COMPUTER ARCHITECTURE LABORATORY 2

Prepared by Husni Khanfar

1 Objectives

- Working with Pipeline simulator
- Implementing the pipelining.

2 Building your MIPS Pipeline Simulator.

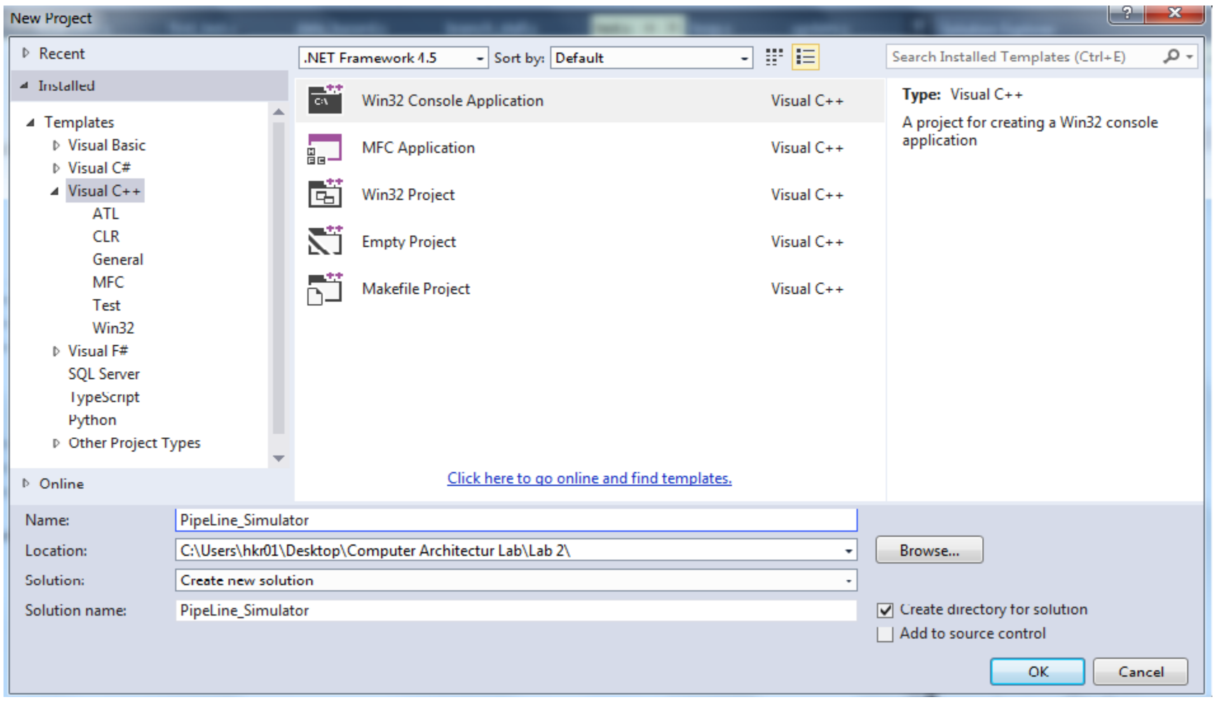
MIPS pipelines simulator consists of C source files. This simulator is successfully built in the following steps:

1- Open your Microsoft Windows
2- Install Microsoft Visual Studio 2013 if it is not installed.

   Hint: You can download it from your DreamSpark Account

3- Create the subfolder: lab 2 in the main folder: “Computer Architecture Lab” folder which you created in the lab 1.
   Note: the name of the main folder might be: CDT204.

4- Open Microsoft Visual Studio 2013
5- Select “File” from the main menu, then “New” → “Project”
6- You will see the window in Figure 1
7- Select “Visual C++” in the left hand side and “Win32 Console Application” in the right hand. Please, fill the “Name” and “Location” text boxes like figure 1, then press “Next”
8- A new dialog box titled by: “Welcome to the Win32 Application Wizard” appears. Press on “Next” button to continue.
9- “Application Settings” dialog box appears. Select ONLY the options shown in figure 2. After this step, your C++ project is created in Microsoft Visual Studio.



