CSCE 312 Lab 6 manual

(Final Project)

Instructor: Dr. Eun Jung Kim

Spring 2024

Department of Computer Science & Engineering
Texas A&M University

Chapter 6: Processor internals and program execution

In the lab-4 you learnt the basics of designing circuits that sit around a microprocessor. These peripheral circuits connect the processor to the rest of the system and thereby allow the processor to do something useful in conjunction with other sub-systems (like memory, IO, etc.). In this chapter you will learn what really happens inside a microprocessor, by actually designing a very simple processor yourself. In doing that you will also learn how programs actually execute inside processors. Processor internals are commonly termed as “computer architecture” where as “computer organization” deals with the sub-systems that are external to the processor. However these technology domains are very much interrelated, therefore to develop a comprehensive understanding in each of these, it is necessary to know both of these areas sufficiently.

<u>Basic processor operations</u>: To understand how a processor works, we should observe Jack, who is a diligent and obedient worker (Fig. 1). Jack is provided with a list of tasks that he should do. Jack looks at each task in the list, does the work in a way exactly as mentioned in the list. After completion, he delivers the work, and then goes back to the list to check the next task that he has to do. Jack does this cycle till he completes all the tasks in the given list.

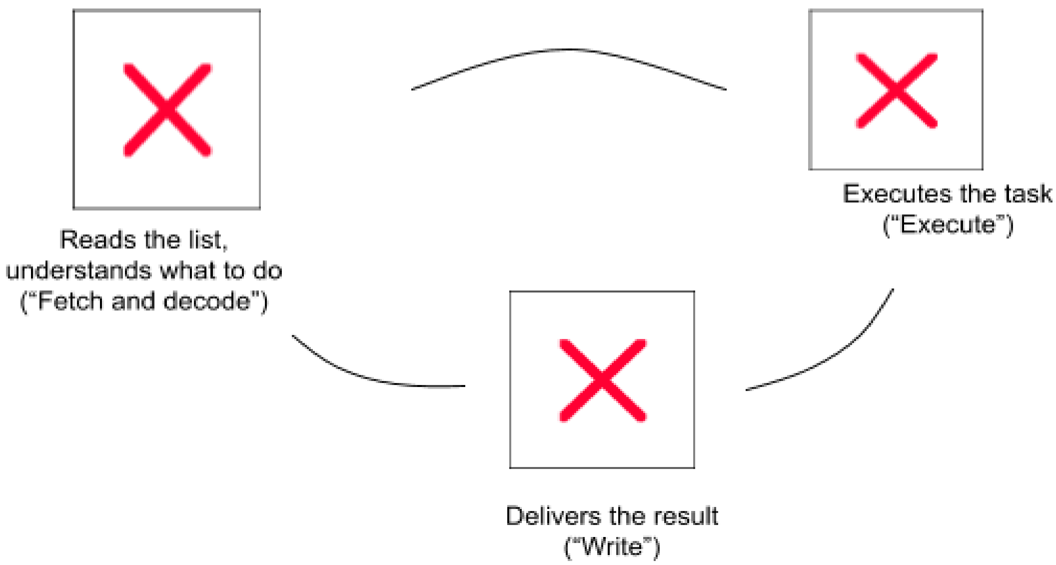

Fig. 1: The story about Jack: the fetch-decode-execute-write cycle inside the processor

Quite some time back, even before when geeks kept sideburns, (i.e. 1960s), processor designers had decided that processor circuits should work the same way as Jack. For example to execute the following “C” statement –

a = b + c+ d;

the processor would execute the following operations in sequence, one operation at a time –

Opr. 1: Get the value from memory location that corresponds to variable “b”
Opr. 2: Get the value from memory location that corresponds to “c”
Opr. 3: Add these two values
Opr. 4: Store the intermediate result to the location corresponding to “a”.
Opr. 5: Get the value from memory location that corresponds to “d”
Opr. 6: Add the intermediate result at “a” with the value retrieved from “d”
Opr. 7: Store the final result to memory location “a”

For every “C” variable a certain memory location/address is reserved. Later you will learn who does this and how this is achieved. These operations (Opr. 1 to 7) belong to either one of the four standard class of operations: fetch (Opr. 1, 2), decode (not shown as a task), execute (Opr. 3) and write (Opr. 7). All computations inside a computer are performed using these four basic primitive class of operations. The fetch, execute and write operations are easy to understand. The decode operation is explained in subsequent paragraphs.

