Course Name: *Computer Architecture Lab*

Course Number and Section: **14:332:333:*04***

Experiment: *4 – CPU Structure, Parallel Programming & Hazards, Exceptions & Interrupts*

Lab Instructor: *Christos Mitropoulos*

Date Performed: *October 27th, 2016*

Date Submitted: *November 17th, 2016*

Submitted by: *FAHD HUMAYUN – 168000889 – fh186*

----------For Lab Instructor Use ONLY----------

GRADE: _____

COMMENTS:

Electrical and Computer Engineering Department
School of Engineering
Rutgers University, Piscataway, NJ 08854

Introduction:

Some of the assignments are based on understanding the difference between pipelined and no-pipelined processors, understanding how pipelining works, understanding the hazards in pipelining, the types of hazards, and how to prevent hazards by using NOP (stall/bubble), forwarding, and reordering the instructions. The other assignments are based on understanding the mechanism of exceptions and interrupts handler in MIPS.

Assignment – 1:

The assignment requires to show the values of the control signal lines for the instructions given. Control Signal lines are:

$RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp, Jump$

The basic MIPS instructions are:

- The memory-reference instructions
  - $(I-type: load\ word\ (lw)\ and\ store\ word\ (sw))$.
- The arithmetic-logical instructions
  - $(R-type: add, sub, AND, OR, and\ slt)$
- The branch $(I-type: beq, bne)$ and the jump (j) instructions.

These control lines are determined by the instruction's type and then the particular instruction is processed accordingly.

These signals control the behavior of the main components of the datapath i.e. Register File, Arithmetic-Logical Unit, Data Memory, and also the multiplexors in the datapath.

The table below shows the values of the control signals for each given instruction:

Instruction	*RegDst*	*ALUSrc*	*MemtoReg*	*RegWrite*	*MemRead*	*MemWrite*	*Branch*	*ALUOp*	*Jump*
addiu \$t0, \$t0,10	*0*	*1*	*0*	*1*	*0*	*0*	*0*	*00*	*0*
lb \$t1, 32(\$t0)	*0*	*1*	*1*	*1*	*1*	*0*	*0*	*00*	*0*
or \$t2, \$t0, \$t1	*1*	*0*	*0*	*1*	*0*	*0*	*0*	*10*	*0*
bne \$t2, \$t0, *QUIT*	*X*	*1*	*X*	*0*	*0*	*0*	*1*	*01*	*0*
j 2500	*X*	*X*	*X*	*0*	*0*	*0*	*X*	*XX*	*1*

Assignment – 2:

The assignment is based on understanding the pipelining, with and without hazard by using NOP instructions, instruction reordering, and forwarding.

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

There are five stages/steps:

1. **Instruction Fetch** (***IF***)**:** *Fetch instruction from memory* – All instructions start by using the program counter (PC) to supply the instruction address to the instruction memory.

2. **Instruction Decode** (***ID***)**:** *Read registers while decoding the instruction* – After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. All MIPS instructions are same length, which makes it much easier to fetch instructions in the first pipeline stage (IF) and to decode them in the second pipeline stage (ID).

3. **Execution** (***EX***)**:** *Execute the operation or calculate an address* – Once the register operands have been fetched, they can be operated on to compute a memory address by using the sign-extended 16-bit number (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch)

4. **Memory** (***MEM***): *Access an operand in data memory* – Memory operands appear only in load or store instructions in MIPS. Execution (EX) stage calculates the memory address and then access memory (MEM) in this stage.

5. **Write Back** (***WB***): *Write the result into a register* – The result from ALU or memory is written back into a register file.

Pipeline Hazards – Situations in pipelining when the next instruction cannot execute in the following clock cycle, these events are called *hazards.*

There are three types of hazards:

1. **Structural Hazard:** When a pipeline instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

2. **Data Hazards:** When a planned instruction cannot execute in the proper clock cycle because the data that is needed to execute the instruction is not yet available. Arises from the dependence of one instruction on an earlier one that is still in the pipeline.

3. **Control Hazards:** When the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed.

Part 1 & 2:

Table of execution with five pipeline stages (hazards ignored) and register values.
Initial register values 2, 5, 8, 2, 4, and 0 for $\$t0, \$t1, \$t2, \$t3, \$t4$, and $\$t5$, respectively.

