Project 1: Pipelined CPU using Verilog

2016.11.23

Project 1

- 最多三人一組, 請在（ https://goo.gl/AY1c2N ）確認分組，分組有問題的話請寄信給助教
- Deadline:
  - Deadline: **12/12(一) (3 weeks)** 午夜**12:00**
- Demo:
  - 另行公佈時段給各組填寫（約在繳交後一週左右）
  - 內容：執行程式給助教檢查，回答數個相關問題
  - 需全員到齊

Require

- Required Instruction Set：
  - and
  - or
  - add
  - sub
  - mul
  - addi
  - lw
  - sw
  - beq
  - j

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx

Require

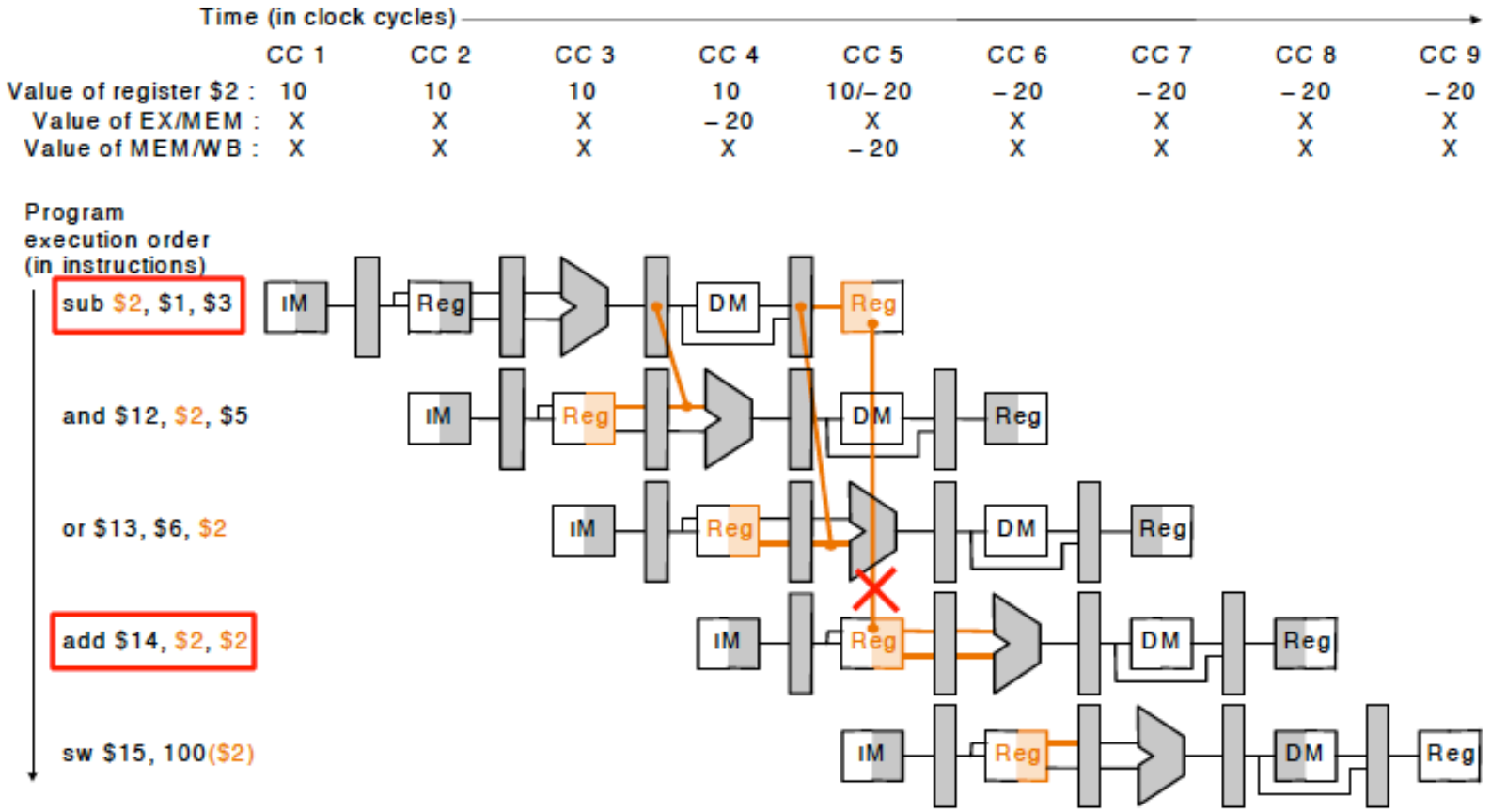
- Register File : 32 Register (Write When clock rising edge)
- Instruction Memory : 1KB
- Data Memory : 32 Bytes
- Data Path & Module Name
- MUL OpCode :

0	rs	rt	rd	0	0x18

- Hazard handling
  - Data hazard
    - Implement the Forwarding Unit to reduce or avoid the stall cycles.
    - The data dependency instruction follow “lw” must stall 1 cycle.
    - No need forwarding to ID stage
  - Control hazard :
    - The instruction follow ‘beq’ or ‘j’ instruction may need stall 1 cycle.
    - Pipeline Flush

Data hazard

- 不需要forwarding 到 ID stage !



