# 

***Digital Design and Computer Architecture***

Kent Jones

# MP 3: Control Unit Design

|  |  |  |
| --- | --- | --- |
| **CATEGORY** | **POINTS** |  |
| **Exercise 1: CU Design Diagrams** |  | 25 |
| **Exercise 2: CU VHDL Integrated Into your Processor** |  | 25 |
| **Exercise 3: Testbench for Program(s)** |  | 25 |
| **Exercise 4: Group Presentation** |  | 25 |
| **TOTAL** |  | 100 |

# 

# Introduction

In this lab you will design the Control Unit for your processor and verify that your instructions are working correctly. The control unit sets the signals which tell the components in the DPU what to do. A control unit implements what is known as the “Decode” and “Execute” stages of an instruction’s life-cycle.

Single-cycle and multiple-cycle machines will have different types of control units.

* In a single-cycle machine, the control unit is essentially just fancy “**look-up” combinational logic** that acts like a python dictionary or a C++ map. The instruction op-code functions like the input “key” and the control unit then returns the “value” which is placed on the control bus. The control bus is the set of wires that carry control signals to the components in the DPU. Every instruction will have a different set of control signals that drive the control bus.
* In a multi-cycle machine, the control unit is implemented using a finite state machine. In this case, each instruction op-code is the input to the machine which triggers a series of control signal states. All of the instruction share the “fetch” phase of the FSM, but, after decoding, each individual instruction will have a separate set of states required for the execute phase of the instruction.

#### Control Unit Design

Have your Assembly Language, ALU, and DPU designs easily accessible while you work on this design.

1. Make a list of all the op-codes for each of the instructions supported:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Opcode** | **Binary opcode** | **Purpose** |
| Arithmetic 0001 | And | 0001 0000 | Add |
| Or | 0001 0001 | Subtract |
| Xor | 0001 0010 | Multiply |
| Add | 0001 0011 | Divide |
| Increment | 0001 0100 | Modulus |
| Multiply | 0001 0101 | And |
| Division | 0001 0110 | Or |
| Modulus | 0001 0111 | % |
| Nand | 0001 1000 |  |
| Nor | 0001 1001 |  |
| Not | 0001 1010 |  |
| Subtract | 0001 1011 |  |
| Decrement | 0001 1100 |  |
| --- | 0001 1101 |  |
| Exponential | 0001 1110 | ^ |
| Set less than | 0001 1111 |  |
| Data Control 0010 | ld | 0010 0001 | Load from memory |
| St | 0010 0010 | Store into memory |
| li | 0010 0011 | Load immediate |
| Conditionals 0011 | jmp | 0011 0001 | jump |
| Move  0100 | mov | 0100 0001 | r2 = r1 |

1. Make a list of all control signals required by the DPU and the ALU. Keep the control signals for the ALU separate from the control signals needed by the components in the DPU:

ALU Signal List:

|  |
| --- |
| Input a |
| Input b |
| AluControl |
| Zero |
| Result |

DPU Signal List:

|  |
| --- |
| PC (program counter) |
| Instruction signal |
| Memory write |
| Memory read |
| Register write |
| Register read |
| Alu source |
| jump |

1. Control Unit Design:
   1. SINGLE CYCLE CU DESIGN: Create decoder tables for your CPU (e.g. See pages 383-387 of your textbook for examples of decoder tables.)
   2. MULTI CYCLE DESIGN: If you are doing a multi-cycle CPU draw out the entire bubble chart FSM for your design. (e.g. see pages 398-408 in your textbook for an example of the finite state machine design for the multi-cycle mips processor). HINT: You may find it very helpful to place the FSM into tabular form (e.g. excel spreadsheet), remember, every FSM can be drawn as a table (adjacency list) or as a graph.(nodes/edges))