Type: Visual C++

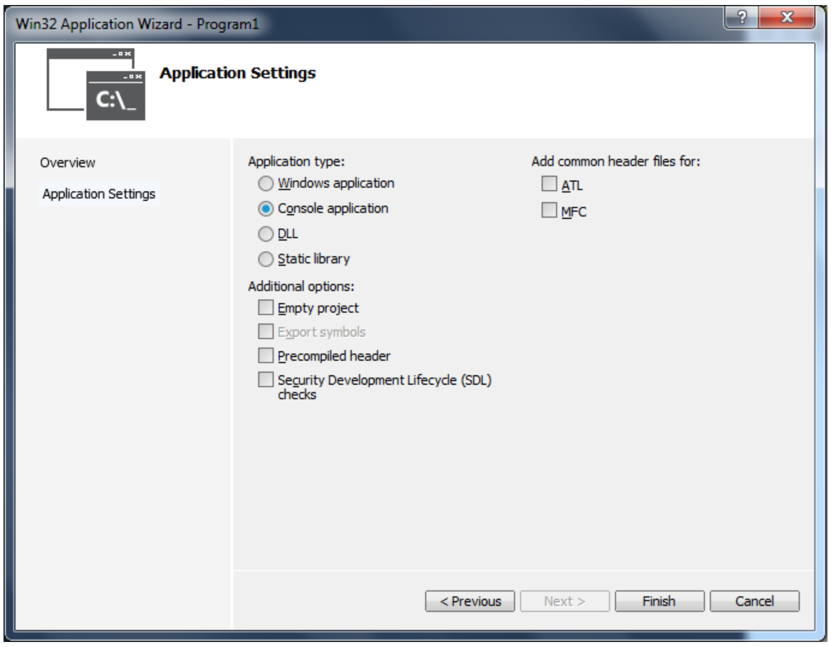
A project for creating a Win32 console application

Name:	PipeLine_Simulator
Location:	C:\Users\hkr01\Desktop\Computer Architectur Lab\Lab 2\
Solution:	Create new solution
Solution name:	PipeLine_Simulator

[✓] Create directory for solution

[ ] Add to source control

Figure 1



Application type:
- ( ) Windows application
- (●) Console application
- ( ) DLL
- ( ) Static library

Additional options:
- [ ] Empty project
- [ ] Export symbols
- [ ] Precompiled header
- [ ] Security Development Lifecycle (SDL) checks

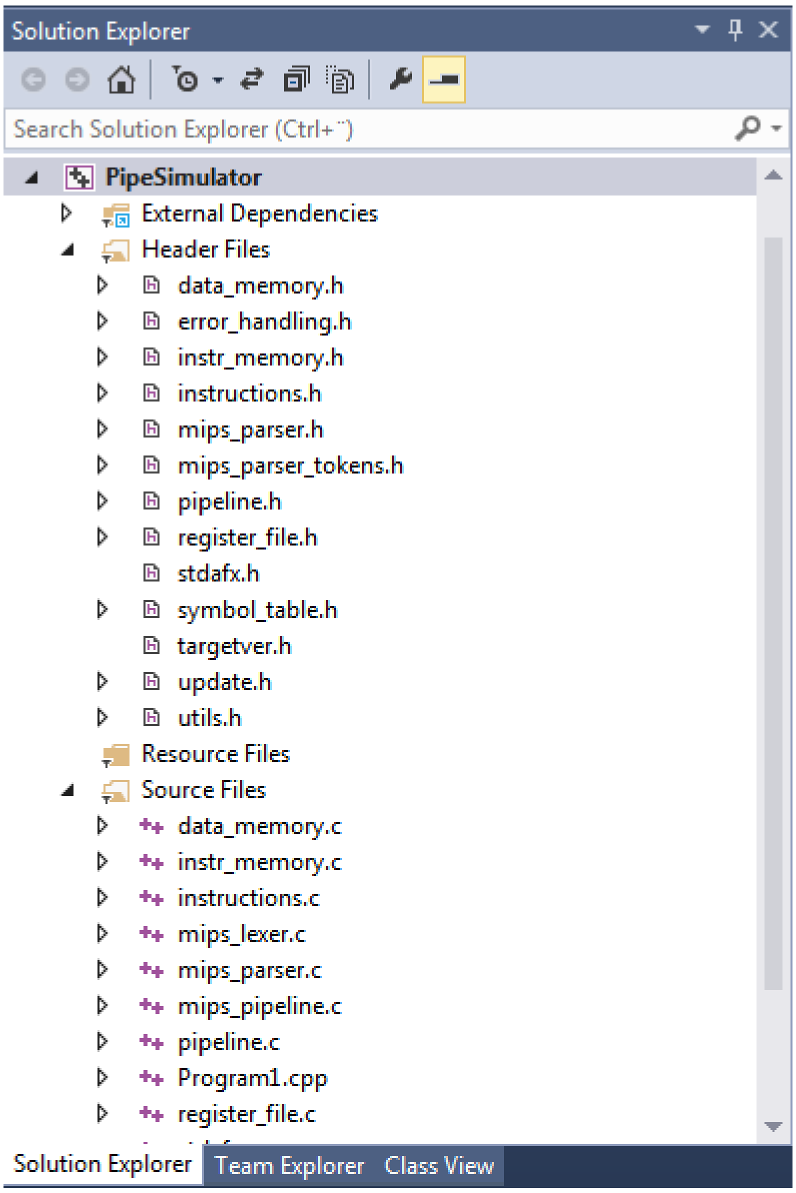
Add common header files for:
- [ ] ATL
- [ ] MFC

