

# Final Exam

---

CIS-655 ADVANCED COMPUTER ARCHITECTURE

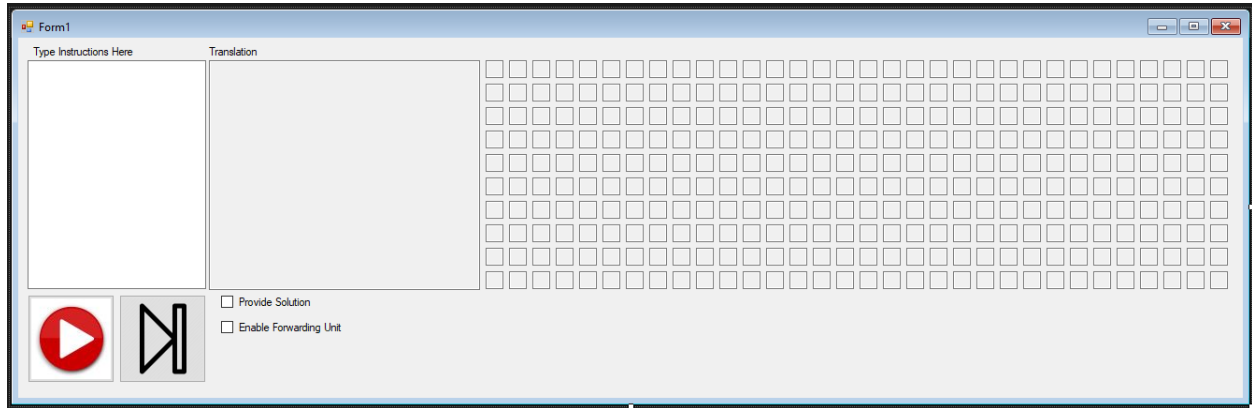
PROF. MOHAMMED ABDALLAH

12/8/2021

Anthony Redamonti  
SYRACUSE UNIVERSITY

The following program was written to decode a maximum of 10 MIPS instructions and provide any information regarding data dependencies. The program also has an option for providing two solutions to any dependencies: without a forwarding unit and with a forwarding unit. The four supported opcodes are: lw, sw, add, and sub.

Please find the entire C# application attached.



The GUI is shown above. The user enters commands into the textbox. If the user does not check the checkbox “Provide Solution” then the program will simply output the opcode and the registers involved, along with any dependencies that may occur. If there are any dependencies, then the program will provide the registers and instructions at fault (debugging information).

Below is an example of a data dependency.

```
Lw $t0, 0($s2)
Add $t5, $t0, $s4
```

There is a data dependency on register \$t0 because it is being used as rt (destination register) in the first instruction and then as an argument in the add instruction. Notice that with no forwarding unit, there are two stall cycles required.

The above dependency was tested in the program. The “Provide Solution” checkbox was not checked, so no solution was provided. There is a data dependency error message shown in the translation textbox: “DATA DEPENDENCY: \$t0 cannot be used as argument in add/sub when used as rt in previous lw instruction.”

Form1

Type Instructions Here

```
lw $t0, 0($s2)
add $t5, $t0, $s4
```

Translation

Opcode 00: Read the contents of memory location ( $\$s2 + 0$ ) and write them to register  $\$t0$

Opcode 10: Add reg.  $\$t5 = \$t0 + \$s4$

DATA DEPENDENCY:  $\$t0$  cannot be used as argument in add/sub when used as it in previous lw instruction

F	D	X	M	W		
	F	D	X	M	W	

☐ Provide Solution
   
☐ Enable Forwarding Unit

Below is the output when the “Provide Solution” checkbox is checked (solution without forwarding unit).

[illegible]

There are two stall cycles inserted before the decode stage of the second instruction. Notice that the writeback stage of the first instruction is in parallel with the decode stage of the second instruction (separate I&D).

Below is the output of the program when the forwarding unit is enabled.

