

Table of Contents

ISA Requirements

Your instruction set architecture shall feature fixed-length instructions (machine code) 9 bits wide.

Given the tight limit on instruction bits, you need to consider the target programs and their needs carefully. The best design will come from an iterative process of designing an ISA, then coding the programs, redesigning the ISA, etc.

Your ISA specification should describe:

- What operations it supports and what their respective opcodes are.
    - For ideas, see the MIPS, ARM, RISC-V, and/or SPARC instruction lists
- How many instruction formats it supports and what they are
    - **In detail!** How many bits for each field, where they are found in the instruction.
    - Your instruction format description should be detailed enough that someone other than you could write an assembler (a program that creates machine code from assembly code) for it. (Again, refer to ARM or MIPS.)
- Number of registers, and how many general-purpose or specialized.
- All internal **data** paths and storage will be 8 bits wide.
- Addressing modes supported
    - This applies to both memory instructions and branch instructions.
    - How are addresses constructed or calculated? Lookup tables? Sign extension? Direct addressing? Indirect? Immediates?

The more time and care you put into your specification, the easier the rest of the project will be. This is the *design* element, and it harder than it seems (you have a *lot* of options!).

Some Things to Think About

For instructions to fit in a 9-bit field, the memory demands of these programs will have to be small. For example, you will have to be clever to support a conventional main memory of 256 bytes (8-bit address pointer). You should consider how much data space you will need before you finalize your instruction format. Your instructions are stored in a separate memory, so that your data addresses need be only big enough to hold data. Your data memory is byte-wide, i.e., loads and stores read and write exactly 8 bits (one byte). Your instruction memory is 9 bits wide, to hold your 9-bit machine code.

You will write and run three programs on your ISA. You may assume that each program starts at address 0, and that you will reload the instruction memory specifically for each program. The specification of your branch instructions may depend on where your programs reside in memory, so you should make sure they still work if the starting address changes a little (e.g., if you have to rewrite one of the programs and it causes the others to also shift). This approach will allow you to put all three programs in the same instruction memory later on in the quarter.

Architecture Limitations and Requirements

We shall impose the following constraints on your design, which will make the design a bit simpler:

1. Your core should have separate instruction memory and data memory.
2. You should assume single-ported data memory (a maximum of one read or one write per instruction, not both — Your data memory will have only one address pointer input port, for both input and output).
3. Your instruction memory *should* not exceed 2^10 entries; it *must* not exceed 2^11 entries. If you need the larger number of instruction entries, your writeup *must* explain how these extra entries improve some other performance element.
4. Your data memory *should* not exceed 2^8 entries; it *must* not exceed 2^9 entries. If you need the larger number of data entries, your writeup *must* explain how these extra entries improve some other performance element.
5. You should also assume a register file (or whatever internal storage you support) that can write to only one register per instruction.
    a. The sole exception to this rule is that you may have a multibit ALU condition/flag register (e.g., carry out, or shift out, sign result, zero bit, etc., like ARM's Z, N, C, and V status bits) that can be written at the same time as an 8-bit data register, if you want.
    b. You may read up to two data registers per cycle.
    c. Your register file will have no more than two output ports and one input port.
    d. You may use separate pointers for reads and writes, if you wish.
    e. Please restrict register file size to no more than 16 registers.
6. Manual loop unrolling of your code is not allowed – use at least some branch or jump instructions.
7. Your ALU instructions will be a subset of those in ARMsim, or of comparable complexity.
8. You may use lookup tables / decoders, but these are limited to 32 elements each (i.e., pointer width up to 5 bits).
    a. You may not, for example, build a big 512-element, 32-bit LUT to map your 9-bit machine codes into ARM- or MIPS-like wider microcode. (It was amusing the first time a team tried it, but it got old ☺.)

Some Things to Think About

In addition to these constraints, the following *suggestions* will either improve your performance or greatly simplify your design effort:

1. In optimizing for performance, distinguish between what must be done in series vs. what can be done in parallel.
    a. E.g. An instruction that does an add and a subtract (but neither depends on the output of the other) takes no longer than a simple add instruction.
    b. Similarly, a branch instruction where the branch condition or target depends on a memory operation will make things more difficult later on.
2. Your primary goal is to execute the assigned programs accurately. Secondary goals are:

a. Minimize clock cycle count.
b. Minimize cycle time (short critical paths).
c. Simplify your processor hardware design.

