Assignment Submission
CZ3001 Advanced Computer Organisation and Architecture
Project Report

Submitted by : Mavric Tan Soon Heng

Oh Zhi Jie

Ng Chin Teck

Tan JunJie Alvin

Lab Group : FSP1

Date Submitted : 31 March 2016

Pages : 15 inclusive of Cover page and Annexes

Background

This report is submitted for CZ3001 Project.

The 4 staged pipelined processor from lab-3 can be modified into by adding a new module “Data memory” as well as a 5th stage pipeline register “MEM_WB_Stage”. The newly refined schematic is shown as Appendix A.1.

To support the additional instructions **(BEQ, LW, SW)**, there will be a need to edit the control module as it has to generate the additional control signals. The signals that are added are “**Branch**”, “**MemWrite**”, “**RegDst**”, and “**MemtoReg**”.

The data path is also modified to include JR**, JMP** and **JAL** with the additional of some multiplexers. Likewise with the above, more control signals such as “**JR**”, “**JAL**”,”**JMP**”, “” are to be generated by the control module for the implementation of the various jump instructions.

Appendix A and B includes the various waveform diagrams from our group implementation of the pipeline.

A list of control instructions and details on when they are triggered as well as the list of opcode to instructions is attached as Appendix C.

Project Part-1:

Modify the 4-stage pipelined processor of Lab-3 to include BEQ, LW, and SW instructions and convert that to a 5-stage pipelined processor. (6 marks)

LW, SW, BEQ are all I-type instructions where by two register addresses are required as well as a 16bit immediate data.

For LW, the 16bit immediate value is added to the value of the RS register to calculate the effective memory address to "load data" from the data memory module. I

The implementation is reversed for SW. The 16 bits immediate value is added to the RT register to calculate the effective address to store the value inside RS into the "destination address" of the data memory.

For instruction BEQ, the instruction will read the contents of two register, RS and RT, and compares their values. If the value is the same, the program counter will be increment by the immediate value. Otherwise, the program counter will remain as PC +1.

To further illustrate the working of the instructions, the following MIPS code will be used for testing. The waveform diagram is attached together with this document as Appendix B.

```
ADDI $2, $0, 5          // set $2 to 5
ADD $1, $0, $0          // set $1 to zero
NOP                     // dependency on $2
SW $2, 0($0)            // set value at DM address 0x0000 to 5
BEQ $2, $1, 10          // (if $2 == $1) NPC+= 11. Will not be taken
LW $1, 0($0)            // set $1 with DM address 0x0000. Set $1 to 5
NOP                     // dependency on $1
NOP                     // dependency on $1
BEQ $2, $1, 10          // (if $2 == $1) NPC+=11. Will be taken
```

Due to the potential data hazards, NOPS have been inserted.
The waveform diagram is attached in Appendix B.1.

The MIPS code above will set register 2 to 5 and register 1 to 0. Then, register 2 will be written into memory. A branch instruction will compare between register 2 and register 1 (5==0) the branch will not take place as the values in the registers are not equal. After that, register 1 will load the value of 5 from memory and store within the register. Another branch instruction will compare between registers 2 and 1 again (5==5). As the two registers have the same value, the branch will take place and PC will increment by 10.

Project Part-2:

Modify the processor designed in Part-1 of the project to include jump register (jr), jump (J), and jump & link (jal) instructions. (5 marks)

Jump Register (JR), Jump (J) and Jump & Link (JAL) are all unconditional jumps. They are implemented such that it will be a "jump to" instead of "jump by". In other words, the destination in which the program counter to is not dependant on the current value of the program counter. However, because of hardware limitations, J and JAL will be unable to jump to any arbitrary address (absolute jump) and is restricted to a range of 2^{28} addresses relative from where the current program counter is.

J and JAL are J type instruction where by the MSB 6bits are used as opcode and the remaining 26bits will be used as offset. As mentioned above, because of the restriction on the number of bits (26+2) provided for both instructions, there is a range in which the jump is able to be performed. Also because of the delay in computation of the destination, only known during decode stage, there is a need for 1 stall cycle after each J and JAL.