Instruction	**Clock Cycles**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
lw $\$t0, 0(\$t3)$	*IF*	*ID*	*EX*	*MEM*	*WB*				
add $\$t1, \$t0, \$t2$		*IF*	*ID*	*EX*	*MEM*	*WB*			
sub $\$t3, \$t3, \$t1$			*IF*	*ID*	*EX*	*MEM*	*WB*		
addi $\$t4, \$t4, 4$				*IF*	*ID*	*EX*	*MEM*	*WB*	
sub $\$t5, \$t5, \$t4$					*IF*	*ID*	*EX*	*MEM*	*WB*
Register Values									
$\$t0$	*2*	*2*	*2*	*2*	*2/N*	*N*	*N*	*N*	*N*
$\$t1$	*5*	*5*	*5*	*5*	*5*	*5/10*	*10*	*10*	*10*
$\$t2$	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*
$\$t3$	*2*	*2*	*2*	*2*	*2*	*2*	*2/-3*	*-3*	*-3*
$\$t4$	*4*	*4*	*4*	*4*	*4*	*4*	*4*	*4/8*	*8*
$\$t5$	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0/-4*

Where, N represents a number loaded from the memory, and X/Y represents the value of register being changed from the previous to new value in the write back stage.

Part 3:

Instruction	Clock Cycles														
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
lw $t0, 0($t3)	*IF*	*ID*	*EX*	*MEM*	*WB*										
NOP		*Stall*	*Stall*	*Stall*	*Stall*	*Stall*									
NOP			*Stall*	*Stall*	*Stall*	*Stall*	*Stall*								
add $t1, $t0, $t2				*IF*	*ID*	*EX*	*MEM*	*WB*							
NOP					*Stall*	*Stall*	*Stall*	*Stall*	*Stall*						
NOP						*Stall*	*Stall*	*Stall*	*Stall*	*Stall*					
sub $t3, $t3, $t1							*IF*	*ID*	*EX*	*MEM*	*WB*				
addi $t4, $t4, 4								*IF*	*ID*	*EX*	*MEM*	*WB*			
NOP									*Stall*	*Stall*	*Stall*	*Stall*	*Stall*		
NOP										*Stall*	*Stall*	*Stall*	*Stall*	*Stall*	
sub $t5, $t5, $t4											*IF*	*ID*	*EX*	*MEM*	*WB*
Register Values															
$t0	*2*	*2*	*2*	*2*	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*N*	*N*
$t1	*5*	*5*	*5*	*5*	*5*	*5*	*5*	*N+8*	*N+8*	*N+8*	*N+8*	*N+8*	*N+8*	*N+8*	*N+8*
$t2	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*
$t3	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*-N-6*	*-N-6*	*-N-6*	*-N-6*	*-N-6*
$t4	*4*	*4*	*4*	*4*	*4*	*4*	*4*	*4*	*4*	*4*	*4*	*8*	*8*	*8*	*8*
$t5	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*-8*

Part 4:

If the instruction that follows the current instruction has dependencies i.e. there is a register or data that is being needed for the instruction in the next cycle but it is not yet available from the current cycle instruction, then instruction reordering can be done in such a way that the instruction in next cycle is replaced with an instruction that does not dependent on the current instruction in the clock cycle. As in the code given the second instruction in the code is dependent upon the first (i.e. $\$t0$ value is being changed in the first instruction and then the same register $\$t0$ is used in the second instruction, but, the data will be available in the write back stage or in the 5^{th} clock cycle, but the second instruction would need the data in the 3^{rd} clock cycle, so the second instruction can be replaced with the fourth instruction *addi* $\$t4, \$t4, 4$ in the code given because that instruction is not dependent upon the first instruction. Now, there is no other instruction that is independent, so, the *add* $\$t1, \$t0, \$t2$ instruction can be placed in the next clock cycle, but, still it would need the data of the first instruction in 4^{th} clock cycle, and as mentioned the data will be available from the *lw* $\$t0, 0(\$t3)$ in the 5^{th} clock cycle, therefore, one NOP/stall is still needed. Similarly, the instruction *sub* $\$t3, \$t3, \$t1$ is dependent upon the previous instruction, so, it is replaced with the *sub* $\$t5, \$t5, \$t4$ instruction, and then one NOP/stall is needed as the data needed would not be available in the ID stage of the last instruction. The table below shows the result of instruction reordering and necessary NOP.

From the table it can be seen that the clock cycles has been reduced from 15 to 11 because of the reordering of instructions compared to the previous case with NOP but with no reordering.

