ECE 368 Digital Design - Spring 2016

Project Lab #1: (100pts)

Lab date: March 11th and March 25th,2015

Lab Report Due: Wednesday April 1st,2015

Overview and Objectives:

In this project assignment, you will be using the Xilinx ISE software tools with the Digilent Nexys 2 devboard to build the first part of your machine. You will be applying the experience you gained in the introductory labs in order to create and test your hardware.

Introduction to the UMD RISC

In Figure 1 below, it shows the top-level diagram of the Control entity and the FPU entity together in a Pipeline functional processing unit (FPU). The FPU entity will expand upon the design experience you gained in the Labs. The FPU will automatically run a program from the Control entity which is stored in the Instruction Memory entity. The Instruction Memory for now should hold up to 20 instructions through a five-stage pipeline. This is just a recommendation, this is a open design project which means you can interface as many instructions as space is allowed and the pipeline system can be smaller if you optimize the architecture. The results from the FPU and the system should be outputted to verify that the system is working. This can be done through a debug unit that can either output to the VGA console or serial to a computer to log the results. The Data Memory portion of the design and instruction set will only be used for the load word and store word instructions for this lab.

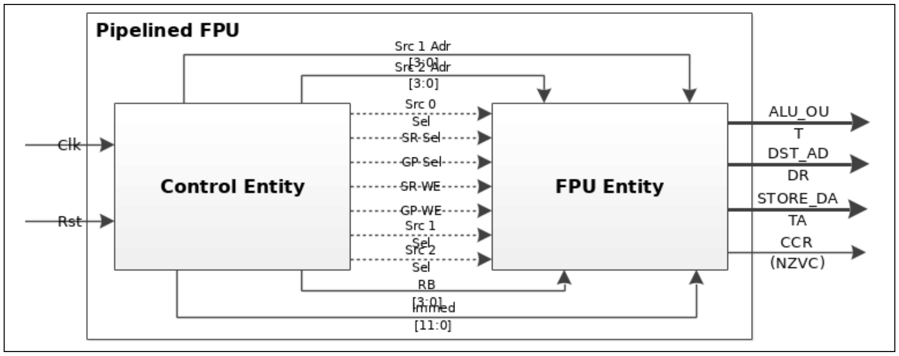


Figure 1: UMD RISC16 - Pipelined ALU Top-Level

For the first project you will implement a full 5 cycle pipeline that takes in one instruction per clock cycle and performs partial operations on five instructions in parallel when the pipe is full. A full pipeline implies that there are 5 different instructions being performed, with each one in a different stage of the pipeline (i.e. one in fetch, decode, operand access, execution and write back). For the data path portion of this design you must implement the major elements of the Harvard Architecture. This includes the Instruction Memory, the Data memory, the functional processing unit (consisting of an arithmetic, logic, shift and load store unit), the sixteen 16 bit register bank and any other needed components. For example a program counter, instruction register, temporary registers and connecting devices (e.g. multiplexers, etc.).

Table 1 shows the instruction set architecture (ISA) you are to implement this in your machine. This portion of the instruction set includes the register type, immediate type, and data type instructions. You will need a variety of hardware components to implement these instructions.

OPCODES:	Operation:	Routine:
0000	ADD REG A, REG B	R[A] ← R[A] + R[B] set status flags NZVC
0001	SUB REG A, REG B	R[A] ← R[A] - R[B] set status flags NZVC
0010	AND REG A, REG B	R[A] ← R[A] & R[B] set status flags NZVC
0011	OR REG A, REG B	R[A] ← R[A] | R[B] set status flags NZVC
0100	MOV REG A, REG B	R[A] ← R[B]
0101	ADDI REG A, IMMED	R[A] ← R[A] + IMMED set status flags NZVC
0110	ANDI REG A, IMMED	R[A] ← R[A] & IMMED set status flags NZVC
0111	SL REG A, IMMED[0..3]	R[A] ← R[A<<IMMED] zero fill LSB C and V affected
1000	SR REG A, IMMED[0..3]	R[A] ← R[A>>IMMED] zero fill MSB
1001	LW REG A, IMMED[7..0]	R[A] ← MEM[IMMED]
1010	SW REG A, IMMED[7..0]	MEM[IMMED] ← R[A]

Table 1: UMD RISC-16 Partial Instruction Set Architecture (R, I, D type instructions)

A high level (not complete) notional design is depicted in Figure 8. The clock, reset and other possible needed control signals are implied in this design for readability. You will need to decide how you will clock the devices or not at all.