J instruction works by shifting the LSB 26bits left to form a 28bit address and appending 4bits from the current program counter to form an address of 32bits. This result is only known during the decode stage and thus a stall cycle will be inserted after any J instruction.

JAL instruction works similar to J instruction in the way the new address is computed. However, JAL instruction has an addition flow of saving the address of the instruction after it (nPC) to register $31 so that it is possible to "return" to the address if "jump" has not occurred. This is particularly useful for performing procedural calls and resuming execution after the procedure has ended.

Although JR instruction does perform unconditional jump like J and JAL, it is an R type instruction. The instruction requires a 6bits opcode as well as a 5 bits register address (RS). Its purpose is to dereference the value of the register given and use the value as the next address for program counter. This will remove the range limit faced by J and JAL instruction as the value from the registers are 32bits. Likewise with the other instructions, a single stall cycle is required to retrieve the address from the register.

The following code below will illustrate the execution of the above mentioned instructions. A waveform diagram has also been attached at the end of this document as a proof of correctness and to aid visualisation of the instructions flow.

```
PC          Instruction
00000000    JAL 00000010            //goto 00000010
00000001    NOP
00000002    NOP
00000003    NOP
00000004    NOP
00000005    JMP 00000015            //goto 00000015
00000006    NOP
00000007    NOP
00000008    NOP
00000009    NOP
0000000A    NOP
0000000B    NOP
0000000C    NOP
0000000D    NOP
0000000E    NOP
0000000F    NOP
00000010    JR $31                  //goto 00000001
00000011    NOP
00000012    NOP
00000013    NOP
00000014    NOP
00000015    NOP
```

Project Part-3:

Each group of students will be given a program which gets slowed due to pipeline stalls. You are required to modify the program to remove the hazards so as to reduce the number of pipeline stalls. Finally, you will estimate the reduction in the CPI and execution time which you achieve. (4 marks)

(a) Write the MIPS-like assembly code for the following program segment to run on the 5-stage pipelined processor which you have developed.

```
sum=0;
for (i=0; i<=7; i=i+1) {
        sum = sum + x[i];
}
```

Convert that assembly code to machine language format, and execute them on the 5-stage pipelined processor which you have developed and Find the number of clock cycles and execution time to execute those machine instructions.

The below is the unenhanced version:

```
        ADD $1, $zero, $zero        // $1 = sum = 0
        ADD $2, $zero, $zero        // $2 = index = 0
        ADDI $3, $zero, 8           // $3 = max = 8
        ORI $4, $zero, 100          // store address into $4
        NOP
Loop    BEQ $2, $3, done            // if equal jump out
        NOP                         // branch delay
        NOP                         // branch delay
        LW $5, 0($4)                // load element into $5
        NOP                         // dependency $5
        NOP                         // dependency $5
        ADD $1, $1, $5              // sum = sum + $5
        ADDI $2, $2, 1              // inc index
        ADDI $4, $4, 1              // inc array address
        Jmp loop                    // Loop
        NOP                         // branch delay
```

```
Number of instructions
= 4 + 8 * 6
= 52 instructions

Number of cycles
= 4 + 1 stalls + 8 * (6+5)
= 93 cycles

Steady CPI
= 93 / 52
= 1.788

Execution time
= 93 cycles * 30ns
= 2760ns
```

Some instructions can be rearranged to remove stall cycles

```
        ADD $2, $zero, $zero        // $2 = index = 0
        ADDI $3, $zero, 8           // $3 = max = 8
*       ADD $1, $zero, $zero        // $1 = sum = 0
        ORI $4, $zero, 100          // store address into $4
Loop    BEQ $2, $3, done            // if equal jump out
        NOP                         // branch delay
        NOP                         // branch delay
        LW $5, 0($4)                // load element into $5
*       ADDI $2, $2, 1              // inc index
*       ADDI $4, $4, 1              // inc array address
        Jmp loop                    // Loop
*       ADD $1, $1, $5              // sum = sum + $5
```

```
Number of instructions
= 4 + 8 * 6
= 52 instructions

Number of cycles
= 4 + 8 * (6+2)
= 68 cycles

Steady CPI
= 68 / 52
= 1.308

Execution time
= 68 cycles * 30ns
= 2040ns
```

The 1st NOP can be replaced by instruction ***ADD $1, $zero, $zero*** as there no dependency.