Instruction	Clock Cycles										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
lw $\$t0, 0(\$t3)$	*IF*	*ID*	*EX*	*MEM*	*WB*						
addi $\$t4, \$t4, 4$		*IF*	*ID*	*EX*	*MEM*	*WB*					
NOP			*Stall*	*Stall*	*Stall*	*Stall*	*Stall*				
add $\$t1, \$t0, \$t2$				*IF*	*ID*	*EX*	*MEM*	*WB*			
sub $\$t5, \$t5, \$t4$					*IF*	*ID*	*EX*	*MEM*	*WB*		
NOP						*Stall*	*Stall*	*Stall*	*Stall*	*Stall*	
sub $\$t3, \$t3, \$t1$							*IF*	*ID*	*EX*	*MEM*	*WB*
Register Values											
$\$t0$	*2*	*2*	*2*	*2*	*N*	*N*	*N*	*N*	*N*	*N*	*N*
$\$t1$	*5*	*5*	*5*	*5*	*5*	*5*	*5*	*N+8*	*N+8*	*N+8*	*N+8*
$\$t2$	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*
$\$t3$	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*-N-6*
$\$t4$	*4*	*4*	*4*	*4*	*4*	*8*	*8*	*8*	*8*	*8*	*8*
$\$t5$	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*-8*	*-8*	*-8*

Part 5:

Forwarding – when result is passed forward from an earlier instruction to a later instruction.

The data in an arithmetic-logical instruction is actually available after the EX stage i.e. as soon as the ALU does an operation, the result can be supplied as an input to the ALU in the EX stage of the next instruction. Similarly, in memory-reference instruction the data is available after the MEM stage, so, the data can be passed as an input for the next instruction, and do not need to wait till the WB stage.

Adding extra hardware to retrieve the missing item early from the internal resources is called forwarding or bypassing.

Forwarding paths are valid only if the destination stage is later in time than the source stage. There cannot be a valid forwarding path from the output of the memory access stage (MEM) in $lw\ \$t0, 0(\$t3)$ instruction as the input to of execution stage (EX) stage of $add\ \$t1, \$t0, \$t2$ instruction if the EX stage is back in time from the MEM stage.

Forwarding cannon prevent all pipeline stalls. There might still be some stalls needed, but, it will significantly decrease the number of stalls in a program. So, even with forwarding a stall of one stage is needed for a load-use data hazard if there is an instruction next to the load instruction that is dependent on it.

Two tables are given below that shows the forwarding with NOP and no reordering and with NOP and reordering:

Table of execution with pipeline stages with NOP and no reordering of instructions:

Instructions	**Clock Cycles**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
lw $t0, 0($t3)	*IF*	*ID*	*EX*	*MEM*	*WB*					
NOP		*Stall*	*Stall*	*Stall*	*Stall*	*Stall*				
add $t1, $t0, $t2			*IF*	*ID*	*EX*	*MEM*	*WB*			
sub $t3, $t3, $t1				*IF*	*ID*	*EX*	*MEM*	*WB*		
addi $t4, $t4, 4					*IF*	*ID*	*EX*	*MEM*	*WB*	
sub $t5, $t5, $t4						*IF*	*ID*	*EX*	*MEM*	*WB*
Register Values										
$t0	*2*	*2*	*2*	*2*	*N*	*N*	*N*	*N*	*N*	*N*
$t1	*5*	*5*	*5*	*5*	*5*	*5*	*N+8*	*N+8*	*N+8*	*N+8*
$t2	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*
$t3	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*-N-6*	*-N-6*	*-N-6*
$t4	*4*	*4*	*4*	*4*	*4*	*4*	*4*	*4*	*8*	*8*
$t5	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*-8*

Table of execution with pipeline stages with NOP and reordering of instructions:

Instructions	**Clock Cycles**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
lw $\$t0, 0(\$t3)$	*IF*	*ID*	*EX*	*MEM*	*WB*				
addi $\$t4, \$t4, 4$		*IF*	*ID*	*EX*	*MEM*	*WB*			
add $\$t1, \$t0, \$t2$			*IF*	*ID*	*EX*	*MEM*	*WB*		
sub $\$t5, \$t5, \$t4$				*IF*	*ID*	*EX*	*MEM*	*WB*	
sub $\$t3, \$t3, \$t1$					*IF*	*ID*	*EX*	*MEM*	*WB*
Register Values									
$\$t0$	*2*	*2*	*2*	*2*	*N*	*N*	*N*	*N*	*N*
$\$t1$	*5*	*5*	*5*	*5*	*5*	*5*	*N+8*	*N+8*	*N+8*
$\$t2$	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*	*8*
$\$t3$	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*2*	*-N-6*
$\$t4$	*4*	*4*	*4*	*4*	*4*	*8*	*8*	*8*	*8*
$\$t5$	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*-8*	*-8*

In the above tables the arrows show the forwarding.

