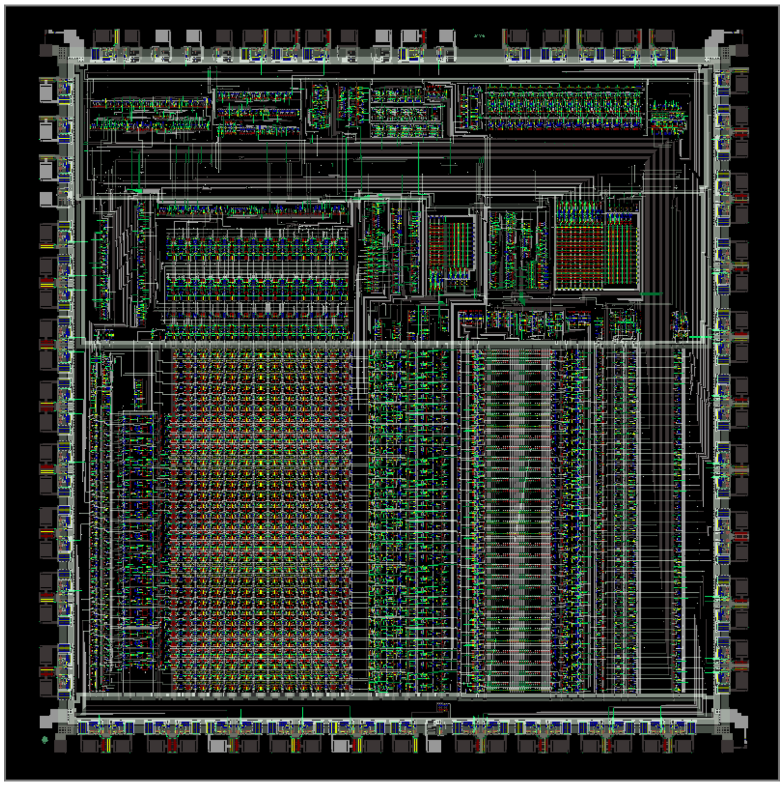
Lane Department of Computer Science and Electrical Engineering

CpE 271L Digital Logic Laboratory

Design of a Simple Central Processing Unit

Fall 2023



West Virginia University - College of Engineering and Mineral Resources

Central Processing Unit (CPU) Overview

(Getting your feet wet with some background info, valuable info regarding the project starts on page 5)

A CPU can be thought of as a complex, programmable state machine beast that can perform operations according to a set of instructions that have been stored in some memory. Modern CPUs are *very* difficult to understand or try to reverse-engineer because of the stack-design method needed for backwards compatibility with older generations as well as the protection around certain manufacturing companies' intellectual properties (IP) for the likes of AMD, Intel, Qualcomm, ARM, etc..

Although this incremental development over the years have caused the hardware architecture to become cluttered and confusing, at their hearts, processors still go through the same process of performing a "fetch-decode-execute" cycle.

The processor retrieves an instruction from memory, decodes the instruction to determine what actions need to be performed, performs the necessary actions, and then begins retrieving the next instruction from the next memory location. On their own, individual instructions perform simple tasks[3] (move values around different registers, fetch values from a specific memory address, etc..), but when instructions are arranged in special sequences, extremely complex tasks can be performed. When instructions are arranged in this manner, they are collectively referred to as a *program*.

This program can have a sequence like the following:

1. Fetch instruction from memory to start executing
2. Instruction is decoded (lookup Instruction Set Architecture if interested how this happens)
3. Operands/Values are collected and stored in internal registers inside the CPU

After the program is fetched, the execution stage starts happening where operations get performed on the inputs/operands brought in. These operations can be anything from adding numbers, to logic manipulations, and more. After the results are obtained, they either are stored in registers for further usage later (branch prediction in more complex CPU's for example) or are stored back into memory by overwriting some other locations. Once this is all done, the CPU moves on to start working on the next instruction residing in memory. This typically takes several clock cycles to accomplish (up to 50+ cycles) for one instruction to be completed depending where the values are being loaded from (L1->L3 cache, RAM, SSD/HDD).

This whole process is a highly sophisticated, choreographed procedure that thousands of engineers work together to get it working as efficiently as possible with the tightest margins of error/time-wasting. This is where the idea of pipelining, parallelism, executing instructions out-of-order, and other techniques become very important and are the tiny details that differentiate different companies from each other's designs and performances. One of these technologies is executing two instructions at the same time using pipelining if they have no dependencies between each other, with an implementation called Simultaneous Multithreading (SMT), or

Hyper-Threading. Intel and AMD support two-way SMT whereas IBM supports up to eight-way.

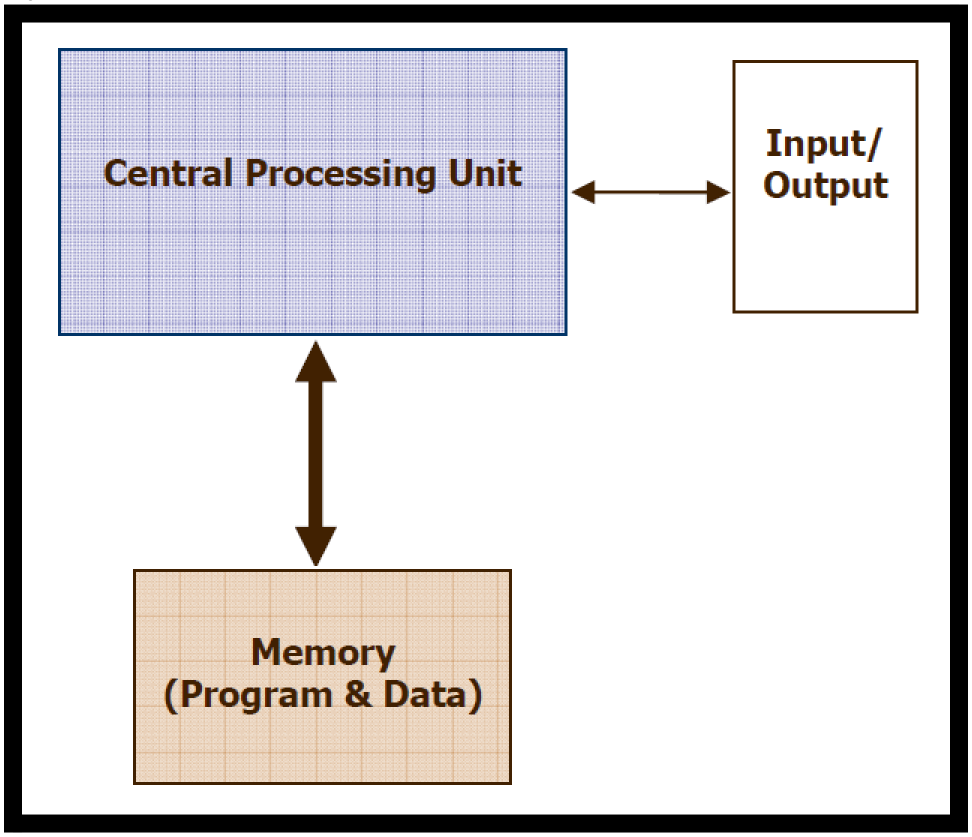


Figure 1: Microprocessor - High Level View

The architecture above follows the Von Nuemann model, in which **program instructions and data are stored in the same memory space**. This means that one must pay careful attention when creating a program to avoid accidentally overwriting instructions or important data. If you have ever worked with a high level language such as C, you might have experienced a mitigation against that when you encountered a "segmentation fault" error; this is when you try to access a memory location inside your onboard RAM that's an "instruction/important data" that you are technically not allowed to overwrite. When you start working with low-level stuff such as VHDL/Assembly, you have to keep track of what is data that you want to preserve and not overwrite.

Zooming into the CPU, one will typically find the following components residing within. The CPU block here in Figure 2 refers to one "core", nowadays you hear about quad/hex/8 "core" CPUs for your standard non-server PC.

