ECEN 651: Microprogrammed Control of Digital Systems
Department of Electrical and Computer Engineering
Texas A&M University

Laboratory Exercise #5

Objective

The objective of lab this week is to complete the design of a single-cycle (non-pipelined) processor. This is the first step towards creating a fully pipelined processor implementation of MIPS. Thus far, we have created a register-file and a simple data-memory unit. We have also shown how to create multiplexers and decoders. It is now time to put them all together to build a simple microarchitecture of MIPS.

Background

The single-cycle MIPS processor is shown in Figure 1. This version of MIPS is labeled as such because each instruction takes a single clock cycle to execute, and instruction execution is not overlapped. In other words, the currently executing instruction must fully complete before the next instruction can start. As we will see in this lab, this sort of implementation is extremely inefficient. Examination of the diagram reveals where the data-memory module and register-file fit in. The instruction memory will be provided for you in the form of a Read Only Memory (ROM) module with preloaded test programs. A portion of the questions at the end of lab will address these test applications.

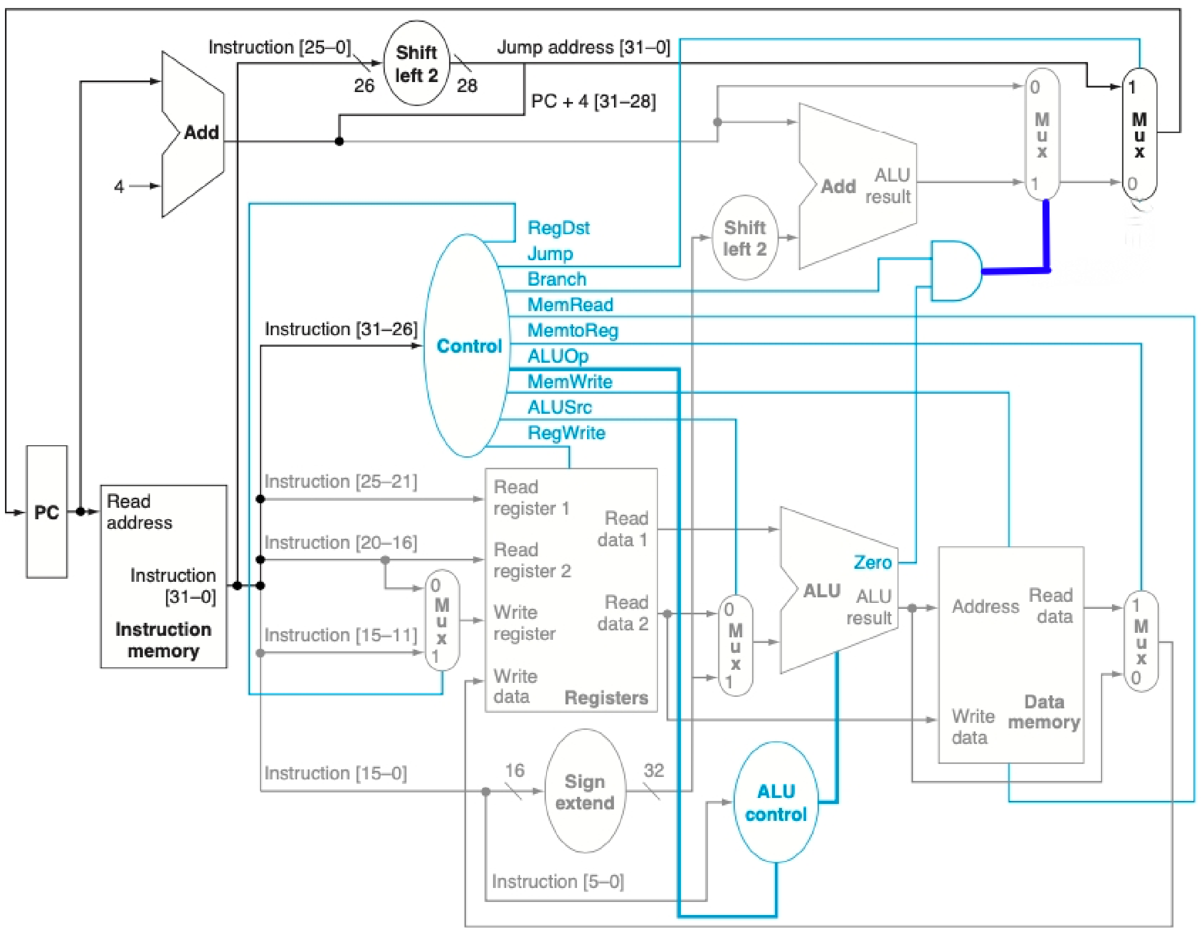


Figure 1: MIPS Block Diagram

Procedure

1. **Design the control unit logic.** The control unit orchestrates the flow of data through the microprocessor by decoding the 6-bit op code within the current instruction. Simply put, each of the control signals shown in Figure 1 will be set to LOW or HIGH based solely on the op code field (see Figure 3). The control unit is nothing more than a large decoder, and this level of simplicity is maintained by the fact that we already designed the ALU control logic.