Only one stall cycle is needed. The forwarding unit will forward the output of the memory access stage of the first instruction to the beginning of the execution stage (ALU) of the second instruction.

If there is an independent instruction between these two instructions the outputs change. See the example below.

```
Lw $t0, 0($s2)
Add $t2, $t3, $t4
Add $t5, $t0, $s4
```

Form1

Type Instructions Here



lw \$t0, 0(\$s2)  
 add \$t2, \$t3, \$t4  
 add \$t5, \$t0, \$s4

Translation

Opcode 00: Read the contents of memory location (\$s2 + 0) and write them to register \$t0  
 Opcode 10: Add reg. \$t2 = \$t3 + \$t4  
 Opcode 10: Add reg. \$t5 = \$t0 + \$s4  
 DATA DEPENDENCY: \$t0 cannot be used as an argument in add/sub when it was used as rt in lw 2 instructions previous.

F	D	X	M	W		
	F	D	X	M	W	
		F	D	X	M	W

☐ Provide Solution
 ☐ Enable Forwarding Unit

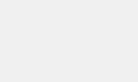
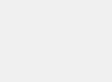



Notice the message in the translation textbox: "DATA DEPENDENCY: \$t0 cannot be used as an argument in add/sub when it was used as rt in lw 2 instructions previous."

Below is the output when the “Provide Solution” checkbox is checked (solution without forwarding unit).

**Form1**

Type Instructions Here	Translation
lw \$t0, 0(\$s2)	
add \$t2, \$t3, \$t4	
add \$t5, \$t0, \$s4	

☒ Provide Solution  
☐ Enable Forwarding Unit

There is only one stall cycle inserted before the decode stage of the third instruction.

Below is the output of the program when the forwarding unit is enabled.

Form1



Type Instructions Here

lw \$t0, 0(\$s2)  
 add \$t2, \$t3, \$t4  
 add \$t5, \$t0, \$s4

Translation

F	D	X	M	W		
	F	D	X	M	W	
		F	D	X	M	W

☒ Provide Solution  
☒ Enable Forwarding Unit

No stall cycles are needed because the output of the memory access stage in the first instruction is forwarded to the input of the execution stage (ALU) of the third instruction.

If there are two independent instructions between these commands, no stall cycles are required, regardless of data forwarding capabilities.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$t0, 0(\$s2)	F	D	X	M	W			
Add \$s2, \$s2, \$s4		F	D	X	M	W		
Sub \$s1, \$s5, \$s6			F	D	X	M	W	
Add \$t0, \$t0, \$s4				F	D	X	M	W

Therefore, since the worst-case scenario requires two stall cycles, the previous two instructions need to be compared for analysis with the current instruction.

See output below. There are no dependencies reported.

Form1

Type Instructions Here

```
lw $t0, 0($s2)
add $s2, $s2, $s4
sub $s1, $s5, $s6
add $t0, $t0, $s4
```

Translation

Opcode 00: Read the contents of memory location ( $\$s2 + 0$ ) and write them to register  $\$t0$

Opcode 10: Add reg.  $\$s2 = \$s2 + \$s4$

Opcode 11: Subtract reg.  $\$s1 = \$s5 - \$s6$

Opcode 10: Add reg.  $\$t0 = \$t0 + \$s4$

F	D	X	M	W				
	F	D	X	M	W			
		F	D	X	M	W		
			F	D	X	M	W	

☐ Provide Solution
 ☐ Enable Forwarding Unit

Complex Test:

1. lw \$s2, 0(\$t2)
2. lw \$t1, 0(\$s2)
3. sw \$s3, 0(\$t1)
4. add \$t0, \$t1, \$t2
5. add \$t3, \$t4, \$t5

There are 3 data dependencies in the code above. Data dependency with \$s2 in instruction 1 and 2. Another data dependency with \$t1 in instruction 2 and 3. And one more with \$t1 between instructions 2 and 4.