Why processor needs memory? : Before we attempt to understand the decode operation, we need to appreciate the multiple role of memory. When Jack is working he needs to remember the small details of the present work, he needs a work bench to hold his incomplete work, he also needs sheets of paper to hold his task list. Similarly the processor needs memory to store input data, intermediate and final results.

Stored program “Von Neumann” computer: The list of tasks or operations is also stored in the memory. This sort of computer system is known as “stored program computer” where the same memory is used to store the input, output, intermediate data and also the task list, as opposed to the calculator, where the user has to define every task using the push buttons. The advantage of this “stored program computer” is that it can literally be the “Jack of all trades”. This computer can be reused to do a variety of work simply by storing a different list of task in the memory for each variety of work. Moreover this allows the stored program computer to do more complicated tasks compared to the calculator. This scheme of processor operation was conceived by Von Neumann, and most of the modern processors today are derived from this kind of “Von Neumann architecture” or scheme of operation. The operation of this architecture (Fig. 2) is explained over subsequent paragraphs.

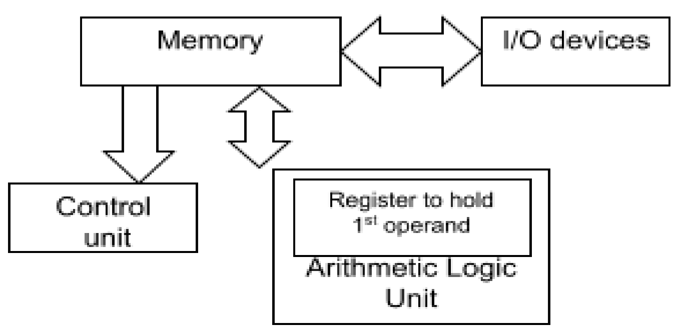

Fig. 2: Von Neumann stored program architecture

Instruction, instruction set, ISA, code, application: The trick behind successful designing of a generic hardware to a variety of applications, depends on two things. First, a standard set of primitive operations has to be decided, and then a right kind of task list has to be concocted which when executed will finally deliver a specific machine behavior. A couple of low level tasks/operations, like Opr.1 to 4 in the previous example, are packed into a binary encoded format which is known as “instruction” (analogous to each task item in Jack’s list). An instruction word will be “w” bit wide, where “w” can be anywhere between 8 to 64 bits (Fig. 3).

Fig. 3: A 16-bit instruction word

A single instruction may be decoded (broken down) to generate a list of few memory read/write and ALU operations. The set of standard instructions is called as “instruction set” or “instruction set architecture (ISA)”. This is because a certain processor architecture design is considered to be defined by the given set of instructions. By “Micro-architecture” we mean the actual architecture of the processor internals which implements the given instruction set. Whereas, the art of constructing the instruction list is known as coding/ programming. An application is actually a long list of instructions.

Instruction and data memory: For stored program computer operation, the common memory address space is partitioned into two parts – “instruction space” and “data space”. An example is shown in Fig. 4. The proportion of partitioning is arbitrary, but often based on system requirement and design outlook.

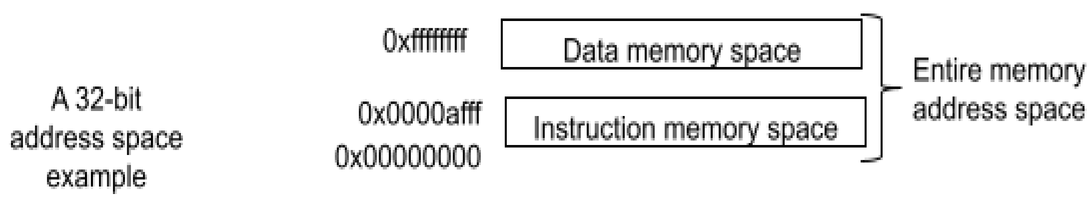

Fig. 4: Instruction and data memory address space partitions in a stored program computer

The processor fetches instructions from the “instruction memory” one by one, decodes and executes them. Some of the operations, like Opr. 1,2,5 in the previous example, will require that the processor also fetch data from the “data memory” and write results back into the data memory (Opr. 7) as a part of the execution operation. Quite often computers are dedicated to executing only one application therefore it will have only one single list of instructions in the instruction memory for its entire life time. An example is your mobile phone, which is a dedicated embedded system in contrast to your multi-purpose desk top computer. In dedicated systems, the instruction memory might be a ROM, because the computer has to only read from the instruction memory, whereas the data memory would be a RAM so that data can be both read from and written into it.