This lab introduces design decisions in creating and linking different combinational, behavioral and structural components together to work properly in a larger design. Pay careful attention to timing between different components and instructions in different phases of operation (fetch, decode, etc), Figure 2 and Figure 3.

For example, if you are reading in a new instruction value it must be stored in a register in order to stay stable, likewise every value must be stable before a write occurs. This can be ensured by performing writes during the next cycle or on an edge transitions. Although your design may work in simulation, you will find that the hardware is not as stable and predictable as you would expect. It will take a well laid out RTL designs, timing designs and test procedures to verify and validate your designs. Be precise and well organized for your design and make as much documentation to easily convey your design to the TA/Professor or later on in your career your employer/boss. Lay out timing and control signal transitions carefully for each clock cycle to completely develop a working design.

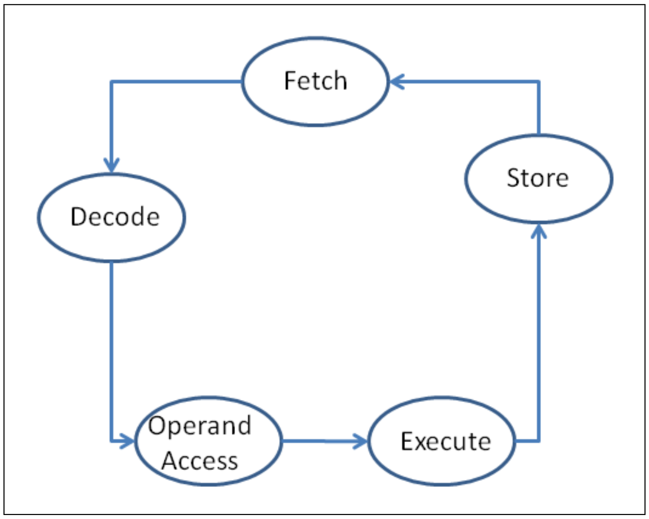


Figure 2: Typical Instruction Execution Cycle

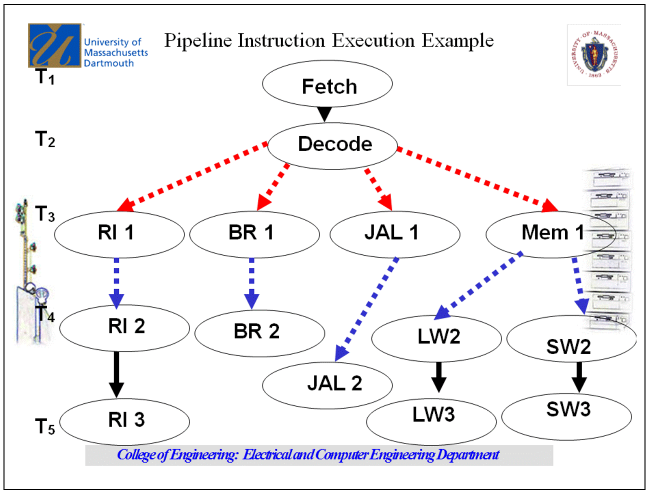

University of Massachusetts Dartmouth
Pipeline Instruction Execution Example
T1
T2
T3
T4
T5
Fetch
Decode
RI 1
BR 1
JAL 1
Mem 1
RI 2
BR 2
LW2
SW2
JAL 2
RI 3
LW3
SW3
College of Engineering: Electrical and Computer Engineering Department

Figure 3: Concept for Pipelined Instruction Parallel Execution steps

Procedure:

1. Complete the design of the Control Path entity. For this and later labs, this will be the heart and soul of your machine. This entity currently consists of possibly the following main components which must be implemented as separate VHDL entities, then will be integrated into a larger control entity:

    - **Program Counter**: Controls the current instruction to be executed. Automatically increments on each clock cycle once started and later will be able to be edited by the system if need be(JMP).
    - **Pipeline Registers**: Multiple 16-bit registers must be created to hold instruction words (and state) for the appropriate pipeline stage.
    - **Pipeline Controller**: Facilitates transition of instruction words for each pipeline cycle. Although we only have a few pipeline registers for now, this will be a great help once more stages are added and more sophisticated transitions are required.
    - **Control Modules**: Act on the control signals based on the current instruction in the pipeline register for their associated pipeline stage. (i.e. CTRL1 acts on T1, CTRL2 acts on T2). These modules dictate the functionality of the rest of the machine. For example, if your current instruction is a register add, the control modules will decode the instruction to output the control signals for which register to read from, the operation to be performed, the logical flow of data through your FPU, and ultimately where to store it and when to enable the write enable back into the register bank and any other pertinent control signals.

    Capture RTL diagrams of your design for your lab report.

2. Create a testbench for your control entity and verify its outputs are what you expect. Show that each instruction is read sequentially and propagated through pipeline. Verify and show that the control and data signals match your intended design.

    Document your test procedure, print your testbench results, and discuss any problems for your lab report.

3. Construct the FPU entity and associated components. This design builds upon the ALU from your previous experience in Lab 2. For your reference, a sample design has been included in the class directory along with a testbench. If you decide to use some of these design concepts ensure that they fit your design and modify the entities and testbench as necessary.

    There are several components that must be wrapped around your ALU design. These are controlled by the control modules discussed in Step 1. Some notable entities are discussed as follows:

- **General Purpose Register Bank**: sixteen 16-bit general purpose registers (R0-R15). Similar in structure to the Instruction Memory in the Control entity except it is dual ported. You will have two address inputs of which only will be written to and from the data in din. Both outputs will reference the 16-bit register indexed by the two corresponding addresses.
- **Shadow Register Bank**: (SR0-SR3) for stack operations, and fast-interrupts supporting I/O operations. Implement but do not worry about implementing in your full architecture just yet. This will be focus more on in Project Lab 3.
- **Instruction Memory**: Read-only register containing 32, 16-bit words indexed by the address output from the program counter. Create this as a simple VHDL entity and initialize with your desired instruction word values. Since we are only accessing up to 20 instruction words for now, only the least significant five bits of the program counter are needed for instruction memory addressing. Figure 4 shows the format of these instructions. You must correctly populate each field depending on the type of instruction you are executing and the desired output.
- **Data Memory**: Read-Write memory containing M, 16-bit words indexed by the address output from the instruction register. Create this as a VHDL entity and initialize with your desired data word values.
- **Store_Data Register**: Output from FPU for use in writing results to data memory.
- **Dst_Addr Register**: Output from FPU for use in determining destination addresses.

Capture RTL diagrams of your design for your lab report.

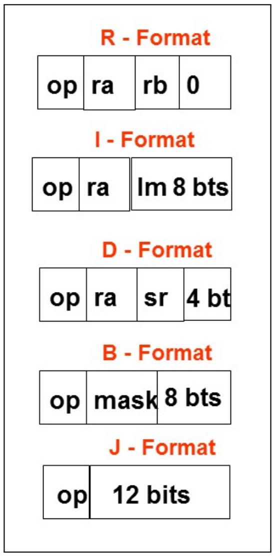


Figure 4: UMD-RISC 16 Instruction Modes

4. Create a testbench to simulate your FPU Entity. When you are ready, integrate both Control and FPU entities together and verify that your design outputs the values you expect in a testbench.

5. Design a small program (20+ instructions) to test your design using all of the instructions in Table 1 and at least two registers.

   A Python assembler is provided to help with generating instructions from assembly to machine code for your design.

   Show that the expected outputs of the FPU are seen at each stage of you testbench in Step 4.

   Document and create a hardware test to run your program. Run all of your instructions and latch the final values of your FPU on a display media of your choice (this includes LEDs, LCD, VGA, etc.).

Your final result should be something that builds on each previous instruction to verify your entire program ran as expected.

Show the final hardware test to the professor or TA for verification.