Here is the output with no solution:



Form1

Type Instructions Here

lw \$s2, 0(\$t2)  
 lw \$t1, 0(\$s2)  
 sw \$s3, 0(\$t1)  
 add \$t0, \$t1, \$t2  
 add \$t3, \$t4, \$t5

Translation

Opcode 00: Read the contents of memory location (\$t2 + 0) and write them to register \$s2  
 Opcode 00: Read the contents of memory location (\$s2 + 0) and write them to register \$t1  
 DATA DEPENDENCY: \$s2 cannot be used as rs in lw and rt in previous lw instruction  
 Opcode 01: Write the contents of \$s3 to memory location (\$t1 + 0)  
 DATA DEPENDENCY: \$t1 cannot be used as rs in sw and rt in previous lw instruction  
 Opcode 10: Add reg. \$t0 = \$t1 + \$t2  
 DATA DEPENDENCY: \$t1 cannot be used as an argument in add/sub when it was used as rt in lw 2 instructions previous.  
 Opcode 10: Add reg. \$t3 = \$t4 + \$t5

☐ Provide Solution  
☐ Enable Forwarding Unit

F	D	X	M	W					
	F	D	X	M	W				
		F	D	X	M	W			
			F	D	X	M	W		
				F	D	X	M	W	



Below is the output with a solution (forwarding unit disabled).

**Form1**

Type Instructions Here	Translation
lw \$s2, 0(\$t2)	F D X M W
lw \$t1, 0(\$s2)	F S S D X M W
sw \$s3, 0(\$t1)	F S S D X M W
add \$t0, \$t1, \$t2	F D X M W
add \$t3, \$t4, \$t5	F D X M W

☒ Provide Solution  
☐ Enable Forwarding Unit

Below is the output with the forwarding unit enabled.

Form1



Type Instructions Here

lw \$s2, 0(\$t2)  
 lw \$t1, 0(\$s2)  
 sw \$s3, 0(\$t1)  
 add \$t0, \$t1, \$t2  
 add \$t3, \$t4, \$t5

Translation

F	D	X	M	W							
	F	D	S	X	M	W					
		F	S	D	S	X	M	W			
				F	S	D	X	M	W		
						F	D	X	M	W	

☒ Provide Solution
 ☒ Enable Forwarding Unit

Code Review:

A struct was created to properly format instruction history.

```
// create a struct for an instruction
struct Instruction
{
    public string opcode;
    public string rt;
    public string rs;
    public string rd;
}
```

Two global variables were created to record the number of instructions executed and the current cycle.

```
// create a global variable that counts the total number of instructions executed
thus far.
```

```
int numberOfInstructionsExecuted = 0;
```

```
// create a global variable that counts the current instruction cycle thus far.
```

```
int cycleNumber = 0;
```

```
// array holding the previous instruction's timing sequence
```

```
char[] prevTimingSequence = new char[7];
```

```
string[] registerNameStringArray = new string[32]; // an array containing the
register strings
```

```
string[] opcodeStringArray = new string[4]; // an array holding all 4 op-codes
```

```
int parsingIndex = 0; // index used to parse commands
```

The previous 2 instructions are stored in the “lastInstruction” and “secondToLastInstruction” variables shown below.

```
// array containing the last instruction executed
```

```
Instruction lastInstruction = new Instruction();
```

```
// array containing the second to last instruction executed
```

```
Instruction secondToLastInstruction = new Instruction();
```

The 10 instructions are stored in an array of arrays named “commandsArr” shown below.

```
// an array of arrays
```

```
TextBox[][] commandsArr = new TextBox[10][];
```

If the user presses the Play button in the GUI, the event handler below is triggered.

```
// the user pressed the play button
```

```
private void playButton_Click(object sender, EventArgs e)
```

```
{
```

```
    reset(); // reset the settings for a new execution.
```

```
    // collect the user input
```

```
    string commands = scriptTextBox.Text;
```

```
    // parse the user input
```

```
    char[] delimitingChars = { ',', '\r', '\n', ' ', '(', ')' };
```

```

        string[] words = commands.Split(delimitingChars,
StringSplitOptions.RemoveEmptyEntries);

        // if the length of the script is zero, ask the user for input.
        if (words.Length == 0)
        {
            translationTextBox.Text = "Please enter code into the script textbox.";
            return;
        }

        // used to detect error.
        int error = 0;

        while ((parsingIndex < words.Length) && (error == 0))
        {
            // call single step function
            error = singleStep(ref words, ref parsingIndex);
        }
    }
}

```

The instructions are executed one at a time by calling the “singleStep” function until the parsing index variable exceeds the number of commands entered in the GUI.