4th and 5th NOP can be replaced with ***ADDI $2, $2, 1*** and ***ADDI $4, $4, 1.***

The final NOP due to JMP can be used to compute ***ADD $1, $1, $5***.

(b) Perform maximal loop unrolling as well as instruction reordering of the assembly code segment obtained for part (a). Convert that assembly code segment to machine language format, and run on the 5-stage pipelined processor to find the number of clock cycles and execution time.

As we have 32 registers in our register file, it is possible for us to unroll the loop entirely so that control hazards are removed completely. The below table contains two versions of our unrolled loop. The first version contains some stall cycles that can be reordered with useful instructions. It will be obvious to see that the reordered version has a better steady state CPI as well as a faster execution time.

Unrolled Loop		Unrolled Loop Reordered	
ADD $2, $zero, $zero ORI $4, $zero, 100 LW $5, 0($4); LW $6, 1($4); LW $7, 2($4); LW $8, 3($4); LW $9, 4($4); LW $10, 5($4); LW $11, 6($4); LW $12, 7($4); ADD $2, $2, $5; NOP; NOP; ADD $2, $2, $6; NOP; NOP; ADD $2, $2, $7; NOP; NOP; ADD $2, $2, $8; NOP; NOP; ADD $2, $2, $9; NOP; NOP; ADD $2, $2, $10; NOP; NOP; ADD $2, $2, $11; NOP; NOP; ADD $2, $2, $12;	Total Number of instructions **= 18** Total Number of cycles = 18 inst + 14 stalls **= 32 cycles** Steady State CPI = 32 / 18 **= 1.778** Execution time = 32 * 30ns **= 960ns**	ADD $2, $zero, $zero ORI $4, $zero, 100 LW $5, 0($4); ADD $2, $2, $5; LW $6, 1($4); LW $7, 2($4); ADD $2, $2, $6; LW $8, 3($4); LW $9, 4($4); ADD $2, $2, $7; LW $10, 5($4); LW $11, 6($4); ADD $2, $2, $8; LW $12, 7($4); NOP; ADD $2, $2, $9; NOP; NOP; ADD $2, $2, $10; NOP; NOP; ADD $2, $2, $11; NOP; NOP; ADD $2, $2, $12;	Total Number of instructions **= 18** Total Number of cycles = 18 inst + 7 stalls **= 25 cycles** Steady State CPI = 25 / 18 **= 1.389 *** Execution time = 25 * 30ns **= 750ns**

* Steady state CPI can be reduced to 1 if data forwarding is implemented.
** highlighted to denote RAW dependencies

(c) Compare the execution times found for part (a) and part (b), and explain the effect of loop-carried dependency on the execution time.

From the table of comparison below, we can noticed that execution time is vastly reduced when the loop is unrolled. The speed up over the unenhanced loop over the unrolled is 368%.

This is due to the removal of unnecessary loop initialisation (loop overheads), and removal of stalls cycle from control hazards. Without the stalls cycles from data dependency and control dependency, a program can actually run much faster. However, it is important to note that this speed up has come at a cost of additional registers and because registers are of a limited number, it may not be always possible for loop unrolled to be performed.