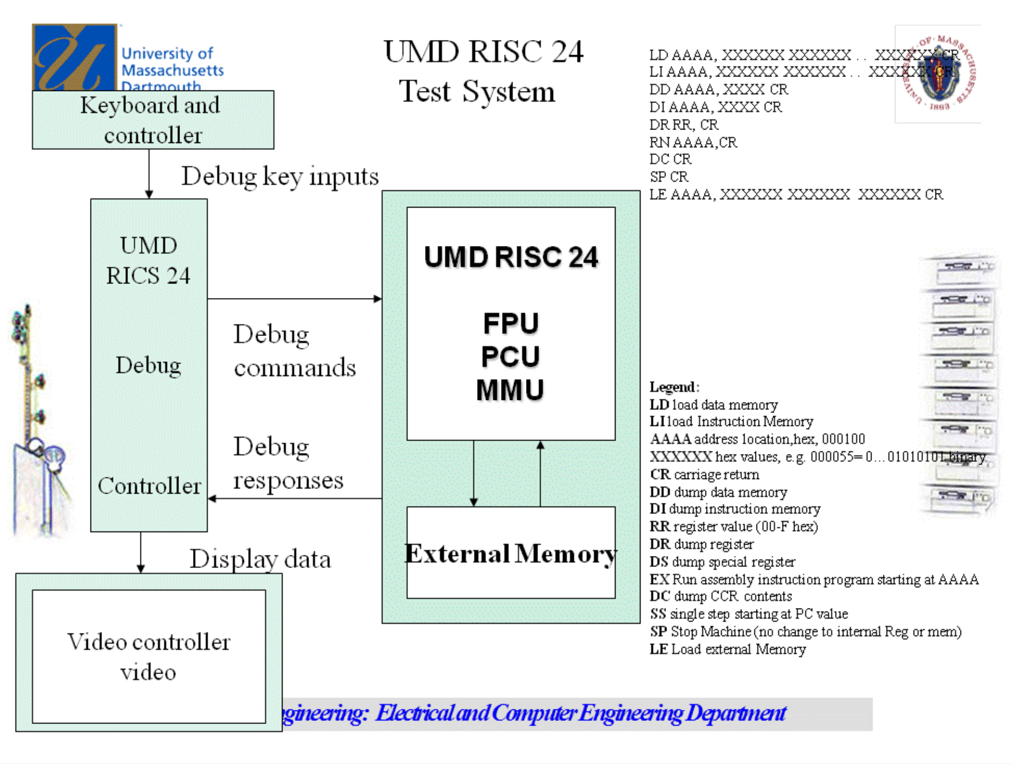
```
Simple Program:
; In this sample program, the final value would be 2 in R0, a 1
;   in R5 after all instructions are executed and 4 in R10.

ADDI  R0, 2    ; ADDI R0<-2             R0=2
ADDI  R1, 1    ; ADDI R1<-1             R1=1
SW    R1, $0F  ; DataMem[0F] <- [R1]    DM[0F]=1
LW    R5, $0F  ; R5 <- DataMem[0F]      R5=1
ADD   R0, R5   ; ADD  R0<-R0 + R5       R0=3
AND   R0, R1   ; AND  R0<-R0 & R1       R0=1
OR    R0, R1   ; OR   R0<-R0 or R1      R0=1
ADD   R0, R1   ; ADD  R0<-R0 + R1       R0=2
MOV   R10,R1   ; R10 <- R1              R10=1
SL    R10, 3   ; R10 R10[23:3&000]      R10=8
SR    R10, 1   ; R10 [0& 22:0]          R10=4

The resulting instruction word translation:

5002
5101
A10F
950F
0050
2010
3010
0010
4A10
7A03
8A01
```

Figure 5: Program Instruction Example



UMD RISC 24
Test System

LD AAAA, XXXXXX XXXXXX . . XXXXXX CR
LI AAAA, XXXXXX XXXXXX . . XXXXXX CR
DD AAAA, XXXX CR
DI AAAA, XXXX CR
DR RR, CR
RN AAAA,CR
DC CR
SP CR
LE AAAA, XXXXXX XXXXXX XXXXXX CR