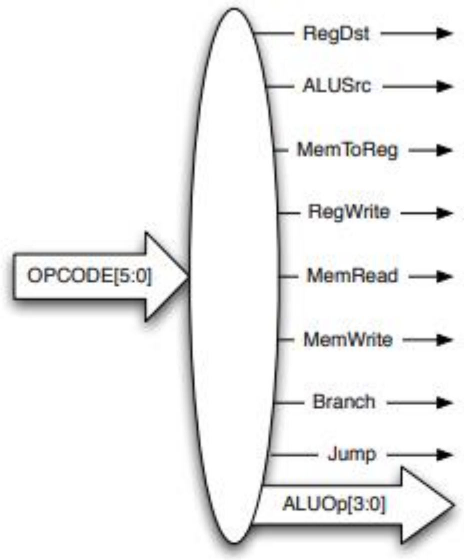


Figure 2: Control Unit Decoder

(1) Use Figure 1 to create a table which lists the values (i.e. 0 or 1 for individual signals and use the defines for the op code field) of all the control signals for the following instructions: R-type, load/store word, immediate (only those currently supported by the ALU), branch (just beq for now), and jump.
(2) Write the Verilog to describe you control logic using the following module interface:

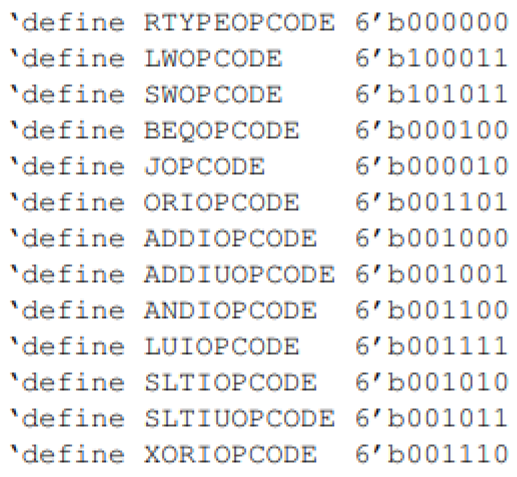
module SingleCycleControl(RegDst, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch, Jump, SignExtend, ALUOp, Opcode);

Helpful Hints:
• The control unit should be completely combinational and can be easily described with a giant case statement.

• You may use the code in Figure 3 to improve the clarity of your code:

• You may also use some of the other defines provided previously in this lab manual to fill in the ALU op field.

• You must specify a value for each control signal regardless of whether or not it is use in the instruction under consideration. If it is not used, specify it as a “don’t care” like this: 1’bx. This allows the optimizer to further optimize your logic.



```
`define RTYPEOPCODE 6'b000000
`define LWOPCODE    6'b100011
`define SWOPCODE    6'b101011
`define BEQOPCODE   6'b000100
`define JOPCODE     6'b000010
`define ORIOPCODE   6'b001101
`define ADDIOPCODE  6'b001000
`define ADDIUOPCODE 6'b001001
`define ANDIOPCODE  6'b001100
`define LUIOPCODE   6'b001111
`define SLTIOPCODE  6'b001010
`define SLTIUOPCODE 6'b001011
`define XORIOPCODE  6'b001110
```

Figure 3. Defines in Single Cycle Control

(3) Synthesize your hardware for now and ensure your code generates no warnings or errors and that it creates no latches (i.e. it is completely combinational). Provide the summary results of the synthesis process in your lab write-up.

2. **Design the remainder of the MIPS single cycle datapath.** Up to this point, we have created all the major subcomponents of the MIPS single-cycle processor and have tested all of them except the control unit. The remaining logic is essentially the glue, logic which interconnects all major subcomponents, and the address generation logic (top of diagram).

   (1) Describe the single-cycle MIPS datapath in Verilog using the following module interface:

   module SingleCycleProc(CLK, Reset_L, startPC, dMemOut);

   Helpful Hints:

   • Download the instruction memory Verilog file on the laboratory website and incorporate it into your ISE project.

   • Some instructions require that the 16-bit immediate field be sign extended, while some do not. This is not depicted in Figure 1, and your control unit design does not support this. You must add this capability.

   • Keep your signal names as consistent with the diagrams and code provide in this lab and try to minimize the number of intermediate signals.

   • For clarity, use assign statements in place of always@(*) for the multiplexers and branch computation logic.

• The PC block is a 32-bit register. Make it sensitive to the negative edge of the clock. Similarly, the Reset signal is active low and should asynchronously reset the PC.

• The state of this machine is contained in only three blocks, the PC, register-file, and the Data-memory. Storage elements implied in any blocks other than these three are a result of errors in your Verilog.

(2) Use the test bench provided on the laboratory website to test the functionality of your overall design. Paste the output of the simulator into your lab write-up.
(3) Once your design passes all tests provided by the test bench, synthesize your design and ensure no errors or warnings exist. Provide the summary results of the synthesis process in your lab write-up.

Deliverables

1. Submit a lab report that captures your efforts in lab.
2. Include all Verilog source files with comments.
   Note: Code submitted without adequate comments will not be graded!
3. Be sure to include all material requested in lab.
4. Answer the following review questions:
   a. Explain why we designed the data memory such that reads happen on the positive edge of the clock, while writes happen on the negative edge of the clock.
   b. Take a look at the test programs provided in the instruction memory Verilog file (only test programs 1 through 3) and briefly explain what each of them do and what instructions they test.
   c. If you wanted to support the **bne** instruction, what modifications would you have to do to your design? Include changes to all affected subcomponents as well.
   d. What clock rate did the synthesis process estimate your overall design would run at? Explain why this design is inefficient and provide suggestions for improvement.