Programming Fundamentals

C Structured Programming

* Arrays
* Structures
* Multiple Files Compilation

Lab Exercise 11:

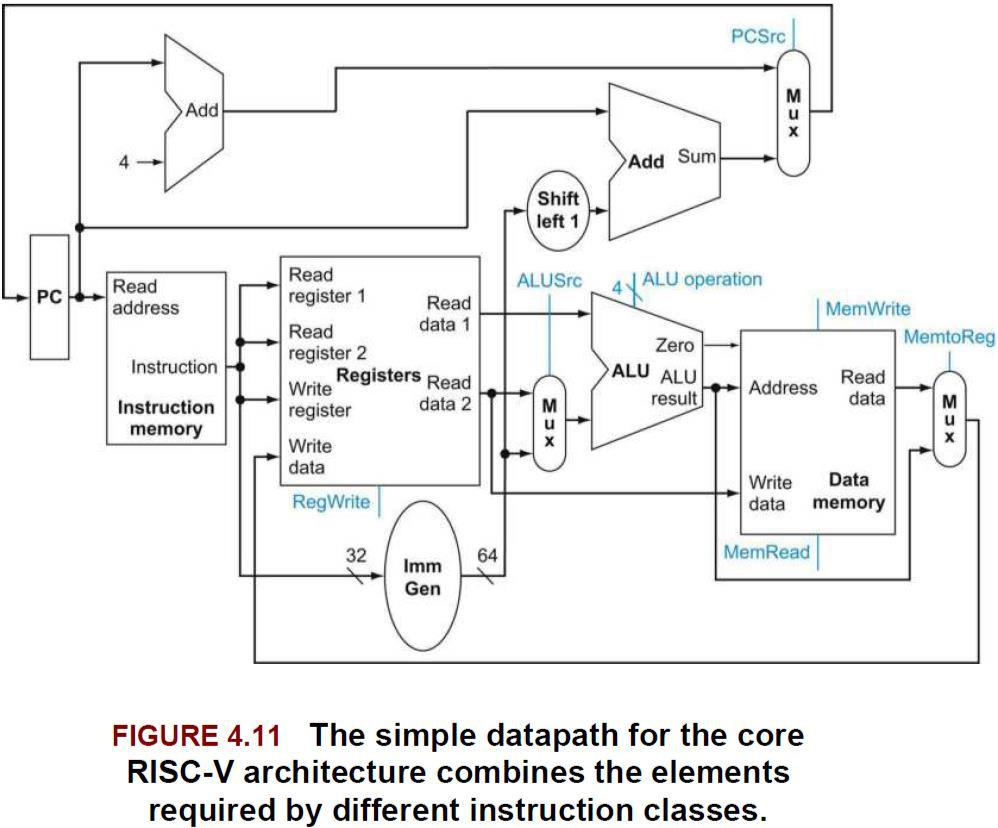
For each program, create a separate .c file. For first program, it will be source\_11\_1.c, for

second program it will be source\_11\_2.c and so on. If a different name is suggested for file in

problem, use that name.

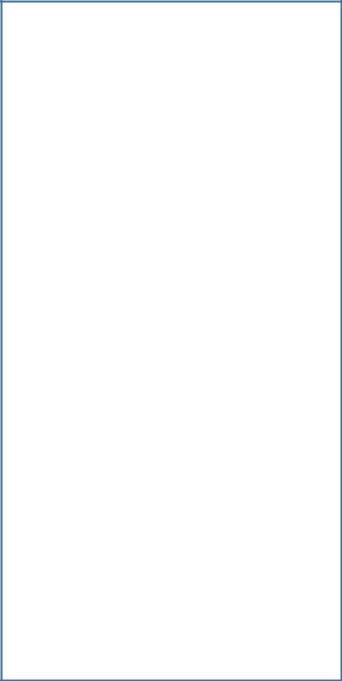
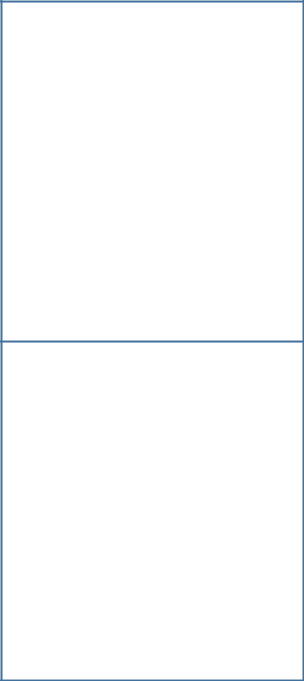
COPING IS STRICKLY PROHIBITED

