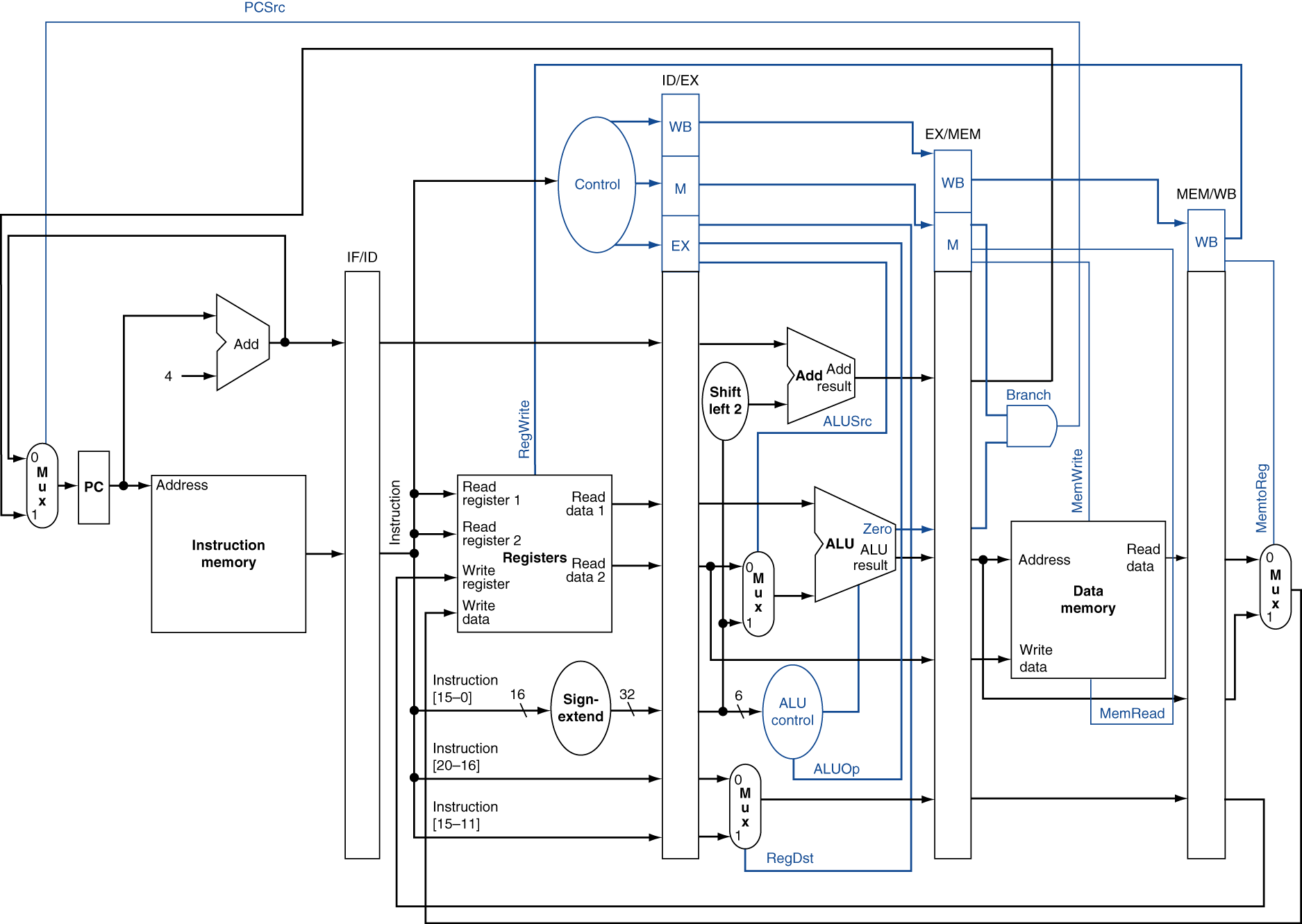
**PHASE1: LABS 8-14 (400 points)**

* **Objectives:**
* Design and build a **datapath** to execute a subset of MIPS instructions