Table 1 Comparison between the various implementations

Type	Loop unenhanced	Loop enhanced	Loop unrolled
Instructions	52	52	**18**
Clock cycles	93	68	**25**
Steady State CPI	1.788	**1.308**	1.389 *
Execution time	2760ns**	2040ns**	**750ns****

* Steady state CPI can be reduced to 1 if data forwarding is implemented.
** Clock period is set to 30ns

(d) **Convert the assembly code in part (a) to machine language format and find the number of clock cycles and execution times to find sum of 4 arrays A, B, C, and D, where each of the arrays consists of 8 integer elements**.

```
        ADD $1, $zero, $zero            // $1 = sum = 0
        ADD $2, $zero, $zero            // $2 = index = 0
        ADDI $3, $zero, 8               // $3 = max = 8
        ORI $4, $zero, 100              // store address into $4
        NOP
Loop_A  BEQ $2, $3, done_A              // if equal jump out
        NOP                             // branch delay
        NOP                             // branch delay
        LW $5, 0($4)                    // load element into $5
        NOP                             // dependency $5
        NOP                             // dependency $5
        ADD $1, $1, $5                  // sum = sum + $5
        ADDI $2, $2, 1                  // inc index
        ADDI $4, $4, 1                  // inc array address
        Jmp loop_A                      // Loop
        NOP                             // branch delay
Done_A  ADD $2, $zero, $zero            // Reinitialise index to zero
        ADDI $3, $zero, 8               // Reinitialise max to zero
        ORI $4, $zero, 200              // store address into $4
        NOP
Loop_B  BEQ $2, $3, done_B              // if equal jump out
        NOP                             // branch delay
        NOP                             // branch delay
        LW $5, 0($4)                    // load element into $5
        NOP                             // dependency $5
        NOP                             // dependency $5
        ADD $1, $1, $5                  // sum = sum + $5
        ADDI $2, $2, 1                  // inc index
        ADDI $4, $4, 1                  // inc array address
        Jmp loop_B                      // Loop
        NOP                             // branch delay
Done_B  ADD $2, $zero, $zero            // Reinitialise index to zero
        ADDI $3, $zero, 8               // Reinitialise max to zero
        ORI $4, $zero,300               // store address into $4
        NOP
Loop_C  BEQ $2, $3, done_C              // if equal jump out
        NOP                             // branch delay
        NOP                             // branch delay
        LW $5, 0($4)                    // load element into $5
        NOP                             // dependency $5
        NOP                             // dependency $5
        ADD $1, $1, $5                  // sum = sum + $5
        ADDI $2, $2, 1                  // inc index
        ADDI $4, $4, 1                  // inc array address
        Jmp loop_C                      // Loop
        NOP                             // branch delay
Done_C  ADD $2, $zero, $zero            // Reinitialise index to zero
        ADDI $3, $zero, 8               // Reinitialise max to zero
        ORI $4, $zero, 400              // store address into $4
        NOP
Loop_D  BEQ $2, $3, done_D              // if equal jump out
        NOP                             // branch delay
        NOP                             // branch delay
        LW $5, 0($4)                    // load element into $5
        NOP                             // dependency $5
        NOP                             // dependency $5
        ADD $1, $1, $5                  // sum = sum + $5
        ADDI $2, $2, 1                  // inc index
        ADDI $4, $4, 1                  // inc array address
        Jmp loop_D                      // Loop
        NOP                             // branch delay
Done_D
```

Loop initialisation for SUM A
Clock cycles = 5

Loop Body for SUM A
Clock cycles
= 8 * 11
= 88

Loop initialisation for SUM B
Clock cycles = 5

Loop Body for SUM A
Clock cycles
= 8 * 11
= 88

Loop initialisation for SUM C
Clock cycles = 5

Loop Body for SUM C
Clock cycles
= 8 * 11
= 88

Loop initialisation for SUM D
Clock cycles = 5

Loop Body for SUM D
Clock cycles
= 8 * 11
= 88

Total Cycles = (5 + 8 * 11) + (4 + 8 * 11) + (4 + 8 *11) + (4 + 8 *11) = 369
Total Execution Time = 369 * 30ns =11070ns

(e) **Using loop fusion of the form given below and write another MIPS-like code for the program segment to run on the 5-stage pipelined processor which you have developed.**

```
sum1=0;
sum2=0;
sum3=0;
sum4=0;
for (i=0; i<=7; i=i+1) {
      sum1 = sum1 + A[i];
      sum2 = sum2 + B[i];
      sum3 = sum3 + C[i];
      sum4 = sum4 + D[i];
      }
```

Convert the assembly code to machine language format, perform necessary instruction reordering to minimize the pipeline stalls, execute them on the 5-stage pipelined processor, and find the number of clock cycles and execution time.