1. The purpose of this lab exercise is to use C language or any other language to create a structured code that can implement and simulate a design. In our case we will use design of RISC processor. The following diagram show a single cycle processor (RISC-V) data path:



*Figure 1 RISC-V Data path*

The above diagram is representation of a digital logic circuit with different components. We are familiar with these different components as these were discussed in earlier lab experiments. We can refer these components as modules in C language paradigm. There are two main modules. The diagram below shows these two modules, processor and main memory/RAM.



|  |  |  |
| --- | --- | --- |
|  | Address |  |
| Data Memory | Data |  |
|  |  |
|  | Control |  |



Microprocessor

|  |  |  |
| --- | --- | --- |
|  | Address |  |
| Code Memory | Instruction |  |
|  |  |



Memory Processor

*Figure 2 Single Cycle Machine Top Level Diagram*

To simulate this processor, first we need to create these two modules. The interface through which these modules can communicate is also visible through diagram [Figure 2.](#page2) The processor module can communicate with memory using address, data and control buses with processor. Each of these buses has a particular size/width. We can assume that each of data and control bus is of integer size or 32 bits.

Memory Module

Lets’ focus on memory module first. The memory module can store both data and instructions. Each data word or instruction can be read or written if address is applied and proper control signals are applied. We can create memory using arrays. We will need two separate arrays for data and instruction memory. We will read either data or instruction of size integer using read function if read signal is 1. We will write data or instruction using write function if write signal is 1. We can now summarize data stores and functions required for memory module.

**Memory Module**

**Data Storge Elements**

|  |  |
| --- | --- |
| IntructionMem | Memory to hold instructions |
| unsigned int array |  |
| DataMem | Memory to hold variables |
| unsigned int array |  |
| **Functions** |  |

|  |  |
| --- | --- |
| void initializeMemory(enMem memType); | Function to initialize all memory to zero |
|  | Input: |
|  | Memory type enum – Can CODE or DATA |

|  |  |
| --- | --- |
|  | Output: void |
| unsigned int readMemory(enMem | Function to read memory element from an |
| memType, int address, int readFlag); | address if read flag is 1 |
|  | Input: |
|  | Memory type enum – Can be CODE or |
|  | DATA |
|  | Address – Can be from 0 to 255 |
|  | readFlag – Can be 0 or 1 |
|  | Output: |
|  | Value at input address if readFlag is 1, 0 |
|  | otherwise |
| void writeMemory(enMem memType, int | Function to write on memory address if |
| address, unsigned int val, int writeFlag); | write flag is 1 |
|  | Input: |
|  | Memory type enum – Can be CODE or |
|  | DATA |
|  | Address – Can be from 0 to 255 |
|  | writeFlag – Can be 0 or 1 |
|  | Output: |
|  | Void |
| void printMemoryData(enMem memType, | Function to print all memory data values |
| int size); | Input: |
|  | Memory type enum – Can CODE or DATA |
|  | Output: void |

The header file for memory module is provided. You are required to write memory.c. Note that data storage elements are defined in memory.c and can only be modified using functions from outside of memory.c. This is an example of data encapsulation. These functions also termed as getter and setter functions since data storage elements are not visible outside of memory module and can only be set through or readback (get) through read and write functions.