Instruction, operation, clock cycles, multi and single cycle processors: Execution of an entire instruction will require several cycles of operations and clock pulses (Fig.5).

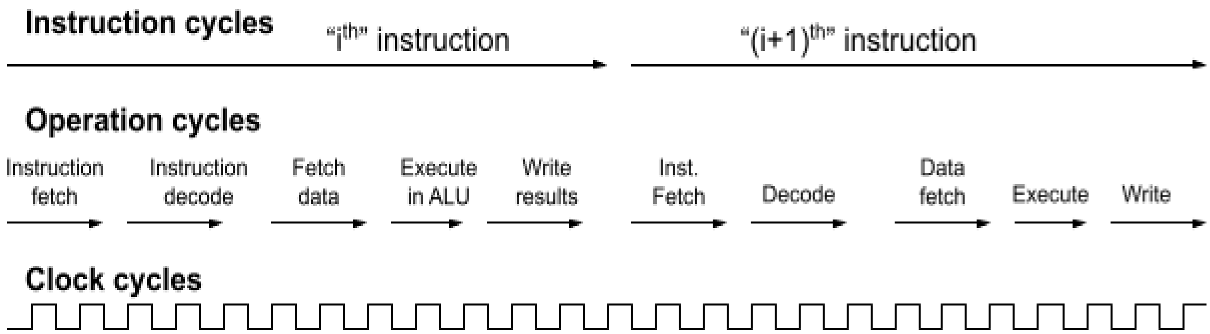

Fig. 5: Relationship between instruction, operations, and clock cycles in a multi-cycle processor

On the other hand, a single operation cycle may require several clock cycles to complete. For example, a single memory read/write or floating point ALU or integer multiplication/division operation require multiple clock cycles. The multiple operation and clock cycles which materialize execution of a single instruction, together are called as an instruction cycle. Therefore, an instruction cycle actually means an entire set of fetch-decode-execute-write operation cycle. A simplistic processor may carry out these operations one at a time, and therefore it might need multiple operational cycles to implement one instruction cycle, in that case it is called as a "multi-cycle processor". Whereas a more complex and advanced processor may overlap multiple operation cycles from different but consecutive instruction cycles, those are called single cycle processors. We will mostly discuss a basic multi-cycle processor in this course.

Overview of the processor circuitry: The Von Neumann processor has circuits to do three things: (1) to either fetch an instruction or a data from memory one at a time ("fetch" operation); (2) to interpret or understand the task description ("decode" operation); and (3) to carry out the task ("execution" operation).

The instruction fetch circuit: The processor has a counter called "program counter" (PC). The program counter points to the address location which contains the next instruction to execute. Therefore, in each instruction cycle, the program counter content is taken and place on the address bus to read the instruction memory location. Once the instruction is fetched, decoded and executed, the program counter is either incremented or set to a jump location based on the current instruction's execution operation. This prepares the processor for the next instruction fetch cycle. Setting the PC to a jump location is needed to implement the if-then-else, while/for loops, and switch statements, which are known as program control statements.

The decoder and control circuitry/unit: To appreciate the decoding operation you will need to understand bit level encoding of instructions. The "C' source code is compiled to generate a list of instructions in machine language (binary format), which is understood by the processor. During the stone age of computers (1940s and 50s), real life programs used to be written in binary format by people. To ease program comprehension and troubleshooting, people decided to represent the same code in more intelligible form with English literals, which became known as assembly code. Each assembly code statement has a corresponding binary form (machine language) representation, so conversion between them is straight forward. But generation of machine format code from "C" source code is not always simple. For example, a single "C" statement -

$$a = b + c + d;$$

will be compiled to two assembly code or binary format (which are not Y86) instructions as below -

Assembly	Meaning	Machine/Binary form
addl b, c, a	get value from memory "b" (address = 0x2) and "c" (address= 0x3), add them and store the result to location "a" (address = 0x4)	0010 0010 0011 0100
addl d, a, a	Get "d" (address = 0x1), add with "a" and store result in "a"	0010 0001 0100 0100

A much simpler assembly and machine language syntax is introduced here instead of more sophisticated Y86, so that you could design processor hardware for it. Once you get comfortable with the simpler schemes, we will explain why more sophisticated ISA, assembly or binary format instructions are required at present. After that we will explain how and where Y86 fits in to what we are discussing, and in what way Y86 is much more sophisticated. Now let's continue with the simple scheme which we were discussing.