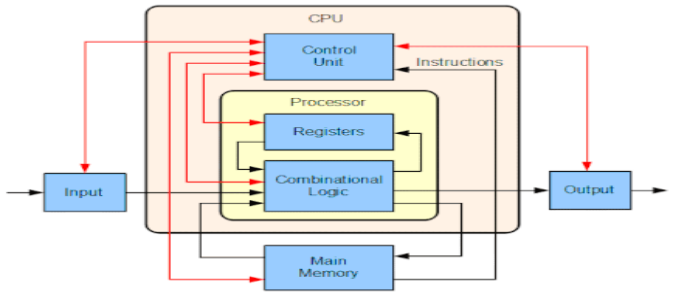
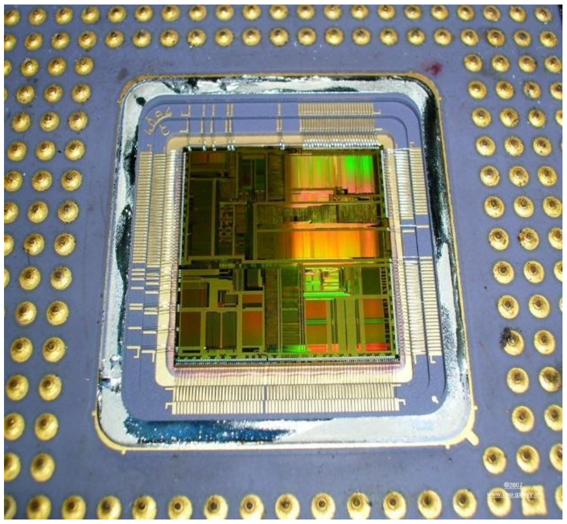
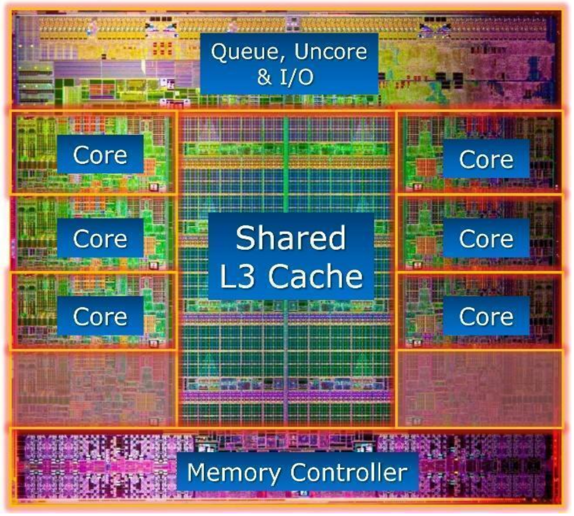


Figure 2: CPU Internal Components Structure

This is as simple as a CPU can get, but modern CPUs tend to cram more components into one chip to look something like that:

Figures 3&4: Exposure of CPU Die (colorful silicon wafer)

Where each core might have an architecture like the following:

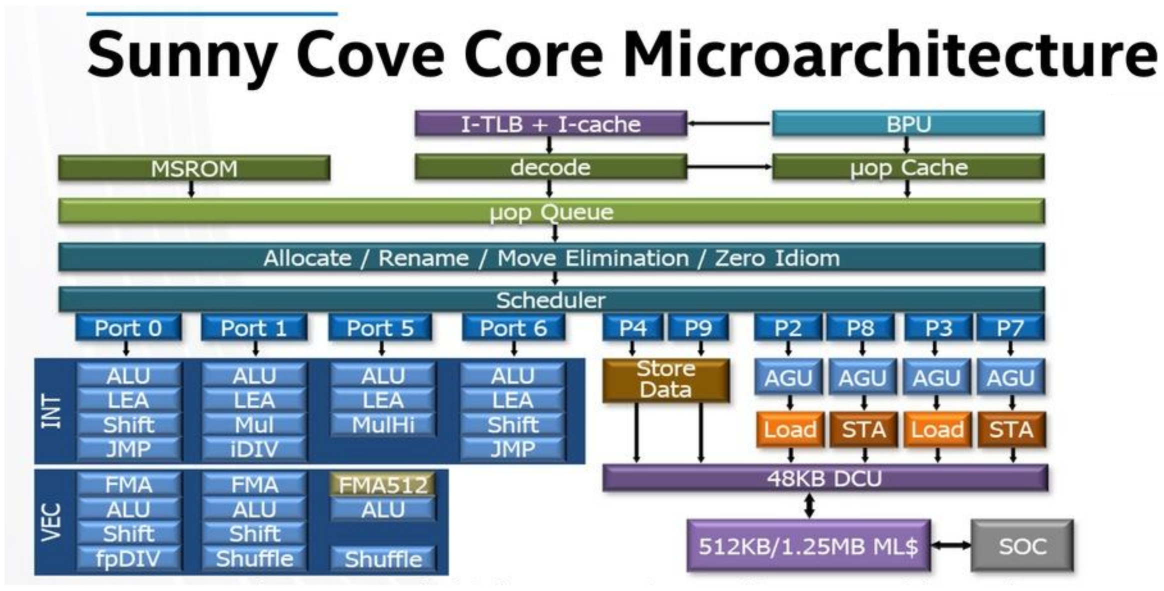


Figure 5: CPU's Single Core Architecture

<u>**You are tasked with designing a CPU that has the following architecture:**</u>

CPU Architecture

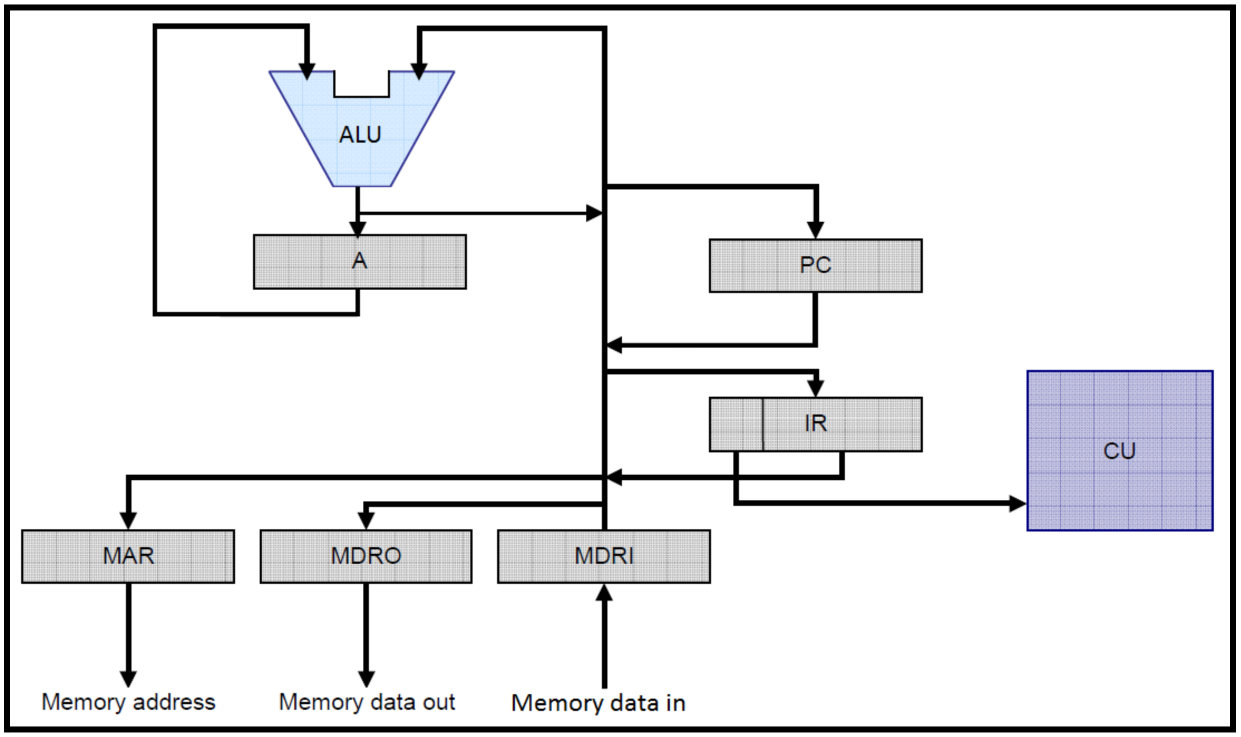


Figure 6: Simplified CPU Architecture

Figure 6 shows the simplified internal architecture of the CPU, as well as the bus lines connecting them together. **Keep in mind that each bus/wire might have a**

different width associated with it (varying number of bits can be transferred at a time). Figure 6 does not show/include the memory module/RAM used to fetch instructions/values from. This CPU contains the following components:

A – Accumulator Register

IR – Instruction Register

CU – Control Unit

PC – Program Counter Register

ALU – Arithmetic & Logic Unit

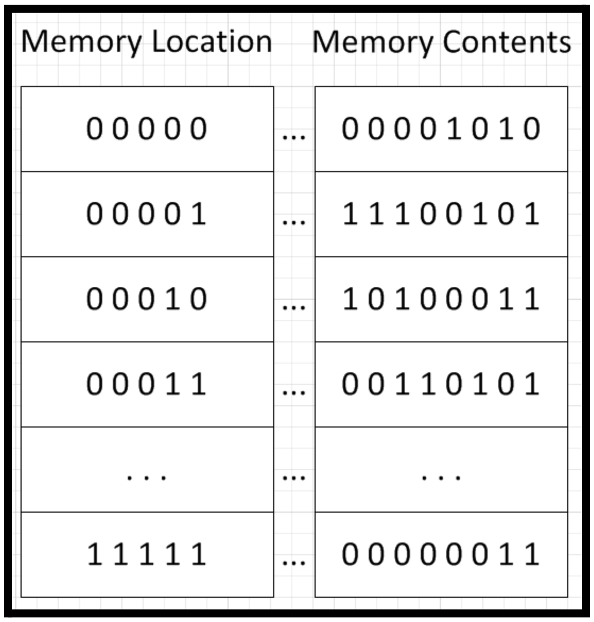
MAR – Memory Address Register

MDRI – Memory Data Register (Input)

MDRO – Memory Data Register (Output)

Memory Structure (RAM)

For this project, the memory module will be based on the memory component created in Lab #9. **The memory address bus is 5-bits wide, allowing for a total of 32 memory locations. Each memory location contains an 8-bit value**, which can be read from or written to, according to the "Write_Enable" bit.