The forwarding is basically done using the pipeline registers that stores data, registers and other necessary information in each stage.

In the table of execution with NOP and reordering of instructions, the data of $\$t0$ is available in the MEM stage of *lw* $\$t0, 0(\$t3)$ instruction, so it is forwarded from MEM/WB pipeline register as an input to the EX stage of *add* $\$t1, \$t0, \$t2$ instruction as in that clock cycle

$$MEM/WB.RegisterRd = ID/EX.RegisterRs$$

Which means the data should be forwarded. Similarly, for the other instructions.

Comparing with NOP and instruction reordering:
The number of clock cycles for each one of them is as follow:

- NOP (only): **15** clock cycles.
- Instruction reordering (with NOP): **11** clock cycles.
- Forwarding
    - (with NOP but no reordering): **10** clock cycles.
    - (with NOP and reordering): **9** clock cycles.

The number of clock cycles in part 1 with ignoring hazards was **9** clock cycles but the result for the register values was not according to what was expected and it can be seen that forwarding (with NOP and reordering) the instructions were completed in **9** clock cycles with the result desired. So, clearly in the code given the forwarding was really important for decreasing the number of clock cycles.

Assignment – 3:

1. The number of cycles for the program to execute:
   5 cycles for one instruction, then adding one clock cycle for each instruction executed so, the number of instructions executed after the $addiu\ \$t0, \$t0, 3$ are 6 instructions till the loop ends, because $\$t0$ is initialized to 3 and is decremented each time in a loop and condition is checked, so, it will take total of 6 instructions (both the decrement and branch condition combined) till the loop ends. So, total number of cycles would be 5 + 6 = **11 cycles.**
   If the initial value of $\$t0$ is assumed to be 0, then without hazard handling the program will execute in **7** clock cycles, because, the data of $\$t0$ from the first instruction $addiu\ \$t0, \$t0, 3$ will be available in the fifth stage (WB) or 5th cycle, while for the next instruction $addi\ \$t0, \$t0, -1$ the data is needed at the 3rd cycle (ID stage), so, in the 3rd cycle the value of $\$t0$ will still be the initially assumed value i.e. 0, so that value will be decremented in EX stage and this new value i.e. -1 will be updated in the 6th clock cycle. Now, the $bne\ \$t0, \$0, loop$ instruction will read the data of register in the 4th clock cycle (ID stage) and at during that clock cycle the value of $\$t0$ will still be 0 because the value of $\$t0$ is updated in 5th cycle from the first instruction and in 6th cycle from the second instruction, so, in the 4th cycle as the value of $\$t0$ is 0 which means in the EX stage of the third instruction condition would not be true and the loop would end, therefore, it will take total of three instruction to execute which means 5 cycles for the one instruction and then 2 additional cycles for the other two instruction, hence, 5 + 2 = **7 cycles.**

2. Execution table (expected, otherwise it would end in 7 cycles as mentioned above):

Instructions	**Clock Cycles**										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
$addiu\ \$t0, \$t0, 3$	*IF*	*ID*	*EX*	*MEM*	*WB*						
$addi\ \$t0, \$t0, -1$		*IF*	*ID*	*EX*	*MEM*	*WB*					
$bne\ \$t0, \$0, loop$			*IF*	*ID*	*EX*	*MEM*	*WB*				
$addi\ \$t0, \$t0, -1$				*IF*	*ID*	*EX*	*MEM*	*WB*			
$bne\ \$t0, \$0, loop$					*IF*	*ID*	*EX*	*MEM*	*WB*		
$addi\ \$t0, \$t0, -1$						*IF*	*ID*	*EX*	*MEM*	*WB*	
$bne\ \$t0, \$0, loop$							*IF*	*ID*	*EX*	*MEM*	*WB*

Type of hazards in the given code:

Register $\$t0$ is used in all of the three instructions, and in the execution table above without NOP or forwarding the data of $\$t0$ will be available in the fifth stage (WB) of each instruction. So, clearly there are ***data hazards***.

