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

Prof. Mi Lu
TA: Ehsan Rohani

Laboratory Exercise #5

Non-pipelined MIPS Processor

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 major components which still need to be designed include the Arithmetic Logic Unit (ALU) and the control unit. 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.

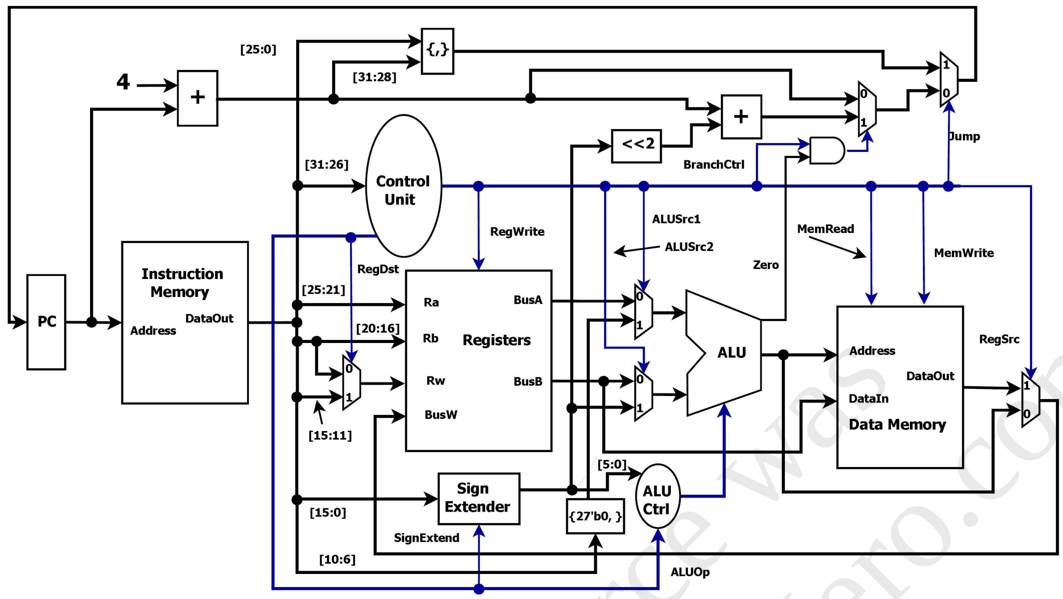

Figure 1: MIPS Block Diagram

Procedure

1. **Design the ALU.** The Arithmetic Logic Unit (ALU) in MIPS is a combinational logic module that produces an arithmetic or logic result from two input operands based on the control signals input to the module. Figure 2 provides the port interface diagram of the ALU you will build in this lab. The two operand buses, BusA and BusB, are each 32-bits wide, while the control bus is 4-bits wide. The outputs from the ALU module include a 32-bit result bus, BusW, and a Zero signal, which is HIGH when the result equals zero. Please note that this ALU does not provide support for arithmetic exceptions; however, we will fix this in later labs. The ALU can be thought of as combination of arithmetic and logic blocks, each sharing the same inputs, such that the outputs of each of the blocks are multiplexed to create the final result. In this illustration, the ALU control signals act as the selection lines of the multiplexer.

   (a) Write a Verilog module to implement the ALU discussed above. Use the following module interface:

   ```
   module ALU(BusW, Zero, BusA, BusB, ALUCtrl);
   ```

   Helpful Hints:

   - Use Table 1 in conjunction with the MIPS reference card on the laboratory website as a guide.

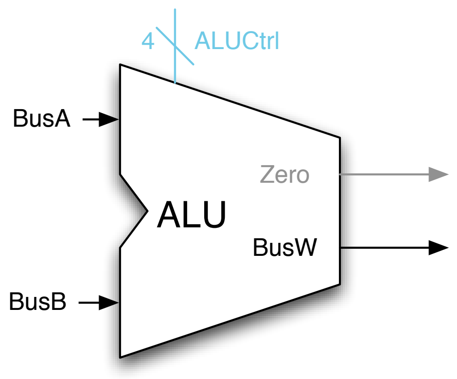

Figure 2: ALU port interface

- Make sure that you consider the fact that for shifts BusA holds the Amount of shift and BusB is the value that ALU is going to shift.
- Your code should make use of the case statement.
- Since exceptions are not being supported initially, ADD and ADDU are identical operations. The same goes for SUB and SUBU.
- SLT must compare 2 two's complement numbers. This can be done by first negating the sign bits of the two operands and then performing an unsigned comparison.
- To improve the clarity of your code, use the following defines:

```
`define AND  4'b0000
`define OR   4'b0001
`define ADD  4'b0010
`define SLL  4'b0011
`define SRL  4'b0100
`define SUB  4'b0110
`define SLT  4'b0111
`define ADDU 4'b1000
`define SUBU 4'b1001
`define XOR  4'b1010
`define SLTU 4'b1011
`define NOR  4'b1100
`define SRA  4'b1101
`define LUI  4'b1110
```

(b) Simulate the operation of your hardware using ISE and the ALU test bench provided on the laboratory website. Paste the output of the simulator into your lab write-up.