Figure 2

10- Lab2.ZIP file exists in the Contents; unzip the contents of this file, copy those extracted files in: “C:\Users\hkr01\Desktop\Computer Architectur Lab\Lab 2\PipeSimulator\”

a. Now we need to add all the source code files to our project. To do so: First, Select the main menu: “Project”. Second, Select “Add Existing Item”. Finally, A new dialog box appears, select all the .c and .h files and press on “Add” button.

11- Now your solution explorer window is shown like figure 3.



Solution Explorer

Search Solution Explorer (Ctrl+¨)

- PipeSimulator
  - External Dependencies
  - Header Files
    - data_memory.h
    - error_handling.h
    - instr_memory.h
    - instructions.h
    - mips_parser.h
    - mips_parser_tokens.h
    - pipeline.h
    - register_file.h
    - stdafx.h
    - symbol_table.h
    - targetver.h
    - update.h
    - utils.h
  - Resource Files
  - Source Files
    - data_memory.c
    - instr_memory.c
    - instructions.c
    - mips_lexer.c
    - mips_parser.c
    - mips_pipeline.c
    - pipeline.c
    - Program1.cpp
    - register_file.c

Figure 3

12- The last step; compile and run your project as follows: select “Build” and then “Build Solution”. If your work is OK, then the following message appears:

```
========== build All: 1 succeeded, 0 failed, 0 skipped ==========
```

3 Track the Status of the Microprocessor

In MIPS architecture, the instruction is executed in 5 individual stages, which are:

1- **Instruction Fetch (IF)** stage. This stage brings the instruction whose address is stored in the register: PC.

The Inputs:	The value of the PC register
The Outputs:	the op-code of the fetched instruction + the new address in PC

2- **Instruction Decode (ID)** stage: the op-code of the fetched instruction in IF stage is decoded. So, the used registers, the contents of the used registers and the immediate values are realized in this stage.

3- **Execution Stage (Ex)** stage: this stage implements the logical or the arithmetic operation of the instruction as well as calculates a new address.

The inputs:	The values of: rs, rt or immediate value
The outputs:	The value of rt, ALU result, the Branch target and Zero flag.

4- **Memory stage (Mem)**: this stage writes on the memory any changes.

The inputs:	The values of rt, ALU result, the Branch target and Zero flag.
The output:	Changes on one of the memory locations

5- **Write Back (WB)**: This stage updates one of the registers by the value of ALU result

The inputs:	The result which equals ALU result.
Output:	Updates the rt register. update the one of registers value.

Check your Understanding 1

Program1:

```
.globl __start
__start:
addi    $t1, $t1, 150
addi    $t2, $t2, 48
addi    $t3, $t3, 21
nop
nop
nop
and     $t4, $t1, $t2
ori     $t5, $t3, 16
sub     $t6, $t1, $t2
nop
addi $v0, $v0, 10
syscall
```

Save the above program as: Program1.s. Use table 1 to predict the status of the microprocessor during executing the three instructions:

```
and     $t4, $t1, $t2
ori     $t5, $t3, 16
sub     $t6, $t1, $t2
```

Consider that \$t1 = 150, \$t2 = 48 and \$t3 = 21 when those three instructions are executed.

In table 1 do the following:

In (1): Instruction Fetch (IF):

- Mention the fetched instruction
- Calculate the new value of PC register. Consider that the first memory location storing first instruction is 0x400000. Each instruction occupies 4 bytes.