Generic, general-purpose ISAs (that is, those that will execute other programs just as efficiently as those shown here) will be seriously frowned upon. We really want you to optimize a creative special purpose design for these programs only.

Top-Level Interface

Your microprocessor needs only three one-bit I/O ports: **`clock`** input and **`start`** input from the testbench and **`done`** output back to the testbench.

We will use the **`start`** and **`done`** signals to drive your processor. During final testing, the sequence will be as follows:

1. The testbench will set the **`start`** bit high.
    a. Your processor **must not** write to data memory while the **`start`** bit is asserted.
2. The testbench will load operands into specified location in the data memory.
3. The testbench will lower the **`start`** bit.
    a. This should cause your processor to begin executing the first program.
4. When your program has run and your device has stored the result into the specified location(s) in data memory, your device should bring the **`done`** flag high.
5. The testbench will respond by reading and verifying your results.
6. You may then set up your second set of instruction 9-bit words.
7. The testbench will assert the **`start`** bit
    a. Your processor should deassert the **`done`** flag in response
8. The testbench will load the next set of operands into the specified locations in data memory while the **`start`** bit is high.
9. The testbench will lower the **`start`** bit.
    a. Your device should start running the second program.
10. When the second program completes, your processor should assert **`done`**.
11. The testbench will read and verify your results from the second program, then issue the final **`start`** command while loading the third set of operands into data memory. Your **`done`** flag at the end of this program will terminate simulation after the testbench reads and verifies your results.

If you cannot get all three programs to run, separate testbenches for individual programs will also be provided, with correspondingly lower course grades awarded.

What must the processor do?

Your processor must be able execute the following three programs.

Program 1 (forward error correction block coder/transmitter)

Given a series of fifteen 11-bit message blocks in data `mem[0:29]`, generate the corresponding 16-bit encoded versions and store these in data `mem[30:59]`.

Input and output formats are as follows:

```
     input MSW =    0   0   0   0   0 b11 b10 b09
           LSW =   b8  b7  b6  b5  b4  b3  b2  b1, where bx denotes a data bit

    output MSW =  b11 b10  b9  b8  b7  b6  b5  p8
           LSW =   b4  b3  b2  p4  b1  p2  p1  p0, where px denotes a parity bit
```

Example, to clarify "endianness": binary data value = 101_0101_0101

```
        mem[1] = 00000101  -- 5 bits zero pad followed by b11:b9 = 00000_101
        mem[0] = 01010101  -- lower 8 data bits b8:b1
```

You would generate and store:

```
       mem[31] = 10101010 -- b11:b5, p8                 = 1010101_0
       mem[30] = 01011010 --  b4:b2, p4, b1, p2:p1, p0 = 010_1_1_01_0

            p8 = ^(b11:b5) = 0;
            p4 = ^(b11:b8,b4,b3,b2) = 1;
            p2 = ^(b11,b10,b7,b6,b4,b3,b1) = 0;
            p1 = ^(b11,b9,b7,b5,b4,b2,b1) = 1;
            p0 = ^(b11:1,p8,p4,p2,p1) = 0;
```

Program 2 (forward error correction block decoder/receiver)

Given a series of 15 two-byte encoded data values – possibly corrupted – in data `mem[30:59]`, recover the original message and write into data `mem[0:29]`.

This is just (sort of..) the inverse problem. However, in Program 1 there was only 1 unique output for each unique input. Now there are several (how many?) possible inputs for each of the 2**11 possible outputs.

This coding scheme can correct any single-bit error and detect any two-bit error.

The testbench will generate parity bits for various random data sequences, occasionally flip one, occasionally two, of the parity or data bits, and then ask you to figure out the original message.
Your final format = F1 F0 0 0 0 D11 D10 D9 D8 D7 … D0
If no errors, F1F0 = 00; if one error (data or parity), F1F0 = 01; if two errors, FIFO = 1X
(If there are two errors, the test bench looks only to see that you put a 1 in F1, and does not look at any other bits.)