The single step button is very similar, except that it calls the “singleStep” function in an if statement rather than a while loop. The event handler for the single step button is below.

```

private void singleStepButton_Click(object sender, EventArgs e)
{
    string commands = scriptTextBox.Text; // collect the user input

    // parse the user input
    char[] delimitingChars = { ',', '\r', '\n', ' ', '(', ')' };
    string[] words = commands.Split(delimitingChars,
StringSplitOptions.RemoveEmptyEntries);

    // if the length of the script is zero, ask the user for input.
    if (words.Length == 0)
    {
        translationTextBox.Text = "Please enter code into the script textbox.";
        return;
    }

    int error = 0; // used to detect user error

    // if executing the first command, reset the spacing and the cycle
parameters.
    if(parsingIndex == 0)
    {
        numberOfInstructionsExecuted = 0;
        cycleNumber = 0;
    }

    // if the parsing index is less than the length of the instructions, perform
a single instruction
    if ((parsingIndex < words.Length) && (error == 0))
    {
        // call single step function
    }
}

```

```

        error = singleStep(ref words, ref parsingIndex);
    }
}

```

The single step function checks that the number of commands executed thus far does not exceed the maximum supported value (10). If it does, then the function returns a value of -1, and the program takes no further action.

The function is used to check that the opcode entered is supported and calls the appropriate function for that opcode.

```

// Perform a single command.
// Update the index of the opcode for the next command.
int singleStep(ref string[] commands, ref int index)
{
    // if the number of commands executed thus far exceeds
    // the maximum number of commands supported, return here.
    if(numberOfInstructionsExecuted >= commandsArr.Length)
    {
        return -1;
    }

    // see if the opcode is in the opcode string array
    string opcode = "invalid";
    int arrayParse = 0;
    while ((opcode == "invalid") && (arrayParse < opcodeStringArray.Length))
    {
        // if there's a match, update the opcode
        if (commands[index] == opcodeStringArray[arrayParse])
        {
            opcode = opcodeStringArray[arrayParse];
        }

        // increase arrayParse for next iteration
        arrayParse = arrayParse + 1;
    }

    // if there was no match, inform the user.
    if (opcode == "invalid")
    {
        translationTextBox.Text += commands[index] + " is not a supported
opcode.";
        return -2;
    }
    // perform the instruction.
    else
    {
        // perform the command and update the index of the next opcode.
        int error = performCommand(ref commands, ref opcode, ref index);
        return error;
    }
}

```

The performCommand function simply checks that the arguments (registers) are valid before calling the function requested by the opcode. It also increments the number of instructions executed thus far as well as the current cycle number.

```

// call the function needed for this command
int performCommand(ref string[] array, ref string opcode, ref int index)
{
    // Example: lw $t0, 1000($t1)
    if (opcode == "lw")
    {
        // if one of the registers is invalid, return.
        if (!checkTwoRegisters(ref array[index], ref array[index + 1], ref
array[index + 3]))
        {
            return -1;
        }
        else
        {
            // call the load word function and update the index for the next
instruction
            loadWord(array[index + 1], array[index + 3], array[index + 2]);
            index = index + 4;
        }
    }
    // Example: writew $t0, 1000($t1)
    else if (opcode == "sw")
    {
        // if one of the registers is invalid, return.
        if (!checkTwoRegisters(ref array[index], ref array[index + 1], ref
array[index + 3]))
        {
            return -1;
        }
        else
        {
            // call the store word function and update the index for the next
instruction
            storeWord(array[index + 1], array[index + 3], array[index + 2]);
            index = index + 4;
        }
    }
    // Example: add $t0, $t1, $t2
    else if (opcode == "add")
    {
        // if one of the registers is invalid, return.
        if (!checkThreeRegisters(ref array[index], ref array[index + 1], ref
array[index + 2], ref array[index + 3]))
        {
            return -1;
        }
        else
        {
            // call the add function and update the index for the next
instruction
            add(array[index + 1], array[index + 2], array[index + 3]);
            index = index + 4;
        }
    }
    // Example: sub $t0, $t1, 123
    else if (opcode == "sub")
    {
        // if one of the registers is invalid, return.