Processor Module

All components other than data memory and code memory are part of processor module or its sub modules. Let us identify different components used in this diagram that will be part of processor module.

|  |  |
| --- | --- |
| Components | Description |
| Mux | There are 3 mux2x1 used in this diagram. We will use 3 |
|  | mux2x1 with 32 bit input/output width |
| ALUs | We see three ALUs in this diagram |
| Control Unit | Blue signal originate from control unit that is not visible in |
|  | this diagram |
| Program Counter (PC) | PC is used to provide address of next instruction to |
|  | Instruction memory |
| Instruction Register | Instruction register, not shown here fetches current |
|  | instruction from instruction memory. Control unit will use |
|  | this instruction to generate required signals |
| Registers | Register bank is used to store our variables/data from data |
|  | memory or results after operation by ALU |

|  |  |
| --- | --- |
| Shift Left and Imm Gen | We can ignore these modules for now |

All components other than registers, program counter and instruction register take data as input and map out based on input, hence are functions. We can divide these components in two categories:

1. Functions:

Mux, ALU, Control Unit, Shift left and Im Gen are functions

1. Data storage elements

Program counter, Instruction register and registers are storage elements, hence variables. Registers in an array

Mux

We are using 2x1 mux in processor with 2 inputs, 1 control input and 1 output. Create a sperate module mux.c and mux.h that has mux2x1.

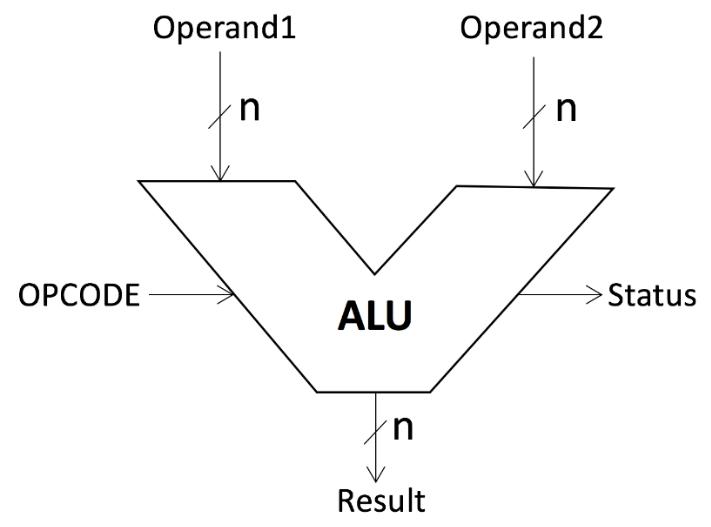
**Mux Module**

**Functions**

|  |  |
| --- | --- |
| int mux2x1(int src, int in1, int in2); | Return output either in1 or in2 based on |
|  | control input src. A value 0 at source |
|  | selects in1 |
|  | Input: |
|  | Src – control input, can be 0 or 1 |
|  | In1 – First input |
|  | In2 – second input |

Arithmetic Logic Unit (ALU)

We will use ALU function defined in previous labs.



The ALU will support all 18 operations mentioned in table below:

|  |  |
| --- | --- |
| OPCODE | Operation |
| 0 | Operand1+Operand2 |
| 1 | Operand1-Operand2 |
| 2 | Operand1\*Operand2 |
| 3 | Operand1/Operand2 |
| 4 | Operand1%Operand2 |

|  |  |
| --- | --- |
| 5 | Operand1&&Operand2 |
| 6 | Operand1||Operand2 |
| 7 | !Operand1 |
| 8 | Operand1>>Operand2 |
| 9 | Operand1<<Operand2 |
| 10 | Operand1&Operand2 |
| 11 | Operand1|Operand2 |
| 12 | ~Operand1 |
| 13 | Operand1^Operand2 |
| 14 | Operand1=Operand2 |
| 15 | Operand1++ |
| 16 | Operand2++ |
| 17 | Operand1 |
| 18 | Operand2 |

Instruction Set

A processor data path depends upon the instruction it supports. For RISC-V processor design, we will drive our own instruction format for ease of design. We will support instructions that allow data to be moved from data memory to processor registers (LOAD), instruction to move data from registers to data memory (STORE) and instructions to perform operations on two operands stored in registers (ADD, SUB, MUL, DIV). We can later add support for instruction that allow repetition or selection (JUMP).