**Table 1.** Required MIPS Operations for the datapath design

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Instruction** | **Code** | **Type** | **Instruction** | **Code** |
| **Arithmetic** | **Add** | **add** | **Logical** | **And** | **and** |
|  | **Add Immediate Unsigned Word** | **addiu** |  | **And immediate** | **andi** |
|  | **Add Unsigned Word** | **addu** |  | **Or** | **or** |
|  | **Add Immediate** | **addi** |  | **Not or** | **nor** |
|  | **Subtract** | **sub** |  | **Exclusive or** | **xor** |
|  | **Multiply** | **mul** |  | **And Immediate** | **andi** |
|  | **Multiply Word** | **mult** |  | **Or immediate** | **ori** |
|  | **Multiply Unsigned Word** | **multu** |  | **Exclusive or Immediate** | **xori** |
|  | **Multiply and add word to Hi,Lo** | **madd** |  | **Sign-extend half word** | **seh** |
|  | **Multiply and subract word to Hi,Lo** | **msub** |  | **Shift left logical** | **sll** |
| **Data** | **Load word** | **lw** |  | **Shift right Logical** | **srl** |
|  | **Store word** | **sw** |  | **Shift Word Left Logical Variable** | **sllv** |
|  | **Store byte** | **sb** |  | **Shift Word Right Logical Variable SRLV** | **srlv** |
|  | **Load half** | **lh** |  | **Set on less than** | **slt** |
|  | **Load byte** | **lb** |  | **set on less than immediate** | **slti** |
|  | **Store half** | **sh** |  | **move conditional on not zero** | **movn** |
|  | **Move to Hi Register** | **mthi** |  | **move conditional on zero** | **movz** |
|  | **Move to Lo Register** | **mtlo** |  | **Rotate Word Right Variable** | **rotrv** |
|  | **Move from Hi Register** | **mfhi** |  | **Rotate word right** | **rotr** |
|  | **Move from Lo Register** | **mflo** |  | **Shift word right arithmetic** | **sra** |
|  | **Load Upper Immediate** | **lui** |  | **Shift Word Right Arithmetic Variable** | **srav** |
| **Branches** | **branch if greater than or equal to zero** | **bgez** |  | **Sign-Extend Byte** | **seb** |
|  | **branch on equal** | **beq** |  | **Set on Less Than Immediate Unsigned** | **sltiu** |
|  | **branch on not equal** | **bne** |  | **Set on Less Than Unsigned SLTU** | **sltu** |
|  | **branch on greater than zero** | **bgtz** |  |  |  |
|  | **branch on les than or equal to zero** | **blez** |  |  |  |
|  | **branch on less than zero** | **bltz** |  |  |  |
|  | **Jump** | **j** |  |  |  |
|  | **jump register** | **jr** |  |  |  |
|  | **jump and link** | **jal** |  |  |  |

**Datapath**



The datapath figure above is slightly different from the single cycle datapath we covered in the class. This datapath is actually a pipelined version. We will partition the execution of an instruction into 5 stages. These stages are:

* Instruction Fetch (IF)
* Instruction Decode (ID)
* Execute (EX)
* Memory Access (MEM)
* Write Back (WB)

Between each stage we have a register file named as IF/ID, ID/EX, EX/MEM and MEM/WB.

The size of the register file depends on the number of bits transferred from one stage to another.

For example:

* IF/ID register is storing the 32 bit PC value and the 32-bit instruction read from the instruction memory.
* ID/EX register stores all the control signal generated by the Controller along with the two regiser values read from the register file, sign extended offset field, potential destination registers (Rd – I[15:0] and Rt – I[20-16])
* EX/MEM stores the control signal needed by the subsequent stages (note that execution stage control signals have already been used to control the MUXes in the EX stage.) the branch target address (output of adder), ALU output, zero flag, and destination register (output of 2:1 mux controlled by RegDst).
* MEM/WB stores control signals needed by the WB stage, ALU output and the Data memory output along with the destination register.

In this design each stage takes 1 clock cycle, meaning a single instruction will go through 5 phases and take 5 clock cycles to complete. You will still be reading an instruction in each clock cycle. The execution flow for this datapath will be covered by Lab 11.

Other than the registers introduced between 5 stages, the entire datapath is exactly same as the single cycle datapath.

After completing the ALU design, your focus should be on the “Controller” design. A complete list of control signals required to manage your datapath is essential before attempting to implement the Controller module. Otherwise you may risk the redesign of the entire controller.

**TASK – Datapath Design for Arithmetic and Logical Operations**

* **Method:**
* Design and build a datapath to execute the arithmetic and logic operations listed in Table 1.
* Conduct post routing simulation for functional verification of each instruction
* Verify that you are able to run a sequence of these instructions in post-routing simulation.
* You need to create your own test program in assembly that includes all the arithmetic and logical operations
* Translate this program to binary form and initialize your instruction memory with this test program.
* You will need to learn how to initialize your instruction memory using the MIPSHelper tool (**more information below, source code given in this folder**).
* Example:
* We strongly recommend you to generate a program with dependent sequence of operations covering all the arithmetic and logical operations. For example:

addi **$t1**, $zero, 6 # t1 = 6

nop # 5 nop instructions

nop # in between each instruction

nop # must be inserted

nop # later we will address this

nop # and remove all nops

addi **$t2**, $zero 8 # t2 = 8

nop

nop

nop

nop

nop

sub **$t3**, **t2,** **t1** # t3 = 8-6 = 2

nop

nop

nop

nop

nop

sll **$t4**, **$t3**, 3 # t4 = 2 << 3 = 16

nop

nop

nop

nop

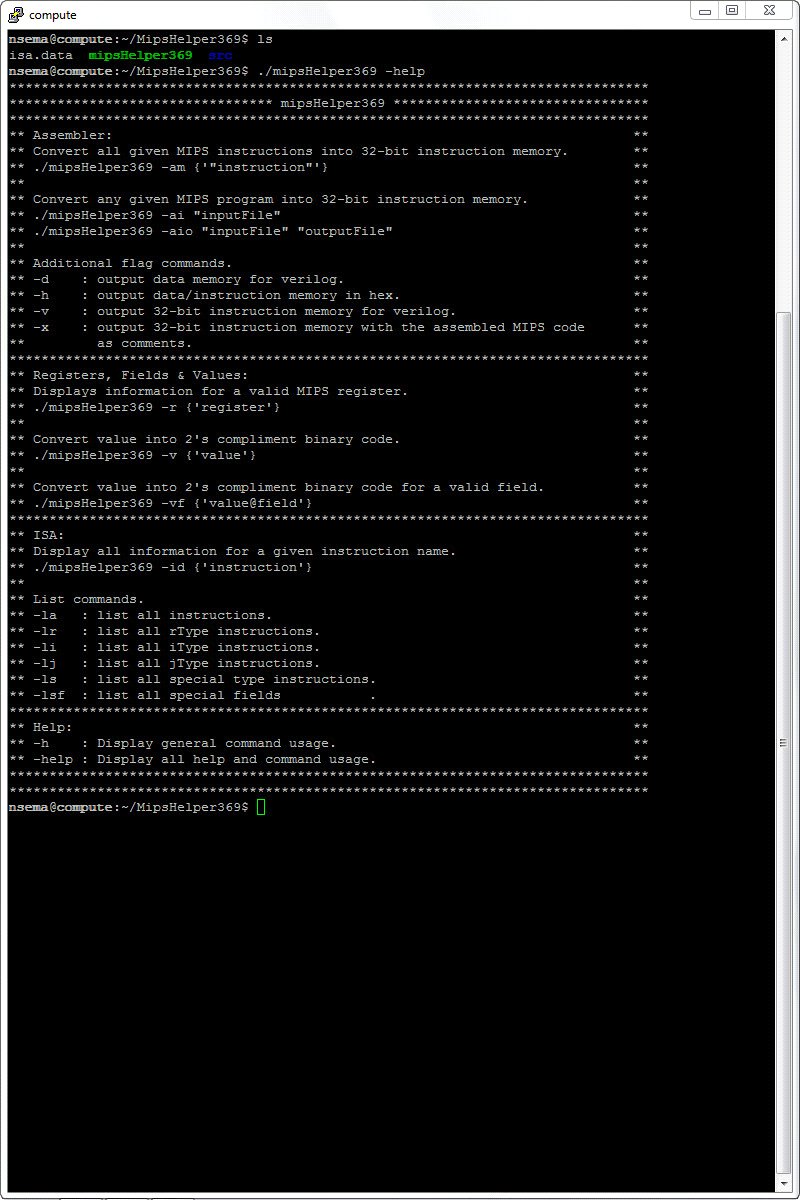
nop

srl $t5, **$t4**, 2 # t5 = 4

Initialize your registers using “add” and addi” instructions. Note the color coded dependency between the sequential instructions.

* Run the MIPShelper tool to generate your instruction memory module
* Synthesize and run the program in post-routing simulation
* After verifying in post-routing simulation, load your bitstream onto the FPGA
* Display the current PC value and the value that is written into the register file for that instruction on the FPGA as the datapath sequences through the instructions
* You need to use the “**Two4DigitDisplay**” module provided in this folder. This module displays two 4-digit numbers.
* It will be convenient if each operation results with a value that can be represented with at most 4 digits (base 10).
* Multiplication operation generates a 64 bit output that is written into a “hi” and “lo” registers outside the register file. You need to display the content of the “lo” register only with 4 digits on the FPGA for the multiplication operation.
* Assume that first instruction of the test code corresponds to PC value of 0.
* **Demonstration:**
* Functional verification by displaying the value written into the register file after executing each instruction on the FPGA on **October 18** during your designated lab time.
* During the demonstration day you will be given an assembly program for testing the arithmetic and logical operations.