In (2): Instruction Decoder (ID):

- Decode the fetched instruction. So, it finds the values of rs,rt and immediate values and move them to ES stage.

In (3): Execution Stage (ES):

- The values of rs, rt and imm are decoded in ID stage then they appears and used in ES stage. The outcome of the logical or arithmetic operation executed in ES stage appears in Memory Stage.

In (4): mem stage (MEM):

- ES stage produces three kinds of results:
  - ALU result: the outcome of any arithmetic or logical operation
  - Z: this is a Boolean flag which associates with some logical operation. It is set or reset according to the value in ALU result (zero or not zero)
  - Branch Target: The result of jump and conditional jump instructions is stored in Branch Target.

  ALU result, Z and Branch Target are calculated in ES and appears in MEM stage.

In (5): Write Back stage (WB):

- Write in the rd register the value calculated and stored in ALU result.

1- After filling Table 1. Check your result. To do so: Write Program1.s, Copy Program1.s in: C:\Users\(Your user name)\Desktop\ Computer Architecture Lab\Lab 2\PipeSimulator \Debug folder
2- Press on "Start" menu in Windows. Go to "Accessories" folder. Then run "Command Prompt"
3- Run: cd C:\Users\"your user name"\Desktop\ Computer Architectur Lab\Lab 2\PipeSimulator \Debug
4- Run the command: PipeSimulator program1.s
5- From the same directory: Open the trace.html
6- Compare Table 1 with the cycles in trace.html from 6 to 12 shown.

Check your Understanding 2

Save the following program as program2.s. From your understanding; fill table2. Exclude the last two instructions "the exit code from your analysis".

```
.globl __start
__start:
addi       $t1, $t1, 1
addi       $t2, $t2, 2
or         $t4, $t1, $t2
addi       $v0, $v0, 10
syscall
```

The most important point that you need to consider and learn in this exercise is the following: the instruction `or $t4,$t1,$t2` depends on the value of $t2 calculated in `addi  $t2, $t2, 2.` Therefore, you have to postpone executing `or $t4,$t1,$t2` till it writes the new value in $t2.
Try to return back to your text book in order understand more how to guess the stages of each of those two instructions.

Check your Understanding 3

Program 3.s: Expect the pipeline stages from cycle 6 to cycle 8. Report your expectations in Table 3, then compare them with your observations, gotten from running the command line: `Simulator program2.s`

```
.globl __start
__start:
addi       $t1, $t1, 9
addi       $t2, $t2, 5
nop
nop
bne        $t1, $t2, L1
sub        $t3, $t1, $t2
or         $t4, $t1, $t2
L1:
addi       $v0, $v0, 10
syscall
```

	Cycle 6	Cycle 7	**Cycle 8**	**Cycle 9**	**Cycle 10**	**Cycle 11**	**Cycle 12**
IF:	(1)	(1)	(1)	(1)	(1)	(1)	(1)
IE:	(2)	(2)	(2)	(2)	(2)	(2)	(2)
Ex:	(3)	(3)	(3)	(3)	(3)	(3)	(3)
MEM:	(4)	(4)	(4)	(4)	(4)	(4)	(4)
Ex:	(5)	(5)	(5)	(5)	(5)	(5)	(5)

Table 1

	Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
IF:	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)
IE:	(2)	(2)	(2)	(2)	(2)	(2)	(2)	(2)	(2)
Ex:	(3)	(3)	(3)	(3)	(3)	(3)	(3)	(3)	(3)
MEM:	(4)	(4)	(4)	(4)	(4)	(4)	(4)	(4)	(4)
Ex:	(5)	(5)	(5)	(5)	(5)	(5)	(5)	(5)	(5)

Table 2

	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
IF:	(1) `bne $t1,$t2, L1`	(1) `sub $t3, $t1, $t2`	(1)	(1)	(1)
IE:	(2) -	(2) `bne $t1,$t2, L1`	(2)	(2)	(2)
Ex:	(3) -	(3) -	(3)	(3)	(3)
MEM:	(4) -	(4) -	(4)	(4)	(4)
Ex:	(5) -	(5) -	(5)	(5)	(5)

Table 3

4 The Internal Implementation of Update.C

4.1 Computing Next State

In update.c there are two kind of data structures instances categorized in terms of the current or next execution cycles. The first group of the data structures holds the current status of each stage in the microprocessor. The current status of each stage is expressed by the following data structures instances:

if_stage, id_stage, ex_stage,mem_stage, wd_stage.