	Assembly	Remarks
	ADD $1, $zero, $zero;	//SUM = 0
	ADDI $2, $zero, 8;	//MAX = 8
	ADD $3, $zero, $zero;	//INDEX =0
	ORI $4, $zero, 10;	//Array A
	ORI $5, $zero, 20;	//Array B
	ORI $6, $zero, 30;	//Array C
	ORI $7, $zero, 40;	//Array D
	ADD $8, $zero, $zero;	//SUMA
	ADD $9, $zero, $zero;	//SUMB
	ADD $10, $zero, $zero;	//SUMC
	ADD $11, $zero, $zero;	//SUMD
LOOP	BEQ $2, $3, Done	//Jump when equal
	NOP	//branch delay
	NOP	//branch delay
	LW $12, 0($8)	// A[i]
	LW $13, 0($9)	// B[i]
	LW $14, 0($10)	// C[i]
	LW $15, 0($11)	// D[i]
	ADD $8, $8, $12;	//SUMA+=A[]
	ADDI $4, $4, 1;	// inc address A
	ADD $9, $9, $13;	//SUMB+=B[]
	ADDI $5, $5, 1;	// inc address B
	ADD $10, $10, $14;	//SUMC+=C[]
	ADDI $6, $6, 1;	// inc address C
	ADD $11, $11, $15	//SUMD +=D[]
	ADDI $7, $7, 1;	// inc address D
	ADDI $3, $3, 1	//INC INDEX
	J LOOP	
	NOP	//branch delay

Loop initialisation

11 clock cycles

Loop body

Clock cycles

= 18 *8

= 155

Total Cycles = 11 + 18 * 8
= 155

Total Execution Time = 155 * 30ns
= 4650ns

(f) Compare the execution times found for part (d) and part (e). Explain the result.

Without Loop Fusion (d)	With Loop Fusion (e)
Number of instructions **= 4 + 6*8 + 3 + 6*8 + 3 + 6*8 + 3 + 6*8** **= 205** **Number of clock cycles** **= 369** **Steady State CPI** **= 1.8** **Execution Time** **= 11070ns**	**Number of instructions** **= 11 + 15 *8** **= 131** **Number of clock cycles** **= 155** **Steady State CPI** **= 1.183** **Execution Time** **= 4650ns**

The table above shows the differences between the two versions of the same program. With the implementation of fusion, the steady state CPI vastly reduced to near 1 and execution time has also decreased by 42%.

There is a slight difference made when the instructions for loop initialisation is reduced. However, majority of the improvement is due to the removal of unnecessary control hazards that (d) has introduced. Because of the merging of the loop conditions, the program has effectively halved the execution time between the two versions.

Bonus

Any substantial improvement made in the hardware to improve the performance of the system.

Data forwarding

In the 5 stage pipelined processor, we can determine potential data hazards at the ID (decode) stage. Likewise, it will be possible for us to determine what forwarding needs to be done at the same stage. This can be done by comparing the source and destination registers of the adjacent instructions. Thereafter, control will be able to generate the necessary signals to trigger the data forwarding.

The table below shows an example of comparison.

