Computer Architecture Project1

Pipelined CPU using Verilog

1. 工作分配

B05902053 陳奕均 維護spec, 寫report

B05902071 陳郁文 解釋架構, 寫小部分code

B05902083 余柏序 report顧問, 寫大部分code

1. 實作方式
2. instruction -> ALU-control

由原始的opcode可以決定各種signal和ALU\_Op

R-type ALU\_Op 都是 10, 要看function code

其他類instruction ALU\_Op可能是00或01 (對應到加和減)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R-type | | | | | |
| opcode | instruction | funct7 | funct3 | ALU-action | ALU-control |
| 0110011 | and | 0000000 | 111 | AND | 001 |
| 0110011 | or | 0000000 | 110 | OR | 010 |
| 0110011 | add | 0000000 | 000 | ADD | 011 |
| 0110011 | sub | 0100000 | 000 | SUB | 100 |
| 0110011 | mul | 0000001 | 000 | MUL | 101 |
| I-type | | | | | |
| 0010011 | addi | X | 000 | ADD | 011 |
| 0000011 | lw | X | 010 | ADD | 011 |
| S-type | | | | | |
| 0100011 | sw | X | 010 | ADD | 011 |
| SB-type | | | | | |
| 1100011 | beq | X | 000 | SUB | 100 |

欲決定ALU\_Ctrl

如果是R-type, 即ALU\_Op = 10

這時看function code: funct\_i = {inst[30], inst[25], inst[14], inst[13], inst[12]};

由下表得ALU\_Control

|  |  |  |
| --- | --- | --- |
| funct\_i | ALU-action | ALU-control |
| 00111 | AND | 001 |
| 00110 | OR | 010 |
| 00000 | ADD | 011 |
| 10000 | SUB | 100 |
| 01000 | MUL | 101 |

如果不是R-type, 看ALU\_Op 即可

1. instruction -> Control Signal (with pipeline)

必須實作出下方這張表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | R-type | addi | lw | sw | beq |
| ID | Branch | 0 | 0 | 0 | 0 | 1 |
| EX | ALUOp | 10 | 00 | 00 | 00 | 01 |
| ALUSrc | 0 | 1 | 1 | 1 | 0 |
| M | MemRead | 0 | 0 | 1 | 0 | 0 |
| MemWrite | 0 | 0 | 0 | 1 | 0 |
| WB | RegWrite | 1 | 1 | 1 | 0 | 0 |
| MemtoReg | 0 | 0 | 1 | X | X |
| opcode | | 0110011 | 0010011 | 0000011 | 0100011 | 1100011 |

原先single cycle processor可以直接decode, 直接傳signal, 但現在還要考慮pipeline

一種做法是先把signal decode出來之後全部存在pipeline register中, 逐步傳遞, 不過我們採取作法的是從Control開始, 每次都傳opcode, 一步一步往下走, 當需要用到Signal在當場decode即可

比對opcode會發現只需要傳前3個bit就可以了

在 project中實現如下:

opcode\_ID => opcode\_EX => opcode\_MEM => opcode\_WB (傳遞)

產生Signal:

opcode\_ID -> Branch Signal (Beq\_Op in our code)

opcode\_EX-> ALUOp, ALUSrc Signal

opcode\_MEM -> MemWrite Signal

opcode\_WB-> RegWrite Signal (regwrite\_MEM, regwrite\_WB in our code)

opcode\_WB-> MemtoReg Signal (reg\_src\_WB in our code)

1. Data Forwarding

考慮將前2個stage的結果預先傳回來

在source code中, RegWrite\_p, RegWrite\_pp 是前一個, 前兩個stage的RegWrite Signal

Rd\_p, Rd\_pp 是前一個, 前兩個stage的Rd位址

Rs1\_i, Rs2\_i 是當前指令需要的rs1和rs2位址

比對位址後可以知道要不要以及如何做forwarding, 送給兩個MUX32\_3i適當的選擇訊號, 讓ALU拿到兩個正確的input

1. Data Hazard

由Hazard\_Detect來做檢驗, 當指令是lw instruction, 且後面的指令需要用到前面指令應寫入而尚未寫入的register, 讓PC無法寫入來stall一個cycle

1. Control Hazard

beq instruction在decode的時候就檢查有沒有equal (提早檢查可以少flush一個instruction) 如果有, Beq 會被set, 送到Control模組, 清空IF/ID pipeline register, 也會送給PC\_Mux對應的選擇訊號, 使PC能跳到該跳的位置

1. 其他細節
2. ALU的Zero\_o原本是用來判斷beq, 但現在beq在decode的時候已經提早判斷, 這個port已經用不到, 所以省去
3. Control的output port valid bit用來判斷指令是否有效，超出project1範圍無效。或者當beq成立時會flush IF/ID pipeline register, 這時讀出的instruction 會是32’b0, 也是無效的
4. 0號register不可寫入(Register.v)