Homework #4
CS 152

5.2

RegDst = 1: Any result written to the register file will be written to the RD location. All r-type instructions would continue to work fine but any i-type instruction would write in the wrong location. JAL will no longer work as well since it will link the pc to the wrong address.

ALUSrc = 1: The ALU will always take its second operand from the immediate field. This does not affect i-type instructions but all the r-type instructions will read the wrong value as its operand.

MemToReg = 1: The write back to the register file will always take data from the memory. All r-type instructions and most i-type (except for memory) will not work since it will be writing back garbage from that output.

Zero = 1: This will always say the two operands to the ALU are equal. Thus, roughly half of all BEQ instructions would still work (those who are actually equal) and the ones who are unequal will lead to branches that should not occur. Non-branching instructions are unaffected.

5.11

RegDst could possibly be replaced with ALUSrc since ALUSrc is 0 only for r-type instructions which are the only ones who need to write back to the register RD and vice versa. Branch cannot be replaced. MemWrite can be replaced combinational logic that looked into RegWrite, MemToReg and Branch. RegWrite can be replaced if you kept MemWrite and used combinational logic with MemToReg and Branch.

5.17

JAL would require changes to the following things:

-Mux into Write register port of Register File needs to be expanded to take 31 as a destination for the link meaning the control signal needs to be 1 bit bigger.

-Mux into Write Data port of Register File needs to be expanded to take PC as an input.

The state diagram would involve just one more state which is the JAL stage which is similar to the jump stage except it also writes back the PC into register 31. This instruction would just require 3 cycles to complete.

5.18

Assume the instruction format just has an opcode and two operands that you wish to swap which would be in the RS and RT field. The following changes would need to be made:

-Mux into Write register port of Register File needs to be expanded to take RS as an input.

-Mux into Write Data port of Register File needs to be expanded to take two more inputs: one from register A and the other from register B.

The state diagram would need two more states. One to write A to RT field and another to write B to the RS field. This is similar to the write back state of r-type instructions except you perform it twice to swap the registers. This instruction would take 4 cycles to finish.

5.20

Assuming the RT field of the instruction is 0 since it is not being used, the following changes need to be made:

-Mux into PC needs to be expanded to take MemData coming out of the Memory.

The state diagram would include just one extra state. One state needs to be added after the MemRead state so that the MemRead state can branch to where currently goes which writes back to memory and this new state that would write to the PC. This new state would just be similar to the jump state except it would set the PCSource to take its input from MemData. Since there is more branching, the dispatch unit would need to change so that it can distinguish between a load word and a jump mem. This would require 5 cycles.

5.21

The following changes need to be made:

-A new Mux needs to be placed in front of Read register 1 port of Registers so it can read from RS and the new location of the other source register.

-Mux of ALUSrcB needs to be expanded to take ALUOut as an input.

-Register B would need an enable signal so it can be disabled from getting a new value.

The state diagram would be similar to an r-type instruction and would require 3 new states. After the decode, the first new state would add the RS and RT field and store the result into ALUOut while reading out the new register value and placing it into A (this can work since the register hardware is separate from the execute hardware; also remember to disable B). The second state would add the new source register to ALUOut (selected through the expanded mux). The last state would be a simple write back state from the ALUOut back to the register file into register RD. This instruction would take 5 cycles.

5.22

Since the RT field in a JR instruction is 0, RS + RT = RS which is the destination address. Then route that signal through the PC Source Mux directly from the ALU so it does not waste a cycle going through ALUOut. The changes to the state diagram would include just one new state (JR). This state would is similar to the current Decode state except both ALUSrc should be 0 take in values from the register file. Also set PC Src = 0 and PC Write high in this state.

5.24

A lot of answers will vary here depending on how you deal with instructions that are not explicitly supported in the current multi-cycle datapath. Assuming that we support all the instructions and just put the instruction in the right category (for example, any set less than instruction is pretty much the same as an add and would take 4 cycles), we get the following CPIs:

M1: (9+17+1+2+5+2+2+1+1+1+2)*4*.01 alu insts + (21+1+1)*5*.01 mem loads + (12+1)*4*.01 mem write + (9+8+1+1+1+1)*3*.01 branching = 3.98 CPI

M2: (9+17+1+2+5+2+2+1+1+1+2)*3*.01 alu insts + (21+1+1)*4*.01 mem loads + (12+1)*4*.01 mem write + (9+8+1+1+1+1)*3*.01 branching = 3.36 CPI

M3: (9+17+1+2+5+2+2+1+1+1+2)*3*.01 alu insts + (21+1+1)*3*.01 mem loads + (12+1)*3*.01 mem write + (9+8+1+1+1+1)*3*.01 branching = 3

The MIPS for each of these processors are:
M1: 500/3.98 = 126 MIPS
M2: 400/3.36 = 119 MIPS
M3: 250/3 = 83 MIPS