The branch instruction is used in the given code, so, with branch instruction there is ***control hazard,*** because, the instruction executed/fetched next might not be the one that is needed, for example if the branch condition is true then the decrement instruction should be the next instruction fetched, but, it could be possible that the branch condition is not true and then the instruction when the loop ends has to be fetched.

3. Using NOP instructions to avoid the hazards (using *X* to represent stall).

$addiu\ \$t0, \$t0, 3$
NOP
NOP
$addi\ \$t0, \$t0, -1$
NOP
NOP
$bne\ \$t0, \$0, loop$
NOP

The last NOP instruction after the branch instruction is to simplify what actually happens, at least during the first clock cycle immediately following the branch. To flush instructions in the IF stage, IF.Flush (control line) zeros the instruction field of the IF/ID pipeline register. Clearing the register transforms the fetched instruction into a NOP, an instruction that has no action and changes no state.

4. Instruction forwarding in the given code.
   Instruction forwarding in this code plays a very important role, because, if the data hazards are to be avoided using NOP then the delay increases and if the number of iterations of the loop is increased it means that there will be 4 to 5 additional cycles in each loop because of which the program will take longer to execute and will consume too much of time. Now, on the other hand if forwarding is used as shown in the execution table with arrows, the program executes in 11 cycles as expected. NOP are not needed with forwarding in this code because the data from earlier instruction is available after the EX stage and is needed in the EX stage of the next instruction,

which, makes it execute perfectly by forwarding from one EX to another EX stage using the EX/MEM and ID/EX pipeline registers.

Assignment – 4:

MIPS use memory-mapped I/O. When using memory-mapped I/O, the same address space is shared by memory and I/O devices. Some addresses represent memory cells, while others represent registers in I/O devices. No separate I/O instructions are needed in a CPU that uses memory-mapped I/O. Instead, we can perform I/O operations using any instruction that can reference memory. On the MIPS – ROM, RAM, and I/O devices are accessed using load and store instructions.

Each register within an I/O controller must be assigned a unique address within the address space. This address may be fixed for certain devices, and auto-assigned for others.

Communicating with a Keyboard Controller:

The keyboard controller consists of two 32-bit registers, of which only few bits are used. The receiver data register resides at the fixed memory address 0xFFFF0004. The low 8 bits of this register contain the ASCII code of the last that was pressed. This register is read-only, and can be accessed with a load instruction.

The Receiver Data register (the last byte is the input byte representing the key pressed):

Unused	*Received byte (8-bits)*

The line number 33 of the program has instruction *lw* $\$t1\ 0xFFFF0004$ that loads the address of the receiver data register in register $\$t1$, and line number 46 with the instruction *lw* $\$t6, 0(\$t1)$ that is used for reading character. A *lbu* instruction would have been better as it makes sure that the 3 high bytes of the destination register are cleared, while, *lw* would copy all 32 bits from the source. The high 3 bytes in the receiver control register are probably read as zeros, but *lbu* is safer.

The receiver controller register is located at the memory address $0xFFFF0000$, and only bits 0 and 1 are used.

The Receiver Control register:

Unused	*I (1-bit)*	*R (1-bit)*

LSB (R, bit_0) is the input ready bit. If it is set to 0, then there is no data waiting to be read from the keyboard and cleared when the receiver data register is read. If it is set to 1, then there is data to be read from the keyboard.

bit_1, I is the output interrupt enable bit. If it is set to 0, then an interrupt is not triggered by the device. If it is set to 1, then an interrupt is triggered whenever new data is ready to be read.

The ready bit in the receiver control register and the entire receiver data register are read-only for the CPU. Trying to change their values for example by using *sw* or *sb* would have no effect.

The line number 32 of the program has instruction $li\ \$t0, 0xFFFF0000$, that loads the address of the receiver control register into $\$t0$, which is then used in line number 43 i.e. $lw\ \$t4, 0(\$t0)$, that is used to wait for receiver ready by checking the LSB of the register which is the input ready bit. This is done by the next two instructions in the program by first masking the LSB and the checking if it is equal to 0 then it means input ready bit is not set and jump back to check again, keeps repeating till input ready bit is changed to 1.

This loop does nothing but *poll* the I/O device until the device becomes ready that is if new input is received, or an output device is done processing previous output. When the device is ready the loop exits and the I/O transaction occurs.