Memory Location		Memory Contents
0 0 0 0 0	...	0 0 0 0 1 0 1 0
0 0 0 0 1	...	1 1 1 0 0 1 0 1
0 0 0 1 0	...	1 0 1 0 0 0 1 1
0 0 0 1 1	...	0 0 1 1 0 1 0 1
. . .	...	. . .
1 1 1 1 1	...	0 0 0 0 0 0 1 1

Figure 7: Memory Layout

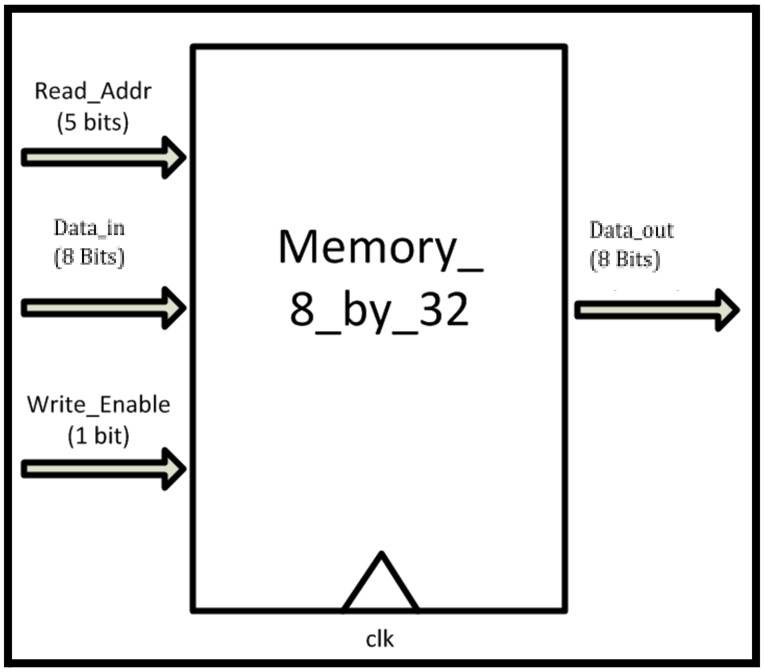


Figure 8: Memory Entity (VHDL)

Instruction Format

Each instruction is represented by an 8-bit value (pulled from RAM). The 3 most significant bits contain the “Op Code”, which describes the operation that the instruction performs. **The remaining 5 bits represent the memory address of the “Operand”, which contains the data used by the instruction.**

Op Code			Address of the Operand				
Op2	Op1	Op0	A4	A3	A2	A1	A0

Figure 9: Instruction Format

The CPU for this project will need to perform three distinct functions: **LOADA, ADDA, and STOREA**. Each instruction performs an operation involving an 8-bit register known as the “Accumulator”. This register contains the output of the CPU’s Arithmetic & Logic Unit (ALU) and is routed back to the ALU to serve as one of its two inputs. The Accumulator and ALU are described in further detail later in the handout.
The Op Codes for the three instructions are represented as follows:

Op 2	Op 1	Op 0	Instruction
0	0	0	LOADA
0	0	1	ADDA
0	1	0	STOREA

Figure 10: Instruction Op Codes

LOADA: Loads the data specified by the Operand into the Accumulator.
Example: LOADA 10h ; *Loads the Accumulator with the contents of memory location 10h (10h = 0x10)*

ADDA: Adds the data specified by the Operand with the contents of the Accumulator.
Example: ADDA 10h ; *Adds the contents of memory location 10h to the value stored in the accumulator*

STOREA: Stores the contents of the Accumulator in the memory location specified by the Operand.
Example: STOREA 10h; *Stores the contents of the Accumulator in memory location 10h*

Arithmetic & Logic Unit (ALU)

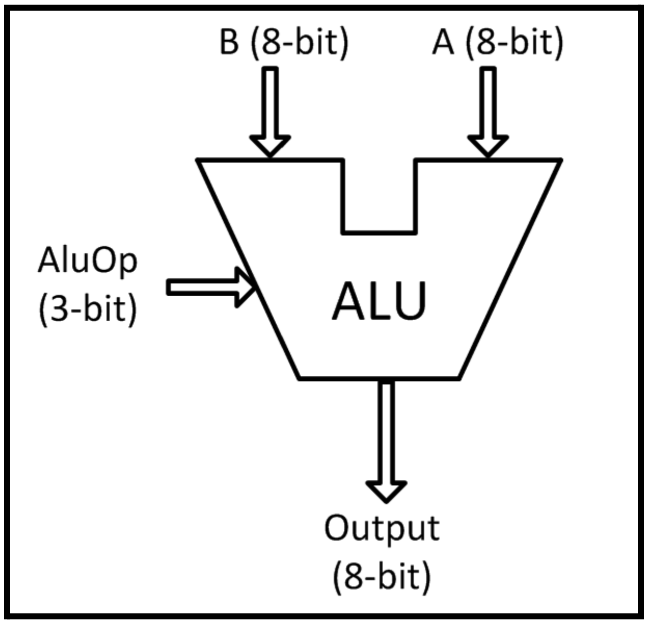
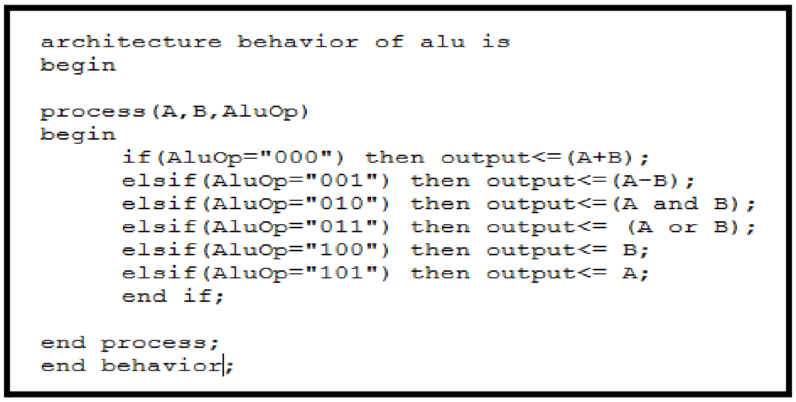


Figure 11: ALU Entity (VHDL)

The ALU is based on the ALU component created in Lab #9. The ALU performs operations on the 8-bit inputs, A and B, and outputs the 8-bit result. The possible operations include addition, subtraction, bitwise AND, bitwise OR, output A, and output B. The ALU should use Input B from the Accumulator and Input A from the MDRI.

The ALU is created by invoking an instance of the component "alu" (port-mapping). Make sure to have the alu project file created, to be able to import the alu.vhd code later on to be used in the port-mapping process.



```
architecture behavior of alu is
begin

process(A,B,AluOp)
begin
        if(AluOp="000") then output<=(A+B);
        elsif(AluOp="001") then output<=(A-B);
        elsif(AluOp="010") then output<=(A and B);
        elsif(AluOp="011") then output<= (A or B);
        elsif(AluOp="100") then output<= B;
        elsif(AluOp="101") then output<= A;
        end if;

end process;
end behavior;
```

Figure 12: ALU Behavioral Description

Program Counter (PC)

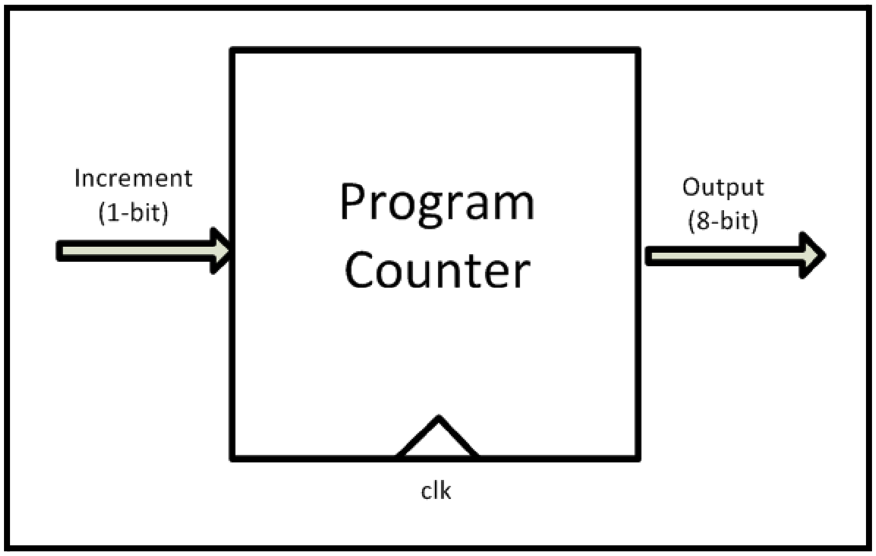


Figure 13: Program Counter Entity (VHDL)

The program counter (PC) is a special register that holds the 8-bit memory address of the **next instruction to be executed**. When a program begins, the memory address of the first instruction is stored in the PC. The CPU "fetches" the instruction from memory based on the contents of the PC. After the first instruction has been executed, the PC is incremented by one to point to the next instruction. To control this, the PC register has one input labeled "INCREMENT". When INCREMENT is set to one, the PC value is incremented by one. When INCREMENT is set to zero, the PC value does not change.

When creating a program that will run on the CPU, it will be necessary to place the instructions in memory sequentially. If instruction #1 is in memory location 0, then instruction #2 will need to be in memory location 1, instruction #3 in memory location 2, and so on. Be cautious when loading and storing data to prevent instructions from being overwritten.