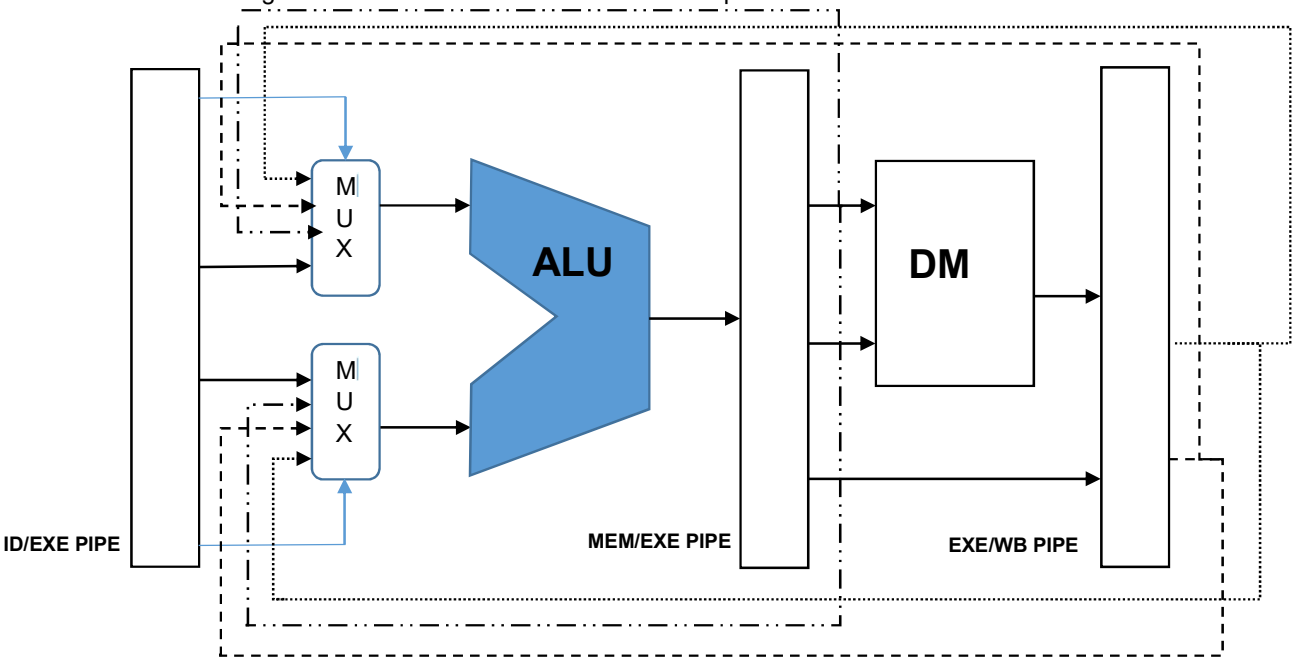
Scenario	MIPS code	Remarks

No dependencies	`LW $1, 0 ($2)`
	`ADD $2, $3, $4`
	`SUB $5, $6, $4`
		No dependencies between all instructions

Dependency that can be overcome by forwarding	`LW $1, 0($2)`
	`ADD $2,$3,$4`
	`SUB $10, $1,$1`
		There is a RAW dependency between $1 in instruction 1 and 3. However, because $1 can be evaluated in the MEM stage, it can be forwarded into instruction 3.

Dependency that cannot be overcome by forwarding	`LW $1, 0($2)`
	`SUB $10, $1,$1`
	`ADD $2,$3,$4`
	`XOR $5, $6, $7`
		There is also a RAW dependency between $1 in instructions 1 and 2. But forwarding is unable to overcome the dependency because $1 can only be evaluated in MEM stage and hence a stall cycle has to be inserted before instruction 2

As data forwarding always occur at the end of **EXE** or **MEM** stage and requires piping back as source to the ALU during **EXE** stage, all that we will need to the existing 5 stage pipeline is the additional control signals as well as the redirection of output from the ALU and Data Memory back as the source to ALU. The figure below shows the addition to the datapath.



There is 3 ways forwarded data can be redirected in as ALU source.

1) The output of ALU after EXE Stage
2) The output of ALU after MEM stage
3) The output of Data memory after MEM stage

b. Control hazard removal

In the five stage pipelined schematics given in the assignment sheet, there is a need for two stall cycles for the instruction **BEQ**. This is because the result of **RS** and **RT** can only be known after the **EXE** stage and thus allowing a new fetch only at the **MEM** stage of the instruction. Illustration shown in the table below.