The second group of data structures are used to calculate the internal registers in each next stage. The second group consists of:

next_if, next_id, next ex, next_mem, next_wd

Your role here in the lab is to modify the calculations of next stages. When you do so, consider the followings:

1- Any next stage is calculated only from the current stages. **Do NOT** calculate it from any another next stage.
2- When the next stages are calculated; do not try to make any change in the instances of the current stages.

Notice, the above ten instances are created from five data structures: IFStage; IDStage; ExStage; MemStage and WBStage. Accordingly, if_stage and next_if are instantiated from IFStage class. Those classes are declared in pipeline.h file. It is highly important to take a look into this file before solving your assignments.

4.2 UpdatePipelineState

The next stages are calculated in the function `UpdatePipelineState (figure 4)`. There are five procedures which each calculates one of the next stages: `ComputeNextIFStage, ComputeNextIDStage, ComputeNextExStage, ComputeNextMemStage and finally ComputeNextWBStage.`

The function `UpdatePipelineState` simulates the working in the entire microprocessor by:

1- Calls consecutively the five functions
2- Calls the function `HandleHazards` which handles any dependencies issue.

Before working in the assignments. Consider that it is your role to read and understand the source code of update.c and pipeline.h. It is very important to link between the source code and the ideas of pipelines.

4.3 Check the current instruction in another stage

Sometimes, you need to need to know what is the current instruction in another stage. The following is an example of checking whether the current instruction in ID stage is BEQ:

```
if( id_stage.instr.type == BEQ )

                    ..................
```

4.4 Creating Bubbles

In your implementation, you need sometimes to create bubbles in a particular stage. To do so, you can use the following function:

```
ClearIDStage(next_id);       //Creates a bubble in ID stage

ClearExStage(next_ex);       //Creates a bubble in Ex Stage
```

```
ClearMemStage(next_mem);    //Creates a bubble in Mem Stage
```

4.5 GetRegWrittenByInstr

`GetRegWrittenByInstr` function checks what is the register checked in a particular stage.

For example, to know what is the register updated by the instruction which is currently in MEM stage, we use this function as the following:

```
int updated = GetRegWrittenByInst(&mem_stage.instr);
```

4.6 GetRegReadByInstr

`GetRegReadByInstr` function checks what are the registers used by an instruction in one of the stages. For example, to know what are the register used in the instruction existing in MEM stage, we write the following:

```
int rs,rt;

GetRegsReadByInstr(&mem_stage.instr, &rs, &rt);
```

5 Assignments

5.1 Assignment 1

program flow refers to the specification of the order in which the individual instructions are executed. `j` instruction changes the program flow. Therefore, instead of fetching the next immediate instruction to the current executed instruction, another instruction is fetched. This mechanism depends on changing the value of PC register. So, the unique operand of `j` instruction is the address of the new alternative address.

In order to implement `j` instruction, you have to follow the following steps:-

1- `ComputeNextExStage` produces an error message if your code contains J instruction, comment this code.

2- IF stage just fetches the instruction, but it does not know what is the type of instruction or even its operands. The type of the instruction and its operands are recognized in ID stage. Because the address of the new fetched instruction is stored in PC register. Therefore, in computing next IF stage, we have to calculate properly the value of new PC. This is done by the following:

   a. If the type of the current instruction in ID stage is "J", then the PC register in the next IF register has to be loaded by the operand of ID stage. Consider that the operand of J instruction is stored in the immediate value.

3- When the current ID stage is loaded by a J instruction, then the current IF stage is loaded by the next immediate instruction. To fix this error, next ID stage has to contain a bubble. In other words, what is currently in IF stage, has to be erased and not continue in the pipeline.

Implement J instruction, test it with program4.s assembly code. Compare your trace file produced from your implementation with the file (trace4.html) uploaded in the blackboard\Contants\lab2. Focus mainly on the first three cycles.

Program4.s:

```
.globl __start
.data
.text
__start:
j Label1
sub        $t3, $t1, $t2
or         $t4, $t1, $t2
sub        $t5, $t1, $t2
Label1:
ori        $t6, $t2, 99
addi       $v0, $v0, 10
syscall
```

5.2 Assignment 2: Forwarding

The ideal case in the microprocessor is to execute an instruction in every clock cycle, in other words, at every clock cycle, one of the instructions is processed completely after passing the five pipelines stages.