```

```

        if (!checkThreeRegisters(ref array[index], ref array[index + 1], ref
array[index + 2], ref array[index + 3]))
        {
            return -1;
        }
        else
        {
            // call the subtract function and update the index for the next
instruction
            sub(array[index + 1], array[index + 2], array[index + 3]);
            index = index + 4;
        }
    }
    else
    {
        // nothing to do at this time
    }

    // increment the number of instructions executed up to this point and return
0 (success).
    numberOfInstructionsExecuted = numberOfInstructionsExecuted + 1;

    // increment the cycle number for the next command.
    cycleNumber = cycleNumber + 1;

    // return zero (success).
    return 0;
}

```

Each opcode calls a different function: lw, sw, add, or sub. The functions are all structured similarly. StoreWord is below.

```

// write a value from rt to memory
void storeWord(string rt, string rs, string inputOffset)
{
    // if the provide solutions checkbox is not checked, print the details of the
instruction.
    // No timing diagram is generated in this case.
    if (!provideSolutionCheckBox.Checked)
    {
        // write the instruction to the translation text box
        translationTextBox.Text += "\nOpcode 01: Write the contents of " + rt + "
to memory location (" + rs + " + " + inputOffset + ")\r";
    }

    string opcode = "sw";
    string emptyString = "";

    // check for data dependencies.
    checkDataDependencies(ref opcode, ref rt, ref rs, ref emptyString);
}

```

If the user did not check the “provideSolution” checkbox, then the program will output the instruction information (opcode and registers involved).

The function will call “checkDataDependencies,” which checks for all possible data dependencies. Due to the size of the function, please view it in the code attached in the submission. The function formulates the correct sequence (F, D, X, M, W) for the current instruction based on the previous two instructions. After formulating the correct answer, it will pass it to the “formatTimeDiagramWithPrevTimeDiagram” function so that any stalls in previous answers are carried over as needed.

```

// format the correct current timing diagram with the previous timing sequence
// so that stall cycles are taken into account.
// Format is "FDXMW". 'S' for stall.
void formatTimeDiagramWithPrevTimeDiagram(ref string currentTimeDiagram)
{
    // create a string builder and append the characters of the current timing
    diagram to it.
    var stringBuilder = new StringBuilder();
    for(int i = 0; i < currentTimeDiagram.Length; i++)
    {
        stringBuilder.Append(currentTimeDiagram[i]);
    }

    // if the second character in the previous timing sequence was a stall cycle,
    // increment the number of instructions executed by 1, which adjusts the
padding
    // for future commands.
    if (prevTimingSequence[1] == 'S')
    {
        cycleNumber = cycleNumber + 1;

        // if the third character is also a stall cycle in the previous
instruction
        // add another offset to the initial spacing
        if (prevTimingSequence[2] == 'S')
        {
            cycleNumber = cycleNumber + 1;
        }
    }

    // find the index of the decode in the previous instruction
    int dIndex = 0;
    while((dIndex < prevTimingSequence.Length) && (prevTimingSequence[dIndex] !=
'D'))
    {
        dIndex = dIndex + 1;
    }

    // if the previous command had a stall cycle after the decode stage, carry it
over to the current instruction.
    if (lastInstruction.opcode != "")
    {
        if (prevTimingSequence[dIndex + 1] == 'S')
        {
            stringBuilder.Insert(1, 'S');
        }
    }

    // create a local copy of the cycle number
    int cycleCopy = cycleNumber;