In the program the loop keeps on running and keep checking for the input to be received i.e. polling, and it actually ends when the user enters three lines of codes, because the loop counter is set to 3 and it decrements every time a new line is received as an input. So, after 3 lines of input the loop i.e. polling will end.

The disadvantage of this procedure is that the CPU is completely consumed with polling the device until it is ready, and no useful work can be done until after the I/O operation. Even on a slow CPU, a device like keyboard, will produce about 10 input events in a second, and the keyboard is polled about millions of times between keystrokes.

Communicating with a Display Controller:
The display controller works like the keyboard controller.

The transmitter data register is located at the address $0xFFFF000C$, in the program $\$t3$ in the line number 35 is loaded with this address. The last byte is an output byte representing the data being sent to the terminal

The Transmitter Data register (the last byte is the output byte):

Unused	*Transmitted byte (8-bits)*

The transmitter control register is located at the address $0xFFFF0008$, in the program $\$t2$ in the line number 34 is loaded with this address. It consists of a ready bit (bit_0) and interrupt enable bit (bit_1). If ready bit is 1, then it indicates that the display is ready to receive character, and the ready bit is cleared each time a write to the transmitter data register is performed.

A character is sent to the display by writing it to the transmitter data register which is located at $0xFFFF000C$.

When the character is displayed, the display controller sets the ready bit back to 1.

The transmitter control register is read-only and the transmitter data register is write-only.

This is performed the same way as readying input from keyboard, by using load instruction and to display on the console store instruction is used.

In the program, line number 49 to 51 are used to check if the transmitter is ready – lw $\$t4, 0(\$t2)$ gets the transmitter control register then masks the LSB in next instruction and checks if the bit is 0 or 1 in the following instruction. Line number 52 i.e. sw $\$t6, 0(\$t3)$ is used to display the character on the console, which stores the character in the last byte of the transmitter data register (the address was stored in $\$t3$).

Polling:
Checking each device at regular intervals is called polling, also called programmed I/O. The program is in charge of making sure that the I/O devices are each dealt with properly. Not a good method, as for input devices, most of the time when it is checked there might not be any data to be read –the CPU cycles are wasted. Another drawback could be there is a possibility of missing a data, because some input devices cannot wait for the program. Similarly, for output devices.

Interrupts:
Interrupts are a type of exception, and the three types of exceptions are interrupts, traps, and system calls. Where interrupts are raised by hardware and can happen asynchronously.

Interrupts and system calls are handled by the exception (interrupt) handler, which is the place where control is transferred to when an interrupt occurs. The interrupt handler first saves the state of the running program before redirecting control to sections of code that does the actual handling of that particular interrupt. It is done by a jump table that points to

the interrupt handling routines to jump to. Before the interrupt handler returns, the state of the interrupted program is restored, and the program continues executing as if nothing has happened.

When an interrupt is being handled, all other interrupts are disabled. The interrupts that occurring during the current time are not lost, they are just pending. When interrupts are re-enabled again after completion of the current interrupt, the pending interrupts will assert themselves.

There are priority levels, the current interrupt can only be interrupted by an interrupt whose priority level is higher than it, for example, keyboard interrupt is of higher priority than the hard disk, because the hard disk knows how to wait a while but the keyboard does not, because the user might not be as patient as the hard disk.

When the interrupt happens:

- Hardware will transfer control to address 0x80000080, processor is placed in kernel mode. Interrupts are disabled by default. Previous interrupt enable (bit_3 in status register), and the bit_2 that indicates whether the program is running in user (when set to 1) or kernel (when set to 0) mode – there states are saved in the bit_5 and bit_4, respectively (that are bits for old IE, and UK). Address of interrupted instruction is in EPC.
- Get exception code field from cause register.
- Use the interrupt mask field (bits 8 to 15 or 10 to 15) in the status register to find current priority level and then mask out the lower priority interrupts.
- Enable interrupts in the status register interrupt enable field.
- Jump to appropriate interrupt handler routine based on exception code.
- After the handler is done, return.
- Disable interrupts again.
- Restore the previous interrupt enable, and user/kernel bits in the status register.
- The program resumed after the interruption.

Cause Register:

Bits 31:16	*15:10*	*9:8*	*7*	*6:2*	*1:0*
	$IP_7 : IP_2$	$IP_1\ IP_0$		*Exception Code*	

The cause register provides information about what interrupts are pending ($IP_7 : IP_2$), and the cause of the exception ($exception\ code$).
If exception code is 0 it means an interrupt has occurred, and by looking at the interrupt pending bits the processor would know what specific interrupt occurred.