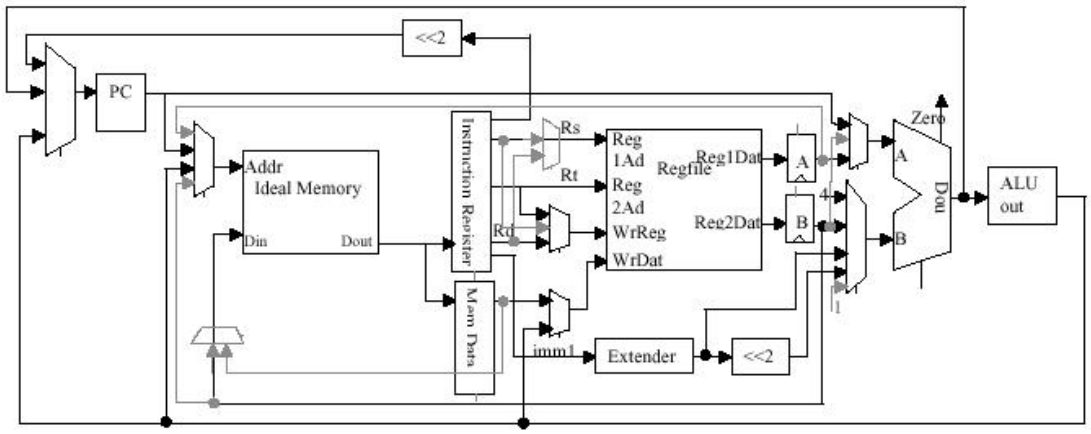
Based on current situations, M1 would be the fastest machine. With more memory access (particularly loads) instructions, M2 and M3 should pick up on speed with M2 most likely to edge out unless most of the instructions become loads.

5.27

Block Copy - Multicyle & Microcode

Using the multicycle datapath from the textbook (Fig 5.33) changes are made below in aqua dashed lines. The basic RTL for the block copy instruction using this modified datapath is also referenced below. When instruction is fetched you have register $t1, $t2, and $t3. Make sure that the instruction register has a write enable so that the register file addresses are kept around. First you want to read R($t1) and R($t2) and get these values into the register for Reg1Data (register A) and Reg2Data (register B). Make sure these registers have write enables also. Next you want to use the value in register A to be used as address for memory to get the first value to be copied. At the same time you can use increment the register A value using the ALU. In the next cycle you can write the value back to memory using the memory address in register B. At the same you can write the newly incremented source address back into the register file, and use the ALU to increment the destination address in register B, and to get the length of the array into register A. In this final step you can write the value of incremented destination address back into the regfile. In the same cycle you can also decrement the length by one and check to see if it's negative. If so then block copy is done, if not it needs to write the length back into the register file and continue the loop. Another way is to store the length in it's own separate register (RegC, not shown in diagram), instead of ALUout, so that you won't need to write it back into the regfile. This saves you a cycle on every loop.
In hardware it would take 4cycles for the last loop + (5cycles)*(length -1) + 2cycles for instruction fetch and decode. The five cycles for the loop comes from having to store the length back into R($3). [If use extra register (RegC) to store length, instead of storing it to ALUout then to R($3); it's just (4cycles)*(length) + 2cycles.] In the software version it would take 3cycles for the beq + (length-1)*(24cycles). The 24 cycles are based on the multicycle state diagram (Fig 5.42) where lw is 5cycles, sw is 4cycles, each addi is 4 cycles, and branch is 3cycles. (i.e. 5+4+(4*3)+3)

Basic RTL for Block Copy:
regA <- R($t1), regB <- R($t2)
MemDataReg <- M(regA), ALUout <- regA + 4
Mem <- MemDataReg at M(regB), R($t1) <- ALUout, ALUout <- regB + 4, regA <- R($t3)
R($t2) <- ALUout, ALUout <- regA - 1, decide if should loop again



5.28

For bcp, we need to add a field for the mux going into the Din of the ideal memory, MemSrc, with the values B or MemDataReg. We need to add a new field RegSrcA with values Rs or Rd and a new field for the counter, counter with possible values of load and dec, adding two bits (because we need a nop). Also, we need to add a value to the SRC1 field, ALU_2d, adding onebit of control to this field. For the sequence field, we need to a add a branch, which will fetch if the counter is zero, but jump back to the loop

5.29

The value Rs in the field SRC1 has been changed to RegA to avoid confusion when we try to readthe Rd register instead. The idea in this solution is to only use an extra register so that we can keep the next address we want to read from and the next address we want to write to in two different registers. In addition, the timing works out so that you read from an offset off of Rs in one cycle, store the value in the MemDataRegister, then feed that value back into the memory and write from an offset off of Rd. Of course, we need a counter that is loaded to the value Reg[Rt] that decrements only when we want it to. When the counter reaches 0, we know we have finished our looping and we can fetch the next instruction.

Label	*ALU*	*SRC1*	*SRC2*	*ALUDst*	*Mem*	*MemReg*	*PCWr*	*MemSrc*	*rsorrd*	*counter*	*Seq*
Fetch	*Add*	*PC*	*4*		*ReadPC*	*IR*	*ALU*				*Seq*
Dispatch	*Add*	*PC*	*ExtShft*						*rs*		*Dispatch*
Jal				*Ra-PC*			*Instr25to0*				*Fetch*
Addiu	*Add*	*RegA*	*Extend*								*Seq*
				Rt-ALU							*Fetch*
Bcp	*Add*	*RegA*	*0*						*Rd*	*Load*	*Seq*
	Add	*RegA*	*0*		*ReadALU*						*Seq*
Loop	*Add*	*ALU_2d*	*4*		*WrALU*			*feedback*		*Dec*	*Seq*
	Add	*ALU_2d*	*4*		*ReadALU*						*Branch*