Forwarding Control

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

1. EX hazard

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd=ID/EX.RegisterRs))
ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd=ID/Ex.RegisterRt))
ForwardB = 10

2. MEM hazard

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd=ID/Ex.RegisterRs))
ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd=ID/Ex.RegisterRt))
ForwardB = 01

Stall & Flush

- In testbench.v
- Can be changed depends on your own design.

```
// put in your own signal to count stall and flush

   if (CPU.HazzardDetection.mux8_o == 1 && CPU.Control.Jump_o == 0
&& CPU.Control.Branch_o == 0)
            stall = stall + 1;

   if(CPU.HazzardDetection.Flush_o == 1)
            flush = flush + 1;
```

Require (cont.)

- (80%) Source code (put all .v file into “code” directory)
  - CPU module
    - Basic (40%)
    - Data forwarding (20%)
    - Data hazard (lw stall) (10%)
    - Control hazard (flush) (10%)
  - TestBench (may need to modified)
    - Initialize storage units
    - Load instruction.txt into instruction memory
    - Create clock signal
    - Output cycle count in each cycle
    - Output Register File & Data Memory in each cycle
    - Print result to output.txt
  - TestData
    - Fibonacci
- (20%) Report (project1_teamXX.doc)
  - Members & Team Work
    - 須註明組員工作分配比例
  - How do you implement this Pipelined CPU.
  - Explain the implementation of each module.
  - Problems and solution of this project.
- Put all files and directory into *project1_teamXX_V0*

Require (cont.)

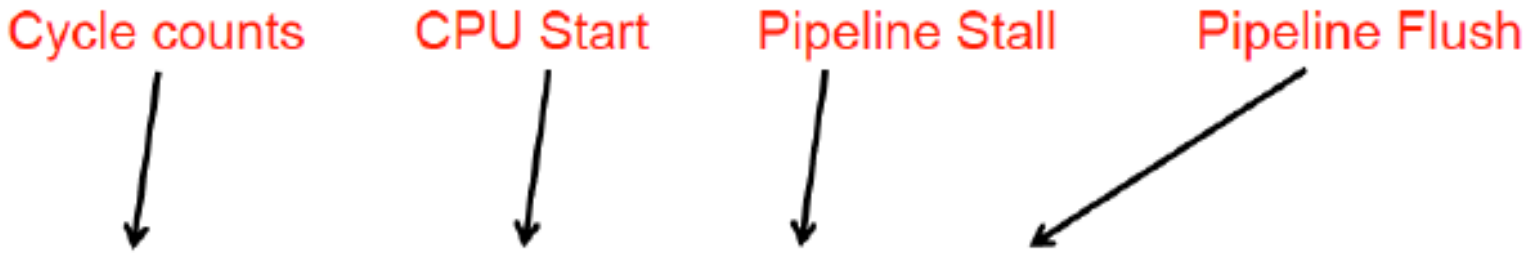
- Fibonacci
  - Set Input n into data memory at 0x00
  - CPU.Data_Memory.memory[0] = 8'h5;

```
// Initialzation
addi $a0, $r0, 0    // $a0 = 0
addi $t0, $r0, 0    // $t0 = 0
addi $t1, $r0, 1    // $t1 = 1
addi $s1, $r0, 0    // $s1 = 0
addi $s2, $r0, 1    // $s2 = 1

// Load number n from data memory
lw $s0, 0($a0)
addi $s4, $s0, 0    // save input value in $s4

// Calculate the Fibonacci Number
loop:
sub $s0, $s0, $t1  //subu $s0, 1
add $s1, $s1, $s2
addi $s3, $s2, 0    //move $s3, $s2
addi $s2, $s1, 0    //move $s2, $s1
addi $s1, $s3, 0    //move $s1, $s3
beq $s0, $t0, finish   //bne $s0, $zero, loop
j loop
```

```
// Store the result
finish:
sw $s1, 4($a0)

// Calculate the input and output and Store
and $t2, $s1, $r0
or $t3, $s1, $r0
add $t4, $s1, $t4
sub $t5, $s1, $t4
mul $t6, $s1, $t4
sw $t2, 8($a0)
sw $t3, 12($a0)
sw $t4, 16($a0)
sw $t5, 20($a0)
sw $t6, 24($a0)
```

Change instruction sequence if needed, to avoid ID-forwarding.

Output Example




```
cycle =          0, Start = 0, Stall = 0, Flush = 0
PC =         0
Registers
R0(r0) =         0, R8 (t0) =         0, R16(s0) =         0, R24(t8) =         0
R1(at) =         0, R9 (t1) =         0, R17(s1) =         0, R25(t9) =         0
R2(v0) =         0, R10(t2) =         0, R18(s2) =         0, R26(k0) =         0
R3(v1) =         0, R11(t3) =         0, R19(s3) =         0, R27(k1) =         0
R4(a0) =         0, R12(t4) =         0, R20(s4) =         0, R28(gp) =         0
R5(a1) =         0, R13(t5) =         0, R21(s5) =         0, R29(sp) =         0
R6(a2) =         0, R14(t6) =         0, R22(s6) =         0, R30(s8) =         0
R7(a3) =         0, R15(t7) =         0, R23(s7) =         0, R31(ra) =         0
Data Memory: 0x00 =          0
Data Memory: 0x04 =          0
Data Memory: 0x08 =          0
Data Memory: 0x0c =          0
Data Memory: 0x10 =          0
Data Memory: 0x14 =          0
Data Memory: 0x18 =          0
Data Memory: 0x1c =          0
```

Data Path & Module