Decoder table:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Opcode (8b)** | **Alu source** | **jump** | **Mem read** | **Mem write** | **Reg write** | **Reg read** | **ALU Op (4b)** |
| And | 0001 0000 | 0 | 0 | 0 | 0 | 1 | 1 | 0000 |
| Or | 0001 0001 | 0 | 0 | 0 | 0 | 1 | 1 | 0001 |
| Xor | 0001 0010 | 0 | 0 | 0 | 0 | 1 | 1 | 0010 |
| Add | 0001 0011 | 0 | 0 | 0 | 0 | 1 | 1 | 0011 |
| Increment | 0001 0100 | 0 | 0 | 0 | 0 | 1 | 1 | 0100 |
| Multiply | 0001 0101 | 0 | 0 | 0 | 0 | 1 | 1 | 0101 |
| Division | 0001 0110 | 0 | 0 | 0 | 0 | 1 | 1 | 0110 |
| Modulus | 0001 0111 | 0 | 0 | 0 | 0 | 1 | 1 | 0111 |
| Nand | 0001 1000 | 0 | 0 | 0 | 0 | 1 | 1 | 1000 |
| Nor | 0001 1001 | 0 | 0 | 0 | 0 | 1 | 1 | 1001 |
| Not | 0001 1010 | 0 | 0 | 0 | 0 | 1 | 1 | 1010 |
| Subtract | 0001 1011 | 0 | 0 | 0 | 0 | 1 | 1 | 1011 |
| Decrement | 0001 1100 | 0 | 0 | 0 | 0 | 1 | 1 | 1100 |
| --- | 0001 1101 | - | - | - | - | - | - | 1101 |
| Exponential | 0001 1110 | 0 | 0 | 0 | 0 | 1 | 1 | 1110 |
| Set less than | 0001 1111 | 0 | 0 | 0 | 0 | 1 | 1 | 1111 |
| ld | 0010 0001 | 1 | 0 | 1 |  | 1 | 0 | 0000 |
| St | 0010 0010 | 1 | 0 |  | 1 |  | 1 | 0000 |
| li | 0010 0011 | 1 | 0 |  |  | 1 | 0 | 0000 |
| jmp | 0011 0001 | ? | 1 | 0 | 0 | 0 | 0 | 1101 |
| mov | 0100 0001 |  | 0 | - | - | 1 | 1 | 0000 |

#### Integrated VHDL for Control Unit

Using the tables and finite state machine diagrams from Exercise 1, design the VHDL for the control unit of your processor.

* SINGLE CYCLE CONTROL UNIT DESIGN: For an example, look at the MIPS single cycle control unit. The VHDL is on pages 431, 432 of the textbook. However there are small differences so you will also want to look at your MIPS3 processor’s control unit code.
* MULTI-CYCLE CONTROL UNIT DESIGN: For this, you will want to refresh your memory on how to write code for a Finite State Machine. See pages 212 and 213 in your textbook.

#### VHDL Testbench Illustrating Two Test Program Running Correctly

Re-use and/or redesign two test programs for your microprocessor. Remember, you will need to project folders one for each algorithm, and every time you change the algorithm test code you will have to re-synthesize your project. Refresh your memory on how to use the debugger in the test-bench to verify the contents of the program have loaded correctly into your instruction memory.

* A simple ad-hoc program that executes every instruction at least one time, and
* A more general algorithm (e.g. similar to the MIPS algorithm you wrote previously). Make sure each algorithm works correctly on your processor.

### What to Hand In:

This document with the following items included. Only one document necessary per group this time.

* List of instructions and op-codes.
* Control signals lists.
* SINGLE-CYCLE: Neatly drawn decoder tables.
* MULTI-CYCLE: Neatly drawn FSM bubble diagram / spreadsheet for the multi-cycle design.
* VHDL code for your completed design in each of your group member’s folder on CS1
* Undeniable evidence that your program(s) are running correctly on your microprocessor.

#### Control Unit Project Group Presentation

1. Show the class your control unit design. If you have a multi-cycle, you must show your bubble diagram state machine diagram. For a single-cycle machine you must show your decoder tables.
2. Demo one of your programs running (either in simulation or on the FPGA)
3. What hardware bugs did you encounter in testing? How did you find them? How did you squash them?
4. Did you go above and beyond the assignment requirements in any way?

|  |  |  |  |
| --- | --- | --- | --- |
| **CATEGORY** | **Beginning**  **70% – 79%** | **Satisfactory**  **80% – 89%** | **Excellent**  **90% – 100%** |
| 50 pts possible  **CONTROL UNIT DESIGN** | Rudimentary decoder tables. | Basic decoder tables and/or basic finite state machine bubble diagram for the control unit. | Neat, well commented, complete set of decoder tables (single cycle) or neat, well commented complete finite state machine bubble diagram and tables (multi-cycle). |
| 25 pts possible  **CONTROL UNIT VHDL** | VHDL code for the control unit. Few comments. | Partially working VHDL code for the control unit. Satisfactory comments. | Working VHDL code for the control unit. Excellent comments, code formatted neatly, etc. |
| 25 pts possible  **CONTROL UNIT TEST** | Simulation test bench created but not documented well or does not work properly | Single ad-hoc test code that tests every possible instruction. | Both an ad-hoc program AND a more advanced program that runs correctly. |