Status Register:

Bits 31:16	*15:10*	*9:8*	*7:6*	*5:0*
	$IM_7 : IM_2$	$IM_1 : IM_0$		$UK_{old}\ IE_{old}\ UK_{prev}\ IE_{prev}\ UK_{curr}\ IE_{curr}$

The status register contains an interrupt mask bits (15:10) and status information bits (5:0). If bit IM_i is 1 then interrupts at level i are enabled. The interrupt enable bits indicate whether interrupts are enabled or not in the respective state. If the IE_{curr} is 0 then it means the processor is currently running with the interrupts disabled.

Register $12 is the *status* register (interrupt enable) and register $13 is the *cause* register (exception type)

In the second part of the program:
Line number 62 that is instruction $mfc0\ \$t4, \$13$ moves the value in register to $\$t4$ which is $0x00000800$ which corresponds to the IP_3 (i.e. bit_{11}) in cause register as set to 1 which means one of the pending interrupt is 1 and the exception code is 00000 which corresponds to the exception code as interrupt. In the next two instructions the interrupt pending bits ($bit_8\ to\ bit_{15}$) are set to 0 and then the value is moved back to the cause register which makes the cause register as $0x00000000$.

The interrupts are enabled in the device using the instructions from line 69 to 72 by loading $0x00000002$ to $\$t4$ and then storing it in the receiver control register and the transmitter control register, which is basically the interrupt enable (bit_1) setting it to 1 in those two registers.

The status register's value is moved to $\$t4$ in the instruction at line 73 (i.e. $mfc0\ \$t4, \$12$), and the bits are masked making all the interrupt mask bits as 1, the old interrupt enable and the user/kernel bits to whatever they already were, the previous and the current interrupt

enable and user/kernel bits to 1. Then loading this value back to the status register by instruction at line 75 (i.e. $mtc0\ \$t4, \$12$).

At line number 78, jump instruction is used, that reads input/character and then performs the interrupt handler in the next instructions which checks for the execution code in the cause register and if there is no interrupt then jump to an exception, if it is an interrupt then it checks for the interrupt pending bit (IP_3) and then checks if there is character to be written and then checks for the next interrupt pending bit (IP_2) and services that pending interrupt.

The line number 78 is a loop waiting for the interrupts, and when the program is running the program keeps on taking input from the user until three lines are entered because the counter is set to 3 and it decrements each time new line is read.

The values of cause register $0x00000000$ (all bits 0), $0x00000C00$ ($IP_3\ IP_2$ set to 1), $0x00000800$ (IP_3 set to 1), and $0x00000400$ (IP_2 set to 1).
The exception code is 00000 (interrupt).

The values of status register $0x3000FF10$ ($user\ mode\ bit_5 = 1$, interrupt mask bits set to 1), after clearing interrupt enable bit in status register to disable interrupts the value changes to $0x30000010$ (interrupt mask bits set 0), and $0x300FF13$ (interrupt mask bits set to 1, old user/kernel bit set to 1 and the new user/kernel as well as the current interrupt enable set to 1).

The value of register $\$t4$ was observed during polling (as 0,1,2,3) when masking and checking for the receiver and transmitter control bits ($bit_1\ bit_0$) which corresponds to the ready bit (bit_0) and the interrupt bit (bit_1) being changed for each condition. The register $\$t4$ and $\$t5$ are used in the second half of the program for interrupts.

The value of register $\$t6$ was observed (ASCII equivalent of characters input) when the load instruction was used to load from the receiver data register and then store instruction for writing on transmitter data register for displaying the character.

Reference: http://www.cs.uwm.edu

Assignment – 5:

Reordering:

Total number of cycles for the program without reordering is [5+31 (the *syscall* instructions)] + [38 (the instructions from line 42 to 67 for computing the solution)] = 74 cycles.

After reordering total number of cycles = 29 (the instructions for computing the solution + 36 (the *syscall* instructions) = 65.

So, 9 cycles reduction.

For reordering the table has been divided into two that represents the complete program and number of clock cycles.