			Result only know here				
BEQ	IF	ID	EXE	MEM	WB		
Next Inst		Stall	Stall	IF	ID	MEM	WB

However, this can be reduced to just one stall cycle. This is because the content in both **RS** and **RT** are already know at the decode stage of the instruction and hence we can do the comparison during the decode stage allowing us to perform the next instruction fetch at the **EXE** stage, requiring only one additional stall cycle.

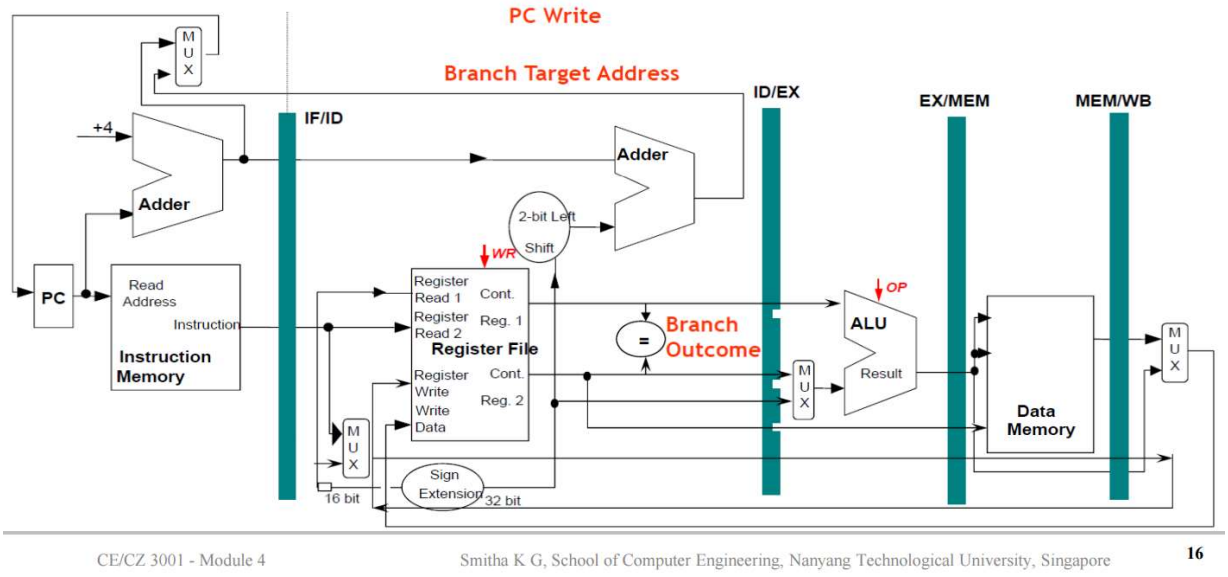


Image from lecture slide.

Appendix A

This waveform diagram is to help aid visualisation of LW, SW and BEQ instructions. LW is in orange. SW is in teal. BEQ is in purple and green.

Workings of SW

Fetch:
IM outputs machine code instruction

Decode:
Control reads in opcode and generates signals MEMWRITE, ALUSRC

Execute:
Effective address is calculated

Memory:
Data is written into memory at effective address

Writeback:
No write back since there is no writing into register file

Workings of LW

Fetch:
IM outputs machine code instruction

Decode:
Control reads in opcode and generates signals MEMTOREAD, ALUSRC, WriteEn

Execute:
Effective address is calculated

Memory:
Data is read from memory at effective address

Writeback:
Data from memory is written into register file

Workings of BEQ

Fetch:
IM outputs machine code instruction

Decode:
Control reads in opcode and generates signals ALUSRC,BRANCH

Execute:
Zero signal is generated by comparing RS and RT

Memory:
No memory operation

Writeback:
No write back since there is no writing into register file

Instructions

```
1)  NOP              // NOP
2)  ADDI $2, $0, 5   // set $2 to 5
3)  ADD $1, $0, $0   // set $1 to zero b
4)  NOP              // Data hazard for $2
5)  SW $2, 0($0)     // set value at DM address 0x0000 to 5
6)  BEQ $2, $1, 10   // (if $2 == $1) NPC+= 11. Will not be taken
7)  LW $1, 0($0)     / set $1 with DM address 0x0000. Set $1 to 5
8)  NOP              // Data hazard for $1
9)  NOP              // Data hazard for $1
10) BEQ $2, $1, 10   // (if $2 == $1) NPC+=11. Will be taken
```

Appendix B.