Each machine language statement is represented with a "w" bit wide word. Each of the bits of this instruction word has specific meaning. For an example 16-bit computer (w =16), the binary instruction word may have the following form (bits 0 to 15 are shown as B0 to B15):

<table>
<tr><td>B15</td><td>B14</td><td>B13</td><td>B12</td><td>B11</td><td>B10</td><td>B9</td><td>B8</td><td>B7</td><td>B6</td><td>B5</td><td>B4</td><td>B3</td><td>B2</td><td>B1</td><td>B0</td></tr>
<tr><td colspan="4">(Instruction type field)
Defines the operations</td><td colspan="4">(Operand 1 field)
Defines address for operand 1 (direct addressing)</td><td colspan="4">(Operand 2 field)
Address for operand 2 (direct addressing)</td><td colspan="4">(Result/Destination field)
Defines destination address to store results</td></tr>
<tr><td colspan="4">Example -
0010 – Add
0011 – Subtract
0100 – Shift left</td><td colspan="4">Limitations –
The 4-bit address space only supports 16 locations, not enough</td><td colspan="4">Limitations –
Inadequate memory addressing</td><td colspan="4">Limitations –
Inadequate memory addressing</td></tr>
</table>

In the above table B15 to B12 are used to define the type of operation, B15 to B12 = 0010 means that the instruction is an add instruction. For this example, the decoder circuitry reads this entire 16-bit word, it recognizes them (in this example) to be "add" instructions with two operands, and one destination location. Based on the 16 bits of the instruction word the decoder circuitry excites three control circuits (Fig. 6).

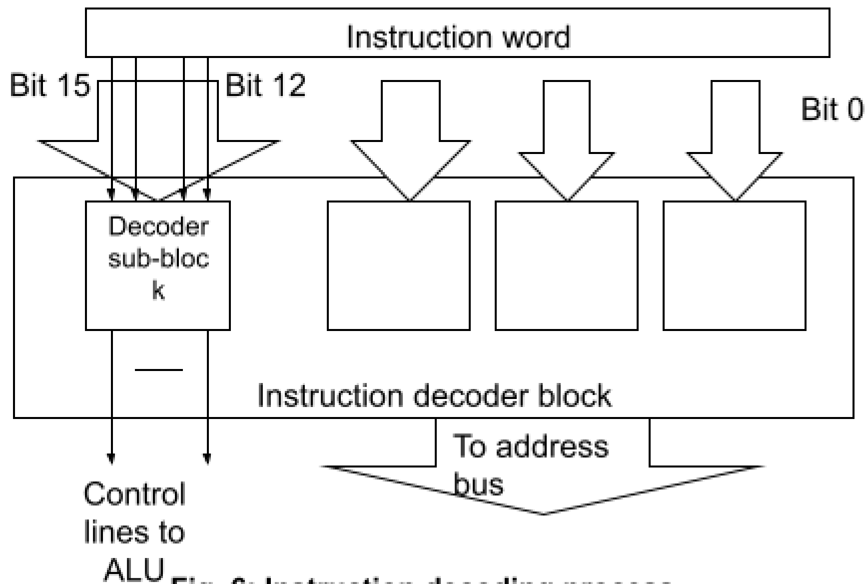

Fig. 6: Instruction decoding process

The first control circuit enables the fetch circuitry to fetch the values from two memory locations to get the operands. The second control circuit selectively enables the adder, subtractor, etc. of the arithmetic logic unit (ALU) to suitably execute the add/sub instructions. Finally, the third control circuit enables the data path to route the result from the ALU to the destination memory location.

ALU circuitry: The portion of the circuit which actually executes the task is the ALU. ALU does the arithmetic and logical operations like: add, subtract, bit shift, AND, OR, compare etc. To do this the ALU has to interact with the memory, where the data is stored.