Instructions	Clock Cycles																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
mul $t4, $t2, $t2	IF	ID	EX	MEM	WB															
mul $t5, $t1, $t3		IF	ID	EX	MEM	WB														
li $s0, 1			IF	ID	EX	MEM	WB													
				X	X	X	X	X												
mul $t5, $t5, 4					IF	ID	EX	MEM	WB											
mult $s0, $s0						IF	ID	EX	MEM	WB										
							X	X	X	X	X									
sub $t6, $t4, $t5								IF	ID	EX	MEM	WB								
mflo $s2									IF	ID	EX	MEM	WB							
										X	X	X	X	X						
tlt $t6, $0											IF	ID	EX	MEM	WB					
move $s1, $t6												IF	ID	EX	MEM	WB				
													X	X	X	X	X			
														X	X	X	X	X		
bge $s2, $s1, endsqrt															IF	ID	EX	MEM	WB	
addi $s0, 1																IF	ID	EX	MEM	WB
li $t0, 2																	IF	ID	EX	MEM
neg $s2, $t2																		IF	ID	EX
																			X	X
mul $s5, $t1, $t0																				IF
add $s3, $s2, $s0																				
sub $s4, $s2, $s0																				
div $s6, $s3, $s5																				
div $s7, $s4, $s5																				

Instructions	Clock Cycles																			
	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
mul $t4, $t2, $t2																				
mul $t5, $t1, $t3																				
li $s0, 1																				
mul $t5, $t5, 4																				
mult $s0, $s0	WB																			
	X	X																		
sub $t6, $t4, $t5	EX	MEM	WB																	
mflo $s2	ID	EX	MEM	WB																
	X	X	X	X	X															
tlt $t6, $0		IF	ID	EX	MEM	WB														
move $s1, $t6			IF	ID	EX	MEM	WB													
				X	X	X	X	X												
					X	X	X	X	X											
bge $s2, $s1, endsqrt						IF	ID	EX	MEM	WB										
addi $s0, 1							IF	ID	EX	MEM	WB									
li $t0, 2								IF	ID	EX	MEM	WB								
neg $s2, $t2									IF	ID	EX	MEM	WB							
										X	X	X	X	X						
mul $s5, $t1, $t0											IF	ID	EX	MEM	WB					
add $s3, $s2, $s0												IF	ID	EX	MEM	WB				
sub $s4, $s2, $s0													IF	ID	EX	MEM	WB			
														X	X	X	X	X		
div $s6, $s3, $s5															IF	ID	EX	MEM	WB	
div $s7, $s4, $s5																IF	ID	EX	MEM	WB

Forwarding with reordering:

Total number of cycles = 36 + 22 = 58 cycles.
16 cycles of reduction from the program (without reordering and no forwarding 74 cycles)
7 cycles of reduction from the program (with reordering and no forwarding 65 cycles)

Instructions	Clock Cycles																					
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
mul $t4, $t2, $t2	IF	ID	EX	MEM	WB																	
mul $t5, $t1, $t3		IF	ID	EX	MEM	WB																
li $s0, 1			IF	ID	EX	MEM	WB															
mul $t5, $t5, 4				IF	ID	EX	MEM	WB														
mult $s0, $s0					IF	ID	EX	MEM	WB													
sub $t6, $t4, $t5						IF	ID	EX	MEM	WB												
mflo $s2							IF	ID	EX	MEM	WB											
tlt $t6, $0								IF	ID	EX	MEM	WB										
move $s1, $t6									IF	ID	EX	MEM	WB									
bge $s2, $s1, endsqrt										IF	ID	EX	MEM	WB								
addi $s0, 1											IF	ID	EX	MEM	WB							
li $t0, 2												IF	ID	EX	MEM	WB						
neg $s2, $t2													IF	ID	EX	MEM	WB					
mul $s5, $t1, $t0														IF	ID	EX	MEM	WB				
add $s3, $s2, $s0															IF	ID	EX	MEM	WB			
sub $s4, $s2, $s0																IF	ID	EX	MEM	WB		
div $s6, $s3, $s5																	IF	ID	EX	MEM	WB	
div $s7, $s4, $s5																		IF	ID	EX	MEM	WB

Although reordering decreased the number of cycles of the program but there could still be seen NOP in the program, but, from the table above it can be seen that the forwarding has decreased the number of cycles to 22, the program execution time will be even lower than the previous cases with reordering or without reordering, and also with forwarding there is no need of NOP that is the stalling/bubbles in the program.

The highlighted blocks in the table shows where the data has been forwarded to the instruction's stage that is dependent upon the previous data.

The tables contain only the instructions of the program that computes the solution.

Note: program with reordered instructions attached.