<u>It is important to note that the PC contains an 8-bit memory address, but the memory component for this project is limited to 5-bit addresses. Keep this in mind when fetching the instruction from memory.</u>

Accumulator (A)

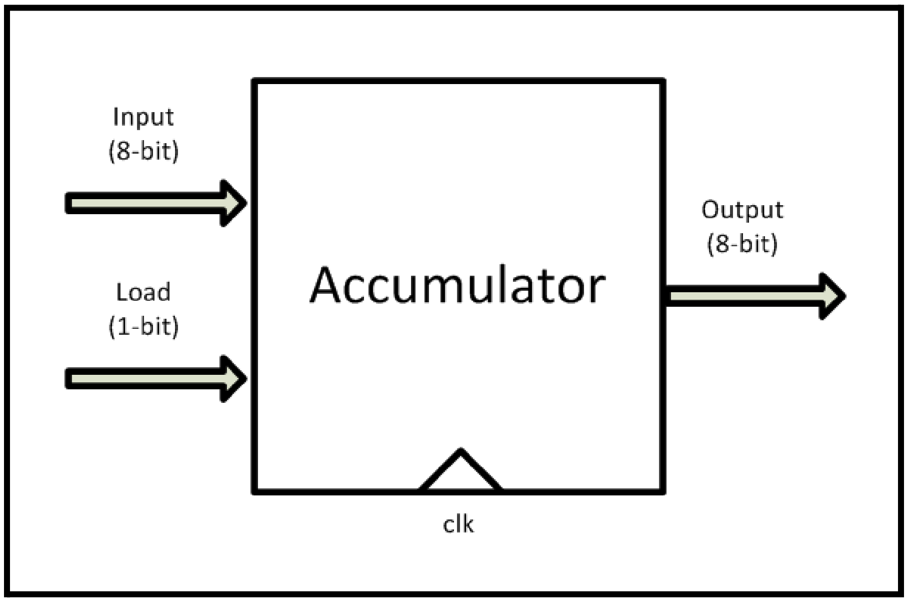


Figure 14: Accumulator Component (VHDL)

The Accumulator is a register that contains the 8-bit output of the ALU. Figure 6 shows that the output of the Accumulator serves as an input for the ALU, forming a feedback loop.

The "Load" input, in conjunction with the clock input, signifies when the Accumulator will latch its input to its output. "Load" performs the same role as the clock input of a D Flip-Flop, in that a high signal will set the output equal to the input, and a low signal will keep the output the same regardless of what happens at the input. The control unit controls the value of "Load".

The Accumulator is created by invoking an instance of the "reg" component (port-mapping).

```
architecture behavior of reg is
begin

process(clk,load)
begin
     if (clk'event and clk = '1' and load = '1') then
          output <= input;
     end if;
end process;
end behavior;
```

Figure 15: Register Behavioral Description

Memory Address Register (MAR)

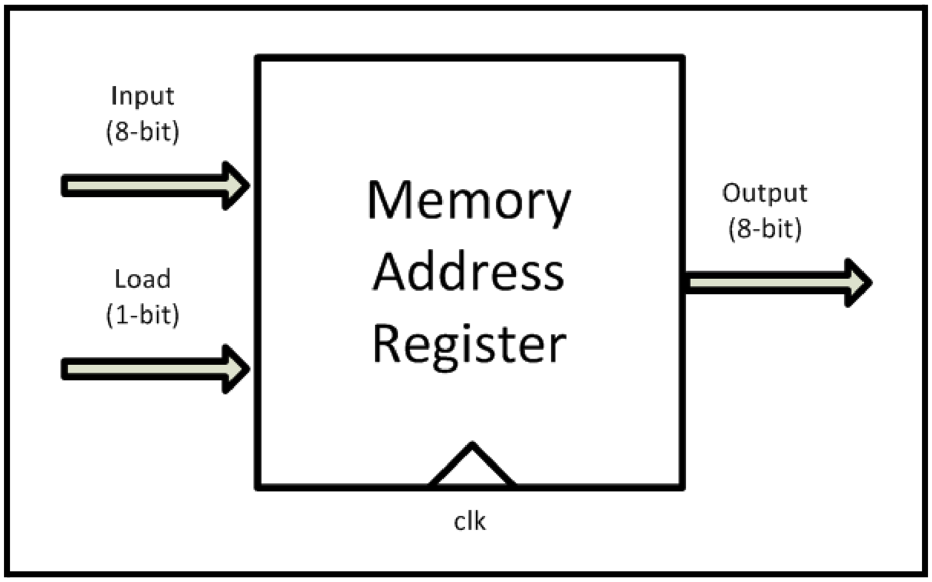


Figure 16: Memory Address Register Component (VHDL)

The Memory Address Register (MAR) is an 8-bit register that stores the value of a memory location. This memory location can be a location to retrieve data from, or a location to store data. Both the program counter and the instruction register need to access the MAR to read and write to memory. To handle both inputs, a multiplexer (TwoToOneMux) is placed in front of the MAR. The control unit determines which component has access to the MAR by controlling the "address" input. An "address" of '1' should route the IR to the to the MAR. An "address" of '0' should route the PC to the MAR.
The MAR is created in the same manner as the Accumulator by invoking an instance of the "reg" component. The multiplexor is based on the code below:

```
entity TwoToOneMux is
port (
        A : in std_logic_vector (7 downto 0);
        B : in std_logic_vector (7 downto 0);
        address : in std_logic;
        output : out std_logic_vector (7 downto 0)
);
end;

architecture behavior of TwoToOneMux is
begin

process(A,B,address)
begin
        if (address='0') then
        output <= A;
        elsif(address='1') then
        output <= B;
        end if;
end process;
end behavior;
```

Figure 17: MAR Multiplexor Entity and Behavioral Description (VHDL)

Memory Data Register: Input & Output (MDRI & MDRO)

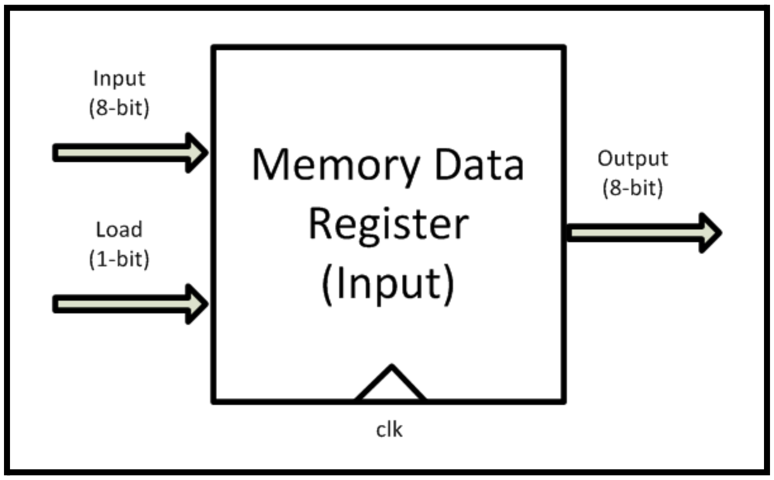


Figure 18: Memory Data Register Input Component (VHDL)

The Memory Data Register Input (MDRI) is an 8-bit register that holds a value read from memory. When an instruction reads from memory, this is where the result will temporarily be stored before being passed along to component that will use it.

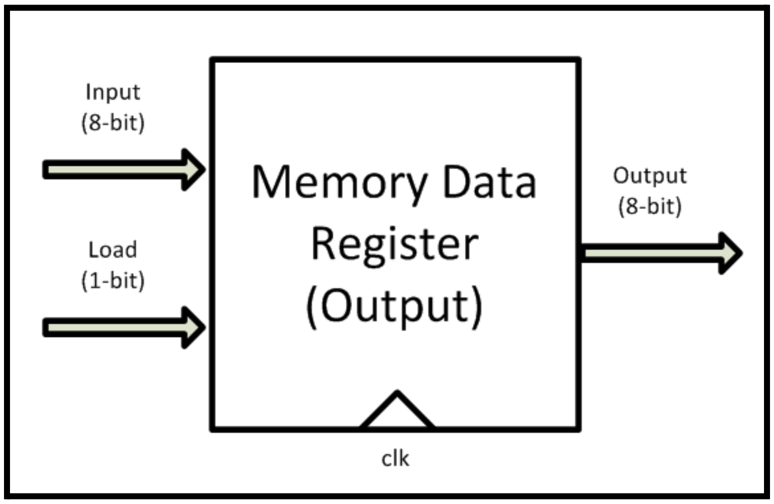


Figure 19: Memory Data Register Output Component (VHDL)

The Memory Data Register Output (MDRO) is an 8-bit register that holds a value that is to be written to memory. When an instruction writes to memory, this is where the value will be stored before being stored in memory.

Both the MDRI and the MDRO are created by invoking an instance of the "reg" component (port-mapping).