Memory read/write circuitry: The bidirectional data bus, address bus, data and address routing mechanisms constitute this circuitry. Address is routed from PC or other address generating sources and placed on the address bus for memory read/write operations. For our simple processor, the decoder block with address registers acts as sources for data memory addresses when operand data is read from memory and result is written to it. However, for more advanced processor alternate address generators are used, which you will learn later. PC always acts as the address generator for instruction memory addressing. Normally there is one single address bus for both instruction and data memory addressing.

Instruction is routed from memory to the decoder block in the instruction fetch operation cycle. Operand data is routed from memory to the ALU in the data memory read cycle. The result data is routed from ALU to the memory in the data write cycle. A register holds the first operand in place while the second operand is fetched (Fig. 2). Generally, there is one single bidirectional data bus to do both read and write operations. Therefore, the data path utilizes lot of mux/demux to implement the routing mechanism.

The memory-ALU speed gap: Large RAM sizes require larger silicon die area, so they were traditionally kept outside the processor and integrated as separate IC chips (hope you have read something about device fabrication in the earlier chapters). The other rationale was that as RAM consumed more power, so for better heat dissipation (and cooling) it was packaged as a separate IC and kept away from the ALU, which is another source of heat. As the device and IC fabrication process became sophisticated, we realized that we were able to make faster ALUs but couldn't make faster and larger RAMs to match the speed of the ALUs. This gave rise to the ever-increasing speed gap between ALU and memory. The ALU operations took 1 to 20 clock cycles, whereas the memory read write operations took 100 to 400 clock cycles. The sluggish RAM was slowing down the entire system. This is because we can't build both faster and larger RAM at the same time due to engineering limitations. Moreover, the electrical bus lines, called interconnects, which connected the processor and RAM was another bottleneck. Actually, these electrical bus lines started radiating radio waves and lost power when those are driven any faster.

Registers: Therefore, to increase system speed, designers decided to build a small but faster memory inside processor. These memories could match the ALU speed. The rationale was that the processor seemed to reuse the same data in consecutive instruction cycles, so there was no need to write data all the way down the RAM and again bring it back from RAM to the processor for next instruction. These small memories are called registers. The ALU used them to store intermediate results which were used over consecutive instruction cycles. Typically, there are 8 to 128 registers in such a processor (Fig.7).

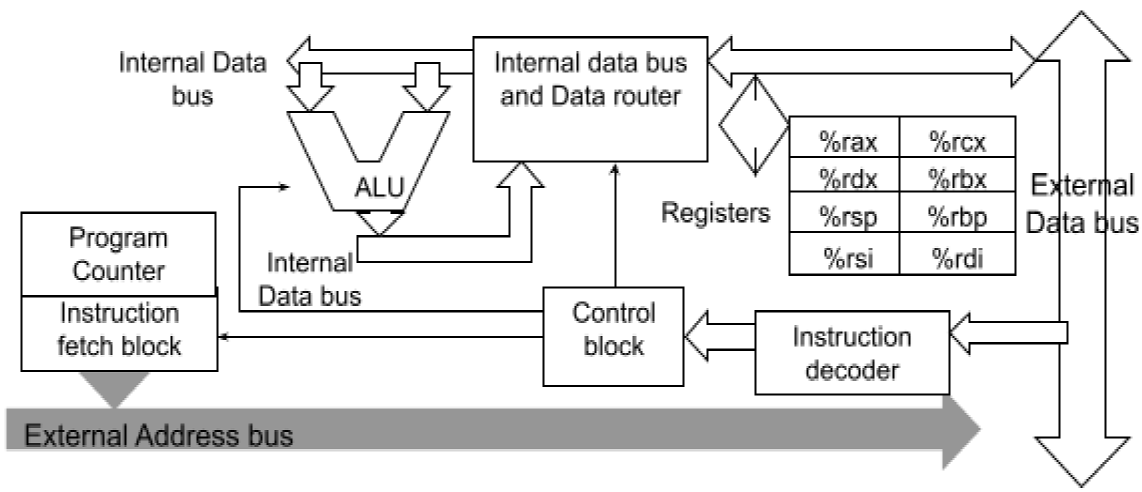

Fig. 7: A block diagram of simple multi-cycle load-store processor

In such a processor, operand data are first brought into the registers from memory ("Load" operation) before using them in the ALU. These registers feed operand data to the ALU. After execution ALU writes the result into one of the registers. Later the final result is transferred from register to the required memory ("Store" operation).