**mipsHelper** is an assembler tool implemented by Nathaniel Sema while he was taking the ece369a. Tool generates Verilog-based instruction and data memories rapidly for a given MIPS assembly code. Refer to the “mipsHelper” folder for the tutorial and installation instructions (MipsHelper369\_Tutorial.pdf).



The screenshot shows that launching -help command lists the capabilities of the tool

**Known bugs:** This bug is not relevant at this stage. Branch instructions that have negative offset values need to be adjusted by 4 manually.

* **Deliverable:**
* Turn in only .v files used in your design (not zipped, no other format will be accepted) using designated dropbox on D2L (deadline: **October 18, 2:00pm**)
* **Penalty Conditions:**
* Percent effort not reported (20% penalty)
* Late submission or late demonstration (15% per day)
* Submitting files in a folder or in compressed form (zip/tar). (10% penalty)
* Changing the file name or extension. (10% penalty)
* Failing to demonstrate (80% penalty)
* Design works in behavioral simulation but fails to synthesize (70% penalty)
* Design works in behavioral simulation, synthesizes with warnings but post-routing simulation fails (60% penalty)
* Design works in post-routing simulation, but FPGA fails to display (20% penalty)
* Both team members must attend the demonstration (missing partner receives maximum of 50)
* Keep in mind that routing process will take time. Therefore, post routing simulations must be ready before the demonstration starts so that the demonstration is completed within 10 minutes.

**PHASE 2: LABS 15-17 (150 points) Oct 30**

* Objective:
* Update your datapath to support Data Transfer, Branch and Jump instructions. Show that you are able to run a sequence of these instructions.
* **Method:**
* Include the MUXes and other needed datapath components
* Revise the controller
* Conduct post routing simulation for functional verification of each instruction
* Create your own test program in assembly that includes all the data transfer, branch and jump instructions listed in Table 1.
* Assume that first instruction of the test code corresponds to PC value of 0.
* Translate this program to binary and initialize your instruction memory with this test program.
* Synthesize and run the program in post-routing simulation
* After verifying in post-routing simulation load your bitstream onto the FPGA
* Verify by displaying the PC value and the value written into the register file for each instruction
* If no value is written into register file, only display the PC value.
* Suggested method: We strongly recommend you to generate a program with dependent sequence of operations covering all the operations.

For example:

PC=0 loop: addi $t0, $zero, $zero # t0=0, display 0, 0

nop

nop

nop

nop

nop

PC=24 addi **$t1**, $zero, 6 # t1= 6, display 24, 6

nop

nop

nop

nop

nop

PC=48 addi **$t2**, $zero 10 # t2 = 10, display 48, 10

nop

nop

nop

nop

nop

PC=72 sw **t1**, 0($t0) # display 72, (no register written)

nop

nop

nop

nop

nop

PC=96 sw **t2,** 4($t0) # display 96,

nop

nop

nop

nop

nop

PC=120 lw t3, 0($t0) # t3 = 6, display 120, 6

nop

nop

nop

nop

nop

PC=144 lw t4, 4($t0) # t4 = 10, display 144, 10

nop

nop

nop

nop

nop

PC=168 j loop # display 168,

* Initialize your memory with a sequence of sw operations and then read from the memory with the lw. Different from the phase-1, display both current PC and the value written into the destination register. For the “sw”, “branch” and “jump” type of instructions there is no destination register therefore you should display the PC value and 0.
* It will be convenient if each operation results with a value that can be represented with at most 4 digits.
* **Demonstration:**
* During the demonstration day you will be given an assembly program for testing the Data Transfer, Branch and Jump instructions.
* Functional verification by displaying the value written into the register file and the PC value of that specific instruction on the FPGA
* Due by **October 30**
* **Deliverable:**
* Turn in only .v files used in your design (not zipped, no other format will be accepted) using designated dropbox on D2L (deadline: **October 30, 2:00pm**)
* **Penalty Conditions:**
* Percent effort not reported (20% penalty)
* Late submission or late demonstration (15% per day)
* Submitting files in a folder or in compressed form (zip/tar). (10% penalty)
* Changing the file name or extension. (10% penalty)
* Failing to demonstrate (80% penalty)
* Design works in behavioral simulation but fails to synthesize (70% penalty)
* Design works in behavioral simulation, synthesizes with warnings but post-routing simulation fails (60% penalty)
* Design works in post-routing simulation, but FPGA fails to display (20% penalty)
* Both team members must attend the demonstration (missing partner receives max of 50)
* Keep in mind that routing process will take time. Therefore, post routing simulations must be ready before the demonstration starts so that the demonstration is completed within 10 minutes.