Instruction Register (IR)

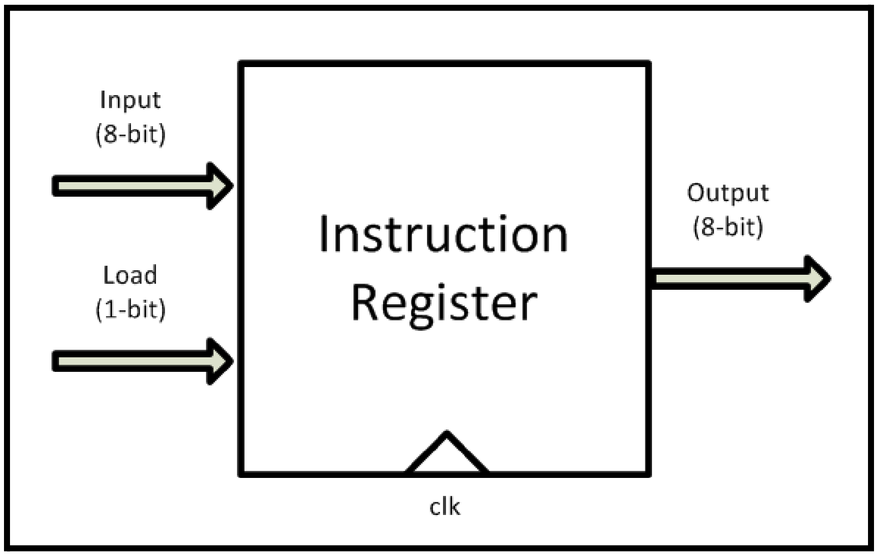


Figure 20: Instruction Register Component (VHDL)

The Instruction Register (IR) is an 8-bit register that contains the instruction that is currently being executed. The format of the instructions is described in Figure 9. The IR is created by invoking an instance of the “reg”/register component (port-mapping).

Control Unit (CU)

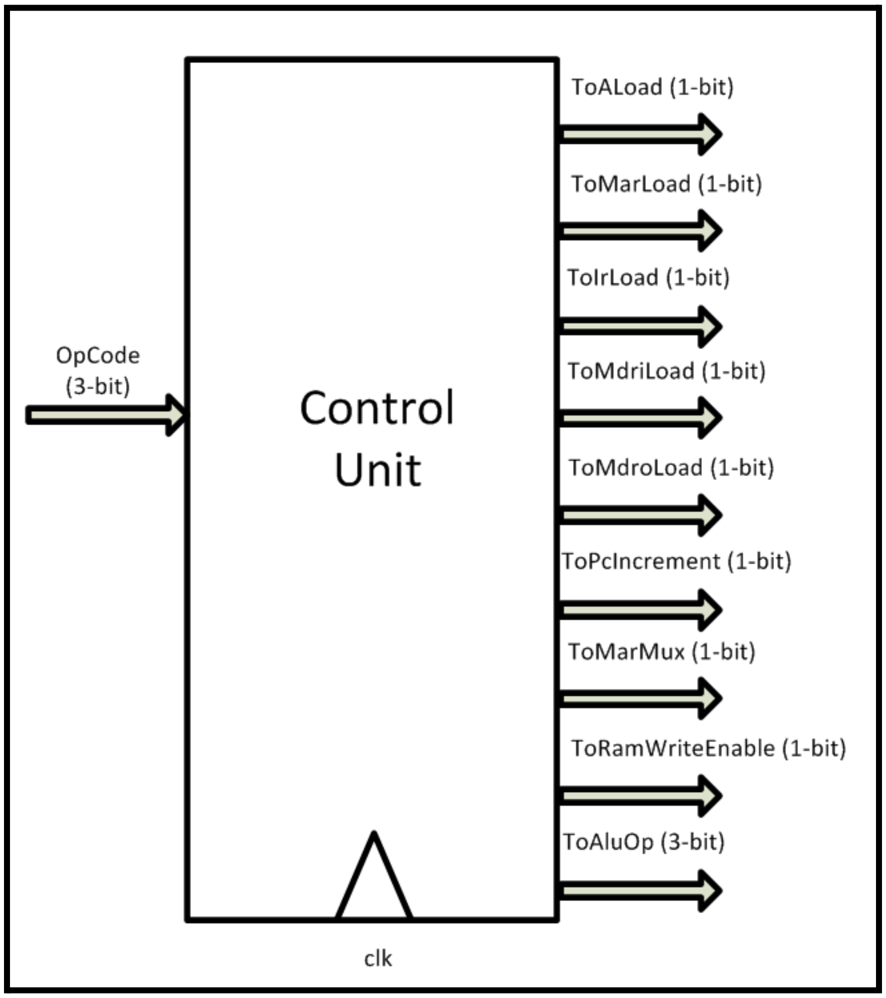


Figure 21: Control Unit Entity (VHDL)

The Control Unit (CU) coordinates all the operations of the CPU, including incrementing the PC, accessing memory/registers, and ALU operations. **The CU can be modeled as a state machine in which each step of an instruction represents a state.** For example, fetching an instruction from memory takes five steps (clock cycles), so there will be five state transitions involved in the fetch process. At each state, the control unit must define which registers are active and what the next state will be.

Instruction Fetch

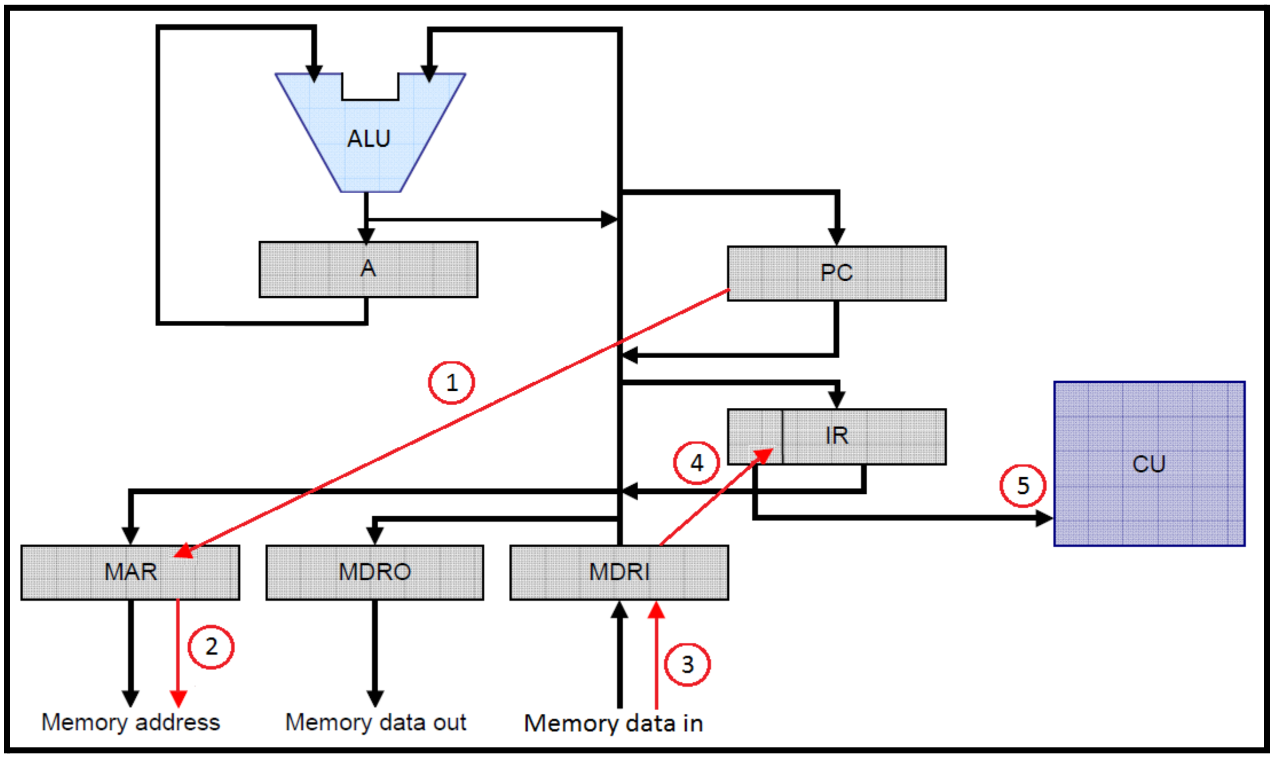


Figure 22: Instruction Fetch Sequence

To execute a program instruction, the CPU must first fetch the instruction from memory. The following steps are taken to fetch an instruction:

1) The program counter contains the address of the next instruction, load that address into MAR.

2) A memory read is performed at the address loaded in MAR.

3) The result of the memory read is loaded into MDRI.

4) The value held in MDRI is loaded into IR.

5) The instruction held in IR is decoded.

After the instruction has been decoded, the CPU can then begin executing the instruction. Remember, for this project there are only three possible instructions that can be performed: LOADA, ADDA, STOREA. The Control Unit’s next state after decoding will depend on which of these instructions is to be performed.