Now looking from another perspective, we can say that the ALU can interact with the data memory in different alternative ways. The most primitive way was the direct ALU-memory interaction (called memory-memory architecture), which we had started with. In that memory-memory architecture the ALU read operands from memory and directly wrote the results to the memory. Stack and accumulator-based processors were other two dominant historical processor types. Whereas having registers inside the processor and interacting with memory via registers is a modern design. Register based processors have two types of operations - register-memory or register-register/load-store operations. You can read more about all these alternatives from other sources (see the suggested readings of Chap 5). Register-register/load-store processor is the most modern version, so we will focus on the architecture, operations and assembly language for the load-store pattern.

Load-store assembly language: For this single "C" statement -

$$a = b + c + d;$$

the assembly code for a load-store architecture will look like (%rax to %rdx are the registers which are used) –

Assembly	Meaning
movq b, %rax	Load/move value from address "b" to %rax
movq c, %rcx	Load/move value from address "c" to %rax
movq d, %rdx	Load/move value from address "d" to %rax
addq %rax, %rcx, %rcx	Add values from %rax and %rcx and store result in %rcx
addq %rdx, %rcx, %rcx	Add values from %rdx and %rcx and store result in %rcx
movq %rcx, a	Store or move value from %rcx to address "a"

There is no specific technical reason that the registers have names %rax to %rdx. The above assembly languages are still simpler ones compared to Y86-64.

Memory addressing methods: The memory addressing method which specifies memory addresses in the instruction itself, is known as "immediate memory addressing" (refer the discussion under "The decoder and control circuitry/unit" sub-section). The disadvantage of this simple method is that only a small number of bits can be reserved in the instruction word to specify the memory address (4 bits in the shown example). This limits the range of address that can be used. Alternate forms of addressing methods are used in advanced processor which can use the entire "w" or more bits to specify the data memory locations. Those methods allow the processor to use larger memory spaces. You will learn these later when you get deeper into more sophisticated ISA like Y86-64.

Y86-64: Y86-64 ISA is a subset of x86-64 ISA. This specific subset has been particularly chosen for teaching purpose to avoid the complexities of x86-64. As most desktops have Intel kind of chips which comply to x86-64, therefore it is useful to learn x86-64 kind of ISA (Y86-64) and assembly language. For the hardware design part of the course (project work, Lab manual Chap 6) we will use an ISA which uses immediate memory addressing. Whereas for assembly coding part of the course you will learn and use Y86-64 and more sophisticated memory addressing schemes.

Input, Output, program loading, role of OS, BIOS, compilation and bootloader: The program input and output are retained in the memory. To provide input to the processor one has to insert the input values in a predetermined memory location (which has to be known to the processor before the execution starts). The processor stores the output in a pre-determined memory address. Therefore, to make this machine useful additional mechanisms are required to transfer input values into the memory and transfer output values out of the memory. Input/Output hardware devices play a key role in this. However, the processor itself is utilized to orchestrate this input/output (I/O) operations. This means even to do the I/O, the processor will need to have a code for that too (what a chicken-egg situation!).

A meaningful standalone program that delivers result to users, is actually the core code strapped with additional codes to orchestrate the I/O operations. You may have to write a part of the I/O code, but the rest of the required I/O code is eventually written by somebody else. The compiler and operating system (OS) work together to write the residual I/O code during the compilation process. This happens behind your back and under the hood. These are some of the roles of the compiler and the OS. The OS geeks (Linux/Windows kernel developers) wrote this code in assembly language.

Loading a program in the memory is basically a specialized I/O task. A code component called program loader, which is part of the OS, takes care of this specialized I/O (program loading). OS also dynamically partitions the available memory to instruction and data spaces. In fact, the OS in your desktop have layers. The firmware code called Basic I/O system (BIOS) sit inside a ROM that comes with the motherboard. On top of that the Linux/Windows OS sit in a specific portion of the instruction memory space known as kernel memory. The BIOS treats the Linux/Windows OS as a program to load and execute, whereas the Windows/Linux treats your program as a code to load and run on top of it. This stacking of codes is what was shown in Fig.1 in the previous Chap 5 of the manual.