Program 3 (pattern search)

Given a continuous message string in data `mem[0:31]` and a 5-bit pattern in bits `[7:3]` of data `mem[32]`:

a. Enter the total number of occurrences of the given 5-bit pattern in any byte into data `mem[33]`. Do not cross byte boundaries for this count.
b. Write the number of bytes within which the pattern occurs into data `mem[34]`.
c. Write the total number of times it occurs anywhere in the string into data `mem[35]`. For this total count, consider the 32 bytes to comprise one continuous 256-bit message, such that the 5-bit pattern could span adjacent portions of two consecutive bytes.

Note the order ("Endian-ness") of the sequence:

```
data mem[0] = first byte in message string;
                bit [7] is the first bit in the entire string;
data mem[1] = second byte;
                bit [7] of [1] comes right after bit [0] of [0]
```

Example:

```
data_mem[0:31] = 8'h0,8'h0..  Memory to search is all zeros
    data_mem[32] = 8'b00000_000 The pattern to search for is ‘00000’
    data_mem[33] = 4*32 = 128   because the string can show up in any
                                 of 4 different locations in each byte
    data_mem[34] = 32           because every byte contains at least
                                 one copy of the pattern
    data_mem[35] = 252          because pattern can start at bit 0, 1, 2, .., 251
    It cannot start at bit 252, because one bit of the pattern would fall outside the string; likewise, 253, 254, and 255 are "nonstarters," as well
```

What to Submit?

You will turn in milestone reports and (eventually) all your code.

Reports will address questions for each milestone. In describing your architecture, keep in mind that the person grading it has much less experience with your ISA than you do. It is your responsibility to make everything clear. One objective of this course is to help you improve your technical writing and reporting skills, which will benefit you richly in your career.

For each milestone, there will be a set of requirements and questions that direct the format of the writeup and make it easier to grade, but strive to create a report you can be proud of.

Milestone 1 — The ISA

For the first milestone, you will design the instruction set architecture (ISA) for your processor. A quick reminder that an IS**A** is more than just an instruction set. It describes a fair bit about how the machine will work [at least from the programmer's perspective]. It specifies how many registers are available, how memory operates, how addressing works, etc. *Your ISA design will dictate your implementation — plan ahead!*

Milestone 1 Objectives

For this milestone, you will design the instruction set and instruction formats for your processor. You will then write code for the three programs to run on your instruction set.

Milestone 1 Components

0. Team.
   i. List the names of all members of your team, but only one copy of the report should be submitted.

1. Introduction.
   i. This should include the name of your architecture (have fun with this ☺), overall philosophy, specific goals strived for and achieved.
   ii. Can you classify your machine in any of the classical ways (e.g., stack machine, accumulator, register, load-store)? If so, which? If not, devise a name for your class of machine.

2. Architectural Overview. **This must be in picture form.**
   i. What are the major building blocks you expect your processor to be made up of? ***NOTE:*** This *is not* your final processor design, rather an early rough draft of the major elements. Missing details and imprecision are okay at this stage, but you should continue to refine this picture as your design evolves. You will submit an updated diagram with every milestone.

3. Machine Specification
   i. Instruction formats.
      i. List all formats and an example of each. (ARM has R, I, and B type instructions, for example.)
   ii. Operations.
      i. List all instructions supported and their opcodes/formats.
   iii. Internal operands.
      i. How many registers are supported?
      ii. Is there anything special about any of the registers, or all of them general purpose?
   iv. Control flow (branches).
      i. What types of branches are supported?
      ii. How are the target addresses calculated?
      iii. What is the maximum branch distance supported?
   v. Addressing modes.
      i. What memory addressing modes are supported, e.g. direct, indirect?

   ii. How are addresses calculated?
   iii. Give examples.

4. Programmer's Model [Lite]
   i. How should a programmer think about how your machine operates?
   ii. Give an example of an "assembly language" instruction in your machine, then translate it into machine code.
5. Program Implementations

   For each program, give assembly instructions that will implement the program correctly. Make sure your assembly format is either very obvious or well described, and that the code is (very) well commented. If you also want to include machine code, the effort will not be wasted, since you will need it later. We shall not correct/grade the machine code. State any assumptions you make.

   i. Program 1
   ii. Program 2
   iii. Program 3

What to Submit?

You will submit a written report that contains all of the required components of Milestone 1. It is your responsibility to make this report clear and well-organized.

Your report should be a single document, in PDF form.

- Exception: You may include your program implementations as separate "source code" files if you wish.