LOADA Instruction

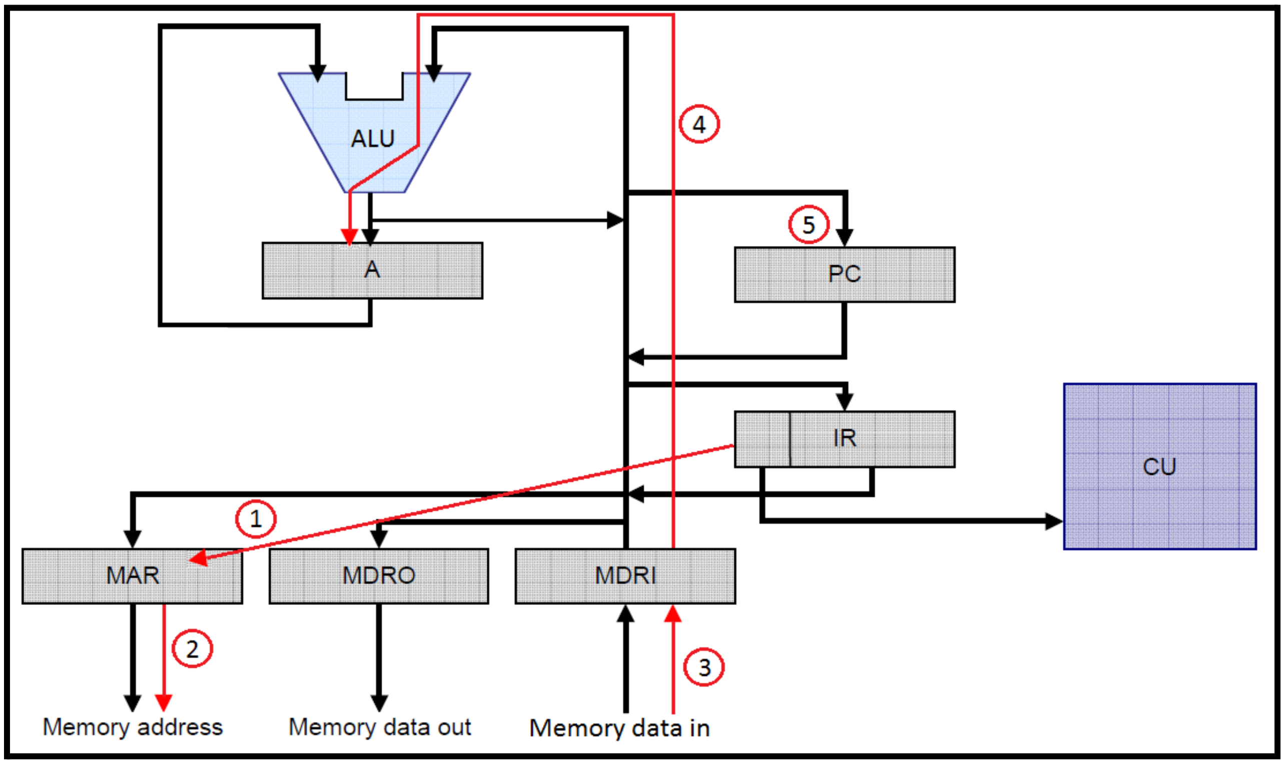


Figure 23: LOADA Instruction Sequence

The LOADA instruction loads the value from the specified memory location into the Accumulator (A). The following steps are performed to complete this instruction:

1) Load MAR with the lower 5-bits of IR
2) A memory read is performed at the address loaded in MAR.
3) The result of the memory read is loaded into MDRI.
4) Load the accumulator with the contents of MDRI.
5) Increment PC

The correct ALU op code will need to be set by the Control Unit in order to load the appropriate ALU input into the accumulator.

ADDA Instruction

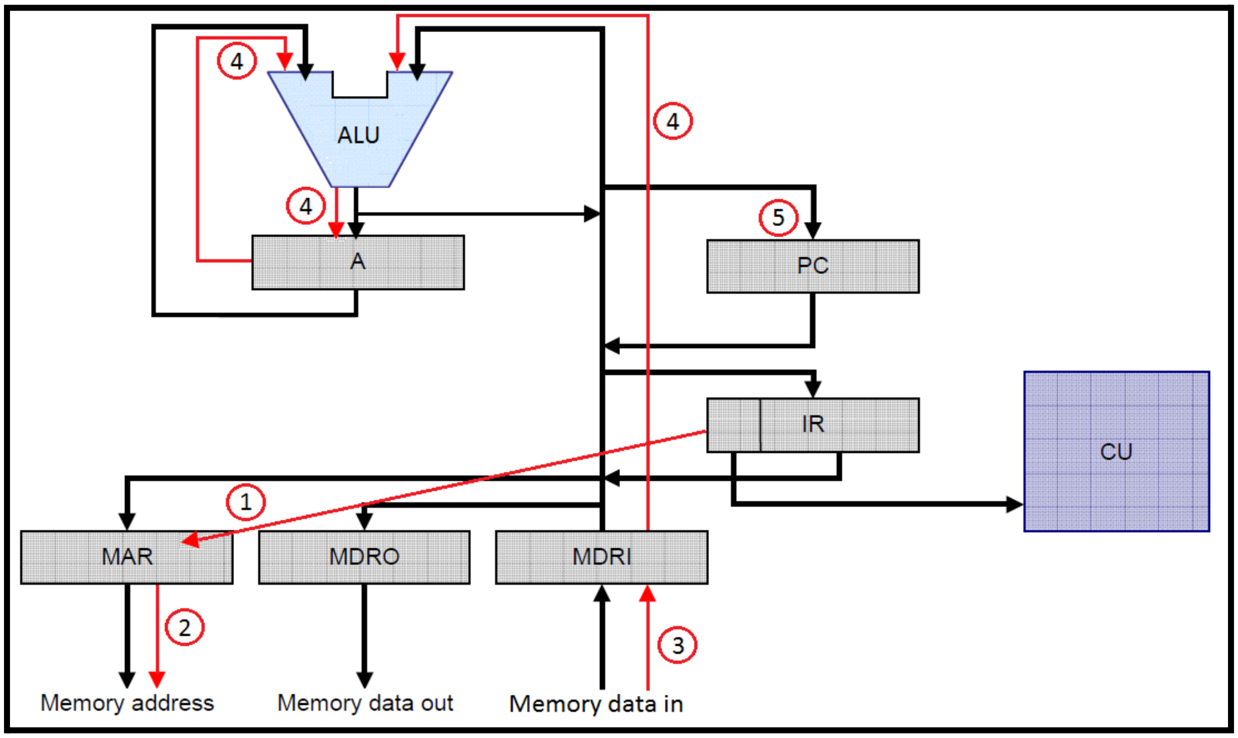


Figure 24: ADDA Instruction Sequence

The ADDA instruction adds the value from the specified memory location to the value in the Accumulator. The result is stored in the Accumulator. The following steps are performed to complete this instruction:

1) Load MAR with the lower 5-bits of IR

2) A memory read is performed at the address loaded in MAR.

3) The result of the memory read is loaded into MDRI.

4) Perform addition of MDRI contents and accumulator contents and store the result in the accumulator.

5) Increment PC

STOREA Instruction

The STOREA Instruction stores the contents of the accumulator in the specified memory location. As part of the final project, students will need to determine the steps required to complete this instruction.

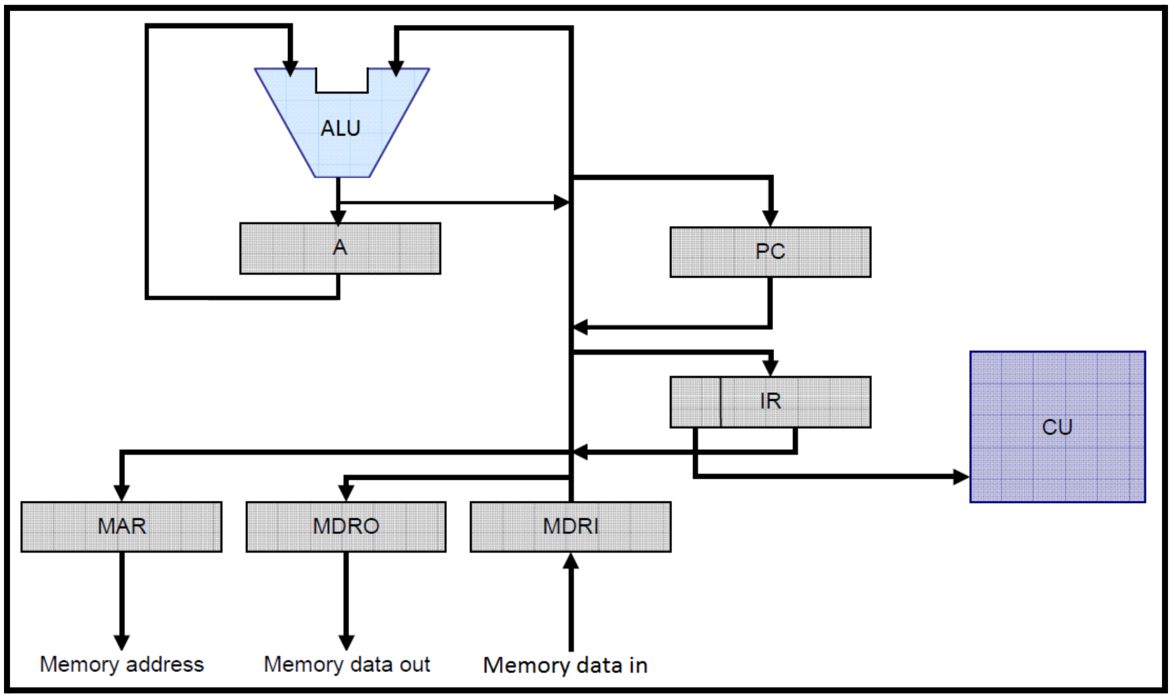


Figure 25: STOREA Instruction Sequence (FILL IN!)