Legend:
LD load data memory
LI load Instruction Memory
AAAA address location,hex, 000100
XXXXXX hex values, e.g. 000055= 0. . . 01010101 binary
CR carriage return
DD dump data memory
DI dump instruction memory
RR register value (00-F hex)
DR dump register
DS dump special register
EX Run assembly instruction program starting at AAAA
DC dump CCR contents
SS single step starting at PC value
SP Stop Machine (no change to internal Reg or mem)
LE Load external Memory

Figure 6: Possible UMD RISC Test configuration

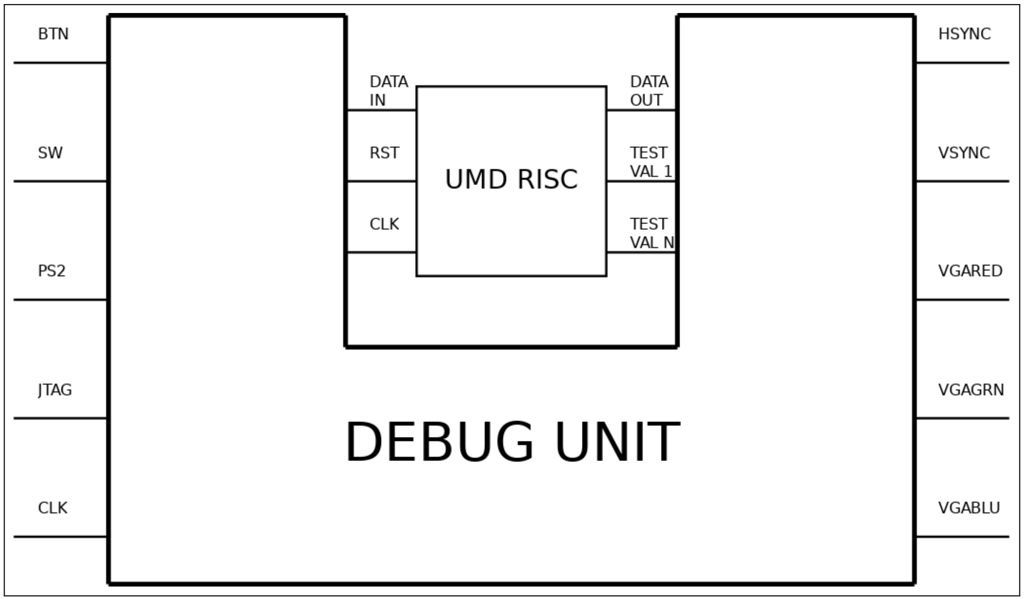


Figure 7: Another Possible UMD RISC Test configuration

Deliverables:

Please only print out your testbench code. A print out of all your code is not necessary. Please zip all VHDL files and UCF file and email to the **Instructor** and the **TA** with the filename **ECE368_Project_Lab1_Team#.zip**. Describe your new VHDL designs in your lab report. You can use portions of the code or complete listings if necessary, but discuss how the components operate and any required conditions or constraints on them.

Explain your testbench, print out your testbench results, and discuss the results you observed. Likewise, explain your hardware test fully and discuss any final results and/or problems you encountered. Include any handwritten design flowcharts or pseudocode used in your lab report.

Lab Report:

Each lab group should prepare a lab report with the following information:

- Follow the **CPE Laboratory Report Guidelines** for base format
- Machine RTL block level design
- VHDL component specification and schematic designs.
- VHDL system specification and resulting schematic design.
- Test plan design and test bench code
- Hardware test procedure used to verify your design
- Simulation results for your simple machine using a testbench (Prove it works).
- Schematic and UCF file for your simple machine.
- Conclusion needs to include a discussion of the learning utility of this lab and discussion of any issues you had with lab design and clarity.
- Reflection section discussing what you did well in your design, what you did not so good and why and what you plan to do to improve your design and design skills in the next lab and for the final project.