Milestone 2 — 9-bit CPU: Register file, ALU, and fetch unit

In this milestone, you will design the top level, register file, control decoder, ALU (arithmetic logic unit), data memory, muxes (signal routing switches), lookup tables, and fetch unit (program counter plus instruction ROM) for your CPU.

For this and future designs, we want the *highest* level of your design to be a schematic and SystemVerilog code. You may either hand-draw the schematic or generate it using the Quartus RTL Viewer function.

Anything below that can be schematic (again either drawn or generated by Quartus) and SystemVerilog, or just SystemVerilog. The SystemVerilog files implement the symbols included in the block diagram file. Everyone will use Questa/ModelSim for simulation and Intel (formerly Altera) Quartus II for logic synthesis in the Cyclone IVE family, device **EP4CE40F29C6**.

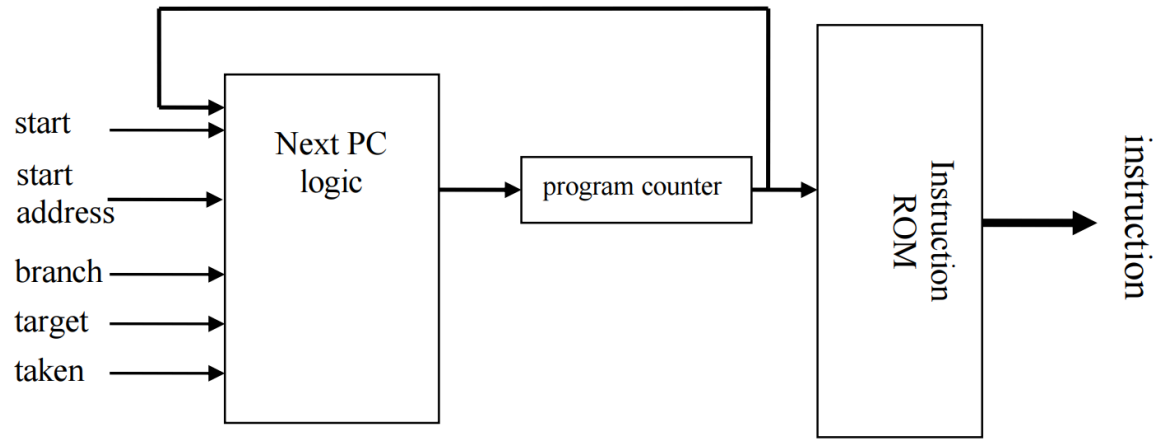
In addition to connecting everything together at the top level, you will demonstrate the functionality of each component separately through schematic, SystemVerilog, and timing printouts.

Milestone 2 Objectives

The primary goal of this milestone is to show individual components operating as desired. All of the pieces of your processor will need to work in isolation before final integration.

CPU Design Refreshers and Helpful Tips

The fetch unit points to the current instruction from the instruction memory and determines the next out of the program counter (PC). It should look something like the following diagram:



The *program counter* is a state element (register) that outputs the address pointer of the next instruction. *Instruction ROM* is a Read Only Memory block that holds your 9-bit machine code. It does not have to hold your actual code (generated in Milestone 3) yet at this point (but if you have already written it then it might as well). It should hold something so we can see the effect of changing PCs while your processor runs. The *next PC logic* takes as input the previous PC and several other signals and calculates the next PC value.

The inputs to the next PC logic are:

- *start* – when asserted, it sets the PC to the starting address of your program.
- *start_address* – has the starting address of your program.
- *branch* – when asserted it indicates that the prior instruction was a branch.
- *taken* – [optional.. more on this in lecture] when the instruction is a branch, this signal when asserted indicates the branch was resolved as taken.
- *target* – [some options.. more on this in lecture] where this branch is going

On non-branch instructions, the next PC should be PC+1 (regardless of the value of *taken*). For branch instructions, the new PC is either PC+1 (branch not taken) or *target* (branch taken). If your branches are ALWAYS PC-relative, then you can redefine *target* to be a signed distance rather than an absolute address if you want. Make sure you tell us this is what you're doing. (*Note: ARM and MIPS increment their respective PCs by 4, simply by convention because their machine codes are 32 bits = 4 bytes wide. We'll just increment by 1, for each 9-bit value of our machine code.*)

How to Present Your Implementation

You will demonstrate each element of your design in two ways.