In simpler embedded systems, instead of the BIOS, a specialized hardware chip is used which takes care of the initial program loading. This is hardware circuitry is known as bootloader circuit. Instead of using a bootloader, system developers sometimes use a simple trick. They program an "EPROM (electrically programmable ROM)" with the code by a process which is called as "burning an EPROM". Then they plug that programmed EPROM in the "IC socket" connected to the rest of the circuit or "printed circuit board (PCB)" containing the processor and peripherals. Then they power up the entire circuit. Once powered up the processor will start reading from instruction memory location zero and go on executing the entire program. Most processors have this behavior when they are either powered up or reset by enabling its reset pin.

For designing the circuits follow the discussions in the lab. Try to clarify your doubts in the lab itself.

---

1. Required Tools: Logisim 3.8.x or above

2. Group size: 3

3. Objective: *To learn -*

Primary topics

1. Designing a scaled down version of an instruction set architecture.
2. Designing and verifying a simple scaled down version of the processor – instruction fetch, decoder, control, ALU, registers, data memory read-write and data path components.
3. How programs are executed in a computer.

Secondary topics

4. Issues about program loading and boot-loading.

4. Instructions:

1. Use Logisim to design and validate the designed processor. Create each processing block separately and validate their functionality. Once they are validated, integrate the components to implement the entire processor. Use sub-circuits to clearly demarcate the components.
2. Each group will submit a single common project report, but each member individually have to submit a single additional 1-page report. Use the same format for that 1-page report which you used in the Lab 4.

5. Project to do: To design and implement a processor that can execute any program written in Y86-64 ISA. For instance, addition of two 2x2 matrices in the processor and store the result in RAM. The initial values for the matrices to be added should be taken from RAM. Whole program instructions and data must be loaded into ROM before starting the simulation. After starting the simulation, first task will be to copy the content of ROM to RAM. (Please follow the discussion in the lab for details.)

Activities to do(20 in report, 80 in demo)-

1. **Design and verify a minimal processor that can execute the Y86-64 ISA (except cmov, call, return)**. Only design the bare minimum circuits which are absolutely essential to execute the instructions and **design the peripheral input/output circuit(s) for this designed processor**, which can interface with the RAM/ROM modules. Your circuit in LogiSim should be clean, modularized, and easy to understand. Otherwise, you will receive a maximum deduction of 20 points.

   For each functional block (stages) of the processor:

   a. Explain its operation in the report (you may want to insert screenshots from Logisim to help you in your explanation).
   b. If you split the work of creating these various sub-blocks, clearly demarcate which parts were contributed by each of the members.

2. **Provide the timing diagram of your processor in your report.**
   a. The diagram at least includes outputs of each stage.
   b. The following signals must be included - icode, ifun, rA, rB, valC, valP, valA, valB, valE, valM, (new) PC.

3. **Final Project Demo in lab to TA, it will be part of your grade.**
   a. Create Y86-64 simple programs (at least 3) that cover all the instructions in the Y86-64 ISA.
   b. Show your TA how your processor runs your programs.
   c. Attach the files of your tested programs (assembly, binary, converted memory files) in the submission.

4. **Honors Requirement:**
   a. Implement “call” and “return” instructions.
   b. Write a matrix multiplication program that is called from main method and after the computation is done return to main method.
   c. For that implementation the return address must be pushed into the stack, and upon finishing the called method, should be popped from the stack to change the PC to the return address.

Note: Once the simulation starts, you should not touch your design at all (no button or switch press, no manual reset or control signals). It should run all the instructions in the memory file one by one and then have the final results same as when run using yis executable.

Possible work divisions for a team with 3 members:

Member-1	Member-2	Member-3
Design and implement ISA. Figure out the encoding for ISA. Implement Fetch and decode pipeline stages in the processor (implement proper control signals for the execution unit).	Write down the code in Y86 assemble and also in the ISA designed. Make a script to parse the assembly code (written using the ISA designed) and generate hex code that is readable to the ROM block in Logisim.	Implement execution and write back unit. Implement memory access and I/O. Need to integrate all the components using BUSes and control signals. Control signals must be automated with the instructions.
All members		
Jump instructions (honors: call-return) should be done together. Because for implementing jump instructions you need the knowledge of each of the member's individual portions.		

<u>Note: You have to design a general-purpose processor which should be able to execute any program written using the Y86-64 ISA.</u>

Tips:

1. Read Byrant's book section 4.1, 3.1 to 3.6, 4.3.2 to 4.3.5 (in the given sequence).
2. Complete reading all the suggested reading materials, tips and additional materials given in the previous chapter (Chap 5) of this lab manual.