As an example, we need to perform a = b+c. All, a, b, c are in memory. These instructions will be compiled as following instructions for our processor:

LOAD R0, 0 // Move B (address 0) form data memory to register R0

LOAD R1, 1 // Move C (address 1) form data memory to register R1

ADD R0, R1 // Add registers R0 and R1 and store result in R0

STORE R0, 2 // Move data from register R0 to data memory A (address 2)

We can define these different types of instructions as follow:

Each instruction will be of 4 bytes (size int).

1. The first byte will be opcode, this unique value will decide current instruction to be executed. The later bytes will provide operands.
2. Second byte will be reserved for future use.
3. Third byte will provide register address as one of the operands, we have 32 registers in our design. This value can vary from 0 to 31 for register R0 to R31 respectively.
4. Fourth byte will provide data memory address as second operand. We are using memory of 256 bytes so this value can vary from 0 to 255.

The instructions supported as summarized below:

Instruction (size 4 bytes)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Instruction | Opcode | Reserved | Register | Data Address | |
| LOAD | 01 | 00 | 0-31 | 0-255 |  |
| STORE | 02 | 00 | 0-31 | 0-255 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ADD | 03 | 00 | 0-31 | 0-31 |
| SUB | 04 | 00 | 0-31 | 0-31 |
| MUL | 05 | 00 | 0-31 | 0-31 |
| DIV | 06 | 00 | 0-31 | 0-31 |

It is obvious from above table that instruction are converted to machine language (binary) when provided to microprocessor. A much convenient way is to convert them to hex values. As an example, we can calculate hex value for instruction LOAD R7, 1

Opcode: LOAD = 01

Reserved = 00

Register Address: R7 = 07

Memory Address: 1 = 01

Hence LOAD R7,1 in hex = 0x01000701

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | LOAD R7, 01 | |  |  |  |
| 01 | 00 | 07 | 01 |  |  |
| Opcode | Reserved | Register | | Data |  |
| Address | | Address |  |
|  |  |
|  |  |  |



1. Create a structure **struct Instructions** with following four members:
   1. Opcode
   2. Reserved
   3. RegisterAddress
   4. DataAddress
2. Write a function that takes instruction (an integer value) as input and return **struct** **Instructions** after populating all four members according to input. As an example, ifinput is 0x01000701, structure member Opcode will have value 01, Reserved will have value 0, RegisterAddress will have value 07 and DataAddress will have value 01.

**Processor Module**

**Functions**

|  |  |
| --- | --- |
| Struct Instructions | Returns struct Instructions after decoding |
| decodeInstruction(unsigned int inst); | structure members |
|  | Input: |
|  | Inst – Instruction from Instruction memory |

Control Unit

The purpose of control unit is to generate control signals for different instructions. The control signals are responsible to define data path for each of supported instructions. The control signals are shown as blue signals in [Figure 1.](#page1) Control signal are evaluated during decode phase for each instruction. The table below list control signals and their values for each instruction to be executed.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Opcode | Mux1 | Mux2 | ALU | Mem | Mem | Reg | Reg |
|  | ALUSrc | MemtoReg | Operation | Read | Write | Read | Write |
| LOAD | 1 | 0 | OP2 | 1 | 0 | 0 | 1 |
| STORE | 1 | 0 | OP2 | 0 | 1 | 1 | 0 |
| I\_ADD | 0 | 1 | ADD | 0 | 0 | 1 | 1 |
| I\_SUB | 0 | 1 | SUB | 0 | 0 | 1 | 1 |
| I\_MUL | 0 | 1 | MUL | 0 | 0 | 1 | 1 |
| I\_DIV | 0 | 1 | DIV | 0 | 0 | 1 | 1 |