Ctrl_Jump

JAL signal to write to $31

Ctrl_ Jr signal

Ctrl_jump signal

This wave form diagram help aids visualization of the instruction JAL, JR and a JMP instruction. JAL is in Orange, JR is in teal, JMP is in purple.

Working of JAL

Fetch:
IM outputs machine code instruction.

Decode:
Control reads opcode and generate signals ctrl_jump and MEM_WB_jal. The nPC and MEM_WB_jal is passed into the 3 stage pipeline. The nPC is updated to immediate.

Execute:
No execute operation.

Memory:
nPC is written into $31

Write back:
No write back operation.

Working of JMP

Fetch:
IM output machine code instruction.

Decode:
Control read opcode and generate signals ctrl_jump. The nPC is updated to immediate.

Execute:
No execute operation

Memory:
No memory operation.

Write back:
No write back operation

Working of JR

Fetch:
IM outputs machine code instruction.

Decode:
Control reads opcode and generate signal ctrl_jr. The RF outputs address to PCin.

Execute:
No execute operation.

Memory:
No memory operation.

Write back:
No write back operation.

Instructions

```
0) NOP
1) JAL 00000010 //Jump to 10 and save nPC in $31
2) NOP
3) NOP
4) NOP
5) NOP
6) JMP 00000015   //Jump to 0x15
7) NOP
8) NOP
9) NOP
A) NOP
B) NOP
C) NOP
D) NOP
E) NOP
F) NOP
10) JR $31      //Jump to $31
11) NOP
12) NOP
13) NOP
14) NOP
15) NOP
```

Appendix C

Table for control signals and their conditions

Control Signals	Conditions		Used by Instructions
	0	1	
ctrl_regDst	Use INST[20:16](RT) as wAddr	Use INST[15:11](RD) as wAddr	R and I type instructions
ctrl_memWrite	disable writing to data memory	enable writing to data memory	SW
ctrl_memToReg	Use alu_result as wData1	Use DM_dataout as wData1	LW
ctrl_branch	Use nPC as nPC2	Use nPC1 as nPC2	BEQ
ctrl_jump	Use nPC2 as nPC3	User jumpPC as nPC3	JMP
ctrl_jal	Use wData1 as wData2 Use MEM_WB_wAddr as regWAddr	Use nPC as wData2 Use 5'b11111 as regWAddr	JAL
ctrl_jr	Use nPC3 as PCIN	Use reg_rdata1 as PCIN	JR
ctrl_writeEn	disable writing to register file	enable writing to register file	R-type instruction, I-type instruction, LW, JAL
ctrl_ALUSrc	Use ID_EXE_rdata2 as alu_operand2	Use ID_EXE_immData as alu_operand2	R and I-type instructions

ctrl_ALUop	**ALU_opcode**	**Instruction**
	000	ADD
	001	SUB
	010	AND
	011	OR
	100	XOR
	101	MUL
	110	NOT
	111	COM

Instructions and opcode

Instructions	6bit opcodes	Instruction Type
ADD	000000	R-type
SUB	000001	R-type
AND	000010	R-type
OR	000011	R-type
XOR	000100	R-type
MUL	000101	R-type
NOT	000110	R-type
COM	000111	R-type
JR	001000	R-type special
ADDI	100000	I-type
SUBI	100001	I-type
ANDI	100010	I-type
ORI	100011	I-type
XORI	100100	I-type
LW	111000	I-type
SW	110000	I-type
BEQ	111111	I-type
JMP	010000	J-type
JAL	010001	J-type