Steps (FILL IN!):

6) Load MAR with the lower 5-bits of IR

7) A memory read is performed at the address loaded in MAR.

8) The result of the memory read is loaded into MDRI.

9) Perform addition of MDRI contents and accumulator contents and store the result in the accumulator.

10) Increment PC

Tasks

Due to the complexity of this project, a large portion of the code has been provided for you and you will be tasked with filling out some one-liners, creating the components, importing/port-mapping, assigning values to certain variables, etc...

Take your time reading through the handout and studying the provided code.

Make sure to read over this handout **several times**. Twice at the minimum to start piecing it all together.

Critical areas of the provided code are incomplete, and it will be your task to fill in these areas.

Tasks to complete (There's a simple guide of how I solved the problem right after the tasks):

1) Complete the behavioral description of the Program Counter.

2) Create a complete diagram of the CPU, including all components and connections preferably labeled the same as the variables declared in the VHDL files. Include that in your report.

3) Define the Control Unit state transitions as well as the corresponding output for each state (Have to define sequence for STOREA first on page 18). Complete the general finite state machine model given in the PowerPoint file (slide #6 is your template, slides #7-10 explain what each FSM transitions refer to). Include the completed FSM in your report.

4) Instantiate all the CPU components through port-mapping and create the appropriate connections. Drawing the CPU diagram beforehand will make this significantly easier!

5) Use the waveform simulator to demonstrate functionality of all inputs/outputs as shown on page 24.

6) Fill out the table provided in the Project_Demo_Handout (the flow is explained in slides 11-18 of the CPU Final Notes PowerPoint.

<u>How I went about it:</u>

For Task 1: Program Counter

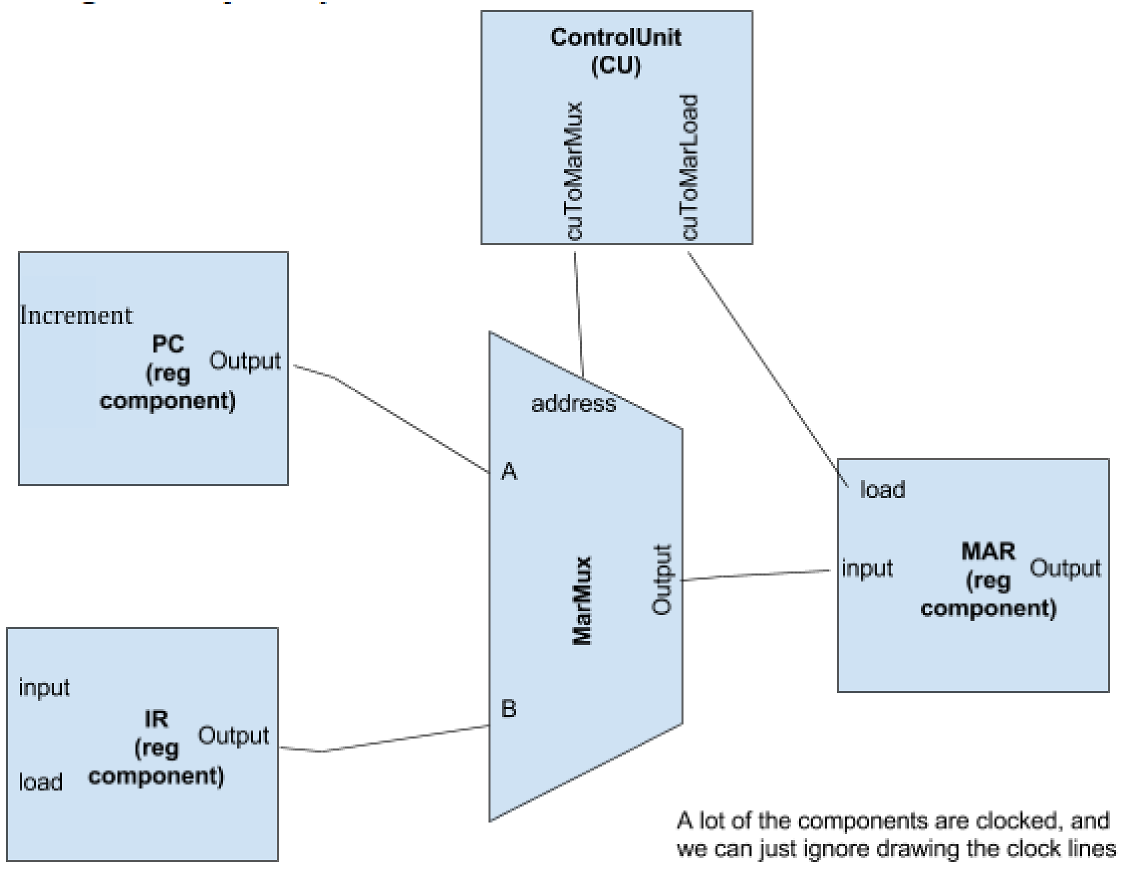
- Use the code template to create the .vhd file.
- Have the PC vhd file and the PC write-up in the handout side-by-side to familiarize yourself and gain a connection between the two. Transcribe the pseudo code in the vhd file to actual VHDL code.
    - hint: When you must write the 3 lines of code here, note that there is a *new* data type to remember about (like signal, in, out, inout) called **variable**. You need this keyword to declare your counter (similar as in the "clock divider" example in one of the labs). Also when you are instantiating its variable or changing its value, you must use the **:=** operator instead of **<=**.

For Task 2:

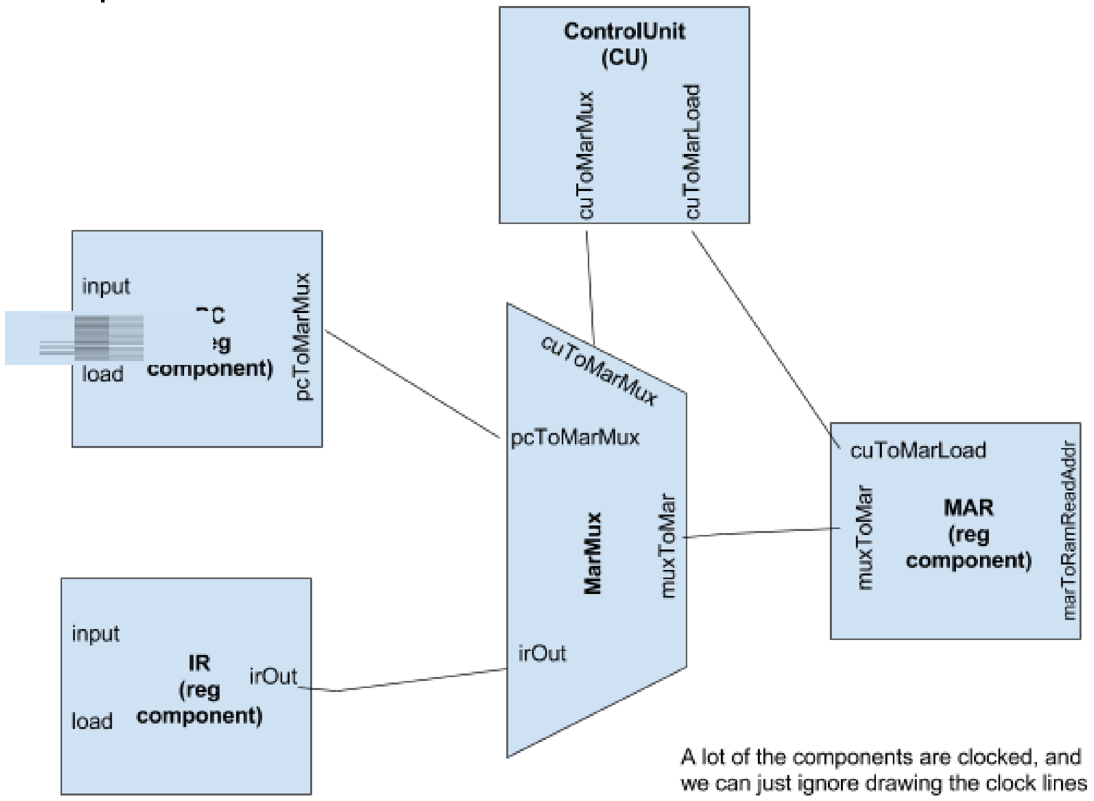
Begin by going through the Code template and create a VHDL file for each file given.

- Create block diagrams for each component in the handout and connect them altogether to create your simple CPU. Take your time on this part. Use MS Visio, Paint, Word, or any software tool to make this as neat as possible.
- You will need to match components (MAR, Accumulator, ControlUnit, MDRI, etc.) up to files you just created.
- Some of the components will use the same code multiple times.
    - We have seen this before. Re-using code was done with VHDL Components and port mapping.
- I would suggest creating diagrams in Visio, but PowerPoint also works.
- Furthermore, when you create block diagrams, you will want to label their Inputs and Outputs. Eventually, you will connect those inputs and outputs to other blocks/components. I would suggest labeling them relative to the block the wire/connection is coming from/going to. Here is an example below:

If you have read the section about the MAR, you'll know that two components are MUXed to its input. Okay first, let us break down what this means in terms of drawing something up from a word problem. Let us take inventory of all the components involved, then let us connect them up and label their lines (BTW, note the re-use of the register component).