```

```

        // print the current timing diagram to the output.
        for (int i = 0; i < stringBuilder.Length; i++)
        {
            // send each character in the string builder to the console and increment
            the number of cycles.
            commandsArr[numberOfInstructionsExecuted][cycleCopy].Text +=
stringBuilder[i];
            cycleCopy = cycleCopy + 1;
        }

        // convert the properly formatted string builder to a string.
        string properlyFormattedAnswer = stringBuilder.ToString();

        // update the timing sequence history for the next command.
        updateTimingSequenceHistory(ref properlyFormattedAnswer);
    }

```

Notice that if there is a stall cycle in the previous instruction in the 2<sup>nd</sup> stage, the cycle of the fetch stage for the current instruction will be incremented (cannot fetch under a stall cycle). If there is a stall in the 2<sup>nd</sup> and 3<sup>rd</sup> stages of the previous instruction, the fetch stage of the current instruction will be incremented by 2.

Also, if there is a stall cycle in the previous instruction after the decode stage, a stall cycle is inserted into the same cycle of the current instruction.



All possible combinations of lw, sw, and add/sub instructions will be listed below.

Lw and Add/Sub (8 Possible Combinations)

1. \$s2 used as rs in lw then used as argument in add/sub.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$t0, 0(\$s2)	F	D	X	M	W			
Add \$s3, \$s2, \$s4		F	D	X	M	W		

No dependency.

2. \$s2 used as rs in lw then used as rt in add/sub.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$t0, 0(\$s2)	F	D	X	M	W			
Add \$s2, \$s3, \$s4		F	D	X	M	W		

No dependency.

3. \$s2 used as rt in lw then used as argument in add/sub.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$s2, 0(\$t0)	F	D	X	M	W			
Add \$s3, \$s2, \$s4		F	D	S	S	X	M	W

Data dependency.

There is a data dependency because lw uses \$s2 as rt (destination register) and then add/sub uses \$s2 as rs or rd (an argument) in the very next instruction.

If a forwarding unit is used, only one stall cycle is needed (out of M and into X).

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$s2, 0(\$t0)	F	D	X	M	W			
Add \$s3, \$s2, \$s4		F	D	S	X	M	W	

4. \$s2 used as rt in lw then used as rt in add/sub.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$s2, 0(\$t0)	F	D	X	M	W			
Add \$s2, \$s1, \$s4		F	D	X	M	W		

No dependency.

5. \$s2 used as argument in add/sub then used as rs in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s3, \$s2, \$s4	F	D	X	M	W			
Lw \$t0, 0(\$s2)		F	D	X	M	W		

No dependency.

6. \$s2 used as argument in add/sub then as rt in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
-------------	---------	---	---	---	---	---	---	---

Add \$s3, \$s2, \$s4	F	D	X	M	W			
Lw \$s2, 0(\$t0)		F	D	X	M	W		

No dependency.

7. \$s2 used as rt in add/sub then as rs in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s2, \$s3, \$s4	F	D	X	M	W			
Lw \$t0, 0(\$s2)		F	S	S	D	X	M	W

Data dependency.

There is a data dependency because \$s2 is used as rt in add/sub instruction then as rs in lw in the next instruction.

Using a forwarding unit, no stalls are needed (out of X and into X).

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s2, \$s3, \$s4	F	D	X	M	W			
Lw \$t0, 0(\$s2)		F	D	X	M	W		

8. \$s2 used as rt in add/sub instruction then as rt in lw instruction.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s2, \$s3, \$s4	F	D	X	M	W			
Lw \$s2, 0(\$t0)		F	D	X	M	W		

No dependency.

## Sw and Add/Sub (8 Possible Combinations)

1. \$s2 used as rs in sw then used as argument in add/sub.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$t0, 0(\$s2)	F	D	X	M	W			
Add \$s3, \$s2, \$s4		F	D	X	M	W		

No dependency.