Opperation	ALU Control Line
AND	0000
OR	0001
ADD	0010
SLL	0011
SRL	0100
SUB	0110
SLT	0111
ADDU	1000
SUBU	1001
XOR	1010
SLTU	1011
NOR	1100
SRA	1101
LUI	1110

Table 1: Arithmetic Logic functions and associated control signals

(c) Synthesize your design using the Device properties provided in Lab 3. Fix **all** warnings and errors and ensure the compiled hardware contains no latches or flip-flops. Provide the summary results of the synthesis process in your lab write-up.

2. **Design the ALU control logic.** The process of decoding an instruction into a control code that the ALU can understand (Table 1) is split up into two blocks. The first block is contained within the control unit and will be addressed shortly. The second block is the ALU control block shown in Figure 1, which takes in an ALU op control code (*Note the control signal is thicker indicating a bus*) from the control unit and the function field from the current instruction. The ALU control logic should asynchronously function as follows:

   - When the ALU op bus is not equal to 4'b1111, the ALU op code should be passed directly to the ALU.
   - When the ALU op bus is equal to 4'b1111, the function code should be used to determine the ALU control code.

   For R-type instructions, the control unit simply sets the ALU op to 4'b1111 and the ALU control logic will interpret the arithmetic or logic function needed to carry out the instruction and pass this information on to the ALU. However, when an instruction other than an R-type instruction is executing, the control unit must specify the correct ALU control signal on the ALU op bus.

(a) Describe the ALU control logic in Verilog using the following module interface:

```
module ALUControl(ALUCtrl, ALUop, FuncCode);
```

Helpful Hints:

- Case statements can be easily nested within if else statements within *always@(*)* blocks.
- Use the following define statements (in addition to the ones provided above) for code readability:

```
`define SLLFunc  6'b000000
`define SRLFunc  6'b000010
`define SRAFunc  6'b000011
`define ADDFunc  6'b100000
`define ADDUFunc 6'b100001
`define SUBFunc  6'b100010
`define SUBUFunc 6'b100011
`define ANDFunc  6'b100100
`define ORFunc   6'b100101
`define XORFunc  6'b100110
`define NORFunc  6'b100111
`define SLTFunc  6'b101010
`define SLTUFunc 6'b101011
```

(b) Simulate the operation of your hardware using ISE and the ALU control test bench provided on the laboratory website. Paste the output of the simulator into your lab write-up.

(c) Synthesize your design using the same Device properties as in Section 2. Fix **all** warnings and errors and ensure the compiled hardware contains no latches or flip-flops. Provide the summary results of the synthesis process in your lab write-up.

3. **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.

   (a) 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.

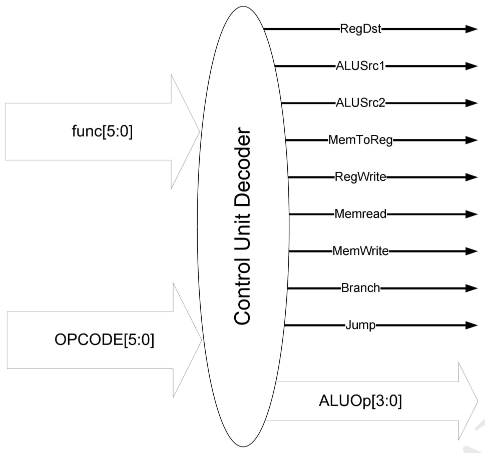

Figure 3: Control Unit Decoder

(b) Write the Verilog to describe you control logic using the following module interface:

```
module SingleCycleControl(RegDst, ALUSrc1,ALUSrc2, 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 following defines to improve the clarity of your code:

```
`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
```

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

- Use the textbffunc field only in case the instruction is shift.
- You must specify a value for each and every 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.

(c) 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.

4. **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 of the major subcomponents, and the address generation logic (top of diagram).

(a) 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 instruction 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.If the **SignExtend** signal is 'one' we extend the **immediate** field with sign of the MSB of That field, and if it is 'zero' we extend the field with 'zero'.
- 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.

(b) 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.

(c) Once your design passes all of the 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.

1 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.