Sometimes, this ideal case stumbles due to the existing of data dependencies between instructions that already exists in the pipeline. Precisely, this hazard occurs when an instruction in DI stage needs to read the contents of a register that is being updated, at the meanwhile, in EXE or MEM stages. program2 is an example of this status..

program2.s was solved by forcing the instruction `bne` to wait in DI stage till the new values of $t1 and $t2 were written back. This enabled the microprocessor to synchronize the writing and reading to and from $t1 and $t2.

An improvement has to be done in this assignment, instead of freezing an instruction in ID stage, we aim to forward the content of any register updated in EXE or MEM stages if it is used as `rs` or `rt` in DI stage. To do so, you have to follow those steps: In `HandleHazards` function, read the used registers, `rs` and `rt`, in ID stage (use the function `GetRegsReadByInstr`). Then, check whether those two registers are updated in EXE or MEM stages. To check whether a particular register is updated in a particular stage (`stg`) then you can use the function: `GetRegWrittenByInstr`. Accordingly:

1- If one of the registers in ID stage is updated in EXE stage, then the contents of this used register in the next EXE stage has to equal the contents of the same register in the next MEM stage. Why ???.
2- If one of the registers in ID stage is updated in MEM stage, then the contents of this used register in the next EXE stage has to equal the contents of the same register in the next WB stage.

After implementing the forwarding, compare your output trace file with the corresponding uploaded in blackboard\content\trace_files\trace2_bypassed.html.

Notice here the following, each stage has an instruction and this instruction has two registers and to contents of the registers. For example:

The used register `rs` in the instruction currently loaded in ID stage is:

```
ex_stage.instr.rs
```

But the contents of this register is: `ex_stage.instr.rs_value`

5.3 Assignment 3: Branch Prediction

After execution a conditional instruction; one of two instructions have to executed, the first is the next immediate instruction. The second is whose label is in the third operand in the conditional instruction. This depends on the condition of the conditional branch instruction.. If it is false, then the first instruction is executed, otherwise, the program flow is changed and jumps to the second.

In the existing of a conditional instruction, the basic solution, which is already written in update.c file, wait until the condition status of the instruction is determined. This is done in MEM stage. Hence, the instruction which is currently in IF stage (next immediate instruction) has to wait two extra clock cycles till conditional instruction reaches MEM stage. In MEM stage, we have two possibilities:

1- The condition is false: then two bubbles are created in ID and EXE stages
2- The condition is true: then IF, ID and EXE stages have to be cleared. As well, PC is loaded by the new address.

Note: it is a very good exercise if you try to find the source code fragment that implement the above ideas.

The performance of the processing could be improved by considering initially the condition of the branch instruction false. Accordingly, the next immediate instruction is fetched and follows the conditional branch in the pipeline instead of being frozen in IF stage. When the conditional branch instruction reaches MEM stage, then there are two possibilities; first, the condition is false, in this case nothing is changed. Second, the condition is true, then the instructions in EXE, ID and IF stages are cleared by bubbles as well as the address of the instruction whose label in third operand of the conditional instruction is loaded in PC register in order to be fetched in the next clock cycle.

In this assignment, only the conditional branches BEQ (branch if equal) and BNE (branch if not equal) are required. Notice, in MEM stage, there is a Zero flag. Since this flag is set if the outcome of the logical or arithmetic operation is ZERO, it is set if BEQ is true and it is reset if BNQ is true.

Hint: Consider always the conditional branch is as a subtraction. So, for `beq $t1,$t2,L1`. *$t1 is subtracted from $t2, if their values are identical then Z = 1 (because $t1 - $t2 = 0).*

To perform the above algorithm, `HandleHazards` function checks whether the current instruction in current MEM stage is BEQ and Z = 1 or BNE and Z = 0; if one of those two cases is satisfied, then the stages IF, ID and EXE stages are cleared.

Implement your ideas, run your program to process Program3.s and then compare your trace file with trace3_BranchTrue.html. Again, replace the instruction BNE by BEQ and compare the new trace file with trace3_BranchFalse.html.

5.4 Assignment 4: Comparisons

You have some assembly test files in the directoy: test files. Compare between the number of cycles before and after your improvements.

	Before	After
Loop.s		
Lui_test.s		
Branch_stall.s		
sll_test.s		