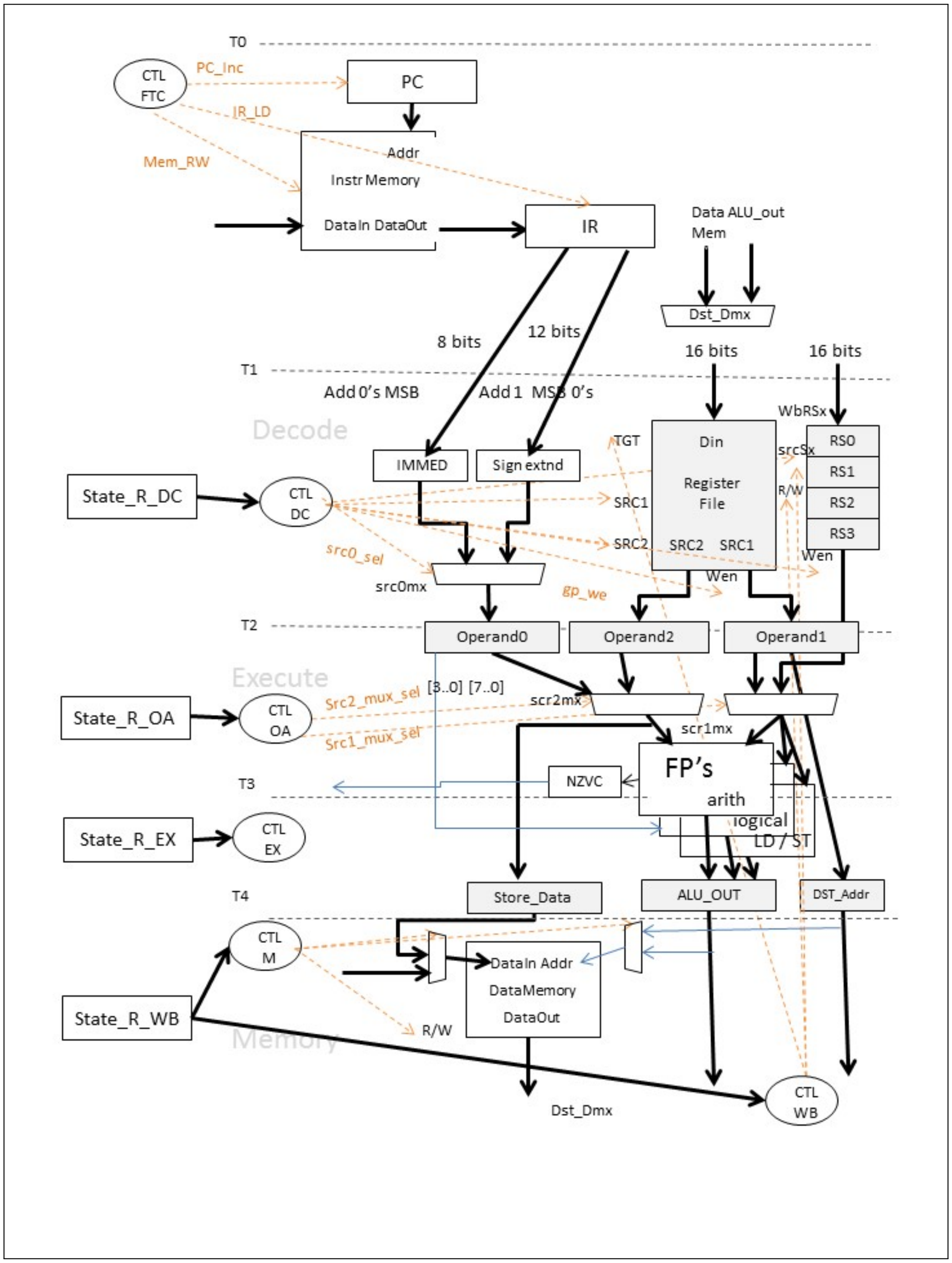


Figure 8: Notional Pipelined Data Path and Control Paths

Project Lab 1 Grade

Section	Value	Score
Procedure:	**-40-**	
1. Design the Control Path Entity	10	
2. Simulate Control Entity	10	
3. Design the FPU Entity and associate components	10	
4. Simulate FPU entity with Control Entity	5	
5. Show Final Hardware Test	5	
Deliverables:	**-20-**	
Lab Report:	**-40-**	
Total	100	

Table 2: Project Lab 1 Grade Breakdown