1. Create a structure **struct controlFlags** with 7 control signals mentions above
2. Write a function controlUnit() that takes instrunction as input and populate required values in control structure **struct controlFlags**

**Processor Module**

**Functions**

|  |  |
| --- | --- |
| struct controlFalgs controlUnit(unsigned int | Returns struct controlFalgs after decoding |
| inst); | instruction and populate structure |
|  | members accordingly |
|  | Input: |
|  | Inst – Instruction from Instruction memory |

Program Counter

Program counter holds address of next instruction to be executed. The address is applied to instruction memory to get instruction to be executed. We assume a program counter of type int (32 bit) although our instruction memory size is 256 bytes and requires just 8 bit program counter.

**Processor Module**

**Data Storge Elements**

|  |  |
| --- | --- |
| ProgramCounter | Stores address of next instruction to be |
| unsigned int | executed |

**Functions**

|  |  |
| --- | --- |
| void fetch(void); | Read instruction from instruction memory |
|  | using ProgramCounter as address. The |
|  | instruction is stored in instruction register |
|  | Input: |
|  | Void |
|  | Output: |
|  | void |

1. Define variable ProgramCounter in processor.c
2. Write a function fetch() that reads instruction from instruction memory using program counter as address and stores value in instruction register

Instruction Register

Instruction register hold current instruction to be decoded and executed. Instruction fetched from instruction memory is stored in instruction register during execution.

**Processor Module**

**Data Storge Elements**

|  |  |
| --- | --- |
| InstructionRegister | Stores instruction that is decoded ad is |
| unsigned int | currently being executed |

7. Define variable InstructionRegister in processor.c

Registers

Registers are used for temporary data storage in processor. They store data read form data memory or result generated during operations. Data from registers is stored back in memory for later use or storage. We can model registers as array of type int. Since there are limited number of registers available in processor, we will have array of 32 registers. Registers are named as R0-R31 in RISC-V architecture. Since operation like addition and subtraction require two operands, RISC-V register bank support reading two variables during single read function call.

**Processor Module**

**Data Storge Elements**

|  |  |
| --- | --- |
| Registers | Stores values read from data memory or |
| unsigned int | results generated using ALU operations |
| **Functions** |  |
| void readReg(int address1, int address2, int | Read data1 and data2 from register |
| \* data1, int \* data2, int readFlag); | address 1 and register address 2 |
|  | respectively if read flag is 1. |
|  | Input: |
|  | Address1 – address of first register |
|  | Address2 – address of second register |
|  | Data1 – data pointer to hold first value |
|  | Data2 – data pointer to hold second value |
|  | ReadFlag – can be 0 or 1 |
|  | Output: |
|  | Void |
| void writeReg(int address, int data, int | Write value on register with provided |
| writeFlag) | address if write flag is 1 |
|  | Input: |
|  | Address – address of register to wirte value |
|  | Data – Data to be stored |
|  | writeFlag – can be 0 or 1 |

Decode Function

This function consists of decodeInstruction() and controlUnit() function. This function will be called after fetch() function to decode instruction stored in InstructionRegister. Once these functions are called, control signals are generated and data path is configured to execute instructions.

**Processor Module**

**Functions**

|  |  |
| --- | --- |
| void decode(void); | Calls decodeInstruction() and controlUnit() |
|  | functions. |
|  | Input: void |
|  | Output: void |

Execute Function

This function is responsible to execute the instructions. It consists of following steps:

|  |  |
| --- | --- |
| 1. | Apply addresses from instruction to registers. This will read two values from register |
|  | bank (data1 and data2) |
| 2. | Provide data2 and instruction data address to mux1 |
| 3. | Apply data1 and mux1 output to ALU |
| 4. | Apply ALU output to data memory address and data2 to memory write |
| 5. | Provide output of data memory and alu output to mux2 |
| 6. | Apply address from instruction to register bank and data from mux2 output |
| . |  |