2. \$s2 used as rs in sw then used as rt in add/sub.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$t0, 0(\$s2)	F	D	X	M	W			
Add \$s2, \$s3, \$s4		F	D	X	M	W		

No dependency.

3. \$s2 used as rt in sw then used as argument in add/sub.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$s2, 0(\$t0)	F	D	X	M	W			
Add \$s3, \$s2, \$s4		F	D	S	S	X	M	W

No dependency.

4. \$s2 used as rt in sw then used as rt in add/sub.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$s2, 0(\$t0)	F	D	X	M	W			
Add \$s2, \$s1, \$s4		F	D	X	M	W		

No dependency.

5. \$s2 used as argument in add/sub then used as rs in sw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s3, \$s2, \$s4	F	D	X	M	W			
Sw \$t0, 0(\$s2)		F	D	X	M	W		

No dependency.

6. \$s2 used as argument in add/sub then as rt in sw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s3, \$s2, \$s4	F	D	X	M	W			
Sw \$s2, 0(\$t0)		F	D	X	M	W		

No dependency.

7. \$s2 used as rt in add/sub then as rs in sw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s2, \$s3, \$s4	F	D	X	M	W			
Sw \$t0, 0(\$s2)		F	S	S	D	X	M	W

Data dependency.

There is a data dependency because \$s2 is used as rt in add/sub instruction then as rs in sw in the next instruction.

Using a forwarding unit, no stall cycles are needed (out of X and into X).

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s2, \$s3, \$s4	F	D	X	M	W			
Sw \$t0, 0(\$s2)		F	D	X	W	M		

8. \$s2 used as rt in add/sub instruction then as rt in sw instruction.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s2, \$s3, \$s4	F	D	X	M	W			
Sw \$s2, 0(\$t0)		F	S	S	D	X	M	W

Data dependency.

There is a data dependency because add/sub uses \$s2 as rt then sw uses \$s2 as rt in next instruction.

Using a forwarding unit, no stall cycles are needed (out of X and into M).

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s2, \$s3, \$s4	F	D	X	M	W			
Sw \$s2, 0(\$t0)		F	D	X	M	W		

Lw and Sw (8 different combinations)

1. \$s2 is used as rt in lw, then used as rt in sw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$s2, 0(\$t1)	F	D	X	M	W			
Sw \$s2, 0(\$t2)		F	S	S	D	X	M	W

Data dependency.

There is a dependency because \$s2 is used as rt (destination register) in lw and used as rt in sw in the very next instruction.

Using a forwarding unit, no stall cycles are needed (out of M and into M).

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$s2, 0(\$t1)	F	D	X	M	W			
Sw \$s2, 0(\$t2)		F	D	X	M	W		

2. \$s2 is used as rt in lw then as rs in sw instruction.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$s2, 0(\$t2)	F	D	X	M	W			
Sw \$t1, 0(\$s2)		F	S	S	D	X	M	W

Data dependency.

There is a dependency because \$s2 is used as rt in lw and used as rs in sw in the very next instruction.

Using a forwarding unit, only one stall cycle is needed (out of M and into X).

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$s2, 0(\$t2)	F	D	X	M	W			
Sw \$t1, 0(\$s2)		F	D	S	X	M	W	

3. \$s2 used as rs in lw then as rt in sw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$t2, 0(\$s2)	F	D	X	M	W			
Sw \$s2, 0(\$t1)		F	D	X	M	W		

No dependency.

4. \$s2 used as rs in lw then as rs in sw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$t2, 0(\$s2)	F	D	X	M	W			
Sw \$t1, 0(\$s2)		F	D	X	M	W		

No dependency.

5. \$s2 used as rt in sw then as rt in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
-------------	---------	---	---	---	---	---	---	---

Sw \$s2, 0(\$t1)	F	D	X	M	W			
Lw \$s2, 0(\$t2)		F	D	X	M	W		

No dependency.