Now, we could call it a day for this drawing, but labeling our input/outputs with generic labels like “input” and “A” will not help us out on some of the other things we need to do later. Therefore, we should label them relative to their function and connections to other blocks. See the revised version below and note how the labels have changed to be more descriptive. The new label names were sourced from the signal names in the Control Unit code.



The reason for doing this is that later on in Task 3, you'll need to set opcodes (single-bit variables for the control unit) and a fully mapped out block diagram along with descriptive names to make the process a whole lot simpler. In general, you will find that most of the labels are relative to the control register. Just look at the different opcode bits in Figure 21 and the ALU opcodes in Figure 12.

For Task 3:

The first thing that is asked of you is to "define the Control Unit state transitions as well as the corresponding output for each state." Okay, so one thing at a time here. This is a little bit of a loaded statement.

There are a couple steps to complete on paper before going to code, and they are the following:

1) Write in English (i.e. not code) the steps for the Control Unit to accomplish its four different tasks it facilitates in performing: fetch, store, load, add.
    a) Luckily, Fetch, Load, and Add are already done for you below Figures 22-24.
    b) Finish the Store command steps in English
2) Correspond each step that is written in plain English to a specific state. And by state, I mean finite state machine state just like we did in previous labs 8 & 9.

a) You might be asking, how am I supposed to come up with these states? Well, no worries, they have already been given to us in the code, we just have to look for them.

b) Where should I look for the states that have been given for me?

   i) We are talking about the Control Unit here, so you are going to want to look there.

   ii) Before going to look, do you know what you are looking for? Remember from Lab 9, when we defined all our possible states in an enumerated type? Well, the same goes for this project. Write down/list out every enumerated type and begin assigning the enumerated state types to the aforementioned steps we wrote in plain English.

3) Now take a step back and analyze what you have transcribed on paper. This is an appropriate time to finish the finite state machine given to you in PowerPoint. Use your transcribed states that you have on paper to corresponding states in the finite state machine (FSM). Some are already given to you to get you started. Note, you do not have to list the inputs/outputs for the arrows like the Mealy state machine. We are just looking at how the simple CPU operates as a whole.

4) Transcribe these FSM states into sequential VHDL code. You will need to do this for each instruction of Fetch, Load, Add, Store. You have already done the thinking part of matching the word problem step to the state, now just place it in VHDL code.

   a) Begin with the Fetch instruction/command because of the way the code flows in the code template.

   b) These are just simple case statements to assign your current_state to the states you wrote out on paper for the plain-English steps. There is nothing else that goes here other than assigning states.

   c) Note, there is a state decode (which occurs after fetch) that you will have to program to determine which instruction to execute (i.e Store, Load, or add). What do you need to look for to determine which instruction to assign to the state? Refer to Figure 9 and Figure 10 to answer this question.

5) You must now define the corresponding output for each state. What outputs are they really referring to here? They are generally referring to what bits need to go HIGH/LOW to perform certain operations. If you look in the Control Unit, the first OpCode is given to you to increment the PC. You'll note that ToPCIncrement is HIGH, which makes sense since we want to enable that functionality. Additionally, how can we add "+1" to something? We need to do an additional operation. Which component performs addition? The ALU is the only component that performs addition; therefore, we need to include the ALU OpCode that performs addition along with the other control unit bits. You won't always need to include ALU with the usual set of Control Unit bits unless you are performing one of the ALU operations.

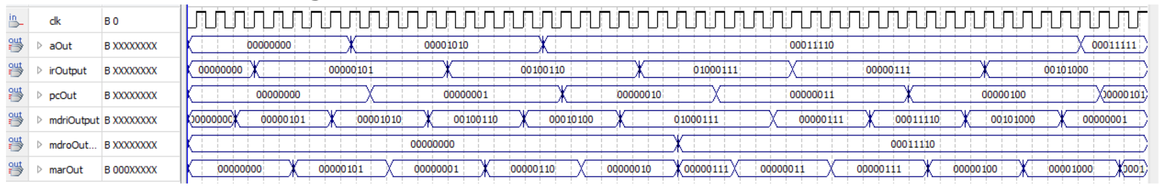
a) So, what is the best way to go about defining the Enable/Load bits for the rest of the states? Probably listing out every control unit bit that was given to us in the ToPCIncrement example and writing out the corresponding operations.
   i) E.g. Instruction Register → cuToirLoad
      (1) '1' for load in current instruction
      (2) '0' to hold and do nothing
   ii) Rely on their individual descriptions in the handout to make this determination.
   iii) Furthermore, I'm going to give you some of the codes that I believe aren't clear or well-defined in this handout:
      (1) Reading from memory is all '0's for the OpCodes.

For Task 4:

1) Open the Code Template CPU document that has all the template code for all the different components.
2) Copy/paste each block into a new project wizard file (make sure they're all in separate folders), fill out any missing lines of VHDL code (I mention "—INSERT CODE HERE" wherever more code is needed).
3) Compile each file
4) Create a top-level project for the simpleCPU_Template entity and paste all .VHD files there to be able to utilize the components pre-loaded for you in the code with port-mapping statements. <= This is the file you can eventually run a waveform simulation on and get something like what I screenshotted below.

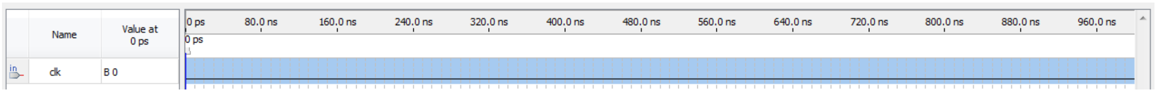
What is convenient about using the waveform simulator is that you can see ALL your signals/variables in real time. For example, I can see MarOut, MdriOutput. You can also notice that the clk is the only driving thing for this circuit; as soon as the "CPU" is powered on, it starts fetching instructions from RAM, decoding them, and performing whatever each instruction is designed to do. No external, physical inputs must be provided, it should run "on its own".

This is a 100% working simulation:

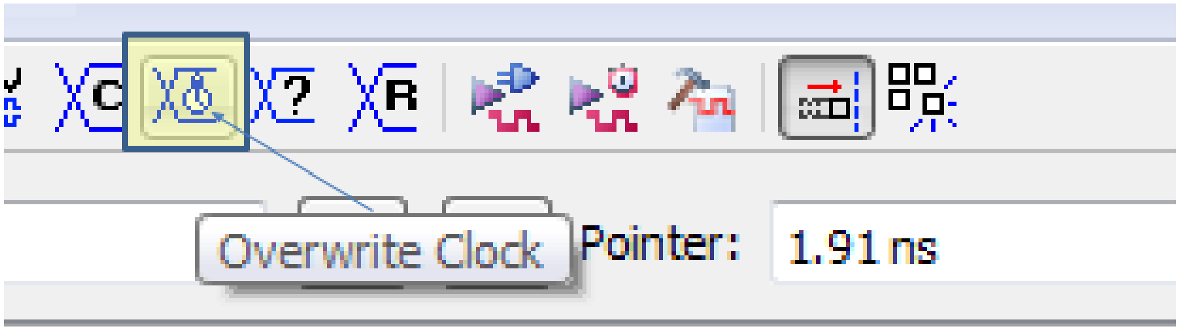

clk B 0
aOut B XXXXXXXX 00000000 00001010 00011110 00011111
irOutput B XXXXXXXX 00000000 00000101 00100110 01000111 00000111 00101000
pcOut B XXXXXXXX 00000000 00000001 00000010 00000011 00000100 00000101
mdriOutput B XXXXXXXX 00000000 00000101 00001010 00100110 00010100 01000111 00000111 00011110 00101000 00000001
mdroOut... B XXXXXXXX 00000000 00011110
marOut B 000XXXXX 00000000 00000101 00000001 00000110 00000010 00000111 00000011 00000111 00000100 00001000 0001

<u>The only new feature that was not introduced in the waveform simulator lab was how to create a clock signal like the one above. Here are the directions to create the clock signal.</u>

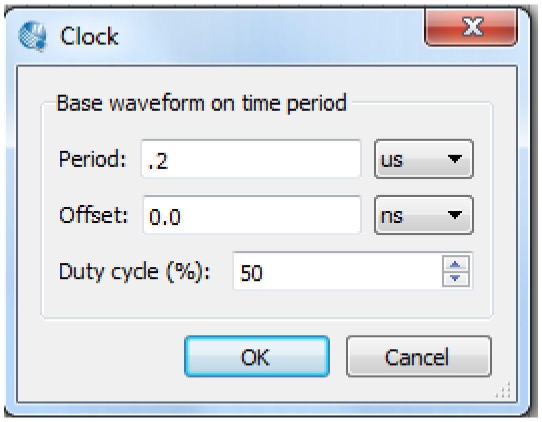
Step 1) Select the whole driving-signal by double-clicking it



Step 2) While selected, choose the “Overwrite Clock” option in the menu above



Step 3) Specify the clock period. Take your end-time (mine was 10us), divide it by 100, and multiply it by 2. That is why mine is 0.2us. Select OK



Clock

Base waveform on time period

Period: .2 us

Offset: 0.0 ns

Duty cycle (%): 50

And you are done!