First, with schematics such as the one shown above, plus your SystemVerilog code. Obviously, you must also show all relevant internal circuits with further SystemVerilog code.

Second, you must demonstrate correct operation of all ALU operations, register file functionality, and fetch unit functions with timing diagrams. An example of a (partial) timing diagram will be demonstrated in class; yours will be longer. The timing diagrams, for example, should demonstrate all ALU operations (this includes math to support load address computation, or any other computation required by your design), each with a couple of interesting inputs. Make sure any relevant corner/unusual cases are demonstrated. If you support instructions that do multiple computations at the same time, you need to demonstrate them happening at the same time. Note that you're demonstrating **ALU operations**, not instructions. So, for example, instructions that do no computation (e.g., branch to address in register) need not be demonstrated. There will also be a timing diagram for the fetch unit, showing it doing everything interesting (increment, absolute jump/branch, conditional jump/branch, etc.). The schematics and timing diagrams will be difficult for us to understand without a *great deal* of annotation. Good organization of files and Verilog modules also helps.

Milestone 2 Components

Your Milestone 2 report should add on to your Milestone 1 report (you are building your final report over time).

Your Milestone 2 report must include a changelog that indicates where any significant changes have been made since your Milestone 1 submission. Please restrict this to highlighting substantial architectural or operational changes. You may include a changelog per section, or a final changelog at the end, or something in between as best suits your report. You do not need a changelog for new sections.

- Some things, such as your Introduction, may have no changes; this is fine/expected.

Your Milestone 2 reports must add the following. You may add these to existing sections in your report or add new sections, as you deem appropriate:

- A list of ALU operations you will be demonstrating, including the instructions they are relevant to. Also, a brief description of the register file functionality is needed.

- Full Verilog models, hierarchically organized **if** your top level module contains subassembly modules, some of which contain smaller modules.

- Well-annotated timing diagrams or transcript (diagnostic print) listings from your module level Questa/ModelSim runs. It should be clear that your program counter / instruction memory (fetch unit) and ALU works. If your presentation leaves doubt, we'll assume it doesn't.

- Your Architectural Overview figure should be revised with more detail / needed updates.

Answer the following question:

- Will your ALU be used for non-arithmetic instructions (e.g., MIPS or ARM-like memory address pointer calculations, PC relative branch computations, etc.)? If so, how does that complicate your design?

What to Submit?

You will submit a written report that contains all of the required components of Milestones 1 and 2. It is your responsibility to make this report clear and well-organized.

Your report should be a single document, in PDF form.

- Exception: You may include your program implementations as separate "source code" files if you wish.

Milestone 3 — An Assembler & Early Integration

Assemblers convert human-readable assembly code to computer-readable machine code. Your code from Milestone 1 is the former, but your processors will need the latter.

Milestone 3 Objectives

Implement an assembler. Begin the process of integrating your processor components.

Tasks

1. Write an assembler which converts your assembly code from Milestone 1 into 9-bit binary machine code. We will provide sample code, but you may use any language you wish. This should be a fairly simple string access, map, print sequence.
2. If you have not already done so in Milestone 2, write a **top-level** SystemVerilog model of your design which instantiates the ALU, fetch (program counter) unit, instruction memory (either inside fetch or separate), register file, data memory, control decoder, and any other blocks you need. This does not need to actually run the three problems yet — that will be the final piece — but it should compile cleanly in both Questa/ModelSim and Quartus II.

Milestone 3 Components

Your Milestone 3 report should add on to your Milestone 2 report.

Your Milestone 3 report must include a changelog that indicates where any significant changes have been made since your Milestone 2 submission.

Your Milestone 3 reports must add the following. You may add these to existing sections in your report or add new sections, as you deem appropriate:

- An example of input to and output from your assembler.
    - [unlikely]: If your assembler does anything beyond what a 'normal' assembler would be expected to do, explain this as well.

- Your Architectural Overview figure should be revised with more detail / needed updates. [Might you be able to automate this drawing now?]

What to Submit?

You will submit a written report that contains all of the required components of Milestones 1, 2, and 3. It is your responsibility to make this report clear and well-organized.

Your report should be a single document, in PDF form.

- Exception: You may include your program implementations as separate "source code" files if you wish.

Document Revision History

- v1.1.0
  - Limit instruction memory capacity
  - Limit data memory capacity
- v1.0.0
  - Initial Release.