6. \$s2 used as rt in sw then as rs in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$s2, 0(\$t1)	F	D	X	M	W			
Lw \$t2, 0(\$s2)		F	D	X	M	W		

No dependency.

7. \$s2 used as rs in sw then as rt in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$t2, 0(\$s2)	F	D	X	M	W			
Lw \$s2, 0(\$t1)		F	D	X	M	W		

No dependency.

8. \$s2 used as rs in sw then as rs in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$t2, 0(\$s2)	F	D	X	M	W			
Lw \$t1, 0(\$s2)		F	D	X	M	W		

No dependency.

Lw and Lw (4 different combinations)

1. \$s2 is used as rt in lw, then used as rt in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$s2, 0(\$t1)	F	D	X	M	W			
Lw \$s2, 0(\$t2)		F	D	X	M	W		

No dependency.

2. \$s2 used as rt in lw then as rs in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$s2, 0(\$t1)	F	D	X	M	W			
Lw \$t2, 0(\$s2)		F	S	S	D	X	M	W

Data dependency.

There is a data dependency because \$s2 is used as rt in lw then as rs in lw.

Using a forwarding unit, only one stall cycle is needed (out of M and into X).

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$s2, 0(\$t1)	F	D	X	M	W			
Lw \$t2, 0(\$s2)		F	D	S	X	M	W	

3. \$s2 used as rs in lw then as rt in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$t1, 0(\$s2)	F	D	X	M	W			
Lw \$s2, 0(\$t2)		F	D	X	M	W		

No dependency.

4. \$s2 used as rs in lw then as rs in lw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Lw \$t1, 0(\$s2)	F	D	X	M	W			
Lw \$t2, 0(\$s2)		F	D	X	M	W		

No dependency.

Sw and Sw (4 different combinations)

1. \$s2 is used as rt in sw, then used as rt in sw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$s2, 0(\$t1)	F	D	X	M	W			
Sw \$s2, 0(\$t2)		F	D	X	M	W		

No dependency.

2. \$s2 used as rt in sw then as rs in sw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$s2, 0(\$t1)	F	D	X	M	W			
Sw \$t2, 0(\$s2)		F	D	X	M	W		

No dependency.

3. \$s2 used as rs in sw then as rt in sw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$t1, 0(\$s2)	F	D	X	M	W			
Sw \$s2, 0(\$t2)		F	D	X	M	W		

No dependency.

4. \$s2 used as rs in sw then as rs in sw.

Instruction	Cycle 1	2	3	4	5	6	7	8
Sw \$t1, 0(\$s2)	F	D	X	M	W			
Sw \$t2, 0(\$s2)		F	D	X	M	W		

No dependency.



Add/Sub and Add/Sub (4 different combinations)

1. \$s2 is used as rt in add, then used as rt in add.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s2, \$t0, \$t1	F	D	X	M	W			
Add \$s2, \$t1, \$t2		F	D	X	M	W		

No dependency.

2. \$s2 used as rt in add then as argument in add.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s2, \$t0, \$t1	F	D	X	M	W			
Add \$t3, \$s2, \$t2		F	S	S	D	X	M	W

Data dependency.

There is a data dependency because \$s2 is used as rt in add then as an argument in the second add instruction.

Using a forwarding unit, no stall cycles are needed (out of X and into X).

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$s2, \$t0, \$t1	F	D	X	M	W			
Add \$t3, \$s2, \$t2		F	D	X	M	W		

3. \$s2 used as argument in add then as rt in second add instruction.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$t0, \$s2, \$t1	F	D	X	M	W			
Add \$s2, \$t2, \$t3		F	D	X	M	W		

No dependency.

4. \$s2 used as argument in add then as argument in second add.

Instruction	Cycle 1	2	3	4	5	6	7	8
Add \$t0, \$s2, \$t1	F	D	X	M	W			
Add \$t2, \$s2, \$t3		F	D	X	M	W		

No dependency.